Asynchronous Pipeline Controller Based on Early Acknowledgement Protocol

Chammika MANNAKKARA

DOCTOR OF PHILOSOPHY

Department of Informatics,
School of Multidisciplinary Sciences,
The Graduate University for Advanced Studies (SOKENDAI)

September 2010

Abstract

Over the past couple of decades, the digital design technology scales to date remarkably satisfying the Moore's Law. The circuits became denser with the scaling of transistor and interconnect, and operating frequencies increased several orders of magnitudes during this period. This poses challenges to digital circuit design in a variety of areas including clock distribution, power management, process migration, fault-tolerance, etc. A lot of research effort goes to tackle these issues under the synchronous design methodology which currently dominates the digital design world. However, the magnitude of the challenges poised has also revitalized the asynchronous design methodology explored in this work, as it inherently address some of key issues.

The main philosophy of the asynchronous design practices is to compose a digital circuit as a collection of autonomous parts communicating with each other locally, as opposed to synchronous design which controls the circuit with a centralized clock signal. Without a global clock or clock domains the designs eliminate the ever increasing problems of high power and area consumption, skew minimization, etc. associated with the clocks. Each component operates only when required in an inherently power efficient manner generating a low electromagnetic(EM) noise. The control and data flow is inherently elastic providing immunity to transistor-to-transistor variability in the manufacturing process, thus providing better technology migration characteristics. These are only a few of the main advantages of asynchronous design.

The spectrum of design styles under asynchronous paradigm varies from bundled data communication model which can employ synchronous-like data processing elements with careful delay matching for completion detection, to delay-insensitive model which can accommodate arbitrary delays in the design. The focus of the is work is on the former style -the bundled data model- which is more close to synchronous design practices. Synchronous circuits, specially pipelined circuits can be transformed to these form asynchronous designs with relative ease. In a time when digital design primarily done in synchronous manner, the work presented here will be significant in harnessing the strengths of asynchronous practices by migrating from synchronous to asynchronous with low effort.

This PhD dissertation presents is a new pipeline controller based on Early Acknowledgement protocol for bundled data asynchronous circuits. The Early Acknowledgement protocol is a hybrid of 2-phase and 4-phase hand-shake protocols, two widely used protocols for bundled data communication. The new Early Acknowledgement protocol combines the advantages of 2-phase and 4-phase protocols and the controller that is presented exploits them. It mainly employs the return-to-zero control signals like 4-phase protocol retaining the simplicity for interfacing and composition of non-linear controllers. At the same time, the controller overhead can be hidden in the Early Acknowledgement protocol which gives the performance comparable to that of 2-phase protocol. First a linear controller for the Early Acknowledgement protocol is proposed which can be deployed in straight pipelines. In order to further the claims of the proposed controller, a non-linear controller for Early Acknowledgement protocol to perform conditional branch operation is also proposed using the above mentioned linear controller. Though simple in construction, it has been observed to be superior in performance compared to its 2-phase and 4-phase counter parts.

The performances of the both linear and non-linear controllers are evaluated analytically. Constraints for the proper operation of the controllers are obtained and the conditions for the optimal operation i.e. when the controller hides all its overhead and operate efficiently are derived. The performances of the controllers are obtained when the controllers are operating in two different modes: pipelines with logic processing and pipelines without logic processing. Similar performance analysis for the controllers of 2-phase and 4-phase protocols (both linear and non-linear controllers) is carried out. The findings outline the design choices available, cost vs. performance benefits and design constraints to be satisfied in employing 2-phase, 4-phase and Early Acknowledgement controller in bundled data communication design.

A case study which carried out to analyse the performance of the each controller in a practical application environment is presented at the end. The target was to build an accelerator module to solve set of linear equations using Gauss-Seidel method which can be used in a core of a Finite Element Method (FEM) analysis system. Three accelerator modules are designed using 2-phase, 4-phase and Early Acknowledgement protocols for the control path. All the designs are implemented on a Xilinx Virtex-4 FPGA platform. Performance of these modules which essentially compares the protocol is analysed and presented. The conclusions highlight the advantages and best use case scenarios of the proposed controllers.

In conclusion, this work as highlighted the importance of little known EA protocol by proposing a controller for it to harness its advantages. This work serves the main source of any analytical and practical comparison of these protocols. The results of the work

will strengthen the importance of EA protocol and encourage the use of it in applications where it exhibit to work efficiently.

Acknowledgements

This work would not be possible without the immense support extended my advisor Prof. Tomohiro Yoneda. He was always available for help and guided me in the right direction. The informal, friendly and productive environment created by him was a key to this success.

I would also like to thank my wife, Indu who stood by me all along, always supporting me throughout this period. I am ever thankful for her to for taking a greater share of responsibilities of household work, specially taking care of our daughter Hansie.

I express my deepest gratitude to my parents and aunt Hilda, for encouraging and supporting me in during this period. Without their advice and encouragement I couldn't have made this far in academia.

My sincere gratitude goes to NII administration who were so understanding and helpful in making our life comfortable in Japan. Last but not least, I complement the friendship of Sri Lankan student community in Tokyo for making my stay in Japan more pleasant and enjoyable.

Dedicated to my beloved wife Indu . . .

Contents

List of Figures

List of Tables

Abbreviations

CB	**C**onditional **B**ranch
CTL	**C**omputation **T**ree **L**ogic
DI	**D**elay **I**nsensitive
EA	**E**arly **A**cknowledgement
EM	**E**lectro **M**agnetic
FEM	**F**inite **E**lement **M**ethod
MD	**M**atched **D**elay
QDI	**Q**uasi **D**elay **I**nsensitive
RD	**R**esetting **D**elay
RTZ	**R**eturn **To** **Z**ero
SI	**S**peed **I**ndependent
STG	**S**tate **T**ransition **G**raph
2P	**2** **P**hase
4P	**4** **P**hase

Chapter 1

Introduction

1.1 Overview

Digital design embraced the synchronous design methodology because it greatly simplifies the design task with a single orchestrator –the clock signal– in command. In the early days of CMOS technology, with low gate densities and low operating frequencies, the asynchronous design methodology gave way to a rapid advancement in digital design based on synchronous circuits. However, as the digital design scales to date remarkably satisfying the Moore's Law, having a single clock has become one of the fundamental limitation. In a synchronous design, clock signal is required to be received by the different parts of the system, ideally, at the same time. The variations of the clock signal due to interconnect length, capacitive coupling, manufacturing and material imperfections, etc. is known as the *clock skew.* When designs became denser and the operating frequencies increased in several orders of magnitudes, clock skew has become a significant portion of the cycle requiring action to mitigate the problem. General practice in synchronous design is to use wider and thicker clock trees which reduce the skew, which in turn makes the clock signal one of the major source of power consumption in a system. Moreover, the clocked components consume power at every clock cycle whether they perform a useful task or not which contribute further to the power consumption problem. In the performance perspective the simplicity of the synchronous designs lies in finding the slowest component (known as the critical path of the system) and synchronizing all the other components to work at that speed. This is otherwise known as "worst–case–design" practice i.e. the design works at the speed of the slowest component.

The distinctive advantages of asynchronous design, which counter the above problems, signify its importance in recent years [1, 2]. Asynchronous designs do away with any centralized (clock) signal. Instead, each component of the design communicates locally with each other. In this design paradigm, a digital circuit is viewed as a collection of autonomous components without any centralized orchestrator to control operations. Thus, there are number of advantages which are inherent to asynchronous design. *(i)* Without a global clock signal, the designs eliminate the ever increasing problems of clock skew and power consumption associated with it [3]. *(ii)* Each component operates only when required an inherently power–efficient manner, generating little electromagnetic (EM) noise [4–8]. *(iii)* Component timing is naturally elastic with each component operating at its own speed. This gives a system average–case performance as opposed to worst–case performance dictated by the slowest component in a synchronous system. *(iv)* Also, it makes the system more resilient to fabrication and environmental variations. *(v)* Moreover, metastability issues arising from the handling of inputs and/or multiple clock domains do not exist in the asynchronous design paradigm [9, 10]. Hence, designs become more robust.

There is a wide array of asynchronous design methods with trade–offs between robustness and performance in operation. The most robust category is Delay- Insensitive (DI) circuits [11]. In this domain of design the functional operation of the circuit is guaranteed regardless of the gate and wire delays of the implementation. DI circuits implemented using gates and their performance is very limited as shown in [12].

Quasi-Delay-Insensitive (QDI) is a variant of the DI circuits which is highly robust yet too restricted to realize useful circuits. Similar to DI, this model does not make assumptions about gate or wire delays except for a class of interconnect called *isochronic forks*. They allow signals to travel to two destinations and only receive an acknowledgement from one. Speed Independent (SI) circuits is another class of asynchronous circuits which comes in the same class as QDI circuits. Operation of a SI circuit is valid irrespective of the gate delays of the design. Yet, in these circuits the wire delays are assumed to be zero. However this assumption has become increasingly difficult to guarantee as the technology scales and wire delays have become predominant.

In Self-Timed (ST) circuit model, the design consists of isochronic regions communicating with zero delay wires. Within each region the elements are self-timed and take arbitrary amount of time to process. In this type of circuits all the delay assumptions are related to local signals within each region.

The general class of asynchronous circuits which assume bounded gate and wire delays, is the least robust and deliver highest performance. Design effort to ensure the post fabricated delays of design meets the requirements is the price to pay for the high performance in this type of design. However, the consideration of robustness and performance depends on the particular application of the circuit.

Asynchronous technology has been applied for processors in numerous occasions. One of the notable asynchronous processors is the AMULET series which implements the ARM processor architecture. The AMULET1 was first demonstrated in 1993 and the processor is currently in its third iteration (AMULET3i) [5, 7, 13].

1.2 Contribution of this work

The main contribution of this work is a proposal of a new pipeline controller for Early Acknowledgement protocol for bundled data asynchronous circuits. Early Acknowledgement protocol is a relatively new handshake protocol for bundled data communication. 2-phase and 4-phase protocols are the most widely used protocols for this type of asynchronous design. The new protocol combines the advantages of each protocol and hence claimed to be superior. The contributions of this work can be summarized as follows.

1. The first of the pipeline controller proposed based on this protocol is a linear controller to operate straight pipelines. The analytical and experimental study on this controller shows it outperforms 4-phase controller and the performance of it is comparable to that of high-speed 2-phase controller

2. Another controller is proposed to perform the conditional branch non-linear operation based on the above proposed linear controller. The performance of it is analytically and experimentally proven to be superior to both 4-phase and 2-phase conditional branch controllers.

3. Thereby this work also highlights the importance of the Early Acknowledgement protocol as a promising candidate for bundled data handshake protocol even though it is not widely employed in asynchronous design community.

1.3 Organization

This thesis is organized in to nine chapters. The Chapter 1 gives an introduction to the dissertation. The rest of the thesis is organized as follows:

- Chapter 2 gives basic theoretical background for asynchronous design and the pipelined circuits which is the focus of this work.
- The first controller is presented in Chapter 3 which includes its design, operation, constraint and performance analysis.
- The conditional branch controller is presented in Chapter 4 with similar structure to the linear controller in the previous chapter.
- An analysis of the performance of 2-phase linear and conditional branch controllers (MOUSETRAP controllers) are presented in Chapter 5.
- Similarly, an analysis of the performance of linear and conditional branch controllers for 4-phase controllers is presented in Chapter 6.
- Comparison of the performance and an experimental evaluation of their performance is presented in Chapter 7.
- The case study carried out to test the performance of the proposed controllers in a practical design is presented in Chapter 8.
- The conclusions and future research work in this area is given in Chapter 9.

Chapter 2

Background and Theory

2.1 Overview

This chapter provides the background details for the research work discussed in the subsequent chapters. The basics of pipelined circuits and their construction in synchronous and asynchronous manner are given. Bundled-data communication - the asynchronous design model that this work is based- is detailed with a brief analogy to its synchronous counterpart. Then, the hand-shake protocols (4-phase and 2-phase protocols) that are mainly used for bundled data communication are presented. The Early Acknowledgement protocol, on which this work make a contribution is detailed next. At the end, a fundamental component used in asynchronous circuits and a formalism used to analysis the circuit is presented.

2.2 Pipelined Circuits

Very simple systems which perform basic functions that depend on the inputs only, can be implemented using only combinational logic. However, more often complex systems require partitioning of the function and keeping the state of the logic held temporally. A *pipeline* is a result of such a partitioning of a complex function in temporal order. It consists of a set of data processing elements (partitions of a task) connected in series, so that the output of one element is the input of the next one. The elements of a pipeline are executed in parallel. *State registers* are inserted in-between two processing elements to buffer the data. The state registers (or pipeline registers) store the results of a particular partition and retain it as input the next stage. Such a partitioning has following advantages.

1. Partitioning allows a single processing element (combinational block) to be shared by several different operations. (e.g. ALU with different operands)
2. Use the results of the processing element as an input the next operation of the element (e.g. accumulator unit)
3. Deliver higher throughput by executing the each stage of the pipeline independently, as opposed to waiting for the whole task to finish before stating another cycle.

The timing to operate such a pipeline is addressed differently in synchronous and asynchronous design methods which are discussed in next two sections.

2.3 Synchronous Pipelines

As shown in Figure 2.1, the synchronous circuits derive timing to operate the pipeline registers from the global clock signal. The conventional microprocessors are synchronous circuits that use this type of synchronous pipelines. Here, registers are clocked synchronously. The time between each clock signal should be greater than the longest delay between pipeline stages. Thus ensuring when the registers are clocked, the data that is written to them is the final result of the previous stage.

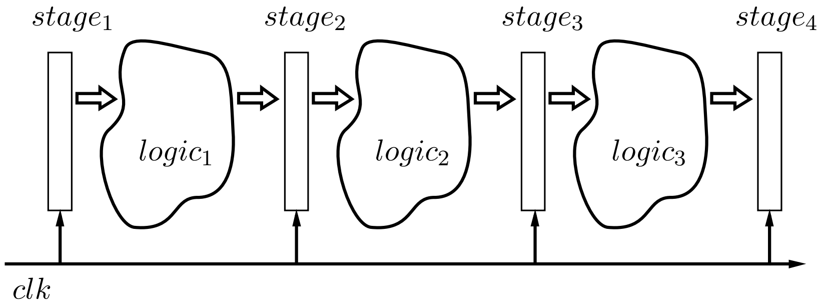


FIGURE 2.1: Synchronous Pipeline.

Figure 2.2 shows the movement of data from one stage to another by shifting data to the next stage at the rising edge of the clock. The boxes *D0* though *D3* denote four different sets of data passing through the pipeline stages. At each pipeline stage require different amount of time depending on the data to process them and make it available to the next stage. The gray area denotes time each stage waits before the clock signal arrives. Note that when *D0* passes through *stage 2* it requires the entire clock cycle to process data. This in turn defines the critical path and hence the maximum clock frequency to the entire pipeline. If the clock frequency is increased, *D0* cannot be completely processed from *stage 2 before* the clock signal arrives to capture the data for stage registers, resulting a failure in the pipeline. *Stage 3*, uses relatively less time to complete any data-set, yet the clock cycle is required to be set to address the above mentioned worst-case making *Stage 3* idling for the considerable time in each cycle.

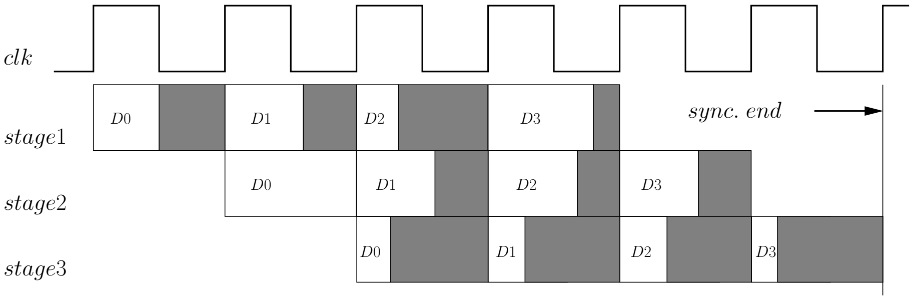


FIGURE 2.2: Synchronous Pipeline Data Flow Diagram.

2.4 Asynchronous Pipelines

In asynchronous pipelines the stage registers are clocked asynchronously by detecting the completion of each stage. To achieve this, global clock of the synchronous designs are replaced with locally distributed set of controllers to provide timing for each stage. These controllers use a request/acknowledge signals, to indicate handshake between adjacent stages to pass data.

Figure 2.3 shows an asynchronous pipeline and Figure 2.4 shows the movement of data within this pipeline. Note that the timing for each stage is generated locally for each stage. Hence, the pipeline exhibit average case performance. Pipeline “stretches” depending on the stage and/or data being processed. This causes two types of *pipeline stalls* in this type of pipelines.

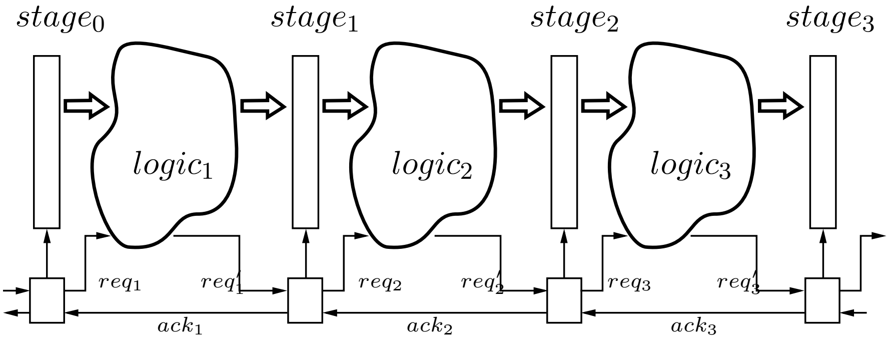


Figure 2.3: Asynchronous Pipeline.

- *blocking*: when the data is ready to be processed but the next stage is busy processing the previous set of data
- *starvation*: when the stage is ready to process data but the data is not available from the previous stage.

In Figure 2.4 the blocking and starvation are illustrated with light gray and dark gray respectively. Blocking can be observed between stages 1 and 2 for *D1* data. Data is available from the *Stage 1*, yet *Stage 2* is busy processing the previous data *D0*, hence cannot accept *D1*. Starvation is present in *Stage 3* after processing data *D0*. It has finished processing *D0* and waiting on *D1* data which is still not available. 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.

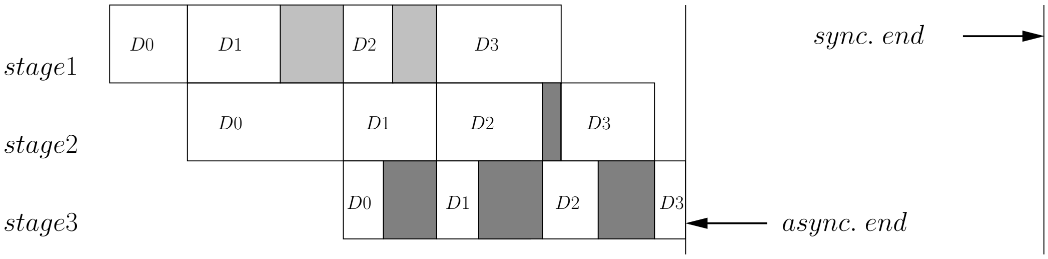


Figure 2.4: Asynchronous Pipeline Data Flow Diagram.

There are two basic schemes to encode the data path of an asynchronous pipeline. Namely they are: Single-rail encoding (or Bundled-data communication) and Dual-rail encoding. In bundled-data communication, a single wire is used to represent a single bit of data, in the same way as synchronous logic. In dual-rail encoding two wires are used to represent a single bit of a data path –each rail to transmit the value 0 and value 1.

This work entirely based on the former scheme. A brief introduction to this single-rail scheme is given below.

2.4.1 Bundled Data Communication

Bundled data communication model uses "matched delays" (MD) to locally generate the completion signals for each stage as shown in Figure 2.5. The data wires do not carry the completion signal by themselves, rather all (or a set of) wires are bundled together by a dedicated wire with a matched delay carrying the completion signal to the local controllers. This allows "conventional" data processing elements used in synchronous designs to be used in asynchronous designs employing bundled data model. The bundling can be done for the entire data set or sets of data to allow more fine control in matched delay at the expense of additional control wires and stage controllers.

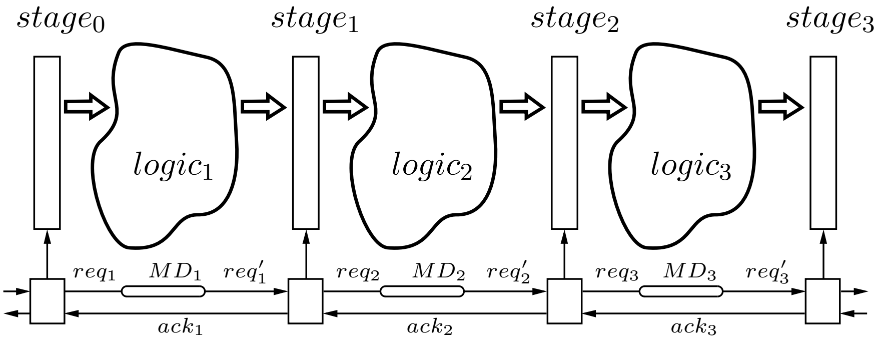


Figure 2.5: Bundled Data Pipeline.

2.5 Handshake Protocols

The protocol used to communicate between the controllers varies. Three of such protocols which span the scope of this study are discussed below. The first two protocols 4-phase and 2-phase protocols are the most widely used protocols. Early Acknowledgement protocol is a relatively new protocol on which the main work of this thesis based.

2.5.1 4-phase Protocol

4-phase protocol is a level-based or Return-To-Zero (RTZ) handshake protocol. The request (*req*) and acknowledge (*ack*) signals starts from zero state and return back to zero state at the end of a handshake cycle. The 4-phase protocol is shown in Figure 2.6. As the name suggests, in this protocol, there are four transitions on the *req* and *ack* signals per cycle. Here, a request is made by the rising edge of the *req* signal. The corresponding acknowledgement is indicated by the receiver raising the *ack* signal. The working period for this protocol is from the rising edge of *req* to the rising edge of *ack*. At the end of the working period, both *req* and *ack* signals are high. Upon receipt of the acknowledgement, *req* is lowered and *ack* is also lowered, completing the cycle. The period when both *req* and *ack* are zero before the next cycle starts is the resetting period. The different sequencing of these 4-phase signalling transitions leads to different controllers for a range of cost and performance options as shown in [14].

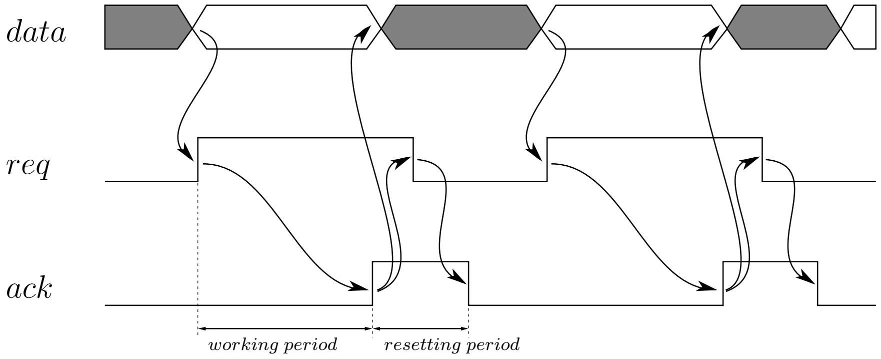


Figure 2.6: 4-phase Protocol.

The 4-phase handshake protocol is also the most commonly used protocol for its simplicity. The AMULET processor and Tangram-based systems use the 4-phase protocol. However, the main disadvantage of the this protocol is the resetting phase where the protocol is incapable of performing useful task while waiting for signals to reset for the next cycle.

2.5.2 2-phase Protocol

In the 2-phase, or transition signalling, protocol, each cycle consists of two transitions (rising or falling edge) on *req* and *ack* signals, as shown in Figure 2.7. A request is made by a transition on the *req* signal. The receiver also acknowledges completion of work by a transition on the *ack* signal. The working period is defined as the duration from the request being made to the acknowledgement of completion. Hence, for the 2-phase protocol it is between any transition on the *req* signal to the same transition on the *ack* signal. It should be noted that in this protocol, the next cycle can be started immediately after the working period. The MOUSETRAP [15], a simple and robust linear pipeline controller, is based on this protocol, which has been proven to operate high–throughput pipelines at 2.1–2.4 GHz [16].

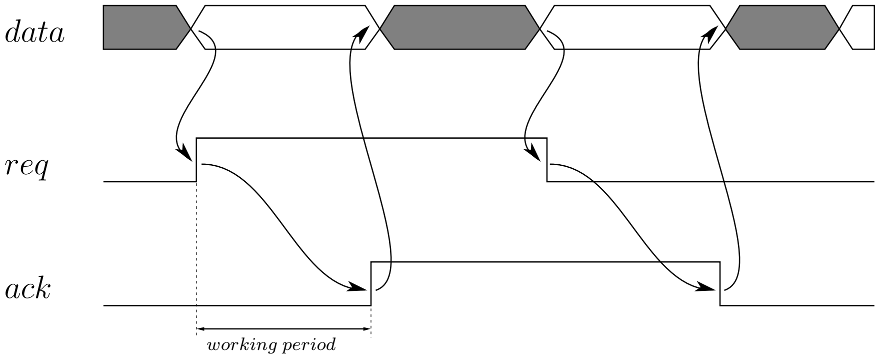


FIGURE 2.7: 2-phase Protocol.

However, when the transition signalling protocol is used, translations from 2-phase to 4-phase are usually required at some points, because in many cases, environment circuits use level-sensitive controls.

2.5.3 Early Acknowledgement Protocol

Pipeline controller presented as the main work of this thesis employs the Early Acknowledgement (EA) protocol introduced in [17], where its original idea was presented in [18]. This protocol is an improvement over the simple 4-phase protocol, and can hide the resetting period of the signalling. As shown in Figure 2.8, the request is made at the rising edge of *req* signal similar to 4-phase protocol. But, the *ack* signal goes high at any time point after the request, and the *req* signal can be lowered in response to this *early acknowledgement*. Especially in the EA protocol the completion of work is indicated by the *falling edge* of the *ack* signal in contrast to the 4-phase protocol where it is indicated by the rising edge. Since working period is from the request made to notification of completion, in the case of EA protocol it is from rising edge of *req* signal to falling edge of *ack* signal. Unlike in 4-phase protocol, at the end of working period, both signals are reset back to zero and next cycle can be started immediately. Hence, this protocol eliminates the resetting period inherent in the 4-phase protocol and yet retains its simplicity by maintaining the return-to-zero control signals.

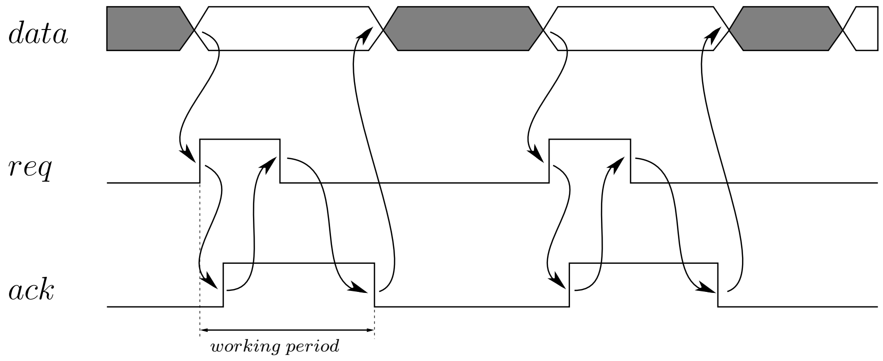


FIGURE 2.8: Early Acknowledgement Protocol.

When any of the above protocols is used in an asynchronous pipeline, the requests need to be appropriately delayed with matched delays such that request on the next stage arrives after the data is valid. The Figure 2.9 illustrates the 4-phase protocol used in such a case.

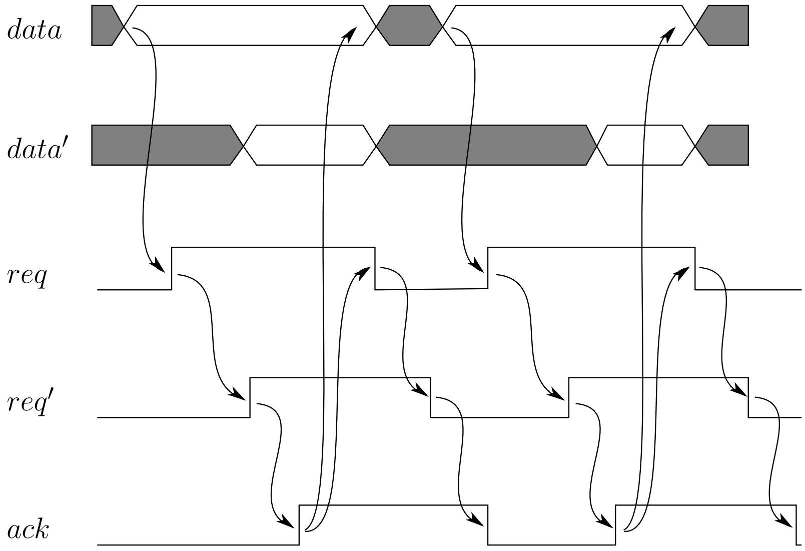


FIGURE 2.9: 4-phase Protocol.

2.6 C-element

In asynchronous circuit design there are some fundamental elements used that are not common in synchronous design domain. Muller C-element [19] is a commonly used asynchronous component. It performs “event anding” operation, i.e. C-element produce and event when it receives events on all of its inputs. The event in this case can be 0-to-1 or 1-to-0 transition. Figure 2.10 shows the C-element symbol and the implementations of it using feedback logic and at transistor level.

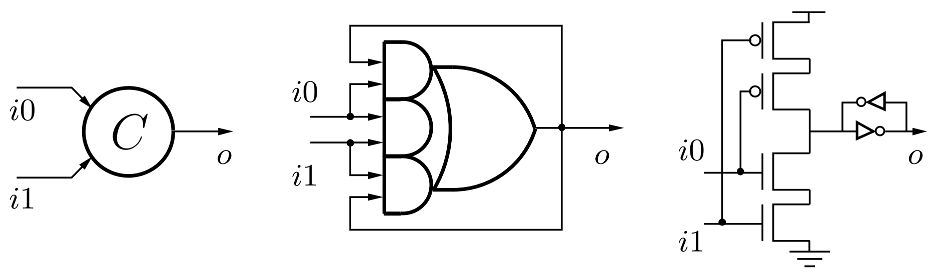


FIGURE 2.10: Muller C-Element (symmetric).

Asymmetric C-element is a natural extension of this function. There, the output event is sensitive to only one of input events, either 0-to-1 or 1-to-0. Figure 2.11 shows the asymmetric C-element and its implementations. Output is sensitive to the 0-to-1 transitions of the asymmetric inputs that are attached to the plus (+) side of the symbol and to the 1-to-0 transitions of the inputs attached to the minus (−) side. The inputs in the middle are accounted for in both transitions like in symmetric C-element.

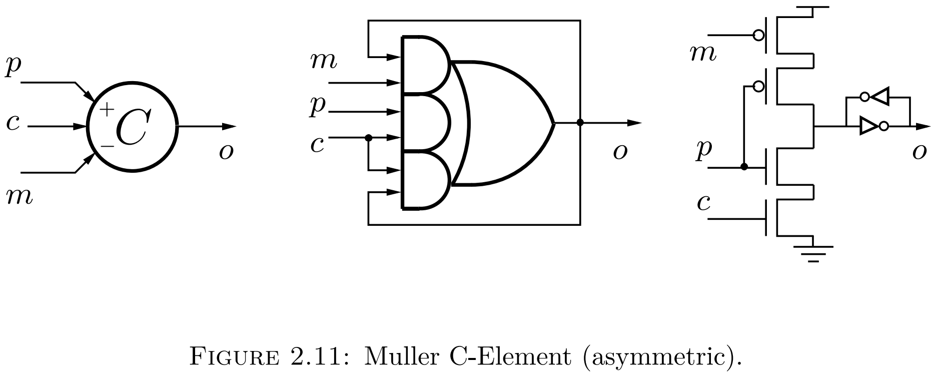


FIGURE 2.11: Muller C-Element (asymmetric).

2.7 Formal Specification: STGs

In order to formally analyse the proposed designs *State Transition Graphs* (STG) are used [20, 21]. STG is a derivative of a Petri Net [22] a graphical and mathematical tool for describing and studying various concurrent systems. In the strict sense, STG is an Interpreted Free-Choice Petri Net (IFCN) [23]. There are many advantages using the STGs for formal specification of an asynchronous circuit.

- STGs address the basic paradigms of asynchronous systems: the concurrency causality and choice.
- The behaviour of the circuit and the environment can be captured to one STG diagram
- Relatively easy to use due to its resemblance to timing diagrams

2.8 4-phase Linear Controller

We have used the 4-phase controller proposed in [24] for this comparison. The controller is shown in Figure 2.12. The *G1* and *G2* are complex gates which comprises the controller.

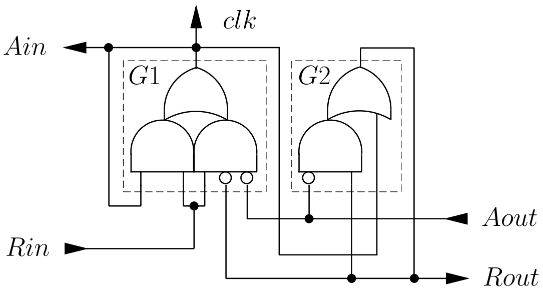


FIGURE 2.12: 4-phase Controller.

2.9 2-phase Controller: MOUSETRAP

For the 2-phase or the transition signalling protocol, MOUSETRAP controller is selected for its simplicity and high performance. As shown in Figure 2.13, the controller consists of a simple transparent latch (denoted by the rectangle box) and a XONR gate. The signal *enable* is used to drive data latches (instead of D-flipflops) in the data path.

Initially all control signals are low except for *enable* signals making the all stages of pipeline transparent. When the first data item flows through the pipeline stage, it flips the values of Rin_N, $Rout_N$ and Ain_N exactly once (high). Subsequently, the second data item flips all these signals once again (low). This is 2-phase (or transition) signalling where each transition (either up or down) indicates a distinct request or acknowledgement.

Once a data item captured by stage latches, three actions take place in parallel: (i) the request to next stage $Rout_N$ is made; (ii) an acknowledgement, Ain_N, is sent to the previous stage, allowing the next data item to be sent; and finally (iii) $enable_N$ is lowered to make the stage latched opaque protecting the current data from being overwritten. Subsequently, when an acknowledgement, $Aout_N$ is received from $stage_{N+1}$, the latch in $stage_N$ is re-enabled (i.e., made transparent).

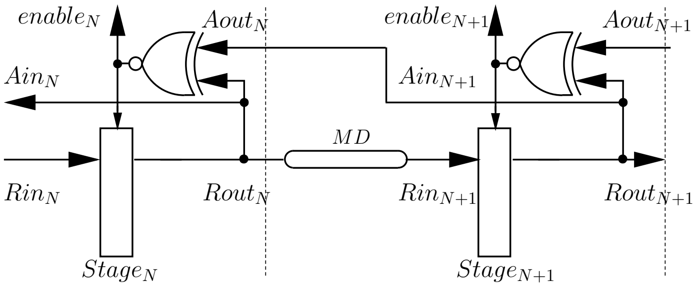


Figure 2.13: MOUSETRAP Controller.

In [15] the operation of the MOUSETRAP controller in both high-speed pipeline and pipelines with logic processing is described in detail.

Chapter 3

Design and Analysis of EA Controller

3.1 Overview

The pipeline controller presented in this chapter is an improvement of the controller that was proposed earlier in [25]. The new controller has reduced the overhead of the previously proposed controller under two timing constraints introduced. The operation of the controller is presented first, followed by a timing analysis for its correct operation. The performance analysis is presented at the end.

3.2 Pipelined Operation of Early Acknowledgement Protocol

First, we define a few naming conventions that we use for all controllers throughout the rest of this dissertation. A general diagram of a pipeline using a bundled–data scheme with logic processing in between stages is shown in Figure 3.1. In the interface of the controller, Rin_N is the request to $stage_N$, and Ain_N is the corresponding acknowledgement signal from it to the input side ($stage_{N-1}$). Similarly, $Rout_N$ and $Aout_N$ are the request and acknowledgement to and from the output side ($stage_{N+1}$). The local clock signal of the stage generated by the controller is clk_N. The logic processing delay ($logic$) between the pipeline stages is accounted for by the matched delay (MD) inserted in the request line between stages. For the 2-phase protocol, the delay can be symmetric such as a string of buffers, where as for the 4-phase protocol (hence, for the EA protocol as well), the delays are asymmetric with a quicker resetting time, as shown in Figure 3.2. $t_{MD\uparrow}$ represents the variable delay for the rising transition and $t_{MD\downarrow}$ represents the delay for the falling transition. In our implementation, $t_{MD\downarrow}$ is equal to $t_{AND\downarrow}$.

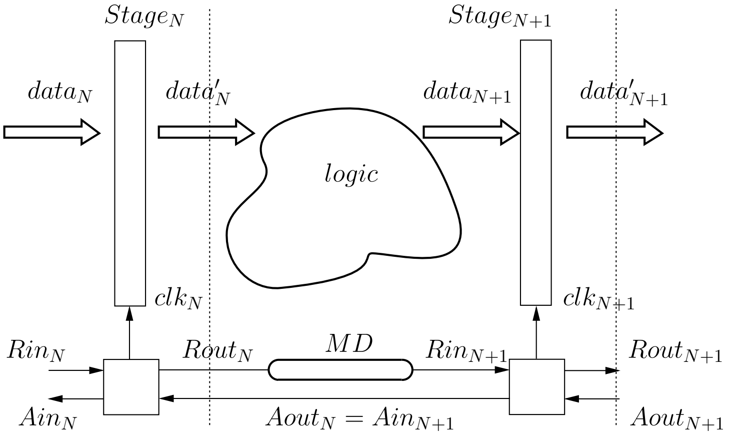


FIGURE 3.1: General Pipeline with Logic Processing.

Figure 3.3 shows the operational waveforms of the EA controller in a general pipeline with logic processing as in Figure 3.1. The EA protocol uses the falling edge of the acknowledgement signal to indicate the completion of the working period. Hence, data D on $data_N$ will be captured in $stage_N$ at the falling edge of Ain_N (i.e., $clk_N = \overline{Ain_N}$). The captured data is processed by the *logic* unit in between two stages and becomes available at $data_{N+1}$ after t_{logic} delay. Concurrently, on the control path, $Rout_N$ is raised and $stage_{N+1}$ controller receives the request on Rin_{N+1} after the $t_{MD\uparrow}$ delay. The matched delay MD is chosen such that the falling edge of Ain_{N+1} occurs just after $data_{N+1}$ becomes valid. This essentially hides the controller overhead (i.e., from the rising edge of Rin_{N+1}, up to the falling edge of Ain_{N+1}) inside the required t_{logic} delay by offsetting it from the matched delay MD. Thus,

$$t_{logic} = t_{MD} + t_{ctrl} \tag{3.1}$$

where, t_{ctrl} is the controller overhead. When $t_{logic} \geq t_{ctrl}$, controller overhead (t_{ctrl}) can be completely hidden inside the required matched delay. However, for fine–grained

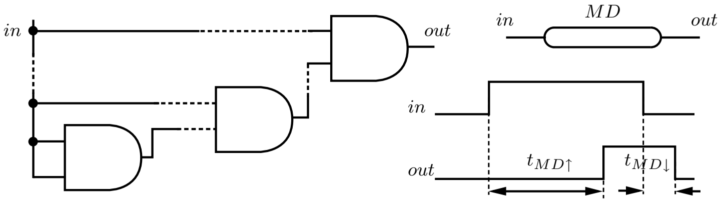


FIGURE 3.2: Asymmetric Delay for MD.

pipelines with 1–2 gates per stage, this condition may not hold and, in that case, the controller overhead is exposed to the pipeline operation. Hence, our controller is preferable in applications where there is a fairly large processing delay (t_{logic}) between stages.

3.3 Controller Operation

Our controller the EA protocol is depicted in Figure 3.4. The controller consists of two AND gates, a C-element, an inverter, and an asymmetric delay (RD) for the self–resetting of the rst signal. The clock signal clk of the pipeline stage is derived from $\bar{Ain}$,

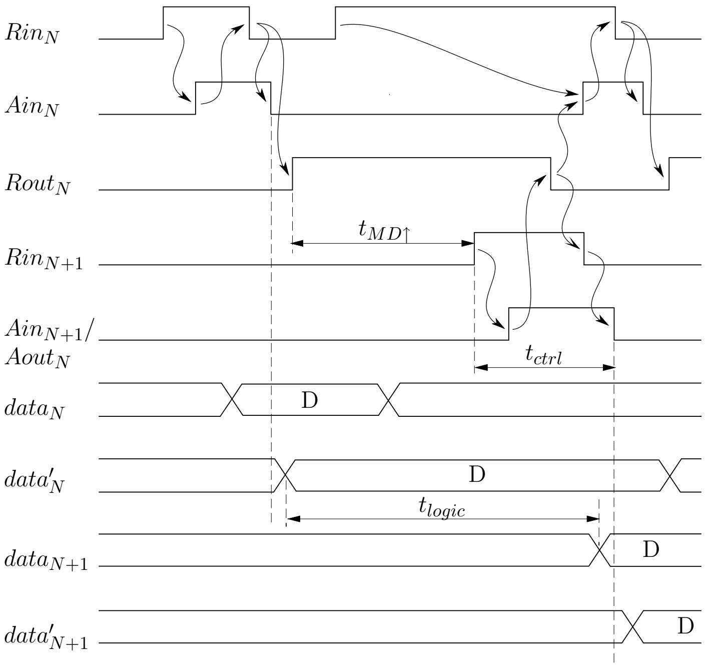


FIGURE 3.3: Behavior of EA controller.

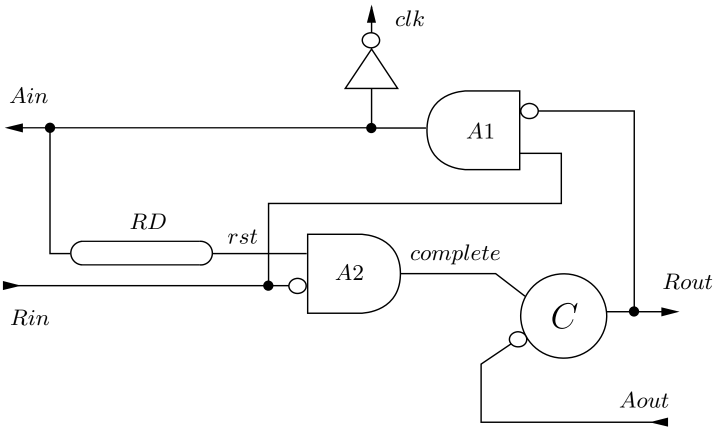


Figure 3.4: EA pipeline controller.

and allows the clocking of the stage to be made at the falling edge of the acknowledgement. The implementation of this delay is shown in Figure 3.5. $t_{RD\downarrow}$ is the variable part of the delay, and $t_{RD\uparrow}$ is exactly equal to $t_{OR\uparrow}$.

Figure 3.6 shows the operation of the controller, which conforms to the pipelined operation in Figure 3.3. Initially, all the control signals are low except for the *clk* signal. When the input stage raises the request *Rin*, the controller immediately acknowledges the request by raising *Ain*. At first, this is possible because there are no pending requests at the output stage through *Rout* (For the blocked case, see below).

Since the early acknowledgement is provided by raising *Ain*, *rst*– the input for AND gate *A2* from the asymmetric delay– is also raised. When the input stage lowers the request in response, the acknowledgement and the data is expected to be ready and the following events occur.

- *Ain* is lowered by the falling edge of *Rin* through *A1*,

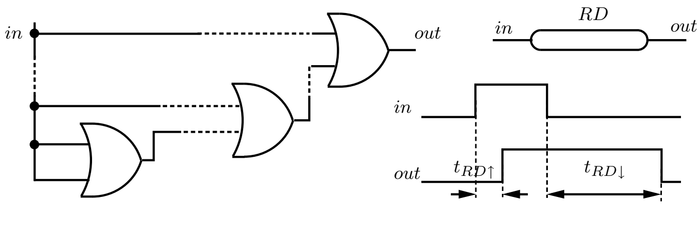


Figure 3.5: Asymmetric delay for *RD*.

- *clk* is raised, latching the new valid data from the input stage to the current stage register, and
- *complete* is raised, generating the rising edge of the output request *Rout*

Once the *Rout* has been driven high, it can be maintained high by C-element[1] even after the *complete* signal has been lowered by the self–resetting circuit of the controller. This also constitutes a local timing constraint to be satisfied by $t_{RD\downarrow}$ of the self–resetting delay to correctly produce the *Rout* signal.

Since the controller has fully completed the handshake cycle at the input side, it is free to make a new request on *Rin*. However, the pending output request *Rout* high effectively blocks the generation of an early acknowledgement back to the input side. Upon receipt of acknowledgement high on *Aout*, *Rout* will be lowered, and the blocked request from the input stage will be free to send the early acknowledgement by raising *Ain*.

3.4 Timing Constraints

First, we turn to the timing constraints required for the desired operation described in Section 3.2. For constraint analysis, we assume that our controller is in a middle stage of

[1] The C-element used here with a negative input changes its output only when the two inputs have different values, and its output value is equal to that of the positive input.

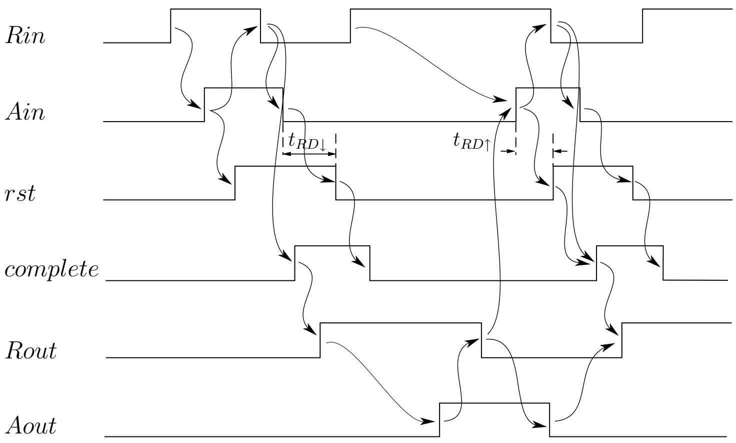


FIGURE 3.6: Controller operation.

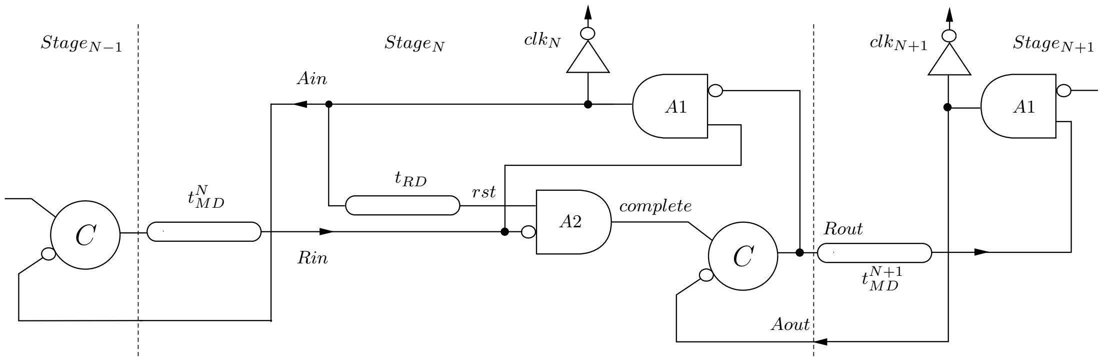


FIGURE 3.7: Fastest environment for constraint analysis.

a pipeline and that its environment (i.e., the controllers in the previous and next stages) operates at a speed equal to or slower than that of our controller. This is because we consider that the linear controller is the fastest, and assuming that the environment is slower than it allows us to evaluate the impact on constraints when more complex operations are built around the linear controller, as detailed in Section 4.3. Figure 3.7 shows the fastest environment where the delays can be quantified using the controller delays.

The Signal Transition Graph (STG) for our controller for constraint analysis is depicted in Figure 3.8. Thick arrows indicate the signal transitions generated from the environment of the controller, where as regular arrows indicate transitions made by the controller. Transitions are labelled with their associated gate delays. Note that, according to our assumptions, the environment delays are either equal to or larger than the delays incurred from the two similar controllers in the previous and next stages as shown in Figure 3.7.

We identify two types of expressions throughout the constraint analysis: the constraints and properties. The equation numbers are appropriately prefixed with the letter C or P to distinguish between these types. Constraints are what are *required* to be satisfied, where as properties express conditions that *already* hold. We utilize the properties of the controller and the environment in validating the constraints during our analysis.

In the timing calculations, the inverted inputs of AND gates A1, A2 and of the C-element are not considered separately. They are attributed to the total delay of the gate.

Constraint 1. The first constraint imposes conditions to prevent data overwriting. In our controller, the pending output request (*Rout* high) blocks any new requests on *Rin*.

This requires *Rout* to go high *before* a new request (*Rin* high) is received. Thus, the timing constraint can be formulated as follows:

$$\text{(C)} \qquad t_{Rin\downarrow\rightarrow Rin\uparrow} \geq t_{Rin\downarrow\rightarrow Rout\uparrow} \tag{3.2}$$

The left-hand side of the above constraint can be given as:

$$\text{(P)} \qquad t_{Rin\downarrow\rightarrow Rin\uparrow} = t_{AND\downarrow} + t_{Ain\downarrow\rightarrow Rin\uparrow} \tag{3.3}$$

The path is labelled Ⓐ in Figure 3.8. Note that *Ain* ↓ is always caused by *Rin* ↓ through AND gate *A1*. Since the delays incurred from the environment at the input and output sides are considered to be either equal to or larger than the delays incurred by a linear controller, as mentioned previously, the following holds (see Figure 3.7).

$$\text{(P)} \qquad t_{Ain\downarrow\rightarrow Rin\uparrow} \geq t_{C\uparrow} + t^{N}_{MD\uparrow} \tag{3.4}$$

Thus, (3.3) can be rewritten as:

$$\text{(P)} \qquad t_{Rin\downarrow\rightarrow Rin\uparrow} \geq t_{AND\downarrow} + t_{C\uparrow} + t^{N}_{MD\uparrow} \tag{3.5}$$

As for the right-hand side of (3.2), we need to consider two cases where different events cause *Rout* ↑.

Case 1: If *Aout* ↓ is early enough compared with the next *Rin* ↑, and if *Rout* ↑ is caused by *complete* ↑, then the following holds:

$$\begin{aligned} \text{(P)} \qquad t_{Rin\downarrow\rightarrow Rout\uparrow} =& max(0, -t_{Ain\uparrow\rightarrow Rin\downarrow} + t_{OR\uparrow}) \\ & + t_{AND\uparrow} + t_{C\uparrow} \end{aligned} \tag{3.6}$$

The above expression is labelled Ⓑ in Figure 3.8. The *max* operator is used to get the larger of the delays from two concurrent paths. The first path corresponds to the delays from the input side, and the second path comprises delays local to the controller in the self-resetting loop. Since the second path actually originates from *Ain* ↑, $t_{Rin\downarrow\rightarrow Ain\uparrow} = -t_{Ain\uparrow\rightarrow Rin\downarrow}$ is used. Again, from the delay assumption of the environment,

$$\text{(P)} \qquad t_{Ain\uparrow\rightarrow Rin\downarrow} \geq t_{C\downarrow} + t^{N}_{MD\downarrow} \tag{3.7}$$

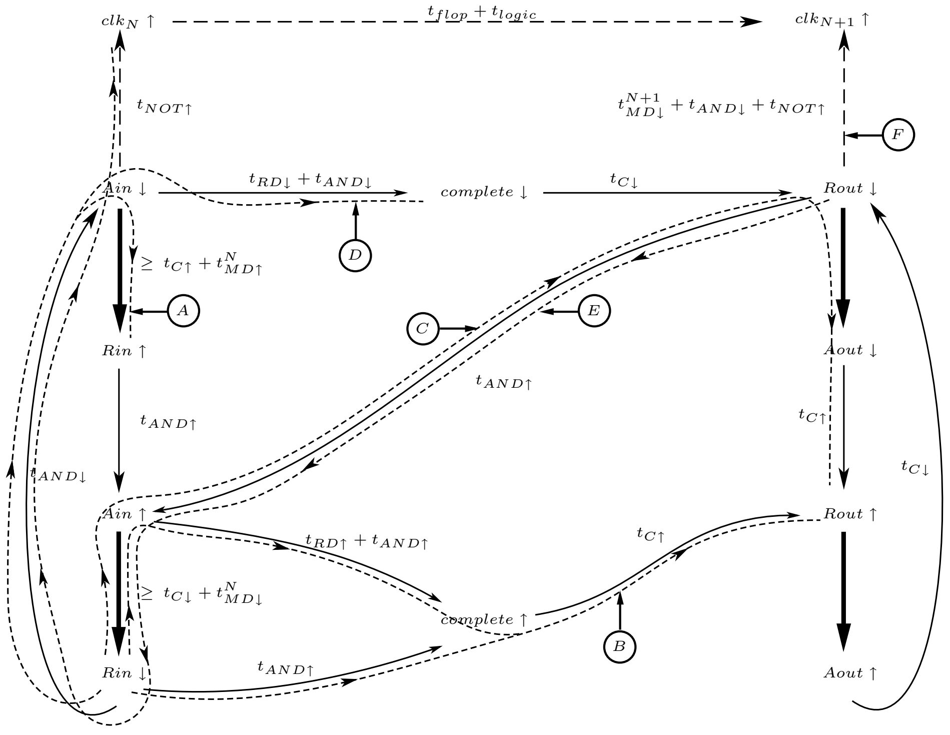


FIGURE 3.8: STG for EA controller (constraint paths).

holds (see Figure 3.8). Thus, (3.6) can be rewritten as:

$$\text{(P)} \qquad t_{Rin\downarrow \to Rout\uparrow} \leq max(0, -(t_{C\downarrow} + t^N_{MD\downarrow}) + t_{RD\uparrow}) + t_{AND\uparrow} + t_{C\uparrow}. \tag{3.8}$$

From (3.5) and (3.8), a conservative version of the constraint (3.2) is obtained in the form of constraints for the variable parameter $t^N_{MD\uparrow}$, the matched delay to be inserted between two stages of the pipeline, as follows:

$$\begin{aligned} \text{(C)} \qquad & t_{AND\downarrow} + t_{C\uparrow} + t^N_{MD\uparrow} \geq t_{AND\uparrow} + t_{C\uparrow} \\ & \text{that is,} \qquad t^N_{MD\uparrow} \geq t_{AND\uparrow} - t_{AND\downarrow} \end{aligned} \tag{3.9}$$

and

$$\begin{aligned} \text{(C)} \qquad t_{AND\downarrow} + t_{C\uparrow} + t^N_{MD\uparrow} &\geq -(t_{C\downarrow} + t_{AND\downarrow}) + t_{OR\uparrow} \\ &\quad + t_{AND\uparrow} + t_{C\uparrow} \\ \text{that is,} \qquad t^N_{MD\uparrow} &\geq t_{AND\uparrow} + t_{OR\uparrow} \\ &\quad - (t_{C\downarrow} + 2 \cdot t_{AND\downarrow}). \end{aligned} \tag{3.10}$$

In (3.10), the occurrences of $t^N_{MD\downarrow}$ and $t_{RD\uparrow}$ have already been replaced with the equivalent gate delays $t_{AND\downarrow}$ and $t_{OR\uparrow}$ respectively.

Case 2: If *Aout* ↓ is late and causes *Rout* ↑, the following holds:

$$\begin{aligned} \text{(P)} \qquad t_{Rin\downarrow \to Rout\uparrow} &= -\left(t_{Ain\uparrow \to Rin\downarrow} + t_{AND\uparrow}\right) \\ &\quad + t_{Rout\downarrow \to Aout\downarrow} + t_{C\uparrow}. \end{aligned} \tag{3.11}$$

The above expression is labelled Ⓒ in the STG diagram in Figure 3.8. From the delay assumption (3.7), it can be rewritten as:

$$\begin{aligned} \text{(P)} \qquad t_{Rin\downarrow \to Rout\uparrow} &\leq -\left(t_{AND\uparrow} + t_{C\downarrow} + t_{AND\downarrow}\right) \\ &\quad + t_{Rout\downarrow \to Aout\downarrow} + t_{C\uparrow}. \end{aligned} \tag{3.12}$$

From (3.5) and (3.12), another conservative version of the constraint (3.2) for $t^N_{MD\uparrow}$ is obtained as follows:

$$\begin{aligned} \text{(C)} \qquad t_{AND\downarrow} + t_{C\uparrow} + t^N_{MD\uparrow} &\geq -(t_{AND\uparrow} + t_{C\downarrow} + t_{AND\downarrow}) \\ &\quad + t_{Rout\downarrow \to Aout\downarrow} + t_{C\uparrow} \\ \text{that is,} \qquad t^N_{MD\uparrow} &\geq t_{Rout\downarrow \to Aout\downarrow} - (t_{C\downarrow} \\ &\quad + t_{AND\uparrow} + 2 \cdot t_{AND\downarrow}). \end{aligned} \tag{3.13}$$

All the constraints derived for $t^N_{MD\uparrow}$ in Cases 1 and 2 (i.e., (3.9), (3.10), and (3.13)) can be satisfied in the preferred application of our controller where there are processing elements within the pipeline and hence the matched delay $t^N_{MD\uparrow}$ is large enough to meet the above constraints.

Constraint 2. The next is a timing constraint to be satisfied by the self resetting delay. The *complete* signal should not be self–reset *before* the *Rout* high is produced. This constraint imposes conditions on the minimum delay for the self–resetting loop

$t_{RD\downarrow}$ to satisfy the above condition. We can formulate this constraint as:

$$\text{(C)} \qquad t_{Rin\downarrow\rightarrow complete\downarrow} \geq t_{Rin\downarrow\rightarrow Rout\uparrow}. \tag{3.14}$$

From Figure 3.6, the causality relation for *Rin* ↓, *Ain* ↓, *RD* ↓, and *complete* ↓ is straightforward. Thus, the left-hand side of the above constraint can be given as:

$$\begin{aligned} \text{(P)} \qquad t_{Rin\downarrow\rightarrow complete\downarrow} &= t_{AND\downarrow} + t_{RD\downarrow} + t_{AND\downarrow} \\ &= t_{RD\downarrow} + 2 \cdot t_{AND\downarrow}. \end{aligned} \tag{3.15}$$

The path of the above expression is labelled Ⓓ in Figure 3.8. The right-hand side of the above constraint is the same as that of (3.2). Thus, exactly the same two cases as those shown for Constraint 1 are considered, and the following three constraints are obtained for (3.14).

Case 1: From (3.15) and (3.8), a conservative version of the constraint (3.14) is obtained as follows:

$$\begin{aligned} &\text{(C)} \qquad t_{RD\downarrow} + 2 \cdot t_{AND\downarrow} \geq t_{AND\uparrow} + t_{C\uparrow} \\ &\text{that is,} \qquad t_{RD\downarrow} \geq t_{AND\uparrow} + t_{C\uparrow} - 2 \cdot t_{AND\downarrow} \end{aligned} \tag{3.16}$$

and

$$\begin{aligned} &\text{(C)} \qquad t_{RD\downarrow} + 2 \cdot t_{AND\downarrow} \geq -(t_{C\downarrow} + t_{AND\downarrow}) + t_{OR\uparrow} \\ &\qquad\qquad\qquad + t_{AND\uparrow} + t_{C\uparrow} \\ &\text{that is,} \qquad t_{RD\downarrow} \geq t_{OR\uparrow} + t_{AND\uparrow} + t_{C\uparrow} \\ &\qquad\qquad\qquad - (t_{C\downarrow} + 3 \cdot t_{AND\downarrow}). \end{aligned} \tag{3.17}$$

Case 2: From (3.15) and (3.12), another conservative version of constraint (3.14) for $t_{RD\downarrow}$ is obtained as follows:

$$\begin{aligned} &\text{(C)} \qquad t_{RD\downarrow} + 2 \cdot t_{AND\downarrow} \geq -(t_{AND\uparrow} + t_{C\downarrow} + t_{AND\downarrow}) \\ &\qquad\qquad\qquad + t_{Rout\downarrow\rightarrow Aout\downarrow} + t_{C\uparrow} \\ &\text{that is,} \qquad t_{RD\downarrow} \geq t_{Rout\downarrow\rightarrow Aout\downarrow} + t_{C\uparrow} \\ &\qquad\qquad\qquad - (t_{AND\uparrow} + t_{C\downarrow} + 3 \cdot t_{AND\downarrow}). \end{aligned} \tag{3.18}$$

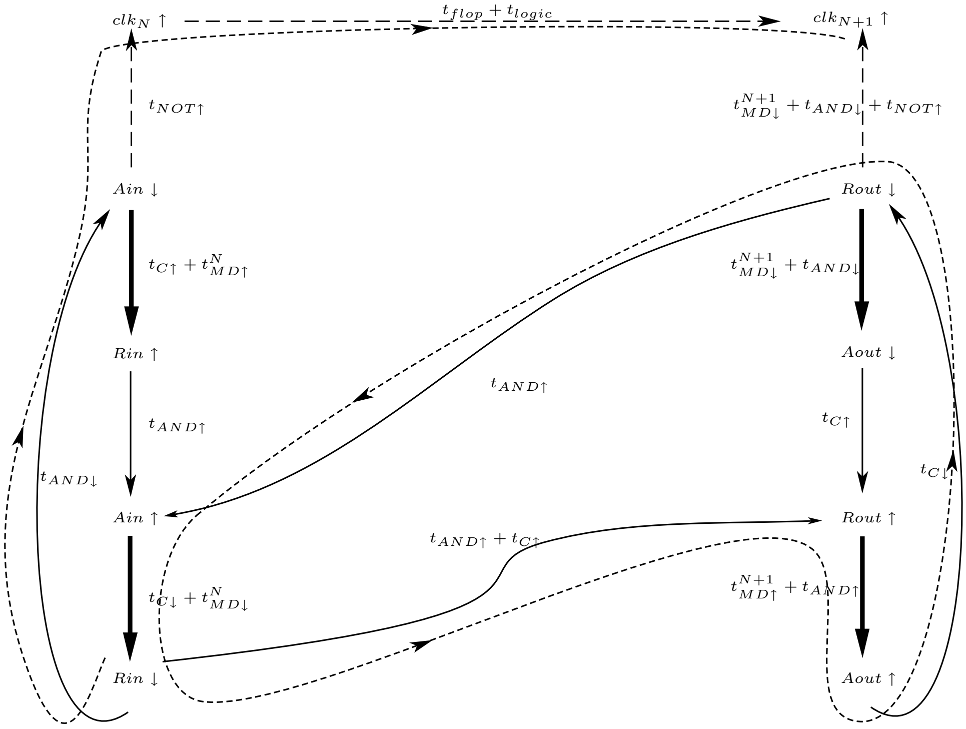


Figure 3.9: STG for EA Controller.

The constraints derived for $t_{RD\downarrow}$ in cases 1 and 2 (i.e. (3.16), (3.17) and (3.18)) should be considered when selecting the minimum delay for the self-resetting loop.

Constraint 3. The last timing constraint prevents $Stage_N$ clock (clk_N) from capturing new data *before* $Stage_{N+1}$ captures data already processed between two stages. When we use *Rout* ↓ event, which signals the arrival of new data to the $Stage_{N+1}$, and unblocks the requests pending at AND gate $A1$, as the starting point of time measurements, this timing constraint can be formulated as follows.

$$\text{(C)} \qquad t_{Rout\downarrow \to clk_N\uparrow} \geq t_{Rout\downarrow \to clk_{N+1}\uparrow}. \tag{3.19}$$

Right-hand side of the above constraint (labelled Ⓕ in Figure 3.8), is the path from the *Rout* falling edge to the capture of data by the clock clk_{N+1}. The left-hand side (labelled Ⓔ in Figure 3.8), is the path from *Rout* falling edge to the capture of new data from clk_N.

$$\text{(P)} \qquad \begin{aligned} t_{Rout\downarrow \rightarrow clk_N\uparrow} = \; & t_{AND\uparrow} + t_{Ain\uparrow \rightarrow Rin\downarrow} \\ & + t_{AND\downarrow} + t_{NOT\uparrow} \\ \geq \; & t_{AND\uparrow} + t_{C\downarrow} + t^{N}_{MD\downarrow} \\ & + t_{AND\downarrow} + t_{NOT\uparrow}. \end{aligned} \tag{3.20}$$

For the worst case of the constraint (3.19), the equality of the property (3.20) should hold. In other words, the input environment is the fastest possible (i.e., delays are equal to those incurred by the linear controller). This gives an upper bound for the right-hand side of the constraint.

$$\text{(P)} \qquad \begin{aligned} t_{Rout\downarrow \rightarrow clk_{N+1}\uparrow} \leq \; & t_{AND\uparrow} + t_{C\downarrow} + t^{N}_{MD\downarrow} \\ & + t_{AND\downarrow} + t_{NOT\uparrow}. \end{aligned} \tag{3.21}$$

In the case of linear controller in the $Stage_{N+1}$, from Figure 3.8, the following holds.

$$\text{(P)} \qquad t_{Rout\downarrow \rightarrow clk_{N+1}\uparrow} = t^{N+1}_{MD\downarrow} + t_{AND\downarrow} + t_{NOT\uparrow}. \tag{3.22}$$

Hence, in the linear pipeline, the above environment delay satisfies (3.21) because, when the equations (3.21) and (3.22) are simplified replacing $t^{N}_{MD\downarrow}$ and $t^{N+1}_{MD\downarrow}$ with constant $t_{AND\downarrow}$ delay, the right-hand sides of the resulting expressions amounts to 5 gate delays and 3 gate delays, respectively.

The controller is model checked using UPPAAL model checker [26, 27] tool to verify the correctness of the functionality and to conclude that the above constraints comprehensively guarantee the operation of the controller.

3.5 Performance

Here, we derive equations for two important performance factors of the pipeline i.e., *forward latency (L)* and *cycle time (T)*. More importantly, we show which components of the latter performance metric can be hidden in the case of a pipeline with logic processing where the EA protocol has a competitive edge. We assume that the controller in the middle stage of a pipeline with the similar controllers in the previous and next stages. In contrast to the constraint analysis, we assume that the controllers are operating at maximum speed in the performance analysis. With these two assumptions we can derive the maximum performance of our controller.

The Figure 3.9 depicts the STG for our controller in desired operation, when it meets the above specified constraints. Here, the environment delays are equal to the delays incurred by the two similar controllers in the previous and next stages according to our second assumption. Dashed-line arrows are for the clock signals of the controller stage and the following stage (clk_N and clk_{N+1}) as well as for the data path between these stages, which are not directly in the control path of the main control logic, but are useful in measuring the cycle time in terms of logic processing delay (t_{logic}). For clarity, not all the transition arcs for these two clock signals are shown.

The cycle time is defined as the interval between two successive data items passing through a pipeline stage when the pipeline is operating at maximum speed. For this purpose, we can measure the gate delays between two successive *clk* rising edges or equivalently the delay between two successive falling edges of *Rin*.

First, we identify the controller's critical cycle using the STG branch and merge points. This critical cycle lies on the path ($Rin \downarrow \rightarrow Rout \uparrow \rightarrow Aout \uparrow \rightarrow Rout \downarrow \rightarrow Ain \uparrow \rightarrow Rin \downarrow$) indicated by the cycle composed of short dashes. It can be proven that the above path is the critical cycle of the controller according to the following argument.

The Figure 3.8 shows 4-cyclic paths that can be considered for the critical cycle of the controller. They are

1. Path: $Rin \downarrow \rightarrow complete \uparrow \rightarrow Rout \uparrow \rightarrow Aout \uparrow \rightarrow Rout \downarrow \rightarrow Ain \uparrow \rightarrow Rin \downarrow$
2. Path: $Ain \uparrow complete \uparrow \rightarrow Rout \uparrow \rightarrow Aout \uparrow \rightarrow Rout \downarrow \rightarrow Ain \uparrow$
3. Path: $Rin \downarrow \rightarrow Ain \downarrow \rightarrow \rightarrow Rin \uparrow \rightarrow Ain \uparrow \rightarrow Rin \downarrow$
4. Path: $Rin \downarrow \rightarrow Ain \downarrow \rightarrow complete \downarrow Rout \downarrow \rightarrow Ain \uparrow \rightarrow Rin \downarrow$

The path with *larger* delays dominates and becomes the critical cycle. It can be observed that the difference between #1 path and #2 path is from $Ain \uparrow$ to $complete \uparrow$ the rest of the paths are common. Thus if,

$$t_{Ain\uparrow \rightarrow Rin\downarrow} + t_{Rin\downarrow \rightarrow complete\uparrow} > t_{Ain\uparrow \rightarrow complete\uparrow} \tag{3.23}$$

then, the #1 path is more critical. Substituting the gate and environment delays from Figure 3.8,

$$t_{C\downarrow} + t^{N}_{MD\downarrow} + t_{AND\uparrow} \geq t_{OR\downarrow} + t_{AND\uparrow} \tag{3.24}$$

This condition can be satisfied with three gate delays (or more from the environment of the input side) on the left-hand side and two internal gate delays of the right- hand side, hence the #1 is more critical than #2 path.

Similar argument can be held for paths #3 and #4. In this case, the segment of path from $Ain \downarrow$ to $Ain \uparrow$ differs while the rest of the two cycles share common paths. Thus if,

$$t_{Ain\downarrow \rightarrow Rin\uparrow} + t_{Rin\uparrow \rightarrow Ain\uparrow} > t_{Ain\downarrow \rightarrow complete\downarrow} + t_{complete\downarrow \rightarrow Rout\downarrow} + t_{Rout\downarrow \rightarrow Ain\uparrow} \tag{3.25}$$

then, the #3 path is more critical than the #4 path. Substituting the gate delays the condition can be obtained as

$$\begin{aligned} t_{C\uparrow} + t^{N}_{MD\uparrow} + t_{AND\uparrow} \geq & t_{RD\downarrow} + t_{AND\downarrow} + t_{C\downarrow} + t_{AND\uparrow} & (3.26) \\ t^{N}_{MD\uparrow} \geq & 2 \cdot t_{AND\downarrow} + t_{C\downarrow} + t_{AND\uparrow} \\ & - (t_{C\uparrow} + t_{AND\uparrow}) & (3.27) \end{aligned}$$

This condition for the matched delay can be satisfied in the applications where our controller is preferred (with large logic delays in between stages), hence the critical cycle is dictated by the #3 path.

Paths #1 and #3 can be compared to determine the critical cycle of the two.

$$\begin{aligned} \#1 \quad path = & t_{AND\uparrow} + t_{C\uparrow} + t^{N+1}_{MD\uparrow} + t_{AND\uparrow} \\ & + t_{C\downarrow} + t_{AND\uparrow} + t_{C\downarrow} + t^{N}_{MD\downarrow} & (3.28) \\ \#3 \quad path = & t_{AND\downarrow} + t_{C\uparrow} + t^{N}_{MD\uparrow} + t_{AND\uparrow} \\ & + t_{C\downarrow} + t^{N}_{MD\downarrow} & (3.29) \end{aligned}$$

Thus, for

$$\#1 \quad path > \#3 \quad path \tag{3.30}$$

and hence be the most critical cycle of all paths the following should be satisfied.

$$t_{MD\uparrow}^{N+1} \geq t_{MD\uparrow}^{N} + t_{AND\downarrow} - (2 \cdot t_{AND\uparrow} + t_{C\downarrow}) \tag{3.31}$$

Given the above condition, the critical cycle is defined by #1 path.

The cycle time can be obtained from the critical path as a function of gate delays and required matched delays (t_{MD}^{N} and t_{MD}^{N+1}) as follows.

$$T = 3 \cdot t_{AND\uparrow} + 2 \cdot t_{C\downarrow} + t_{C\uparrow} + t_{MD\uparrow}^{N+1} + t_{MD\downarrow}^{N}. \tag{3.32}$$

Here, all terms except $t_{MD\uparrow}^{N+1}$ are constant gate delays. To obtain the cycle time and forward latency in terms of logic processing delay (t_{logic}), we need to express the required matched delay t_{MD}^{N+1} for the operations in terms of t_{logic}. When the data is captured with $clk_N \uparrow$, the next stage clock $clk_{N+1} \uparrow$ needs to be made after a delay of $t_{flop} + t_{logic}$, where t_{flop} is the delay of the date register. We can relate t_{logic} to t_{MD}^{N+1} by measuring the same delay in two paths to the event of $clk_{N+1} \uparrow$.

- Path on control cycle: $Rin \downarrow \rightarrow Rout \uparrow \rightarrow Aout \uparrow \rightarrow Rout \downarrow \rightarrow clk_{N+1} \uparrow$

$$\begin{aligned} T_1 = t_{AND\uparrow} + t_{C\uparrow} + t_{MD\uparrow}^{N+1} + t_{AND\uparrow} \\ + t_{C\downarrow} + t_{MD\downarrow}^{N+1} + t_{AND\downarrow} + t_{NOT\uparrow} \end{aligned} \tag{3.33}$$

- Path on data cycle: $Rin \downarrow \rightarrow Ain \downarrow \rightarrow clk_N \uparrow \rightarrow clk_{N+1} \uparrow$

$$T_2 = t_{AND\downarrow} + t_{NOT\uparrow} + t_{flop} + t_{logic} \tag{3.34}$$

To ensure the correct operation of the pipeline, $T_1 \geq T_2$ must hold. Thus, from the above two equations, we can derive an expression for the minimum value of $t_{MD\uparrow}^{N+1}$ as:

$$\begin{aligned} t_{MD\uparrow}^{N+1} \geq (t_{flop} + t_{logic}) \\ - (2 \cdot t_{AND\uparrow} + t_{MD\downarrow}^{N+1} + t_{C\uparrow} + t_{C\downarrow}). \end{aligned} \tag{3.35}$$

Thus, if

$$t_{logic} \geq (2 \cdot t_{AND\uparrow} + t_{MD\downarrow}^{N+1} + t_{C\uparrow} + t_{C\downarrow}) - t_{flop} \tag{3.36}$$

holds, we can find the cycle time in terms of t_{logic} by replacing $t_{MD\uparrow}^{N+1}$ in equation (3.32) by the right-hand side of (3.35). Then the cycle time for the linear controller of EA protocol can be expressed as follows.

$$T^{l}_{EA} = t_{flop} + t_{logic} + t_{AND\uparrow} + t_{C\downarrow}. \tag{3.37}$$

Note that in the above expressions, $t^{N+1}_{MD\downarrow}$ is equal to $t_{AND\downarrow}$ for our implementation shown in Figure 3.2. The convention that we use for cycle time and forward latency consists of the protocol in the subscript (*EA, 2P, 4P*, respectively) and the controller type (*l, cb* for linear- and CB-type controllers, respectively) in the superscript.

If the logic processing time is smaller and the inequality (3.36) does not hold, we obtain the minimum cycle time (maximum throughput) of this controller directly from equation (3.32) by using $t^{N+1}_{MD\uparrow} = t^{N}_{MD\downarrow} = 0$, that is:

$$T^{l}_{EA}|_{min} = 3 \cdot t_{AND\uparrow} + 2 \cdot t_{C\downarrow} + t_{C\uparrow}. \tag{3.38}$$

The above cycle minimum time is valid because it is possible to remove the matched delay without violating the timing constraints derived for $t^{N}_{MD\uparrow}$[2]. This could be confirmed in our experiments as well.

Forward latency is the time taken by a data item to emerge from an initially empty pipeline. Transitions that take place in the forward latency path starting from the *Rin* ↓ of the STG is shown in Figure 3.9 by the line of short dashes. When the inequality (3.36) holds, we can use a similar argument to obtain the forward latency as follows.

$$L^{l}_{EA} = t_{AND\downarrow} + t_{NOT\uparrow} + t_{flop} + t_{logic}. \tag{3.39}$$

When the logic processing delay is small and inequality (3.36) does not hold, the critical path for forward latency lies on the path: *Rin* ↓→ *Rout* ↑→ *Aout* ↑→ *Rout* ↓→ clk_{N+1} ↑, which is:

$$\begin{aligned} L = {} & t_{AND\uparrow} + t_{C\uparrow} + t^{N}_{MD\uparrow} + t_{AND\uparrow} + t_{C\downarrow} \\ & + t^{N}_{MD\downarrow} + t_{AND\downarrow} + t_{NOT\uparrow}. \end{aligned} \tag{3.40}$$

Like the minimum cycle time, the minimum forward latency on this path can be derived using $t^{N}_{MD\uparrow} = t^{N}_{MD\downarrow} = 0$. It is given by

$$L^{l}_{EA}|_{min} = 2 \cdot t_{AND\uparrow} + t_{AND\downarrow} + t_{C\uparrow} + t_{C\downarrow} + t_{NOT\uparrow}. \tag{3.41}$$

[2] In the pipeline, the constraints derived for t^{N}_{MD} of $stage_{N}$ are valid for t^{N+1}_{MD} of $stage_{N+1}$

In a general pipeline with logic processing, condition (3.36) often holds. In that case, the cycle time and forward latency for our controller are given by equations (3.37) and (3.39), respectively.

3.6 Model Checking of Controller

A part from the constraint analysis of the controller, which ensures the operation of the controller, a model checking was performed to formally verify the operation of controller under variety of operating environments. For this purpose, the controller and the input/output environment of the controller is modelled with UPPAL model checking program [26].

3.6.1 UPPAAL Model Checker

UPPAAL is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata, extended with data types (bounded integers, arrays, etc.) [27]. The UPPAAL model checker is based on timed automata (finite state machine extended with clock variables) theory and the query language which is used to specify the properties of the system is a subset of CTL (computation tree logic).

Though the expressiveness of the CTL subset supported by UPPAAL is limited, it provides an extended timed automata system. This was instrumental in modelling of the controller and its environment, and to overcome the limitations of property expression by modelling complex properties into observer automata.

The following UPPAAL extensions of the timed automata are used for the modelling of the system and observer automata.

1. **Templates** automata are defined with a set of parameters that can be of any type (e.g., `int, chan`). These parameters are substituted for a given argument in the process declaration.

2. **Binary synchronisation** channels are declared as `chan c`. An edge labelled with `c!` synchronises with another labelled `c?`. A synchronisation pair is chosen non-deterministically if several combinations are enabled.

3. **Broadcast channels** are declared as broadcast `chan c`. In a broadcast synchronisation one sender `c!` can synchronise with an arbitrary number of receivers `c?`. Any receiver than can synchronise in the current state must do so. If there are no receivers, then the sender can still execute the `c!` action, i.e. broadcast sending is never blocking.

4. **Committed locations** are even more restrictive on the execution than urgent locations. A state is committed if any of the locations in the state is committed. A committed state cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations [28].

3.6.2 Modelling EA Controller with UPPAAL

The model of the EA controller is mainly divided into 3 parts:

1. Model of EA controller. Individual gates which comprise the EA controller are modelled and instantiated in the system. Wires connecting gate elements are modelled as broadcast signals to propagate the changes within the system.

2. Models of the input and output environments.

3. Models of "observers". Some of the properties to be tested against the model of the controller are modelled as automata. This provides a way to test the properties which are difficult and/or impossible to express in the subset of the CTL language supported by UPPAAL.

The following Figure 3.10 shows the model for 2-input logic gate which use to model the AND gate and C-element of the EA controller.

There are 5 states of the 2-input logic gate model. `stable_op` state shown in blue (and a smaller inner circle) is the initial state which corresponds to the state when the logic gate has a stable output `out` depending on its inputs `in1, in2` according to the logic function it represents. The three states with a letter `c` in the middle are committed states in UPPAAL which are used to model transient changes in model execution. The inputs are captured by the model through `change_in1` and `change_in2` broadcast channels. If there is any change in inputs the model moves to `change_in` transient state and try to determine whether the output is going to change due to the input change using

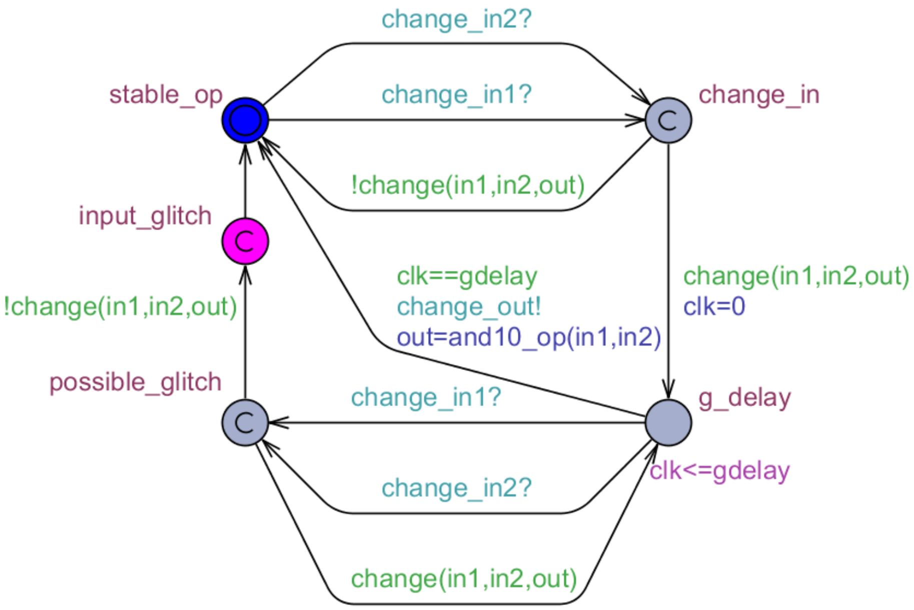


FIGURE 3.10: 2-input logic gate modelled in UPPAAL

`change()` function [3]. If it does change the output the model arrives at `g_delay` state otherwise, it moves back to `stable_op`. In `g_delay` state, the model waits for clock to elapse `gdelay` time units before updating the output and broadcasting the change of the output through `change_out` channel at which point it returns to initial state.

The rest of the two transient states `possible_glitch` and `glitch` states are there to detect any glitches in the inputs that may not get reflected in an output change. Having the `input_glitch` state allows to easily compose properties to detect these conditions which would otherwise require complex CTL expression not permissible in the subset supported in UPPAAL.

The input and output environments are modelled in such a way that the speed of the environment is configurable through parameters passed to the model. The input environment is shown in Figure 3.11.

It should be noted that `dn_max` and `up_max` are configuration parameters for maximum times between *Ain* up to *Rin* down and *Ain* down to *Rin* up, respectively. Using these parameters the input environment can be slowed down or speed up to test the response

[3] The implementation of this model can be found in Appendix A

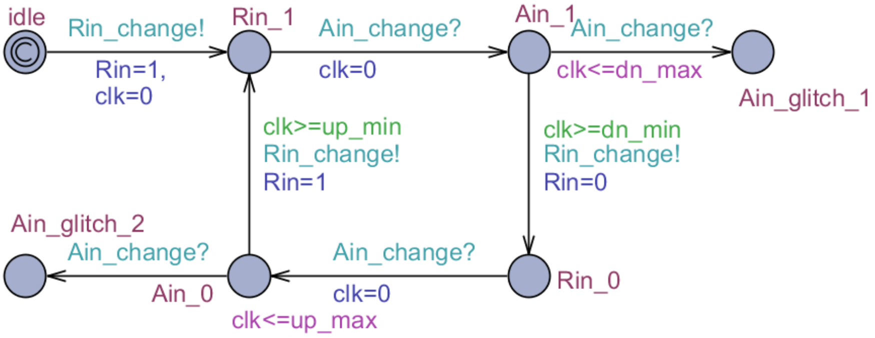


FIGURE 3.11: Input Side Environment modelled in UPPAAL

of the controller model. The details of the model description is not provided, though it can be understood easily using the same UPPAAL construct that we used to model the 2-input logic gate. There are two "glitch detection" states in the model, namely `Ain_glitch_1` and `Ain_glitch_2`. They are modelled also as sink states so that if the model detects a glitch from the controller the entire system is sent into a deadlock which can be easily detected using liveliness properties of the system.

"Observer" automata were instrumental in testing the above constructed UPPAAL model of the EA controller and its environment. Using these automata we overcame the most of the limitations of the CTL subset supported by UPPAAL model checker to express the properties. Instead of modelling the complex properties in CTL, they are modelled as automata. The use of broadcast channels (provided by UPPAAL) to transmit the signal changes within the model was exploited in these observer models. The observer models also listen to these broadcast signals to extract the transitions taking place and model properties into states of the automata. Figure 3.12 shows such observer automata which models the missing clock pulses.

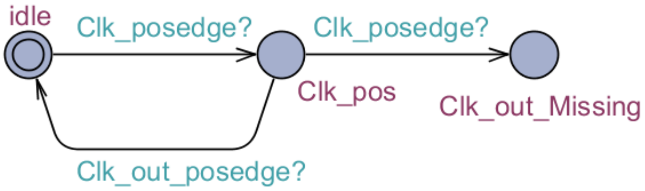


FIGURE 3.12: Input Side Environment modelled in UPPAAL

In the above model if `Clk_posedge` signal is received (i.e. there is a positive edge in the clock signal of the controller) and followed by a `Clk_out_posedge` signal the model returns to the initial state. If however, two consecutive `Clk_posedge` signals are received it would go to the `Clk_out_missing` state which can detected easily testing for the property that whether the system reaches that state.

3.6.3 Properties tested for EA controller model

Following are the properties tested using the above UPPAAL model of the EA controller.

1. Property: `A[] not (Out_env.Rout_glitch_1 or Out_env.Rout_glitch_2)`. No glitches in Rout before Aout changes take place from the output environment.

2. Property: `A[] not (In_env.Ain_glitch_1 or In_env.Ain_glitch_2)`. No glitches in Ain before Rin changes takes place from the input environment.

3. Property: `A[] not (A1_and.input_glitch or A2_and.input_glitch or C_ele.input_glitch )`. No glitches at the inputs of the logic gates of the controller.

4. Property: `A[] not (A1_and.possible_glitch or A2_and.possible_glitch or C_ele.possible_glitch )`. There are no glitches occurred in the combinational gates of the controller

5. Property: `A[] not (obs_Clk_Missing.Clk_Missing)`. For every Rin posedge there is a Clk posedge –> No missing Clk edges for requests.

6. Property: `A[] not (obs_A1_Blocking.A1_Blocking_Failed)`. A1 AND gate successfully blocks if there are consecutive request on Rin.

7. Property: `A[] not (obs_Clk_out_Missing.Clk_out_Missing)`. For every Clk posedge there is Clk_out posedge –> No missing data captures.

8. Property: `(Rin==1) -> (Clk==1)`. Rin posedge always followed by Clk posedge (i.e. there are no missing clock pulses). This property is better correctly tested with the obs1.

9. Property: `A[] not deadlock`. No deadlocks in the system.

Among the properties tested all properties except for property #4 have passed. We expect the glitch behaviour of the controller and verified it is not compromising the correct behaviour of the controller.

Chapter 4

Design and Analysis of Non-linear Controller

4.1 Overview

The Conditional Branch (CB) non-linear pipeline operation is used to demonstrate the simplicity of the EA protocol (which is essentially the4-phase protocol) in composing complex pipeline constructs. First, the abstract operation of the Conditional Branch without any particular reference to a signalling protocol is given followed by the implementation of CB controller for each signalling protocol.

4.2 Operation of CB Controller

In contrast to Fork operation, Conditional Branch operation diverts the data to only *one* branch depending on the *select* signal to the controller. It implements the logical equivalent of an *if-then-else* construct. The interface of a two way Conditional Branch controller is shown in Figure 4.1.

Conditional Branch controller communicates with the input stage by means of *Rin* and *Ain* signals, whereas the two output stage control signals are *Rout1*, *Aout1* and *Rout2*, *Aout2*, respectively. When a request *Rin* is made from the previous stage of the pipeline, data is latched by *clk* signal. Acknowledgement *Ain* is sent to the input stage when data is latched. Depending on the *select* signal, the request is routed on either the first branch

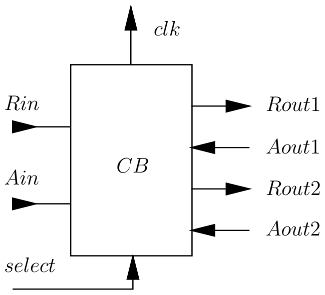


Figure 4.1: Conditional Branch Controller.

Rout1 or the second branch *Rout2*. It is assumed that the *select* signal is generated from several data-path signals.

4.3 Early Acknowledgement CB Controller

The CB controller for the EA protocol is a simple extension of its linear controller. The controller can be composed of a linear controller (for EA protocol), demultiplexer, delay element (SD), and an OR gate. The CB controller at $Stage_N$ of a pipeline is shown in Figure 4.2. The Rin and Ain of the controller are handled by the linear controller used within the CB controller. A function generator *gen* that produces *select* from $stage_N$ data is explicitly considered for analyzing constraints imposed by such an application. In Fig. 4.2, the rectangular box connected to clk_N signal represents the $stage_N$ register which consists of D-flipflops. $data'_N$ is the captured data of $data_N$ in the stage register according to our naming convention. The *select* signal is generated using function generator *gen* from this captured data as shown. The asymmetric delay element SD, which is the same type as MD shown in Fig. 3.2, is used to compensate for the delay of *gen*. An additional constraint on SD for this correct sampling of the *select* signal is presented in the constraint analysis of this controller. The *select* signal diverts the delayed request *req_d* from the linear controller to either the *Rout1* or *Rout2* conditional paths through the demultiplexer. Since only one request is acknowledged from either *Aout1* or *Aout2*, the acknowledgements from the conditional branches can be simply “ORed” to produce the *ack* to the linear controller.

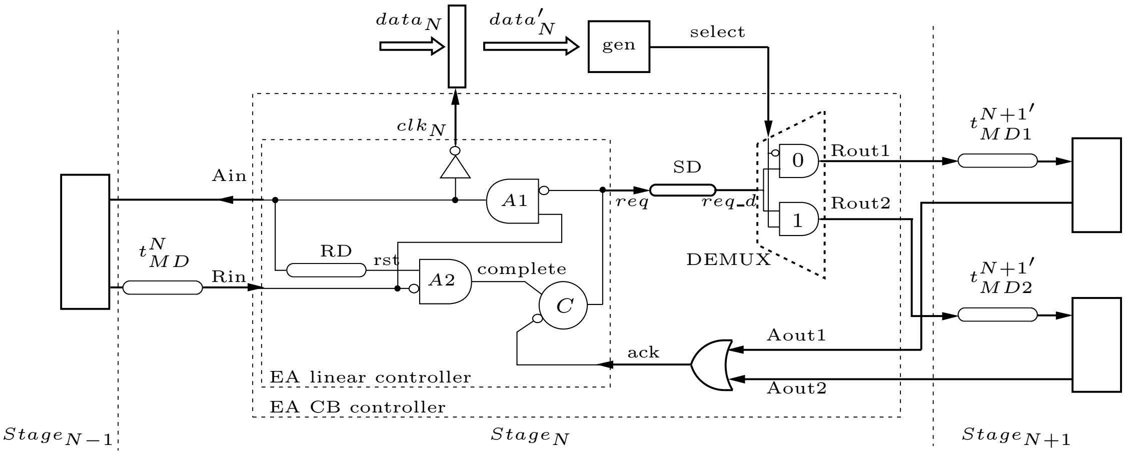


FIGURE 4.2: Early Acknowledgement CB controller.

4.4 Timing Constraints

Timing constraints for the CB controller are analyzed as an extension of the linear controller constraints presented in 3.4. Again, we obtain timing constraints to satisfy the desired operation described in Section 4.2 assuming that the CB controller is in a middle stage of a pipeline with the environment operating at a speed equal to or slower than our linear controller. First, the three constraints presented in Section 3.4 are reevaluated to ensure proper operation of the linear controller used within the CB controller. Then, an additional constraint on t_{SD} for proper operation of demultiplexer is presented.

Constraints 1 and 2. The CB controller is viewed as a linear controller (or linear pipeline) from the input side because; it uses a linear EA controller to communicate with *Rin* and *Ain*. The difference can be perceived only when viewed from the controller's branches owing to the additional branching logic for request and acknowledge. Thus, constraints involving $t_{Rout\downarrow\rightarrow Aout\downarrow}$ should be reconsidered. This corresponds to **Case 2** of each constraint where *Aout* ↓ causes *Rout* ↑. The two constraints (3.13) and (3.18) can be restated for the linear controller within the CB controller using the labels *req*

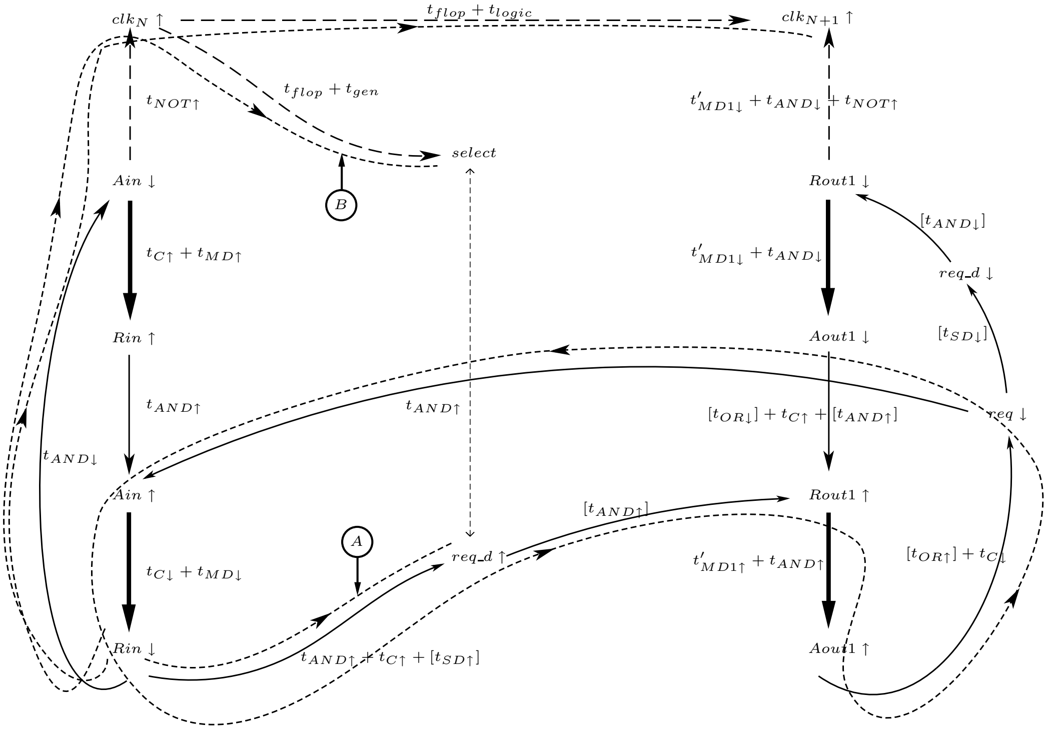


FIGURE 4.3: STG for EA–CB Controller.

and *ack* for the output side as in Figure 4.2.

$$\begin{aligned} \text{(C)} \qquad t^N_{MD\uparrow} &\geq t_{req\downarrow \to ack\downarrow} \\ &\quad - (t_{C\downarrow} + t_{AND\uparrow} + 2 \cdot t_{AND\downarrow}) \qquad (4.1) \\ \text{and,} \quad t_{RD\downarrow} &\geq t_{req\downarrow \to ack\downarrow} + t_{C\uparrow} \\ &\quad - (t_{AND\uparrow} + t_{C\downarrow} + 3 \cdot t_{AND\downarrow}) \qquad (4.2) \end{aligned}$$

where,

$$t_{req\downarrow \to ack\downarrow} = 2 \cdot t_{AND\downarrow} + t_{Rout1\downarrow \to Aout1\downarrow} + t_{OR\downarrow} \qquad (4.3)$$

Here, $t^N_{MD\uparrow}$ and $t_{RD\downarrow}$ should be selected to satisfy these new constraints.

Constraint 3. This constraint can be restated for CB controller from (3.21) as follows.

$$\begin{aligned} \text{(C)} \qquad t_{req\downarrow \to clk_{N+1}\uparrow} &\leq t_{AND\uparrow} + t_{C\downarrow} + t^N_{MD\downarrow} \\ &\quad + t_{AND\downarrow} + t_{NOT\uparrow}. \qquad (4.4) \end{aligned}$$

This condition must be satisfied by any controller in the output stage.

For example, this constraint can be considered for two cases 1) when there is a linear controller at the output stage, and 2) when there is a CB controller at the output stage. It should be noted that there is no difference in these two cases as far as this constraint is concerned because; the input side of a CB controller is composed of a linear controller. Hence, the output stage, *Rout*1/*Aout*1 (and/or *Rout*2/*Aout*2) will be connected to a linear controller in either case. The path delay from *req* ↓ to clk_{N+1} ↑ (similar to (3.22) in linear controller) in above two cases can be obtained from the STG diagram for the CB controller shown in Figure 4.3.

$$\text{(P)} \qquad t_{req\downarrow \to clk_{N+1}\uparrow} = [t_{SD\downarrow} + t_{AND\downarrow}] + t'_{MD1\downarrow} \\ + t_{AND\downarrow} + t_{NOT\uparrow}. \qquad (4.5)$$

Due to the extra overhead incurred by the *SD* and demultiplexer, the property becomes a hard condition to satisfy with 5 gate delays in either side of the inequality. Satisfaction of this constraint can be guaranteed by increasing $t^N_{MD\downarrow}$ to be more than one gate delay at the expense of introducing additional overhead to the critical cycle of the controller. (This can be done by adding buffers in between the *out* port and the rightmost AND gate of the asymmetric delay in Figure 3.2. As a result, it also increases the $t_{MD\uparrow}$ by the same amount of delay incurred by the buffers. It should be adjusted back by reducing the AND gates which comprises the variable part of the delay.)

Constraint 4. An additional constraint on a CB controller requires the *select* signal to be valid *before req_d* goes high. This ensures proper operation of the demultiplexer that switches the request *req* to either branch depending on the *select* signal. The EA protocol stipulates that the *data* becomes valid *before Rin* ↓ arrives. Thus, the worst case for this constraint is when *data* becomes valid simultaneously with *Rin* ↓. In that case, the constraint becomes:

$$\text{(C)} \qquad t_{Rin\downarrow \to req\_d\uparrow} \geq t_{Rin\downarrow \to select}. \qquad (4.6)$$

For *Rin* ↓ to cause *req_d* ↑, at least $t_{AND\uparrow}$ of AND gate *A2*, $t_{C\uparrow}$ of the *C*-element, and $t_{SD\uparrow}$ of *SD* should occur (path labelled Ⓐ in Figure 4.3). Hence, the lower bound for the left-hand side of the above constraint can be expressed as follows.

$$\text{(P)} \qquad t_{Rin\downarrow \to req\_d\uparrow} \geq t_{AND\uparrow} + t_{C\uparrow} + t_{SD\uparrow}. \qquad (4.7)$$

The *Rin* ↓ also causes data capture by lowering *Ain* though the AND gate *A1* and raising the clock NOT gate. The function generator produces the correct *select* signal from the captured data after time t_{gen} has elapsed (path labelled Ⓑ in Figure 4.3). Hence, the right-hand side of the constraint can be expressed as

$$t_{Rin\downarrow\rightarrow select} = t_{AND\downarrow} + t_{NOT\uparrow} + t_{flop} + t_{gen}. \tag{4.8}$$

Thus, the constraint on the asymmetric delay can be derived as

$$\begin{aligned}\text{(C)} \qquad t_{AND\uparrow} + t_{C\uparrow} + t_{SD\uparrow} &\geq t_{AND\downarrow} + t_{NOT\uparrow} + t_{flop} + t_{gen} \\ t_{SD\uparrow} &\geq t_{flop} + t_{gen} + t_{AND\downarrow} + t_{NOT\uparrow} \\ &\quad - (t_{AND\uparrow} + t_{C\uparrow}). \end{aligned} \tag{4.9}$$

This constraint defines the selection of $t_{SD\uparrow}$ based on the select generator function.

4.5 Performance

In this section, the controller's cycle time and forward latency are derived analytically as was done for the linear controller. The minimum time for processing logic t_{logic} that ensures the advantage of the new controller by hiding its overhead is also derived in the form of an inequality similar to that of (3.36).

The STG for the CB controller is shown in Figure 4.3. The arrows with associated delays in square brackets indicate the delays incurred by the controller's extra components (demultiplexer, delay element, and OR gate). We try to differentiate between the linear operation overhead and the additional overhead incurred due to the CB operation, and then reflect them in the equations that we derive as well. The diagram shows the STG for only one branch of the controller (*Rout1/Aout1*) without losing any functional information needed to perform the analysis.

The notable difference from the linear controller's STG is in the matched delays $MD1$ and $MD2$ for the controller's output branches. As shown in Figure 4.2 and detailed in the constraint analysis, part of the output-side matched delays may be used for SD inside our controller to compensate for the select generator function and the demultiplexer delays. The matched delays outside the controller $MD1'$ and $MD2'$ are selected such that the original matched delay remains the same. i.e.,

$$
\begin{aligned}
&t_{MD1\uparrow} = t'_{MD1\uparrow} + t_{SD\uparrow} + t_{AND\uparrow} \\
\text{and} \quad &t_{MD2\uparrow} = t'_{MD2\uparrow} + t_{SD\uparrow} + t_{AND\uparrow}.
\end{aligned}
$$

We can measure the delays in the control cycle and data cycle to derive the cycle time in t_{logic} and the inequality for optimal controller operation. The cycle time (indicated by the cycle composed of short dashes in Figure 4.3) in terms of gate delays can be expressed as follows.

$$
\begin{aligned}
T = &t_{AND\uparrow} + t_{C\uparrow} + [t_{SD\uparrow}] + [t_{AND\uparrow}] \\
&+ t'_{MD1\uparrow} + t_{AND\uparrow} + [t_{OR\uparrow}] + t_{C\downarrow} \\
&+ t_{AND\uparrow} + t_{C\downarrow} + t_{MD\downarrow}.
\end{aligned} \tag{4.10}
$$

The delays enclosed within square brackets indicate the extra delays of the path due to CB operation. We can express this cycle time in terms of t_{logic} by measuring the delays in the control and data paths.

- Path on control cycle: $Rin \downarrow \rightarrow req\_d \uparrow \rightarrow Rout_1 \uparrow \rightarrow Aout_1 \uparrow \rightarrow req \downarrow \rightarrow Rout_1 \downarrow \rightarrow clk_{N+1} \uparrow$

$$
\begin{aligned}
T_{CB1} = &t_{AND\uparrow} + t_{C\uparrow} + [t_{SD\uparrow}] + [t_{AND\uparrow}] \\
&+ t'_{MD1\uparrow} + t_{AND\uparrow} + [t_{OR\uparrow}] + t_{C\downarrow} + [t_{SD\downarrow}] \\
&+ [t_{AND\downarrow}] + t'_{MD1\downarrow} + t_{AND\downarrow} + t_{NOT\uparrow}.
\end{aligned} \tag{4.11}
$$

- Path on data cycle: $Rin \downarrow \rightarrow Ain \downarrow \rightarrow clk_N \uparrow \rightarrow clk_{N+1} \uparrow$

$$
T_{CB2} = t_{AND\downarrow} + t_{NOT\uparrow} + t_{flop} + t_{logic}. \tag{4.12}
$$

Again, for proper operation of the pipeline, $T_{CB1} \geq T_{CB2}$ must hold. This translates to:

$$
\begin{aligned}
t'_{MD1\uparrow} \geq &(t_{flop} + t_{logic}) \\
&- (2 \cdot t_{AND\uparrow} + t_{C\uparrow} + t_{C\downarrow} + t'_{MD1\downarrow}) \\
&- [t_{SD\uparrow} + t_{AND\uparrow} + t_{AND\downarrow} + t_{OR\uparrow} + t_{SD\downarrow}].
\end{aligned} \tag{4.13}
$$

Thus, if

$$\begin{aligned} t_{logic} \geq (2 \cdot t_{AND\uparrow} + t_{C\uparrow} + t_{C\downarrow} + t'_{MD1\downarrow} \\ + [t_{SD\uparrow} + t_{AND\uparrow} + t_{AND\downarrow} + t_{OR\uparrow} \\ + t_{SD\downarrow}]) - t_{flop} \end{aligned} \tag{4.14}$$

holds, the cycle time T^{cb}_{EA} for CB controller can be expressed in terms of t_{logic} by substituting the minimum of (4.13) into equation (4.10).

$$T^{cb}_{EA} = t_{flop} + t_{logic} + t_{AND\uparrow} + t_{C\downarrow} - [2.t_{AND\downarrow}]. \tag{4.15}$$

To simplify the above expression, we use the fact that $t'_{MD1\downarrow} = t_{MD\downarrow} = t_{SD\downarrow} = t_{AND\downarrow}$ in accordance with our implementation.

Note that in inequality (4.14), the additional delays due to CB operation (enclosed in brackets) constitute two parts. 1) t_{SD}, which compensates for t_{gen} and 2) delays due to the DEMUX and the OR gate. When the condition in (4.14) is satisfied, the controller's cycle time is given by (4.15).

The condition given by inequality (4.14) gives the minimum of t_{logic} required to hide the overhead of the controller in this Conditional Branch controller. A similar condition derived for the linear controller is the inequality (3.36) in Chapter 3 which is:

$$t_{logic} \geq (2 \cdot t_{AND\uparrow} + t^{N}_{MD\downarrow} + t_{C\uparrow} + t_{C\downarrow}) - t_{flop} \tag{4.16}$$

According to inequalities of (4.16) and (4.14), minimum of t_{logic} required in order to hide the additional overhead incurred by the Conditional Branch Controller is higher than that of linear controller.

In the case where the logic processing time is too small and inequality (4.14) does not hold, we get the minimum cycle time directly from the equation (4.10) with $t'_{MD1\uparrow} = t_{MD\downarrow} = 0$. It is:

$$\begin{aligned} T^{cb}_{EA}|_{min} = 3 \cdot t_{AND\uparrow} + t_{C\uparrow} + 2 \cdot t_{C\downarrow} \\ + [t_{SD\uparrow} + t_{AND\uparrow} + t_{OR\uparrow}]. \end{aligned} \tag{4.17}$$

Forward latency is also measured in a manner similar to that for the linear controller; this is marked in dashed lines on the STG diagram. For sufficiently large t_{logic}, forward latency has the same terms (despite the minimum t_{logic} being larger) as for the linear controller. That is,

$$L_{EA}^{cb} = t_{AND\downarrow} + t_{NOT\uparrow} + t_{flop} + t_{logic}. \tag{4.18}$$

When t_{logic} is small and the inequality (4.14) does not hold, the critical path lies on the path: $Rin \downarrow \rightarrow req\_d \uparrow \rightarrow Rout_1 \uparrow \rightarrow Aout_1 \uparrow \rightarrow req \downarrow \rightarrow Rout_1 \downarrow \rightarrow clk_{N+1} \uparrow$.

$$\begin{aligned} L = {} & t_{AND\uparrow} + t_{C\uparrow} + [t_{SD\uparrow}] + [t_{AND\uparrow}] + t'_{MD1\uparrow} \\ & + t_{AND\uparrow} + [t_{OR\uparrow}] + t_{C\downarrow} + [t_{SD\downarrow} + t_{AND\downarrow}] \\ & + t'_{MD1\downarrow} + t_{AND\downarrow} + t_{NOT\uparrow}. \end{aligned} \tag{4.19}$$

Similar to the minimum cycle time, the minimum forward latency can be derived for this case when $t'_{MD1\uparrow} = t'_{MD1\downarrow} = 0$ as follows.

$$\begin{aligned} L_{EA}^{cb}|_{min} = {} & 2 \cdot t_{AND\uparrow} + t_{AND\downarrow} + t_{C\uparrow} + t_{C\downarrow} + t_{NOT\uparrow} \\ & + [t_{SD\uparrow} + t_{AND\uparrow} + 2.t_{AND\downarrow} + t_{OR\uparrow}]. \end{aligned} \tag{4.20}$$

Again, the fact that $t_{SD\downarrow} = t_{AND\downarrow}$ is used to simplify the above equation.

Chapter 5

Design and Analysis of non-linear 2-phase Controller

5.1 Overview

To demonstrate the advantage of the Early Acknowledgement (EA)–protocol–based controllers, we compared its performance with 2- and 4-phase pipeline controllers. This chapter details the 2-phase controllers (linear and conditional branch controllers) that are used for comparison and their key features.

5.2 2-phase Controller: MOUSETRAP

For the 2-phase or the transition signalling protocol, the MOUSETRAP controller was selected for its simplicity and high performance. As shown in Figure 5.1, this controller consists of a simple transparent latch (denoted by the rectangular box) and an XONR gate. The signal *enable* is used to drive data latches (instead of D-flipflops) in the data path.

Initially, all control signals are low except for the *enable* signals that make all the pipeline stages transparent. When the first data item flows through the pipeline stage, it flips the values of Rin_N, $Rout_N$, and Ain_N exactly once (to high). Subsequently, the second data item flips all these signals once again (to low). This is 2-phase (or transition) signalling where each transition (either up or down) indicates a distinct request or acknowledgement.

Once a data item has been captured by stage latches, three actions occur in parallel: (i) the request to the next-stage $Rout_N$ is made; (ii) an acknowledgement, Ain_N, is sent to the previous stage, allowing the next data item to be sent; and finally (iii) $enable_N$ is lowered to make the stage latches opaque, protecting the current data from being overwritten. Subsequently, when an acknowledgement, $Aout_N$ is received from $stage_{N+1}$, the latch in $stage_N$ is re-enabled (i.e., made transparent).

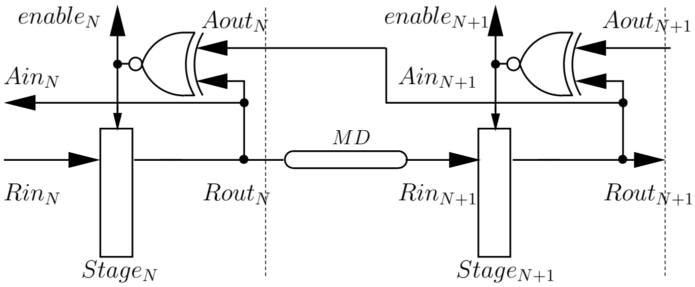


FIGURE 5.1: MOUSETRAP controller.

In [15], the operation of the MOUSETRAP controller in both high-speed pipelines and pipelines with logic processing is described in detail. The authors also presented that controller's cycle time and the forward latency. The most important point to note regarding the EA protocol and 4-phase signalling protocol is that there is no resetting overhead in the 2-phase protocol, hence there is no resetting overhead in controller either. We have extended the original derivation of cycle time and latency for MOUSETRAP to cases with and without logic processing.

5.2.1 Performance of MOUSETRAP controller

Here we use the STG (Figure 5.2) presented in [15] (with our naming conventions for the control signals) to derive the performance of MOUSETRAP controller. Also note that the signals request/acknowledge signals confirm to 2-phase protocol hence an arrow to/from those signals imply a "transition". The cycle time lies on the path marked in dashed line and the forward latency can be measured on the same path from $Ain_N \rightarrow$

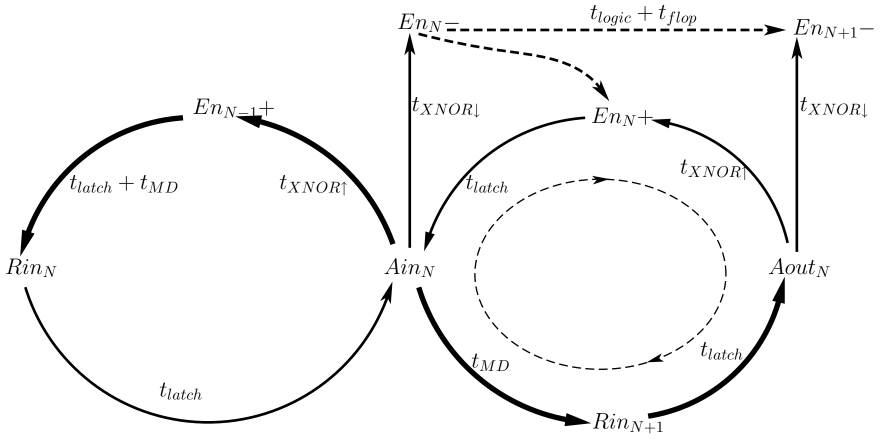


Figure 5.2: STG for MOUSETRAP Controller.

$Rin_{N+1} \rightarrow Aout_N$. Hence,

$$\begin{aligned} T &= t_{MD} + t_{latch} + t_{XNOR\uparrow} + t_{latch} \\ &= t_{MD} + 2 \cdot t_{latch} + t_{XNOR\uparrow} \qquad (5.1) \\ L &= t_{MD} + t_{latch}. \qquad (5.2) \end{aligned}$$

To express the above equations in terms of t_{logic}, the time can be measured on control path and data path as follows.

- Path on control cycle: $Ain_N \rightarrow Rin_{N+1} \rightarrow Aout_N \rightarrow En_{N+1}-$

$$T_1 = t_{MD} + t_{latch} + t_{XNOR\downarrow} \qquad (5.3)$$

- Path on data cycle: $Ain_N \rightarrow En_N- \rightarrow En_{N+1}-$

$$T_2 = t_{XNOR\downarrow} + t_{logic} + t_{latch}. \qquad (5.4)$$

To correctly latch the data at the next stage ($En_{N+1}-$) it is required that $T_1 \geq T_2$ which lead to below condition:

$$\begin{aligned} t_{MD} + t_{latch} + t_{XNOR\downarrow} &\geq t_{XNOR\downarrow} + t_{logic} + t_{latch} \\ t_{MD} &\geq t_{logic}. \qquad (5.5) \end{aligned}$$

In other words, the matched delay should be selected to be equal or greater than the logic delay. The optimal matched delay is when $t_{MD} = t_{logic}$. The cycle time and forward latency will be:

$$T = t_{logic} + 2 \cdot t_{latch} + t_{XNOR\uparrow} \tag{5.6}$$

$$L = t_{logic} + t_{latch}. \tag{5.7}$$

In contrast to Early Acknowledgement controllers, the above equations holds for *any* logic delay making it a very high performance pipeline controller specially when the logic processing is very low limited to one or two gate delays. The maximum performance (minimum cycle time and latency) for this controller is when $t_{logic} = 0$ which can be given as:

$$T|_{min} = 2 \cdot t_{latch} + t_{XNOR\uparrow} \tag{5.8}$$

$$L|_{min} = t_{latch}. \tag{5.9}$$

5.3 2-phase Conditional Branch (CB) Controller

The CB controller for the transition signalling protocol is not as straightforward as in the EA or 4-phase protocol. Since there is no resetting of the request or acknowledgement signal, we cannot make use of a demultiplexer to route the request on the sampled *select* signal. The CB controller for 2-phase protocol based on [29] is shown in Figure 5.3. Note that D-flops are used instead of the transparent latches used in the MOUSETRAP controller for a linear pipeline because; the D-flipflop-based controller is more robust than the transparent-latch-based controller in this case.

Initially, all control signals are in the same state and the *complete* signal is high, which indicates that the controller's output side operations are complete. The *select* signal can be either high or low depending on the data or other control information that handles the branching operation. When a request is made with a transition on *Rin*, the difference between the states of *Rin* and *Ain* generates the *clk* signal, which is gated by *complete*. Since *complete* is initially high, the *clk* signal is raised and captures the control and data signals. Once the *Rin* is captured, the same transition occurs in *Ain*, which acknowledges the request to the input side. The *s1* and *s2* flipflops work as a "transition demultiplexer" that generates the requests on either *Rout1* or *Rout2* depending on the *select* signal. The transition on *Rout1* or *Rout2* is made using its previous level from *Aout1* or *Aout2*, respectively, and inverting it through the two XOR gates. The first XOR gate generates "$Rout1 = \overline{Aout1}$ when $select = 0$", whereas the second XOR gate generates "$Rout2 = \overline{Aout2}$ when $select = 1$". For example, if the *select* signal is low, *s1* captures $\overline{Aout1}$, generating transitions on *Rout1*, i.e., requests on the first branch. The *SD* delay element is used to appropriately delay the request to match the the select generator block *gen*, so that a valid *select* signal is used to drive $s1$ and $s2$ flops.

Either request event causes the *complete* signal to go low, indicating that the latched data is being passed to the output stage, which effectively blocks new requests from the input side. Upon the acknowledgement of the corresponding branch, each pair of request and acknowledgement signals returns to the same state, raising the *complete* signal high and re-enabling the requests from the input side. In comparison to the minimal overhead of the linear controller (MOUSETRAP), *s1* and *s2* the request-generating toggle flops and completion detection mechanism incur a considerable overhead in their operation, which adversely affects the controller's performance.

5.3.1 Performance of CB controller for 2-phase protocol

The STG for the functional operation of the 2-phase Conditional Branch Controller is given in Figure 5.4. In this STG also, transitions for only one branch of the controller

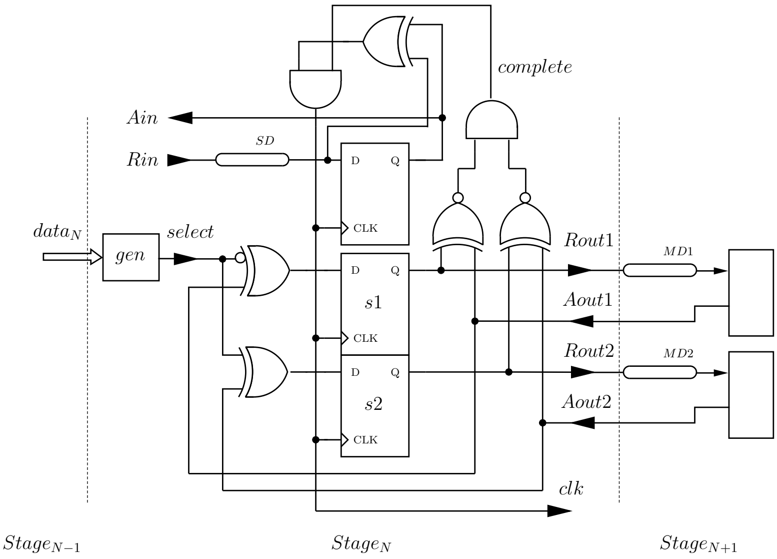


FIGURE 5.3: 2-phase CB controller.

are given. Note that for signals *Rin*, *Ain*, $Rout_1$ and $Aout_1$ STG implies a "transition" without explicitly unfolding the particular transitions (from low-to-high and high-to-low) which are identical in the protocol. The matched delays are symmetrical for the transition signalling protocol. Hence there is no distinction made between $t_{MD\uparrow}$ and $t_{MD\downarrow}$ as in the case of other two protocols.

We can calculate the cycle time and forward latency in terms of t_{logic} using the same rationale used in Conditional Branch controller of the Early Acknowledgement protocol. The cycle time and forward latency of the controller as shown in the STG with thin dashed lines can be expressed as:

$$\begin{aligned} T = {} & t_{flop} + t_{MD} + t_{latch} + t_{XNOR\uparrow} \\ & + 2 \cdot t_{AND\uparrow} \end{aligned} \tag{5.10}$$

$$\begin{aligned} L = {} & t_{XOR\uparrow} + t_{AND\uparrow} + t_{flop} + t_{MD} \\ & + t_{latch} + t_{XNOR\downarrow}. \end{aligned} \tag{5.11}$$

In order to express above in terms of t_{logic} we measure the delays on the control and data paths.

- Path on control cycle: $clk_N+ \rightarrow Rout_1 \rightarrow Aout_1 \rightarrow En-$

$$T_{CB1} = t_{flop} + t_{MD} + t_{latch} + t_{XNOR\downarrow} \tag{5.12}$$

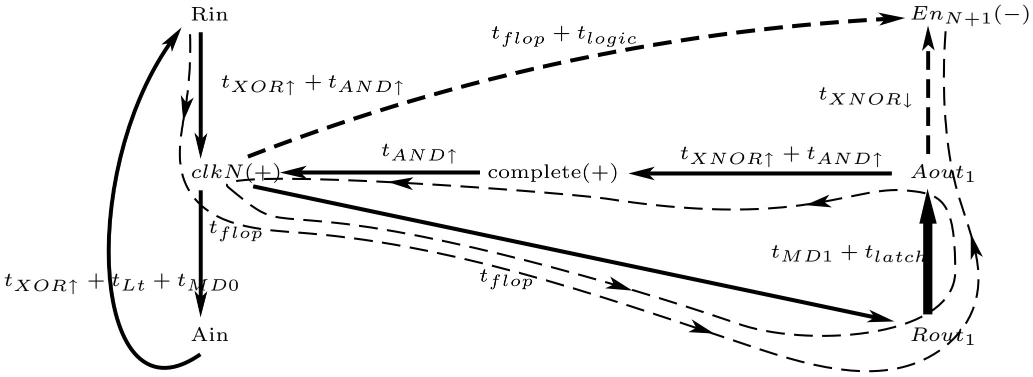


FIGURE 5.4: STG for Conditional Branch Controller for 2-phase protocol.

- Path on data cycle: $clk_N+ \rightarrow En-$

$$T_{CB2} = t_{flop} + t_{logic}. \tag{5.13}$$

Since $T_{CB1} \geq T_{CB2}$ for proper operation, we can obtain the constraint on t_{MD1} as:

$$t_{MD} \geq t_{logic} - (t_{latch} + t_{XNOR\downarrow}) \tag{5.14}$$

$$\text{thus, if,} \quad t_{logic} \geq t_{latch} + t_{XNOR\downarrow} \tag{5.15}$$

holds, the cycle time and latency for this controller can be obtained as:

$$\begin{aligned} T &= t_{flop} + t_{logic} + 2 \cdot t_{AND\uparrow} \\ &\quad + (t_{XNOR\uparrow} - t_{XNOR\downarrow}) \qquad (5.16) \\ L &= t_{flop} + t_{logic} + t_{XOR\uparrow} + t_{AND\uparrow}. \qquad (5.17) \end{aligned}$$

When the above condition does not hold we can derive the cycle time and latency directly from (5.10) and (5.11) at $t_{MD} = 0$ as follows.

$$T|_{min} = t_{flop} + t_{latch} + t_{XNOR\uparrow} + 2 \cdot t_{AND\uparrow} \tag{5.18}$$

$$L|_{min} = t_{XOR\uparrow} + t_{AND\uparrow} + t_{flop} + t_{latch} + t_{XNOR\downarrow}. \tag{5.19}$$

5.4 Summary

The performance analysis can be summarized as follows. For Linear Controller :

$$T^l_{2P} = 2 \cdot t_{latch} + t_{logic} + t_{XNOR\uparrow} \tag{5.20}$$

$$L^l_{2P} = t_{latch} + t_{logic}, \tag{5.21}$$

The minimum cycle time and forward latency can be derived from the above equations when $t_{logic} = 0$ as follows.

$$T^l_{2P}|_{min} = 2 \cdot t_{latch} + t_{XNOR\uparrow} \tag{5.22}$$

$$L^l_{2P}|_{min} = t_{latch}. \tag{5.23}$$

For Conditional Branch controller:

$$\text{if,} \quad t_{logic} \geq t_{latch} + t_{XNOR\downarrow} \tag{5.24}$$

holds, the cycle time and forward latency for this controller can be obtained as:

$$\begin{aligned} T_{2P}^{cb} &= t_{flop} + t_{logic} + 2 \cdot t_{AND\uparrow} \\ &\quad + (t_{XNOR\uparrow} - t_{XNOR\downarrow}) \end{aligned} \tag{5.25}$$

$$L_{2P}^{cb} = t_{flop} + t_{logic} + t_{XOR\uparrow} + t_{AND\uparrow}. \tag{5.26}$$

When the above condition does not hold the minimum of these two parameters are obtained as follows.

$$T_{2P}^{cb}|_{min} = t_{flop} + t_{latch} + t_{XNOR\uparrow} + 2 \cdot t_{AND\uparrow} \tag{5.27}$$

$$L_{2P}^{cb}|_{min} = t_{XOR\uparrow} + t_{AND\uparrow} + t_{flop} + t_{latch} + t_{XNOR\downarrow}. \tag{5.28}$$

Chapter 6

Analysis of 4-phase Controllers

6.1 Overview

This chapter details the 4-phase linear and conditional-branch controllers used for the comparison of Early Acknowledgement controllers. Similar to the 2-phase controllers in previous chapter, the performance of those controllers are analysed for comparison.

6.2 4-phase Linear Controller

We used the 4-phase controller proposed in [24] for this comparison. That is shown in Figure 6.1. *G1* and *G2* are complex gates composing the controller.

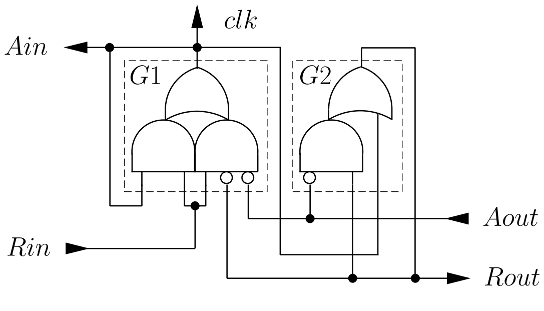


FIGURE 6.1: 4-phase controller.

6.3 4-phase Conditional Branch controller

The construction of the CB controller for the 4-phase protocol is similar to that for the EA protocol. It is the same as Figure 4.2, except that a 4-phase linear controller (Figure 6.1) is used in place of the EA linear controller. Moreover, the adjacent stages also contain the 4-phase linear controllers for this case. The operation as described in previous Section 4.3 is valid for the 4-phase Conditional Branch controller as well.

The cycle time and forward latency of this controller are obtained in a way similar to that of CB controller for the EA protocol. A brief performance analysis to obtain its cycle time and latency is presented below.

6.4 Performance Analysis of 4-phase Linear Controller

We could derive the cycle time and latency for this 4-phase controller using a similar mechanism to the one used in the EA protocol controller. The STG for obtaining the cycle time and latency is shown in Figure 6.3. Quite evidently the cycle time of the 4-phase controller has controller overhead which lies on the critical cycle and it cannot be hidden by the matched delay, unlike Early Acknowledgement controller. The analysis of the cycle time and latency is similar to the Early Acknowledgement controller presented in Section 3.5; hence we left out trivial deductions that can be made directly from the STG diagram.

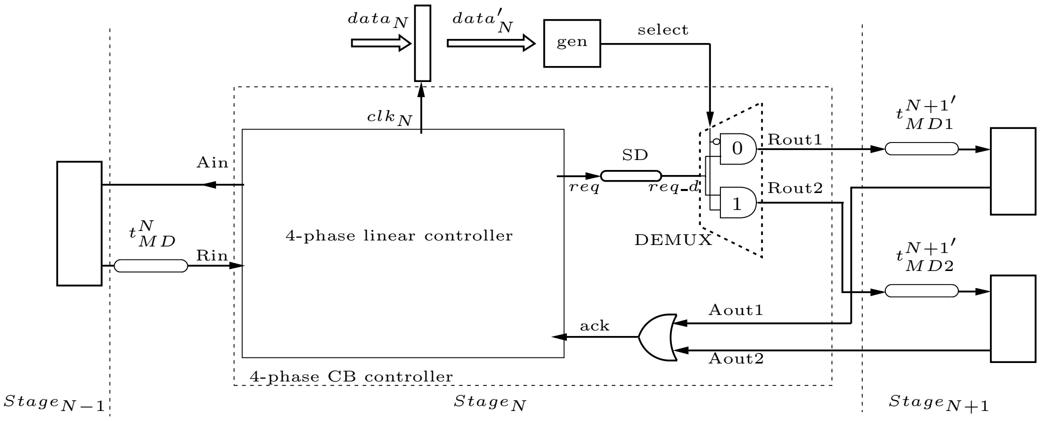


FIGURE 6.2: 4-Phase CB controller.

We used the *Rin*+ as the starting transition confirming to the semantics of the 4-phase protocol. The critical path lies on the dashed line path which constitutes a twisted loop. Hence, the cycle time and forward latency can be obtained as a function of gate delays (starting from *Rin*+) as follows.

$$\begin{aligned} T = {} & t_{G1\uparrow} + t_{G2\uparrow} + t_{MD\uparrow} + t_{G1\uparrow} \\ & + t_{G2\downarrow} + t_{MD\downarrow} + t_{G1\downarrow} + t_{G2\uparrow} \end{aligned} \tag{6.1}$$

$$L = t_{G1\uparrow} + t_{G2\uparrow} + t_{MD\uparrow} + t_{G1\uparrow} \tag{6.2}$$

To bring in the t_{logic} to the above equations two paths on control and data cycles are considered.

- Path on control cycle: *Rin*+ → *Ain*+ → *Rout*+ → *Aout*+

$$T_1 = t_{G1\uparrow} + t_{G2\uparrow} + t_{MD\uparrow} + t_{G1\uparrow} \tag{6.3}$$

- Path on data cycle: *Rin*+ → *Ain*+ → *Aout*+

$$T_2 = t_{G1\uparrow} + t_{flop} + t_{logic} \tag{6.4}$$

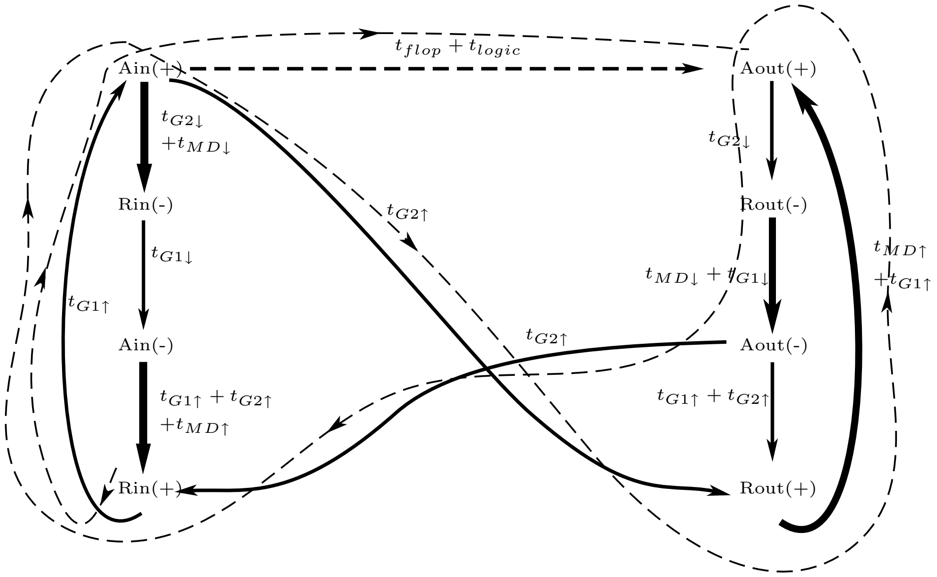


Figure 6.3: STG for 4-phase Controller.

From the $T_1 \geq T_2$ condition for proper operation we have:

$$t_{MD\uparrow} \geq t_{flop} + t_{logic} - (t_{G1\uparrow} + t_{G2\uparrow})$$

$$\text{Thus, if,} \quad t_{logic} \geq t_{G1\uparrow} + t_{G2\uparrow} - t_{flop} \tag{6.5}$$

holds, the cycle time and forward latency of the 4-phase controller can be expressed as follows.

$$\begin{aligned} T &= t_{flop} + t_{logic} + t_{G1\uparrow} + t_{G2\uparrow} \\ &\quad + t_{G1\downarrow} + t_{G2\downarrow} + t_{AND\downarrow} \qquad (6.6) \\ L &= t_{flop} + t_{logic} + t_{G1\uparrow} \qquad (6.7) \end{aligned}$$

When the above condition does not hold the above parameters can be deduced from equations (6.1) and (6.2) at $t_{MD\uparrow} = t_{MD\downarrow} = 0$.

$$\begin{aligned} T|_{min} &= t_{flop} + t_{logic} + t_{G1\uparrow} + t_{G2\uparrow} \\ &\quad + t_{G1\downarrow} + t_{G2\downarrow} + t_{AND\downarrow} \qquad (6.8) \\ L|_{min} &= t_{flop} + t_{logic} + t_{G1\uparrow} \qquad (6.9) \end{aligned}$$

6.5 Performance analysis of 4-phase CB Controller

The Figure 6.4 shows the STG for the Conditional Branch controller for 4-phase protocol. The analysis of the cycle time and forward latency is similar to that of Early Acknowledgement protocol Conditional Branch controller presented in Section 4.5.

The arrows with associated delays in square brackets indicate the delays incurred by the extra components (demultiplexer, delay element and OR gate) of the controller. With the same reasoning that we followed for Conditional Branch controllers for Early Acknowledgement protocol, we can derive expressions for cycle time and forward latency in terms of t_{logic}. The cycle time and forward latency as marked in thin dashed lines in the STG can be expressed in terms of gate delays as follows.

$$\begin{aligned}
T &= t_{G1\uparrow} + t_{G2\uparrow} + [t_{SD\uparrow} + t_{AND\uparrow}] + t'_{MD1\uparrow} \\
&\quad + t_{G1\uparrow} + [t_{OR\uparrow}] + t_{G2\downarrow} + [t_{AND\downarrow}] \\
&\quad + t'_{MD1\downarrow} + t_{G1\downarrow} + [t_{OR\downarrow}] + t_{G2\uparrow} \qquad (6.10) \\
L &= t_{G1\uparrow} + t_{G2\uparrow} + [t_{SD\uparrow} + t_{AND\uparrow}] \\
&\quad + t'_{MD1\uparrow} + t_{G1\uparrow} \qquad (6.11)
\end{aligned}$$

In order to express above two parameters in terms of t_{logic} two paths on control and data cycles are considered.

- Path on control cycle: *Rin+ → Ain+ → req_d+ → Rout+ → Aout+*

$$\begin{aligned}
T_1 &= t_{G1\uparrow} + t_{G2\uparrow} + [t_{SD\uparrow} + t_{AND\uparrow}] \\
&\quad + t'_{MD\uparrow} + t_{G1\uparrow} \qquad (6.12)
\end{aligned}$$

- Path on data cycle: *Rin+ → Ain+ → Aout+*

$$T_2 = t_{G1\uparrow} + t_{flop} + t_{logic} \qquad (6.13)$$

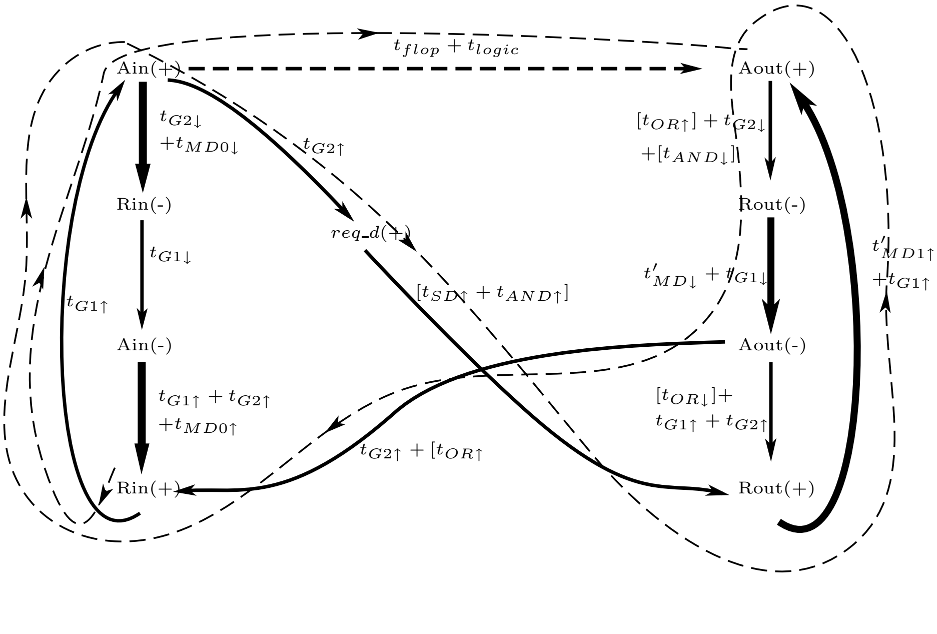

$t_{flop} + t_{logic}$
Ain(+)
Aout(+)
$t_{G2\downarrow}$ $+t_{MD0\downarrow}$
$t_{G2\uparrow}$
$[t_{OR\uparrow}] + t_{G2\downarrow}$ $+[t_{AND\downarrow}]$
Rin(-)
Rout(-)
$req\_d(+)$
$t_{G1\downarrow}$
$t'_{MD1\uparrow}$ $+t_{G1\uparrow}$
$t'_{MD\downarrow} + t_{G1\downarrow}$
$t_{G1\uparrow}$
$[t_{SD\uparrow} + t_{AND\uparrow}]$
Ain(-)
Aout(-)
$t_{G1\uparrow} + t_{G2\uparrow}$ $+t_{MD0\uparrow}$
$[t_{OR\downarrow}]+$ $t_{G1\uparrow} + t_{G2\uparrow}$
$t_{G2\uparrow} + [t_{OR\uparrow}$
Rin(+)
Rout(+)

Figure 6.4: STG for Conditional Branch Controller for 4-phase protocol.

From the $T_1 \geq T_2$ condition for proper operation we have:

$$\begin{aligned} t'_{MD\uparrow} &\geq t_{flop} + t_{logic} - (t_{G1\uparrow} + t_{G2\uparrow} \\ &\quad + [t_{SD\uparrow} + t_{AND\uparrow}]) \\ \text{Thus, if,} \quad t_{logic} &\geq (t_{G1\uparrow} + t_{G2\uparrow} + [t_{SD\uparrow} + t_{AND\uparrow}]) \\ &\quad - t_{flop} \qquad (6.14) \end{aligned}$$

holds, the cycle time and forward latency of the 4-phase controller can be expressed as follows.

$$\begin{aligned} T &= t_{flop} + t_{logic} + t_{G1\uparrow} + t_{G2\uparrow} + t_{G1\downarrow} \\ &\quad + t_{G2\downarrow} + t_{AND\downarrow} + [t_{AND\downarrow} + t_{OR\uparrow} + t_{OR\downarrow}] \qquad (6.15) \\ L &= t_{flop} + t_{logic} + t_{G1\uparrow} \qquad (6.16) \end{aligned}$$

When the above condition does not hold the minimum values for above parameters can be deduced from equations (6.10) and (6.11) at $t'_{MD\uparrow} = t'_{MD\downarrow} = 0$.

$$\begin{aligned} T|_{min} &= 2 \cdot (t_{G1\uparrow} + t_{G2\uparrow}) + t_{G1\downarrow} + t_{G2\downarrow} \\ &\quad + [t_{SD\uparrow} + t_{AND\uparrow} + t_{AND\downarrow} + t_{OR\uparrow} \\ &\quad + t_{OR\downarrow}] \qquad (6.17) \\ L|_{min} &= 2 \cdot t_{G1\uparrow} + t_{G2\uparrow} + [t_{SD\uparrow} + t_{AND\uparrow}] \\ &\qquad (6.18) \end{aligned}$$

6.6 Summary

The results can be summarized as follows. When the t_{logic} is large enough such that the inequality

$$t_{logic} \geq t_{G1\uparrow} + t_{G2\uparrow} - t_{flop} \qquad (6.19)$$

holds, the cycle time and the latency can be expressed as:

$$\begin{aligned} T^l_{4P} &= t_{flop} + t_{logic} + t_{G1\uparrow} + t_{G2\uparrow} \\ &\quad + t_{G1\downarrow} + t_{G2\downarrow} + t_{AND\downarrow} \qquad (6.20) \\ L^l_{4P} &= t_{flop} + t_{logic} + t_{G1\uparrow} \qquad (6.21) \end{aligned}$$

When the t_{logic} is small that the above inequality does not hold, the cycle time and latency take the following form.

$$T^{l}_{4P}|_{min} = 2 \cdot (t_{G1\uparrow} + t_{G2\uparrow}) + t_{G1\downarrow} + t_{G2\downarrow} \tag{6.22}$$

$$L^{l}_{4P}|_{min} = 2 \cdot t_{G1\uparrow} + t_{G2\uparrow} \tag{6.23}$$

For the Conditional Branch controller the obtained expressions can be summarized as follows.

$$\begin{aligned} \text{if,} \quad t_{logic} \geq (t_{G1\uparrow} + t_{G2\uparrow} + [t_{SD\uparrow} + t_{AND\uparrow}]) \\ - t_{flop} \end{aligned} \tag{6.24}$$

Then,

$$\begin{aligned} T^{cb}_{4P} &= t_{flop} + t_{logic} + t_{G1\uparrow} + t_{G2\uparrow} + t_{G1\downarrow} \\ &\quad + t_{G2\downarrow} + t_{AND\downarrow} + [t_{AND\downarrow} + t_{OR\uparrow} + t_{OR\downarrow}] \end{aligned} \tag{6.25}$$

$$L^{cb}_{4P} = t_{flop} + t_{logic} + t_{G1\uparrow} \tag{6.26}$$

Otherwise, the minimum of cycle time and forward latency are,

$$\begin{aligned} T^{cb}_{4P}|_{min} &= 2 \cdot (t_{G1\uparrow} + t_{G2\uparrow}) + t_{G1\downarrow} + t_{G2\downarrow} \\ &\quad + [t_{SD\uparrow} + t_{AND\uparrow} + t_{AND\downarrow} + t_{OR\uparrow} \\ &\quad + t_{OR\downarrow}] \end{aligned} \tag{6.27}$$

$$L^{cb}_{4P}|_{min} = 2 \cdot t_{G1\uparrow} + t_{G2\uparrow} + [t_{SD\uparrow} + t_{AND\uparrow}] \tag{6.28}$$

Chapter 7

Comparison of Controllers

7.1 Overview

This chapter provides the comparison of performance of the of the controllers of 4-phase, 2-phase and Early Acknowledgement protocols both linear and non-linear derived in last four Chapters. Moreover, a test carried out substantiate the analytical results is presented. The simulation results obtained are discussed in view of the analytical comparison.

7.2 Linear Controllers

The merits of using the EA controller could be observed in the case where t_{logic} satisfies the condition derived in (3.36) in Chapter 3. Then, the cycle time for our controller is given by (3.37). That is:
If,

$$t_{logic} \geq (2 \cdot t_{AND\uparrow} + t^{N}_{MD\downarrow} + t_{C\uparrow} + t_{C\downarrow}) - t_{flop} \tag{7.1}$$

Then,

$$T^{l}_{EA} = t_{flop} + t_{logic} + t_{AND\uparrow} + t_{C\downarrow} \tag{7.2}$$

This can be compared analytically with the 2- and 4-phase protocols using equations (5.20) and (6.20). For 2-phase controller:

$$T^{l}_{2P} = 2 \cdot t_{latch} + t_{logic} + t_{XNOR\uparrow} \tag{7.3}$$

For 4-phase controller:

$$\begin{aligned} T^{l}_{4P} &= t_{flop} + t_{logic} + t_{G1\uparrow} + t_{G2\uparrow} \\ &\quad + t_{G1\downarrow} + t_{G2\downarrow} + t_{AND\downarrow} \end{aligned} \tag{7.4}$$

It is not possible to compare the cycle times without specific delays obtained from technology libraries. However, we can get an idea of the controller overhead in the overall cycle time in each case. Note that the data-path delays for our controller and for the 4-phase controller are both $t_{flop} + t_{logic}$ because; we use D-flipflops on the data-path. In the case of the MOUSETRAP controller, the data-path delay is $t_{latch} + t_{logic}$ as a result of the use of transparent latches. Any additional terms appearing in the cycle-time expressions apart from the data-path delays are incurred by the controller's overhead. Hence, our controller has an overhead of only two gate delays ($t_{AND\uparrow} + t_{C\downarrow}$), which is comparable to the 2-phase controller's overhead ($t_{latch} + t_{XNOR\uparrow}$). For the 4-phase controller the overhead is 5 gate delays, which is incurred by the resetting period.

However, our controller does not exhibit the performance advantage in first-in-first-out (FIFO) types of pipelines, which have no logic processing. In that case, its minimum cycle times are given by (3.38), (5.22) and (6.22) derived in Chapter 3, Chapter 5 and Chapter 6 respectively.

$$T^{l}_{EA}|_{min} = 3 \cdot t_{AND\uparrow} + 2 \cdot t_{C\downarrow} + t_{C\uparrow} \tag{7.5}$$

$$T^{l}_{2P}|_{min} = 2 \cdot t_{latch} + t_{XNOR\uparrow} \tag{7.6}$$

$$T^{l}_{4P}|_{min} = 2 \cdot (t_{G1\uparrow} + t_{G2\uparrow}) \tag{7.7}$$

$$+ t_{G1\downarrow} + t_{G2\downarrow} \tag{7.8}$$

$$\tag{7.9}$$

EA controller overhead is exposed in the critical cycle of the controller. Compared with 2-phase controller (7.6) this is clearly larger though it is in the same order of gate delays (6 gates) in comparison to the 4-phase controller (7.8).

The forward latencies of the three controllers can be compared using equations (3.39), (5.21), and (6.21).

$$L^l_{EA} = t_{AND\downarrow} + t_{NOT\uparrow} + t_{flop} + t_{logic}, \tag{7.10}$$

$$L^l_{2P} = t_{latch} + t_{logic}, \tag{7.11}$$

$$L^l_{4P} = t_{flop} + t_{logic} + t_{G1\uparrow} \tag{7.12}$$

The MOUSETRAP controller clearly exhibits the lowest forward latency, where as the 4-phase controller shows slightly lower latency than our controller for equal gate delays. The forward latencies when there is no logic processing are given by equations (3.41), (5.23), and (6.23).

$$\begin{aligned} L^l_{EA}|_{min} &= 2 \cdot t_{AND\uparrow} + t_{AND\downarrow} + t_{C\uparrow} + \\ &\quad t_{C\downarrow} + t_{NOT\uparrow} \end{aligned} \tag{7.13}$$

$$L^l_{2P}|_{min} = t_{latch} \tag{7.14}$$

$$L^l_{4P}|_{min} = 2 \cdot t_{G1\uparrow} + t_{G2\uparrow} \tag{7.15}$$

Again, we can see that the MOUSETRAP controller has the lowest latency and our controller has the highest.

It should be noted that neither our controller nor the MOUSETRAP controller is a Speed-Independent (SI) circuit, whereas the 4-phase controller is. SI circuits work correctly when we use the unbound delay model, for which gate delays are unbounded (yet finite) and wire delays are zero. In our controller (and also in the MOUSETRAP), there are timing constraints that should be held by the gate delays of the selected technology for the design. The performance advantages of both controllers over the 4-phase controller depend partly on this as well.

7.3 Conditional Branch (CB) Controllers

Again, an accurate comparison of cycle times requires specific delay values from technology libraries. However we can employ the same mechanism to compare the overhead of the controllers that we employed in the linear controller comparison. In comparison of cycle times, from equations (4.15), (5.25) and (6.25)

$$T_{EA}^{cb} = t_{flop} + t_{logic} + t_{AND\uparrow} + t_{C\downarrow} - [t_{AND\downarrow}] \tag{7.16}$$

$$\begin{aligned} T_{2P}^{cb} &= t_{flop} + t_{logic} + 2 \cdot t_{AND\uparrow} \\ &\quad + (t_{XNOR\uparrow} - t_{XNOR\downarrow}) \end{aligned} \tag{7.17}$$

$$\begin{aligned} T_{4P}^{cb} &= t_{flop} + t_{logic} + t_{G1\uparrow} + t_{G2\uparrow} + t_{G1\downarrow} \\ &\quad + t_{G2\downarrow} + t_{AND\downarrow} + [t_{AND\downarrow} + t_{OR\uparrow} + t_{OR\downarrow}] \end{aligned} \tag{7.18}$$

it can be observed that 4-phase cycle time has high overhead compared to the 2-phase and our controller. Hence, we put more emphasis on comparing the first two controllers as it shows the advantage of our controller over 2-phase protocol. For EA controller (4.15) and 2-phase controller (5.25), an approximate comparison can be done assuming $t_{AND\uparrow} \approx t_{AND\downarrow}$ and $t_{XNOR\uparrow} \approx t_{XNOR\downarrow}$ which further simplifies the cycle times to as follows.

$$T_{EA}^{cb} = t_{flop} + t_{logic} + t_{C\downarrow} \tag{7.19}$$

$$T_{2P}^{cb} = t_{flop} + t_{logic} + 2 \cdot t_{AND\uparrow} \tag{7.20}$$

A comparison of the controllers' simplified cycle times (7.20) and (7.19) shows that the latter is slightly better provided that

$$t_{C\downarrow} - t_{AND\downarrow} < 2 \cdot t_{AND\uparrow} \tag{7.21}$$

This can hold in the case of many technologies mainly owing to the fact that the right-hand side is two gate delays while the left-hand side is less than one gate delay. This gives a slight performance advantage to the EA-protocol-based controller. In a process technology where gate delays can be chosen such that $t_{AND\uparrow} < t_{AND\downarrow}$ and $t_{XNOR\uparrow} < t_{XNOR\downarrow}$ (for example, using transistor sizing in ASIC technologies), the cycle times in *both* cases can be reduced according to the expressions that we obtained ((4.15) and (5.25)). Again, this give rises to the above condition, which determines the higher performance of the two controllers. As shown in Section 7.5, in the case of our experiments on an FPGA where the gate delays are identical, we observed that the EA controller's cycle time was slightly better.

When there is no logic processing, the cycle times can be compared using equations (4.17), (5.27), and (6.27). With the minimum cycle time, the 2-phase controller exhibits the highest performance, whereas the 4-phase controller shows the slowest performance.

As in the case of the linear controller, this result supports the 2-phase controller as the best candidate for pipelines with very small or no logic processing in between stages.

The forward latencies of the three CB controllers were derived in equations (4.18), (5.26), and (6.26). Given that the gate delays are equal, we have 4-phase CB controller with the lowest latency and 2-phase CB controller with the highest latency. Our controller's latency is better than 2-phase controller's but slightly larger than the latency of 4-phase controller. The minimum latencies obtained for each type of controller (equations (4.20), (5.28) and (6.28)) when there are no logic processing units in between stages shows that our controller has the highest latency compared to other two.

7.4 Performance Test Framework

To prove the concept, we evaluated the performance of each of the controllers on a Xilinx Vertex-4 (XC4VFX100) FPGA. We made maximum efforts to minimize the uncertain path delays in FPGA routing. All control and data path circuits of the designs were placed identically in each case using the *rloc* placement constraints of the Xilinx ISE tool. Synthesis options, both general and Xilinx-specific ones were tuned to suit asynchronous design synthesis; for example, the use of global and regional clock buffers was disabled. Thus, we believe that the results we obtained are comparable with each other with minimum uncertainty in the measurements.

For the linear controllers, we created simple 8-bit 4-stage FIFOs operated by controllers of each type. For the CB controllers, we built 8-bit Y-shaped pipelines with 4-stages where two stages were in the stem and two stages were branched out. The CB controller was placed in the second stage of the pipeline. Data width of all pipelines was 8-bit.

Environment of the pipelines comprised input-generating shift registers and output-capturing registers (two registers in the CB case) were operating with minimum overhead, which maximized the performance of the controller under test.

Performance of controllers were evaluated in two cases

1. Pipelines operating without any processing
2. Pipelines operating with processing between stages

In the first case, there was minimum delay between stages without any logic processing in between which represents the maximum performance of the controllers for high-speed pipelines. Since there was no logic processing ($t_{logic} = 0$), no matched delays were inserted between stages either ($t_{MD} = 0$).

In the second case, we tested the performance of pipeline controllers for a general scenario of pipelines operating with processing in between stages. To emulate the processing elements, we used simple buffers to delay the data path. The introduced logic delay ranged from 6.9ns to 7.2ns (varied depending on the exact routing of the data path) for each stage of the pipeline. This delay was chosen such that it satisfied the condition (4.14) (which in turn satisfies condition (3.36)) that we derived for CB controller for the EA protocol. Thus, we could obtain the performance of the EA protocol (and other protocols) in the case of a general pipeline with logic processing, for which these two conditions can be easily satisfied.

The delay elements for controllers were tuned starting from a higher delay to the lowest possible delay for which proper operation of the pipeline was guaranteed. Even when the logic delay was measured accurately, it was not possible to get the exact delay value. The tracking error, i.e., the difference between the actual and required delays, should be always kept positive to correctly operate a design. This error is additive to the cycle time of each design and is common to all designs of the three protocols. In an ASIC design, adjustable delays are desired so that the delay can be set properly after the design has been fabricated.

TABLE 7.1: Cycle-time comparison.

Cycle Time	Without processing (ns)	With processing (ns)
Linear		
2-phase (MOUSETRAP)	2.6	9.9
4-phase	3.6	12.4
EA	4.0	10.0
CB		
2-phase	4.0	11.2
4-phase	7.1	15.1
EA	5.6	10.6

7.5 Results

Post-layout simulation results for Vertex-4 obtained using ModelSim are shown in Table 7.1. The first column of results shows, that the 2-phase controllers outperformed the 4-phase and EA controllers in linear and CB operations when there was no processing in between pipeline stages ($t_{logic} = 0$). Its performance advantage was evident in these cases where minimum controller overhead is desirable. Since $t_{logic} = 0$, condition (3.36) for the EA controller does not hold, so the overhead of the controller is exposed on the critical cycle time which explains its larger cycle time.

The second column shows that, in the cases where logic processing is present between pipeline stages, the EA controllers performed better as their overhead got hidden in the required delay between stages. For EA controllers, condition (3.36) holds in this case, so their performances are comparable to that of the 2-phase controller in linear operation, which confirms the analytical cycle times that we obtained in (3.37) and (5.20).

The last thee rows of the second column show, that the CB controller for EA protocol outperformed the 4-phase controller and performed slightly better than the 2-phase controller. As we demonstrated in our analysis, the ability of the EA protocol to hide the control overhead leads to this performance gain. In our FPGA implementation, all gates (including the C-elements) are implemented using lookup tables (LUTs) that have identical delays, which simplifies the comparison of controller cycle times. Given the equal delays in gates, the difference in cycle times of CB controllers derived in (4.15) and (5.25) amounts to a 2-gate delay, which is roughly between 600 ps and 1100 ps [30] in the Vertex-4 architecture. Hence, the results (600 ps difference) agree with our formal analysis, subject to routing delay variations. Even though the gain is relatively small, the simplicity of the EA protocol as a 4-phase protocol makes it more appealing in this case, which is a non linear asynchronous pipeline application. As described earlier, when the 2-phase protocol is used, translations from 2-phase protocol to 4-phase protocol are usually required at some points where level-sensitive control is necessary. In such cases, our controller has the added advantage using a variation of the 4-phase protocol and of having a performance gain over the 2-phase protocol by hiding the additional controller overhead incurred by non linear operations.

To evaluate the area consumption of our controller, we measured the resource utilization of the FPGA for our designs. The resource utilization of the control path for controllers and matched delays in terms of FPGA slices used is shown in Table 7.2. In the Vertex-4 architecture that we used, one slice comprises two lookup tables and two flipflops and/or latch units.

Table 7.2: Resource utilization comparison.

# slices	Without processing		With processing	
	ctrl.	delay	ctrl.	delay
Linear				
2-phase	4	0	4	20
4-phase	4	0	4	23
EA	12	0	12	14
CB				
2-phase	8	0	8	35
4-phase	8	0	8	33
EA	20	0	20	22

The second column of Table 7.2 shows the number of slices used by controllers without processing. In the linear 4-stage FIFO, both 2-phase and 4-phase controllers have the same resource utilization; 4 slices. EA controllers, which consume the most resources, utilize 12 slices in total for their logic and for the self-resetting delay elements inside. Since we did not use any matched delays in this case, the third column which shows the delay element utilization is always zero. Fourth and fifth columns show the cases where pipelines are operated with logic processing. Since the difference is only the logic processing between stages, the number of resources used by the controllers are same in each case (second and fourth columns). According to the fifth column which gives the matched delay element resource usage in this case, it can be noted that delay elements of EA controllers are smaller consuming 14 slices compared to delay elements of 2-phase (20 slices) and 4-phase (23 slices) controllers. The reason for this is that EA controllers require a *smaller* matched delay $t_{MD\uparrow}$ for a logic processing stage given the same t_{logic} compared with the other two protocols. Thus the total resource usage for control path (controllers and delay elements) for EA controllers (26 slices) is comparable to that of the 2- and 4-phase controllers (24 and 27 slices, respectively).

The same reasoning applies to the case of CB controllers. Even though EA controllers themselves consume higher resources, the total resources consumed in the case of a pipeline with processing are comparable in all thee cases. Hence, we could obtain the performance gains described earlier with more or less the same resource utilization for the control path, which highlights the advantages of using EA protocol.

7.6 Conclusions

We proposed a new pipeline controller for the Early Acknowledgement (EA) protocol. Its timing constraints were analyzed and performance metrics were derived. When the pipeline has logic processing, the controller can operate with minimal overhead by hiding its overhead in the required matched delay. In such a case, we found both analytically and experimentally that the controller's cycle time was comparable to that of 2-phase controller MOUSETRAP.

Furthermore, we highlighted the advantages of using the EA protocol, which also inherits the simplicity of the 4-phase protocol, by comparing the conditional branch controllers for each protocol. The area usage of the protocol is also comparable to those of other protocols in the preferred application of this protocol since the required matched delay is smaller, so requiring less area is required for the design.

Chapter 8

Case Study: Gauss-Seidel Accelerator

8.1 Introduction

An accelerator module is designed to validate the performance of the Early Acknowledgement controllers, both linear and Conditional Branch types. The function of the accelerator is to solve systems of linear equations using Gauss-Seidel method [31, 32]. Accelerator consists of 6 pipeline stages mainly consisting of multipliers, adders and read/write operations for accessing the variables from the memory.

The accelerator is designed for each protocol, i.e. i). 2-phase, ii). 4-phase and iii). Early Acknowledgement protocol using the de-synchronization methodology as detailed in [24]. The synchronous version of the accelerator is first implemented and then de-synchronized using local controllers for each pipeline stage replacing the clock signal. Thus, all accelerator designs have same data path while the control path is replaced by different protocols. This gives an accurate comparison of the performance of the each protocol/controller pair.

8.2 Gauss-Seidel Method

The Gauss-Seidel method is a technique used to solve a linear system of equations using an iterative algorithm. It is an improvement of Jacobi method [32] to achieve quick convergence using the previously computed results as soon as they are available. The a linear system of equations can be expressed as:

$$Ax = b. \tag{8.1}$$

where $\boldsymbol{x}= (x_0, x_1, \cdots, x_{n-1})$ is a vector of n variables, $\boldsymbol{b}= (b_0, b_1, \cdots, b_{n-1})$ is a constant vector, and $\boldsymbol{A}$ is n-by-n coefficient matrix with non-zero diagonals. If matrix $\boldsymbol{A}$ either diagonally dominant, that is, $|a_{ii}| > \sum_{j \neq i} a_{ij}$ for every i, or symmetric and semi-positive definite, the solution for vector $\boldsymbol{x}$ can be obtained using Gauss-Seidel method as follows:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^k), \quad i = 1, 2, \ldots, n\,. \tag{8.2}$$

k is the iteration of the algorithm. x_i, x_j, b_i and a_{ij} denotes the elements of $\boldsymbol{x}$, $\boldsymbol{b}$ vectors and $\boldsymbol{A}$ matrix of i^{th} row and j^{th} column in the usual vector notation. The algorithm computes $x_i^{(k+1)}$, the value of x_i in the $(k+1)$-th iteration, using the already computed values of the same iteration ($x_j^{(k+1)}$ for $j < i$) and the previous iteration ($x_j^{(k)}$ for $j > i$) according to the equation (8.2). Thus, each x_i needs to be computed sequentially unlike in the Jacobi method where it can be done simultaneously. This enable the use of a single physical location to store the vector x in memory, overwriting the old value of iteration (k) with the new value of iteration $(k + 1)$.

8.3 Algorithm

The iterative algorithm based on Gauss-Seidel method can be expressed as in the algorithm 1:

Algorithm 1: The Gauss-Seidel (GS) iterative algorithm

```
Data: Matrix $\boldsymbol{A}$, vector $\boldsymbol{b}$ and initial solution $\boldsymbol{x^0}$
Result: Solution $\boldsymbol{x}$
1  $n \longleftarrow sizeof(b)$ ;
2  foreach iteration $k$ $(= 1, 2, \ldots)$ do
3      foreach row $i$ do
4          $sum \longleftarrow b[i]$ ;
5          for $j$=0 to $i$-1 do $sum \longleftarrow sum - A[i][j] \cdot x^{k+1}[j]$ ;
6          for $j$=$i$+1 to $n$ do $sum \longleftarrow sum - A[i][j] \cdot x^k[j]$ ;
7          $x^{k+1}[i] \longleftarrow sum \;/\; A[i][i]$ ;
8      end
9      if solution $x^{k+1}$ converged to $x^k$ then
10         return solution $x^(k+1)$ ;
11     end
12     $x^k \longleftarrow x^{k+1}$ ;
13 end
```

In the algorithm, accepts the system of equation $\boldsymbol{A}$, $\boldsymbol{b}$ and an initial solution $\boldsymbol{x^0}$ (it can be $\boldsymbol{x^0} = \boldsymbol{0}$). Then for each iteration k and each row i the each term of x_i is calculated in lines 4, 5, 6 and 7 of the algorithm. The final division operation can be optimized out by properly conditioning the A matrix as explained in Section 8.4.3. At the end of each iteration solution sets $x^{(k+1)}$ and $x^(k)$ are compared to test convergence. If the difference in the solution is smaller than some threshold value for every i the iteration stops and the solution is returned.

Even though two or more x_i terms cannot be computed simultaneously, the computations within each x_i term can be carried out in parallel. If the elements of $\boldsymbol{x}$, $\boldsymbol{b}$, and $\boldsymbol{A}$ are stored in m ROMs for $\boldsymbol{A}$ and $2m$ RAMs for $\boldsymbol{b}$ and $\boldsymbol{x}$ such that m elements can be read in parallel the computation time is reduced by factor of m per iteration. The typical use of this accelerator is to calculate x for a fixed system at various points. The system characteristics are captured in coefficient matrix A which will be pre-compiled and stored in the ROMs. At each point in time and/or scenario the observations of the system are captured as b. The solution for x the accelerator calculates corresponds to the particular instance of observations. This is a typical usage of the accelerator in a finite- element analysis system. Figure 8.1 shows the architecture of such a pipeline with $m = 4$.

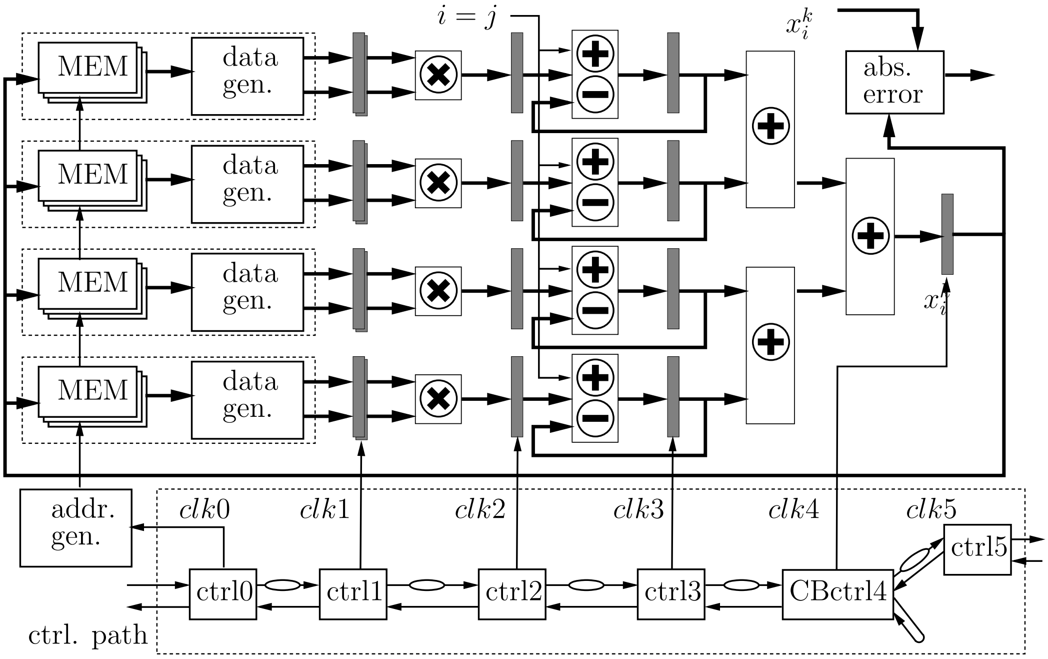


FIGURE 8.1: Gauss Seidel Pipeline.

Following are the stages of the 6-stage pipeline architecture.

1. **Data generation stage** where at the start of each computation cycle eight data elements are read from the RAM/ROM blocks.

2. **Product calculation stage** where four product terms of the equation is computed in parallel from using the above read data.

3. **Product accumulation stage** either adds or subtract the computed terms from a accumulation register which is reset the beginning of the computation cycle.

4. **Reduction stage** employees a reduction tree of adders to compute the final x_i^{k+1} term using the partial summations of the accumulation registers.

5. **Write back stage** write the computed x_i^{k+1} value back to the RAM so that it can be immediately used by the next computation cycle. The old value x_i^k is simultaneously read back to compute the error in the next stage

6. **Error calculation stage** computes the absolute errors between x_i^k and x_i^{k+1} terms. If the result is sufficiently converged this stage raise *finish* signal to indicate completion of the computation block.

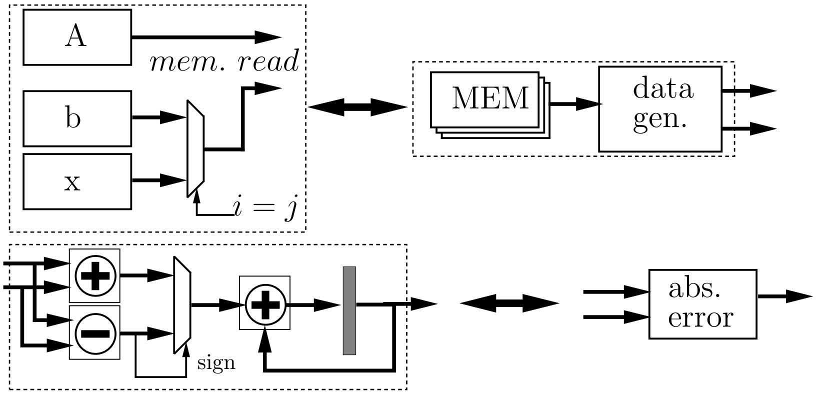


Figure 8.2: Gauss Seidel Pipeline (units).

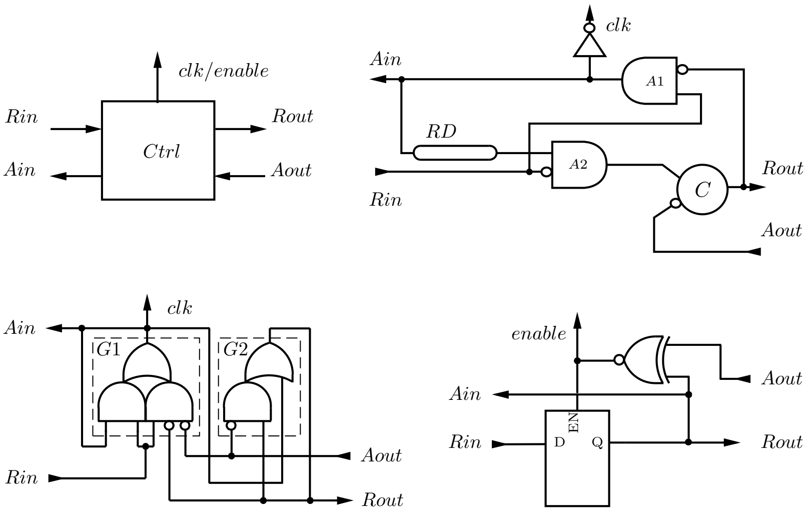


Figure 8.3: Linear Controllers for each Protocol.

Three different versions of the Gauss-Seidel accelerator were built using each protocol to compare their performances. As shown in Figure 8.3, the linear controllers of GS accelerator (*ctrl0, ctrl1, ctrl2, ctrl3* and *ctrl5* of Figure 8.1, were implemented with respective linear controllers of the each protocol. Similarly the CB controller (*ctrl4* of

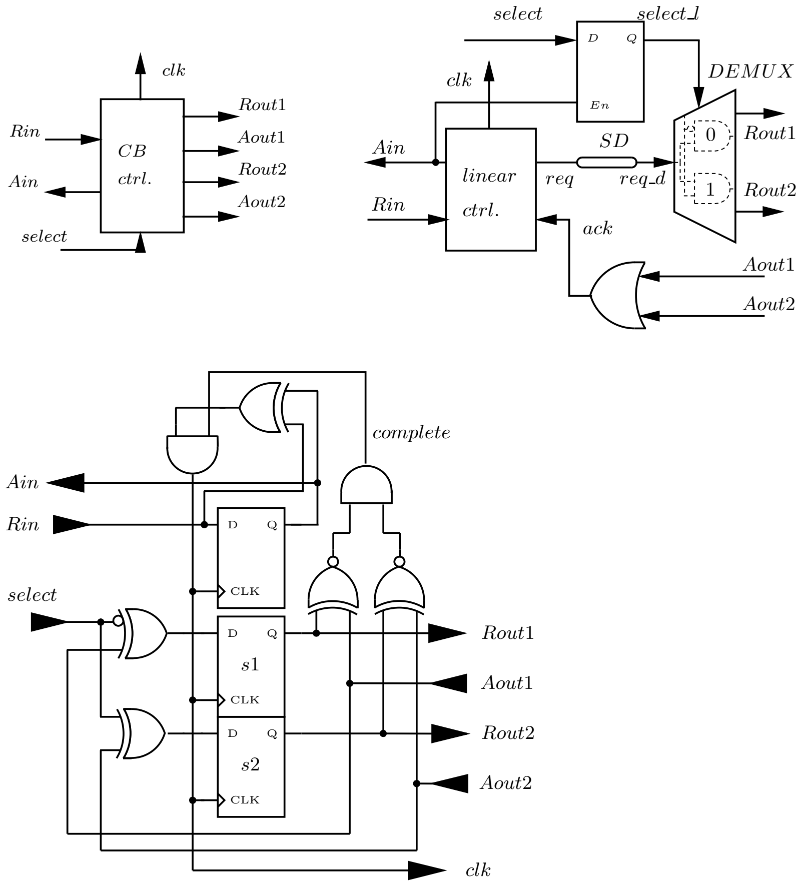


FIGURE 8.4: Conditional Branch Controllers for each Protocol.

Figure 8.1) is replaced with the respective CB controller of the protocol as shown in Figure 8.4.

8.4 Pipelined Implementation

8.4.1 Data Path

In our design, $n = 32$, i.e. the accelerator solves a linear system of 32-variables. It is a design choice we made which can be easily extensible for any $n = 2^p$ number of variable systems. The width of the data A, b and x i.e. input and output for the accelerator is set to be 16-bit. These results in 16x16 multipliers and 16x256 RAM/ROM units for

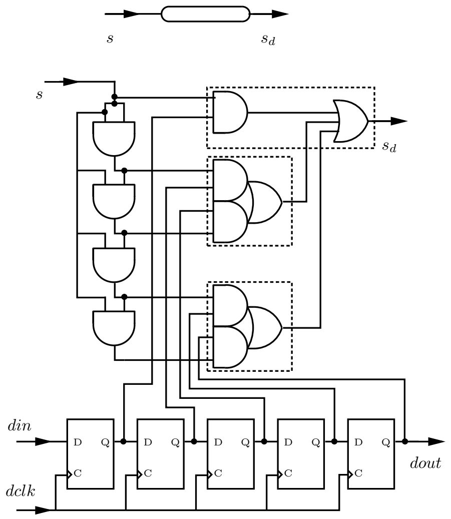


Figure 8.5: Asymmetric (Rising-edge) Programmable Delay.

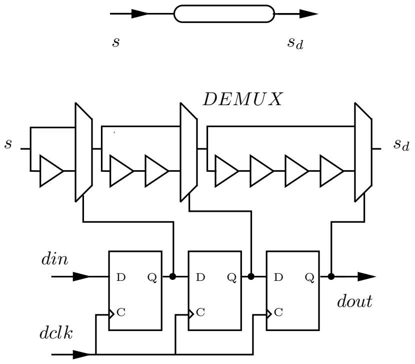


Figure 8.6: Symmetric Programmable Delay.

logic processing stages which can be directly mapped to the DSP48 blocks and block RAM macros available on Virtex-4 FPGAs [33].

As shown in Figure 8.1, the main computational block has 4 parallel pipeline stages ($m = 4$). Thus, the memory of the accelerator which holds the elements A, b and x are split into 4 blocks to generate data simultaneously for the computations. There are 4-multipliers and 4- adder/subtracter units and a 4-to-1 reduction tree of adders to produce the $x_i^{(k+1)}$ term as shown. The accelerator consists of 6-pipeline stages. The gray rectangles denote the stage registers.

The first stage consists of address generator, memory and data generators. When the next set of addresses is provided by the address counter this stage drive the data generators to read data from memory for the next computation. As shown in Figure 8.2, in each block of memory A elements are stored in a ROM and elements of b and x are stored in RAM. The initial solution x^0(if any), should be loaded to RAM before executing the accelerator. At every iteration the $x^{(k+1)}$ replaces the $x^{(k)}$ in the same RAM blocks. A dual-port RAM is used for this purpose to avoid some extra addressing logic which would require otherwise. Note that both b and x are stored in the same RAM block as the addressing of their elements is mutually exclusive (i.e. there is no $b_i x_j$ term involve in the calculations). The memory is realized using FPGA block RAM IP core modules with necessary write-before-read property set. The data generator modules consists of address logic: mainly multiplexors and a common address counter (a simple up counter to generate addresses) shared by all generators.

The second stage is the bank of multipliers which calculate the product terms in parallel. Adder/subtracter units accumulate the product terms depending on the sign of the calculation in the third stage registers. Reduction tree of adders constitute the fourth stage which produce the $x_i^{(k+1)}$. Since $n = 32$ and $m = 4$ the accelerator execute ($n/m =$)8-times over the previous stages to complete the calculation for each x_i per iteration. RTL descriptions according to the design guidelines are used to realize these blocks such that multipliers are mapped to DSP48 blocks and adder/subtracter units are mapped to FPGA macro blocks.

Once, the $x_i^{(k+1)}$ is computed the value calculated it is written back to the corresponding RAM holding x overwriting the previous value. The old value $x_i(k)$ is simultaneously read back to absolute error calculator to compute the error. To ensure the consistency of this operation the read-before-write property of the RAM is ensured. The convergence of the algorithm is computed using the absolute error calculator unit. It consists of adder/subtracter pair to calculate this function.

$$e = \sum |x_i^{(k+1)}| - |x_i^{(k)}|. \tag{8.3}$$

Figure 8.2 shows the implementation of this unit comprising of adder/subtracter unit and an accumulator. This error is computed as a running total within each iteration so that the final error e can be tested against the threshold value at the end of each iteration and reset back to zero.

8.4.2 Control Path

The control path consists of 6 controllers to drive each stage of the pipeline as shown in Figure 8.1. First 4 stages and the last stage of the pipeline consists of linear controllers where as fifth stage employs a conditional branch controller. The *select* signal for the Conditional Branch controller is the boolean predicator which test whether the number of iterations are less than 8 ($iter < 8$). Thus, control flow drives the data path up to that stage 8-times before the absolute error is tested.

Control logic including the C-elements for the Early Acknowledgement controllers are implemented using LUTs and FF/LT resources of FPGA. Hierarchical design and placement using *rloc* placement constraint allows fine control over the placement of controllers in the FPGA. Three accelerators are implemented employing each of the 3 protocols (2-phase, 4-phase and Early Acknowledgement protocols) for comparison of performance. In each case the data path remains identical. The relative placement of the control paths is also comparable.

8.4.3 Input and Output

Gauss-Seidel method requires matrix A to be either diagonally dominant, that is, $|a_{ii}| > \sum_{j \neq i} a_{ij}$ for every i, or symmetric and semi-positive definite for guaranteed convergence. This precondition must be satisfied for using the accelerator which would not explicitly check it.

Moreover, there matrix A provided for the accelerator is pre-conditioned to eliminate the division operation of step 7 of the algorithm. This can be achieved by defining a new matrix A' for the computations in the accelerator, such that:

$$A' = (\frac{-a_{i0}}{a_{ii}}, \cdots, \frac{a_{i(i-1)}}{a_{ii}}, \frac{1}{a_{ii}}, \frac{a_{i(i+1)}}{a_{ii}}, \cdots \frac{a_{i(n-1)}}{a_{ii}})^T \tag{8.4}$$

Thus, equation 8.2, deduce to:

$$x_i^{(k+1)} = (a'_{ii} b_i - \sum_{j<i} a'_{ij} x_j^{(k+1)} - \sum_{j>i} a'_{ij} x_j^k), \quad i = 1, 2, \ldots, n\,. \tag{8.5}$$

which only consists of multiplication operations. Accelerator is fed with this A' matrix instead of A as a result of this optimization.

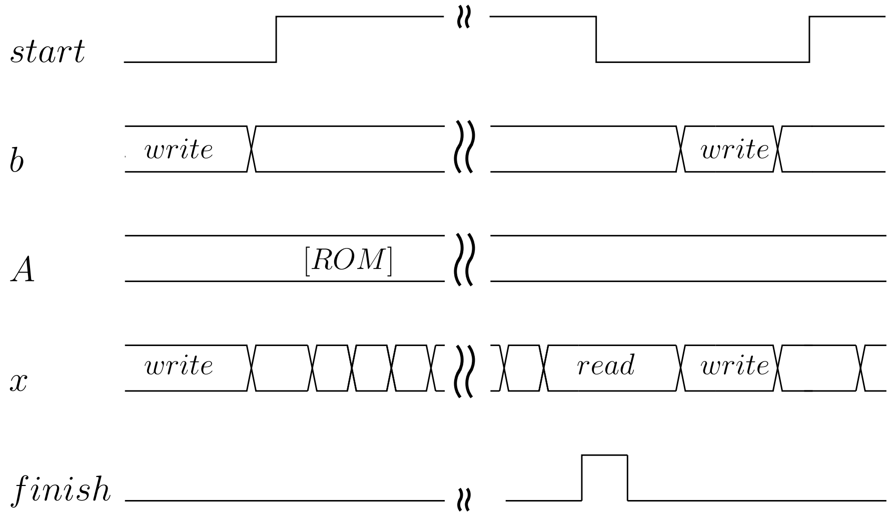


FIGURE 8.7: Operation of Gauss Seidel Accelerator.

The Figure 8.7 shows the top-level operational waveforms for using the accelerator. The data in A' matrix are pre-computed and loaded to the ROMs this values set won't change during the computation. The accelerator solves the equation system using Gauss-Seidel method for different sets of observations (b) and initial solutions (x^0). At the beginning of each computation, RAMs are loaded with b and x^0 using the CPU interface of the accelerator. Once completed, *start* signal is raised to initiate the execution of the accelerator. During the iterative solving phase, the RAM blocks of x is overwritten by the successive solutions computed by the accelerator. Once the solution converged, the $finish$ signal is raised by the accelerator. The solution x can be read out using the CPU interface. A new set of b and x^0 can be loaded afterwards to for another run of the accelerator.

The CPU interface of the accelerator, which is used to assign the problem to the accelerator and read back the solution, is not implemented for this case study. This does not affect the performance comparison of the three protocols. Instead, the inputs (A', b and x^0) are generated using a C program and directly written to memory locations of the final design to be simulated. At the end of the execution of the accelerator the output (stored in x RAMs) are compared with the solution generated by the C program.

8.4.4 Fixed-point arithmetic and normalization

The accelerator employs fixed point arithmetic. Hence a diagonally dominant matrix of A produces a factional numbers for all a'_{ij} elements as of equation (8.4) making significant bits to underflow. Thus the A' matrix is normalized by left-shifting elements to avoid underflow by 16-bits (i.e. multiplied by 2^{16}). When $x_i^{(k+1)}$ the computed as a 32-bit value, the result is de-normalized by discarding the upper 16-bits. The convergence test becomes a simple due to the use of fixed-point arithmetic. When every element of $x_i^{(k+1)}$ is equal to previous iteration x_i^k, the convergence achieved and accelerator terminates. This corresponds to the test of $e = 0$ of equation (8.3).

8.5 FPGA Design

As a proof of concept of the performance of the protocols, the implementation of the accelerators for this case study is carried out on Xilinx Virtex-4 FPGA architecture. Xilinx ISE and ModelSim simulator is used during the design, implementation and evaluation cycle of the accelerators. The designs are implemented with Verilog HDL. The Figure 8.8 shows the architecture of the Virtex-4 FPGA slice.

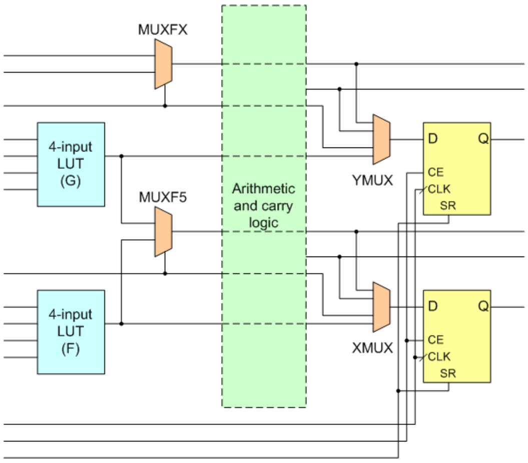


Figure 8.8: Xilinx Virtex 4 Slice Configuration.

It mainly consists of 2 Look-Up-Tables (LUTs) with maximum of 4 inputs and 2 registers that can either be used as Flip-flops or Latches (FF/LT). As a general design guideline, both LUTs and FF/LT units of the slice are used whenever possible. However, at instances where there is a better routing possibility with a shorter delays the design is spread to adjacent resources if available.

Necessary steps to minimize the uncertain path delays are taken during the design process. First, the de-synchronization methodology used ensures that the data path remains identical in each case of the accelerator. The design is done hierarchically and all memory DSP48 units and logic slices which comprises the control and data paths are placed identically in each case using *rloc* placement constraints of the Xilinx ISE tool. Synthesis options, both general and Xilinx specific ones are tuned to suit the asynchronous design synthesis. For example, use of global and regional clock buffers is disabled. Thus, the simulation results obtained from this are made comparable with each other with minimum of uncertainty in measurements.

The Figure 8.9 shows the floor plan of the designs on Virtex-4 FPGA. The resource usage of the data path is shown in Table 8.1

Table 8.1: Data path Resource Utilization.

Block(s)	# Resource	Units
A ROMs	4	MEM
b and x RAMs (dual port)	4	MEM
Multipliers	4	DSP48
Adder-subtracters	128	Slices
Adder (Reduction) tree	96	Slices
Address generator	5	Slices
Address logic	26	Slices
Error calculation logic	32	Slices
De-normalizer logic	9	Slices

As for the control path, the placement of controllers remains at the same relative location as indicated in the Figure 8.9 though the exact usage of the slices for controllers and matched delays varies according to the different protocols. In each case of the accelerator with different protocols, the designs are tuned to have optimal operational performance by tuning the matched delays. The tuning is started from a higher delay value and gradually reduced to the lowest possible value where the proper operation of the pipeline is guaranteed. At each step the design is simulated with a batch of random input sets generated by the C program and the generated output is tested against the results. Also,

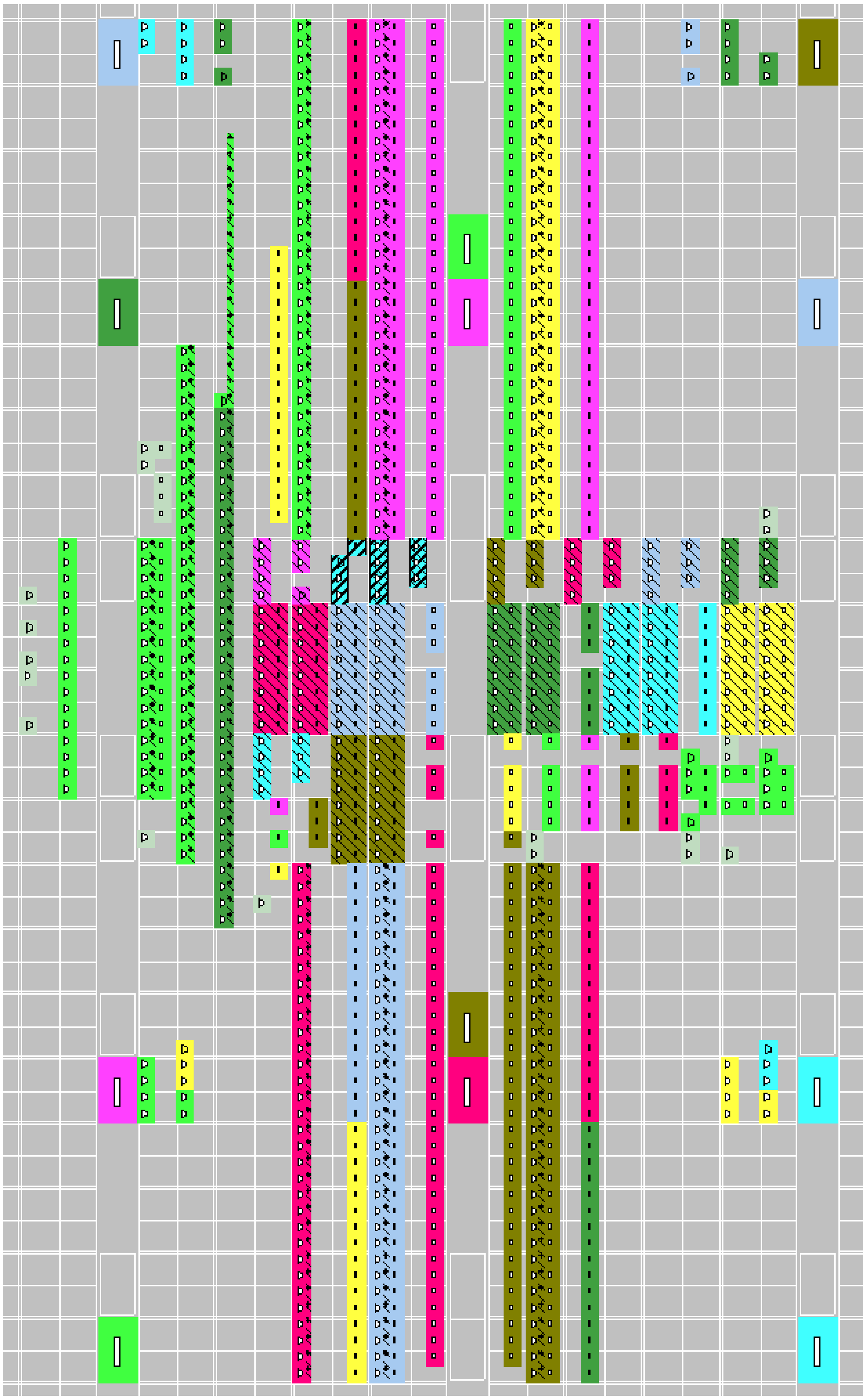


Figure 8.9: Floor Plan on FPGA Virtex-4.

the simulation is checked against possible timing violations which could invalidate the design generated for the given input data sets.

8.6 Simulation Results

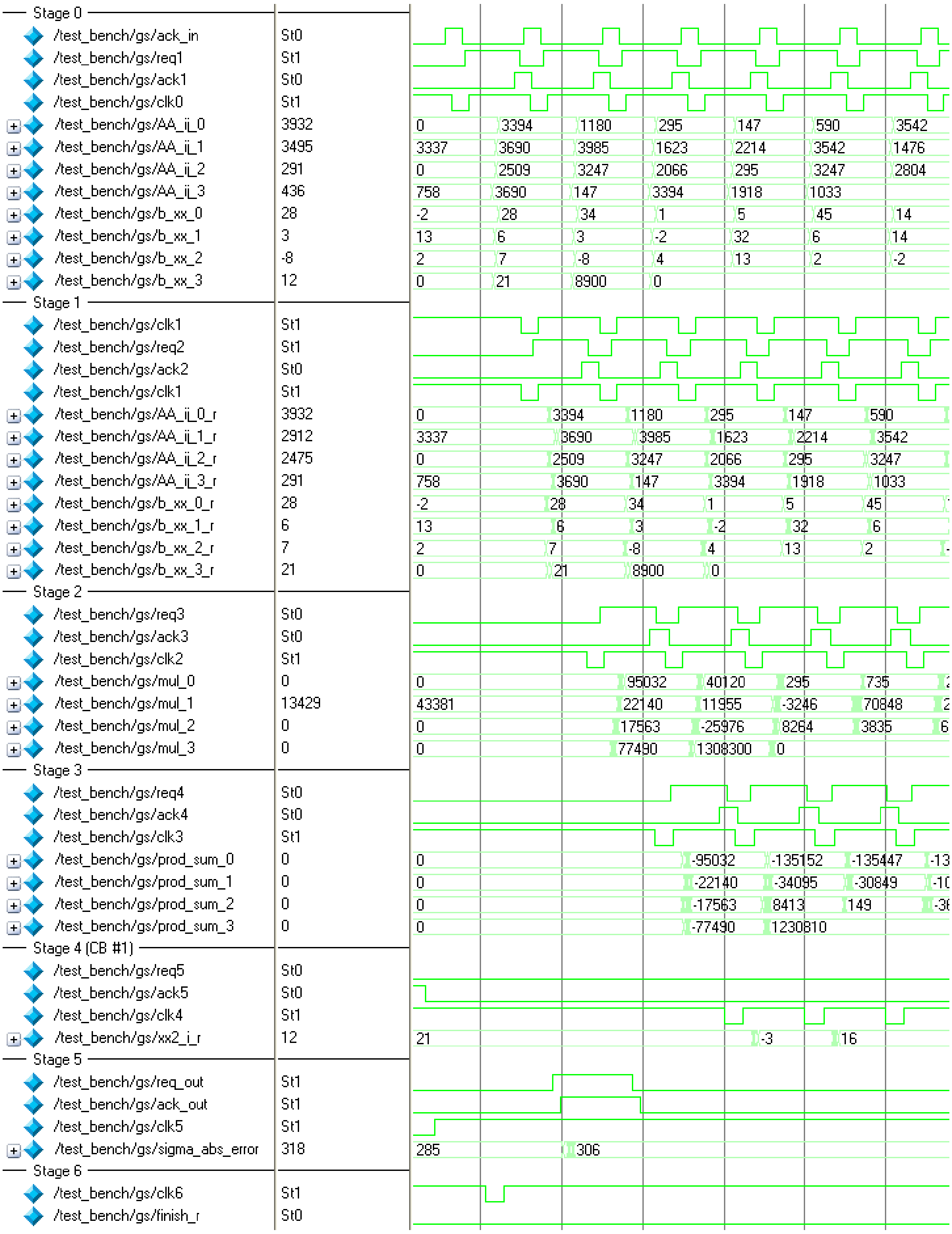
Simulation results for place-and-route(PAR) designs for the Virtex-4 architecture are obtained using the ModelSim simulator. A waveform capture of a typical simulation cycle is shown in Figure 8.10. Note, only few important signals of the design are shown here.

The simulation results obtained for each of the accelerator are summarized in in Table 8.2. The simulation results show the accelerator solving the linear system of equation for a same set of A, b and x^0. Hence, the iteration count in all cases is the same. For this particular set of data the accelerator iterates 6 times (i.e. $k = 6$) before converging in to the solution. The first row shows accelerator time taken to calculate $x_i^{(k+1)}$ per single element in nano seconds(ns). The second column indicates the time taken to complete an iteration (i.e k=1,2,...) also in ns. The last column shows the complete time taken in micro seconds (us), from $start$ signal is raised to $finish$ is raised by the accelerator to indicate the completion of the task.

Table 8.2: Accelerator Performance Comparison.

Protocol	simulation time		
	$/x_i^{(k+1)}$ (ns)	/iteration (ns)	total (us)
4-phase	122.6	3922.9	23.67
2-phase	118.7	3799.4	22.93
Earl Acknowledgement	107.7	3447.8	20.81

It can be observed that the accelerator employing the EA protocol has better performance than the other two. The computation times taken for each x_i^{k+1} element, and each iteration and the total time are less than the consumed time for 2-phase and 4-phase protocols. In this accelerator, the gap between 4-phase and 2-phase is also observed to be narrower than in the results obtained in previous chapter. This is caused by two reasons:



Stage 0	
/test_bench/gs/ack_in	St0
/test_bench/gs/req1	St1
/test_bench/gs/ack1	St0
/test_bench/gs/clk0	St1
/test_bench/gs/AA_ij_0	3932
/test_bench/gs/AA_ij_1	3495
/test_bench/gs/AA_ij_2	291
/test_bench/gs/AA_ij_3	436
/test_bench/gs/b_xx_0	28
/test_bench/gs/b_xx_1	3
/test_bench/gs/b_xx_2	-8
/test_bench/gs/b_xx_3	12
Stage 1	
/test_bench/gs/clk1	St1
/test_bench/gs/req2	St1
/test_bench/gs/ack2	St0
/test_bench/gs/clk1	St1
/test_bench/gs/AA_ij_0_r	3932
/test_bench/gs/AA_ij_1_r	2912
/test_bench/gs/AA_ij_2_r	2475
/test_bench/gs/AA_ij_3_r	291
/test_bench/gs/b_xx_0_r	28
/test_bench/gs/b_xx_1_r	6
/test_bench/gs/b_xx_2_r	7
/test_bench/gs/b_xx_3_r	21
Stage 2	
/test_bench/gs/req3	St0
/test_bench/gs/ack3	St0
/test_bench/gs/clk2	St1
/test_bench/gs/mul_0	0
/test_bench/gs/mul_1	13429
/test_bench/gs/mul_2	0
/test_bench/gs/mul_3	0
Stage 3	
/test_bench/gs/req4	St0
/test_bench/gs/ack4	St0
/test_bench/gs/clk3	St1
/test_bench/gs/prod_sum_0	0
/test_bench/gs/prod_sum_1	0
/test_bench/gs/prod_sum_2	0
/test_bench/gs/prod_sum_3	0
Stage 4 (CB #1)	
/test_bench/gs/req5	St0
/test_bench/gs/ack5	St0
/test_bench/gs/clk4	St1
/test_bench/gs/xx2_i_r	12
Stage 5	
/test_bench/gs/req_out	St1
/test_bench/gs/ack_out	St1
/test_bench/gs/clk5	St1
/test_bench/gs/sigma_abs_error	318
Stage 6	
/test_bench/gs/clk6	St1
/test_bench/gs/finish_r	St0

FIGURE 8.10: A Typical Simulation Output.

- The accelerator uses flip-flops instead of latches. The level-sensitive latches could not be used in this accelerator especially at the accumulator unit where output is feedback to as one input. Use of latches is prone to errors in the accumulator output with multiple feedback cycles.
- Use of extra edge-to-pulse control logic in the controller path. The control path of pipeline restart requires less logic and identical for 4-phase and EA protocols. For 2-phase it requires edge-to-pulse transformations incurring additional overhead.

The matched delay inserted in between each stage is given in Table 8.3. Note that the all the delays required for the EA controller based accelerator are less than both 2-phase and 4-phase cases.

Table 8.3: Accelerator Matched Delays.

pipeline stage		matched depaly /(ns)		
#	Fuction	Early Ack.	4-phase	2-phase
1	Data gen.	1.0	3.02	3.74
2	Multiplier	4.65	6.79	5.38
3	Add/Sub	1.07	3.62	3.91
4	Reduction	4.88	7.64	8.93
5	Store $x_i^{(k+1)}$	0.87	1.58	1.92
6	Abs. Error	4.96	7.27	8.68

The same reasoning that we used in Section 7.5 of Chapter 7 can be used to explain the above table timing. The EA controllers exhibit the highest controller overhead and able to offset some of the logic processing delay needed to be matched by the delays inserted in between stages. Hence, additional required to match the stage logic processing is smaller according to the following equation.

$$t_{logic} = t_{overhead} + t_{MD} \tag{8.6}$$

The Figures 8.11, 8.12 and 8.13 shows the control paths of the each protocol (EA, 2-phase and 4-phase protocols respectively) on the place-and-routed (PAR) design of the Virtex4 FPGA.

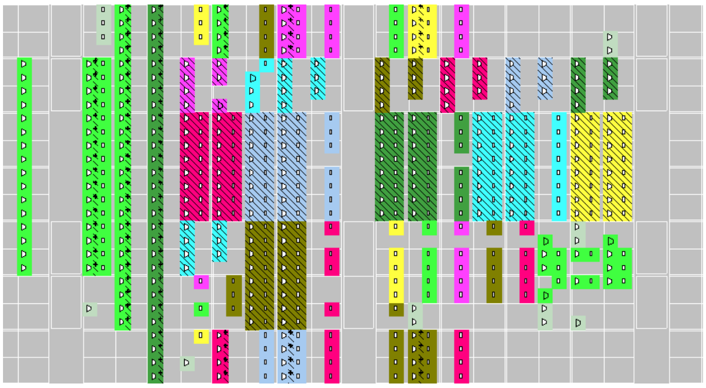


FIGURE 8.11: Control Path: With Early Acknowledgement controllers.

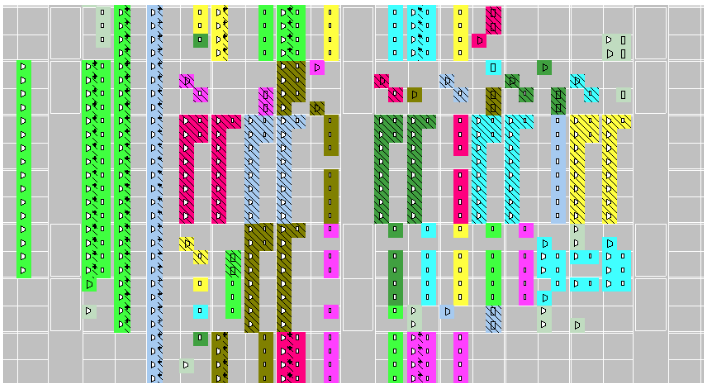


FIGURE 8.12: Control Path: With 2-phase controllers.

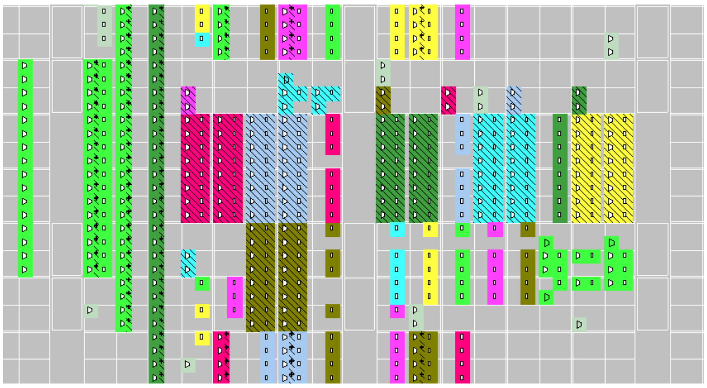


FIGURE 8.13: Control Path: With 4-phase controllers.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

Before this work there were considerable research work done on the bundled data communication model of asynchronous design mainly targeting the use of 2-phase and 4-phase protocols. The Early Acknowledgement protocol was initially introduced in [18], as a zero-time overhead protocol. Lack of an implementation of the protocol into a controller that exploit its advantages and a formal comparison of its performance to 2-phase and 4-phase controllers, made it difficult to appreciate the superiority of the Early Acknowledgement protocol.

Initial attempt to implement a Early Acknowledgement protocol based controller is attempted in [25]. In the evaluation, it turned out that controller is too rigid ensuring the control sequencing that its overhead is too high even though the controller is robust. Thus, the initial EA controller compared to 4-phase and 2-phase controllers did not deliver the performance attributed by the protocol.

A new controller is proposed based on the previous version of the controller which removes some of the overhead of the controller, in lieu of timing constraints introduced to operate it correctly. The new controller is validated to satisfy these constraints and expressions for the timing constraints are analytically derived. It has been compared with 4-phase and 2-phase controllers analytically and experimentally. The performance of the linear controller is comparable to high-speed MOUSTRAP 2-phase controller. Furthermore, the conditional branch controller derived based on this EA linear controller has better performance than their 4-phase and 2-phase counterparts.

To substantiate the claims of EA controllers in a more practical application, an accelerator for solving set of linear equations using Gauss-Seidel method was build. Three instances of the accelerators were built each using 2-phase 4-phase and EA protocols. The performance results highlight the advantages of using EA protocol and the proposed controllers for it.

In conclusion, this work as highlighted the importance of little known EA protocol by proposing a controller for it to harness its advantages. This work serves the main source of any analytical and practical comparison of these protocols. The results of the work will strengthen the importance of EA protocol and encourage the use of it in applications where it exhibit to work efficiently.

9.2 Recommendations for the applications

According to our observations in the comparison of the controllers in Chapter 7, EA controller has the advantage when there is sufficiently enough logic processing between stages. In the analytical evaluation of controllers it could be shown that the difference in the cycle-time is in the order of 1–2 gate delays for both linear controller and conditional branch controller.

In the case study performed for the controllers we could show that in a accelerator application like Gauss-Seidel solver where *the pipeline operation is iterative and performed in large numbers* the cumulative advantage gained by using the EA controllers becomes significant.

On the other hand, with respect to the power consumption EA protocol efficiency is lower than that of 2-phase protocol since EA is essentially a 4-phase protocol. Thus we have following recommendations in the application of the proposed controllers.

1. Use in pipelines where there is sufficiently large logic processing units (as a rule of thumb at least 10 logic gates deep) like multipliers, adders or memory accessing logic in between stages.

2. Use EA controllers in pipelines with iterative computation to amplify the gain of the advantage in processing the data as a whole.

3. Use them when there is more emphasis on performance than the power consumption of the pipeline.

On the other hand, following is a list of scenarios where EA controller is not desirable.

1. For simple FIFOs or micro-pipeline applications with logic depth of processing is very low -not more than 2–3 logic gates.
2. DSP applications, where pipeline process streams of data minimal or no iterations. The performance gains expected by using the EA controller heavily over weigh the power penalty incurred.

9.3 Future Work

We would like to evaluate and confirm the performance of the controllers on ASIC, like on 65nm technology. Experimental results in such a case are deemed necessary to strengthen our claims of the advantages of using Early Acknowledgement protocol.

This work mainly focused on the formulating and evaluating the performance of the Early Acknowledgement protocol. However, there is a great emphasis on power consumption in the present day design methodologies. Even though asynchronous designs are generally more power efficient than their synchronous counterparts, we would like to evaluate that for EA protocol in future. Especially a comparison of power consumption between the three asynchronous communication protocols will be interesting and will provide guidelines for designer in choosing protocols. As a general rule, EA protocol is expected to consume about the same power consumption as 4-phase protocol owing to its return-to-zero nature of signals.

Implementing different pipeline structures like *fork* is also necessary and is a requirement in composing complex pipelines using EA protocol. Moreover, the performance of EA protocol needs to be evaluated on different applications, for example on an MPEG decoder. The performance on pipelines processing data streams need to be examined.

List of Publications

1. C. Mannakkara, T. Yoneda, "Asynchronous Pipeline Controller Based on Early Acknowledgement Protocol", IEICE Transactions on Information and Systems. Vol. E93-D, No. 8, pages 2145-2161, August, 2010.

2. C. Mannakkara, T. Yoneda, "Asynchronous Pipeline Controller Based on Early Acknowledgement Protocol", NII Technical Report, #NII-2008-009E, pages 1-18, 2008.

3. C. Mannakkara, T. Yoneda, "Asynchronous Pipeline Controller Based on Early Acknowledgement Protocol", Proceedings on Applications of Concurrency to System Design, pages 118-127, 2008.

4. C. Mannakkara, T. Yoneda, "Comparison of Standard Cell based Non-linear Asynchronous Pipelines", IEICE Technical Report, VLSI, pages 49-54, 2007.

5. C. Mannakkara, S. Signell, "Software implementation of DVB-RCT Modulator", Conference on Industrial and Information Systems, Pages 463-469, 2006

Appendix A

UPPAAL model for EA controller

This contains the UPPAL model used to model check the EA controller.

A.1 Declarations

Following are the declarations of the model.

```
// PCE3x Controller IO & internal wires
bool Rin= 0;
bool Ain =0;
bool Rout =0;
bool Aout =0;
bool Clk =1;

bool rst =0;
bool complete=0;

// Broadcast channels for signalling output changes
broadcast chan Rin_change, Ain_change, Rout_change, Aout_change, Clk_change;
broadcast chan rst_change, complete_change;

// Input environment signals
broadcast chan Clk_in_posedge;
```

```
// Output environment signals
broadcast chan Clk_out_posedge;

// Observer variables
broadcast chan Clk_posedge, Clk_negedge;
broadcast chan Rin_posedge, Rin_negedge;
broadcast chan Ain_posedge, Ain_negedge;
broadcast chan Rout_posedge, Rout_negedge;
broadcast chan Aout_posedge, Aout_negedge;

// Controller Delays
const int A1_D = 4;
const int A2_D = 4;
const int CE_D = 4;
const int INV_D = 4;
// Controller RD delay
const int RD_UP = 4;
const int RD_DN = 4;

// Environment delays
const int IN_DN_MIN = 8; //  min == t_C_dn + t_MD_N_dn
const int IN_DN_MAX = 100;
const int IN_UP_MIN = 8; //?min == t_C_up + (t_MD_N_up >= 1 gate delay)
const int IN_UP_MAX = 100;
const int IN_CLK = 4;

const int OUT_DN_MIN = 8; // min == t_MD_N+1_dn + t_AND_dn
const int OUT_DN_MAX = 15; // < 16  (*** !! should be less than 4 gate delays !! ***)
const int OUT_UP_MIN = 8; // min == (t_MD_N+1_up >= 1 gate delay) + t_AND_up
const int OUT_UP_MAX = 100;
const int OUT_CLK = 4; // OUT_CLK + OUT_DN_MAX < 20  (i.e. 4 gate delays)
```

The following sections describe model of each logic gate of the EA controller.

A.2 2 Input gate (AND gate and C-element)

A.2.1 Model

The model of the 2 input gate is shown in Figure A.1. It should be noted that two models need to be created with the similar structure and the function (AND gate and C-element) should be replaced according to the model.

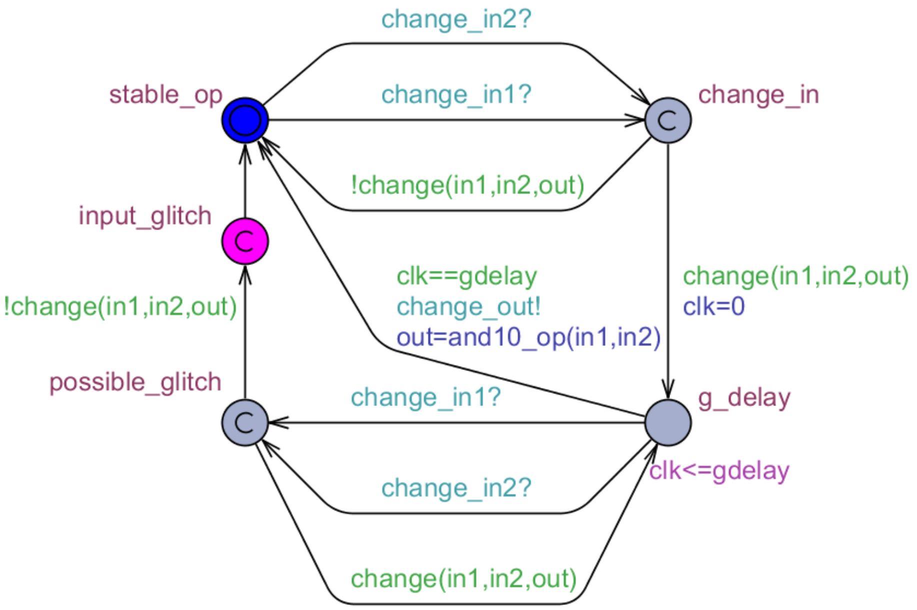


FIGURE A.1: 2-input logic gate modelled in UPPAAL

A.2.2 Declarations

This contains the UPPAL declarations used to model *both* AND gate and C-element functions.

```
// Place local declarations here.
clock clk;

// Check for changes in output for AND gate with inverted "in2"
// if out != (in1 && !in2) then change
//    no-change otherwise
//
bool change(bool in1, bool in2, bool out)
{
    return (in1 && !in2) != out;
}

// AND gate with inverted "in2"
// out = in1 && !in2
//
bool and10_op(bool &in1, bool &in2)
{
    return in1 && !in2;
}

// Check for changes in output for Celement with inverted "in2"
// if in1 != in2 and in1 != out then change
//    no-change otherwise
//
bool change(bool in1, bool in2, bool out)
{
    return (in1 !=in2) && (in1 !=out);
}

// C-element with inverted input "in2"
// out = in1 if in1 != in2
//     = out otherwise
//
bool cele10_op(bool &in1, bool &in2, bool &out)
{
    return (in1 != in2) ? in1: out;
}
```

A.3 Inverter

A.3.1 Model

Figure A.2, shows the model for the inverter (1-input logic) of the EA controller.

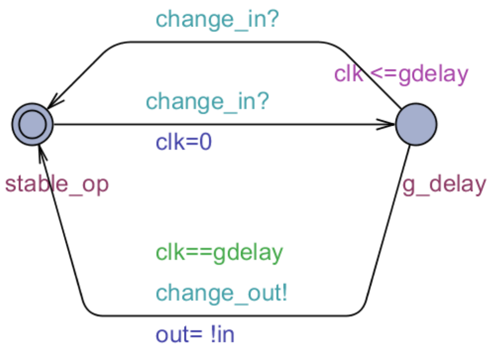


Figure A.2: Inverter gate modelled in UPPAAL

Note: There is just the clock `clk` in the declaration of the inverter.

A.4 Variable Delay

A.4.1 Model

Figure A.3, shows the model for the variable delay (1-input logic) of the EA controller.

A.5 Output Environment

A.5.1 Model

Figure A.4, shows the model for the output environment of the controller.

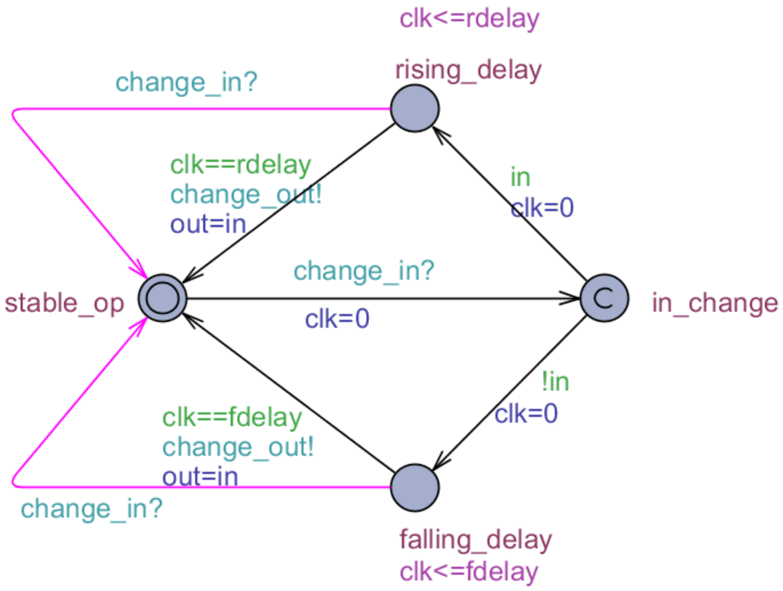


FIGURE A.3: Variable delay modelled in UPPAAL

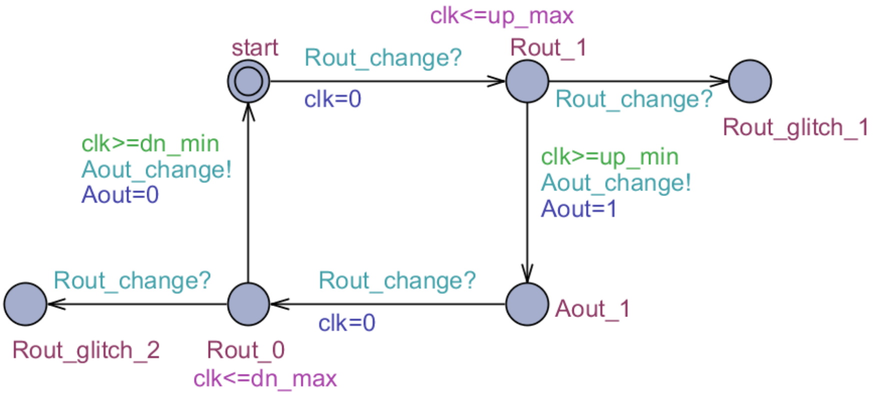


FIGURE A.4: Output Environment modelled in UPPAAL

A.6 Environment Clock

A.6.1 Model

Figure A.5, shows the model clocks of the input and output environments which captures the data.

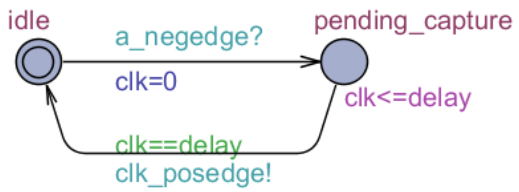


Figure A.5: Data capture clocks for the input and output environments

A.7 Edge detect (generic)

A.7.1 Model

Figure A.6, shows the generic model used to detect edges (positive and negative) from various signals. The detected edges are used in other observer models in turn.

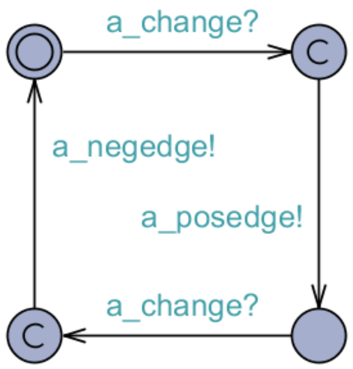


Figure A.6: Generic observer model to detect edges

A.8 Clock missing property

A.8.1 Model

Figure A.7, shows the model for testing "clock missing" property using edge detectors (above) for *Rin* and *Rout* signals.

Similar models are used to model the "*A1* blocking" and "*Clk_out* missing" properties.

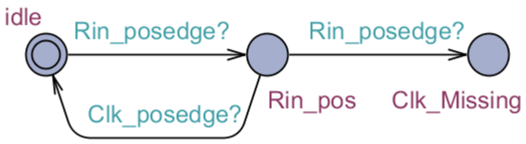


FIGURE A.7: Observer model to test clock missing property

A.9 System Declarations

```
// PCE3x Controller
A1_and = AND10Gate(Rin, Rout, Ain, Rin_change, Rout_change, Ain_change, A1_D);
A2_and = AND10Gate(rst, Rin, complete, rst_change, Rin_change, complete_change, A2_D);
C_ele = Celement10(complete, Aout, Rout, complete_change, Aout_change,
                   Rout_change, CE_D);
RD_delay = VDelay(Ain, rst, Ain_change, rst_change, RD_UP, RD_DN);
CLK_inv = Inverter(Ain, Clk, Ain_change, Clk_change, INV_D);

// signals deriving edges
obs_Rin_edge = Obs_edge_detect(Rin_change, Rin_posedge, Rin_negedge);
// obs_Ain_edge = Obs_edge_detect(Ain_change, Ain_posedge, Ain_negedge);
obs_Rout_edge = Obs_edge_detect(Rout_change, Rout_posedge, Rout_negedge);
obs_Aout_edge = Obs_edge_detect(Aout_change, Aout_posedge, Aout_negedge);
obs_Clk_edge = Obs_edge_detect(Clk_change, Clk_negedge,
                       Clk_posedge); // swapped edge signals as init. Clk==1

// Input side Environment
In_env = InputEnv(Rin, Rin_change, Ain_change, IN_DN_MIN, IN_DN_MAX,
                  IN_UP_MIN, IN_UP_MAX);
In_env_clk = EnvClk(Rin_negedge, Clk_in_posedge, IN_CLK);

// Output side Environment
Out_env = OutputEnv(Aout, Aout_change, Rout_change, OUT_DN_MIN, OUT_DN_MAX,
                    OUT_UP_MIN, OUT_UP_MAX);
Out_env_clk = EnvClk(Aout_negedge, Clk_out_posedge, OUT_CLK);
```

```
// Observers
obs_Clk_Missing = Obs_Clk_Missing(Rin_posedge, Clk_posedge);
obs_A1_Blocking = Obs_A1_Blocking(Rin_posedge, Rout_posedge);
obs_Clk_out_Missing = Obs_Clk_out_Missing(Clk_posedge, Clk_out_posedge);

// System components
system A1_and, A2_and, C_ele, RD_delay, CLK_inv, // Controller
        obs_Rin_edge, /* obs_Ain_edge, */ obs_Rout_edge, obs_Aout_edge,
        obs_Clk_edge, // Edge signals
        In_env, In_env_clk,    // Input Env
        Out_env, Out_env_clk,  // Output Environment
obs_Clk_Missing,
        obs_A1_Blocking,
        obs_Clk_out_Missing;   // Observers
```

Bibliography

[1] Chris J. Myers. *Asynchronous Circuit Design.* Wiley-Interscience, 2001. ISBN 978-0471415435.

[2] Steve Furber Jens Spars. *Principles of Asynchronous Circuit Design: A Systems Perspective.* Springer, 2001. ISBN 978-0792376132.

[3] E.G. Friedman. Clock distribution networks in synchronous digital integrated circuits. *Proceedings of the IEEE*, 89(5):665–692, May 2001. ISSN 0018-9219.

[4] M. Singh and S.M. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, pages 198–209, 2000.

[5] S.B. Furber, J.D. Garside, P. Riocreux, S. Temple, P. Day, Jianwei Liu, and N.C. Paver. AMULET2e: an asynchronous embedded controller. *Proceedings of the IEEE*, 87(2):243–256, Feb 1999. ISSN 0018-9219.

[6] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, and G. Stegmann. An asynchronous low-power 80C51 microcontroller. *Advanced Research in Asynchronous Circuits and Systems*, pages 96–107, Mar-2 Apr 1998.

[7] J. D. Garside, W. J. Bainbridge, A. Bardsley, D. M. Clark, D. A. Edwards, S. B. Furber, D. W. Lloyd, S. Mohammadi, J. S. Pepper, S. Temple, J. V. Woods, J. Liu, and O. Petlin. AMULET3i - an asynchronous System-on-Chip. page 162, 2000.

[8] N.C. Paver, P. Day, C. Farnsworth, D.L. Jackson, W.A. Lien, and J. Liu. A low-power, low noise, configurable self-timed DSP. *Advanced Research in Asynchronous Circuits and Systems*, pages 32–42, Mar-2 Apr 1998.

[9] Stephen H. Unger. Hazards, critical races, and metastability. *IEEE Trans. Comput.*, 44(6):754–768, 1995. ISSN 0018-9340.

[10] Leonard R. Marino. General theory of metastable operation. *IEEE Transactions on Computers*, pages 107–115, 1981.

[11] F.U. Rosenberger, C.E. Molnar, T.J. Chaney, and T.-P. Fang. Q-modules: internally clocked delay-insensitive modules. *Computers, IEEE Transactions on*, 37(9): 1005–1018, Sep 1988. ISSN 0018-9340.

[12] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In *AUSCRYPT '90: Proceedings of the sixth MIT conference on Advanced research in VLSI*, pages 263–278, Cambridge, MA, USA, 1990. MIT Press. ISBN 0-262-04109-X.

[13] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V. Woods. AMULET1: a micropipelined ARM. *Compcon Spring '94, Digest of Papers.*, 1(1):476–485, Feb-4 Mar 1994.

[14] S.B. Furber and P. Day. Four-phase micropipeline latch control circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 4(2):247–253, Jun 1996. ISSN 1063-8210.

[15] M. Singh and S.M. Nowick. MOUSETRAP: ultra-high-speed transition-signaling asynchronous pipelines. *International Conference on Computer Design (ICCD)*, pages 9–17, 2001.

[16] M. Singh and S.M. Nowick. MOUSETRAP: High-speed transition-signaling asynchronous pipelines. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(6):684 –698, jun. 2007. ISSN 1063-8210.

[17] T. Yoneda, A. Matsumoto, M. Kato, and C. Myers. High level synthesis of timed asynchronous circuits. *Asynchronous Circuits and Systems (ASYNC), Proceedings. 11th IEEE International Symposium on*, pages 178–189, March 2005. ISSN 1522-8681.

[18] N. Sretasereekul, H. Saito, M. Imai, E. Kim, M. Ozcan, K. Thongnoo, H. Nakamura, and T. Nanya. A zero-time-overhead asynchronous four-phase controller. *International Symposium on Circuits and Systems (ISCAS), Proceedings*, 5:V–205–V–208, May 2003.

[19] D. E. Muller. Asynchronous logics and application to information processing. *Switching Theory in Space Technology*, pages 289–297, 1963.

[20] T. Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. Technical report, Cambridge, MA, USA, 1987.

[21] Leonid Ya. Rosenblum and Alexandre Yakovlev. Signal graphs: From self-timed to timed ones. In *International Workshop on Timed Petri Nets*, pages 199–206, Washington, DC, USA, 1985. IEEE Computer Society. ISBN 0-8186-0674-6.

[22] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[23] L. Lavagno, K. Keutzer, and A.L. Sangiovanni-Vincentelli. Synthesis of hazard-free asynchronous circuits with bounded wire delays. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 14(1):61–86, Jan 1995. ISSN 0278-0070.

[24] I. Blunno, J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. Handshake protocols for de-synchronization. *Asynchronous Circuits and Systems. Proceedings. 10th International Symposium on*, pages 149–158, April 2004. ISSN 1522-8681.

[25] C. Mannakkara and T. Yoneda. Comparison of standard cell based non-linear asynchronous pipelines. *IEICE Technical Report*, 107(337):49–54, 2007. ISSN 09135685.

[26] Johan Bengtsson, Fredrik Larsson, Paul Pettersson, Wang Yi, Palle Christensen, Jesper Jensen, Per Jensen, Kim Larsen, and Thomas Sorensen. UPPAAL: a tool suite for validation and verification of real-time systems. *Proceedings of Workshop on Verification and Control of Hybrid Systems III*, pages 232–243, 1995.

[27] UPPAAL Model Checker. http://www.uppaal.com/.

[28] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAALL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[29] M.Singh. Private communication with montek singh.

[30] ALTERA (White Paper). Stratix II vs. Virtex-4 performance comparison.

[31] H. Jeffreys and B. S. Jeffreys. Methods of mathematical physics. pages 305–306, 1988.

[32] W. H. Press, B. P. Flannery, and S. A. Teukolsky. Numerical Recipes in FORTRAN: The Art of Scientific Computing. pages 864–866, 1992.

[33] Edgard Garcia. Writing RTL Code for Virtex-4 DSP48 Blocks with XST 8.1i. *Xcell Journal Online*, 1995.

[34] C. Mannakkara and T. Yoneda. Asynchronous pipeline controller based on early acknowledgement protocol. *NII Technical Report, NII-2008-009E*, pages 1–18, 2008.

[35] C. Mannakkara and T. Yoneda. Asynchronous pipeline controller based on early acknowledgement protocol. *Application of Concurrency to System Design, 2008. ACSD 2008. 8th International Conference on*, pages 118–127, June 2008. ISSN 1550-4808.

[36] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989. ISSN 0001-0782.

[37] Erik Brunvand. Using FPGAs to implement self-timed systems. *J. VLSI Signal Process. Syst.*, 6(2):173–190, 1993. ISSN 0922-5773.

[38] Erik Lee Brunvand. *Translating concurrent communicating programs into asynchronous circuits.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[39] R.O. Ozdag, M. Singh, P.A. Beerel, and S.M. Nowick. High-speed non-linear asynchronous pipelines. *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 1000–1007, 2002.

[40] Quoc Thai Ho, Jean-Baptiste Rigaud, Laurent Fesquet, Marc Renaudin, and Robin Rolland. Implementing asynchronous circuits on LUT based FPGAs. *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 36–46, 2002.

[41] Y. Sato, Yamasoto Y., M. Saito, H. Kagotani, T. Okamoto, and M. Kawai. Systematic reducing of metastable operation occurred in CMOS D flip-flops. *Systems and Computers in Japan*, 81(9):1090–1098, 1998. ISSN 09151915.

[42] A. Peeters. Support for interface design in Tangram. *Asynchronous Interfaces: Tools, Techniques, and Implementations*, pages 57–64, 2000.

[43] K. Van Berkel, F. Huberts, and A. Peeters. Stretching quasi delay insensitivity by means of extended isochronic forks. In *ASYNC '95: Proceedings of the 2nd Working*

Conference on Asynchronous Design Methodologies, pages 99–106, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7098-3.

[44] M. Ampalam and M. Singh. Counterflow pipelining: Architectural support for preemption in asynchronous systems using anti-tokens. *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 611–618, Nov. 2006. ISSN 1092-3152.

[45] Montek Singh, Steven M. Nowick, Jose A. Tierno, Sergey Rylov, and Alexander Rylyakov. An adaptively-pipelined mixed synchronous-asynchronous digital FIR filter chip operating at 1.3 GigaHertz. page 84, 2002.

[46] P. Balaji, W. Mahmoud, E. Ososanya, and K. Thangarajan. Survey of the counterflow pipeline processor architectures. *System Theory, Proceedings of the Thirty-Fourth Southeastern Symposium on*, pages 1–5, 2002. ISSN 0094-2898.

[47] Blunno Ivan and Lavagno Luciano. Automated synthesis of micro-pipelines from behavioral Verilog HDL. In *ASYNC '00: Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 84, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0586-4.

[48] Ligthart Michiel, Fant Karl, Smith Ross, Taubin Alexander, and Kondratyev Alex. Asynchronous design using commercial HDL synthesis tools. In *ASYNC '00: Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 114, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0586-4.

[49] Ozdag Recep O. and Beerel Peter A. High-speed QDI asynchronous pipelines. In *ASYNC '02: Proceedings of the 8th International Symposium on Asynchronus Circuits and Systems*, page 13, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1540-1.

[50] Victor Khomenko and Mark Schaefer. Combining decomposition and unfolding for STG synthesis. In *ICATPN'07: Proceedings of the 28th international conference on Applications and theory of Petri nets and other models of concurrency*, pages 223–243, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-73093-4.

[51] A. Semenov, A. Yakovlev, E. Pastor, M. A. Pe na, and J. Cortadella. Synthesis of speed-independent circuits from STG-unfolding segment. In *DAC '97: Proceedings of the 34th annual Design Automation Conference*, pages 16–21, New York, NY, USA, 1997. ACM. ISBN 0-89791-920-3.

[52] K. Y. Yun, P. A. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *ASYNC '96: Proceedings of the 2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 17, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7298-6.

[53] Jung-Lin Yang and Erik Brunvand. Using dynamic domino circuits in self-timed systems. In *GLSVLSI '03: Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 253–256, New York, NY, USA, 2003. ACM. ISBN 1-58113-677-3.

[54] Charles E. Molnar and Ian W. Jones. Simple circuits that work for complicated reasons. In *ASYNC '00: Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 138, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0586-4.

[55] Montek Singh and Steven M. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *ASYNC '00: Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 198, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0586-4.

[56] Frédéric Béal, Tomohiro Yoneda, and Chris J. Myers. Hazard checking of timed asynchronous circuits revisited. *Fundam. Inf.*, 88(4):411–435, 2008. ISSN 0169-2968.

[57] Chris Myers and Teresa H. Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1:106–119, 1993.

[58] Curtis A. Nelson, Chris J. Myers, and Tomohiro Yoneda. Efficient verification of hazard-freedom in gate-level timed asynchronous circuits. In *ICCAD '03: Proceedings of IEEE/ACM international conference on Computer-aided design*, page 424, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 1-58113-762-1.

[59] Tomohiro Yoneda. Synthesis of speed independent circuits based on decomposition. In *In ASYNC 2004*, pages 135–145. Society Press, 2004.

[60] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers, 1996.

[61] Xiao Yong and Zhou Runde. Single-track asynchronous pipeline controller design. In *ASP-DAC '05: Proceedings of the Asia and South Pacific Design Automation Conference*, pages 764–768, New York, NY, USA, 2005. ACM. ISBN 0-7803-8737-6.

[62] Je-Hoon Lee, Seung-Sook Lee, and Kyoung-Rok Cho. Asynchronous ARM processor employing an adaptive pipeline architecture. In *ARC'07: Proceedings of the 3rd international conference on Reconfigurable computing*, pages 39–48, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71430-9.

[63] Montek Singh and Steven M. Nowick. The design of high-performance dynamic asynchronous pipelines: lookahead style. *IEEE Trans. Very Large Scale Integr. Syst.*, 15(11):1256–1269, 2007. ISSN 1063-8210.

[64] Montek Singh and Steven M. Nowick. The design of high-performance dynamic asynchronous pipelines: high-capacity style. *IEEE Trans. Very Large Scale Integr. Syst.*, 15(11):1270–1283, 2007. ISSN 1063-8210.

[65] D. Sokolov, A. Bystrov, and A. Yakovlev. STG optimisation in the direct mapping of asynchronous circuits. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10932, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1870-2.

[66] Ivan Sutherland, Bob Sproull, and David Harris. *Logical effort: designing fast CMOS circuits*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1-55860-557-6.

[67] Christos P. Sotiriou. Implementing asynchronous circuits using a conventional EDA tool-flow. In *DAC '02: Proceedings of the 39th annual Design Automation Conference*, pages 415–418, New York, NY, USA, 2002. ACM. ISBN 1-58113-461-4.

[68] Peggy B. McGee and Steven M. Nowick. A lattice-based framework for the classification and design of asynchronous pipelines. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 491–496, New York, NY, USA, 2005. ACM. ISBN 1-59593-058-2.

[69] Sunan Tugsinavisut, Youpyo Hong, Daewook Kim, Kyeounsoo Kim, and Peter A. Beerel. Efficient asynchronous bundled-data pipelines for DCT matrix-vector multiplication. *IEEE Trans. Very Large Scale Integr. Syst.*, 13(4):448–461, 2005. ISSN 1063-8210.

[70] Tiberiu Chelcea, Girish Venkataramani, and Seth C. Goldstein. Self-resetting latches for asynchronous micro-pipelines. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 986–989, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1.

[71] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *ARVLSI '97: Proceedings of the 17th Conference on Advanced Research in VLSI (ARVLSI '97)*, page 164, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7913-1.

[72] Rakefet Kol and Ran Ginosar. Kin: a high performance asynchronous processor architecture. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 433–440, New York, NY, USA, 1998. ACM. ISBN 0-89791-998-X.