Early Output Logic and Anti-Tokens

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy in the
Faculty of Engineering and Physical Sciences

Septermber 2005

Charles Brej

School of Computer Science

Contents

List of Figures

List of Tables

Abstract

Asynchronous logic has for some time been promoted as being able to take advantage of average case performance. Unfortunately the overheads of using asynchronous techniques, such as the return to zero phase and unnecessary synchronisations, have often outweighed the benefits. The aim of the research described is to take full advantage of the performance benefits attainable through the use of asynchronous methodologies, then to overcome the overheads introduced.

The thesis introduces the Early Output design methodology which allows the generation of circuits which synchronise the production of outputs with the minimal set of inputs, thus generating the result as soon as possible. The throughput problem is tackled through a series of optimisations. The optimisations allow the removal of unnecessary synchronisation points which degrade performance. One novel optimisation is the anti-token latch which allows further improvements in performance by inhibiting operations once their results are found to be unnecessary.

To determine where the optimisations should be applied, a novel dynamic analysis technique was developed. This targets improving average case performance through simulating the design running a benchmark and attaining the Slowest Path (a sequence of elements which contributed to the delay of the simulation run).

The effect of the optimisation is demonstrated on a range of circuits presenting each optimisation's applicability to various commonly used structures.

The result of these techniques is a system capable of generating circuits which generally perform faster than their synchronous equivalents.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank:

My proof readers Andrew Bardsley, Luis Plana and Viv Woods who were simply execlent.

Shuet-Ying Cheung for putting up with me over the last 7 years.

Good friends Will Toms, Matthew Horsnell, Andrew Bardsley, John Bainbridge, Paul Capewell, Wannarat (Eve) Suntiamorntut and many others.

The asynchronous community for being so friendly.

My supervisor James Garside and my advisor Doug Edwards.

Chapter 1: Introduction

1.1 Justification

For a number of years the VLSI design community has been looking towards asynchronous logic to solve some of the problems that arise when using global clocks on very large circuits [1]. There are some advantages inherent in asynchronous circuits over their synchronous counterparts. Lower emissions of electromagnetic noise [2][3][4], no clock distribution (saving area and power) [5], no clock skew [5], robustness to environmental variations (e.g. temperature and power supply) or transistor variations, better modularity [6] and better security [7][8] are just some of the properties for which most asynchronous designs have shown advantages over synchronous designs. The ability to show these advantages over synchronous designs in a number of properties has been demonstrated. Low power, low latency [9][10] and high throughput [11] are three properties which have been claimed but need to be specifically targeted in order to exploit them at the expense of the others. It is important to distinguish the difference between throughput and latency rather than just calling them performance. The Amulet group has, in the past, created three low power microprocessors using low power asynchronous techniques [2][13]. Others have used fine grain pipelining to achieve high throughput at the cost of latency and power consumption [11][12]. By trying to exploit all three properties the final design will hold little if any advantage over the synchronous implementation. Alternatively by trying to exploit just one of these properties it is possible to gain it at cost to the others.

- The power consumption of synchronous circuits is often higher than asynchronous equivalents as the full global clock network has to be driven at a very high rate and many pipeline stages are executed in instances where the result is not desired [2][10].

- Low latency can be achieved by exploiting the average case performance present in some asynchronous circuits [14].

- High throughput is present in circuits with very high density pipelining, which is made difficult by the presence of a global clock skew.

1.2 Synchronous logic

The basis of computing is combinatorial logic which takes a set of inputs and, depending on the state of these inputs, generates appropriate outputs. For very simple systems which perform only one function and do not keep state this is sufficient. More complex systems require some form of timing to partition the circuit temporally. The partitioning allows a single piece of combinatorial logic to be used for several different operations (e.g. different operands being passed though an ALU) or keep state and use the results of the previous operation as the next set of inputs (e.g. a cyclic multiplier). The alternatives to using such schemes are usually not realistic (e.g. creating a separate ALU for each executed instruction in a fixed program). The timing in the system can come from a number of sources.

Synchronous circuits rely on external timing to determine the completion of each pipeline stage and registers to stop data from one stage overwriting the data in the next stage.

1.2.1 Synchronous logic construction

In the diagram of a synchronous circuit (Figure 1.1) the clock net is connected to every flip-flop. As the clock ‘ticks’ the data changes from being the results of one stage to the inputs of the next, this construction is called a pipeline. Pipelines not only divide the system temporally but also spatially. A large operation, which has already been recycled temporally by passing different data though it, can also be divided spatially by allowing many operations to pass though different parts of the unit at the same time. This is a common method of increasing throughput at the penalty of latency due to the added delay of the latches.

Figure 1.2 shows how data moves from one stage to the next by shifting all data to the next stage at the rise of the clock. A global clock is used to ensure that sufficient time is

given for the result to be correct by the time it is accepted by the next stage and a stage holds only one data entry.

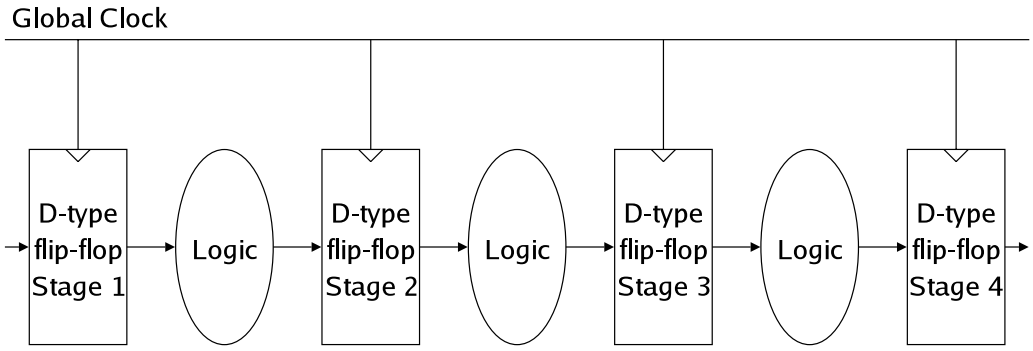

Figure 1.1: Synchronous pipeline

1.2.2 Synchronous pipeline properties

In figure 1.2 the coloured blocks represent a series of operations passing through a pipeline. The shaded areas of each stage represent the stage having completed its logical operation and the result being valid but waiting for the clock before it can move to the next stage. For example, when ‘D0’ passes through Stage 1 its result is ready $\sim^1/_4$ of a clock cycle before the next clock edge arrives. During this time, the data is unable to progress to the next stage. When ‘D0’ passes through Stage 3 it requires the entire clock cycle to perform its operation. This operation passes along the critical path and if the clock frequency was increased, circuit operation would fail because the result of the logical operation would not be ready in time to be accepted into the next latch. These operations may occur very rarely, but they force the clock to have a longer period in all cycles to always guarantee correct operation. This critical path delay must be found for the worst operating conditions of the circuit. This requirement usually degrades performance even further.

1.3 Asynchronous circuits

Asynchronous logic is a very broad term which can be used to describe any circuit which has the ability to keep and change state without the use of a global clock. Another

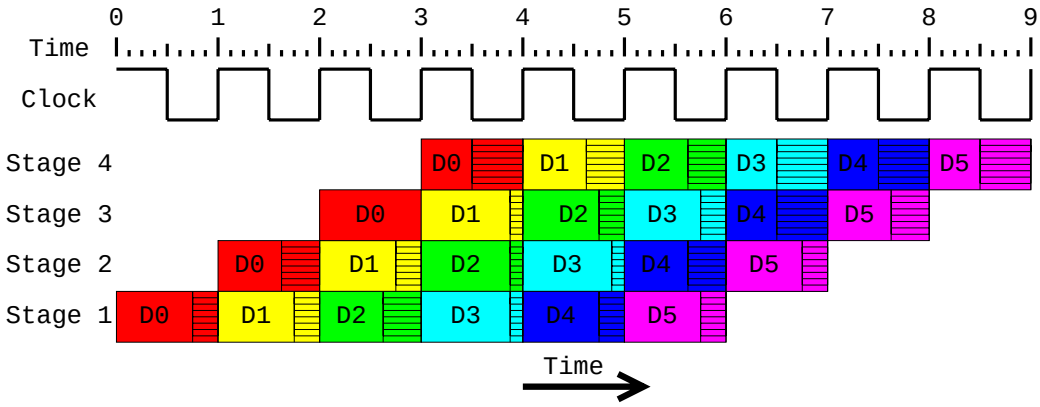

Figure 1.2: Synchronous pipeline occupancy diagram

generally accepted term is “self-timed”. This is more descriptive of the nature of these circuits as even asynchronous circuits synchronise.

1.3.1 Requirements of asynchronous circuits

As stated above the synchronous approach gives a timing reference which estimates the completion of a stage and ensures the stages are separated. If the timing and data separation properties can be reproduced without using a global clock it will allow the pipeline to execute faster than worst case performance. Stage completion can be determined in many ways: The easiest method is a matched delay; a series of gates provides a delay to match the stage logic depth. When external variables such as temperature or voltage slow down the circuit, this delay increases to allow the logic extra time to resolve the result. A more complex method is to use a data dependent matched delay which employs several matched delay lines, one of which is chosen depending on the data or the operation conducted. For example, if an ALU stage executed a fast, logical operation rather than a slow, arithmetic one then a shorter delay would be chosen.

The most precise method of completion detection is not to use matched delays but to use the logic to create a completion signal. The last two methods allow the data dependent speed improvements. Figure 1.3 shows an example of an asynchronous pipeline. The global clock is replaced with a set of asynchronous pipeline control elements. Once new data enters a stage, the request signal is generated on the wire labelled Req1 in figure 1.3. This signal goes through a matched delay, or is combined with a completion detection

signal, and, when the logic function has been evaluated, the request signal is emitted on wire Req2. The data is now ready to be accepted for use in the next stage.

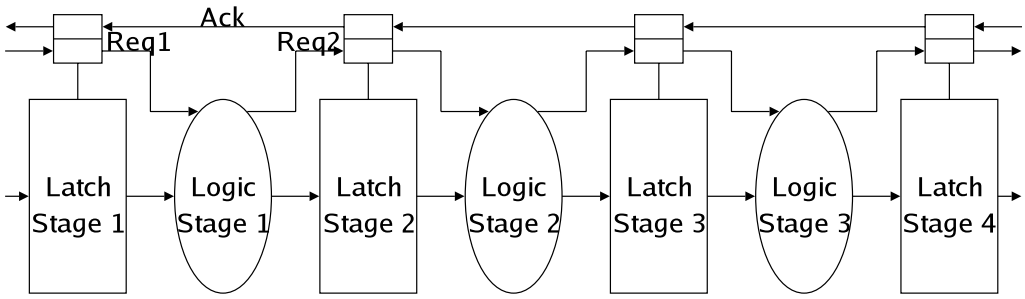

Figure 1.3: Asynchronous pipeline

This approach solves the completion detection problem but there is still the problem of one piece of data overwriting another in the next pipeline stage. To solve this, an acknowledge signal (Ack) is sent back to the requesting control unit to signal that it has accepted the data and that stage can be used for the next data. In turn the data that has been accepted is used in the next stage by emitting its request and the cycle then begins in the next stage. This is called *handshaking* and is used in asynchronous systems to guarantee a correct transfer of information while making no assumptions in the communication protocol on the delay of either the sender or the receiver.

1.3.2 Properties of asynchronous pipelines

Figure 1.4 shows an asynchronous pipeline executing the same computation as the synchronous pipeline in figure 1.2, there are noticeable differences between the two traces. Firstly, the asynchronous pipeline is faster as the optimisations described above are implemented. The speed improvement is due to the completion of each stage being determined on an individual basis rather than estimating the worst case delay of the slowest stage (using a global clock).

Unlike the synchronous pipeline, there are two different types of stalls in the asynchronous pipeline both of which were dealt with simply by using a clock in the synchronous version. The first is demonstrated in stage 2 after D0 has moved to stage 3. Here the stage 2 hardware is ready to accept new data but D1 has not completed its

function in stage 1. This is a *starvation* as the hardware has to wait for the data to become available. In the figure this is demonstrated with the dashed lines across the stalling area. The second type of stall is shown where D2 is trying to move from stage 1 to stage 2 but the stage is not ready to accept new data as it is still processing D1. This causes *blocking* as the data is ready but has to wait for the hardware to become available. In the figure it is shown with dashed lines across the stalling area with the data shading still present. When the pipeline contains too few data elements then starvation is common and the throughput is low. When the pipeline contains too many data elements then blocking appears often and causes high latency. A balanced pipeline would have low latency and high throughput and so avoiding these stalls is important.

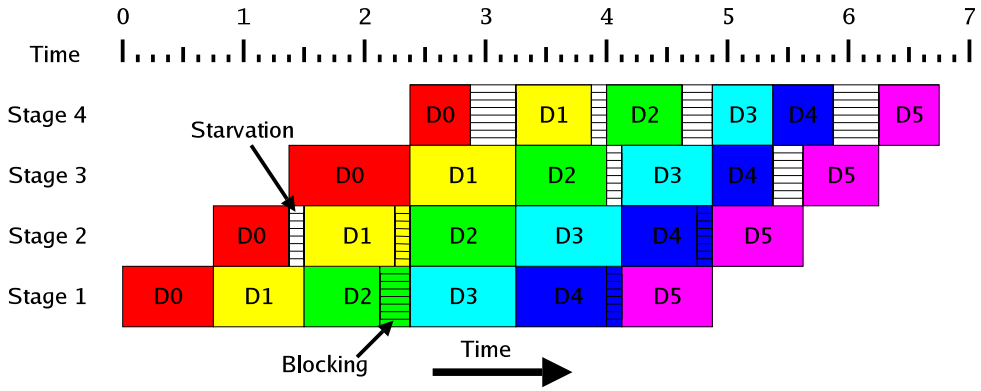

Figure 1.4: Asynchronous pipeline occupancy diagram

1.4 Aims of this research

From the advantages in performance, stated in the previous section, of asynchronous circuits over synchronous counterparts it would seem clear that all well balanced asynchronous circuits should operate much faster than synchronous designs. This unfortunately is not the case and asynchronous circuits rarely reach the performance of synchronous equivalents and even then this is only in structures particularly suited to the asynchronous approach. This has caused the advantages in latency and throughput to be generally dismissed by the synchronous community.

The latency advantage due to average case performance has never been fully demonstrated. While the data dependent timing was included in the Amulet 1 [15] and 2

[2] designs, in the last version (Amulet 3 [13] which was targeting higher performance) the data dependent delay was removed due to the complexity overhead outweighing any latency advantage. Other systems [23], which use bit level flow control, attempt to exploit unbalanced logical depth of individual outputs in stages to reduce latency. This unfortunately requires the logic to be constructed from special gates which indicate the completion of the result. These gates are so much slower than the normal gates and generally enforce synchronisation of all their inputs which causes them to have overheads which outweigh their advantages.

The throughput advantage has also been dismissed by the synchronous community. Although asynchronous circuits could implement much finer grained pipelining, most circuits concentrated on in this thesis suffer from a reset phase separating computation phases and often wasting more than half of the system capacity.

The aim of this research was to exploit the potential performance advantages of the asynchronous design methodology while tackling its weaknesses. The current methodologies were evaluated and their weaknesses were targeted. Firstly, the latency advantage was targeted and using the early output system this was improved. To do this, power and area were not considered and all emphasis was only on the latency. Later, to tackle the throughput problem the circuits have a series of optimisations placed on them which remove unnecessary synchronisations through the use of anti-tokens and early drop latches.

The combination of all these factors produces circuits which perform faster than their synchronous counterparts.

1.5 Contributions made by this work

The thesis presents the following advancements in knowledge in the field of asynchronous logic:

- Early output circuit synthesis allowing stages to generate results before all inputs are present. This includes the methodology, analysis of circuits constructed using this method and a number of considerations when using the technique.

- An understanding of the cause of missed early outputs where sufficient inputs are present yet the stage does not generate a result. And a method of generating circuits which avoid this kind of behaviour.

- QDI guarding of early output circuits to allows the generation of more robust circuits.

- Demonstration and analysis of an anti-token like behaviour in backward safe guarded circuits where late arriving inputs do not block the entire stage from continuing to operate.

- Anti-token latch designs, behaviour of anti-tokens and their effectiveness.

- Novel dynamic timing analysis technique based on a blame passing method.

- Optimisation system based on dynamic timing analysis.

1.6 Thesis Structure

This thesis will present a method of improving the performance of four-phase asynchronous circuits at the gate-level composition. Most of the methods presented primarily target dual-rail circuits, but the application of the methods to other systems (namely control circuits and bundled data) will also be presented.

The thesis does not deal with architectural or transistor level optimisations. Possible timing hazards and the timing assumptions made will be shown but the full method of avoiding hazards in highly timing variable technologies will not be presented.

The rest of the thesis is structured as follows:

Chapter 2 explains the fundamentals of asynchronous logic which are used throughout the thesis.

Chapters 3, 4 and 5 all present the particular aspects of three design styles. These are control circuits, bundled data, and dual rail.

Chapter 3 describes a conventional approach of constructing asynchronous circuits.

Chapter 4 presents the early output design methodology.

Chapter 5 introduces anti-tokens and extends the early output methodology. The motivation for these is explained and their behaviour is described.

Chapter 6 evaluates the performance of the methods presented. It presents methods of analysing constructions and optimising them.

Chapter 7 concludes the thesis by summarising the contributions made and suggesting the future work which could be conducted.

1.7 Publications

"An Automatic Synchronous to Asynchronous Circuit Convertor" (11th UK Asynchronous Forum)

"Early Output logic using Anti-Tokens" (13th UK Asynchronous Forum)

"Counterflow Networks" (13th UK Asynchronous Forum)

"Obtaining asynchronous benefits from synchronous design flow" (Third Working Group on Asynchronous Circuit Design Workshop)

"Early Output Logic using Anti-Tokens" (Twelfth International Workshop on Logic and Synthesis, IWLS 2003)

"Reduction in synchronisation in bundled data systems" (15th UK Asynchronous Forum)

"Reduction in synchronisation in bundled data systems" (Workshop on Token Based Computing, ToBaCo)

"Safe Early Output: An Improved QDI Logic System" (Fourth ACiD-WG Workshop)

"Early Output Logic and Anti-Tokens" (2004 MAPLD International Conference)

“A Quasi-Delay-Insensitive Method to Overcome Transistor Variation” (18th International Conference on VLSI Design)

Chapter 2: Fundamentals of Asynchronous Systems

The asynchronous design methodology is based on the use of handshakes to communicate data. To enable hazard free operation, most of these handshakes make no assumptions on the speed of each communicating unit. This ‘delay insensitivity’ can be extended to the computing parts of the system. These constraints make the generation of logic very difficult and so a series of less restrictive delay models have been defined for use in logic synthesis.

2.1 Asynchronous protocols

Communication in asynchronous systems is achieved using handshake signals. These handshakes are conducted between the source and the destination along two wires. A request signal is driven by the initiator of the transaction and the acknowledge signal is transmitted by the other end to signal the receipt of the request. The initiator can be either the source or destination unit. *Pull channel* handshakes are initiated by the destination while *push channel* handshakes are initiated by the source.

Although the handshake channel construction using request and acknowledge wires has become standardised, there are two protocols commonly used on these channels.

2.1.1 Two-phase signalling

The two-phase protocol [17] uses signal transitions to indicate the request and acknowledge messages. Each transition (alternating between up-going and down-going) of the request wire is acknowledged by a transition of the acknowledge wire to match the state of the request.

Figure 2.1 shows the behaviour of the protocol on a simple push channel. The source initiates the transaction by placing a transition on the request wire. Once the destination has received the message it replies with a transition on the acknowledge wire. This completes the transaction and the states of the request and acknowledge wires match once again. Once the source observes the transition on the acknowledge wire, it is able to initiate another transaction.

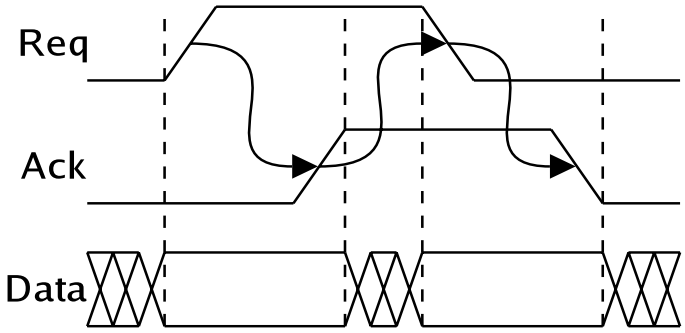

Figure 2.1: Two-phase protocol

2.1.2 Four-phase signalling

In single phase clocked synchronous systems, the flip-flops driven by a clock only update the state of their data output on the rising (or falling) edge of the clock. Using latches which trigger on both clock edges would allow the clock speed to be halved and hence reduce the power consumption of the clock generation and distribution. Although it is possible to create synchronous flip-flops which update their state on both edges of the clock, thus reducing the power consumption of the design, most of these latch designs are expensive as they effectively recreate a double speed clock internally to generate the latching signal or duplicate the latching logic (one for each phase).

In asynchronous two-phase circuits, each latch creates a latching signal which transitions at double the rate of its inputs or two latching elements are used. The four-phase protocol takes this into account and communicates across the request and acknowledge wires using level rather than edge sensitive signals. This causes the protocol to become somewhat more complicated but allows the construction of latches to be greatly simplified.

Figure 2.2 shows the behaviour of a four phase latch [19]. The sequencing of the request and acknowledge signals is the same as that in the two-phase protocol, but the data is now communicated once every two transitions. Additionally, the four-phase protocol has a number of schemes defining when the data is valid [18]. The *early* and *late* data validity schemes are used depending who has control of the bus on which the data is transmitted. Early data validity scheme is used when the source controls the data bus and so places its data on the bus and then sends a request to the destination to accept the data. The late scheme is often used when the destination has control of the bus (in situations where many 'slaves' wish to communicate with one 'master') and the source must first place a request to drive the data lines. Only once the request is granted, indicated by a transition on the acknowledge wire, can the source drive the data lines. The third, *broad*, scheme is often used as a scheme neutral method of communication. Although the early and late schemes are not compatible they can both receive data from a broad scheme source (assuming the source has control of the data bus).

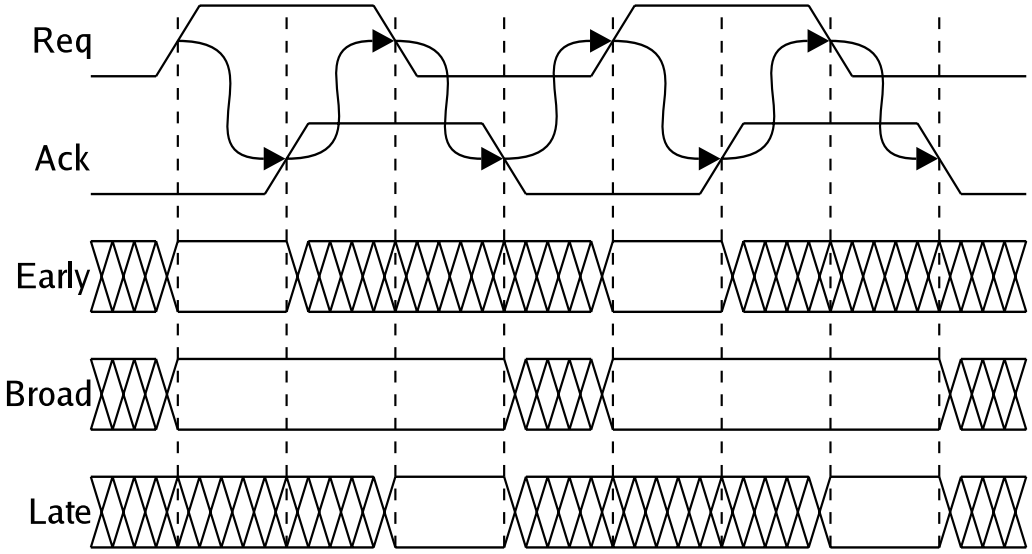

Figure 2.2: Early, Broad and Late Four-phase protocol

2.2 Delay models

Asynchronous circuits are often classified in order of the type of their 'robustness'. More robust circuits need less testing to ensure correct operation both during the design phase and post production.

2.2.1 Delay-Insensitive

The ‘Delay-Insensitive’ (DI) class [20] is the most robust of all delay models. It makes no assumptions on the delay of wires or gates. In this model all transitions have to be acknowledged before transitioning again. This condition stops unseen transitions from occurring. In DI circuits, any transition on an input to a gate must be seen on the output of the gate before a subsequent transition on that input is allowed to happen. This forces some input states or sequences to become illegal. For example OR gates must never go into the state where both inputs are one, as the entry and exit from this state will not been seen on the output of the gate.

Although this model is very robust, no practical circuits are possible due to the heavy restrictions. This does not leave the model as useless as it is often used for communication protocols despite the fact that communicating modules may not delay-insensitive. For example the interaction between the request and acknowledge is delay insensitive as each transition of each wire is acknowledged with a transition of the other wire.

2.2.2 Quasi-Delay-Insensitive

The Quasi-Delay-Insensitive (QDI) model is a compromise to delay-insensitivity with the addition of *isochronic forks* [21]. Isochronic forks allow signals to travel to many destinations and be acknowledged by only one. Isochronic forks are forks in wires where, if the acknowledging target has seen a transition on their end of the fork, then the transition is assumed to have happened on the other ends of the fork too. There are two types of isochronic forks; the asymmetric type only ensures that the signal will reach the acknowledging fork tip before or at the same time as it will reach the other; the symmetric type ensures that both fork tips will be reached at the same time. Symmetrical isochronic forks allow either of the targets to acknowledge the signal. In QDI circuits all forks have to be either isochronic and acknowledged by one of the destinations, or acknowledged by both destinations.

2.2.3 Q^nDI

In the Q^nDI delay model isochronic forks can be extended through gates [22]. The n in Q^nDI represents the number of gates allowed in the extended isochronic forks. These are

usually asymmetric as, with an increasing level of complexity on the two paths, it becomes impossible to ensure symmetric fork behaviour. As the number of gates increases the robustness of circuits drops and requires more rigorous testing to ensure that the timing assumptions are maintained in the fabricated circuit.

2.2.4 Speed Independent

Speed-Independent (SI) [23] circuit design is one of the least robust models as it assumes wires have no delay. This is increasingly difficult to justify with shrinking process feature sizes. Designs manufactured in the latest process technologies have longer wire delays than gate delays. Despite this the SI model is a popular delay model.

2.3 Fundamental asynchronous components

Construction of handshaking asynchronous circuits uses most of the generic synchronous components with the obvious exception of clocked elements (combinatorial gates and transparent latches). There are also a small number of additional elements which have become standard in the implementation of asynchronous circuits.

2.3.1 C-elements

The Muller C-element [23] is a commonly used asynchronous component. It is used to merge and synchronise signals switching its output only when all inputs have reached the same state. Figure 2.3 shows the implementation and symbol of the C-element.

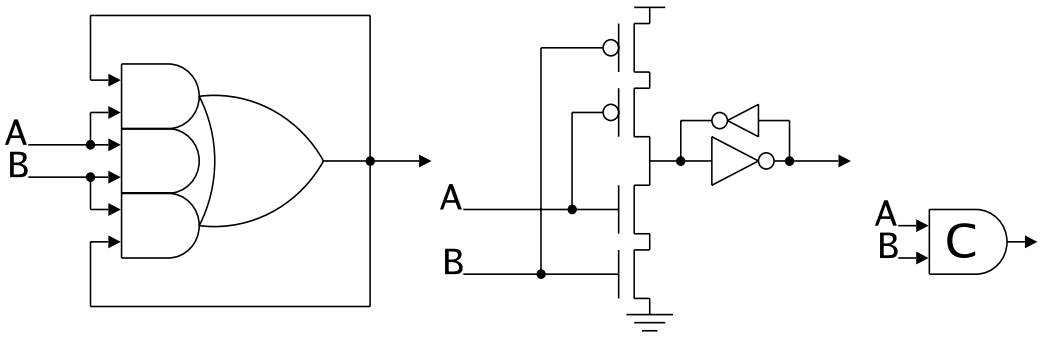

Figure 2.3: Gate and transistor-level design of the C-element and its symbol

Asymmetric C-elements have inputs which affect the operation of the element only when transitioning in one of the directions (shown in figure 2.4). Asymmetric inputs are attached to either the minus (-) or plus (+) strips of the symbol. When transitioning from 0 to 1 the C-element will take into account the common and the asymmetric plus inputs. Similarly when transitioning from one to zero the C-element will take into account the common and the asymmetric minus inputs.

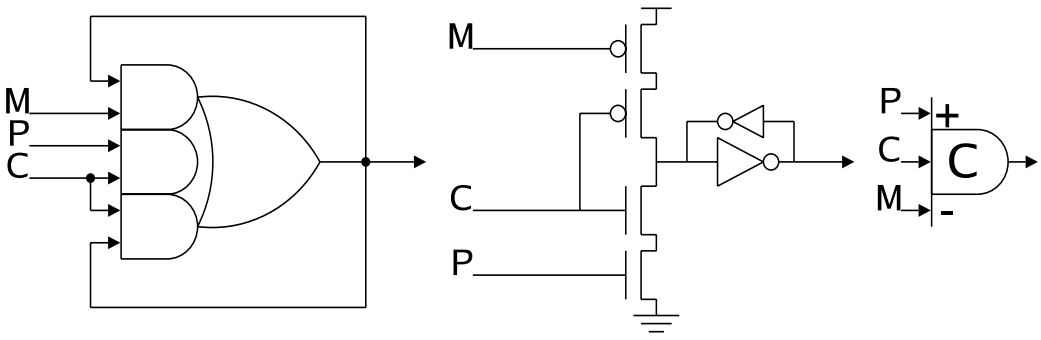

Figure 2.4: Gate and transistor-level designs of an asymmetric C-element and its symbol

2.3.2 Mutex

In asynchronous circuits arbitration is often required in situations where two requests arrive asynchronously desiring access to a shared resource. As the signals can arrive at identical times and only one of the requests can be granted, a hardware element is used to guarantee the exclusivity of the grants.

The *mutex* (mutual-exclusion) element is used to arbitrate between two asynchronously arriving signals. A simple gate level construction would involve two cross coupled NAND gates. Each signal tries to block the other from being granted which in gate level implementations can cause the outputs to become metastable if the signals arrive within a gate delay of each other. This metastability will eventually resolve but in the meantime the outputs of the NAND gates must pass through metastability filters. These keep both outputs low until the metastability has been resolved.

The circuit in figure 2.5 shows a design of a mutex element [24]. The metastability filters comprise a pair of inverters which will drive their outputs high only once there is a

sufficient difference in voltage between the two outputs. This is done by attaching the Vdd (power) signal of each inverter to the signal feeding its counterpart.

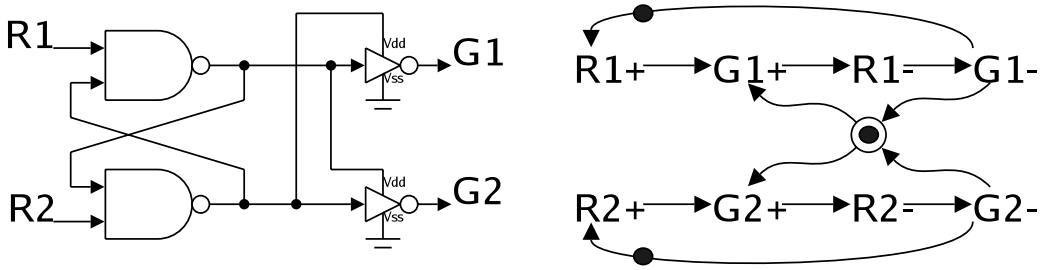

Figure 2.5: Mutex element design and STG

2.4 Formal specification and synthesis

The synthesis of small asynchronous units can be conducted using the Petrify [25] tool which as an input takes a “Signal Transition Graph” (STG). STGs are Petri net based descriptions an example of which can be seen in figure 2.5.

The transition of each net is placed on the graph and these are then connected with arrows signifying which events trigger the transition. The tokens present on some arrows in the figure represent the initial placement which is the state of the graph at reset time. In the figure there also is a ‘place’. A place is a space for a token which allows the token to move to one (and only one) of its outputs. In the mutex STG this allows the formation of mutually exclusive sequences of events. The place allows either the G1+ or G2+ transitions to happen but not both. When the G+, R-, G- sequence has completed a new token is inserted into the place to allow another sequence to begin.

Further details of STGs can be found in the referenced material [25].

Chapter 3: Asynchronous logic

As well as the many asynchronous communication protocols there are also a number of methods to encode the data in the communication channels. Bundled data and dual-rail are two of the most popular and will be discussed later in this chapter. The basis of all these protocols is exemplified in a system known as *control* circuits. Control circuits do not pass data and can only pass empty messages. Control circuits can then be altered to construct either bundled data of dual rail structures.

3.1 Control Circuits

Control circuits are asynchronously communicating networks which do not carry any data. This makes their construction very simple but their use is limited due to their inability to perform computation.

All computing circuits comprise two parts: storage elements (latches and flip-flops) which store data and computing elements which then perform computation. These parts alternate forming pipeline stages. In the case of control circuits no computation is done in the logic stages and these are simply synchronisation points.

3.1.1 Tokens

Control circuits are incapable of passing data but the message handshake signals are present even though the hardware associated with data transfer is not present. These transfers can be thought of as tokens. Tokens can progress from one latch to the next and can be split and re-converge with other tokens in logic stages.

3.1.2 Operation cycle

A stage using a four phase handshake protocol goes through 2 periods during each operation. These are the *set* and the *reset* periods. The set period encompasses the time where the inputs to the stage are arriving as the acknowledge is low. The reset period of a stage has the acknowledge high and the inputs start being removed.

Tokens need to be separated to stop their merging and becoming a single token. Between the set period of each token a reset period is inserted. Each stage has to reset completely before another set can begin to ensure that data from the previous cycle does not effect the computation in the current cycle.

3.1.3 Latches

To pass tokens from one stage to another asynchronous latches are used to store the token while it is progressing. Each latch handshakes the transaction of its token with the next stage. These transactions are communicated across the request and acknowledge wires as described in section 1.3.

Half Latch

The ‘half-latch’ is a simple latch design and is shown in figure 3.1. With a request-acknowledge interface on both the input and the output, the latch forwards the requests while acknowledging the input. The request out and the acknowledge in signals are only released once the input stage has released its request and the output has acknowledged.

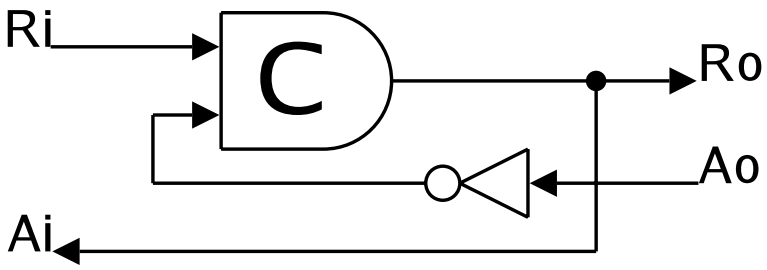

Figure 3.1: Half latch design

This simple behaviour enables the latch to allow the input stage to drop to the reset phase (releasing the request) while the output stage enters the set phase (request becomes asserted). This is demonstrated in figure 3.2 where each latch separates a stage holding data from a stage hosting a spacer. The half cycle separation enables the half-latch to store half a token. Two half-latches are needed to store a single token (allowing the input and output to the two latch pipeline to be sequenced as seen in the figure) as each latch can only separate the phase of the input from the output by half a cycle. The front latch separates the resetting phase stage in front of it from the setting phase behind it. The second latch separates the setting phase in front of it from the resetting phase behind it. The leading edge and trailing edge of the token can be separated by many latches, allowing the token to stretch and shrink depending on the progress of the leading and trailing edges. The sequencing of the latch forces the separation of the edges by at least one logic stage. This ensures that tokens are kept separated (do not merge) and are kept in at least one stage (do not die).

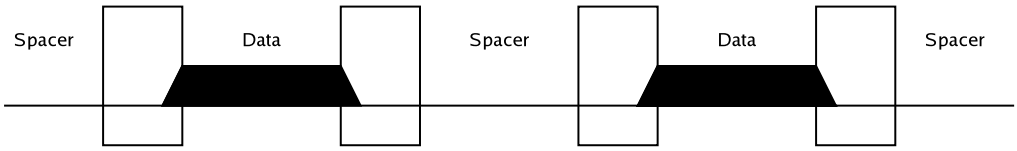

Figure 3.2: Half latch pipeline token capacity

Semi and Fully Decoupled Latches

The half latch is only able to separate the data stage from a stage holding a spacer but some latch designs are able to ‘decouple’ the two stages by more than one phase difference. Figure 3.3 shows three levels of decoupling latches could be capable of. The first level is ‘no decoupling’ where the output of the latch is the same state as the input. Although this behaviour is possible using a set of wires with no logic, it is important that any latch design maintains this ability. The half decoupling behaviour allows regions with data to be separated from regions holding a spacer. This is present in the half latch.

The third level of decoupling presented is the full decoupling where a latch separates stages with the same state by fully enclosing a token or a spacer. Semi-decoupled latches

[16] can enclose either a spacer or a token while fully-decoupled latches can perform both actions.

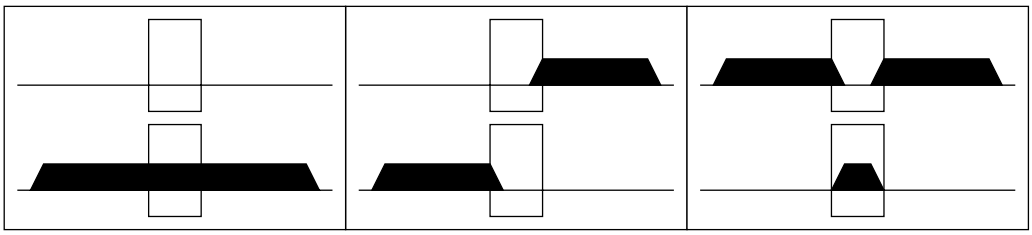

Figure 3.3: Levels of latch decoupling

3.1.4 Split

The transmission of a token to two or more latches requires the source latch to interface correctly with more than one destination. While the request signal needs to be forked to all destinations, the acknowledge signals from all destinations need to be combined to generate the single acknowledge signal the source latch expects. The introduction of forks in the request distribution requires that each transition of the request be acknowledged by all destinations. Only after all destinations have acknowledged may the request transition again.

A C-element can be used to combine all the acknowledge signals. It will ensure that all destinations have acknowledged before forwarding the acknowledge to the source latch (see figure 3.4). As the C-element is symmetrical to up and down transitions, it will wait for all destinations to release the acknowledge before releasing its output. This ensures that all destinations the request leads to have observed the signal before the request transitions again.

3.1.5 Converge

Synchronisation of tokens is achieved by converging two or more pipelines into one. The output latch will receive a request only once all input latches have presented their request. The output latch has to acknowledge all input latches by sending each latch its acknowledge signal. As described above both the rising and falling transitions on forked

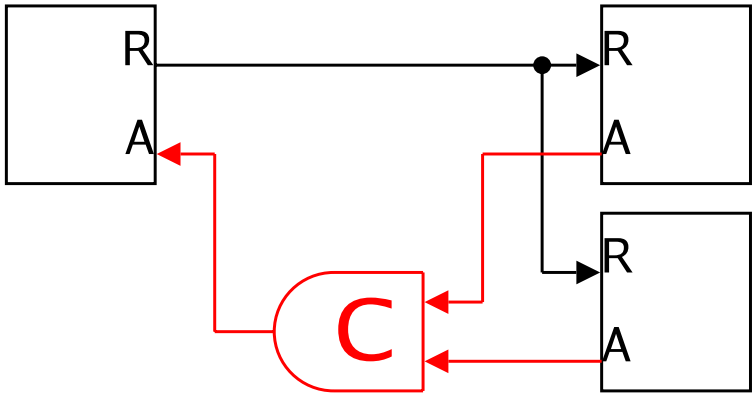

Figure 3.4: Split example

signals have to be acknowledged. This again is done using a C-element to gather the requests of all inputs as illustrated in figure 3.5.

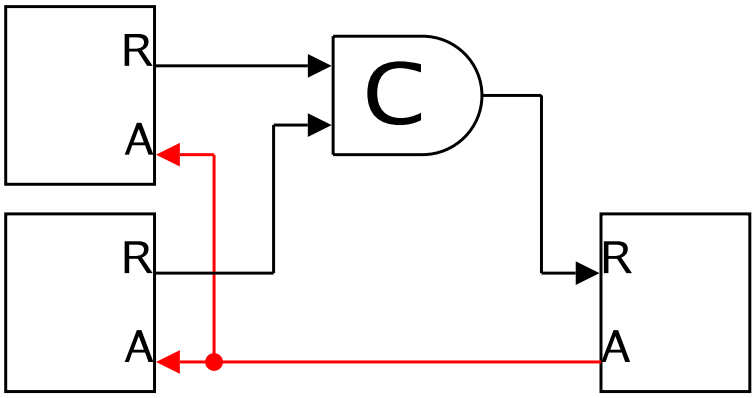

Figure 3.5: Converge example

3.1.6 Complex constructions

A single pipeline stage can both split and converge many signals. The rules specified above are simply applied to a single block of combinatorial logic surrounded by latches (stage).

There are two approaches to applying both split and converge rules in a single stage. These are called *grouping* and *separating*. Of the two methods, grouping generally

generates circuits with lower power consumption and area, while separating generates circuits with fewer synchronisations (and is thus faster).

Grouping

Grouping treats all inputs and outputs of a pipeline stage equally and ignores their dependencies. This generates one request signal which is shared between all outputs and one acknowledge signal which is shared between all inputs. The common request signal is formed by gathering all request signals using a C-element. Likewise for the acknowledge signal. This is demonstrated in figure 3.6.

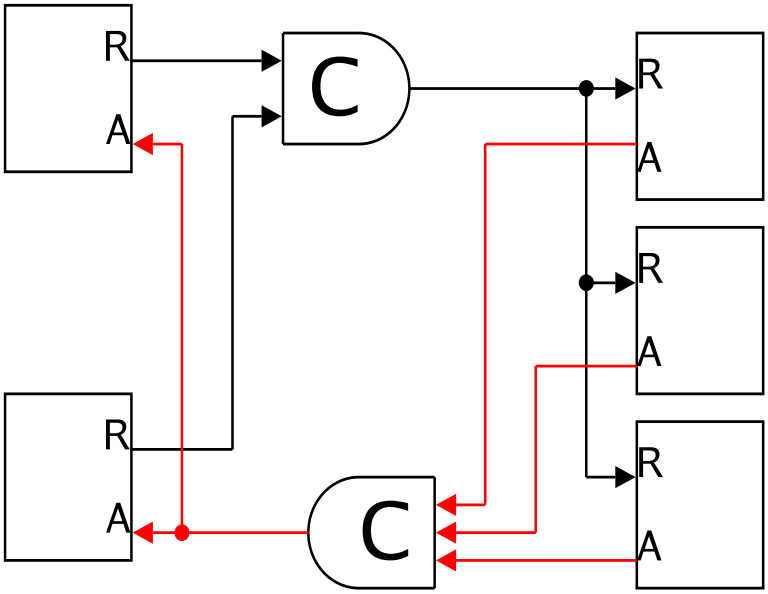

Figure 3.6: Grouping example

As all inputs and outputs are driven by the same request and acknowledge wire they become synchronised. This can slow the circuit down as it is unable to exploit signalling dependencies.

Separating

The Separating method treats each latch individually and connects it only to co-dependent latches. The request of each input latch is connected only through a gathering C-elements to all latches that depend on its data. The input latch's acknowledge signal is generated by C-elements which gather the acknowledges of the dependent output latches. This technique is illustrated in figure 3.7.

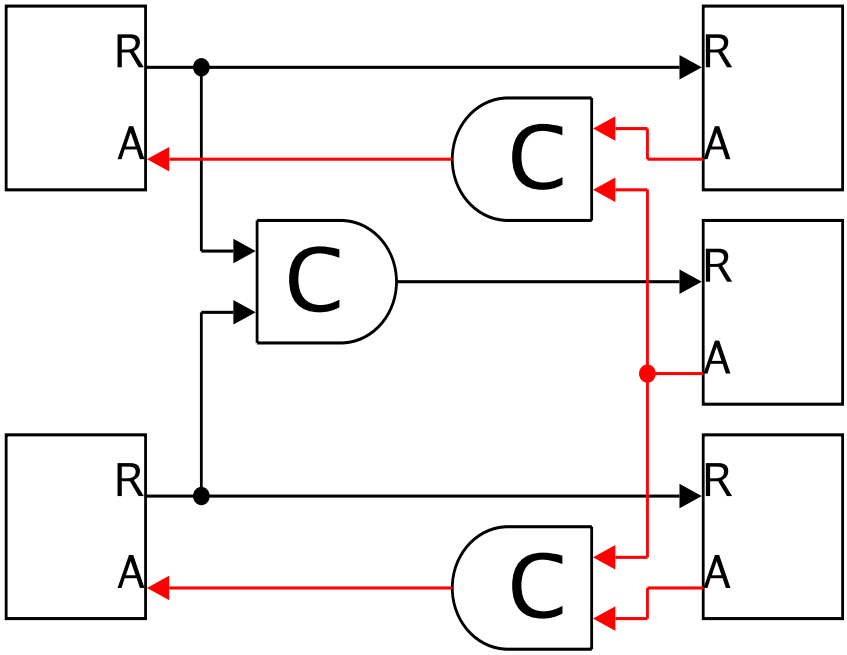

Figure 3.7: Separating example

This approach only synchronises latches where necessary and allows the acknowledge of tokens earlier than in the grouped version. The need for each latch to have two separate C-elements to gather requests and acknowledgements creates a larger circuit. Fortunately many optimisations can be carried out to reduce this impact: In cases where an input latch contributes to only one output latch, the C-element will have to gather only one signal and can be optimised away to a wire. Also, in cases where many latches at a stage have the same set of dependants they can use a single C-element to generate a common request or acknowledge signal.

3.2 Bundled data

The bundled data system [18] allows communication of data in systems with token flow. This also allows computation rather than just synchronisation. With the control part of the system already described, the generation of full computing circuits requires only the data communication and computation parts of the system.

Tokens are primarily designed as symbols representing data flow and systems such as control circuits can be easily adapted to carry data. Control circuits announce the presence of and negotiate the progress of tokens but data latching and computation requires additional components.

3.2.1 Latches

To create bundled data latch designs, the control circuit designs are adapted by connecting a data latch to the component. The latch enable signal is taken from one of the wires available in the design or a logical operation of several signals. Depending on the wire chosen the latch will have a different data validity period (early, broad or late explained in “Four-phase signalling” on page 22).

A bundled data half latch is shown in figure 3.8. All three schemes for generating the latch enable signal are presented in the figure but in a normal design only one is required. Connecting the latch enable signal directly to the Ro wire will give an early data validity and connecting it to the inverse of the Ro signal will give the late data validity where the enable being high causes the latch to become opaque. The broad validity can be achieved by latching on the OR of the Ro and Ao signals.

In the broad data validity scheme, the addition of the delay element, called the ‘bundling constraint’ delay, ensures that the data has been presented on the output before request signal is transmitted.

3.2.2 Logic

Logic in bundled data systems is constructed in much the same manner as in synchronous circuits. Because the logic has delay, the request signal must be also slowed down in order

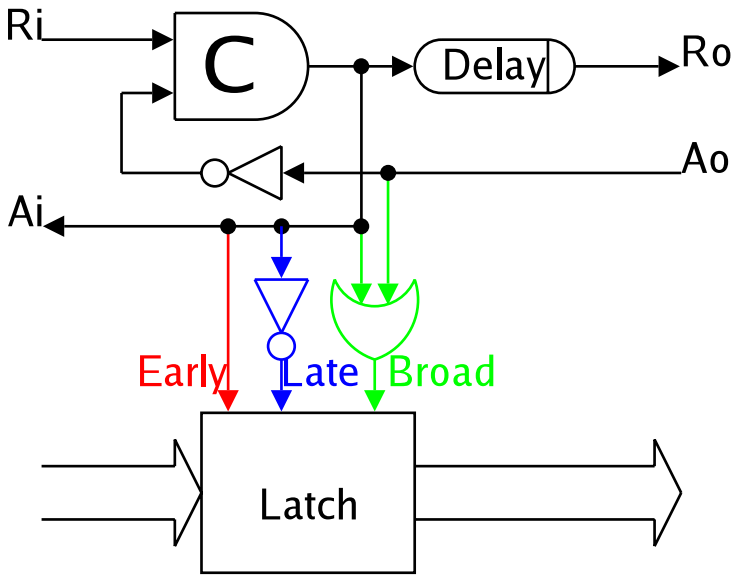

Figure 3.8: Bundled data half latch

to ensure the output of the data processing logic has been calculated before being latched to the next stage. This delay on the request signal is often referred to as a *matched delay* as it 'matches' the delay of the logic. The delay of the logic must be shorter than the delay along the request line otherwise insufficient time would be allocated for the logic operation.

As the delay is needed only on the rising transition, it can be constructed using an asymmetric delay element. Such elements delay only one kind of transition (either the rising or the falling) and leave the other transition with minimal delay.

The request and acknowledge gathering is constructed the same way as in the control circuits (shown in section 3.1.6). The delay element can be placed either before or after the request gathering C-element. If placed before the gathering C-element, the delay element has to be replicated on every input into the C-element. This consumes more space and power but does allow a more closely matched delay to be constructed. This is because the effect of each input may take a different period of time to reach the output. Placing the delay on the output of the C-element would force the delay of the stage to be the same irrespective of the order of arrival of inputs. As it is often the case that the last input to arrive has a short path to affect the output, the stage not completing until the worst case

delay has passed can have a negative effect on the performance. An example of such a technique is demonstrated in figure 3.9.

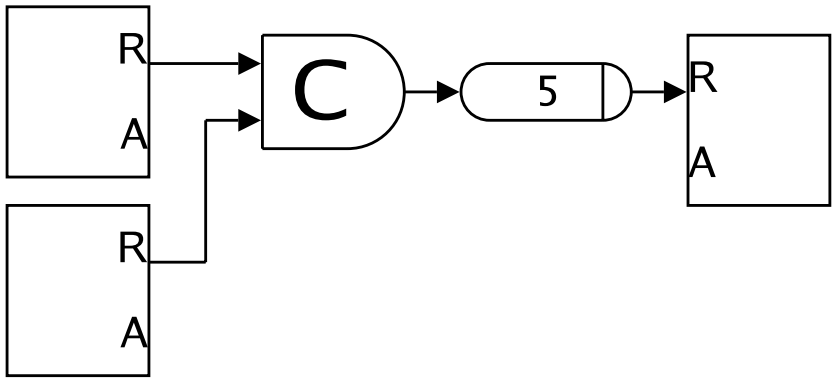

Figure 3.9: Common delay

A compromise between the two approaches places the shortest input to output delay on the output of the gathering C-element. The inputs which require a longer delay have additional delay elements placed on their respective C-element inputs. This approach gives both the accurate arrival time based delay along with reduced area and power consumption.

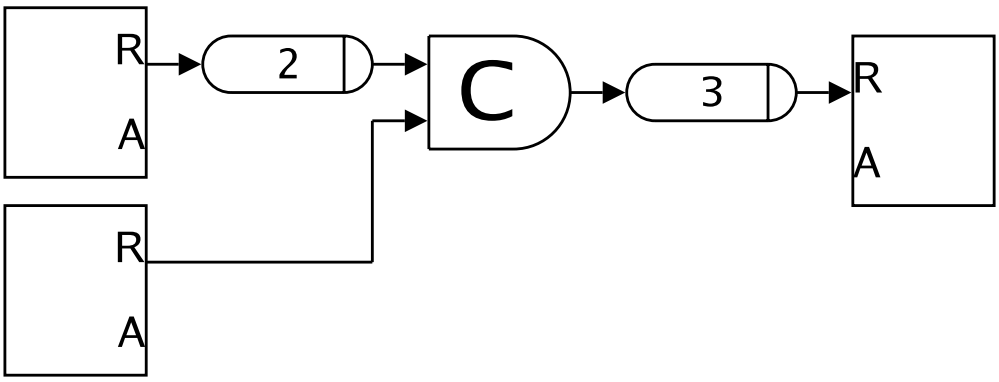

Figure 3.10: Separated delays

3.3 Dual-Rail (DIMS)

Invented by D. E. Muller, the *DIMS* [23] (Delay Insensitive Minterm Synthesis) system is an asynchronous design methodology making the least possible timing assumptions. Assuming only the QDI delay model the generated designs need little, if any, time closure testing. The basis for DIMS is the use of a one-hot code on a set of wires to represent data. The most common number of wires used in sets for use in DIMS logic is two, where one wire set represents one bit of information. Although other codes are possible and useful, such as the 1-of-4 codes, only 1-of-2 codes will be examined here.

To enable the QDI operation of the system the request path is duplicated. Asserting one of the request wires transmits one bit of information the value of which is dependent on which request was activated. The data is acknowledged and the active request wire must be de-asserted before the acknowledge is released. As the request signal is encoded on the data, the dual-rail data encoding can only be applied using the early data validity

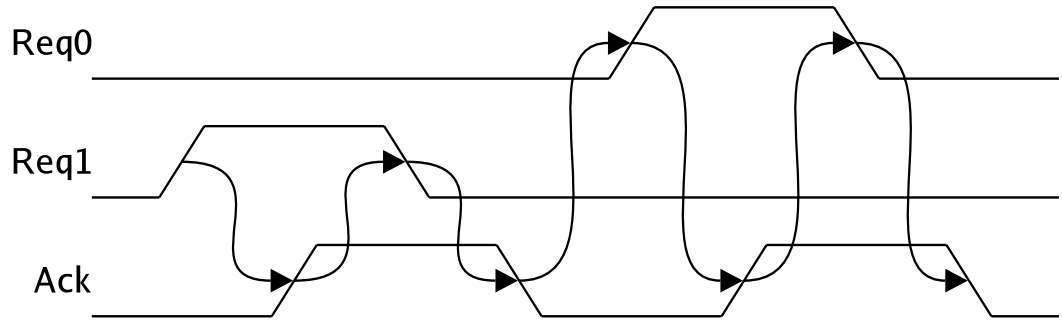

Figure 3.11: Dual rail protocol

3.3.1 Latches

Dual rail latches are composed by duplicating the request path in the control circuit latch designs. The new request signal names are usually suffixed with a 0 or 1 distinguishing the value transmitted by each signal.

Half latch

The half latch design described in section 3.1.3 is taken and its request path is duplicated. This duplicates the C-elements in that path giving a C-element for each bit. In the control circuit the ‘request out’ signal also drives the ‘acknowledge in’. In the dual-rail version there are two requests out so they have to be merged in order to create an acknowledge in. In this case as only one of the request wires will be active at a time and the acknowledge should be activated once a request out is generated (irrespective of which one), the signals will be gathered using an OR gate.

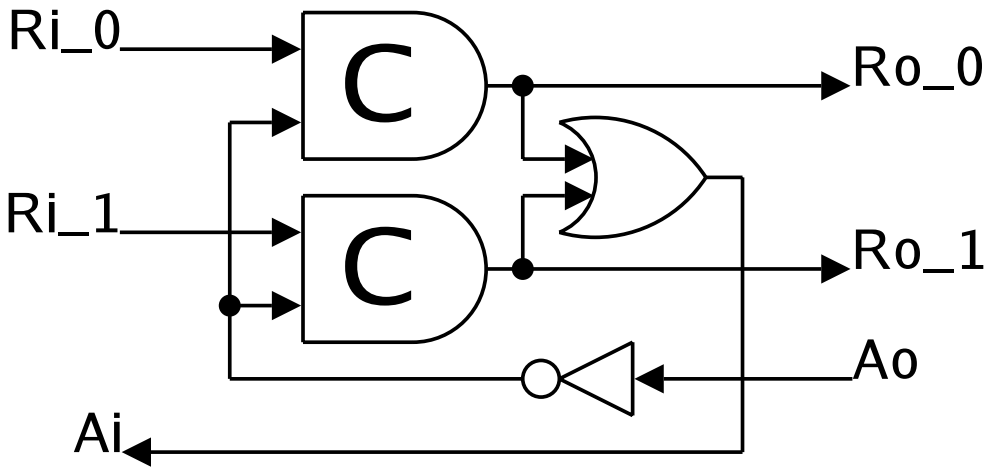

Figure 3.12: Dual-rail half latch

3.3.2 Logic

Logic in a DIMS system has to preserve the strict sequencing assumed by the latches. DIMS logic performs two tasks: it gathers the request signals of all input latches that affect the result and performs the logical operation. The output of a DIMS gate must not generate a result until all inputs are present and not release the result until all inputs have been released.

The standard construction of DIMS gates involves generating a full set of all minterms from the inputs. These minterms are generated with C-elements and cover the full set of legal input states. In the example of the two input gate, there are four minterms, one for

each of the possible input states. Each output wire then takes a selection of these inputs and generates the output when any one of them is activated. Each minterm must activate exactly one output as ignoring the minterm will stop the gate from producing an output and attaching it to more than one output will generate illegal output states.

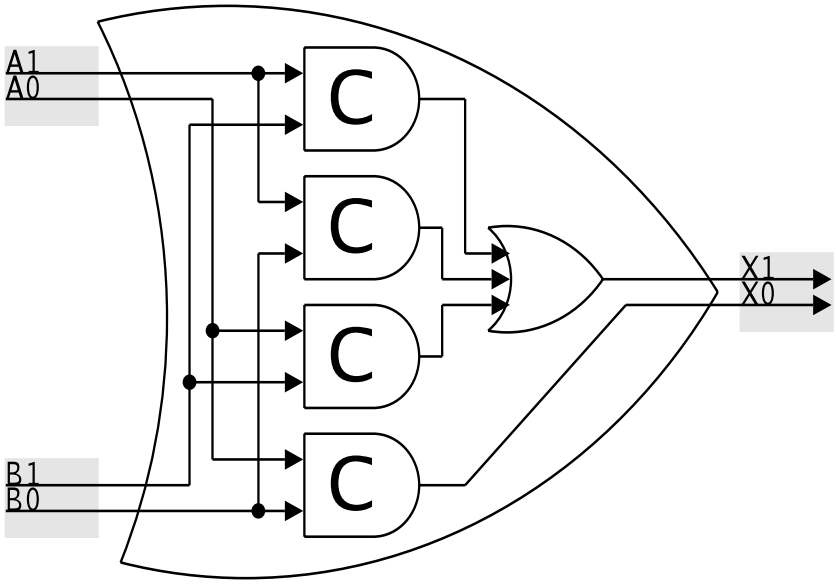

Figure 3.13: 2 input DIMS OR gate

The construction of larger gates becomes more problematic as the number of minterms increases exponentially with the number of inputs (2^X) and the number of inputs to each minterm C-element matches the number of dual rail inputs. Figure 3.14 shows a four input DIMS gate. Here there are 16 three input C-elements and a 15 input OR gate gathering the results. This explodes the eight transistor synchronous equivalent gate into a 264 transistor DIMS implementation.

3.3.3 Bit-level pipelining

One of the useful aspects of dual-rail logic is its intrinsic ease of creating *bit-level pipelining* circuits (demonstrated in systems such as *Phased Logic* [26]). In contrast with the bundled data circuits, which usually use one latch controller to latch multiple bits of data, dual-rail latches capture one bit of data each. Although bit-level pipelining is

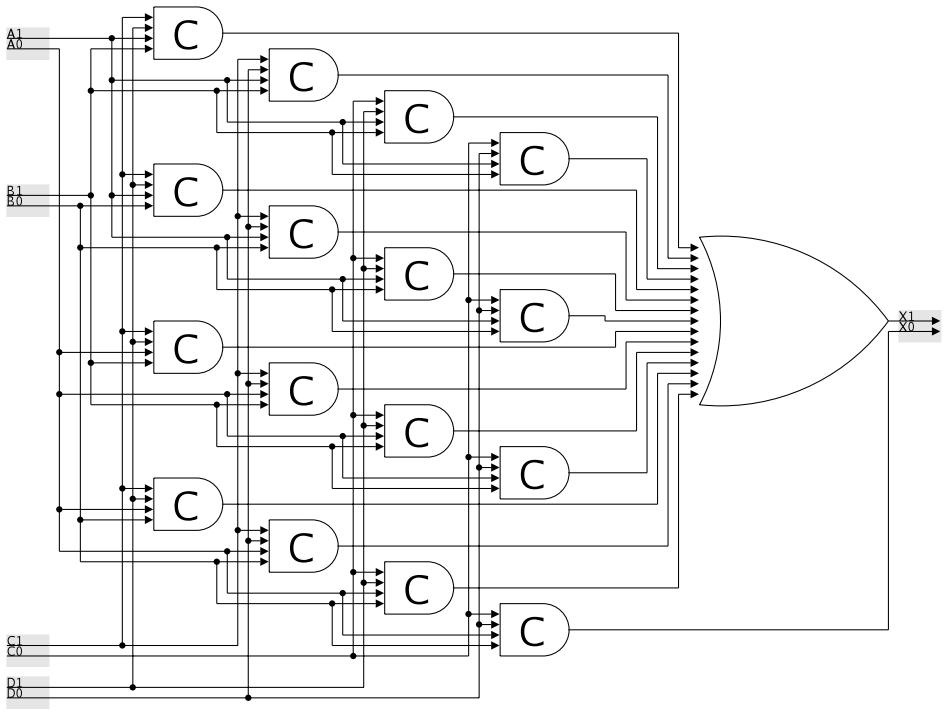

Figure 3.14: 4 input DIMS OR gate

possible in the bundled data designs the increased complexity of having a matched delay for each latch makes the system large and slow.

Four-phase dual-rail designs present their completion at a bit level (and not a stage level like common bundled data designs). This allows parts of the result to flow to the next stage in the pipeline and be operated on before the complete result has been generated. This behaviour makes designs, which use carry ripple adders frequently, faster as the bottom parts of the result which are generated first are used by adders in subsequent stages. This transfer of data from one stage to another in an ordered sequence (bottom bits first then going up) is called *skewed wavefront pipelining*. The skewed wavefront allows stages to disguise their high latency by ordering the inputs to come at the exact time they are needed and generating the outputs in the same order. This gives ripple carry adders a latency of a single bit full adder rather than the critical path of the full carry chain [27]. The practicality of skewed wavefront pipelines is reduced once the full value needs to be de-skewed for operations such as memory accesses.

Data travelling through dual-rail bit level pipelines not only becomes skewed when passing through adders but can also pass through stages in an unordered manner should the complexity of generating some bits be higher than others.

3.3.4 Vertical pipelining

Although the data passing through a ripple carry adder can generate some results before others, the full stage must complete before the adder can work on the next data inputs. This effectively de-skews the data as the bottom bits of the next set of values cannot enter a adder stage while the stage is completing. This can be avoided by pipelining the adder into a a series of smaller blocks separated from each other by latches. Vertical pipelining allows a small section of the adder to complete while the rest of the adder is only starting to compute. This not only reduces the reset period by allowing the different segments to complete in parallel, but also frees the bottom segments to start computing on the next set of data. Vertical pipelining also increases the overall pipelining of the system which is important to stop data from stalling due to blocking (explained in “Properties of asynchronous pipelines” on page 15). As well as adders, other constructions have been made using the vertical pipelining style such as register banks [28].

3.3.5 Empty latches

Blocking is a big problem in four phase circuits as the circuit often spends as much time resetting as working on the data. This forces each stage which feeds back to itself to waste half the time resetting. In the vertical pipelining example above, the insertion of latches to break up the adder into a set of smaller segments allows parts of the stage to compute while other parts reset. The latches are *empty latches* as they do not hold a token at reset time. The insertion of empty latches is necessary for building fast four-phase circuits as the stages need to be split into two or more balanced segments to create a pipeline with few data stalls.

Chapter 4: Early output

Both synchronous and asynchronous bundled data designs do not take into account data operated on when forming the timing of the operation. In synchronous designs the delay used for each operation is fixed (in synchronous circuits all inputs arrive at the same time) and in bundled data it is fixed to the arrival of the last input and is irrespective of the complexity of the task being carried out. This forces each stage to assume a worst case delay before completing. DIMS logic elements, due to their restriction that all inputs must be present before generating an output, consume a worst case delay during each execution.

To capture the performance potential of average case performance fully, the timing must be data dependent and the generation of results must be allowed before all inputs are present [24][29][30][31][32][33][34][35][36][37].

4.1 Early Output Theory

Many functions can generate the result based on the data from only a subset of inputs but often it is impossible to determine which inputs must be supplied to a stage to yield a result. In push channel communication the data is supplied even if it is not necessary to the unit it is supplied to. Speculative supply of data to a unit is often unavoidable as its necessity often cannot be easily determined. The synchronisation between the generation of the output and the late arriving data which is not needed to complete the operation has a negative effect on the performance of a system. Generating a function's output irrespective of the arrival of data on all inputs can allow faster operation but still generate the correct result.

4.1.1 Determining input necessity

The input set gathered to trigger the output generation can be limited to just the necessary minimum. This set can be determined by observing the presence of inputs and their data. The necessity of data inputs can be determined in one of three manners: data independent, data dependent and data co-dependent.

Data independent

The data independent method relies only on the presence of other inputs and not their data. Once an input threshold has been met the stage can complete. This method is only useful in redundant computing where the result can always be generated even in the absence of at least one input. An example of such a scheme would be in having two implementations of a functional unit, each performing the same task with the same data but using a different algorithm. A data independent stage would be useful to pass out the first result generated by one of the units and thus achieve the best performance through using each unit for operations better suited to it.

Data dependent

A multiplexer always requires just one of its data inputs to arrive (as well as the select signal) to generate an output. The desired input is encoded in the select signal which is observed to determine the necessary input set. This data dependency uses the data of an always necessary set of inputs. The other inputs are gathered but their necessity can be determined before their arrival.

Often, in situations like this, the data channels are implemented using pull channels and only the desired input is fetched. This optimisation allows the data communication to be at the request of the destination rather than pre-emptively sent by the source. The removal of unnecessary data transfers can allow a reduction in power consumption.

Data co-dependent

The data dependent method of determining the required input set assumes the presence of a necessary input set. In most computing stages each input can sometimes be unnecessary and sometimes provide data to disqualify other inputs. The inputs to a stage which takes two 1 bit inputs and passes them through an OR gate are co-dependent. This means that (depending on the data) each input can obviate the necessity for the other (in cases where the first input to arrive is a one).

4.1.2 Early output cases

The situation where a stage has received sufficient data to generate a result while some inputs are still to arrive is called an *early output* case [38]. Early output cases allow the output generation in a circuit to synchronise only with the necessary subset of inputs and not with the last input to arrive.

To demonstrate the approach, each of the three asynchronous circuit styles described in the previous chapter (control circuits, bundled data and dual rail) will be adapted to generate early outputs.

4.2 Control Circuits

As control circuits do not pass data, the generation of early outputs must be data independent. Normally a token is passed on the output once all inputs in the latch's input set have all presented tokens to the stage. In the early output version, only a threshold of inputs has to be present to output a token. In this function a rising transition of an input cannot cause the output to fall, and a falling transition on the input cannot cause the output to rise.

The threshold function gives the earliest possible time an output token can be generated but does not guarantee that all inputs are ready to be acknowledged (or have the acknowledge released). To ensure that all inputs are ready for a transition on the acknowledge signal each latch outputs its validity. A validity transition signals the latch is ready to accept an acknowledge transition. To ensure the acknowledge does not reach any input latches before they are ready to accept it, the validity signals of all inputs are

gathered and combined with the acknowledge of the output latch. This ‘guarding logic’ replaces the strong indication that existed in the standard system. With guarding logic, each latch is protected from receiving a transition on its acknowledge until it has signalled it is ready to do so (by transitioning the valid signal).

The early output protocol can be seen in figure 4.1. When compared to the diagram of “Early, Broad and Late Four-phase protocol” on page 23, the valid signal now does the work of the request signal in the sequencing of the transitions. The request signal transitions in parallel with the validity, it is guaranteed to be low once the validity is released and high by the time validity is asserted.

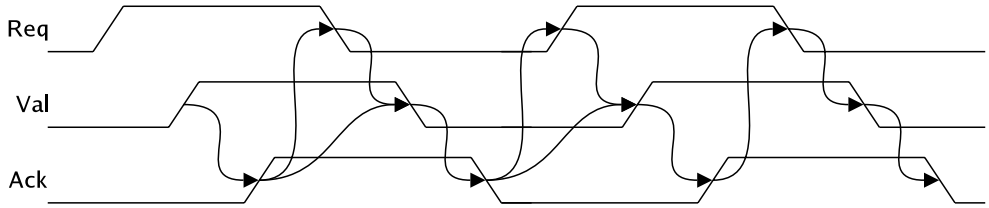

Figure 4.1: Early output protocol

4.2.1 Latches

The addition of the validity output in the latch design can be accomplished by connecting it directly to the request out signal. In most designs, this is the most logical method of generating the validity and the separation of these signals on the outside of the latch might not seem justified. The separation becomes useful in designs where the latch can take advantage of the different uses of the two signals, as demonstrated in “Anti-Tokens” on page 63.

The second change to the standard latch designs is the sequencing of the ‘validity in’ signal with the acknowledge on the input. The acknowledge may only transition to match the state of the validity in signal. This can be enforced by inserting a *guarding C-element* [39]. This can be seen in figure 4.2 where the acknowledge signal, which leaves the latch in the “Half latch design” on page 29, becomes synchronised with the validity in signal (Vi) to generate a guarded acknowledge.

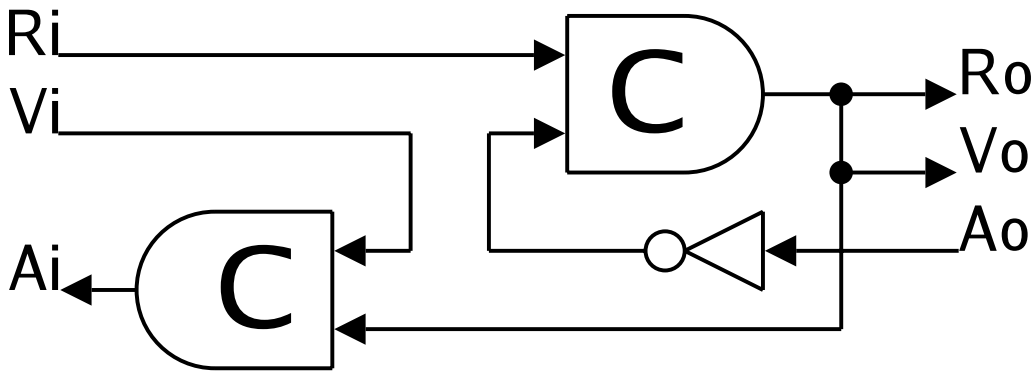

Figure 4.2: Early output latch

4.2.2 Logic

The guarding C-element is the connection point between the validity gathering and the acknowledge gathering C-element trees. These two trees are used to protect latches from receiving transitions on the acknowledge signal until they are ready to do so, despite the unbunding of the request and validity signals.

The acknowledge tree has existed in previous control circuit designs and the only difference in early output systems is the separation of the validity and the request signals. The validity signal now takes the place of the request signal by being gathered in a tree to form a single signal at the output latch stating the validity of all inputs used to generate the particular output. The request path now does not need to signify the state of all inputs and can concentrate on the generation of the data (or in this case the time of the threshold being met). This is done in a separate threshold function allowing the control circuit to fire when a subset of inputs has arrived yet still wait for the remaining inputs to arrive before acknowledging them.

4.2.3 Advanced Latch Designs

Half latches wait for the trailing edge of the request signal on the input side before releasing the request out. This unnecessary wait can be avoided by releasing the request out as soon as it is acknowledged by the output stage rather than also waiting for the input to be release. The latch must still keep the acknowledge high until the request on its input has finally dropped.

This action introduces an additional trailing edge into a token and splits the token into two as demonstrated in figure 4.3. The figure shows a token which stretches through the latch (part 1 in the figure where both the input and output requests are high). The latch (represented by the rectangle), once it has received an acknowledge from the stage it feeds, can drop its request output early (before the input request has been released). The front part of the token gets a trailing edge attached to it and the back part of the token gets a front edge which is locked to the latch and will not progress (part 2). This will allow the front token to progress with a new trailing edge, generating a smaller token which occupies fewer stages. The back token's trailing edge will eventually catch up to the latch where the rising edge is held and disappear (part 4).

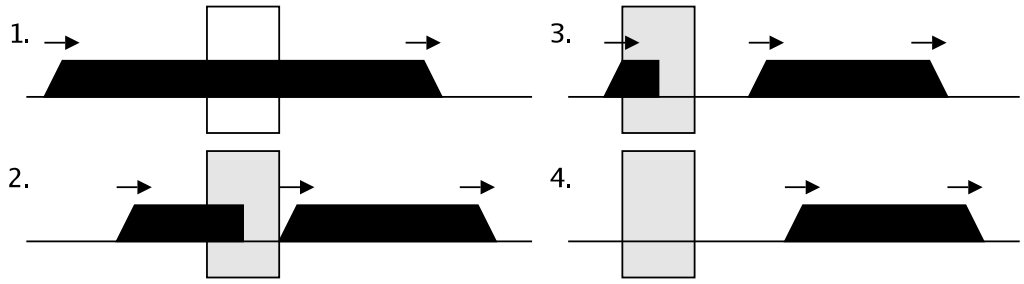

Figure 4.3: Early-drop latch token split

Although the latch is capable of separating two data stages using a spacer because the two stages hold the same token (stretched and split into two) it does not increase the level of decoupling of the latch. The latch is still only capable of ever storing half a token.

Figure 4.4 shows the design of an *early-drop* latch [39] which has the early drop behaviour described above. The latch is based on the half latch design and keeps the C-element along with the inversion.

An AND gate is added in the path from the original C-element to the Ro output. Normally the gate's output will reflect the data from the original C-element, but when the second input of this gate has dropped, the output is forced low. This second input is driven by the inverted Ao signal. This forces the output low as soon as the acknowledge arrives but the validity signal remains high to stop additional transitions on the acknowledge signal. Only

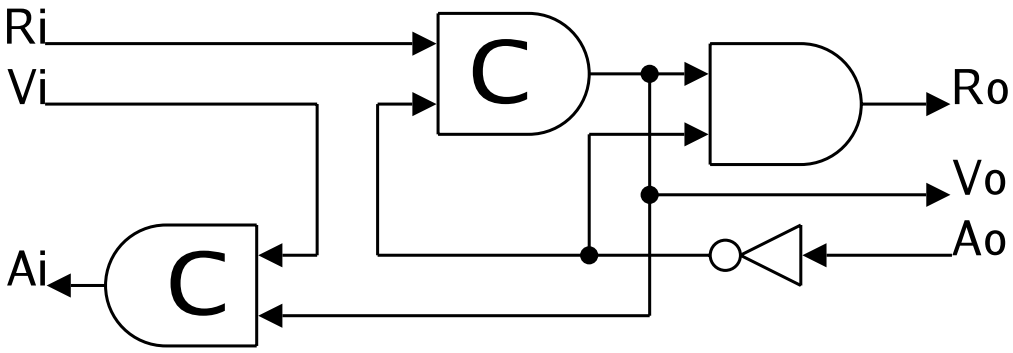

Figure 4.4: Early-drop latch design

once the request in the input side of the latch has been removed can the validity be released.

The early output latch exploits the sequencing between the validity and the acknowledge signals to release the request but not trigger another computation cycle.

4.3 Bundled Data

An early output version of bundled data systems can be implemented in the same way as the control circuits. The generation of results can be controlled by both the arrival of tokens and their data. Inputs can be data dependent, co-dependent or data independent, giving the maximum flexibility.

4.3.1 Latches

Bundled data latches as in the control circuits are constructed by forking the request out (Ro) wire to the valid out (Vo) output and inserting the guarding C-element.

4.3.2 Logic

The generation of an output can be dependent on the presence of an input or its data. These condition signals are gathered to create a request signal.

A multiplexer example demonstrates both data dependent and co-dependent input sets. The three inputs to the multiplexer in this example come from three different sources. A

and B are data inputs, of which one will be passed to the output conditional on the value of input S. One early output case is covered by the presence of input A and S (valids are high) and the data of input S being zero. Another early output case is covered by the presence of input B and S and the data of input S being one. These two cases also cover the ‘all present’ case. The all present case is normally needed to generate an output once all inputs have arrived. In this case the early output cases cover the all present case so there is no need to add it to the request generating logic.

Figure 4.5 shows the early output generation and the validity gathering for the multiplexer example. Normally, in the request generation logic, there would also be delay elements to match the delay of the logic but in this case the delay of the logic is equal to the delay of the early output request generation and no additional delay is necessary.

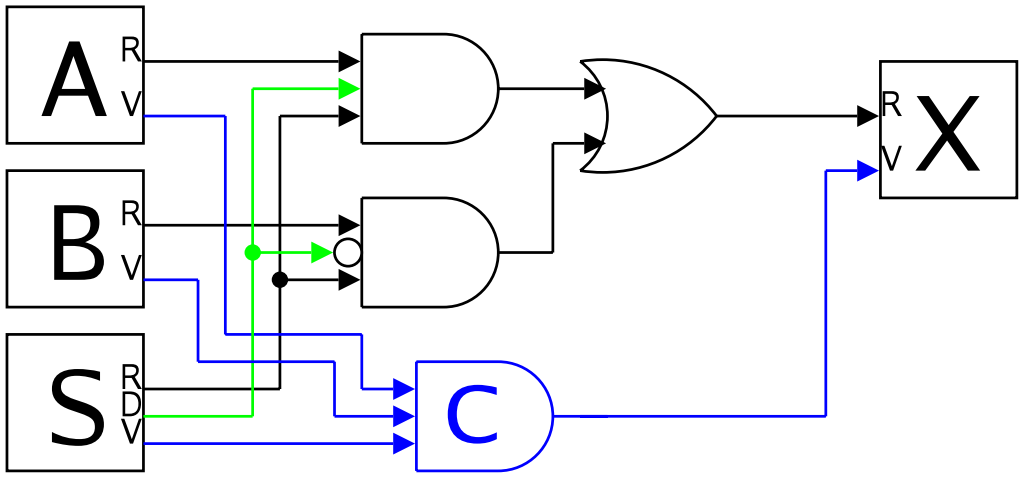

Figure 4.5: Early output function of a multiplexer

The data part of the function must be combined with the request from the latch the data came from with an AND gate. This ensures data inputs have no effect on the function until they are stable. Attaching the data to additional gates before passing it through an AND gate with its request signal causes races and extended wire forks. Such an arrangement is smaller and easier to implement than forcing all data signals to pass through an AND gate with their validity but additional timing assumptions must be upheld to ensure correct operation. An example of this strategy can be seen in an early output case in the multiplexer example which is not covered by the rules stated above.

The two early output cases stated before cover most likely combinations to arise but a third early output case is possible (which will be shown). The first two cases are data dependent. This makes the S input necessary in all output conditions. The third case relies only on the presence of input A and B and this causes the input combination to become co-dependent (as each pair of inputs could remove the requirement for the last input).

When A and B are present and they are equal the S input is irrelevant to the result. The result can then be generated without the need to wait for the S input. There are two methods to construct an equality comparator for use by the request generator. The first is to pass the inputs directly into a standard equality comparator (a row of XNOR gates each taking a bit of the two inputs and an AND gate collecting all the XNOR outputs) and then AND the generated equality signal with the validities of the two data inputs. This assumes that by the time the request signal has gone up, the comparator has completed its operation and presents the result for the current set of inputs. This is rarely the case and for this method to work a delay element would have to be placed into the delay of the request signal. The alternative to this approach is to pass all data inputs used in the early output function through AND gates (with the validity signal of the input) before any logical operations are conducted on them.

4.4 Dual-Rail

Implementation of the early output method is more suited to dual rail systems where the validity of the result is intrinsically encoded in the data. The implementation of dual-rail early output circuits can be achieved in a variety of delay models. The safest of these models is QDI which offers safety matching that of DIMS along with forward propagation speed of bundled data/synchronous designs (examined in sections 4.11 and 4.12). Less robust methods use Q^nDI models which reduce the reset times, by not observing the transitions of all wires in the system, and allow higher throughput. All methods described use the same set of early output latches.

4.4.1 Latches

In the control circuit and bundled-data latch designs, the generation of the validity signal was done by forking the request out wire. In dual-rail circuits the request is separated into

a request 0 and a request 1. As no single signal exists, the validity signal must be generated when one of the requests becomes active. This can be done by gathering then using an OR gate.

Half Latch

In a half latch, the addition of an OR gate to generate the validity signal can be avoided as this signal is already available. Originally generated to drive the ‘Acknowledge In’ output, the OR gate also generates the validity output. The reuse of this gate causes the overhead of the early output version of the half latch to be a single (guarding) C-element.

Early-Drop Latch

The dual rail early-drop latch has no OR gate gathering the request outputs and so the early output version must add that additional component.

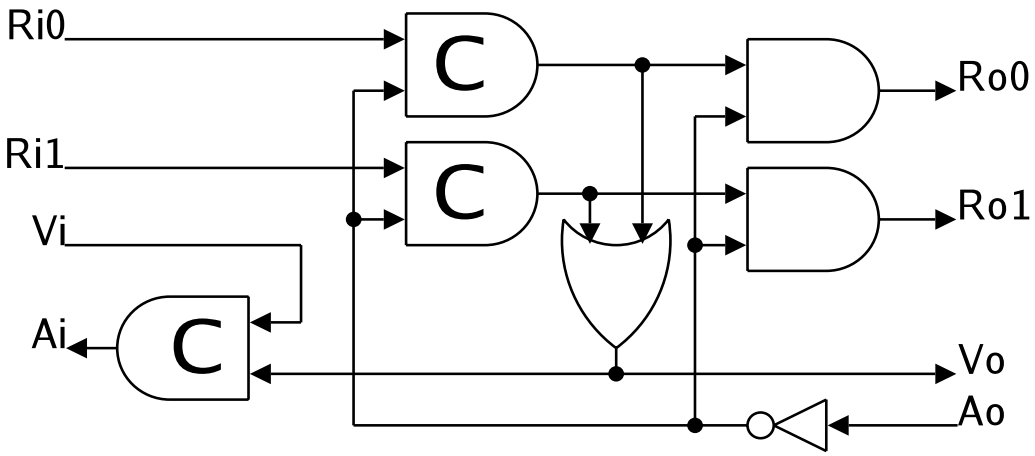

Figure 4.6: Dual rail early-drop latch

In the early-drop latch design, the OR gate gathering the requests before the resetting AND gates is used to generate both the validity and the acknowledge signals.

4.4.2 Logic

Early output dual rail logic has fewer restrictions than the DIMS approach. The output of logic gates or units must remain in the null state until enough valid inputs have arrived to determine the correct output. The units do not have to generate an output in every early output case but must generate an output once all inputs have become valid. Once an output becomes valid it may not change with the arrival of additional inputs. In the reset phase the unit can drop its output once any of the inputs have been released and must drop its output before, or shortly after, all the inputs have been released. Unlike the DIMS approach, where the output is kept active until all inputs have been released, the early output units have no such restriction. This makes the logic cheaper to construct, but the stage completion must be ensured using a separate mechanism.

The structure of a two input early output OR gate is demonstrated in figure 4.7. The OR gate generates an early output when either of two inputs are 1. To complete the set of input states the AND gate generates an output when both inputs are valid but they are not covered by the early output set (both are 0).

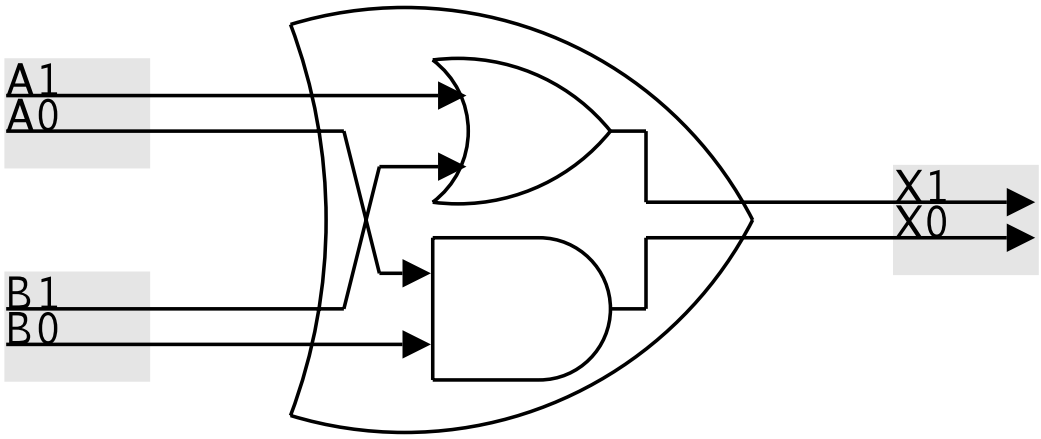

Figure 4.7: Two input early output OR gate

This combination can be used to create any dual-rail early output AND/OR gate with or without inversions on inputs and outputs. AND/OR gates output one value when all inputs are in a particular state and output the other value in all other input combinations. Any

inversions of inputs or outputs can be performed by swapping the wires representing 1 and 0.

With the arrival of the first input, the gate can either generate an output (if that input was attached to the OR gate) or wait for more inputs to arrive as the output cannot be yet determined with the present valid subset of inputs. This will continue until either the gate receives an input connected to the OR gate or all inputs connected to the AND gate have been activated. Once one of the inputs to the OR gate has been activated the AND gate cannot gain the full set of inputs it requires to activate. In this example it is easy to see that any complete input set will generate an output as either the input set matches the AND gate set (and the AND gate generates the output) or some inputs differ and instead activate one or more of the OR gate inputs.

The AND/OR gate has a full coverage of the early output states. It is not necessary to generate an output in every early output case. Construction of more complex logic composed with early output dual-rail AND/OR gates yields correctly behaving logic but often does not have full early output coverage. Figure 4.8 shows a multiplexer constructed with AND/OR gates and table 4.1 shows the behaviour of the unit. The unit generates outputs in all but one of the early output states (marked red). Due to the composition by parts (composing early output circuits from a set of early output gates) of the unit, the OR gate can only generate an early output once it has received a one as an input from either of the two dual rail AND gates. The OR gate receiving a zero from the AND gates could mean “transmit a zero as it is being selected” or “this input is not being selected”. This is why the A=1,B=1,S=X case results in an undetermined output.

This example demonstrates that composition by parts generates non optimal designs. This can be corrected by implementing from the whole specification rather than dividing the problem into parts. When designed using the full specification, the multiplexer covers all early output cases with no overhead in area over the original design. The following functions would be used:

```
X1 = (A1+S1).(B1+S0)
X0 = (A0+S1).(B0+S0)
```

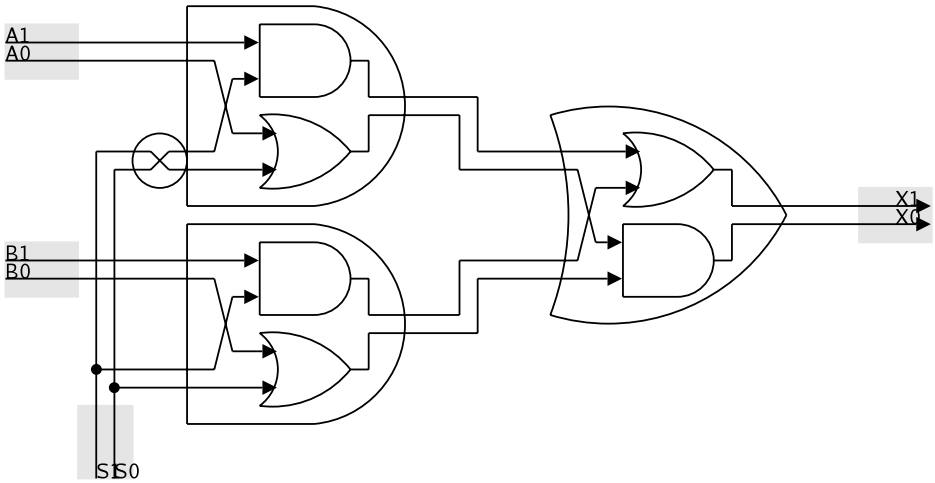

Figure 4.8: Composed early output multiplexer

	B=X	B=X	B=X	B=0	B=0	B=0	B=1	B=1	B=1
	A=X	A=0	A=1	A=X	A=0	A=1	A=X	A=0	A=1
S=X	X	X	X	X	0	X	X	X	X
S=0	X	0	1	X	0	1	X	0	1
S=1	X	X	X	0	0	0	1	1	1

Table 4.1: Output generation of a 2:1 early output multiplexer

4.4.3 Loose Guarding

Early output circuits are designed to generate outputs before all inputs have been presented. This property prevents the system from determining the state of inputs by observing the output alone. Instead, early output circuits require a method of *guarding* to ensure all inputs are ready to accept a transition of their acknowledge signals. In a *loose guarding* system the input latches signal their ability to receive an acknowledge transition by raising their validity output to state they are ready for the acknowledge to transition. To ensure all inputs are ready for the transition to take place, all validity signals are gathered using C-elements and finally combined with the output latch's acknowledge in a guarding C-element. The output of the guarding C-element is a *guarded acknowledge*. This acknowledge will transition only once all inputs are ready to receive it and the acknowledge from the latch (taking the result of the stage) has signalled it has accepted

the data. This enables any latch in the system to stall the following stage from entering either the set or the reset phase (by not transitioning their validity signal). It also makes the acknowledge from the output latch safe to be transmitted before all input latches are ready to accept it as it will be stopped from progressing to the inputs by the guarding C-element. This signal becomes latched in the guarding C-element which will keep it from progressing until all latches have accepted the previous transition (signalled by transitioning their validity signal).

Figure 4.9 shows the implementation of a gate with loose guarding. The Additional C-element generates a validity signal for the output of the gate wire bundle by simply gathering the acknowledge signals of the inputs to the gate.

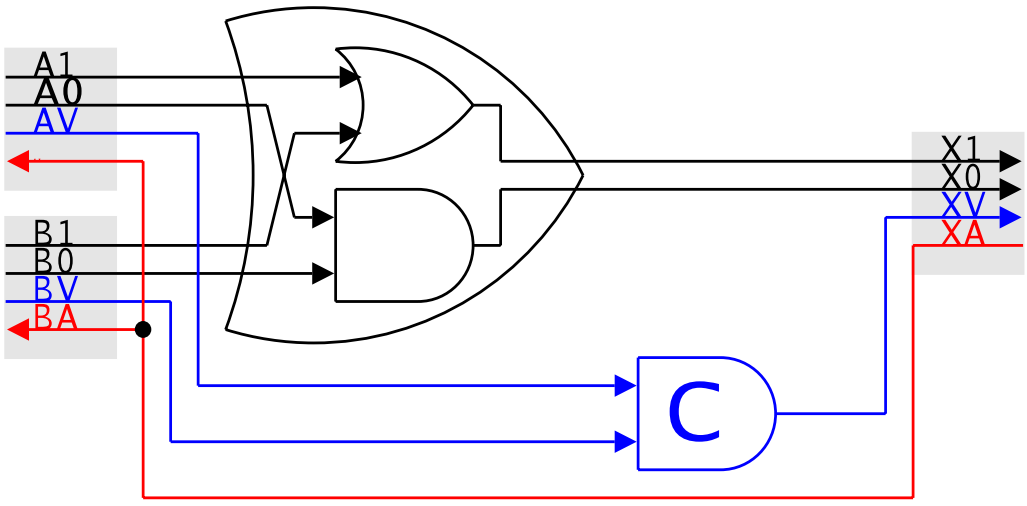

Figure 4.9: Standard early output gate with input guarding

This form of guarding is sufficient only for simple stages due to its timing assumptions (explained in section 4.5). Hazards can arise due to the limited scope of observability of the state of the circuit. Only the state of inputs and the output is observed (and not internal wires) when declaring it safe to move to either the set or the reset phase.

Although early output logic has no storage, the signals travelling through it have delay. Signals can propagate through many gates and once their source latch has released them the propagation of the falling edge can take a longer period of time than the stage's transition back to the set phase. These dying signals can then interact with other signals

in the new set phase in order to reach the output. The output in such a situation could be incorrect as it is based on data from the previous cycle.

Signals abandoned by their input latch are often referred to as orphans. An example of an orphan affected circuit is demonstrated in figure 4.10. The chained OR gate takes inputs from a number of sources, generating one output. The arrival of the furthest input (A) after the circuit has already completed will be met with an immediate acknowledge. This causes a short pulse on the signal emerging from this latch. As the stage is resetting this pulse can slowly travel along the chain of OR gates (coloured blue) eventually reaching the output once the stage has moved back into the set phase.

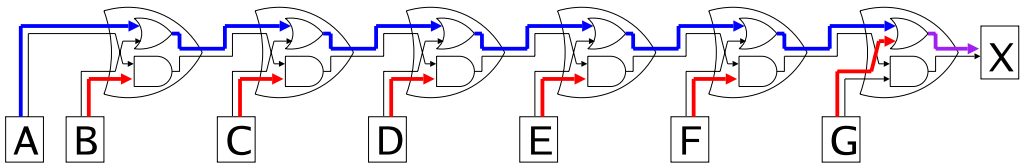

Figure 4.10: Example orphan circuit

This problem can be tackled in two ways. The first is to generate a set of timing modifications and force the circuit to only complete the reset phase after a period of time during which all orphans would have been eliminated. This is not delay insensitive and reduces the robustness of the circuit but as the occurrence of dangerous orphan propagating structures like the one in figure 4.10 is rare (normally large OR gate trees would be balanced), the methodology could have an impact on the performance of the system. This is described in section “Early output timing assumptions” on page 60. The second is to use a safer form of guarding to enforce a QDI level of robustness onto the system. Two forms of safe guarding will be described.

4.4.4 Forward Safe Guarding

As the validity gathering C-elements only observe the state of the input latches, they cannot ensure the internal state of the logic. This makes the circuits non QDI and susceptible to hazards. A QDI guarding system such as forward safe guarding can remove these hazards. Attaching an OR gate to the output of a dual-rail gate (as demonstrated in

figure 4.11) generates a local valid signal based on the wire pair within the logic block. This signal can be connected to the validity gathering C-element to ensure the validity of both inputs and the validity of the output. As these gates are connected to other forward safe gates the validity reflects the state of all stage inputs and data wire pairs feeding to the gate (even through other gates). The gathered validity will reflect both the state of inputs and the validity of all dual-rail wire pairs in the entire logic block. The acknowledge cannot transition until all inputs and signal pairs in the block of logic have become valid or returned to zero. This can ensure no orphan signals are present in the logic before moving onto the next computation cycle. This type of guarding is also implemented in the NCL-X design style [32].

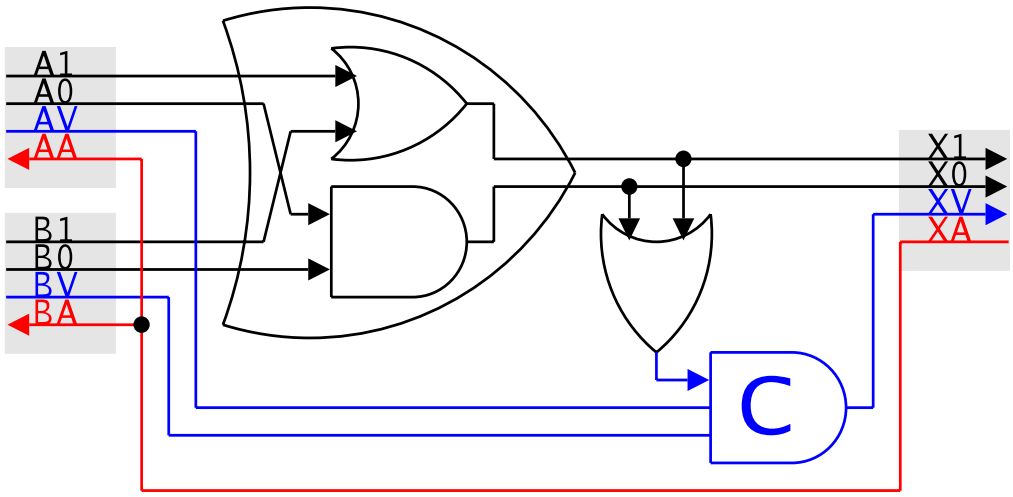

Figure 4.11: Forwards safe early output gate

The forward propagation of the computing signals remains unhindered by the additional guarding logic and the stage delay should be similar to that of the simple guarding version (assuming additional capacitance of wires due to fan out to the validity testing OR gates have little impact). Although the result can propagate to the next stage, the gathering of all inputs and the restriction that all paths must be activated before the acknowledge is permitted to propagate can have a negative effect on the speed of the reset cycle. The need for all wires in the design to be checked along with all inputs (as opposed to just the inputs in the loose guarding system) creates a large gathering C-element which would have to be constructed from a tree of smaller C-elements.

The effect of this gathering style is to force the unnecessary late inputs to arrive and set the parts of the logic they control. After this is done, the reset of the stage also checks all wires in the logic block before releasing the acknowledge. The check has to be conducted on both the rising and falling transitions of all wire pairs, as the only way to ensure the sub circuit has completed and gone through the cycle is to observe all wire pairs rising and then falling.

This approach creates a fast propagating result (low latency maintained) but a slowly propagating dropping edge (reduced throughput).

4.4.5 Backwards safe guarding

The *backwards safe guarding* model checks the data wire pairs for the presence of data on the propagation of the acknowledgement signal rather than the validity. This allows parts of the logic and some inputs to reset before the arrival of all inputs.

In backward guarding, validity is generated by taking an OR of the two data wires. This then only signals the local validity of that wire pair and so does not imply that it is safe to pass the acknowledge to all the inputs of the gate (as some may have not arrived). To protect the inputs of the gate from receiving acknowledge signals before they have asserted their validity, a C-element is placed to propagate the acknowledge only if both inputs have become valid. This is arrangement is demonstrated in figure 4.12. The construction is similar to the “Reverse Path Completion” created by Luis Plana used in the Balsa system (unpublished).

Unlike the forward guarding system, backward guarding examines the state of the circuit on the acknowledge propagation. This can make it slower than the forward guarding system as backward guarding, instead of testing the circuit for validity of internal dual-rail wire pairs in parallel with it computing, waits until the circuit enters the reset phase before doing so. The advantage of backward guarding is its ability to acknowledge a subset of inputs, in early output cases, while waiting for the complete set to arrive. This property is exploited in section “Backward safe guarding” on page 64.

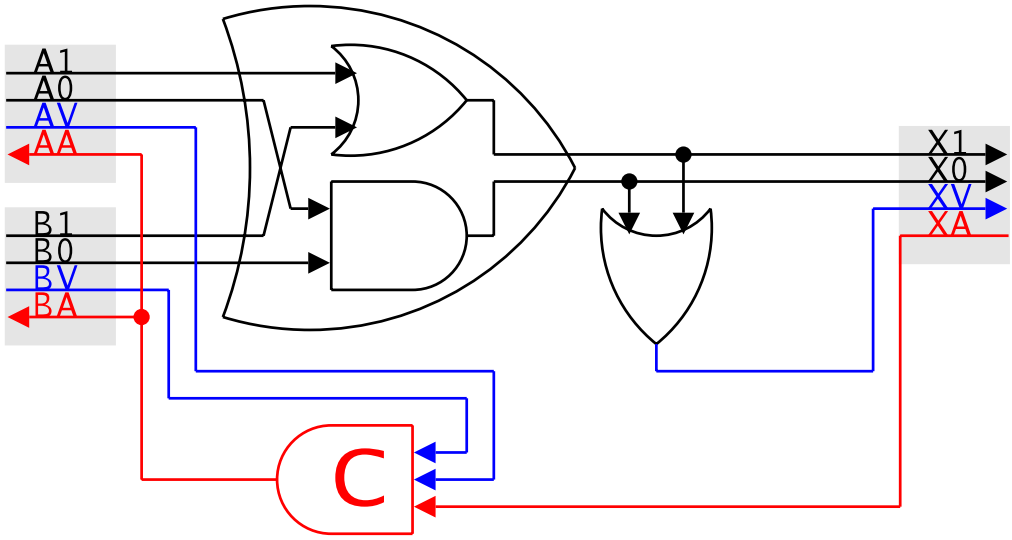

Figure 4.12: Backwards safe early output gate

Backward safe guarding also creates circuits with low latency but a reset phase even slower than that of the forward safe guarding system (lower throughput). The two safe guarding approaches generate QDI circuits which are very robust but at a cost of reduced throughput. Additionally, a safe guarded system cannot take advantage of the anti token latch described in the next chapter (although the backward safe guarding system can make limited use of them). For these reasons it would also be advantageous to determine the timing assumptions to avoid using a safe guarding system. In most circuit stages the loose guarding system is sufficient and it would be advantageous to determine if the circuit has possible hazards before adding additional guarding logic to remove them.

4.5 Early output timing assumptions

As stated in section 1.6, timing validation is outside the scope of this thesis, but the timing assumptions made and how they have been upheld in experiments conducted in later chapters will be presented. The timing assumptions in loose guarding early output systems are based on the removal of orphans. The safe guarding strategies achieve this by observing every intermediate signal in the system to ensure all were valid (or returned to zero) before allowing the next transition on data signals.

The example presented in section “Loose Guarding” on page 55 will only cause a hazard if a timing constraint is broken. There is a race between the data signals progressing through the logic gates and the validity signal travelling to the output through a series of C-elements which then switches the guarding C-element within the output latch. This enables the acknowledge to be asserted which releases the data signals of some input latches which then releases the data to the output. In figure 4.13, the two paths are shown in red and blue. If the blue path takes longer than the red path, the blue signal would reach the final OR gate after the red signal on the other input would have already been released. This would be seen to the output latch as two distinct tokens. This can cause an extra token to be inserted into the pipeline which can either deadlock the system or unsynchronise a pipeline causing it to function incorrectly throughout the remainder of the execution. The alternative is that the two results of the stage were merged and either matching and leaving the error unseen or causing both bits in the result to be one (a disallowed state).

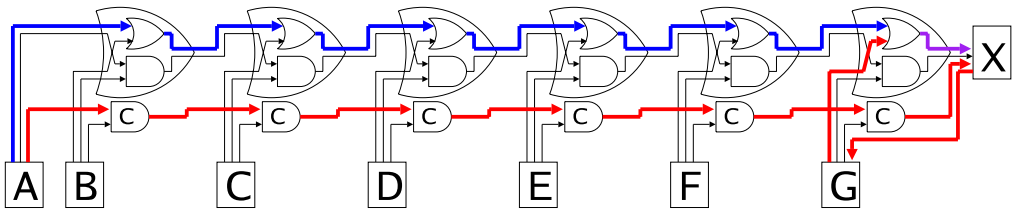

Figure 4.13: Orphan race

In the figure, the paths have visibly different lengths and it would be very difficult for the post layout routing to make the delay of the blue path longer than that of the red path if the gates were formed from cells which forced the C-elements to be placed directly next to the logic gates. This does also assume the C-element validation network does not become optimised into a shorter path (e.g. by forming a balanced tree). All early output circuits can be assumed to be safe if they uphold the condition that the propagation delay through each gate’s validation gathering C-element is slower than the data propagation. These timing constraints can be extracted and used to guide the layout software to ensure these conditions are met.

If it is difficult to adhere to these constraints in some gates, those gates can be re-implemented using one of the safe guarding methods or additional delays could be placed on the generation of the valid signal from some gates.

Chapter 5: Anti-Tokens

Guarding logic provides a method of ensuring an input is present before acknowledging it. Until a late arriving input arrives, the stage (or at least part of the stage as will be shown in section "Backward safe guarding" on page 64) is unable to complete the acknowledgement. This not only stops the stage progressing to the next set of inputs but also requires it to continue computing a data input already determined to be unnecessary. The first challenge, of releasing the stage to move to the next set of inputs, can be accomplished by keeping a flag inside the latch representing an instruction to acknowledge the next token and to not propagate it to the next stage. This would allow the stage to continue processing, safe in the knowledge that the late and unnecessary token will be destroyed and instead only the following token will be presented to the stage.

The flag kept in the latch to destroy one token can be thought of as an "anti-token". Anti-tokens in collisions with tokens destroy both the token and themselves. This is done by adding a not moving front edge to the arriving token. This, as demonstrated in section "Advanced Latch Designs" on page 47, causes the token to be removed.

Although a latch could be designed to hold a number of anti-tokens, the increase in logic area and propagation delay through the latch is undesirable. Instead of latches gathering anti-tokens and waiting for inputs to arrive to be destroyed, the latch should be able to forward the anti-token to the stage computing the unnecessary input. This both solves the problem of unnecessarily computing the input and removes the need for the latch to store a number of anti-tokens.

5.1 Anti-token theory

In order to understand how an anti-token processing system can be built, a few implementation styles should be examined to determine the desired behaviour and possible implementations.

5.1.1 Backward safe guarding

Backward safe guarding, as described in section 4.4.5, allows a subset of inputs to move back into the set phase once all dual-rail wire pairs they effect have become valid and returned back to the null state [38] (an example of this will be shown). This is because the completion of the stage is not determined on the propagation of the validity signals but rather on the propagation of the acknowledge. This often allows the acknowledge to reach some inputs of the stage which have become valid. The other input subset with members which have not become valid, is halted until all inputs are presented and subsequently reset. This action can be repeated allowing the result of the stage to become several stages ahead of some inputs. Unfortunately, in each cycle the halted set of inputs grows eventually absorbing the whole stage. This partial completion behaviour has the effect of collecting anti-tokens to absorb the unnecessary inputs. Although the anti-tokens are unable to progress through the input latch onto the next stage backwards, careful designing can enable many anti-tokens to be collected in a single stage.

Anti-token generation and stacking can be demonstrated in an example circuit shown in figure 5.1 (note this is a different circuit from those shown in chapter 4). The circuit has been abstracted to show only the request and the acknowledge wires. The values arriving on all inputs (latches A to F) in the example are always 1. In the initial state, shown in the figure, the circuit is provided with inputs B to F. Input A is not provided. This is enough to generate a result and the output (latch Z) receives a request. This output generates an acknowledge despite the absence of the complete set of inputs to the stage.

Despite the acknowledge from the output latch, the late and unnecessary input cannot be acknowledged until its token has arrived. This stops the progress of the acknowledge signal from reaching the input and any inputs which are combined (directly or indirectly) with the path of inactivity due to the non-presence of the input. In this case, the path of

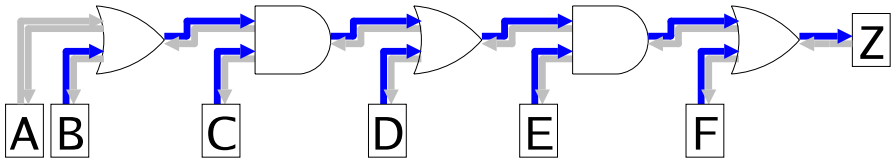

Figure 5.1: Backward guarded stage with full input set

inactivity only reaches one gate and the only other input connected to that gate is input B (as shown in figure 5.2). The acknowledge signal can reach all other inputs in the stage.

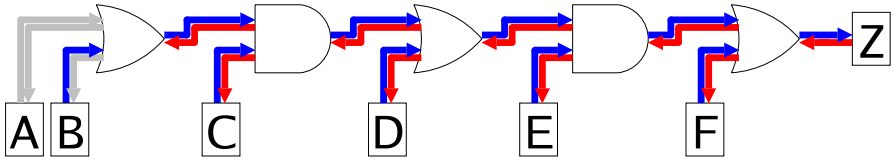

Figure 5.2: Stage after acknowledge

The release of all other inputs in this case also drops the output and the acknowledge signal is released by the output latch. This causes the acknowledged region to be shortened down to the single gate which is waiting for one of its inputs to be released (signalling the acknowledge is being propagated), and one gate which is not releasing its output signal due to it not being able to propagate the acknowledge signal (as shown in figure 5.3). Although input C becomes reset it cannot become valid again as a gate it feeds is propagating the acknowledge and it will not stop acknowledging until all its inputs release their requests.

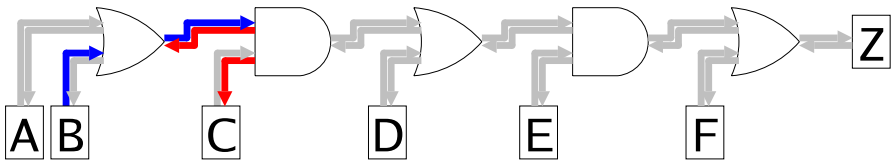

Figure 5.3: Stage after removal of acknowledged inputs

All other inputs can now become valid. Again, if the set of inputs is sufficient to generate a result this stage can complete and generate another acknowledge cycle. Each acknowledge region (acknowledge high between the AND and the OR gates) is effectively an anti-token. Figure 5.4 shows the maximum anti-token capacity of the stage. It can store two anti-tokens while still being able to generate a result which allows the output latch to acknowledge, although the acknowledge cannot reach any inputs. This effectively gives the stage a maximum anti-token capacity of two and a half.

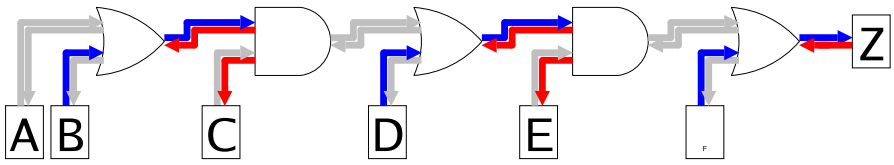

Figure 5.4: Stage with maximum anti-token capacity

Each anti-token waits for the presence of valid data on all its inputs before acknowledging them and does not release the acknowledge signal until all inputs have returned to null (signalling they have accepted the acknowledge). This allows the anti-tokens to stack up and not merge into a single anti-token.

The maximum anti-token capacity of a stage can be determined through the difference in the number of inversions in the late arriving token path and the input subset still capable of generating an output assuming the circuit uses only one kind of gate (either AND or OR where the other gate can be created through applying DeMorgan's theorem on the available gate). Each inversion can store half an anti-token. In the given circuit this would yield a difference of 4 inversions. An additional half anti-token can be stored in the output latch so the total number of half anti-tokens which could be stored in the stage is 5.

Unfortunately, stages are rarely able to keep more than one anti-token and more often they can only separate inputs from their output by half a cycle. This also does not resolve the problem of stopping the computation from being carried out by propagating the anti-token through a latch. Additionally, the computation of the stage's completion being done only after the stage has generated a result, yields lower performance, as will be shown in the next chapter.

5.1.2 Counterflow pipeline processor

Interaction between moving tokens is usually done at specific points where they become synchronised (should either token arrive before the other it will then wait for the other before the action takes place). Interaction between tokens in a system where the interaction point is not specified becomes more problematic. This is what anti-tokens attempt to achieve by allowing the token to progress though the latch to the previous stage. By specifying the point of interaction the backwards safe guarding anti-token system can avoid the use of arbitrating components (the input latch of the stage is always the interaction point in this case), which are necessary if a synchronisation point is not specified. Unfortunately, one of the advantages of anti-tokens is that each token tries to progress to the other thus halving the amount of time they take to reach each other.

The counterflow pipeline [41] is an example of a system where tokens moving in opposite directions interact. Composed of two pipelines, instructions flowing in one direction and the results and register contents travelling in the opposite direction (opposite direction to the instructions) interact. Interactions can destroy or change data carried by tokens flowing in the opposite direction. The result tokens take the latest value of the register taken from the instruction token. The instruction tokens can also read or update the values of registers they are operating on. Often there are also ways of deleting tokens in situations such as taken-branch instructions; this removes the speculatively, but unnecessarily, fetched instructions.

There are a number of implementations, both synchronous and asynchronous, of counterflow pipelines. The “asynchronous counterflow pipeline processor” is the most relevant. The system allows tokens to travel in both directions unhindered until they try to progress to a stage occupied by another token flowing in the opposite direction. The dangerous action of both tokens moving to their next stage and bypassing one another is protected against by a *cop* element. The cop element takes the requests, from the two pipelines, for transition of a token to the next stage and only permits one of them. The other token’s movement is halted until the stage has signalled it has finished operating on it. This ensures that both tokens are present in a processing stage and only that stage is able to process the interaction between the two tokens.

The cop element uses a mutex element (described in section 2.3.2) to ensure only one of the transactions is granted. Unfortunately, the use of arbiters can have a negative effect on the performance of very finely pipelined systems with a lot of collisions. With the heavy use of arbiters with their non-deterministic delay, the design style has an unpredictable computation time leaving it unmarketable for real time and quality of service applications. Although this design style could be adapted to carry tokens and anti-tokens in the two pipelines, the overhead in delay of the sequencing arbiters into every latch transaction could remove any benefits gained by removing unnecessary speculative operations. Additionally the technique is unnecessarily complex simply to remove colliding tokens, as it is also designed to perform arbitrary computation on the collision.

As no computation is needed to remove tokens and anti-tokens upon their collision there is no necessity to ensure they meet in a single stage. Counterflow pipelines use two separate pipelines to allow tokens to pass (after interaction) and for both to carry data. As both tokens are always removed and data is carried only in one direction, there is no need to have separate pipelines.

5.1.3 Counterflow networks

Counterflow networks [42] are neural network type organisations communicating using asynchronous protocols. Neural networks comprise a mass of neurons, each “fires” upon reaching a threshold of neighbours already fired. The threshold is determined by summing weighted inputs and comparing to an output threshold and once a neuron fires it causes more of its neighbours to fire. Connections between the neurons are bi-directional.

The two elements used to compose counterflow networks are the node and the link. Each node is connected to other nodes using links. Links do not have a direction and communicate the state of both nodes, synchronising them to ensure each node has acknowledged the transition of the other. Nodes collect the states of their neighbours and once having reached their threshold they fire.

Both the node and the link are symmetrical and use a three wire interface. Figure 5.5 shows the construction of both the links and nodes and demonstrates the how they are connected. The three wires labelled L, R and N are used for communication. The L wire

(Link Request) signifies a link passing the request from one node to its neighbour (through the link). This request is passed to the link over the N wire (Node request). Wire R (Ready) is used to enable flow control between the two nodes. The link uses this wire to stop the two nodes it is attached to from firing or returning to their non excited state.

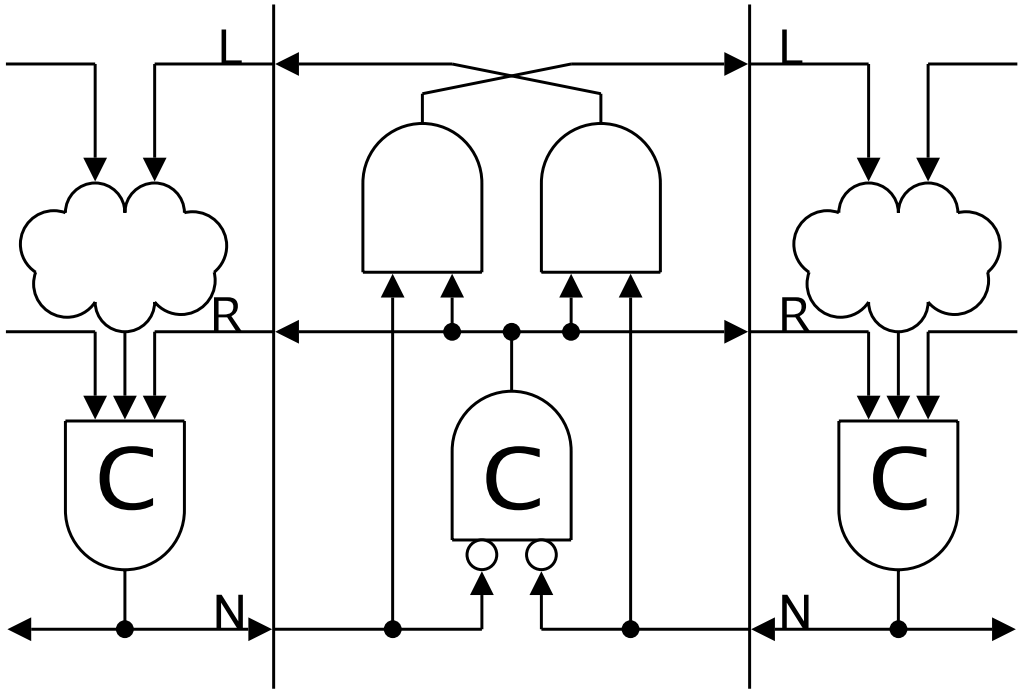

Figure 5.5: Circuit of two counterflow nodes connected by a link

Each node collects the Latch request signals in the threshold unit. The output of this unit is then gathered with all Ready signals in a C-element. The C-element output becoming active represents the Node firing. This signal is then passed back to all links to forward to other nodes. Links which connect two Nodes which have fired then release their Ready signal. This also drops the Link request signals passed to both nodes. The link will wait until both nodes have returned to their inactive state before re-activating the Ready signal which in turn allows requests to propagate through the link.

This sequencing forces each node to wait for all its neighbours to fire before releasing its request and ensures all links are “ready” before firing. Not only does this ensure the message is propagated and not lost but it also separates waves of activation.

In a simple example a string of nodes and links is made in a FIFO like arrangement (like the one shown in figure 5.5), the threshold of each node is set to one, meaning a node will fire if either of its neighbours has. A wave of activity can be caused by either end's token firing. All nodes will then fire in order progressing the activity towards the other end. Each node, once it has fired (and so have its neighbours), can return to its un-excited state. This is not necessarily on the trailing edge of the activated region. A new pair of trailing edges can be created by any node. This splits the excited area into two parts creating a new trailing edge for the firing front and a trailing edge for the decreasing back part of the activated region. This behaviour is similar to that of the early-drop latch described in "Advanced Latch Designs" on page 47.

Each node fires and waits for all of its neighbours to fire before dropping to its inactive state. Only then can a node fire again, once all neighbouring nodes are ready to receive a new request (signalled through the Ready lines). The enforcement of at least one node separating regions of activity ensures the trailing edge of an activated area is not accidentally connected to the rising edge of another activated region. Again this is another behavioural trait in common with the token based asynchronous system.

The most important behavioural aspect of the couterflow network system is the merging of activation regions progressing in opposite directions. Unlike the trailing edge, there is no protection for the front edge from merging with another activation area. This means that two areas moving in opposite directions will merge. Any areas with no leading edges will be removed by the trailing edged deactivating the remaining excited nodes.

A token collision happening in a FIFO example is demonstrated in figure 5.6. Two activation regions are generated in nodes at opposite ends of the FIFO (point 1). These are progressing in opposite directions towards each other. By point 3 on the diagram the trailing edges start releasing some nodes at the back of the regions. Because the regions are flowing in opposite directions, at point 4 they merge their leading edges leaving only trailing edges on the activation area. These then release the remaining activated nodes and by point 6 remove the whole region.

This merging and destruction of two regions moving in opposite directions can be used to implement anti-tokens. To remove and stop a region from progressing, another activation

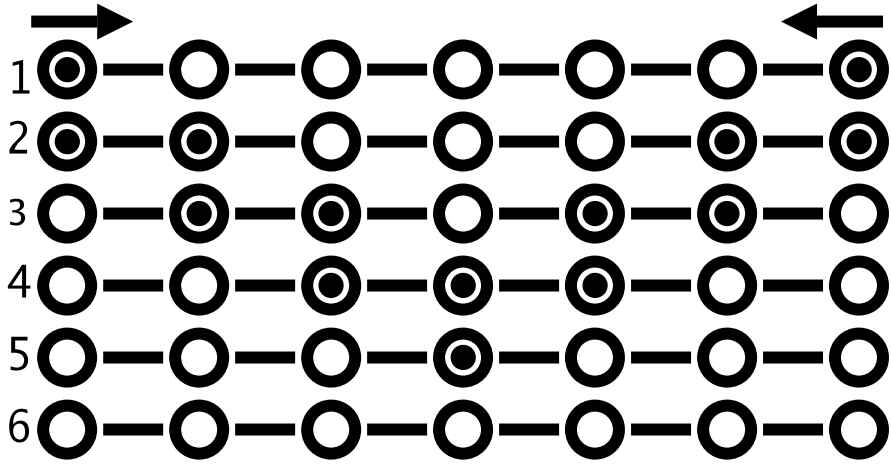

Figure 5.6: Activation area merging in a counterflow FIFO

area can be introduced travelling in the opposite direction towards the undesired region. The introduction of these counterflowing areas occurs in counterflow networks when a number of neighbours to a node have not fired at the time the node fired. The newly fired node then passes its request to the remaining unexcited neighbours in an attempt to cause them to fire. This removes the slowly progressing activation areas which should have reached the firing node through the node's still inactive neighbours. This is very similar to the desired behaviour of the anti-token circuits.

5.2 Control Circuits

Three methods of generating anti-tokens have been proposed. The backward safe guarding anti-tokens are implicit in the design and need no additional effort to be implemented. Unfortunately this method does not allow tokens to progress backwards through latches. The counterflow pipeline does allow the progress of anti-tokens through latches but requires a large amount of additional logic. An anti-token pipeline along with 'cop' units would be required alongside each forward pipeline, consuming both power and area. The additional logic will probably have such a negative effect on speed it would be difficult to find cases where anti-tokens can have a greater beneficial effect to counter the overheads.

The most promising approach is the counterflow network system which uses a single pipeline to communicate in both directions. Additionally, many similarities can be drawn between it and the early output system making it easily adaptable to them. Both systems break into two parts: the communicating components connecting two units together and the computational components taking inputs form one set of connections and generating outputs on other connections. The only difference is that the control circuits have a nominal direction of data flow. Data flow direction is partly enforced by the sequencing of the delay insensitive protocol underlying the model. Although the sequencing of each early output style transaction requires the data to be present before the validity rises (to ensure the acknowledge cycle begins only once the input has presented data to be acknowledged), this condition can be removed if the latch can remember to remove the token at a later time. In such cases, the token may not need to be present in order to complete the stage's computation. The acknowledge will reach inputs which have not presented data to the stage if the latches which they pass through present the validity early. Presenting the validity early allows the acknowledge to propagate to the late arriving token. This would only be done if the latch is both capable of propagating the acknowledge to the next stage and it is not currently doing so. Each latch reserves the ability to not raise the validity line if it is not ready to accept an early acknowledge. This can mean the latch is not ready to receive an acknowledge at that point in time or the latch is incapable of doing so (not an anti-token latch).

5.2.1 Latch

The anti-token latch can be implemented by altering the counterflow network node and link designs. Taking the node and link design and directly mapping it into the control circuit's latch template yields the implementation shown in figure 5.7. In order to map the design to the control circuit specification, the node firing function is mapped to the early output function and an OR gate combines it with what would have been the link request signal in the counterflow network design. The ready gathering C-element from the counterflow network design is mapped to the guarding C-element and the validity gathering tree.

The request in line (Ri) is combined with the acknowledge out (Ao) signal (after being passed through an AND gate to ensure it is only active when the latch is 'ready') in an OR

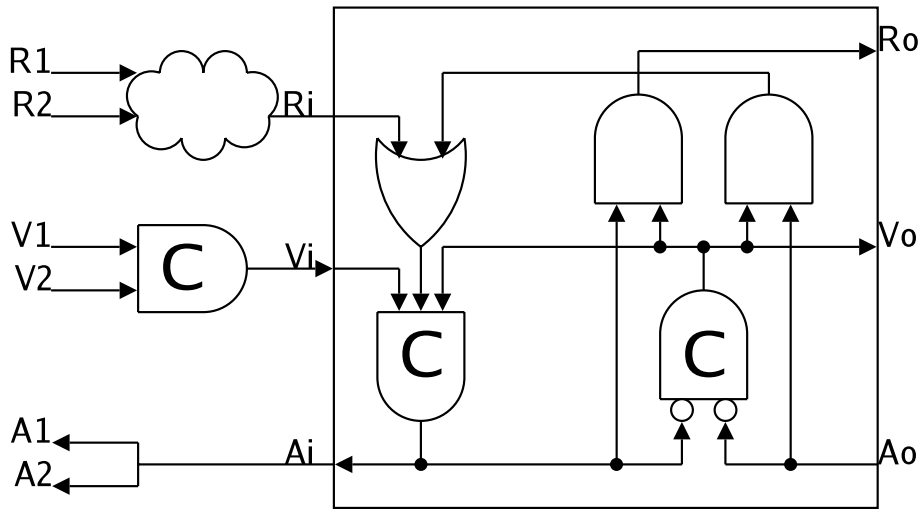

Figure 5.7: Control circuit adapted anti-token latch

gate to allow either of these inputs going high to trigger a completion of the cycle. This means once the result has been generated or the activation of acknowledge out (Ao) has signalled a receipt of an anti-token, the latch can complete. This triggers an acknowledge on the input side. On the output, if a token was being passed, the Request would be activated and the latch will wait for an acknowledge. On passing an anti-token the acknowledge has already been received and so the stage can fully complete by releasing the validity out (Vo) signal.

Early output re-insertion

In counterflow networks, the nodes will only complete once all links are ready. There is no special output link which can fire once the threshold has been reached and before all inputs are ready. Because, in computing systems there is a nominal direction of data flow, the circuits do not have a symmetrical behaviour across the input and output nodes. In early output systems it is possible to take advantage of the threshold being reached before all inputs have raised their validity. The result is propagated to the next stage and the gathering of validity signals ensures that, only once all inputs are present, does the acknowledge become asserted.

To regain the early output property, the Ro signal must also be activated when the Ri signal has been triggered. To achieve this the Ri signal must be able to trigger the activation of Ro. This can be done by merging the Ri and Ai signals in an OR gate as demonstrated in figure 5.8. This causes the Ro signal to become active when the Ri signal arrives, and stays active until the acknowledge Ao has arrived.

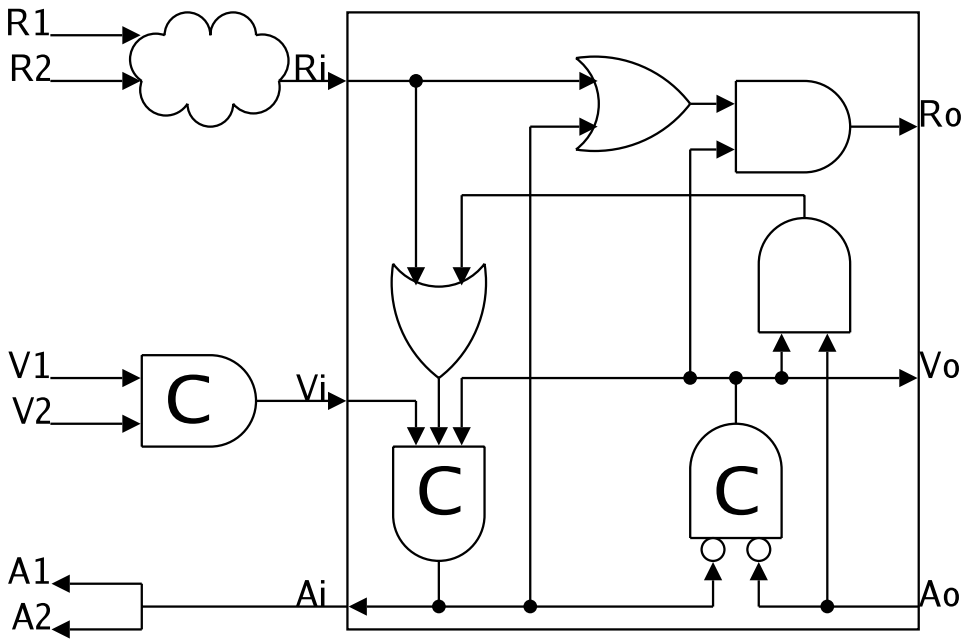

Figure 5.8: Early output control circuit anti-token latch

5.2.2 Logic

The logic in anti-token based systems is the same as that in the early output logic. The assertion of validity before the request has been generated is not within the early output protocol. The method of circuit generation does not need to change but in order for the system to allow anti-token latches to communicate with standard half and early drop latches, additional timing assumptions must be upheld. These are described in section 5.5. This allows the use of anti-token latches only in places where they are beneficial while in other places half and early drop latches are appropriate. The benefits of anti-token latches become greater when used in data processing systems such as bundled data and dual rail.

5.3 Bundled data

The implementation given above can be used for bundled data systems. The latch can be used in place of a half latch and connected to any early output stages (as long as the timing assumptions described at the end of this chapter are upheld). The latching signal can be taken from a number of points in the design to generate the correct latching for a four phase early handshake. The “acknowledge in” signal is easiest to sample as other signals can often be optimised away to form complex gates.

5.4 Dual Rail

In the past two chapters, the dual rail versions of the latch could be generated by duplicating the request data path. Unfortunately, in this design, this would require the duplication of most of the gates and the addition of several OR gates to detect completion of different wire pairs. Instead, a redesign of the request passing system allows a smaller design. The new design is shown in figure 5.9.

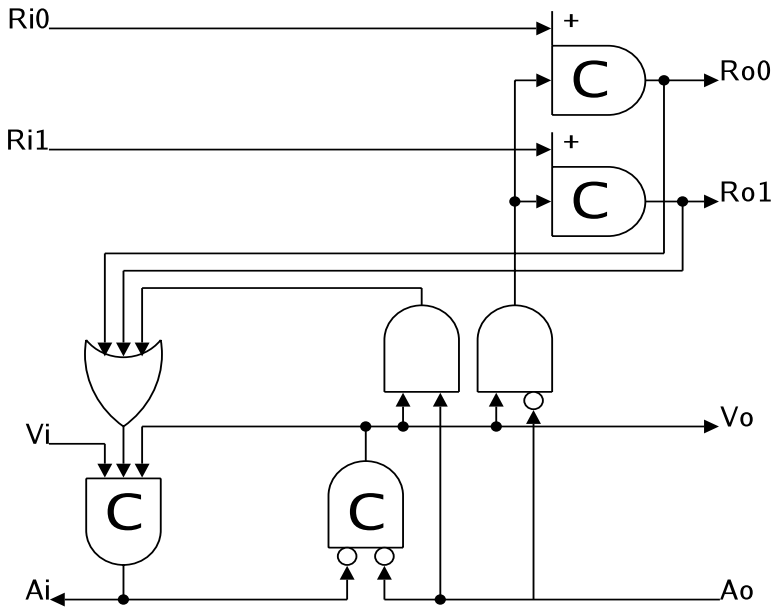

Figure 5.9: Dual-Rail anti-token latch design

The data is stored in asymmetric C-elements (see “C-elements” on page 25). These release their data as soon as the acknowledge on the output arrives. This gives the anti-

token latch an early drop property. This comes at a higher expense than the early drop latch (described in section 4.2.3) but it does have a different timing response to stimuli. The acknowledge out rise to request out fall time is two gate delays compared with the early drop and half latches which take just one.

5.5 Anti-Token protocol

Anti-token latches do not adhere to the early output protocol which assumes sequencing of the transitions on the request and validity signals. Figure 5.10 shows the early output protocol and figure 5.11 shows the STG of the sequencing. The transitions marked in red on the figures are the Req to Val transitions. This sequencing is upheld by early output half and early drop latches and by ‘safe guarded’ gates (both forward and backward). It is even upheld across loose guarding logic if the delay of the validity gathering C-elements can be assured to be greater than the delay of the logic gates.

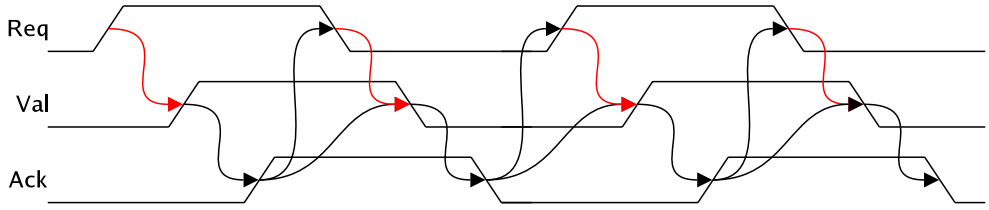

Figure 5.10: Early output protocol with safe sequencing

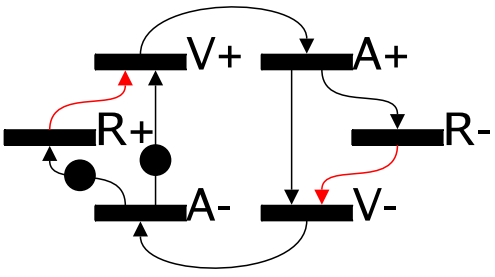

Figure 5.11: Early output protocol STG

Because anti-token latches rely on being able to assert their validity before asserting the data request, this request validity sequencing assumption cannot be upheld. Additionally,

the use of loose guarding allows the request signals to become asserted and reasserted during the reset phase (as just-arriving signals ripple through the stage and arrive as pulses on the output). Finally, the request signals need not rise within a cycle due to an anti-token pass. The sequencing in figure 5.12 shows the updated anti-token allowing based early output protocol and figure 5.13 shows its STG.

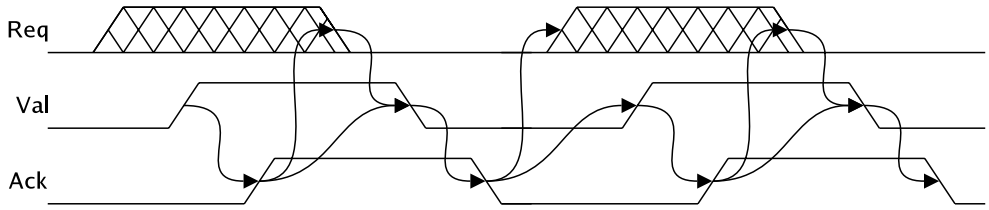

Figure 5.12: Anti-token protocol

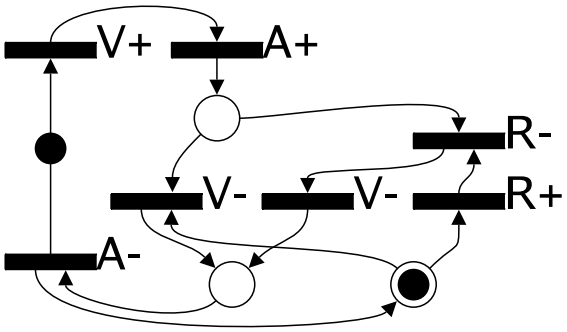

Figure 5.13: Anti-token protocol STG

5.5.1 Timing assumptions

Stages with anti-token latches on their inputs are susceptible to generating several request pulses on the output of the stage during the acknowledgement. Latches should only accept the first request, and while the stage is acknowledging, the latches should ignore any transitions on the request wires. To stop the data C-elements in the half and early drop designs from receiving additional tokens during the acknowledge, the acknowledge in signal could be inverted and connected to each of the data latching C-elements. This stops the C-elements from capturing new data while the previous stage is in the reset state

(while Ai is high). Anti-token latches are already protected from accepting tokens during the reset phase.

This is a simple strategy to stop all orphan glitches from reaching the next stage. It was tested and shown to work in a circuit specifically designed to cause such a situation. These cases very rarely occur as circuits which generate such behaviour are composed of long strings of OR gates along which the orphan can travel, a specific sequencing of arrival and data would also be required. In the next chapter, a number of circuits will be presented, including one which has a long line of OR gates which is a very susceptible to failures to meet these constraints. During the testing and benchmarking of these circuits, the situation of an orphan managing to propagate to the output of a stage never occurred throughout the thousands of simulations conducted on these circuits. This was partly because most of the stages were very small and balanced (roughly logic equal distance for all inputs).

These timing assumptions, and methods of protecting latches against orphans, are presented to demonstrate that the timing hazard problem may be solved without paying a high penalty in performance. A number of assumptions, such as the comparative delay of a C-element versus a gate, were made, but as these are outside of the scope of this investigation, they will not be justified. Only a simple model of orphan generation and propagation was presented and further work in the area should reveal better approaches to achieve a highly robust system still capable of working with anti-tokens.

5.5.2 OR-causality

The functionality of the anti-token latch is based on OR-causality [33][36][37]. OR-causality is a method of triggering a transition once one of a number of input events happens, in contrast with AND-causality which requires all input events to happen before the transition is triggered. In the case of the anti-token latch either the request in (Ri) or acknowledge out (Ao) transitioning up causes a sequence of transitions.

This kind of behaviour is difficult to describe in DI STGs as either (or both) of the inputs can cause the output transition and each input transition should be acknowledged. An example of a method of guaranteeing all inputs are acknowledged is shown in figure 5.14.

Either of the two inputs (A and B) transitioning up can cause X to transition but both inputs are necessary to arrive before (observed through signal Y) the cycle can complete and they are acknowledged. This forces a sequencing on the inputs to the unit and both A and B have to cycle even when only one causes the transition of X. In the figure, although X does not cause Y to transition, the implementation of the circuit ensures that the transition on X will happen before the transition on Y. For this reason the transitions are connected through Read Arcs (dashed lines), which signify the transition is guaranteed to happen before, although it does not directly cause, the other signal transition.

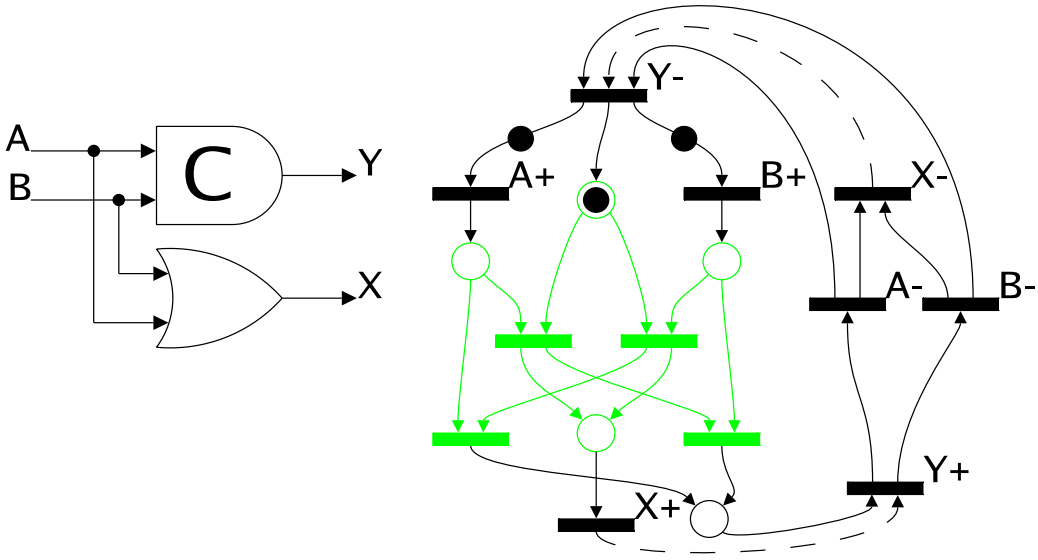

Figure 5.14: OR-causality example circuit and STG

The design of the anti-token latch presents an additional problem where only one of the two signals (Ao) is guaranteed to be cycled each transaction. The Ri signal may never arrive. In the previous example the Y- transition was used to guarantee both signals had completed their cycle before starting a new one. In that example, both signals were guaranteed to rise and fall during each cycle with Y synchronising them. In the anti-token latch the high periods of the two inputs entering the OR-causality segment are confined to periods between sensed transitions. In the case of the Ao signal the high period is guaranteed to be present each cycle and so can be observed by directly sensing the Ao signal, in a similar way to the example design. In this case the Vo transitions synchronise with the Ao signal much in the same way that Y was used in the example circuit. The Ri

high period is not guaranteed to be present during each cycle and another method of ensuring the signal does not become high some time later during the cycle must be present. Although the Ro signal high period is not guaranteed to be present each cycle (and thus the circuit cannot observe it transitioning up and then down to guarantee it has completed) the high period of Ro must be confined between Ai- and Vi-, as one of the rules of the anti-token protocol state that each cycle the Vo- transition must happen after any Ri transitions. Figure 5.15 presents the STG of the behaviour of the anti-token latch presented in figure 5.8 on page 74. The OR-causality part of the figure is highlighted in green to match the highlighted OR-causality in figure 5.14.

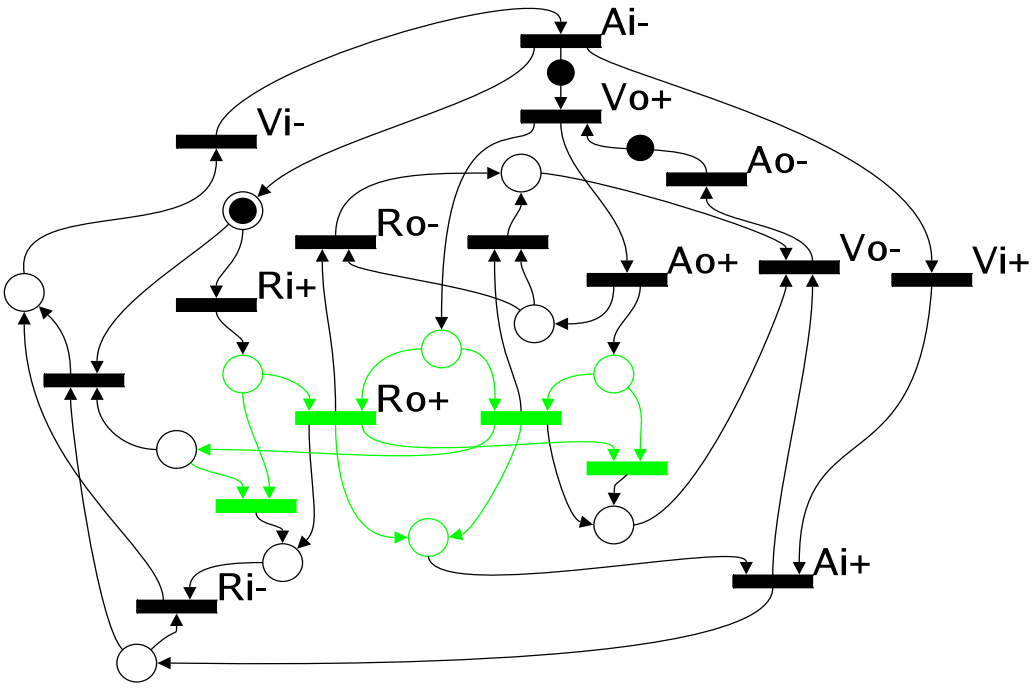

Figure 5.15: Anti-Token latch STG

In addition to dealing with the OR causality between the Ao and Ri signals, the latch must also generate the Ro signals which could be acknowledged before or after it is generated. In the STG the Ro signal is withdrawn only if it was raised during that cycle. This decision, along with the OR-causality, could be resolved using a mutex element but a mutex free implementation is preferable as a mutex element can consume an unpredictable amount of time to resolve. Although the probability of a long metatstable

period within the mutex is very low, the use of a mutex in a frequently used component, such as this, increases the probability to a level where it could become problematic.

Chapter 6: Application and Analysis

To understand fully the behaviour of circuits made with the techniques described in the previous chapters, properties of circuits affected by this change in the design style must be identified. These properties can then be analysed in circuits to become performance metrics.

This chapter will concentrate on two areas of the early output designs. The occurrence of early outputs will be studied and then methods of improving this metric will presented. The second area focused on will be the performance of complete early output circuits. After analysing the performance of circuits, a series of optimisations will be presented and their effects on the performance will be demonstrated.

6.1 Early output occurrence

The ability of circuits to generate early outputs must be measured to justify the claim they increase performance. Different computations have different early output properties. To demonstrate these properties, 12 common circuits were observed with varying numbers of valid inputs present. For each circuit the full input state space was explored and for each input combination, the ability of the circuit to generate a result was recorded. As some circuits tested have over 100 million input combinations a program was written to simulate the circuit in each case and then optimised to allow reasonable computation times. Its operation will be explained in more detail in section 6.1.2.

6.1.1 Benchmarks

The circuits chosen were taken from a synchronous design [43]. These were not altered or optimised to allow the tests to observe the true performance of circuits created by synchronous engineers and then passed through the program the function of which will be

clarified in the next section. Each benchmark circuit will be described and its function presented.

7seg

The 7seg benchmark is the generator for segment A (top segment) from a 7 segment display decoder. The input is 4-bit binary coded decimal and the output is the state of the top segment of that digit. High output of the segment is distributed randomly across the input state space. As the binary number being presented to the unit is limited to the range of zero to nine, the extra input states can be ignored on the logic generation. This can allow the generation of smaller implicants (larger implicant loops on a Karnaugh map) and thus increase the likelihood of early outputs.

ALU

A one bit slice of an ALU from the reference design. This takes a three bit code defining the operation on the two, 1 bit values along with a carry in. The three bit code only represents 6 operations and so there is some redundancy in the code as well as the carry in only being necessary for addition and subtraction operations. The other (logical) operations are AND, OR, NOR and XOR.

AND

Early output states are generated most frequently in very large AND/OR gates. This benchmark uses an eight input AND gate to show an ideal circuit where the probability of early outputs is very high even with a very low number of inputs.

Adder

Although it is impossible to generate a full result of an addition without the presence of all inputs, it is possible to generate parts of the result with only a subset of inputs. This benchmark measures the ability of bit eight of the adder to generate a valid result. The

inputs higher than bit eight in the adder have no effect on the result of bit eight and are thus ignored.

Branch

The logic to determine if a branch should be taken in the original design took the relevant bits from a fetched instruction, the result of the comparators and the flags from the four co-processors. This totals nine inputs of which many can take considerable time to be computed. The branch unit in a processor is often the bottleneck and it is important to be able to determine which instruction to fetch as soon as possible and preferably without waiting for results of unnecessary computations to be performed.

CmpEQ

The branch unit takes two inputs from comparators to determine if a branch is required. One of these comparators is a “compare if equal”. This takes two parameters (in this case both 8 bits long) and generates a signal representing their equality. The construction is made of a layer of XOR gates comparing each bit pair and a NOR gate gathering all pairs to generate a single bit result.

MUX

Multiplexers are a very common component and here an eight-to-one multiplexer is used to demonstrate early output instances in such components. One of the eight inputs is chosen to be passed to the output depending on a three bit select input.

Memory

Memory mask logic controls the mask in the data memory load unit which zeros parts of the loaded word. This is used in byte and half word loads as well as rotated partial loads which have addresses on non-word boundaries. Unlike most of the other benchmarks the

logic in this benchmark is relatively deep as many parts of the circuit are also used to generate other outputs.

Random

In order to prove that even circuits with random mapping of output states to the input set have early output cases, a circuit with a randomly chosen output for each input combination was created. Each eight bit input vector corresponds to an output of either zero or one. This effectively generates a 256 bit ROM containing random data. The circuit is constructed as a two level AND-OR structure containing an eight input AND gate for each of the 130 (of the 256) different randomly selected input states to represent a positive output. The results of these AND gates is then passed to a 130 input OR gate. This is not possible on modern design methodologies and both the AND and OR gates would need to be formed from trees. In this benchmark no effort was made to optimise the design.

RandomMin

Synchronous designs often try to decrease the amount of logic used by optimising the designs with tools such as Espresso [44][45]. Espresso takes a circuit specification in the form of a table with the input states and the desired output states and generates another minimised table again with the input and output states but also (if an optimisation was possible) with input states holding a larger number of “Don’t care” inputs and a decreased number of terms. This allows synchronous designs to reduce the size and number of implicants and thus the number and size of gates necessary. In an early output system this also can have a positive effect on early output generation.

The previous benchmark (“Random”) was passed through Espresso and the resultant two level AND-OR circuit was benchmarks. The number of minterms had dropped from 130 down to 41 with an average of six inputs to each minterm.

Shifter

The shifter does a very similar job to a multiplexer. In this case only one of the output bits (bit four) of the eight bit shifter is observed. As well as the three bit shift offset and the 8 bit original value there is a direction (left or right) and an arithmetic bit to trigger sign extension on right shifts. In the case of a right shift in arithmetic mode if the top bit of the input number was one, a one is used when the value is shifted beyond the extent of the input number (rather than the normal zero).

XOR

The final benchmark is the worst case circuit that can be implemented in this design style in terms of its ability to generate early outputs. The XOR gate is the only gate which cannot generate early outputs and is placed here as a control.

6.1.2 Composed Circuits

The circuits were passed into the ‘Early’ [46] program to be evaluated. Early tests the circuit’s ability to generate a result under different input states. Each input can be in one of three states (Zero, One and Null). The entire input state space is 3^C where C is the number of inputs. This means that circuits with more than 20 inputs become very difficult to explore fully as the number of input combinations exceeds 4×10^9 combinations. For this reason all benchmark circuits were limited to 17 or fewer inputs. If data on larger circuits was desired then other approaches such as Monte-Carlo or implicant analysis could be used, but both have weaknesses. The Monte-Carlo method takes random input states and only measures a subset of the full space and thus gives inaccurate results. An implicant analysis method has also been researched. Instead of measuring the circuit’s performance for the complete input set the early output coverage of each implicant can be derived and summed (subtracting the cross section of the implicant with the summed total). This is a fast approach but in XOR based circuits the number of implicants becomes so great the approach becomes considerably slower than the full input space evaluation. The approach could be improved but as all circuits had fewer than 20 inputs the exhaustive approach was sufficient.

After the circuit's ability to generate a result is recorded across the complete input state range, the proportion of input states for a valid result with a varying number of inputs present can be recorded and presented on a graph. Figure 6.1 shows the percentage of input combinations giving a valid result on the Y-axis and the percentage of inputs present on the X-axis.

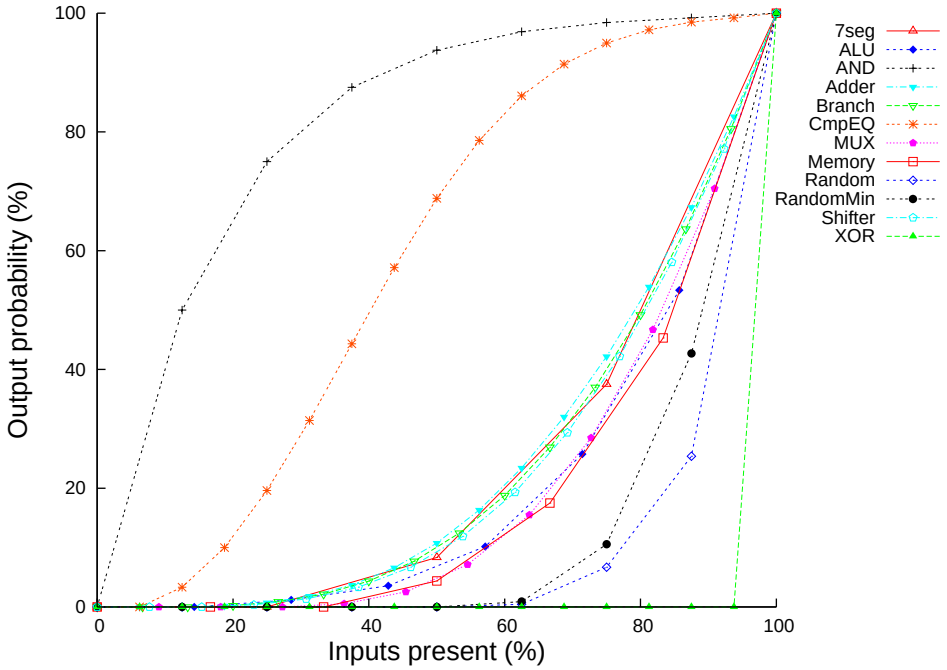

Figure 6.1: Early outputs in composed circuits

Each of the benchmark circuits, with the exception of the XOR, exhibits some early output behaviour. Of the 12 benchmarks, the majority of circuits follow a very similar pattern on the graph despite their differing function and number of inputs.

General trend

Most circuits, despite their differing numbers of inputs and depth of logic, will follow a very similar pattern. This has been observed on many other circuits not presented here and the average number of inputs present before a result is generated is generally between 75% and 85%. For the XOR circuit this is 100% and for the AND circuit it is 25%. The adder design needs on average 78.66% of inputs to be present before generating an output.

Large OR circuits

Both the AND and the CmpEQ benchmarks give favourable results due to their use of large OR gates. The AND gate has a large OR gate to generate the zero result. Both these circuits need just one of the inputs to the gate to be activated in order to generate a result. In the case of the AND benchmark this is achieved by any arriving input being low. In the CmpEQ benchmark the result could be generated by any pair of bits being present but differing. This brings the probability of the function to not be able to generate a result down even with very few inputs valid.

Because it is difficult to observe the function of the circuit in its usual application, a complete input set was chosen to determine the behaviour of these functions. This does mean that the probability of the two numbers not being equal in the CmpEQ benchmark is 0.39% while in an application this may be much higher.

The “Random” Benchmarks

Even with a random distribution of outputs to inputs it is possible to generate early outputs. The random benchmarks have no regular structure. The best performing circuits (such as the AND) have a large regular structure. The generation of early outputs does not rely on the existence of a regular structure (although it is beneficial) but rather on the existence of adjacent input states with the same output state (i.e. don’t cares).

Adjacent input states differ by one input and would be adjacent in a multi-dimensional Karnaugh map. In the random benchmark there is a 50% chance that any two adjacent inputs would have an equal output value. The differing input can be marked as a don’t care in the situation where all the other inputs are present. In a situation like this the desired output has been determined and could be generated. In situations where two of these combined input pairs are adjacent and generate the same result the same strategy can be used to remove yet another input from the care list in that situation. This process can be repeated to determine the full set of early outputs irrespective on the circuit construction. The Early tool can not only determine which early outputs are generated in the composed circuit but also which would be possible using a ‘perfect’ circuit.

6.1.3 Perfect circuits

A perfect circuit is one which captures all possible early outputs. A circuit does not have perfect coverage if upon the arrival of an input the same output is generates irrespective of the data of the input. A perfect circuit would have generated an output without the presence of that input.

The ability of a circuit to generate an output is dependent on its construction and in the random benchmark it is possible to see the effect of a Boolean minimization tool over a simple composition strategy. The graph in figure 6.2 shows the performance of the two Random benchmarks along with the perfect circuit.

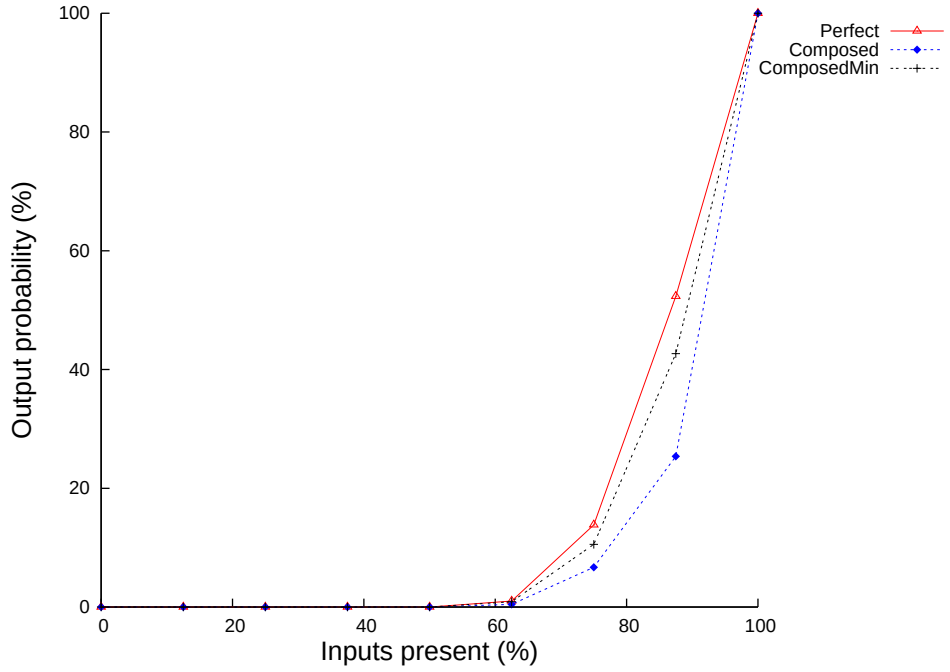

Figure 6.2: Early outputs in the Random benchmarks compared with the perfect circuit

As predicted the probability for a perfect random circuit to generate an output with one missing input is around 50% (actual result is 52.34%). The probability of generating a result with an arbitrary number of inputs missing would $(1/2^{(2^X-1)})$ where X is the number of missing inputs. In the case of two missing inputs that would be 12.5% which again is close to the observed value of 13.84% in the perfect circuit.

Abilities of perfect circuits can be determined for each of the benchmarks and as before these can be plotted on a graph (shown in figure 6.3). This shows the theoretical limit to the improvement in the early output abilities of the circuits.

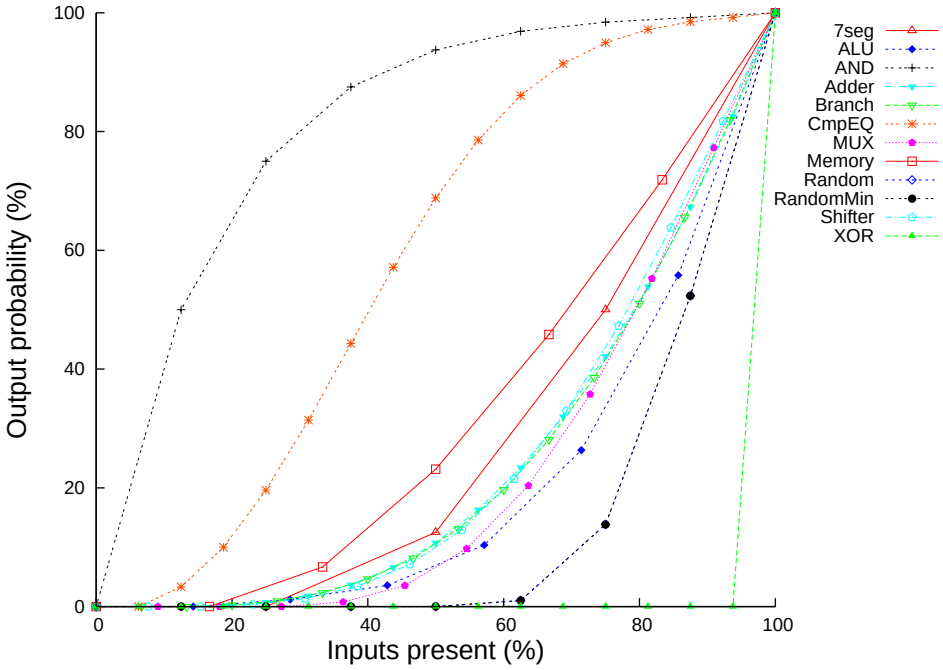

Figure 6.3: Early output cases in perfect circuits

A more important graph would be one which shows the difference between the composed and the perfect circuit in each benchmark. This graph is shown in figure 6.4.

6.2 Attaining perfect circuit properties

A composed circuit's inability to generate all early outputs can be analysed and a better method of constructing circuits can be used to ensure a highly early output behaviour. Each of the benchmarks can be observed and the reason for its non optimal performance can be shown.

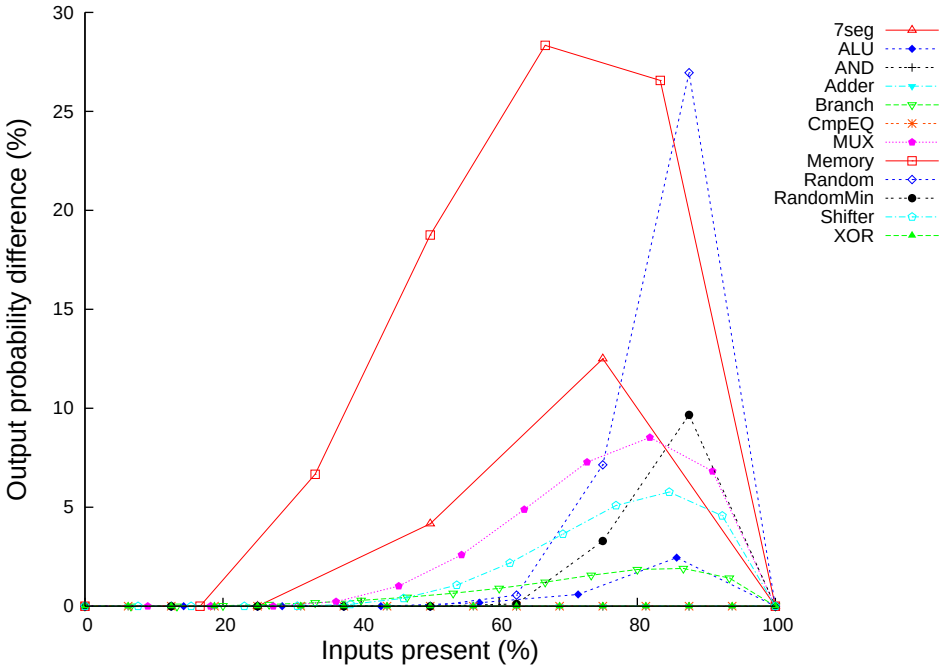

Figure 6.4: Missed early output cases in composed circuits

6.2.1 Optimum by composition circuits

Four of the benchmark circuits gave perfect early output coverages (AND, Adder, CmpEQ and XOR). The reasons for each benchmark's ability to cover fully the complete early output set show which constructions are unable to miss early outputs.

The XOR has the simplest explanation for its perfect performance as it has no early outputs and so none can be missed.

The fundamental components (AND and OR gates) have been designed to capture all the early output cases and using any logic of depth one (composed of a single gate) will have a full coverage. The AND benchmark is composed of a single gate and thus will capture all early output cases. Even if the implementation would be formed from a tree this property would be preserved.

The CmpEQ benchmark uses a combination of a XOR gates and a basic gate.

In order to determine how to achieve optimum early output performance, the un-caught early output states must be examined. To demonstrate a missed early output a simpler

circuit will be used. A 2:1 multiplexer presents as an output one of two inputs determined by a select input. The composition of the circuit can be shown and the reason for the missing early output demonstrated.

6.2.2 Multiplexer missed early output example

The standard 2:1 multiplexer circuit consists of three gates in an AND-OR arrangement. It can be expressed as "(A.$\overline{S}$)+(B.S)" and the Karnaugh map in figure 6.5 shows the cover of the input state space.

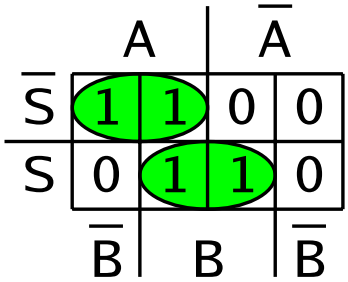

Figure 6.5: 2:1 Multiplexer Karnaugh Map

Using the Early tool it is possible to determine the early outputs missed by the composed logic. In the multiplexer there is one early output not covered by the positive result generating logic. The output of the early tool below shows the complete set of early outputs and marks ones missed by the composed logic.

```
A    B    S    =>  Result
0    0    X    =>  0
0    X    0    =>  0
1    1    X    =>  1 Uncaught
1    X    0    =>  1
X    0    1    =>  0
X    1    1    =>  1
```

The Karnaugh map not only shows the result in all fully valid input combinations but also the early output states for the positive output. Early output cases covered by the positive result generating part of the logic can be seen in a Karnaugh map as any loops with two or more states covered. In this case there are two of these early outputs each allowing the

circuit to disregard one input. This is not the complete set of early output cases as the early tool reports another case not covered by the Karnaugh map or logic. The A.B case is split between two regions and is not fully covered by either. In the case where the circuit has received a one on both the A and the B inputs but has yet to receive a value on S, the output has already been determined but the composed circuit will not be able to generate it. Effectively the circuit is asked to make a decision between the two regions and until enough information has been provided to determine which region will be activated the result will not be generated. This is despite the fact that in either case the result would be the same and the differentiation between the regions is unnecessary.

6.2.3 OR-AND logic

As shown in the multiplexer example, although there was a missing early output opportunity in the positive result generating logic, there was none in the negative result generating logic. The A=0, B=0 case is covered even with S not present. The input circuit to the example was in an AND-OR (Sum or Products) logic style where a set of implicants is generated using AND gates and these are then gathered using an OR gate to generate a result. This logic style allows fast two level logic which is easily minimised using tools such as Espresso. Many of the benchmarks tested were in the same AND-OR style (7seg, MUX, Random, RandomMin and Shifter). Although all these circuits managed to miss some early output cases, none of the missed early output cases were in the negative result generating logic.

In the AND-OR composed circuits the positive result generating logic is made in the AND-OR style while the negative result generating logic is made in the OR-AND style. The OR-AND (Product of Sums) logic does not miss any early outputs and the reason for this could enable us to build logic which will catch all early outputs.

OR-AND logic is composed of a row of OR gates, each of which accepts a number of inputs, the outputs of which are all collected in an AND gate. To prove there is no way of missing an early output in OR-AND logic, the situation of missing an early output will be shown to be impossible.

Circuit implementations where, when a number of inputs have arrived but cannot yet generate an output, yet once an additional input becomes valid, the same result can be generated irrespective of the value in the arriving input denotes the circuit is missing an early output. In OR-AND logic, in order to generate a result, all OR gates must become activated and only then will the AND gate fire and present the output. In order to miss an early output, the logic must achieve a state where an upward transition from one of two wires (forming a single data bit) will activate the remaining (not yet activated) OR gate(s) and thus cause a transition on the output of the AND gate. The two wires in this situation are the zero and one dual rail pair of one of the inputs. The function computed in a case where both wires in the dual rail pair are connected to a single OR gate has no computational value as the result would be consistently one. Effectively a gate in the circuit used for composition would have to take the input and its complement. The behaviour of such gates can be predicted with respect to the mutually exclusive inputs and these inputs (and gates) can be optimised away.

As an example of this the function $A.\overline{A}.B$ was passed into the early tool. The output is presented below.

```
A    B    =>  Values
X    X    =>  0 Uncaught
```

The same result (with a different output value) is derived from the function $A+\overline{A}+B$. Again the output is presented below.

```
A    B    =>  Values
X    X    =>  1 Uncaught
```

Not only is the value along with its complement unnecessary for the computation but all other inputs to the gate are. As these circuits are computationally redundant and are not practical, their non-optimal early output performance is acceptable. With the exception of these circuits there is no other way of missing early outputs in OR-AND logic. These OR-AND logic functions can have a use to ensure an input is valid before a result is generated before generating a result for situations which require the presence of all inputs before a request can be passed on. This technique is used in gathering all address bits before passing them to the memory in section “Microprocessor datapath” on page 121.

6.2.4 Full AND-OR Coverage

It is possible to use an OR-AND structure to implement both the positive and the negative result generating circuits in the simple guarding system (described in section “Loose Guarding” on page 55). Simple guarding, due to its use of timing assumptions during the reset phase, does not impose a restriction forcing each intermediate signal to be encoded in a delay insensitive code. Backward and forward guarding styles ensure the full completion of the computation by ensuring the transition of all intermediate values to a valid state and then returning to zero. Because composing both the zero and one generating logic in the OR-AND style creates a circuit which does not have delay insensitive intermediate values, the approach cannot be used in the forward and backward guarding styles. Instead, either the zero or the one result generating logic must be composed in the AND-OR style. Because the complementary result generating logic would be generated in the OR-AND style, if it is possible to ensure the AND-OR logic captures all early output states, then the full function can be guaranteed to have complete early output coverage. As the OR-AND side will capture all early output states the method need only concentrate on the AND-OR logic half.

The method of ensuring all early output cases are covered in an AND-OR is to add extra regions which cover the remaining cases not covered by any existing region despite the fact they are not necessary to create a result in a fully valid input set. This is demonstrated in figure 6.6 where a third region is added to ensure an early output case is caught.

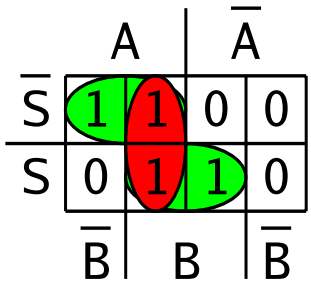

Figure 6.6: 2:1 Multiplexer Karnaugh Map for early output perfect circuit

The full implicant set which covers not just all the minterms but also all terms with don’t care inputs, in logic synthesis is referred to as the “full prime implicant set”. Methods of

deriving prime implicants are part of two level logic minimisation schemes and there are a number of algorithms. To determine the full implicant set to generate an early output perfect circuit, there are two methods presented. The first is based on one of the logic minimisation, prime implicant set generation methods. The second takes advantage of the input being in the form of a circuit which can be turned into a sum of products form without generating a full state space map.

Brute Force Approach

The brute force method is one used in the early tool to show the early output coverage of a particular circuit. It is based on the Quine-McCluskey procedure [47][48] and thus it inherits the disadvantages of the method. It determines the result of the given circuit in all possible input combinations. As stated in “Composed Circuits” on page 86, the input state space is exponential with the number of inputs (3^x where x is the number of inputs). Although (as with the Quine-McCluskey procedure) a lot of these can be ignored, the function needs to be broken down into a complete set of minterms and thus functions with high numbers of inputs can be difficult to process.

For each fully valid input combination, the circuit’s result (0 or 1) is recorded in a table. The pass then runs through all input combinations with one or more not valid inputs in a sequence, increasing number of not valid inputs (firstly all combinations with one missing input are considered followed by all combinations with two missing inputs and so on). For each input state a single Null input (there is guaranteed to be at least one) is taken and for both possible valid states of that input (1 and 0) the result is looked up in the table. If both results are valid and equal then this output state is entered in the table. If either of the states result was Null or if the output values differed the entry is marked as Null. The choice of the invalid input on which to apply the algorithm is unimportant, as performing the operation with respect to any invalid input will give the same result. It can be proven that for a given function C and additional inputs A and B not present in C that $C = C.A+C.\overline{A} = C.B+C.\overline{B}$, as a union of any variable and it’s complement is always equal to one ($X+\overline{X} = 1$ so $C = C.X+C.\overline{X}$). Thus performing the operation on either A or B gives the same result (C).

Figure 6.7 shows the pass being performed on the 7seg benchmark. The first of the four tables shows the Karnaugh map of the circuit. Because Karnaugh maps can show the output of the circuit only with a fully valid input set, several other tables are needed to show the complete input state space. The second, third and fourth tables show the desired output state of the circuit with one, two and three of the four inputs missing respectively. Each table is generated from the data of its predecessor. Each of the entries in the second table is generated from two adjacent inputs in the first table (the Karnaugh map). Although the last three tables are represented in a style to match the positioning of the Karnaugh map, they do not follow the Karnaugh map rules of grouping.

As an example of the insertion of data into each table, the A.B entry in the third table (2 inputs missing and marked in green) is taken. There is a choice of which of the remaining not valid inputs (C or D) to use to look up the values in the previous table. In either case (input C shown in red or input D shown in blue) the same result would be reached. The last table (three inputs missing) has no valid states as a result cannot be generated with just one input present.

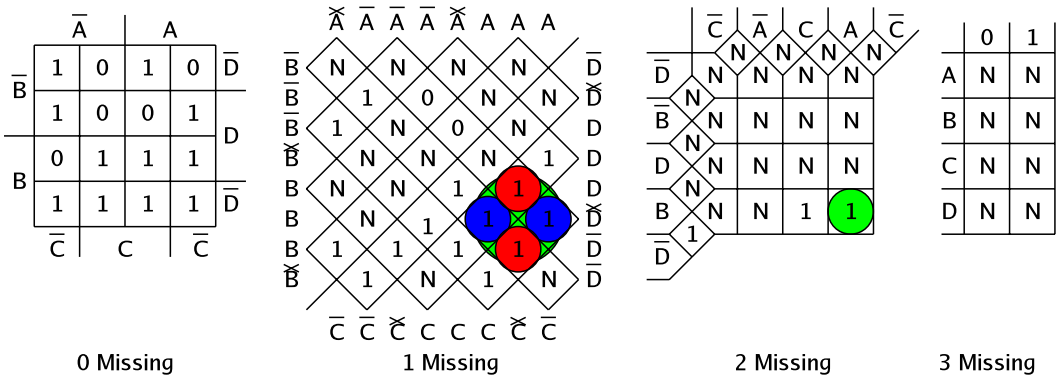

Figure 6.7: Brute force complete early output set generation phase one

The second pass considers all possible input states, this time running from the greatest number of missing inputs to the least, for insertion into the final list of implicants. Implicants are only inserted if they are not a subset of an existing implicant in the list. This process is shown in figure 6.8. The first table cannot contribute any implicants but the second table has three. It must be stressed that normal Karnaugh map rules do not apply and the number of states covered by an implicant changes from table to table. The

implicants from the previous table are then marked on the next table (shown in green) and any states not covered are added to the list.

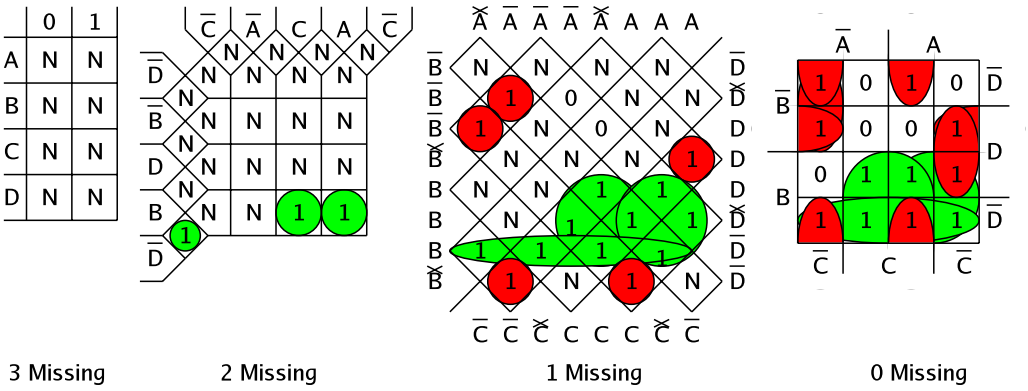

Figure 6.8: Brute force complete early output set generation phase two

This strategy is simple to implement and, as a by-product, determines figures for the early output abilities of the perfect circuit across the full input space. Unfortunately, even on optimised code, this approach takes several minutes on an 18 input circuit and the time triples with the addition of each input (complexity $O(3^x)$). This makes the re-synthesis of large circuits infeasible. The generation of the perfect AND-OR early output cover can, however, be achieved without analysing the result of every possible input combination.

Analytical Approach

The analytical approach avoids the complexity explosion of the brute force method. Unlike the brute force method the analytical algorithm requires a circuit in the AND-OR form as an input. The first (and most complex) step in the method is to flatten the circuit into a canonical form (AND-OR structure) composed of a set of implicants. This does not necessarily have to be a minimal or an optimised set as different implicant sets with the same coverage will yield the same result.

Once the desired function is represented as a set of implicants, the algorithm can be applied. As described in “OR-AND logic” on page 93, a missed early output exists in situations when there exists an input state where the arrival of an additional input will

generate the same valid result for all possible input states. To ensure that this case is caught, the process needs to know the input for which the early output was missed and the state of the other inputs in that situation.

The algorithm to capture all remaining missed early output states is based on the fact that a missed early output exhibits itself through a situation where an input can arrive in a given circuit state and upon presenting either value (1 or 0) to the function the result will be the same. For this reason each input is tested for this occurrence. For each input the implicant set is divided into three parts. The zero and one sets where the desired value for the input is zero or one respectively, and the don't care set where the value of the input is not necessary for the generation of the output. The intersection of the zero and one sets (common areas in the two implicant sets) is taken and merged with the original set. This gives the areas of the circuit where the input was not necessary to determine the output of the function. This is done for each input and the whole cycle is repeated until no new implicants have been added into the set.

To demonstrate the process in more detail the algorithm will be shown applied to the 7seg benchmark. The original set of implicants are listed below. This forms the master set to which additional implicants will be added.

$$A.B + B.C + \overline{B}.\overline{C}.D + A.C.\overline{D} + \overline{A}.\overline{C}.\overline{D}$$

This implicant list is then broken up into the three sets with respect to one of the inputs (in this case A):

$$0: \overline{A}.\overline{C}.\overline{D}$$
$$1: A.B + A.C.\overline{D}$$
$$X: B.C + \overline{B}.\overline{C}.D$$

The Zero and One sets, can now be merged to find the common areas in both sets with respect to A. To achieve this, the sets are ANDed together to generate a list of implicants which exist in both the One and the Zero sets. Before this is done, the zero and one sets

of implicants are divided by $\overline{A}$ and A respectively to report only the common regions with respect to A.

```
  (implicants with A) . (implicants with Ā)
  (C̄.D̄) . (B + C.D̄)
= B.C̄.D̄ + C̄.D̄.C.D̄
```

Of the two resultant implicants, only one can occur (B.$\overline{C}$.$\overline{D}$), while the other ($\overline{C}$.$\overline{D}$.C.$\overline{D}$) is the result of finding the intersection of mutually exclusive regions and can be removed from the set at this stage. The set is then merged with the master set. In order to add the implicant to the function it must be ensured to be prime (not a subset of an already present implicant which could only exist in the don't care set). Should any inserted implicant make any already present implicants redundant (by being their superset) these should be removed from the master list. In this case the implicant is neither in the subset of any implicant in the don't care set nor a superset of another implicant in the master set, and thus is inserted into the original function. If it is a subset of an implicant in the don't care set then it would not be inserted. If it is a superset of an implicant (or a number of implicants) in the master list then it would replace these entries in the original function. This is demonstrated when the algorithm is applied with respect to C, where the intersection between the newly added B.$\overline{C}$.$\overline{D}$ implicant and B.C from the original function (with respect to C), generates an implicant (B.$\overline{D}$) which is the superset of B.$\overline{C}$.$\overline{D}$. The new implicant would replace the subset implicant in the master set.

After the method is applied with respect to all inputs and no new implicants are added to the function the process is then complete. The final generated circuit adds three additional implicants into original function ($\overline{A}$.$\overline{B}$.$\overline{C}$ + A.$\overline{C}$.D + B.$\overline{D}$). The complexity of this method is approximately $O(X^2)$.

6.3 Early output function used in larger circuits

The sequencing of early output circuits is less uniform than DIMS style designs due to the data dependent timing in the result generation. Although this behaviour has been shown to be beneficial in a single pipeline stage, the ability for a larger circuit to take advantage of early output schemes will be demonstrated. First the early output circuit operation will be demonstrated in detail in order to explain its behaviour, then larger and more realistic

circuits will be assembled and their ability to use the methodologies described in earlier chapters will be presented.

6.3.1 Early output demonstration

To demonstrate and observe the behaviour of early output systems, an example will be used to illustrate different aspects of the methodology.

Decrement circuit

The circuit chosen to demonstrate the properties of the early output systems is a decrementer. The unit decrements its internal value until it reaches zero, the next value is then loaded from the input; in this case the input is an external constant. The unit can be used to count a specific number of cycles and output values dependent on the state of the internal value e.g. counting the operations in a cyclic divider. In this design no outputs are generated in order to enable the circuit to function without being connected to an external test-bench circuit. The constant can be fixed to any value and different values give varying performance in different design styles. The pseudo code of the function is shown below.

```
a = c = 255     // or another cimput constant
while (true) {
        if (a≠0)        a = a - 1;
        else            a = c;
        }
```

Figure 6.9 presents the register level implementation of the decrementer circuit. The green boxes show the positions of half latches.

As well as the half latches shown in the figure, there are additional latches on the carry path in the decrementing unit. This breaks the decrementer into 32 small stages with few inputs and outputs. This method was explained in section “Vertical pipelining” on page 42.

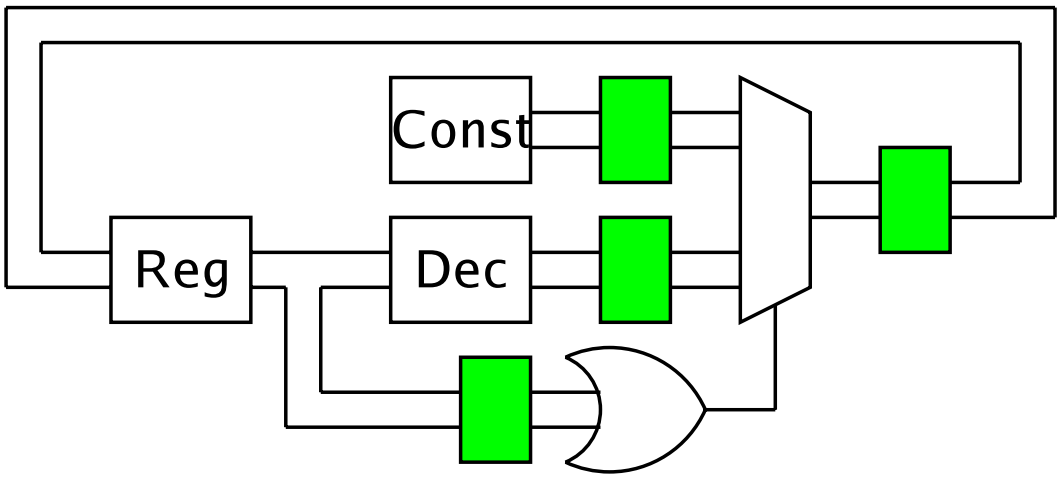

Figure 6.9: Decrementer register level design

S2A tool

A tool to take the specification netlist and convert it to any of the dual rail design styles described in the previous chapters was developed. The available back-end implementation styles are:

- Delay insensitive minterm synthesis (DIMS)
- Backward guarding early output
- Forward guarding early output
- Loose guarding early output

The system can not only generate an output netlist but can also simulate the circuit to determine its relative performance.

The performance figures throughout the rest of this chapter will be in terms of the number of operations executed in a 100,000 gate delay interval.

6.3.2 Circuit operation analysis

Many factors can vary the operating speed of a circuit and as an example the value of the constant in the decrementer example was varied to demonstrate its effect on the different design styles. Figure 6.10 shows the performance of each circuit with different values for the constant being read in.

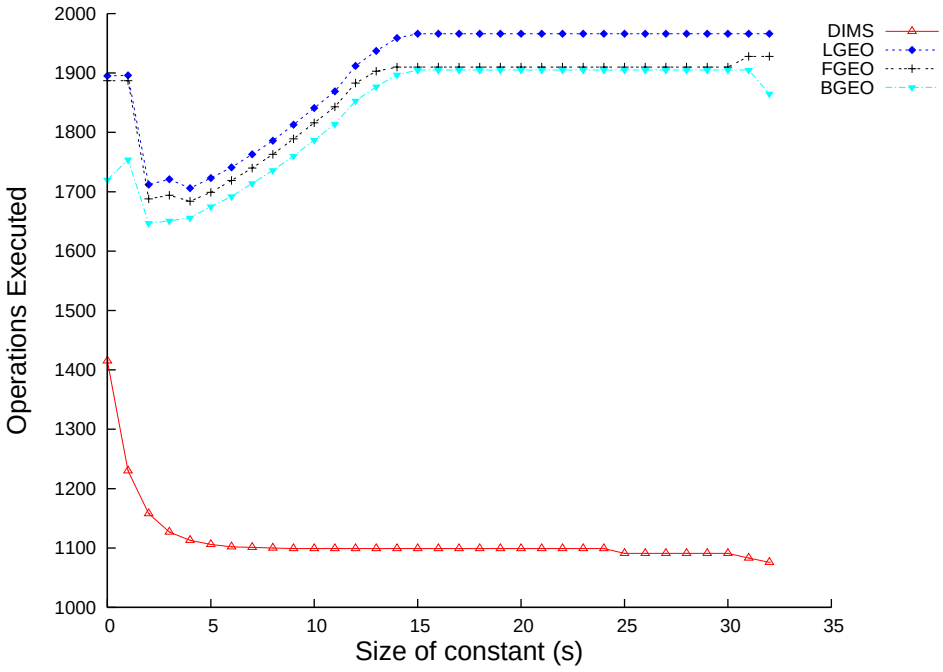

Figure 6.10: Cycle count variation effect on operation speed

The early output versions have similar behaviours and have the same factors affecting their performance while the DIMS version has different factors affecting its performance. In this analysis, inputs with sizes from 0 to 32 are used by presenting constants with values of 2^S-1 (i.e. 2^0-1=0 to 2^{32}-1=0xFFFFFFFF).

DIMS circuit

DIMS circuits have no early output states and so their performance is generally quite independent of the data operated on. This can make the DIMS logic style useful for the generation of secure systems which behave identically irrespective of the data being

executed. Simply using the DIMS approach does not give a fully balanced system as there are two paths through a DIMS gate (see figure “2 input DIMS OR gate” on page 40).

Imbalances are visible in figure 6.10 with the variation in the operating performance with respect to the loaded constant value. The worst case path in the circuit passes through the full length of the borrow chain of the decrementer unit, composed of AND gates which propagate the signal. These AND gates have two paths through the gate with different delays. The short path through the gate is the situation where the borrow input is high and the current bit is low, this propagates the high borrow signal to the next bit. The long path through the gate propagates or generates the low borrow signal to the next stage. The length of the high borrow propagation is equal to the number of leading low inputs (from the least significant end) in the input number. This is normally very short and in 50% of cases it is zero. For a random number the length of the high borrow chain is one (50% of the time it is 0, 25% of the time it is 1, 12.5% of the time it is 2 and so on, averaging to 1). In the case where the input number is zero the borrow chain will propagate all the way up though the width of the unit.

In the DIMS design, the length of the high borrow chain has a direct effect on the delay of the unit. For large numbers which do not reach zero during the simulation (2^{11} and above or point 11 on the graph) the length of the high borrow chain is on average one. For the number zero, as the constant in the circuit, the length of the borrow path would be the bit width of the unit each time. For input sizes between 0 and 11 the average high borrow chain length is the average delay plus the frequency of the occurrence of the zero delay times its additional length from the length over the average.

The graph in figure 6.10 shows the performance of the system matching the explanation above. The shortest delay through the unit occurs when then constant is zero. As the constant increases so does the delay and the performance drops. The effect is decreased until no longer visible past the point when the size reaches 10.

The 32 input OR gate (a≠0 unit) comprises two layers of four input OR gates and one layer of two input OR gates. For this reason the introduction of high bits into the top eight or top two bits causes additional delay as the longer paths are taken through these gates.

The DIMS circuit performs best when there is a worst case borrow propagation and, in the case of the large layered OR gate, in situations where no early output would be possible. Not only does this create slow circuits but the change in performance of these circuits with different inputs is counter intuitive to the designer.

Early Output circuits

The first property of early output circuits visible on the graph is their speed relative to the DIMS designs. The second important feature is the different effect of the size of the constant on performance. The early output plots on the graph can be broken up into three parts representing the areas where three different factors have the primary influence on performance. The first area is between s=0 and s=4.

With small numbers in the decrementer, the probability of the number being zero and a new number being loaded is relatively high. The early output circuit can take advantage of this and, even though the full decrement operation being performed on zero takes a relatively long time, the result can be dismissed and instead the constant is loaded into the register. This part of the graph looks similar to the DIMS line but it is for different reasons. If the result of the addition was required when the input was zero this would be the lowest point on the early output line.

In the area from five to 15 the performance increases in a linear manner with the size of the constant. This is due to the borrow chain in the long string of low bits above the constant rippling the borrow signal. The signal cannot be determined without the input from bits lower down. Units with a high input bit can determine the borrow out but due to the lack of these in the upper bits the delay of the long path has a direct impact on the performance.

Constants from 15 bits long and higher are unable to continue the trend of increasing performance as the path through the decrementer is no longer the slowest path through the system. Instead a path elsewhere in the system is the obstacle in the performance.

As shown, the effects on the early output designs are different from those experienced by the DIMS designs. Although the performance is greatly increased over the DIMS design,

the limit on the speed demonstrates that the circuit could perform much faster if another slow path was removed.

6.3.3 Slowest path

Synchronous circuits have a simple global clock system where all signals are synchronised to latches at each cycle and emerge at the next upward clock edge. This makes determining the slowest path in the system trivial. Asynchronous logic does not have a system where the data must propagate from one point to another in a fixed amount of time. Instead the execution of the entire program can be considered as a path of transitions from one point in the system to another. The start point would be the release of the reset signal and the end point would be the signal which indicates the completion of the task. Like the synchronous system, the shorter this path, is the faster the system will perform. Unlike the synchronous model the asynchronous slowest path can pass through any element or wire several times. This makes a simple time-stamping approach to determine the worst case, such as that described in [40], impractical.

As demonstrated by the performance cap on the decrementer design, in early output logic the path of the slowest route is not always clearly visible. Additionally, as synchronous methods are not applicable, a novel method must be developed to tackle this problem. A dynamic approach, blame passing simulation, to determine the slowest path will be presented and its use in optimising the example design will be demonstrated.

Blame passing simulator

The blame passing simulator allows circuits to be simulated in a test-bench environment and the slowest path to be extracted. Although static timing analysis is not used in this approach, some aspects of it must be understood as they also form the basis of the dynamic method.

The static approach runs through the input netlist marking the arrival time of signals. This process starts at the outputs of the storage elements (flip-flops) which are marked as time zero. Any gate with all its inputs marked can then mark its output as the last signal to arrive plus the delay of the gate. Once completed the net with the latest arrival time is

marked as a point in the slowest path. The last input to arrive at the gate generating this signal is marked as the next point in the path. The process is repeated until the input flip-flop is reached. More complicated approaches such as holding the time stamp for both the up-going and the down-going edges or even taking account of cross-talk can give more accurate results. The major disadvantage of this approach is its inability to handle non uni-directional circuits due to cyclic dependencies. Secondly the approach can often find a slowest path which cannot occur due to some signals being mutually exclusive.

An example of this would be a circuit with selectable pre and post increment units attached. Only one increment unit would be activated and the other is bypassed. Unfortunately the worst case path would assume both are activated. The alternative of having a single unit, which can be multiplexed into place in front or behind the function, would introduce cyclic dependencies.

Blame passing analysis works using a similar strategy to the second phase of the static timing analysis. Starting from a transition which indicated the completion of the test program, the gate which caused this transition then looks at the input which caused it to flip its output. This process is repeated until the initiation signal for the circuit is reached (this is usually the release of the reset signal). This approach requires the simulator to record the cause of each transition in the circuit throughout the simulation. Such a record would be very large and only a small portion of it is relevant.

To allow longer simulations, the blame path for each transition is calculated during the simulation and only the relevant paths are stored. Non-critical paths are freed allowing only useful information to be stored. Each new transition during the simulation allocates a record which is marked with the cause of the transition and a reference count. The cause transition gets its reference count incremented. Should a transition be unable to cause any gates to transition their output (the reference count remains at zero after all gates it feeds to have been processed), the transition record is freed and its parent's transition reference count is decremented. Should the parent's reference count also drop to zero the process is repeated releasing a thread of transitions which accumulate to a non critical path. As the product of these transitions has arrived to all gates it affects early, it does not need to be optimised and is of no interest. The advantage of this approach is that it looks at the

average case operation (as described in the test-bench). It cannot determine the worst case operation.

The blame passing extension to the simulator has a small impact on its speed (30% decrease) and gives a good view of system execution. To demonstrate the method, a simulation of the decrementer circuit was conducted. The constant chosen for the simulation was the full width of the pipeline (32 bits) as, in this setup, the circuit performs much slower than expected. The extracted slowest path should present the cause of the poor performance. To allow easier analysis, the path can be superimposed on a diagram of the design. To generate such a diagram the coordinates, on the schematic, of all wires in the design can be fed to the simulator which then generates a list of vectors which can be plotted. In this case, as the circuit was not designed using a graphical tool, the coordinates for the wires were manually generated. Figure 6.11 shows the path for the simulation.

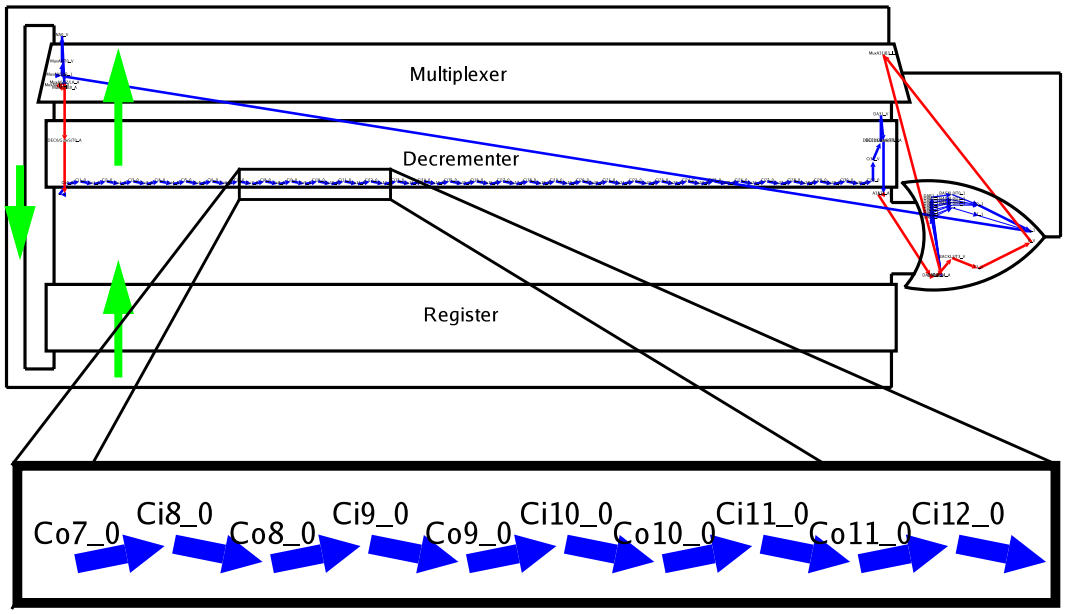

Figure 6.11: Slowest path of the decrementer example with zoomed segment

The blue and red lines represent the up and down transitions of the signals respectively. The width of the line shows the frequency of a transition's occurrence in the slowest path. The details in the diagram are not important and the significant part has been enlarged. In addition to the diagram, the simulation data can be used to generate a table reporting the

proportion of time spent in each unit (or set of units) in the system. In this case 77% of the slowest path transitions occurred within the decrementer unit. A closer inspection of the figure or the log file shows the presence of a long chain of transitions which take up the majority of the cycle time. The path follows the ripple carry chain along the zero request path. When all the input bits are high, the carry out can be generated locally and the critical path does not stretch from the bottom bit but this path is not on the positive transition but rather the negative one. Although using early output gave the advantage of being able to generate the results without waiting for the full ripple carry to reach every bit, the reset phase still requires a full completion to be performed. This forces the highest bit to observe the data being released by all inputs. As the carry zero output is driven by an OR gate of the carry zero input of the previous stage, the output will remain high until the carry input has been released. This dependency then stretched all the way to the lowest bit and causes a full ripple to be performed.

Once the cause of the delay has been determined, an appropriate optimisation can be applied to correct it.

6.3.4 Circuit optimisation

Due to the non globally synchronised operation of asynchronous circuits, their behaviour cannot be broken down into single clock cycle segments and analysed independently. Without applying very restrictive assumptions it becomes difficult to analyse the circuit operation using a static approach. Instead the optimisation system which will be described uses the information from dynamic timing analysis.

The full system breaks into three parts:

- Analysis: observes the circuit and identifies its weak points.
- Optimisation: determines changes which could be beneficial
- Re-analysis: applies a prospective optimisation and observes the circuit performance

This becomes an iterative process as each optimisation changes the behaviour of the circuit which then needs to be re-evaluated before the next optimisation is applied.

Example optimisation

The example decrementer circuit's slowest path determined in the previous subsection can be used to find the most appropriate optimisation. As described before, the main delay in the slowest path is in the decrementer carry chain on the downward transition. This downward transition, passing through a latch along the request (data) wires, is indicative of a position where an 'early drop' latch can improve the circuit's performance. The full table of possible optimisations and the slowest path route which indicates where each use would be beneficial will be shown in the subsection 6.3.4. In this case early drop latches are likely to be better suited in the place of the carry propagating half latches. Replacing any of these latches will change the behaviour of the circuit resulting in an circuit where the slowest path no longer takes the same route. Blindly applying all the recommended changes suggested by the slowest path, in a single step, can yield a much poorer circuit.

It is a good idea to apply only one of the possible optimisations before re-analysing the circuit. This is a demonstration of one of the weaknesses of the approach and the reason why a fast simulator is necessary. The simulator can evaluate each of the possible optimisations to determine which have the most positive effect and then once it has been committed the cycle can begin again. In this case there are 31 possible optimisations in the carry path alone.

To demonstrate the effect of some of the optimisations, three were plotted on the graph against the decremented constant number as shown in figure 6.10. Figure 6.12 shows the performance across the different numbers being operated on across three possible optimisations along with the original circuit. The three chosen optimisations shown on the graph are: replacing the latch on the carry-out of bit-slice 2 (LGEO_SD2), 16 (LGEO_SD16) and 24 (LGEO_SD24) along with the original circuit (LGEO).

In the LGEO_SD16 optimisation the early-drop latch is placed (approximately) in the middle of the carry chain. This gives good balance, effectively cutting the delay caused

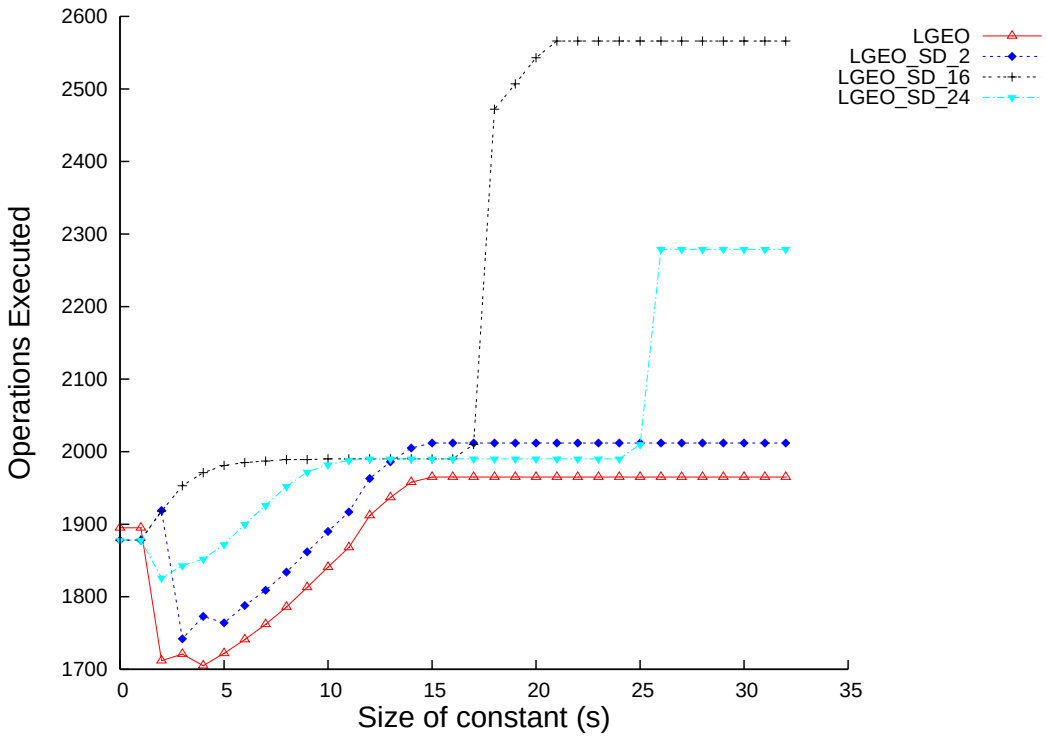

Figure 6.12: Analysis of possible optimisations

by the release of the carry chain requests in half. This only happens once the constant has reached a size where the input values are high beyond the latch. With constants reaching the newly placed latch, the carry chain can be interrupted and the release of the carry chain can begin from half way across the decrementer. This improves performance but, with small constants, the lower part of the decrementer still takes a long time to complete. The upper part then will not get the carry in it requires to complete and start the release of the carry. When constants with high bits in the upper part of the decrementer are used the upper part can start the next computation cycle independently, bar its bottom bit which still needs to interact with the lower half.

The LGEO_SD_24 optimisation gives much poorer results as it requires the constant to be very large before the jump in the performance and because the latch is placed towards one of the ends the delay caused by the release of the valid signal along the carry chain is not cut in half. Instead the path is shortened by eight stages.

The placement on bit slice two (in LGEO_SD_2) is also a poor choice. This not only does not shorten the carry drop path very much but also interferes with a frequently used part of the decrementer by adding an extra delays in the critical path.

Although some of the factors contributing to the performance of the circuit under different conditions have been explained above, the full range of influences is too large to examine exhaustively in order to explain every point of inflection in the graph. It is equally difficult to determine if an optimisation will have a positive effect on the performance. For this reason each potential optimisation must be simulated to ensure that it improves the performance before it is committed to the design. Although each of the optimisations shown was beneficial with the benchmark parameters applied, each optimisation has a negative effect in the range of the input constant length between zero and one. The addition of the early drop latches causes additional delay in the positive edge carry propagation which, in the case of the zero and one constants, is a primary contributor to the slowest path.

The importance of having a correct benchmark circuit can be seen in figure 6.13 where the circuit's carry path was optimised repeatedly with different benchmarks used to generate the slowest path and observe the post optimisation performance. The three benchmarks used decrement constants of length 0, 16 and 32. Also shown on the graph is the original circuit performance to demonstrate how some optimisations can have a detrimental effect on the circuit in situations not executed by the benchmark.

The primary optimisation used on the circuit with the zero length constant was the removal of latches from the carry path. The 32 length constant circuit mostly replaced the carry latches with early-drop latches. The 16 length constant optimisations placed semi-decoupled latches on bits below 16 and removed latches above 16.

The removal of latches is yet another possible optimisation. It is used when the slowest path passes through a latch from request in to request out in the positive direction. This is common with small numbers where the carry propagates along the full length of the unit. The removal of this latch removes a gate delay in the slowest path but can result in another slowest path being formed due to the pipeline stage being merged with another. In the case

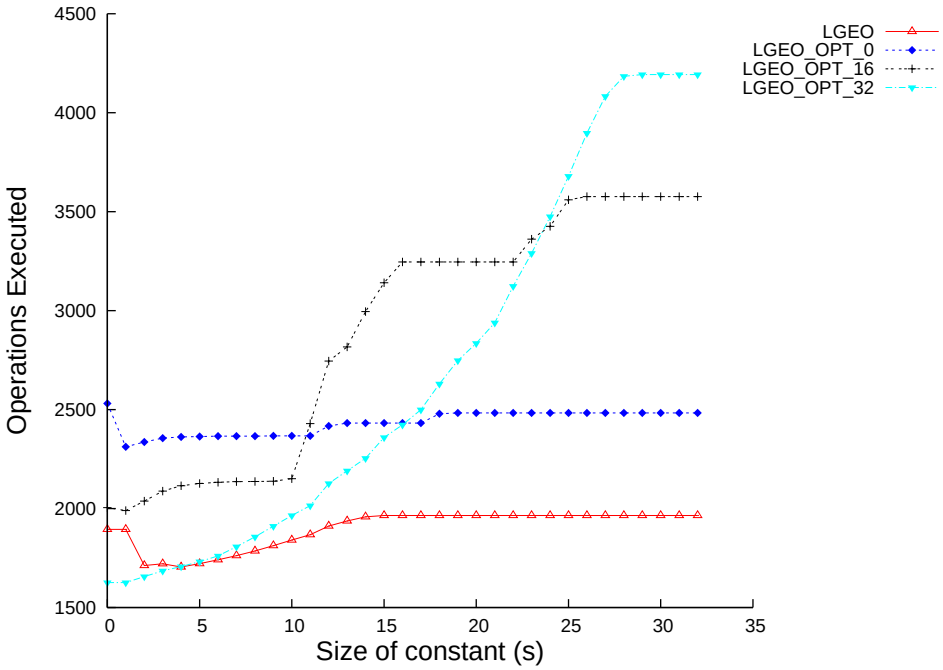

Figure 6.13: Performance of circuits optimised using different benchmarks

of the decrementer, vertical pipelining to a granularity of one bit forms stages which are small enough to be merged and not reduce performance.

6.3.5 Optimisation tables

Relevant optimisation to be applied to a circuit can be derived from its slowest path. A peephole style optimisation can observe frequently taken paths and apply the most suitable optimisation. Figure 6.14 shows the most frequently used optimisations. The effect of these options will be demonstrated later in this chapter when applied to a number of circuits. The table is laid out to show on the top row the path which when matched will replace the affected components with the design below. The red lines represent positive transitions while the blue ones represent the negative transitions. Not shown on the diagram but equally important is the pattern which predicts if the optimisation will have negative effect on the performance. Each optimisation and the reasoning will be presented in section 6.4; the performance increase due to each will be evaluated.

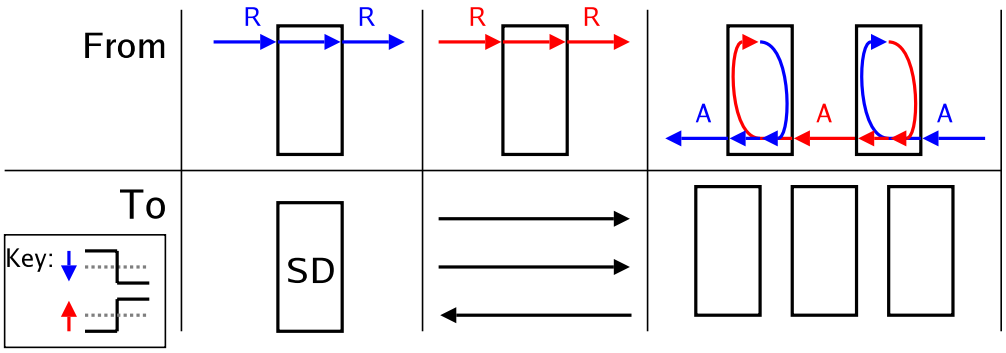

Figure 6.14: Table for early drop, latch removal and insertion optimisations

Early-drop latch

The use of early-drop latches allows the release of the data values before the data in requests have dropped. To spot a latch which could be replaced with an early-drop version, the pattern of request in dropping to request out dropping must occur frequently in the slowest path.

The early drop latch does have the disadvantage of an additional gate delay in data propagation. In some situations, despite the frequent activation of the path to be matched, the use of an optimisation can have a negative effect. These can often be predicted using slowest path pattern matching. The path which can signify a position where the early drop latch optimisation is likely to have negative results is the positive transitioning request out signal caused by either a request in going high or an acknowledge being released. In these cases the slowest path is increased by one gate delay. This is only a minor increase in the delay compared with the removal of the data released path and must occur much more frequently than the request out falling path to have a greater effect.

Latch removal

Often latches can have no effect on performance in situations where the latch is not necessary due to the system cycle time being larger than the combined cycle time of the two stages the latch connects. Their presence does not increase the effective pipelining of the system as the two stages the latch connects can be very small and in a free running

system the presence of these latches does not increase the number of data values being worked on at any one time. The latch can, however, contribute to the slowest path which does add a C-element delay to the path each time it passes through the latch. The removal of latches is one of the most difficult optimisations from a performance prediction aspect. This is due to the observation that, in perfectly balanced and optimised circuits, the slowest path is likely to pass through many latches in this exact manner. Secondly, there is very little information from the slowest path to determine if the latch is useful in the system and whether its removal will have a dramatic negative effect on the performance. One of the hints that shows the latch is useful, is the slowest path passing through the acknowledge signals of the latch. The removal of a latch which has the slowest path passing through the acknowledge signals will cause the path to extend to the next closest latch (from the other stage which has now been merged).

Latch insertion

The correct level of pipelining is difficult to predict by the designer and automatic slack-matching methods are common place in other tools [49][50][51][52][53]. These tools use static timing analysis to determine the need for extra latching. A simple latch insertion technique would be to take datapaths with single start and end points. These would have their number of latches compared and the path with fewer latches receives more to allow both pipelines to be fully occupied rather than one pipeline's full token occupancy to force the second pipeline to be starved. This can be seen in figure 6.15 where the top pipeline is starved of input due to the bottom pipeline being full. The slowest path in such an example would run along the acknowledge path in the bottom pipeline from the output latch, then along the request signals of the upper pipeline.

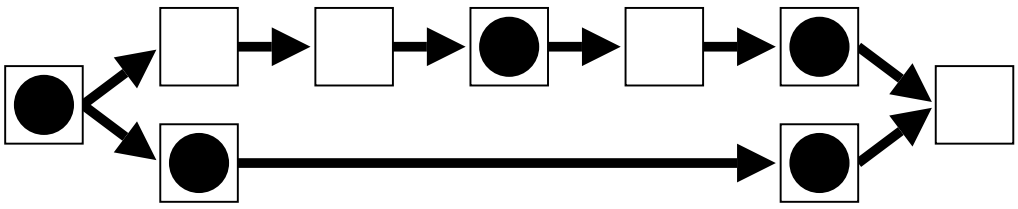

Figure 6.15: Not slack matched pipeline

This static approach may be effective in this example but, here, the dynamic optimization approach of the blame passing simulation analysis can reach the same results with more accurate placement of latches. One of the issues with slack matching is the position of the new latches. These are usually evenly scattered throughout the shorter pipeline. The dynamic approach has the ability to pinpoint the exact point where the extra latch is needed (or if it is needed at all).

The slowest path route, which signifies a beneficial latch insertion position, passes from the acknowledge-out of a latch though the latch and out though the acknowledge-in pin. This can be with either the acknowledge rising and releasing the data out and consequently dropping the acknowledge-in, or the acknowledge-out allowing new data to be latched. These paths often occur in series in designs which are latch bound (not enough latches are present to allow free token flow). The latch insertion does add an extra gate delay in the data forward propagation and so the optimisation can gave negative results if the forward propagating request-in to request-out path (in either the positive or the negative direction) occurs often in the slowest path.

Anti-token latch

The last optimisation demonstrated in this chapter is the anti-token latch. The path is shown in the top part of figure 6.16 along with the description of which latch design it should be replaced with.

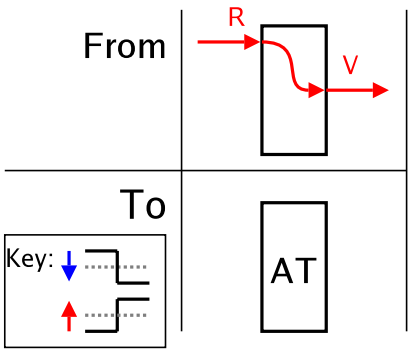

Figure 6.16: Anti-token latch optimisation

Anti-token latches generate the valid-out signal early. This allows it to receive an acknowledge before transmitting any data. The advantage of doing this is the latch does not need to wait for data to be presented to it before the validity is generated and the cycle is completed. The path to be matched thus flows from the request-in to validity-out of the latch. The anti-token latch has many transition sequences where it performs more poorly than other latches and it does rely on a larger set of timing assumptions being made. These were discussed in the previous chapter and, although it is possible to define and uphold them, due to the added complexity in the placement and routing of the design, it is generally favourable to avoid the use of anti-token latches where they do not add to circuit performance.

The four optimisations shown have been studied and their benefits will be shown at the end of this chapter. Additionally to these there are a number of optimisations which were found but whose benefits have not yet been fully explored. These optimisations exist below the architectural level and require further study to examine their effectiveness. The three additional optimisations are presented in figure 6.17.

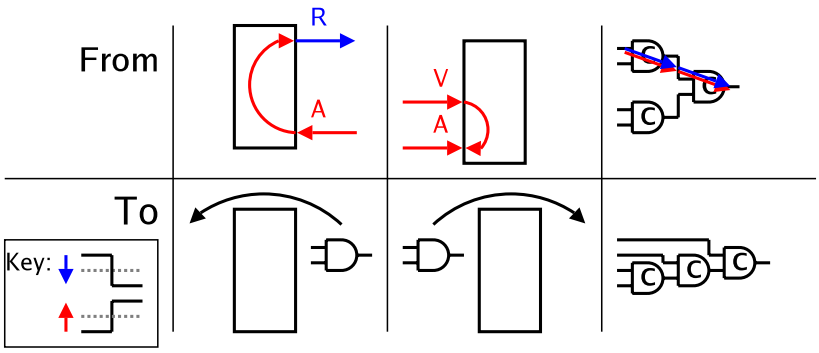

Figure 6.17: Table for retiminag and tree reshaping

Retiming

Retiming [54] in synchronous designs allows stages of the pipeline to become balanced and increase performance by moving gates from a deep stage to a shorter neighbouring stage. Imbalance in stage sizes is also a problem in asynchronous designs. The slowest path can be used to spot stages which, if shortened, would yield higher performance. In an asynchronous system, where the performance is bound by the speed of a stage which

is slower than any other, the slowest path will run up and down the stage propagating along the request/validity signals and back along the acknowledge path. Because all neighbouring stages are faster, the data is always ready to be accepted by output latches and new data is always ready to be consumed at the input latches. In such a system the slowest path is positioned along the acknowledge network until it reaches the input latch, and because the acknowledge is always the later to arrive, (rather than the data in) the path then continues along the request/validity wires. Once it reaches the output latch, because the data is always ready to be accepted, the slowest path then returns along the acknowledge network.

The first two optimisations in figure 6.17 spot situations where the stage could be shortened by pushing gates through latches in the cases where the slowest path exhibits behaviour common with that of large stages. This optimisation does have the disadvantage of changing the circuit so, when observed in simulation, it no longer directly reflects the original layout of latches placed by the designer. Secondly the pipeline stage borders are often placed by engineers at the points in the datapath where the number of signals to be latched is lowest. An automatic retiming algorithm can increase the number of latches needed by placing them in inappropriate places.

Tree reshaping

The validity and acknowledge networks in early output designs are formed from large trees of C-elements. These are not balanced as some networks need to adhere to the race conditions and leaving them in their directly translated state makes it easier to uphold these timing assumptions. Some trees (such as acknowledge networks) do not have timing assumptions placed upon them which prevent the trees from being balanced. Tree balancing is a well researched topic in software engineering where keeping all paths to nodes roughly equal length allows the tree to have a shorter average length than an unbalanced tree. This is a reasonable strategy if the likelihood of requesting each node was roughly equal or this probability was unknown. Better systems, such as the Huffman encoding trees [55][56], allow the more frequently used nodes to be placed closer to the root of the tree.

In early-output circuits the slowest path frequently passes through a tree of components. The tree could be restructured to allow this path to become shorter at the penalty of other inputs having a longer path (as demonstrated in the third optimisation in figure 6.17). This is often not a problem as these inputs will probably arrive far ahead of time. If this is not so, and the restructuring moved an input to become a part of the slowest path, then this input becomes a priority and also gets moved closer to the root of the tree.

6.4 Large design demonstration and analysis

The abilities of the optimisations and design methodologies presented will be shown across a number of designs and test-benches. The performance of three benchmark circuits will be examined across different execution parameters for each. Each of these benchmarks will then be optimised with different sets of optimisation rules to allow the benefit of each to be demonstrated. The early output designs will then be compared with the designs made in alternative design methodologies: synchronous, bundled data and DIMS.

Firstly each of the circuits and their benchmarked modes of operation will be presented.

6.4.1 Benchmark designs

Three designs of varying complexity were chosen to demonstrate the performance of the early output logic, anti-tokens and optimisation scheme which makes use of different latch designs. Each benchmark also has two modes of operation upon which the optimisations will be based.

Decrementer

The decrementer circuit was presented earlier in this chapter. Two modes of operation use the input constant values of 0 and 0xFFFFFFFF. The two different modes of operation place very different demands on the circuit and will generate circuits unsuitable for execution in the opposing mode of operation.

Greatest common divisor

The GCD example design has two registers (A and B) which contain values of the two numbers for which the greatest common divisor is to be calculated. With each iteration of the functional unit, the registers get updated with the values of their remainders after being divided by each other. This is done using two divide units which throw away the result of the division and feed out only the remainder. The values of the registers are also tested to see when either reaches zero in which case a new set of values would be loaded into the registers rather than reading in the next set of remainders. The following is the pseudo-code of the circuit.

```
a = ca = 223;   //for the Fibonacci or 0 for the Zero benchmarks
b = cb = 144;   //for the Fibonacci or 0 for the Zero benchmarks
while (true) {
        if (a == 0 || b == 0){
                a = ca;
                b = cb;
                }
        else {
                a = a % b;
                b = b % a;
                }
        }
```

Values ‘ca’ and ‘cb’ are the new values to be loaded and computed, these are dependent on the benchmark. The first benchmark uses 0 as one of the values which will instantly throw the number pair away and fetch the numbers from constants again. This benchmark doesn’t make use of the dividers and its ability to decouple itself from them will be observed. The second benchmark will use the dividers in nearly all operations and a result generation will be a rare occurrence. The numbers which give the largest number of iterations before a number is generated are the consecutive number pairs from the Fibonacci sequence. The numbers used will be 233 and 144 which are the two greatest Fibonacci numbers which still fit in an unsigned 8 bit number space.

Microprocessor datapath

The final example is that of a microprocessor pipeline. The pipeline was extracted from an open source five stage RISC microprocessor design [43]. The pipeline control signals are connected to random value generators to allow the pipeline to execute random instructions without the need to simulate the instruction decode logic. The first stage in the pipeline (Instruction Fetch) is completely skipped. The four simulated stages (Register Fetch, EXEcute, MEMory and Write Back) are also trimmed to simplify the simulation. The register bank contains only four registers and the values from accesses to data memory are always the address supplied.

The delay of the data memory can be varied. The delay starts once all bits of the address are present. The two benchmarks observe the operation with the delay set to zero and 50 gate delays.

6.4.2 Optimisation results

Each of the test circuits was optimised for its benchmark inputs using three levels of optimisation along with the non optimised design (labelled “Early None”). The first optimisation balances the stages through the addition and removal of half latches (labelled “Early Half”). The second optimisation replaces some half latches with early drop latches (labelled “Early Drop”). The third optimisation also where beneficial adds anti-token latches (labelled “Early Anti”).

To compare the designs to the DIMS alternative each circuit is also implemented in the DIMS design style. Results are shown for the original unoptimised DIMS design (labelled “DIMS None”) and a design which has gone through the same optimisation system (labelled “DIMS Half”). Due to the early output specific nature of the early-drop and anti-token latches more advanced optimisations were not possible on the DIMS design.

Finally the critical path can be extracted from the input designs and the logical depth of the slowest stage can be derived. This is then used to present the maximum speed for a synchronous design. The delay is that of the gates only and does not include the delay of

the latching elements nor any overheads to allow a margin of error due to poor manufacture or environmental conditions (voltage temperature).

Decrementer

The decrementer circuit was placed in a test bench and allowed to operate for 100,000 gate delays. The numbers of operations executed were recorded and are shown in figure 6.18.

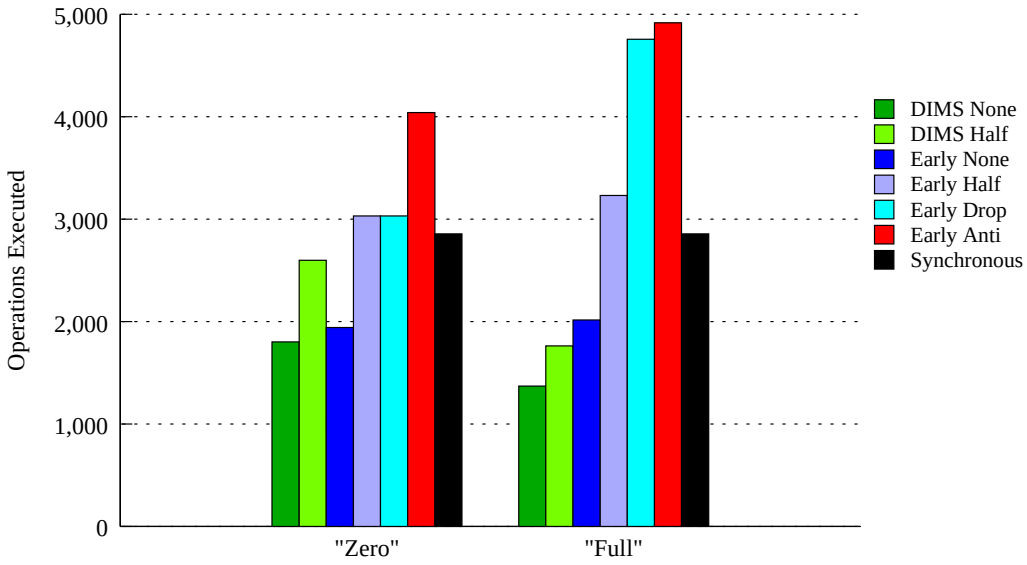

Figure 6.18: Decrementer benchmark performance

The two value clusters show the operation speed with the circuit counting down from two different numbers. The “Zero” benchmark counts down from zero and so fetches a new number each time it executes a cycle. The “Full” benchmark decrements a maximum 32 bit integer. This does not reach zero within the benchmark and so a new number is never fetched.

As was shown in section 6.3.2, the zero benchmark is favourable to the DIMS design. Because the slowest path, in the benchmark of the DIMS design, travels through gates along the fast path (rather than also going through the OR gate, as described in section 6.3.2) the performance benefit of the basic early output design over the DIMS counterpart

is relatively small. The effect of balancing the stages and inserting/removing latches in order to streamline the design benefited both the DIMS and the early output designs and increased their performance by approximately 50% but this still leaves the early output design only 16% faster. The early output design also had the advantage of being able to apply further optimisations. Here the early drop latch insertion yielded no improvement in performance but the insertion of anti-token latches allowed a further performance increase to over 4000 operations. This translates to a cycle time of less than 25 gate delays for a synchronous critical path composed of a 32 bit carry ripple incrementer followed by a multiplexer.

The “Full” test bench allows the decrementer unit to execute with a shorter carry path. The DIMS circuit is penalised due to its use of the slower path through its gates. The performance remains poor even after half-latch insertion/removal optimisations. The early output circuit starts faster than the optimised DIMS version and benefits greatly due to the insertion of early-drop latches. The reason for this is explained in section 6.3.4. Due to the circuit never reaching zero in the testbench, there are no situations where a result is thrown away and the anti-token latch based optimisation does not increase the performance much. Despite the lack of anti-token causing situations, the latch is still beneficial to the circuit in places where the stage is awaiting data from one of the inputs. Once the input arrives the data is consumed and propagates to the next stage faster than the stage validity is calculated. Placing an anti-token latch in such a situation allows the stage to pre-compute the validity before the data arrives. The anti-token latch optimisation (fig. 6.16) detects both cases. This is because ability of the stage fully to complete and generate an acknowledge without the presence of the last remaining input is not required in the optimisation pattern match (also it is very difficult to detect such a situation). The fully optimised circuit executes at a 20 gate delay cycle time compared with 73 gate delay cycle time of the original DIMS design or 35 date delay critical path of the synchronous version.

Greatest common divisor

The GCD benchmark uses a deep pipeline stage which is very finely pipelined with half latches. This makes the tokens going through the stage spaced out and it rarely stalls due

to contention for hardware. Additionally, the early output properties of the circuit are very good. The results of the tests are presented in figure 6.19.

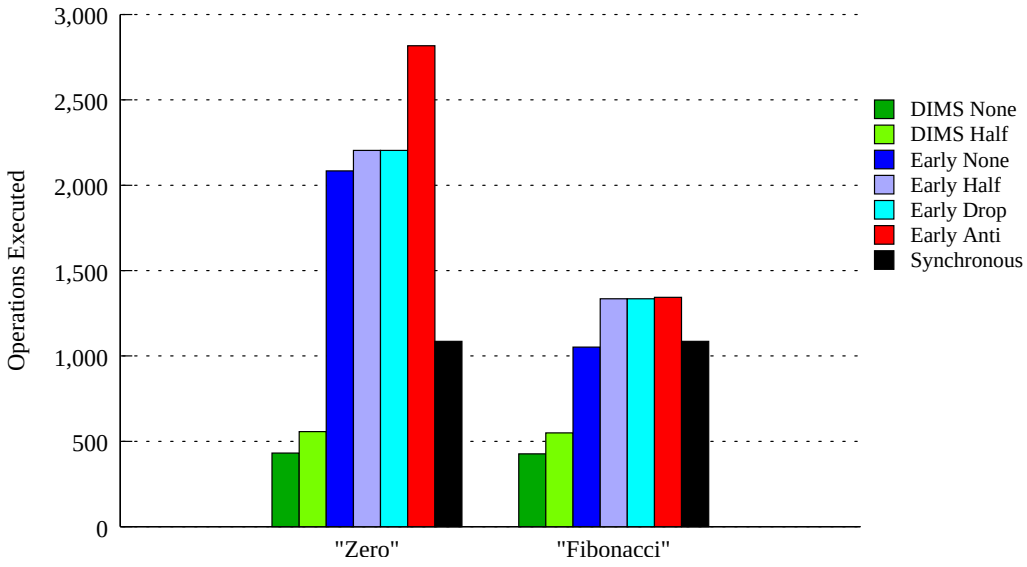

Figure 6.19: GCD testbench results

The DIMS circuits performed poorly in both benchmarks due to their inability to take advantage of any early output cases. The half latch based optimisation in the DIMS designs gave a small performance boost and the benefit only came from the removal of many already present latches which added an unused level of pipelining and instead increased the latency of the data propagation.

The early output circuits performed much better due to their ability to exploit early output cases and not suffering from hardware contention (which can often bring the performance down close to the DIMS level). Because of the fine pipelining of the stage, the half latch optimisation (just like in the DIMS design) removed many latches which were impeding performance. Because of the separation of tokens during execution, the early drop latch had no placement where it would be beneficial to performance. The anti-token latch had an impact on the performance of the “zero” benchmark where the next set of numbers to be computed would always be picked from the constants rather than the results of the dividers. The same effect could not be gained in the Fibbonachi test as, even after a new value had been loaded, there is no benefit in removing the current values from the pipeline

as they are so spread out they will not impede the progress of the next wave of computation passing through the divider.

Microprocessor datapath

The microprocessor datapath benchmark was tested with two delays for the memory operations. The first has the delay set to zero while the second testbench uses a delay of 50 gate delays for a memory access. Because control signalling is generated randomly, the frequency of different situations in the benchmark does not reflect their occurrence in real applications. The probability of a load operation is 50% and the probability of a subtract with both operands being zero (creating a maximum carry propagation chain) is 25%. Because of these factors, the early output circuits executed a little slower than a version executing real code. Despite this, the early output circuits are faster than the DIMS approach and arguably also faster than the synchronous equivalent. Figure 6.20 shows the results of these benchmarks.

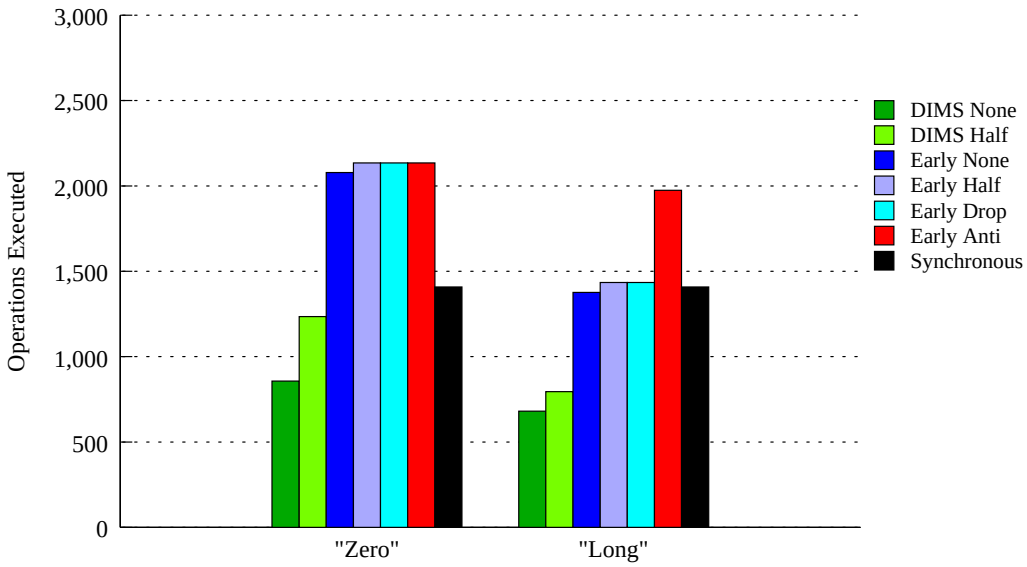

Figure 6.20: Microprocessor data-path testbench results

The DIMS circuits in both cases trimmed the level of vertical pipelining across the adder in the ALU. This has differing effects on the two benchmarks. In the version with no memory delay this approach gives a reasonable performance increase. In the fifty gate

delay version, the slowest path moved from the ALU to the memory stage and decreasing the latency of the ALU did not affect the performance. Instead, this version concentrated on balancing the latches around the memory stage to increase performance. There was only a little room for performance increases.

The early output circuits generally performed better than the DIMS versions. The removal of half latches from the ALU (just like in the DIMS version) had a small impact on the performance in situations where the full carry propagation path is in the slowest path. The early-drop latches did not find any situations where they were beneficial and nor did the anti-token latches in the zero delay memory version. In the long memory delay version the anti-tokens proved very valuable as they managed to bring the performance to close to that of the zero delay version. This was done by placing the anti-token latches across the output of the memory stage to throw away data which was not needed. This allows the system not to take the penalty of the 50 gate delay on half of the operations executed. Additionally anti-token latches placed in the forwarding multiplexers allowed the memory operation data to be discarded at the multiplexers before it arrived, thus allowing them to not to synchronise with the slow memory operation. This has the ability of hiding the performance hit of a single memory access. The de-synchronisation of the inputs of this multiplexer has a capacity of half an anti-token and thus will only be able to hide a single memory access and only if the data from the memory stage is not required by the execute stage in this cycle. Because of this, performance is still impacted (even if only by a small amount). Placing multiple anti-token latches in series will increase the anti-token capacity, but due to the additional latency of the stage and the rare occurrence of multiple memory operations with their data requested by the forwarding multiplexers, this approach only added latency to the system rather than improving its performance.

As a comparison with the synchronous solution, the critical path of the design was 71 gate delays (full ripple carry plus the forwarding and ALU unit select multiplexers). This does not include the delay of the flip-flops and the timing margin overhead. This equates to 1408 operations in the 100,000 gate delay simulation.

6.4.3 Power and Area

So far the main thrust of the early output methodology has been performance based. Although in the early output approach power and area are not targets they must be considered in the design process.

The power consumption of any given circuit can be predicted to a reasonably high accuracy in a static manner. This is because in early output circuits the power consumption is data independent. The area and power consumption of a circuit can be broken down into the four parts. These are data latches, gates, additional pipelining latches and wire forks. In the synchronous design only the first two of these exist (and consume power).

The data latches in early output designs contain ten gates of which seven transition twice for each data element they pass. Gates contain three components (AND and OR gates along with the validity C-element) of which two transition twice for every cycle. Additional pipelining latches (in this case half latches) consist of three elements of which two transitions twice per cycle. Finally wire forks require an acknowledge gathering C-element, this transitions two times per cycle.

The size in transistors of the various components in all three technologies (synchronous, DIMS and early output) is shown in table 6.1. These are very simple estimates to give general figures and do not take account of inversions between components etc. The area increase is dependent on the numbers of each component in the design. In the case of the microprocessor pipeline the early output design was 5.9 times larger and the DIMS design was 7.4 times larger. The greatest common denominator benchmark was 6.2 times larger in both the DIMS and the early output versions.

Table 6.1: Transistor count for each component

Element	Synchronous	DIMS	Early Output
Flip-Flop	14	72	82
2 input gate	4	46	18
3 input gate	6	108	24
4 input gate	8	262	30

Table 6.1: Transistor count for each component

2 way fork	0	10	10
half latch	0	24	34

The power consumption in the three design methodologies are shown in table 6.2. The figures show the number of transitions per cycle of operation in different components. Counting transitions is not a very accurate method of estimating power consumption but is presented here as a rough figure of the relative values. Again this is highly dependent on the numbers of each component in the system. The benchmarked circuits had their numbers of transitions recorded and the transitions per operation was derived. In all benchmarks the DIMS circuits had 13±1 times the number of transitions as the synchronous circuit (assuming the probability of changing data being half and not considering the clock). The early output designs generated 18±1 times the number of transitions.

Table 6.2: Transition count for each component per cycle

Element	Synchronous	DIMS	Early Output
Flip-Flop	$^1/_2$ (exc. clock)	12	14
2 input gate	$^1/_2$	$3^1/_2$	4
3 input gate	$^1/_2$	$4^3/_4$	4
4 input gate	$^1/_2$	6	4
2 way fork	0	2	2
half latch	0	4	6

Both these figures are very high due to the heavy use of half-latches which were not removed if they did not impede performance of the design. If these half latches were all removed, early output circuits would be 4 times larger for all benchmarks and would consume 11 to 12 times the amount of energy compared to a synchronous design. DIMS circuits would be 5 to 6 times larger and consume 9 times the amount of power.

Obviously these figures are very poor but whether they can be justified is an issue which will be tackled in the concluding chapter. The figures for systems without any half latches were presented but these systems would run very slowly. Without any half latches the microprocessor pipeline executed just 400 operations in the 100,000 gate delay simulation

(compared with 2135 for a fully optimised version). The easiest method of tackling the excessive area and power consumption is to remove all unnecessary latches. The slowest path optimisation system cannot determine the usefulness of a latch and only concentrates on the performance of the system. A separate system analysis method would have to be constructed to trim unnecessary latches to move closer to the non half latch pipelined version.

6.5 Summary

The presence of early outputs in standard logic was presented along with methods of improving circuits to capture more of them. Even without specifically targeting the capture of all early outputs the circuits using the method were shown to be superior to the DIMS counterparts. The addition of optimisations such as the early-drop and the anti-token latches, allow the performance of the system to surpass the worst-case based synchronous approach. The placement of these latches as well as balancing pipeline stages and slack matching was done using the analysis of the slowest path. This has been shown to improve the performance of both early output and (to a smaller extent) DIMS circuits. The area and power concerns have been presented along with additional optimisation techniques which were not explored. These form the basis of future work.

Chapter 7: Conclusion

With ever decreasing geometries of the microelectronics manufacturing processes, the new attributes of the emerging technologies are making conventional design methodologies increasingly difficult to apply. Timing closure is becoming increasingly difficult and will soon start to consume the majority of the design time. Global clock distribution and long distance communication are still manageable but techniques to implement them are adding additional complication to already complex designs. Transistor variation is making the above points more difficult and also the worst case delay longer. The actual delay of the executed operation is becoming a small part of the clock cycle, while taking the maximum possible period of time to compensate for clock skew/jitter affecting a worst case delay of the stage with slowest transistors of an operation which is not even being executed.

These reasons have prompted the work presented in this thesis. The presented approach has minimal impact on the architectural and transistor level methodologies. This allows it to be used in conjunction with the current and future optimisations in these regions.

7.1 Contributions to knowledge

The thesis outlined five contributions to knowledge.

7.1.1 Early output

The early output logic approach was presented. A novel method of synthesizing early output circuits was introduced and the ability for early result generation in different circuits was evaluated. The reason for the inability of some constructions to capture all early output states was presented and two approaches to overcome this through the collection of all prime implicants were provided.

7.1.2 Safe guarding

Due to the early output's timing assumptions, two QDI logic design styles and their impact on performance were presented. The safe guarding methods have the advantage of performing close to the speed of non guarded early output circuits but they are very robust. The slower of the two approaches (backward safe guarding) does also offer a limited capacity for non propagating anti-tokens.

7.1.3 Anti-tokens

As well as the anti-token behaviour of backward safe guarding logic, a full anti-token scheme was presented. Behaviour of anti-tokens which comprises: their generation, propagation and destruction (through a merger with a token) was demonstrated. Anti-token latch designs were given for the control, bundled data and the dual-rail design styles.

7.1.4 Blame passing timing analysis

As the complexity of the behaviour of designs built in a bit-level pipelined system with data-dependent delays is too high for engineers to be able to determine the performance bottleneck, a new timing analysis system for use in asynchronous systems was presented. This "blame passing simulation" system extracts the slowest path from a benchmarked circuit, is technology independent and can be used in all asynchronous design styles. The slowest path is the asynchronous circuit's equivalent of the critical path. Critical path optimisation has been instrumental to the generation of high performance synchronous circuits and the slowest path optimisation should prove itself to be equally useful.

7.1.5 Slowest path based optimisation

A series of keyhole optimisations based on the route of the slowest path were presented. These were also demonstrated and their effect on the performance of a series of designs was shown. An automatic method of generating possible optimisations derived from the slowest path, applying them and committing the most effective, was outlined.

7.2 Future work

The work conducted has opened many doors for future exploration of the subject and industrial exploitation. The following are some of the proposed projects to be conducted in the future (some of which are already active).

7.2.1 Complete tool suite

During the course of the thesis a set of tools was constructed to improve the understanding of the behaviour of the circuits. These range from ‘Early’; a system for evaluating and improving the early output coverage in single stage logic, through ‘S2A’; a system of converting synchronous circuit descriptions to a range asynchronous styles and simulating them (optionally extracting the slowest path), to a set of scripts to analyse the slowest path to extract possible optimisation, annotate schematics and evaluate possible optimisations.

A complete tool suite would take the aspects of the current implementations and combine them all into a single easy to use tool. A single input specification can be used to target a number of design styles, in a similar way to Balsa [57] and Tangram [58]. Currently the input specification comes from a custom netlist format file and this would have to be extended to embrace additional popular HDL languages and preferably a custom language specifically designed to allow designers to exploit fully the additional methodology features. The optimisation stage could be automatic or user directed. The schematic annotation with the data from the slowest path extraction could be extended to textual input specifications.

7.2.2 Timing assumption extraction

The extraction of timing assumptions is an area which has not been fully covered in this thesis and its lack restricts the possible optimisations in the non QDI approaches presented. The optimisations in question are the C-element tree flattening and reorganisation (in the acknowledge and validity gathering). Extracting the timing assumptions would allow further performance increases and would generate a more robust system.

7.2.3 Slowest path extraction in other systems

Although the slowest path extraction is bound to the asynchronous domain (running the blame passing simulator on a synchronous circuit reveals the clock as the reason for the slow operation) it can be used on other design styles. Handshake circuits (used in systems such as Balsa and Tangram) are very different from the pipeline based structures used in early output circuits, but the slowest path extraction would be equally effective for improving their performance. The keyhole optimisation table would have to be rewritten for the particular design style.

7.3 Summary

This thesis has presented a design approach takes advantage of asynchronous logic to construct fast circuits. On each of the six testbenches presented the fully optimised early output circuit with anti-tokens managed to outperform the synchronous equivalent at gate level simulations. Future work should further extend the performance advantage. The approach does come at a high cost of power consumption and area but with the reduced effort of timing closure along with a range of benefits of using asynchronous logic could justify its use.

References

[1] J. Sparsø and S. Furber, “Principles of Asynchronous Circuit Design”, Kluwer Academic Publishers, 2001, (ISBN 0-7923-7613-7)

[2] S. B. Furber, J. D. Garside, S. Temple and J. Liu. “AMULET2e: An Asynchronous Embedded Controller”. Proceedings of Async 97, pp. 290-299, IEEE Computer Society Press, 1997.

[3] Hans van Gageldonk, Daniel Baumann, Kees van Berkel, Daniel Gloor, Ad Peeters, and Gerhard Stegmann, “An asynchronous low-power 80C51 microcontroller”, Proceedings of Async 98, pp. 96 -107, IEEE Computer Society Press, Apr. 1998.

[4] N.C. Paver, P Day, C. Farnsworth, D. L. Jackson, W. A. Lien, J. Liu, “A low-power, low-noise, configurable self timed DSP”, Proceedings of Async 98, pp. 32-42, IEEE Computer Society Press, Apr. 1998

[5] Eby G. Friedman, “Clock Distribution Networks in Synchronous Digital Integrated Circuits”, Proceedings of the IEEE, Vol. 89, No. 5, pp. 665-692, May 2001.

[6] Hiroshi Saito, Alex Kondratyev, Jordi Cortadella, Luciano Lavagno, Alexander Yakovlev, “Bridging modularity and optimality: delay-insensitive interfacing in asynchronous circuits synthesis”, Proceedings of IEEE International Conference on Systems, Man and Cybernetics (SMC), volume 3, pp. 899-904, 1999.

[7] S. Moore, R. Anderson, P. Cunningham, R. Mullins, G. Taylor, “Improving Smart Card Security using Self-timed Circuits”, Proceedings of Async 02 , pp. 23-58, 2002.

[8] L.A. Plana, P.A. Riocreux, W.J. Bainbridge, A. Bardsley, J.D. Garside, S. Temple, “SPA - a synthesisable amulet core for smartcard applications”, Proceedings of Async 02, pp. 201-210, 2002.

[9] A. J. Martin , S. M. Burns , T. K. Lee , D. Borkovic , P. J. Hazewindus, “The design of an asynchronous microprocessor”, Proceedings of the decennial Caltech conference on VLSI on Advanced research in VLSI, p.351-373, June 1989.

[10] A. J. Martin , A. Lines , R. Manohar , M. Nystroem , P. Penzes , R. Southworth , U. Cummings, “The Design of an Asynchronous MIPS R3000 Microprocessor”, Proceedings of the 17th Conference on Advanced Research in VLSI (ARVLSI ’97), p.164, September 15-16, 1997.

[11] M. Singh, J. A. Tierno, A. Rylyakov, S. Rylov, and S. Nowick. "An Adaptively-Pipelined Mixed Synchronous-Asynchronous Digital FIR Filter Chip Operating at 1.3 Gigahertz". Proceedings ofAsync 02, 2002.

[12] M. Singh and S. M. Nowick, "High-Throughput Asynchronous Pipelines for Fine-Grain Dynamic Datapaths", Proceedings of Async 2000, 2000.

[13] J.D. Garside, W.J. Bainbridge, A. Bardsley, D.M. Clark, D.A. Edwards, S.B. Furber, J. Liu, D.W. Lloyd, S. Mohammadi, J.S. Pepper, O. Petlin, S. Temple and J.V. Woods, “AMULET3i - an Asynchronous System-on-Chip”, Proceedings of Async 2000, 2000.

[14] S.M. Nowick, K.Y. Yun, P.A. Beerel. “Speculative completion for the design of high-performance asynchronous dynamic adders”, Proceedings of Async 97, pp. 210-223, April. 1997.
[15] S. B. Furber, P. Day, J. D. Garside, N. C. Paver and J. V. Woods, “AMULET1: A Micropipelined ARM”, Proc. IEEE Computer Conference, March 1994
[16] S.B. Furber and P. Day, “Four-Phase Micropipeline Latch Control Circuits”, IEEE Transactions on VLSI Systems, vol. 4 no. 2, 1996.
[17] I.E. Sutherland, “Micropipelines”, The 1988 Turing Award Lecture, Communications of the ACM, Vol. 32, No 6, pp 720-738, January, 1989.
[18] A. M. G. Peeters. “Single-Rail Handshake Circuits”, Ph.D. thesis, Technische Universiteit Eindhoven, Eindhoven, 1996.
[19] S. B. Furber and P. Day, Four-phase micropipeline latch control circuits, IEEE Transactions on VLSI Systems, vol. 4, pp. 247-253, June 1996.
[20] M. J. Stucki, S. M. Ornstein, and W. A. Clark, "Logical design of macromodules", AFIPS Proc., vol. 30, Spring Joint Comput. Conf., pp. 357-364, 1967.
[21] C. H. van Berkel, “Beware the isochronic fork”, Tech. Rep. UR 003/91, Philips Research Laboratories, 1991.
[22] K. van Berkel, F. Huberts, A. Peeters, “Stretching Quasi Delay Insensitivity by Means of Extended Isochronic Forks”, Proceedings of Async 95, May 1995.
[23] D.E. Muller, “Asynchronous logics and application to information processing”, Switching Theory in Space Technology, Stanford, University Press, Stanford, CA, 1963.
[24] C. Seitz, “System Timing”, Chapter 7 of Introduction to VLSI Systems by Mead, C, Conway, L., Addison Wesley. Second Edition, 1980.
[25] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Luciano, A. Yakovlev, “Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers”. IEICE Trans. on Information and Systems, pp. 315-325, 1997.
[26] R. Reese, C. Traver, "Synthesis and Simulation of Phased Logic Systems", International Workshop on Logic Synthesis (IWLS 2000), June 2 2000.
[27] A. Taubin, K. Fant, J. McCardle, “Design of Three Dimension Pipeline Array Multiplier for Image Processing” Proc. IEEE Int’l Conf. Computer Design: VLSI in Computers and Processors (ICCD 02), pp. 104-111, 2002.
[28] D. Fang and R. Manohar. Non-Uniform Access Asynchronous Register Files. Proceedings of Async 04, 2004.
[29] C.F. Brej, “An automatic synchronous to asynchronous circuit convertor”, 11th UK Asynchronous Forum, 2001.
[30] M.A. Thornton, K. Fazel, R.B. Reese, C. Traver, "Generalized Early Evaluation in Self-Timed Circuits", Proceedings IEEE/ACM Conf. Design Automation and Test in Europe, pp. 255-259, Paris, March, 2002.
[31] S. C. Smith, “Speedup of Self-Timed Digital Systems Using Early Completion”, The IEEE Computer Society Annual Symposium on VLSI, pp. 107-113, April 2002.
[32] A. Kondratyev, K. Lwin, “Design of asynchronous circuits by synchronous CAD tools”, ACM/IEEE Design Automation Conference, pp. 411-414, June 2002.
[33] A. Yakovlev, A. Petrov, and L. Lavagno, “A low latency asynchronous arbitration circuit”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems vol. 2, No. 3, pp. 372--377, Sept. 1994.

[34] A. Bystrov, D. J. Kinniment, and A. Yakovlev, “Priority arbiters”, In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pp. 128-137. IEEE Computer Society Press, April 2000.
[35] A. Yakovlev, L. Lavagno, A. L. Sangiovanni-Vincentelli, “A Unified Signal Transition Graph Model for Asynchronous Control Circuit Synthesis”, Formal Methods in System Design 9(3): 139-188, 1996.
[36] A. V. Bystrov, D. Sokolov, A. Yakovlev, “Low-Latency Control Structures with Slack”, ASYNC 2003, pp.164-173, 2003.
[37] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, M. Pietkiewicz-Koutny, “On the Models for Asynchronous Circuit Behaviour with OR Causality”, Formal Methods in System Design 9(3), pp. 189-233, 1996.
[38] C.F. Brej, “A Quasi-Delay-Insensitive Method to Overcome Transistor Variation”, 18th International Conference on VLSI Design, January 2005.
[39] C.F. Brej, “Early Output Logic using Anti-Tokens”, Twelfth International Workshop on Logic and Synthesis (IWLS 2003), May 2003.
[40] R. B. Hitchcock, G. L. Smith, D. D. Cheng, "Timing Analysis of Computer Hardware", IBM Journal of Research and Development, Vol. 26, 1, pp. 100-105, 1982.
[41] R.F. Sproull, I.E. Sutherland and C.E. Molnar, "Counterflow Pipeline Processor Architecture", Sun Microsystems Laboratories Technical Report, April 1994.
[42] C.F. Brej, “Counterflow Networks”, 13th UK Asynchronous Forum, 2001.
[43] C.F. Brej, “Yellow Star: A MIPS R3000 microprocessor on an FPGA”, 2001.
[44] R. K. Brayton, G. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli, “Logic Minimization Algorithms for VLSI Synthesis”, Kluwer Academic, 1984.
[45] “Espresso”, http://www-cad.eecs.berkeley.edu/Software/
[46] “Early resynthesis tool”, http://www.brej.org/early/
[47] E. McCluskey, “Minimization of Boolean Functions”, Bell System Technical Journal, pp. 1417-1444, 1959.
[48] “Quine-McCluskey algorithm - Wikipedia”, http://en.wikipedia.org/wiki/Quine-McCluskey_algorithm
[49] K. Fazel, L. Li, Mitchell A. Thornton, R. B. Reese, C. Traver, “Performance enhancement in phased logic circuits using automatic slack-matching buffer insertion”, ACM Great Lakes Symposium on VLSI, pp. 413-416, 2004.
[50] A. J. Martin, M. Nystrom, K. Papadantonakis, P. I. Penzes, P. Prakash, C. G. Wong, J. Chang, K. S. Ko, B. Lee, E. Ou, J. Pugh, E. Talvala, J. T. Tong, A. Tura, “The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller”, Proceedings of Async 03, 2003.
[51] A. M. Lines. “Pipelined asynchronous circuits”, Master’s thesis, California Institute of Technology, Computer Science Department, 1995.
[52] G. Venkataramani, M. Budiu, T. Chelcea, S. C. Goldstein, “C to Asynchronous Dataflow Circuits: An End-to-End Toolflow”, International Workshop on Logic synthesis (IWLS), pp. 501-508, June 2004.
[53] M. Renaudin, J.B. Rigaud, A. Dinhduc, A. Rezzag, A. Sirianni, J. Fragoso : "TAST CAD Tools", Async 02 Tutoial, 2002.
[54] S. Hassoun, C. Ebeling, "Architectural Retiming: An Overview", TAU95, November 1995.
[55] D.A. Huffman, "A method for the construction of minimum-redundancy codes ", Proceedings of the I.R.E., pp 1098-1102, Sept 1952.

[56] “Huffman coding”, http://en.wikipedia.org/wiki/Huffman_coding
[57] A. Bardsley, “Implementing Balsa Handshake Circuits”, Ph.D. Thesis, University of Manchester, 2000.
[58] K. van Berkel, “Handshake Circuits - An Asynchronous Architecture for VLSI Programming”, Cambridge University Press, 1993.