Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-2009-19

M.Sc. Thesis

Desynchronization Methods for Scheduled Circuits

Michael Simmonds B.ASc.

Abstract

Synchronous systems waste a lot of power in the clock tree, and must be designed based on the worst case scenario in terms of speed. Asynchronous circuits offer relief to these problems, by replacing clock signals with handshakes which only charge when data is being transferred, and delay signals which may adapt more easily to variance in speed compared to the clock period. Desynchronization is the process of turning a synchronous circuit into an asynchronous one. Scheduled circuits are a common way to provide a good compromise between conserving area of a circuit and increasing its speed. Desynchronizing such a system is made difficult because every functional unit in the circuit must respond to controls from the central state machine, which cannot easily handshake with all of them. This report demonstrates two related methods designed specifically for the conversion of a synchronous, scheduled circuit into an asynchronous, delay insensitive circuit. Decomposition of the central state machine into local, smaller ones is used to combat the problem of skew in the control signals, as well as to speed up the performance of the asynchronous circuit. The slack in the clock period can also be used for possible speedup. Conditions which threaten deadlock of the circuit are identified and rescheduling solutions are proposed. A tradeoff between the two methods of area conservation and hardware reusability versus speed is also explained.

Faculty of Electrical Engineering, Mathematics and Computer Science

Delft University of Technology

Desynchronization Methods for Scheduled Circuits

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Engineering

by

Michael Simmonds B.ASc.
born in North York, Canada

This work was performed in:

Circuits and Systems Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

TUDelft

Delft University of Technology

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"Desynchronization Methods for Scheduled Circuits"** by **Michael Simmonds B.ASc.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: November 10th, 2009

Chairman:

prof.dr.ir. Alle-Jan van der Veen

Advisor:

dr.ir. Rene van Leuken

Committee Members:

dr. Ben Juurlink

dr.ir. Nick van der Meijs

Abstract

Synchronous systems waste a lot of power in the clock tree, and must be designed based on the worst case scenario in terms of speed. Asynchronous circuits offer relief to these problems, by replacing clock signals with handshakes which only charge when data is being transferred, and delay signals which may adapt more easily to variance in speed compared to the clock period. Desynchronization is the process of turning a synchronous circuit into an asynchronous one. Scheduled circuits are a common way to provide a good compromise between conserving area of a circuit and increasing its speed. Desynchronizing such a system is made difficult because every functional unit in the circuit must respond to controls from the central state machine, which cannot easily handshake with all of them. This report demonstrates two related methods designed specifically for the conversion of a synchronous, scheduled circuit into an asynchronous, delay insensitive circuit. Decomposition of the central state machine into local, smaller ones is used to combat the problem of skew in the control signals, as well as to speed up the performance of the asynchronous circuit. The slack in the clock period can also be used for possible speedup. Conditions which threaten deadlock of the circuit are identified and rescheduling solutions are proposed. A tradeoff between the two methods of area conservation and hardware reusability versus speed is also explained.

Acknowledgments

I would like to acknowledge my family, my girlfriend, and all of my friends for their support throughout the entire duration of my studies. Knowing I can count on them, and that they are counting on me is what keeps me going.

I would like to thank Alexander de Graaf, Huib Lincklaen Arriens, Tao Xu, and most of all my advisor Rene van Leuken, for their assistance and guidance during the formulation of the ideas that went into this thesis, their implementation and testing, and the writing of this report.

Michael Simmonds B.ASc.
Delft, The Netherlands
November 10th, 2009

Contents

List of Figures

List of Tables

Introduction 1

1.1 Motivation

Transistor sizes continue to shrink, and engineers continue to search for new ways in which these exponential gains can be effectively used to increase computing power and decrease strain on the world's resources. Processing power is also so inexpensive that its use is being considered in all aspects of technology, society, and life. Engineers must look for new and better embedded solutions to meet the demands of the modern age, and sometimes must look outside of widely used and accepted digital hardware design methods.

Traditionally processor and digital electronic design is done in a synchronous fashion, using a clock signal which ideally rises and falls identically to every component and gives the circuit a sense of time. In general, this makes the design easier, but it has several drawbacks. Since a large portion of the circuit is occupied by wires carrying this falling and rising clock signal, and in most cycles most components are not changing, there is a lot of power used to drive these lines which is constantly wasted, often over 50 percent of the total power used in the circuit. In addition, the clock can only be as fast as the slowest register-to-register path in the circuit, which is a design constraint forcing engineers to make the gate delay of each component as similar in delay as possible, and limiting design possibilities. This also restricts the circuit from taking advantage of delay variability, that is, the phenomenon of the circuit's speed being dependent on operating conditions such as temperature. Circuits are also affected by clock skew, as in increasingly denser circuits it can take different amounts of time for the signal to propagate in different directions in a non-symmetrical circuit. In synchronous design processes, a great deal of effort is expended in ensuring the longest path is shorter than the clock period.

In traditional asynchronous design however, the entire synchronous circuit must be designed in a much more complicated process, which slows down time to market and makes new circuits incompatible with and unable to use most of the digital components designed up until now, since they employ clock signals. In this way we have to redesign all components that we want to use in an asynchronous circuit, and therefore lose backwards compatibility, as well as the ability to use experience in current design methods on future designs. Even so, its advantages are numerous and compelling enough that it has been used to create several processors [13].

Various efforts are being made to make asynchronous design more feasible for digital component design in both the academic and commercial sectors. There are increasingly more tools to aid in the design of asynchronous circuits[6, 3], but this pales in comparison to the amount of tools there are available for the design of synchronous systems. If there was an algorithm to systematically convert current and future synchronous designs to asynchronous designs, it could be a very attractive compromise with many benefits of both synchronous and asynchronous systems. The only real synchronous part of the synchronous circuit is the clock signal, so this means somehow the clock must be removed and communication between registers must be instead done in an asynchronous manner.

There are several methods for turning a synchronous circuit into an asynchronous one, using handshaking to replace the functions that originally are handled by the clock signal. For a pipelined circuit, these implementations are fairly straightforward, because all of the components have the same connections and perform the same tasks at any given step in the circuit.

If these methods are extended to circuits controlled with a central FSM, however, things are somewhat more difficult. There is one centralized control register which all the components read at every state. This means that if we are to guarantee the absence of skew in the control signals, all of the components must handshake with the state register or signals derived from it at each step. In a scheduled circuit, which uses an FSM as its control path, the structure is also much more complicated than a simple pipeline, and differs for each implementation. There should be a faster and more structured way to pass data between components than simply implementing handshaking for transfering data.

1.2 Goals

- To find a method to convert a synchronous scheduled circuit to an asynchronous one with equivalent functionality and producing the same results
- This method should be systematic, repeatable, and suitable for future automation
- Determine if there are scheduling conditions which prevent this method from working, what they are, and how to get around them
- Execute this method on a simple scheduled filter, and demonstrate its results are equivalent to those in the original implementation using simulations
- Keep the circuit speed high, increase it if possible
- Circuit should be delay invariant to insure stability
- Determine the conditions for which this method is best suited and worst suited

Although part of the motivation for this project is minimizing power and taking advantage or delay variability, these are not the goals of this project. The project focuses on the development of a method to first desynchronize the system instead.

The method should be systematic and repeatable so that it can be used in further research and automation of the production of similar asynchronous circuits.

If there are conditions in the schedule which cause this method to produce incorrect results, they must be identified, as well as the reasons for their happening. If possible, solutions to fix these problems should be found and described in detail. In the same manner, conditions which adversely affect the performance of the circuit should be outlined and discussed.

1.3 Results

- Two related methods which desynchronize such a circuit by decomposing the central FSM into local, smaller FSMs which communicate with each other to advance state and send data
- Clock and clock enable signals are replaced by a network of handshaking controllers
- Three separate scheduling hazards have been identified which cause deadlock in the system
- Rescheduling solutions to fix these hazards have also been identified
- Sample implementations have been performed using each method, with successful simulations
- Asynchronous circuits found to be faster than the synchronous implementation in certain cases
- Scheduling is a big factor in determining circuit speed in the desynchronized circuits, corrections meant to avoid hazards are especially slow
- Speed is also affected by the difference in the length of operations, frequency of operations, number of inputs and outputs, and size of the circuit

The first of the two methods uses less area, has more logic that is used without custom design, and takes less work to implement, but is much slower and more susceptible to hazards than the second method. The first was conceived more directly from Cortadella's desynchronization method, and the second created in order to deal with some weaknesses in the first design method.

Scheduling based on synchronous circuits prevents the desynchronization methods from reaching their full potential. This scheduling creates hazards, and can limit the concurrency obtained from an asynchronous implementation.

1.4 Thesis Outline

The the following chapter will give a thorough background on all of the concepts used in this report which derive from the work of others, and explain how they relate to the project. If the reader is unfamiliar with any of the terms discussed in the introduction, they will all be thoroughly explained.

The third chapter is about two short experiments which were done on previously developed methods for desynchronization, the first manual, and the second method automated.

The fourth chapter focuses on the theory behind the solutions for desynchronization and the concepts developed in during this project.

The fifth chapter goes into detail about the implementations of the two different methods on the same synchronous circuit, how these implementations are different from each other and the synchronous circuit, and deviations from the theory in the fourth chapter.

The sixth and final chapter explains the results of the simulations and the conclusions derived thereof.

Background 2

The previous chapter provided a reasoning for the undertaking of this thesis, the goals set and results obtained, as well as an overview of the entire report.

The chapter will explain the following concepts:

- Synchronous circuits, clock signals, latches, registers, clock skew, and timing closure
- Asynchronous circuits, handshaking, Muller elements, Muller pipelines, and delay sensitivity
- Finite state machines and state variables
- Asynchronous finite state machines, bundled inputs, and the Aghdasi style state machine
- Scheduled circuits, scheduling, scheduling algorithms with and without constraints
- Scheduling for asynchronous circuits

2.1 Synchronous Circuits

The following section will explain some background about synchronous circuits, concepts, and design methodology. Synchronous circuits are the industry standard in digital circuit design. They use a special signal global to all components called the *clock signal*, which is used to synchronize all components in the circuit. This ensures that the data in the system is valid and that the different parts of the circuit receive the correct control signals at the same time. The clock signal has a cycle in which it charges and discharges and usually the components in the circuit are activated on the *rising edge*, which the clock signal charges from a logical 0 to a logical 1 although sometimes this happens on the *falling edge* instead, when the signal discharges, changing its logical value from 1 to 0.

A *latch* is a hardware component used to be sometimes *transparent*, having the output match the input, sometimes *opaque*, where output stays at its previous value. When the input cannot affect the output, the value stored in the latch is said to be *latched* or *clocked*. *D-Latches* have two inputs, a clock signal and a data signal. These latches are designed so that the output is held when the clock signal is low, and changed to match the data signal when the clock is high.

In order to store data temporarily, until the following clock cycle, synchronous circuits use a hardware component known as a *register* or *flipflop*. A flipflop is composed of two D-latches back to back with the second clock signal inverted, a fact which is used to an advantage later in this project. This ensures that the two latches will be transparent and opaque at complementary times. If the clock cycle is first high and then low, the input will be first transmitted to the output of the first latch when the first output has a high clock. Then the clocked output of the first latch will be will then become transparent through the second latch, and by the next rising edge will be clocked in.

This means that the flipflop is made to only allow the input which is to be stored in for a very short time, ideally as short as possible. This time is simultaneous with the rising or falling edge of the clock signal. While the electrical signals are propagating through the circuit, the input should be kept constant. When it has finished propagating and the circuit is stable, the next input can be clocked in and the process begun anew. This is how all of the circuit's parts can keep their data synchronized and valid. If not for using registers and a clock signal, design becomes much harder, because the designer must ensure the data is valid at any point in the system.

There are however, some resulting disadvantages to the synchronous approach to circuit design. These are mostly due to speed, power, and distribution.

The clock is calibrated for the longest possible delay path between any two latches in the system. All of the other paths in the system are not being traversed as quickly as is possible, and parts of the circuit are held up for extra amounts of time. Compared to the ideal solution, they are being used for some amount of extra time during which they could be used for another operation, or finished early. Any operations which are dependent on the output of a previous operation must stall until the end of the clock cycle, if all of their data is ready.

Having a universally distributed clock signal means that there is a lot of activity in the circuit. The clock must be charged and discharged at every component at every cycle. This requires a lot of unneeded power because some of the components are not actually in use at any given time.

A global clock can also cause *skew* in the circuit, which is the difference between arrival times of the clock signal to different components. This can affect the timing, and make it very difficult to ensure circuit data is valid in every situation.

Setup time is the name for the amount of time a value must stay at the desired value before a clock edge, and *hold time* is the amount of time it must stay at the required value after the edge. *Timing closure* is the term given to the state of a synchronous circuit where the delay in all paths added to the setup and hold times is less than the clock period. This takes a great deal of design effort, and is absolutely critical for the circuit to work correctly.

2.2 Asynchronous Circuits

This section will go over some of the basics of asynchronous circuits. Asynchronous circuits are those which do not use a clock signal to synchronize their components. This means that the delay between a change in the input and the subsequent delay of the change in the output of the system is dependent only on the intrinsic logic delay of the circuit. This means that in general, an asynchronous system will be faster than a synchronous system because fast components are not waiting for the end of the clock period to continue with their execution. Since a clock signal which runs through the entire circuit is not being charged and discharged at every cycle, there is also usually a large reduction in power. The downside of asynchronous circuits is that stability can arise in asynchronous circuits that are not designed properly and this can be simply solved with a clock signal. This makes the design more difficult and tedious. Also, designers are more used to the problems and design process of synchronous systems because they are the standard. Problems and design of asynchronous systems are less well explored.

Asynchronous circuits can be divided into different classes based on their sensitivity to delay. *Speed independent* circuits are those which will always function correctly regardless of timing. These are therefore the most robust and ideal, but their design is often very difficult, because for any input or bit that can change, the entire circuit must always be valid. *Delay insensitive* circuits are those which operate correctly due to delays that are bounded and greater than zero. These circuits are likewise extremely robust, because the delays keep data in the circuit from becoming invalid. *Delay sensitive* circuits are those which do not fall under either of these two categories. They must receive the correct inputs at the right times, or their data will become invalid. For this reason, they are not very robust, but they are usually the easiest to design and implement [14].

Handshaking is a method by which components with different timings, whether synchronous or asynchronous, can communicate with each other. The component sending data is known as the *master* and the one receiving data is the *slave*. The data which is communicated in this way is called *bundled-data* because it is sent and received in all data wires simultaneously. Two lines in addition to the data lines are used for the handshaking protocol, one controlled by the master called the *request* showing that it wants to send data. The other line is controlled by the slave and called the *acknowledgement*, showing that it has received data. There are two major methods of handshaking: *two phase* and *four phase*.

Two phased handshaking consists, as is implied of two phases. The first is the transmission of the request and the second is the transmission of the acknowledgement. Because there are only two phases, the lines alternate between going from 0 to 1 and from 1 to 0 in every other handshake. This method is faster, but since the master does not wait for the slave to receive data, problems can arise in the circuit if timings are not handled carefully.

Four phased handshaking is more commonly used, and likewise composed of four phases. The request and acknowledgement lines are both initialized to 0. When data is ready to be passed the first component sets the request line high, which is the first phase. Then when data is received, the acknowledgement line goes high, which is the second phase. When the first component sees this, it discontinues its request, that is the third phase. When the second component sees that, it likewise discontinues its acknowledgement in the fourth and final phase. This is illustrated in Figure 2.1[14].

The *muller C element* is a piece of hardware used extensively in asynchronous circuits. A Muller C element holds the value of its output, until all inputs are the same value, at which point the output becomes that value, as seen in Table 2.1. Using a Muller C element, handshakes with more than one input or output can be performed,

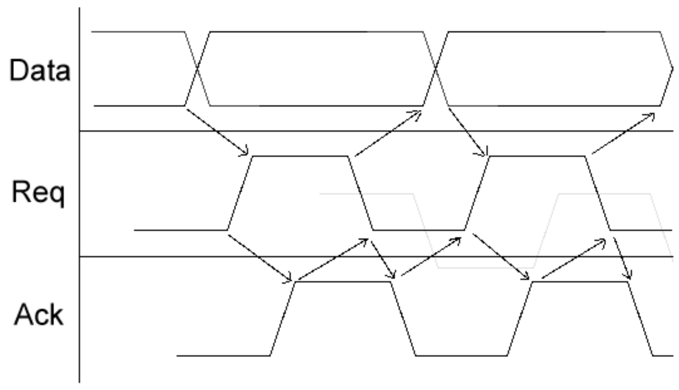


Figure 2.1: Four Phase Handshaking

because the element will wait until all handshaking components are ready for the next phase of the handshake. It can also be used in another way, to create dual rail asynchronous circuits. Since the output will wait until both inputs have changed to make a change, the system can be made stable until all data is valid[14].

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

Table 2.1: Muller C Element Truth Table

A *Muller pipeline* is a way to create an asynchronous pipeline of data using Muller C elements as a memory component. Data is passed to the next element of the pipeline using handshaking. The request of the previous element is put into a C element with the inverse of the acknowledgement from the next component, which means that the previous component has data ready and the next component is free. The output of the Muller C is then used as the request to the next component and acknowledgement back to the previous component. This frees the previous element and prepares the next element to accept the handshake, and in this way, the control signals are passed along the pipeline.

An asynchronous circuit's sensitivity to delay, handshaking, and Muller C elements and pipelines are concepts that are all used extensively in this project.

2.3 A Method for Desynchronization

This section will discuss methods of conversion from a synchronous circuit to an asynchronus one, and discuss in detail the method that is the inspiration for this project.

Desynchronization is the process of eliminating the clock of a synchronous circuit, and in so doing making it asynchronous. In order to make this conversion there are two principle implementations: *dual rail*, *bundled data*.

Dual rail systems use a *dual rail Muller pipeline*. Throughout the design, all of the signals in the design are implemented using two lines where there would be only one in an equivalent synchronous system. One of these lines signifies true, and the other signifies false. If both lines are zero, the data is invalid. If the true line is 1, then this is the equivalent of a logical one, and if the false line is 1, this is equivalent of a logical 0. Both lines being set to true is an error that should not occur. Every operation is performed on both signals, and is equivalent to something in the synchronous implementation. For instance, an OR gate will produce a 1 in the false line if the false signals of both two rail inputs are 1, and does this with a Muller C element. An or gate is used to test if the signal is valid, in order to send an acknowledgement, which similar to in a Muller pipeline, allows the transmission of new data into the pipeline stage. There is no request line but valid data in the two rails is the equivalent of having an active request and invalid data is the equivalent of having an inactive request. Effectively, this combines the control and data of the implementation. It takes a lot of time to redesign all of the components and it takes up a lot of area since the wires and logic have effectively doubled, but to its credit it is speed independent and does not require delay elements.

Bundled data implementations separate the data from the control lines. The data is passed through the pipeline through latches, which are controlled by a Muller pipeline. This is the basic version of the Muller pipeline, with two small additions. The output of the C element in every pipeline stage is also used as the clock input to a latch, which means that when a pipeline stage is "active", the latch is transparent. This allows data to be transmitted as the handshake moves through the pipeline. Also, a delay element must be used to match the combinational logic delay in the datapath between two latches [8]. This ensures the data is valid when it reaches the latch and classifies this circuit as delay insensitive. This delay element is difficult to implement, but this type of circuit is much more easily converted from a synchronous implementation and takes less area than the dual rail implementation.

The method used in this paper to desynchronize the circuit was devised by Cortadella et al.[5, 4, 2], and its results can be seen from comparing Figure 2.2 and Figure 2.3. One goal of this method is to minimise the amount of design effort in the desynchronization process. It is similar to the bundled data method, but it does not use C elements in the pipeline stages. The main advantage over the Muller pipeline is that the circuit has an asynchronous reset signal. Since the datapath logic is left intact, the simplicity of the synchronous design process can be retained, and since it is unrestrained by the clock, the speed and power advantages of an asynchronous circuit can also be acquired, all in the same circuit.

Starting with the synchronous implementation, which typically uses D flip-flops as

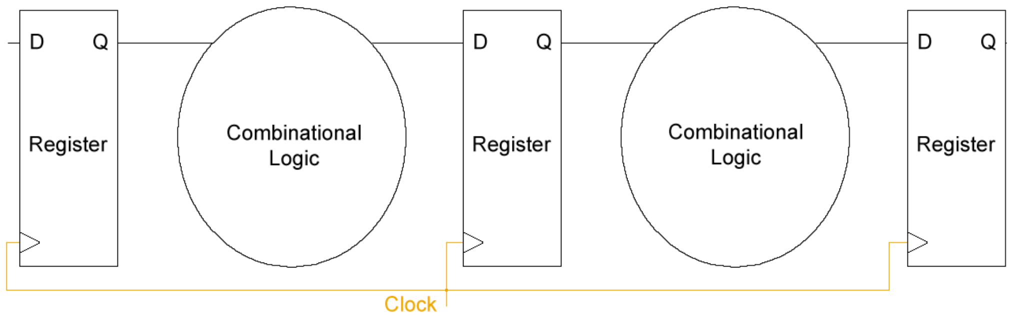


Figure 2.2: A Simple Pipeline Prior to Desynchronization. From [5]

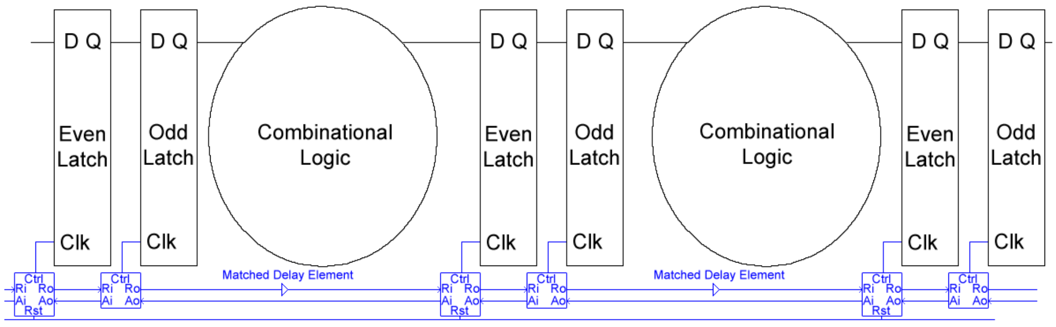


Figure 2.3: A Simple Pipeline Following Desynchronization. From [5]

registers which store information in a circuit, all of the flip flops in the design are split into two latches (a master and a slave), and the clock signals of all of these latches are replaced by control signals. Using this method, an asynchronous, open-source, high end processor known as ASPIDA was generated from the open source DLX processor.

Each latch has a controller as can be seen in Figure 2.4, to determine when this control signal should be high or low, which is analogous to the clock signal in the original circuit. The inputs and outputs to the controller are dual handshaking lines in and out and a reset signal. The controllers are slightly different between master and slave latches, so that the control signals into these latches are the complement of each other, ensuring that the latches are not both transparent at any one time. They are also designed such that only the master controllers need a reset signal. The circuits of both the master and slave controllers are shown in Figure 2.4.

The pairs of controllers are arranged so that every latch pair is handshaking with the latches to and from which it receives data. This creates a sort of handshaking network which mirrors the flow of data in the system.

When information is multiplexed from two or more registers into one, a Muller C element is used to group together all of the preceding adjacent request signals, because the system must wait for all of them to be ready to change the input to the next controller. The acknowledge signal can simply fan back out to all preceding acknowledge out ports. If we have a register that sends input to multiple adjacent registers, we do

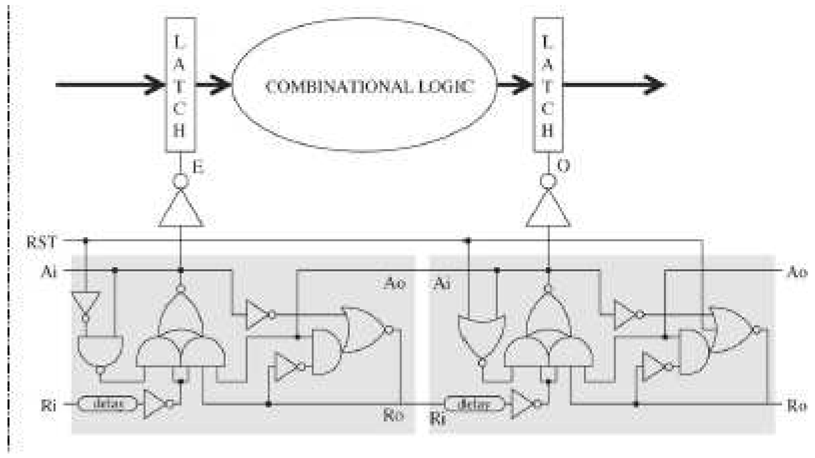


Figure 2.4: Cortadella Style Latch Controllers. From [5]

the opposite, fanning out the request signal and using a C element for the acknowledge signals.

It is critical to ensure that the control signal for a request does not arrive at the next latch faster than the data from the corresponding combinational logic, because then the next register could be clocking invalid data. A delay is inserted between the request out and request in of the next controller, equal to or longer than the slowest path through the combinational logic between the two latches. This is the delay seen in Figure 2.4 and Figure 2.3. The major perceived disadvantage to this method is the difficulty of calculation of the delay, the complications of realising that delay accurately in real hardware, and an inability to port the logic for the delay between various hardware mappings, routings, and technologies.

The handshaking protocol is of the four phased variety, very standard, simple, and straightforward. When a controller receives a request in, it sends a request out to controllers after itself, and an acknowledge in signal back to preceding controllers. When the request in goes low, acknowledge in can go low, and when acknowledge out goes high, request out goes low. The system then only has to wait for acknowledge out to be low before it can respond to another request in.

The logic of the latch controllers are often described using *Petri Nets*, which are a graph used to verify the correctness of asynchronous systems. These are diagrams which demonstrate the impact on the current values of a circuit of the change in one or more of the inputs, and the flow of data, by way of showing which handshakes need to be be completed in order to start a given handshake in the diagram. They are different from a state diagram in that branching indicates not a choice, but two separate dataflows. An arrow indicates a transition between these states, and a ”+” in the transition variable indicates a rising edge, and a ”-” indicates a falling edge. Dots are also sometimes used to indicate the starting position of data in the circuit [16].

2.4 Finite State Machines

This section will explain some basic concepts about finite state machines. Finite state machines are a method of controlling a circuit which performs different tasks at different points in time. A machine is split up into various states, which along with the inputs determine what the values of outputs are, and which state to move to at the end of the current one. When one state is over, the system moves to a new state, and in this way all of the states are connected in a network. The state machine moves to the next state with a trigger, which in a synchronous FSM is the clock signal, and in an asynchronous one more complicated and discussed in the next section. The machine uses a bitwise encoding, called a state variable from which it can identify which state it is currently in. A combinational logic network uses these and the system inputs to calculate an output, as well as the encoding for the next state, which becomes the new value of the memory element containing the current state at the trigger.

2.5 Asynchronous State Machines

Finite state machines are most commonly implemented synchronously, but it is possible to implement them asynchronously. Memories are used instead of flipflops to hold the current state. The relative security of the clock is not present, and thus there can possibly be critical races, oscillations, and static hazards. If these things are allowed to happen they can cause the state machine to function incorrectly [14, 10, 15].

There are however, several methods of transforming the circuit to work around these timing errors. One of these is to make a speed independent circuit. With this method, one must make sure all state transitions only change one bit in the state encoding. This ensures that there are only two possibilities for state while the transition is happening. The inputs also cannot change during the transition between states. Therefore inputs are only changed in bursts [14].

There have been several ideas which attempt to retain as many advantages as possible from both synchronous and asynchronous finite state machines. Some of these ideas include using delays in the clock path(similar to what is done in the desynchronization method described above), and using inertial delay elements, in which the output is only changed if it is held for a certain amount of time. These can make the state encoding problem less difficult and desensitize the state machine against changing inputs, but they suffer from other problems, notably increased risk of metastability, slower circuits (when the delays are unnecessary), and the difficulty in calculating accurate delays [1].

There is another way to implement state machines however, which is not sensitive to changing inputs and does not use delay elements, the structure of which is shown in Figure 2.6. Clock signals are generated for the memory elements using a sum of products equation of which input and current state combinations cause a $0 \rightarrow 1$ and $1 \rightarrow 0$ transition for outputs and the state variable. Each output and state variable bit has a latch pair. When a change has been made to the input or state, the change is clocked in the first latch. When the information is safely inside, the first latch becomes opaque, and the second latch transparent. The system then waits until the input has settled. The second latch is again made opaque and the first latch transparent, waiting

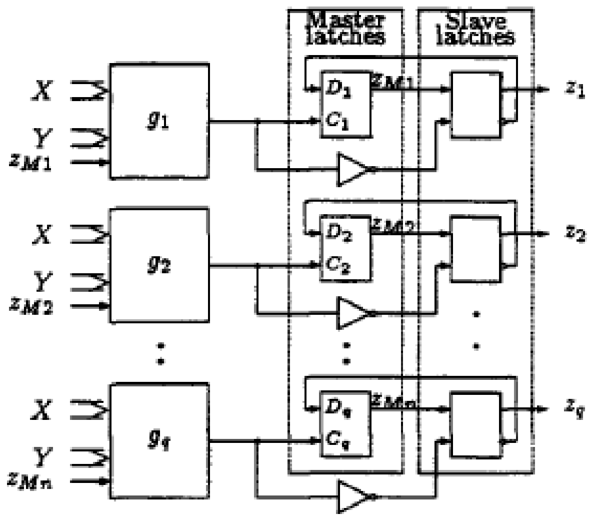


Figure 2.5: Output Latching in Aghdasi Style AFSM. From [1]

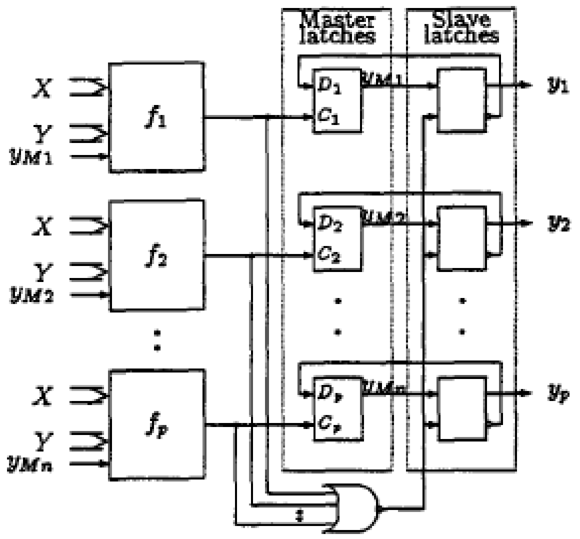


Figure 2.6: Next State Latching in Aghdasi Style AFSM. From [1]

for a change to state or input. When a state variable is wider than one bit, the system will wait until all bits are clocked into the second latch to reset the latch pair, by way of a common and gate. This will ensure that the machine does not move to a state to which it did not intend to go [1].

2.6 Scheduled Synchronous Filters

The circuit of focus in this project is a scheduled filter. A *digital filter* is a circuit which applies a mathematical calculation on a sampled signal and previous outputs, which amplify or attenuate the different frequency components in the signal. They are used in any kind of circuits that do signal processing and can be found in devices such as radios, DVD players, and cellular phones.

Scheduling is a process by which resources are allocated in a circuit to be used efficiently. A *scheduled circuit* is one which has been created in this way.

Every digital circuit takes some set of inputs, performs a function on them, and produces outputs. This function can be split into a set of smaller operations which when executed in a defined order, produce the same outputs. An common example of this is a mathematical function which can be split into additions, subtractions, multiplications, and divisions, but the method can be applied for any types of operations. The arrangement of inputs, operations, connections, orderings and outputs of the circuit is known as a dataflow [9].

If the same operations can be performed multiple times on different data within the dataflow, the hardware performing the operation can be shared, and ideally not much hardware needs to be added to multiplex between them and there will not be much contention and extra delay. If the dataflow is scheduled correctly, a circuit can be very quick and save a lot of area.

These shared, scheduled, hardware units which perform a defined operation are known as *computational units (CUs)* or *functional units*. Commonly they are *ALU units* which perform additions or subtractions, *MUL units* which perform multiplications, *DIV units* which perform divisions, shift registers, or simple registers.

Timesteps are used to separate each use of the computational units. A clock is used to set the length of a timestep in the circuit, and the cycle time is the length of the delay of slowest computational unit. A central finite state machine is used to send control signals enabling the different functional units in the circuit and multiplexing between the various data at the different states. The result is that at each state some of the computational units will be working, some will be off, and inputs and outputs will be routed in order to produce the correct data.

Calculating the optimal schedule is an NP problem, so heuristics must be used to find a circuit which has a low overall delay and/or a small die area, with a reasonable compilation time, depending on the priorities of the designer. Scheduling every operation to finish as early or as late as possible are two simple alternatives, and these are relatively easy to compute so they are sometimes used as bounds for the heuristic algorithm [9].

Some more complicated algorithms include list based scheduling and path based scheduling. List based scheduling assigns different operations different priority based on how many other operations are dependent on them, and how precious the resources they use are. Path based scheduling seeks to find the critical path in the circuit by determining what the longest string of operations is, and minimize the path in the circuit, by giving its operations first priority, and then scheduling the other operations [9].

There are other algorithms which work using constraints which limit the amount of each computational unit which can be used, such as integer linear programming, and force directed scheduling. Integer linear programming creates bounds using ASAP and ALAP times and the hardware constraints of the system. Then it uses algebraic substitution to satisfy all of these conditions and find an optimal schedule. Force directed scheduling seeks to balance the concurrency in the circuit to make all steps as

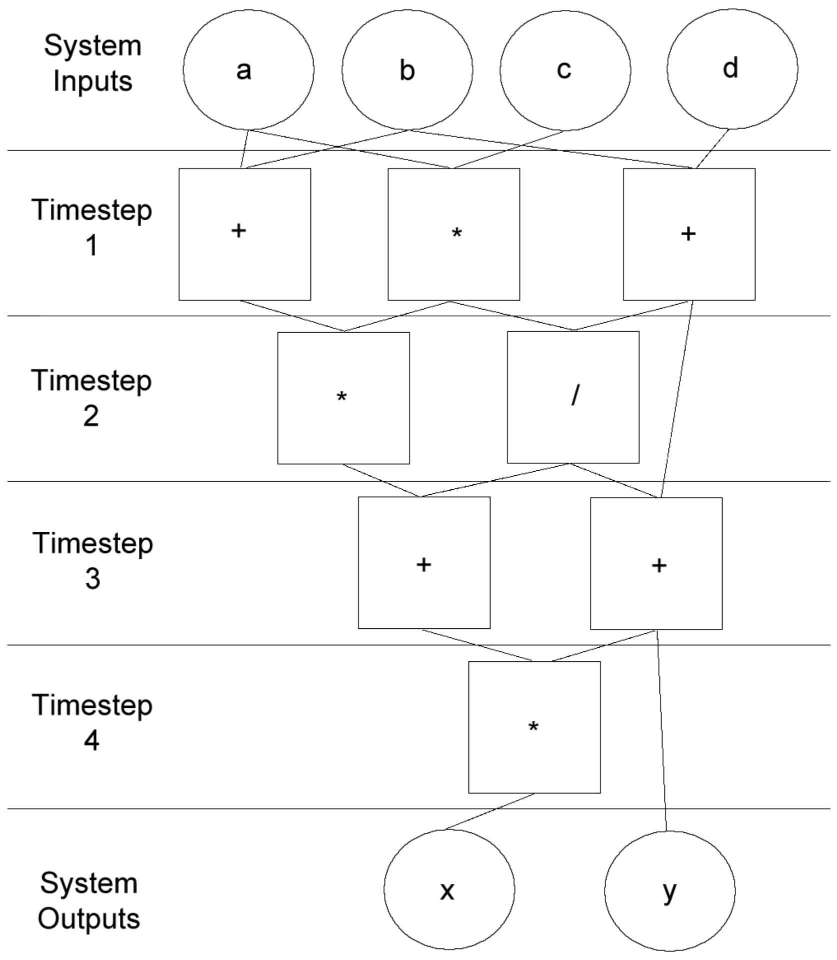


Figure 2.7: Sample Scheduled Dataflow with 2 ALU, 1 MUL, and 1 DIV units

concurrent as possible. First ASAP and ALAP are calculated to establish bounds, and then earliest and latest times are calculated for each component. Distribution graphs are taken to see how many of the different operations could used at different timesteps, and probabilities are established based on how many other options for scheduling that operation has. Every operation has a "force" calculated at every step within its bounds, calculated by the probability multiplied by the distribution. If all the forces in the circuit are added, the lowest total will be a circuit with the most concurrency. This is the solution that will be selected by the algorithm [9].

Scheduling is an important process for generating a large synchronous circuit that is run efficiently in terms of both time and hardware. There are many algorithms for scheduling, including as soon as possible, as late as possible, list based scheduling, path based scheduling, integer linear programming, and force directed scheduling.

2.7 Scheduled Asynchronous Circuits

There are also a few published papers about asynchronous circuits and how to schedule them effectively to reduce total time and chip area. This is somewhat more difficult to do since the timing intervals are completely continuous, compared to the discrete intervals in synchronous scheduling.

In a method published by researchers at the University of Aizu, two algorithms are used that are based on force directed scheduling and force directed list scheduling in synchronous scheduling. The starting times of the different operation instances are estimated, and this is used at every iteration of these algorithms to scheduling operations which communicate asynchronously. States are also separated into the periods of time in between the start times of all operations in the system [7, 12]. This approach is different than the one used in this project, and they could possibly be combined in the future to great effect.

The reader should now be familiar with all of the background information required to understand the methods of desynchonization and decentralization explained and implemented in the following chapters.

The next chapter will discuss two experiments which show the functionality of the desynchronization method explained in this chapter and its automation using a language constructed specifically for that purpose.

3 Experiments

In order to verify the feasibility of the desynchronization method, first some preliminary experiments were conducted. The first of these is a simple pipeline which is refitted with the latch controllers described in the desynchronization method above, in order to test the effectiveness of the method. The second test was using a tool, called Pipefitter, which can be used to describe asynchronous logic in a subset of Verilog.

3.1 Desynchronization Tests

The even and odd latches specified in [5] were coded in VHDL and simulated functionally using ModelSim, a tool to simulate the functionality of a circuit with user specified test inputs. An 'after' statement was inserted in place of the delay element. Even though there is no delay in the corresponding combinational logic, the circuit needs at least some delay to function, because there are circular dependencies. In a physical circuit, there is some propogatation delay so this is taken care of, but in a virtual circuit without a sense of timing, there will be a logical loop which ModelSim cannot resolve.

The pipelined circuit was coded with a simple (A+B)*C operation. In the first stage taking three input values A, B, and C, in the second stage adding A and B and carrying C, and in the last stage multiplying the values (A+B) and C and writing to an output register. The circuit is used to demonstrate that the asynchronous circuit will produce the same results as the synchronous circuit even with pipeline stages of varying lengths, as well as to allow easy verification of the correctness of the results in both implementations since the operations are arithmetic.

The pipeline was first tested in its synchronous implementation for correctness with a variety of inputs based on corner cases. Then it was desynchronized in the manner described earlier, replacing each flip-flop with two latches and implementing the odd and even controllers for each latch pair, which handshake with the controllers of the adjacent latch pairs, the last pair also handshaking with the first. Diagrams of the synchronous and asynchronous implementations can be found in Figure 3.1 and Figure 3.2.

After being desynchronized, the circuit was tested again. The output results of the synchronous and desynchronized circuits were observed to be equivalent (as well as correct arithmetically), indicating a successful application of desynchronization in this experiment. The results of the simulations can be seen in the figures Figure 3.3 and Figure 3.4.

3.2 Pipefitter Tests

In addition, the Pipefitter tool[3], which was developed at the University of Torino for use in the ASPIDA project, was tested with some examples to determine whether

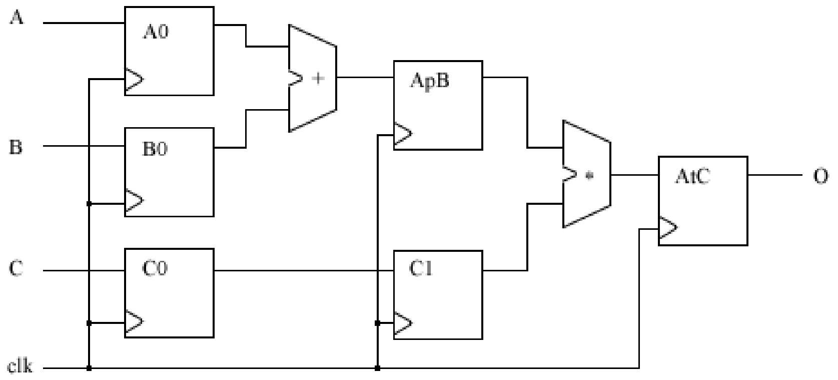


Figure 3.1: Design of the Pipeline Circuit Before Desynchronization

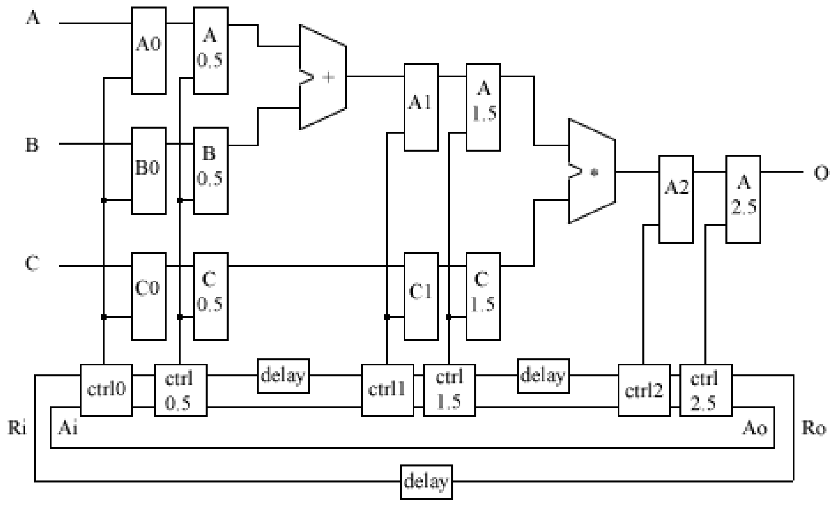


Figure 3.2: Design of the Pipeline Circuit After Desynchronization

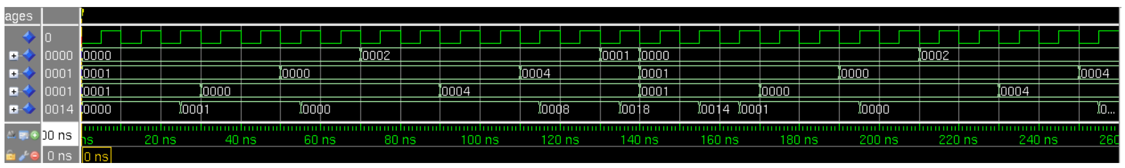


Figure 3.3: Results from the Pipeline Circuit Before Desynchronization

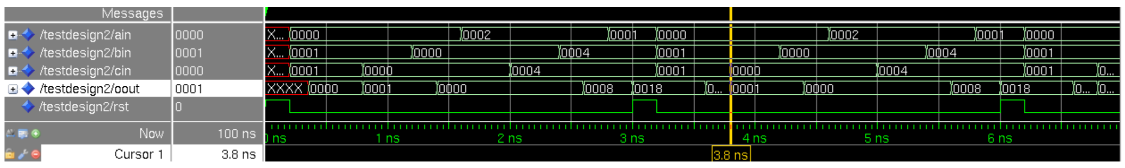


Figure 3.4: Results from the Pipeline Circuit After Desynchronization

it was suitable for generating asynchronous control blocks for use in desynchronized systems. Pipefitter has the ability to add timing information to the models which allows the designer to check the timing of the handshaking protocol in addition to the

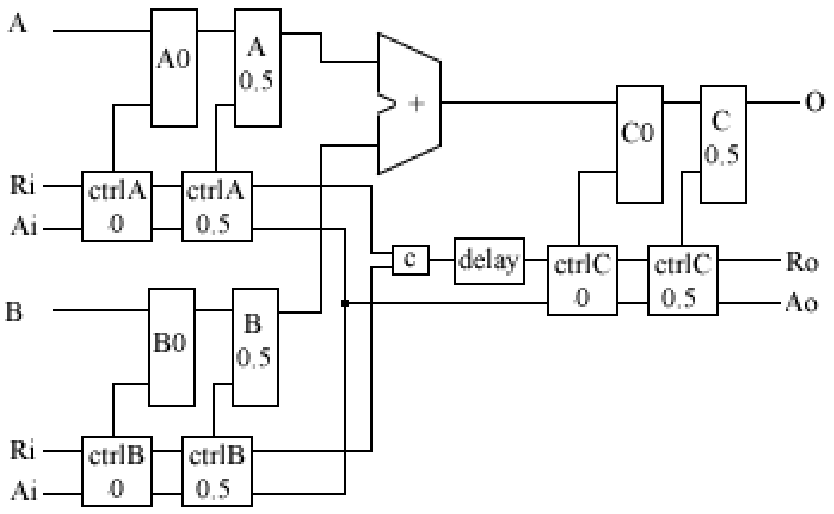
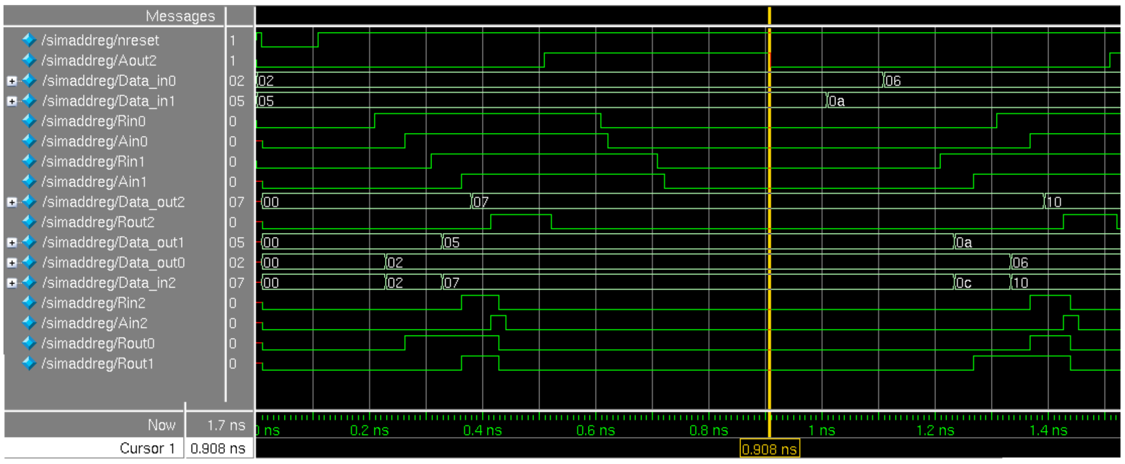


Figure 3.5: Pipefitter Circuit



```
Messages
/simaddreg/nreset       1
/simaddreg/Aout2        1
/simaddreg/Data_in0     02
/simaddreg/Data_in1     05
/simaddreg/Rin0         0
/simaddreg/Ain0         0
/simaddreg/Rin1         0
/simaddreg/Ain1         0
/simaddreg/Data_out2    07
/simaddreg/Rout2        0
/simaddreg/Data_out1    05
/simaddreg/Data_out0    02
/simaddreg/Data_in2     07
/simaddreg/Rin2         0
/simaddreg/Ain2         0
/simaddreg/Rout0        0
/simaddreg/Rout1        0
Now        1.7 ns
Cursor 1   0.908 ns
```

Figure 3.6: Pipefitter Circuit Results

functionality.

First the examples provided with the source code of a sequential register, an adder/-subtractor, and a concurrent register (which corresponds to the one we coded earlier in VHDL) were verified for correctness. Next a simple circuit was designed using instances of the concurrent registers connected in series, to test interactivity and interconnection of different registers. Lastly, an example with two registers which add their values was tested, to show the ability of the system to merge different asynchronous signals, but with the addition done functionally, without delay necessitating the calculated delay elements shown in Figure 3.5. All of these tests were successful, as seen in Figure 3.6, but logic for intercommunication between components had to be coded by hand.

3.3 Summary

- Verification that the desynchronization method works as described
- Verification that Pipefitter works as described
- Better to perform manual desynchronizations than to use an automated desynchronization process for the task ahead

Having performed these experiments, we can see that the desynchronization method does what it is supposed to and is powerful and straightforward for making asynchronous implementations of of existing circuits. Now we are ready to explore how to apply this technique in real, complicated circuits, and produce observable changes and results. As for the pipefitter tool, it proved to be not useful for applications in this project, since similar results can be achieved without the tool, and all of the base circuits in this project are currently implemented in vhdl, while the tool uses verilog, which requires rewriting. Since there is more freedom in devising a method by hand and something must be written by hand in any event, the decision was made to use manual conversion to realise the methods.

The next chapter will delve into the ways in which the synchronous circuit can be improved by desynchronization, ways in which to perform that desynchronization, adjustments to the Cortadella style desynchronization process, and the inner workings of the control network and state machines used in these methods.

Desynchronization Concepts and Solutions 4

The previous chapter discussed experiments and the validation of the desynchronization method presented in the background chapter.

This chapter will focus on:

- The ways in which desynchronization can benefit scheduled circuits
- Control skew from a central state machine, and how localization can solve this and further improve the circuit
- Required modifications to the latch controller to deal with logical changes, and to the delay element in order to keep the circuit fast
- Hazards that result in deadlock caused by conversion of a synchronous circuit to an asynchronous circuit, and how to avoid them
- Greater handshakes and how they create delays in the system
- The first solution, its approach to states, state templates, and handshaking, and connections between computational units
- The second solution, state clusters, and the way it removes dependencies between computational units

4.1 Inefficiencies in a Scheduled Synchronous Circuit

There are several ways in which a scheduled synchronous circuit is inefficient in using time and hardware resources. These are mainly due to the fact that it is synchronous.

Firstly, the clock is calibrated for the longest possible delay path between any two latches in the system. Faster parts of the circuit must operate at the speed of the slowest parts, due to the global clock signal. If all operations and all paths in the circuit are similar, the amount of wasted time can be negligible, but if there are large differences, the amount of wasted time can be a large percentage of the total time. In a standard ripple carry addition operation, there are N propagations through full adders, where N is the number of bits in the addition, and in an array multiplier, there are 2N propagations [11], meaning that the approximate delay difference of these two operations is the size of the addition. There are also many other common ways to implement the operations, so these differences can be more or less but are usually quite large. The delay for a division is even larger, and for bitwise operations smaller, and these are some of the most common operations in scheduled circuits.

The circuit, as all synchronous circuits also suffers from clock skew and excessive power dissipation by universal charging and discharging of the clock.

An optimal solution to the scheduling problem is not solvable, to our knowledge, in a polynomial amount of time. Therefore we must use heuristics, and while they usually find an efficient solution, it is also usually suboptimal, and can even be quite inefficient in some cases.

Also, all of the control signals must travel to all of the computational units in the circuit. If there are enough of them, they could be located far away from the central controller, and this can cause some skew.

4.2 Desyncronization and Decentralization of a Finite State Machine

The goal is to apply the desyncronization method to a generic scheduled digital filter which has a central state machine that is linear in nature, and can be run in iterations from start to finish. There are some static inputs which can are constant throughout an iteration of the circuit. There is also some number of reusable computational units suitable for scheduling which perform different or similar operations on inputs and generating outputs for use as inputs in other functional components. Every computational unit has in it a register which stores the result of the function's combinatorial logic.

In a centralized, synchronous finite state machine, each functional unit in the design is communicating at every step with the state register. Making the FSM asynchronous, using method described in [5] to change all registers into a network of handshaking latches, can only serve to make the system the same speed or slower than the optimized synchronous system, because the entire system is waiting for the state register to handshake with the computational unit with the slowest path as well as for all of the other units. Also, at every state these lines must be charged and discharged, which will dissipate a lot of power and counteract the benefits of using the method. These observations provide motivation to change this desynchronization method in such a way that there is less of a negative impact on circuit speed and power usage.

In order to have the chance of speedup in the circuit, one option is to separate the state machine into multiple independently clocked state machines which send requests and acknowledgements to each other and the functional units. The requests and acknowledgements will advance the states in this interdependent "state machine network". By separating the state machine into these smaller state machines, we do not have to wait for the entire clock period in the faster components, which corresponds to the combinational delay of the slowest functional units. Thus we can theoretically trim the difference in time between the longest path for the current component, and the longest path of the slowest component, each time an operation is performed.

In a software design, this is somewhat analogous to the procedure of separating a program into different parallel pieces which depend on information from each other, and will run simultaneously on different CPUs, interacting with each other using message passing. This is a common practice in embedded systems in order to effectively use multiple and often heterogeneous processors for separate tasks, in order to gain some speed up the entire system.

These smaller state machines do not have any knowledge about the execution of the others, and their only communication with the other FSMs is through handshaking.

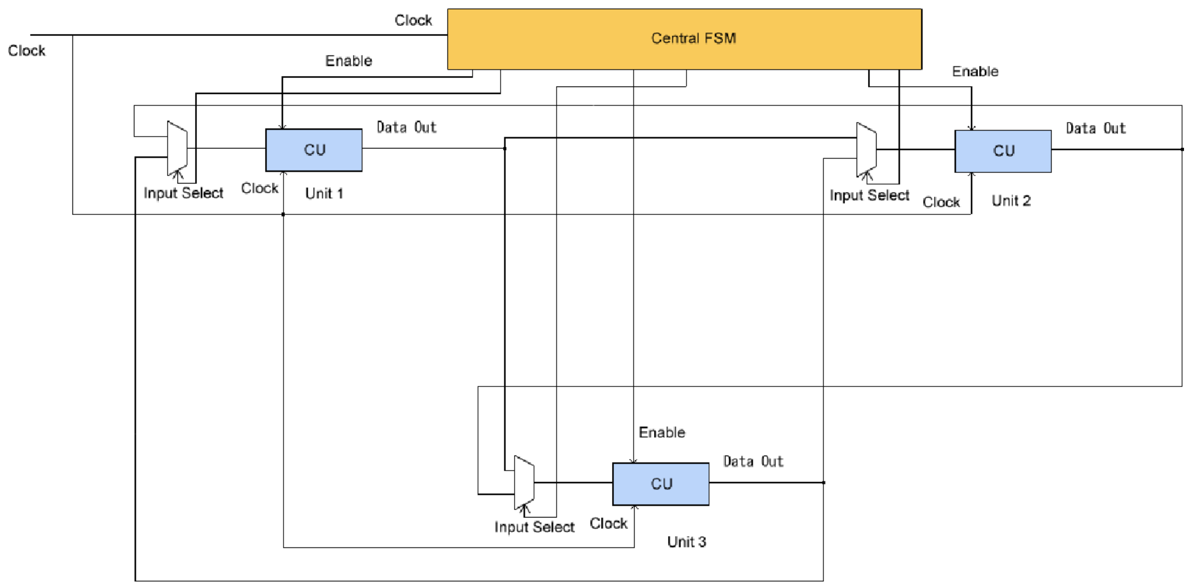


Figure 4.1: A Sample Scheduled Synchronous System

This is not a problem, because the global execution of the system is designed and determined at compile time, and there are no variations on execution. If the system is run once without error, it should run in the same way, without error, for every execution.

When splitting the state machines into smaller ones, how many is too much, and how many are too few? Which state machines will control which of the computational units in the circuit, and at which times? The most immediately obvious way to determine the number of state machines and allocate them responsibility for different functional units, is to create a separate state machine for every functional unit. Since they are asynchronous states, there is no clock to determine when one state is finished and the next state begins. Thus state transitions must happen asynchronously, as well. There is a method of doing this, mentioned in [1] which circumvents two major problems, namely, instability in the state machine due to input during a transition, and timed or significant delay. This method does, however, provide a sort of buffer for the input. That means that all possible combinations of inputs must be handled by the state machine, similar to a synchronous solution.

A diagram which shows the simplified structure of a scheduled circuit with three functional units can be seen in Figure 4.1. There is a central FSM, and this FSM controls the clock enable signals and selects which data to input to each CU at any given state. There is also a global clock signal which is seen by all components in the system.

4.3 Modifications to the Latch Controller

Both of the solutions presented in the following chapters make use of latch controllers, In the first solution, each functional unit in the circuit has an independent instruction state

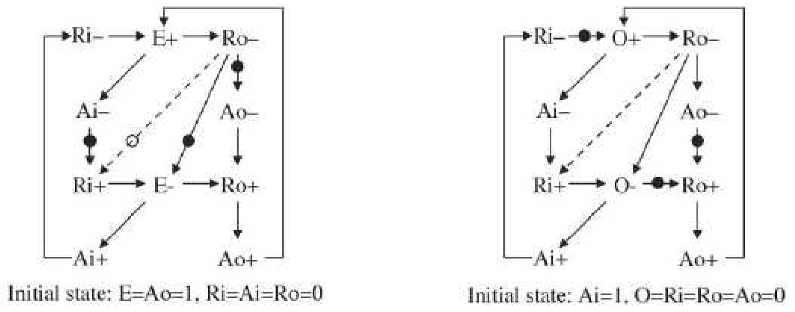

Ri–
E+
Ro–
Ai–
Ao–
Ri+
E-
Ro+
Ai+
Ao+
Initial state: E=Ao=1, Ri=Ai=Ro=0
Ri–
O+
Ro–
Ai–
Ao–
Ri+
O-
Ro+
Ai+
Ao+
Initial state: Ai=1, O=Ri=Ro=Ao=0

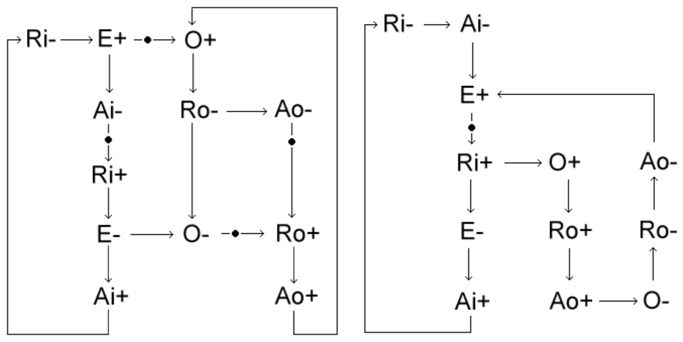
Figure 4.2: Petri Nets for Original Desynchronized Even and Odd Latch Controllers. From [5]

machine and a separate latch-pair controller based on the latch controller introduced in the background chapter. In the second solution, this latch-pair controller is embedded in the logic of the instruction state machine, but the functionality is the same.

The controller is slightly modified from the [5] design. The Petri nets for the original latch controllers are given in Figure 4.2. The controllers for both latches are combined into a single unit called a *latch pair controller* for ease of use, since they are always used together throughout the circuit. In Figure 4.3a is the combined and simplified Petri nets from Figure 4.2, and in Figure 4.3b is the version used in this project. E stands for the control signal of the even latch or master latch, and O stands for the control of the odd or slave latch. The handshaking outputs consist of: Ai, the acknowledgement in signal which confirms the capture of data from the sender latch pair controller, and Ro, the request out of the controller to the receiving controller. The inputs are: Ri, which is the request into the controller, and Ao, the acknowledgement to the CU when its request is seen by another. A "+" in the diagram represents a rising edge, and a "-" represents a falling edge.

The modifications are mostly to allow the latch pair controller to signal an FSM that is keeping track of an overall state when it should advance its state. In the original latch controller, the state of the input and output is only loosely connected because both handshakes are happening as soon as the other is finished, which there is no common point for the Petri nets to pass through where the state of the controlling FSM can be changed. Therefore in the modified latch pair controller there is a common entry and exit point in the Petri net in the E+ stage where the controlling state machine ends and begins states. The rest of the latch controller logic is adapted to account for this change.

A perhaps easier way to see the logic for the latch pair controller is displayed in Figure 4.4. If the latch pair controller is used as a separate entity, this acts as a state machine diagram for the controller. If it is not explicitly separate from other hardware, the logic controlling the latches still follows this scheme. The edges of the graph, which represent the transitions between phases in the handshaking, are influenced by the positive or negative changes in the two handshaking inputs to the CU and its pair of latches. The final state is necessary if there is a another controlling state machine. It must be visited because its controlling state machine must see both handshakes are completed to advance to the next state, and as soon as it advances, the latch pair



(a) Original Petri Net (b) Redesigned Petri Net

Figure 4.3: Petri Nets for Original and Redesigned Latch Pair Controllers

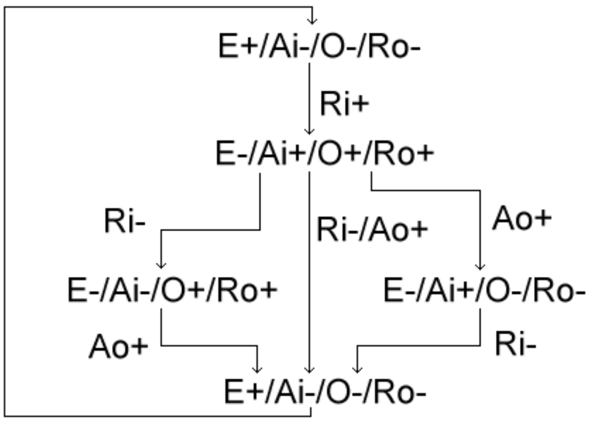


Figure 4.4: Modified Latch Controller Logic

controller will also be told to advance.

4.4 Delay Elements and Bypassing

In the original desynchronization method, the delay elements in the circuit are placed in the request in line of all latch pair controllers which in turn serve all of the computational units. This ensures that the request is delayed and unseen, to allow the combinatorial logic of the receiving CU to saturate before the result is saved in the latch.

The delay while the circuit is waiting for its input request to go through the delay element is given the term *input delay* by this report, and the delay of a sending CU while this request is going through the receiver's delay is known as the *output delay.* The output delay can usually be alleviated, because the logic can begin to saturate with its new inputs and request as soon as the input handshake is complete (see the section on input multiplexing options). The input delay on the other hand is necessary for the logic to saturate and data validity.

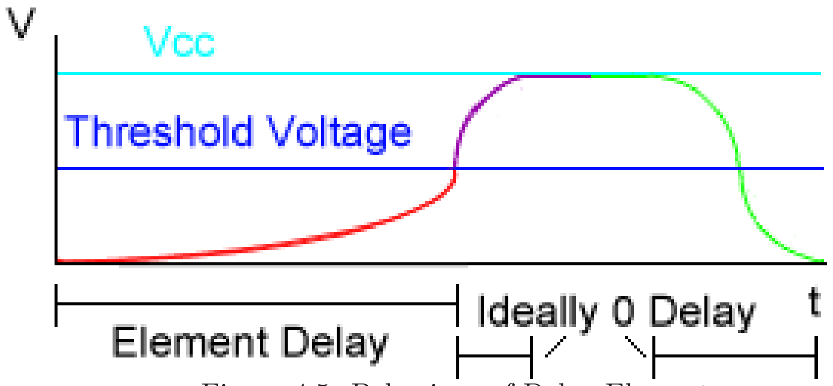


Figure 4.5: Behaviour of Delay Element

When the acknowledgement is sent by the receiver back to the sender, and the CU waits for the request to end, since the lowering of the request also travels through the delay element. This delay, prior to the falling edge of the request in, is no longer needed, because the logic has already saturated and the result is secure in the latch pair, and the sender can safely finish the handshake as quickly as is possible.

If this extra time is added, the circuit can be up to two times slower, since every handshake has twice the input delay and output delay. Thus in order to save time, we should use a delay element which has delay only until the threshold voltage on the rising edge. When the request is rising past the threshold voltage or falling, ideally there should be zero delay in the element. This ideal behaviour can be seen in the curve in Figure 4.5. The further from this ideal the delay element is, the slower the circuit will be.

4.5 Input Multiplexing Options

There are two separate ways to multiplex the inputs to each FSM in the circuit. The first way is simpler and takes less area, the second way is faster and takes more area.

The simpler option is to change the inputs which are being passed into the functional unit at each state transition. This solution is more stable, but it wastes more time than the alternative. This is due do the logic only propagating through the computational unit's logic after the state has started. The unit will have to wait for the delay of itself between the start of every state and the input handshake. On the other hand the logic for multiplexing the inputs into the computational unit is simpler.

The more sophisticated option is to feed the input for the next state into the functional unit as soon as the input handshake is finished, locking the computed data into the first latch, so that we can cut down on the time it takes between the start of the next state and sending acknowledgements to the input units. If all of the inputs have propagated through the logic by the time the next state starts, this unit can save time equal to its own delay. In general it will save the difference in time from when it has finished the input handshake and inputs are ready until when the output handshake is finished and the next state starts. This is the method used for this project because the increase in speed is quite significant over the simple method.

4.6 Hazards, Deadlock, and Redesigning the Circuit to Avoid Them

Depending on the synchronous design, situations can arise after the simple desynchronization which cause deadlock. This is due to the fact that in the synchronous implementation, these situations do not cause deadlock, and are not known to be avoided. Therefore, the asynchronous system must be slightly altered to avoid these situations and the ensuing deadlock. A properly designed system will never deadlock if as long as the components function correctly, since its execution path is predetermined, and the same every time. The fundamental cause of this deadlock is that there are one or more state machines which are involved in the same greater handshake in two different states. Since the transition from one state to the other is dependent on the handshake being completed, and every handshake in the greater handshake must complete before it will finish, there is a paradox. This happens in three different ways in a desynchronized circuit. In this report they are refered to as *crossover hazards*, *self-handshake hazards*, and *triangle hazards*.

If two or more functional units are in the synchronous design sending each other output over the same time boundary, it will create an interdependency between the output handshakes, and thus deadlock in the circuit. This phenomenon is referred to in this report as a crossover hazard and shown in Figure 4.6. The CUs are represented by squares, states and time are represented by the horizontal lines, and handshakes are represented by arrows between CUs. The handshakes causing a crossover hazard are coloured in red.

The simplest two CU problem lies in the fact that both of the units are at the end of their respective states, where they are trying to send their output, and need to finish before they can start a new state and accept new input. In a crossover deadlock involving more than two CUs, they will be connected in a circle, all of them trying to send requests to each other. A will be sending a request to B, B will be sending one to C, and C sending one to A.

One of the CUs will have to somehow finish first and accept input from the other CU in order to break the deadlock. One solution is to somehow reroute the output from the first unit to another computational unit. In the case of this implementation, there are CUs which are simple latch pairs with no other logic. We can use a latch pair which is not in use, or add one to the design in order to prevent the crossover deadlock. We can also potentially switch the operation in one of the computational units causing the deadlock to another which performs the same task and does not create a deadlock hazard with the other component.

Rerouting is however often a very unfavourable solution in terms of speed. The element through which the data is rerouted is often a simple latch pair, and fast, but the process destroys the concurrency of the scheduled circuit. The circuits which are rerouted all execute at the same time in the synchronous implementation, but in a rerouted implementation, they execute in a chain; first the one which has its output rerouted begins to execute and its sender is freed and begins to execute and so on.

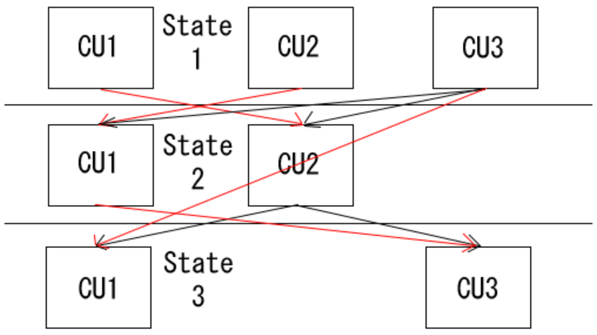


Figure 4.6: Deadlock due to Crossover Hazards

This is very costly in a hardware system designed to be concurrent. Therefore if it's possible, it is much better to switch the operation with that in another functional unit with no dependency.

Another problem arises when in the synchronous solution a functional unit sends output to itself. This is impossible in the asynchronous solution because the output handshake is at the end of one state, and the input handshake is at the beginning of the next, these two need to handshake with each other, and the machine cannot be in two states at once. This is called a self-handshake hazard in the scope of this thesis. As with a crossover hazard the data has to either be rerouted or another equivalent functional unit that eliminates this problem must be used.

A third potential deadlock problem happens when there are two or more components receiving an output handshake, and one of those is also receiving an input handshake dependent on the other. This is called a triangle hazard by this report and illustrated in Figure 4.7, with the handshakes between CUs represented by arrows, and those causing the hazard represented by red arrows. This will cause deadlock because the first handshake can never complete. One or more of the signals in the greater handshake is dependent on the completion of the same greater handshake. This means that one of the computational units is part of the greater handshake in more than one state, which is impossible. This problem can be solved by separating the output handshakes in the first state. This is performed by adding a resend data state in the first computational unit. This splits the greater handshake into two parts, and the CU which previously had two states in the greater handshake will now have one state in each, and the deadlock is avoided.

All of these hazards can be intertwined and combined with each other, and their manual correction and elimination can be very complicated and time consuming. The resulting fixes are often more linear in execution than the synchronous implementation, and this may lead to slow down the circuit. This is akin to fixing mistakes in a factory made circuit board by soldering it by hand. It is somewhat less elegant than a circuit which has been scheduled to be executed asynchronously in this manner.

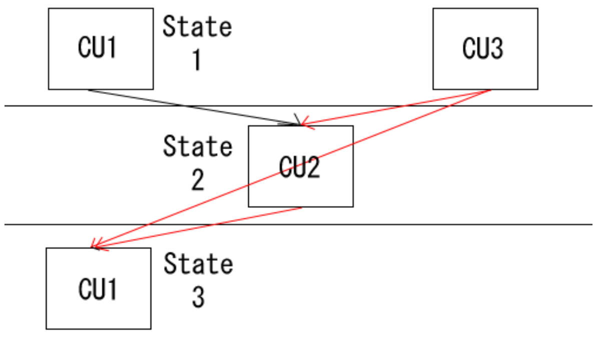


Figure 4.7: Deadlock due to Triangle Hazards

4.7 Separated Latch Pair Controller FSM Solution

In this solution, each computational unit has a *latch pair controller*, a separated asynchronous FSM which controls the handshake cycle at every state, and a likewise asynchronous *instruction state machine*, which has states corresponding with the states that the functional unit is active in the synchronous implementation. This instruction state machine must interface with the latch pair controller at every state, and multiplex the correct inputs and outputs.

Every state in the instruction FSM is a complete handshaking cycle of a pair of latch controllers. They each have one input handshake and one output handshake with all of the functional units' inputs and outputs at the state in question, although handshakes to outputs can potentially be separated (refer to the section on state templates). The FSM will move to a subsequent state when both the input and output handshakes in the current state are completed.

As for the latch pair controller, each cycle of states corresponds to one state in the instruction FSM. Each state in the latch controller corresponds with a possible phase in the input and output handshakes, and the state diagram is equivalent to the Petri Net in Figure 4.4. All of the inputs and outputs are multiplexed into and out of the latch controller with controls coming from the instruction FSM.

The results of the process on the system in Figure 4.1 can be seen in the picture in Figure 4.8. The central FSM is gone, along with the clock signal, and there are local instruction state machines and latch pair controllers. The units communicate with handshakes, and the FSMs determine which inputs and handshakes to accept and which units to send an output handshake at each state.

4.7.1 State Encodings

The latch pair controller, when separated, can use a simple and efficient encoding, as there are only 5 states, as seen in Figure 4.4. The first state will have the encoding of 000, the second, 001. When it branches, the two states will be 101 and 011. Then for the final state, it can be duplicated, with an encoding of either 100 or 010. These states both return to 000, for efficient coding and transition logic.

The encoding for the instruction state machine, which moves in a linear fashion, is

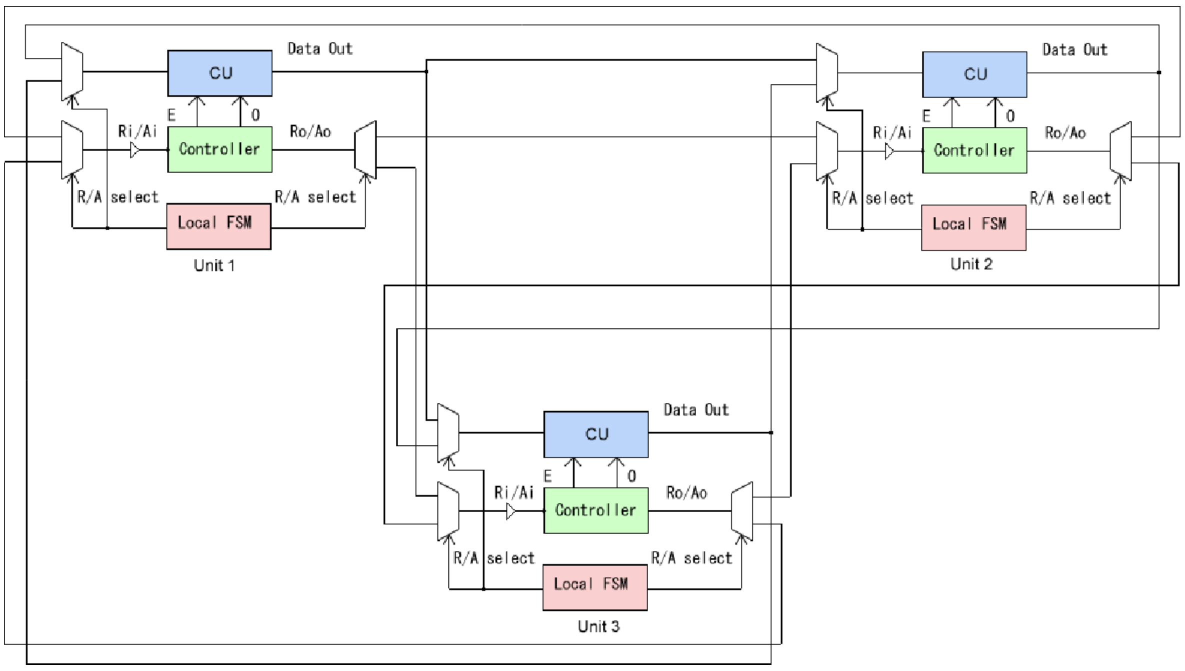


Figure 4.8: A Sample Desynchronized Scheduled System with Separated Latch Pair Controllers

even easier. To make things simple in the transition logic, each state will be one bit apart from its predecessor and successor. This is analogous to an algorithm visiting all the nodes in a hypercube without revisiting any of them. This ensures a similarly small amount of logic, which also means lower delay times. An example with three bits is:

000 → 001 → 011 → 010 → 110 → 111 → 101 → 100

000 is the encoding of the reset state, and 100 is the encoding of the done state. If the number of states is not a power of two, the state machine will just end at the largest encoding in this sequence. Because of the nature of the state machine we are using, it is not necessary to have a one bit transition, so when the system is reset to 000 it will be safe.

4.7.2 States and State Templates

The states that each functional unit is active in the synchronous circuit translate into states for that functional unit's state machine in this asynchronous version. In order to convert what happens in the synchronous state to states in the decentralized asynchronous machines there is a simple procedure. First, for every functional unit that is active (clock enabled) in the synchronous state, we make one state in the functional unit's state machine and find the inputs and the outputs. All of the states should stay in the same order they occur in the synchronous implementation. If they are time-dependent inputs or outputs, the unit must handshake with all of the units which provide the inputs or receive the outputs. Every state is one of four types, depending on the inputs and outputs: a *standard state* which uses both input and output hand-

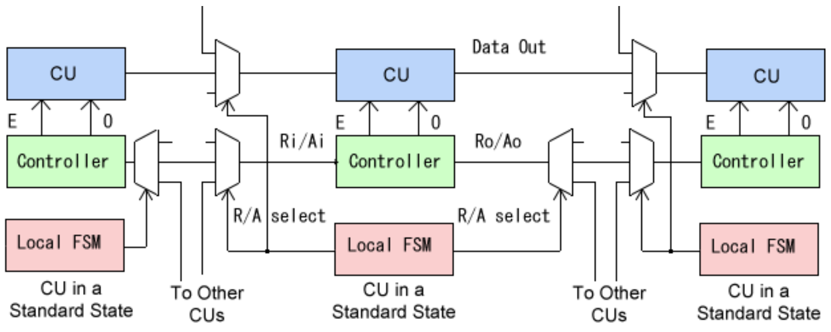


Figure 4.9: Standard State Template

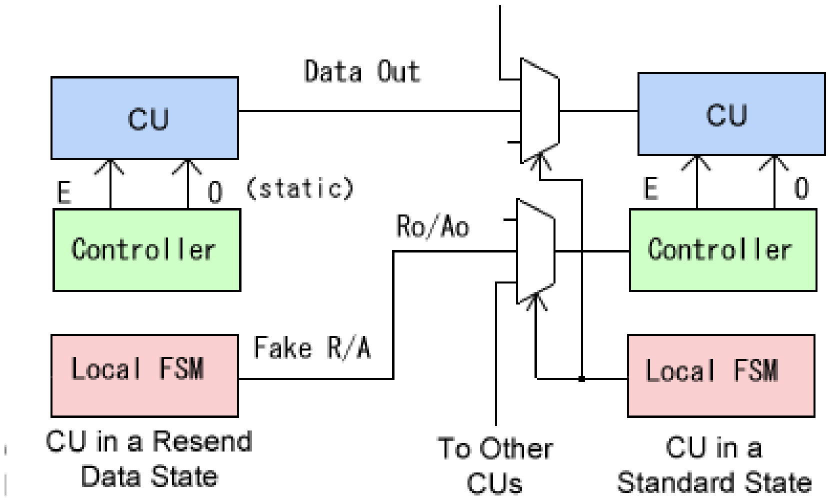


Figure 4.10: Resend Data State Template

shakes, a *data resend* state which only sends its data to another CU, a *fake request* state which only handshakes in its output, a *fake acknowledgement* state which only handshakes in its input.

The most common is a standard state which is used for a normal input-to-output dataflow through a functional unit. A diagram demonstrating the functionality can be seen in Figure 4.9. There is one input handshake with all required inputs, and an output handshake with some or all of the required outputs. The latch controller regulates the timing of the input due to the longest path delay through the functional unit, and stores the result in the second latch by the time the handshakes are finished.

The next most important type is a state to resend data, which is useful in cases where two CUs need to access the same data at different points in time, or when there is a dependency between outputs. Its picture is in Figure 4.10. In this template, the latch controllers are not changed at all, but the instruction state machine sends a fake request and reads the forthcoming acknowledgement. This state is used in cases where the output is sent at different times, and the input and result are thus unchanged. This is achieved by completely circumventing the latch controls, so the input is not stored by the latch pair.

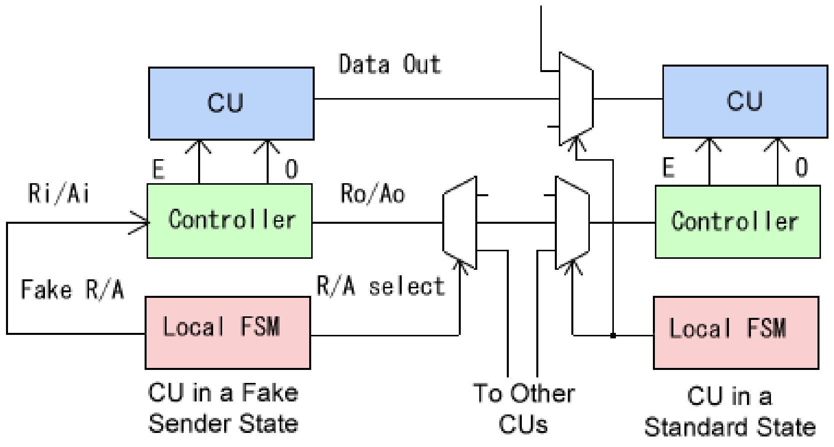


Figure 4.11: Fake Request State Template

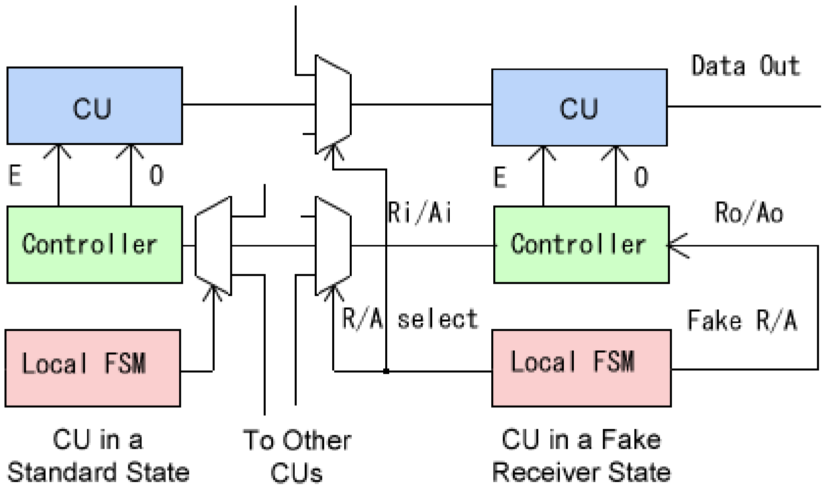


Figure 4.12: Fake Acknowledgement State Template

The third state type is used when none of the inputs are time-reliant, likely because they are predetermined before an iteration of the circuit. This template is shown in the diagram in Figure 4.11. The output is relayed by handshake as normal, and a fake request is sent into the latch controller and the acknowledgement is read by the state machine to set the request inactive. Like all requests in, the fake request must be delayed by the combinatorial logic time, because when a new state is reached, the new inputs still need to saturate the logic before the data is valid.

The last type of state is comparable to the third state type but used to store a result in the latch that does not have an output, usually in the case of a finishing state that calculates a final output. It can be seen in Figure 4.12. The input handshake is the same as a standard state. The request out is read by the state machine, and a fake acknowledgement is sent in to set the control of the second latch active. When the request goes low, the fake acknowledgement also goes low.

There are three types of fake latches that can be present in the finite state machine,

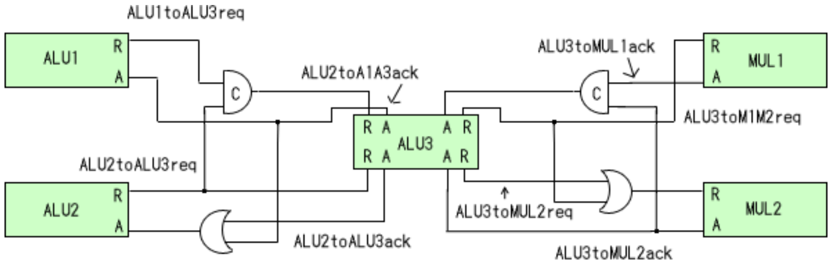


Figure 4.13: Connections Using Muller C Elements and OR Gates

corresponding to the three states other than the standard one. The first is a fake request out latch, which is used in the resend data state, and sends a request to the specified computational unit immediately at the start of the state. When that unit sends a corresponding acknowledgement back, the request goes low and the state ends. The next is a fake request in latch, which is used in the fake request state and sends a request into the current functional unit after its combinational delay in order to guarantee valid data. When the acknowledge is received the signal can go low, but the output handshake must finish before the state ends. The last is a fake acknowledgement out latch, which waits for the request out from its own computational unit to go high then sends a high acknowledgement signal. When the request goes low, the fake acknowledgement goes low again too, and waits for the input handshake to end to change state.

4.7.3 Connections and Communication Between Computational Units

For every dynamic output and input used by a functional unit, there are request and acknowledgement signals being multiplexed into every computational unit's latch pair controller, with the signals controlling the multiplexers coming from the instruction FSM. The latch controller can thus perform the same operations on different signals at different states, and handshake with only the functional units that are inputs or outputs at the current state. Each computational unit needs to know where to send its requests and acknowledgements, and also where to find them at any given state.

Different states for different components have differing amounts of inputs. For a computational unit state that has all static inputs, no request in is needed, for a state with one time-dependent input or output, a simple request and ack are needed. For those with more dynamic inputs or outputs, these signals must be somehow merged when going into the latch pair controller, and forked when going out of it, because there is only one input to and one output from the latch pair controller. For this task, a Muller C element can be used, because its output will only change to high when both inputs are high, and to low when both inputs are low.

The acknowledgements back to all of these components should be forked, which is done with a simple fanout. For multiple outputs, it is the opposite way: requests are forked, and acknowledgements are merged. A state that sends output to multiple functional units must send the same request signal to all of them, and merge the acknowledgements by receiving them through a Muller C element. This logic is demonstrated in Figure 4.13.

In order to guarantee no timing issues, each Muller C element should have its input signals separate from simple signals and other C elements. If this is not the case, then the output can potentially be the incorrect value. The problem arises in a very specific situation. One or more input signals to the C element are active, and waiting for a long time. The remaining signal or signals are active, but they are meant to be for another input into the receiving CU. This will occur at a state where the other signal is expected, and not the signal which gives us the problem. Since all of the inputs to the C element are active, the output will go high, even though the data is not completely ready. The output of the Muller C element will stay high, and as soon as the state expecting its output starts, the request will be seen as active, and the data entered into the receiving computational unit could be invalid. This happens rather infrequently, but there is a chance of it happening depending on the schedule of the circuit, and separation is the best means of guaranteeing valid data. Since the problem only arises with signals detected by the wrong C element, the signals entering a Muller C element should be separate outputs of their CUs, and there is no need to change the simple signals.

Detail	First Solution	+/-
Area	Smaller than second solution	+
Speed	Slower than second solution	-
Greater handshakes	Susceptible, causing delay	-
Deadlock inducing Hazards	All 3 - Crossover, self-handshake, and triangle	-
Reusable hardware	Latch pair controller and outer multiplexing	+
State count	Much less than second solution, similar to synchronous circuit	+
Multiple handshakes	Handled using C elements	NA

Table 4.1: Details of the Separated Latch Controller Solution

4.8 Greater Handshakes and Dependency Between Computational Units

In the context of the separated latch controller desynchronization method, can say that one computational unit is *directly dependent* on another unit if it is either part of an input or an output handshake. To advance to further states, the CU must complete this handshake, and thus it is dependent on the functionality of the other CU.

If there are two or more CUs which are involved in the same handshake to a computational unit, Muller C elements are used to merge either the acks or the reqs, since there is only one request input to the latch pair controller, and they become *indirectly dependent* on one another. Since these CUs can themselves be handshaking with other CUs, this creates a network of handshakes where all data must be ready before any of the handshakes can begin. All of the input handshakes and output handshakes must also be in the same phase of the handshake at the same time, and the network will wait for the slowest handshake. This network of coincidal and interdependent handshakes is called a *greater handshake* by this report. A diagram illustrating what constitutes

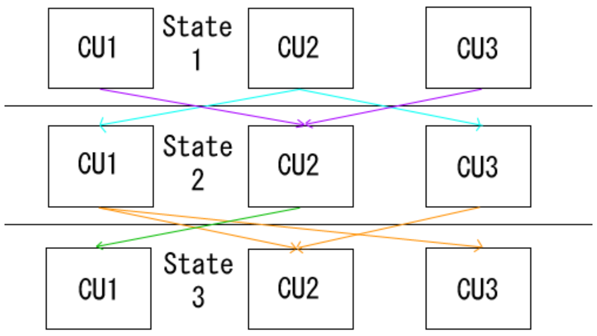


Figure 4.14: 4 Different Greater Handshakes

a greater handshake can be seen in Figure 4.14. Arrows in the diagram represent a handshake between two different CUs, with arrows of a similar colour representing a greater handshake.

Greater handshakes can make the circuit very slow. All of the functional units involved in the greater handshake must stall until the others are ready, even though they may be ready much earlier than the other units. If there are more signals and CUs involved in the same handshake, it will make the system much slower, because more and more of the circuit will be stalled, which can even affect CUs not in the handshake because they have to wait for its completion. The frequency and size of the greater handshakes in the system depend on the way that the synchronous circuit has been scheduled, which could potentially cause problems because the asynchronous implementations are not considered during the scheduling process. Greater handshakes can also cause the system to be even slower if there are larger differences between the logical delays of the components involved. The larger these differences, the longer the faster components will have to stall, and the longer this will tend to delay the execution time of the entire system.

This has major consequences on the timing of the system, and therefore is an important phenomenon to understand in the evaluation of these decentralized asynchronous circuits.

4.9 Combined FSM Solution

The second method is similar to the first method in that it uses handshaking and control signals to direct the flow of data in the system, and that each computational unit has its own local FSM controlling it. The difference is that the latch pair controller and instruction FSM are merged into one state machine which controls everything that happens to its computational unit. This merging is done with the goal of avoiding greater handshakes, and the resulting delays. In general, the resulting circuit is faster, but takes more area than the separated latch pair control circuit.

If this method is used on the system in Figure 4.1, the resulting system can be seen in the picture in Figure 4.15. Similarly to the first solution, the central FSM and clock signal are gone, and is replaced by single FSMs at each computational unit,

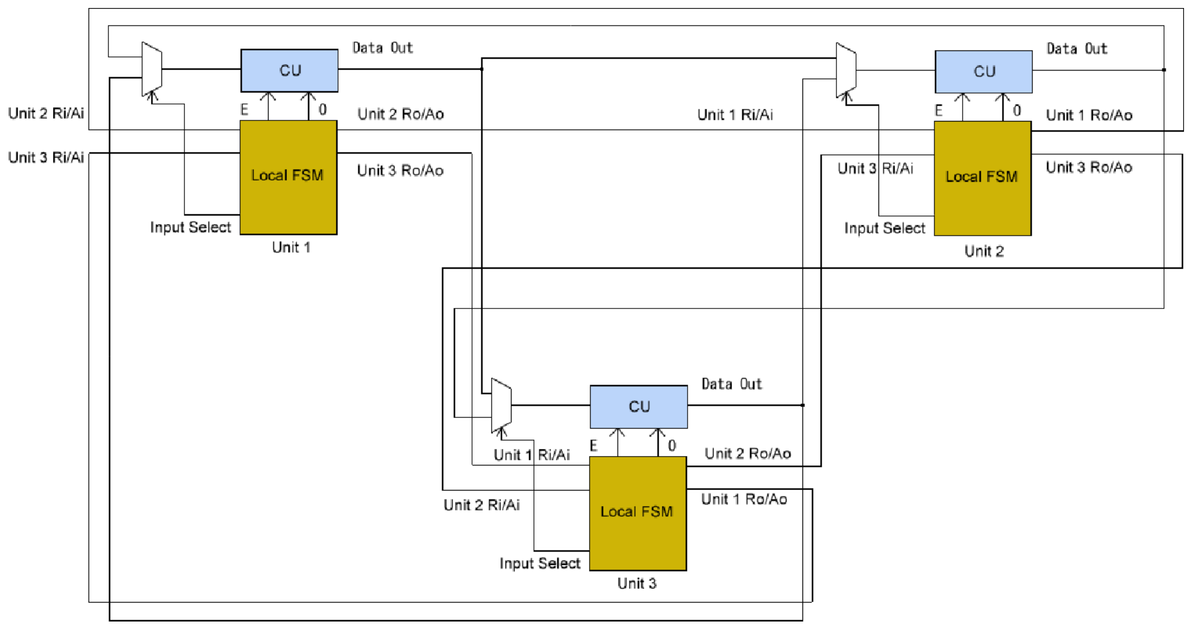


Figure 4.15: A Sample Desynchronized Scheduled System with Combined FSMs

which control the now present handshakes, although this is done inside the FSM, since different types of handshakes are done at different states, depending on the number of inputs and outputs. More of the logic controlling the inputs and outputs is also inside the FSM.

4.9.1 Handshaking State Clusters

In this method, the functionality of both the instruction state machine and the latch pair controller (which is itself also a state machine) is achieved using only one state machine. Needless to say, this state machine is substantially larger and more complex than the two state machines in the other decentralization method. One synchronous state at each computational unit, which one pair of input/output handshakes (or a single handshake if there is static input or output) corresponds in the first solution to a complete cycle of the latch controller, and one state in the instruction state machine.

In the combined solution, the input and output handshaking is represented as a cluster of states which begin with a common entry point of a waiting state, and end with a transition into the common waiting state of the next state cluster, or in the case of the last cluster, a done state. This pattern of states and transitions is always the same for a handshake pair which has the same number of inputs and outputs, and thus we can have a universal defined *handshaking state cluster* for a handshaking sequence with any number of inputs and outputs. With a small number of inputs and outputs, this cluster is small (with one output two states only), but it grows exponentially with the amount of inputs and outputs.

The number of possible states can be calculated mathematically, by the number of combinations. Every input or output in the computational unit can be waiting (low), active (high), or finished (low again). We have the further constraints that the outputs

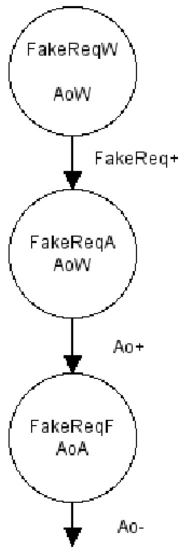


Figure 4.16: A Standard 0-Input 1-Output State Cluster

cannot be active or finished if any input is waiting. So the number of states in a state cluster corresponding to a synchronous state with n inputs and m outputs is:

$3^{n+m} - 3^m$

If there are 0 inputs, we must also use a fake request in to wait for the amount of time that it takes for the input data to proliferate through the logic, adding an additional state to do this.

A diagram of the state cluster with 0 input handshakes and 1 output handshake is shown in Figure 4.16, and one for the cluster with 2 input handshakes and 1 output handshake is shown in Figure 4.17. The requests in are represented by Ri, the acknowledgements out are represented by Ao. The state of the input and output handshakes are also represented by Ri and Ao, followed by a W, A, or F, which stand for waiting, active, and finished stages in the handshaking. Diagrams for more state clusters can be seen the the appendix.

Most of these states will never be visited in an iteration, but the cluster is complete in case of timing variations. It is possible that a designer or program which generates the circuit could safely cut some of the unused states out and make the state encodings and logic smaller and simpler, and in so doing speed up the state machine and reduce the overall design area. This will not guarantee delay insensitivity, however, which is the real goal of this complicated state cluster design.

4.9.2 State Encoding

Since we are able to calculate the number of states in each cluster, we can get an idea for how to encode states in the system. If the encoding is done well, it will save a lot of logic and therefore both area and time. A state encoding to reduce the amount of logic needed, and still keep the encoding fairly small is proposed.

First, we can use a "header" on the state encoding, to indicate which cluster the

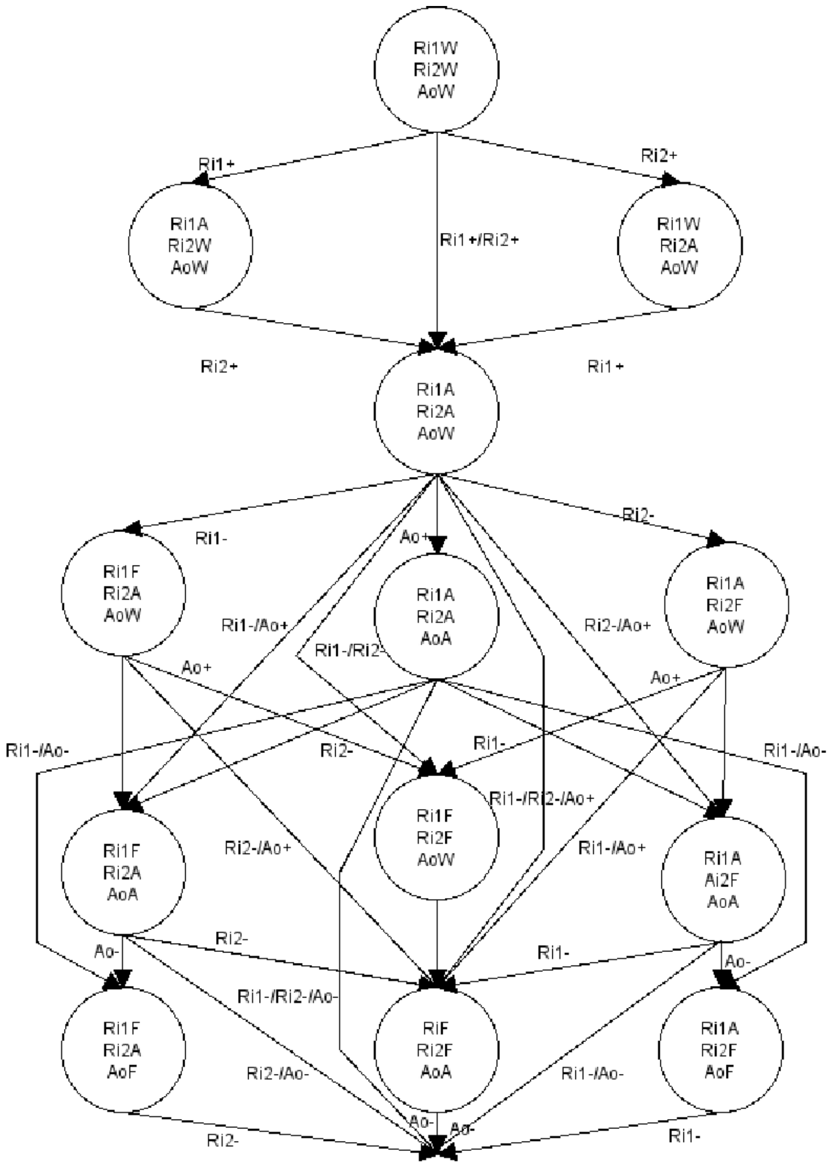


Figure 4.17: A Standard 2-Input 1-Output State Cluster

state is a part of. Next, we can use 2 bits for every input and output present in any state cluster, 2 bits since each of them has three possible values, waiting, active, or finished, and giving them separate encodings because this will provide parallel encodings between different state clusters, which should result in less complicated logic in the multiplexors, which can use the state header and the progress of all the handshakes to decide which inputs and outputs to use.

The result is that the encoding, where n is maximum inputs, m is maximum outputs, and c is number of clusters, will have the length of:

$log(c) + 2(n + m)$

which is rather large, but we do have its bounds. It should be efficient in terms of hardware, although compared to the first solution it will be more complicated.

4.9.3 Separating Greater Handshakes

In the synchronous and two FSM solutions, the functional units all have two latches to store the output after the operation is completed, but in order to reap the full benefits of the combined FSM implementation, we should make one latch for each of the inputs to the computational unit, and leave the other latch for the output. This increases the amount of latches that we need in the design, but it helps us because we can store the inputs at separate times, which has the effect that we can deal with them independently, unlike in the two FSM implementation. There is a drawback to this however, the output requests and input acknowledgements cannot be sent until all of the inputs are captured and have been allowed to saturate the logic in between latches.

One of the main advantages to doing things this way is that we can now separate all of the handshakes from each other. That means that as soon as both state machines are in the correct state, they can complete their transactions with one another. Thus, a state machine only has to wait for the state machine to which it is directly handshaking, and does not have to stall for any of the state machines in the greater handshake. This can speed up the circuit significantly, because the circuits don't have to wait for each other as much. We also do not need to use Muller C elements in between all of the computational units, and it cuts down on the number of signals going in between them because now instead of needing A1, A2, and A1A2 in signals, they only need signals coming from A1 and A2.

Also, since the handshakes are separated, there is also no problem with triangle hazards, because the handshake which would otherwise cause deadlock can be partially completed and the FSM can move on and complete the rest of the handshake in future states. This also eliminates the need for resending data, as each of the dependent FSMs can acquire the data independently. Again this speeds things up in the circuit, because no FSM is waiting for another, except for the one with which they are handshaking directly.

The designer must still watch out for crossover hazards, however, because the FSMs are still stuck trying to send data and finish their state before they can advance to another state and receive each other's data.

4.9.4 Further Expansion

In theory, this state diagram could be expanded even more, and each functional unit could have more input latches to take even more inputs, to separate these handshakes from the flow of the machine, but making this many states causes the amount of possibilities to balloon, which makes the state transfer logic larger and the state machine slower, although how much slower depends on the case in question. This is a tradeoff for which the designer can decide what the best design is.

This method also requires more customization. More of the logic is repeated because the different state clusters are performing similar logic to the others, whereas in the two FSM implementation, the latch controller performs the core of the handshaking logic over and over again at every state, and does not have to deal multiplexing and how many inputs or outputs there are. The state encoding could also have a very drastic effect on the speed and area of the circuit. If good state encodings are used, the number of bits can be lower, and more importantly, making the combinations more consistent between state clusters, which will result in simpler logic determining outputs and next state logic for the system. Since the amount of states will already be large, minimizing this extra logic is essential to the efficiency of this method.

The FSMs will be constructed in the style described in [1]. Since the state variables will be larger than those in the separated latch pair controller implementation, that means they will also have a longer state transition. If this difference is enough, it could potentially be slower than the separated latch pair controller implementation. In general however, the advantage of the combined FSM solution is speed.

Detail	Second Solution	+/-
Area	Larger than first solution	-
Speed	Faster than second solution	+
Greater handshakes	Avoids them, allowing more concurrency	+
Deadlock inducing Hazards	2 kinds - Crossover, self-handshake	+
Reusable hardware	Inner multiplexing logic, state encoding from compiler	-
State count	Many more than first solution, synchronous circuit	-
Multiple handshakes	Handled using state logic	NA

Table 4.2: Details of the Merged FSM Solution

The next chapter will discuss the original synchronous implementation and the two asynchronous implementations, what changes have been made, where the implementations depart from the theory, why, the state schedules of the FSMs in all three implementations.

Implementation of the Desynchronized Circuits 5

In the previous chapter the reasoning behind the decentralization of the finite state machine, the workings of the latch pair controller and delay elements, and the theory behind the two desynchronization methods was explained.

This chapter will explain the workings of the synchronous implementation, and explain the asynchronous implementations based on the theory presented in the last chapter. It will also give the schedules for all three implementations and thorough steps for performing the two methods on a synchronous circuit.

5.1 Filter Specifics of Original Synchronous Implementation

The implementation of this method was performed on a filter with three ALU units, two MUL units, and five free registers. By default, registers/latch pairs have a 2 ns delay, for ALUs it's 6 ns, and for MULs 10 ns, which includes the latch pair embedded in these components. The width used for the internal data path of the circuit is 10 bits. The synchronous state machine is 8 states long. The first two ALUs are the units which are the most busy, the first to start and the last to finish, and are only unused in a single state. One of the MUL units is also fairly busy, and the rest are relatively unused, only enabled in one or two states. The scheduling information is displayed in Table 5.1, Table 5.2, and Table 5.3.

State	I/O	REG1	REG2	REG3	REG4	REG5
1	Input	-	-	-	-	-
	Output	-	-	-	-	-
2	Input	-	-	-	-	-
	Output	-	-	-	-	-
3	Input	-	-	-	-	-
	Output	-	-	-	-	-
4	Input	ALU2	ALU3	ALU1	-	-
	Output	ALU1	ALU2	-	-	-
5	Input	-	-	-	ALU2	-
	Output	-	-	-	-	-
6	Input	ALU2	ALU3	-	-	-
	Output	-	-	-	-	-
7	Input	-	ALU1	-	-	ALU2
	Output	-	-	-	-	ALU1, ALU2
8	Input	-	-	-	-	ALU2
	Output	-	-	-	-	-

Table 5.1: Register States in the Synchronous Filter

State	I/O	ALU1	ALU2	ALU3
1	Input	input, I2A	I3B, I2B	I3C, I2C
	Output	MUL1	MUL2	MUL1
2	Input	-	-	-
	Output	-	-	-
3	Input	input, MUL1	I2A, MUL1	I3B, MUL2
	Output	REG3	REG1, ALU1	ALU1, REG2
4	Input	ALU3, ALU2	I2B, MUL2	I3C, MUL1
	Output	MUL1	REG4	ALU1
5	Input	ALU3, input	I2C, MUL1	-
	Output	MUL1	REG1	REG2
6	Input	REG1, MUL1	REG2, MUL1	-
	Output	REG2	REG5	ALU2
7	Input	input, MUL1	ALU3, MUL1	-
	Output	-	ALU1, ALU2, REG5	-
8	Input	ALU2, REG5	ALU2, REG5	-
	Output	-	-	-

Table 5.2: ALU States in the Synchronous Filter

State	I/O	MUL1	MUL2
1	Input	-	-
	Output	-	-
2	Input	A1A, ALU1	A2B, ALU2
	Output	ALU1, ALU2	ALU3
3	Input	A2C, ALU3	-
	Output	ALU3	ALU2
4	Input	-	-
	Output	-	-
5	Input	A1B, ALU1	-
	Output	ALU1, ALU2	-
6	Input	A1C, ALU1	-
	Output	ALU1, ALU2	-
7	Input	-	-
	Output	-	-
8	Input	-	-
	Output	-	-

Table 5.3: MUL States in the Synchronous Filter

The file which contains the datapath and the different computational units is inside of a separate wrapper file which clocks inputs and outputs, and tells the circuit when to start and finish. The filter is reset with a reset signal before the first iteration begins. Then a start signal is sent to the controller to begin the filter's operation. A done signal is sent to the circuit wrapper when the circuit has finished all working states and moved into it's done state. A last state signal is also sent when in the final state before

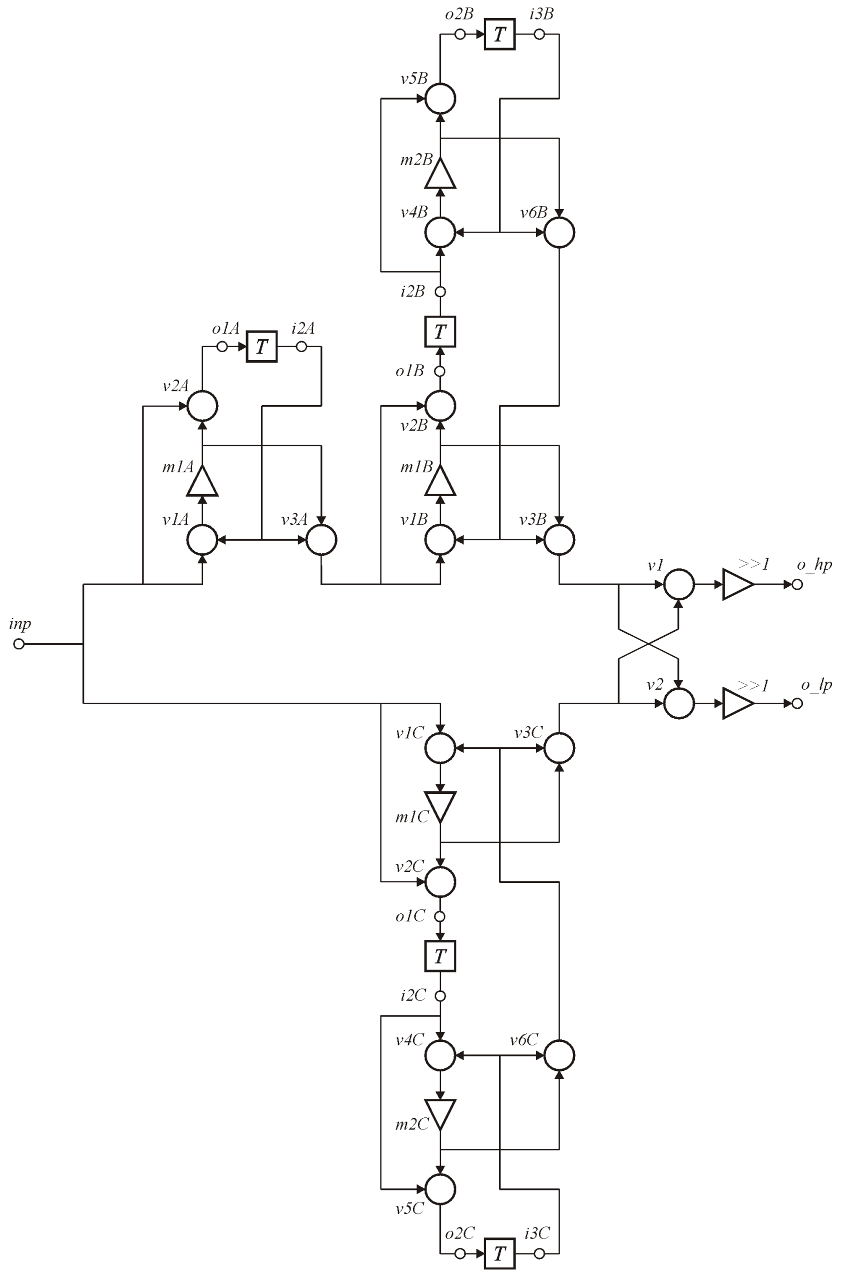


Figure 5.1: Circuit Diagram of the Synchronous Filter

the done state, and the outputs are all clocked in this state except for one which is clocked in the previous state. The filter has a set of set of static inputs, some of which are set to the values of static outputs after every iteration of the filter, some of which stay constant. There is also an input value. Outputs are divided into the true output of the system, and those that are used to determine the result of the next iteration in the filter. The circuit is described in Figure 5.1.

The asynchronous circuits use the same delay estimates for the different functional units, although several other combinations were experimented with and simulated to ensure robustness and timing invariability. The synchronous circuit uses clock enable signals, whereas in the asynchronous versions the schedules are already determined by the local state machines and current handshake partners, so there is no need for these signals, and they do not have them.

5.2 Two State Machine Implementation

In the asynchronous implementation, many of the units are more active, including ALU3 and the MUL units, due to resend data states.

Several changes were made to original the synchronous dataflow due to deadlock, the first to avoid crossover between ALU1 and MUL1, rerouting through REG5. Using a register that had not been written to is a simple change, and due to the register's low latency, one which does not slow down the circuit too much. The second was to avoid a triangle with MUL1, ALU1 and ALU2, and the third was to avoid ALU2 sending data to itself. The method was performed such that the asynchronous decentralized implementation is faster than the synchronous implementation. The synchronous circuit takes 262 ns, and the asynchronous implementation is 240 ns. Any given unit is at most 7 states in the asynchronous implementation and some of the registers are only one state long. The states and scheduling are shown in Table 5.4, Table 5.5, and Table 5.6.

State	I/O	REG1	REG2	REG3	REG4	REG5
1	Input	ALU2	ALU3	ALU1	ALU2	ALU1
	Output	ALU1	ALU2	-	-	MUL1
2	Input	ALU2	ALU3	-	-	ALU2
	Output	-	-	-	-	ALU1, ALU3
3	Input	-	ALU1	-	-	-
	Output	-	-	-	-	-

Table 5.4: Register States in the Two State Machine Asynchronous Filter

The biggest way in which this implementation differs from the theory is the connections between units. Not all of the signals are separated for each computational unit, only in the case that they conflict with other signals. There was only one of these in the entire circuit, one signal and Muller C element between MUL1, REG2, and ALU2. ALU2 was receiving a separate signal from MUL1, but since REG2 was requesting to the ALU at the same time, the Muller C output went high, and it appeared as though both were ready when the ALU went into its' next state, when MUL1 was still starting its new state and did not have the correct output latched.

In all of the other signals in this implementation, which do not conflict with each other, another configuration can be used. This other configuration uses OR gates and C elements to connect the acknowledgements and requests for all signals going between two functional units. The functional unit with an input in the C element does not know about it, and does not distinguish between this signal and those to other C elements or simply connected signals. The functional unit which receives the output of the C

State	I/O	ALU1	ALU2	ALU3
1	Input	input, I2A	I3B, I2B	I3C, I2C
	Output	MUL1	MUL2	MUL1
2	Input	input, MUL1	I2A, MUL1	I3B, MUL2
	Output	REG3	REG1, ALU1	ALU1, REG2
3	Input	ALU3, ALU2	I2B, MUL2	I3C, MUL1
	Output	MUL1	REG4	ALU1
4	Input	ALU3, input	I2C, MUL1	-
	Output	REG5	REG1	REG2
5	Input	REG1, MUL1	REG2, MUL1	-
	Output	REG2	REG5	ALU2
6	Input	input, MUL1	ALU3, MUL1	A2, R5
	Output	-	ALU1, ALU3, REG5	-
7	Input	ALU2, REG5	-	-
	Output	-	-	-

Table 5.5: ALU States in the Two State Machine Asynchronous Filter

State	I/O	MUL1	MUL2
1	Input	A1A, ALU1	A2B, ALU2
	Output	ALU1, ALU2	ALU3
2	Input	A2C, ALU3	-
	Output	ALU3	ALU2
3	Input	-	-
	Output	ALU2	-
4	Input	A1B, ALU1	-
	Output	ALU1	-
5	Input	-	-
	Output	ALU2	-
6	Input	A1C, REG5	-
	Output	ALU1, ALU2	-

Table 5.6: MUL States in the Two State Machine Asynchronous Filter

element as input has separate input signals for the outputs of the Muller C elements and the simple signal.

5.3 Process Overview for Desynchronizing and Decentralizing a Filter Using the Dual State Machine Method

In order to execute the two state machine desynchronization method on the synchronous circuit, perform the following steps:

1. Find all states for each functional unit where that unit is active, and the functional units to which it inputs and outputs. Gather all states for every functional unit

and make state machines using a linear progression for all of the states that the CU is active in. If the machine is in the done state and there is a start signal, it should move to the very first state in the diagram. If there are crossover or triangle hazards, correct them by rerouting or splitting output states.

2. Create input and output request and acknowledgement lines connecting the functional units, and multiplexed into a latch controller. Use C-elements to connect these handshakes when the unit has two or more dynamic inputs or outputs in the current state. Static inputs and outputs don't require a separate handshake, but a fake handshake must be used to store data if no other functional unit is handshaking at the appropriate time.

3. Divide the states into the different state types and add a fake outward request, fake inward ack, or fake inward request latch for the state machines which have data resend, fake acknowledgement, and fake request states respectively. To increase circuit speed, requests in and data in should be multiplexed after the input handshake is completed and until the next input handshake is completed, but the input ack and output request and ack should be multiplexed by state only.

4. When input and output handshakes are both completed or in the case of a resend state, the ack is received, the current state should change to the next one.

5. Split the done and last state signals into separate signals for the different state machines, namely the ones from which the output is clocked in the wrapper of the synchronous circuit. If computational units providing final output have been changed due to hazards in the asynchronous circuit, include the signals from the corresponding state machine instead.

6. In the wrapper, change from clocking data using the edge of the clock and instead change when the appropriate last state or done signal is active. Set the start signal to activate the state machines at the beginning of every iteration. For the master done signal all done signals can be anded together, or just the ones which will be the last to finish.

5.4 The Combined State Machine Implementation

In this implementation, there are several differences from the theory. First of all, the state clusters have been modified to only be sensitive to the last phase of the output handshake at the very end (waiting for all of the acknowledgements out to go low). This offers a significant reduction in the amount of states needed in each cluster, especially when there are a lot of outputs. All of the functional units receiving data during this time are also not dependent on the state of the sending functional unit. Therefore the receiving units can also finish their handshakes and are thus not waiting at all, only the functional unit sending data must wait. See the state diagram in Figure A.6 which illustrates the simplified two-input one-output state cluster used in this implementation, and compare with Figure 4.17 in the previous chapter, which illustrates the complete state cluster.

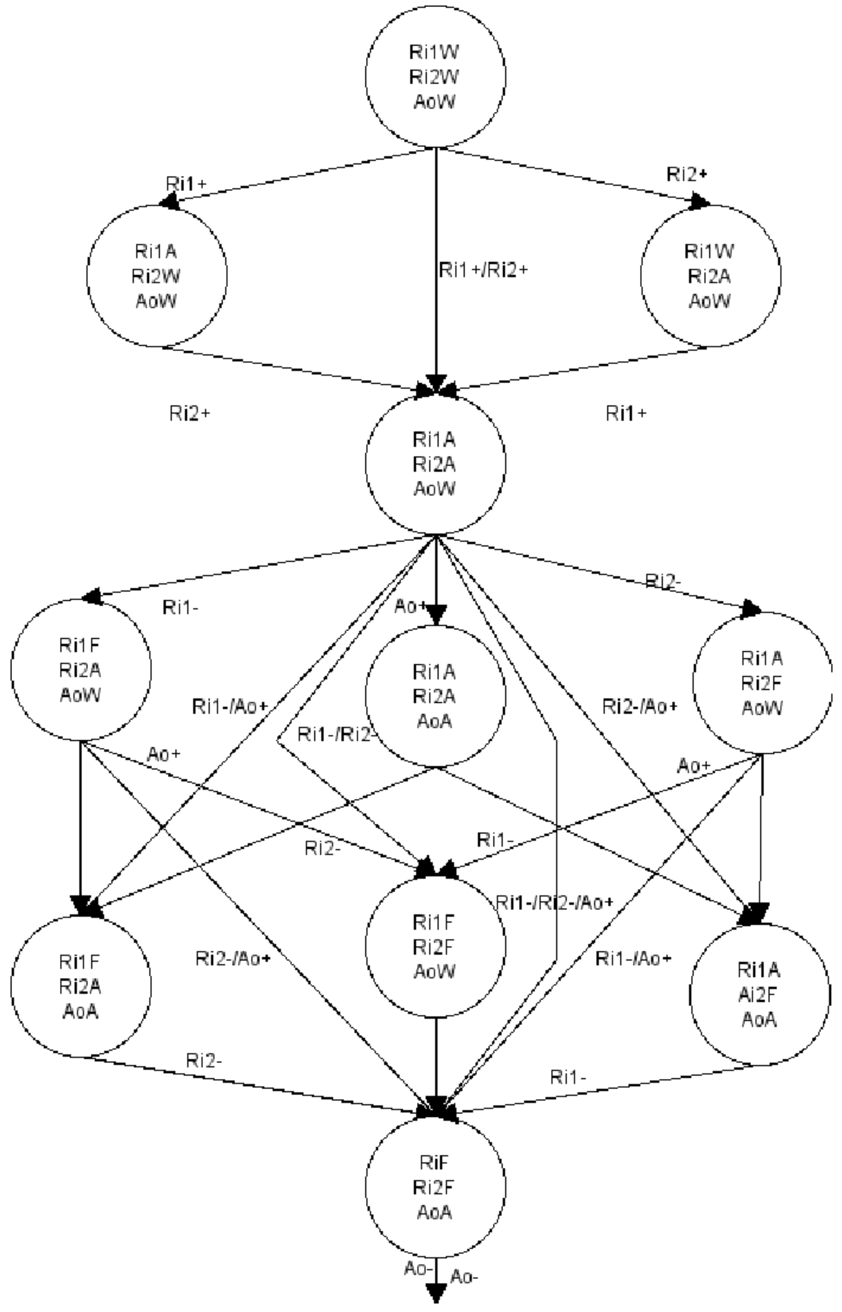
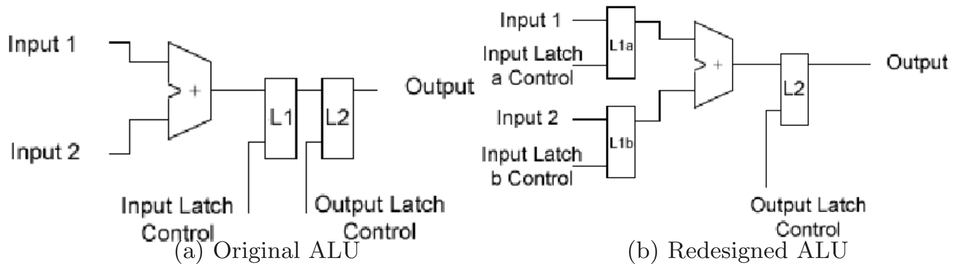


Figure 5.2: A Modified 2-Input 1-Output State Cluster

There are some changes to the ALU structure so that each of the inputs can have its own latch. Giving each input its own latch means that the input handshakes can be separated. There are therefore three clocking signals into it, one for each of the input latches, and one for the output latch. The new design is displayed in Figure 5.3b, in contrast to the original shown in Figure 5.3a. The input to the output latch is calculated through an adder from the outputs of the two input latches. The MUL design is unchanged because there are never two simultaneous dynamic inputs into



(a) Original ALU (b) Redesigned ALU

Figure 5.3: Latching and Structure in the Original and Redesigned ALUs

either of the MULs, and in likewise in the REGs (which only ever have one input at a time).

The scheduling is mostly the same as the separated latch pair implementation, since the combined FSM also has problems with self-handshaking and crossover hazard. The difference is that the data resend states are all rejoined into the state where their data is latched. This can only make the circuit faster because the CUs receiving output do not have to wait for any other CU, they only have to wait for the sending CU. The schedules are shown in Table 5.7, Table 5.8, and Table 5.9.

State	I/O	REG1	REG2	REG3	REG4	REG5
1	Input	ALU2	ALU3	ALU1	ALU2	ALU1
	Output	ALU1	ALU2	-	-	MUL1
2	Input	ALU2	ALU3	-	-	ALU2
	Output	-	-	-	-	ALU1, ALU3
3	Input	-	ALU1	-	-	-
	Output	-	-	-	-	-

Table 5.7: Register States in the Combined State Machine Asynchronous Filter

5.5 Process Overview for Desynchronizing and Decentralizing a Filter Using the Combined State Machine Method

In order to execute the combined state machine desynchronization method on the synchronous circuit, perform the following steps:

1. For every state that a CU is active, take note of the inputs and outputs. Create a state cluster from the template for the number of inputs and outputs involved in the original, synchronous state, and replace the generic inputs and outputs with those involved in this state. Repeat this for all states in the computational unit, and for all of the functional units in the circuit.

2. Design the state machine to move between the clusters in a simple linear progression. If there is a reset signal, the machine should move to the done state. If the

State	I/O	ALU1	ALU2	ALU3
1	Input	input, I2A	I3B, I2B	I3C, I2C
	Output	MUL1	MUL2	MUL1
2	Input	input, MUL1	I2A, MUL1	I3B, MUL2
	Output	REG3	REG1, ALU1	ALU1, REG2
3	Input	ALU3, ALU2	I2B, MUL2	I3C, MUL1
	Output	MUL1	REG4	ALU1, REG2, ALU2
4	Input	ALU3, input	I2C, MUL1	A2, R5
	Output	REG5	REG1	-
5	Input	REG1, MUL1	REG2, MUL1	-
	Output	REG2	REG5	-
6	Input	input, MUL1	ALU3, MUL1	-
	Output	-	ALU1, ALU3, REG5	-
7	Input	ALU2, REG5	-	-
	Output	-	-	-

Table 5.8: ALU States in the Combined State Machine Asynchronous Filter

State	I/O	MUL1	MUL2
1	Input	A1A, ALU1	A2B, ALU2
	Output	ALU1, ALU2	ALU3, ALU2
2	Input	A2C, ALU3	-
	Output	ALU3, ALU2	-
3	Input	A1B, ALU1	-
	Output	ALU1, ALU2	-
4	Input	A1C, REG5	-
	Output	ALU1, ALU2	-

Table 5.9: MUL States in the Combined State Machine Asynchronous Filter

machine is in the done state and there is a start signal, it should move to the very first state in the diagram. In the case of crossover or self-handshaking hazards, correct them by adding new components, rescheduling, or rerouting data.

3. Connect the state machines using simple request and acknowledgement lines between those that communicate. Add delay elements at the inputs to a state, the delay should take place after the input signals are multiplexed into the circuit.

4. When input and output handshakes are both completed or in the case of a resend state, the ack is received, the current state should change to the next one.

5. Split the done and last state signals into separate signals for the different state machines, namely the ones from which the output is clocked in the wrapper of the synchronous circuit. If computational units providing final output have been changed due to hazards in the asynchronous circuit, include the signals from the corresponding state machine instead.

6. In the wrapper, change from clocking data using the edge of the clock and instead

change when the appropriate last state or done signal is active. Set the start signal to activate the state machines at the beginning of every iteration. For the master done signal all done signals can be anded together, or just the ones which will be the last to finish.

A myriad of simulations designed to show different properties of the circuits were performed on both of the implementations, and the next chapter will give details about the results of these simulations, and discuss the impact of these findings as well as future work related to this project.

Conclusions 6

The last chapter discussed the implementations of the two desynchronization methods, their schedules, and steps to complete them.

This, the final chapter will discuss the simulations done on these implementations and the results obtained from those simulations. It will then proceed to discuss the results and how this knowledge can be used to compliment the use of the desynchronization methods in the future, and possible extensions of the research done in this project.

6.1 Simulations and Results

Simulations of the VHDL code in ModelSim are used to test the circuits for both correctness and to evaluate their timing.

The original filter has a clock frequency of 20 ns, and takes 80 ns of time to set up prior to an iteration. Each iteration is finished after 9 states. That means, if there is no slack, and the circuit's speed is optimal, every iteration takes 9δ where δ is the delay of the longest path in the circuit.

We can find the run times of the asynchronous implementations by the time at which the done signals of all the FSMs in the circuit are active. With the subtraction of the setup time of the circuit, (80 ns), the real speed of the circuit can be obtained. This real speed can then be compared against the optimal synchronous circuit execution time.

Simulations were run for both of the asynchronous implementations using several different combinations of ALU and MUL timings, and calculated for the synchronous circuit using the above formula. For each of these tests, the latches each had a delay of 1 ns, so together a latch pair or register has a delay of 2 ns. The stated MUL and ALU values are not including this time, so the true delay of the entire MUL or ALU CUs is the MUL or ALU delay added to 2 ns. The simulations can be seen in Table 6.1.

MUL	ALU	Synchronous Circuit	Separated Latch Control Circuit	Combined FSM Circuit
8	4	90	127	84
8	30	270	493	340
50	4	468	295	247
50	30	468	481	419
17	17	171	306	219

Table 6.1: Execution Times in the Three Implementations (ns)

Each of the different tests is meant to illustrate a different point about the different asynchronous versions and the effects of timing, scheduling, and frequency of operations.

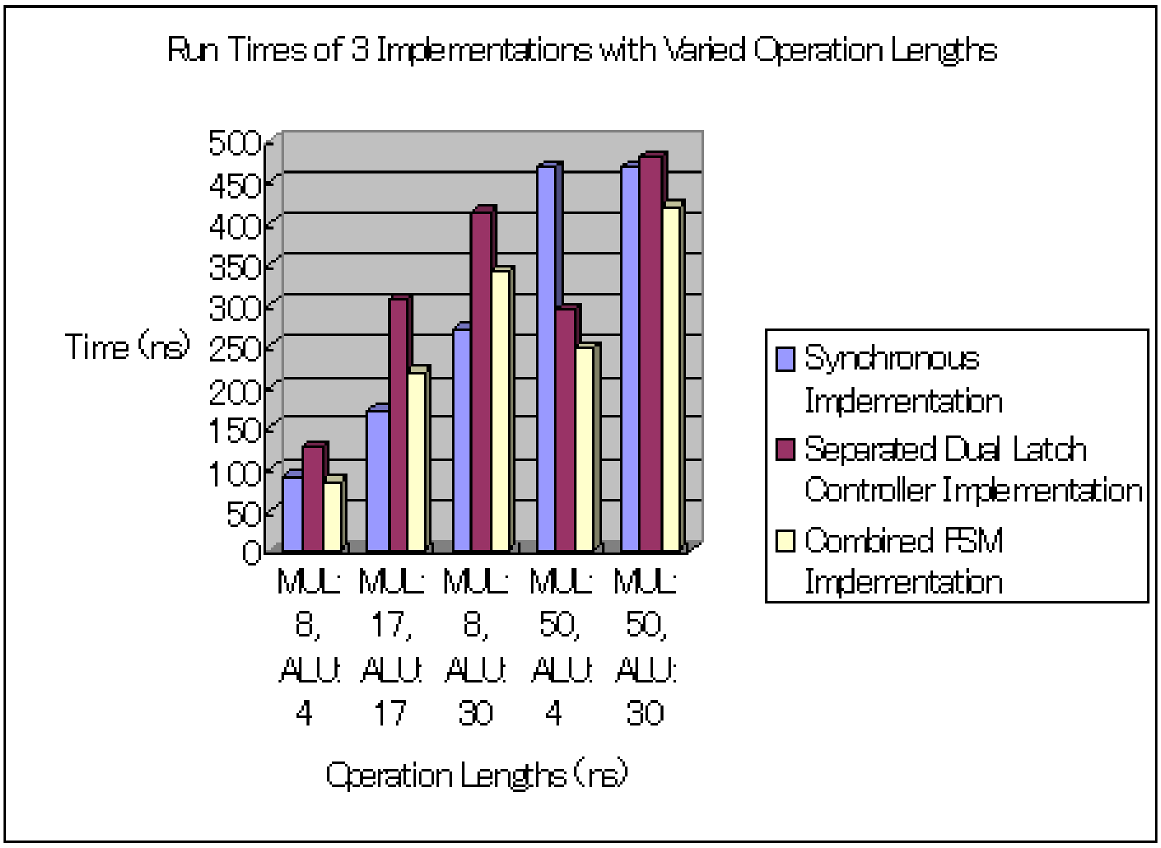


Figure 6.1: Comparison of Speeds in the 3 Implementations

The 8 ns MUL and 4 ns ALU is the default test; these are the values in the generated synchronous circuit. It shows what happens when the operations are short, and the latching takes a relatively long time. The 8 ns MUL and 30 ns ALU test and the 50 ns MUL and 4 ns ALU test are to show the result when one of the two operations takes much longer than the other, and how the frequency of that operation and the scheduling can affect the timing, in contrast with the length, since the ALU operation is much more common, but the MLU operation longer since it is set to 50 ns. The 50 ns MUL and 30 ns ALU test is to show what happens when the operations take a long time, and the latching a short time. Finally, the last test with 17ns in both operations is intended to showcase the worst case for the asynchronous implementations: when both of the operations take the same amount of time. This means that there is no slack in either of the operations, and no way to benefit from asynchronism. The timings of the five scenarios are shown once again in a histogram in Figure 6.1.

We can see that the first asynchronous implementation is slower than the synchronous implementation. This is due to a combination of two factors: the compromise of the concurrency in the system, resulting from the corrections in the schedule due to triangle and crossover hazards, and stalls in the system due to greater handshakes. However, it can be faster in certain cases, namely where one of the functional units is much slower than the others, and rarely used. In this case, enough wasted time will be cut off when the ALU is being used that it makes up for stalls and schedule inefficiencies.

The second asynchronous implementation is in general slightly faster than the synchronous implementation, but in worst case scenarios it will be slower. In these scenarios, the amount of time cut off from the end of the fast finishing states will not make up for the time lost in serializing and rerouting the schedule. It indeed still suffers from some rescheduling, to avoid the crossover hazards in the circuit, but there is not the same level of stalling as in the first implementation. If there is no corrective rescheduling, and the functions performed by the CUs do not take the same length of time, this implementation will most definitely be faster than the synchronous alternative.

6.2 Factors Affecting System Performance

It is therefore important to note that there are many factors which have a considerable impact on the effectiveness of the two methods on the speedup of the circuit, and which implementation will be the fastest. These factors include:

- The schedule and number of hazards therein
- The difference in lengths between CU operation times
- The number of inputs and outputs per CU
- The number of outputs of the circuit
- The relative frequencies of these operations

The synchronous implementation will tend to be the fastest in situations where the operations have similar execution lengths, because there is less slack in each cycle, and the slack is the area in which the asynchronous implementations can improve the circuit. It will also tend to be faster if there are relatively many of the operations that take a long time, because the total amount of slack in the circuit will be less. It will also tend to be faster in cases where there are many inputs to and outputs from the operations and a small amount of CUs in the circuit. If there are a small amount of CUs, there are a small amount of scheduling options, which creates more hazards in the asynchronous implementation. If each operation has more inputs and outputs, there will tend to be more hazards and dependencies, which also favours the synchronous implementation. If there is a larger number of outputs from the circuit, the amount of CUs which are used to hold values goes up, which means the amount of scheduling options is smaller, again increasing hazards, and favouring this implementation.

In contrast, the combined FSM implementation will be the fastest in situations where the CU operations take very different amounts of time, and if the shorter duration operations are more plentiful than longer ones, because of large amounts of slack. It will also excel when there are few inputs into and outputs from each operation and many CUs, because there will be more scheduling options, and greater chances to avoid crossover, triangle, and self handshake hazards. Lastly, it will tend to be faster if there are fewer outputs from the circuit, because final values will take up less of the CUs, and there will be more scheduling options.

The combined FSM implementation will benefit over its two FSM counterpart more in the cases where waiting times are long. The CUs are able to finish without waiting for each other and can thus move on to other work. This may also help to speed up circuits which are scheduled with this in mind, because units which are waiting in the synchronous implementation can move on faster and be assigned more operations.

6.3 Impacts on Area Overhead, Power Usage, and Delay Variability

Since one central state machine is being split up into many, these state machines may be quite large, and hardware is being added for the latch control network, the area overhead for these two methods will likely be quite considerable, especially if the width of the datapath is small. The area for the combined FSM implementation especially could be very large, since the state variable is very large and many outputs depend on logic calculated from the state variable.

The clock is entirely removed from the system, and the CUs only communicate when necessary, therefore there are likely considerable power savings compared to the original circuit. There are also no skew problems, as all communication is done with handshaking, senders can be sure their information reached its destination safely.

If the delay of the elements in the system can somehow respond to different operating conditions which affect the delay of the logic, the system can respond to variations in delay because the delay elements are the part of the circuit which determines its speed. In this situation, the circuit could gain further speedup and robustness without sacrificing either. This capability cannot be realised without a sufficiently sophisticated delay element technology, however.

6.4 Recommendations for Future Work

This section discusses several ideas which were not explored in the duration of this project, but could be developed further in the future.

Something that would be of great benefit for creating a synthesized circuit is a compiler which can create asynchronous control and datapath logic based on this method. If a designer was to attempt to compile a VHDL file outlining the logic of the circuit using a standard VHDL compiler, which is designed for register transfer logic, they will not get the desired result. Currently, the circuit would be extremely difficult to synthesize. But a synthesized circuit can give estimates for saved power and area, which would be extremely useful for comparison with synchronous implementations.

The true secret to an efficient schedule in the asynchronous implementations is a scheduling algorithm made specifically for asynchronous implementations, which eliminates the concepts of state and timestep, takes into account different lengths of operations, and avoids the three asynchronous hazards, while maximizing concurrency. Breakthroughs have already been achieved in this field [7], using a very different scheme, and likely a decentralized FSM scheme such as the one described in this report can improve timing or skew of the control signals.

The best results as far as speed is concerned would be obtained if the system could be scheduled using such an algorithm. A tool which generates this sort of asynchronous system would be much better than first creating the synchronous system and later desynchronizing it. Such a tool could be created by modiying a pre-existing tool which generates a synchronous scheduled circuit.

Another option that was not fully explored during this project due to lack of time, is to have two separate FSMs controlling the input and output handshakes of each CU and communicating to each other using some form of handshaking, so that neither one gets too far ahead of the other. This may prove to be more efficient because the first latch does not necessarily have to wait until the ack out is low before it can perform a new handshake, but in the schemas presented in this report that is a necessary condition for transition to the next state, so it must wait.

In the case that it does not prove advantageous to make everything asynchronous, the decentralization strategy could also work in a GALS (Globally Asynchronous, Locally Synchronous) scheme, where the local FSM and CUs are synchronous but communicate to each other in an asynchronous fashion. This is unlikely to provide system speedup, but may be able to alleviate problems due to clock and control skew.

6.5 Closing Remarks

Two thorough methods for efficiently desynchronizing a scheduled circuit have been introduced in this thesis. They use a network of latch controllers to communicate with each other, and create localized state machines for each computational unit to eliminate clock and control signal skew. There is a tradeoff of area and speed in choosing one method over the other. The separated latch pair controller method conserves more area but the combined FSM method runs faster.

The scheduling conditions which endanger these methods, namely crossover, triangle, and self-handshaking hazards, were identified, and rescheduling solutions to correct them through rerouting or using similar components were suggested.

Sample implementations of both these methods using a converted scheduled filter have been realised in VHDL code and simulated with several tests. The results of these tests have been compared with the originals and found to be faster in certain cases.

Scheduling was found to have a drastic effect of which implementation is the fastest. Other influences on the speed include frequency of operations, lengths of different operations, size of the circuit, and numbers of inputs and outputs.

These desynchronization methods provide a potential way to decrease power, increase speed, deal with delay variability, and a way to solve clock skew and alleviate the pressures of timing closure based design. At the same time, they provide as direct a conversion from the synchronous version as possible. In this way, the ideas and findings presented in this paper provide an attractive combination of advantages from both synchronous and asynchronous design.

More State Cluster Diagrams A

This appendix shows additional state cluster diagrams not displayed in the previous chapters for various combinations of inputs and outputs for both the standard and modified schemes.

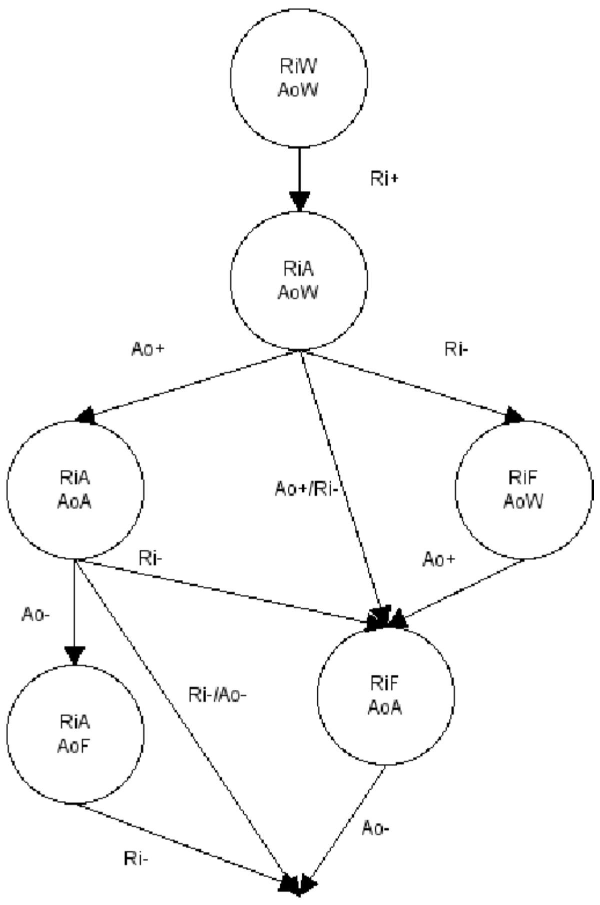


Figure A.1: A Standard 1-Input 1-Output State Cluster

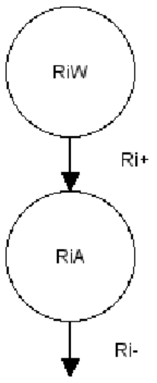


Figure A.2: A Standard 1-Input 0-Output State Cluster

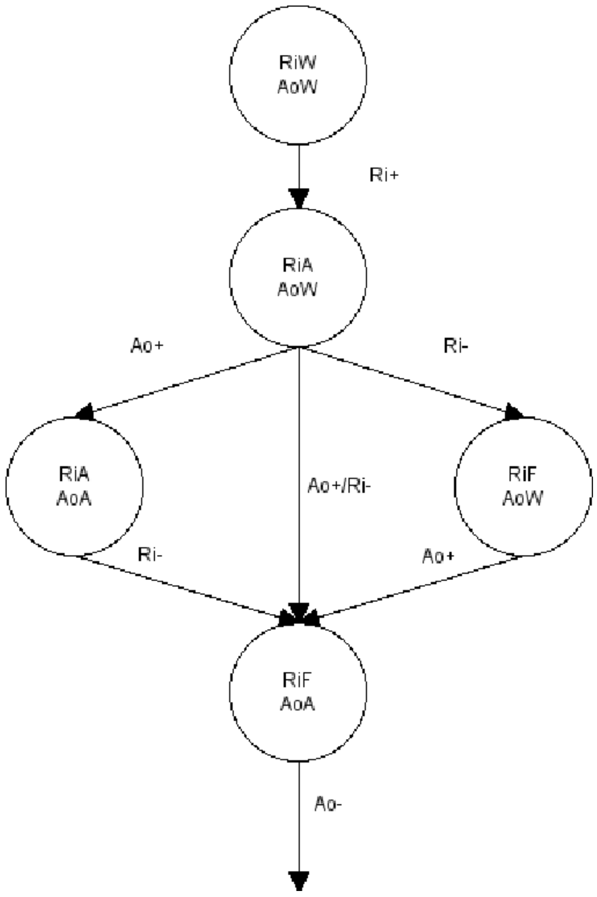


Figure A.3: A Modified 1-Input 1-Output State Cluster

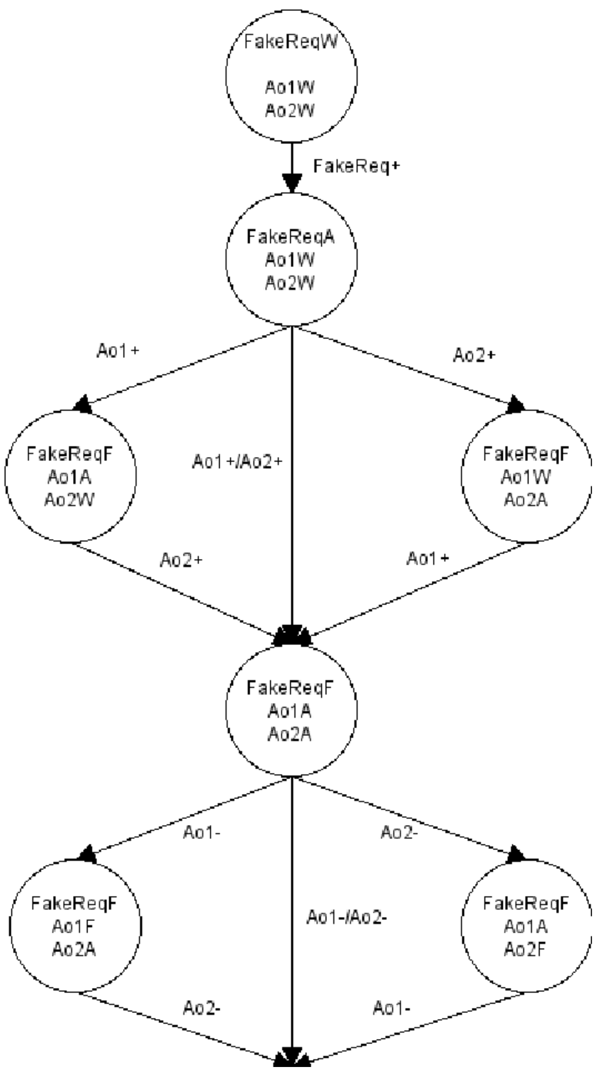


Figure A.4: A Modified 0-Input 2-Output State Cluster

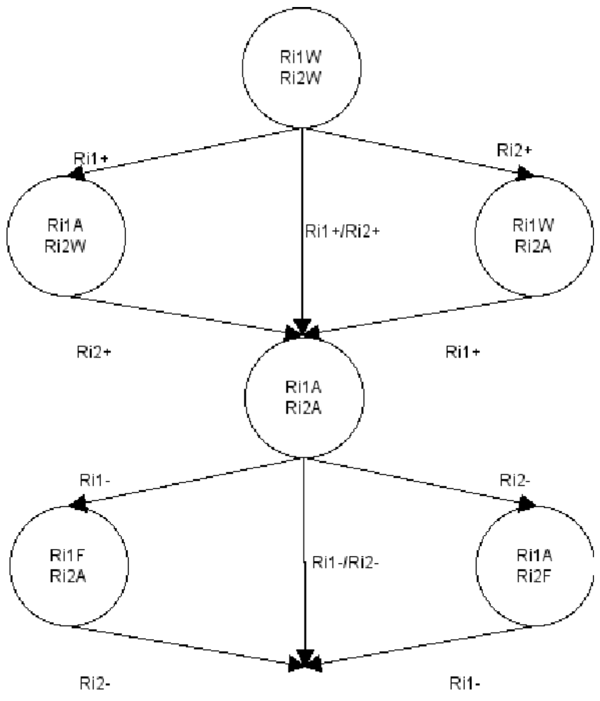


Figure A.5: A Modified 2-Input 0-Output State Cluster

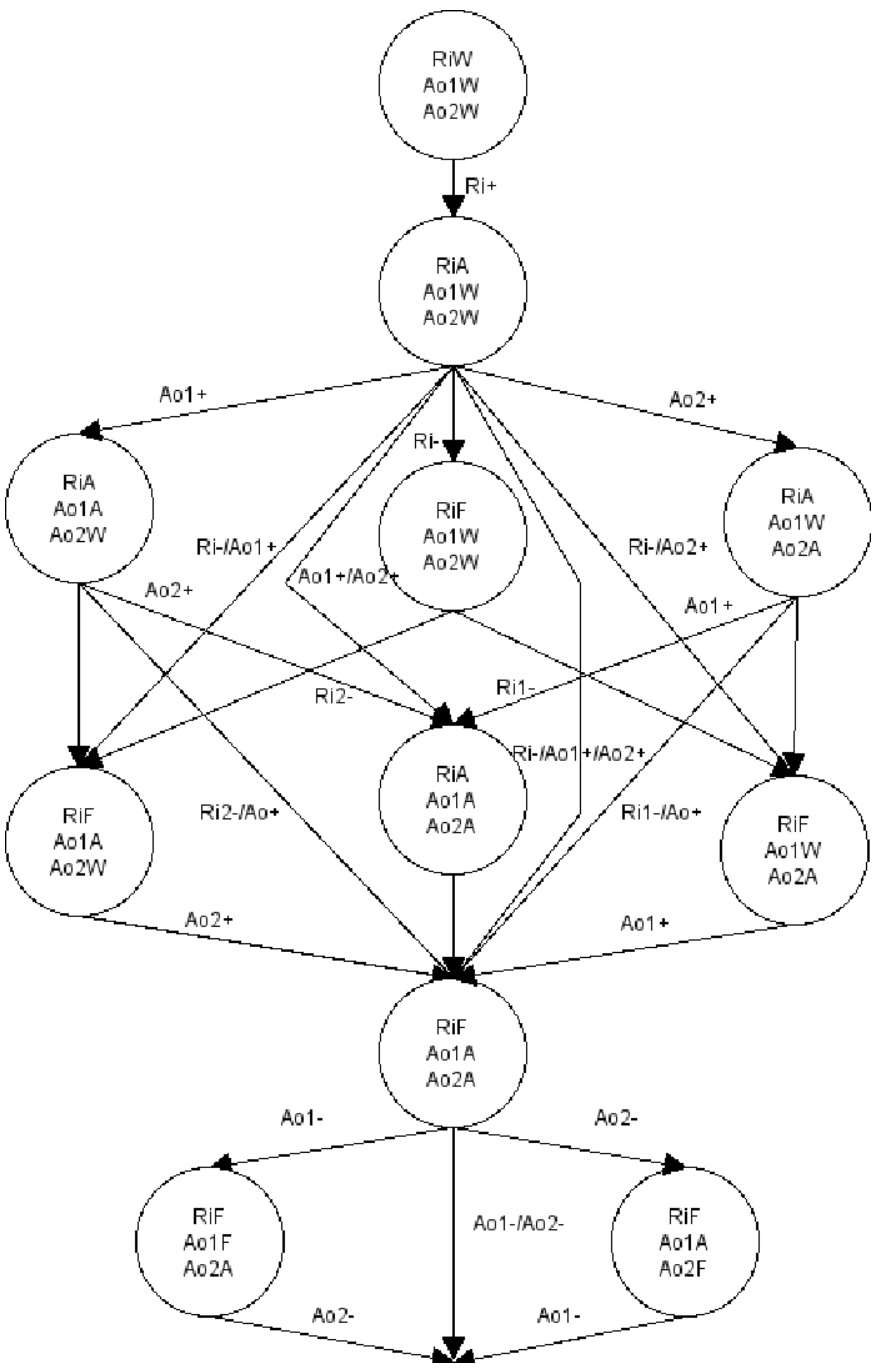


Figure A.6: A Modified 1-Input 2-Output State Cluster

Code Samples B

This appendix features some of the code used in the two implementations, to give the reader an idea of how the correct logic was produced.

B.1 Separated Latch Controller Implementation Code

This section provides the instruction state machine for the second MUL unit, and the code for the latch pair controller.

B.1.1 The Latch Pair Controller State Machine

```
entity cortlatchctrl is
generic ( REG_delay_g : Time :=  2 ns );
port (rst :  in std_logic;
--inputs
    Ri : in std_logic;
    Ao : in std_logic;
--outputs
    inputdone : out std_logic;
    outputdone : out std_logic;
    E : out std_logic;
    O : out std_logic;
    Ai : out std_logic;
    Ro : out std_logic
);
end cortlatchctrl;

architecture cortlatchctrl_behavioral of cortlatchctrl is
    type  STATE_TYPE is ( STATE_START, STATE_REQIN, STATE_INDONE, STATE_OUTDONE, STATE_FINISHING);
    attribute ENUM_ENCODING of STATE_TYPE:type is "000 001 011 101 111";
    signal  CURRENT_STATE, NEXT_STATE : STATE_TYPE;
begin
    --next state and output logic
    STATES:
    process ( CURRENT_STATE, Ri, Ao, rst)
    begin
        if rst = '1' then
            NEXT_STATE  <= STATE_START;
            inputDone <= '0';
            outputDone <= '0';
            E <= '1';
            O <= '0';
            Ai <= '0';
            Ro <= '0';
        else
            case CURRENT_STATE is
                when STATE_START =>
                    inputDone <= '0';
                    outputDone <= '0';
                    E <= '1';
                    O <= '0';
                    Ai <= '0';
                    Ro <= '0';
                    if Ri = '1' then
                        NEXT_STATE  <= STATE_REQIN;
                    else
                        NEXT_STATE  <= STATE_START;
                    end if;
                when STATE_REQIN =>
                    inputDone <= '0';
                    outputDone <= '0';
                    E <= '0';
                    O <= '1';
                    Ai <= '1';
                    Ro <= '1';
                    if Ri = '0'  and Ao = '1' then
                        NEXT_STATE  <= STATE_FINISHING;
                    elsif Ri = '0' then
                        NEXT_STATE  <= STATE_INDONE;
                    elsif Ao = '1' then
                        NEXT_STATE  <= STATE_OUTDONE;
                    else
                        NEXT_STATE  <= STATE_REQIN;
                    end if;
                when STATE_INDONE =>
                    inputDone <= '1';
                    outputDone <= '0';
                    E <= '0';
                    O <= '1';
                    Ai <= '0';
                    Ro <= '1';
                    if Ao = '1' then
                        NEXT_STATE  <= STATE_FINISHING;
                    else
                        NEXT_STATE  <= STATE_INDONE;
                    end if;
                when STATE_OUTDONE =>
                    inputDone <= '0';
                    outputDone <= '1';
                    E <= '0';
                    O <= '0';
                    Ai <= '1';
                    Ro <= '0';
                    if Ri = '0' then
                        NEXT_STATE  <= STATE_FINISHING;
                    else
                        NEXT_STATE  <= STATE_OUTDONE;
                    end if;
                when STATE_FINISHING =>
                    inputDone <= '1';
                    outputDone <= '1';
                    E <= '0';
                    O <= '0';
                    Ai <= '0';
                    Ro <= '0';
                    NEXT_STATE  <= STATE_START;
                when others =>
                    inputDone <= '0';
                    outputDone <= '0';
                    E <= '1';
                    O <= '0';
                    Ai <= '0';
                    Ro <= '0';
                    NEXT_STATE  <= STATE_START;
            end case;
        end if;
    end process STATES;

    CLK_INFO:
    process (rst, Ri, Ao, NEXT_STATE)
    begin
        if rst = '1' then
            CURRENT_STATE <= STATE_START;
        else
            CURRENT_STATE <= NEXT_STATE;
        end if;
    end process CLK_INFO;

end cortlatchctrl_behavioral;
```

B.1.2 A Sample Instruction State Machine

```
entity mul2FSM is
generic ( NX_g    : positive := 16;
    M_g      : positive := 15;
    MUL_delay_g :      Time :=  5 ns;
    REG_delay_g : Time :=  2 ns );
port (rst :  in std_logic;
    start : in std_logic;
--constants
    A2B :  in std_logic_vector(NX_g-1 downto 0);
--inputs
    ALU2toMUL2req : in std_logic;
    ALU2toMUL2ack : out std_logic;
    ALU2out :  in std_logic_vector(NX_g-1 downto 0);
--outputs
    MUL2toALU3req : out std_logic;
    MUL2toALU3ack : in std_logic;
    MUL2toALU2req : out std_logic;
    MUL2toALU2ack : in std_logic;
    dataout : out std_logic_vector(NX_g-1 downto 0);
    last_state : out std_logic;
    done       : out std_logic
);
end mul2FSM;
```

```vhdl
architecture mul2FSM_behavioral of mul2FSM is
    type  STATE_TYPE is ( STATE_1, STATE_2, STATE_3, STATE_4, STATE_5, STATE_FINISHING, STATE_DONE );
    attribute ENUM_ENCODING of STATE_TYPE:type is "000 001 011 010 110 100 101";

    signal  CURRENT_STATE, NEXT_STATE : STATE_TYPE;
    signal  Ai0, Ri0, Ai1, Ri1, Ai2, Ri2, pRi0, lRi0, fakeReqOut, Ro0: std_logic;
    signal  eCtrl, oCtrl : std_logic;
    signal  mulIn1, mulIn2, out_s, mulOut : std_logic_vector(NX_g-1 downto 0);
    signal  inputDone, outputDone, requestSeen : std_logic;
    signal id, od : std_logic;

    component CRTDLA_ELCTRL is
        port (RESET, Ao, Rid : in std_ulogic;
            Ai, Ro, E: out std_ulogic );
    end component;

    component CRTDLA_OLCTRL is
        port (RESET, Ao, Rid : in std_ulogic;
            Ai, Ro, O: out std_ulogic );
    end component;

    component cortlatchctrl is
    generic ( REG_delay_g : Time :=  2 ns );
    port (rst :  in std_logic;
    --inputs
        Ri : in std_logic;
        Ao : in std_logic;
     --outputs
        inputdone : out std_logic;
        outputdone : out std_logic;
        E : out std_logic;
        O : out std_logic;
        Ai : out std_logic;
        Ro : out std_logic
     );
    end component;

    component MUL_L1Cas is
    generic (
        NX_g         : positive := 16;      -- databus width
        M_g          : positive := 15;      -- width of fraction part
        MUL_delay_g :      Time :=  5 ns;  -- additional delay of mul
        REG_delay_g :      Time :=  2 ns   -- simulation delay register
    );
    port (
        reset   :  in std_logic;            -- asynchronous, active high
        clk     :  in std_logic;
        clk2     :  in std_logic;
        clk_en  :  in std_logic;
        op1     :  in std_logic_vector(NX_g-1 downto 0);
        op2     :  in std_logic_vector(NX_g-1 downto 0);
        out_s   : out std_logic_vector(NX_g-1 downto 0);
        out_r   : out std_logic_vector(NX_g-1 downto 0)
    );
    end component;

begin
        --MUL unit
    MUL_1: MUL_L1Cas
    generic map ( NX_g, M_g, MUL_delay_g, REG_delay_g )
    port map ( reset => rst,
        clk => eCtrl, clk2 => oCtrl, clk_en => '1',
        op1 => mulIn1, op2 => mulIn2,
        out_s => out_s, out_r => mulOut);

    dataout <= mulOut;

    --latch pair controller
    ctrl: cortlatchctrl
    generic map ( REG_delay_g )
    port map ( rst => rst,
    --inputs
        Ri =>Ri0,
        Ao => Ai2,
    --outputs
        inputdone => id,
        outputdone => od,
        E => eCtrl,
        O => oCtrl,
        Ai => Ai0,
        Ro => Ri2);

    --input delay
    Ri0CTRL:
    process( CURRENT_STATE, inputDone, pRi0)
    begin
        if inputDone = '0' then
            if requestSeen = '0' then
                Ri0 <= pRi0 after (MUL_delay_g + REG_delay_g);
            else
                Ri0 <= pRi0;
            end if;
            else
            Ri0 <= '0';
        end if;
    end process Ri0CTRL;

    --multiplexing input data lines and requests
    INPUTS:
    process ( CURRENT_STATE, rst, inputDone, Ai0, ALU2toMUL2req, ALU2out, A2B)
    begin
        if CURRENT_STATE = STATE_1 and inputDone = '0' then
                pRi0         <= ALU2toMUL2req;
        elsif (CURRENT_STATE = STATE_1 and inputDone = '1') or (CURRENT_STATE = STATE_FINISHING and inputDone =
             '0') then
            pRi0         <= '0';
            mulIn1      <= A2B;
            mulIn2      <= ALU2out;
        elsif (CURRENT_STATE = STATE_FINISHING and inputDone = '1') then
            pRi0         <= '0';
            mulIn1      <= "0000000000";
            mulIn2      <= "0000000000";
        elsif (CURRENT_STATE = STATE_DONE) then
            pRi0         <= '0';
            mulIn1      <= "0000000000";
            mulIn2      <= "0000000000";
        else
            pRi0        <= '0';
            mulIn1      <= "0000000000";
            mulIn2      <= "0000000000";
        end if;
    end process INPUTS;

    --multiplexing input acks
    ACKS:
    process ( CURRENT_STATE, rst, inputDone, Ai0)
    begin
        if inputDone = '1' then
            ALU2toMUL2ack   <= '0';
        else
            case CURRENT_STATE is
                when STATE_1 =>
                    ALU2toMUL2ack   <= Ai0;
                when STATE_FINISHING =>
                    ALU2toMUL2ack   <= '0';
                when STATE_DONE =>
                    ALU2toMUL2ack   <= '0';
                when others =>
                    ALU2toMUL2ack   <= '0';
            end case;
        end if;
    end process ACKS;

    --multiplexing outputs
    OUTPUTS:
    process ( CURRENT_STATE, rst, outputDone, Ri2, MUL2toALU3ack, MUL2toALU2ack, fakeReqOut)
    begin
        if outputDone = '1' then
            Ai2             <= '0';
            MUL2toALU3req     <= '0';
            MUL2toALU2req     <= '0';
        else
            case CURRENT_STATE is
                when STATE_1 =>
                    Ai2         <= MUL2toALU3ack;
                    MUL2toALU3req     <= Ri2;
                    MUL2toALU2req     <= '0';
                when STATE_FINISHING =>
                    Ai2         <= '0';
                    MUL2toALU3req     <= '0';
                    MUL2toALU2req     <= fakeReqOut;
                when STATE_DONE =>
                    Ai2         <= '0';
                    MUL2toALU3req     <= '0';
                    MUL2toALU2req     <= '0';
                when others =>
                    Ai2     <= '0';
                    MUL2toALU3req     <= '0';
                    MUL2toALU2req     <= '0';
            end case;
        end if;
    end process OUTPUTS;

    --set to high when request has gone high in this state
    REQ_SEEN:
    process (Ri0, CURRENT_STATE, rst)
    begin
        if CURRENT_STATE'event or rst = '1' then
            requestSeen <= '0';
        elsif rising_edge(Ri0) then
            requestSeen <= '1';
        end if;
    end process REQ_SEEN;

    --set to high when input handshake completed
    INPUT_DONE:
    process (Ri0, CURRENT_STATE, rst, fakeReqOut)
        begin
        if CURRENT_STATE'event or rst = '1' then
            inputDone <= '0';
        elsif falling_edge(Ri0)  or falling_edge(fakeReqOut) then
            inputDone <= '1';
        end if;
        end process INPUT_DONE;

    --set to high when output handshake completed
    OUTPUT_DONE:
    process (Ai2, CURRENT_STATE, rst, fakeReqOut)
    begin
        if CURRENT_STATE'event or rst = '1' then
            outputDone <= '0';
        elsif falling_edge(Ai2) or falling_edge(fakeReqOut) then
            outputDone <= '1';
        end if;
    end process OUTPUT_DONE;

    --next state, last state, and done signal logic
    STATES:
    process ( CURRENT_STATE, A2B, rst, start)
    begin
        if rst = '1' then
            NEXT_STATE  <= STATE_DONE;
            last_state <= '0';
            done      <= '1';
        else
            case CURRENT_STATE is
                when STATE_1 =>
                    last_state <= '0';
                    done      <= '0';
                    NEXT_STATE  <= STATE_FINISHING;
                when STATE_FINISHING =>
                    last_state <= '1';
                    done       <= '0';
                    NEXT_STATE  <= STATE_DONE;
                when STATE_DONE =>
                    last_state <= '0';
                    done       <= '1';
                    NEXT_STATE  <= STATE_1;
                when others =>
                    last_state <= '0';
                    done      <= '0';
                    NEXT_STATE  <= STATE_DONE;
            end case;
        end if;
    end process STATES;

    --fake out request latch
    FAKE_OUT_LATCH:
    process ( CURRENT_STATE, MUL2toALU2ack)
    begin
        if CURRENT_STATE = STATE_FINISHING and MUL2toALU2ack = '0' then
            fakeReqOut <= '1';
        else
            fakeReqOut <= '0';
        end if;
    end process FAKE_OUT_LATCH;

       --state transitions
    CLK_INFO:
    process (rst, start, outputDone, inputDone)
    begin
        if rst = '1' then
            CURRENT_STATE <= STATE_DONE;
        elsif (start'event and start = '1') or (outputDone = '1' and inputDone = '1') then
            CURRENT_STATE <= NEXT_STATE;
        end if;
    end process CLK_INFO;

end mul2FSM_behavioral;
```

B.2 Combined FSM Implementation Code

This section provides a code sample of the combined state machine for the second MUL unit.

B.2.1 The Latch Pair Controller State Machine

```
entity mul2FSM is
generic ( NX_g     : positive := 16;
    M_g      : positive := 15;
    MUL_delay_g :      Time :=  5 ns;  -- additional delay of mul
    REG_delay_g : Time :=  2 ns );
port (rst :  in std_logic;
    start : in std_logic;
--constants
    A2B :  in std_logic_vector(NX_g-1 downto 0);
--inputs
    ALU2toMUL2req : in std_logic;
    ALU2toMUL2ack : out std_logic;
    ALU2out :  in std_logic_vector(NX_g-1 downto 0);
--outputs
    MUL2toALU3req : out std_logic;
    MUL2toALU3ack : in std_logic;
    MUL2toALU2req : out std_logic;
    MUL2toALU2ack : in std_logic;
    dataout : out std_logic_vector(NX_g-1 downto 0);
    last_state : out std_logic;
    done       : out std_logic
);
end mul2FSM;

architecture mul2FSM_behavioral of mul2FSM is
    type  STATE_TYPE is ( S1_WAIT, S1_RI_HIGH, S1_RI_LOW, S1_AO1_HIGH, S1_AO2_HIGH, S1_RIL_AO1H, S1_RIL_AO2H,
         S1_AO1H_AO2H, S1_FIN, S1_AO1_LOW, S1_AO2_LOW, S_DONE );

    signal  CURRENT_STATE, NEXT_STATE : STATE_TYPE;
    signal  pRi0, lRi0, fakeReqOut, Ro0, requestSeen: std_logic;
    signal  eCtrl, oCtrl : std_logic;
    signal  mulIn1, mulIn2, out_s, mulOut : std_logic_vector(NX_g-1 downto 0);

    component MUL_L1Cas is
    generic (
        NX_g         : positive := 16;      -- databus width
        M_g          : positive := 15;      -- width of fraction part
        MUL_delay_g :      Time :=  5 ns;  -- additional delay of mul
        REG_delay_g :      Time :=  2 ns    -- simulation delay register
    );
    port (
        reset   :  in std_logic;            -- asynchronous, active high
        clk     :  in std_logic;
        clk2     :  in std_logic;
        clk_en :  in std_logic;
        op1     :  in std_logic_vector(NX_g-1 downto 0);
        op2     :  in std_logic_vector(NX_g-1 downto 0);
        out_s   : out std_logic_vector(NX_g-1 downto 0);
        out_r   : out std_logic_vector(NX_g-1 downto 0)
    );
    end component;

begin
    --MUL unit
    MUL_1: MUL_L1Cas
    generic map ( NX_g, M_g, MUL_delay_g, REG_delay_g )
    port map ( reset => rst,
        clk => eCtrl, clk2 => oCtrl, clk_en => '1',
        op1 => mulIn1, op2 => mulIn2,
        out_s => out_s, out_r => mulOut);

    dataout <= mulOut;

    --input delay
    Ri0CTRL:
    process( CURRENT_STATE, pRi0)
    begin
        if requestSeen = '0' then
            lRi0 <= pRi0 after  (MUL_delay_g + REG_delay_g);
        else
            lRi0 <= pRi0;
        end if;
    end process Ri0CTRL;

    --set to high when request has gone high in this state
    REQ_SEEN:
    process( CURRENT_STATE, rst, lRi0)
    begin
        if rst = '1' or CURRENT_STATE = S_DONE or falling_edge(lRi0) then
            requestSeen <= '0';
        elsif rising_edge(lRi0) then
            requestSeen <= '1';
        end if;
    end process REQ_SEEN;

    --next state logic, output reqs, input acks
    STATES:
```

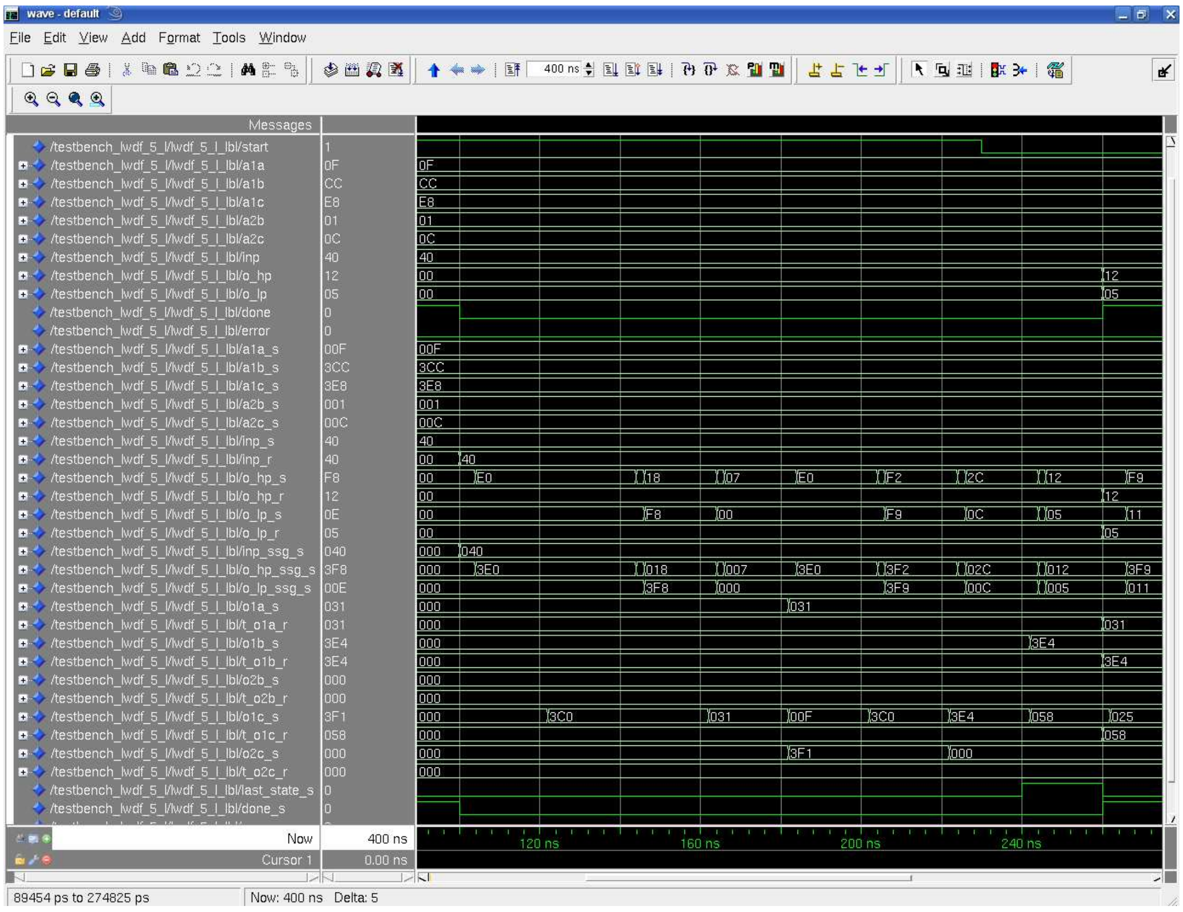
```
process ( CURRENT_STATE, rst, start, ALU2toMUL2req, ALU2out, MUL2toALU3ack, MUL2toALU2ack, A2B, lRi0)
begin
    if rst = ’1’ then
        NEXT_STATE  <= S_DONE;
        ALU2toMUL2ack <= ’0’;
        MUL2toALU3req <= ’0’;
        MUL2toALU2req <= ’0’;
        last_state <= ’0’;
        done     <= ’1’;
        mulIn1    <= "0000000000";
        mulIn2    <= "0000000000";
        pRi0 <= ’0’;
        eCtrl <= ’0’;
        oCtrl <= ’0’;
        else
        case CURRENT_STATE is
            when S1_WAIT =>
                ALU2toMUL2ack <= ’0’;
                MUL2toALU3req <= ’0’;
                MUL2toALU2req <= ’0’;
                last_state <= ’0’;
                done     <= ’0’;
                mulIn1    <= A2B;
                mulIn2    <= ALU2out;
                pRi0 <= ALU2toMUL2req;
                eCtrl <= ’1’;
                oCtrl <= ’0’;
                if lRi0 = ’1’ then
                    NEXT_STATE  <= S1_RI_HIGH;
                else
                    NEXT_STATE  <= S1_WAIT;
                end if;
            when S1_RI_HIGH =>
                ALU2toMUL2ack <= ’1’;
                MUL2toALU3req <= ’1’;
                MUL2toALU2req <= ’1’;
                last_state <= ’0’;
                done     <= ’0’;
                mulIn1    <= A2B;
                mulIn2    <= ALU2out;
                pRi0 <= ALU2toMUL2req;
                eCtrl <= ’0’;
                oCtrl <= ’1’;
                if lRi0 = ’0’ then
                    if MUL2toALU3ack = ’1’ then
                        if MUL2toALU2ack = ’1’ then
                            NEXT_STATE  <= S1_FIN;
                        else
                            NEXT_STATE  <= S1_RIL_AO1H;
                        end if;
                    else
                        if MUL2toALU2ack = ’1’ then
                            NEXT_STATE  <= S1_RIL_AO2H;
                        else
                            NEXT_STATE  <= S1_RI_LOw;
                        end if;

                    end if;
                else
                    if MUL2toALU3ack = ’1’ then
                        if MUL2toALU2ack = ’1’ then
                            NEXT_STATE  <= S1_AO1H_AO2H;
                        else
                            NEXT_STATE  <= S1_AO1_HIGH;
                        end if;
                    else
                        if MUL2toALU2ack = ’1’ then
                            NEXT_STATE  <= S1_AO2_HIGH;
                        else
                            NEXT_STATE  <= S1_RI_HIGH;
                        end if;
                    end if;
                end if;
            when S1_RI_LOW =>
                ALU2toMUL2ack <= ’0’;
                MUL2toALU3req <= ’1’;
                MUL2toALU2req <= ’1’;
                last_state <= ’0’;
                done     <= ’0’;
                mulIn1    <= "0000000000";
                mulIn2    <= "0000000000";
                pRi0 <= ’0’;
                eCtrl <= ’0’;
                oCtrl <= ’1’;
                if MUL2toALU3ack = ’1’ then
                    if MUL2toALU2ack = ’1’ then
                        NEXT_STATE  <= S1_FIN;
                    else
                        NEXT_STATE  <= S1_RIL_AO1H;
                    end if;
                else
                    if MUL2toALU2ack = '1' then
                        NEXT_STATE  <= S1_RIL_AO2H;
                    else
                        NEXT_STATE  <= S1_RI_LOW;
                    end if;
                end if;
            when S1_AO1_HIGH =>
                ALU2toMUL2ack <= '1';
                MUL2toALU3req <= '0';
                MUL2toALU2req <= '1';
                last_state <= '1';
                done      <= '0';
                mulIn1     <= A2B;
                mulIn2     <= ALU2out;
                pRi0 <= ALU2toMUL2req;
                eCtrl <= '0';
                oCtrl <= '0';
                if lRi0 = '0' then
                    if MUL2toALU2ack = '1' then
                        NEXT_STATE  <= S1_FIN;
                    else
                        NEXT_STATE  <= S1_RIL_AO1H;
                    end if;
                else
                    if MUL2toALU2ack = '1' then
                        NEXT_STATE  <= S1_AO1H_AO2H;
                    else
                        NEXT_STATE  <= S1_AO1_HIGH;
                    end if;
                end if;
            when S1_AO2_HIGH =>
                ALU2toMUL2ack <= '1';
                MUL2toALU3req <= '1';
                MUL2toALU2req <= '0';
                last_state <= '1';
                done      <= '0';
                mulIn1     <= A2B;
                mulIn2     <= ALU2out;
                pRi0 <= ALU2toMUL2req;
                eCtrl <= '0';
                oCtrl <= '0';
                if lRi0 = '0' then
                    if MUL2toALU3ack = '1' then
                        NEXT_STATE  <= S1_FIN;
                    else
                        NEXT_STATE  <= S1_RIL_AO2H;
                    end if;
                else
                    if MUL2toALU3ack = '1' then
                        NEXT_STATE  <= S1_AO1H_AO2H;
                    else
                        NEXT_STATE  <= S1_AO2_HIGH;
                    end if;
                end if;
            when S1_RIL_AO1H =>
                ALU2toMUL2ack <= '0';
                MUL2toALU3req <= '0';
                MUL2toALU2req <= '1';
                last_state <= '0';
                done      <= '0';
                mulIn1     <= "0000000000";
                mulIn2     <= "0000000000";
                pRi0 <= '0';
                eCtrl <= '0';
                oCtrl <= '1';
                if MUL2toALU2ack = '1' then
                    NEXT_STATE  <= S1_FIN;
                else
                    NEXT_STATE  <= S1_RIL_AO1H;
                end if;
            when S1_RIL_AO2H =>
                ALU2toMUL2ack <= '0';
                MUL2toALU3req <= '1';
                MUL2toALU2req <= '0';
                last_state <= '0';
                done      <= '0';
                mulIn1     <= "0000000000";
                mulIn2     <= "0000000000";
                pRi0 <= '0';
                eCtrl <= '0';
                oCtrl <= '1';
                if MUL2toALU3ack = '1' then
                    NEXT_STATE  <= S1_FIN;
                else
                    NEXT_STATE  <= S1_RIL_AO2H;
                end if;
            when S1_AO1H_AO2H =>
                ALU2toMUL2ack <= '1';
                MUL2toALU3req <= '0';
                MUL2toALU2req <= '0';
                last_state <= '1';
                done      <= '0';
                mulIn1     <= A2B;
                mulIn2     <= ALU2out;
                pRi0  <= ALU2toMUL2req;
                eCtrl  <= '0';
                oCtrl  <= '0';
                if lRi0 = '0' then
                    NEXT_STATE  <= S1_FIN;
                else
                    NEXT_STATE  <= S1_AO1H_AO2H;
                end if;
            when S1_FIN =>
                ALU2toMUL2ack <= '0';
                MUL2toALU3req <= '0';
                MUL2toALU2req <= '0';
                last_state <= '1';
                done      <= '0';
                mulIn1     <= "0000000000";
                mulIn2     <= "0000000000";
                pRi0  <= '0';
                eCtrl  <= '1';
                oCtrl  <= '0';
                if MUL2toALU3ack = '0' then
                    if MUL2toALU2ack = '0' then
                        NEXT_STATE <= S_DONE;
                    else
                        NEXT_STATE  <= S1_AO1_LOW;
                    end if;
                else
                    if MUL2toALU2ack = '0' then
                        NEXT_STATE  <= S1_AO2_LOW;
                    else
                        NEXT_STATE  <= S1_FIN;
                    end if;
                end if;
            when S1_AO1_LOW =>
                ALU2toMUL2ack <= '0';
                MUL2toALU3req <= '0';
                MUL2toALU2req <= '0';
                last_state <= '1';
                done      <= '0';
                mulIn1     <= "0000000000";
                mulIn2     <= "0000000000";
                pRi0  <= '0';
                eCtrl  <= '1';
                oCtrl  <= '0';
                if MUL2toALU2ack = '0' then
                    NEXT_STATE <= S_DONE;
                else
                    NEXT_STATE  <= S1_AO1_LOW;
                end if;
            when S1_AO2_LOW =>
                ALU2toMUL2ack <= '0';
                MUL2toALU3req <= '0';
                MUL2toALU2req <= '0';
                last_state <= '1';
                done      <= '0';
                mulIn1     <= "0000000000";
                mulIn2     <= "0000000000";
                pRi0  <= '0';
                eCtrl  <= '1';
                oCtrl  <= '0';
                if MUL2toALU3ack = '0' then
                    NEXT_STATE <= S_DONE;
                else
                    NEXT_STATE  <= S1_AO2_LOW;
                end if;
            when S_DONE =>
                ALU2toMUL2ack <= '0';
                MUL2toALU3req <= '0';
                MUL2toALU2req <= '0';
                last_state <= '0';
                done      <= '1';
                mulIn1     <= "0000000000";
                mulIn2     <= "0000000000";
                pRi0  <= '0';
                eCtrl  <= '1';
                oCtrl  <= '0';
                NEXT_STATE <= S_DONE;
            when others =>
                ALU2toMUL2ack <= '0';
                MUL2toALU3req <= '0';
                MUL2toALU2req <= '0';
                last_state <= '0';
                done      <= '0';
                mulIn1     <= "0000000000";
                mulIn2     <= "0000000000";
                pRi0  <= '0';
```

```
                    eCtrl <= '0';
                    oCtrl <= '0';
                    NEXT_STATE  <= S_DONE;
            end case;
        end if;
    end process STATES;

    --state transition
    CLK_INFO:
    process (rst, start, NEXT_STATE)
    begin
        if rst = '1' then
            CURRENT_STATE <= S_DONE;
        elsif (start'event and start = '1') then
            CURRENT_STATE <= S1_WAIT;
        else
            CURRENT_STATE <= NEXT_STATE;
        end if;
    end process CLK_INFO;

end mul2FSM_behavioral;
```

Waveforms for Working Implementations

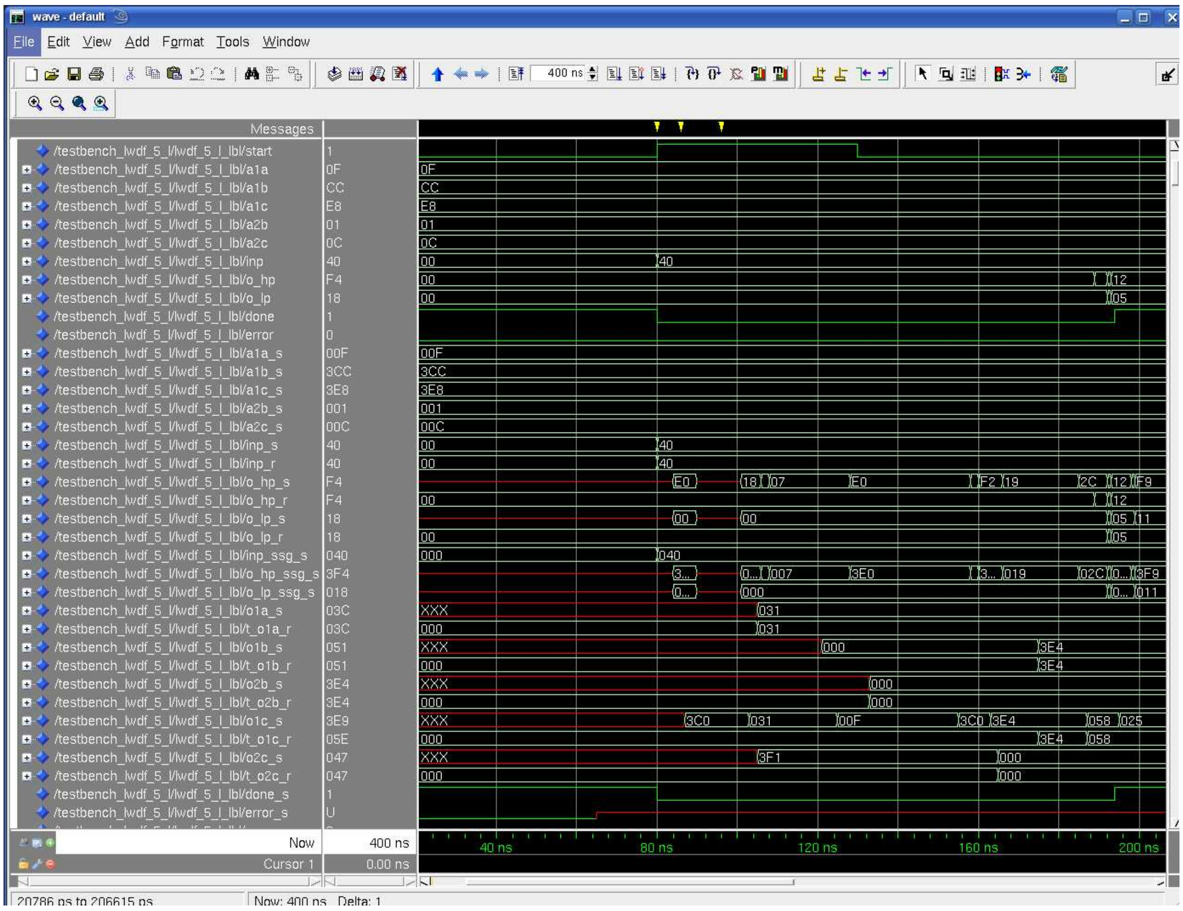
C

This appendix shows the waveforms for the duration of an iteration in all three implementations. The correctness of the values can all be verified to be equivalent to the synchronous implementation.



```
Messages
/testbench_lwdf_5_l/lwdf_5_l_lbl/start            1
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1a              0F
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1b              CC
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1c              E8
/testbench_lwdf_5_l/lwdf_5_l_lbl/a2b              01
/testbench_lwdf_5_l/lwdf_5_l_lbl/a2c              0C
/testbench_lwdf_5_l/lwdf_5_l_lbl/inp              40
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_hp             12
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_lp             05
/testbench_lwdf_5_l/lwdf_5_l_lbl/done             0
/testbench_lwdf_5_l/lwdf_5_l_lbl/error            0
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1a_s            00F
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1b_s            3CC
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1c_s            3E8
/testbench_lwdf_5_l/lwdf_5_l_lbl/a2b_s            001
/testbench_lwdf_5_l/lwdf_5_l_lbl/a2c_s            00C
/testbench_lwdf_5_l/lwdf_5_l_lbl/inp_s            40
/testbench_lwdf_5_l/lwdf_5_l_lbl/inp_r            40
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_hp_s           F8
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_hp_r           12
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_lp_s           0E
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_lp_r           05
/testbench_lwdf_5_l/lwdf_5_l_lbl/inp_ssg_s        040
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_hp_ssg_s       3F8
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_lp_ssg_s       00E
/testbench_lwdf_5_l/lwdf_5_l_lbl/o1a_s            031
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o1a_r          031
/testbench_lwdf_5_l/lwdf_5_l_lbl/o1b_s            3E4
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o1b_r          3E4
/testbench_lwdf_5_l/lwdf_5_l_lbl/o2b_s            000
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o2b_r          000
/testbench_lwdf_5_l/lwdf_5_l_lbl/o1c_s            3F1
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o1c_r          058
/testbench_lwdf_5_l/lwdf_5_l_lbl/o2c_s            000
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o2c_r          000
/testbench_lwdf_5_l/lwdf_5_l_lbl/last_state_s     0
/testbench_lwdf_5_l/lwdf_5_l_lbl/done_s           0

Now        400 ns
Cursor 1   0.00 ns

89454 ps to 274825 ps     Now: 400 ns  Delta: 5
```

Figure C.1: The Results of Running the Synchronous Implementation



```
Messages
/testbench_lwdf_5_l/lwdf_5_l_lbl/start          1
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1a            0F
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1b            CC
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1c            E8
/testbench_lwdf_5_l/lwdf_5_l_lbl/a2b            01
/testbench_lwdf_5_l/lwdf_5_l_lbl/a2c            0C
/testbench_lwdf_5_l/lwdf_5_l_lbl/inp            40
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_hp           F4
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_lp           18
/testbench_lwdf_5_l/lwdf_5_l_lbl/done           1
/testbench_lwdf_5_l/lwdf_5_l_lbl/error          0
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1a_s          00F
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1b_s          3CC
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1c_s          3E8
/testbench_lwdf_5_l/lwdf_5_l_lbl/a2b_s          001
/testbench_lwdf_5_l/lwdf_5_l_lbl/a2c_s          00C
/testbench_lwdf_5_l/lwdf_5_l_lbl/inp_s          40
/testbench_lwdf_5_l/lwdf_5_l_lbl/inp_r          40
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_hp_s         F4
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_hp_r         F4
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_lp_s         18
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_lp_r         18
/testbench_lwdf_5_l/lwdf_5_l_lbl/inp_ssg_s      040
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_hp_ssg_s     3F4
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_lp_ssg_s     018
/testbench_lwdf_5_l/lwdf_5_l_lbl/o1a_s          03C
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o1a_r        03C
/testbench_lwdf_5_l/lwdf_5_l_lbl/o1b_s          051
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o1b_r        051
/testbench_lwdf_5_l/lwdf_5_l_lbl/o2b_s          3E4
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o2b_r        3E4
/testbench_lwdf_5_l/lwdf_5_l_lbl/o1c_s          3E9
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o1c_r        05E
/testbench_lwdf_5_l/lwdf_5_l_lbl/o2c_s          047
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o2c_r        047
/testbench_lwdf_5_l/lwdf_5_l_lbl/done_s         1
/testbench_lwdf_5_l/lwdf_5_l_lbl/error_s        U
Now                                             400 ns
Cursor 1                                        0.00 ns
20786 ps to 206615 ps        Now: 400 ns  Delta: 1
```

Figure C.2: The Results of Running the Separated Latch Pair Controller Implementation

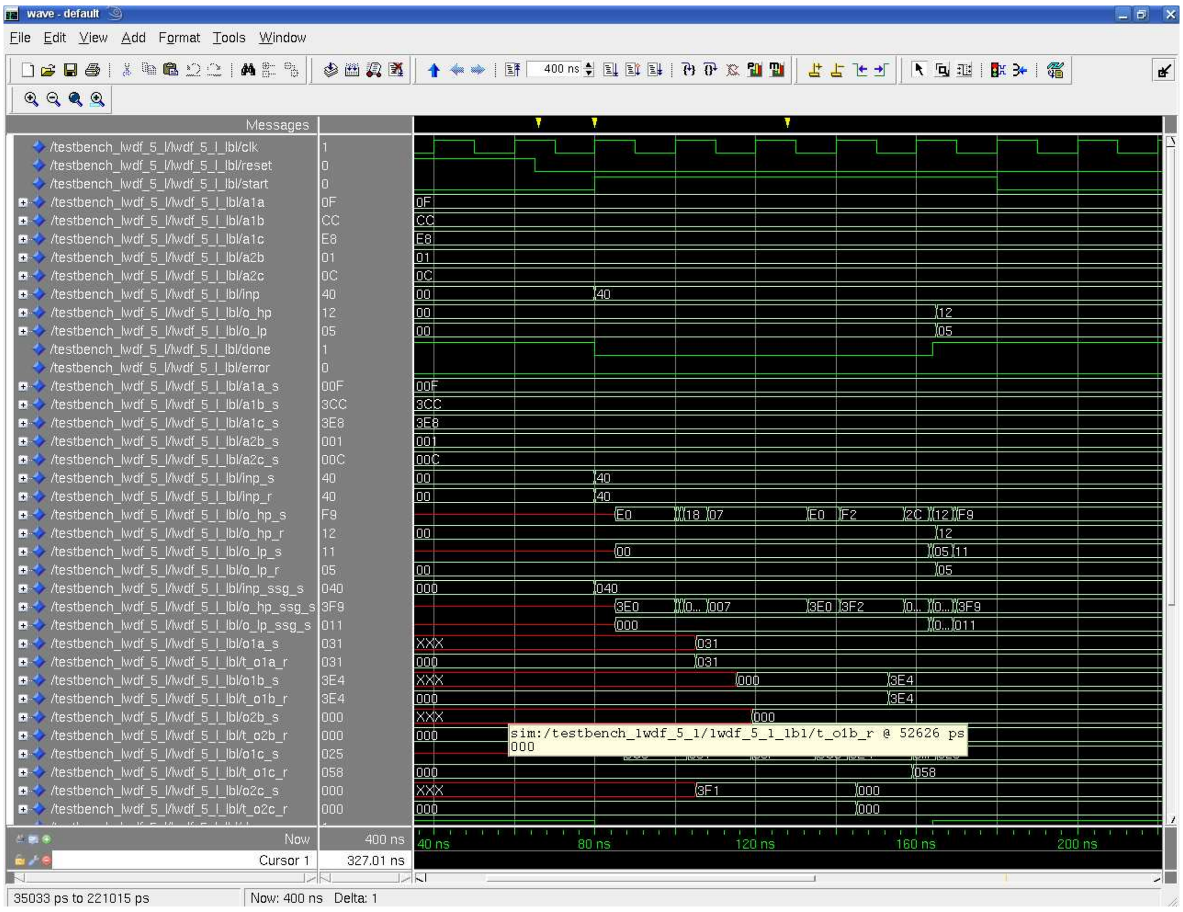

```
/testbench_lwdf_5_l/lwdf_5_l_lbl/clk            1
/testbench_lwdf_5_l/lwdf_5_l_lbl/reset          0
/testbench_lwdf_5_l/lwdf_5_l_lbl/start          0
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1a            0F
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1b            CC
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1c            E8
/testbench_lwdf_5_l/lwdf_5_l_lbl/a2b            01
/testbench_lwdf_5_l/lwdf_5_l_lbl/a2c            0C
/testbench_lwdf_5_l/lwdf_5_l_lbl/inp            40
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_hp           12
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_lp           05
/testbench_lwdf_5_l/lwdf_5_l_lbl/done           1
/testbench_lwdf_5_l/lwdf_5_l_lbl/error          0
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1a_s          00F
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1b_s          3CC
/testbench_lwdf_5_l/lwdf_5_l_lbl/a1c_s          3E8
/testbench_lwdf_5_l/lwdf_5_l_lbl/a2b_s          001
/testbench_lwdf_5_l/lwdf_5_l_lbl/a2c_s          00C
/testbench_lwdf_5_l/lwdf_5_l_lbl/inp_s          40
/testbench_lwdf_5_l/lwdf_5_l_lbl/inp_r          40
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_hp_s         F9
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_hp_r         12
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_lp_s         11
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_lp_r         05
/testbench_lwdf_5_l/lwdf_5_l_lbl/inp_ssg_s      040
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_hp_ssg_s     3F9
/testbench_lwdf_5_l/lwdf_5_l_lbl/o_lp_ssg_s     011
/testbench_lwdf_5_l/lwdf_5_l_lbl/o1a_s          031
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o1a_r        031
/testbench_lwdf_5_l/lwdf_5_l_lbl/o1b_s          3E4
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o1b_r        3E4
/testbench_lwdf_5_l/lwdf_5_l_lbl/o2b_s          000
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o2b_r        000
/testbench_lwdf_5_l/lwdf_5_l_lbl/o1c_s          025
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o1c_r        058
/testbench_lwdf_5_l/lwdf_5_l_lbl/o2c_s          000
/testbench_lwdf_5_l/lwdf_5_l_lbl/t_o2c_r        000

Now        400 ns
Cursor 1   327.01 ns

sim:/testbench_lwdf_5_1/lwdf_5_1_lbl/t_o1b_r @ 52626 ps
000

35033 ps to 221015 ps    Now: 400 ns  Delta: 1
```

Figure C.3: The Results of Running the Combined FSM Implementation

Bibliography

[1] Farhad Aghdasi. Synthesis of self-clocked asynchronous state machines. *TENCON*, 2:1018–1022, 1992.

[2] I. Blunno, J. Cortadella, A.Kondratyev, L. Lavagno, K. Lwin, and C. Sotirou. Handshake protocols for de-synchronization. *International Symposium on Advanced Research in Asynchronous Circuits and Systems 2004*, 0:149–158, 2004.

[3] Ivan Blunno and Luciano Lavagno. Designing an asynchronous microcontrollerusing pipefitter. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 12(7):696–700, 2004.

[4] Dmitry Bufistov, Jordi Cortadella, Mike Kishinevsky, and Sachin Sapatnekar. A general model for performance optimization of sequential systems. In *Proceedings of the 2007 IEEE/ACM International Conference on Computer-aided design*, pages 362–369. IEEE Press, 2007.

[5] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, and Christos P. Sotirou. De-synchronization: Synthesis of asynchronous circuits from synchronous specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):1904–1921, 2006.

[6] Doug Edwards and Andrew Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45, 2002.

[7] Naohiro Hamada, Yuki Shiga, Takao Konishi, Hiroshi Saito, Tomohiro Yoneda, Chris Meyers, and Takashi Nanya. A behavioral synthesis system for asynchronous circuits with bundled-data implementation. *IPSJ Transactions on System LSI Design Methodology*, 2:64–79, february 2009.

[8] Thomas Knight and Henry M. Wu. A method for skew-free distribution of digital signals using matched delay lines. *Massachusetts Institute of Technology Laboratory*, 1992.

[9] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, New York City, NY, USA, 1994.

[10] Victor P. Nelson, H. Troy Nagle, Bill D. Caroll, and J. David Irwin. *Digital Logic Circuit Analysis and Design.* Prentice Hall, Upper Saddle River, NJ, USA, 1995.

[11] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs.* Oxford University Press, New York City, NY, USA, 2000.

[12] Hiroshi Saito, Naohiro Hamada, Nattha Jindapetch, Tomohiro Yoneda, Chris Meyers, and Takashi Nanya. Scheduling methods for asynchronous circuits with bundled-data implementations based on the approximation of start times. *IEICE Transaction*, E90A(12):2790–2799, december 2007.

[13] D.A. Gilbert S.B. Furber, J.D. Garside. Amulet3:a high-performance self-timed arm microprocessor. *International Conference on Computer Design*, 0:247–252, 2006.

[14] Jens Sparsø and Steve Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective.* Kluwer Academic Publishers, Boston, MA, USA, 2001.

[15] Richard F. Tinder. *Engineering Digital Design.* Academic Press, San Diego, CA, USA, second edition, 2000.

[16] Michael Yoeli and Abraham Ginzburg. Petri-net based verification of asynchronous circuits. *Technical Report*, (CS0959):1–14, May 1999.