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

CAS-2010-10

M.Sc. Thesis

Implementation and automatic generation of asynchronous scheduled dataflow graph

T.M. van Leeuwen B.Sc.

Abstract

Most digital circuits use a clock signal to synchronize operations, the so called synchronous circuits. Although this clock signal makes the design convenient, especially since practically all commercial EDA tools assume a synchronous design, some advantages can be exploited when using asynchronous circuits; circuits without clock signal. Those advantages can include typical case performance, low power consumption, less sensitive to variability, lower EMI admittance and protection against differential power analysis attacks. Disadvantages of asynchronous circuits include the lack of EDA tools, their sensitivity to hazards and in some cases performance loss.

In this thesis, an asynchronous implementation for a scheduled data flow graph is proposed. This type of circuit contains a lot of operations with different latencies. Thus, the faster operations are delayed by the clock signal in the synchronous case. Performance benefits could be gained when using asynchronous circuits instead of a clock signal. In this case, handshake signals are used to indicate the completion of an operation, instead of a clock signal.

An asynchronous LWDF filter is synthesized. This implementation is analyzed and an optimized implementation is proposed. A complete design flow is created to generate an asynchronous circuit from any given data flow graph.

Implementation and automatic generation of asynchronous scheduled dataflow graph

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Microelectronics

by

T.M. van Leeuwen B.Sc.
born in Nieuwkoop, The Netherlands

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

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 **"Implementation and automatic generation of asynchronous scheduled dataflow graph"** by **T.M. van Leeuwen B.Sc.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: October 29th, 2010

Chairman: _____
Prof.dr.ir. A.J. van der Veen

Advisor: _____
Dr.ir. T.G.R.M. van Leuken

Committee Members: _____
Dr.ir. A.J. van Genderen

Abstract

Most digital circuits use a clock signal to synchronize operations, the so called synchronous circuits. Although this clock signal makes the design convenient, especially since practically all commercial EDA tools assume a synchronous design, some advantages can be exploited when using asynchronous circuits; circuits without clock signal. Those advantages can include typical case performance, low power consumption, less sensitive to variability, lower EMI admittance and protection against differential power analysis attacks. Disadvantages of asynchronous circuits include the lack of EDA tools, their sensitivity to hazards and in some cases performance loss.

In this thesis, an asynchronous implementation for a scheduled data flow graph is proposed. This type of circuit contains a lot of operations with different latencies. Thus, the faster operations are delayed by the clock signal in the synchronous case. Performance benefits could be gained when using asynchronous circuits instead of a clock signal. In this case, handshake signals are used to indicate the completion of an operation, instead of a clock signal.

An asynchronous LWDF filter is synthesized. This implementation is analyzed and an optimized implementation is proposed. A complete design flow is created to generate an asynchronous circuit from any given data flow graph.

Acknowledgments

I would like to thank my advisor Dr.ir. T.G.R.M. van Leuken for his assistance and advice during my research and writing of this thesis. I would like to thank H.J. Lincklaen Arrins for his support on the Scheduling Toolbox and A.P. Frehe for his support on the ICT infrastructure.

I would like to thank my mom and dad for giving me the opportunity to do this study.

T.M. van Leeuwen B.Sc.
Delft, The Netherlands
October 29th, 2010

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Most digital circuits today use a clock signal, a signal that oscillates between 0 and 1 at a predefined period. This signal can be used to indicate that a certain operation has ended and the next operation can begin. One or both clock transitions can be used to propagate data to the next operation using a flip-flop.

The end of an operation, and thus valid data, can only be guaranteed by predefined timing constraints shorter than the period time of the clock. These timing constraints should hold for the worst-case scenario, the longest path under the worst operation conditions and manufacturing imperfections, because the exact value of these is not known on forehand.

Because different types of operations can have different delays, faster operations will always be finished before the next clock transition. Thus, these faster operation are slowed down by the slowest operation.

Spreading the clock signal across a large IC is not trivial. Complex algorithms are developed to create a clock tree that spreads the signal over the entire IC without to much skew, i.e. the difference in arrival time at different locations on the IC. A clock tree designed to cope with these problems can draw a significant amount of power, up to 40% of the total power[3].

An alternative for circuits with a clock signal is asynchronous circuits. In these circuits, the operations indicate themselves when they are done. Handshaking is used to indicate the successive block of logic to start the operation, and to indicate the preceding block to remove the data from its output. Asynchronous circuits can reduce power consumption by 70% compared to synchronous counterparts[14].

Because operations can indicate when they are finished, the speed does not necessarily depend on the longest path, but it can also depend on the actual path. This can result in average-case performance instead of worst-case performance. Also, the execution time of an operation is not matched with the execution time of other operations, i.e. a fast operation will actually be faster than a slow operation.

Battery life of mobile devices like mobile phones and MP3 players could be significantly increased by the use of asynchronous circuits. However, the market for these devices demands quick time-to-market. If implementing an asynchronous circuit would be as easy as implementing a synchronous circuit, a designer could take advantage of these type of circuits without sacrificing design time.

Scheduled circuits, a type of circuits that implement hardware reuse, can easily be implemented as a synchronous circuit using specialized software, but there was no method or software to automatically implement these types of circuits

asynchronous. In this thesis, an optimized solution is proposed for implementing asynchronous scheduled circuits automatically.

1.2 Goals

The ultimate goal of this thesis is to have an automated design flow from data flow graph to a layout which is faster than the synchronous counterpart. To achieve this goal, a number of sub-goals have been specified

- Create a synthesizable description of the controllers for a given asynchronous scheduled circuit
- Implement the asynchronous controllers in a standard-cell UMC90 library
- Implement the complete asynchronous scheduled circuit in a standard-cell UMC90 library in such a way that it is systematic, repeatable and suitable for automation
- Optimize performance to outperform the synchronous counterpart without losing the systematic and repeatable approach.
- Create a completely automated design flow from specification to layout
- Analyse the results of different automatically generated asynchronous circuits

1.3 Contributions

The contributions of this thesis include:

- A set of controller blocks is designed which can implement the control path for an asynchronous scheduled circuit. These controller blocks are implemented in UMC90.
- Handshaking is simplified which results in increased performance and less area.
- The controller blocks are optimized to increas the concurrency of the system, which results in increased performance and the possibility to implement any valid scheduling result using only these controller blocks for the control path. Using the optimized controller blocks, a reduction in the latency of 33% is achieved.
- Different controller implementations are compared using technology mapping and custom layouts.
- The datapath is converted to a latch-based design that fits the asynchronous scheduled circuit better, resulting in increased performance.

- Software is written to automatically generate an asynchronous circuit from any valid scheduling result using the optimized controller blocks and modified datapath.
- Performance of the asynchronous circuit is analyzed at different levels of abstraction.

The main contributions of this thesis can be summarized as a set of optimized controller blocks and software that can automatically generate an asynchronous circuit from any valid scheduling result using only these controller blocks for the control circuit and commercial EDA software for the datapath.

1.4 Outline

In the rest of my thesis, I will first focus on synthesis of asynchronous circuit in general in Chapter 2. All methods, tools and pitfalls related to this thesis are explained. In Chapter 3, the controller network and datapath for an asynchronous scheduled circuit is created and optimized. The complete design flow including the automatic generation of the netlist is explained in Chapter 4. Evaluation of the manually and automatically generated circuit at different levels of abstraction is done in Chapter 5. Finally, in Chapter 6, the conclusions that can be drawn from this thesis are presented and some recommendations for future work are given.

Design of Asynchronous Circuit

2

This chapter explains the relevant theory behind asynchronous circuits which is used in this thesis. The theory behind hazard free logic and why this is required for an asynchronous circuit is explained. Then, a number of asynchronous methods for indicating the completion of an operation are given, where after the handshaking between different operations is explained. Some theory behind scheduling is explained and modifications for scheduling asynchronous circuits are presented. At last, an overview of all software used in this thesis is given and relevant issues with the software are explained.

2.1 Hazard-free logic

Hazards or glitches are undesired signal transitions in a circuit, i.e. the value of the signal is temporarily changed unintentionally. The most common cause for hazards is a critical race, a situation where two signal paths infuence the output but the relative timing of the two paths determines the output waveform, i.e. when the wrong signal arrives first the output unintentionally changes. In a synchronous circuit, a clock signal is used to indicate when the signals are stabilized, and thus glitch-free [26]. Since there is no clock signal in an asynchronous design indicating when control signals are stabilized, the circuit can always respond on input transitions. Thus, the designs require all control signals to be valid at all times during operation, i.e. hazards are not allowed in the control signals of asynchronous controllers. In some cases the data is used as input to the controller. In this case, the datapath also needs to be hazard-free.

To design hazard-free logic, a number of classifications exist, indicating different delay assumptions under which the circuit is functional.

The most robust model, allowing any delay at any input and output of a gate, is Delay Insensitive. This model only allows a very specific type of controllers to be implemented.

One commonly used assumption is that forks can be made isochronic. This means that if a signal transition is seen by one gate, all gates have seen the transition. Quasi Delay Insensitive (QDI) and Speed Independend (SI) use these assumptions [23].

Huffman circuits use (relative) timing assumptions to guarantee the state of the circuit.

The model used in the target library is called Vital. This model allows the simulation of the target library gates to incorporate hazards.

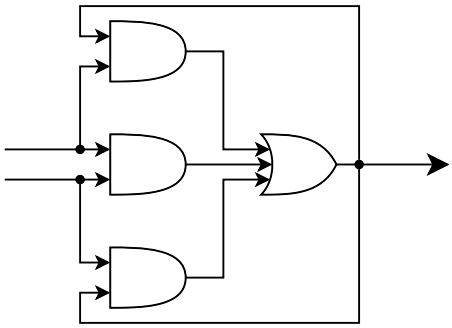


Figure 2.1: Muller-C gate build from AND and OR gates

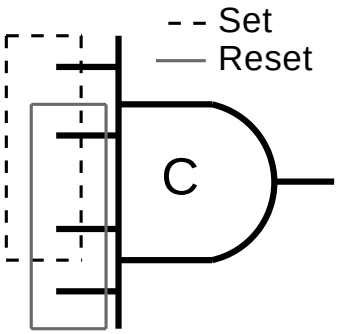


Figure 2.2: Generalized-C symbol

2.1.1 Muller-C gate

A very commonly used element in asynchronous circuits is the Muller-C gate. It is a gate with two inputs and one output. When both inputs are high, the output will become high, and when both inputs are low, the output will become low. When the inputs are unequal, the output remains unchanged, i.e. the gate has hysteresis[33]. An implementation with AND and OR gates can be found in Figure 2.1.

2.1.1.1 Generalized-C elements

Generalized-C elements (or Asymmetric C elements or Standard C elements) are an extension to Muller-C elements. The output of a generalized C element goes high when a specific set of inputs are high, and the output goes low when a specific set of inputs are low. Thus, when all set-inputs are high, the output will go high, when all reset-input are low, the output will go low. A combined input is both a set-input and a reset-input. A symbol for a Generalized-C element with set, reset and combined inputs can be found in Figure 2.2.

Both Muller-C and Generalized-C elements will be used later on in this thesis.

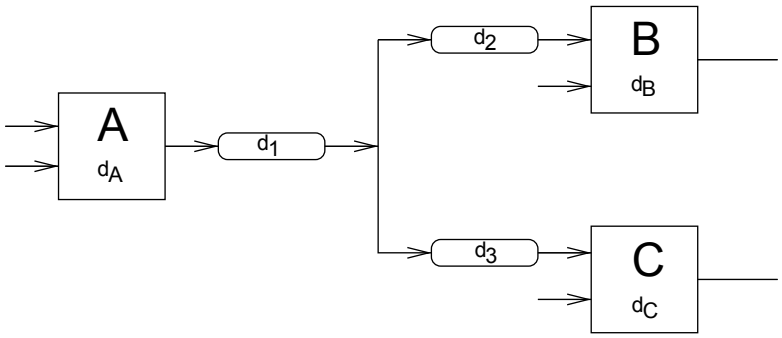


Figure 2.3: A circuit fragment with gate and wire delays. The output of gate A forks to inputs of gates B and C [33].

2.1.2 Delay Insensitive Circuits

The model used for Delay Insensitive (DI) circuits consist of gates, wires and unbounded positive delays. The delays represent both gate- and wire delays. The circuit is assumed to be closed, that is, every gate output is connected to at least one input and every gate input is connected to an output. The environment should thus also be represented by gates. Wires that are forked have uncorrelated delays on each forked element.

In Figure 2.3, an example of a fork can be found. The gate delay only delays the output, thus d_A can be lumped in d_1. Since d_1 delays both parts of the forked signal, d_1 can subsequently be lumped into d_2 and d_3.

A circuit is considered Delay Insensitive if correct operation is still guaranteed when all delays are unbounded positive, i.e. between 0 and infinite.

To make sure the circuit is hazard-free, a signal transition can only take place if the previous transition is completed. This requires a negative edge to be a successor of the positive edge of the same signal and vice versa. This is called acknowledgement. Each signal should be acknowledged to be hazard-free. If a signal is forked, all forked elements can be considered new signals in a delay insensitive circuit. The transition on the input of a fork should thus be a successor of all the previous transitions on the forked elements.

To add functionality to the circuit other than inverting or delaying a signal, gates with more than one input are required. Since the circuit is closed, forks should be present to counterbalance the extra inputs. In a Delay Insensitive circuit, all wires have unbounded positive delays, so the two wires after a fork also have unbounded positive delays. Both delayed signals should be acknowledged in order to make sure they are hazard-free.

A transition can only be acknowledged when it can be observed. For an OR-port, a positive edge on one of the inputs cannot be observed at the output when the other input is high, i.e. there is no way to tell if the second input has changed if only the output is known. The only multiple-input gate that allows all inputs to be observed for both the positive and the negative edge is a Muller-C element.

Thus, Delay Insensitive circuits can only consist of single-input gates (inverters and buffers) and Muller-C elements. More details can be found in [23]

2.1.3 Quasi-Delay Insensitive and Speed Independent circuits

Only a very limited number of circuits can be made Delay Insensitive. To make a more practical circuit, assumptions about the delays in a circuit should be made. In Quasi-Delay Insensitive circuits, some carefully selected forks are assumed to be isochronic and in a Speed Independent circuit, all forks are assumed to be isochronic. In this case, the delays on the forked elements are equal. In Figure 2.3, d2 and d3 are assumed to be equal, d1 and the gate delays are still unbounded. Note that d2 and d3 can then be lumped into d1, like the gate delay, so only one delay element per gate is present in this model. Isochronic forks requires the physical wire to be of equal length from the fork to the gates, and the threshold voltages to be the same for both gates. Using this assumption, an isochronic fork only has to be acknowledged by one of the forked elements. As a result, quasi-delay insensitive and speed-insensitive circuit can contain all types of gates. During layout, the isochronic fork assumption has to be fulfilled.

2.1.4 Huffman and burst-mode circuits

Huffman circuits operate in fundamental mode; it is required that no external input can change until all internal signals have stabilized. When the circuit is stabilized, only one input signal is allowed to change. Since the internal signals are unknown to the environment, timing assumptions have to be made. [36]

Burst-mode circuits are assumed to be stable during input burst. Thus, multiple input changes can arrive as long as the output of the circuit is not expected to change. During the design, it is also made sure that the state signals don't change. When a burst is completed (e.g. all corresponding inputs have changed), the inputs are not allowed to change until the circuit is stabilized. Again, the internal signals are unknown to the environment and timing assumptions have to be made.

Huffman and Burst-mode circuits consist of output and next-state logic, just like Mealy machines. The logic is made free of critical race hazards, for example, by adding additional terms when two adjacent but disjoint terms exist when using Karnaugh maps for two-level logic minimization, as shown in Figure 2.4[37]. In addition, hazard-free multi-level logic minimizers also exist.

When the logic is made critical race free, essential hazards can still exist when a change in a next-state signal is detected before the corresponding change in input is detected by a different part of the circuit. To cope with essential hazards, delay lines in the next-state signal are inserted[15].The value of these delays can only be estimated by making timing assumptions about the logic.

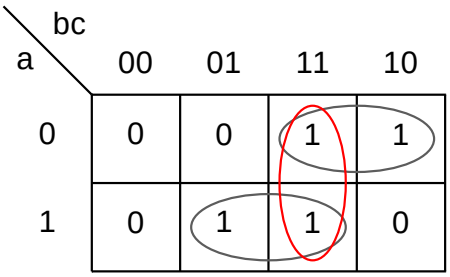


Figure 2.4: Karnaugh map with two adjacent but disjoint terms in grey, with additional term in red to make the circuit critical race free

2.1.5 VITAL model

The simulation model used for logic level simulations in this thesis is a VITAL model[12]. Among other things, this model allows hazards (glitches) to be generated and displayed in the log file. A glitch is defined as follows:

"A glitch occurs when a new transaction is scheduled at an absolute time which is greater than the absolute time of a previously scheduled pending event which results in a preemptive behavior."[16]

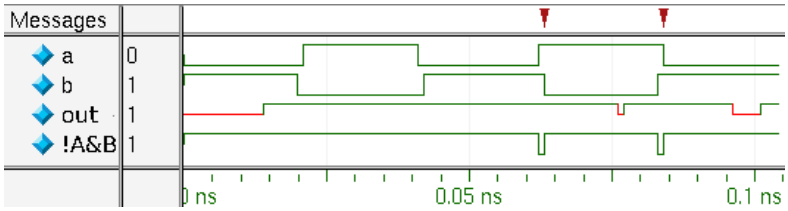


Figure 2.5: VITAL simulation of NAND-port

Since an event is only scheduled when the new value differs from the previous value of a signal in VITAL simulations[1], a signal change on the input of a gate does not produce an event if the output value is not different from the previous output value. For example, when input A of a NAND port changes from high to low, but input B was already low, there is no change on the output and no event is scheduled. When B then changes from low to high, there is still no change in the output, because A is low. However, when the propagation delay for signal A is slightly longer than for B, an output change might be visible in a real circuit. Since this is not modelled in the VITAL model, it can be stated that the input signals are not delayed, only the output signals are delayed (although the actual delay depends on the effective input port causing the output transition). In Figure 2.5, the situations which could lead to a glitch if the delay on one input port is slightly more than on the other input port is shown. The time between the input changes is 1ps. In this simulation, the undelayed logic function is also shown.

The following can be concluded about the VITAL model in combination with DI circuits:

- VITAL glitches occur when a new gate input results into an output event while the previous event is still pending.
- In DI circuits, all signals are acknowledged; any event on a signal must be detected before another event can take place.

Thus, DI circuits simulated with the VITAL model will not create glitches and will operate correctly.

The following can be concluded about the VITAL model in combination with QDI and SI circuits:

- The VITAL model only adds delay to the output of a gate.
- In QDI and SI circuits, all *output* signals have to be acknowledged by at least one of the succeeding gates; any event on the output of a gate must be detected before before another event can take place.

For QDI and SI circuits, the forks have to be isochronic, (unequal) transport delay is not allowed on isochronic forks. When there are no (unequal) transport delays on isochronic forks, QDI and SI circuits simulated with the VITAL model should not generate any glitches and will operate correctly.

Consequently, it is possible to use the VITAL simulations to verify the functionality of the system build from Speed Independent circuits. However, the VITAL simulations cannot be used to demonstrate that a circuit is in fact Speed Independent or Delay Insensitive since it uses fixed delays instead of unbounded delays.

2.2 Completion detection

Because operations need to indicate to the succeeding operation that the data is ready, a scheme is needed to indicate when the data is available. In this subsection, I will explain a number of methods of completion detection and some modifications to these schemes.

2.2.1 Bundled-data

One way of indicating that an operation is done is by a matched delay element. If the operation starts, the input of the delay element is toggled. When the output of the delay element also toggles, the operation is assumed to be completed. The output of the delay element can thus be used to indicate that the succeeding operation can start [33].

A matched delay element is not data-dependent, and thus the delay is matched to the longest path in the operation. Although average-case performance can not be achieved with bundled data, performance improvements can be achieved by

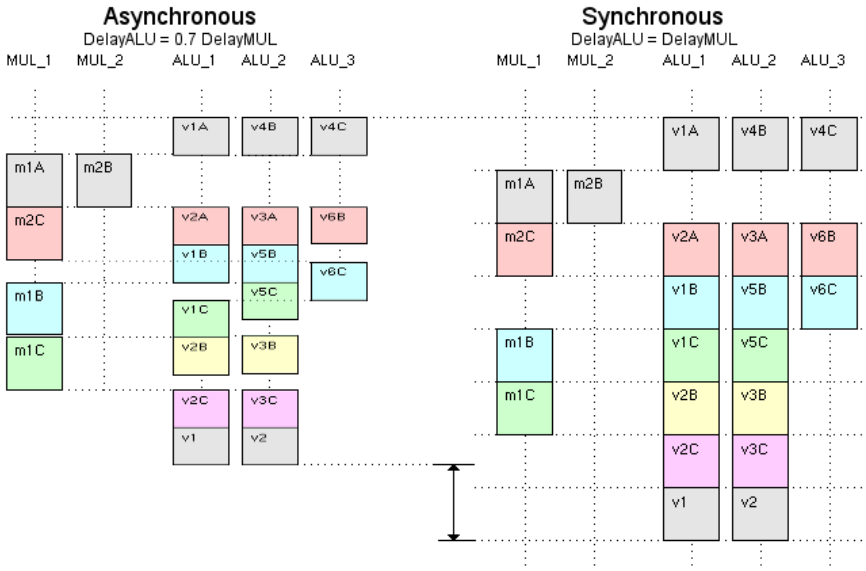


Figure 2.6: Asynchronous Bundled-data vs Synchronous potential performance benefits

delay correlation between the delay element and the operation since the variation between gates within an IC is smaller than the maximum variation taken into account by the design of synchronous circuits. More performance improvements can be achieved by exploiting the difference in worst-case delay between different types of operations, i.e. there is more flexibility in choosing the delay for different types of operations compared to synchronous designs where all operations should complete in an integer multiple of the clock period. In Figure 2.6, the execution sequence is shown for a simple asynchronous LWDF filter and the synchronous counterpart. In this example, it is assumed that the longest path in the ALU is 0.7 times the longest path in the MUL. In the synchronous case, the clock speed is equal to the longest path in the MUL. A speedup of almost 20% is achieved for the asynchronous circuit compared to the synchronous counterpart.

Advantages of bundled-data:

- Datapath can be optimized by widely available EDA tools designed for synchronous logic.
- Hazards on the output of operations are undetected since the data is latched when the logic is completely settled.
- Potential performance benefit due to delay correlation and more flexible delay matching.

Disadvantages of bundled data:

- No data-dependent delay
- Hard to design the right delay element

- Still relies on static timing analysis

2.2.1.1 Speculative completion

To overcome the lack of data-dependent delay, the delay element can be made variable. This can be done by using some internal signals of the operation that indicate if a certain path is active and selecting the right delay value corresponding to the chosen path. For example, when the operation is a simple ripple-carry adder, the propagate signals can be used to estimate the longest possible path [27].

A trade-off can be made in the number of selectable delay elements. More delays result in more area overhead but better performance due to more precise delay matching.

Advantages and disadvantages compared to Bundled Data include:

- Data-dependent delay resulting in better performance
- More area overhead due to extra delay elements and delay-selection circuit
- Data-path operations might need to be adapted to indicate the length of the chosen path

2.2.1.2 Current sensing completion detection

If an operation is active, it consumes considerably more current than when it is finished. If this current can be measured, completion can be detected on an unmodified data-path. The current measurement should be performed in series with the actual operational logic, which decreases the voltage for the operation. A current-sense amplifier is needed to amplify the current signal. Small delay elements are still required to compensate for non-idealities in the current measurement[8].

Advantages and disadvantages compared to Bundled Data include:

- Average case performance
- Performance loss due to supply voltage drop
- Current sense amplifier consumes more power than other schemes

2.2.1.3 Activity monitoring completion detection

Any activity on a wire can be detected by an activity monitor, a device that exploits the delay of an inverter and compares the input and output. When the input and output of an inverter are equal, activity is detected. These activity monitors can be placed at strategic places, so that no activity for a certain amount of time guarantees the completion of the circuit. Delay elements should be matched to the delay between the activity monitors[13].

Advantages and disadvantages compared to Bundled Data include:

- Data-dependent delay resulting in better performance
- More area overhead and power consumption due to activity detection circuits and delay control circuit.
- Data-path need to be adapted to indicate activity detection on strategic places

2.2.2 Dual-rail and one-out-of-X

A complete different method for indicating the completion of an operation is by modifying the data-path in such a way that it indicates its own completion. This is not possible with normal Boolean logic, since it is impossible to tell if a 1 or a 0 is valid. The solution is to add an invalid value for every bit, thus having valid 0, valid 1 and invalid. In practice, when using CMOS logic, this requires two Boolean signals, one for 'valid 0' and another for 'valid 1'. If both signals are low, the data is invalid. It is not allowed for both signals to be high.

Some circuits require all outputs to stay invalid until all inputs are valid and some circuits require all outputs to remain valid until all inputs are invalid, but more relaxed schemes exists depending on the handshake protocol used. For example, if both of the least significant input bits of an adder are valid, the least significant output bit can become valid if it is allowed by the handshake protocol, but it might need to wait for all input bits to become valid [33].

Other encoding are used as well, for example one-out-of-four encoding, in which there is a signal for 'valid 00', 'valid 01', 'valid 10' and 'valid 11', thus requiring 4 signals to transfer 2 bits. This requires the same number of wires as a dual rail implementation, but since only one of the four toggles for every invalid/valid transition, it is more power efficient [22].

If all outputs are valid, the data can be send to the successive operational unit. All outputs should go to the invalid state before the succeeding operation can start, to make sure a completion is not detected by accident. Since a hazard on the output signal can cause invalid completion detection, the output of dual-rail logic should be hazard free.

Advantages of dual-rail:

- Average-case performance
- Does not depend on any timing assumptions

Disadvantages of dual-rail:

- Data-path should be hazard-free
- Data-path requires more area and power
- Commercial EDA tools can't optimize the data-path
- Performance loss due to more complex data-path
- Performance loss due to two transitions (invalid-valid and valid-invalid)

2.3 Handshaking

Since the operations communicate via handshaking, a control circuit should be present to control the latching of data and to communicate with the other operations. Because there is no clock or other mechanism indicating that the control signals are valid, the handshake signals of these circuits should always be valid; the control circuits should operate hazard-free[33].

A number of different handshake protocols are used in asynchronous circuits. For some handshake protocols, different levels of concurrency are possible. This section explains the handshake protocols and the different levels of concurrency.

2.3.1 Handshake protocols

When a completion detection scheme is used with a separate delay element, like bundled-data, two signals are used; a Request (req) from the sending unit to the receiving unit which is delayed by the delay element, and an Acknowledge (ack) from the receiving unit to the sending unit. The request indicates that the data is available and the acknowledge indicates that the data is transferred and can be removed.

There are two widely used handshake protocols for bundled-data, two phase and four phase.

In the two-phase variant, any transition in one of the req and ack signals indicate the availability of data and the completion of the data transfer respectively. For each transfer, there are thus two events, one transition in the req signal and one transition in the ack signal. The actual value of any of those signals have no meaning[34].

In the four-phase variant, a high level of req and ack signals indicate the availability of data and the completion of the data transfer respectively. After the transfer is complete, both req and ack are high and need to go low in the same order to verify that the high level of ack is seen and to allow new data to be transferred[33].

In the dual-rail implementation, there is no request signal, since the data-path indicates valid data. An acknowledge signal goes high to indicate that the data is

latched and can be removed, and goes low again to indicate that all data signals have reached the invalid state and new data can be send.

2.3.2 Concurrency

In a synchronous circuit with edge-triggered flip-flops, every part of a circuit can be active in each clock cycle. Thus, when a circuit consists of multiple operations, all operations are run at the same time, and every operation is finished and accepting the results of the preceding operations on each clock edge.

In an asynchronous circuit, the way data is transferred from one operation to the next depend on the level of concurrency. In the least concurrent case, only every other operation is active, the other operations are waiting for the succeeding operation to finish. This is the basic Muller pipeline described in [33]. In this case, there is one latch between every operation and half of the latches are transparent.

It is not necessary to leave a latch transparent when the corresponding operation is active. If a latch is only allowed to be transparent when both the preceding operation is finished and the succeeding operations is accepting new data, all operations can run concurrent. A latch completion signal should be present to indicate that the data has propagated thought the latch and it can be made opaque[18]. However, when an operation is finished, it has to wait for the succeeding operation to be finished, because the data cannot be saved in the output latch when the previous data is still being processed. In a bundled data pipeline, this is not a problem since it always has to wait for the same operation and it makes no sense to produce data faster than it can be consumed, but when the data could possibly be consumed faster, such as with a scheduled circuit or a circuit with variable delays, this can still slow down the circuit.

If one or more extra latches are added, the concurrency can be increased even more. A new operation can start before the succeeding operation has latched its input. This way, a fast operation can be run more times than its succeeding slow operations in the same time[17].

Besides impact on execution speed, the amount of concurrency also has impact on the possible operation sequence. See Section 2.4.3 for more information about deadlocks as a result of low concurrency.

2.4 Scheduling asynchronous circuits

In this section, problems and solutions for scheduling asynchronous circuits are explained. The result of the scheduling solutions form the base of my thesis, implementing the scheduled asynchronous circuits.

2.4.1 Scheduling algorithms

There are a lot of algorithms available for scheduling operations in a synchronous circuits [24]. Operations are scheduled in time slots, defined by the clock. Each

operation can be scheduled to complete in one or more time slots.

In asynchronous circuits, an operation start when the data and resource become available[28]. In a bundled-data implementation, it is known on forehand when the data and resource become available, since the delay is fixed by a delay element. This delay element does not have to be an exact multiple of a certain time slot, i.e. it can be any real positive number instead of positive integers.

If one would use a scheduling algorithm designed for synchronous circuits to schedule an asynchronous circuit, a number of discrete time slots has to be assigned to each operation. If there are many slots per operation (short time per slot, fine grained), the operation delay used for scheduling (time per slot times number of slots) will be close to the real operation delay. If there are few slots per operation (course grained), the operation delay used for scheduling will be far from the real operation delay. The former will result in a better scheduling result, but the latter will make the scheduling algorithm run faster.

In [28], two new scheduling algorithms are proposed based on the approximation of start times and existing scheduling algorithms. Using the properties of a fixed delay and the fact that an operation will start when data and resource become available, a finite number of possible start-times can be obtained. Using these start times, existing scheduling algorithms (ILP based and Force Directed) can be adapted to fit the asynchronous case better than the existing synchronous variants with finite number of slots per operation.

However, in this thesis, the synchronous list scheduling is used since this algorithm is available in the Scheduling Toolbox used later on in this thesis.

2.4.2 Scheduling results

An operational unit takes data from a source, processes it and sends it to the destination. When all operations are scheduled and allocated to operational units, it is known at what moment an operation should be executed by an operational unit. But since there is no global clock, operations cannot be allocated to time slots. Only the order of operations can be defined, and thus the results of the scheduling should be converted to retrieve the order of operations and their corresponding inputs and outputs.

For example, if ALU1 is scheduled to compute $X = A + B$ at time $t = 1.6$ and $Y = C + D$ at time $t = 4.2$, the ALU should be configured to handshake with the source of A and B, and the destination of X at its first stage and the source of C and D, and the destination of Y at its second stage. Note that the time of the operations is disregarded since it will automatically wait for the sources and destination to become available. Thus, per operational unit, the sources and destinations of the data should be specified, the relative order of the operations and the type of operation in case of an ALU.

2.4.3 Deadlocks

A scheduled asynchronous circuit can potentially reach a deadlock state, where the system stops working. Deadlocks can occur at both controller level and system-level, but this section tries to explain deadlocks on system level (i.e. when the individual controllers are working correctly but waiting on each other) This kind of deadlocks can be avoided by either modifying the scheduling results or modifying the control scheme.

Depending on the concurrency of the control scheme, it might not be possible for an operational unit to handshake with it selves, with the same operational unit at its input as at its output, or even with indirectly dependent operations. In [32], two different controller styles are analyzed for deadlock and modifications to the scheduling results are proposed. Since there is only one delay element and no complete signal from the latches in both controller styles, this single delay element is used to indicate the propagation through both latches and the data operation. As a result, when the data on the output is not latched by the succeeding operation, the data on the input cannot be latched since it would overwrite the output data. If there is a closed chain of operations waiting for the data on the output to be latched by the succeeding operation, the system will be in a deadlock state. Even more possible deadlock states arise when both operands for a certain operation have to be latched at the same time.

Instead of modifying the scheduling results, latches with a complete signal can also be used, or extra delay elements can be added to signal the completion of the latch. When the controllers are modified in such a way that the input and output handshakes can occur concurrently, most deadlocks can be avoided. When a scheduling result can be implemented synchronous, the asynchronous counterpart with concurrent handshaking does not contain deadlocks. This is explained in more detail in Section 3.3. Besides the solution to the deadlocks, the performance is also significantly improved with concurrent handshaking. A disadvantage is the increased area as a result of the increased number of delay elements and more complex controllers.

2.5 Synthesis software

There are a number of different methods and tools available for synthesis of asynchronous circuits. This section explains the different asynchronous specifications and tools that I have used during my thesis, as well as a commercial EDA tools designed for synchronous circuits.

2.5.1 Specifications

In this subsection, Signal Transition Graphs and (Extended) Burst-mode specifications, two methods for specifying asynchronous controllers, are compared.

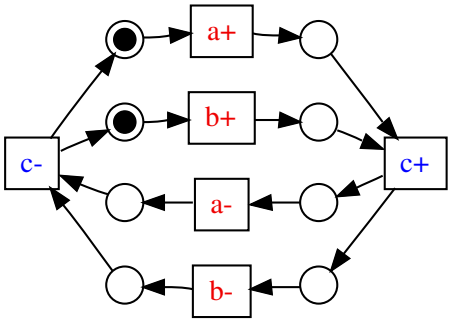


Figure 2.7: STG of a Muller-C element. A + indicates a positive signal transition, a - indicates a negative signal transition. The circles represents places which can contain a token. A transition takes a token from each input place and puts a token on each output place.

2.5.1.1 Asynchronous Signal Transition Graph

Asynchronous Signal Transition Graphs (ASTG's) are a subset of Petri nets where all transitions are signal transitions[6]. An ASTG contains transitions and places, connected by directed arcs. Every place can contain a token. A transition is enabled when all places with arcs to the transition contain a token. A transition can be an input transition which *can* be fired by the environment when enabled, or a transition of an output or internal signal (non-input transition) which *will* be fired by the circuit when it is enabled. If a transition is fired, the tokens from the input places are removed and a token is added to each of its output places The collection of all tokens in the ASTG is called the marking.

In Figure 2.7, an example of an ASTG can be found; a Muller-C gate. When both input transitions, A+ and B+, are fired, the corresponding output transition, C+, is enabled and when the output transition is fired, the input transitions are enabled again. Note that unlike in Figure 2.7, the implicit places, places with only one input, one output and no token, are usually not shown.

A reachable marking is a marking that can be reached by firing enabled transitions. For example, in Figure 2.7, reachable markings are markings where there is one token adjacent to A+ and one token adjacent to B+, or one token adjacent to A- and one token adjacent to B-. It is not possible to reach a marking with two token adjacent to A+ for example.
To create a correct ASTG, the following rules apply:

- The values of all signals should be consistent for any marking. Thus, for every marking, the value for all signals is not ambiguous and does not depend on the preceding transitions. (Correctness)
- In an ASTG, a place can contain only zero or one tokens. There should be no possible sequence of events that can lead to multiple tokens in one place. (Safeness)
- For every transition, the marking that enables this transition should be reachable from every reachable marking. In other words, no parts of the STG should be excluded after a certain input sequence. (Liveness)

The original ASTG specifications can be found in [31].

Since the ASTG is not in a single state at the same time, but its state is the collection of independent tokens, an arbitrary concurrency can be modelled with ASTG's. For example, consider a join element which combines two handshakes, for which the two requests can arrive at any time. When the first request arrives, an ASTG will place a token at the output places of this request transition which can then be used to enable the corresponding acknowledge transition independent of the other input request. Another token can then be used to enable the output request transition, which will only be enabled when both input requests have taken place.

The environment has to be included in the ASTG as well. Since an ASTG can only specify signal transitions and not signal levels, all input signal should be hazard free.

SIS was the first program that could synthesize an ASTG into speed-independent hardware[9].

2.5.1.2 (Extended) Burst-mode specifications

Burst-mode specifications (BMS) specify state transitions with input and output bursts. A BMS specifies a number of input transitions that need to take place to initiate a state transition and the corresponding output burst. If all of the input transitions have taken place, the state will change and the output transitions will take place. An input transition has to be monolithic and input transitions are not allowed to arrive until the circuit has stabilized. Any valid BMS can be implemented as a Burst-mode circuits[25].

The fact that new input transitions may only take place after the output burst results in reduced concurrency in the control circuit. For example, consider the

same join element as before, in which the two requests can arrive at any time. Only when both requests are received, there are no further input changes allowed. This means that an acknowledge can only be send when both inputs have arrived. If a state transition is specified to occur when only one input request has arrived, a state-change and output burst is initiated before the next input request has taken place. Since the next request does not depend on this output burst, it is not guaranteed that the next input request will arrive after the state change and output burst have completed. This would lead to circuit malfunction and is therefore not allowed.

To increase the specification freedom, the Extended Burst Mode is specified. Besides the normal input-bursts, the following input signals are allowed[20]:

- An input can be allowed to change during different states and state-changes, but does not prevent the state to change. If an input is allowed to change it is bound to change eventually. The input change still has to be monolithic. (Directed don't care)

- A level input is added, in which a state transition takes place if the level is correct. Any state change has to contain at least one transition input, which determines when the level inputs are checked. This means that the input does not have to be hazard-free. In fact, if every input is a level input except for a single clock input, a synchronous FSM is specified. (Conditionals)

When a state-change has taken place with a directed don't care on a certain input transition X, the next state cannot be initiated only by input transition X, because it would make the second state-change unstable; i.e. it might be enabled during the first state-change. Considering the previous join element again. If the a request arrives, an acknowledge is send directly, thus a state change is initiated. The second request can arrive at any time, thus has a directed don't care. Now, the second request has a directed don't care on the first state change, but is the only input transition required for the second state change, which results in an unstable state-change and is thus not allowed. Extra required transitions should be added, which would reduce the concurrency of the circuit. Thus, an Extended BMS circuit can model less concurrency than an STG, as a result of the prohibited unstable state-changes.

2.5.1.3 Standard Cell Libraries

A standard cell library contains the transistor layout for a large number of gates, along with the specifications in terms of logic function, area, speed and power consumption. Standard cells libraries allow the design of the high level functionality to be separated from the physical aspects of the design. In this thesis, the target library is the Faraday FSD0A_A UMC90 library [10].

Faraday library The Faraday library contains a large number of logic gates, flip-flops and latches for the UMC90 technology. It does not contain Muller-C or generalized-C elements.

The specifications of the gates are available in a large number of file formats, supporting every popular commercial EDA tool. However, the specifications of the gates are not available in *genLib*, a file format containing logic function, speed and area of gates which is used in a large number of research tools including most asynchronous controller synthesis tools.

2.5.2 Synthesis Tools

In this subsection, I give a brief introduction of the synthesis tools I have used in my thesis to synthesize asynchronous circuits.

2.5.2.1 SIS

SIS is a "system for sequential circuit synthesis". SIS can generate the logic equations, truth-table and technology mapping for both synchronous and asynchronous circuits. The input format used for asynchronous circuits is ASTG and the generated circuits are hazard-free under the unbounded gate delay mode, i.e. the resulting circuits are Speed Independent [31].

The netlist output format of SIS is not compatible with commercial EDA tools, but a translator exists that converts the output to verilog, a format which is compatible with commercial EDA tools.

Reset circuitry is not available for SIS, thus the feed-back signals have to be reset by a user-generated circuit. Also, the feed-back signals have to be connected to the input of the circuit manually. This makes SIS a relative user-intensive synthesis tool.

2.5.2.2 Petrify

Petrify can generate Speed-Independent circuits from ASTG's using either Generalized-C elements, Technology Mapping or Complex Gates.

It uses all the signals in the system to create a state diagram. Initially, only the input and output signals are used to create this diagram. States that are not reachable via the specified input sequences are excluded. Then, it maps the specified outputs on the reachable states. In Figure 2.8, the state graph of a Muller-C element can be found. The binary values of the input- and output signals are shown in each state, in the order A B C, where A and B are the two inputs and C is the output. Adding more signals will double the state diagram. This phenomena is known as state explosion [4]. Due to state explosion, only small circuits can be synthesized with Petrify.

If multiple states exist with the same binary values for all input and output signals, but different future behaviour, Petrify can add internal signals to the sys-

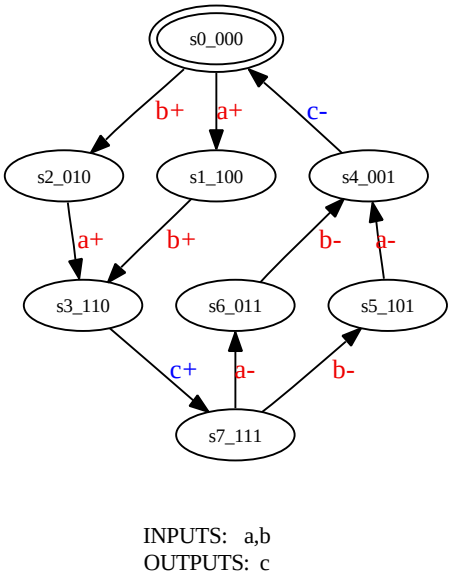


Figure 2.8: State Graph of a Muller-C element. The binary value after the state number represent the value of signals a, b and c.

tem in order to distinguish between otherwise identical states, so called Complete State Coding (CSC) signals.

Petrify can use timing assumptions to exclude states from the reachability graph or optimize the CSC signals. When Petrify uses timing assumptions, every assumption has to be fulfilled during layout by hand.

When using technology mapping, timing assumptions are not allowed. When the environment can be considered slow, it can still be assumed that the environment responds only after all CSC signals have changed; there is no reason why this should not be the case with Technology Mapping. To overcome this restriction, Petrify can be configured to create the CSC signals with timing assumptions but don't do technology mapping. Then, the CSC signals are changed from internal signals to output signals to make the specification SI again and synthesizable with technology mapping.

Not every STG can be synthesized by petrify. Making timing assumptions increases the possibility of finding a solution.

When a valid solution is found, petrify can map the output on a genLib library of gates and output a Speed Independent verilog netlist. This can thus be read commercial EDA tools, given that there is a genLib specification of the target library used in the commercial tools. However, the reset circuitry is still behavioral and the inverters are assumed to have zero delay, as explained in the next paragraph. When using Technology Mapping, all the gates from the genLib library are assumed to be hazard-free under the speed-independent model.

Petrify can also output Genealized-C or Complex Gates implementations. In both these implementations, Petrify outputs a behavioral specification of these

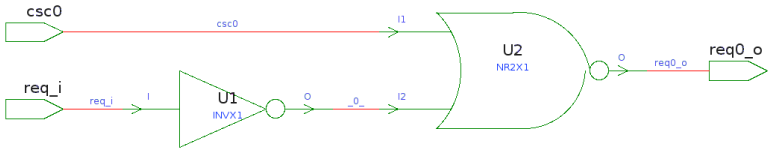


Figure 2.9: The logic for req_o in forkreq block

elements and additional information in the log file which can be used to create the Complex Gates or Generalized-C elements at transistor level.

Since the circuits used can contain internal signals and fed-back output signals, these variables need to be reset to get the circuit into the initial state. Petrify can generate reset logic for these internal signals at behavioral level. However, some outputs of the circuit may be in an unknown state until all inptus have the right value, since petrify assumes that the input signals are as specified. If two petrify circuits are connected together, both having inputs and outputs to each other, the signals might never be initialized.

Speed Independent Technology Mapping Although Petrify can perform speed-independent technology mapping according to the manual, the resulting netlist can contain inverters which need to have a small delay compared to all other gates. In a true speed-independent circuit, the delay of any gate, including inverters, can be zero to infinite.

For example, in Figure 2.9, logic for one output signal for the forkreq controller created with Petrify's technology mapping function is depicted (later used in this thesis). The corresponding STG is shown in Figure 2.10. All irrelevant signals are left out. One part of the circuit (not shown) will make sure that the signal *csc0* will go low after *req_i* has gone low. Now, assume that the inverter is slower than the logic for *csc0*. The following chain of events will be possible (since _0_ will not go high fast enough):

$$req\_i \uparrow \Rightarrow \_0\_ \downarrow \Rightarrow req\_o \uparrow \Rightarrow csc0 \uparrow \Rightarrow req\_o \downarrow \Rightarrow req\_i \downarrow \Rightarrow csc0 \downarrow \Rightarrow req\_o \uparrow$$

Since $req\_i \uparrow$ has not fired for the second time, $req\_o \uparrow$ should also not fire for the second time, but since *_0_* is still low, it does. To prove that the circuit is not SI, lets continue to see what chain of events could be executed. After $req\_0 \uparrow$ has fired, the *csc0* signal can go high again, *req_i* can go high and $\_0\_ \downarrow$ is excited while $\_0\_ \uparrow$ is still pending. Thus, $\_0\_ \uparrow$ is not acknowledged. To notify the user about this behavior, Petrify adds the text "This inverter should have a short delay" above each inverter of which the output is not acknowledged.

Since we use another translation step from the resulting Petrify netlist to the target library netlist, we can affect the relative delay of the inverter by transistor sizing, but only during layout this relative timing constraint can be guaranteed.

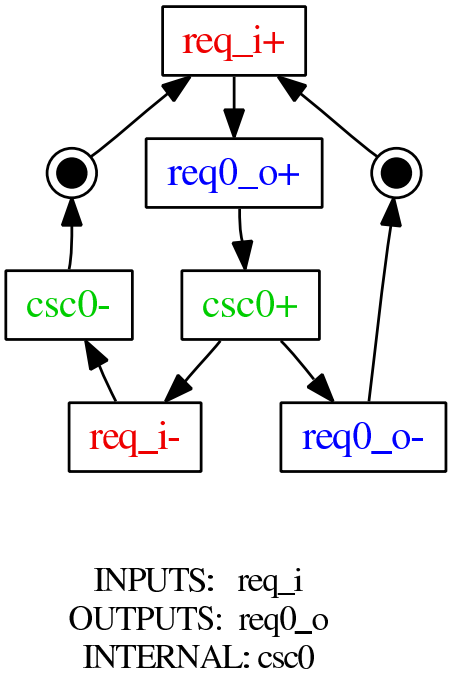


Figure 2.10: The STG for the req_o in forkreq block

2.5.2.3 Digital Gate Compiler

Digital Gate Compiler (DGC) is a tool that can, among other things, generates Burst-mode circuits from (Extended) Burst-mode specifications. DGC also uses genLib as library format and the output is a VHDL netlist of the gates from the library[19]. DGC will generate hazard-free logic for the Huffman machine and can also add the correct delay in the feedback path from the timing information in the genLib library [9]. DGC can generate a reset for the feedback network as well as for the outputs that are not fed back.

2.5.2.4 Minimalist

Minimalist is a tool that generates Burst-mode circuits machines from (normal) Burst-mode specification. Minimalist is designed to investigate in design-space exploration for BMS machines[9]. Thus, Minimalist has a large number of options for trading off performance and area. The output of Minimalist can be a 2-level NAND circuit with inverters for some inputs or a circuit with Generalized-C elements. There is also a hazard-free multi-level logic minimizer available, but without support for reset circuitry[11]. The NAND circuit can easily be mapped on a library, given that there are NAND-ports available with enough inputs.

2.5.2.5 DesiJ

DesiJ is a tool that can solve the state-explosion problem by automatically decomposing ASTG's. For every output signal, it creates a seperate ASTG containing the desired output signal and all signals required to create that output. DesiJ can decompose very large STG's in reasonable time; it does not suffer from state explosion since it does not generate the reachability graph[30]. However, being a tool under development, some features are missing. For example, markings are missing after a circuit is decomposed, the netlist has to be specified manually to compose the original ASTG from the components. Also, some output signals are specified by invalid ASTG's which cannot be synthesized.

2.5.2.6 Design Compiler

Design Compiler is a commercial EDA tool for logic synthesis. It can create a netlist of Standard Cell's from behavioural hardware description language (HDL) code (VHDL and Verilog). Design Compiler only supports synchronous circuits; all timing constraints are related to a clock and it cannot generate hazard-free logic. However, a number of functions can still be used in asynchronous circuit design.

Muller-C synthesis Although DC can compile a behaviour description of a Muller-C element based on if-statements in a VHDL process, it will use a D-flip-flop to save the state of the output and generate the clock signal for this register from the inputs.

A more practical way to implement a Muller-C in DC is a structural description. The schematic of Figure 2.1 was created with VHDL AND and OR statements. Using this description, Design Compiler is able to map the logic for a Muller-C element to gates from the library. Although the input-to-output timing can easily be constrained, the internal feedback node generates a timing loop and is thus ignored by DC.

The internal feedback can be made external to avoid the timing loop and make the timing constraints easier. This can be done by removing the feedback path from the VHDL code, and instead using extra input (C_Feedback). The timing constraints are simple to setup because the feedback path is no longer in the schematic and it is a simple input-output delay.

It is still possible that Design Compiler generates a solution with internal critical-races, thus generating hazards. However, when timing constraints are not too strict, Design Compiler will map a single complex gate to implement the Muller-C element which is critical-race free.

Library conversion Since the Faraday library is not available in genLib, a method has been developed to be able to use the technology mapping function of genLib-

aware tools without the actual genLib file for the target implementation and without losing the Speed Independent properties of the circuit.

A generic genLib library has to be obtained of with each gate is available in the actual target library, although not all target library gates have to be available in the generic genLib library. Such libraries are available from different sources. For this thesis, the default library that came with DGC is used. Then, each gate of the generic genLib library has to be specified in an HDL. This is easy since the equations are already present in the genLib file. Simple regular expressions are used to convert it to VHDL behavioral descriptions.

Using Design Compiler, the netlist of generic genLib gates can be converted to the Faraday library. Every genLib gate can be read in by Design Compiler from the HDL specifications created before. If the netlist of genLib gates is then opened and the compile command is executed, (not compile-ultra), the original netlist is not changed but every genLib gate is compiled individually[35]. In most cases, the genLib gates will directly be mapped on the DC target library gates if those gates are available and the timing constraints can be met. If the timing constraints are very strict, DC can split the gates and cause critical races. Thus, when using timing constraints, the total number of gates has to be checked to make sure no ports are split.

Since the genLib library was not written for this specific implementation (i.e. it is a generic genLib library), the timing and area information from the original genLib gates will not represent the actual figures. This can degrade the area/delay trade-off.

Delay generation For bundled-data asynchronous circuits, it is required that a matched delay element is generated. Design Compiler can generate delays for fixing flip-flop hold time. By specifying the right constrains, this feature can be exploited to generate matched delay elements. First, a virtual clock has to be declared to be able to specify timing constrains. Then, the in- and output minimum and maximum delays should be specified. If these delays are specified as 0, Design Compiler will assume that there is a zero-delay flip-flop directly at the input and output ports. Since we use hold times, *set_fix_hold* should be executed for this clock. Because there are no real flip-flops, the required hold time is 0, which does not suit our delay requirements. If a multicycle path is specified by *set_multicycle_path*, the hold time is constrained as the number of cycles minus 1[35]. So if we specify a multicycle path of 2, the actual hold time is constrained to one clock cycle. We can leave the setup time 1 clock cycle, but that would require the minimum delay to be exactly the same as the maximum delay, one clock cycle, resulting in excessive large circuits since different types of inverters and buffers are chained instead of delay elements, to be as close to the required delay as possible. If the multicycle path is specified for both setup and hold time, the setup time can be two clock cycles. If this is too flexible, the setup time can be reduced by specifying a nonzero maximum in- or output delay.

Using this method, a minimum and maximum delay can be specified which Design Compiler will try to satisfy under the selected operation conditions. The command set_cost_priority can be used to prioritize the minimum over the maximum delay. For the complete list of commands, see Appendix C.1.

Datapath synthesis The datapath for bundled-data asynchronous circuits does not have to be hazard free, because the delay element indicates valid data. Design Compiler is able to synthesize the datapath with latches instead of flip-flops if a behavioural description for level-sensitive latches is entered. The control signals for the latches are considered clock signals by design compiler and should be used as such for the timing constraints.

When a design consists of a behavioural datapath and a netlist of hazard-free controllers, Design Compiler should not try to optimize the hazard-free controllers any further, since this might compromise the hazard-freedom. Setting the Don't touch parameter to true for all asynchronous controllers will make sure the controller netlist doesn't change while compiling the datapath.

2.6 Conclusion

In this chapter, an overview of relevant theories and synthesis tools for asynchronous circuits is given. Hazard-free controllers can be created using one of the availible implementation styles. Togeter with a completion detection method, different handshaking protocols can be implemented by the hazard-free controllers. In the next chapter, these theories are used to create and optimize the controller network and datapath for an asynchronous scheduled circuit.

Asynchronous Scheduled circuits

3

This chapter explains the synthesis of an asynchronous LWDF filter. Non-synthesizable VHDL code was supplied for two asynchronous implementations. These implementations are compared and improved. Two different synthesizable versions are made.

The LWDF filter contains 3 ALU's which can do addition and subtraction (selected by the opcode) and 2 MUL's, which can do multiplications. In total, the LWDF filter needs 17 ALU operations and 5 MUL operations to process each sample, so the ALU's and MUL's (Operational Units) are scheduled to process multiple operations per sample.

3.1 VHDL models

In this section, the original VHDL code of the LWDF filter designed by Michael Simmonds[32] is explained. In particular, the functionality of the controllers is explained. The scheduling and datapath are not discussed.

3.1.1 Method by Cortadella.

One of the two versions of the original code is based on the de-synchronisation model proposed by Cortadella et al[2]. This section explains that version.

In the LWDF filter, one latch controller and two level-sensitive latches are added to each operation. Although the in- and output handshakes are concurrent in this model due to the use of the concurrent desynchronisation mechanism, one latch controller per operation is used in the LWDF, so only one of the two latches is effective and only every other operation can be active at the same time.

Cortadella's method does not include a strategy for hardware reuse, and would thus require 22 separate resources. To make the latch controller of Cortadella compatible with the scheduled circuit, an additional controller is added to each latch controller that will decide which in- and outputs the latch controller will be connected to. Since this controller controls handshake signals, it should also be hazard-free. The resource combined with the latch controller and additional controller is called an Operational Unit. The additional controller was designed to connect the input and output of the latch controller the right preceding and succeeding operational units until both the input and output handshakes were completely finished. If there are two inputs, and thus two input handshakes (for example, an ALU with two operands), the requests are combined with a Muller-C element. Thus, an acknowledge can only be send if both requests have been

received and processed, slowing down the earlier input.

When an operational unit needs to send its data to multiple destinations, the output handshakes are executed in sequence by the additional controller. In that case, the handshakes are completed by the additional controller instead of the latch controller.

Using this method, much of the advantages of the concurrent desynchronisation mechanism are counterbalanced by the inefficient additional controller.

3.1.2 Integrated controller

Simmonds also created a controller which combined the latch controller and additional controller. Instead of just selecting the in- and outputs for the latch controller, the integrated controller completes all the handshakes by it selves. The input handshakes and latches are implemented twice, so that the earlier input can receive an acknowledge before the later request is received. Also, when data needs to be send to multiple destinations, the output handshakes are completed concurrently.

The integrated controller is much lager than the additional controller mentioned before. For example, the controller for an operational unit with 7 different operations has almost 100 transitions if it is described as a Signal Transition Graph. Although this level of concurrency cannot be described in a Burst-Mode Specification, a less concurrent version has already more than 50 states.

Although I was able to synthesize a controller for a much smaller operational unit with only a single operation using Petrify, SIS and DGC, the controller for 7 operations could not be synthesized by the available tools described in chapter 2, due to the result of state explosion.

3.1.2.1 Hold Data

Because the integrated controller is equipped with two latches per operation, it is possible to use both latches to store global outputs. The input of the second latch (output of the operation) is also made available as output port and used when the second latch already contains a global output or feed-back. The first latch holds the input of the operation and the output of the operation will thus stay stable. This is the only effective use of the odd latch in this VHDL model.

3.2 Decomposed handshake blocks

In an attempt to synthesize the integrated controller, some of the controllers were completely specified as STG and XBMS. The XBMS was not synthesizable because the inputs handshakes should be finished independently; this would result in unstable state changes as described in Section 2.5.1.2.

The smaller STG's could be synthesized by Petrify and were functionally equivalent to the VHDL code. However, for the operations with more than two stages,

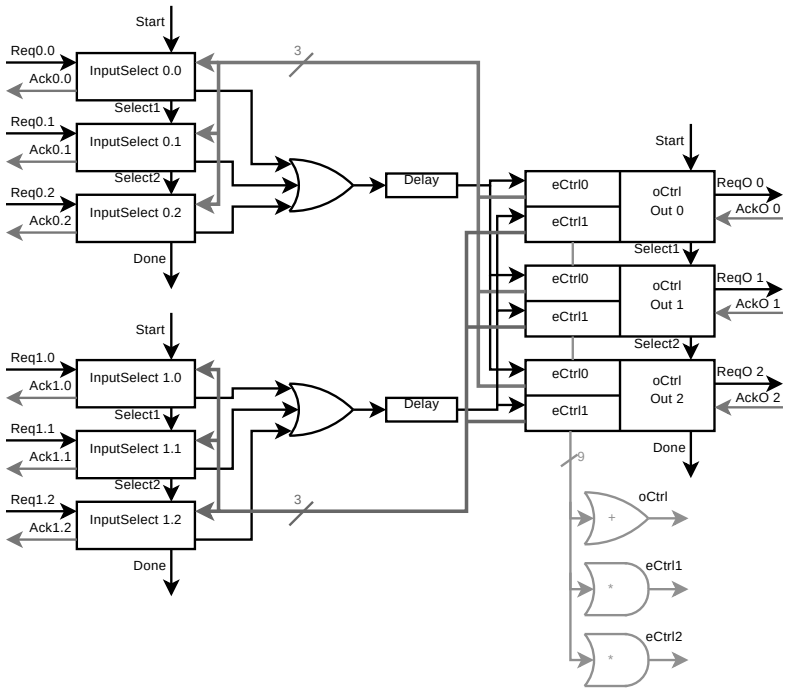


Figure 3.1: Operational Unit FSM split up in handshake blocks

the STG's could not be synthesized as a result of state explosion. To overcome the problem of state explosion, the controller can be decomposed into smaller blocks[29]. DesiJ was unable to generate valid ASTG's for the integrated controllers, so manual decomposition is performed. In Figure 3.1 it is shown how the integrated controller can be split up into a number of handshake blocks that are only active once per cycle. The handshake blocks implement the integrated controllers by Simmonds. A circuit consisting of partly Simmonds code and partly handshake blocks was successfully simulated. When the complete system consists of only handshake blocks, some simplifications can be applied. Since the integrated controllers requires the input handshakes to be finished independently, STG is the only possible specification style and the controllers should be implemented in Speed Independent or Quasi Delay Insensitive circuits. Petrify and SIS both satisfy these requirements, but Petrify was preferred since it can directly generate a verilog netlist and reset circuitry, and it is more user friendly.

3.2.1 Handshake blocks for the integrated controller

In this section, all handshake blocks from Figure 3.1 are explained. All handshake blocks are specified in STG and implemented using Petrify and a default genLib gate library using the method explained in Chapter 2. The STG's of the handshake blocks can be found in Appendix A.1.

3.2.1.1 Inputselect

The inputselect block selects the correct input handshake from the preceding operational unit and passes it on to the outputselect block.

Each inputselect block will send a request out when it has received a start signal and input request. The request and start signals can arrive in arbitrary order. when it receives an ack signal from the outputselect element, it indicates that the data is latched and the inputselect block can complete the handshakes on both its input and output concurrently. If both handshakes are completed, the select out (finish) signal is made high to start the next stage.

3.2.1.2 Outputselect

The outputselect block controls the latches and handshakes with the right succeeding operational unit.

The outputselect block is slightly more complicated than the inputselect block, since it also has to control the latches. In case of a 2-input operation (i.e. MUL and ALU but not REG) the part that handles the input handshake and controls the even latches is implemented twice, since it has to control two latches and handshake with two inputselect blocks concurrent. When a start signal is received and the outputselect block receives a request, it makes the even latch opaque and sends an ack signal, indicating that the data can be removed from the input. When both input requests are received, and thus the even latches are both opaque, the odd latch is made transparent and the output request is send. When the output handshake is completed, the odd latch is made opaque again and the even latch is made transparent again. The input handshake is completed, indicating that the next data can be processed, and the finish signal is made high.

3.2.1.3 Fake request

The fake request block is a replacement for the inputselect block when there is no input handshake.

Sometimes, it is necessary to start a calculation without having an incoming request, for example when the data is a global input, feed-back from a previous iteration or a constant[32]. For this situations, a special block is designed, called 'fake request'. It has the same functionality as the inputselect block, but without the input handshake. It will start a handshake on the output when a start signal is received, and make the finish signal high when the handshake is completed.

3.2.1.4 Fake acknowledge

The fake acknowledge block is a replacement for the outputselect block when there is no output handshake.

When the result of an operation is only used as a global output or feed-back, there is no output handshake. In this case, the odd latch still has to be activated

to save the data. Because it takes some time for the data to propagate through the odd latch, an extra delay element is required to indicate that the data has propagated trough the odd latch. By simply connecting a delay element between the request out and acknowledge out of an outputselect block, the fake acknowledge block is obtained.

3.2.1.5 Hold data

The hold data block is a replacement for the outputselect block when previous data needs to be hold.

If the result of the second last operation should not be overwritten, because it is a global output or feedback instead of an intermediate value, the odd latch should not be activated during the last operation. Also, because the data should not be overwritten before it is read out, the even latches stays opaque until the start signal is made low again. Because the fake acknowledge block controls the latch in a different way, the hold data block has to be designed seperately.

Making the results available without overwriting the previous results is not implemented in the synchronous counterpart, and thus also not exploited by the scheduling algorithm. This block is thus only used for the given LWDF filter and not needed later on in this thesis for automatic generated circuits.

3.2.1.6 Fork request

The fork request block (not shown in Figure 3.1) allows one outputselect block to handshake with multiple inputselect blocks of different succeeding operational units.

Some intermediate results are used in multiple succeeding operational units. Since the outputselect block has only one output handshake, this handshake should be forked. The usual way to fork a handshake signal in an asynchronous circuit is to connect the requests together and combine the acknowledge with a Muller-C element[33]. Because the request will only go low after all the acknowledge signals have gone high, each individual succeeding operation can only complete their handshake after all the other succeeding operations are finished. If the latency of the succeeding operations differ, or one of the succeeding operations is stalled by another handshake, this kind of fork will halt the faster operation until the slower operation has send an acknowledge. This can decrease performance and even cause deadlocks.

To speed up the forked requests and be compatible with the given LWDF filter, a fork request block is designed. This block has one input handshake and two output handshakes. If a request in is received, it sends two output requests. If an output acknowledge is received, the corresponding request goes low, completing the handshake. If both output acknowledges are received, the input handshake is completed.

3.2.2 Handshake blocks interconnects

Because the requests from the inputselect to the outputselect blocks have to go through a delay element, they are OR-ed together to avoid the need of separate delay elements. The MUX can be controlled by the select signal that ripples through the input controllers. The opcode for the ALU can be derived from the select signal of the output controllers by simple VHDL statements, since it only modifies the data path and as a result it doesn't has to be hazard-free.

To be compatible with the original code, the input acknowledges and output requests have to be OR-ed together if they go to the same operational unit. If the whole system is composed out of handshake blocks, the handshake signals can directly be connected to the right select block. This removes the requirement that the select blocks have to ignore signals on their inputs when start is low and allows the finish signal to go high before the handshake is completed.

3.2.3 Asymmetrical delay elements

Since the design is based on a 4-phase bundled data circuit, an asymmetrical delay element is desired. This means that the rising edge is delayed, but the falling edge is passed as fast as possible. If a symmetrical delay element would be used, the completion of the handshake (e.g. req and ack going low again after ack has gone high) would take as much time as the operation itself and would slow down the circuit in most cases.

The delay elements available in the target library are designed as symmetrical delays[10].

To create an asymmetrical delay, an extra signal is added that is XOR-ed with the output of the delay. A controller is designed to toggle the input of the delay at the rising input edge and toggle the input of the XOR at the falling edge.

When a rising edge appears at the input of the controller, the input of the symmetrical delay is made the inverse of the XOR input, and thus the output of the delay will become the inverse of the XOR input after the delay. This will result in a rising edge at the output of the XOR. Since the circuit is used in a handshake, the input of the delay controller will not change until the output is settled, and thus the input of the XOR will not change during the delay.

When a falling edge appears at the input of the controller, the input of the XOR is made equal to the input of the delay, and since the rising edge has already propagated through the output, also equal to the output of the delay. This will result in a falling edge at the output of the XOR.

Using this method, an asymmetrical delay element is created with the available target library.

Besides a Speed-Independent circuit from Petrify, the controller can also be implemented with two edge-triggered flip-flops, with the inverting output connected to the input of the flip-flop. The flip-flop that controls the symmetrical delay element should be positive-edge triggered, while the flip-flop that controls to the

XOR-port should be negative-edge triggered.

3.2.4 Reset of handshake blocks

In the specifications of the handshake blocks, initial states of the inputs are assumed to be known. The synthesis tool Petrify will only generate a reset for internal signals that are still ambiguous when all inputs are correctly initialized. Because the input signals are not always unambiguous if they are outputs of other Petrify circuits, intermediate states should be reset, as explained in Chapter 2.5.2.2.

If two handshake blocks are connected together, assuming all other inputs are correct, the signals between the two handshake blocks can be ambiguous. Only one of them has to be able to generate unambiguous outputs even when not all input signals are correct, since the other circuit will then have all correctly initialized input signals and thus produce correct outputs. If a chain of handshake blocks are connected together, only half the blocks should generate correct outputs with ambiguous input signals. When these blocks define their outputs correctly, the other half of the blocks will initialize correctly and thus initialize their outputs correctly.

In some cases, not every input signal of a Petrify circuit has to be correct to make the output of the circuit correct. For example, if one input (or internal) signal (signal A) is unambiguously low and AND-ed with another input signal (signal B), and signal B is not used elsewhere in the circuit, signal B doesn't need to have the correct value to make sure the output is correct since the output of the AND will be low independent from signal B. Signal B will be defined eventually, because it is the output of another circuit that will be initialized at a later stage. This idea is also used in Petrify to determine which internal signals needs to be initialized.

Also, it is assumed that the 'start' signal is always valid, since the first one is generated by the external environment, and when the first handshake block is initialized, its 'finish' signal will be defined even if the next handshake block is still uninitialized. This start signal will ripple through all the blocks until all start and finish signals are initialized.

These lead to the following simplifications in the reset circuitry:

1. To produce a correct output, only one (N)AND input needs to be intentionally low, or one (N)OR input needs to be intentionally high, the other can be left ambiguous.

2. Only every other handshake block needs to be able to produce correct outputs with ambiguous inputs.

3. The start/finish signal does never have to be reset, except the global start input which is reset by the environment.

There are three kinds of handshake blocks connected together; the inputselect block, the asymmetric delay and the outputselect block. Since there is only one

asymmetric delay per operational unit, this is a relative cheap signal to reset. Only one other handshake block has to create correct outputs as a result of simplification 2

I chose to reset the inputselect blocks since these blocks are much simpler. I chose to initialize the inputs, since not all inputs are required to be correct in order to create correct outputs, but the output signals can go to different types of blocks and should thus all be correctly initialized. This makes sure I can take advantage of optimization 1 without analyzing all possible succeeding blocks.

If 'start' and 'ack_o' are defined low, circuit inspection shows that 'req_i' can be left ambiguous as a result of optimization 1. Since 'start' can be assumed correct as a result of optimization 3, only 'ack_o' needs to be initialized low.

The fake request is an alternative for the inputselect block, and thus cannot take advantage of optimization 2. The 'start' signal can be left uninitialized again. Circuit inspection shows that the other input of this block, 'ack_o', does not need to be initialized as a result of optimization 1.

The fork request block is a special case since its output handshakes can be left undefined as a result of optimization 2, but its input handshake can't be left undefined as a result of this optimization. Thus, the only ambiguous signal is req_i. As a result of optimization 1, both possible preceding outputselect blocks initialize their request output correctly when their start is low. The input acknowledge of the fork request block (output acknowledge of the outputselect block) will then be correct since all its inputs are correct. If multiple fork request blocks are chained together, the valid request from the outputselect block will be rippled through all the blocks as a result of optimization 1 and the last one will have all inputs correct, rippling the correct acknowledge back until every fork request block is completely initialized. Thus, the fork request block doesn't have to be modified.

Since VHDL supports undefined signals (with the use of the IEEE std_logic library), the correct functionality of the initialization circuit can be simulated with VHDL.

3.2.5 Performance

When the LWDF filter is build with the handshake blocks that implement the integrated controller, the total latency of the circuit is 90 ns when the ALU delay is 3.8 ns and the MUL delay is 5 ns. The reduced performance compared to the synchronous counterpart, which finishes in 40 ns, can primarily be attributed to the reduction in concurrency and controller overhead. To improve the latency of the circuit, new controllers have to be designed which increase the concurrency of the circuit.

3.3 Optimized handshaking

In the circuit described before, each operational unit is combined with one delay and two latches. Since there is only a single delay element for two latches plus

one operation, and there is no finish signal for the latch, two latches has to be transparent at the same time to make sure the data has propagated through both latches. In fact, when all odd latches are removed from the original design, the circuit still operates correctly (except when a hold data block is used).

Instead of removing the redundant latches, they can also be used to speed up the circuit. This can be done by adding an extra delay and making the latch opaque when the data has propagated. In this case, the latches are decoupled and can be controlled individually. Then, an operational unit can start a new operation, even when the succeeding operation is still processing data from the old operation. This will speed up the circuit since all operations can run concurrently.

When the latches are decoupled, the system will be live and flow-equivalent to the synchronous system. This means that there are no more deadlock situations and the output will be correct when scheduling results for a synchronous circuit are used.

This optimized handshaking implements the fall-decoupled model from [2]. The de-synchronization model, which is even more concurrent, could not be implemented because the output data would be overwritten before it is latched by the environment, since there is no handshake associated with the global outputs.

3.3.1 Valid scheduling result

In this thesis, a valid scheduling result is considered a scheduling result that can be implemented synchronous using the Scheduling Toolbox. Two usefull properties are considered:

- A variable can only be consumed one or more cycles after it is produced, e.g. not in the same cycle
- Data that is consumed at a later stage cannot be overwritten in the current stage

To prove that any valid scheduling result can be implemented, the theory of de-synchronization is used since we use scheduling results for a synchronous circuit. To prove that the asynchronous circuit is equivalent to the synchronous counterpart, liveness and flow-equivalence have to be proven. To do so, marked graphs are used as model for the latch control signals. Marked graphs are a subclass of Petri nets for modelling decistion-free concurrent systems[2].

3.3.2 Marked Graphs

A marked graph is a subset of Petri Nets. A marked graph consists of a set of events, directed arcs and tokens. An event is enabled when all incoming arcs contain a token. When an event is fired, all tokens from the incoming arcs are removed and tokens are placed on all outgoing arcs. In this section, the marked graph of a scheduled circuit implemented using the optimized handshaking is explained.

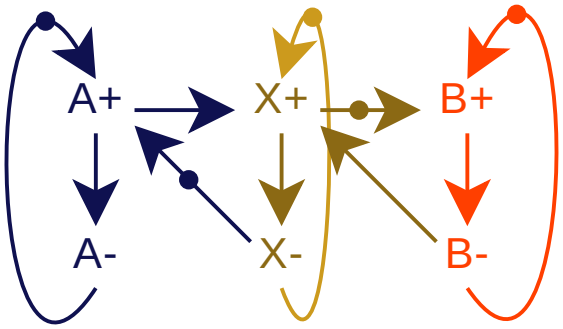


Figure 3.2: Marked Graph of fall-decoupled model

A marked graph can be represented by an STG which has only one input and one output per place.

Fall decoupled model In Figure 3.2, the fall-decoupled model from [2] is depicted. A and B represent control signals for even latches, while X represent control signals for an odd latch. A+ represents a positive edge on the latch control signal, which will make the latch transparant, while the negative edge, A-, will make the latch opaque.

This model is both live and flow-equivalent to a synchronous circuit where an odd and even latch is represented by a flip-flop controlled by a global clock signal.

Join and Forks In order to allow multiple operands to come from different sources, incoming requests can be joined. This is represented by an extra even latch and latch control signal. To allow data to be consumed by different succeeding operational units, outgoing requests can be forked. No extra latch is needed, the outgoing arcs of the marked graph are simply implemented twice. In Figure 3.3, an example marked graph can be found representing an operational unit with multiple inputs and multiple outputs. A and B represent latch control singals for the two operants for the operation, while C and D represent latch control signals for the two different succeding operations.

Hardware reuse When implementing hardware reuse, each input request represents data from different preceding operational units. Thus, different latch control signals have to be used for each preceding operaional unit. The output handshake should also go to the right succeeding operational unit. Since each input request now corresponds to a specific output request, those handshakes should be synchronized. To synchronize those handshakes, global start and done signals are added. In Figure 3.4, a marked graph is shown which implements hardware reuse. The illustrated operation consumes data from A0 and sends the results to C1. Then, it consumes data from A1 and and sends the results to B2. The letters (A, B, C) indicates the input of a certain operational unit, and the numbers indicate

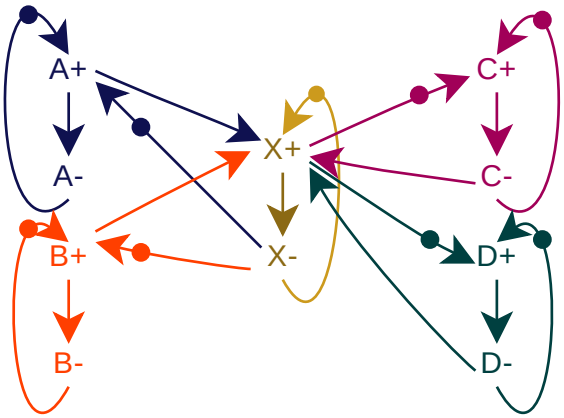


Figure 3.3: Marked Graph of operation with two inputs and two outputs

the cycle from the scheduling result in which the data would be processed in the synchronous counterpart. A higher number consequently indicates later in time. X indicates the odd latch control signal. Note that the numbers do not have to be subsequent, an unused timeslot in the synchronous case is represented by an increase in cycle number of more than 1.

When a valid scheduling result for a synchronous circuit is used, the following restrictions are imposed to the cycles:

1. A_i+ fires X_i+
2. X_i- fires $A_{i+n}+$ where $n >= 1$
3. X_i+ fires $B_{i+n}+$ where $n >= 1$
4. B_i- fires X_{i+n} where $n >= 0$ unless X_{i+n} does not exist

The first restriction is by design. The even and odd latch work together on the same data, since they represent a single flip-flop in the synchronous case.
The second restriction represents the property that for every mux-input, a separate cycle number has to be assigned. This corresponds to the synchronous case, since every operation can be only active once per cycle.
The third restriction indicates that data produced in a certain cycle can only be consumed one or more cycles later, and not in the same cycle.
The fourth restriction uses the fact that data cannot be overwritten in the synchronous case, if it is used at a later stage. If B consumes data in a certain cycle, it should be valid during this cycle. If the producing operation produces other data before the old data is consumed, a register is required to hold the data. Thus, the same registers for the synchronous design are used in the asynchronous design. In the exception, X has finished all operations and should be enabled for the next sample.

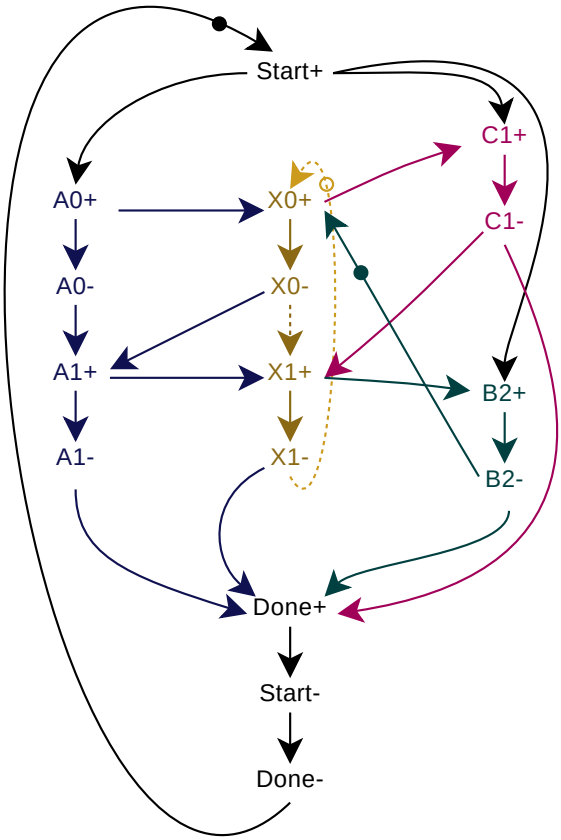


Figure 3.4: Marked Graph of operation with hardware reuse

Complete model For completeness, hardware reuse is combined with joins and forks, to form a complete marked graph for the improved handshake blocks. This model is depicted in Figure 3.5. However, since the joins and forks complicate the analysis but do not introduce new challenges for the proof of liveness and flow-equivalence, the model without joins and forks is used.

3.3.3 Liveness

Liveness can be proven by proving that all directed circuits contains a token[2]. A directed circuit is a closed chain of arcs in a marked graph. In the model for hardware reuse, there are only tokens at the start signal and at the first event of the odd latch control signals. If any directed circuit is followed, it should thus end in one of these tokens.

To prove that all directed circuits contain a token, the arcs are followed backwards from any given event in the circuit until a token is reached:
Any negative event of an odd latch control signal is always fired by the positive event of that odd latch control signal (by design). The positive event of an odd latch is always fired by the last cycle of an even latch, which contains a token, or

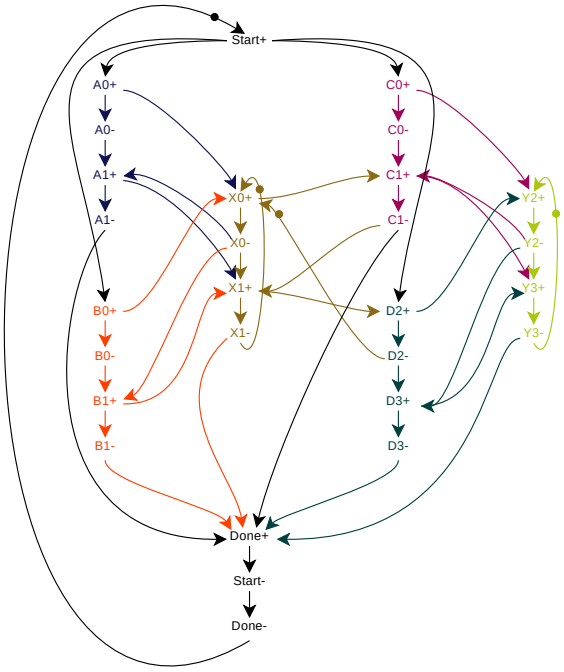


Figure 3.5: Partial Marked Graph of asynchronous scheduled circuit

by a positive or negative event of an even latch at the same cycle or a lower cycle (By restriction 1 and 4). A negative edge of an even latch is always fired by the positive event of that even latch (by design). A positive edge of an even latch is either fired by the start signal containing a token, or by a negative or positive edge of an odd latch at *at least one cycle earlier* (by restriction 2 and 3). Consequently, when this sequence is repeated long enough, any directed circuit without token end in cycle 0 of an even latch. Cycle 0 of an even latch is fired only by the start signal which contains a token. Thus, every directed circuit contains a token and the model is thus live.

3.3.4 Flow-equivalence

Two circuits are flow-equivalent if they have the same set of latches, and the projection of the traces onto a latch are the same in both circuits. Flow-equivalence can easily be demonstrated as a result of the adoption of the fall-decoupled model and the use of cycle numbers.

In Figure 3.4, the first input of even latch A (A0) receives its data from a global input which is always valid. Every even latch that operates at cycle 0 receives its data from a global input since there is no local data available at cycle 0. The adoption of the fall-decoupled model makes sure X0 contains the correct results, as proved in [2], since the latch control signal can only go low after the data has propagated. Then, the scheduling results make sure that the results from X0 are send to the right succeeding operation, C1 in this case. The handshaking

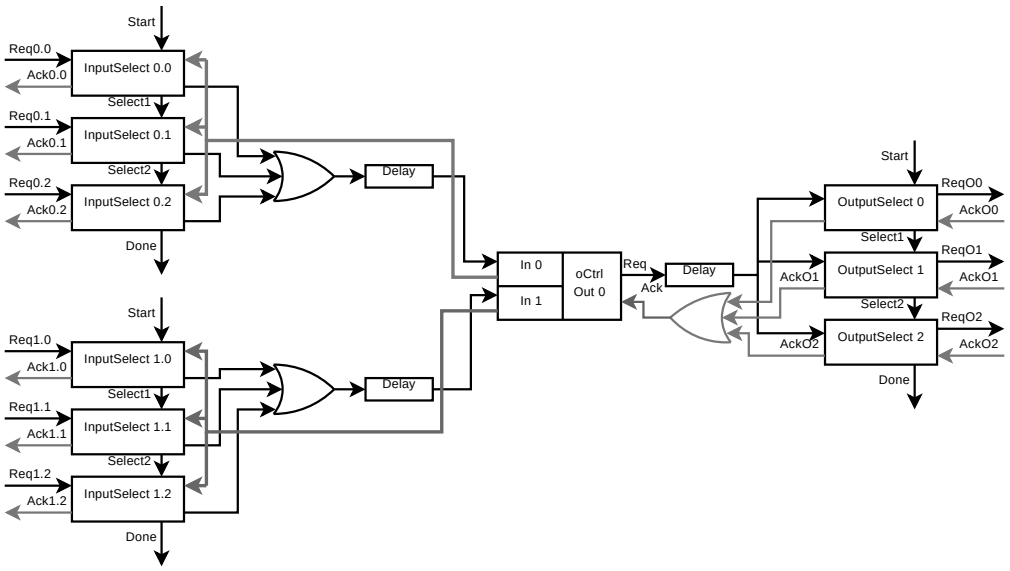


Figure 3.6: More concurrent version of Operational Unit FSM

between X0 and C1 also follows the fall-decoupled model, which ensures correct data transfer. C1 now contains the data as specified by the scheduling results and this will continue until all outputs contain the data as specified by the scheduling results.

In other words, the fall-decoupled model ensures that the data in a latch pair is always valid and the handshaking signals are connected in such a way that the scheduling results are implemented correctly.

3.3.5 Handshake blocks for more concurrent design

For the optimized handshaking, new handshake blocks are designed. Also, the topology has to be modified, to be able to add a second delay to the operational unit. The delay could also be added to the request signals between operational units, but that would require a large number of delays. Therefore, the latch controller is separated from the output select blocks, creating a new handshake signal where the delay can be added. In Figure 3.6, this new design is shown. There is a new latchcontroller and thus the outputselect blocks do not control the latches anymore. Handshake signals from input- and outputselect blocks are OR-ed together to be able to use two simple delay elements and only one latchcontroller. The inputselect block is slightly modified to allow more concurrency at the input and the even latch is controlled by the inputselect block. The latchcontroller implements the fall-decoupled model from [2], and a pair of connected inputselect and outputselect blocks together form another latch controller implementing the fall-decoupled model.

The STG's of the optimized handshake blocks can be found in Appendix A.2.

3.3.5.1 Latch controller

The latch controller controls the odd latch according to the fall-decoupled model and implements a join function if an operator has two operands. It handshakes with one or two inputselect blocks and with one outputselect block at the time. OR-ports on the request in and acknowledge out signal make sure only one latch controller is required for multiple operations.

When the latch controller receives a request from an inputselect block, the latch controller indicates to the inputselect block that the data has propagated through the even latch by making the corresponding input acknowledge high. When the second request is received, it makes the other input acknowledge high, and since this request arrived later, the operation is also assumed to be completed. Next, an output request is send and the odd latch is made transparent. When this request is propagated through the delay, it is immediately acknowledged by the outputselect block, indicating that the data has propagated through the odd latch. The odd latch is now made opaque, the output request is made low and the input acknowledge signals are allowed to go low when the requests have gone low to indicate that the data is not required anymore. Now, a new input request can arrive or the output acknowledge can go low. When new input requests arrives, it is acknowledged immediately and when the output acknowledge is low, the data is latched again by the odd latch and an output request is send.

3.3.5.2 Outputselect

The outputselect block is much simpler in the concurrent design, since it doesn't have to control the latches anymore. It acknowledges the request signal immediately in order to lower the odd latch control and it handshakes with the inputselect block. A fork request block can still be used if the data is used by multiple succeeding operational units.

When both a request and start signal have arrived, an acknowledge is immediately send, indicating that the data has propagated through the odd latch. Also, a request out is send, indicating that the output data is valid. At some point, the input request will go low, but the input acknowledge will be held high until the output acknowledge is received. When the output acknowledge is received, indicating that the data is latched and can be removed, the input request is made low and the finish signal is made high. The output handshake doesn't have to be completed before the finish signal goes high because the handshake signals between operational units are connected directly and can thus only be active once per sample; i.e. if the next outputselect block handshakes with the same succeeding operational unit, it has a separate handshake signal and can not respond to the previous acknowledge, which was possible with the implementation compatible with the original VHDL code.

3.3.5.3 Inputselect

The inputselect block controls the even latch in the more concurrent design. Together with the outputselect block of the succeeding operation, a fall-decoupled latch controller is implemented.

When an input request arrives, indicating that data is ready, the latch is made transparent. A request out is send through the delay element in order to indicate that the data is propagating through the latch and operation. When the output acknowledge is high, indicating that the data has propagated through the latch, the latch is made opaque, the output request is made low and the input acknowledge is made high. When the output acknowledge is low, the finish signal is made high immediately, without waiting for the input request to go low. This is now possible since the handshake signals between operational units are connected directly, just like with the outputselect block.

3.3.5.4 Fake Acknowledge

Like with the original controllers, the fake acknowledge block is used as a replacement for the outputselect block when data is only used for a global output and does not need a handshake.

The fake acknowledge block will send an input acknowledge when both the start and request in are high. When the request in is low again, it will make the finish signal high.

3.3.5.5 Fake Request

Like with the original controllers, the fake request block is used as a replacement for the inputselect block when the data is a global input or a constant, and does not need a handshake since the data is always available.

The fake request block is the same as in the original controllers.

3.3.5.6 Reset

Since there is a delay element between each handshake block which is reset correctly, the only signals that need to be initialized is the handshake between the outputselect and inputselect blocks. As a result of optimization 1, no extra reset logic is needed to initialize the input- and outputselect blocks correctly, even when a fork request block is used.

3.3.6 Performance

When the LWDF filter is build with the optimized handshake blocks, the total latency of the circuit is 60 ns when the ALU delay is 3.8 ns and the MUL delay is 5 ns, compared to 90 ns with the original design. The reduced performance compared

to the synchronous counterpart, which finishes in 40 ns, can be attributed to the controller overhead. A more detailed analysis is presented in Chapter 5.

3.4 Datapath

The datapath is to a large extend the same as with the synchronous design. The difference is in the way the data is stored. In the synchronous case, every operation has a edge-triggered flip-flop at its output which always contains valid data, thus the input mux doesn't have to store data. In the asynchronous case, the flip-flop is replaced with two latches, one at the input and one at the output of an operation, the even and odd latches. Consequently, the possibility to combine the even latch with the mux, which is also located at the input, is created.

In this section, the different implementations for the input mux and latch are explained, and the timing constrains for the data path are explained.

3.4.1 Input Multiplexer

For every operational unit, there need to be two MUXes to select the right operands. A MUX based on three-state buffers and a MUX implemented in logic are compared. The MUX implemented in logic needs a latch at its output to hold the data for the operation, but the three-state buffers only require a bus-hold element. The test design was a 16-bit 4-input MUX.

For Design Compiler, the designs are not functionally equivalent. Thus Design Compiler cannot be used to automatically select the best circuit.

To constrain the timing of a latch in Design Compiler, it needs to have a clock signal. Thus, the latch control signals eCtrl and oCtrl have to be defined as a clock, and all other timing constraints have to be adapted to match this clock. In the case of a three-state buffer, the timing constraints are simple input-to-output delays.

3.4.1.1 Logic MUX

For the MUX implemented in logic, a behavioral description was given in VHDL. If-statements decide whether new data should be loaded to the MUX output. The absence of an output assignment in an else statement will result in a latch. When compiled in Design Compiler, it will automatically select logic to implement the MUX and latch elements to hold the data.

3.4.1.2 Three-state buffers

For the three-state buffer implementation, a structural description was given in VHDL. A three-state buffer was described in behavioral VHDL and used for every input. The bus hold elements from the gate library where mapped directly to every output bit.

Table 3.1: 16-bit 4-input MUX compared

	Area (nm^2)	Delay (ps)
Logic	1989	320
	1087	350
	528	640
Three-state	4544	100
	1728	140
	640	200

Timing requirements are straight-forward for this implementation, since it is a simple input-output delay. The three-state buffers don't break the timing path. The bus holding elements only add extra load and don't complicate the timing analysis. Also, the latch control signal is considered an input-to-output delay that will be optimized by Design Compiler if the constraints are set correctly.

3.4.1.3 Test results

For both implementations, the fastest possible solution was found using a very strict timing constraint (10 ps) and the smallest was found without a timing constraint. The same compile command was used every time. The results can be found in Table 3.1. The three-state design is always faster than the logic design. Although the smallest logic design is smaller than the smallest three-state design, the delay is increased significantly. Thus, three-state registers were chosen over a logic mux.

3.4.2 Timing constraints

An operational unit receives a request when data is ready to be processed. Then, it makes the eCtrl signal high. Thus, when the eCtrl signal goes high, data is always availible at the input. Note that the data still has to propagate through the MUX. After a predefined delay, the data should be ready at the input of the odd latch; it should have propagated through the MUX, latch and the operation. Then, eCtrl can go low and oCtrl goes high and stays high for another predefined delay. The data has to propagate through the latch in this time slot, before the oCtrl signal goes low again.

The next oCtrl signal can be delayed by the output block if the old data is still in use, but that doesn't change the timing constraints since it only increases the time available for the operation in some cases. Thus, the datapath can be constrained by the two predefined delays. During the first predefined delay, the oCtrl signal is low and the eCtrl signal is high. The data has to travel through the MUX and the operation during this period. Then, the oCtrl signal goes high and the data has to travel through the latch. When the oCtrl signal goes low again,

the data should be ready at the output of the odd latch. The data does not have to be valid before the oCtrl signal goes high, since it can borrow some time from the latch if the oCtrl signal is longer high than the time required for the data to propagate through the latch.

This timing constraints are specified in Design Compiler using the script found in Appendix C.2.

Note that oCtrl is specified as a clock since it controls a latch. The input and eCtrl are specified relative to the falling edge of the oCtrl signal. The latch is recognised by Design Compiler and it will automatically take care of time borrowing.

3.4.3 Performance of optimized handshake blocks

When the LWDF filter is implemented with the optimized handshake blocks, the total latency of the circuit is reduced to 60 ns when the ALU latency is 3.8 ns and the MUL latency is 5 ns, as opposed to 90 ns with the circuit based on the integrated controller by Simmonds. Also, the improved version is based on unmodified scheduling results.

3.5 Conclusion

In this chapter, an implementation for the given asynchronous LWDF filter is proposed. Performance and compatibility issues in this solution are identified and an optimized implementation is designed to tackle these issues. Handshake blocks able to implement any data flow graph as asynchronous scheduled circuit are designed. In the next chapter, these blocks are used as a basis for a complete design flow from circuit specification to netlist.

4 Design flow

In this chapter, the complete design flow for the asynchronous scheduled circuit from a data flow graph to a netlist of UMC90 library gates is explained. The circuit is controlled by the handshake components described in Chapter 3.

The design flow consists of a number of different steps. First, STG's are synthesized using Petrify and Design Compiler. The netlist is generated from circuit specification using the Scheduling Toolbox and custom Matlab code. Design Compiler is used again to generate delay elements and compile the datapath to the target library. Finally, placement and routing is performed using SoC Encounter. An overview of all steps involved can be examined in Figure 4.1.

The concurrent controller implementation from Section 3.3 is used to be compatible with the original scheduling results.

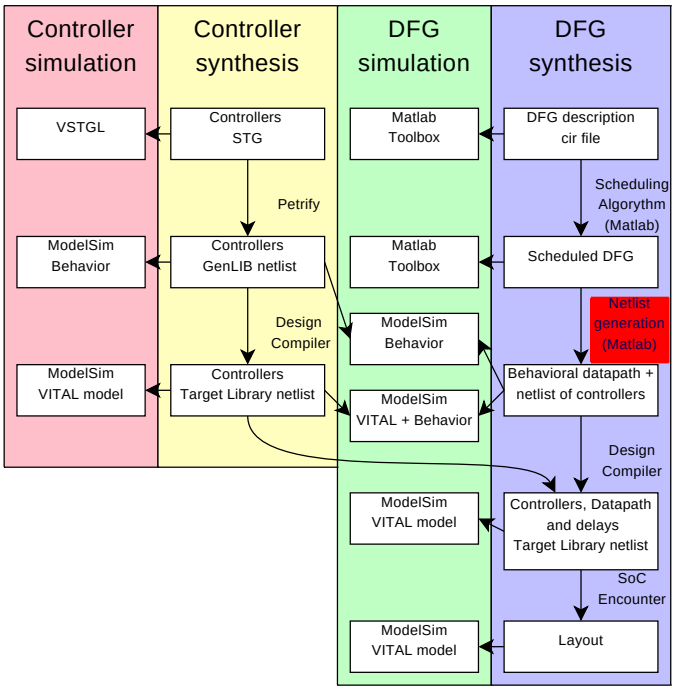


Figure 4.1: Overview of the design flow. The custom Matlab code is highlighted in red

4.1 Synthesis of controllers

To synthesize the STG to the target library, Petrify and the technology mapping method described in Section 2.5.2.6 is used since the target library is not available in the genLib format required by Petrify. Technology mapping is favoured over Generalized-C or Complex Gate implementations because both Generalized-C and Complex Gate implementation would require the user to create those gates in the target technology, which is a substantial amount of work.

4.1.1 STG Synthesis

Petrify is used to synthesize the STG's of the controllers. First, the Complete State Coding (CSC) is solved and the results are saved to a file. Timing assumptions are not used since the controllers can be used in different configurations. For example, the request out from an inputselect block is followed by an acknowledge out, which is delayed by a substantial delay in case of an ALU and MUL, but in case of a register this delay is minimal. Using timing assumptions while solving the Complete State Coding would require a large variety of controllers in which different assumptions about the environment are made.

Next, the Speed-Independent Technology Mapping is run with the default genLib file from Digital Gate Compiler. This file is used because all of the gates are also available in the Target Library while still providing a significant amount of complex gates. The resulting netlist is saved in a verilog file.

After the netlist is generated, it can be simulated in ModelSim to verify its correctness and it can be determined which input signals need to be reset, as described in Chapter 3.2.4. The input signals that need to be reset are manually modified in the verilog code, but when using the concurrent controllers and default genLib file from DGC, no input signals need to be reset. When a reset of input signals is needed, a behavioural description for the reset circuitry can be given, since the reset of the internal signals is also behavioral and needs to be synthesized later.

4.1.2 Library conversion

When the verilog netlist is read in together with an HDL file with the description of all gates in the default genLib file, Design Compiler can map each genLib component to the target library component, as described in Chapter 2.5.2.6, using the *compile* command.

The *compile* command will also map the behavioral reset logic. It has to be verified that the reset logic does not introduce hazards; the reset logic should either be mapped to an AND-port, when the signal has to be reset to a low state, or an OR-port with one inverted input, when the signal has to be reset to a high state.

Also, as stated in Chapter 2.5.2.6, the number of gates should remain the same (except for the reset circuit). To make sure this is the case, no timing constraints

are given.

After the controllers are implemented in the target library, they are simulated in ModelSim with back-annotated timing information. The wire load will change when the controllers are mapped to a larger design, but the circuits are Speed-Independent and the functionality will not depend on the wire load.

4.2 Netlist generation

Although the controllers form the basis of the asynchronous design, they serve no purpose themselves. To make a functional circuit, they need to be connected in such a way that they control the datapath as specified in the Data Flow Graph. Thus, the generation of the datapath and netlist of controllers are a vital part of the design flow, which is done by custom Matlab code based on scheduling results from the Scheduling Toolbox.

First, the operations are scheduled and bounded to operational units using one of the available scheduling algorithms. Then, VHDL code for the asynchronous circuit is generated.

For every operational unit, the controllers and datapath for that operational unit are contained in a single entity. Every register also contains two latches and the same controllers as an operational unit, and is also contained in a single entity. The entity of an operational unit or register contains the handshake blocks, OR-ports for the handshake signals, latches and the operation itself.

The result of existing scheduling algorithms are used for the generation of an asynchronous circuit using previously explained controllers and topology.

In this section, the results from the scheduling are explained. Then, the generation of the operational units is explained after which the collection of operational units are interconnected and implement the State Sequencing Graph (SSG). Then, the top entity is created, which connects the environment with the SSG and makes sure the right variables are fed back. At last, a Testbench is created.

4.2.1 Scheduling results

The existing scheduling algorythms are run to generate the results. The steps are:

- Parse function arguments
- Read circuit data
- Run selected scheduling algorythm
- Determine needed registers

When the asynchronous netlist generation is executed, the desired scheduling algorithm has to be specified as well as the parameters required by the scheduling algorithm and the circuit file. The circuit will then be read and the specified

scheduling algorithm will be called from the netlist generation tool. The result of the scheduling are:

- Global inputs and outputs
- Global feedbacks.
- Local variables
- Input- and output variables for each operation at each cycle
- Lifetimes for local variables that require extra registers
- Required number of logic shifts for each operation input.

For the asynchronous circuit, every global input and feedback input is a static inputs, i.e. they don't require handshaking since they are valid as long as start is high. Local variables require handshaking, since they are only valid when the operation calculating that variable is completed.

All outputs, global and feedback, are a subset of the local variables. Since they should be valid when all operations are finished, they should be the last output of an operation or register. Their handshaking is implicitly satisfied as a result of the done-signals, and they do thus not require output handshaking.

When a certain local variable should be stored in a register, the lifetime of the variable is saved by the scheduling algorithm. If the lifetimes of two variable do not overlap, they can be mapped to the same register.

4.2.2 Operational Units

For each operation and register, the sequence of input- and output handshakes need to be known in order to map the right handshake components. When the output of an operation needs to be saved in a register, the operation also needs to handshake with the register. The registers need to handshake with the consuming operations that can't get their data from the producing operation because it would slow down the circuit or cause deadlocks. Each fork in the handshake signal introduces an extra delay, so only the consuming operations that are in the critical path should get the data directly from the producing operation, the other consuming operations should get their data from the register. The consuming operation is assumed to be in the critical path when it was originally scheduled in the clock cycle directly after the producing operation. The *inreg* function will determine if a variable should be read from the register or from the producing operation.

4.2.2.1 Input handshakes

To map the inputselect blocks, the succeeding operations, registers and static inputs have to be identified. For a MUL or ALU, there are three possibilities: A

static input, another operational unit or a register. For a register, the input is always an operational unit since a certain variable is stored in maximal one register and static inputs are already valid during the entire operation.

Operational Units For each static input, no input handshake is needed and a fakereq block needs to be mapped instead of an inputselect block. A MUX-input will be created to select the data, which will be controlled by the fakereq block. For a handshake input, the producing operation needs to be identified to be able to generate the right handshake signal names. Also, if the variable is available in a register, the *inreg* function is used to check if the input has to come from the register or the producing operation, based on the lifetime and critical path requirements explained above. When the right signal names are determined, an inputselect block is mapped and a MUX input is created.

If the input needs to be shifted, the data is shifted internally in the component at the input of the new MUX input.

Registers For each variable stored in the register, the operational unit producing the variable should be identified, an inputselect block should be mapped and a new MUX input should be created.

4.2.2.2 Output handshakes

To map the outputselect blocks, the consuming operations, registers and global outputs have to be identified. Each operational unit can handshake with other operational units and registers, and it can contain a global output. Registers can't handshake with other registers, since every local variable is assigned to at most one register.

Operational Units For each output variable, the succeeding operational unit using the output variable should be identified. If the output variable is stored in a register, the operational unit should handshake with the register and the *inreg* function should be executed for each consuming operational unit. If operational units exists that do not use the register according to inreg function, the output handshake should be forked to the register and these operational units. If no register is used to store the variable, but multiple outputs exist, the output handshake should be forked to all succeeding operational units. If there are no succeeding operational units or registers, the output of the operational unit is used as global output. In that case, a fakeack block is mapped instead of an outputselect block.

Registers As with the operational units, for each variable stored in the register, the succeeding operational unit using this output variable should be identified. For each operational unit, the *inreg* function is executed to check if that operational

unit needs to handshake with the register. For each variable, at least one operational unit should require a register, or the variable stored in the register should be used as a global output, because the scheduling algorithm doesn't create a register when it is not used. If multiple operational units require the same variable from the register, the handshake should be forked to all consuming operational units.

Fork blocks When an output handshake needs to be forked to multiple destinations, a (complete binary) tree is created with forkreq blocks. Each pair of handshake signals (req and ack) is represented by a node and each forkreq block is represented by two edges. The tree is created in such a way that all required output handshakes are a leaf of the tree. Using this tree, any number of output handshakes can be generated.

4.2.3 SSG Entity

When all operational units are created, they need to be connected to each other and to the outside world. This is done in the entity called SSG (State Sequencing Graph). The original Scheduling Toolbox also creates an SSG entity with the same functionality. The signals that need to be connected include

- Handshake signals
- Data signals
- Global inputs
- Global outputs
- Start signal
- Done signal

Since the names of the handshake signals are consistent, the handshake signals can directly be connected to the local signals with the same name. For every output handshake, a local signal is created. Every input handshake corresponds with another output handshake, so in this way every handshake signal is locally created.

Each operational unit and register has a single data output port. The name of this signal is the name of the operational unit with the postfix 'out'. This signal convention is also consistently used as input signal name and can thus directly be connected to the local signal of the same name.

The global inputs are connected to the corresponding operational units input, and the global outputs are connected to the right operational units outputs. The global inputs can directly be connected to the consuming operational units since the names are consistent, but each global output signal has to be mapped to the corresponding data out signal.

The start signal is directly connected to all operational units and registers, but the done signals has to be merged to a single done signal. This is done by a tree of 3- and 2-port Muller-c elements to create an arbitrary-width Muller-C element. In this tree, each input is represented by a leaf, each intermediate signal is represented by an internal node and each Muller-C element represents two or three edges. Using this arbitrary-width Muller-C element, the global done signal goes high when all internal done signals are high, and the global done signal goes low when all internal done signals go low. Since the start signal can only go low if the global done signal is high, and it can only go high again when the global done signal is low, no operational units receive a high start signal before the low start signal is seen by all in- and outputselect elements.

To create the VHDL entity, the algorithm first loops through all operational units and registers to define them as components. Since the local signals are identical to the signals in the operational unit entities, they are saved when generating the operational unit entities and can now easily be looped.

After all operational units and signals are defined, the algorithm loops through all operational units again to instantiate each one and map their ports to the signals of the same name. This loop is almost the same as the one before, but it is separated because port mapping is only valid after the VHDL 'begin' statement.

The done signals of all operational units and registers are saved together in a single array during previous loop whereafter the tree is generated. As long as there is more than one signal in this array, a Muller-c element is mapped to the top three input signals of that array (or two when there are only two available), and the output signal is added to the bottom of the array. This step is repeated until there is only one output signal left, which is connected to the global done output.

The original Scheduling Toolbox can specify certain inputs to be fed through to the output, in order to delay them with one sample period. In some cases, one input can be delayed multiple sample periods, like with a FIR-filter. Since the asynchronous design uses level-sensitive latches controlled by the done signal to save the SSG outputs, the feed-through signals would ripple through all latches at once. To overcome this problem, an extra latch is added to each feed-through signal inside the SSG entity which is transparent when the done signal is low.

4.2.4 Top entity

Since the SSG entity has the same functionality as the synchronous one, the top entity which contains the feedback registers can be almost the same as the synchronous one. The only change is the replacement of the feedback edge-triggered registers for latches. This latch is transparent when the done signal is high since a high value of the done signal ensures the validity of the output data.

4.2.5 Test-bench

The synchronous design requires a pulse on the start signal and then waits for a pulse on the done signal. In the asynchronous design, the start- and done signals are used in a 4-phase handshake fashion. The start signal represents the request and the done signal represents the acknowledge signal. Thus, the test-bench has to wait for the done signal to become high before it can lower the start signal and vice versa; the handshaking of the synchronous test-bench is modified to a 4-phase protocol. Since the asynchronous design doesn't have a clock signal, the clock signal can be completely removed from the test bench.

4.3 Datapath synthesis

The VHDL code generated by Matlab still contains behavioural code for the delay elements and the datapath. Using Design Compiler, we can compile the datapath into library gates, taking into account the timing requirements and operating conditions, and generate corresponding delay elements. The latency of the operations are constrained with a predefined delay, and a delay element is matched to the same predefined delay minus the control overhead hideable by the delay. Which control overhead can be hidden by the delay will be explained later, in Section 5.3.1. Different operating conditions can be chosen for the delay elements and the datapath.

4.3.1 Operating Constrains script

For every type of operation, the timing constraints are set using the script from Section 3.4.2.

For every type of operational unit, there are two types of delay elements: One for the operation and one for the output latch. The delay for the output latch is equal for all types of operations, so in total there are four kinds of delays (Multiplier, ALU, Register and output latch). For every type of delay, the delay script described in Section 2.5.2.6 is run with the specified delays. Then, the right operating conditions are selected and the delay is compiled.

The behavioural description of the Muller-C elements are compiled as described in Section 2.5.2.6.

All asynchronous controllers and Muller-C elements are set to don't touch, in order to preserve their hazard-free designs.

The complete operating constraints script can be found in Appendix C.2.

4.4 Placement and Routing

The placement and routing of the asynchronous circuit differs from the synchronous counterpart in three ways; the clock tree generation is not required for the asyn-

chronous case, the isochronic fork assumption should be satisfied and the delay elements should be matched to the actual operation delay.

Also, the timing constraints from Design Compiler can't be applied to SoC Encounter since it doesn't accept the complex requirements used for the asynchronous circuit generation.

The clock tree generation is easily removed from the synchronous synthesis script, but the isochronic forks can't be satisfied directly. For the isochronic forks to be satisfied, all asynchronous controllers have to be layed-out manually.

Delay elements can be matched to the latency of each individual operation more closely during layout. If manual delay matching is performed, the performance can be improved.

The absence of timing constraints can result in increased latency for the operations. This should be compensated for by longer delay elements, resulting in reduced performance.

Although the placement and routing scripts used in this thesis do not satisfy the above requirements, in some cases the design is functional in simulation after placed and routed without manual layout or delay matching. These designs are used later on for power simulations.

4.5 Conclusion

Using the optimized handshake blocks, Petrify, Design Compiler and custom Matlab code, any data flow graph can be implemented as asynchronous scheduled circuit, although last design step, from netlist to layout, still needs improvement. Using this design flow a large number of circuits were successfully generated. In the next chapter, the performance of the LWDF filter and other automatically generated circuits is analyzed.

5 Performance Evaluation

In this chapter, the results of the manually and automatically generated circuits are considered. First, the potential performance benefits from asynchronous operation are investigated in more detail, after which the actual performance in latency and power consumption are discussed.

As explained in Chapter 2, the asynchronous LWDF filter could finish its task in 80% of the time required by the synchronous counterpart if the delays of the operations are added. In Chapter 3, an implementation with technology-mapped speed independent controllers is proposed. Although this implementation is functional, the performance expected in Chapter 2 is not attained. In this chapter, the causes of the performance loss are explained and calculated.

5.1 Fine-grained scheduling

For all analysis in this chapter, a perfect delay element is assumed, and the delay of operations is assumed to be equal to that in the synchronous counterpart. This means that the perfect delay element has to be equal to the worst-case delay of the operation, as in the synchronous case. This makes the comparison of the performance only related to the expected delay of the operations under worst-case conditions, which is the same in the synchronous case as in the asynchronous case since the same operations are used. Note that the simulations are run with typical-case delays, the delay elements are designed for the typical case and the controller overhead is also specified in the typical case delay. Using method, performance improvements as a result of the correlation between the delay line and actual operation delay are ignored.

Asynchronous circuits can be modelled by an extreme high clock speed and fine-grained resource constraint list scheduling. The list scheduling will schedule the highest prioritized event when a resource is available. Thus if data for a certain type of operation is available and a resource is available, the operation will immediately be scheduled to start, even if the operation is not in the critical path. This is the same behaviour as an asynchronous scheduled circuit.

The list scheduling algorithm will assign integer delays to operations, representing clock cycles in synchronous operations [24]. If the clock period would be much smaller than the operation time, the difference in critical path between operations can be exploited, just like with a bundled-data asynchronous circuit. In practice, the clock period is limited by a number of factors such as clock skew and power constraints.

The fine-grained clock scheduling can be used to schedule asynchronous circuits without adopting the scheduling algorithms, because the fine grained clock will approximate the asynchronous behaviour. Also, fine-grained clock scheduling can be used to compare the ideal asynchronous case with the synchronous case.

For the comparison, different tools, technologies and circuits were used.

5.1.1 Scheduling Tools

The two scheduling tools used for this fine-grained scheduling test are the Scheduling Toolbox for Matlab and GAUT.

Scheduling Toolbox for Matlab The Scheduling Toolbox by H.J. Lincklaen Arrins is a high-level synthesis tool from the Circuits and Systems group. It can perform list scheduling under resource constraint, unconstrained ASAP and ALAP scheduling, and Force Directed scheduling. The available operations are limited to an ALU and a Multiplier. When using list scheduling, ASAP or ALAP, the user is allowed to specify the number of clock cycles for an ALU and a MUL. This allows the user to define the ratio of delays between different operations, while the exact values for the delays are not fixed.

During the test, it turned out that the Scheduling Toolbox did not achieve the expected results when a fine grained clock is used in combination with the list scheduling. When operations took more than 3 cycles, the scheduling results were worse than the results from GAUT. Sometimes, even when compared to the scheduling results with only one cycle per operation, the performance was worse. For example, if the ALU is specified to complete in 10 clock cycles, and the MUL is specified to complete in 13 clock cycles, the total numer of clock cycles was more than 13 times the number of clock cycles with respect to the scheduling results when both ALU and MUL take 1 clock cycle. A factor between 10 and 13 would be expected, because if all ALU operations are just delayed by 3 clock cycles, the original scheduling is still valid, but an improvement is expected as a result of the increased flexibility for the scheduling of the ALU.

The cause of the shortcomings was related to a small bug in the Scheduling Toolbox which was found and fixed quickly by the author. The new version produced exactly the same results as GAUT.

GAUT The GAUT tool by the University of South Brittany is a high-level synthesis tool that uses C and C++ as specification language. It can perform list scheduling under timing constraint, and allows the user to specify the operation delay and clock speed. A library with operations can be specified by the user or a library can be generated for a number of FPGA's. The tool will automatically decide the required number of clock cycles for the operations based on the delay of the operations and on the clock period. The timing accuracy is 1 ns, but a scale factor can be introduced to allow more accurate scheduling. GAUT version 2.4.3 is used for these experiments.

Table 5.1: Latencies used in fine-grained scheduling

	Custom	Xilinx
ALU delay	3.8 ns	7 ns
MUL delay	5 ns	18 ns

In most cases, GAUT adds one or more clock cycles at the beginning and end of the dataflow graph to allow input and output communication to take place over a single bus. Since both the asynchronous implementation and the synchronous Scheduling Toolbox implementation use direct inputs and outputs, these clock cycles are not required for the correct operation of these circuit. Thus, those clock cycles were subtracted from the total amount of clock cycles.

5.1.2 Technologies

Two different technoligies were used as model for the fine-grained scheduling. One is based on previous experience with the LWDF filter in UMC90 technology and the other one is based on Xilinx FPGA's.

Custom In the experiments with the LWDF filter, the MUL delay was set to 5 ns and the ALU delay to 3.8 ns. For the different asynchronous LWDF filter implementations, the timing constraints were set more tight until Design Compiler was unable to satisfy the timing constraints with 18-bit operations using a single compile command. The 5 and 3.8 ns constraint for the MUL and resp. ALU could always be synthesized and are used for these experiments. Since only a MUL and an ALU is availible, this technology cannot be used for all examples. In this model, both operation delays are a multiple of 0.2ns, so a clock period of 0.2ns should result in the best possible scheduling for asynchronous operation with this model.

Xilinx FPGA In GAUT, a default library is included based on a Xilinx VirtexE xcv1000e-6 FPGA [7]. In this library, a MUL has a delay of 18 ns and most ALU operations have a delay of 7 ns. A divider is missing, so not all examples from GAUT can be used. All operation delays in the library are modelled as a multiple of 1ns. Thus, a clock period of 1ns should result in the best possible scheduling for asynchronous operation with this model.

5.1.3 Results

Five different example circuits were scheduled using the available scheduling tools. In this section, the results for two of the circuits are explained.

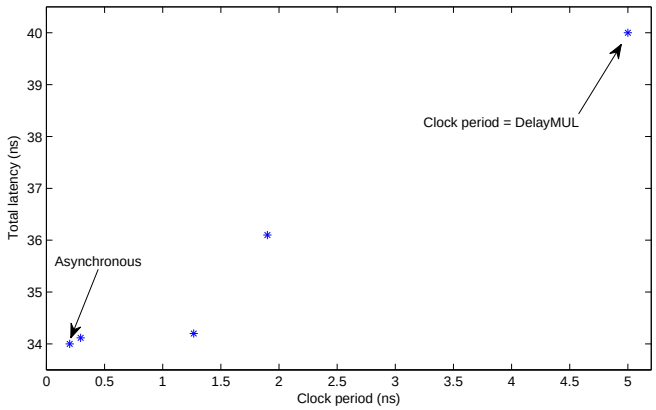


Figure 5.1: LWDF filter scheduled with custom technology

LWDF The LWDF filter used in Chapter 3 was tested to see the possible performance improvements of an ideal asynchronous implementation. A performance improvement of almost 20% is expected in Chapter 2. The LWDF filer was supplied in the scheduling toolbox format, but it was converted to simple C statements using regular expressions. The results of Gaut and the fixed scheduling toolbox were equal and can be found in Figure 5.1. Only the results which have the lowest latency for a certain minimal clock period are shown; a result with a higher latency but lower clock period is not shown because those results are not beneficial in any way.

A performance improvement for the asynchronous circuit of 19% can be achieved compared to a clock period equal to the ALU delay (not shown in Figure 5.1), and an improvement of 15% can be achieved compared to a clock period equal to the MUL delay.

CORDIC One of the examples in Gaut is the CORDIC core. It doesn't use any multipliers, but it does use six other kind operations with four different latencies. Since it uses operations that are not available in the custom library, only the GAUT library was used. An area constraint of 2 instances for every operation was chosen. Because Gaut can only schedule under timing constraints, the timing constraints where relaxed until the area constraint was met.

The latency for the CORDIC core against clock period can be found in Figure 5.2. Again, only the beneficial points are shown. Compared to a clock period equal to the comparator delay, a performance improvement of 11% can be achieved with an ideal asynchronous implementation.

Results of fine-grained scheduling Besides the LWDF filter and CORDIC core, three other circuits were tested using fine-grained scheduling. A simple FIR-fiter was scheduled in GAUT using both the Xilinx as the custom technology

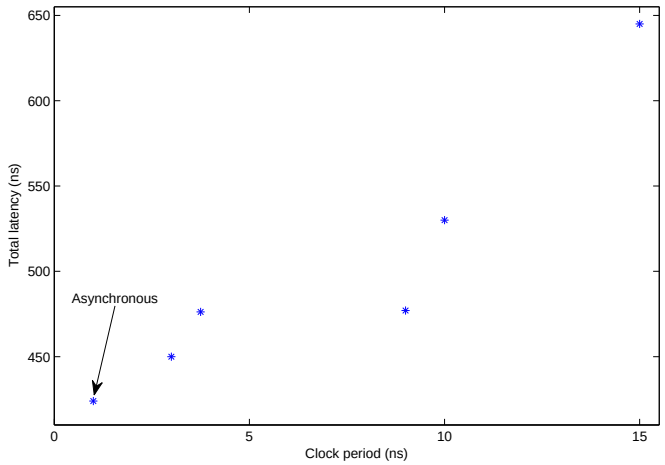


Figure 5.2: CORDIC core scheduled with Xilinx technology

Table 5.2: Asynchronous latency in percentage of synchronous implementation with specified clock periods

Clock period	0.5 MUL	ALU	MUL
LWDF Custom	85%	81%	85%
CORDIC Xilinx	n.a.	89%	n.a.
FIR16 Xilinx	83%	95%	44%
FIR16 Custom	78%	96%	78%
IMDCT Custom	91%	80%	91%
FFT Custom	84%	67%	84%

in Gaut, an 18-point IMDCT core for an MP3 player from [21] was scheduled using the custom technology and the Scheduling Toolbox, and a 32-point FFT was also scheduled using the custom technology and the Scheduling Toolbox. Using the fine-grained scheduling, the potential performance benefits of asynchronous bundled-data scheduled circuits are demonstrated. In Table 5.2, the latency of ideal asynchronous bundled-data circuits are shown in percentage of their synchronous counterparts with a clock period of 0.5 MUL delay, 1 ALU delay and 1 MUL delay.

5.2 Simulations

In Chapter 3, two different versions of the LWDF filter are created. In Chapter 4, a complete design flow for automatically generating asynchronous circuits is proposed. The resulting circuits are simulated and the simulation results are presented in this section.

All simulations described in this chapter were successful, the output data corresponds to the expected data.

5.2.1 Original LWDF Filter

In the original VHDL code, the controllers are specified in behavioural VHDL; all controllers are behavioural without delay specifications, thus no control overhead is incorporated. When simulating with the custom technology from previous chapter, the original code has a latency of 60 ns. The corresponding scheduling results have 8 cycles, which would correspond to a latency of 40 ns in the synchronous case. The lower performance, even without control overhead, is a result of the reduced concurrency.

When the circuit was synthesized using handshake blocks and the custom technology, the latency was increased to 90 ns. This reduced performance is a result of overhead in the controllers.

5.2.2 Optimized LWDF Filter

The optimized handshake blocks do not contain the holddata block, required for the original VHDL code (see Section 3.1.2.1 for details). As a result, the LWDF filter had to be scheduled using the Scheduling Toolbox. Again, the corresponding scheduling results have 8 cycles. According to the fine-grained scheduling, the asynchronous LWDF filter should have a latency of 34 ns.

The LWDF filter synthesized with optimized handshake blocks and the custom technology has a latency of 60 ns. When the Optimized LWDF filter is simulated without control overhead, by using a behavioural description without delay for all library gates except the delay elements, the latency of the LWDF filter corresponds with the fine-grained scheduling results. The reduced performance in the real circuit is a result of overhead in the controllers and delays. The exact sources of the overhead are explained in Section 5.3.

5.2.3 IMDCT core

With the optimized handshake blocks, it is possible to implement any circuit available in Scheduling Toolbox format as asynchronous circuit. An IMDCT core was synthesized using the design flow described in Chapter 4. The same custom technology is used as with the LWDF filter. The latency of the asynchronous IMDCT core is 80 ns. The corresponding scheduling results have 12 cycles, which corresponds to a latency of 60 ns in the synchronous case.

5.2.4 Results

Simulations show that the performance of the Asynchronous LWDF filter is improved by 33% compared to the original LWDF filter, as a result of the increased concurrency. The simulation also shows that it is possible to implement a circuit

Table 5.3: Simulation results

Circuit	Real Asynchronous	Synchronous	Ideal asynchronous
LWDF original	90 ns	n.a.	60 ns
LWDF improved	60 ns	40 ns	34 ns
IMDCT	80 ns	60 ns	47 ns

available in the Scheduling Toolbox format as asynchronous circuit using the Design Flow proposed in Chapter 4 and the optimized handshake blocks proposed in Chapter 3.

In Table 5.3, the latencies of the different circuits are shown. The improved handshake blocks show a significant increase in performance compared to the original asynchronous LWDF filter.

5.3 Circuit overhead

Although the fine-grained scheduling indicates a large performance improvement, the assumption that a new operation start when resource or data become available is optimistic. In a real circuit, there are different sources of overhead, which is why the simulation results differ so much from the fine-grained scheduling results.

Overhead can be categorized in three groups:

- Controller overhead
- Inconsistent delay
- Done-signal generation

In this section, the categories are explained and the sources of overhead are identified. Then the figures can be extracted from simulation and can be used to determine the overhead in the individual operations and eventually to determine the critical path. This path can then be compared to the simulation results to verify the correctness of the analysis.

5.3.1 Controllers

If an input to a controller block changes, the corresponding output transitions take place after the propagation time. Sometimes, this includes one or more state-signal changes. For example, when a request in arrives at an inputselect block, it takes 220 ps before the MUX control signal goes high. During this propagation time, the data is available, since the request signal is high, but it is not latched as a result of the controller overhead. Thus, the operation is delayed by 220 ps.

Some controller transitions can be hidden by decreasing the length of the delay element. For example, when the request to the latchcontrol block arrives, it it

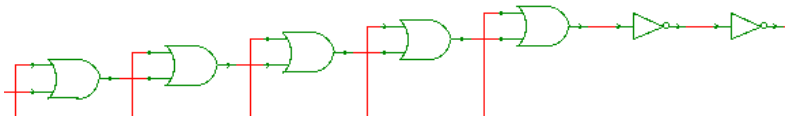


Figure 5.3: 6-input or generated by Design Compiler without timing constraints

takes about 1000 ps before the latch control signal and output request go high. However, it is possible to have the input request arrive 1000 ps earlier by trimming the delay element.

5.3.2 Delay element

For each operation, one delay element is present which is made equal to the worst case delay of the operation minus the controller overhead that can be compensated for. Each request signal is fed through an OR-port before it reached the asymmetrical delay element. Using this OR-port means that only one delay element is needed per input of the operation, instead of one for every input. Since there are only symmetrical delay elements available in the gate library, a controller is used to feed only the rising edge through the delay element, see Section 3.2.3 for details.

Both the OR-port and the asymmetrical delay add an inconsistency to the total delay. When the delay time is inconsistent, the shortest possible delay time still has to be long enough to ensure reliable operation. The other paths will then be longer than needed, and thus add overhead.

The asymmetrical delay should ideally not delay the falling edge. If the falling edge is delayed, it can slow down the circuit in some cases, in particular when the falling edge is delayed more than the rising edge of the succeeding delay element.

OR-port When a wide OR-port is specified in a behavioral way, Design Compiler maps a tree of OR-ports or other gates until enough inputs are available and the functionality of a wide OR-port is obtained. An example circuit generated by Design Compiler is depicted in Figure 5.3. In this case, it is clear that the rightmost input has a shorter propagation time than the leftmost input.

To solve this problem, timing constraint can be given to Design Compiler. A short time constraint will also make the falling edge propagate as fast as possible, which ideally should not be delayed. However, it is possible that the OR-port will not be critical-race free anymore and thus unusable. Thus, in this case, every OR-port should be checked manually.

Another solution is using a library of hazard-free OR-port implementations. These wide or-ports can be generated by Design Compiler and manually checked if they are critical-race-free, or a tree of gates implementing the wide or-port can be generated manually or using an automated process.

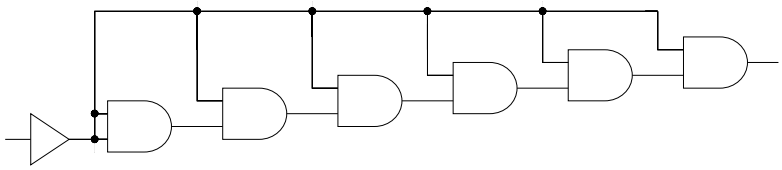


Figure 5.4: Tree of AND-ports implementing assymetrical delay[5]

Delay controller To create an asymmetrical delay, a controller is designed to generate a signal that propagates through the symmetrical delay when a rising edge arrives and a signal that flips the output of that delay using a XOR-port when a falling edge arrives. The controller is explained in detail in Section 3.2.3

The time from the rising edge of the input signal to the change of the input signal for the symmetrical delay should be consistent to be able to hide this overhead in the delay. The time from the falling edge of the input signal to the change of the input signal for the XOR-port should be as short as possible.

In the case of the technology-mapped controller, the inconsistency in the rising-edge is quite large: the first rising edge is delayed by 95 ps and the second rising edge is delayed by 175 ps. When the delay controller is implemented using two flip-flops as described in Section 3.2.3, the inconsistency is minimal since the delay from the clock signal to the change of the output for the flip-flops are quite consistent for low and high data inputs.

Another solution would be to implement an asymmetrical delay using a chain of AND-ports. In this case, one input of each AND-port can be connected to the output of the preceding AND-port, and the other input can be connected to the global delay-input, as depicted in Figure 5.4. When a rising edge appears, one input of all AND-ports is immediately high, but the output stays low until the other input has propagated. When a falling edge appears, one input of all AND-port is low and thus all outputs are low after only one AND-delay. This resets the circuit in its initial state and the delay is thus consistent.

Using the AND-ports available in the library, the area needed for a certain delay is 2,5 times more than with the delay elements from the library. Custom AND-ports which are designed to have a low area/delay ratio would reduce the area disadvantage of the proposed solution.

5.3.3 Done-signal generation

A small part of the overhead is a result of the done-signal generation. To indicate to the environment that all operations are finished, all done signals are combined with a tree of Muller-C elements, as explained in Section 4.2.3. When the last operational unit is finished, the done signal of this operational unit is made high and has to propagate to the top of the tree. In the LWDF filter, the done-signal generation represents less than 2% of the total latency.

Table 5.4: External paths used in high-level latency model

Path	Delay element in path	Overhead
Input request to Input acknowledge	Operation	225 ps
Input request to Output request	Operation + Latch	1750 ps
Input request to next operation	Operation + Latch	2900 ps
Output acknowledge to next output request		925 ps
Output acknowledge to start of third operation		3700 ps
Fork request in to request out		350 ps
Fork acknowledge out to acknowledge in		650 ps

5.4 Latency calculations

To better understand the important sources of overhead, the latency of the LWDF filter is calculated using actual controller delays and delay inconsistencies.

To create a model for the latency of a certain circuit, a number of assumptions are made to simplify the calculations:

- If possible, control overhead is hidden by making the delay shorter
- The overhead caused by inconsistent delay is half the difference between the minimum and maximum delay.
- If controller delays are inconsistent, the average delay is used.
- When a request arrives at an inputselect block before the start signal, the finish signal of the previous operation acts as the request signal.
- Scheduling will make sure registers are not part of the critical path.

Using these simplifications, delay figures can be assigned to certain paths, which can then be chained together to from a model for the complete circuit. The paths from input to output of operational units are considered in the model; internal signals of the controllers and operational units are hidden. When multiple paths exist, the longest path is used. The figures for the delays are extracted from ModelSim simulations with back-annotated timing information from Design Compiler.

In Table 5.4, all paths used in this analysis are displayed. Using these paths, a spreadsheet is made for the LWDF filter. When all paths are calculated, the longest path is easily identified. The spreadsheet can be found in Appendix B. In the spreadsheet, all values represent time in ps after the positive edge of the start signal. All values are between 95% and 105% of the timing in the ModelSim simulation.

Using this model, the delay of operations can be varied while keeping the controller performance constant to estimation performance at different bit widths.

The longest path of the circuit at different multiplier latencies is shown in Figure 5.5. The delay of the ALU is set at 70% of the multiplier delay. It can

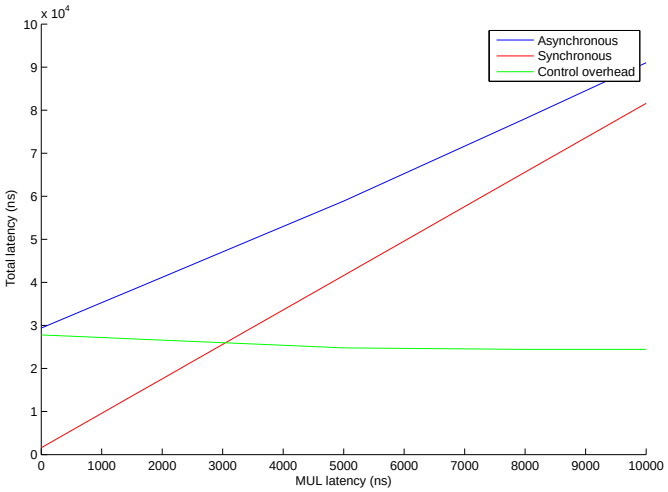


Figure 5.5: Latency of asynchronous and synchronous LWDF filter with different multiplier latencies

be observed that the absolute value of the controller overhead increases when the delay of operations decrease. This is a result of controller paths which are not delayed by the delay element that will become part of the critical path, while they would normally be shorter than a different path delayed by the delay element with the same endpoint. The slope of the synchronous circuit is equal to the number of cycles times the multiplier delay, since the multiplier delay fixes the clock period. The latency of the asynchronous circuit is the controller overhead plus the datapath latency. The datapath latency is equal to the results of the fine-grained scheduling.

From the figure, it can be concluded that the control overhead is a significant part of the critical path when reasonable values for the multiplier delay are used. To outperform a synchronous design when using multiplier latencies of 5ns, the control overhead should be reduced to 30% of its current value.

When critical races are allowed and Design Compiler is configured to optimize the logic like synchronous logic, the overhead could only be reduced to about 60% of its current value. In this technology, a 128 bit multiplier can easily be created with a delay of less than 5ns, synthesized with Design Compiler and the Faraday library.

Thus, the asynchronous LWDF filter with technology-mapped controllers has no performance benefits over the synchronous counterpart, even when the transistor sizes of the controllers are optimized.

5.5 Generalized-C implementation

In an attempt to reduce the control overhead, one of the control blocks (inputselect) was implemented as a generalized-C element in the UMC90 technology. The

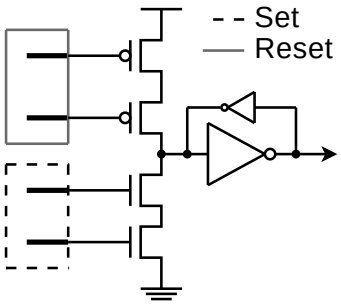


Figure 5.6: Generalized-C schematic

advantage of Generalized-C implementations is that there is more freedom for the cover, since only states where the non-input signal has to rise or fall have to be covered, while still having the freedom to cover don't cares and states where the non-input signal doesn't change[33]. This would make the logic less complex and thus faster.

A set of generalized-C elements where designed with separate set and reset inputs. This way, the inputs can be connected externally if a combined input is required. Schematics for generalized-C elements were designed for all required combinations of set- and reset inputs. The transistor sizes were made equal to the gates from the Faraday library, used in the technology mapped controllers. The schematic of a generalized-C element with 2 set and 2 reset inputs can be found in Figure 5.6.

5.5.1 Inputselect block

With the generalized-C elements, an inputselect block was constructed with the help of Petrify. For every non-input signal, Petrify indicates what the on- and off-set should be. So for every non-input signal, one generalized-C element and a number of inverters are used. The schematic of the complete inputselect block implemented in generalized-C elements can be found in Figure 5.7.

5.5.2 Comparison with Technology Mapping

The generalized-C based inputselect block is compared to the technology-mapped inputselect block. Both are based on the same STG.

Performance For the performance comparison, a complete handshake cycle is simulated on the input and output of the inputselect block.

The generalized-C based version was simulated on transistor-level using Spectre. A transition is assumed to have taken place when the voltage has crossed $^1/_2$ Vcc. The load on all output ports was set to 20fF. The input rise and fall time is 100ps. All simulations use typical operating conditions (Vcc = 1V, Temp= 27 °C)

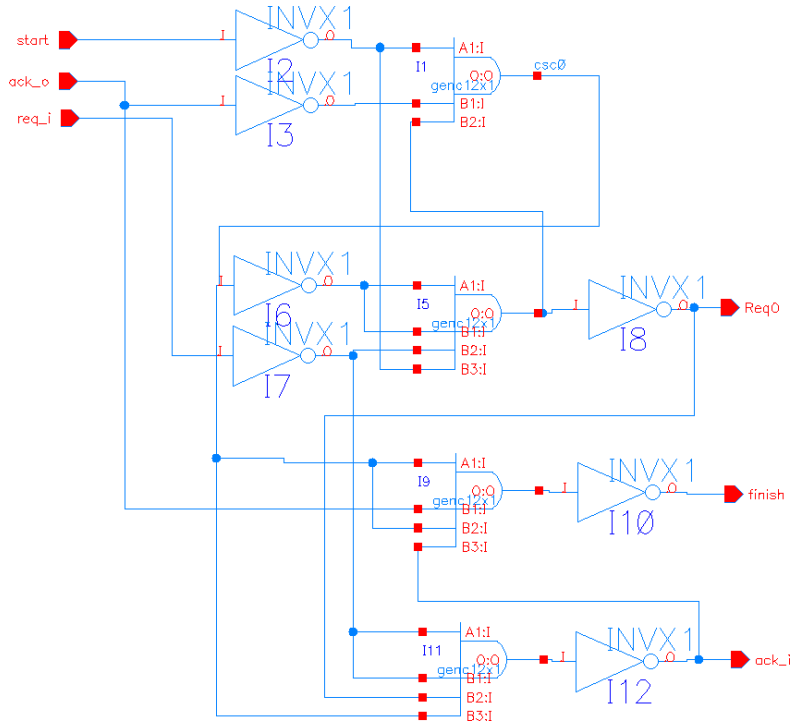


Figure 5.7: Generalized-C inputselect block

Table 5.5: Technology Mapped vs Generalized-C inputselect block latencies

Signal	Technology Mapping	Generalized-C
Req. In to Req. Out	216 ps	205 ps
Ack. Out to Req. Out low	264 ps	195 ps
Req. Out low to Ack. In	177 ps	190 ps
Req. In low to Ack. In low	129 ps	110 ps
Ack. Out low to finish	95 ps	197 ps

The technology-mapped version was simulated using ModelSim with back-annotated delays from Design Compiler using typical case delays. Here, a transition is also assumed when the voltage has crossed ½ Vcc [10].

As can be seen in Table 5.5 the difference between Technology Mapping and full-custom Generalized C is minimal.

Area The total area for the Generalized-C elements, including output inverters, is about 9 um^2. The inverters from the target library are 1 um^2. The inputselect block contains 4 generalized-C elements and 7 inverters, as shown in Figure 5.7. A total area of 43 um^2 is used by the Generalized-C implementation. The technology mapped implementation takes 44 um^2 gate area according to Design Compiler. It can be concluded that there can be no significant area advantage gained from the

Generalized-C implementation

Conclusion Transistor-level simulations showed that the performance benefits are minimal and the difference in area is also minimal, while converting all controllers to generalized-C elements would take a significant amount of work. Thus, generalized-C implementations were not made for the other blocks.

5.6 Power simulations

To compare the power consumption of the asynchronous circuit with the synchronous one, both the synchronous and the asynchronous versions of different circuits where synthesized. The complete design flow from circuit specification to layout was followed, in order to be able to complete power simulations. The circuits tested include the LWDF filter and the IMDCT core.

As explained earlier, any latency improvement cannot be expected. For this reason, the timing constraints and clock speeds were chosen such that the synchronous circuit would have the same latency as the asynchronous circuit.

The asynchronous version is generated with the Matlab code explained in chapter Chapter 4 and the synchronous version was generated with the original Scheduling Toolbox. The asynchronous version used the same scheduling results as the synchronous one and the constants were hardcoded in the VHDL files to reduce the number of IO's. The synchronous versions incorporated clock gating with the help of Design Compiler. The scripts used for placement and routing are the same, except for the clock tree synthesis and timing constraints, which where left out for the asynchronous circuit.

For the simulation, the same input was used for the synchronous circuits as for the asynchronous circuit (except for the clock signal). For the LWDF filter, a step-function was applied with 25 samples. For the IMDCT core, a ramp function with 18 steps is transfomed 3 times in a row.

Activities from the first start signal until the final output are recorded in a simulation and back-annotated to Cadence SOC Encounter for power estimation. In the simulation, the VITAL glitches are disabled because the requirements for speed-independent layout are not satisfied. A simulation with VITAL glitches enabled did not complete successfully.

As a result of the non-isochronic forks, a number of gates were oscillating, resulting in excessive power consumption. Consequently, the results of the power simulations are mixed.

5.7 Conclusion

In this chapter, the performance in terms of latency is analyzed and compared to the original asynchronous circuit and the synchronous counterpart. A number of assumptions are made regarding the delay, including worst-case delay for

the operations, to be able to do a fair comparison between the synchronous and asynchronous parts without details about the technology. It is shown that the optimized handshaking is a big improvement compared to the original asynchronous implementation of the LWDF filter, but the controller overhead is still a significant part of the total latency, resulting in performance loss compared to the synchronous design which cannot be undone by changing the transistor sizing or gate type of the controllers. The results of power simulations are mixed because the forks in the speed-independent controllers are not isochronic, which is required for correct operation.

6 Conclusion

In this thesis, optimized controller blocks which implement the controller network for asynchronous scheduled circuits are designed. Compared to previous designs, the optimized controller blocks increase the concurrency of the system, which improves performance by 30% and allows implementing any valid scheduling result. Using these controller blocks, an automated design flow from specification to layout is created. Using this design flow, a number of circuits are generated and performance analyse of these circuits is performed at different levels.

6.1 Results

Different models exist in order to guarantee hazard-freedom under a number of assumptions. Delay Insensitive circuits are the most robust variant, where each gate and each wire can have an arbitrary delay, but those type of circuits are also very limited in functionality. In Speed Independent circuits, wire forks are assumed to be isochronic, but gates can have an arbitrary delay at the output. These types of circuits can implement more functionality.

The simulation model available for the target library is a VITAL model. This model is compatible with Speed Independent and Delay Insensitive circuits in the sense that any Speed Independent and Delay Insensitive circuit can be simulated for functionality with the VITAL model. Functionally correct simulation results do not imply Speed Independent or Delay Insensitive circuits.

To synthesize asynchronous scheduled circuits with the required level of concurrency, the controllers can be specified as ASTG and synthesized to Speed-Independent circuits. Petrify and SIS are two tools able to do so, but Petrify was preferred since it requires the least amount of manual work.

When more than three operations are scheduled to a resources, none of the asynchronous synthesis tools could synthesize the controllers with reasonable time and memory constraints, even when concurrency is reduced. This is a result of state explosion, which doubles the amount of states for every signal added.

To overcome the problem of state explosion, the controllers have to be decomposed into smaller communicating controllers. DesiJ, a tool for automatic decomposing of STG's, generates smaller STG's which implement the original STG when connected together, but the results are not useable. Manual decomposition is performed to be able to synthesize the controllers. The controllers are split up into handshake blocks, responsible for one input- or output handshake, support blocks for global inputs and outputs, and blocks for the forking of handshakes. With these blocks, the behaviour of a given LWDF circuit could be mimicked.

Although the given circuit could be synthesized, the performance is even without controller overhead much worse than the synchronous counterpart, and the controllers could not be used for arbitrary scheduling results since certain requirements to the scheduling results are imposed. To overcome these problems, more concurrent versions of the controllers are designed, which utilize an extra delay elements per resource in order to be able to use two latches effectively. This more concurrent design required an extra controller block, the latch controller, which controls the second latch. Also, some modifications to the existing controller blocks are made.

With the concurrent handshake blocks, the performance of the asynchronous LWDF filter is improved by 33%. Also, any scheduled data flow graph can now be implemented asynchronous. A Design Flow is created that takes care of the synthesis of the controllers, scheduling of the data flow graph, mapping of the controllers and creating a behavioral datapath to implement the scheduled data flow graph, synthesis of the datapath and placement and routing. Simulation can be performed at any stage in the Design Flow.

Although the performance without control overhead is, as expected, better than the synchronous circuit, the controller overhead is a significant part of the total latency and this results in worse overall performance compared to the synchronous design. An elaborated analysis shows which parts of the controllers are responsible for the overhead. In order to outperform the synchronous design, the latency of the controllers has to be reduced to 30% of their current value, which is not possible with transistor sizing.

In an attempt to reduce the controller overhead, a Generalized-C implementation is created for one of the handshake blocks. The target library does not contain Generalized-C elements, so the library components for the Generalized-C gates are created. The resulting circuit is compared to the technology-mapped Speed Independent counterpart. Both latency and area are almost identical. Because implementing the controllers with Generalized-C elements takes a significant amount of time while the benefits are minimal, other handshake blocks are not implemented in Generalized-C elements.

6.2 Recommendations

- Although synchronous-like performance will not be achievable, the transistor size of the gates in the controller blocks can still be tuned to improve performance. Delay constraints in Design Compiler cannot be used, since this will compromise Speed Independence. Manual sizing is required to guarantee speed-independent operation and still improve performance.

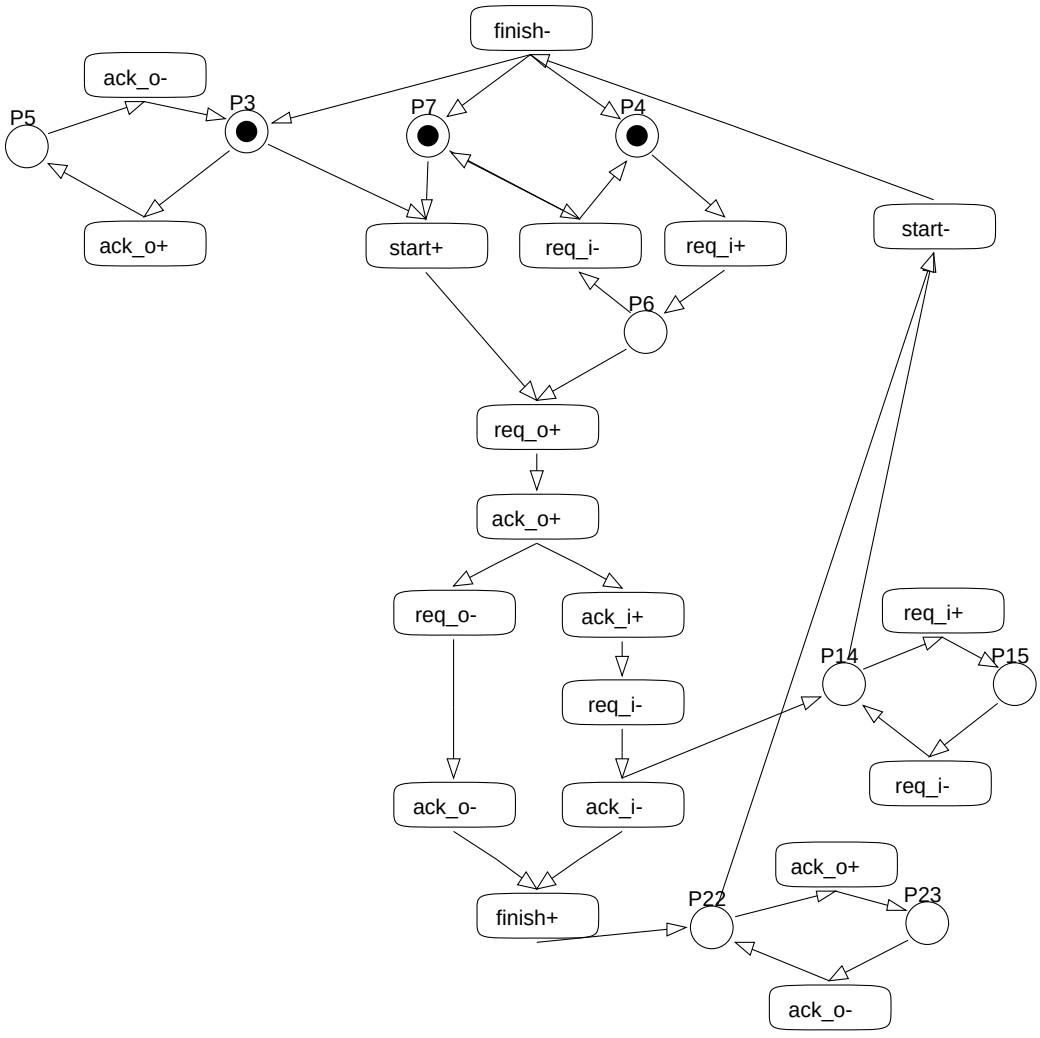
- The last step in the Design Flow, from netlist to layout, still has to be improved. Especially the isochronic fork and short inverter delay assumptions have to be satisfied. Since there are only a limited number of controllers, these controllers could be layed-out manually.

- Correlation between delay-elements and actual latency of the operation can be exploited and enhanced during layout to improve performance.
- In order to improve the design-space exploration, different completion detection schemes could be added to the design flow. Since this primarily affects the datapath, the controller architecture and netlist generation can still be used
- To improve scheduling results without sacrificing too much execution time for larger circuits, scheduling algorithms designed for asynchronous circuits[28] can be implemented in the Scheduling Toolbox

Handshake Component STG's A

A.1 Decomposed handshake blocks

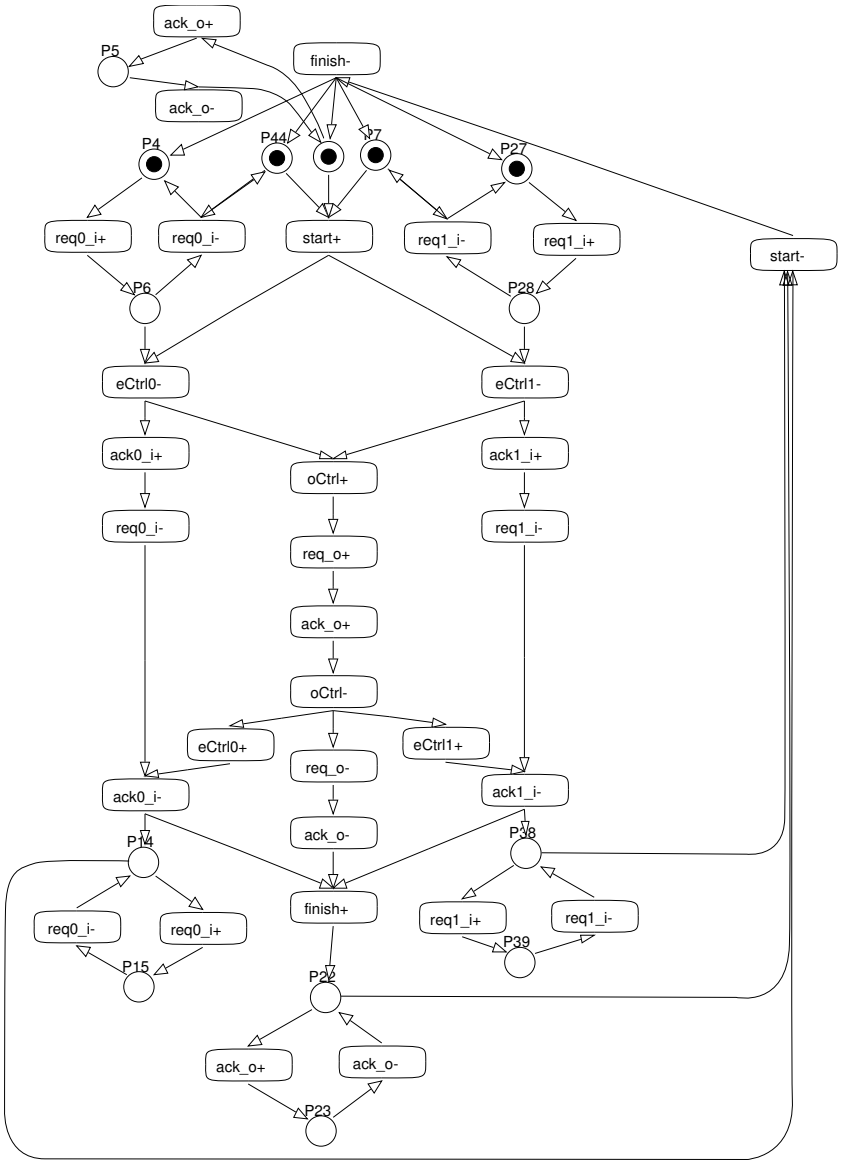
A.1.1 Inputselect



In: *start, req_i, ack_o*
Out: *req_o, ack_i, finish*

Figure A.1: Inputselect block

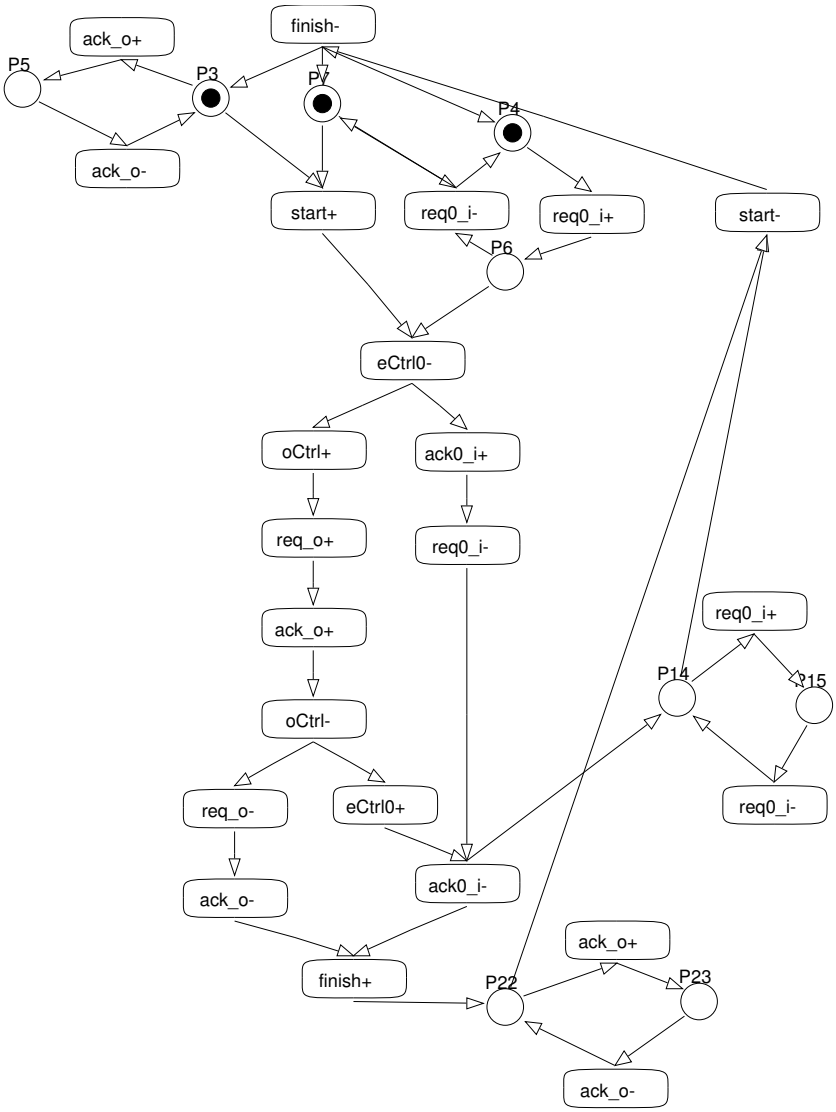
A.1.2 Outputselect



In: *start, req0_i, req1_i, ack_o*
Out: *eCtrl0, eCtrl1, oCtrl, req_o, ack0_i, ack1_i, finish*

Figure A.2: Outputselect block with two inputs

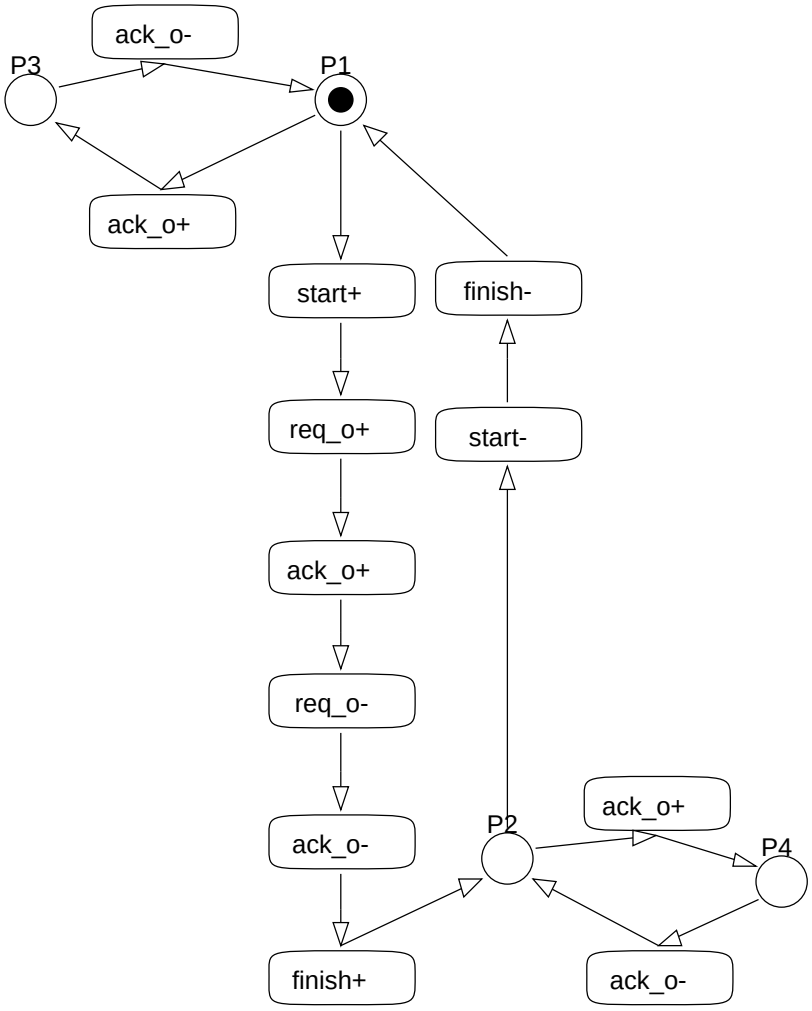
A.1.3 Outputselect single



In: *start, req0_i, ack_o*
Out: *eCtrl0, oCtrl, req_o, ack0_i, finish*

Figure A.3: Outputselect block with one input

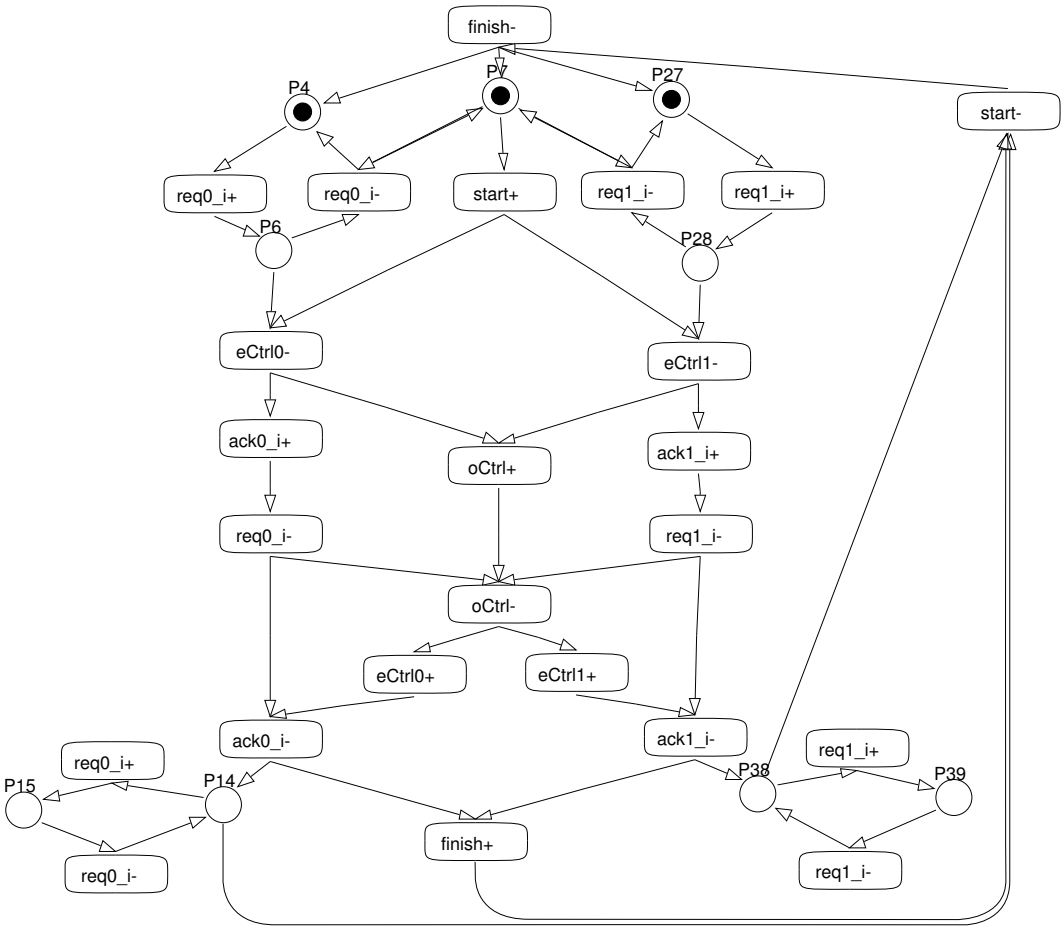
A.1.4 Fake request



In: *start, ack_o*
Out: *req_o, finish*

Figure A.4: Fake request block

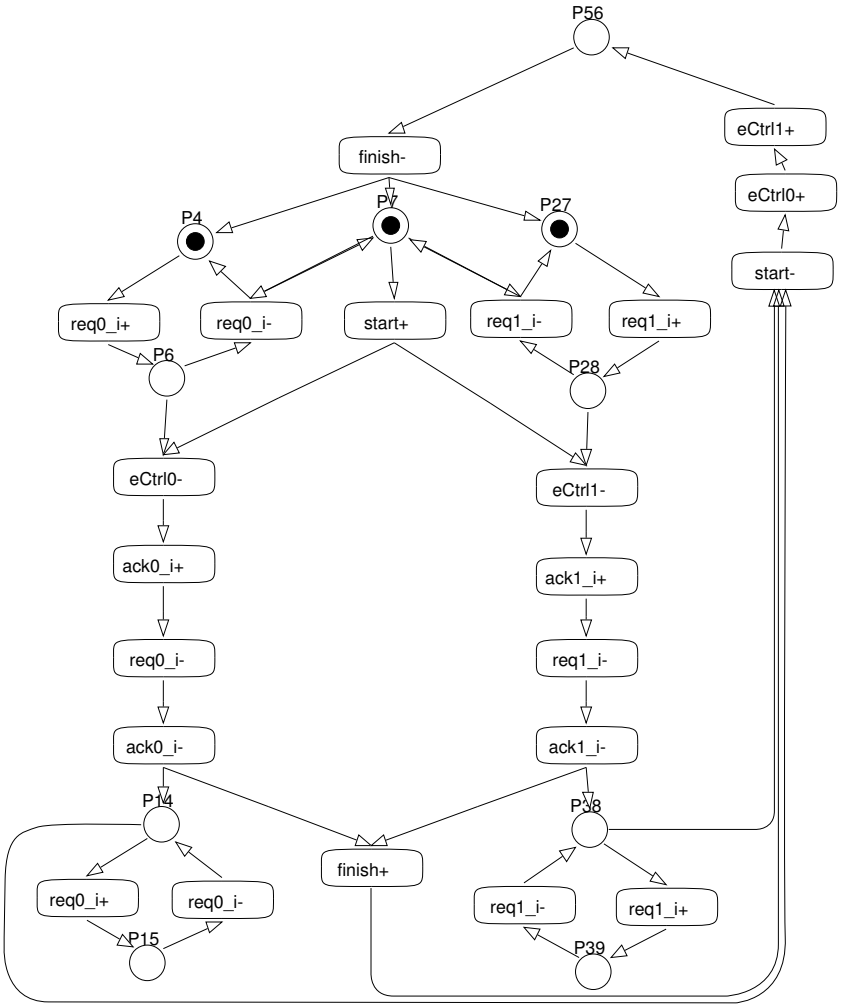
A.1.5 Fake acknowledge



In: *start, req0_i, req1_i*
Out: *ack0_i, ack1_i, eCtrl0, eCtrl1, oCtrl, finish*

Figure A.5: Fake acknowledge block

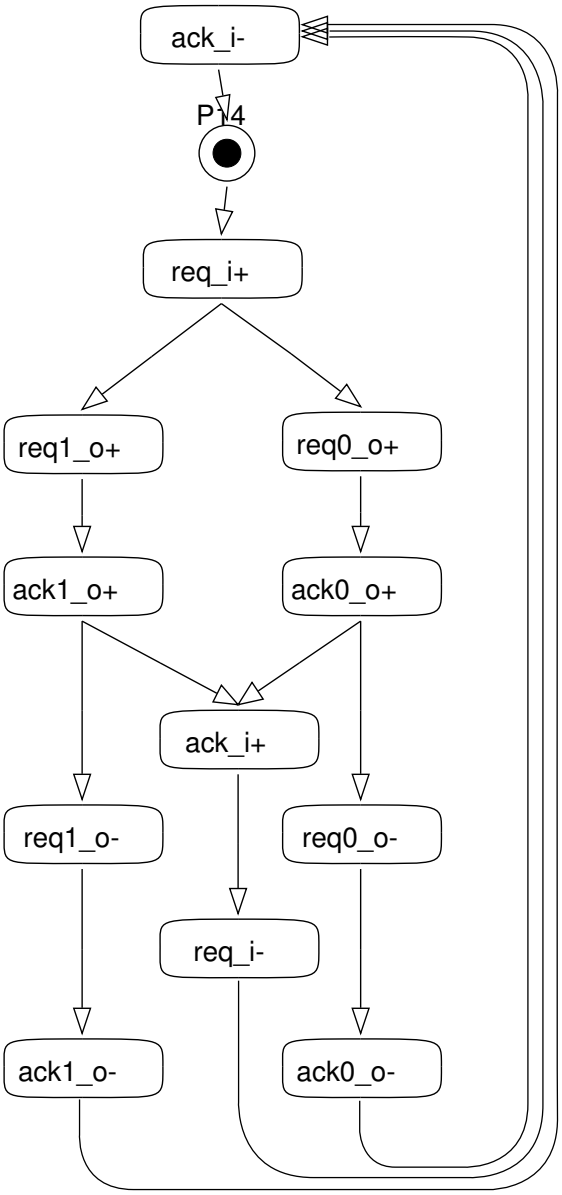
A.1.6 Hold data



In: *start, req0_i, req1_i*
Out: *ack0_i, ack1_i, eCtrl0, eCtrl1, finish*

Figure A.6: Hold data block

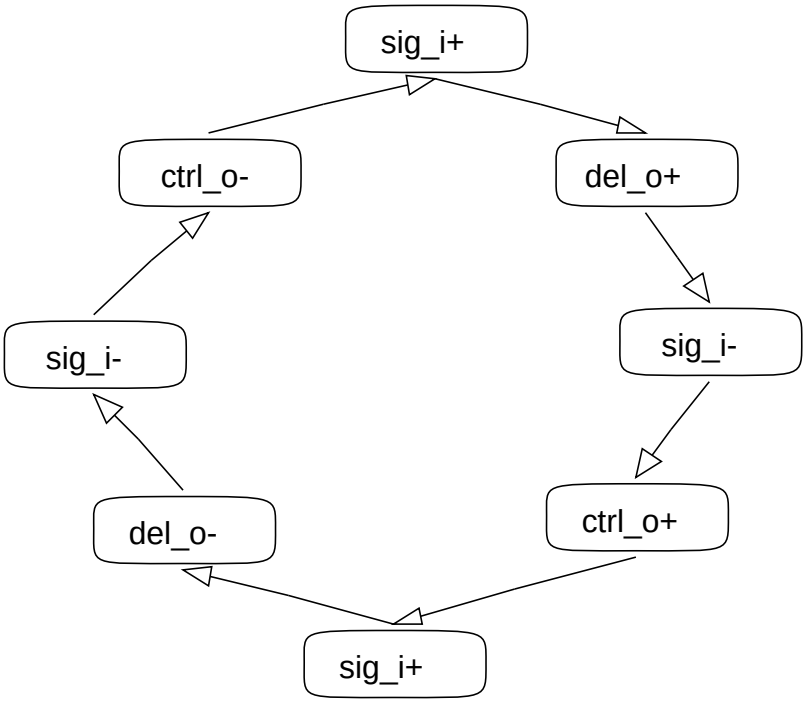
A.1.7 Fork request



In: *req_i, ack0_o, ack1_o*
Out: *req0_o, req1_o, ack_i*

Figure A.7: Fork request block

A.1.8 Delay controller

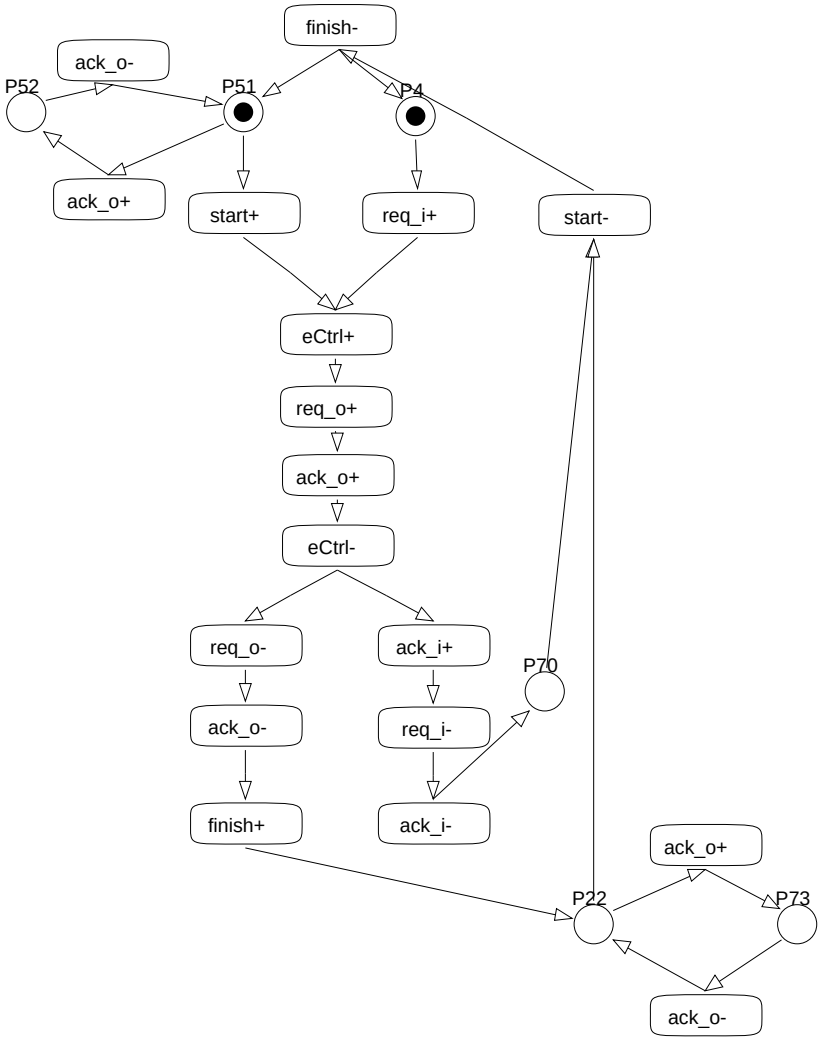


In: *sig_i*
Out: *del_o, ctrl_o*

Figure A.8: Delay controller block

A.2 More concurrent handshake blocks

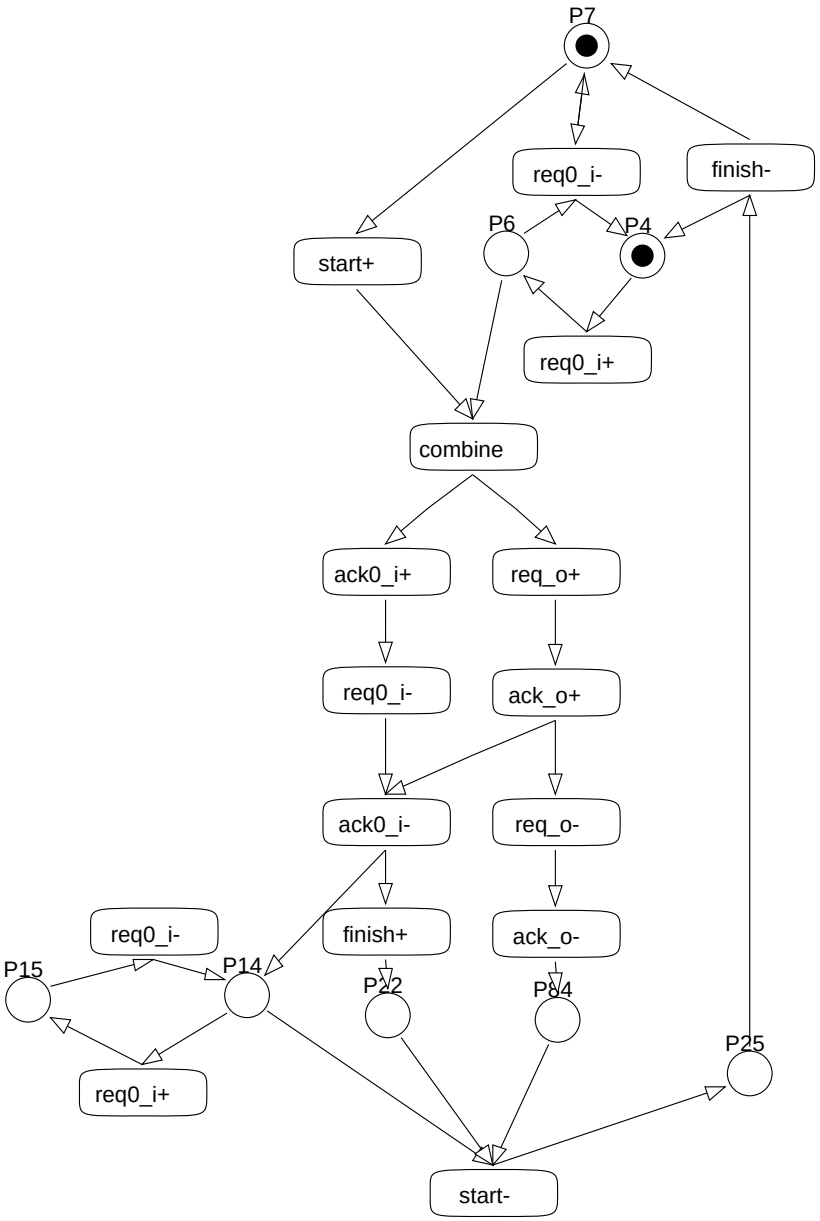
A.2.1 Inputselect



In: *start, req_i, ack_o*
Out: *req_o, ack_i, eCtrl, finish*

Figure A.9: Optimized inputselect block

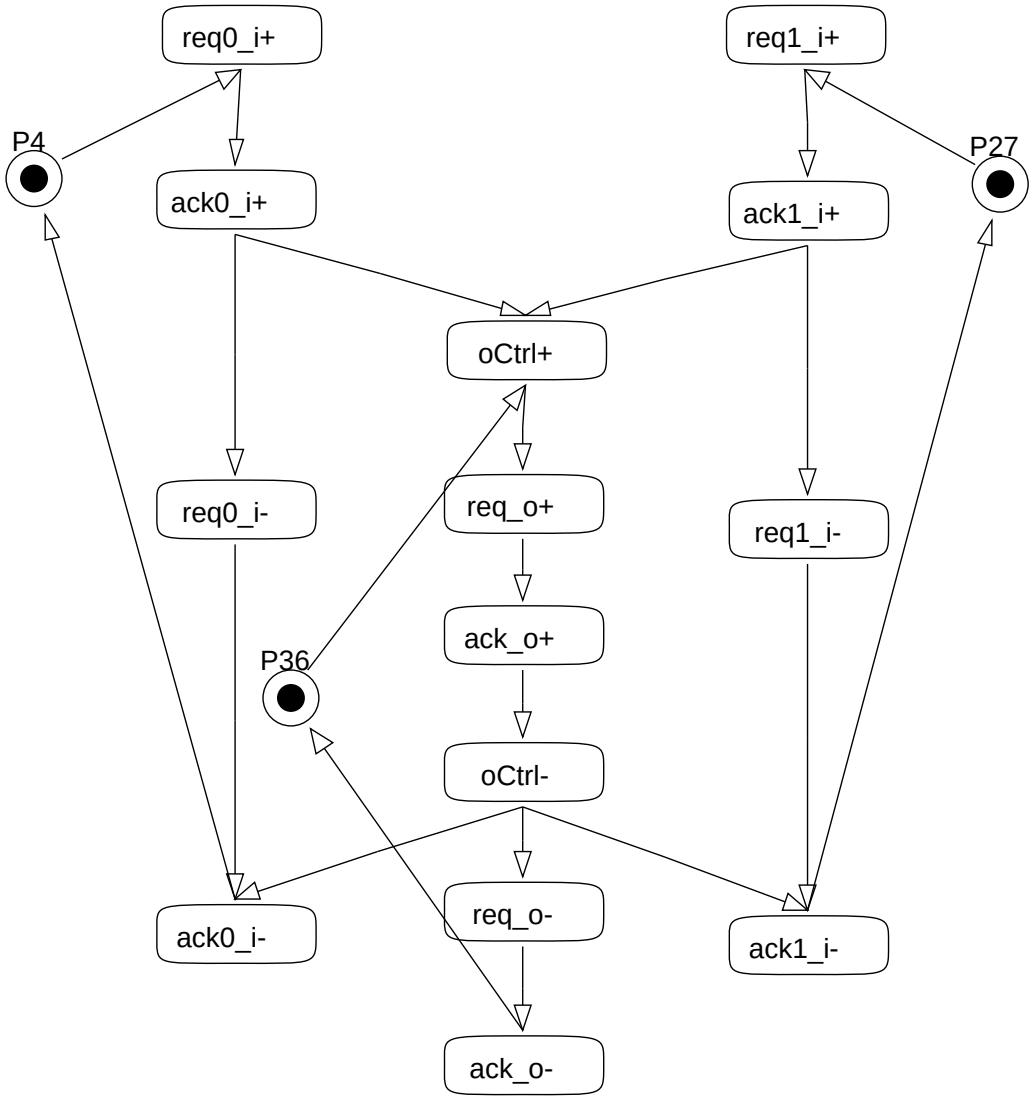
A.2.2 Outputselect



In: *start, req0_i, ack_o*
Out: *req_o, ack0_i, finish*
Dummy: *combine*

Figure A.10: Optimized outputselect block

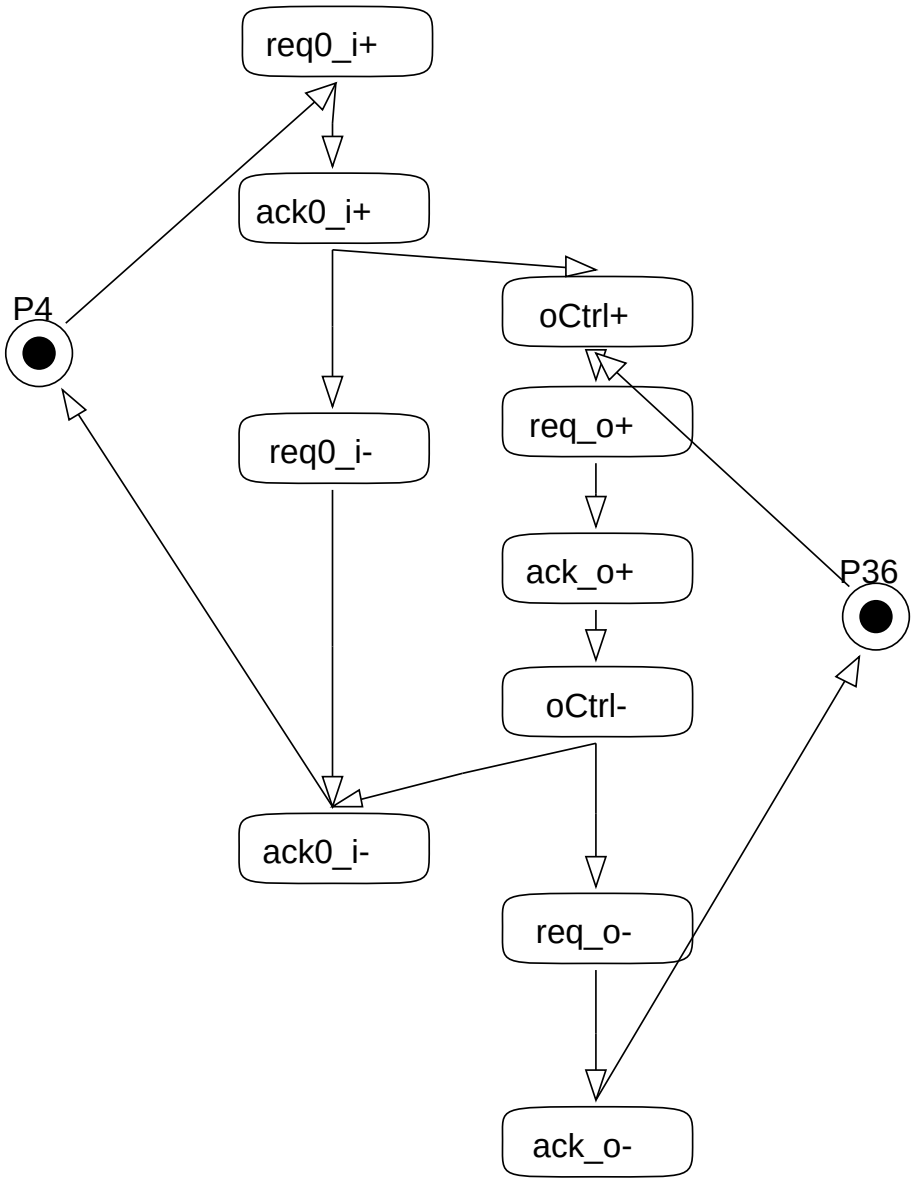
A.2.3 Latch Control



In: *req0_i, req1_i, ack_o*
Out: *oCtrl, req_o, ack0_i, ack1_i*

Figure A.11: Latchcontrol block with two inputs

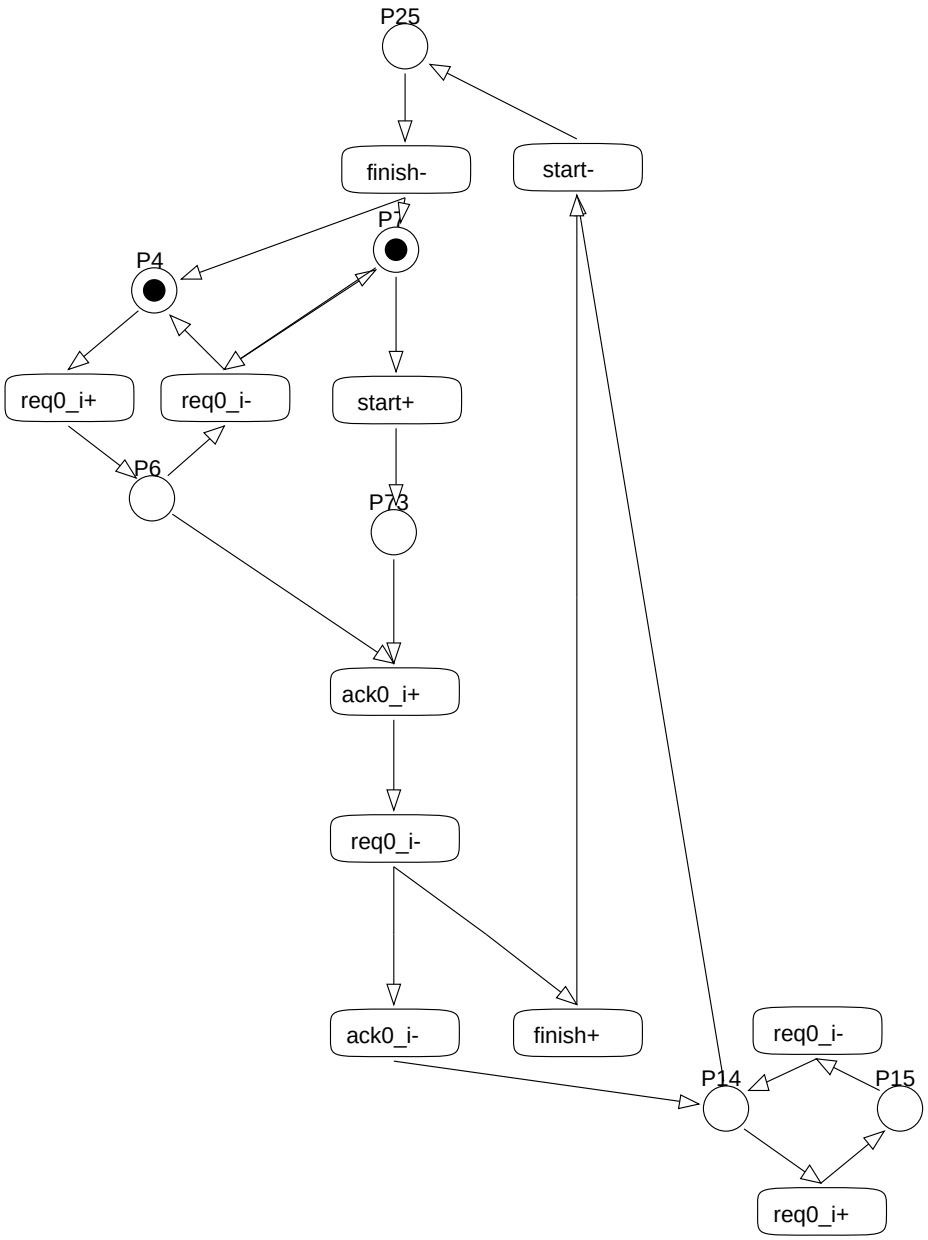
A.2.4 Latch Control single



In: *req0_i, ack_o*
Out: *oCtrl, req_o, ack0_i*

Figure A.12: Latchcontrol block with one input

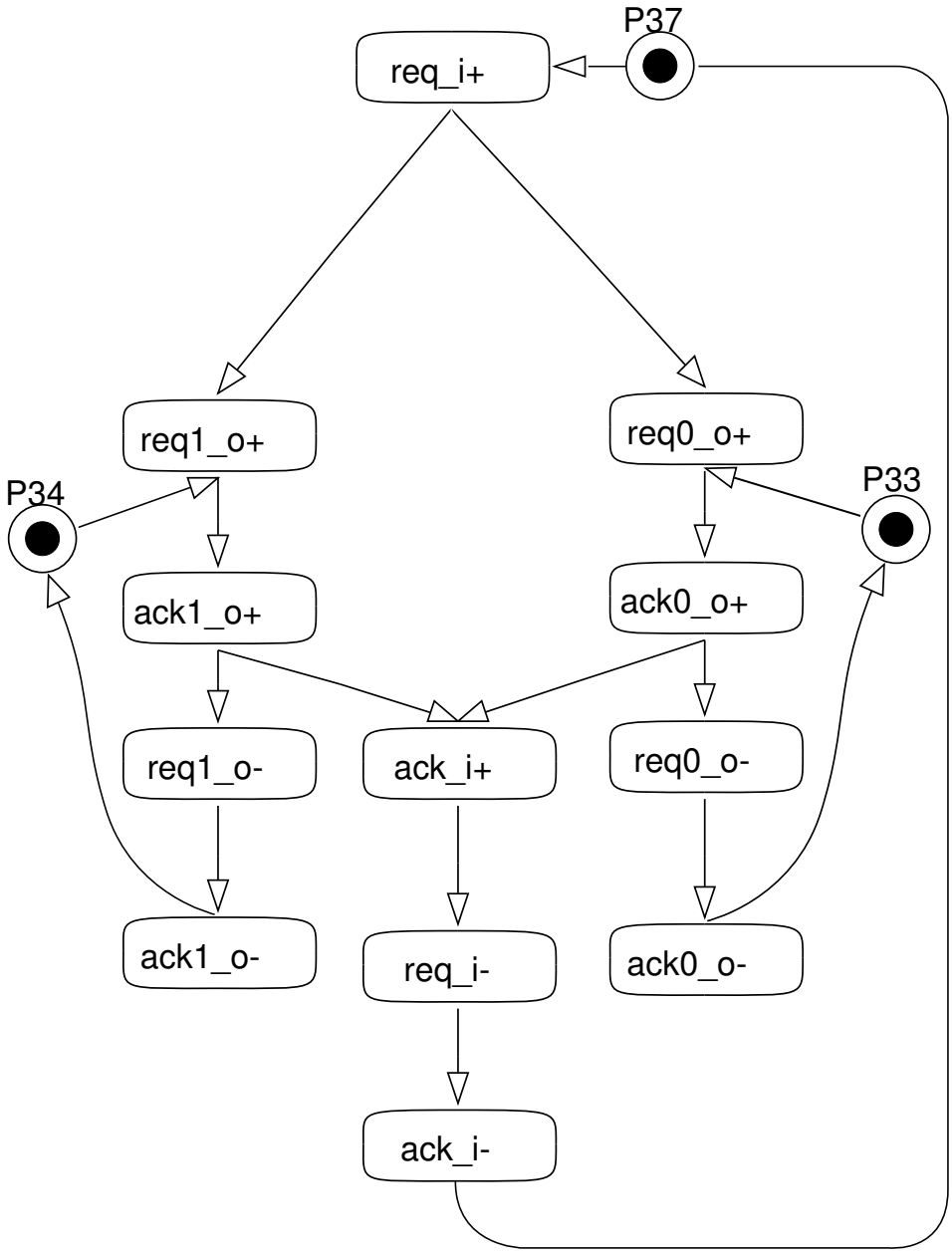
A.2.5 Fake acknowledge



In: *start, req0_i*
Out: *ack0_i, finish*

Figure A.13: Optimized fake acknowledge block

A.2.6 Fork request



In: *req_i, ack0_o, ack1_o*
Out: *ack_i, req0_o, req1_o*

Figure A.14: Optimized fork request block

B

LWDF latency model

Low-to-high transitions of request and acknowledge signals (ps after start)

Operation	MUL1				MUL2				ALU1						ALU2						ALU3					
	ReqI	*AckI*	*ReqO*	*AckO*	*ReqI*	AckI	*ReqO*	Ack)	*Req0*	Ack0	*Req1*	Ack1	*ReqO*	AckO	*Req0*	Ack0	*Req1*	Ack1	*ReqO*	AckO	*Req0*	Ack0	*Req1*	Ack1	*ReqO*	AckO
0	6250	11475	14350	18725	6250	11475	14350	26125	0	3725	0	0	6250	11475	0	3725	0	0	6250	11475	0	3725	0	0	6250	20375
1	15150	20375	23250	33525					14350	18075	0	0	20600	0	14350	18075	0	0	20950	26125	14350	18075	0	0	21300	26125
2	28000	33225	36100	41925					21750	25475	21750	25475	28000	33225	21750	25475	0	0	28000	0	23250	26975	0	0	30150	48325
3	37225	42450	45325	49700					30150	33875	0	0	36400	37225	29150	32875	0	0	35400	0	51575	55300	0	0	57825	0
4									37550	41275	0	0	43800	0	36550	40275	0	0	42800	0						
5									45325	49050	0	0	51575	0	45325	49050	43950	47675	51575	57100						
6															52725	56450	0	0	58025							

Critical path

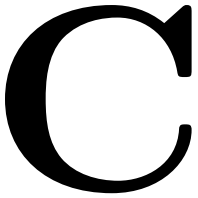
Operation	*External Delay*	*Fork delay*	*Path delay*	*Cumulative delay*
ALU1	ReqI to ReqO		6250	6250
MUL1	ReqI to ReqO	ForkReq	8100	14350
ALU2	ReqI to NextOp		7400	21750
ALU2	ReqI to NextOp		7400	29150
ALU2	ReqI to AckI	ForkAck	4375	33525
MUL1	AckO to ThirdOp		3700	37225
MUL1	ReqI to ReqO	ForkReq	8100	45325
ALU2	ReqI to NextOp		7400	52725
ALU2	ReqI to AckI	ForkAck	4375	57100
ALU2	AckO to ReqO		925	58025

External delays

	MUL1	*MUL2*	*ALU1*	*ALU2*	*ALU3*
DelayOp	5000	5000	3500	3500	3500
DelayLatch	1000	1000	1000	1000	1000
ReqI to ReqO	6950	6950	5450	5450	5450
ReqI to NextOp	8100	8100	6600	6600	6600
ReqI to AckI	5225	5225	3725	3725	3725
AckO to ThirdOp	3700	3700	3700	3700	3700
AckO to ReqO	925	925	925	925	925
ForkDelayReq	350				
ForkDelayAck	650				

Figure B.1: Spreadsheat with request and acknowledge event times

Design Compiler Scripts



C.1 Delay generation

```
# Example script for specifying delay
# A design with one input, one output and a buffer is used to test
    this code

#Assume only one input and one output.
set delayin [all_inputs];
set delayout [all_outputs];

# Create the virtual clock as reference for timing constraints
create_clock -name clk0 -period $min_delay
# Make sure hold-times are taken into account
set_fix_hold clk0
# Set minimal delay priority over maximal delay
# Because the minimal delay makes reliable operation possible
# and the maximal delay is only for performance
set_cost_priority {min_delay max_delay max_design_rules}
# Set multicycle path to two to make the required hold-time
# equal to the clock period
set_multicycle_path 2 -from $delayin -to $delayout
# Specify minimal delays for hold time
# Zero to make exactly one clock the minimal delay
set_input_delay -min 0 -clock clk0 $delayin
set_output_delay -min 0 -clock clk0 $delayout
# Specify maximal delay for setup-time
# Zero makes two clock cycles the maximal delay.
# Since the clock period is the minimal delay, we have
# to substract the maximum delay from two mimimal delays.
set_input_delay -max [expr {(2*$min_delay)-$max_delay}] -clock clk0
    $delayin
set_output_delay -max 0 -clock clk0 $delayout
```

C.2 Timing constraints

```
#The original .cir filename
set design FFT;
#the specified maximum latency for the operations
set alu_delay 3.802;
set mul_delay 5;
set reg_delay 0.5;
set reg_in_delay 0.2;
#Set the overhead which can be hidden by the first delay
set input_overhead 1;
#Set the overhead which can be hidden by the second delay
set output_overhead 0.5;
#Minimum delay for delay element is operation latency minus overhead.
#Specify the maximum delay of the delay element in terms of mimimal
    delay
set delay_interval 1.2;


##########################################################
# Begin of constraint specifications                     #
##########################################################


#Calculate the total latency including output register in order to
    create an oCtrl signal
set mul_reg_delay [expr {$mul_delay+$reg_delay}];
set alu_reg_delay [expr {$alu_delay+$reg_delay}];
set reg_reg_delay [expr {$reg_in_delay+$reg_delay}];
set savedesign [current_design];

#MUL
#Create Octrl signal
create_clock -name "oCtrlmul" -period $mul_reg_delay -waveform "
    $mul_delay $mul_reg_delay  "  [concat $design\_SSG_Lbl/mul*FSM_Lbl
    /LATCH_CONTROL/oCtrl]
#Ignore all control signals by giving it a large negative input delay
set_input_delay -max -100 -clock oCtrlmul -clock_fall [get_pins
    $design\_SSG_Lbl/mul*FSM_Lbl/*]
#Set the datapath delay to zero
set_input_delay -max 0 -clock oCtrlmul -clock_fall [
    remove_from_collection [get_pins $design\_SSG_Lbl/mul*FSM_Lbl/*] [
    get_pins [concat $design\_SSG_Lbl/mul*FSM_Lbl/*req* $design\
    _SSG_Lbl/mul*FSM_Lbl/*ack* $design\_SSG_Lbl/mul*FSM_Lbl/rst
    $design\_SSG_Lbl/mul*FSM_Lbl/start]]]
set_output_delay -max 0 -clock oCtrlmul -clock_fall $design\_SSG_Lbl/
    mul*FSM_Lbl/dataout

#ALU
#Create Octrl signal
```

```
create_clock -name "oCtrlalu" -period $alu_reg_delay -waveform "
    $alu_delay $alu_reg_delay  "  [concat $design\_SSG_Lbl/alu*FSM_Lbl
    /LATCH_CONTROL/oCtrl]
#Ignore all control signals by giving it a large negative input delay
set_input_delay -max -100 -clock oCtrlalu -clock_fall [get_pins
    $design\_SSG_Lbl/alu*FSM_Lbl/*]
#Set the datapath delay to zero
set_input_delay -max 0 -clock oCtrlalu -clock_fall [
    remove_from_collection [get_pins $design\_SSG_Lbl/alu*FSM_Lbl/*] [
    get_pins [concat $design\_SSG_Lbl/alu*FSM_Lbl/*req* $design\
    _SSG_Lbl/alu*FSM_Lbl/*ack* $design\_SSG_Lbl/alu*FSM_Lbl/rst
    $design\_SSG_Lbl/alu*FSM_Lbl/start]]]
set_output_delay -max 0 -clock oCtrlalu -clock_fall $design\_SSG_Lbl/
    alu*FSM_Lbl/dataout

#Register
#Create Octrl signal
create_clock -name "oCtrlreg" -period $reg_reg_delay -waveform "
    $reg_in_delay $reg_reg_delay  "  [concat $design\_SSG_Lbl/
    reg*FSM_Lbl/LATCH_CONTROL/oCtrl ]
#Ignore all control signals by giving it a large negative input delay
set_input_delay -max -100 -clock oCtrlreg -clock_fall [get_pins
    $design\_SSG_Lbl/reg*FSM_Lbl/*]
#Set the datapath delay to zero
set_input_delay -max 0 -clock oCtrlreg -clock_fall [
    remove_from_collection [get_pins $design\_SSG_Lbl/reg*FSM_Lbl/*] [
    get_pins [concat $design\_SSG_Lbl/reg*FSM_Lbl/*req* $design\
    _SSG_Lbl/reg*FSM_Lbl/*ack* $design\_SSG_Lbl/reg*FSM_Lbl/rst
    $design\_SSG_Lbl/reg*FSM_Lbl/start]]]
set_output_delay -max 0 -clock oCtrlreg -clock_fall $design\_SSG_Lbl/
    reg*FSM_Lbl/dataout

#set the self-loop path invalid since the data is always there before
     eCtrl goes high. (Because oCtrl has to go low first)
set_false_path -through $design\_SSG_Lbl/*FSM_Lbl/*_Lbl/out_r_reg*/Q
    -to $design\_SSG_Lbl/*FSM_Lbl/*_Lbl/out_r_reg*/data_in


#########################################################
# Begin of delay generation                             #
#########################################################

#MUL delay

current_design asym_delay_type_gMUL

#Group they delay cell(s) in order to contain them in a single design
group [get_cells delay*] -design_name delay_mul_ref -cell_name
    delay_mul_cell;
#And select that design
current_design delay_mul_ref;
```

```
#Set the delay requirements from specs
set min_delay [expr {$mul_delay-$input_overhead}];
set max_delay [expr {$delay_interval*$min_delay}];

#Run the delay constraint script
source delay.tcl
#Set OC to Typpical in order to make sure delay variablity doesn't
    screw up the circuit
set_operating_conditions TCCOM
#Since it is a simple delay line, compile will do just fine
compile

#Go back one level
current_design asym_delay_type_gMUL
#And make sure the delay is not changed anymore
set_dont_touch delay_mul_cell;

#REG delay

current_design asym_delay_type_gREG

#Group they delay cell(s) in order to contain them in a single design
group [get_cells delay*] -design_name delay_reg_ref -cell_name
    delay_reg_cell;
#And select that design
current_design delay_reg_ref;

#Set the delay requirements from specs
set min_delay [expr {$reg_in_delay-$input_overhead}];
set max_delay [expr {$delay_interval*$min_delay}];

#Run the delay constraint script
source delay.tcl
#Set OC to Typpical in order to make sure delay variablity doesn't
    screw up the circuit
set_operating_conditions TCCOM
#Since it is a simple delay line, compile will do just fine
compile

#Go back one level
current_design asym_delay_type_gREG
#And make sure the delay is not changed anymore
set_dont_touch delay_reg_cell;

#ALU delay

current_design asym_delay_type_gALU

#Group they delay cell(s) in order to contain them in a single design
group [get_cells delay*] -design_name delay_alu_ref -cell_name
```

```
    delay_alu_cell;
#And select that design
current_design delay_alu_ref;

#Set the delay requirements from specs
set min_delay [expr {$alu_delay-$input_overhead}];
set max_delay [expr {$delay_interval*$min_delay}];

#Run the delay constraint script
source delay.tcl
#Set OC to Typpical in order to make sure delay variablity doesn't
    screw up the circuit
set_operating_conditions TCCOM
#Since it is a simple delay line, compile will do just fine
compile

#Go back one level
current_design asym_delay_type_gALU
#And make sure the delay is not changed anymore
set_dont_touch delay_alu_cell;

#Output register delay

current_design asym_delay_type_gREG_OUT

#Group they delay cell(s) in order to contain them in a single design
group [get_cells delay*] -design_name delay_reg_out_ref -cell_name
    delay_reg_cell;
#And select that design
current_design delay_reg_out_ref;

#Set the delay requirements from specs
set min_delay [expr {$reg_delay-$output_overhead}];
set max_delay [expr {$delay_interval*$min_delay}];

#Run the delay constraint script
source delay.tcl
#Set OC to Typpical in order to make sure delay variablity doesn't
    screw up the circuit
set_operating_conditions TCCOM
#Since it is a simple delay line, compile will do just fine
compile

#Go back one level
current_design asym_delay_type_gREG_OUT
#And make sure the delay is not changed anymore
set_dont_touch delay_reg_cell;


#Select the muller_C element
current_design MULLER_C_ELEMENT;
```

```
    #Run compile since this will not screw up the feedback
167 compile;
    #Select 3-input muller_c element
    current_design MULLER_C_ELEMENT3;
    #Run compile since this will not screw up the feedback
    compile;
172
    #Load the original design again
    current_design $savedesign;

    #Make sure the Speed Independent components are not touched.
177 set_dont_touch [get_designs MULLER_C_ELEMENT*];
    set_dont_touch [get_designs *_net*];
```

Bibliography

[1] P.J. Ashenden. *The student's guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[2] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. Handshake protocols for de-synchronization. In *Asynchronous Circuits and Systems, 2004. Proceedings. 10th International Symposium on*, pages 149 – 158, april 2004.

[3] S.A. Butt, S. Schmermbeck, J. Rosenthal, A. Pratsch, and E. Schmidt. System level clock tree synthesis for power optimization. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE '07*, pages 1–6, April 2007.

[4] J. Carmona, J. Cortadella, and E. Pastor. A structural encoding technique for the synthesis of asynchronous circuits. In *Application of Concurrency to System Design, 2001. Proceedings. 2001 International Conference on*, pages 157–166, 2001.

[5] K.T. Christensen, P. Jensen, P. Korger, and J. Sparso. The design of an asynchronous tinyrisctm tr4101 microprocessor core. In *Advanced Research in Asynchronous Circuits and Systems, 1998. Proceedings. 1998 Fourth International Symposium on*, pages 108 –119, mar. 1998.

[6] J Cortadella, M Kishinevsky, A Kondratyev, L Lavagno, and A Yakovlev. Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers, 1996.

[7] Universite de bretagne sud. nontech_16b.lib. Part of GAUT software package, feb. 2010.

[8] M.E. Dean, D.L. Dill, and M. Horowitz. Self-timed logic using current-sensing completion detection (cscd). In *Computer Design: VLSI in Computers and Processors, 1991. ICCD '91. Proceedings, 1991 IEEE International Conference on*, pages 187 –191, oct. 1991.

[9] D. A. Edwards and W. B. Toms. Design, automation and test for asynchronous circuits and systems. Technical Report 3, Information Society Technologies, Jun 2004.

[10] Faraday Technology Corporation. Fsd0a_a data book, 2004.

[11] R.M. Fuhrer, S.M. Nowick, and T Chelcea. "minimalist": A cad package for the synthesis and optimization of asynchronous burst-mode controllers. "Minimalist" Release v2.0., november 2007. READMY file included in the release.

[12] A.C. de Graaf, H.J. Lincklaen Arrins, and T.G.R van Leuken. *Digital Design Flow Tutorial*. Delft University of Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Circuits & Systems Group, March 2009.

[13] E. Grass, R.C.S. Morling, and I. Kale. Activity-monitoring completion-detection (amcd): a new single rail approach to achieve self-timing. In *Advanced Research in Asynchronous Circuits and Systems, 1996. Proceedings., Second International Symposium on*, pages 143 –149, mar. 1996.

[14] Handshake Solutions. Ht-80c51 microcontroller leaflet, may 2004.

[15] D.A. Huffman. The design and use of hazard-free switching networks. *J. ACM*, 4(1):47–62, 1957.

[16] IEEE. Ieee standard for vital asic (application specific integrated circuit) modeling specification. *IEEE Std 1076.4-2000*, pages 1–420, 2001.

[17] D. Kearney and N.W. Bergmann. Performance evaluation of asynchronous logic pipelines with data dependent processing delays. In *Asynchronous Design Methodologies, 1995. Proceedings., Second Working Conference on*, pages 4 – 13, 30-31 1995.

[18] R. Kol and R. Ginosar. A doubly-latched asynchronous pipeline. In *Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings., 1997 IEEE International Conference on*, pages 706 –711, oct. 1997.

[19] O Kraus. About dgc. Retrieved September 10, 2010, from http://dgc.sourceforge.net/dgcprogs_toc.html, May 2002.

[20] O. Kraus and M. Padeffke. Xbm2pla: A flexible synthesis tool for extended burst mode machines. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 11092, Washington, DC, USA, 2003. IEEE Computer Society.

[21] H.J. Lincklaen Arrins. Implementation of an 18-point imdct on fpga. Internal report, CAS, TU Delft, Jun 2005.

[22] D.W. Lloyd and J.D. Garside. A practical comparison of asynchronous design styles. In *Asynchronous Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pages 36 –45, 2001.

[23] A.J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *AUSCRYPT '90: Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 263–278, Cambridge, MA, USA, 1990. MIT Press.

[24] G. de Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[25] C.J. Myers. *Asynchronous Circuit Design*. Wiley-Interscience, 2001.

[26] S.M. Nowick. Automatic synthesis of burst-mode asynchronous controllers, 1995.

[27] S.M. Nowick. Design of a low-latency asynchronous adder using speculative completion. *IEE Proceedings - Computers and Digital Techniques*, 143(5):301–307, 1996.

[28] H Saito, N Hamada, N Jindapetch, T Yoneda, C.J. Myers, and T Nanya. Scheduling methods for asynchronous circuits with bundled-data implementations based on the approximation of start times. *IEICE Transactions*, 90-A(12):2790–2799, 2007.

[29] M. Schaefer, W. Vogler, R. Wollowski, and V. Khomenko. Strategies for optimised stg decomposition. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pages 123–132, June 2006.

[30] M. Schaefer, D. Wist, and R. Wollowski. Desij–enabling decomposition-based synthesis of complex asynchronous controllers. In *Application of Concurrency to System Design, 2009. ACSD '09. Ninth International Conference on*, pages 186 –190, jul. 2009.

[31] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.

[32] M. Simmonds. Desynchronization methods for scheduled circuits. Master's thesis, TU Delft, November 2009.

[33] J Sparsøand S Furber. *Principles of Asynchronous Circuit Design*. Kluwer Academic publishers, 2001.

[34] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

[35] Synopsys Customer Education Service. *Design Compiler 1 Workshop Student Guide*. Synopsys, march 2007.

[36] S.H. Unger. Hazards and delays in asynchronous sequential switching circuits. *Circuit Theory, IRE Transactions on*, 6(1):12 – 25, mar. 1959.

[37] S.H. Unger. Hazards, critical races, and metastability. *Computers, IEEE Transactions on*, 44(6):754 –768, jun. 1995.