µSystems Research Group

School of Electrical and Electronic Engineering

Newcastle University

Elastic Bundles: Modelling and Architecting Asynchronous Circuits with Granular Rigidity

Johnson Fernandes

Technical Report Series

NCL-EEE-MICRO-TR-2017-206

May 2017

Contact: johnson.fernandes@ncl.ac.uk

Supported by EPSRC grant EP/I038551/1

NCL-EEE-MICRO-TR-2017-206

µSystems Research Group

School of Electrical and Electronic Engineering

Merz Court

Newcastle University

Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

Elastic Bundles:

Modelling and Architecting Asynchronous Circuits with Granular Rigidity

Johnson Fernandes

A Thesis submitted for the degree of Doctor of Philosophy

School of Electrical and Electronic Engineering

Newcastle University

March 2017

Abstract

Integrated Circuit (IC) designs these days are predominantly System-on-Chips (SoCs). The complexity of designing a SoC has increased rapidly over the years due to growing process and environmental variations coupled with global clock distribution difficulty. Moreover, traditional synchronous design is not apt to handle the heterogeneous timing nature of modern SoCs. As a countermeasure, the semiconductor industry witnessed a strong revival of asynchronous design principles. A new paradigm of digital circuits emerged, as a result, namely mixed synchronous-asynchronous circuits. With a wave of recent innovations in synchronous-asynchronous CAD integration, this paradigm is showing signs of commercial adoption in future SoCs mainly due to the scope for reuse of synchronous functional blocks and IP cores, and the co-existence of synchronous and asynchronous design styles in a common EDA framework.

However, there is a lack of formal methods and tools to facilitate mixed synchronous-asynchronous design. In this thesis, we propose a formal model based on Petri nets with step semantics to describe these circuits behaviourally. Implication of this model in the verification and synthesis of mixed synchronous-asynchronous circuits is studied. Till date, this paradigm has been mainly explored on the basis of Globally Asynchronous Locally Synchronous (GALS) systems. Despite decades of research, GALS design has failed to gain traction commercially. To understand its drawbacks, a simulation framework characterising the physical and functional aspects of GALS SoCs is presented.

A novel method for synthesising mixed synchronous-asynchronous circuits with varying levels of rigidity is proposed. Starting with a high-level dataflow model of a system which is intrinsically asynchronous, the key idea is to introduce rigidity of chosen granularity levels in the model without changing functional behaviour. The system is then partitioned into functional blocks of synchronous and asynchronous elements before being transformed into an equivalent circuit which can be synthesised using standard EDA tools.

Acknowledgements

I would like to thank my supervisors; Prof. Alex Yakovlev, Dr. Alex Bystrov and Dr. Danil Sokolov for their invaluable guidance throughout this research. This thesis would not have been possible without their support, patience and encouragement over the last five years.

I would also like to thank Prof. Maciej Koutny and Dr. Marta Pietkiewicz-Koutny for formalisation of concepts presented in Chapter 4.

Thanks goes to Dr. James Docherty, Dr. Athanasios Grivas, Dr. Graeme Coapes and Mr. Alessandro De Gennaro for their hours of friendly discussions that kept the research atmosphere lively.

Special thanks goes to my parents Joseph and Apoline, and to my sisters Jenifer and Jessica for their love and constant encouragement during the course of this research. I am also thankful to Ravneet for her patience and wholehearted support, especially during the writing up of this thesis.

Finally, I would like to acknowledge that this work was partly sponsored by the School of Electrical and Electronic Engineering, Newcastle University and the EPSRC under the research grant GAELS EP/I038551/1.

Publications

Parts of this work have appeared in the following publications:

- Journal Paper
  - J. Fernandes, M. Koutny, L. Mikulski, M. Pietkiewicz-Koutny, D. Sokolov, and A. Yakovlev, "Persistent and Nonviolent steps and the design of GALS systems", Fundamenta Informaticae, vol. 137, pp. 143–170, 2015.
- Conference Papers
  - J. Fernandes, D. Sokolov, and A. Yakovlev, "Elastic Bundles: Modelling and synthesis of asynchronous circuits with granular rigidity", International Symposium on Asynchronous Circuits and Systems (ASYNC), 2017. (in press)
  - J. Fernandes, M. Koutny, M. Pietkiewicz-Koutny, D. Sokolov, and A. Yakovlev, "Step persistence in the design of GALS systems", International Conference on Applications and Theory of Petri Nets and Concurrency (ICATPN), vol. 7927, pp. 190–209, 2013.
- Technical Report
  - J. Fernandes, M. Koutny, M. Pietkiewicz-Koutny, D. Sokolov, and A. Yakovlev, "Step persistence in the design of GALS systems", Technical Report Series, CS-TR-1349, School of Computing Science, Newcastle University, 2012.

Contents

List of Figures

List of Tables

List of Abbreviations

ASIC - Application-Specific Integrated Circuit

CAD - Computer Aided Design

CDFG - Control Data Flow Graph

CDN - Clock Distribution Network

CRG - Concurrent Reachability Graph

DFG - Data Flow Graph

DSM - Deep Submicron

EDA - Electronic Design Automation

EMI - Electro-Magnetic Interference

FF - Flip Flop

FFT - Fast Fourier Transform

GALS - Globally Asynchronous Locally Synchronous

HLS - High-Level Synthesis

IC - Integrated Circuit

NoC - Network-On-Chip

PN - Petri Net

REE - Relative Energy Efficiency

RT - Relative Timing

RTL - Register-Transfer Level

SoC - System-On-Chip

STG - Signal Transition Graph

Chapter 1

Introduction

1.1 Motivation

Traditional circuit design styles have been following one of the two main strands, namely synchronous and asynchronous. In a nutshell, these two approaches differ in their techniques of synchronising interaction between circuit elements. Asynchronous designs adopt *'on request'* synchronisation where interaction is regulated by means of handshake control signals. They are designed to be adaptive to delays of signal propagation. Synchronous designs, on the other hand, assume worst case delay between circuit elements and determine a global periodic control signal for synchronisation called the *clock*. The clock signal limits the many sequencing options considered in asynchronous control. Synchronous circuits are, therefore, considered to be a proper subset of asynchronous circuits [3]. Asynchronous logic was the dominant design style with most early computers. In particular, David Muller's speed-independent circuits, dating back to the late 1950s, have served many interesting applications such as the ILLIAC I and ILLIAC II computers [4]. However, since 1960, an era when fabrication of integrated circuits (ICs) became a feasible business, synchronous design became the mainstream technique as it met the market needs with its shorter design flow cycle. Today, majority of designs are synchronous, well etched in the heart of semiconductor industry together with superior CAD tools and EDA flows.

IC designs these days are predominantly system-on-chip (SoC) implementations. Most SoCs consist of various intellectual property (IP) blocks embedded into one chip, with each block having its own timing requirement. This design technique favours the reuse of pre-designed and pre-verified IP blocks, benefiting the semiconductor industry with large productivity gains [5]. The complexity of designing a SoC has grown rapidly over the years due to demand for increased functionality, faster processing speeds and the increasing impact of previously ignorable effects on chip reliability. One of the main reasons is that transitioning to very deep submicron manufacturing (DSM) process has revealed process and environmental variation related problems coupled with the complexity to manage a global clock distribution network (CDN). Moreover, the traditional synchronous clocking style, being more appealing towards processor and instruction-flow designs, is not apt to handle the heterogeneous timing nature of modern SoCs. The semiconductor industry has been constantly researching and adopting new methodologies to improve designer productivity. Consequently, two distinct strategies have emerged, namely *Physical Partitioning* and *Functional Partitioning*, to tackle the design challenges of modern SoCs. Design partitioning creates smaller manageable islands with optimal CDNs that would reduce the effect of DSM variability whilst allowing individual islands to operate at their native clock speeds, thereby offering superior design quality than traditional globally clocked SoCs.

Some of the well-known physical partitioning techniques are ARM's big.LITTLE architecture [6], Teklatech's FloorDirector partitioning engine [7], NoCs [8, 9] and GALS-based physical partitioning [10, 11, 12, 13, 14, 15, 16]. These techniques partition a digital design based on physical characteristics to simplify clock distribution, reduce global interconnect complexity, reduce electro-magnetic interference (EMI), improve energy-efficiency and implement voltage frequency scaling on a coarse-grained level. Functional partitioning, on the other hand, partitions a design based on system functionality creating timing-related functional islands that are fine-grained. This offers greater flexibility for design optimisation, thus demonstrating scope for larger benefits than those gained from the physical partitioning approach. High-level synthesis flows such as BlueSpec [17, 18] and Chisel [19]; dataflow synthesis flows such as Click [20] and eTeak [21]; and Petri net (PN) based partitioning techniques such as [22] and [23] fall under this category. With exception of Click, eTeak, PN based partitioning and GALS-based physical partitioning which are based on asynchronous design principles, all the other techniques are implemented using traditional synchronous EDA tool flow.

In the recent decade, the semiconductor industry has witnessed a strong revival of asynchronous design principles due to its robustness to DSM variability coming from fine-grained circuit timing flexibility, its inherent elasticity and heterogeneity offering great design flexibility, its lower energy consumption profile coming from inherent automatic fine-grained clock gating, and its independence from the need for global clock distribution. Currently, the traditional asynchronous design approach is still infeasible commercially due to the lack of mature CAD tools, significant overheads of asynchronous control protocol, inability to reuse synchronous IP cores and high transitioning costs. However, due to growing semiconductor variability and demand for energy-aware computing, it has been predicted that it is only a matter of time before asynchronous logic penetrates a vast majority of SoCs [24]. Recent innovations in asynchronous design flows such as those in [25, 26, 27, 28, 29, 30, 31, 32, 33] have enabled rapid integration of asynchronous logic into synchronous CAD tools thereby bridging the gap between asynchronous design and designer productivity. In the realm of SoCs, this trend has resulted in an active area of research revolving around mixed synchronous-asynchronous digital design [34, 35, 12, 36, 13, 37, 38, 39, 40, 41, 16, 23, 42, 43]. Figure 1.1 illustrates the positioning of this research area.

Mixed synchronous-asynchronous design can be viewed as a revolutionary step in VLSI design paving the way for commercial adoption of asynchronous design techniques in future SoCs. With a growing trend of synchronous-asynchronous CAD tool flow integration, the mixed synchronous-asynchronous paradigm shows promise to survive the market forces of the semiconductor industry mainly due to the scope for reuse of synchronous functional blocks and IP cores, and the co-existence of synchronous and asynchronous design styles in a common EDA framework. However, there is a lack of tools and design methodologies to facilitate mixed synchronous-asynchronous design. So far, this research area has been mainly explored on the basis of GALS systems [44]. Most GALS design implementations emphasised on physical partitioning and followed an assemble-and-validate design style. They lacked formal reasoning methods and considered functional partitioning aspects

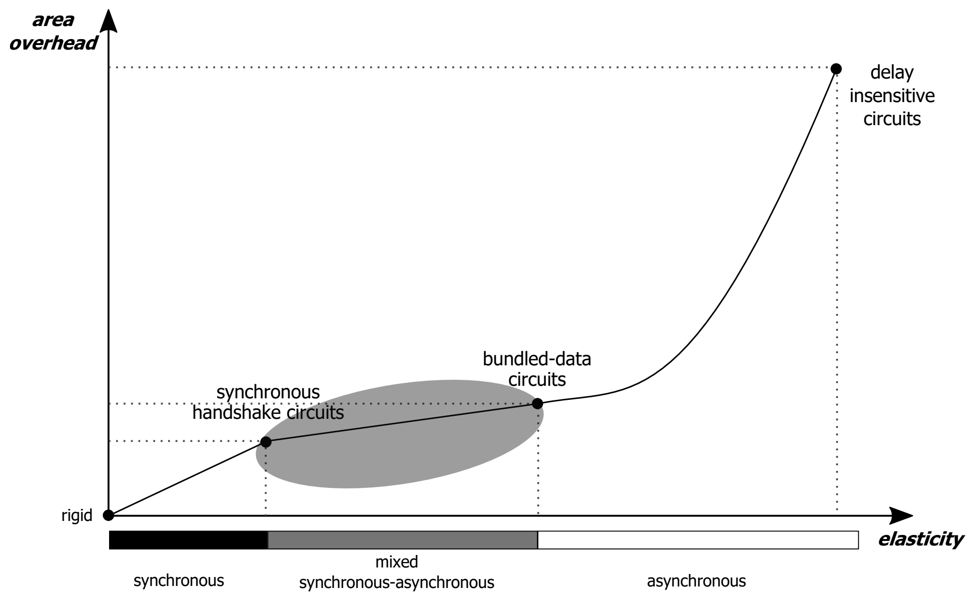

area overhead
delay insensitive circuits
bundled-data circuits
synchronous handshake circuits
rigid
elasticity
synchronous
mixed synchronous-asynchronous
asynchronous

Figure 1.1: Illustration of mixed synchronous-asynchronous circuits

on a very coarse-grain level defined by top-level hierarchical boundaries. Furthermore in the realm of circuits, GALS design would only comprise a subset of the shaded area shown in Figure 1.1. In fact, the mixed synchronous-asynchronous design paradigm would encapsulate a range of design practices starting from synchronous handshake circuits [26], which are synchronous circuits embracing principles of asynchronous elasticity, all the way to bundled-data circuits [45, 46], which are asynchronous circuits embracing principles of synchronous rigidity. Hence, we propose a strong motivation for three distinct research areas in this field:

1. Discovery of sub-classes of mixed synchronous-asynchronous circuits
2. Specification of a common formal method for mixed synchronous-asynchronous circuit description, and
3. Reasoning behind functional and physical partitioning strategies for mixed synchronous-asynchronous SoC design.

1.2 Research Goals and Thesis Contribution

Entering the paradigm of mixed synchronous-asynchronous circuits, we encountered the following questions which defined the research goals of this thesis:

- How do we model and verify circuits exhibiting mixture of synchronous and asynchronous behaviour?
- What is the ideal strategy for designing mixed synchronous-asynchronous SoCs; is it the physical partitioning approach or a functional partitioning strategy?
- What level of granularity is justified for synchronous and asynchronous logic elements in SoC design?

This thesis is centred around finding answers to these questions.

Decades of research have undergone in GALS-based physical partitioning methodology for mixed synchronous-asynchronous SoC design. However, the method has failed to gain traction commercially. It was unclear as to why there was no clear consensus about GALS design in the research community. Thus, the first research goal was to develop an extensive model that could characterise the physical and functional aspects of GALS SoCs and analyse the design technique's benefits and drawbacks. An extensive parametric model considering variability of several technology nodes was developed and GALS design was analysed versus its fully synchronous counterpart.

Secondly, it was conceptualised that mixed synchronous-asynchronous systems would require a unique mathematical model to represent its behaviour. The theory of step persistence and *bundles* was developed to cater this goal. Petri nets with step semantics were used to represent mixed synchronous-asynchronous behaviour. Implications of the model in verification and synthesis of mixed synchronous-asynchronous circuits were studied. A pruning algorithm was developed that would generate a suitable mixed synchronous-asynchronous behavioural model by pruning a system's concurrent specification to achieve desired design characteristics. This result demonstrated how a fully asynchronous design could be formally re-engineered to a mixed synchronous-asynchronous design that couples elastic asynchronous design flexibility with rigid synchronous design simplicity.

The final research goal was motivated from the exploration of research area identified in Figure 1.1. This led to a novel method for synthesising mixed synchronous-asynchronous circuits with varying levels of rigidity. The hypothesis was that *bundles* would reduce the area overheads of asynchronous design by relaxing granularity of handshake control. A Petri net based dataflow modelling technique was developed to model digital systems on a higher level of abstraction than RTL. Functional partitioning was then introduced in these dataflow models by identifying sets of *bundles* that could restrict elasticity whilst retaining functional behaviour. Taking the case of asynchronous bundled-data circuits, these sets of *bundles* were extended to a novel notion of *Elastic Bundles* which basically re-partitioned the design into coarse-grained locally clocked elements (synchronous) and fine-grained locally clocked elements (asynchronous). This net transformation enabled synthesis of the mixed synchronous-asynchronous circuit under standard EDA tool flow.

1.3 Thesis Organisation

The thesis is structured as follows:

Chapter 1 presents the motivation behind this research and outlines the main contributions of this thesis.

Chapter 2 introduces the formal modelling languages used throughout the thesis and provides background on the asynchronous synthesis flow adopted in this research.

Chapter 3 studies the impact of GALS-based physical partitioning in SoC design and presents the first contribution, a simulation framework that evaluates GALS design versus its fully synchronous counterpart considering physical and functional parameters.

Chapter 4 presents the second contribution, theory of step persistence and bundles, which provides a mathematical model to model and verify mixed synchronous-asynchronous circuits.

Chapter 5 applies the concepts discussed in Chapter 4 to synthesise asynchronous circuits of varying levels of rigidity. The notion of 'Elastic Bundles' is introduced to enforce rigidity in fully asynchronous circuits, demonstrating mixed synchronous-asynchronous circuits consisting of granular locally clocked circuit elements. The final contribution is a novel method for modelling and synthesising 'Elastic Bundles'.

Chapter 6 summarises the key results and contributions, and presents directions for future work.

Appendix A presents the MATLAB code written to derive the results shown in Chapter 3.

Appendix B presents main parts of the verilog code for the 16-point FFT design discussed in Chapter 5.

Appendix C presents the synthesis timing constraints used to synthesise the 4-point FFT Elastic-bundle Maximal circuit of Chapter 5.

Appendix D presents the Petri net and Policy net models of the 4-point FFT architecture on the basis of which the results of Chapter 5 were derived.

Chapter 2

Background

This chapter summarises the fundamental concepts on the basis of which the thesis is constructed. Section 2.1 introduces the formal modelling languages used to model and verify digital systems in this thesis. In Section 2.2, the principles of bundled-data asynchronous pipelines is discussed and the CAD flow adopted to synthesise these circuits is summarised.

2.1 Behavioural Modelling

In this section, we introduce the modelling languages used for the specification and verification of digital system behaviour. We recall definitions and notations of these formal models used throughout the thesis, specifically of step transition systems, Petri nets and step semantics of Petri nets. These preliminaries are based on the literature presented in [47, 48, 49, 50, 51, 3].

2.1.1 Step Transition Systems

This section introduces the step transition system model which is the chosen modelling language to describe the behaviour of both asynchronous circuits as well as synchronous circuits in this thesis. We define this model formally as follows:

Let T be a finite set of net transitions representing actions of a concurrent system. A set of transitions is called a *step*, and we use $\alpha, \beta, \gamma, \ldots$ to range over all steps $\mathcal{P}(T)$. Sometimes we identify a step α with its characteristic function $\alpha : T \to \{0, 1\}$, and then write $\alpha = \sum_{a \in T} \alpha(a) \cdot a$.[1] The size $|\alpha|$ of α is defined as the number of its elements.

Definition 2.1.1. (step transition system)

A *step transition system* (or ST-system) over a set of net transitions T is a triple $STS = (Q, A, q_0)$ consisting of a set of *states* Q, including the initial state $q_0 \in Q$, and a set of labelled arcs $A \subseteq Q \times \mathcal{P}(T) \times Q$. It is assumed that:

- $(q, \varnothing, q) \in A$ for all $q \in Q$;[2]
- the transition relation is deterministic, i.e., if $(q, \alpha, q') \in A$ and $(q, \alpha, q'') \in A$ then $q' = q''$;
- each state is reachable, i.e., if $q \in Q$ then there are sequences of steps $\alpha_1, \ldots, \alpha_n$ $(n \geq 0)$ and states $q_1, \ldots, q_n = q$ such that $(q_{i-1}, \alpha_i, q_i) \in A$ for $1 \leq i \leq n$. ■

For an ST-system STS as above, we introduce the following notations:

- $q \xrightarrow{\alpha} q'$ and $q \xrightarrow{\alpha}$ whenever $(q, \alpha, q') \in A$;
- $En_{STS}(q) = \{\alpha \mid q \xrightarrow{\alpha}\} \subseteq \mathcal{P}(T)$ is the set of all steps enabled at a state q;
- $En_{STS} = \bigcup_{q \in Q} En_{STS}(q) \subseteq \mathcal{P}(T)$ is the set of all the enabled steps of STS;
- $max(q) = \{\alpha \in En_{STS}(q) \mid \forall \beta \in En_{STS}(q) : \ \alpha \not\subset \beta\}$ is the set of all maximal steps enabled at a state q.
- $ready_{STS}(q) = \bigcup En_{STS}(q) = \bigcup_{\alpha \in En_{STS}(q)} \alpha \subseteq T$ is the set of all transitions ready to be executed at a state q, i.e., those belonging to steps enabled at q.

2.1.2 Petri nets

Petri nets, introduced by C. A. Petri in his PhD dissertation [52], are widely used as a formal model to express concurrent systems. It is a powerful language which can be used to generate graphical and mathematical representation of asynchronous digital circuits and other distributed systems [50, 51, 3, 53, 54]. The *place/transition net* (or PT-net) is the most frequently used and best studied class of Petri nets where a place can hold any

[1] The ‘ $\cdot$ ’ operator used here denotes scalar multiplication.

[2] For technical reasons we want to keep empty steps, as they might be important in future algorithms.

number of tokens. In this section, we recall definitions, notations and basic properties of *place/transition net*s.

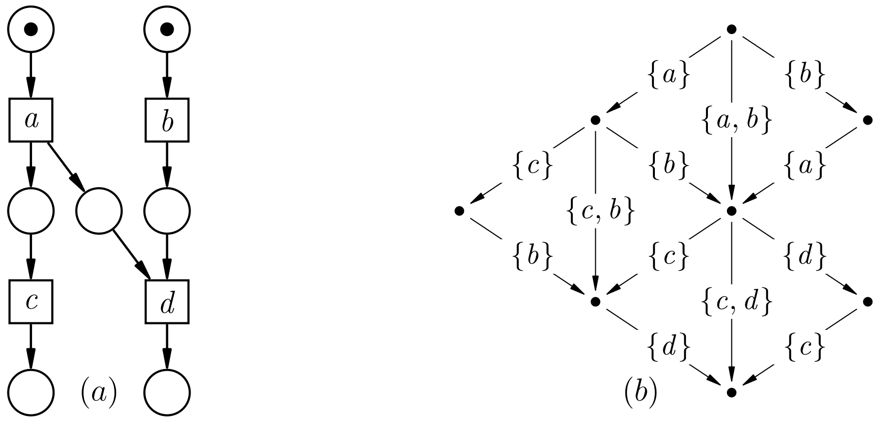


Figure 2.1: A PT-net $\mathcal{N}$ (a); and its concurrent reachability graph $CRG(\mathcal{N})$ (b). Note that arcs (in fact, self-loops) labelled by the empty step are not shown in $CRG(\mathcal{N})$.

Definition 2.1.2. (place/transition net)

A *place/transition net* (or PT-net) is a tuple $\mathcal{N} = (P, T, W, M_0)$, where P and T are finite disjoint sets of respectively *places* and *transitions*, $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ is an arc weight function, and $M_0 : P \to \mathbb{N}$ is an *initial marking* (in general, any mapping $M : P \to \mathbb{N}$ is a marking). ■

We use the standard conventions concerning the graphical representation of PT-nets, as shown in Figure 2.1(a). For every element $x \in P \cup T$, we use

$$^\bullet x = \{y \mid W(y, x) > 0\} \quad \text{and} \quad x^\bullet = \{y \mid W(x, y) > 0\}$$

to denote the pre-set and post-set of x, respectively. If $x \in T$, then any $p \in {}^\bullet x$ is a pre-place of x, and any $p \in x^\bullet$ is a post-place. The dot-notation extends in the usual way to a set $X \subseteq P \cup T$:

$$^\bullet X = \bigcup_{x \in X} {}^\bullet x \quad \text{and} \quad X^\bullet = \bigcup_{x \in X} x^\bullet .$$

Moreover, for every place $p \in P$ and step $\alpha \in \mathcal{P}(T)$, we denote:

$$W(p, \alpha) = \sum_{a \in T} \alpha(a) \cdot W(p, a) \quad \text{and} \quad W(\alpha, p) = \sum_{a \in T} \alpha(a) \cdot W(a, p) .$$

In other words, $W(p,\alpha)$ gives the number of tokens that the execution of α removes from p, and $W(\alpha,p)$ is the total number of tokens inserted into p after the execution of α.

Definition 2.1.3. (place/transition net behaviour)

Let M be a marking of $\mathcal{N}$. A step $\alpha \in \mathcal{P}(T)$ is *enabled* and may be *executed* at M if, for every $p \in P$:

$$M(p) \geq W(p,\alpha)\ . \tag{2.1}$$

We denote this by $M[\alpha\rangle$. Executing such a step leads to the marking M', for every $p \in P$ defined by:

$$M'(p) = M(p) - W(p,\alpha) + W(\alpha,p)\ . \tag{2.2}$$

We denote this by $M[\alpha\rangle M'$. Moreover, for a singleton step $\alpha = \{a\}$, we write $M[a\rangle$ and $M[a\rangle M'$ rather than $M[\{a\}\rangle$ and $M[\{a\}\rangle M'$. ■

The *concurrent reachability graph* of $\mathcal{N}$ is the ST-system

$$\mathit{CRG}(\mathcal{N}) = ([M_0\rangle, A, M_0)$$

over T where:

$$[M_0\rangle = \{M_n \mid \exists \alpha_1,\ldots,\alpha_n, M_1,\ldots,M_{n-1}\ \forall 1 \leq i \leq n\ :\ M_{i-1}[\alpha_i\rangle M_i\} \tag{2.3}$$

is the set of reachable markings and $(M,\alpha,M') \in A$ *iff* $M[\alpha\rangle M'$. Furthermore, we call $\alpha_1 \ldots \alpha_n$, as in the formula (2.3), a *step sequence* and write $M_0[\alpha_1 \ldots \alpha_n\rangle M_n$. Figure 2.1($b$) shows the concurrent reachability graph of the PT-net in Figure 2.1(a).

Remark 2.1.1. Note that for any Petri net $\mathcal{N}$, $\mathit{CRG}(\mathcal{N})$ is an ST-system, but we have different requirements for steps enabled in $\mathit{CRG}(\mathcal{N})$ and in an arbitrary ST-system. This is intentional distinction. In Chapter 4, we define sub-ST-systems, and a sub-ST-system of a *CRG* might not be a *CRG* of a Petri net.

Definition 2.1.4. (sequential and concurrent conflict)

Two distinct transitions, a and b, of $\mathcal{N}$ are in:

- *sequential conflict* at a marking M whenever $M[a\rangle$ and $M[b\rangle$, but $M[ab\rangle$ does not hold;
- *concurrent conflict* at a marking M whenever $M[a\rangle$ and $M[b\rangle$, but $M[\{a,b\}\rangle$ does not hold. $\blacksquare$

Note that sequential conflict implies concurrent conflict, but reverse implication does not hold; e.g., transitions a and b in Figure 2.2 are in concurrent conflict but not in a sequential one.

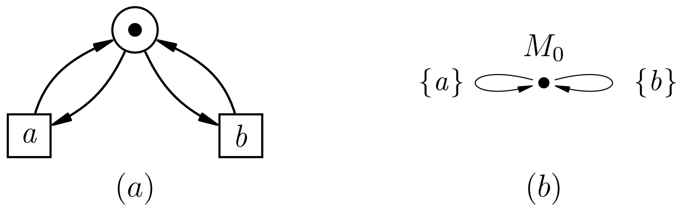


Figure 2.2: A safe persistent PT-net $\mathcal{N}$ (a); and its concurrent reachability graph $CRG(\mathcal{N})$ (b).

Definition 2.1.5. (safe net)

$\mathcal{N}$ is *safe* if, for all reachable markings $M \in [M_0\rangle$ and places $p \in P$, $M(p) \leq 1$. $\blacksquare$

Note that reachable markings of safe nets can be treated as sets of places. Moreover, being a safe PT-net does not depend on the chosen semantics, i.e., the sequential semantics where only singleton steps are executed, or the full step semantics.

A step α of a PT-net $\mathcal{N}$ is:

- *active* if there is a reachable marking of $\mathcal{N}$ enabling it;
- *positive* if $W(\alpha, p) \geq W(p, \alpha)$, for every $p \in P$;
- *disconnected* if $({}^\bullet a \cup a^\bullet) \cap ({}^\bullet b \cup b^\bullet) = \varnothing$, for all distinct $a, b \in \alpha$;[3]
- *lying on self-loops* if $W(p, a) = W(a, p)$, for all $a \in \alpha$ and $p \in P$.

Clearly, the empty step is lying on self-loops, and if α is lying on self-loops then it is also positive.

[3]The notion of disconnectedness as defined here is not related to the standard graph-theoretic notion of disconnectedness (i.e., that a graph is disconnected if there is a pair of vertices without a connecting path).

Fact 2.1.1. (marking monotonicity)

If $M[\alpha\rangle$ and $M'(p) \geq M(p)$, for all $p \in P$, then $M'[\alpha\rangle$. ■

Fact 2.1.2. (step monotonicity)

If $M[\alpha\rangle$ and $\beta \subseteq \alpha$, then $M[\beta(\alpha \setminus \beta)\rangle$. ■

Fact 2.1.3. (disconnectedness)

A step α is enabled at a reachable marking M of a safe PT-net iff α is disconnected and consists of transitions enabled at M. ■

Remark 2.1.2. The above facts are very intuitive and look trivial. However, they have a crucial importance in the discussion on persistence in Chapter 4.

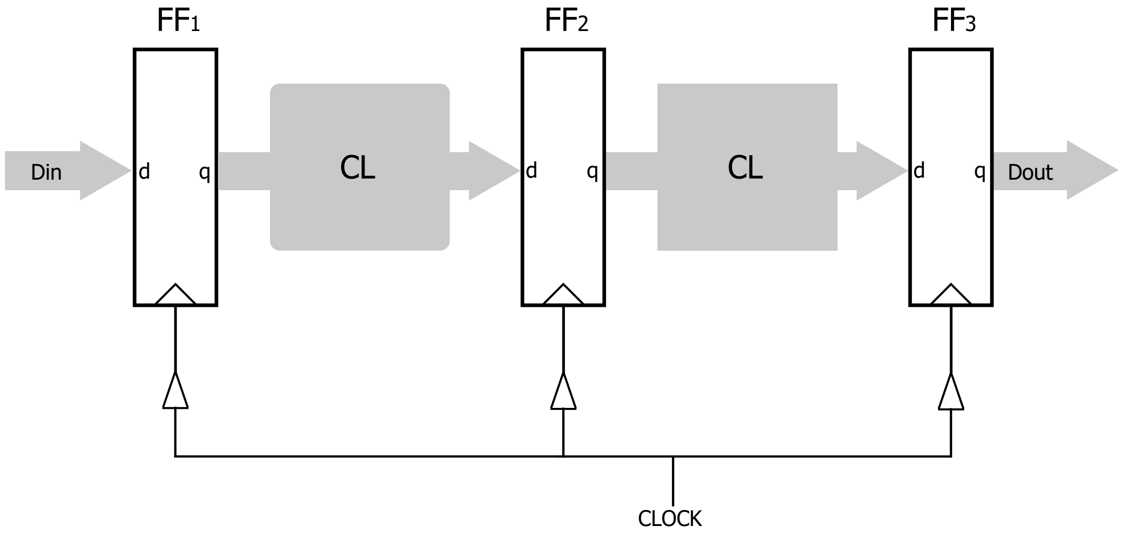
2.2 Bundled-Data Asynchronous Pipelines

In this section, the principles of bundled-data asynchronous pipelines is discussed and a summary of the relative-timing (RT) based CAD flow is provided. The concepts presented in this section are based on literature presented in [45, 46, 55, 32, 1].

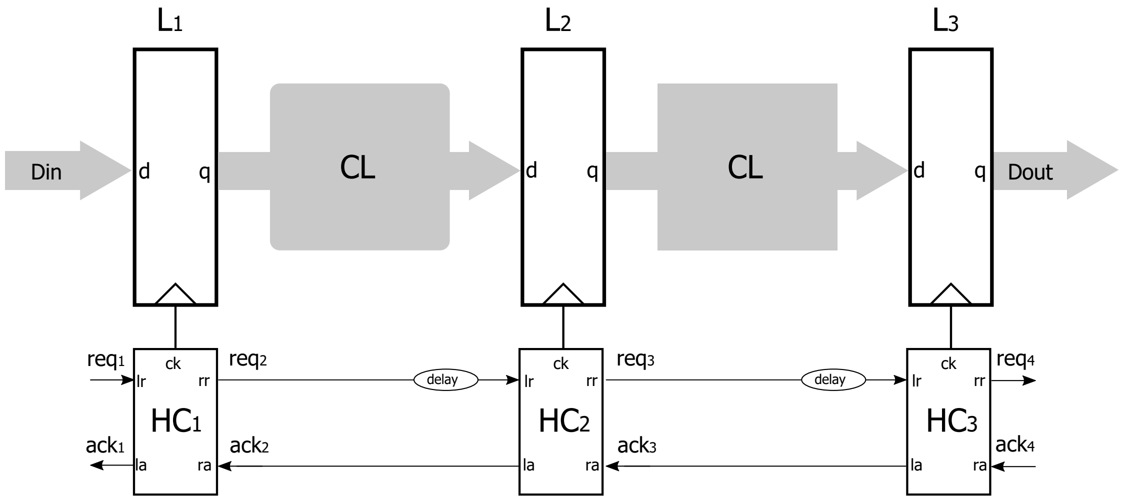
2.2.1 Synchronous vs. Asynchronous Pipelines

Pipelining is a principle element of high-performance digital design catering both synchronous as well as asynchronous systems. The fundamental difference between synchronous and asynchronous pipelines lies in the communication channel that enables data items to move from one pipeline stage to the next. Figure 2.3 illustrates this with a simple example showing a traditional synchronous linear pipeline and an asynchronous linear pipeline. In synchronous pipelines, a global clock signal handles the communication between pipeline stages in a lock step fashion. A critical requirement of such a globally clocked system is that each stage's worst-case data path delay should be lower than the fixed clock period. Hence, traditional synchronous systems feature a rigid communication scheme where all pipeline stages operate at a fixed clock frequency.

In contrast, asynchronous pipelines do not have a global clock and achieve communication between pipeline stages by handshaking between neighbouring registers. Figure 2.3b



(a) Synchronous pipeline



(b) Bundled-data asynchronous pipeline

Figure 2.3: Synchronous versus asynchronous pipelines

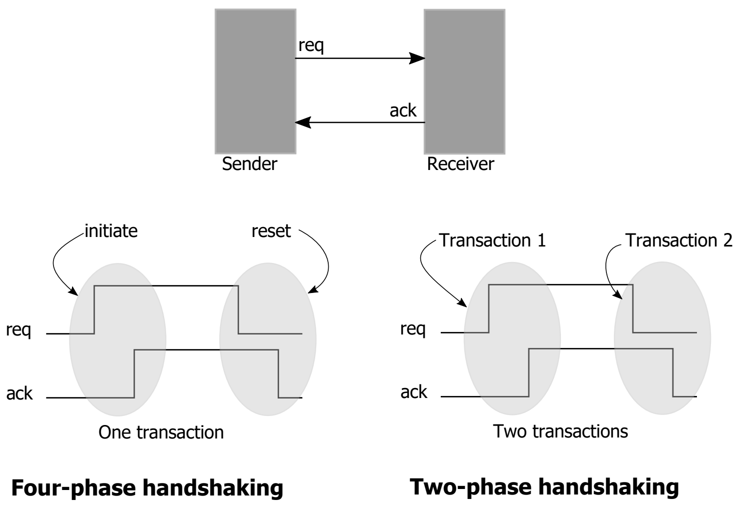


Figure 2.4: Four-phase and two-phase asynchronous signalling protocols

depicts a bundled-data asynchronous pipeline where the global clock signal is replaced by handshake control (*HC*) logic blocks that implement fine-grained local locking at registers. A request-acknowledge based handshake protocol consisting of handshake signals, request (*req)* and acknowledge *(ack*), advance data items on a per-stage basis. Delay elements, implemented as inverter chains, matching each stage's critical path delay, are inserted on *req* signals to ensure data stability and validity before reaching the next pipeline stage. Thus, in comparison to synchronous design, bundled-data handshaking can be viewed as system of multiple fine-grained local clocks whose period is determined by actual critical path delay of individual pipeline stages.

2.2.2 Bundled-data protocols

The scheme of using a synchronous-style data path coupled with a bundling signal (*req*) that enforces a worst-case delay for valid data being received stable at a neighbouring pipeline stage is what defines the class of bundled-data asynchronous circuits [46]. Bundled-data handshaking can follow two basic asynchronous signalling protocols, namely four-phase and two-phase. Figure 2.4 illustrates these two communication protocols. In the case of four-phase handshaking, *req* and *ack* are initially deasserted low. A transaction is initiated with a rising *req* from the sender. The receiver responds with a rising *ack* acknowledging the transaction. These two signals are then deasserted low to allow for

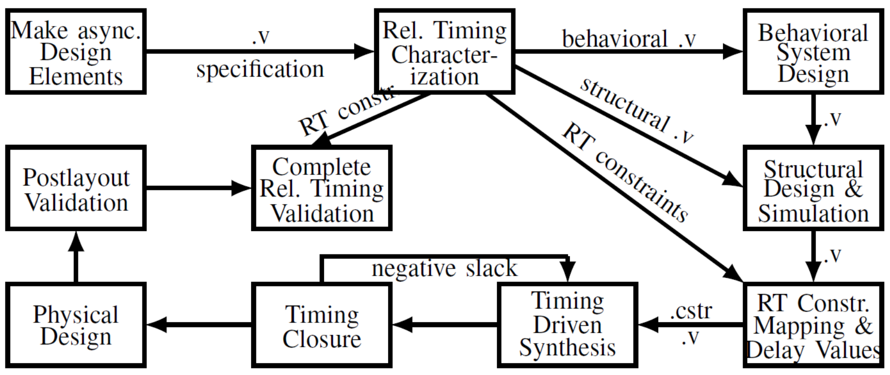


Figure 2.5: Relative-timing based asynchronous design flow (reproduced from [1])

the next transaction. The four-phase handshake protocol is also known as return-to-zero (RTZ) signalling or level signalling. In the case of two-phase handshaking, a single toggle of *req* initiates a transaction which is followed by a single toggle of *ack* acknowledging the transaction. The two-phase handshake protocol is also known as non-return-to-zero (NRZ) signalling or transition signalling. This protocol only requires a single round-trip communication per transaction as compared to double round-trip communications in four-phase signalling. Between the two protocols, the four-phase protocol is widely used due to simpler hardware design.

2.2.3 RT-based Bundled-Data Circuit Synthesis

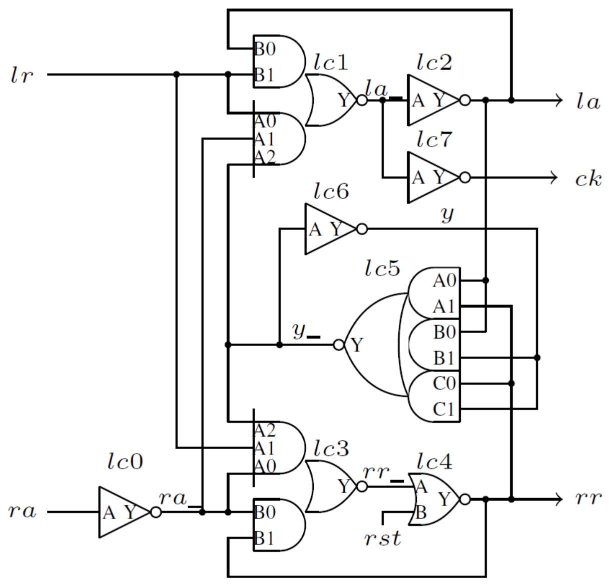
Relative timing (RT) is a method of modelling and controlling the firing order of two events based on logic path delays. This method can accurately capture, model and validate heterogeneous timing behaviour of synchronous as well as asynchronous circuits [32]. RT constraints are enforced during circuit synthesis so that hazardous states are filtered out. RT constraints consist of a common timing reference, called as point-of-divergence (*pod*), and a pair of events ordered in time, called as point-of -convergence (*poc*) [1]. Path-based RT constraints specify the order of arrival of two paths from a common causal *pod* to ordered events *poc*. Such RT constraints are represented as $pod \mapsto poc_0 + m \prec poc_1$ where poc_1 is ordered to fire after poc_0 occurs in time with margin m.[4] Let us consider the bundled-data linear pipeline in Figure 2.3b to illustrate this further. The delay element between stages L_1 and L_2 can be sized by enforcing the following RT constraint:

[4]$\mapsto$ is an arrow symbol used here to define the function. $\prec$ is a relation operator denoting precedence.

$$req_1 \uparrow \mapsto L_2/d + margin \prec L_2/ck \uparrow$$

The RT-based design flow enables rapid development of asynchronous designs by providing a set of characterised asynchronous templates that can be easily integrated with synchronous CAD tools [32]. These templates can be inserted in designs with supporting clocked CAD tool constraints for synthesis, place and route, timing driven sizing, optimisation and validation. A version of RT asynchronous design flow taken from [1] is shown in Figure 2.5.

The RT design flow enables the adoption of bundled-data asynchronous circuits in the traditional synchronous design flow with little expertise in asynchronous design. First, the bundled-data design is partitioned into data path logic and control logic. The data path is synthesised using normal synchronous CAD synthesis procedures. Handshake clocking is, then, implemented by replacing the global clock with the *HC* logic blocks as shown in Section 2.2.1. Characterised asynchronous design elements of the HC block are used to implement the control logic. Figure 2.6a shows the circuit implementation of an *HC* block based on four-phase handshaking as characterised in [32]. The HC circuit's verilog implementation in our in-house 90nm Faraday library is shown in Figure 2.6b. Next, RT constraints are specified based on the timing constraints elaborated in [32]. These constraints are integrated into clocked CAD tool flow by specifying them as **sdc** constraints which are supported by commercial tools. For example, RT constraints represented by *set_ max_ delay* and *set_ min_ delay* commands are used to perform timing driven synthesis. These constraints are responsible for constraining the data path logic and adding delay elements on the control path. Finally, the RT constrained architecture is passed through usual clocked CAD tools and flows. In this fashion, bundled-data asynchronous circuits can be synthesised by leveraging clocked CAD tool flow.



(a) *HC* Circuit Implementation (reproduced from [1])

```
module handshake_ctl(
        input wire lr, ra, reset,
        output wire rr, la, ck
        );

 wire ra_,la_,rr_,y_,y;

 INVX1      lc0  ( .I(ra), .O(ra_) );
 AOI23X1    lc1  ( .B1(lr), .B2(ra_), .B3(y_), .A1(lr), .A2(la), .O(la_) );
 INVX1      lc2  ( .I(la_), .O(la) );
 AOI23X1    lc3  ( .B1(ra_), .B2(lr), .B3(y_), .A1(ra_), .A2(rr), .O(rr_) );
 NR2X1      lc4  ( .I1(rr_), .I2(reset), .O(rr) );
 MAOI222X1 lc5  ( .A1(la), .B1(y), .C1(rr), .O(y_) );
 INVX1      lc6  ( .I(y_), .O(y) );
 INVX1      lc7  ( .I(la_), .O(ck) );

endmodule
```

(b) Verilog Implementation of *HC* in 90nm Faraday library

Figure 2.6: Characterised handshake control element for bundled-data circuits

Chapter 3

Physical Partitioning and its Limitations

One of the first partitioning techniques for mixed synchronous-asynchronous digital designs, and a popular one, was introduced by Chapiro in his doctoral thesis titled 'Globally-asynchronous locally-synchronous systems' [44]. Chapiro's technique abbreviated as GALS was designed to exploit the advantages of asynchronous design while maximally reusing the products of the synchronous design flow. This is achieved by partitioning a system into synchronous islands that inter-communicate by asynchronous handshaking. Over the last couple of decades, GALS system design has been widely researched as an alternative to synchronous SoC designs mainly as a physical partitioning strategy. However, this technique has gained mixed reviews from the research community with no clear indication of its future in SoC design. In this chapter, we are motivated to investigate the GALS physical partitioning technique and review its applicability to SoC design.

3.1 Introduction

Globally asynchronous locally synchronous (GALS) design [44] is a promising approach that can reduce the design complexity of modern SoCs and improve energy efficiency by incorporating heterogeneity. This design principle has been mainly implemented as a physical partitioning strategy for SoC design where top-level hierarchical blocks of varying clock speeds such as heterogeneous IP cores are partitioned into distinct clock domains

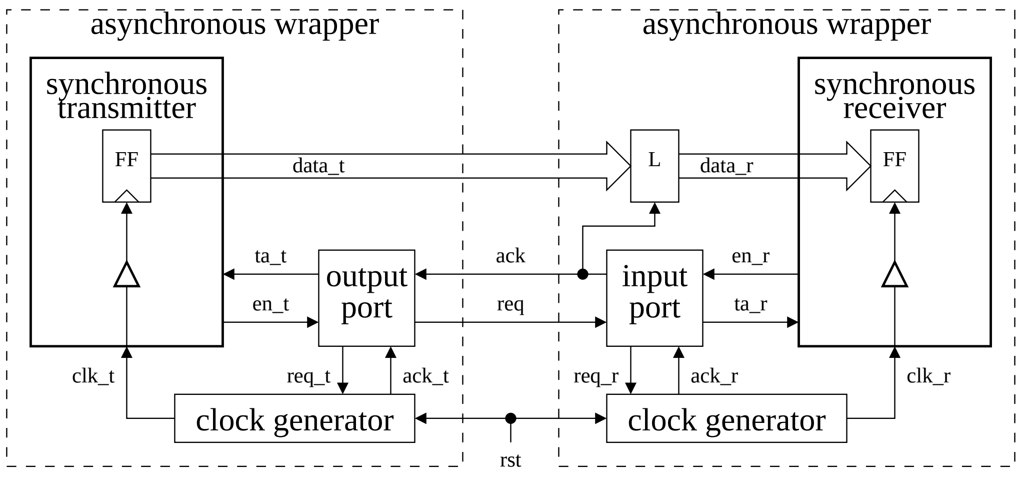


Figure 3.1: Typical GALS design block.

known as synchronous islands. Clock distribution is handled locally in synchronous islands and communication across clock domain boundaries is achieved by asynchronous design principles. A typical GALS system is shown in Figure 3.1. Each synchronous island has its own local clock which would be "paused" during data transfer and resumed after acknowledging the transfer. Synchronous islands are enclosed within asynchronous wrappers which handle asynchronous handshake control and local clock management.

GALS design can provide genuine scope for power savings, performance improvement and energy efficiency compared to its synchronous counterpart owing to its scheme of multiple clock domains. One of the main benefits is a simplified clock distribution network (CDN) offering significant power savings and performance improvement. GALS methodology, however, induces certain design overheads such as latency penalty between clock domains which would impose restrictions on the partitioning granularity. There are several research works such as [10, 56, 57, 58, 35, 12, 36, 37, 41, 15] that demonstrate the potential of GALS design but the benefits reported are not consistent and in some cases contradictory. Most of them follow an assemble-and-validate design style and lack system analysis procedures to justify GALS partitioning scenarios.

The authors in [11, 58] make a good effort to judge the applicability of GALS methodology by comparing the power and performance figures against a fully synchronous design. They showed that GALS performance penalty is around 6.3% when using pausable clocks with five clock domains. The main reason for system throughput reduction is due to frequent inter-island communications and the presence of communication feedback loops. The observations were made from a processor case study and hence, the same effect would not hold true for application specific designs. For example, parallel pipelined data path implementations of complex operations such as FFT [1] or JPEG Encoder [41] will have fewer such loops and would show much lower performance penalty with GALS. The study also demonstrated 10% of average power savings across several GALS implementations. However, the effect of process variations was not accounted in the study. This effect is quite important for comparing digital design implementations, especially for newer semiconductor process technologies, because a growing portion of clock period budget is allocated for variability which directly has an impact on the CDN power and maximum clock frequency per island.

Yu and Bass studied the impact of CDN design on the peak performance of processor cores considering both GALS and globally synchronous design styles [41]. The GALS processor being of a constant size has the highest peak performance compared to the larger synchronous processor array due to lower clock skew. Further decrease in peak performance of the synchronous processor is seen due to additional skew penalty from process and environment variations, and a performance figure of 76.8% to the GALS case is reported for an array of 121 processors. Their main contribution was the performance analysis of several applications on GALS and synchronous processor array architectures. Each application was mapped onto the processor cores and performance was compared. Performance reduction from GALS was in the range of 0% to 2.3% and power savings of upto 40% was demonstrated. Their method of analysis is for a GALS chip multiprocessor with FIFO style of synchronisation wherein computations are localised within each processor's clock domain. Inter-island communications are less frequent thus explaining the low performance degradation figure. Similar performance results are expected from GALS-based application specific designs.

Hemani *et al* suggested that significant clock power reduction of about 70% can be obtained by dividing a system into N equally sized islands [10]. However, till date, there are only a few of GALS implementations that report actual power performance measurements of GALS fabricated chips. In [36], a GALS baseband processor was designed and fabricated on 250 nm process, and compared with its synchronous implementation. Power savings of only 1% were reported when about 17% was to be expected. The authors in [16] evaluated GALS versus synchronous implementation of an OFDM baseband transmitter in 40nm CMOS technology. 20% power savings in the clock power was reported with a total of 6% power reduction in the overall chip power. Furthermore, the number of levels in the clock tree dropped from 27 to less than 10 in the GALS implementation. It was reported that the design was partitioned into six islands based on functional and physical criteria. However in both the implementations, the choice of partitioning granularity and boundaries is unclear.

The applicability of GALS methodology to modern SoC design will depend on several factors such as system dataflow, design size and process technology. One of the main benefits of GALS is a CDN which consumes lower power and delivers a low-skew clock. However, these gains will have to be weighed against the penalties introduced due to the GALS synchronisation protocol in order to determine overall system improvement. A calculated system level analysis is hence required before choosing to GALSify a system and many studies lack this. In this chapter, we discuss the parameters and metrics required to conduct such a trade-off analysis.

The chapter is organised as follows. Section 3.2 discusses the analysis methodology and simulation framework for analysing GALS chip partitioning. Section 3.3 discusses the results obtained from the physical partitioning simulation. Section 3.4 concludes this chapter and presents scope for future work.

3.2 Methodology

3.2.1 Characterising a GALS Design

Several design parameters will influence the decision towards a GALS architecture. This section discusses some of the key factors that govern an effective GALS implementation.

3.2.1.1 Clock Distribution Network

The clock distribution network is a very important component of a synchronous design governing its functioning and performance. Any violation to the clock timing specific-

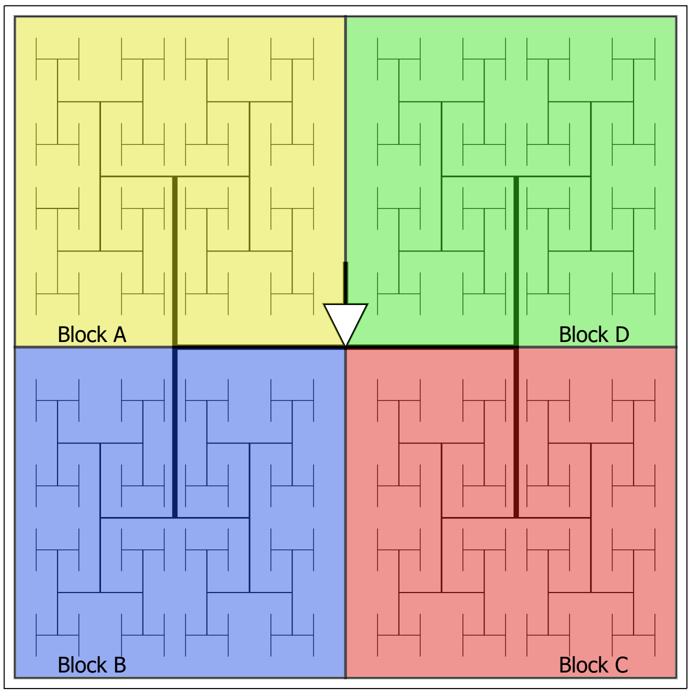


Figure 3.2: Global CDN and GALS partitioning.

ations can affect logical operation of the circuit and lead to design functional failure. Moreover, the CDN can consume up to about 40% of the chip dynamic power in high performance designs [59] and hence any optimisation reducing the clock tree power is considered a significant design improvement. For example, the authors in [60] demonstrated overall chip power savings of 25% by optimising the CDN using a combination of energy recovery and clock gating techniques. CDN design depends on the chip floorplan size, total load to distribute the clock signal and clock design constraints such as buffer/load rise time and skew budgets. For instance, greater area to distribute the clock requires stronger buffers to deliver clock signal due to larger interconnect capacitance.

GALS design has been shown to reduce the clock switching power of larger designs by partitioning into smaller CDNs [10]. Figure 3.2 shows a clock tree distribution network of a generic design and a trivial case of GALS partitioning. It can be visualised that clock power is reduced due to removal of the global clock tree. There will, however, be an optimal number of partitions for any design to benefit from such CDN power savings [57]. This effect is seen due to additional power consumption from GALS architecture overheads: wrapper circuit and local clock generator. In this study, we extend the analysis to determine the CDN power savings for a design over different process technologies.

3.2.1.2 Sensitivity to Process Variations

Clock signals act as control signals that guide data movement between registers in a sequential data path. The timing of these signals has to be precise for proper system operation. Any delay uncertainty can cause system failure if the CDN design has not accounted these effects.

Each element of a clock path is sensitive to process, environmental and geometric variations which lead to different clock arrival times at the sinks of the CDN. This temporal difference is simply the clock skew that has to be characterised by the designer and accounted for, which if not, would violate the setup and hold time constraints required for safe data latching. Timing violations are avoided by either reducing the delay uncertainty of the CDN or relaxing the timing constraints [59]. The latter involves accommodating a larger clock skew budget by increasing the clock period of the design at the cost of reduced system performance. Delay uncertainty of the CDN is reduced by desensitising the interconnects in the clock net [61] and/or increasing the clock buffer sizes [62]. This, however, comes at cost of higher power consumption.

A typical CDN consists of a tree of buffers interconnected by metal wires. Every buffer and interconnect is susceptible to variability which gets accumulated from source to sink [63] giving rise to global clock skew. As technology scales, clock frequencies are increasing and variability is worsening. The resulting global clock skew will become more profound and drastically affect system performance. Skew management techniques such as active de-skewing can manage the global skew while adhering to the traditional clocking principles but at heavy cost of power and area [64]. Sometimes, clock skew is not all that bad as techniques such as time borrowing and skew scheduling employ the clock skew to enable effective timing closure. However, as modern designs are getting increasingly complex with strict timing margins, timing closure is getting increasingly difficult to achieve. GALS architecture minimises on-chip skew owing to the reduced effect of accumulated delay uncertainty coming from a smaller CDN. GALS is, thus, seen to be a promising design solution for future technologies where variability is severe.

In this chapter, we analyse the effect of process variations and clock skew on the CDN and suggest the optimum granularity of GALS blocks for a process technology node. Upadhyay et al. provided a method to approximate the optimum number of GALS partitions for a die size while optimising the CDN power [57]. However, the research did not consider the effect of variability and clock skew on the optimum number of GALS partitions. In [58], the authors make a relation between GALS partition granularity and clock skew due to variability. However, only the effect of temperature variation was considered. The clock skew minimisation feature of GALS was exploited to speed-up the design by allowing the independent clock domains to operate at their own maximum clock frequency. GALS design could potentially tune the power-performance values of a system by either increasing the system frequency or optimising the CDN power or find a sweet spot optimising both. In this research, we chose to optimise the clock buffer sizes in the CDN to fit a given skew budget and fixed clock frequency.

3.2.1.3 System Dataflow

The system dataflow structure is another important consideration that affects the quality of a GALS implementation. Dataflow indicates the flow and frequency of data communication between pipeline stages. This is important when deciding the partitioning boundaries since data transfer between GALS islands requires crossing asynchronous boundaries wherein a penalty will be paid in the form of synchronisation latency by the communication circuitry. Though increasing the number of partitions can provide significant power savings, there will be a greater cost of performance penalty incurred which could render GALS design unacceptable [11]. Maintaining a low inter-island communication probability would keep the penalty down. Hence, each design has to be carefully partitioned to find the right balance of power savings versus latency penalty.

In addition to design partitioning granularity, the behaviour of communication circuitry is also critical to GALS system performance. Basically, three flavours of synchronisation techniques exist for communicating between clock domains: synchronisers, asynchronous wrappers and asynchronous FIFOs. The two-flop synchroniser experiences multi-cycle latency which can decrease system throughput significantly [65, 66]. In case of the asynchronous wrapper, the latency penalty is the least alongside minimal throughput loss [67, 13, 68]. For high performance systems, asynchronous FIFOs are quite effective at maintaining system throughput [41, 69].

We thoroughly investigated these synchronisation techniques. The asynchronous wrapper approach works best when synchronisation between islands is infrequent [70]. However, an upper limit to the operating system clock frequency should be enforced to ensure acceptable metastability resolution [13]. In contrast, FIFOs provide good bandwidth for communication and support higher clock frequencies but its complex design makes it difficult for comparative analysis. Moreover, their application costs more area and power, comparatively, and hence would not be the best choice of synchronisation for systems with tight design budgets. Finally, standard two-flop synchronisers could cause multi-cycle latency of upto 12 clock cycles in the worst case and hence will always lag in overall system performance compared to asynchronous wrappers [66]. They, however, do not pose any clock frequency limitation and can even provide acceptable performance figures in certain design scenarios. For example, mesochronous, multi-synchronous, plesiochronous and ratiochronous systems [71, 72, 65] are a special case of GALS where strategic synchroniser design can eliminate performance penalty significantly at an added design cost.

In this research, the pausable asynchronous wrapper [73, 13, 74, 68] was chosen as it seemed best suited for analysing the impact of wrapper performance penalties with increasing partition granularity. The technique stretches the transmitter clock pulse so that sufficient timing window is available for safe data latching during asynchronous communication between islands. The wrapper circuitry consists of MUTEX elements [67], which, in fact, dictate the minimum resolution time window that should be accommodated in the event of metastability. For a mean time between failures (MTBF) of 10,000 years, which is an acceptable metric for sufficient metastability resolution, a minimum clock period restriction of $(100 \times d_{FO4})$ is incurred [68]. This restriction is not a grave concern in GALS-based SoC designs since the shortest clock period, coincidently, happens to be in the range of clock speeds synthesised using standard cell tools [13]. d_{FO4}, called the fan-out-of-4 ($FO4$) delay, is a common metric used to indicate the performance of a process technology node.

3.2.1.4 Dynamic Voltage and Frequency Scaling

The inter-island asynchronous communicating scheme of GALS allows for fine-grain voltage-frequency tuning of the islands according to design specifications of power and performance. Each island working at its optimum speed rather than on a single frequency domain provides scope for additional power savings without drastically affecting system performance. In [58], the authors showed that GALS architecture with dynamic voltage and frequency scaling (DVFS) can provide an average energy reduction of 25%-30% at an expense of 5%-7% performance penalty. Several other research works have also successfully demonstrated the potential of DVFS using the GALS approach [56, 35, 12]. The percentage of energy reduction is application-specific and depends on functional characteristics of the system and choice of partition granularity. In this research, the effects of DVFS on power-performance of GALS partitioning is left for future work.

3.2.2 Simulation Framework

To conduct realistic comparison of GALS trade-offs across different process technologies, we constructed a parametric model that simulates the physical and functional properties of a digital design.

3.2.2.1 Design Set-up and Partitioning

A generic system mimicking an application-specific SoC core synthesised under standard cell methodology is assumed in this research. The design will be GALSified using a physical partitioning approach where fixed overall floorplan size will be maintained throughout partitioning. Purely for comparative analysis, the design will be sliced into equally sized islands whilst preserving the overall aspect ratio. Figure 3.2 illustrates such a partitioning strategy where a globally clocked system is being planned for partitioning into 4 equally sized islands.

The generic design consists of $41,210$ flip-flops (FFs) which are assumed to be placed regularly on a die. The parameter $FF\,density$ estimates the floorplan size per technology node for a given number of FFs. We have chosen the $FF\,density$ based on a survey of the Moonrake chip [15, 16] and the OFDM Gigabit Baseband Processor chip [39].

Year	1999	2001	2004	2007
L_T (nm)	180	130	90	65
T_{ox} (nm)	4.5	4	3.5	3
V_{DD} (V)	1.8	1.5	1.2	0.9
V_T (V)	0.45	0.4	0.35	0.3
W (μm)	0.65	0.5	0.4	0.3
H (μm)	1.0	0.9	0.8	0.7
ρ $(\frac{m\Omega}{\square})$	50	55	60	75
Wire thickness (μm)	1.25	1.2	1.2	1.2
Dielectric ε	3.5	3.2	2.8	2.2
d_{FO4} (ps)	62.6	39.4	24.9	18.4
Clock freq (MHz)	159.7	254	402.2	544.3
Max. logic delay (ps)	5,949	3,740	2,362	1,745
Total skew budget (ps)	313	197	124	92
Process Variation skew budget (ps)	75	47	30	22
Floorplan (mm^2)	87.04	42.65	20.9	10.24
FF density (FFs/mm^2)	473	966	1972	4025
FF capacitance (fF)	7	5.5	4	2.5

Table 3.1: Process technology and design parameters

Using scaling theory [75], we have estimated the $FF\,density$ for all technology nodes. The clock frequency of the design is scaled as $(100 \times d_{FO4})^{-1}$ to account for performance improvement per process technology while satisfying the MTBF constraint discussed in Section 3.2.1.3. For simplicity of analysis, identical clock frequencies are chosen for all islands. It should be noted that this assumption of identical clock frequency provides the least amount of energy savings while GALS islands operating at their optimal frequency would provide the best case for GALS energy savings. However, our assumption is fine for the comparative analysis as we are mainly evaluating GALS benefits on the basis of physical partitioning criteria. The d_{FO4} is chosen for typical operating and typical process conditions [76].

In the next section, the CDN design of the generic SoC core is discussed. The parameters considered for modelling the generic design and its CDN are given in Table 3.1. The CMOS technology parameters have been adopted from [77], the Arizona State University PTM [78] and the International Technology Roadmap for Semiconductors (ITRS) [79].

3.2.2.2 CDN Design

In our design, we employ the buffered H-tree technique for clock distribution, a popular technique in various CDN studies [80, 81, 82]. The H-tree assumes uniform clock loads and can be geometrically scaled depending on floorplan size.

A local grid comprising between 100 FFs and 360 FFs is chosen as the range of capacitive load at every H-tree sink. This is done because spanning the H-tree to reach every single FF is not considered optimal [83]. We follow the same scheme of interconnect thickness scaling as implemented in [81]. The interconnect closest to the sink is given minimum thickness. The interconnect thickness is then multiplied by a factor of 2 for every third wire segment. This is done to minimise the effect of process variations by desensitising the wires closest to the clock source [61]. A clock tree buffer is placed at every junction of the H-tree. The capacitances of the CDN buffers and interconnects are calculated using models and approximation formulae given in [84] and [81]. These measurements will be required during buffer sizing and clock tree delay estimation.

Clock skew in application-specific integrated circuits (ASICs) are typically in the range of 5% - 10% of their clock periods [76]. For our design, we set the clock skew budget at 5% of the clock period. 40% of this budget is allowed for local skew and the rest 60% is allotted for global skew. Local skew arises from geometric variation of FF placement across local CDN sink grids and from process variation of grid interconnects. The global skew comprises of delay variations due to environmental factors in addition to process variation of CDN interconnects and CDN buffers. Skew due to process variation is accounted to take up 40% of the global skew budget. The remaining 60% of the budget is for environmental variations. This assumption is based on the skew budget generalisation adopted in [41].

3.2.2.3 Variation Model

In this section, we discuss the skew modelling technique used for estimating the clock skew due to transistor length and interconnect variability in the GALS CDNs. The standard deviation values for the process parameters used in our study, based on a normal distribution, have been adopted from [77] and are listed in Table 3.2. Our goal is to model the effect of process variations and resulting clock skew on the CDN power for

Parameter	180nm	130nm	90nm	65nm
L_{eff} (nm)	60	45	40	33
T_{ox} (nm)	0.36	0.39	0.42	0.48
V_T (mV)	45	40	40	40
W (μ)	0.17	0.14	0.12	0.1
H (μ)	0.3	0.27	0.27	0.25
ρ $(\frac{m\Omega}{\sqcup})$	12	15	19	25

Table 3.2: Technology parameter $3 \cdot \sigma$ variations

varying GALS partition granularity in different process technologies. Skew arising from environmental variations will be modelled in future extension of this work.

The clock skew algorithm given by Jiang et al. in [84] is used in our model. Other statistical techniques such as [85] can also be used. First, the variation in individual CDN branch segments are computed based on branch delay estimations derived in Section 3.2.2.2. The clock skew algorithm then processes every variation calculation, cumulatively, along clock paths and statistically estimates the clock skew for that particular CDN design. For a balanced H-tree CDN, the expected clock skew $E(\chi)$ and skew variance $D(\chi)$ is given by Eq. 3.1 and Eq. 3.2, respectively [84], where N is the number of hierarchical tree levels in the CDN and d_i, $i = 1$, ..., N is the propagation delay in branch i of the clock tree. $D(d_i)$ represents the variance of d_i, and ρ is the correlation coefficient metric which is recursively evaluated according to the methodology provided in [84].

$$E(\chi) = \frac{2}{\sqrt{\pi}} \sum_{i=1}^{N} \sqrt{\sum_{k=1}^{i} \left(\frac{\pi - 1}{\pi}\right)^{k-1} \cdot D\left(d_{N-i+k}\right)} \tag{3.1}$$

$$D(\chi) = 2 \cdot (1 - \rho) \sum_{i=1}^{N} \left(\frac{\pi - 1}{\pi}\right)^{i} \cdot D\left(d_i\right) \tag{3.2}$$

Eq. 3.1 and Eq. 3.2 clearly show the nature of growing clock skew with increase in H-tree size as an effect of accumulated variations along clock paths.

3.2.2.4 Clock Tree Synthesis and Optimisation

Here, we describe the clock tree synthesis (CTS) algorithm used in our simulation framework. The CDN is individually built for the synchronous chip and its GALS partitions up to a maximum of 16 islands. Taking Figure 3.2 as an example, each of the four partitioned islands will have their own local clock network and the global CDN, indicated by thick lines, will be removed. The four isolated H-trees now distribute local clocks to 64 sinks each as opposed to the global CDN which managed 256 sinks.

First, the CDN skeleton is built based on buffered H-tree arrangement described in Section 3.2.2.2. Next, the clock tree delay in each branch of the CDN is calculated based on the derived physical design parameters. The CDN buffers are then sized as per a constant ratio λ, the buffer sizing factor, such that transition time at every branch of the CDN is the same [86]. This is an important requirement to ensure effective clock signal delivery. There will, however, be a maximum bound for the output transition time of the clock signal, also known as the clock slew rate, in order to meet the hold time and clock-to-Q delay constraints of FFs [83]. Clock slew for the H-tree is estimated using the PERI method [87, 88] and a budget of 15% of the clock period is set.

Next, CDN buffers are optimised to handle sensitivity from process variations. The skew due to process variations is estimated in Section 3.2.2.3. The CTS algorithm optimises the CDN buffers, recursively, until both skew and slew targets are met. Smaller design partitions will experience skew lower than the skew budget, if the original CDN buffer sizing of the unpartitioned synchronous chip was retained. This is because variability accumulation is lower across fewer clock paths as discussed in Sec. 3.2.2.3. The reduced skew margin in smaller designs can be used to reduce CDN buffer driving strength. This information is, finally, used by the CTS algorithm to decrease local buffer sizes and optimise the CDN accordingly, which in turn reduces CDN power consumption. Experimental results in [16] demonstrate the same effect wherein CDN power reduction was due to locally optimised clock trees. In this fashion, the CDN is modelled and optimised for various physical partition sizes and CDN arrangements.

The lower clock skew in smaller partitions can also be exploited for localised performance improvement by retaining the original CDN buffer sizing. This is very applicable for large high performance designs when problems in timing closure and CTS limit the global clock speed after Standard Cell Place and Route design flow. This effect, requiring a different set of design targets, is not demonstrated in this chapter and is left for future work.

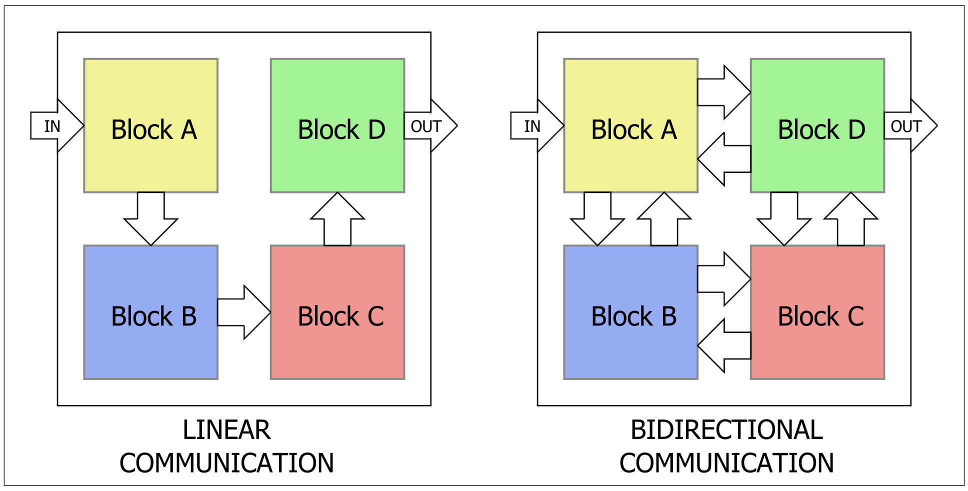


Figure 3.3: Linear versus bidirectional block communication.

3.2.2.5 System Dataflow Model

In this section, we evaluate the system dataflow metrics that cause design penalties from GALSification. In an application-specific SoC core, the data flow is tuned to perform a specific application. Communication between blocks can be evaluated beforehand and utilised for dataflow optimisation. Unlike general purpose processor architectures where blocks communicate with each other every clock cycle, data computations in SoC cores are highly localised within partitions and hence inter-block communication is infrequent. We term such blocks, which inter-communicate in bursts and/or once in several hundred clock cycles, as *localised computation blocks.*

In our analysis, we assume that our physical partitions/islands coincide with the dataflow blocks. We introduce the *Single Handshake Communication Probability* ($Prob_{comm}$) metric which will model the inter-island communication frequency. This metric denotes the average inter-island single handshake communication frequency amongst all partitions in the GALS implementation. In other words, it can be assumed that every GALS island will communicate with its neighbouring island at a fixed rate $Prob_{comm}$. Single handshake simply means the flow of a single data token from the output port of a transmitter block to the input port of a receiver block.

The next step is to model the overall impact of inter-island communications. As the number of islands increase, the number of asynchronous wrappers or communication links also increase. We generalise the inter-island communication to follow either a linear or a bidirectional dataflow communication path as shown in Figure 3.3. The total number of

GALS single wrapper links N_W for linear and bidirectional communication is determined by Eq. 3.3 and Eq. 3.4 respectively [57], where B denotes the total number of GALS partitions, L is the maximum number of localised computation blocks in the design and K is a constant determined from L [57]. In this analysis, a single wrapper link consists of a output port of the transmitter GALS block and a input port of the receiver GALS block.

$$N_W = (B - 1) \tag{3.3}$$

$$N_W = \left(\frac{2.L.\sqrt{B}}{2^K} - 4\sqrt{L} \right) \tag{3.4}$$

We can now estimate the total number of wrappers that are actually involved in inter-island communication per unit of time. Given the average inter-island communication frequency and the total number of GALS wrapper links, the number of active wrappers per clock cycle N_{Wact} can be determined as indicated in Eq. 3.5. This metric is used in determining the wrapper latency and power overheads incurred from GALS partitioning.

$$N_{Wact} = Prob_{comm}.N_W \tag{3.5}$$

In this fashion, we have divided a generic system dataflow into localised computation blocks associated with a fixed number of wrappers that inter-communicate at an average frequency $Prob_{comm}$.

3.2.2.6 Synchronisation Model

In this section, we model the impact of GALS inter-island synchronisation to the latency of the system. Our choice of synchronisation between islands is the pausable wrapper approach owing to its popularity in GALS implementation [58, 13, 68]. Pausable synchronisation can transmit data every clock cycle by stretching the clock pulse so that sufficient timing window is available for safe data latching during asynchronous communication between islands. The pausable wrapper model presented in [68] is used as the reference wrapper in our simulations. This type of wrapper costs a maximum data synchronisation latency (Lat_W) of one receiver's clock cycle. The exact data synchronisation latency will depend on the arrival of data synchronisation request with reference to positive edge of the receiver's clock. In our simulations, we consider the worst case synchronisation latency for comparative analysis.

3.2.3 Trade-off Analysis

3.2.3.1 System Latency

GALS wrappers cost synchronisation latency which can severely affect overall system latency if the chip was not carefully partitioned. Here, we model the latency penalty paid over the course of the system operation and evaluate the quality of GALS partitioning by comparing system latency with the synchronous chip.

System latency is simply the execution time of the system to process a fixed number of input samples. Let us say that system latency of a synchronous chip T_{sync} is the number of clock cycles taken to process X input samples and produce a result. For a GALS chip, if T_{WGALS} is the total synchronisation latency paid by inter-island wrapper communication, then the overall system latency can be given by Eq. 3.6.

$$T_{GALS} = T_{sync} + T_{WGALS} \tag{3.6}$$

To determine T_{WGALS}, we first calculate the total data synchronisation latency penalty (Lat_{WGALS}) paid per clock cycle for the entire GALS system operation. This parameter, given by Eq. 3.7, is derived using the models discussed in Section 3.2.2.5 and Section 3.2.2.6.

$$Lat_{WGALS} = N_{Wact}.Lat_W \tag{3.7}$$

Then, the total wrapper synchronisation latency in clock cycles T_{WGALS} can be calculated as

$$T_{WGALS} = Lat_{WGALS}.T_{sync} \tag{3.8}$$

Finally, Eq. 3.6 becomes

$$T_{GALS} = (1 + N_{Wact}.Lat_W).T_{sync} \tag{3.9}$$

3.2.3.2 Power Estimation

CDN power savings is inherent to a GALS implementation. However, due to the overheads from GALS circuitry, power savings will reduce or power consumption may even increase past a certain number of islands. Therefore, the total power estimation of the CDN, local clock generator and wrapper circuitry are required to evaluate the overall power benefit from GALS.

CDN power comprises of the power consumed by the wire capacitance, clock buffer capacitance and the total FF load and is given by Eq. 3.10.

$$P_{CDN} = (N_{Buff}.C_{Buff} + C_{TotWire} + N_{FF}.C_{FF})f_c.V_{DD}^2 \tag{3.10}$$

where N_{Buff} is the number of equivalent minimum sized inverters as clock buffers, C_{Buff} is the capacitance of the each minimum sized inverter, $C_{TotWire}$ is the total wire capacitance in the H-tree, N_{FF} is the number of FF loads, C_{FF} is the capacitance per FF, f_c is the system clock frequency and V_{DD} is the switching voltage.

The local clock generator in our chosen reference wrapper [68] employs a tunable delay line to provide flexible on-chip clock oscillations. This methodology is well-known in the GALS research community [34, 89, 67, 74, 70]. The easiest way to represent such a system is a ring oscillator which is simply an odd number of inverters connected in a circular fashion [57]. In this way, the power consumed by a local clock generator in the GALS chip can be given by Eq. 3.11.

$$P_{clkgen} = N_{inv}.C_{inv}.f_c.V_{DD}^2 \tag{3.11}$$

where N_{inv} is the number of inverters in the ring oscillator and C_{inv} is the capacitance of each inverter in the oscillator circuit.

The total GALS wrapper power depends on the circuit specification of communication

Process Technology	180nm	130nm	90nm	65nm
Wrapper capacitance (pF)	2.41	1.69	1.18	0.826
Wrapper power P_W (mW)	1.2	0.72	0.68	0.36

Table 3.3: GALS wrapper specification

interface, namely, the number of input and output ports, the number of synchronisation latches and flow of data transfer. In our analysis, we refer to the 180nm GALS wrapper characterised in [57] as our reference pausable wrapper and scale the single wrapper power reading P_W for other technology nodes. P_W is calculated with the worst case assumption that each wrapper operates every clock cycle at system clock frequency f_c. Table 3.3 lists the power parameters of the wrapper employed in our analysis. By estimating the total number of GALS wrappers as given in Section 3.2.2.5 , the total GALS wrapper power consumption is given by Eq. 3.12.

$$P_{WGALS} = N_{Wact}.P_W \tag{3.12}$$

Total power of the GALS chip is given as

$$P_{GALS} = \sum_{b \epsilon B} P_{CDN}(b) + \sum_{b \epsilon B} P_{clkgen}(b) + P_{WGALS} \tag{3.13}$$

where B is the total number of GALS partitions, b represents a particular GALS partition, $P_{CDN}(b)$ denotes the power consumed by the CDN of that partition and $P_{clkgen}(b)$ denotes the power consumed by the local clock generator of partition instance, b.

3.2.3.3 Energy Efficiency

GALS design reduces system power but at the cost of synchronisation latency penalty. A good indicator on the overall benefit of GALSification would be to compare the energy consumption between GALS and the synchronous chip. Relative energy efficiency (REE) is a good metric that can be used to evaluate the quality of the GALS chip with its synchronous counterpart by comparing their energy consumption. Total energy consumption is simply calculated by multiplying the system latency with chip power. REE is given by

Eq. 3.14 and Eq. 3.15.

$$REE = \frac{E_{sync}}{E_{GALS}} \tag{3.14}$$

$$REE = \frac{P_{CDN}(sync).T_{sync}}{P_{GALS}.T_{GALS}} \tag{3.15}$$

where E_{sync} and E_{GALS} are the energy consumption estimates of the fully synchronous chip and GALS partitioned chip, respectively, when X input samples are processed and the required result is obtained. $P_{CDN}(sync)$ represents the power consumed by the CDN of the fully synchronous chip.

Higher REE implies an energy efficient GALS implementation, either being faster or more power efficient. Lower REE means that the penalties paid by GALS exceed any benefits indicating no overall benefit from GALSification. Substituting formulae derived in Section 3.2.3.1 and Section 3.2.3.2, Eq. 3.15 becomes

$$REE = \frac{P_{CDN}(sync)}{\left(\sum_{b \epsilon B} P_{CDN}(b) + \sum_{b \epsilon B} P_{clkgen}(b) + N_{Wact}.P_W\right)(1 + N_{Wact}.Lat_W)} \tag{3.16}$$

3.2.4 Tool Description

The tool has been wholly implemented in MATLAB. Using the parametric model, CTS and statistical clock skew algorithms, dataflow and synchronisation models and energy estimation formulae, we have a framework that realistically simulates the benefits versus penalties of GALS partitioning. The tool can easily be updated to reflect newer process technology nodes and different chip geometries. To simplify our analysis, we divided the chip into equally-sized islands and generalised inter-block communication. However, this would not be the case in real-life SoC applications. Since the tool employs a parametric model, custom floorplan partitioning and multiple inter-island communication probabilities can be assigned, and targeted power-performance measurements can be generated. This tool would provide an intuitive platform for a designer acting as a guide towards effective GALS implementation. Appendix A provides the main source code of the simulation tool.

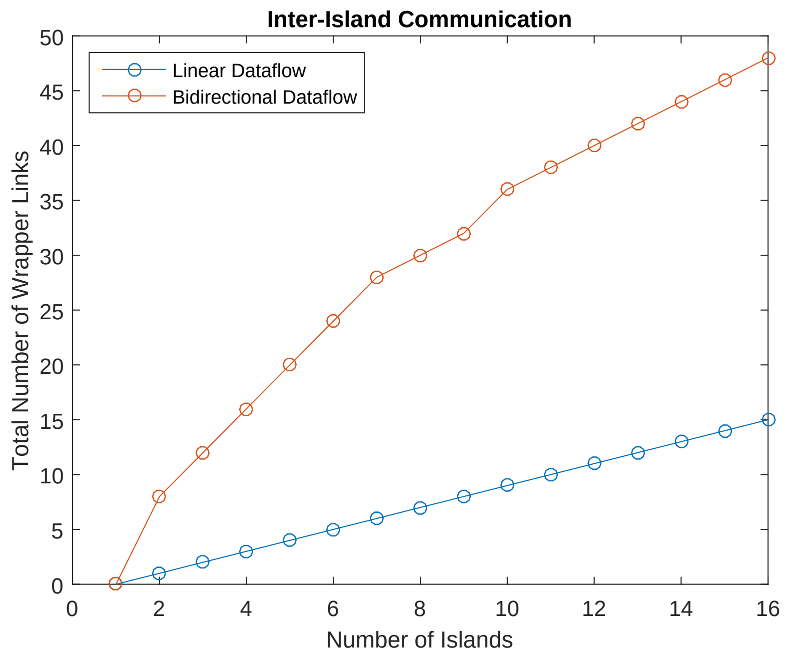
3.3 Results

3.3.1 GALS Design Impact to System Latency

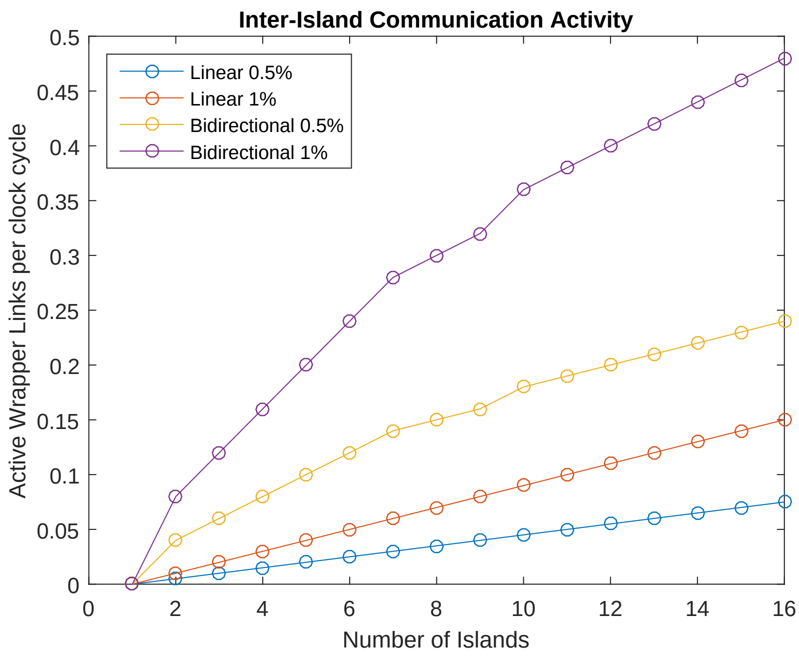
Latency penalties of GALS are estimated from predicting the number of synchronisation events occurring between islands. As a design is subject to finer levels of partitioning granularity, the frequency of communication between islands increase, causing latency penalties to multiply.

As discussed in Section 3.2.2.5, we can estimate the total number of linear or bidirectional GALS wrapper links. This number multiplied with the single handshake communication probability will simply give us the total number of active wrapper links per unit of time. The total number of GALS wrappers for different partition sizes in shown in Figure 3.4a. Figure 3.4b shows the activity of these wrappers per clock cycle depending on inter-island communication frequency. Single handshake communication probabilities of 0.5% and 1% are chosen for our analysis, considering both linear and bidirectional wrapper communication paths. These values were specifically chosen to represent designs having highly localised computation dataflow blocks where GALS synchronisation latency penalty would be minimum. Our range of values are in line with [41] where GALS implementation of applications such as 64-point complex FFT, JPEG encoder and 802.11a/g show similar low inter-island communication frequency. To simplify our analysis, we assume that the single handshake communication probability remains the same with finer grain of partitioning.

The penalties paid by the communication links can, thus, be estimated based on probability of occurrence every cycle. By using the metrics discussed in Section 3.2.3.1, we have plotted the system latency penalty versus increasing number of GALS partitions. The number of islands pointing to 1 indicates the globally synchronous chip. The latency penalty suffered due to GALS wrapper penalties in our case is depicted in Figure 3.5. As expected, it is seen that the performance penalties worsen with more number of GALS partitions. The latency penalty paid by bidirectional wrappers increases at a greater rate



(a) Number of GALS wrapper links



(b) Active GALS wrapper links

Figure 3.4: GALS inter-island communication distribution

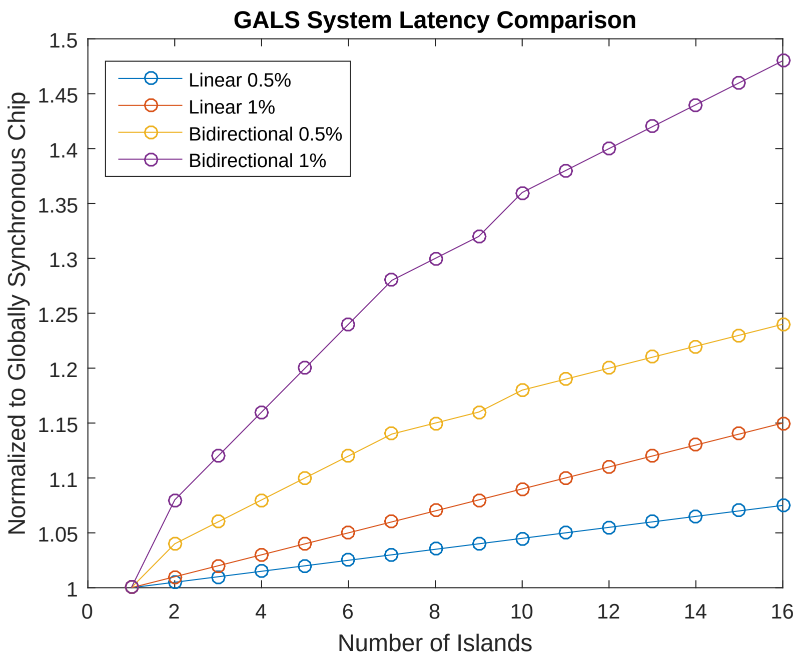
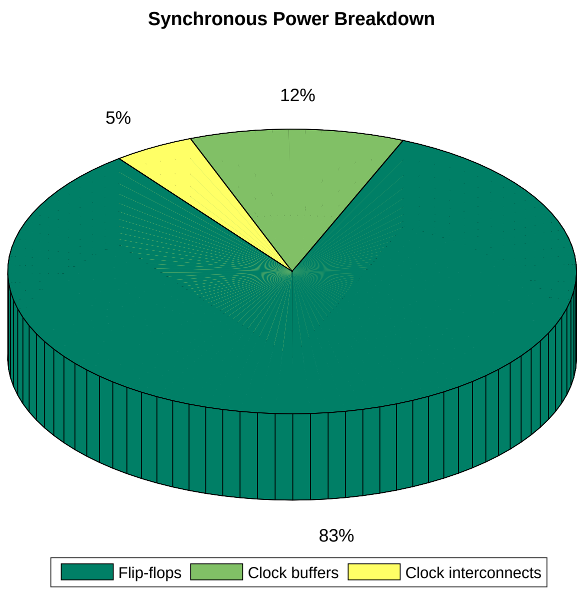


Figure 3.5: System latency analysis of GALS versus synchronous chip

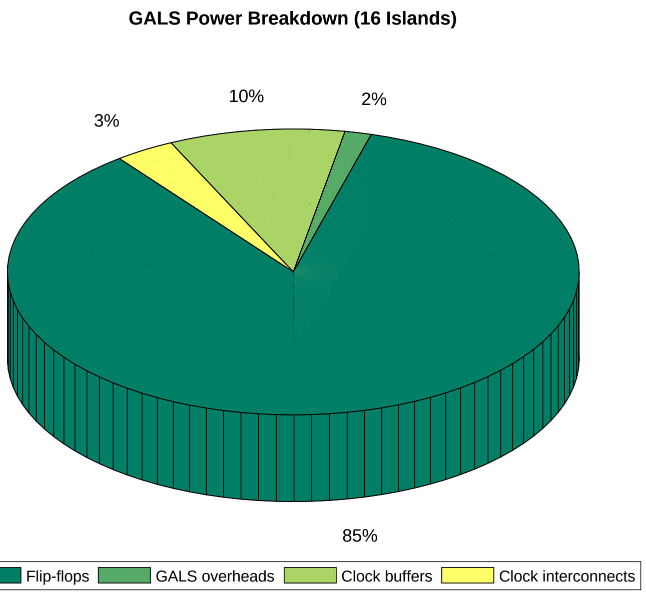
than linear wrappers due to presence of significantly more number of wrapper links. For GALS partitioning of seven islands, linear inter-island communication path costs worst-case system latency increment of 3% and 6% for inter-island frequencies of 0.5% and 1%, respectively, while bidirectional inter-island communication path costs worst-case system latency increment of 14% and 28% for inter-island frequencies of 0.5% and 1% respectively. Thus, the system dataflow, partition granularity and inter-island activity will have a significant impact to system latency during GALSification.

3.3.2 GALS Design Benefit to System Power

The power consumption of individual islands and various partition sizes of the generic design were calculated according to the formulae given in Section 3.2.3.2. Figure 3.6 illustrates the power consumption proportion of different components in the synchronous chip and its GALS version of 16 physical partitions. It is seen that overheads of GALS design only cost about 2% to the GALS chip power for bidirectional dataflow activity of 1%. 4% overall reduction in power contribution from clock buffers and clock interconnects of the GALS chip is seen compared to the synchronous chip. This clearly shows the effect of reduced chip power due to locally optimised CDNs. The number of clock tree levels reduced from eight to four for the case of 16 GALS partitions.

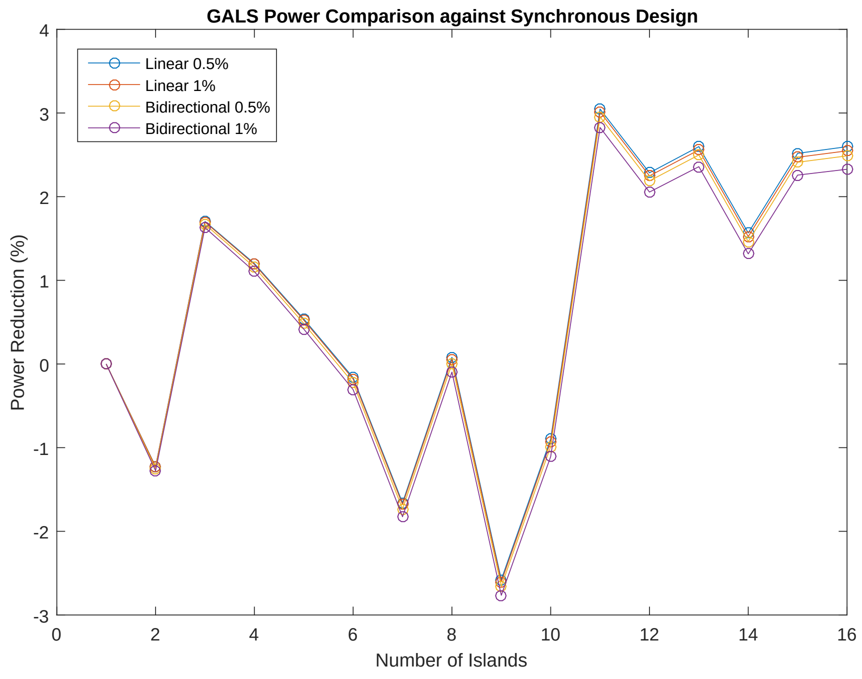


(a) Synchronous design power (only CDN)

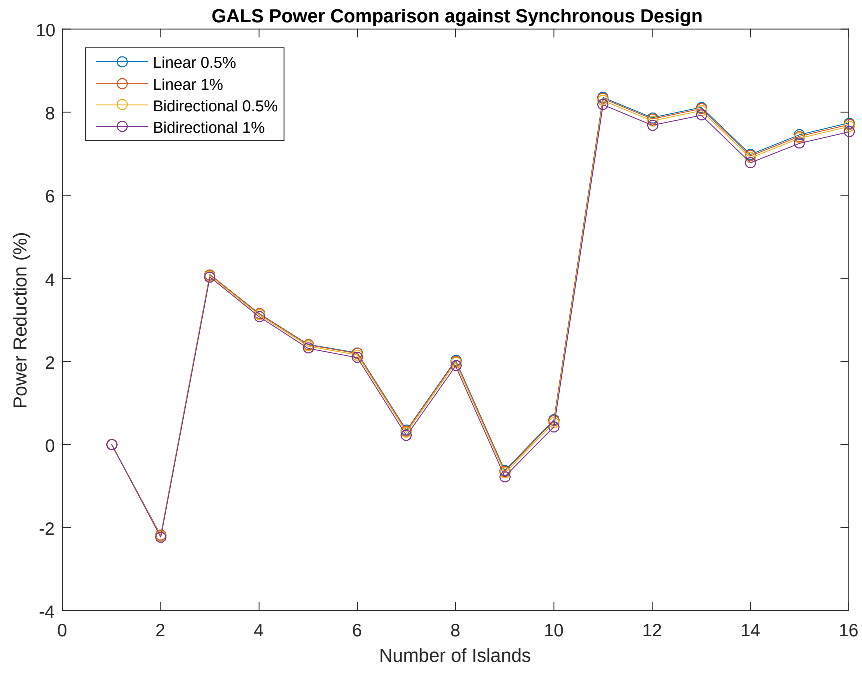


(b) GALS design power with 1% bidirectional dataflow activity (CDN plus GALS overheads)

Figure 3.6: Design power breakdown at 218MHz clock (65nm)



(a) Power analysis at 218MHz clock



(b) Power analysis at 544MHz clock

Figure 3.7: Power efficiency analysis of GALS versus synchronous design (65nm)

The overall power consumed by GALS and its overheads was computed and plotted against varying number of partitions of the same design. The power consumed by the GALS wrapper has been accounted for four cases of wrapper communication styles as presented in the previous section. Figure 3.7 shows the GALS versus synchronous power consumption analysis for two cases of system clock frequency. It can be visualised that GALS offers greater power reduction benefit at higher clock frequencies. This is purely because faster clocks, having larger CDN buffers, offer greater savings from CDN optimisation due to smaller clock variation budget. At 544 MHz clock, GALS partitioning shows power reduction of 4% when partitioned into three islands with maximum reduction of 8.35% observed at partition sizing of eleven islands. There is a trend of reduced power consumption with increasing partitioning granularity due to smaller and less power-hungry CDNs. However, the trend is not consistent across the graph. After analysing the CTS and CDN optimisation algorithms, it was noted that this effect was due to suboptimal buffer synthesis arising from certain cases of floorplan geometry, buffer sizing factor, sink capacitance and global skew attributes. Furthermore, it can be seen that the contribution of wrapper communication activity to the GALS chip power is insignificant. If we compare Figure 3.5 and Figure 3.7, we can observe that higher number of GALS wrapper links cost much greater penalty to the system latency than to the chip power.

3.3.3 Energy Efficiency Analysis

Here, the quality of a GALS implementation over its synchronous counterpart is evaluated by conducting an energy efficiency analysis. The metric REE derived in Sec. 3.2.3.3 is employed for the analysis. This metric shows whether the GALS implementation can justify the design overheads and perform better than a globally synchronous design. Figure 3.8 shows the REE for the four cases of wrapper communication probability at the 65nm technology node. It is observed that, for linear wrapper communication activity of 0.5%, the best case scenario of energy efficiency is three and eleven GALS islands. Energy efficiency improvement of 3.2% and 3.9% is experienced, respectively, over the synchronous implementation. For the case of 1% linear wrapper activity, only one energy efficient implementation is seen for three island GALS partitioning. The bidirectional wrapper

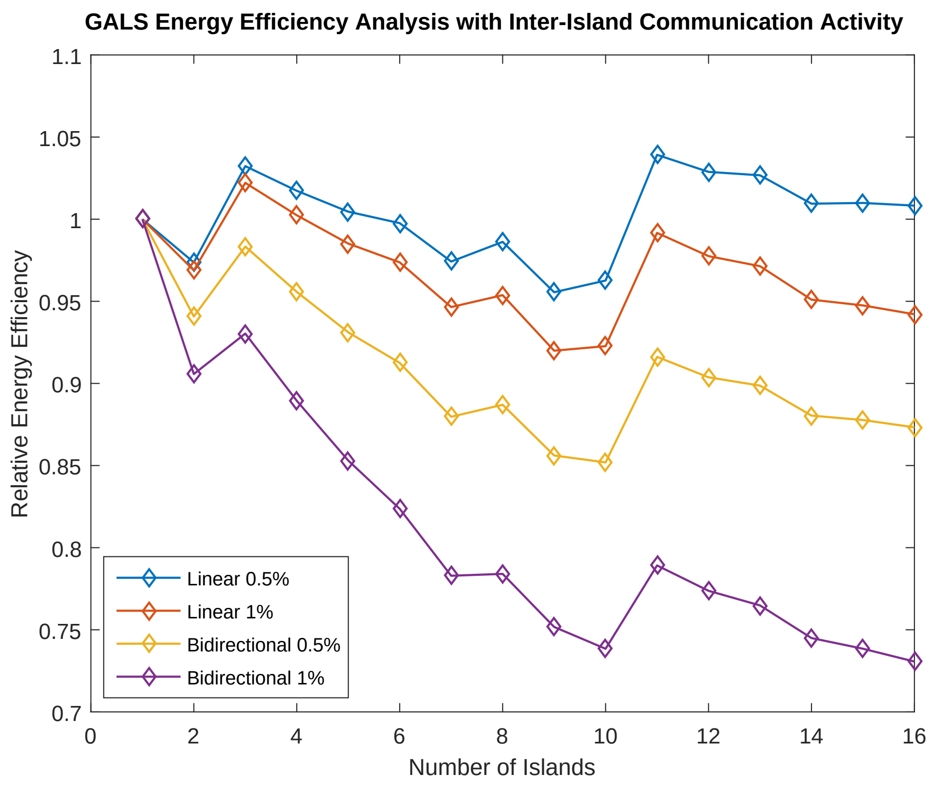


Figure 3.8: REE analysis of GALS versus synchronous chip at 544MHz clock (65nm)

communication dataflow costs significant system latency penalty and hence, poor REE figures for GALS can be observed versus the synchronous chip. This figure demonstrates that REE does not necessarily improve with higher number of partitions. Inter-island communication penalty reduces the design benefit gained from power-efficient CDNs.

Figure 3.9 shows the REE of GALS versus synchronous chip for the four process technologies listed in Table 3.1, taking the case of linear wrapper communication path with 0.5% single handshake probability. The designs compared at each technology node are equivalent having the same distribution of flip-flops. It can be seen that all technologies show similar patterns of REE measurements with number of GALS islands. Furthermore, there is a noticeable trend of REE improvement as we move away from the older 180nm technology node towards the newer 65m technology node. This is mainly because power-efficiency of GALS CDNs is multiplied at newer process technologies due to scope for greater CDN buffer size optimisation arising from larger variability-induced clock skew in the fully synchronous CDN. This result clearly shows the potential of GALS design for future technology nodes. We have also modelled REE with increasing design size. Figure 3.10 depicts the REE of four different design sizes at the 65nm process technology

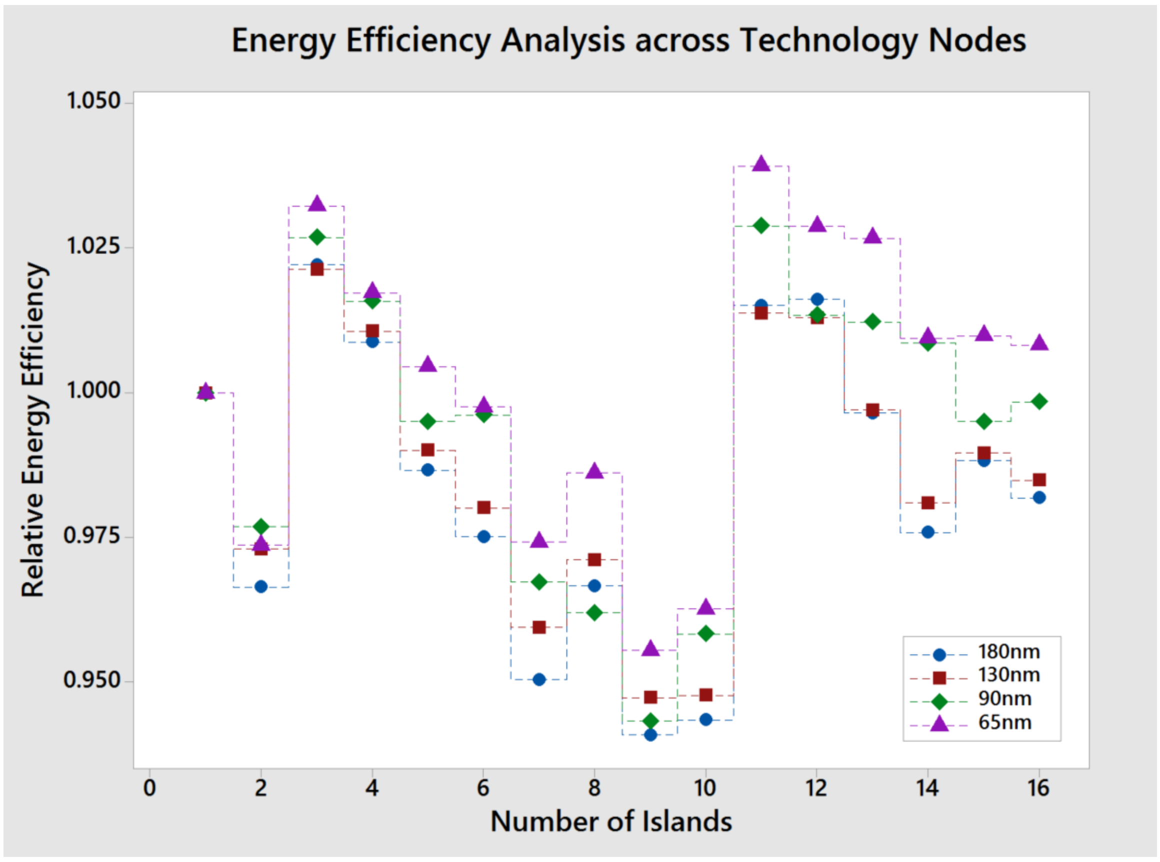


Figure 3.9: Trend of REE with process technology (41,210 FFs)

node considering linear wrapper communication path with 0.5% single handshake probability. Here, we have increased the partitioning granularity from 16 islands to 64 islands for better visibility of the trends. We can clearly see that REE increases with larger design sizes. In this case, REE improvement is due to greater CDN optimisation opportunity from a larger CDN and a larger capacitive clock load.

3.4 Conclusions and Future Work

In this work, we have presented a physical partitioning methodology to help evaluate GALS design technique over a fully synchronous implementation. We considered a generic design example subjected to a certain probability of inter-island communication, from which we analysed different cases of GALS island granularity and communication traffic. Also, several metrics were discussed to model the penalties and gains of GALS design. The extent of GALS energy efficiency was observed to be dependent on design size, process technology. system dataflow, clock frequency and partition granularity. The results

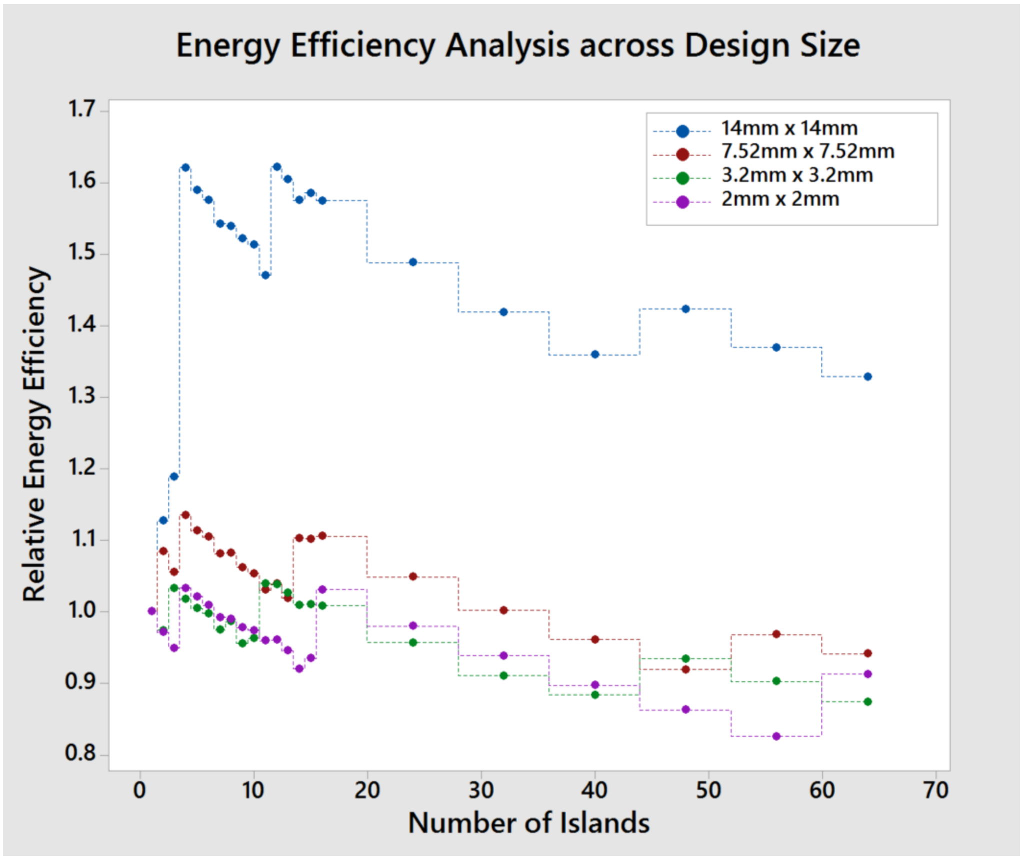


Figure 3.10: Trend of REE with design size (65nm)

indicate that an optimum number of GALS islands can be suggested per technology node for a digital design given its design size, system dataflow and system clock frequency. Furthermore, it was seen that GALS design does well as we transition to newer process technologies demonstrating noticeable energy efficiency improvement. This work highlights how GALS design could address the variability problem of future SoC designs by reducing clock distribution complexity. The methodology presented is this chapter can be adapted for designs of different characteristics and such a tool can be used to evaluate GALS applicability to chip design before expensive design hours are commissioned for GALSification.

Our results, however, show mediocre energy benefit from employing the GALS physical partitioning technique. It was observed that slicing a design into more islands does not necessarily deliver higher energy efficiency. In our analysis, GALSification only offered maximum CDN power reduction of 8.35% at the 65nm technology node. This lower power reduction margin, compared to the 20% CDN power reduction achieved in [16] after GALSification, is due to the H-tree CDN employed in our analysis, being a low power and low skew CDN. This is evident from the fact that the number of clock tree levels dropped from 27 to 10 in [16] for six GALS partitions while the number of clock tree levels dropped from 8 to 4 in our case for 16 GALS partitions. The H-tree, however, has low floorplan flexibility and hence most SoC designs would undergo the standard clock tree synthesis methods. Therefore, better power reduction margins can be expected in practice than demonstrated in this research. Furthermore, additional power benefit could be gained from voltage frequency scaling of individual islands. It should, however, be noted that our analysis methodology did not consider proven chip power optimisation techniques such as clock gating and power gating. Such techniques would reduce the extent of GALS energy savings that come from low-power GALS CDNs. For example, a combination of energy recovery and clock gating techniques is shown to reduce the CDN power of a clocked multiplier by 69% [60]. Moreover, the power savings margin would reduce further when dynamic and leakage power consumption of logic and memory are taken into account. For example, 20% CDN power reduction after GALSification only resulted in overall power savings of 6% [16].

Our analysis also indicated that the nature of inter-island communication activity has a greater impact on GALS design benefit. Chip partitioning should, hence, be carefully planned for minimum inter-island communication frequency so that power savings from GALSification could overcome the latency penalty of GALS wrappers. System functionality and dataflow model thus dictates the extent of energy benefits seen from GALSification. A generic assumption on system dataflow was made in the analysis for comparison simplicity, that the inter-island communication probability remains the same as we partition into more islands. In some situations, the average inter-island communication frequency would increase with more design partitions, when sub-partitions within localised computation blocks communicate at a faster rate. The system dataflow model could be extended to cover several dataflow styles, sub-block communication rates and other inter-island communication interfaces. Furthermore, partitioning a design is also known to affect timing closure positively which could improve individual island clock frequency delivering better system performance. For future work, the simulation framework could be extended or modified to carry out extensive design benefit analysis by incorporating accurate system dataflow pattern, voltage-frequency scaling, elaborate chip power profiling and design-specific communication interfaces.

Through this work, we have identified several limitations of the physical partitioning approach in GALS-based SoC design. Firstly, GALS design provides marginal overall energy benefit over synchronous design with optimal partitioning strategy governed by design size and extent of inter-island communications. Secondly, the parameter having the most impact on chip energy efficiency is the inter-island communication activity which is design dependent and not a physical characteristic. Thirdly, inter-island communication activity occurs between localised computation dataflow blocks and the assumption that physical partitions coincide with these dataflow blocks is suboptimal. For instance, the physical partitioning approach could ignore finer grain dataflow blocks, within larger physical partitions, that could benefit greatly from voltage frequency scaling. Finally, the potential of voltage frequency scaling is, again, determined from behavioural analysis of the system logic and not physical partitioning methods. Therefore, the factors that strongly affect the REE metric are ruled by functional properties of the design making system dataflow analysis a crucial factor before chip partitioning.

The physical partitioning strategy for designing mixed synchronous-asynchronous SoCs only provides a subset of solutions towards energy efficient system design. Overall system energy savings offered by this approach would not be enough to justify the loss of design productivity encountered with this design style. To gain acceptable design benefits from chip partitioning, we need to understand and analyse the functional aspects of a system and adopt custom design partitioning strategies. This strongly motivates the case for functional partitioning, a system level approach to chip partitioning whereby behavioural logic islands are identified based on system dataflow analysis.

In the next chapter, we introduce the theory of *bundles* and approach functional partitioning in a formal manner.

Chapter 4

Theory of Bundles

With chip sizes scaling to deep sub-micron level, semiconductors are experiencing severe variability making it extremely complicated to design ICs in the traditional synchronous fashion. Mixed synchronous-asynchronous design techniques have emerged and are showing great promise in tackling this problem. There is a wealth of formal methods and tools for modelling, verifying and synthesising synchronous and asynchronous circuits. However, the mixed synchronous-asynchronous paradigm is yet to witness a formal method for its complex representation. With regards to circuit verification, satisfying the property of signal persistence is one of the key requirements for modelling hazard-free asynchronous circuits. For synchronous designs, persistence is self-evident as every tick of the clock triggers a sequence of persistent transitions. This, however, is not trivial in mixed synchronous-asynchronous designs due to the presence of asynchrony. We are, hence, motivated to present a mathematical model that describes mixed synchronous-asynchronous behaviour, together with an extension to the traditional notion of persistence for verifying these circuits.

4.1 Introduction

One of the main issues with the complexity of asynchronous circuits was the handling of hazards. Hazards are manifestations of undesirable switching activity called glitches. In the asynchronous style of synchronisation, the output of each circuit element is potentially sensitive to its inputs. This can give rise to non-monotonic pulses (or glitches) when

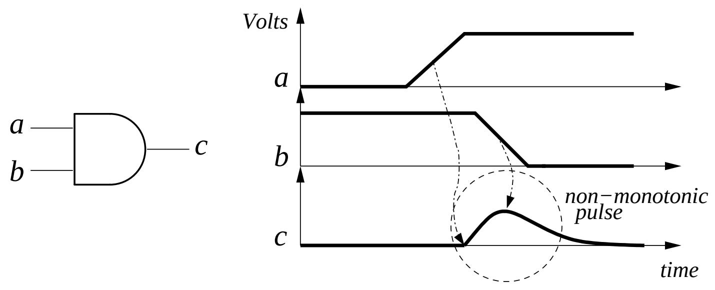


Figure 4.1: Hazardous switching of an AND gate.

transitioning between output states, as illustrated in the waveform of Figure 4.1 taking the case of an AND gate. Due to tight timing between the rising edge of input a and falling edge of input b, the output c produces a non-monotonic pulse before stabilising to a low. This behaviour is hazardous as it is uncertain how the fanout of the AND gate will interpret the glitch; the output c temporary switching to logical 1 or staying at logical 0 all the time.

As shown, for instance, in paper [50], the phenomenon described in the above example can be conveniently interpreted in terms of formal models such as Keller's named transition systems [47] or Petri nets [3]. In particular, what we see in this circuit is the effect where a signal that is enabled (rising edge of c) in a certain state of the circuit may become disabled without firing after the occurrence of another signal (falling edge of b). Such an effect corresponds to the violation of *persistence* property at the level of signal transitions if the latter are used to label the corresponding named transition system. Furthermore, when such a circuit is modelled by a labelled Petri net following the technique of [50], the Petri net would also be classified as a non-persistent one. Thus, it was shown in [50] that the modelling and analysis of an asynchronous circuit with respect to hazard-freedom is effectively reduced to the analysis of persistence of its corresponding Petri net model.

Synchronous circuits, on the other hand, do not require persistence satisfaction as they are intrinsically immune to hazardous behaviour. The principle reason being that the clock, set at worst-case latency period, filters out undesirable circuit switching. This greatly simplifies circuit design compared to asynchronous methods wherein the same circuit had to be analysed for persistence and redesigned to ensure glitch-free operation. Clocked circuits are thus preferred over asynchronous circuits for designing functionally correct (hazard-immune) ICs efficiently. However with chip sizes scaling to deep sub-

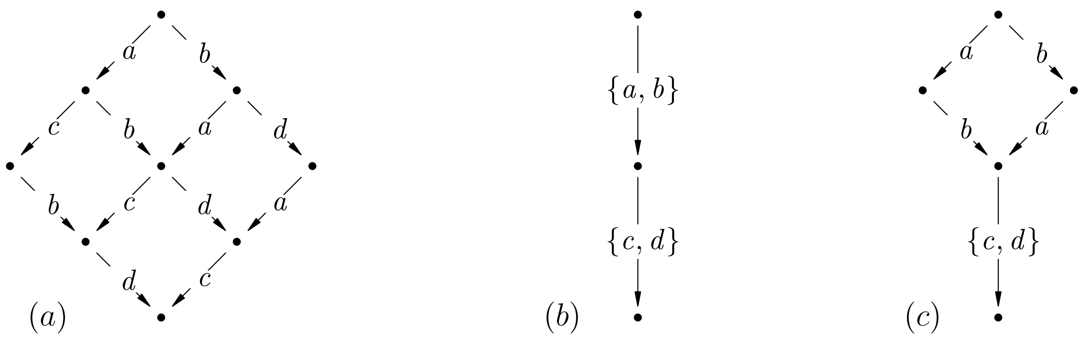


Figure 4.2: Temporal representations of systems having concurrent, parallel and mixed concurrent-parallel behaviours: (*a*) interleaving model for asynchronous behaviour; (*b*) step model for synchronous behaviour; and (*c*) mixed model for mixed synchronous-asynchronous behaviour.

micron level, semiconductors are experiencing severe variability and it is becoming extremely complicated to design chips in the synchronous fashion. This is because designing for variability requires longer safety margins which in turn reduces the clock frequency and degrades circuit performance. To cope with these challenges, asynchronous design methodologies have re-emerged owing to their *inherent* adaptiveness [90]. However, they still suffer significant challenges such as complicated design flow, high overhead costs from control and, lack of CAD support tools and legacy design reuse. Therefore attempts are being made to find a compromise.

Presently, the mixed synchronous-asynchronous design paradigm is showing strong signs of adoption in the semiconductor industry by 2020 [38, 41, 24, 23, 42, 91]. Such systems comprise a mixed temporal behaviour. Asynchronous handshakes handle switching between components where adaptability can significantly improve performance, while clocking is applied to components where worst case performance is tolerable. However, it is worthy of note that modelling such systems would involve detection of potential hazardous states due to presence of asynchronous components, making their design and verification a significant research challenge.

Being a recent trend, there is a lack of formal models that describe correctness of mixed synchronous-asynchronous circuits. The complexity in modelling them begins with the investigation of persistence. It should be noted that the standard notion of persistence has been defined at the level of single actions, which is also known as interleaving semantics of concurrency. This notion has been adequate for representing the correctness of the behaviour of circuits that are fully asynchronous. In asynchronous circuits, there is concurrency between independent actions and sequential order between causally related actions. This notion is well represented by Keller's named transition systems [47]. Figure 4.2(a) depicts such a model capturing the asynchronous behaviour of a system with four events: a, b, c and d. Now, in synchronous circuits, the clock signal would trigger a single action or several actions. These circuits exhibit parallelism between actions in the same clock cycle and sequential order between groups of actions in adjacent clock cycles. To represent this group execution of actions, we will use *steps*, and therefore we need step transition systems to represent such a behaviour. A step represents a single action or a group of actions that are triggered simultaneously from a particular state by the clock signal. Figure 4.2(b) shows such a transition system model capturing the temporal behaviour of a synchronous system with the help of steps. For the case of mixed synchronous-asynchronous systems, there is a mixture of synchrony and asynchrony and hence both concurrent and parallel behaviour have to be represented. Figure 4.2(c) illustrates the mixed temporal behaviour seen in such systems. In all three cases, step transition systems provide a suitable behavioral model, as a single transition can be treated as a singleton step.

Synchronous and asynchronous systems have distinct techniques to guarantee functionally correct behaviour. However, for mixed synchronous-asynchronous systems, it is not so straightforward as correctness should be accounted from both angles. We would like to find an adequate representation of the correct behaviour of such systems. Here, it would be natural to define such a behaviour in analogous way as it was done for asynchronous circuits, i.e., with the use of the notion of persistence. However, when modelling mixed synchronous-asynchronous systems we have to consider complex actions, namely steps, and corresponding transition systems. This chapter is hence centred around extending the notion of persistence to steps.

The chapter is organised as follows. Section 4.2 introduces the notion of persistent steps and discusses their basic properties. Section 4.3 presents the main result of the chapter, an algorithm that prunes the concurrent reachability graph of a net, which serves as an initial system specification, to obtain a representation of desired mixed synchronous-asynchronous behavior. Section 4.4 explains the main motivation behind studying persistent steps in the context of mixed synchronous-asynchronous system design. Finally, Section 4.5 contains conclusions and presents directions for future work.

4.2 Step Persistence in Nets

A concurrent system is persistent [92, 93, 94, 95] if throughout its operation no activity which became enabled can subsequently be prevented from being executed by any other activity. This is often a highly desirable (or even necessary) property; in particular, if the system is to be implemented in hardware [50, 96, 22].

Muller's speed independence theory provided a unique method for guaranteeing hazard-freeness of asynchronous circuits [97]. The *semimodularity* condition in this work required that an excitation of a circuit element must not be withdrawn before being absorbed by the system [98]. This condition was identified by Keller in [47] to be the same as the property of persistence[1] in a transition system model for asynchronous parallel computation. Thus, satisfying the property of persistence became one of the key requirements when designing hazard-free asynchronous circuits. Over the past 40 years, persistence has been investigated and applied in practical implementations assuming that each activity is a single atomic action which can be represented, for example, by a single transition of a Petri net. In other words, persistence was considered assuming the sequential execution semantics of concurrent systems [45].

The idea of persistence has been investigated in many papers, for example, in [99, 95, 93, 96, 22, 92, 50, 94]. However, with the exception of [96, 22], it was only considered in the context of sequential executions of systems, and defined for single transitions (rather than steps) as follows:[2]

Definition 4.2.1. (persistent net, [92])

A PT-net $\mathcal{N} = (P, T, W, M_0)$ is *persistent* if, for all distinct transitions $a, b \in T$ and any reachable marking $M \in [M_0\rangle$, $M[a\rangle$ and $M[b\rangle$ imply $M[ab\rangle$. ■

We can re-write this definition from the point of view of single transition as follows:

[1][47] was the first work to consider persistence in the context of Petri nets.

[2]For simplicity, we do not distinguish transitions as input, output or internal signals here. Application of step persistence to signal persistence and output persistence properties [3] will be explored in the future.

Definition 4.2.2. (persistent transition)

Let a be a transition enabled at a marking M of a PT-net $\mathcal{N} = (P, T, W, M_0)$. Then a is *locally persistent at* M if, for every transition b enabled at M, $b \neq a \implies M[ba\rangle$.

Moreover, an active transition a is *globally persistent* in $\mathcal{N}$ if it is locally persistent at every reachable marking of $\mathcal{N}$ at which it is enabled. ■

The above net-oriented and transition-oriented definitions are closely related as, due to the symmetric roles played by a and b in Definition 4.2.2, we immediately obtain the following.

Proposition 4.2.1. *The following are equivalent:*

- $\mathcal{N}$ is persistent;
- $\mathcal{N}$ contains only globally persistent transitions.

4.2.1 Defining Persistent Steps

We now introduce the central definition of this chapter, in which we lift the notions of persistence from the level of individual transitions to the level of steps. The following definition gives three versions (type-A, type-B and type-C) of a definition of a persistent step. In each case, we try to capture the fact that a persistent step, which is enabled at some reachable marking M, cannot be disabled by another enabled step. The difference in the versions lies either in the different understanding of what 'not to be disabled' means or what we mean by a 'different' step.

Definition 4.2.3. (persistent step in a net)

Let α be a step enabled at a marking M of a PT-net $\mathcal{N} = (P, T, W, M_0)$. Then:

- α is *locally* A*-persistent at marking* M (or LA-persistent) if, for every step β enabled at M,

$$\beta \neq \alpha \implies M[\beta(\alpha \setminus \beta)\rangle$$

- α is *locally* B*-persistent at marking* M (or LB-persistent) if, for every step β enabled at M,

$$\beta \cap \alpha = \varnothing \Longrightarrow M[\beta\alpha\rangle$$

- α is *locally* C*-persistent at marking* M (or LC-persistent) if, for every step β enabled at M,

$$\beta \neq \alpha \Longrightarrow M[\beta\alpha\rangle$$

Furthermore, an active step α is *globally* A/B/C*-persistent* (or GA/GB/GC-persistent) in $\mathcal{N}$ if it is respectively LA/LB/LC-persistent at every reachable marking of $\mathcal{N}$ at which it is enabled. ■

Each of the three types of step persistence is a conservative extension of transition persistence of Definition 4.2.2. Type-A persistence requires that only the unexecuted part of a delayed step is kept enabled, and in this case a persistent step can fail to fully 'survive'. Type-B and type-C persistence, however, insist on preserving the enabledness of whole steps. In type-B persistence, two steps are considered distinct if they are disjoint, whereas in the other two cases it is enough that they are different, and so they can have a nonempty intersection. The empty step is trivially persistent according to all the persistence types in Definition 4.2.3. Note also that although one could drop $\beta \neq \alpha$ in the definitions of type-A persistence, we decided to keep it in order to emphasize a link with the original notion of persistence introduced in [92].

> Since, as proven later, type-A and type-B persistence are equivalent, in the examples we discuss only the type-A and type-C variants of persistence.

Moving from sequential to step semantics changes the perception of persistence in PT-nets introduced by the standard Definition 4.2.1. In particular, in the sequential semantics, by Proposition 4.2.1, all transitions in a persistent net are globally persistent. In the step semantics, the situation is different. Consider, for example, the PT-net in Figure 4.3. It is persistent, and all of its active steps are GA-persistent. However, its nonempty steps fail to be LC-persistent at some of the markings that enable them. More precisely, $\{a\}$, $\{b\}$ and $\{a, b\}$ are not LC-persistent at M_0, while $\{c\}$, $\{d\}$ and $\{c, d\}$ are not LC-persistent at $M_1 = \{p_5, p_6, p_7, p_8\}$. This should not come as a surprise, as type-C persistence is a demanding property. Type-A persistence, on the other hand, is close in spirit to its sequential counterpart.

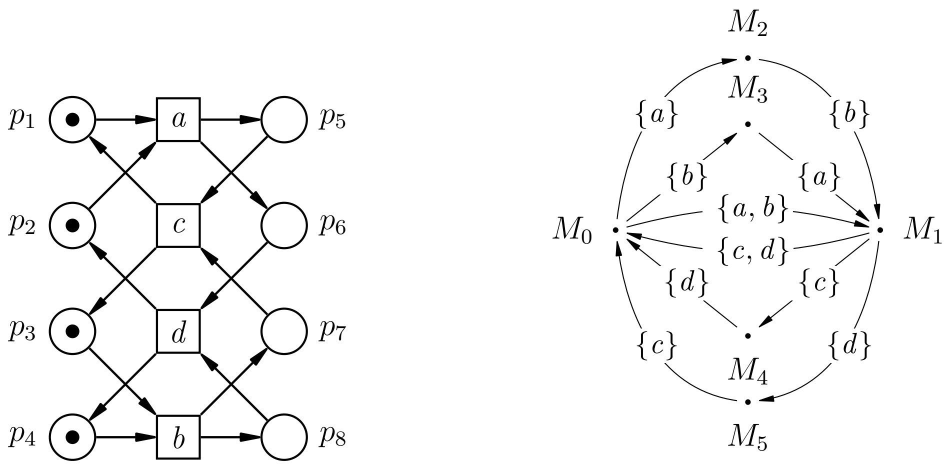


Figure 4.3: A safe persistent PT-net and its concurrent reachability graph.

4.2.2 Basic Properties of Persistent Steps

In this section, we investigate the expressiveness of different notions of persistence defined for steps, assuming first that $\mathcal{N} = (P, T, W, M_0)$ is a general PT-net.

Proposition 4.2.2. Let α be a step enabled at a reachable marking M of $\mathcal{N}$. If α is GA/GB/GC-persistent in $\mathcal{N}$, then α is LA/LB/LC-persistent at M.

Proof. Follows directly from Definition 4.2.3. □

We then obtain a number of inclusions between different types of persistent steps.

Proposition 4.2.3. Let α be an active step and M be a marking of $\mathcal{N}$. Then α is LA-persistent at M iff α is LB-persistent at M.

Proof. Assume that α is enabled at M, and β is another step enabled at M.

Suppose that α is LA-persistent at M and $\beta \cap \alpha = \varnothing$. Then $M[\beta(\alpha \setminus \beta)\rangle$ and $\alpha \setminus \beta = \alpha$. Hence $M[\beta\alpha\rangle$, and so α is LB-persistent at M.
Conversely, suppose that α is LB-persistent at M and $\beta \neq \alpha$. Then $M[(\beta \setminus \alpha)\alpha\rangle$ as $(\beta \setminus \alpha) \cap \alpha = \varnothing$ and $M[\beta \setminus \alpha\rangle$ (cf. Fact 2.1.2). Hence $M[(\beta \setminus \alpha)(\alpha \cap \beta)(\alpha \setminus \beta)\rangle$ (cf. Fact 2.1.2). Thus, by $M[\beta\rangle$, $M[\beta(\alpha \setminus \beta)\rangle$. Hence α is LA-persistent at M. □

Corollary 4.2.1. Let α be an active step of $\mathcal{N}$. Then α is GA-persistent in $\mathcal{N}$ iff α is GB-persistent in $\mathcal{N}$.

Proposition 4.2.4. Let α be an active step and M a marking of $\mathcal{N}$. If α is LC-persistent at M, then α is LA-persistent at M.

Proof. Since enabledness of steps is monotonic in PT-nets (cf. Fact 2.1.2), the implication follows directly from Definition 4.2.3, where the statement for LC-persistence has stronger consequence. □

Corollary 4.2.2. Let α be an active step of a $\mathcal{N}$. If α is GC-persistent in $\mathcal{N}$, then α is GA-persistent in $\mathcal{N}$.

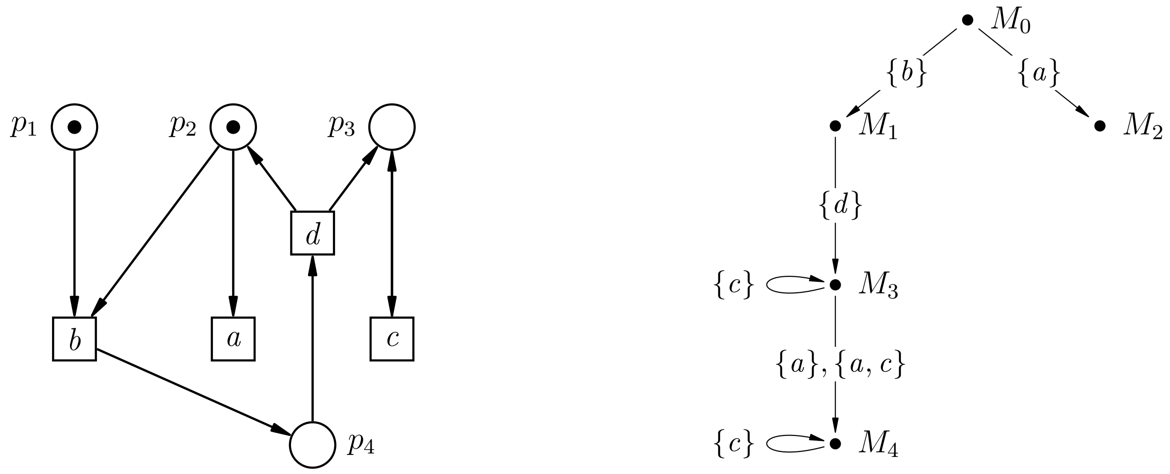


Figure 4.4: A safe PT-net and its concurrent reachability graph.

We now make a series of statements and observations concerning the general PT-net model.

- The implications in Proposition 4.2.4 cannot be reversed. A counterexample is provided in Figure 4.4, where $\{a\}$ is LA-persistent at $M_3 = \{p_2, p_3\}$. However, it is not LC-persistent at M_3. The example in Figure 4.4 can as well be an illustration for Proposition 4.2.2 (type-A) showing the case of an LA-persistent step $\{a\}$ (at M_3), which is not GA-persistent (because of M_0).

- The implications in Corollary 4.2.2 cannot be reversed. A counterexample is again provided in Figure 4.4, where $\{a, c\}$ is GA-persistent, but it is not GC-persistent. As this step is only enabled at marking M_3, it fails to be LC-persistent as well. Moreover, in Figure 4.4, $\{d\}$ is a step that is type-A and type-C globally persistent, because it is only enabled at one marking, M_1, and no other nonempty step is enabled at M_1.

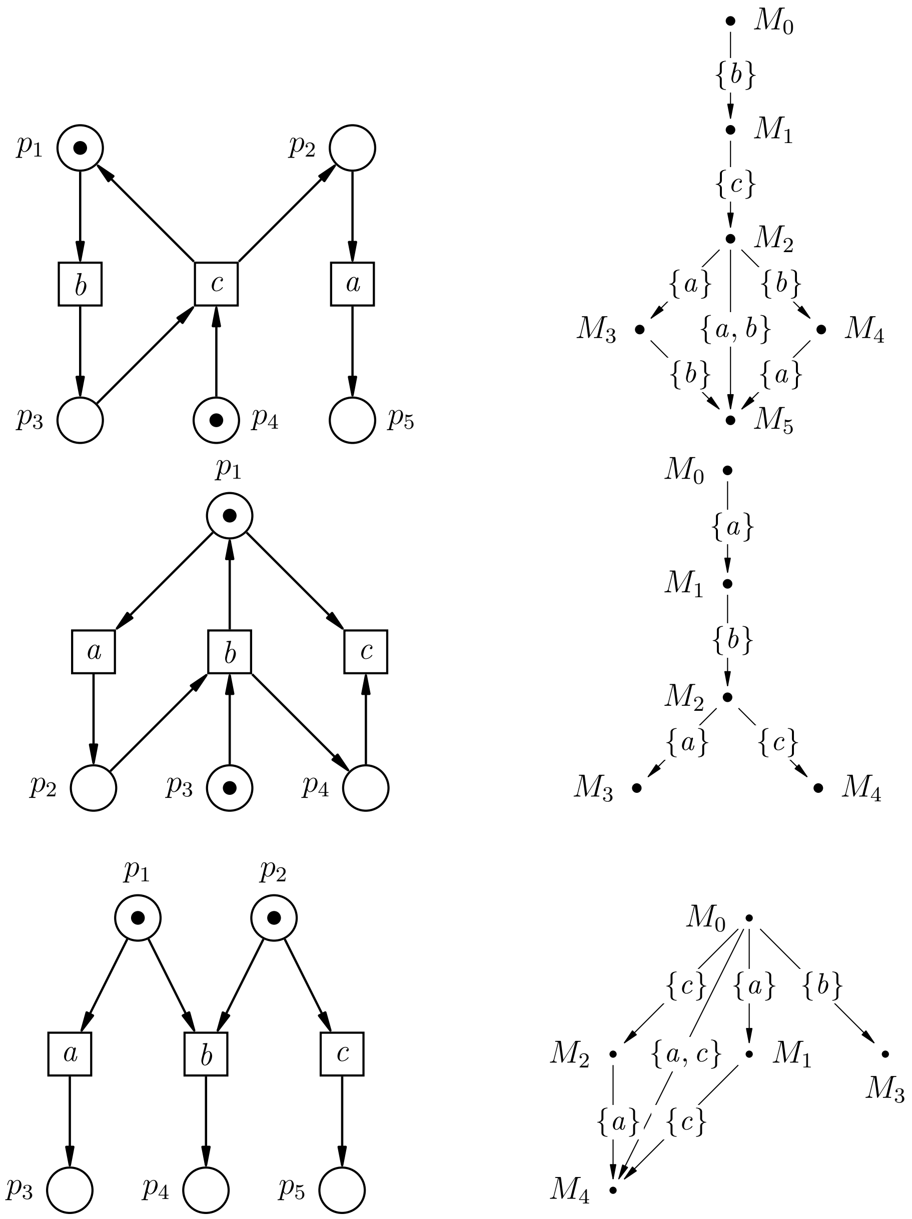


Figure 4.5: Three safe PT-nets and their concurrent reachability graphs.

- The top PT-net in Figure 4.5 shows that a step $\{a\}$ may be GA-persistent, but only LC-persistent (at M_4). Step $\{a\}$ is not GC-persistent, because it is not LC-persistent at M_2.

- The middle PT-net in Figure 4.5 shows an example of a step, $\{a\}$, that is LC-persistent at M_0 (hence also LA-persistent at M_0), but it is not GA-persistent (consequently not GC-persistent).

- There may be steps in PT-nets that fail to satisfy all the types of persistence; for example, $\{a, c\}$ and $\{b\}$ in the bottom PT-net of Figure 4.5.

- There are PT-nets where all steps not persistent whatever type (A or C) we choose. For example, take the bottom PT-net in Figure 4.5 and delete p_2, p_5 and c with all adjacent arcs. Then, the only nonempty steps in the concurrent reachability graph are $\{a\}$ and $\{b\}$, and they prevent each other from being persistent.

4.2.3 Global Persistence in Safe PT-nets

In this section, we focus our attention on safe PT-nets, and assume throughout that $\mathcal{N} = (P, T, W, M_0)$ is such a net.

It turns out that all non-singleton steps in $\mathcal{N}$, which are GC-persistent, are built out of transitions lying on self-loops. To show this, we first prove an auxiliary result.

Proposition 4.2.5. Let α be a GC-persistent step enabled at a reachable marking M of a safe PT-net $\mathcal{N}$. Then ${}^\bullet(\alpha \cap \beta) = (\alpha \cap \beta)^\bullet$, for every step $\beta \neq \alpha$ enabled at M.

Proof. We may assume that $\alpha \cap \beta \neq \varnothing$ as for $\alpha \cap \beta = \varnothing$ the result holds by ${}^\bullet(\alpha \cap \beta) = \varnothing = (\alpha \cap \beta)^\bullet$.

Assume that α is GC-persistent and consider two cases.

Note that by Fact 2.1.3 steps α, β and $\alpha \cap \beta$ are disconnected.

Case 1: $p \in {}^\bullet(\alpha \cap \beta)$, for some step $\beta \neq \alpha$ enabled at M. Clearly, $M(p) = 1$ and there exists $b \in (\alpha \cap \beta)$ such that $p \in {}^\bullet b$. If α is GC-persistent, there is a marking M' such that $M[\beta\rangle M'[\alpha\rangle$. As $M'[\alpha\rangle$ and $p \in {}^\bullet(\alpha \cap \beta)$, we have $M'(p) = 1$. Furthermore, $M'(p) = M(p) - W(p, \beta) + W(\beta, p)$.

And so, by disconnectedness of α and β, and the fact that $M(p) = M'(p) = 1$ we get $W(b, p) = W(p, b) = 1$. Moreover, for any $c \in (\alpha \cup \beta)$ such that $c \neq b$ we get $W(c, p) = W(p, c) = 0$. Therefore $W(p, a) = W(a, p)$ for each $a \in (\alpha \cup \beta)$. Hence $p \in (\alpha \cap \beta)^\bullet$, and so ${}^\bullet(\alpha \cap \beta) \subseteq (\alpha \cap \beta)^\bullet$.

Case 2: $p \in (\alpha \cap \beta)^\bullet \setminus {}^\bullet(\alpha \cap \beta)$. Then, by $M[\alpha \cap \beta\rangle$ and the safeness of $\mathcal{N}$, $M(p) = 0$. Hence, by $M[\alpha\rangle$ and $M[\beta\rangle$, we must have $p \notin {}^\bullet\alpha \cup {}^\bullet\beta$. Consequently, since there is M'' such that $M[\beta\alpha\rangle M''$ in case of GC-persistence of α, we obtain $M''(p) \geq 2$, a contradiction with $\mathcal{N}$ being safe. Thus $(\alpha \cap \beta)^\bullet \setminus {}^\bullet(\alpha \cap \beta) = \varnothing$.

Hence ${}^\bullet(\alpha \cap \beta) = (\alpha \cap \beta)^\bullet$ and the result holds. □

Theorem 4.2.1. Let α be a non-singleton active step of a safe PT-net $\mathcal{N}$. If α is GC-persistent, then it is lying on self-loops.

Proof. If $\alpha = \varnothing$ the result holds. Let $|\alpha| \geq 2$. Suppose that $a \in \alpha$ and M be a reachable marking enabling α. Since $\{a\} \neq \alpha$ and $M[a\rangle$ for any marking M such that $M[\alpha\rangle$, we have ${}^\bullet(\alpha \cap \{a\}) = (\alpha \cap \{a\})^\bullet$ (cf. Proposition 4.2.5). Hence ${}^\bullet a = a^\bullet$. $\square$

We now want to relate the persistence of a step with the persistence of its constituent transitions in safe nets. We first consider GA-persistent steps, but as we already know, from Corollary 4.2.1, the results would also hold for GB-persistent steps.

Theorem 4.2.2. Let α be an active step in a safe PT-net $\mathcal{N}$. If all the transitions in α are globally persistent, then α is GA-persistent.

Proof. Let M be a reachable marking and $\beta \neq \alpha$ be a step in $\mathcal{N}$ such that $M[\alpha\rangle$ and $M[\beta\rangle$.

We assume that all the transitions in α are globally persistent. We need to show that $M[\beta(\alpha \setminus \beta)\rangle$.

Let $\alpha \cap \beta = \{a_1, \ldots, a_m\}$, $\alpha \setminus \beta = \{b_1, \ldots, b_n\}$ and $\beta \setminus \alpha = \{c_1, \ldots, c_k\}$. Note that all the transitions in these three sets are different. From $M[\beta\rangle$ and Fact 2.1.2, we have $M[a_1 \ldots a_m c_1 \ldots c_k\rangle$. Now, since each b_i is globally persistent and enabled at M, we have that $M[a_1 \ldots a_m c_1 \ldots c_k b_1 \ldots b_n\rangle$. Since α and β are steps in a safe net $\mathcal{N}$ enabled at some marking (M), we have, from Fact 2.1.3, that transitions in α and β have disjoint pre-sets and post-sets. Hence we have $M[\beta(\alpha \setminus \beta)\rangle$. $\square$

We now consider GC-persistent steps. In this case the antecedent in the implication is stronger.

Theorem 4.2.3. Let α be an active step in a safe PT-net $\mathcal{N}$. If all the transitions in α are globally persistent and lying on self-loops, then α is GC-persistent.

Proof. Let M be a reachable marking and $\beta \neq \alpha$ be a step such that $M[\alpha\rangle$ and $M[\beta\rangle$. We need to show that $M[\beta\alpha\rangle$.

Let $\alpha \cap \beta = \{a_1, \ldots, a_m\}$, $\alpha \setminus \beta = \{b_1, \ldots, b_n\}$ and $\beta \setminus \alpha = \{c_1, \ldots, c_k\}$. Proceeding similarly as in the previous proof we can show that

$$M[a_1 \ldots a_m c_1 \ldots c_k b_1 \ldots b_n\rangle \ .$$

Since all transitions in α are lying on self-loops and are globally persistent, we further obtain

$$M[a_1 \ldots a_m c_1 \ldots c_k a_1 \ldots a_m b_1 \ldots b_n\rangle \ .$$

Hence, from $M[\alpha\rangle$, $M[\beta\rangle$ and Fact 2.1.3, we have $M[\beta\alpha\rangle$. $\square$

In Theorems 4.2.2 and 4.2.3, the reverse implications do not hold. A counterexample is provided in Figure 4.6, where a persistent step $\alpha = \{a, c\}$ contains a non-globally persistent transition a. Indeed, $\alpha = \{a, c\}$ is both GA-persistent and GC-persistent, but $a \in \alpha$ is not persistent at $M_0 = \{p_1, p_2\}$, because there exists $b \neq a$ such that $M_0[a\rangle$ and $M_0[b\rangle$, but $M_0[ba\rangle$ does not hold.

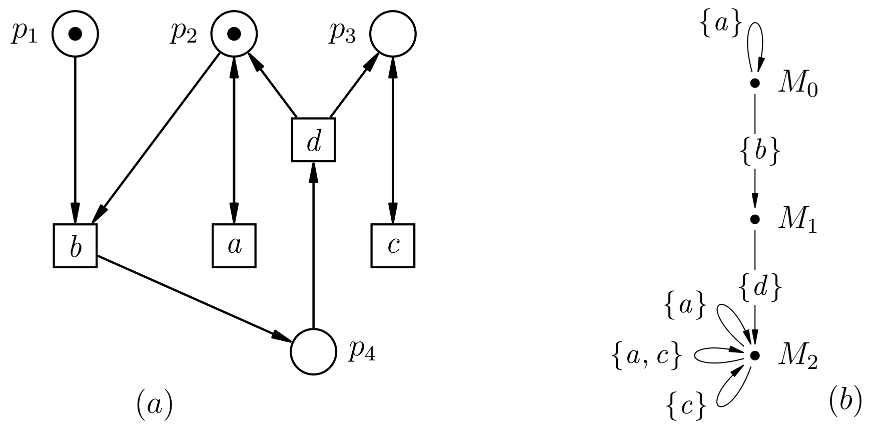


Figure 4.6: A safe PT-net $\mathcal{N}$ (a); and its concurrent reachability graph $CRG(\mathcal{N})$ (b).

4.3 Pruning Reachability Graphs

In this section, we turn from general considerations relating to the persistence of active steps to more application oriented discussion, restricting ourselves to the case of global persistence.

The original motivation for studying persistent steps in this chapter was to discover which sets of transitions — called later *bundles* — can be executed synchronously and therefore be treated as some kind of 'atomic actions', giving rise to new 'bigger' transitions, which would execute in a 'hazard-free' way. In the application area of asynchronous circuits, bundling actions would reduce signal management by merging concurrent signals into one event. This merging must be done in a consistent fashion.

The best candidates for bundles are, in fact, persistent steps, but if we want to form 'bigger' transitions from them, we must make sure that one enabled persistent step does not include another enabled persistent step. All the transitions in a bundle must always appear together, in the same configurations. In the ideal situation (we say ideal, because it might be difficult to achieve), we do not want to allow, for example, three persistent steps $\{a,b\}$, $\{a\}$ and $\{b\}$ to be enabled in a given transition system. We need to choose: either to opt for $\{a,b\}$ and delete $\{a\}$ and $\{b\}$, or the other way round. Therefore, we need to develop an algorithm which, for a given net $\mathcal{N} = (P,T,W,M_0)$, would allow us to prune its reachability graph $CRG(\mathcal{N})$ in such a way that all persistent steps would satisfy an additional 'non-inclusion' condition. The 'pruned' transition system would represent the desired behaviour, which then we would like to implement in a form of a Petri net in a process of synthesis. The 'non-inclusion' condition could be of a local nature (less restrictive) or global nature (more restrictive) as introduced later in Problem 1 as (LNI) and (GNI) respectively. The (LNI) condition determines the possibility of having $\{a\}$ and $\{b\}$ occurring concurrently in one place of the transition system and $\{a,b\}$ occurring as bundled in another place of the transition system, while (GNI) allows either $\{a\}$ and $\{b\}$, or $\{a,b\}$.

We start by defining sub-ST-systems which will be obtained by pruning concurrent reachability graphs.

Definition 4.3.1. (sub-ST-system)

An ST-system $\mathcal{S} = (Q,A,q_0)$ is a *sub-st-system* of an ST-system $\mathcal{S}' = (Q',A',q_0)$ if $Q \subseteq Q'$, $A \subseteq A'$ and, for every $q \in Q$, $ready_{\mathcal{S}}(q) = ready_{\mathcal{S}'}(q)$. We denote this by $\mathcal{S} \preccurlyeq \mathcal{S}'$. ■

In the above definition, $En_{\mathcal{S}}$ of a 'properly pruned' reachability graph $\mathcal{S}'$ will be a set of <u>bundles</u>. What we mean by 'properly pruned' will be described by conditions stated in Problem 1.

We now re-define for ST-systems the three notions concerned with global persistence introduced for PT-nets. The reason is that once we start pruning an ST-system, we need to check whether the remaining steps that were previously persistent remain persistent. Such checks will be carried out for ST-systems that might not be concurrent reachability graphs of any PT-nets.

Definition 4.3.2. (persistent step in an ST-system)

A step $\alpha \in En_{\mathcal{S}}$ is GA/GB/GC-*persistent* in an ST-system $\mathcal{S} = (Q, A, q_0)$ if, for all states $q \in Q$ and steps β such that $\alpha, \beta \in En_{\mathcal{S}}(q)$ we respectively have:

$$\begin{array}{lrcl}
(\mathrm{GA}) & \beta \neq \alpha & \Longrightarrow & q \xrightarrow{\beta(\alpha\setminus\beta)} \\
(\mathrm{GB}) & \beta \cap \alpha = \varnothing & \Longrightarrow & q \xrightarrow{\beta\alpha} \\
(\mathrm{GC}) & \beta \neq \alpha & \Longrightarrow & q \xrightarrow{\beta\alpha} \ .
\end{array}$$

■

Proposition 4.3.1. A step α is GA/GB/GC-persistent in a PT-net iff α is respectively GA/GB/GC-persistent in its concurrent reachability graph.

Proof. Follows directly from the definitions. □

We have the following relationships between just introduced notions of step persistence.

Proposition 4.3.2. *Let $\mathcal{S} = (Q, A, q_0)$ be an* ST-*system.*

1. If $\alpha \in En_{\mathcal{S}}$ is GA-persistent, then it is also GB-persistent.

2. If $\alpha \in En_{\mathcal{S}}$ is GC-persistent, then it is also GB-persistent.

Proof. (1) Let $q \in Q$ and $q \xrightarrow{\alpha}$ and $q \xrightarrow{\beta}$ be such that $\beta \cap \alpha = \varnothing$. Since $\alpha \in En_{\mathcal{S}}$ is GA-persistent, we have $q \xrightarrow{\beta(\alpha\setminus\beta)}$. Hence $q \xrightarrow{\beta\alpha}$ which means that $\alpha \in En_{\mathcal{S}}$ is GB-persistent.

(2) Follows directly from Definition 4.3.2. □

Unlike for PT-nets, in the case of ST-systems, GB-persistence does not imply GA-persistence of steps. Indeed, let $\alpha \in En_{\mathcal{S}}$ be a GB-persistent step in $\mathcal{S}$, and $\beta \neq \alpha$ and $q \in Q$ be such that $q \xrightarrow{\alpha}$ and $q \xrightarrow{\beta}$. We know that $\beta \cap (\alpha \setminus \beta) = \varnothing$. However, with such assumptions, we cannot in general guarantee that $q \xrightarrow{\alpha\setminus\beta}$. Though the latter is true for the concurrent reachability graphs of PT-nets, we must also consider ST-systems resulting from their pruning (see the ST-system depicted on Figure 4.8(c), where the step $\{b, d\}$ is GB-persistent, but not GA-persistent). For similar reasons, in the case of ST-systems, GC-persistence does not imply GA-persistence.

We now can formulate a problem which is our main concern in this section.

Problem 1. *Let $\mathcal{N}$ be a* PT*-net and $CRG(\mathcal{N})$ be its concurrent reachability graph. Construct an* ST*-system $\mathcal{S}$ such that $\mathcal{S} \preccurlyeq CRG(\mathcal{N})$ and all steps in $En_{\mathcal{S}}$ are* GB*-persistent in $\mathcal{S}$*[3]*, and additionally satisfying (*GNI*) or (*LNI*), where the latter conditions are defined as follows:*

(GNI) $\quad \alpha \not\subset \beta$, *for all nonempty steps* $\alpha, \beta \in En_{\mathcal{S}}$

(LNI) $\quad \alpha \not\subset \beta$, *for all states q and all nonempty steps* $\alpha, \beta \in En_{\mathcal{S}}(q)$.

We denote this respectively by

$$\mathcal{S} \preccurlyeq^{global}_{pers} CRG(\mathcal{N}) \qquad and \qquad \mathcal{S} \preccurlyeq^{local}_{pers} CRG(\mathcal{N}) .$$

*We also refer to (*GNI*) as* global non-inclusion*, and to (*LNI*) as* local non-inclusion. ■

The difference between $\preccurlyeq^{global}_{pers}$ and $\preccurlyeq^{local}_{pers}$ is that the latter only requires non-inclusion of bundles locally for each state, whereas the former insists that non-inclusion holds globally.

Proposition 4.3.3. *$\mathcal{S} \preccurlyeq^{global}_{pers} CRG(\mathcal{N})$ implies $\mathcal{S} \preccurlyeq^{local}_{pers} CRG(\mathcal{N})$.*

Proof. Follows directly from the definition. □

In our first attempt to solve Problem 1, we will concentrate on PT-nets that are persistent according to Definition 4.2.1. We then have the following result.

Theorem 4.3.1. Let $\mathcal{N}$ be a *pt*-net which is persistent according to Definition 4.2.1. Then there is an *st*-system $\mathcal{S}$ satisfying $\mathcal{S} \preccurlyeq^{global}_{pers} CRG(\mathcal{N})$.

[3]Alternatively, we could require GA-persistence or GC-persistence. We opted here for GB-persistence, because it is the weakest of the three notions.

Proof. It suffices to take $CRG(\mathcal{N})$ and delete all nonempty non-singleton steps. $\square$

As the above proof produces completely sequential solution, we call such pruning to be trivial. We will now search for nontrivial, hence more concurrent ones. We will also require that the original PT-net is not only persistent, but also safe.

Proposition 4.3.4. *Let $\mathcal{N}$ be a safe* PT*-net which is persistent according to Definition 4.2.1. Then all active steps in $\mathcal{N}$ are* GB*-persistent in $CRG(\mathcal{N})$.*

Proof. Let $\alpha \in En_{CRG(\mathcal{N})}$. As $\mathcal{N}$ is persistent according to Definition 4.2.1, all transitions in α are globally persistent according to Definition 4.2.2. Hence, from Theorem 4.2.2 and the fact that $\mathcal{N}$ is safe, we have that α is GA-persistent in $\mathcal{N}$, and also GB-persistent in $\mathcal{N}$ (see Proposition 4.2.1). Following Proposition 4.3.1, we conclude that α is GB-persistent in $CRG(\mathcal{N})$. $\square$

The last result guarantees the GB-persistence of steps in the concurrent reachability graph of a safe persistent PT-net $\mathcal{N}$, but the non-inclusion conditions ((GNI) and (LNI)) are not, in general, satisfied in $CRG(\mathcal{N})$ due to Fact 2.1.2. To satisfy the non-inclusion conditions, we need to prune $CRG(\mathcal{N})$, but in such a way that GB-persistence of steps is maintained. We now explore what happens if we choose to prune all but the maximal steps at every reachable marking.

In what follows, the ST-system $CRG^{max}(\mathcal{N})$ is obtained from $CRG(\mathcal{N})$, the concurrent reachability graph of a PT-net $\mathcal{N}$, by deleting at every reachable marking M, all the arcs labelled by non-maximal nonempty steps (we do not delete the empty steps for technical reasons), and then removing those nodes that became unreachable from the initial state by the removal of such steps.

Proposition 4.3.5. $CRG^{max}(\mathcal{N}) \preccurlyeq CRG(\mathcal{N})$.

Proof. Follows directly from the definitions and the fact that, for each enabled step, there is a maximal step enabled at the same marking. $\square$

Proposition 4.3.6. *$CRG^{max}(\mathcal{N})$ satisfies (*LNI*) in Problem 1.*

Proof. Follows from the fact that maximal nonempty steps are incomparable. $\square$

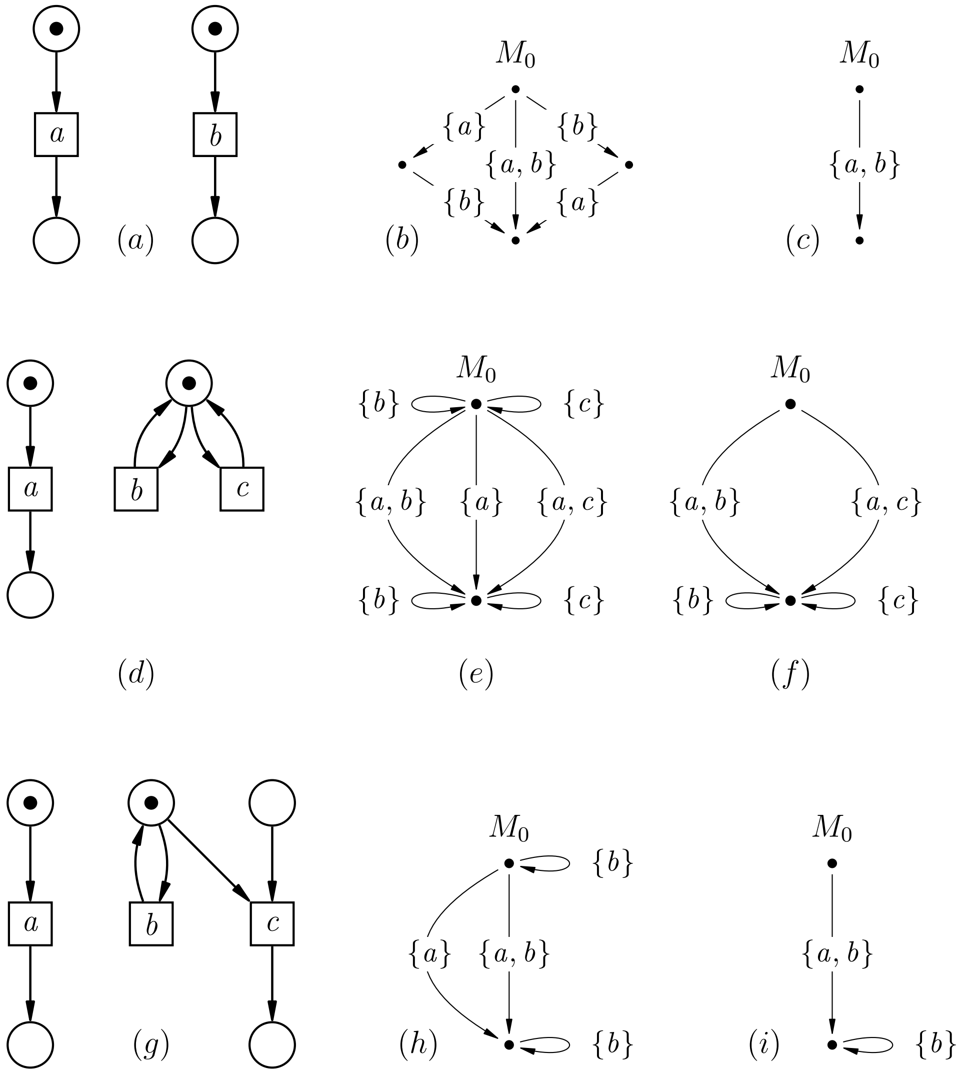


Figure 4.7: Three safe persistent PT-nets $\mathcal{N}$ (a,d,g); their concurrent reachability graphs $CRG(\mathcal{N})$ (b,e,h); and the corresponding $CRG^{max}(\mathcal{N}) \preccurlyeq^{local}_{pers} CRG(\mathcal{N})$ obtained in the pruning procedure (c,f,i).

Figures 4.7 and 4.8 show examples of persistent and safe PT-nets for which the described pruning procedure works as their $CRG^{max}(\mathcal{N})$ graphs contain only GB-persistent steps. In all these examples the pruned reachability graph satisfies $CRG^{max}(\mathcal{N}) \preccurlyeq^{local}_{pers} CRG(\mathcal{N})$, and in case of the example in Figure 4.7(a), we even have $CRG^{max}(\mathcal{N}) \preccurlyeq^{global}_{pers} CRG(\mathcal{N})$. So, the proposed pruning procedure helped to achieve local non-inclusion without jeopardising GB-persistence of the remaining steps. However, in Figures 4.7(f) and 4.8(c), the persistence in initial markings is achieved only because the steps enabled there are not disjoint, and so type-B persistence holds trivially.

In general, pruning non-maximal steps may make some of the remaining steps non-

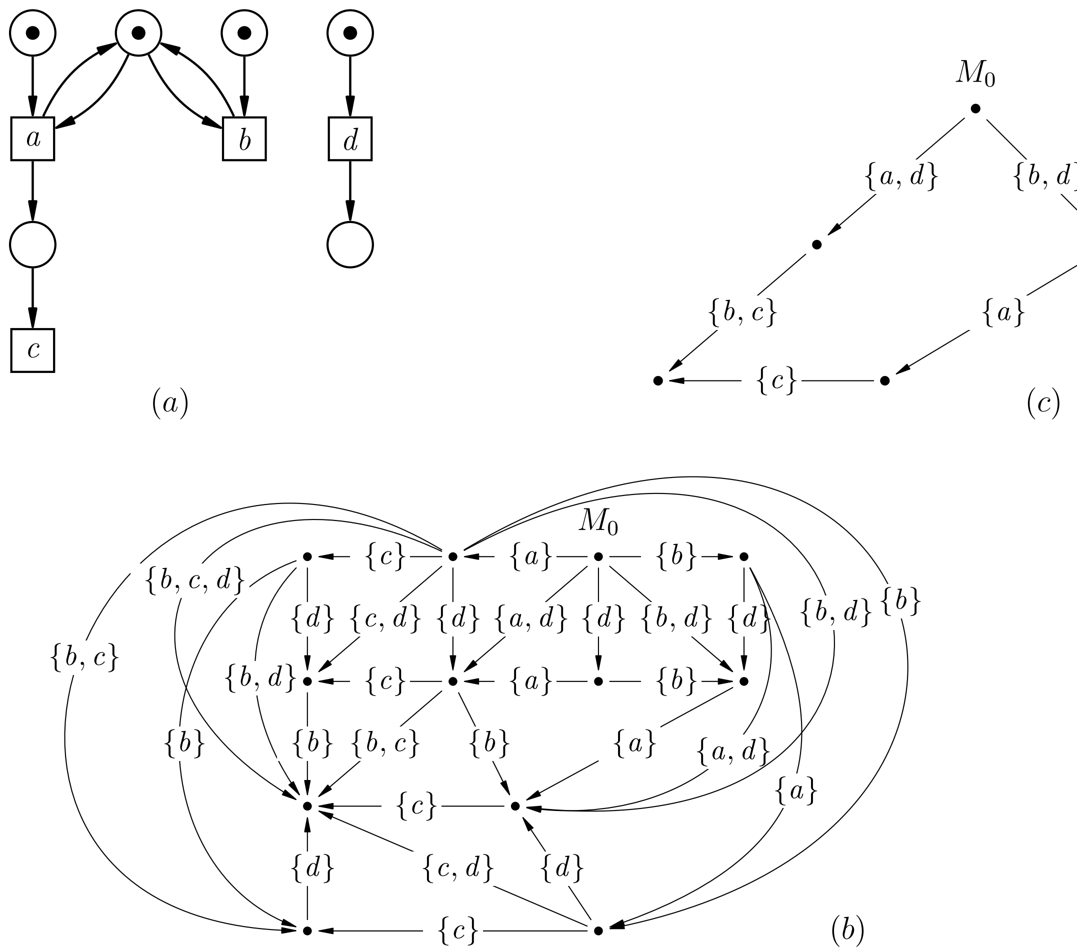


Figure 4.8: A safe persistent PT-net $\mathcal{N}$ (a); its concurrent reachability graph $CRG(\mathcal{N})$ (b); and $CRG^{max}(\mathcal{N}) \preccurlyeq^{local}_{pers} CRG(\mathcal{N})$ obtained in the pruning procedure (c).

GB-persistent. Figure 4.9(c) shows that the initially enabled step $\{b\}$ is not GB-persistent after the pruning procedure, as after executing $\{a\}$ it is not longer enabled. Instead step $\{b, c\}$ is enabled, because it was the maximal step in the marking M. We therefore propose a weaker version of condition (GB) which holds for safe and persistent PT-nets.

Proposition 4.3.7. *Let $\mathcal{N}$ be a safe* PT*-net which is persistent according to Definition 4.2.1. Then, for every marking M in $CRG^{max}(\mathcal{N})$, $M \xrightarrow{\alpha}$ and $M \xrightarrow{\beta}$ implies:*

$$(\text{GB}') \quad \beta \cap \alpha = \varnothing \quad \Longrightarrow \quad \exists \gamma :\ \alpha \subseteq \gamma \wedge M \xrightarrow{\beta\gamma} .$$

Proof. From Proposition 4.3.4 we know that $M \xrightarrow{\beta\alpha}$ in $CRG(\mathcal{N})$. Moreover, there is a maximal step γ available (as it is not removed by the pruning) after executing β from M such that $\alpha \subseteq \gamma$. Hence $M \xrightarrow{\beta\gamma}$ in $CRG^{max}(\mathcal{N})$. □

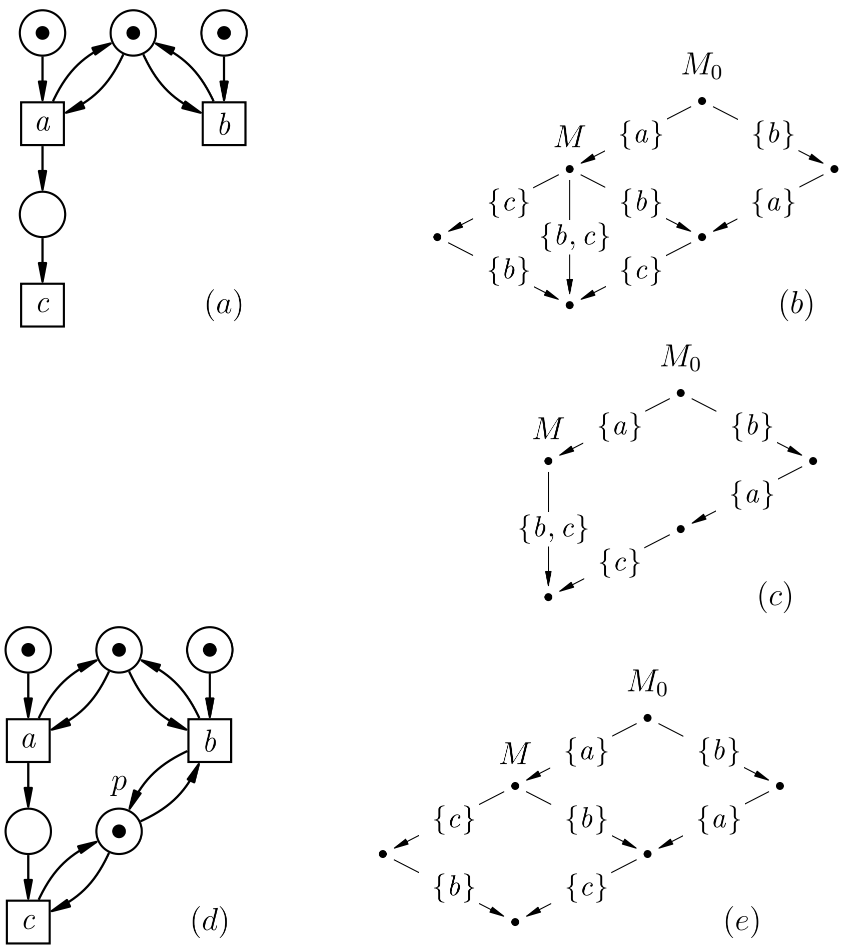


Figure 4.9: A safe persistent PT-net $\mathcal{N}$ (a); its concurrent reachability graph $CRG(\mathcal{N})$ (b); $CRG^{max}(\mathcal{N})$ obtained in the pruning procedure which does not satisfy $CRG^{max}(\mathcal{N}) \preccurlyeq^{local}_{pers} CRG(\mathcal{N})$ (c); a persistent and safe PT-net $\mathcal{N}' = \mathcal{N}_{b \leftrightarrow c}$ (d); and its concurrent reachability graph $CRG(\mathcal{N}') = CRG^{max}(\mathcal{N}')$ which trivially satisfies $CRG^{max}(\mathcal{N}') \preccurlyeq^{local}_{pers} CRG(\mathcal{N}')$ (and also $CRG^{max}(\mathcal{N}') \preccurlyeq^{global}_{pers} CRG(\mathcal{N}')$) (e).

Hence, pruning non-maximal steps may result in the loss of persistence when $\alpha \subset \gamma$ in (GB$'$). In such a case we may, however, 'repair' $\mathcal{N}$ by disabling γ. The mechanism for achieving this is simple, namely we select one transition from α, one transition from $\gamma \setminus \alpha$, and make sure that they cannot be executed in the same step.

Let $\mathcal{N}$ be a PT-net and $x \neq y$ be two transitions. Then $\mathcal{N}_{x \leftrightarrow y}$ is obtained from $\mathcal{N}$ by adding a new place p marked with one token, and such that $W(p,x) = W(x,p) = W(p,y) = W(y,p) = 1$. This construction is illustrated in Figure 4.9(d,e), where we try to fix the problem of the net $\mathcal{N}$ in Figure 4.9(a). We added a new place p and chose b and c to play the roles of x and y (the only choice in this example) creating $\mathcal{N}' = \mathcal{N}_{b \leftrightarrow c}$. The new place disables the concurrent step $\{b,c\}$ at M, leaving enabled only the singleton steps $\{b\}$ and $\{c\}$. They are now maximal steps at M. In fact, in this simple example, we have only singleton steps in the concurrent reachability graph of $\mathcal{N}'$, and so the pruning is trivial.

In the following propositions we show that after the proposed modification the PT-net generates a concurrent reachability graph which is a sub-ST-system of the reachability graph of the initial net. Also, the modified net is still safe and persistent according to Definition 4.2.1.

The two following facts result directly from definitions:

Fact 4.3.1. *Let $\mathcal{N}$ be a safe* PT*-net which is persistent according to Definition 4.2.1. Then we have*

$$CRG(\mathcal{N}_{x\leftrightarrow y}) \preccurlyeq CRG(\mathcal{N})$$

Moreover, the reachable markings of $CRG(\mathcal{N}_{x\leftrightarrow y})$ and $CRG(\mathcal{N})$ are the same, if we identify each reachable marking M of $\mathcal{N}$ with the marking $M \cup \{p\}$ of $\mathcal{N}_{x\leftrightarrow y}$. ■

Fact 4.3.2. *Let $\mathcal{N}$ be a safe* PT*-net which is persistent according to Definition 4.2.1. Then $CRG(\mathcal{N}_{x\leftrightarrow y})$ is also persistent (according to Definition 4.2.1) and safe.* ■

We can now propose a dynamic way of pruning embodied by the following algorithm:

```
while $\neg(CRG^{max}(\mathcal{N}) \preccurlyeq^{local}_{pers} CRG(\mathcal{N}))$ do
    choose $M, \alpha, \beta, \gamma$ in $CRG^{max}(\mathcal{N})$ satisfying (GB$'$) with $\alpha \subset \gamma$
    choose $x \in \alpha,\ y \in \gamma \setminus \alpha$
    $\mathcal{N} := \mathcal{N}_{x\leftrightarrow y}$
```

It follows from what we already demonstrated that the above algorithm always terminates and for the final PT-net $\mathcal{N}'$ we have:

$CRG^{max}(\mathcal{N}') \preccurlyeq^{local}_{pers} CRG(\mathcal{N}') \preccurlyeq CRG(\mathcal{N})$

and therefore:

$CRG^{max}(\mathcal{N}') \preccurlyeq^{local}_{pers} CRG(\mathcal{N})$.

Since the algorithm is non-deterministic, we may try various strategies for choosing x and y. Figure 4.10 shows that different strategies may lead to different solutions. It is possible to have a semi-lattice of pruning solutions or even arrive at a unique solution

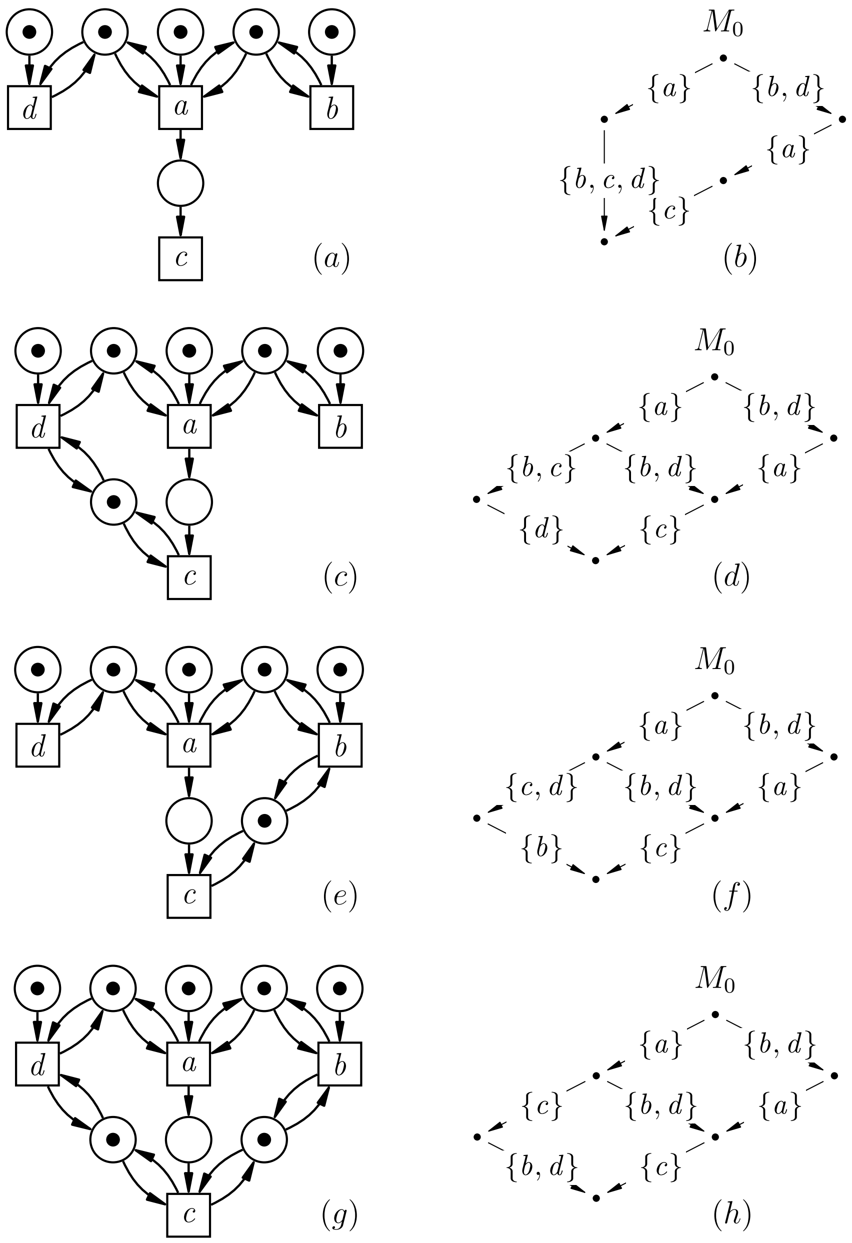


Figure 4.10: A safe persistent PT-net $\mathcal{N}$ (a); $CRG^{max}(\mathcal{N})$ not satisfying $CRG^{max}(\mathcal{N}) \preccurlyeq^{local}_{pers} CRG(\mathcal{N})$ (b); $\mathcal{N}' = \mathcal{N}_{d \leftrightarrow c}$ (c); $CRG^{max}(\mathcal{N}')$ (d); $\mathcal{N}'' = \mathcal{N}_{b \leftrightarrow c}$ (e); and $CRG^{max}(\mathcal{N}'')$ (f). Both $\mathcal{N}'$ and $\mathcal{N}''$ have been obtained as by-products of the successful runs of the pruning and modification algorithm. It is not possible to obtain in this way $\mathcal{N}'''$ in (g) even though $CRG^{max}(\mathcal{N}''') \preccurlyeq^{global}_{pers} CRG(\mathcal{N})$ (h).

by adding timing information to steps and using heuristics to determine optimal pruning strategies. For example, [100] provides a method where heuristics is used on timed marked graphs for determining optimal scheduling solutions to reduce control complexity of synchronous elastic circuits.

4.4 Significance of Bundles in Digital Circuits

Digital system design based on formal models is normally associated with two main tasks: one is the verification of a system's behavioural specification or checking the model of the system implementation, while the other is the synthesis of the circuit implementation from its specification. In the context of verification we would like, for example, to check if the Petri net model of a mixed synchronous-asynchronous system satisfies the requirement of hazard-freedom under a particular form of synchronisation of actions (in steps). In the context of synthesis, we would like to find the optimal partitioning of actions into synchronous steps so that the complexity of control of these steps is minimised. For example, the intuitive complexity of handling synchronisations safely in the three scenarios of Figure 4.2 varies between them, from the most intricate in the fully asynchronous one (case (a)) to the simplest in fully synchronous one (case (b)), placing the mixed synchronous-asynchronous version in the middle (case (c)). With this varying complexity, one can design systems that may exhibit hazards if they are treated as fully asynchronous, but when actions are synchronised into steps the system would behave safely. Amongst the methods for synchronising actions into steps, we can consider those that are based on the insertion of additional control circuits to physically 'bundle' actions together, or based on ensuring the appropriate 'bundling' constraints based on timing, or delays. Traditional globally clocked systems, self-timed systems working under fundamental mode assumptions, and asynchronous systems with relative timing [3] are all of the latter category.

It is this idea of bundling those steps of actions that are 'hazard-free' or persistent that motivated our notion of *bundles*, introduced in this chapter. In terms of nets and corresponding transition systems, bundles are informally sets of transitions that can be executed synchronously and therefore be treated as some kind of 'atomic actions', giving rise to new 'bigger' transitions. Section 4.3 provided a formal treatment for bundles and showed a constructive procedure for deriving them by pruning reachability graphs or transition systems, depending on whether we are solving the verification or synthesis problem. For example, in the process of synthesis of the control policy for a mixed synchronous-asynchronous system, such a 'pruned' transition system would represent the desired behaviour, which then we would like to implement in a form of a Petri net.

In the design of mixed synchronous-asynchronous systems, pruning a system's concurrent specification enables a designer to generate a suitable behavioural model that offers the flexibility of asynchronous design coupled with the simplicity of synchronous design. The control circuit synthesised from such a specification will hence manage a mixture of sequential, parallel and concurrent request signals that execute the bundles of the pruned system while maintaining a functionally correct system flow. Specifically, a bundle deals with the execution of a single action or simultaneous execution of multiple actions in a single clock domain, while concurrent interactions between distinct bundles indicate synchronisation between different clock domains. For example, let us consider the Petri net model in Figure 4.11(a) which satisfies the temporal specification given in Figure 4.2(c). Actions $\{a\}$ and $\{b\}$ belong to different clock domains and have been specified to execute asynchronously. However, their completions must be synchronised before the execution of bundle $\{c, d\}$ which belongs to a third clock domain. Another mixed synchronous-asynchronous implementation of the same system is depicted behaviourally in Figure 4.11(b) and its derived Petri net model is shown in Figure 4.11(c). To give an idea of the practical implications of pruning in digital system design, the two pruned examples can be viewed as a classic case of trade-off between power and latency. A scenario where the latter implementation would be beneficial is the reduction of electromagnetic interference (EMI) which is caused by the simultaneous switching of circuit elements [14]. Here, if the concurrent execution of actions $\{a\}$ and $\{b\}$ leads to higher peak power as compared to the execution of bundle $\{c, d\}$, then the designer would want to order the bundles as indicated. This would be particularly intuitive when considering a complex system where many such blocks exist and a pruned behavioural specification is required to ensure low spectral energy peaks.

It is essential that pruning is performed in a step persistent manner, according to the rules introduced in this chapter. Each rule will ensure a distinct case of hazard-free step execution and also affect control circuit complexity differently. To briefly motivate how these rules affect mixed synchronous-asynchronous circuits, we discuss the case of (LNI) using the examples given in Figure 4.10. It should be noted that the enabling of a step, when translated to circuit behaviour, results in switching the concerned request signal to a logical 1, triggering the execution of its comprising actions. The following paragraph explains how a non-monotonic pulse can arise if specific countermeasures are not considered in the circuit synthesis procedure.

The case of local non-inclusion of steps is compared with the case of global non-inclusion taking Figure 4.10(b) and Figure 4.10(h), respectively. In the former specification, the execution of $\{a\}$ and $\{b, d\}$ is concurrent but not simultaneous due to shared resources, as indicated in the Petri net model shown in Figure 4.10(a). If the arbitration of the shared resource favours executing action $\{a\}$, the step $\{b, d\}$ should get its resource after the completion of action $\{a\}$. However, owing to the given temporal specification, a new request signal pertaining to step $\{b, c, d\}$ will be generated, disabling the request of step $\{b, d\}$. This results in a non-monotonic pulse, as illustrated in Figure 4.12, which could lead to a hazardous computation as it is uncertain how the system will interpret the glitch. Such a glitch will not occur when executing the specification of Figure 4.10(h), as the action $\{b, d\}$ that was enabled does not get disabled until execution and hence is step persistent.

Now, if the former specification is a requirement for the design, only an arbitration between the request signals for the two steps $\{b, d\}$ and $\{b, c, d\}$ can ensure hazard freedom. This, however, complicates control circuit synthesis due to the requirement of an extra arbitration block in the control circuit which would not the case when implementing the

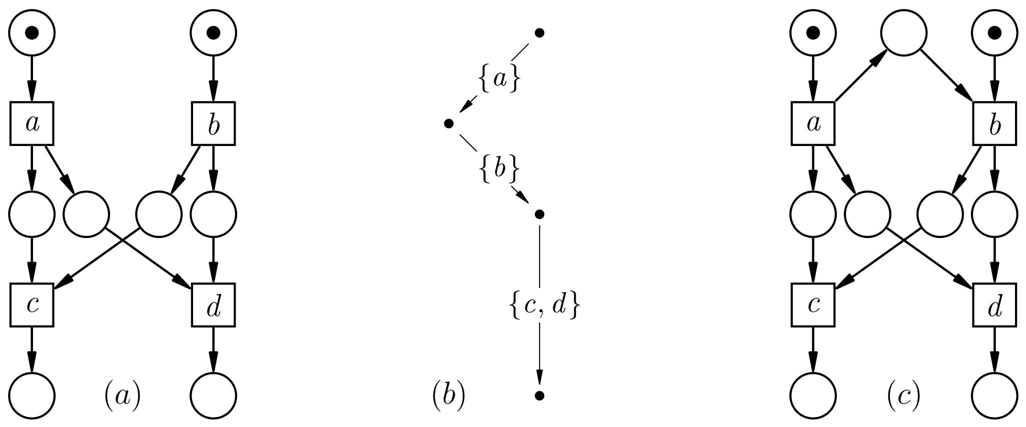


Figure 4.11: A PT-net generating the concurrent reachability graph of Figure 4.2(c) under the assumption that a and b belong to two different clock domains, while c and d belong to a third clock domain (a); and another mixed synchronous-asynchronous implementation of the same system (assuming the same clock domains as in (a)) both behaviourally (b) and as a derived PT-net (c).

latter. The Opportunistic Merge element [101] is an example of an arbitration block that implements hazard-free inclusion of steps. This circuit element is designed to implement opportunistic bundling of closely arriving input signals.

Step persistence therefore plays a vital role in the two main aspects of formal modelling of digital systems: in the verification of mixed synchronous-asynchronous systems to ensure freedom from hazards, and in the synthesis of mixed synchronous-asynchronous circuit implementation to ensure optimal control circuitry.

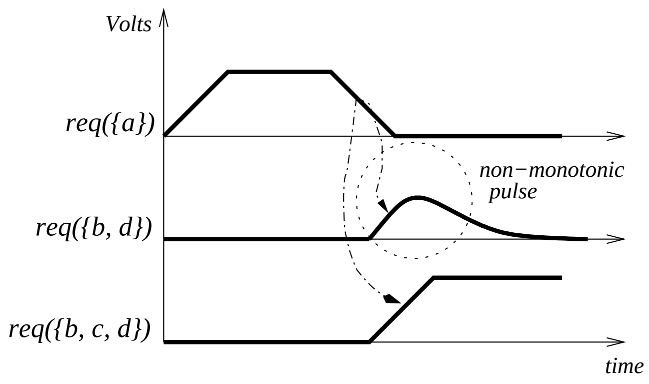


Figure 4.12: Hazardous switching in mixed synchronous-asynchronous step execution semantics.

4.5 Conclusions and Future Work

In mixed synchronous-asynchronous circuits, bundling is envisaged to reduce signal management, and could reduce the cost of scheduling and control, and improve system performance. The ideal way to model mixed synchronous-asynchronous systems is to start with a concurrent model that is persistent and fully asynchronous in behaviour. Then run several iterations that derive a combination of bundles that represents the temporal nature the designer requires. Careful selection of bundles is essential so that the pruned behaviour of the fully asynchronous model still exhibits some characteristics of its parent and is persistent. *Step persistence* is hence an important characteristic that will guarantee true persistent behaviour for mixed synchronous-asynchronous models.

In this chapter, we developed a pruning procedure for reachability graphs of persistent and safe nets. This procedure constructs a step transition system that contains only bundles. The bundles in our algorithm represent maximally concurrent steps of the initial system that are persistent and satisfy non-inclusion constraints. We hope that the theory presented in this chapter will pave the way towards formal design and verification of mixed synchronous-asynchronous systems. Right now, we are not trying to comprehensively describe how this theory would be applied in the scenarios of verification and synthesis mentioned in the introduction. This will be done in our future research, where will have to answer many new questions arising on the way, including, for example, what a rigorous metric for the complexity of bundle control is, how the notions of maximal steps (global and local) affect such a complexity, or what the different forms of step persistence (type-A, type-B and type-C) imply in terms of hazard-avoidance in the system.

A move into the realm of step based execution semantics creates a wealth of new fundamental problems and intriguing questions, some of which have been addressed in [102, 103, 104]. In particular, there are different ways in which the standard notion of persistence could be lifted from the level of sequential semantics to the level of step semantics. For example, if part of an enabled step has been executed by another step, should we insist on the whole delayed step to be still enabled, or just its remaining part? Moreover, one may consider steps which are persistent and cannot be disabled by other steps, as well as steps which are nonviolent [99, 95, 103, 104] and cannot disable other steps.

In the future, we intend to investigate other possible pruning algorithms, weakening our constraints and allowing the initial system's behaviour to be given by a net that is not necessarily persistent. Furthermore, we plan to allow in the algorithms the choice of non-maximal bundles in certain cases. For example, input signals are usually behaving in fully asynchronous way and should not be bundled. We will also explore the practical implications of bundles on the verification and synthesis of mixed synchronous-asynchronous circuits such as fixing hazardous circuits by bundling.

In the next chapter, we employ the concept of bundles for functional partitioning of digital circuits and demonstrate degrees of elasticity in circuits synthesised with bundles.

Chapter 5

Synthesis of Asynchronous Circuits with Granular Rigidity

For decades, GALS circuits have represented the class of mixed synchronous-asynchronous circuits. This methodology, however, has been shown to have limitations and under-represent the mixed synchronous-asynchronous paradigm. In this chapter, we were hence motivated to introduce a new class of asynchronous circuits exhibiting synchrony by method of granular rigidity, and propose a novel method to synthesise these circuits based on functional partitioning of their asynchronous specification.

5.1 Introduction

Mixed synchronous-asynchronous design can be viewed as a revolutionary step in VLSI design paving the way for commercial adoption of asynchronous design techniques in future SoCs. Chapter 1 illustrated this paradigm with a classification of synchronous, asynchronous and mixed synchronous-asynchronous circuits. Each technique has distinct consequences to system behaviour and design implementation as illustrated in Figure 1.1. This figure was motivated from the classification of 'Elastic Circuits' made in [31]. In this chapter, our focus is on elastic circuits that provide elasticity with small overhead, specifically the group of circuits encapsulating bundled-data circuits and synchronous handshake circuits, as highlighted in the shaded area of Figure 1.1.

We classify the low-overhead elastic circuits paradigm to encapsulate a range of design

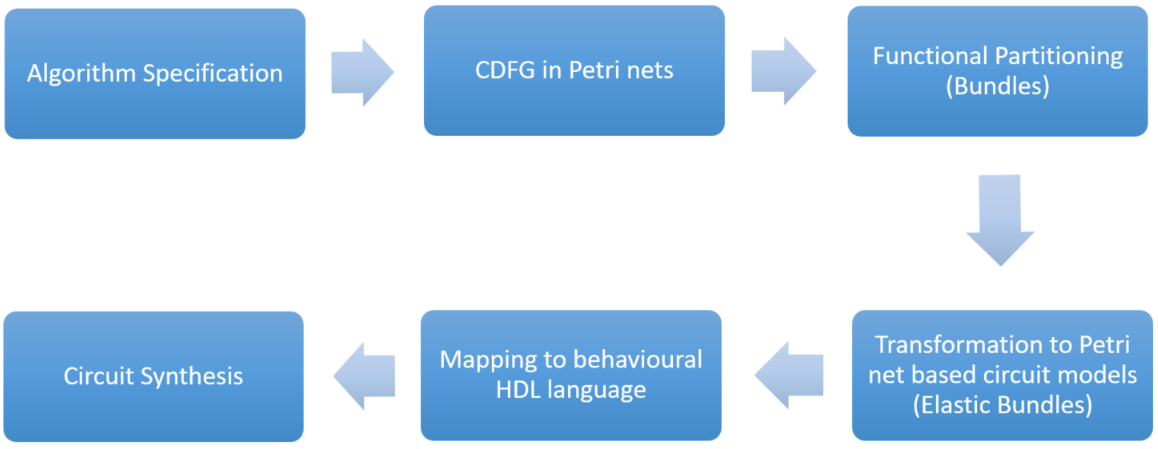

Algorithm Specification
CDFG in Petri nets
Functional Partitioning (Bundles)
Transformation to Petri net based circuit models (Elastic Bundles)
Mapping to behavioural HDL language
Circuit Synthesis

Figure 5.1: Design methodology flow

practices starting from synchronous handshake circuits [26], which are synchronous circuits embracing principles of asynchronous elasticity, all the way to bundled-data circuits [45, 46], which are asynchronous circuits embracing principles of synchronous rigidity. Bundled-data circuits are a class of asynchronous circuits that resemble synchronous behaviour due to its nature of fine-grained local clocking scheme. They can be implemented with the same data path used in synchronous circuits and only differ in the implementation of clock. Several authors have proposed schemes to implement bundled-data circuits using the clocked CAD flow [105, 106, 28, 32]. Synchronous handshake circuits fall in the class of synchronous elastic circuits where asynchronous-like elasticity is implemented in a globally clocked domain. Synchronous handshake circuits have been formalised and investigated by several authors in [26, 107, 108, 29, 31, 109]. Design techniques on either end of the classification have been well researched with strong theoretical and mathematical foundations being laid along with automated synthesis flows integrated into standard synchronous EDA tool flows.

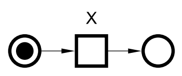
This chapter introduces 'Elastic Bundles', a novel class of asynchronous circuits with varying levels of rigidity. Elastic-bundle circuits exhibit granular rigidity through a combination of coarse-grained locally clocked elements and fine-grained locally clocked elements. Modelling these circuits starts from a high-level dataflow model of a system which is intrinsically asynchronous. The key idea is to introduce rigidity of chosen granularity levels across the model, without changing functional behaviour, for achieving the design simplicity of synchronous principles. In this manner, the system is partitioned into functional blocks of fine-grained and coarse-grained asynchronous elements based on functional criteria. Figure 5.1 depicts the design flow of this methodology.

In this chapter, we focus purely on bundled-data asynchronous circuits, as this circuit classification was identified to display the finest granularity of elasticity in the class of low-overhead elastic digital circuits. Bundled-data circuits emerged as a design simplification practice for asynchronous circuits by bundling control logic with datapath. Here, such circuits are optimised further by identifying groups of circuit elements that can be bundled in succession without changing functional behaviour. Such bundling, named as *Elastic Bundles*, is implemented by performing coarse-grain local clocking on bundled-data circuit models. Furthermore, this design style can share the same data path used for synchronous design as opposed to other asynchronous circuit classes [46]. This is a very strong feature because it enables reuse of existing synchronous functional blocks improving designer productivity. Moreover, a bundled-data design can be synthesised using traditional clocked CAD flows by replacing the global clock of the synchronous flow with multiple number of local handshake clocks from the asynchronous control logic [32]. Elastic-bundle circuits can be synthesised in a similar manner using such methods of synchronous-asynchronous CAD tool flow integration. In this paper, we have adopted the *relative timing* (RT) based CAD flow [1] for the synthesis of elastic circuits. Our method, however, is not limited to this tool flow; any other synchronous-asynchronous CAD tool flows can be applied to synthesise elastic-bundle circuits.

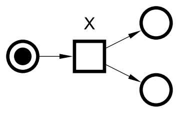
The chapter is organised as follows. Section 5.2 introduces the Petri net modelling framework to describe digital circuits. Section 5.3 provides the method for synthesising elastic-bundle circuits from Petri net specifications. Section 5.4 presents the results of implementing the proposed methodology on a 16-point FFT design. Section 5.5 contains conclusions and presents directions for future work.

5.2 Modelling Digital Systems

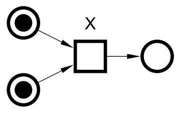
In our design flow, the Petri net (PN) modelling tool is the chosen mathematical language to describe digital systems. The language offered us a convenient tool to capture essential properties of a system from theory formulation to synthesis of control circuits. The



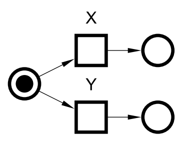
(a) Enabled Transition



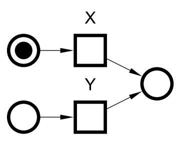
(b) Fork



(c) Join

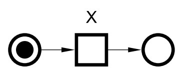


(d) Choice

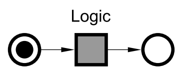


(e) Merge

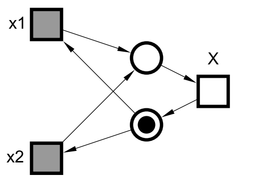
Figure 5.2: Basic PN elements



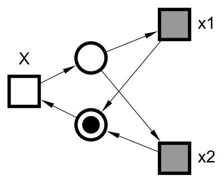
(a) Buffer



(b) Computation



(c) Multiplexer



(d) Demultiplexer

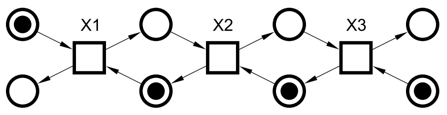
Figure 5.3: PN building blocks for digital systems

WORKCRAFT framework [110, 111, 112] has been used in this research for graphical interpretation and simulation of Petri nets, and for verification of important properties of Petri nets.

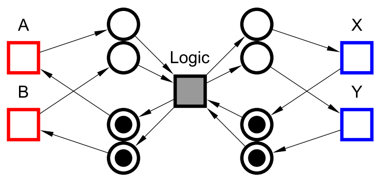
5.2.1 PN building blocks

We employ PT-net class of Petri nets in this thesis. The notations, definitions and basic properties of these nets have been introduced in Section 2.1.2. PT-nets employ token game semantics to describe net behaviour. System behaviour is captured by flow of tokens according to enabling and firing rules. Figure 5.2 shows some of the fundamental elements of PT-nets that are widely used in modelling systems [49].

With regards to a sufficient abstract model to describe digital systems, we wanted to use the most compact form of PT-nets for both asynchronous and synchronous implementations. Digital systems were hence modelled to represent system behaviour on



(a) Three stage pipeline



(b) Computation with multiple inputs and multiple outputs

Figure 5.4: Example of PN model designs

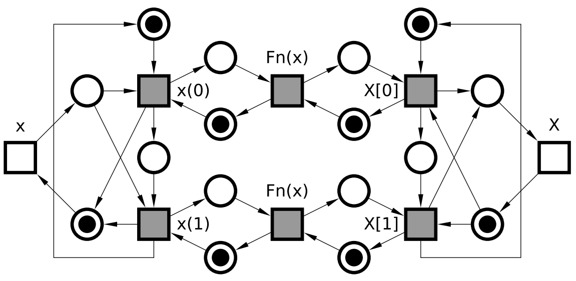


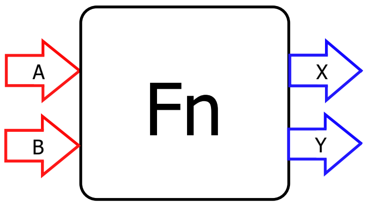
Figure 5.5: Wagging [2]

a high-level abstraction of dataflows and flow control, rather than register-transfer level (RTL) description. Data structures and timing information are excluded in the language description. Such information is labelled on the graph for designer reference. Safe class of PT-nets were employed which ensure that each place can have utmost 1 token. Based on this, we have extracted a set of elements shown in Figure 5.3 that form the building blocks for digital systems in our framework. The buffer is used for data storage and channel decoupling. Computation blocks are used to denote digital logic from basic gates to complex calculations. The multiplexer and demultiplexer provide capability to route multiple data streams and computations to a single stream, or split a single stream to multiple streams. For simplicity, the control switch logic for multiplexer and demultiplexer is not represented in the PN models. Since firing of transitions in PT-nets is instant and atomic, we have colour-coded logic blocks in grey so that differentiation can be made during design optimisation. We also colour-code input places and transitions in *red* and output places and transitions in *blue*. Figure 5.4 and Figure 5.5 show examples of designs built using these building blocks. In such a way, a wide range of digital designs can be built using PNs. Sokolov and Yakovlev listed several other interaction patterns of system behaviour represented with PNs [23].

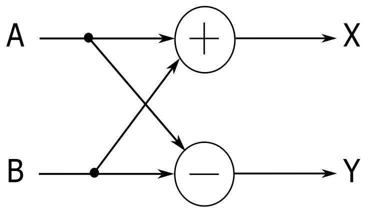
5.2.2 Modelling a Conceptual Design

In this section, we move on to a specific design example to explain the research concepts of this chapter. Let us try to represent a digital system designed to executed a function Fn depicted in Figure 5.6a. The system processes two input signals,A and B, to produce outputs X and Y which are defined as: $X = A + B$, $Y = A - B$

The design is first modelled as a data flow graph (DFG) and subsequently, as a PN to



(a) System specification



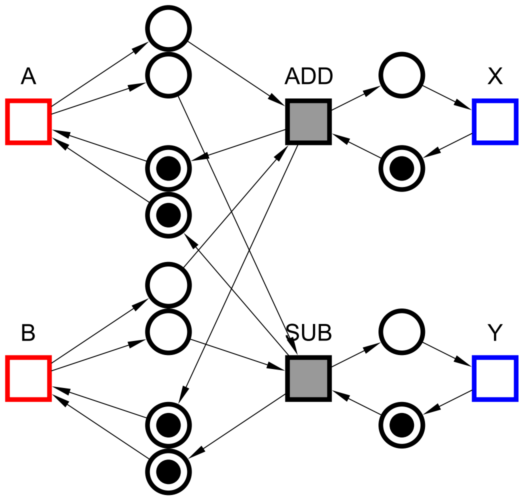
(b) Data flow graph

Figure 5.6: Conceptual design

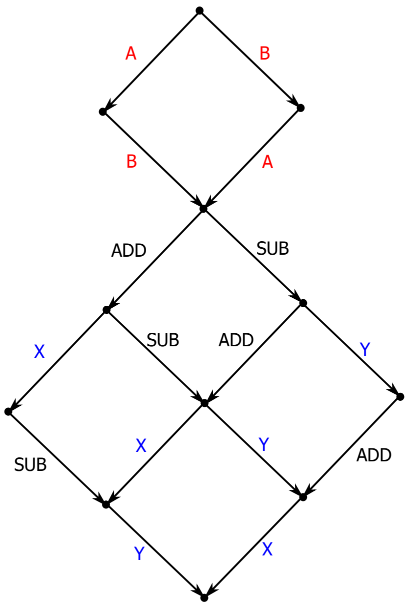
illustrate the significance of modelling flow control. DFGs [113, 114] have been widely used to describe digital systems but they are limited as they can not model control sufficiently. A basic DFG of the system is shown in Figure 5.6b. PN models represent what is known as a control data flow graph (CDFG). Figure 5.7a shows the CDFG of the conceptual design modelled as a PN. The functional behaviour of the design is depicted in Figure 5.7b based on scoping of reachable states. It can be visualised that the PN model describes the digital system in its native form featuring highest level of concurrency. Hence, we also treat this model as a purely elastic or asynchronous description of the design.

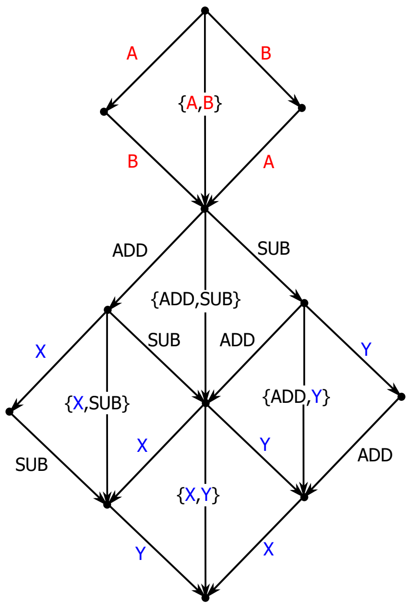
5.2.3 Partitioning with Bundles

In this section, we implement the principles of step persistence and bundles to partition the conceptual design based on functional criteria. Bundles are identified by pruning a concurrent reachability graph (CRG) of a system into a set of persistent steps. The CRG of the design example is shown in Figure 5.7c. By observing the CRG, it makes intuitive sense that the steps *ADD* and *SUB* should be bundled because 1) they are enabled together and 2) their concurrent execution can be restricted to parallel execution because they can be synchronised to complete at a worst-case delay without reducing circuit performance. However, there would be certain situations where data-dependent computation delays or variable-latency ADD/SUB circuit implementations could be exploited to improve system performance or energy efficiency. Such scenarios would require special cases of bundling with some form of run-time adaptability, or simply run-time bundling, that would trigger control for faster synchronisation between pipeline stages



(a) CDFG of design modelled as a PN





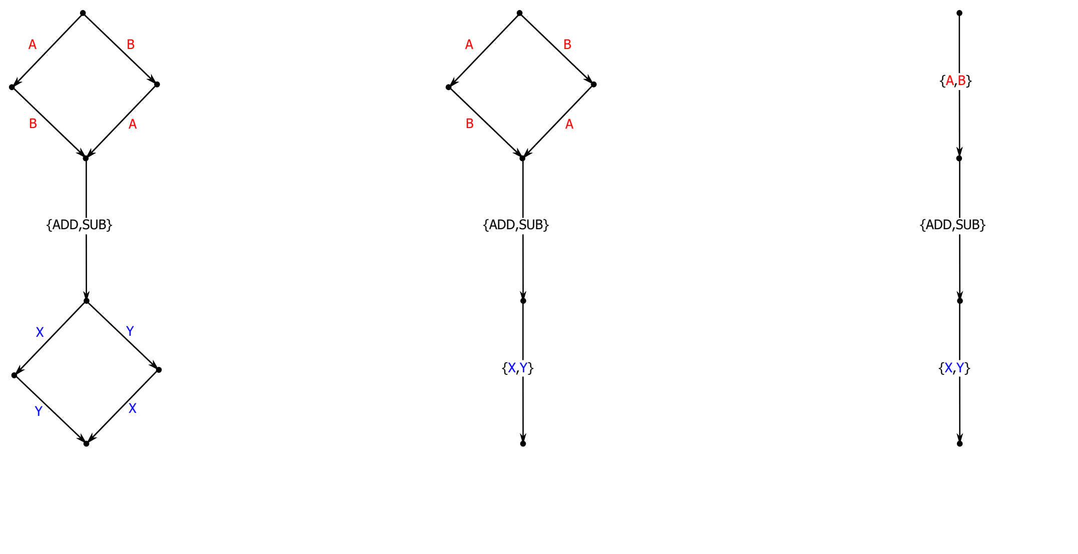
(b) Reachability graph

(c) Concurrent reachability graph

Figure 5.7: PN models of conceptual design

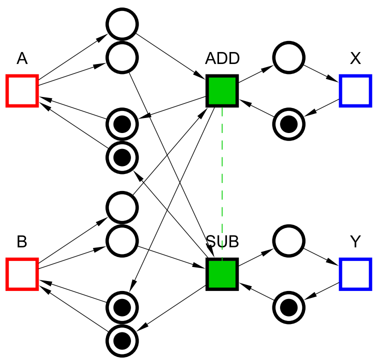
or trigger switching between appropriate situation-driven circuit implementations. For example, research on adaptive parametrised circuits [115] demonstrates the practicality of run-time adaptability of circuits to wide range of operating conditions. This particular area of research is outside the scope of this thesis and will be explored in the future.

The choice of bundling the inputs and outputs depends purely on where the signals arrive from and go to. For instance, if the inputs arrive from different timing domains, it is not possible to bundle the inputs. Similarly, bundling of outputs depends on the nature of the system receiving these output signals. We have considered three pruning behaviours as shown in Figure 5.8 to study the impact of bundling on digital circuit realisation. Step persistence property was checked to avoid hazardous bundling. The implication of these three variations of bundle sets on the PN model is shown in Figure 5.9. Different partitioning scenarios of the design example can be visualised. By bundling, the transitions of a partition would only fire in a step, *i.e.*, they would only execute together in a lock-step manner. Partitioning and step firing behaviour have been described using a novel extension to Petri nets, called Policy nets. Policy nets are, basically, Petri nets with step firing policies. This net classification was introduced into the WORKCRAFT framework during the research phase of Chapter 4. Support for the usage of Policy nets can be found in [112].

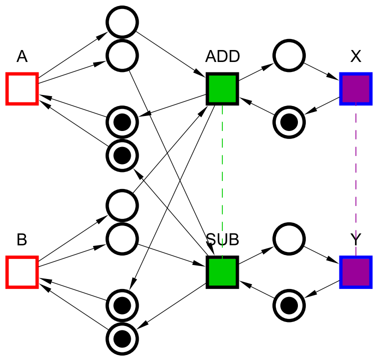


(a) Bundle set 1 (b) Bundle set 2 (c) Bundle set 3

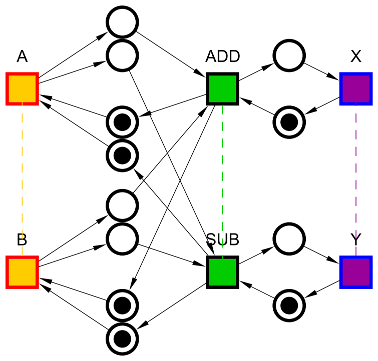
Figure 5.8: Pruned reachability graphs describing bundles



(a) Bundle set 1



(b) Bundle set 2



(c) Bundle set 3

Figure 5.9: Modelling bundles with Policy Nets

5.3 Digital Circuit Synthesis from PN Models

In this section, we describe our method for synthesising elastic digital circuits from PN models. The key idea here is to introduce synchronous-like rigidity into the asynchronous PN models by functional partitioning with bundles without changing functional behaviour. Granularity of these partitions can be varied according to the level of elasticity and rigidity required by the designer. By starting with a pure asynchronous model of a design, we can enforce varying levels of granular rigidity which could eventually transcend to a pure synchronous specification.

5.3.1 Model Transformation to Asynchronous Pipeline Models

The first step in circuit synthesis is to transform the PN specification into asynchronous pipeline models. We intentionally did not include synchronous clocking or asynchronous handshaking in the the PN models in Section 5.2 so that a designer can focus on functional elements and not worry about the aspects of circuit timing implementation. In this section, we introduce hardware description language (HDL) elements into the PN model such as registers, combinatorial logic, control logic and clocking information.

We have focused mainly on bundled-data asynchronous pipelines in this research, as this circuit classification was identified to display the finest granularity of elasticity in the class of mixed synchronous-asynchronous digital circuits, as discussed in Chapter 1. This design style can share the same data path used for synchronous design as opposed to other asynchronous circuit classes [46]. This is a very strong feature because it enables reuse of existing synchronous functional blocks improving designer productivity. Moreover, a bundled-data design can be synthesised using traditional clocked CAD flows by replacing the global clock of the synchronous flow with multiple number of local handshake clocks from the control logic [32].

The first step in transformation is to introduce pipelining into the PN models. Pipelining is conducted without changing the behaviour of the system based on the principles of slack elasticity [116]. Handshaking happens between neighbouring pipeline stages and so, distinction between registers and combinatorial logic is made. The bundled-data asynchronous control elements of handshake control and matched delay are incorporated next. Handshake control signals signifying the *HC* block (introduced in Chapter 2) are indicated in the model. Finally, the forks and join in the CDFG are now mapped to control paths. In this manner, a PN-based CDFG model of a digital design can be transformed to its PN-based HDL-like description.

Figure 5.10 depicts the transformation of the conceptual design from a CDFG to its HDL specification. Pipeline registers A, B, X and Y have been introduced. *HC* control signals manage the handshake clocking of these registers. Distinction between data path and control logic can now be visualised more clearly. Here, the transformation and design verification is done manually at this moment. This would be automated in the future for design productivity.

5.3.2 Partitioning into Elastic Bundles

In this section, the implication of bundle transformation to digital circuits is shown. The transformations of bundle set partitions 1, 2 and 3 are shown in Figures 5.11, 5.12 and

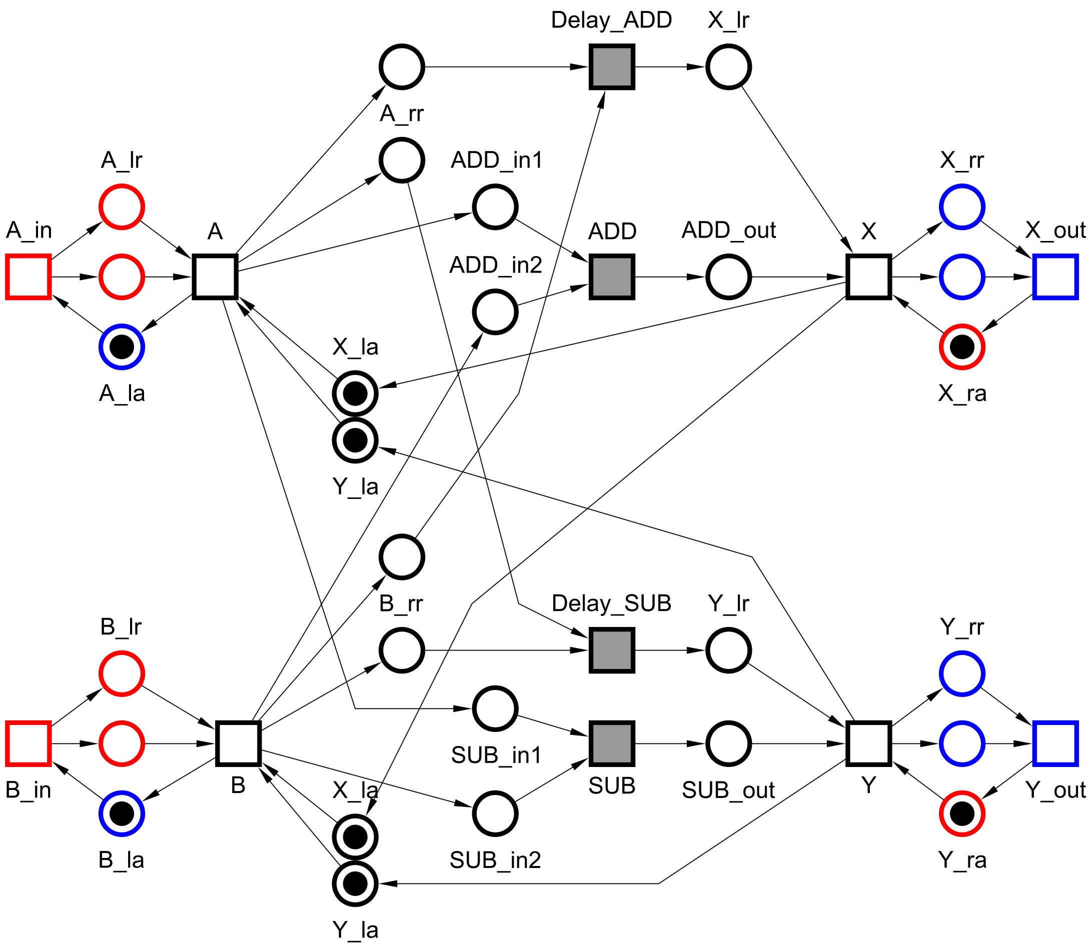


Figure 5.10: Transformation to bundled-data asynchronous pipeline

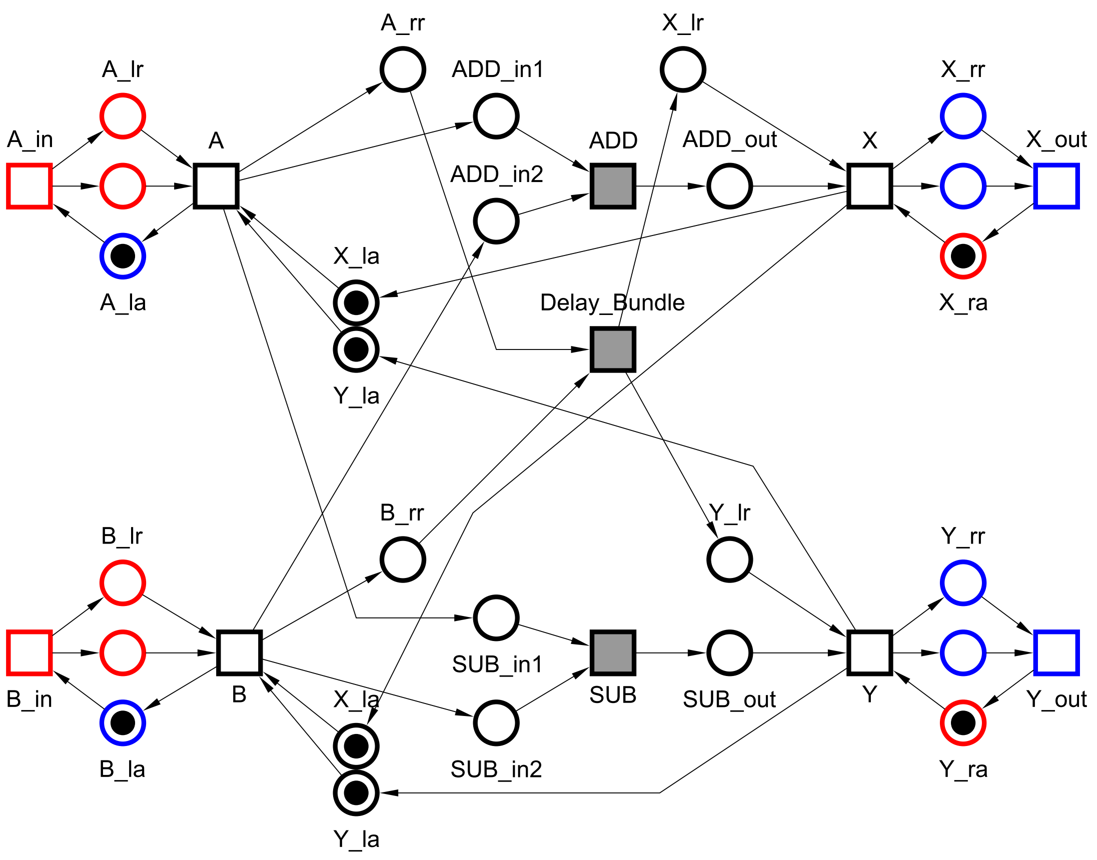


Figure 5.11: Elastic-bundle pipeline transformation (bundle set 1)

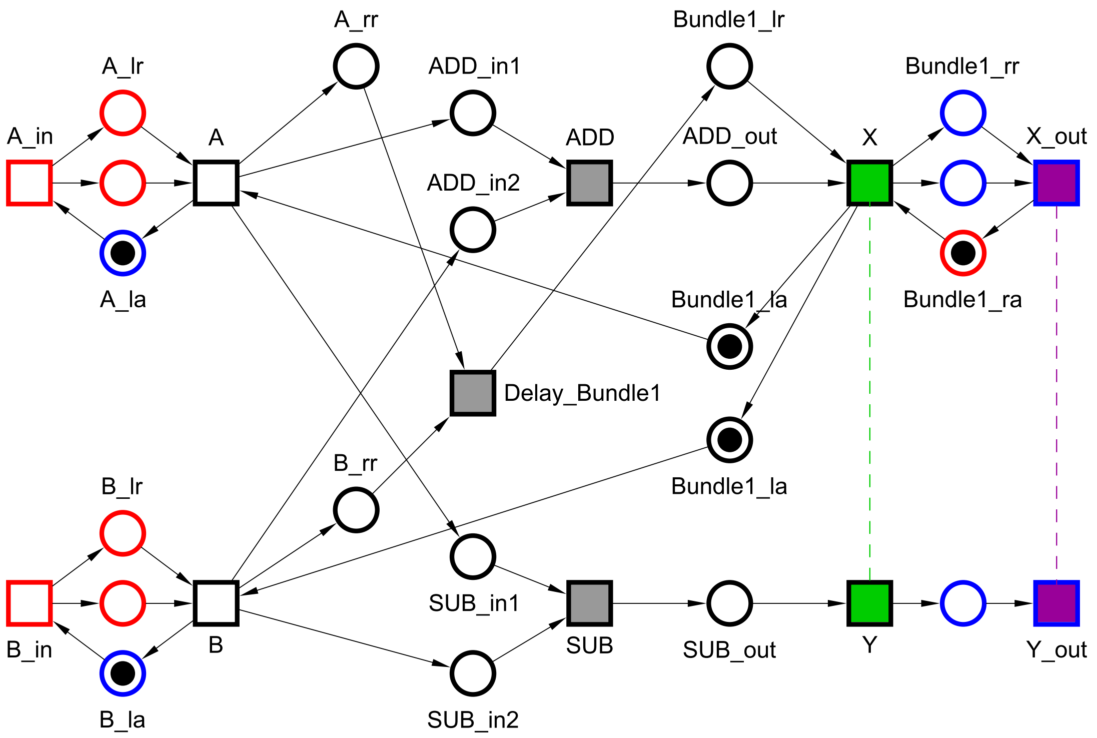


Figure 5.12: Elastic-bundle pipeline transformation (bundle set 2)

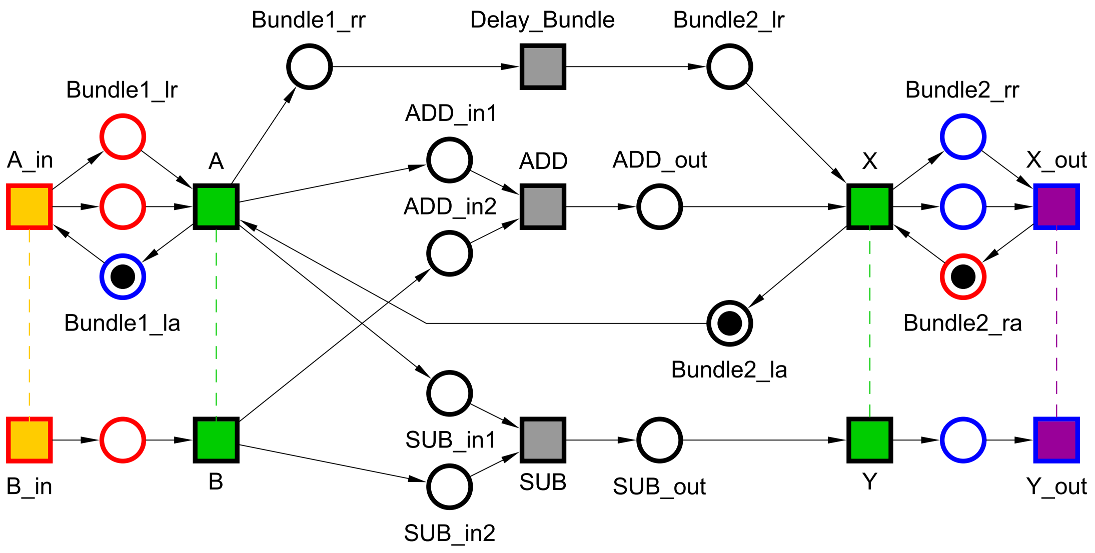


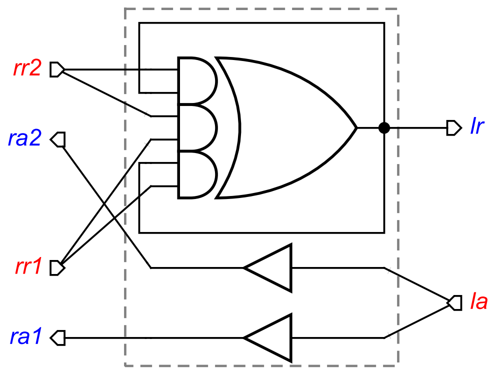
Figure 5.13: Elastic-bundle pipeline transformation (bundle set 3)

5.13, respectively. It can be seen that bundle partitions do not necessarily offer the same partitions in the transformed graph. For instance, in Figure 5.11, according to bundle set 1, bundling of ADD and SUB did not reflect accordingly in the transformation. This was because the asynchronous nature of inputs and outputs makes it impossible to fire the register X and Y in step.

We can now visualise how partitioning into bundles results in lower control overhead. Registers can be controlled by fewer handshake signals by bundling the requests and acknowledge signals. The transformed PN model can thus be viewed as a mixture of coarse-grained locally clocked elements and fine-grained locally clocked elements. Our outlook is that the coarse-grained locally clocked elements are synchronous to a degree of rigidity viewing them as synchronised actions sharing a common clock signal, and the fine-grained locally clocked elements are asynchronous and elastic. The transformed circuits hence exhibit mixed synchronous-asynchronous characteristics but remain asynchronous globally. We introduce such sets of re-partitioned bundles exhibiting granular rigidity in bundled-data pipelines as ***Elastic Bundles***.

5.3.3 From PN Models to Digital Circuits

In this section, we discuss the method employed to synthesise the PN-based HDL description into digital circuits using standard clocked CAD tools. The RT-based synchronous-asynchronous EDA tool flow, summarised in Section 2.2.3, is used for synthesising the



1

Figure 5.14: Fork join circuit

elastic-bundle circuit.

The first step is to create and characterise the asynchronous control elements, specifically the HC blocks and fork/join elements. Pre-built *HC* blocks borrowed from the RT tool flow is utilised. The characterised *HC* element introduced in Figure 2.6a is used here. Fork/Join elements are required for implementing the control of non-linear pipelines. These elements are introduced with the rule that every fork in the data path is associated with a join, and every join in a data path associated with a fork [32]. The characterised fork join elements are shown in Figure 5.14. The join elements employ Muller's C-elements [97] for synchronising request signals as well as synchronising acknowledge signals. The complex gate in Figure 5.14 depicts an implementation of the Muller's C-element.

Next, the PN-based HDL model is mapped to a behavioural HDL language such as Verilog. Active high latches are used for register implementation. In the case of our conceptual design, Figure 5.10 was thus manually mapped into the following behavioural Verilog description:

```
module Fn (A_in, A_lr, A_la, B_in, B_lr, B_la,
    X_out, X_rr, X_ra, Y_out, Y_rr, Y_ra, rst);
    input [31:0] A_in, B_in;
    output [31:0] X_out, Y_out;
    input A_lr, B_lr, X_ra, Y_ra, rst;
    output A_la, B_la, X_rr, Y_rr;
    wire [31:0] A, B, X, Y, ADD1_in1, ADD1_in2, SUB1_in1, SUB1_in2;
    // Datapath Logic
    reg32_async R1 (.D(A_in), .Q(A), .ck(A_ck), .rst(rst));
    reg32_async R2 (.D(B_in), .Q(B), .ck(B_ck), .rst(rst));
    assign ADD1_in1 = A;    assign ADD1_in2 = B;
    assign SUB1_in1 = A;    assign SUB1_in2 = B;
    assign X = ADD1_in1 + ADD1_in2;
    assign Y = SUB1_in1 - SUB1_in2;
    reg32_async R3 (.D(X), .Q(X_out), .ck(X_ck), .rst(rst));
    reg32_async R4 (.D(Y), .Q(Y_out), .ck(Y_ck), .rst(rst));
    // Control Logic
    // Input Handshake Control
    handshake_ctl HC1 (.lr(A_lr), .la(A_la), .rr(lr1), .ra(la1), .ck(A_ck), .rst(~rst));
    handshake_ctl HC2 (.lr(B_lr), .la(B_la), .rr(lr2), .ra(la2), .ck(B_ck), .rst(~rst));
    // Control Stage 1 (Fork - Join)
    assign r11 = lr1;    assign r12 = lr1;
    assign r21 = lr2;    assign r22 = lr2;
    Celement j1 (.in1(a11), .in2(a21), .out(la1));
    Celement j2 (.in1(a12), .in2(a22), .out(la2));
    // Control Stage 2 (Join - Fork)
    Celement j1 (.in1(r11), .in2(r21), .out(lr3_pre));
    Celement j2 (.in1(r12), .in2(r22), .out(lr4_pre));
    assign a11 = la3;    assign a12 = la3;
    assign a21 = la4;    assign a22 = la4;
    // Delay elements for 32 bit adder & subtractor pipeline
    DelayElement d1 (.in(lr3_pre), .out(lr3));
    DelayElement d2 (.in(lr4_pre), .out(lr4));
    // Output Handshake Control
    handshake_ctl HC3 (.lr(lr3), .la(la3), .rr(X_rr), .ra(X_ra), .ck(X_ck), .rst(~rst));
    handshake_ctl HC4 (.lr(lr4), .la(la4), .rr(Y_rr), .ra(Y_ra), .ck(Y_ck), .rst(~rst));
endmodule
```

By employing such an HDL specification, the design can now be synthesised using the RT-based EDA tool flow which would result in the circuit shown in Figure 5.15. Similarly, elastic-bundle sets 1, 2 and 3 modelled in Section 5.3.2 would result in circuits depicted in Figures 5.16, 5.17 and 5.18 respectively, after synthesis. Figure 5.17 is partitioned into three asynchronous domains with HC_3 block introducing a coarse-grain level of rigidity on registers R_3 and R_4. The partitioning in Figure 5.18 results in two asynchronous domains both with the same level of granular rigidity. Note that this particular implementation behaves in the manner of a linear pipeline.

Intuitively, one would notice that the first level of bundling is trivially carried out in bundled-data design where a designer bundles the control signals for latching the flip-flops of a register. Elastic-bundle design extends this notion to bundle functionally related elements of a design, spatially or temporally or both.

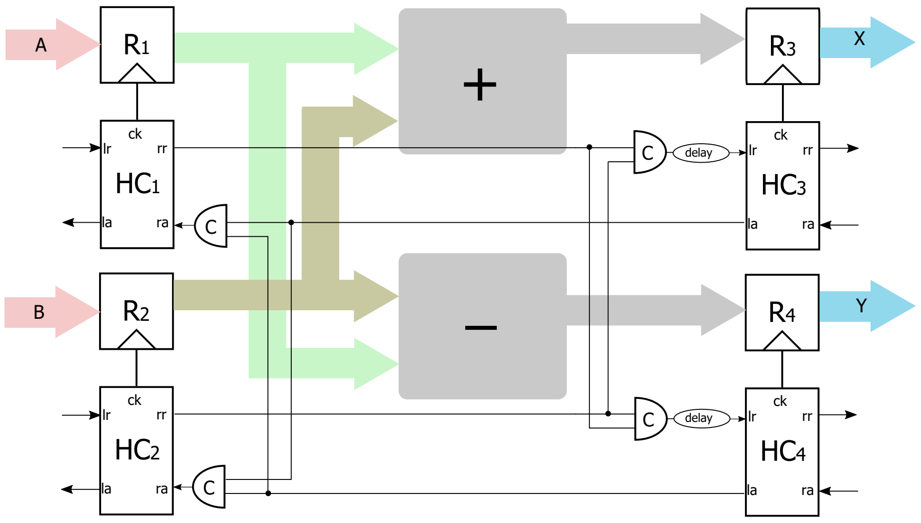


Figure 5.15: Bundled-data asynchronous circuit of conceptual design

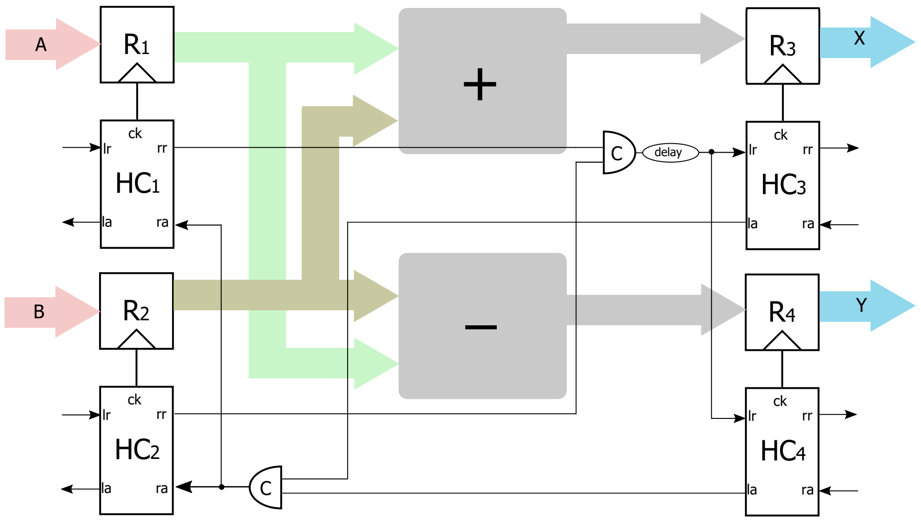


Figure 5.16: Elastic-bundle circuit (bundle set 1)

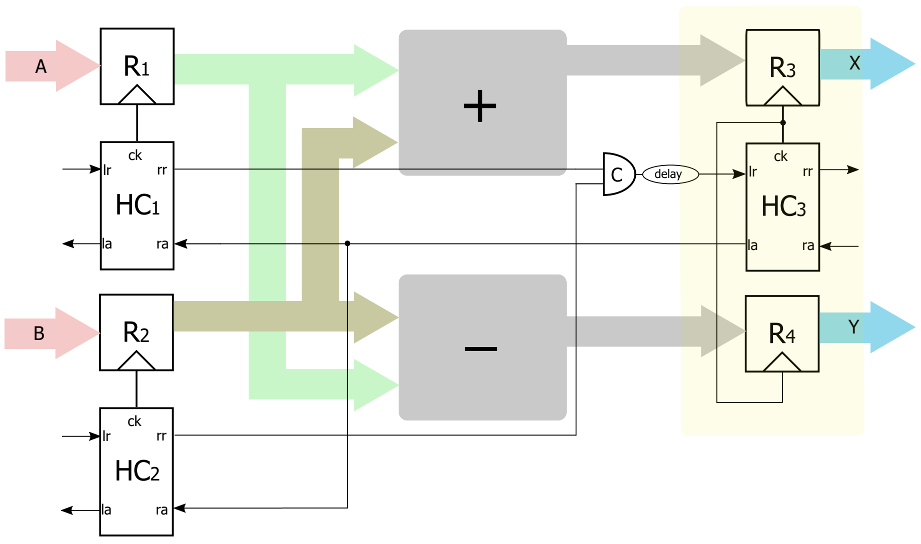


Figure 5.17: Elastic-bundle circuit (bundle set 2)

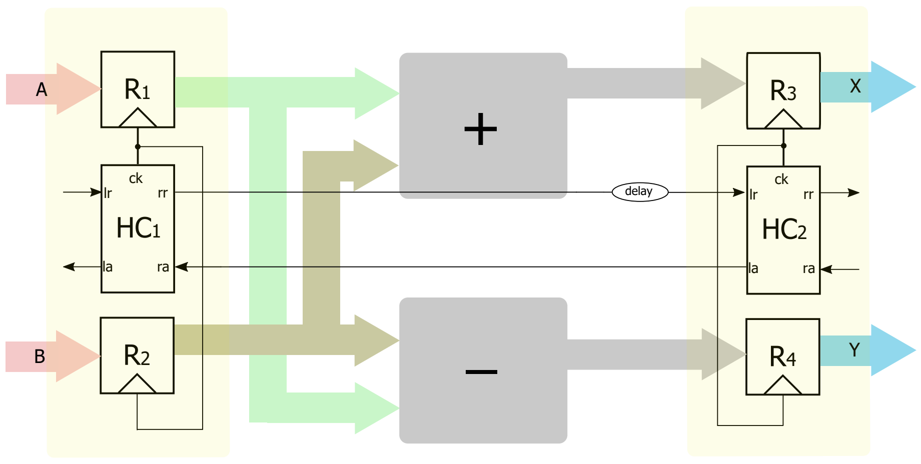


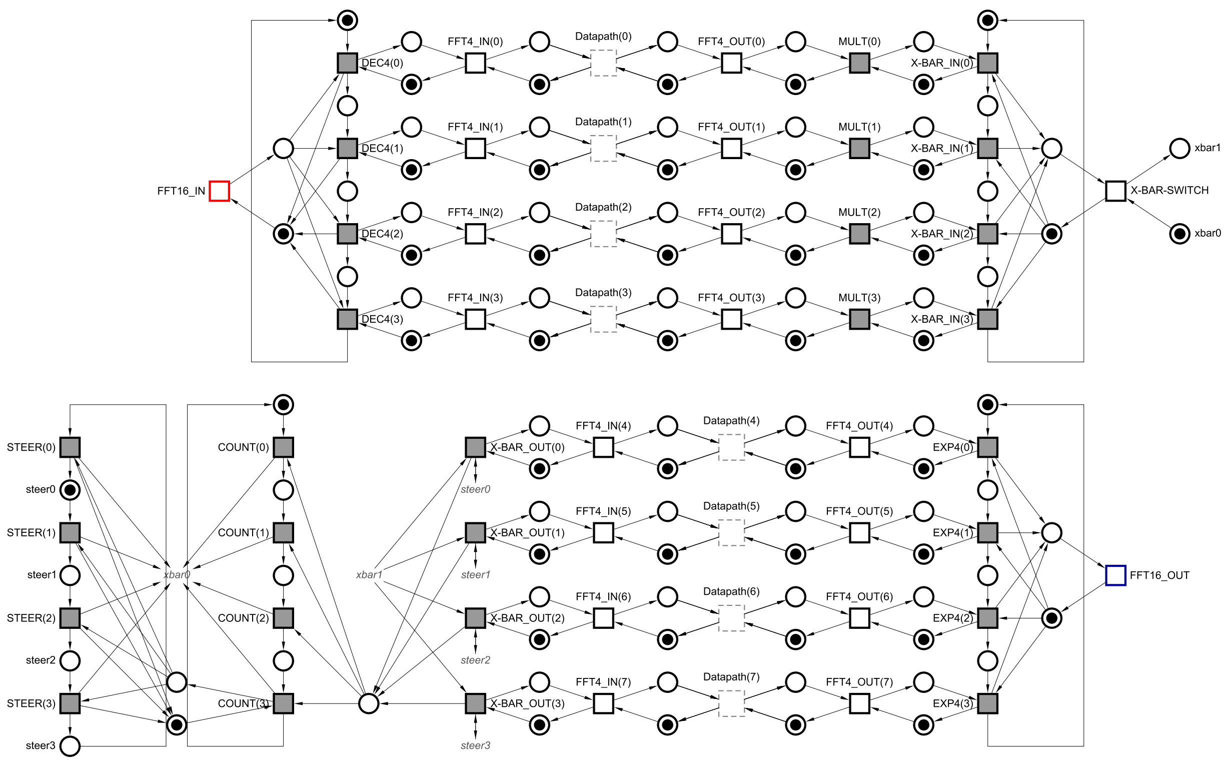
Figure 5.18: Elastic-bundle circuit (bundle set 3)

5.4 16-point FFT Case Study

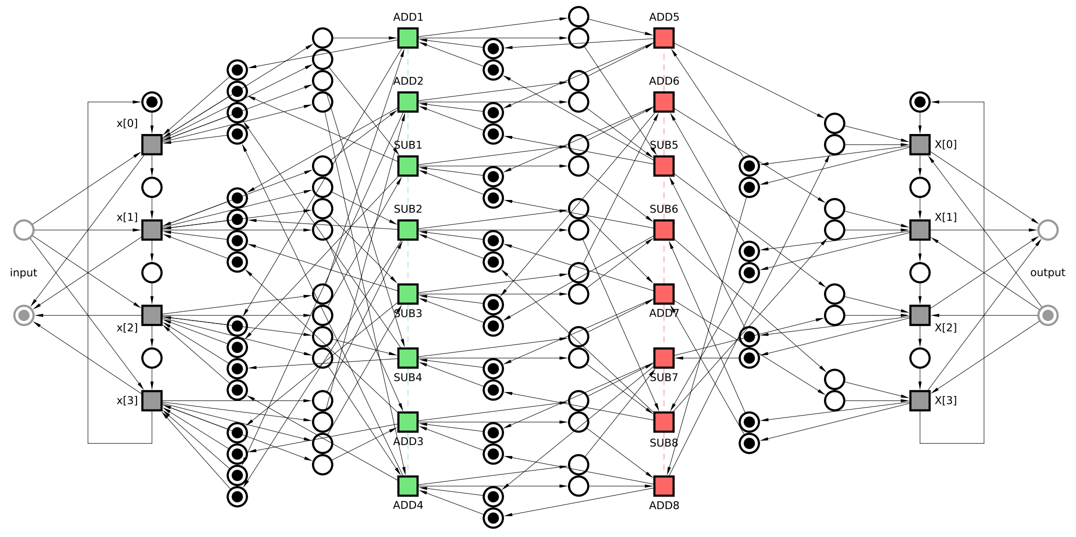
In this section, we consider an asynchronous 16-point FFT architecture as a case study to implement and test the methodology presented in this paper. Detailed description of the architecture and implementation are outside the scope of this paper. The FFT architecture is based on the design presented by the authors in [117, 1]. In this architecture, the FFT algorithm is described in a multirate format which is highly concurrent and heterogeneous by nature. This very nature of the architecture proved to be an ideal case study for us because it allowed for modelling of the algorithm in its native asynchronous/elastic form. Figure 5.19a provides a snapshot of the 16-point FFT architecture described in PNs. Four-way wagging structure captures the behaviour of high frequency input stream being decimated to lower frequency data streams and then expanded back to high frequency output stream. Distributed pipelining manages parallel operation of data streams at different frequencies. Furthermore, the 16-point FFT architecture is hierarchically decomposed of eight identical 4-point FFT blocks, denoted as dotted boxes in the top-level PN model. The PN model for the 4-point FFT datapath is shown in Figure 5.19b.

The case study was subjected to the proposed design flow, starting from PN description to EB functional partitioning, and HDL direct mapping to circuit synthesis using the RT design flow. For comparative analysis, five implementations of the 16-point FFT architecture were conducted: one bundled-data and four elastic-bundle (EB) asynchronous circuit implementations. The circuits were described in Verilog and synthesised using Design Compiler in the 90nm Faraday library. The circuits were tested using pre-defined input stream of 1024 random numbers. Circuit simulations were conducted using VCS simulator. The output data stream was verified for each partitioning scheme by comparing with MATLAB 16-point FFT computation. The SAIF (Switching Activity Interchange Format) file from the VCS simulations was used to calculate the power for each design by PrimeTime-PX. The EB implementations are compared against the bundled-data 16-point implementation. Table 5.1 summarises the pre-layout synthesis results of the case study. Appendix B provides the verilog code of the FFT architecture for further reference. Appendix C presents the synthesis timing constraints and setup scripts for the synthesis of the asynchronous FFT circuits.

The *bundled-data* implementation represents a fully elastic or asynchronous implementation of the 16-point FFT architecture. *EB Temporal* is the first elastic-bundle implementation of the architecture. Here, the adders and subtractors within the 4-point



(a) Top-level PN model



(b) PN model for Datapath block

Figure 5.19: PN model for 16-point FFT

Design	**Total Area (gates)**	**Control Area (gates)**	**Control Logic Power (mW)**	**Energy/Point (pJ)**	**Energy Benefit**	**Control Area Benefit**
EB Temporal	58,533	3,015	1.34	17.107	1.069	2.45
Opt EB Temporal	58,185	2,667	1.28	17.089	1.071	2.77
EB Maximal	57,921	2,403	1.24	17.105	1.069	3.08
EB Reuse	58,753	3,235	1.38	17.194	1.064	2.29
Bundled Data	62,993	7,399	2.03	18.294	1.000	1.00

Table 5.1: Synthesis results for several 16-point FFT designs

FFT datapath blocks are bundled according to their natural order of arrival of data tokens. In this case, the circuit area overhead of asynchronous control got reduced by nearly 60% compared to the bundled-data implementation, with a 34% improvement in control power consumption. *Opt EB Temporal* implementation optimised the previous implementation by restricting concurrency further whilst maintaining the natural order of data token arrival. *EB Maximal* implementation restricted concurrency even further with less regard to natural order of data arrival and more focus on reducing control overhead. This case is described in Figure 5.19 where a maximal case of bundling adders and subtractors with reduced concurrency can be visualised. Finally, the *EB Reuse* implementation focussed on reducing datapath computation logic by exploiting natural order of data arrival tokens to reuse adders and subtractors. The savings in area, as evident from the results, were due to reduction of 50% of adders and subtractor logic. This implementation, however, added control design complexity and control overhead in comparison to other EB implementations of the 16-point FFT. Appendix D provides the CDFGs for the 4-point FFT datapath of remaining elastic-bundle implementations.

The range of circuit area overheads demonstrated in this case study is clearly in line with the illustration of low-overhead elastic circuits presented earlier in Figure 1.1 (shadowed). Amongst the implementations, *EB Maximal* proves the most optimum demonstrating lowest control logic power consumption and an improvement of 3.08$\times$ in terms of control area over its bundled-data counterpart. All of the 16-point FFT EB implementations demonstrate $\sim$ 7% improvement in energy per data point. Noting that all FFT implementations share the same datapath, the energy improvement was due to reductions in control logic switching power coming from optimised bundled-data control.

It should be noted that the synthesis of datapath and control logic delay elements were conducted under worst-case timing-driven analysis. The possibility of average-case performance of asynchronous circuits and recalculation of delay elements when bundling concurrent computations using statistical timing analysis methods [118] have not been incorporated in this study. As future work, these techniques will be employed in the synthesis flow along with place and route for determining more accurate results.

5.5 Conclusions and Future Work

The chapter was motivated from the exploration of research area identified in Figure 1.1. We proposed a novel method for synthesising asynchronous circuits with varying levels of rigidity. The hypothesis was that *bundles* would reduce the area overheads of asynchronous design by relaxing granularity of handshake control. A PN-based dataflow modelling technique was developed to model digital systems on a higher level of abstraction than RTL. Functional partitioning was then introduced in these dataflow models by identifying sets of *bundles* that could restrict elasticity whilst retaining functional behaviour. Taking the case of asynchronous bundled-data circuits, these sets of *bundles* were extended to a novel notion of *Elastic Bundles* which basically re-partitioned the design into coarse-grained locally clocked elements (synchronised) and fine-grained locally clocked elements (asynchronous). This net transformation enabled synthesis of the elastic-bundle circuits under standard EDA tool flow. The method was tested on a 16-point FFT algorithm and the synthesis results look promising.. The elastic-bundle FFT implementations reduced control area overhead by a margin $> 2\times$ whilst demonstrating $> 30\%$ reduction in control power consumption when compared against the bundled-data counterpart.

Though the methodology presented in this research is in its infancy, we have presented a strong case for further exploration. Using small design examples, we have proposed methods that could apply to larger designs. However, design automation is required to be able to test the principles with manageable designer productivity. Extraction of Petri nets from C-like high level specification of systems, automated direct mapping of PN-based HDL to behavioural verilog and automating the generation of RT timing constraints are good areas for future work.

Chapter 6

Conclusion

Design complexity of traditionally clocked SoCs have resulted in emerging trends of mixed synchronous-asynchronous design. With growing research support in synchronous-asynchronous CAD tool integration, the design style is a promising approach for future SoCs. However, this paradigm is still in its infancy begging for formal methods, tools and design methodologies.

In this thesis, we have presented a formal model based on Petri nets with step semantics to represent the behaviour of mixed synchronous-asynchronous circuits. Theory of step persistence and bundles were formulated for verification and synthesis of these circuits. We hope that this theory would pave the way towards formal design and verification of mixed synchronous-asynchronous systems. With regards to tools and design methodology, a novel method for synthesising asynchronous circuits with varying levels of rigidity was proposed. Petri net modelling tool was used to describe the CDFG of a system and facilitate functional partitioning. The resulting elastic-bundle circuit was synthesised using standard EDA tools based on the RT synchronous-asynchronous design flow.

We think that this is the first work of its kind which presents a common formal method to represent mixed synchronous-asynchronous circuits. The work also demonstrated synthesis of mixed synchronous-asynchronous circuits starting from formal models to standard EDA tool flows. The results demonstrate strong potential of the method which would scale well for larger SoC designs.

6.1 Summary of Contributions

In Chapter 3, we started with the evaluation of GALS-based physical partitioning. We think that the work presented here is distinct amongst previous GALS literature as it provides an elaborate and thorough physical partitioning methodology for judging the applicability of GALS to SoC architectures. A parametric model is presented to design the CDN according to specific design constraints. Using this scheme, a simulation framework based on statistical system modelling was built where we analysed the effects of GALS partition granularity over different process technologies. Energy consumption of various GALS partitioning scenarios have been estimated and compared against a globally synchronous implementation. We have analysed the impact of GALS physical partitioning on latency and power of a system considering a generic design scenario. Our analysis was targeted at application-specific SoC designs as it was noted that their data path style architecture would benefit the most from GALS. Investigation in this chapter identified several limitations of the physical partitioning approach in GALS-based SoC design. The physical partitioning strategy would only provide a subset of solutions towards energy efficient system design. The design benefits gained from this approach does not justify the amount of design hours that would be encountered with this design style.

In Chapter 4, we investigated the behaviour of mixed synchronous-asynchronous circuits. The specification of a system is given in the form of a Petri net. Our aim was to re-design the system to optimise signal management, by grouping together concurrent events. Looking at the concurrent reachability graph of the given Petri net, we were interested in discovering events that appear in 'bundles', so that they all can be executed in a single clock tick. The best candidates for bundles are sets of events that appear and re-appear over and over again in the same configurations, forming 'persistent' sets of events. Persistence was considered so far only in the context of sequential semantics. In this chapter, we moved to the realm of step based execution and consider steps which are persistent and cannot be disabled by other steps. We introduce the notion of persistent steps and discuss their basic properties. We then introduce a formal definition of a bundle and propose an algorithm to prune the behaviour of a system, so that only bundled steps remain. The pruned reachability graph represents the behaviour of a re-engineered system, which in turn can be implemented in a new Petri net using the standard techniques of net synthesis. The proposed algorithm prunes reachability graphs of persistent and safe nets leaving bundles that represent maximally concurrent steps. The significance of the formal model and its various properties in the verification and synthesis of digital circuits was investigated. The implications of different classes of bundles on the complexity of control design was suggested.

In Chapter 5, we employed the formal methods and algorithm proposed in Chapter 4 to synthesise asynchronous circuits of varying levels of rigidity. A compact representation of system behaviour based on CDFG was proposed using Petri nets. Bundles were identified to relax elastic handshaking so that area overhead could be reduced whilst retaining functional behaviour. A scheme of varying granularity levels of rigidity was demonstrated to facilitate partitioning of the design into functional blocks of synchronised and asynchronised elements. The bundled design was verified on the basis of the formalised theory and then transformed into an equivalent circuit. Using the RT-based synchronous-asynchronous design flow, the mixed synchronous-asynchronous circuit was synthesised using standard EDA tools. The methodology was demonstrated on a 16-point FFT algorithm. The asynchronous circuit of the 16-point FFT was implemented as a bundled-data asynchronous pipeline with fine-grained handshake clocking. *Elastic bundles* were identified as those bundles in bundle-data circuits which are constrained to be handshake-clocked in step mannerism. Based on this principle, the design was partitioned the into coarse-grained locally clocked elements (synchronised) and fine-grained locally clocked elements (asynchronous). Asynchronous circuits with a range of granular rigidity were identified. Synthesis results demonstrated significant savings in area overhead and power consumption of the control circuitry. The area overhead reduction was shown to lie in the range of the shaded area depicted in Figure 1.1. In summary, we explored functional partitioning of an asynchronous system by identifying elastic bundles that can be bundled in time with similar synchronisation delay. In the realm of SoCs, ‘Elastic Bundles’ are viewed as fine-grained heterogeneous islands. We successfully demonstrated the use of clocked CAD tool to synthesise these elastic-bundle circuits.

6.2 Future Work

In order to develop this work further, the methodology needs to be tested over a variety of small designs covering heterogeneous processor flows and non-deterministic dataflows. Based on observations made in this research, the theory can be expanded to cover an extensive set of properties and problem statements. A good direction for future theoretical investigation would be to understand the significance of nonviolence[103, 104] in the behaviour of circuits.

The next move would be to explore avenues for automation of dataflow analysis and functional partitioning. Here, we would want to introduce the concept of timing in these graphs together with an annotation system for considering design considerations such as peak power. Timed Petri nets would be a good place to start. Automation of bundling could then be used to explore larger designs. Our method can then be compared with other functional partitioning schemes such as [18, 20, 42].

The principle of enforcing granular rigidity can be conducted on a coarser grain level to represent a granular clock signal similar to synchronous clock without affecting functional behaviour. This principle can reduce area overhead and maintain elasticity if designed well. Here, functional partitions would be wrapped in physical boundaries where a clock signal can replace local handshaking. Elastic clocks [91] demonstrates the same principle.

Another area of future work would be to synthesise synchronous handshake circuits or synchronous elastic circuits using the scheme of Elastic bundles. The principle of enforcing rigidity can be used to reduce the area overheads of synchronous elastic circuits.

Bibliography

[1] W. Lee, V. Vij *et al.*, "Design of Low Energy, High Performance Synchronous and Asynchronous 64-Point FFT," *Proc. Design, Automation & Test in Europe (DATE)*, pp. 242–247, 2013.

[2] C. Brej, "Wagging logic: Implicit parallelism extraction using asynchronous methodologies," in *Int. Conf. on Application of Concurrency to System Design (ACSD)*, 2010, pp. 35–44.

[3] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, *Logic synthesis for asynchronous controllers and interfaces*, ser. Integrated Circuits and Systems. Springer-Verlag, 2002.

[4] C. Myers, *Asynchronous circuit design.* Wiley, 2004.

[5] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu, and A. Ivanov, "System-on-chip: Reuse and integration," *Proc. of the IEEE*, vol. 94, no. 6, pp. 1050–1069, 2006.

[6] P. Greenhalgh, "Big.little processing with arm cortex-a15 & cortex-a7," *ARM White paper*, pp. 1–8, 2011.

[7] X. Fan, M. B. Stegmann, O. Schrape, S. Zeidler, I. G. Jensen, J. Thorsen, T. Bjerregaard, and M. Krstić, "Frequency-domain optimization of digital switching noise based on clock scheduling," *IEEE Trans. on Circuits and Systems*, vol. 63, no. 7, pp. 982–993, 2016.

[8] L. Benini and G. De Micheli, "Networks on chips: A new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[9] J. Henkel, W. Wolf, and S. Chakradhar, "On-chip networks: A scalable, communication-centric embedded system design paradigm," in *Proc. Int. Conf. on VLSI Design*, 2004, pp. 845–851.

[10] A. Hemani, T. Meincke *et al.*, "Lowering power consumption in clock by using globally asynchronous locally synchronous design style," *Proc. Design Automation Conference (DAC)*, pp. 873–878, 1999.

[11] A. Iyer and D. Marculescu, "Power and performance evaluation of globally asynchronous locally synchronous processors," *Proc. Computer Architecture*, no. 4901, pp. 158–168, 2002.

[12] A. Chattopadhyay and Z. Zilic, "Galds: a complete framework for designing multiclock asics and socs," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 6, pp. 641–654, 2005.

[13] R. Dobkin, R. Ginosar, and C. Sotiriou, "High Rate Data Synchronization in GALS SoCs," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 10, pp. 1063–1074, 2006.

[14] T. Krol, M. Krstic, X. Fan, and E. Grass, "Modeling and reducing emi in gals and synchronous systems," *Proc. Int. Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, vol. 5953, pp. 146–155, 2009.

[15] X. Fan, M. Krstic, and E. Grass, "Performance Analysis of GALS Datalink Based on Pausible Clocking," *Proc. Asynchronous Circuits and Systems (ASYNC)*, pp. 126–133, 2012.

[16] X. Fan, M. Krstic *et al.*, "Exploring pausible clocking based gals design for 40-nm system integration," in *Proc. Design, Automation & Test in Europe (DATE)*, 2012, pp. 1118–1121.

[17] E. Czeck, R. Nanavati, and J. Stoy, "Reliable design with multiple clock domains," *Proc. Formal Methods and Models for Co-Design (MEMOCODE)*, 2006.

[18] R. S. Nikhil, "Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions," in *High-Level Synthesis*. Springer, 2008, pp. 129–146.

[19] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proc. Design Automation Conference (DAC)*, 2012, pp. 1216–1225.

[20] A. Peeters, F. te Beest, M. de Wit, and W. Mallon, "Click elements: An implementation style for data-driven compilation," in *Int. Symp. on Asynchronous Circuits and Systems (ASYNC)*, 2010, pp. 3–14.

[21] M. Mamaghani, W. Toms, and J. Garside, "eTeak: A Data-driven Synchronous Elastic Synthesiser," *Int. Conf. on Application of Concurrency to System Design (ACSD)*, 2013.

[22] S. Dasgupta and A. Yakovlev, "Desynchronisation Technique Using Petri Nets," *Electronic Notes in Theoretical Computer Science*, vol. 245, pp. 51–67, 2009.

[23] D. Sokolov and A. Yakovlev, "GALS Partitioning by Behavioural Decoupling Expressed in Petri Nets," *Proc. Asynchronous Circuits and Systems (ASYNC)*, pp. 17–26, 2014.

[24] K. Stevens, D. Gebhardt, J. You, Y. Xu, V. Vij, S. Das, and K. Desai, "The Future of Formal Methods and GALS Design," *Electronic Notes in Theoretical Computer Science*, vol. 245, pp. 115–134, 2009.

[25] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev, "Asynchronous design using commercial hdl synthesis tools," in *Int. Symp. on Asynchronus Circuits and Systems (ASYNC)*, 2000, pp. 114–125.

[26] A. Peeters and K. Van Berkel, "Synchronous handshake circuits," in *Int. Symp. on Asynchronus Circuits and Systems (ASYNC)*, 2001, pp. 86–95.

[27] A. Kondratyev and K. Lwin, "Design of asynchronous circuits by synchronous cad tools," in *Proc. Design Automation Conference (DAC)*, 2002, pp. 411–414.

[28] J. Cortadella, A. Kondratyev, L. Lavagno, and C. P. Sotiriou, “Desynchronization: Synthesis of asynchronous circuits from synchronous specifications,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1904–1921, 2006.

[29] J. Cortadella, M. Kishinevsky, and B. Grundmann, “Synthesis of synchronous elastic architectures,” in *Proc. Design Automation Conference (DAC)*, 2006, pp. 657–662.

[30] N. Andrikos, L. Lavagno, D. Pandini, and C. P. Sotiriou, “A fully-automated desynchronization flow for synchronous circuits,” in *Proc. Design Automation Conference (DAC)*, 2007, pp. 982–985.

[31] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin, “Elastic circuits,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1437–1455, 2009.

[32] K. S. Stevens, Y. Xu, and V. Vij, “Characterization of asynchronous templates for integration into clocked cad flows,” in *Proc. Asynchronous Circuits and Systems (ASYNC)*, 2009, pp. 151–161.

[33] M. J. Mamaghani, J. D. Garside, W. B. Toms, and D. Edwards, “Optimised synthesis of asynchronous elastic dataflows by leveraging clocked eda,” in *Euromicro Conf. on Digital System Design (DSD)*, 2014, pp. 607–614.

[34] J. Muttersbach, T. Villiger, and W. Fichtner, “Practical design of globally-asynchronous locally-synchronous systems,” in *Proc. Asynchronous Circuits and Systems (ASYNC)*, 2000, pp. 52–59.

[35] Y. Zhu, D. H. Albonesi, and A. Buyuktosunoglu, “A high performance, energy efficient gals processor microarchitecture with reduced implementation complexity,” in *Proc. Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2005, pp. 42–53.

[36] M. Krstic, E. Grass, and C. Stahl, “Request-Driven GALS Technique for Wireless Communication System,” *Proc. Asynchronous Circuits and Systems (ASYNC)*, pp. 76–85, 2005.

[37] F. Gurkaynak, S. Oetiker *et al.*, "GALS at ETH Zurich: Success or Failure," *Proc. Asynchronous Circuits and Systems (ASYNC)*, pp. 150–159, 2006.

[38] U. Y. Ogras, R. Marculescu, P. Choudhary, and D. Marculescu, "Voltage-frequency island partitioning for gals-based networks-on-chip," in *Proc. Design Automation Conference (DAC)*, 2007, pp. 110–115.

[39] M. Krstic, M. Piz, M. Ehrig, and E. Grass, "Ofdm datapath baseband processor for 1 gbps datarate," *Proc. IFIP/IEEE VLSI-SoC*, pp. 13–15, 2008.

[40] S. Suhaib, D. Mathaikutty, and S. Shukla, "Dataflow architectures for gals," *Electronic Notes in Theoretical Computer Science*, vol. 200, no. 1, pp. 33–50, 2008.

[41] Z. Yu and B. Baas, "High Performance, Energy Efficiency, and Scalability With GALS Chip Multiprocessors," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 1, pp. 66–79, 2009.

[42] M. Jelodari Mamaghani, D. Sokolov, and J. Garside, "Asynchronous dataflow de-elastisation for efficient heterogeneous synthesis," in *Int. Conf. on Application of Concurrency to System Design (ACSD)*, 2016.

[43] M. J. Mamaghani, M. Krstic, and J. Garside, "Automatic clock: A promising approach toward galsification," in *Int. Symp. on Asynchronous Circuits and Systems (ASYNC)*, 2016.

[44] D. Chapiro, "Globally-asynchronous locally-synchronous systems," Ph.D. dissertation, Stanford University, 1984.

[45] J. Sparso and S. Furber, *Principles of asynchronous circuit design: a systems perspective.* Kluwer Academic Publishers, 2001.

[46] S. M. Nowick and M. Singh, "High-performance asynchronous pipelines: an overview," *IEEE Design & Test of Computers*, vol. 28, no. 5, pp. 8–22, 2011.

[47] R. Keller, "A fundamental theorem of asynchronous parallel computation," *Lecture Notes in Computer Science*, vol. 24, pp. 102–112, 1975.

[48] M. Koutny and M. Pietkiewicz-Koutny, “Transition systems of elementary net systems with localities,” in *Proc. Int. Conf. on Concurrency Theory*, 2006, pp. 173–187.

[49] T. Murata, “Petri nets: Properties, analysis and applications,” *Proc. of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[50] A. Yakovlev, A. Koelmans, A. Semenov, and D. Kinniment, “Modelling, analysis and synthesis of asynchronous control circuits using petri nets,” *Integration , the VLSI Journal*, vol. 21, pp. 143–170, 1996.

[51] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, “Hardware and petri nets application to asynchronous circuit design,” in *Int. Conf. on Application and Theory of Petri Nets*, 2000, pp. 1–15.

[52] C. A. Petri, “Kommunikation mit automaten,” Ph.D. dissertation, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, 1962.

[53] D. Sokolov and A. Yakovlev, “Clockless circuits and system synthesis,” *IEE Proceedings-Computers and Digital Techniques*, vol. 152, no. 3, pp. 298–316, 2005.

[54] S. Tugsinavisut, R. Su, and P. A. Beerel, “High-level synthesis for highly concurrent hardware systems,” in *Int. Conf. on Application of Concurrency to System Design (ACSD)*, 2006, pp. 79–90.

[55] S. A. Seshia, R. E. Bryant, and K. S. Stevens, “Modeling and verifying circuits using generalized relative timing,” in *Proc. Asynchronous Circuits and Systems (ASYNC)*, 2005, pp. 98–108.

[56] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott, “Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling,” in *Proc. Int. Symp. on High-Performance Computer Architecture*, 2002, pp. 29–40.

[57] A. Upadhyay, S. R. Hasan, and M. Nekili, “Optimal partitioning of globally asychronous locally synchronous processor arrays,” *Proc. ACM Great Lakes Symposium on VLSI*, pp. 7–12, 2004.

[58] E. Talpes and D. Marculescu, “Toward a multiple clock/voltage island design style for power-aware processors,” *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 5, pp. 591–603, 2005.

[59] E. Friedman, “Clock distribution networks in synchronous digital integrated circuits,” *Proc. of the IEEE*, vol. 89, no. 5, pp. 665–692, 2001.

[60] H. Mahmoodi, V. Tirumalashetty, M. Cooke, and K. Roy, “Ultra low-power clocking scheme using energy recovery and clock gating,” *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 17, pp. 33–44, 2009.

[61] S. Pullela, N. Menezes, and L. Pillage, “Reliable non-zero skew clock trees using wire width optimization,” *Proc. Design Automation Conference (DAC)*, pp. 165–170, 1993.

[62] D. Velenis, R. Sundaresha, and E. Friedman, “Buffer sizing for delay uncertainty induced by process variations,” in *Proc. Int. Conf. on Electronics, Circuits and Systems (ICECS)*, 2004, pp. 415–418.

[63] A. Fisher and H. Kung, “Synchronizing large VLSI processor arrays,” *IEEE Trans. on Computers*, vol. c, no. 8, pp. 734–740, 1985.

[64] S. Tam, *Modern Clock Distribution Systems*, ser. Integrated Circuits and Systems, T. Xanthopoulos, Ed. Springer US, 2009.

[65] Y. Semiat and R. Ginosar, “Timing measurements of synchronization circuits,” *Proc. Asynchronous Circuits and Systems (ASYNC)*, pp. 68–77, 2003.

[66] R. Dobkin and R. Ginosar, “Zero latency synchronizers using four and two phase protocols,” Tech. Rep., 2007.

[67] S. Moore, G. Taylor, R. Mullins, and P. Robinson, “Point to point gals interconnect,” in *Proc. Asynchronous Circuits and Systems (ASYNC)*, 2002, pp. 69–75.

[68] X. Fan, M. Krstic, and E. Grass, “Analysis and optimization of pausible clocking based GALS design,” *IEEE Int. Conf. on Computer Design*, pp. 358–365, 2009.

[69] T. Chelcea and S. Nowick, "A low-latency fifo for mixed-clock systems," *Proc. IEEE Computer Society Workshop on VLSI*, pp. 119–126, 2000.

[70] P. Teehan, M. Greenstreet, and G. Lemieux, "A survey and taxonomy of gals design styles," *IEEE Design Test of Computers*, vol. 24, pp. 418–428, 2007.

[71] J.-M. Chabloz and A. Hemani, "Lowering the latency of interfaces for rationally-related frequencies," *Proc. Int. Conf. on Computer Design*, pp. 23–30, 2010.

[72] R. Ginosar, "Metastability and synchronizers: A tutorial," *IEEE Design and Test of Computers*, vol. 28, no. 5, pp. 23–35, 2011.

[73] K. Y. Yun and A. E. Dooply, "Pausible clocking-based heterogeneous systems," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 7, pp. 482–488, 1999.

[74] R. Mullins and S. Moore, "Demystifying Data-Driven and Pausible Clocking Schemes," *Proc. Asynchronous Circuits and Systems (ASYNC)*, pp. 175–185, 2007.

[75] S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, vol. 19, pp. 23–29, 1999.

[76] D. Chinnery and K. Keutzer, *Closing the gap between ASIC & custom: tools and techniques for high-performance ASIC design*, 2007.

[77] S. Nassif, "Modeling and analysis of manufacturing variations," *Proc. Custom Integrated Circuits Conference*, pp. 223–228, 2001.

[78] Predictive technology model. [Online]. Available: http://ptm.asu.edu/

[79] Semiconductor Industry Association. (2001) International Technology Roadmap for Semiconductors (ITRS). [Online]. Available: http://www.itrs2.net/

[80] X. Jiang and S. Horiguchi, "Optimization of wafer scale h-tree clock distribution network based on a new statistical skew model," in *Proc. Defect and Fault Tolerance in VLSI Systems*, 2000, pp. 96–104.

[81] M. Hashimoto, T. Yamamoto, and H. Onodera, "Statistical analysis of clock skew variation in h-tree structure," in *Proc. Quality of Electronic Design (ISQED)*, 2005, pp. 402–407.

[82] A. Narasimhan and R. Sridhar, "Impact of Variability on Clock Skew in H-tree Clock Networks," *Proc. Quality Electronic Design (ISQED)*, pp. 458–466, 2007.

[83] M. Seok, D. Blaauw, and D. Sylvester, "Robust clock network design methodology for ultra-low voltage operations," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 2, pp. 120–130, 2011.

[84] X. Jiang and S. Horiguchi, "Statistical skew modeling for general clock distribution networks in presence of process variations," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 5, pp. 704–717, 2001.

[85] A. Agarwal, V. Zolotov, and D. T. Blaauw, "Statistical clock skew analysis considering intradie-process variations," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 8, pp. 1231–1242, 2004.

[86] D. Mijuskovic, "Clock distribution in application specific integrated circuits," *Microelectronics Journal*, vol. 18, no. 4, pp. 15 – 27, 1987.

[87] C. Kashyap, C. Alpert *et al.*, "PERI: a technique for extending delay and slew metrics to ramp inputs," *Proc. Timing Issues in the Specification and Synthesis of Digital Systems*, 2002.

[88] Min Pan, Chris Chong-Nuen Chu, and J. Morris Chang, "Transition Time Bounded Low-power Clock Tree Construction," *Proc. Int. Symp. on Circuits and Systems (ISCAS)*, pp. 2445–2448, 2005.

[89] S. W. Moore, G. S. Taylor, P. A. Cunningham, R. D. Mullins, and P. Robinson, "Self calibrating clocks for globally asynchronous locally synchronous systems," in *Proc. Int. Conf. on Computer Design*, 2000, pp. 73–78.

[90] A. Davis and S. M. Nowick, *An Introduction to Asynchronous Circuit Design*, ser. The Encyclopedia of Computer Science and Technology, 1997.

[91] J. Cortadella, L. Lavagno, D. Amiri, J. Casanova, C. Macian, F. Martorell, J. A. Moya, L. Necchi, D. Sokolov, and E. Tuncer, "Narrowing the margins with elastic clocks," in *Int. Conf. on IC Design and Technology (ICICDT)*, 2010, pp. 146–150.

[92] L. Landweber and E. Robertson, "Properties of conflict-free and persistent Petri nets," *Journal of the ACM (JACM)*, vol. 25, pp. 352–364, 1978.

[93] E. Best and P. Darondeau, "Decomposition theorems for bounded persistent Petri nets," *Int. Conf. on Applications and Theory of Petri Nets and Concurrency (ICATPN)*, vol. 5062, pp. 33–51, 2008.

[94] E. Best and P. Darondeau, "Separability in persistent petri nets," *Int. Conf. on Applications and Theory of Petri Nets and Concurrency (ICATPN)*, vol. 6128, pp. 246–266, 2010.

[95] K. Barylska, L. Mikulski, and E. Ochmanski, "On persistent reachability in petri nets," *Information and Computation*, vol. 223, pp. 67–77, 2013.

[96] S. Dasgupta, D. Potop-Butucaru, B. Caillaud, and A. Yakovlev, "Moving from Weakly Endochronous Systems to Delay-Insensitive Circuits," *Electronic Notes in Theoretical Computer Science*, vol. 146, pp. 81–103, 2006.

[97] D. Muller and W. Bartky, "A theory of asynchronous circuits," *Proc. Int. Symp. on the Theory of Switching*, pp. 204–243, 1959.

[98] A. Yakovlev, "Designing self-timed systems," *VLSI System Design*, vol. vi, pp. 70–90, 1985.

[99] K. Barylska and E. Ochmanski, "Levels of persistency in place/transition nets," *Fundamenta Informaticae*, vol. 93, pp. 33–43, 2009.

[100] J. Carmona, J. Julvez, J. Cortadella, and M. Kishinevsky, "Scheduling synchronous elastic designs," in *Int. Conf. on Application of Concurrency to System Design (ACSD*, ser. ACSD '09, 2009, pp. 52–59.

[101] A. Mokhov, V. Khomenko, D. Sokolov, and A. Yakovlev, "Opportunistic merge element," in *Int. Symp. on Asynchronus Circuits and Systems (ASYNC)*, 2015, pp. 116–123.

[102] J. Fernandes, M. Koutny, M. Pietkiewicz-Koutny, D. Sokolov, and A. Yakovlev, "Step persistence in the design of gals systems," *Int. Conf. on Applications and Theory of Petri Nets and Concurrency (ICATPN)*, vol. 7927, pp. 190–209, 2013.

[103] M. Koutny, L. Mikulski, and M. Pietkiewicz-Koutny, "A taxonomy of persistent and nonviolent steps," *Int. Conf. on Applications and Theory of Petri Nets and Concurrency (ICATPN)*, vol. 7927, pp. 210–229, 2013.

[104] J. Fernandes, M. Koutny, L. Mikulski, M. Pietkiewicz-Koutny, D. Sokolov, and A. Yakovlev, "Persistent and nonviolent steps and the design of gals systems," *Fundamenta Informaticae*, vol. 137, pp. 143–170, 2015.

[105] I. Blunno and L. Lavagno, "Automated synthesis of micro-pipelines from behavioral verilog hdl," in *Int. Symp. on Asynchronus Circuits and Systems (ASYNC)*, 2000, pp. 84–92.

[106] F. Te Beest, A. Peeters, K. Van Berkel, and H. Kerkhoff, "Synchronous full-scan for asynchronous handshake circuits," *Journal of Electronic Testing*, vol. 19, no. 4, pp. 397–406, 2003.

[107] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers, "Synchronous interlocked pipelines," in *Int. Symp. on Asynchronus Circuits and Systems (ASYNC)*, 2002, pp. 3–12.

[108] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli, "A methodology for correct-by-construction latency insensitive design," in *The Best of ICCAD*, 2003, pp. 143–158.

[109] M. J. Mamaghani, J. Garside, and D. Edwards, "De-elastisation: from asynchronous dataflows to synchronous circuits," in *Proc. Design, Automation & Test in Europe (DATE)*, 2015, pp. 273–276.

[110] I. Poliakov, D. Sokolov, and A. Mokhov, "Workcraft: a static data flow structure editing, visualisation and analysis tool," in *Int. Conf. on Applications and Theory of Petri Nets and Concurrency (ICATPN).*

[111] I. Poliakov, V. Khomenko, and A. Yakovlev, "Workcraft–a framework for interpreted graph models," in *Int. Conf. on Applications and Theory of Petri Nets and Concurrency (ICATPN).*

[112] "WORKCRAFT homepage, URL: http://www.workcraft.org."

[113] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *ACM SIGARCH Computer Architecture News*, vol. 3, no. 4, 1975, pp. 126–132.

[114] D. Culler, "Dataflow architectures," *Annual Review of Computer Science*, vol. 1, no. 1, pp. 225–253, 1986.

[115] A. Mokhov, D. Sokolov, and A. Yakovlev, "Adapting asynchronous circuits to operating conditions by logic parametrisation," in *Int. Symp. on Asynchronus Circuits and Systems (ASYNC)*, 2012, pp. 17–24.

[116] R. Manohar and A. J. Martin, "Slack elasticity in concurrent computing," in *International Conference on Mathematics of Program Construction*, 1998, pp. 272–285.

[117] D. J. Barnhart, "An improved asynchronous implementation of a fast fourier transform architecture for space applications," Air Force Institute of Technology, United States Air Force, Tech. Rep., 1999.

[118] D. Blaauw, K. Chopra, A. Srivastava, and L. Scheffer, "Statistical timing analysis: From basic principles to state of the art," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 4, pp. 589–607, 2008.

Appendix A

MATLAB code of GALS physical partitioning analysis

The following code is a MATLAB script file that provides the source code for the simulation framework used in Chapter 3. CMOS technology parameters for each of the process technology nodes will need to be uncommented out before deriving results for respective process technologies. The script can be modified for designer-desired chip sizes. However, the script does not automate partitioning. The sizes of the partitions would need to be manually computed from an external script and modified accordingly in the script.

```
clear;
clc;
%———————————————————————
%CMOS Technology Parameters (Ref:Nassif)
%65nm
Vdd=0.9;
Vt_m=0.3;
Vt_3sd=0.04;
Tox_m=3e-9; Tox_3sd=0.48e-9; %gate oxide thickness in metres
Res_m=0.075 ; Res_3sd=0.025; %Sheet resistance ohm/sq block=(resistivity/T_INT)
W_INT_m=0.3 ; W_INT_3sd=0.1; %um
H_INT_m=0.7 ; H_INT_3sd=0.25; %um this is inter layer dielectric (ILD) height not thickness

% Parameters from Ref: ASU PTM
T_INT=1.2; %um thickness of wire
k=2.2;

DielCon0=885.4e-14; %F/m 8.854e-14F/cm
DielCon=k*DielCon0; %F/m
Cox=DielCon/Tox_m; %gate unit area cap
mu=0.0067; % charge carrier mobility (670 cm2/(V*s))

%Minimum sized inverter (Ref: scaled from Jiang)
Lt_m=0.065; Lt_3sd=0.033; %um transistor length (3sd from Nassif)
wt_nmos=0.37*65/250; %um
```

```
wt_pmos=1.1*65/250; %um
K=mu*Cox*wt_nmos/Lt_m; %On resistance depends on nmos
Ro=1/(K*(Vdd-Vt_m)); % output impedance
Co=Cox*Lt_m*(wt_nmos+wt_pmos)*1e-6*1e-6/1e-15; %fF input capacitance
Cd=Co/(2.5); %fF output parasitic capactiance (2.5=14.3/5.8 from Ref: Hashimoto)

%FF Capacitance
FF_Cap=2.5; %fF

FO4dly=Ro*(4*Co+Cd)*1e-3; %ps

C_INT=DielCon*1e-6*((W_INT_m/H_INT_m)+0.77 ...
    +1.06*((W_INT_m/H_INT_m)^0.25+(T_INT/H_INT_m)^0.5))/1e-15;
%fF/um capacitance of nominal width interconnect

R=Res_m/W_INT_m; %ohm/um Resistance of nominal width interconnect

%SoC Design Parameters
%CDN Sink Density
SinkD=(18069/2.2)*0.49; %FFs per mm2

%Clock constraints
f=(1e12/(100*FO4dly)); %scaled clock frequency to satisify MUTEX resolution time constraint
CP=(1/f)*1e12; %Clock period (ps)
Budget=0.05;
TlenVARbudget=0.4; % Budget percentage for process variations. 0.6 for environmental
Target_Skew=CP*Budget; %Total Skew Budget
Target_LocSkew=Target_Skew*0.4 ; %Local Skew Budget
Target_GloSkew=Target_Skew*0.6*TlenVARbudget ; %Global Skew Budget
Target_Rise=CP*0.15; %Clock Signal Rise Time
MAX_LogicDly=CP*(1-Budget); %Worst case delay of comb logic

%——————————————————————
%Island Matrix sizing for 65nm
%Using Values generated from Equal area sizing script
%1 island
GALS(1,1)=3200;GALS(1,2)=3200;
%2 islands
GALS(2,1)=1600;GALS(2,2)=3200;
%3 islands
GALS(3,1)=2133.33;GALS(3,2)=1600;GALS(4,1)=1066.67;GALS(4,2)=3200;
%4 islands
GALS(5,1)=1600;GALS(5,2)=1600;
%5 islands
GALS(6,1)=1600;GALS(6,2)=1280;GALS(7,1)=960;GALS(7,2)=2133.33;
GALS(8,1)=1066.67;GALS(8,2)=1920;
```

```
    %6 islands
    GALS(9,1)=1600;GALS(9,2)=1066.67;
    %7 islands
    GALS(10,1)=1600;GALS(10,2)=914.33;GALS(11,1)=2133.33;GALS(11,2)=685.67;
    GALS(12,1)=1371.4;GALS(12,2)=1066.67;
    %8 islands
    GALS(13,1)=800;GALS(13,2)=1600;
    %9 islands
    GALS(14,1)=1600;GALS(14,2)=711.11;GALS(15,1)=888.89;GALS(15,2)=1280;
    GALS(16,1)=960;GALS(16,2)=1185.2;GALS(17,1)=592.6;GALS(17,2)=1920;
    %10 islands
    GALS(18,1)=640;GALS(18,2)=1600;
    %11 islands
    GALS(19,1)=1600;GALS(19,2)=581.8;GALS(20,1)=1280;GALS(20,2)=727.3;
    GALS(21,1)=960;GALS(21,2)=969.7;GALS(22,1)=1920;GALS(22,2)=484.8;
    %12 islands
    GALS(23,1)=1600;GALS(23,2)=533.33;
    %13 islands
    GALS(24,1)=1600;GALS(24,2)=492.3;GALS(25,1)=914.3;GALS(25,2)=861.5;
    GALS(26,1)=1148.7;GALS(26,2)=685.7;GALS(27,1)=1371.4;GALS(27,2)=574.4;
    %14 islands
    GALS(28,1)=1600;GALS(28,2)=457.1;
    %15 islands
    GALS(29,1)=426.67;GALS(29,2)=1600;GALS(30,1)=914.3;GALS(30,2)=746.67;
    GALS(31,1)=995.56;GALS(31,2)=685.7;GALS(32,1)=1371.4;GALS(32,2)=497.78;
    %16 islands
    GALS(33,1)=800; GALS(33,2)=800;

    %Design sizing for two cases of chip sizes
    %GALS=GALS*2.35; %size = 7.52mm X 7.52mm
    %GALS=GALS*0.625; %size = 2mm X 2mm

    %Number of islands per partition granularity
    NI(1)=1; NI(2)=2;NI(3)=2;NI(4)=1; NI(5)=4;NI(6)=2; NI(7)=2;NI(8)=1;NI(9)=6;
    NI(10)=4;NI(11)=2; NI(12)=1; NI(13)=8; NI(14)=4; NI(15)=2;NI(16)=2;NI(17)=1;
    NI(18)=10;NI(19)=6;NI(20)=2; NI(21)=2;NI(22)=1;NI(23)=12;NI(24)=6;NI(25)=4;
    NI(26)=2; NI(27)=1; NI(28)=14; NI(29)=8; NI(30)=4;NI(31)=2;NI(32)=1;NI(33)=16;
    %------------------------------------------
%------------------------------------------
%Clock Tree Synthesis of Synchronous Chip
    y=1;
    h=GALS(y,1); %floorplan height (um)
    w=GALS(y,2); %floorplan width (um)
    %initialise to enter while loop
    lamda=50;
    Max_Skew=400;
```

```
Local_Skew=400;
RISE=1000;
L=0;
W=0;
Cint=0;
Variance_Cint=0;
Rint=0;
Variance_Rint=0;
x=0;
Pdly=0;
Rdly=0;
Rdly_out=0;
Testdly=0;
Variance_Rdly=0;
Variance_Pdly=0;
SD_Rdly=0;
SD_Pdly=0;

FFs_sink=256; %Maximum allowed FF per sink
FF_Tot=round(SinkD*(h*w)*1e-6) ;
Total_L=FF_Tot*FF_Cap; %fF
C_L=FFs_sink*FF_Cap; %fF Load per sink
N=Total_L/C_L; %no of sinks
alpha=(log(N)/log(4)); %alpha denotes Number of Hs– alpha*2 =clock tree levels

if alpha < 0
alpha=0;
end

%Algorithm to distribute FFs evenly
if mod(2*alpha,2)>0
alpha=ceil(log(N)/log(4));
N=4^alpha;
FFs_sink=round(FF_Tot/N);
C_L=FFs_sink*FF_Cap;
end

D=((2^alpha)-1)*(h+w)/((2^(alpha+1))); %distance from root of tree to sink

%Clock Tree buffer optimisation
if alpha == 0

%Only Local interconnect buffer optimisation
Max_Skew=0;
```

```
sz=0;

while (Local_Skew>Target_LocSkew)||(RISE>Target_Rise)
lamda=lamda-0.1;
if (lamda<1)
 lamda=lamda+0.333333; %also debug purpose
 break;
end
Variance_K= ((mu*(DielCon/(Tox_m)^2)*(wt_nmos/Lt_m))^2)*...
      ((Tox_3sd/3)^2)+ ((mu*(DielCon/Tox_m)*...
      (wt_nmos/(Lt_m)^2))^2)*((Lt_3sd/3)^2);
Variance_Ro= ((1/((K^2)*(Vdd-Vt_m)))^2)*Variance_K+ ...
      ((1/(K*((Vdd-Vt_m)^2)))^2)*((Vt_3sd/3)^2);
x(sz+1)= round(C_L/(lamda*Co-Cd));
RISE=(10^-3)*2.3*(Ro/x(sz+1))*( C_L + x(sz+1)*Cd );
Pdly(sz+1)=(10^-3)*0.69*(Ro/x(sz+1))*( C_L + x(sz+1)*Cd );
Variance_Pdly(sz+1)=(((0.69/x(sz+1))*( C_L + x(sz+1)*Cd ))^2)*(Variance_Ro);
SD_Pdly(sz+1)= (10^-3)*sqrt(Variance_Pdly(sz+1));

%Local Skew calculation (skew = Max path delay - Min path delay )
%Min path is at starting point of grid.... delay is only buffer delay
%Max path is at end of grid (determined by Manhattan Distance)
%Local skew = Max path + 3sigma (max path delay)
%       + 3sigma (max path input buffer) + 3sigma(min path input buffer)

h_hsize= h/(2^alpha);
w_hsize= w/(2^alpha);
Cint_internal=(h_hsize+w_hsize)*DielCon*1e-6*((W_INT_m/H_INT_m)+0.77...
      +1.06*((W_INT_m/H_INT_m)^0.25+(T_INT/H_INT_m)^0.5))/1e-15;
Variance_Cint_internal=(((h_hsize+w_hsize)*DielCon*1e-6/1e-15*...
      ((1/H_INT_m) + (1.06*0.25*((W_INT_m/H_INT_m)^(0.25-1))*...
      (1/H_INT_m))))^2)*((W_INT_3sd/3)^2) + ...
      (((h_hsize+w_hsize)*DielCon*1e-6/1e-15*(-(W_INT_m)*...
      (H_INT_m)^-2 + (1.06*( 0.25*((W_INT_m/H_INT_m)^(0.25-1))*...
      (-(W_INT_m)*(H_INT_m)^-2) + 0.5*((T_INT/H_INT_m)^(0.5-1))*...
      -T_INT*(H_INT_m)^-2))))^2)*((H_INT_3sd/3)^2);
Rint_internal=(h_hsize+w_hsize)*R;
Variance_Rint_internal=(((h_hsize+w_hsize)/(W_INT_m))^2)*((Res_3sd/3)^2) + ...
      ((((h_hsize+w_hsize)*Res_m)/((W_INT_m^2)))^2)*((W_INT_3sd/3)^2);
internal= (10^-3)*0.38*Rint_internal*Cint_internal;
Variance_internal=((0.38*Cint_internal)^2)*(Variance_Rint_internal) + ...
      ((0.38*Rint_internal)^2)*(Variance_Cint_internal);
SD_internal= (10^-3)*sqrt(Variance_internal);
Local_Skew= internal + 3*SD_internal + 6*SD_Pdly(sz+1);
```

```
end

else

%Local and Global interconnect buffer optimisation
%H-Tree interconnect construction
for i=1:alpha
L((2*i)-1)= w/4/(2^(i-1)); %L is interconnect length per stage
L(2*i)= h/4/(2^(i-1));
W((2*i)-1)=2^(alpha-i); %W is interconnect width multiplicative factor
W(2*i)=2^(alpha-i);
end;

sz=size(L,2);
for i=sz:-1:1
Cint(i)=L(i)*DielCon*1e-6*((W_INT_m*W(i)/H_INT_m)+0.77+ ...
    1.06*((W_INT_m*W(i)/H_INT_m)^0.25+(T_INT/H_INT_m)^0.5))/1e-15;
Variance_Cint(i)=((L(i)*DielCon*1e-6/1e-15*((W(i)/H_INT_m) + ...
    (1.06*0.25*((W_INT_m*W(i)/H_INT_m)^(0.25-1))*(W(i)/H_INT_m))))^2)*...
    ((W_INT_3sd/3)^2) + ((L(i)*DielCon*1e-6/1e-15*(-(W_INT_m*W(i))*...
    (H_INT_m)^-2 + (1.06*( 0.25*((W_INT_m*W(i)/H_INT_m)^(0.25-1))*...
    (-(W_INT_m*W(i))*(H_INT_m)^-2) + 0.5*((T_INT/H_INT_m)^(0.5-1))*...
    -T_INT*(H_INT_m)^-2))))^2)*((H_INT_3sd/3)^2);
Rint(i)=L(i)*R/W(i);
Variance_Rint(i)=((L(i)/(W_INT_m*W(i)))^2)*((Res_3sd/3)^2) + ...
    (((L(i)*Res_m)/((W_INT_m^2)*W(i)))^2)*((W_INT_3sd/3)^2);
end;

%Buffer sizing algorithm to meet clock skew and slew targets
while (Max_Skew>Target_GloSkew)||(Local_Skew>Target_LocSkew)||(RISE>Target_Rise)

lamda=lamda-0.1;
if (lamda<1)
 lamda=lamda+0.333333; %also debug purpose
 break;
end

%Buffer size
x(sz+1)= round(C_L/(lamda*Co-Cd));
x(sz)= round( 2.3*Ro*2*(x(sz+1)*Co+...
    Cint(sz))/(2.3*Ro*C_L/x(sz+1)-Rint(sz)*...
    (2.3*x(sz+1)*Co+Cint(sz))));
for i=sz:-1:2
x(i-1)= round( (2.3*2*Ro*(x(i)*Co+Cint(i-1))) / ...
    (2.3*2*Ro/x(i)*(x(i+1)*Co+Cint(i))+ Rint(i)*(2.3*x(i+1)*Co+Cint(i)) ...
    - Rint(i-1)*(2.3*x(i)*Co+Cint(i-1))));
```

```
if(x(i-1)<1)
x(i-1)=1; %really high value for debug purpose
end
end

%x(1) is clock root buffer size
Variance_Co=((Lt_m*(wt_nmos+wt_pmos)*1e-6*1e-6*DielCon/(Tox_m)^2)^2)*...
 ((Tox_3sd/3)^2) +(((wt_nmos+wt_pmos)*1e-6*1e-6*DielCon/Tox_m)^2)*...
    ((Lt_3sd/3)^2) ;
Variance_K= ((mu*(DielCon/(Tox_m)^2)*(wt_nmos/Lt_m))^2)*...
    ((Tox_3sd/3)^2)+((mu*(DielCon/Tox_m)*(wt_nmos/(Lt_m)^2))^2)...
    *((Lt_3sd/3)^2);
Variance_Ro= ((1/((K^2)*(Vdd-Vt_m)))^2)*Variance_K+ ...
    ((1/(K*((Vdd-Vt_m)^2)))^2)*((Vt_3sd/3)^2);

%Interconnect propagation delay estimation and Rise time estimation
%Using Lumped-RC interconnect model

for i=1:sz
%Propagation delay
Pdly(i)=(10^-3)*((0.69*(Ro/x(i))*(2*x(i+1)*Co + x(i)*Cd + 2*Cint(i)))...
    + 0.38*Rint(i)*Cint(i) + 0.69*Rint(i)*(x(i+1)*Co));
%Rise Time
Rdly(i)=(10^-3)*((2.3*(Ro/x(i))*(2*x(i+1)*Co + x(i)*Cd + 2*Cint(i)))...
    + 1.0*Rint(i)*Cint(i) + 2.3*Rint(i)*(x(i+1)*Co));
Variance_Rdly(i)=(((2.3/x(i))*(2*x(i+1)*Co + x(i)*Cd + 2*Cint(i)))^2)*...
    (Variance_Ro) + ((2.3*(((Ro/x(i))*(2*x(i+1)))+...
    (Rint(i)*x(i+1))))^2)*(Variance_Co)+ ...
    ((Cint(i)+(2.3*(x(i+1)*Co)))^2)*(Variance_Rint(i))+ ...
    (((2.3*2*Ro/x(i))+Rint(i))^2)*(Variance_Cint(i));
SD_Rdly(i)= (10^-3)*sqrt(Variance_Rdly(i));
Variance_Pdly(i)=(((0.69/x(i))*(2*x(i+1)*Co + x(i)*Cd +...
    2*Cint(i)))^2)*(Variance_Ro) + ...
    ((0.69*(((Ro/x(i))*(2*x(i+1)))+(Rint(i)*x(i+1))))^2)*(Variance_Co)+ ...
    (((0.38*Cint(i))+(0.69*(x(i+1)*Co)))^2)*(Variance_Rint(i))+ ...
    (((0.69*2*Ro/x(i))+(0.38*Rint(i)))^2)*(Variance_Cint(i));
SD_Pdly(i)= (10^-3)*sqrt(Variance_Pdly(i));
end

Pdly(sz+1)=(10^-3)*0.69*(Ro/x(sz+1))*( C_L + x(sz+1)*Cd );
Rdly(sz+1)=(10^-3)*2.3*(Ro/x(sz+1))*( C_L + x(sz+1)*Cd );
Variance_Rdly(sz+1)=(((2.3/x(sz+1))*( C_L + x(sz+1)*Cd ))^2)*(Variance_Ro);
SD_Rdly(sz+1)= (10^-3)*sqrt(Variance_Rdly(sz+1));
Variance_Pdly(sz+1)=(((0.69/x(sz+1))*( C_L + x(sz+1)*Cd ))^2)*(Variance_Ro);
SD_Pdly(sz+1)= (10^-3)*sqrt(Variance_Pdly(sz+1));
```

```
TPdly=0;
TRdly=0;
for i=1:sz+1
TPdly=TPdly+Pdly(i);
TRdly=TRdly+Rdly(i);
end

%PERI estimation formula for clock slew estimation
Rdly_out(1)= Rdly(1);
for i=2:sz+1
Rdly_out(i)= sqrt((Rdly_out(i-1)^2)+(Rdly(i)^2));
end
RISE=Rdly_out(sz+1);
RISE2=max(Rdly); %checking across all buffers almost equal transition times

%Local Skew calculation
h_hsize= h/(2^alpha);
w_hsize= w/(2^alpha);
Cint_internal=(h_hsize+w_hsize)*DielCon*1e-6*((W_INT_m/H_INT_m)+...
      0.77+1.06*((W_INT_m/H_INT_m)^0.25+(T_INT/H_INT_m)^0.5))/1e-15;
Variance_Cint_internal=(((h_hsize+w_hsize)*DielCon*1e-6/1e-15*...
      ((1/H_INT_m) + (1.06*0.25*((W_INT_m/H_INT_m)^(0.25-1))*...
      (1/H_INT_m))))^2)*((W_INT_3sd/3)^2) + ...
      (((h_hsize+w_hsize)*DielCon*1e-6/1e-15*(-(W_INT_m)*...
      (H_INT_m)^-2 + (1.06*( 0.25*((W_INT_m/H_INT_m)^(0.25-1))*...
      (-(W_INT_m)*(H_INT_m)^-2) + 0.5*((T_INT/H_INT_m)^(0.5-1))*...
      -T_INT*(H_INT_m)^-2))))^2)*((H_INT_3sd/3)^2);
Rint_internal=(h_hsize+w_hsize)*R;
Variance_Rint_internal=(((h_hsize+w_hsize)/(W_INT_m))^2)*...
      ((Res_3sd/3)^2) +((((h_hsize+w_hsize)*Res_m)/((W_INT_m^2)))^2)...
      *((W_INT_3sd/3)^2);
internal= (10^-3)*0.38*Rint_internal*Cint_internal;
Variance_internal=((0.38*Cint_internal)^2)*(Variance_Rint_internal) + ...
      ((0.38*Rint_internal)^2)*(Variance_Cint_internal);
SD_internal= (10^-3)*sqrt(Variance_internal);
Local_Skew= internal + 3*SD_internal + 6*SD_Pdly(sz+1);

MSk=0;
VSk=0;
%Global skew calculation Ref: Jiang et al
for i=1:sz
 temp=0;
 for j=1:i
temp=sqrt((((pi-1)/pi)^(j-1))*((SD_Pdly(sz-i+j))^2));
 end
 MSk=MSk+temp;
 VSk=VSk+((((pi-1)/pi)^i)*(SD_Pdly(i)^2));
end

%Recursive algorithm to calculate correlation coefficient
varskew(sz+1)=(SD_internal^2) + 2*(SD_Pdly(sz+1)^2);
for i=sz:-1:1
 vardelay=0;
 for j=1:sz
 vardelay=vardelay+((((pi-1)/pi)^j)*(SD_Pdly(j)^2));
 end
 cofu(i)=1-(varskew(i+1)/(2*vardelay));
 varskew(i)=2*(1-cofu(i))*vardelay;
end

%Global Clock Skew estimation considering technology variations
Mean_Skew=(2/sqrt(pi))*MSk;
Coeff=cofu(1); %correlation coefficient
Var_Skew=2*(1-Coeff)*VSk;
SD_Skew=sqrt(Var_Skew);
Max_Skew=Mean_Skew+(3*sqrt(Var_Skew)); %Mean + 3*sigma

end
end

%CDN capacitance calculation
Cint_Tot=0;
for i=1:alpha
Cint_Tot=Cint_Tot+(Cint(2*i-1)*(2^(2*i-1)))+(Cint(2*i)*(4^i));
end

%CDN total buffer size calculation
Buff_Tot=x(1);
no=1;
for i=1:alpha
Buff_Tot=Buff_Tot+(x(2*i)*(2^(2*i-1)))+(x(2*i+1)*(4^i));
no=no+(2^(2*i-1))+(4^i);
end

Cb=Co+Cd;
Total_Cap= (Cb*Buff_Tot + Cint_Tot + N*C_L)*1e-15; %With FFs
Cap_Clk= (Cb*Buff_Tot + Cint_Tot )*1e-15;

%CDN Power estimation
Power=Total_Cap*Vdd*Vdd*f;
Power_Clk=Cap_Clk*Vdd*Vdd*f;
Cap_Wire= Cint_Tot;
```

```
Power_Wire=Cap_Wire*Vdd*Vdd*f*1e-15;
Cap_Buff= Cb*Buff_Tot;
Power_Buff=Cap_Buff*Vdd*Vdd*f*1e-15;
Power_Load=N*C_L*Vdd*Vdd*f*1e-15;
%Store CDN parameters and power estimation in Matrix
MAT(y,1)=h;
MAT(y,2)=w;
MAT(y,3)=Power;
MAT(y,4)=((MAT(1,3)-(NI(y)*Power))/(MAT(1,3)))*100;
MAT(y,5)=Power_Clk;
MAT(y,6)=((MAT(1,5)-(NI(y)*Power_Clk))/(MAT(1,5)))*100;
MAT(y,7)=Power_Buff;
MAT(y,8)=((MAT(1,7)-(NI(y)*Power_Buff))/(MAT(1,7)))*100;
MAT(y,9)=Power_Wire;
MAT(y,10)=((MAT(1,9)-(NI(y)*Power_Wire))/(MAT(1,9)))*100;
MAT(y,11)=CP;
MAT(y,12)=lamda;

%Clock Tree Synthesis and Buffer Optimisation of GALS partitions
for y=2:33

h=GALS(y,1); %floorplan height (um)
w=GALS(y,2); %floorplan width (um)
%initialise to enter while loop
lamda=50;
Max_Skew=400;
Local_Skew=400;
RISE=1000;
L=0;
W=0;
Cint=0;
Variance_Cint=0;
Rint=0;
Variance_Rint=0;
x=0;
Pdly=0;
Rdly=0;
Rdly_out=0;
Testdly=0;
Variance_Rdly=0;
Variance_Pdly=0;
SD_Rdly=0;
SD_Pdly=0;

FFs_sink=256; %Max allowed FFs per sink
FF_Tot=round(SinkD*(h*w)*1e-6);
```

```
Total_L=FF_Tot*FF_Cap; %fF
C_L=FFs_sink*FF_Cap;
N=Total_L/C_L; %no of sinks
alpha=(log(N)/log(4)); %alpha denotes Number of Hs– alpha*2 =clock tree levels

if alpha < 0
alpha=0;
end

%Algorithm to distribute FFs evenly
if mod(2*alpha,2)>0
 alpha=ceil(log(N)/log(4));
 N=4^alpha;
 FFs_sink=round(FF_Tot/N);
 C_L=FFs_sink*FF_Cap;
end

D=((2^alpha)-1)*(h+w)/((2^(alpha+1))); %distance form root of tree to sink

%Clock Tree buffer optimisation
if alpha == 0

%Only Local interconnect buffer optimisation
Max_Skew=0;
sz=0;

while (Local_Skew>Target_LocSkew)||(RISE>Target_Rise)

lamda=lamda-0.1;
if (lamda<1)
 lamda=lamda+0.333333; %also debug purpose
 break;
end

Variance_K= ((mu*(DielCon/(Tox_m)^2)*(wt_nmos/Lt_m))^2)*...
      ((Tox_3sd/3)^2)+ ((mu*(DielCon/Tox_m)*(wt_nmos/(Lt_m)^2))^2)...
      *((Lt_3sd/3)^2);
Variance_Ro= ((1/((K^2)*(Vdd-Vt_m)))^2)*Variance_K+ ...
      ((1/(K*((Vdd-Vt_m)^2)))^2)*((Vt_3sd/3)^2);
x(sz+1)= round(C_L/(lamda*Co-Cd));
RISE=(10^-3)*2.3*(Ro/x(sz+1))*( C_L + x(sz+1)*Cd );
Pdly(sz+1)=(10^-3)*0.69*(Ro/x(sz+1))*( C_L + x(sz+1)*Cd );
Variance_Pdly(sz+1)=(((0.69/x(sz+1))*( C_L + x(sz+1)*Cd ))^2)*(Variance_Ro);
SD_Pdly(sz+1)= (10^-3)*sqrt(Variance_Pdly(sz+1));

%Local Skew calculation
```

```
h_hsize= h/(2^alpha);
w_hsize= w/(2^alpha);
Cint_internal=(h_hsize+w_hsize)*DielCon*1e-6*((W_INT_m/H_INT_m)+...
    0.77+1.06*((W_INT_m/H_INT_m)^0.25+(T_INT/H_INT_m)^0.5))/1e-15;
Variance_Cint_internal=(((h_hsize+w_hsize)*DielCon*1e-6/1e-15*...
    ((1/H_INT_m) + (1.06*0.25*((W_INT_m/H_INT_m)^(0.25-1))*...
    (1/H_INT_m))))^2)*((W_INT_3sd/3)^2) + ...
    (((h_hsize+w_hsize)*DielCon*1e-6/1e-15*(-(W_INT_m)*...
    (H_INT_m)^-2 + (1.06*( 0.25*((W_INT_m/H_INT_m)^(0.25-1))*...
    (-(W_INT_m)*(H_INT_m)^-2) + 0.5*((T_INT/H_INT_m)^(0.5-1))*...
    -T_INT*(H_INT_m)^-2))))^2)*((H_INT_3sd/3)^2);
Rint_internal=(h_hsize+w_hsize)*R;
Variance_Rint_internal=(((h_hsize+w_hsize)/(W_INT_m))^2)*...
    ((Res_3sd/3)^2) + ((((h_hsize+w_hsize)*Res_m)/((W_INT_m^2)))^2)...
    *((W_INT_3sd/3)^2);
internal= (10^-3)*0.38*Rint_internal*Cint_internal;
Variance_internal=((0.38*Cint_internal)^2)*(Variance_Rint_internal) + ...
    ((0.38*Rint_internal)^2)*(Variance_Cint_internal);
SD_internal= (10^-3)*sqrt(Variance_internal);
Local_Skew= internal + 3*SD_internal + 6*SD_Pdly(sz+1);

Local_Skew= internal + 3*SD_internal + 6*SD_Pdly(sz+1);
end

else

%Local and Global interconnect buffer optimisation

%H-Tree interconnect construction
for i=1:alpha
L((2*i)-1)= w/4/(2^(i-1));%L is interconnect length per stage
L(2*i)= h/4/(2^(i-1));
W((2*i)-1)=2^(alpha-i); %W is interconnect width multiplicative factor
W(2*i)=2^(alpha-i);
end;

sz=size(L,2);
for i=sz:-1:1
Cint(i)=L(i)*DielCon*1e-6*((W_INT_m*W(i)/H_INT_m)+0.77+ ...
    1.06*((W_INT_m*W(i)/H_INT_m)^0.25+(T_INT/H_INT_m)^0.5))/1e-15;
Variance_Cint(i)=((L(i)*DielCon*1e-6/1e-15*((W(i)/H_INT_m) + ...
    (1.06*0.25*((W_INT_m*W(i)/H_INT_m)^(0.25-1))*(W(i)/H_INT_m))))^2)*...
    ((W_INT_3sd/3)^2) + ((L(i)*DielCon*1e-6/1e-15*(-(W_INT_m*W(i))*...
    (H_INT_m)^-2 + (1.06*( 0.25*((W_INT_m*W(i)/H_INT_m)^(0.25-1))*...
    (-(W_INT_m*W(i))*(H_INT_m)^-2) + 0.5*((T_INT/H_INT_m)^(0.5-1))*...
    -T_INT*(H_INT_m)^-2))))^2)*((H_INT_3sd/3)^2);
```

```
Rint(i)=L(i)*R/W(i);
Variance_Rint(i)=((L(i)/(W_INT_m*W(i)))^2)*((Res_3sd/3)^2) + ...
      (((L(i)*Res_m)/((W_INT_m^2)*W(i)))^2)*((W_INT_3sd/3)^2);
end;

%Buffer sizing algorithm to meet clock skew and slew targets
while (Max_Skew>Target_GloSkew)||(Local_Skew>Target_LocSkew)||(RISE>Target_Rise)

lamda=lamda-0.1;
if (lamda<1)
 lamda=lamda+0.333333; %also debug purpose
 break;
end

%Buffer size
x(sz+1)= round(C_L/(lamda*Co-Cd));
x(sz)= round( 2.3*Ro*2*(x(sz+1)*Co+...
      Cint(sz))/(2.3*Ro*C_L/x(sz+1)-Rint(sz)*(2.3*x(sz+1)*Co+Cint(sz))));
for i=sz:-1:2
x(i-1)= round( (2.3*2*Ro*(x(i)*Co+Cint(i-1))) / ...
      (2.3*2*Ro/x(i)*(x(i+1)*Co+Cint(i))+ Rint(i)*(2.3*x(i+1)*Co+Cint(i)) ...
      - Rint(i-1)*(2.3*x(i)*Co+Cint(i-1))));
if(x(i-1)<1)
x(i-1)=1; %really high value for debug purpose
end
end

%x(1) is clock root buffer size
Variance_Co=((Lt_m*(wt_nmos+wt_pmos)*1e-6*1e-6*DielCon/(Tox_m)^2)^2)*...
      ((Tox_3sd/3)^2) +(((wt_nmos+wt_pmos)*1e-6*1e-6*...
      DielCon/Tox_m)^2)*((Lt_3sd/3)^2) ;
Variance_K= ((mu*(DielCon/(Tox_m)^2)*(wt_nmos/Lt_m))^2)*((Tox_3sd/3)^2)+ ...
      ((mu*(DielCon/Tox_m)*(wt_nmos/(Lt_m)^2))^2)*((Lt_3sd/3)^2);
Variance_Ro= ((1/((K^2)*(Vdd-Vt_m)))^2)*Variance_K+ ...
      ((1/(K*((Vdd-Vt_m)^2)))^2)*((Vt_3sd/3)^2);

%Interconnect propagation delay estimation and Rise time estimation
%Using Lumped-RC interconnect model
for i=1:sz
%Propagation delay
Pdly(i)=(10^-3)*((0.69*(Ro/x(i))*(2*x(i+1)*Co + x(i)*Cd + 2*Cint(i)))...
      + 0.38*Rint(i)*Cint(i) + 0.69*Rint(i)*(x(i+1)*Co));
%Rise Time Rdly(i)=(10^-3)*((2.3*(Ro/x(i))*(2*x(i+1)*Co + x(i)*Cd + 2*Cint(i)))...
      + 1.0*Rint(i)*Cint(i) + 2.3*Rint(i)*(x(i+1)*Co));
Variance_Rdly(i)=(((2.3/x(i))*(2*x(i+1)*Co + x(i)*Cd + 2*Cint(i)))^2)*...
      (Variance_Ro) + ((2.3*(((Ro/x(i))*(2*x(i+1)))+...
    (Rint(i)*x(i+1))))^2)*(Variance_Co)+ ...
    ((Cint(i)+(2.3*(x(i+1)*Co)))^2)*(Variance_Rint(i))+ ...
    (((2.3*2*Ro/x(i))+Rint(i))^2)*(Variance_Cint(i));
SD_Rdly(i)= (10^-3)*sqrt(Variance_Rdly(i));
Variance_Pdly(i)=(((0.69/x(i))*(2*x(i+1)*Co + x(i)*Cd + 2*Cint(i)))^2)*...
    (Variance_Ro) +((0.69*(((Ro/x(i))*(2*x(i+1)))+...
    (Rint(i)*x(i+1))))^2)*(Variance_Co)+ ...
    (((0.38*Cint(i))+(0.69*(x(i+1)*Co)))^2)*(Variance_Rint(i))+ ...
    (((0.69*2*Ro/x(i))+(0.38*Rint(i)))^2)*(Variance_Cint(i));
SD_Pdly(i)= (10^-3)*sqrt(Variance_Pdly(i));
end

Pdly(sz+1)=(10^-3)*0.69*(Ro/x(sz+1))*( C_L + x(sz+1)*Cd );
Rdly(sz+1)=(10^-3)*2.3*(Ro/x(sz+1))*( C_L + x(sz+1)*Cd );
Variance_Rdly(sz+1)=(((2.3/x(sz+1))*( C_L + x(sz+1)*Cd ))^2)*(Variance_Ro);
SD_Rdly(sz+1)= (10^-3)*sqrt(Variance_Rdly(sz+1));
Variance_Pdly(sz+1)=(((0.69/x(sz+1))*( C_L + x(sz+1)*Cd ))^2)*(Variance_Ro);
SD_Pdly(sz+1)= (10^-3)*sqrt(Variance_Pdly(sz+1));

TPdly=0;
TRdly=0;
for i=1:sz+1
TPdly=TPdly+Pdly(i);
TRdly=TRdly+Rdly(i);
end

%PERI estimation formula for clock slew estimation
Rdly_out(1)= Rdly(1);
for i=2:sz+1
Rdly_out(i)= sqrt((Rdly_out(i-1)^2)+(Rdly(i)^2));
end
RISE=Rdly_out(sz+1);
RISE2=max(Rdly); %checking across all buffers almost equal transition times

%Local Skew calculation
h_hsize= h/(2^alpha);
w_hsize= w/(2^alpha);
Cint_internal=(h_hsize+w_hsize)*DielCon*1e-6*((W_INT_m/H_INT_m)+0.77+ ...
    1.06*((W_INT_m/H_INT_m)^0.25+(T_INT/H_INT_m)^0.5))/1e-15;
Variance_Cint_internal=(((h_hsize+w_hsize)*DielCon*1e-6/1e-15*...
    ((1/H_INT_m) + (1.06*0.25*((W_INT_m/H_INT_m)^(0.25-1))*...
    (1/H_INT_m))))^2)*((W_INT_3sd/3)^2) + ...
    (((h_hsize+w_hsize)*DielCon*1e-6/1e-15*(-(W_INT_m)*...
    (H_INT_m)^-2 + (1.06*( 0.25*((W_INT_m/H_INT_m)^(0.25-1))*...
    (-(W_INT_m)*(H_INT_m)^-2) + 0.5*((T_INT/H_INT_m)^(0.5-1))*...
    -T_INT*(H_INT_m)^-2))))^2)*((H_INT_3sd/3)^2);
```

```
Rint_internal=(h_hsize+w_hsize)*R;
Variance_Rint_internal=(((h_hsize+w_hsize)/(W_INT_m))^2)*...
      ((Res_3sd/3)^2) +((((h_hsize+w_hsize)*Res_m)/...
      ((W_INT_m^2)))^2)*((W_INT_3sd/3)^2);
internal= (10^-3)*0.38*Rint_internal*Cint_internal;
Variance_internal=((0.38*Cint_internal)^2)*(Variance_Rint_internal) + ...
      ((0.38*Rint_internal)^2)*(Variance_Cint_internal);
SD_internal= (10^-3)*sqrt(Variance_internal);
Local_Skew= internal + 3*SD_internal + 6*SD_Pdly(sz+1);

MSk=0;
VSk=0;
%Global Skew Algorithm Ref: Jiang et al
%sz+1 is used to calculate local skew
for i=1:sz
temp=0;
for j=1:i
temp=sqrt((((pi-1)/pi)^(j-1))*((SD_Pdly(sz-i+j))^2));
end
MSk=MSk+temp;
VSk=VSk+((((pi-1)/pi)^i)*(SD_Pdly(i)^2));
end

%Recursive algorithm to calculate correlation coefficient
varskew(sz+1)=(SD_internal^2) + 2*(SD_Pdly(sz+1)^2);
for i=sz:-1:1
 vardelay=0;
 for j=1:sz
 vardelay=vardelay+((((pi-1)/pi)^j)*(SD_Pdly(j)^2));
 end
 cofu(i)=1-(varskew(i+1)/(2*vardelay));
 varskew(i)=2*(1-cofu(i))*vardelay;
end

%Global Clock Skew estimation considering technology variations
Mean_Skew=(2/sqrt(pi))*MSk;
Coeff=cofu(1); %correlation coefficient
Var_Skew=2*(1-Coeff)*VSk;
SD_Skew=sqrt(Var_Skew);
Max_Skew=Mean_Skew+(3*sqrt(Var_Skew)); %Mean + 3*sigma

end
end

%Final Clock Tree parameter estimation after buffer sizing
```

```
if alpha == 0

Max_Skew=0;
sz=0;
Variance_K= ((mu*(DielCon/(Tox_m)^2)*(wt_nmos/Lt_m))^2)*...
    ((Tox_3sd/3)^2)+ ((mu*(DielCon/Tox_m)*...
    (wt_nmos/(Lt_m)^2))^2)*((Lt_3sd/3)^2);
Variance_Ro= ((1/((K^2)*(Vdd-Vt_m)))^2)*Variance_K+ ...
    ((1/(K*((Vdd-Vt_m)^2)))^2)*((Vt_3sd/3)^2);
x(sz+1)= round(C_L/(lamda*Co-Cd));
RISE=(10^-3)*2.3*(Ro/x(sz+1))*( C_L + x(sz+1)*Cd );
Pdly(sz+1)=(10^-3)*0.69*(Ro/x(sz+1))*( C_L + x(sz+1)*Cd );
Variance_Pdly(sz+1)=(((0.69/x(sz+1))*( C_L + x(sz+1)*Cd ))^2)*(Variance_Ro);
SD_Pdly(sz+1)= (10^-3)*sqrt(Variance_Pdly(sz+1));

%Local Skew calculation
h_hsize= h/(2^alpha);
w_hsize= w/(2^alpha);
Cint_internal=(h_hsize+w_hsize)*DielCon*1e-6*((W_INT_m/H_INT_m)+0.77+ ...
    1.06*((W_INT_m/H_INT_m)^0.25+(T_INT/H_INT_m)^0.5))/1e-15;
Variance_Cint_internal=(((h_hsize+w_hsize)*DielCon*1e-6/1e-15*...
    ((1/H_INT_m) + (1.06*0.25*((W_INT_m/H_INT_m)^(0.25-1))*...
    (1/H_INT_m))))^2)*((W_INT_3sd/3)^2) + ...
    (((h_hsize+w_hsize)*DielCon*1e-6/1e-15*(-(W_INT_m)*...
    (H_INT_m)^-2 + (1.06*( 0.25*((W_INT_m/H_INT_m)^(0.25-1))*...
    (-(W_INT_m)*(H_INT_m)^-2) + 0.5*((T_INT/H_INT_m)^(0.5-1))*...
    -T_INT*(H_INT_m)^-2))))^2)*((H_INT_3sd/3)^2);
Rint_internal=(h_hsize+w_hsize)*R;
Variance_Rint_internal=(((h_hsize+w_hsize)/(W_INT_m))^2)*...
    ((Res_3sd/3)^2) + ((((h_hsize+w_hsize)*Res_m)/...
    ((W_INT_m^2)))^2)*((W_INT_3sd/3)^2);
internal= (10^-3)*0.38*Rint_internal*Cint_internal;
Variance_internal=((0.38*Cint_internal)^2)*(Variance_Rint_internal) + ...
    ((0.38*Rint_internal)^2)*(Variance_Cint_internal);
SD_internal= (10^-3)*sqrt(Variance_internal);
Local_Skew= internal + 3*SD_internal + 6*SD_Pdly(sz+1);

else

x(sz+1)= round(C_L/(lamda*Co-Cd));
x(sz)= round( 2.3*Ro*2*(x(sz+1)*Co+...
    Cint(sz))/(2.3*Ro*C_L/x(sz+1)-Rint(sz)*(2.3*x(sz+1)*Co+Cint(sz))));
for i=sz:-1:2
x(i-1)= round( (2.3*2*Ro*(x(i)*Co+Cint(i-1))) / ...
    (2.3*2*Ro/x(i)*(x(i+1)*Co+Cint(i))+ Rint(i)*(2.3*x(i+1)*Co+Cint(i)) ...
    - Rint(i-1)*(2.3*x(i)*Co+Cint(i-1))));
```

```
if(x(i-1)<1)
x(i-1)=1; %really high value for debug purpose
end
end

%x(1) is clock root buffer size
Variance_Co=((Lt_m*(wt_nmos+wt_pmos)*1e-6*1e-6*DielCon/(Tox_m)^2)^2)*...
 ((Tox_3sd/3)^2) +(((wt_nmos+wt_pmos)*1e-6*1e-6*...
    DielCon/Tox_m)^2)*((Lt_3sd/3)^2) ;
Variance_K= ((mu*(DielCon/(Tox_m)^2)*(wt_nmos/Lt_m))^2)*...
    ((Tox_3sd/3)^2)+ ((mu*(DielCon/Tox_m)*...
    (wt_nmos/(Lt_m)^2))^2)*((Lt_3sd/3)^2);
Variance_Ro= ((1/((K^2)*(Vdd-Vt_m)))^2)*Variance_K+ ...
    ((1/(K*((Vdd-Vt_m)^2)))^2)*((Vt_3sd/3)^2);

%Interconnect propagation delay estimation and Rise time estimation
%Using Lumped-RC interconnect model
for i=1:sz
%Propagation delay
Pdly(i)=(10^-3)*((0.69*(Ro/x(i))*(2*x(i+1)*Co + x(i)*Cd + 2*Cint(i)))...
    + 0.38*Rint(i)*Cint(i) + 0.69*Rint(i)*(x(i+1)*Co));
%Rise Time
Rdly(i)=(10^-3)*((2.3*(Ro/x(i))*(2*x(i+1)*Co + x(i)*Cd + 2*Cint(i)))...
    + 1.0*Rint(i)*Cint(i) + 2.3*Rint(i)*(x(i+1)*Co));
Variance_Rdly(i)=(((2.3/x(i))*(2*x(i+1)*Co + x(i)*Cd +...
    2*Cint(i)))^2)*(Variance_Ro) + ...
    ((2.3*(((Ro/x(i))*(2*x(i+1)))+(Rint(i)*x(i+1))))^2)*(Variance_Co)+ ...
    ((Cint(i)+(2.3*(x(i+1)*Co)))^2)*(Variance_Rint(i))+ ...
    (((2.3*2*Ro/x(i))+Rint(i))^2)*(Variance_Cint(i));
SD_Rdly(i)= (10^-3)*sqrt(Variance_Rdly(i));
Variance_Pdly(i)=(((0.69/x(i))*(2*x(i+1)*Co + x(i)*Cd + ...
    2*Cint(i)))^2)*(Variance_Ro) + ...
    ((0.69*(((Ro/x(i))*(2*x(i+1)))+(Rint(i)*x(i+1))))^2)*(Variance_Co)+ ...
    (((0.38*Cint(i))+(0.69*(x(i+1)*Co)))^2)*(Variance_Rint(i))+ ...
    (((0.69*2*Ro/x(i))+(0.38*Rint(i)))^2)*(Variance_Cint(i));
SD_Pdly(i)= (10^-3)*sqrt(Variance_Pdly(i));
end

Pdly(sz+1)=(10^-3)*0.69*(Ro/x(sz+1))*( C_L + x(sz+1)*Cd );
Rdly(sz+1)=(10^-3)*2.3*(Ro/x(sz+1))*( C_L + x(sz+1)*Cd );
Variance_Rdly(sz+1)=(((2.3/x(sz+1))*( C_L + x(sz+1)*Cd ))^2)*(Variance_Ro);
SD_Rdly(sz+1)= (10^-3)*sqrt(Variance_Rdly(sz+1));
Variance_Pdly(sz+1)=(((0.69/x(sz+1))*( C_L + x(sz+1)*Cd ))^2)*(Variance_Ro);
SD_Pdly(sz+1)= (10^-3)*sqrt(Variance_Pdly(sz+1));

TPdly=0;
```

```
TRdly=0;
for i=1:sz+1
TPdly=TPdly+Pdly(i);
TRdly=TRdly+Rdly(i);
end

%PERI estimation formula for clock slew estimation
Rdly_out(1)= Rdly(1);
for i=2:sz+1
Rdly_out(i)= sqrt((Rdly_out(i-1)^2)+(Rdly(i)^2));
end
RISE=Rdly_out(sz+1);
RISE2=max(Rdly); %checking across all buffers almost equal transition times

%Local SKew calculation
h_hsize= h/(2^alpha);
w_hsize= w/(2^alpha);
Cint_internal=(h_hsize+w_hsize)*DielCon*1e-6*((W_INT_m/H_INT_m)+...
      0.77+ 1.06*((W_INT_m/H_INT_m)^0.25+(T_INT/H_INT_m)^0.5))/1e-15;
Variance_Cint_internal=(((h_hsize+w_hsize)*DielCon*1e-6/1e-15*...
      ((1/H_INT_m) + (1.06*0.25*((W_INT_m/H_INT_m)^(0.25-1))*...
      (1/H_INT_m))))^2)*((W_INT_3sd/3)^2) + ...
      (((h_hsize+w_hsize)*DielCon*1e-6/1e-15*(-(W_INT_m)*...
      (H_INT_m)^-2 + (1.06*( 0.25*((W_INT_m/H_INT_m)^(0.25-1))*...
      (-(W_INT_m)*(H_INT_m)^-2) + 0.5*((T_INT/H_INT_m)^(0.5-1))*...
      -T_INT*(H_INT_m)^-2))))^2)*((H_INT_3sd/3)^2);
Rint_internal=(h_hsize+w_hsize)*R;
Variance_Rint_internal=(((h_hsize+w_hsize)/(W_INT_m))^2)*...
      ((Res_3sd/3)^2) + ...((((h_hsize+w_hsize)*Res_m)/((W_INT_m^2)))^2)...
      *((W_INT_3sd/3)^2);
internal= (10^-3)*0.38*Rint_internal*Cint_internal;
Variance_internal=((0.38*Cint_internal)^2)*(Variance_Rint_internal) + ...
      ((0.38*Rint_internal)^2)*(Variance_Cint_internal);
SD_internal= (10^-3)*sqrt(Variance_internal);
Local_Skew= internal + 3*SD_internal + 6*SD_Pdly(sz+1);
end

MSk=0;
VSk=0;
%Global Skew Calculation Ref: Jiang et al
for i=1:sz
 temp=0;
 for j=1:i
temp=sqrt(((((pi-1)/pi)^(j-1))*((SD_Pdly(sz-i+j))^2));
 end
 MSk=MSk+temp;
 VSk=VSk+((((pi-1)/pi)^i)*(SD_Pdly(i)^2));
end

%Recursive algorithm to calculate correlation coefficient
varskew(sz+1)=(SD_internal^2) + 2*(SD_Pdly(sz+1)^2);
for i=sz:-1:1
 vardelay=0;
 for j=1:sz
 vardelay=vardelay+((((pi-1)/pi)^j)*(SD_Pdly(j)^2));
 end
 cofu(i)=1-(varskew(i+1)/(2*vardelay));
 varskew(i)=2*(1-cofu(i))*vardelay;
end

%Global Clock Skew estimation considering technology variations
Mean_Skew=(2/sqrt(pi))*MSk;
Coeff=cofu(1); %correlation coefficient
Var_Skew=2*(1-Coeff)*VSk;
SD_Skew=sqrt(Var_Skew);
Max_Skew=Mean_Skew+(3*sqrt(Var_Skew)); %Mean + 3*sigma

%CDN capacitance calculation
Cint_Tot=0;
for i=1:alpha
Cint_Tot=Cint_Tot+(Cint(2*i-1)*(2^(2*i-1)))+(Cint(2*i)*(4^i));
end

%CDN total buffer size calculation
Buff_Tot=x(1);
no=1;
for i=1:alpha
Buff_Tot=Buff_Tot+(x(2*i)*(2^(2*i-1)))+(x(2*i+1)*(4^i));
no=no+(2^(2*i-1))+(4^i);
end

Cb=Co+Cd;
Total_Cap= (Cb*Buff_Tot + Cint_Tot + N*C_L)*1e-15; %With FFs
Cap_Clk= (Cb*Buff_Tot + Cint_Tot )*1e-15;

%Individual GALS island CDN Power estimation
Power=Total_Cap*Vdd*Vdd*f;
Power_Clk=Cap_Clk*Vdd*Vdd*f;
Cap_Wire= Cint_Tot;
Power_Wire=Cap_Wire*Vdd*Vdd*f*1e-15;
Cap_Buff= Cb*Buff_Tot;
Power_Buff=Cap_Buff*Vdd*Vdd*f*1e-15;
```

```
Power_Load=N*C_L*Vdd*Vdd*f*1e-15;

%Store CDN parameters and power estimation in Matrix
MAT(y,1)=h;
MAT(y,2)=w;
MAT(y,3)=Power;
MAT(y,4)=((MAT(1,3)-(NI(y)*Power))/(MAT(1,3)))*100;
MAT(y,5)=Power_Clk;
MAT(y,6)=((MAT(1,5)-(NI(y)*Power_Clk))/(MAT(1,5)))*100;
MAT(y,7)=Power_Buff;
MAT(y,8)=((MAT(1,7)-(NI(y)*Power_Buff))/(MAT(1,7)))*100;
MAT(y,9)=Power_Wire;
MAT(y,10)=((MAT(1,9)-(NI(y)*Power_Wire))/(MAT(1,9)))*100;
MAT(y,11)=CP;
MAT(y,12)=lamda;

end

%Inter-Island Communication Metric
Prob_IslandComm(16,3)=0;
woot(16,3)=0;
MAX=16; %to determine no. of inp and output ports (total no. of wrappers)
K=(log2(MAX)/2)-1;
for j=1:4
woot(1,j)=1;
end
for i=2:16
for j=1:4
woot(i,j)=i;
end
Prob_IslandComm(i,1)=0.005*(i-1); %linear 0.5%
Prob_IslandComm(i,2)=0.01*(i-1); %linear 1%
Prob_IslandComm(i,3)=0.005*2*ceil((MAX*sqrt(i)/2^K)-(2*sqrt(MAX))); %bir 0.5%
Prob_IslandComm(i,4)=0.01*2*ceil((MAX*sqrt(i)/2^K)-(2*sqrt(MAX))) ;%bir 1%
end

for i=1:16
pl(i)=i;
end

figure(1);
plot(woot,Prob_IslandComm,'o-');
title('Inter-Island Communication Activity');
ylabel('Active Communication Links per clock cycle');
xlabel('Number of Islands');
```

```
for i=2:33

%Local clock generator - estimation of number of odd inverters
N_osc=round(1/((2/(MAT(i,11)*1e-12))*Ro*(Co+Cd)*1e-15));
if(mod(N_osc,2)==0)
N_osc=N_osc+1; %increasing N will reduce clock frequency so that you have sufficient time for critical path
end

%Wrapper Power - Ref:Upadhay - 180nm scaled to estimate for relevant process technology
Wrapper_Cap=(0.000117/(15e6*1.8*1.8))*0.7^3;
Wrapper_Power(i)=(Wrapper_Cap*Vdd*Vdd)/(MAT(i,11)*1e-12);

%Local clock generator power estimation
LocalOsc_Power(i)=N_osc*(Co+Cd)*1e-15*Vdd*Vdd/(MAT(i,11)*1e-12);

end

%GALS System Power and Latency Estimation
% Here the sliced islands are put together to form single GALS chips of varying granularity.
for k=1:4

 for j=1:16
ma(j)= Prob_IslandComm(j,k);
 end
Design_Latency = 25000; %system execution time in clock cycles
Wrapper_Lat_WC=1; %One clock cycle worst case latency penalty of Pausable wrapper
Wrapper_Prob=1; %worst case anlaysis taken*

%PERF above is the average clock period of GALS chip in seconds
%ott and ot are used to calculate power of wrappers and compute total GALS chip power.
ott(1,k)=NI(1)*MAT(1,3);
ot(1,k)=((ott(1,k)-ott(1,k))/(ott(1,k)))*100;
PERF(1,k) = (1e-12*MAT(1,11));

ott(2,k)=NI(2)*MAT(2,3)+Prob_IslandComm(2,k)*Wrapper_Power(2)+...
  (pl(2)*LocalOsc_Power(2));
ot(2,k)=((ott(1,k)-ott(2,k))/(ott(1,k)))*100;
PERF(2,k) = (NI(2)*(1e-12*MAT(2,11)))/pl(2);

ott(3,k)=NI(3)*MAT(3,3)+NI(4)*MAT(4,3) + ...
  Prob_IslandComm(3,k)*((NI(3)*Wrapper_Power(3)+...
  NI(4)*Wrapper_Power(4))/pl(3))+...
  (pl(3)*((NI(3)*LocalOsc_Power(3)+NI(4)*LocalOsc_Power(4))/pl(3)));
ot(3,k)=((ott(1,k)-ott(3,k))/(ott(1,k)))*100;
PERF(3,k) = (NI(3)*(1e-12*MAT(3,11))+NI(4)*(1e-12*MAT(4,11)))/pl(3);
```

```
ott(4,k)=NI(5)*MAT(5,3)+Prob_IslandComm(4,k)*Wrapper_Power(5)+...
 (pl(4)*LocalOsc_Power(5));
ot(4,k)=((ott(1,k)-ott(4,k))/(ott(1,k)))*100;
PERF(4,k) = (NI(5)*(1e-12*MAT(5,11)))/pl(4);

ott(5,k)=NI(6)*MAT(6,3)+NI(7)*MAT(7,3)+NI(8)*MAT(8,3)+...
 Prob_IslandComm(5,k)*((NI(6)*Wrapper_Power(6)+..
 NI(7)*Wrapper_Power(7)+NI(8)*Wrapper_Power(8))/pl(5))+...
 (pl(5)*((NI(6)*LocalOsc_Power(6)+NI(7)*LocalOsc_Power(7)+...
 NI(8)*LocalOsc_Power(8))/pl(5)));
ot(5,k)=((ott(1,k)-ott(5,k))/(ott(1,k)))*100;
PERF(5,k) = (NI(6)*(1e-12*MAT(6,11))+NI(7)*(1e-12*MAT(7,11))+...
 NI(8)*(1e-12*MAT(8,11)))/pl(5);

ott(6,k)=NI(9)*MAT(9,3)+Prob_IslandComm(6,k)*Wrapper_Power(9)+...
 (pl(6)*LocalOsc_Power(9));
ot(6,k)=((ott(1,k)-ott(6,k))/(ott(1,k)))*100;
PERF(6,k) = (NI(9)*(1e-12*MAT(9,11)))/pl(6);

ott(7,k)=NI(10)*MAT(10,3)+NI(11)*MAT(11,3)+NI(12)*MAT(12,3)+...
 Prob_IslandComm(7,k)*((NI(10)*Wrapper_Power(10)+..
 NI(11)*Wrapper_Power(11)+NI(12)*Wrapper_Power(12))/pl(7))+...
 (pl(7)*((NI(10)*LocalOsc_Power(10)+NI(11)*LocalOsc_Power(11)+..
 NI(12)*LocalOsc_Power(12))/pl(7)));
ot(7,k)=((ott(1,k)-ott(7,k))/(ott(1,k)))*100;
PERF(7,k) = (NI(10)*(1e-12*MAT(10,11))+NI(11)*(1e-12*MAT(11,11))+...
 NI(12)*(1e-12*MAT(12,11)))/pl(7);

ott(8,k)=NI(13)*MAT(13,3)+Prob_IslandComm(8,k)*Wrapper_Power(13)+...
 (pl(8)*LocalOsc_Power(13));
ot(8,k)=((ott(1,k)-ott(8,k))/(ott(1,k)))*100;
PERF(8,k) = (NI(13)*(1e-12*MAT(13,11)))/pl(8);

ott(9,k)=NI(14)*MAT(14,3)+NI(15)*MAT(15,3)+NI(16)*MAT(16,3)+...
 NI(17)*MAT(17,3)+Prob_IslandComm(9,k)*((NI(14)*...
 Wrapper_Power(14)+NI(15)*Wrapper_Power(15)+...
 NI(16)*Wrapper_Power(16)+NI(17)*Wrapper_Power(17))/pl(9))+...
 (pl(9)*((NI(14)*LocalOsc_Power(14)+NI(15)*LocalOsc_Power(15)+...
 NI(16)*LocalOsc_Power(16)+NI(17)*LocalOsc_Power(17))/pl(9)));
ot(9,k)=((ott(1,k)-ott(9,k))/(ott(1,k)))*100;
PERF(9,k) = (NI(14)*(1e-12*MAT(14,11))+NI(15)*(1e-12*MAT(15,11))+...
 NI(16)*(1e-12*MAT(16,11))+NI(17)*(1e-12*MAT(17,11)))/pl(9);

ott(10,k)=NI(18)*MAT(18,3)+Prob_IslandComm(10,k)*Wrapper_Power(18)+...
 (pl(10)*LocalOsc_Power(18));
ot(10,k)=((ott(1,k)-ott(10,k))/(ott(1,k)))*100;
```

```
PERF(10,k) = (NI(18)*(1e-12*MAT(18,11)))/pl(10);

ott(11,k)=NI(19)*MAT(19,3)+NI(20)*MAT(20,3)+NI(21)*MAT(21,3)+...
 NI(22)*MAT(22,3)+Prob_IslandComm(11,k)*((NI(19)*...
 Wrapper_Power(19)+NI(20)*Wrapper_Power(20)+...
NI(21)*Wrapper_Power(21)+NI(22)*Wrapper_Power(22))/pl(11))+...
 (pl(11)*((NI(19)*LocalOsc_Power(19)+NI(20)*LocalOsc_Power(20)+...
 NI(21)*LocalOsc_Power(21)+NI(22)*LocalOsc_Power(22))/pl(11)));
ot(11,k)=((ott(1,k)-ott(11,k))/(ott(1,k)))*100;
PERF(11,k) = (NI(19)*(1e-12*MAT(19,11))+NI(20)*(1e-12*MAT(20,11))+...
 NI(21)*(1e-12*MAT(21,11))+NI(22)*(1e-12*MAT(22,11)))/pl(11);

ott(12,k)=NI(23)*MAT(23,3)+Prob_IslandComm(12,k)*Wrapper_Power(23)+...
 (pl(12)*LocalOsc_Power(23));
ot(12,k)=((ott(1,k)-ott(12,k))/(ott(1,k)))*100;
PERF(12,k) = (NI(23)*(1e-12*MAT(23,11)))/pl(12);

ott(13,k)=NI(24)*MAT(24,3)+NI(25)*MAT(25,3)+NI(26)*MAT(26,3)+NI(27)*MAT(27,3)+...
 Prob_IslandComm(13,k)*((NI(24)*Wrapper_Power(24)+NI(25)*Wrapper_Power(25)+...
 NI(26)*Wrapper_Power(26)+NI(27)*Wrapper_Power(27))/pl(13))+...
 (pl(13)*((NI(24)*LocalOsc_Power(24)+NI(25)*LocalOsc_Power(25)+...
 NI(26)*LocalOsc_Power(26)+NI(27)*LocalOsc_Power(27))/pl(13)));
ot(13,k)=((ott(1,k)-ott(13,k))/(ott(1,k)))*100;
PERF(13,k) = (NI(24)*(1e-12*MAT(24,11))+NI(25)*(1e-12*MAT(25,11))+...
 NI(26)*(1e-12*MAT(26,11))+NI(27)*(1e-12*MAT(27,11)))/pl(13);

ott(14,k)=NI(28)*MAT(28,3)+Prob_IslandComm(14,k)*Wrapper_Power(28)+...
 (pl(14)*LocalOsc_Power(28));
ot(14,k)=((ott(1,k)-ott(14,k))/(ott(1,k)))*100;
PERF(14,k) = (NI(28)*(1e-12*MAT(28,11)))/pl(14);

ott(15,k)=NI(29)*MAT(29,3)+NI(30)*MAT(30,3)+NI(31)*MAT(31,3)+...
 NI(32)*MAT(32,3)+Prob_IslandComm(15,k)*((NI(29)*...
 Wrapper_Power(29)+NI(30)*Wrapper_Power(30)+...
 NI(31)*Wrapper_Power(31)+NI(32)*Wrapper_Power(32))/pl(15))+...
 (pl(15)*((NI(29)*LocalOsc_Power(29)+NI(30)*LocalOsc_Power(30)+...
 NI(31)*LocalOsc_Power(31)+NI(32)*LocalOsc_Power(32))/pl(15)));
ot(15,k)=((ott(1,k)-ott(15,k))/(ott(1,k)))*100;
PERF(15,k) = (NI(29)*(1e-12*MAT(29,11))+NI(30)*(1e-12*MAT(30,11))+...
 NI(31)*(1e-12*MAT(31,11))+NI(32)*(1e-12*MAT(32,11)))/pl(15);

ott(16,k)=NI(33)*MAT(33,3)+Prob_IslandComm(16,k)*Wrapper_Power(33)+...
 (pl(16)*LocalOsc_Power(33));
ot(16,k)=((ott(1,k)-ott(16,k))/(ott(1,k)))*100;
PERF(16,k) = (NI(33)*(1e-12*MAT(33,11)))/pl(16);
```

```
%below formula is redundant.. kept to prevent changing whole script
for i=1:16
POW(i,k)=ott(i,k);
POW_saving(i,k)=ot(i,k);
end

%Synchronous Chip System Latency
PERF_Overall(1,k) = Design_Latency;
PERF_Overall_Norm(1,k) = 1;
RPP_Prob_Overall(1,k)=1;
for i=2:16
%GALS System Latency
PERF_Overall(i,k) = Prob_IslandComm(i,k)*Design_Latency*...
  Wrapper_Prob*Wrapper_Lat_WC + Design_Latency;
%GALS System Latency normalised comparison to synchronous chip
PERF_Overall_Norm(i,k) = PERF_Overall(i,k)/PERF_Overall(1,k);
%REE metric to compare energy of GALS vs energy of synchronous chip
RPP_Prob_Overall(i,k)=(PERF_Overall(1,k)*PERF(1,k)*POW(1,k))...
  /(PERF_Overall(i,k)*PERF(i,k)*POW(i,k));
end

end

%Synchronous Chip Power Breakdown
figure(2);
island1_tot=NI(1)*MAT(1,3);
island1_FF=NI(1)*(MAT(1,3)-MAT(1,5))/island1_tot;
island1_BU=NI(1)*MAT(1,7)/island1_tot;
island1_WI=NI(1)*MAT(1,9)/island1_tot;
pie3([island1_FF island1_BU island1_WI]);
title('Synchronous Chip Power Breakdown');
colormap summer

%5 Islands Power Breakdown
figure(3);
island5_tot=NI(6)*MAT(6,3)+NI(7)*MAT(7,3)+NI(8)*MAT(8,3)+...
  Prob_IslandComm(5,3)*((NI(6)*Wrapper_Power(6)+...
  NI(7)*Wrapper_Power(7)+NI(8)*Wrapper_Power(8))/pl(5))+...
  (pl(5)*((NI(6)*LocalOsc_Power(6)+NI(7)*LocalOsc_Power(7)+...
  NI(8)*LocalOsc_Power(8))/pl(5)));
island5_FF=(NI(6)*(MAT(6,3)-MAT(6,5))+NI(7)*(MAT(7,3)-MAT(7,5))+...
  NI(8)*(MAT(8,3)-MAT(8,5)))/island5_tot;
island5_BU=(NI(6)*MAT(6,7)+NI(7)*MAT(7,7)+NI(8)*MAT(8,7))/island5_tot;
island5_WI=(NI(6)*MAT(6,9)+NI(7)*MAT(7,9)+NI(8)*MAT(8,9))/island5_tot;
island5_WR=(Prob_IslandComm(5,3)*((NI(6)*Wrapper_Power(6)+...
  NI(7)*Wrapper_Power(7)+NI(8)*Wrapper_Power(8))/pl(5)))/island5_tot;
```

```
island5_OS=(pl(5)*((NI(6)*LocalOsc_Power(6)+NI(7)*LocalOsc_Power(7)+...
  NI(8)*LocalOsc_Power(8))/pl(5)))/island5_tot; is
land5_OVH=island5_WR+island5_OS;
pie3([island5_FF island5_OVH island5_BU island5_WI]);
title('GALS Chip Power Breakdown (5 Islands)');
colormap summer

%7 Islands Power Breakdown
figure(4);
island7_tot=NI(10)*MAT(10,3)+NI(11)*MAT(11,3)+NI(12)*MAT(12,3)+...
  Prob_IslandComm(7,3)*((NI(10)*Wrapper_Power(10)+NI(11)*Wrapper_Power(11)+...
  NI(12)*Wrapper_Power(12))/pl(7))+...
  (pl(7)*((NI(10)*LocalOsc_Power(10)+...
  NI(11)*LocalOsc_Power(11)+NI(12)*LocalOsc_Power(12))/pl(7)));
island7_FF=(NI(10)*(MAT(10,3)-MAT(10,5))+NI(11)*(MAT(11,3)-MAT(11,5))+...
  NI(12)*(MAT(12,3)-MAT(12,5)))/island7_tot;
island7_BU=(NI(10)*MAT(10,7)+NI(11)*MAT(11,7)+NI(12)*MAT(12,7))/island7_tot;
island7_WI=(NI(10)*MAT(10,9)+NI(11)*MAT(11,9)+NI(12)*MAT(12,9))/island7_tot;
island7_WR=(Prob_IslandComm(7,3)*((NI(10)*Wrapper_Power(10)+...
  NI(11)*Wrapper_Power(11)+...
  NI(12)*Wrapper_Power(12))/pl(7)))/island7_tot;
island7_OS=(pl(7)*((NI(10)*LocalOsc_Power(10)+NI(11)*LocalOsc_Power(11)+...
  NI(12)*LocalOsc_Power(12))/pl(7)))/island7_tot;
island7_OVH=island7_WR+island7_OS;
pie3([island7_FF island7_OVH island7_BU island7_WI ]);
title('GALS Chip Power Breakdown (7 Islands)');
colormap summer

%16 Islands Power Breakdown
figure(5);
island16_tot=NI(33)*MAT(33,3)+Prob_IslandComm(16,3)*Wrapper_Power(33)+...
  (pl(16)*LocalOsc_Power(33));
island16_FF=NI(33)*(MAT(33,3)-MAT(33,5))/island16_tot;
island16_BU=NI(33)*MAT(33,7)/island16_tot;
island16_WI=NI(33)*MAT(33,9)/island16_tot;
island16_WR=(Prob_IslandComm(16,3)*Wrapper_Power(33))/island16_tot;
island16_OS=(pl(16)*LocalOsc_Power(33))/island16_tot;
island16_OVH=island16_WR+island16_OS;
pie3([island16_FF island16_OVH island16_BU island16_WI]);
title('GALS Chip Power Breakdown (16 Islands)');
colormap summer

figure(6);
plot(woot,ott,'*-');
title('Chip Power (CDN and GALS Overhead)');
%Taking all FFs switching at every clock cycle.
```

```
ylabel('Power (W)');
xlabel('Number of Islands');

figure(7);
plot(woot,ot,'o-');
title('Chip Power (CDN and GALS Overhead)');
%Taking all FFs switching at every clock cycle.
ylabel('Power Reduction (%)');
xlabel('Number of Islands');

figure(8);
plot(woot,PERF_Overall,'o-');
title('GALS System Latency');
ylabel('Latency (clock cycles)');
xlabel('Number of Islands');

figure(9);
plot(woot,PERF_Overall_Norm,'o-');
title('GALS System Latency Comparison');
ylabel('Normalized to Globally Synchronous Chip');
xlabel('Number of Islands');

figure(10);
plot(woot,RPP_Prob_Overall,'o-');
title('GALS Energy Efficiency Analysis with Inter-Island Communication Activity');
ylabel('Relative Energy Efficiency');
xlabel('Number of Islands');

% %——————————————————
% %90nm CMOS Technology Parameters (Ref:Nassif)
% Vdd=1.2;
% Vt_m=0.35; Vt_3sd=0.04;
% Tox_m=3.5e-9; Tox_3sd=0.42e-9; %gate oxide thickness in metres
% Res_m=0.06 ; Res_3sd=0.019; %Sheet resistance ohm/sq block=(resistivity/T_INT)
% W_INT_m=0.4 ; W_INT_3sd=0.12; %um
% H_INT_m=0.8 ; H_INT_3sd=0.27; %um this is inter layer dielectric (ILD) height not thickness
% DielCon0=885.4e-14; %F/m 8.854e-14F/cm %
% % parameters from Ref: ASU PTM
% T_INT=1.2; %um thickness of wire
% k=2.8;
% % DielCon=k*DielCon0; %F/m
% Cox=DielCon/Tox_m; %gate unit area cap
%mu=0.0067; % charge carrier mobility (670 cm2/(V*s))
% % %Minimum sized inverter (Ref: scaled from Jiang)
%wt_nmos=0.37*90/250; %um wt_pmos=1.1*90/250;
% Lt_m=0.09; Lt_3sd=0.040; %um transistor length
% K=mu*Cox*wt_nmos/Lt_m; %On resistance depends on nmos
% Ro=1/(K*(Vdd-Vt_m)); % output impedance
% Co=Cox*Lt_m*(wt_nmos+wt_pmos)*1e-6*1e-6/1e-15; %fF input capacitance
% Cd=Co/(2.5); %fF output parasitic capactiance (2.5=14.3/5.8 from Ref: Hashimoto)
% % %FF Capacitance
% FF_Cap=4; %fF
% FO4dly=Ro*(4*Co+Cd)*1e-3; %ps
% C_INT=DielCon*1e-6*((W_INT_m/H_INT_m)+0.77+...
 1.06*((W_INT_m/H_INT_m)^0.25+(T_INT/H_INT_m)^0.5))/1e-15;
% %fF/um capacitance of nominal width interconnect
% R=Res_m/W_INT_m; %ohm/um Resistance of nominal width interconnect
%SoC Design Parameters
% SinkD=(18069/2.2)*0.49*0.49; %FFs per mm2
% f=(1e12/(100*FO4dly)); %scaled clock frequency to satisify MUTEX resolution time constraint
% CP=(1/f)*1e12 %Clock period ps
%Island Matrix sizing for 90nm
% %1 island
% GALS(1,1)=4571.4; % GALS(1,2)=4571.4;
% %2 islands
% GALS(2,1)=2285.7; % GALS(2,2)=4571.4;
% %3 islands
% GALS(3,1)=3047.6; % GALS(3,2)=2285.7;
% GALS(4,1)=1523.8;% GALS(4,2)=4571.4;
% %4 islands
% GALS(5,1)=2285.7; % GALS(5,2)=2285.7;
% %5 islands
% GALS(6,1)=1828.6; % GALS(6,2)=2285.7;
% GALS(7,1)=3047.6; % GALS(7,2)=1371.4;
% GALS(8,1)=2742.8;% GALS(8,2)=1523.8;
% %6 islands
% GALS(9,1)=1523.8; % GALS(9,2)=2285.7;
% %7 islands
% GALS(10,1)=2285.7; % GALS(10,2)=1306.1;
% GALS(11,1)=3047.6;% GALS(11,2)=979.6;
% GALS(12,1)=1523.8; % GALS(12,2)=1959.2; %
%8 islands
% GALS(13,1)=2285.7; % GALS(13,2)=1142.8;
% %9 islands
% GALS(14,1)=2285.7; % GALS(14,2)=1015.9;
% GALS(15,1)=1828.6; % GALS(15,2)=1269.8;
% GALS(16,1)=1371.4; % GALS(16,2)=1693.1;
% GALS(17,1)=2742.8; % GALS(17,2)=846.6;
% %10 islands
% GALS(18,1)=914.3; % GALS(18,2)=2285.7;
% %11 islands
% GALS(19,1)=2285.7; % GALS(19,2)=831.2;
% GALS(20,1)=1828.6;% GALS(20,2)=1039;
% GALS(21,1)=1371.4; % GALS(21,2)=1385.3;
% GALS(22,1)=2742.8; % GALS(22,2)=692.6;
% %12 islands
% GALS(23,1)=2285.7; % GALS(23,2)=761.9;
% %13 islands
% GALS(24,1)=2285.7;% GALS(24,2)=703.3;
% GALS(25,1)=1306.1;% GALS(25,2)=1230.8;
% GALS(26,1)=979.6; % GALS(26,2)=1641;
% GALS(27,1)=1959.2; % GALS(27,2)=820.5;
% %14 islands
% GALS(28,1)=653.1;% GALS(28,2)=2285.7;
% %15 islands
% GALS(29,1)=2285.7; % GALS(29,2)=609.5;
% GALS(30,1)=1066.7; % GALS(30,2)=1306.1;
% GALS(31,1)=979.6;% GALS(31,2)=1422.2;
% GALS(32,1)=1959.2; % GALS(32,2)=711.1;
% %16 islands
% GALS(33,1)=1142.8;% GALS(33,2)=1142.8;

% %——————————————————————————
% %130nm CMOS Technology Parameters (Ref: Nassif)
% Vdd=1.3;
% Vt_m=0.4; Vt_3sd=0.03;
% Tox_m=4e-9; Tox_3sd=0.39e-9; %gate oxide thickness in metres
% Res_m=0.055 ; Res_3sd=0.015; %Sheet resistance ohm/sq block=(resistivity/T_INT)
% W_INT_m=0.5 ; W_INT_3sd=0.14; %um
% H_INT_m=0.9 ; H_INT_3sd=0.27; %um this is inter layer dielectric (ILD) height not thickness
% DielCon0=885.4e-14; %F/m 8.854e-14F/cm
% % % parameters from Ref: ASU PTM
% T_INT=1.2; %um thickness of wire
% k=3.2;
% % DielCon=k*DielCon0;
%F/m % Cox=DielCon/Tox_m; %gate unit area cap
% mu=0.0067; % charge carrier mobility (670 cm2/(V*s))
% % %Minimum sized inverter (Ref: scaled from Jiang)
% wt_nmos=0.37*130/250; %um
% wt_pmos=1.1*130/250; %um
% Lt_m=0.13; Lt_3sd=0.045; %um transistor length
% K=mu*Cox*wt_nmos/Lt_m; %On resistance depends on nmos
% Ro=1/(K*(Vdd-Vt_m)); % output impedance
% Co=Cox*Lt_m*(wt_nmos+wt_pmos)*1e-6*1e-6/1e-15; %fF input capacitance
% Cd=Co/(2.5); %fF output parasitic capactiance (2.5=14.3/5.8 from Ref:Hashimoto)
% % %FF Capacitance
% FF_Cap=5.5; %fF
% FO4dly=Ro*(4*Co+Cd)*1e-3
% C_INT=DielCon*1e-6*((W_INT_m/H_INT_m)+0.77+...
 1.06*((W_INT_m/H_INT_m)^0.25+(T_INT/H_INT_m)^0.5))/1e-15;
% %fF/um capacitance of nominal width interconnect
% R=Res_m/W_INT_m; %ohm/um Resistance of nominal width interconnect
%SoC Design Parameters
% SinkD=(18069/2.2)*0.49*0.49*0.49; %FFs per mm2
% f=(1e12/(100*FO4dly)); %scaled clock frequency to satisify MUTEX resolution time constraint
% CP=(1/f)*1e12 %Clock period ps
% %Island Matrix sizing for 130nm
% %1 island
% GALS(1,1)=6530.6;% GALS(1,2)=6530.6;
% %2 islands
% GALS(2,1)=3265.3;% GALS(2,2)=6530.6;
% %3 islands
% GALS(3,1)=3265.3;% GALS(3,2)=4353.7;
% GALS(4,1)=6530.6; % GALS(4,2)=2176.9;
% %4 islands
% GALS(5,1)=3265.3; % GALS(5,2)=3265.3;
% %5 islands
% GALS(6,1)=3265.3; % GALS(6,2)=2612.2;
% GALS(7,1)=4353.7; % GALS(7,2)=1959.2;
% GALS(8,1)=2176.9;% GALS(8,2)=3918.4;
% %6 islands
% GALS(9,1)=3265.3; % GALS(9,2)=2176.9;
% %7 islands
% GALS(10,1)=3265.3; % GALS(10,2)=1865.9;
% GALS(11,1)=4353.7;
% GALS(11,2)=1399.4;% GALS(12,1)=2176.9;
% GALS(12,2)=2798.8;
% %8 islands
% GALS(13,1)=1632.6;% GALS(13,2)=3265.3;
% %9 islands
% GALS(14,1)=1451.2; % GALS(14,2)=3265.3;
% GALS(15,1)=1814.4; % GALS(15,2)=2612.2;
% GALS(16,1)=2418.7; % GALS(16,2)=1959.2;
% GALS(17,1)=1209.4;% GALS(17,2)=3918.4;
% %10 islands
% GALS(18,1)=1306.1;% GALS(18,2)=3265.3;
% %11 islands
% GALS(19,1)=3265.3;% GALS(19,2)=1187.4;
% GALS(20,1)=1484.2;% GALS(20,2)=2612.2;
% GALS(21,1)=1979;% GALS(21,2)=1959.2;
% GALS(22,1)=3918.4; % GALS(22,2)=989.5;
% %12 islands
% GALS(23,1)=1088.4; % GALS(23,2)=3265.3;
% %13 islands
% GALS(24,1)=3265.3; % GALS(24,2)=1004.7;
% GALS(25,1)=1758.2;% GALS(25,2)=1865.9;
% GALS(26,1)=2344.3;% GALS(26,2)=1399.4;
% GALS(27,1)=2798.8;% GALS(27,2)=1172.2;
% %14 islands
% GALS(28,1)=3265.3; % GALS(28,2)=932.9;
% %15 islands
% GALS(29,1)=870.7;% GALS(29,2)=3265.3;
% GALS(30,1)=1865.9; % GALS(30,2)=1523.8;
% GALS(31,1)=2031.7;% GALS(31,2)=1399.4;
% GALS(32,1)=1015.9; % GALS(32,2)=2798.8;
% %16 islands
% GALS(33,1)=1632.6; % GALS(33,2)=1632.6;

% %——————————————————————
% %180nm CMOS Technology Parameters (Ref: Nassif)
% Vdd=1.8;
% Vt_m=0.45; Vt_3sd=0.045;
% Tox_m=4.5e-9; Tox_3sd=0.36e-9; %gate oxide thickness in metres
% Res_m=0.05 ; Res_3sd=0.012; %Sheet resistance ohm/sq block=(resistivity/T_INT)
% W_INT_m=0.65 ; W_INT_3sd=0.17; %um
% H_INT_m=1 ; H_INT_3sd=0.3; %um this is inter layer dielectric (ILD) height not thickness
% DielCon0=885.4e-14; %F/m 8.854e-14F/cm
% % % parameters from Ref: ASU PTM
% T_INT=1.25; %um thickness of wire
% k=3.5;
% DielCon=k*DielCon0; %F/m
% Cox=DielCon/Tox_m; %gate unit area cap
% mu=0.0067; % charge carrier mobility (670 cm2/(V*s))
% % %Minimum sized inverter (Ref: scaled from Jiang)
% wt_nmos=0.37*180/250; %um
% wt_pmos=1.1*180/250; %um
% Lt_m=0.18; Lt_3sd=0.060; %um transistor length
% K=mu*Cox*wt_nmos/Lt_m; %On resistance depends on nmos
% Ro=1/(K*(Vdd-Vt_m)); % output impedance
% Co=Cox*Lt_m*(wt_nmos+wt_pmos)*1e-6*1e-6/1e-15; %fF input capacitance
% Cd=Co/(2.5); %fF output parasitic capactiance (2.5=14.3/5.8 from Ref: Hashimoto)
% % %FF Capacitance
% FF_Cap=7; %fF
% FO4dly=Ro*(4*Co+Cd)*1e-3 %ps
% C_INT=DielCon*1e-6*((W_INT_m/H_INT_m)+0.77+...
 1.06*((W_INT_m/H_INT_m)^0.25+(T_INT/H_INT_m)^0.5))/1e-15
% %fF/um capacitance of nominal width interconnect
% R=Res_m/W_INT_m; %ohm/um Resistance of nominal width interconnect
%SoC Design Parameters
%SinkD=18069/(2.2); %FFs per mm2
```

```
% f=(1e12/(100*FO4dly)); %scaled clock frequency to satisify MUTEX resolution time constraint
% CP=(1/f)*1e12 %Clock period ps
% %Island Matrix sizing for 180nm
% %1 island
% GALS(1,1)=9329.4; % GALS(1,2)=9329.4;
% %2 islands
% GALS(2,1)=4664.7; % GALS(2,2)=9329.4;
% %3 islands
% GALS(3,1)=4664.7;% GALS(3,2)=6219.6; % GALS(4,1)=9329.4; % GALS(4,2)=3109.8;
% %4 islands
% GALS(5,1)=4664.7; % GALS(5,2)=4664.7;
% %5 islands
% GALS(6,1)=4664.7; % GALS(6,2)=3731.8; % GALS(7,1)=6219.6; % GALS(7,2)=2798.8;
% GALS(8,1)=3109.8; % GALS(8,2)=5597.6;
% %6 islands
% GALS(9,1)=4664.7;% GALS(9,2)=3109.8;
% %7 islands
% GALS(10,1)=4664.7; % GALS(10,2)=2665.5;% GALS(11,1)=6219.6; % GALS(11,2)=1999.2;
% GALS(12,1)=3109.8; % GALS(12,2)=3998.3;
% %8 islands
% GALS(13,1)=2332.3;% GALS(13,2)=4664.7;
% %9 islands
% GALS(14,1)=2073.2; % GALS(14,2)=4664.7;% GALS(15,1)=2591.5; % GALS(15,2)=3731.8;
% GALS(16,1)=3455.3; % GALS(16,2)=2798.8; % GALS(17,1)=1727.7; % GALS(17,2)=5597.6;
% %10 islands
% GALS(18,1)=1865.9;% GALS(18,2)=4664.7;
% %11 islands
% GALS(19,1)=4664.7;% GALS(19,2)=1696.3; % GALS(20,1)=2120.3; % GALS(20,2)=3731.8;
% GALS(21,1)=2827.1;% GALS(21,2)=2798.8; % GALS(22,1)=5597.6; % GALS(22,2)=1413.5;
% %12 islands
% GALS(23,1)=1554.9; % GALS(23,2)=4664.7;
% %13 islands
% GALS(24,1)=4664.7; % GALS(24,2)=1435.3; % GALS(25,1)=2511.8; % GALS(25,2)=2665.5;
% GALS(26,1)=3349; % GALS(26,2)=1999.2; % GALS(27,1)=3998.3;% GALS(27,2)=1674.5;
% %14 islands
% GALS(28,1)=4664.7; % GALS(28,2)=1332.8;
% %15 islands
% GALS(29,1)=1243.9; % GALS(29,2)=4664.7; % GALS(30,1)=2665.5; % GALS(30,2)=2176.9;
% GALS(31,1)=2902.5; % GALS(31,2)=1999.2;% GALS(32,1)=1451.2; % GALS(32,2)=3998.3;
% %16 islands
% GALS(33,1)=2332.3; % GALS(33,2)=2332.3;
```

Appendix B

The Verilog codes listed below provide main parts of the behavioural HDL code for bundled-data and elastic-bundle maximal implementations of the 16-point FFT multirate architecture. Note that the datapath and high-level descriptions for both the implementations are the same. Only the control logic codes would change depending on whether the design is a bundled-data implementation or an elastic-bundle implementation.

16-point FFT Top-Level Verilog Code

```
module FFT16async(
    input wire [31:0] D_in,
    output wire [31:0] D_out,
    input wire reqi,
    output wire acki,
    output wire reqo,
    input wire acko,
    input wire reset
    );
    wire [1:0] sel,sel_nxt,sel_temp,select,select_nxt,select_temp;
    wire M1_rout,M2_rout,M3_rout,M4_rout,M1_aout,M2_aout,M3_aout,M4_aout;
    wire FFT4_1_rout,FFT4_2_rout,FFT4_3_rout,FFT4_4_rout;
    wire FFT4_5_rout,FFT4_6_rout,FFT4_7_rout,FFT4_8_rout;
    wire FFT4_1_aout,FFT4_2_aout,FFT4_3_aout,FFT4_4_aout;
    wire FFT4_5_aout,FFT4_6_aout,FFT4_7_aout,FFT4_8_aout;
    wire [31:0] Din,Dout,M1_out,M2_out,M3_out,M4_out,crossbar_out;
    wire [31:0] y1,y2_2,y3_2,y4_2,y2_3,y3_3,y4_3,y2_4,y3_4,y4_4;
    wire [31:0] FFT4_1_Dout,FFT4_2_Dout,FFT4_3_Dout,FFT4_4_Dout;
    wire [31:0] FFT4_5_Dout,FFT4_6_Dout,FFT4_7_Dout,FFT4_8_Dout;
    wire req1,req2,req3,req4,ack1,ack2,ack3,ack4;
    wire r1,r2,r3,r4,a1,a2,a3,a4;
    wire ri,ai,ro,ao,ri_delay;
    wire ci,co,c_M1,c_M2,c_M3,c_M4;
    wire Xr1,Xr2,Xr3,Xr4,Xa1,Xa2,Xa3,Xa4;
    //Twiddle factors
    assign y1[31:16] = $unsigned(16'h1000); //1+0i consider adding a gating logic for this
    assign y1[15:0] = $unsigned(16'h0);
assign y2_2[31:16] = $unsigned(16'h0ec8);
assign y2_2[15:0] = $unsigned(16'hf9e1);
assign y3_2[31:16] = $unsigned(16'h0b50);
assign y3_2[15:0] = $unsigned(16'hf4b0);
assign y4_2[31:16] = $unsigned(16'h061f);
assign y4_2[15:0] = $unsigned(16'hf138);
assign y2_3[31:16] = $unsigned(16'h0b50);
assign y2_3[15:0] = $unsigned(16'hf4b0);
assign y3_3[31:16] = $unsigned(16'h0);
assign y3_3[15:0] = $unsigned(16'hf000);
assign y4_3[31:16] = $unsigned(16'hf4b0);
assign y4_3[15:0] = $unsigned(16'hf4b0);
assign y2_4[31:16] = $unsigned(16'h061f);
assign y2_4[15:0] = $unsigned(16'hf138);
assign y3_4[31:16] = $unsigned(16'hf4b0);
assign y3_4[15:0] = $unsigned(16'hf4b0);
assign y4_4[31:16] = $unsigned(16'hf138);
assign y4_4[15:0] = $unsigned(16'h061f);
handshake_ctl HC1(.lr(reqi),.la(acki),.rr(ri),.ra(ai),.ck(ci),.reset(~reset));
reg32_async R1(.D(D_in),.Q(Din),.ck(ci),.reset(reset));
//delay element for Clock to Q delay
// delay element maynot be required here since, decimatot delays logic
delayele_Reg2Reg d1(.in(ri),.out(ri_delay));
//First stage at ack ai
reg_counter2_async R2(.D(sel_nxt),.Q(sel_temp),.ck(ai),.reset(reset));
//Second stage at ack ~ai
//Time between ai and ~ai is about 400 ps, CtoQ delay is not violated
reg_counter2_async R3(.D(sel_temp),.Q(sel),.ck(~ai),.reset(reset));
//There is enough delay between consecutive ai pulses. Delay element not required . (actually satisfied by delay d1)
counter2_async count1(.in(sel),.out(sel_nxt));
//no delay element required here since request is automatically delayed by logic
decimator4_async decimator(
.R(ri_delay),
.sel(sel),
.a1(a1),
.a2(a2),
.a3(a3),
.a4(a4),
.r1(r1),
.r2(r2),
.r3(r3),
.r4(r4),
.A(ai)
);
FFT4async FFT4_1(
.D_in(Din),
.D_out(FFT4_1_Dout),
.reqi(r1),
.acki(a1),
.reqo(FFT4_1_rout),
.acko(FFT4_1_aout),
.reset(reset)
);
FFT4async FFT4_2(
.D_in(Din),
.D_out(FFT4_2_Dout),
.reqi(r2),
.acki(a2),
.reqo(FFT4_2_rout),
.acko(FFT4_2_aout),
.reset(reset)
);
FFT4async FFT4_3(
.D_in(Din),
.D_out(FFT4_3_Dout),
.reqi(r3),
.acki(a3),
.reqo(FFT4_3_rout),
.acko(FFT4_3_aout),
.reset(reset)
);
FFT4async FFT4_4(
.D_in(Din),
.D_out(FFT4_4_Dout),
.reqi(r4),
.acki(a4),
.reqo(FFT4_4_rout),
.acko(FFT4_4_aout),
.reset(reset)
);
MULT_1_async M1(
.M_in(FFT4_1_Dout),
.clock(c_M1),
.reset(reset),
.M_out(M1_out)
);
MULT16_async M2(
.y1(y1),
.y2(y2_2),
.y3(y3_2),
.y4(y4_2),
.M_in(FFT4_2_Dout),
.clock(c_M2),
.A(FFT4_2_aout),
.reset(reset),
.M_out(M2_out)
);
MULT16_async M3(
.y1(y1),
.y2(y2_3),
.y3(y3_3),
.y4(y4_3),
.M_in(FFT4_3_Dout),
.clock(c_M3),
.A(FFT4_3_aout),
.reset(reset),
.M_out(M3_out)
);
MULT16_async M4(
.y1(y1),
.y2(y2_4),
.y3(y3_4),
.y4(y4_4),
.M_in(FFT4_4_Dout),
.clock(c_M4),
.A(FFT4_4_aout),
.reset(reset),
.M_out(M4_out)
);
FFT16_control_async control(
.reset(~reset),
.FFT4_1_rout(FFT4_1_rout),.FFT4_2_rout(FFT4_2_rout),
.FFT4_3_rout(FFT4_3_rout),.FFT4_4_rout(FFT4_4_rout),
.M1_aout(M1_aout),.M2_aout(M2_aout),.M3_aout(M3_aout),.M4_aout(M4_aout),
.FFT4_1_aout(FFT4_1_aout),.FFT4_2_aout(FFT4_2_aout),
.FFT4_3_aout(FFT4_3_aout),.FFT4_4_aout(FFT4_4_aout),
.c_M1(c_M1),.c_M2(c_M2),.c_M3(c_M3),.c_M4(c_M4),
.M1_rout(M1_rout),.M2_rout(M2_rout),.M3_rout(M3_rout),.M4_rout(M4_rout)
);
crossbar4_async crossbar(
.reset(reset),
.r1(M1_rout),
.r2(M2_rout),
.r3(M3_rout),
.r4(M4_rout),
.a1(M1_aout),
.a2(M2_aout),
.a3(M3_aout),
.a4(M4_aout),
.D1(M1_out),
.D2(M2_out),
.D3(M3_out),
.D4(M4_out),
.REQ1(Xr1),
.REQ2(Xr2),
.REQ3(Xr3),
.REQ4(Xr4),
.ACK1(Xa1),
.ACK2(Xa2),
.ACK3(Xa3),
.ACK4(Xa4),
.Dout(crossbar_out)
);
FFT4async FFT4_5(
.D_in(crossbar_out),
.D_out(FFT4_5_Dout),
.reqi(Xr1),
.acki(Xa1),
.reqo(FFT4_5_rout),
.acko(FFT4_5_aout),
.reset(reset)
);
FFT4async FFT4_6(
.D_in(crossbar_out),
.D_out(FFT4_6_Dout),
.reqi(Xr2),
.acki(Xa2),
.reqo(FFT4_6_rout),
.acko(FFT4_6_aout),
.reset(reset)
);
FFT4async FFT4_7(
.D_in(crossbar_out),
.D_out(FFT4_7_Dout),
.reqi(Xr3),
.acki(Xa3),
.reqo(FFT4_7_rout),
.acko(FFT4_7_aout),
.reset(reset)
);
FFT4async FFT4_8(
.D_in(crossbar_out),
.D_out(FFT4_8_Dout),
.reqi(Xr4),
    .acki(Xa4),
    .reqo(FFT4_8_rout),
    .acko(FFT4_8_aout),
    .reset(reset)
    );
    //delay element for Clock to Q delay from datapath output
    delayele_Reg2Reg d2(.in(FFT4_5_rout),.out(req1));
    delayele_Reg2Reg d3(.in(FFT4_6_rout),.out(req2));
    delayele_Reg2Reg d4(.in(FFT4_7_rout),.out(req3));
    delayele_Reg2Reg d5(.in(FFT4_8_rout),.out(req4));
    //First stage at ack a0
    reg_counter2_async R4(.D(select_nxt),.Q(select_temp),.ck(ao),.reset(reset));
    //Second stage at ack falls ~ao
    //Time between ao and ~ao is 400 ps, CtoQ delay is not violated
    reg_counter2_async R5(.D(select_temp),.Q(select),.ck(~ao),.reset(reset));
    counter2_async count2(.in(select),.out(select_nxt));
    expander4_async expander(
    .sel(select),
    .r1(req1),
    .r2(req2),
    .r3(req3),
    .r4(req4),
    .A(ao),
    .D1(FFT4_5_Dout),
    .D2(FFT4_6_Dout),
    .D3(FFT4_7_Dout),
    .D4(FFT4_8_Dout),
    .R(ro),
    .a1(FFT4_5_aout),
    .a2(FFT4_6_aout),
    .a3(FFT4_7_aout),
    .a4(FFT4_8_aout),
    .Dout(Dout)
    );
    handshake_ctl HC2(.lr(ro),.la(ao),.rr(reqo),.ra(acko),.ck(co),.reset(~reset));
    reg32_async R6(.D(Dout),.Q(D_out),.ck(co),.reset(reset));
    endmodule
```

16-point FFT Control Verilog Code

```
module delayele_inv_7 ( in, out );
    input in;
    output out;
    INVX1 U1 ( .I(in), .O(out) );
    endmodule
```

```
module delayele_inv_6 ( in, out );
input in;
output out;
wire n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14, n15, n16,
n17, n18, n19, n20, n21, n22, n23, n24, n25, n26, n27, n28, n29, n30,
n31, n32, n33, n34, n35, n36, n37, n38, n39, n40, n41, n42, n43, n44,
n45, n46, n47, n48, n49, n50, n51, n52, n53, n54, n55, n56, n57, n58,
n59, n60, n61, n62, n63, n64, n65, n66, n67, n68, n69, n70, n71, n72,
n73, n74, n75, n76, n77, n78, n79, n80, n81, n82, n83, n84, n85, n86,
n87, n88, n89, n90, n91, n92, n93, n94, n95, n96, n97, n98, n99, n100,
n101, n102, n103, n104, n105, n106, n107, n108, n109, n110, n111,
n112, n113, n114, n115, n116, n117;
BUFX1 U1 ( .I(n2), .O(out) );
BUFX1 U2 ( .I(n3), .O(n2) );
BUFX1 U3 ( .I(n4), .O(n3) );
BUFX1 U4 ( .I(n5), .O(n4) );
BUFX1 U5 ( .I(n6), .O(n5) );
BUFX1 U6 ( .I(n7), .O(n6) );
BUFX1 U7 ( .I(n8), .O(n7) );
BUFX1 U8 ( .I(n9), .O(n8) );
BUFX1 U9 ( .I(n10), .O(n9) );
BUFX1 U10 ( .I(n11), .O(n10) );
BUFX1 U11 ( .I(n12), .O(n11) );
BUFX1 U12 ( .I(n13), .O(n12) );
BUFX1 U13 ( .I(n14), .O(n13) );
BUFX1 U14 ( .I(n15), .O(n14) );
BUFX1 U15 ( .I(n16), .O(n15) );
BUFX1 U16 ( .I(n17), .O(n16) );
BUFX1 U17 ( .I(n18), .O(n17) );
BUFX1 U18 ( .I(n19), .O(n18) );
BUFX1 U19 ( .I(n20), .O(n19) );
BUFX1 U20 ( .I(n21), .O(n20) );
BUFX1 U21 ( .I(n22), .O(n21) );
BUFX1 U22 ( .I(n23), .O(n22) );
BUFX1 U23 ( .I(n24), .O(n23) );
BUFX1 U24 ( .I(n25), .O(n24) );
BUFX1 U25 ( .I(n26), .O(n25) );
BUFX1 U26 ( .I(n27), .O(n26) );
BUFX1 U27 ( .I(n28), .O(n27) );
BUFX1 U28 ( .I(n29), .O(n28) );
BUFX1 U29 ( .I(n30), .O(n29) );
BUFX1 U30 ( .I(n31), .O(n30) );
BUFX1 U31 ( .I(n32), .O(n31) );
BUFX1 U32 ( .I(n33), .O(n32) );
BUFX1 U33 ( .I(n34), .O(n33) );
BUFX1 U34 ( .I(n35), .O(n34) );
```

```
BUFX1 U35 ( .I(n36), .O(n35) );
BUFX1 U36 ( .I(n37), .O(n36) );
BUFX1 U37 ( .I(n38), .O(n37) );
BUFX1 U38 ( .I(n39), .O(n38) );
BUFX1 U39 ( .I(n40), .O(n39) );
BUFX1 U40 ( .I(n41), .O(n40) );
BUFX1 U41 ( .I(n42), .O(n41) );
BUFX1 U42 ( .I(n43), .O(n42) );
BUFX1 U43 ( .I(n44), .O(n43) );
BUFX1 U44 ( .I(n45), .O(n44) );
BUFX1 U45 ( .I(n46), .O(n45) );
BUFX1 U46 ( .I(n47), .O(n46) );
BUFX1 U47 ( .I(n48), .O(n47) );
BUFX1 U48 ( .I(n49), .O(n48) );
BUFX1 U49 ( .I(n50), .O(n49) );
BUFX1 U50 ( .I(n51), .O(n50) );
BUFX1 U51 ( .I(n52), .O(n51) );
BUFX1 U52 ( .I(n53), .O(n52) );
BUFX1 U53 ( .I(n54), .O(n53) );
BUFX1 U54 ( .I(n55), .O(n54) );
BUFX1 U55 ( .I(n56), .O(n55) );
BUFX1 U56 ( .I(n57), .O(n56) );
BUFX1 U57 ( .I(n58), .O(n57) );
BUFX1 U58 ( .I(n59), .O(n58) );
BUFX1 U59 ( .I(n60), .O(n59) );
BUFX1 U60 ( .I(n61), .O(n60) );
BUFX1 U61 ( .I(n62), .O(n61) );
BUFX1 U62 ( .I(n63), .O(n62) );
BUFX1 U63 ( .I(n64), .O(n63) );
BUFX1 U64 ( .I(n65), .O(n64) );
BUFX1 U65 ( .I(n66), .O(n65) );
BUFX1 U66 ( .I(n67), .O(n66) );
BUFX1 U67 ( .I(n68), .O(n67) );
BUFX1 U68 ( .I(n69), .O(n68) );
BUFX1 U69 ( .I(n70), .O(n69) );
BUFX1 U70 ( .I(n71), .O(n70) );
BUFX1 U71 ( .I(n72), .O(n71) );
BUFX1 U72 ( .I(n73), .O(n72) );
BUFX1 U73 ( .I(n74), .O(n73) );
BUFX1 U74 ( .I(n75), .O(n74) );
BUFX1 U75 ( .I(n76), .O(n75) );
BUFX1 U76 ( .I(n77), .O(n76) );
BUFX1 U77 ( .I(n78), .O(n77) );
BUFX1 U78 ( .I(n79), .O(n78) );
BUFX1 U79 ( .I(n80), .O(n79) );
BUFX1 U80 ( .I(n81), .O(n80) );
```

```
BUFX1 U81 ( .I(n82), .O(n81) );
BUFX1 U82 ( .I(n83), .O(n82) );
BUFX1 U83 ( .I(n84), .O(n83) );
BUFX1 U84 ( .I(n85), .O(n84) );
BUFX1 U85 ( .I(n86), .O(n85) );
BUFX1 U86 ( .I(n87), .O(n86) );
BUFX1 U87 ( .I(n88), .O(n87) );
BUFX1 U88 ( .I(n89), .O(n88) );
BUFX1 U89 ( .I(n90), .O(n89) );
BUFX1 U90 ( .I(n91), .O(n90) );
BUFX1 U91 ( .I(n92), .O(n91) );
BUFX1 U92 ( .I(n93), .O(n92) );
BUFX1 U93 ( .I(n94), .O(n93) );
BUFX1 U94 ( .I(n95), .O(n94) );
BUFX1 U95 ( .I(n96), .O(n95) );
BUFX1 U96 ( .I(n97), .O(n96) );
BUFX1 U97 ( .I(n98), .O(n97) );
BUFX1 U98 ( .I(n99), .O(n98) );
BUFX1 U99 ( .I(n100), .O(n99) );
BUFX1 U100 ( .I(n101), .O(n100) );
BUFX1 U101 ( .I(n102), .O(n101) );
BUFX1 U102 ( .I(n103), .O(n102) );
BUFX1 U103 ( .I(n104), .O(n103) );
BUFX1 U104 ( .I(n105), .O(n104) );
BUFX1 U105 ( .I(n106), .O(n105) );
BUFX1 U106 ( .I(n107), .O(n106) );
BUFX1 U107 ( .I(n108), .O(n107) );
BUFX1 U108 ( .I(n109), .O(n108) );
BUFX1 U109 ( .I(n110), .O(n109) );
BUFX1 U110 ( .I(n111), .O(n110) );
BUFX1 U111 ( .I(n112), .O(n111) );
BUFX1 U112 ( .I(n113), .O(n112) );
BUFX1 U113 ( .I(n114), .O(n113) );
BUFX1 U114 ( .I(n115), .O(n114) );
BUFX1 U115 ( .I(n116), .O(n115) );
BUFX1 U116 ( .I(n117), .O(n116) );
INVX1 U117 ( .I(in), .O(n117) );
endmodule
module DelayElement_73 ( in, out );
input in;
output out;
wire t1;
delayele_inv_7 d1 ( .in(in), .out(t1) );
delayele_inv_6 d2 ( .in(t1), .out(out) );
endmodule
module delayele_inv_0 ( in, out );
```

```
input in;
output out;
wire n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14, n15, n16,
n17, n18, n19, n20, n21, n22, n23, n24, n25, n26, n27, n28, n29, n30,
n31, n32, n33, n34, n35, n36, n37, n38, n39, n40, n41, n42, n43, n44,
n45, n46, n47, n48, n49, n50, n51, n52, n53, n54, n55, n56, n57, n58,
n59, n60, n61, n62, n63, n64, n65, n66, n67, n68, n69, n70, n71, n72,
n73, n74, n75, n76, n77, n78, n79, n80, n81, n82, n83, n84, n85, n86,
n87, n88, n89, n90, n91, n92, n93, n94, n95, n96, n97, n98, n99, n100,
n101, n102, n103, n104, n105, n106, n107, n108, n109, n110, n111,
n112, n113, n114, n115, n116, n117;
BUFX1 U1 ( .I(n2), .O(out) );
BUFX1 U2 ( .I(n3), .O(n2) );
BUFX1 U3 ( .I(n4), .O(n3) );
BUFX1 U4 ( .I(n5), .O(n4) );
BUFX1 U5 ( .I(n6), .O(n5) );
BUFX1 U6 ( .I(n7), .O(n6) );
BUFX1 U7 ( .I(n8), .O(n7) );
BUFX1 U8 ( .I(n9), .O(n8) );
BUFX1 U9 ( .I(n10), .O(n9) );
BUFX1 U10 ( .I(n11), .O(n10) );
BUFX1 U11 ( .I(n12), .O(n11) );
BUFX1 U12 ( .I(n13), .O(n12) );
BUFX1 U13 ( .I(n14), .O(n13) );
BUFX1 U14 ( .I(n15), .O(n14) );
BUFX1 U15 ( .I(n16), .O(n15) );
BUFX1 U16 ( .I(n17), .O(n16) );
BUFX1 U17 ( .I(n18), .O(n17) );
BUFX1 U18 ( .I(n19), .O(n18) );
BUFX1 U19 ( .I(n20), .O(n19) );
BUFX1 U20 ( .I(n21), .O(n20) );
BUFX1 U21 ( .I(n22), .O(n21) );
BUFX1 U22 ( .I(n23), .O(n22) );
BUFX1 U23 ( .I(n24), .O(n23) );
BUFX1 U24 ( .I(n25), .O(n24) );
BUFX1 U25 ( .I(n26), .O(n25) );
BUFX1 U26 ( .I(n27), .O(n26) );
BUFX1 U27 ( .I(n28), .O(n27) );
BUFX1 U28 ( .I(n29), .O(n28) );
BUFX1 U29 ( .I(n30), .O(n29) );
BUFX1 U30 ( .I(n31), .O(n30) );
BUFX1 U31 ( .I(n32), .O(n31) );
BUFX1 U32 ( .I(n33), .O(n32) );
BUFX1 U33 ( .I(n34), .O(n33) );
BUFX1 U34 ( .I(n35), .O(n34) );
BUFX1 U35 ( .I(n36), .O(n35) );
```

```
BUFX1 U36 ( .I(n37), .O(n36) );
BUFX1 U37 ( .I(n38), .O(n37) );
BUFX1 U38 ( .I(n39), .O(n38) );
BUFX1 U39 ( .I(n40), .O(n39) );
BUFX1 U40 ( .I(n41), .O(n40) );
BUFX1 U41 ( .I(n42), .O(n41) );
BUFX1 U42 ( .I(n43), .O(n42) );
BUFX1 U43 ( .I(n44), .O(n43) );
BUFX1 U44 ( .I(n45), .O(n44) );
BUFX1 U45 ( .I(n46), .O(n45) );
BUFX1 U46 ( .I(n47), .O(n46) );
BUFX1 U47 ( .I(n48), .O(n47) );
BUFX1 U48 ( .I(n49), .O(n48) );
BUFX1 U49 ( .I(n50), .O(n49) );
BUFX1 U50 ( .I(n51), .O(n50) );
BUFX1 U51 ( .I(n52), .O(n51) );
BUFX1 U52 ( .I(n53), .O(n52) );
BUFX1 U53 ( .I(n54), .O(n53) );
BUFX1 U54 ( .I(n55), .O(n54) );
BUFX1 U55 ( .I(n56), .O(n55) );
BUFX1 U56 ( .I(n57), .O(n56) );
BUFX1 U57 ( .I(n58), .O(n57) );
BUFX1 U58 ( .I(n59), .O(n58) );
BUFX1 U59 ( .I(n60), .O(n59) );
BUFX1 U60 ( .I(n61), .O(n60) );
BUFX1 U61 ( .I(n62), .O(n61) );
BUFX1 U62 ( .I(n63), .O(n62) );
BUFX1 U63 ( .I(n64), .O(n63) );
BUFX1 U64 ( .I(n65), .O(n64) );
BUFX1 U65 ( .I(n66), .O(n65) );
BUFX1 U66 ( .I(n67), .O(n66) );
BUFX1 U67 ( .I(n68), .O(n67) );
BUFX1 U68 ( .I(n69), .O(n68) );
BUFX1 U69 ( .I(n70), .O(n69) );
BUFX1 U70 ( .I(n71), .O(n70) );
BUFX1 U71 ( .I(n72), .O(n71) );
BUFX1 U72 ( .I(n73), .O(n72) );
BUFX1 U73 ( .I(n74), .O(n73) );
BUFX1 U74 ( .I(n75), .O(n74) );
BUFX1 U75 ( .I(n76), .O(n75) );
BUFX1 U76 ( .I(n77), .O(n76) );
BUFX1 U77 ( .I(n78), .O(n77) );
BUFX1 U78 ( .I(n79), .O(n78) );
BUFX1 U79 ( .I(n80), .O(n79) );
BUFX1 U80 ( .I(n81), .O(n80) );
BUFX1 U81 ( .I(n82), .O(n81) );
```

```
BUFX1 U82 ( .I(n83), .O(n82) );
BUFX1 U83 ( .I(n84), .O(n83) );
BUFX1 U84 ( .I(n85), .O(n84) );
BUFX1 U85 ( .I(n86), .O(n85) );
BUFX1 U86 ( .I(n87), .O(n86) );
BUFX1 U87 ( .I(n88), .O(n87) );
BUFX1 U88 ( .I(n89), .O(n88) );
BUFX1 U89 ( .I(n90), .O(n89) );
BUFX1 U90 ( .I(n91), .O(n90) );
BUFX1 U91 ( .I(n92), .O(n91) );
BUFX1 U92 ( .I(n93), .O(n92) );
BUFX1 U93 ( .I(n94), .O(n93) );
BUFX1 U94 ( .I(n95), .O(n94) );
BUFX1 U95 ( .I(n96), .O(n95) );
BUFX1 U96 ( .I(n97), .O(n96) );
BUFX1 U97 ( .I(n98), .O(n97) );
BUFX1 U98 ( .I(n99), .O(n98) );
BUFX1 U99 ( .I(n100), .O(n99) );
BUFX1 U100 ( .I(n101), .O(n100) );
BUFX1 U101 ( .I(n102), .O(n101) );
BUFX1 U102 ( .I(n103), .O(n102) );
BUFX1 U103 ( .I(n104), .O(n103) );
BUFX1 U104 ( .I(n105), .O(n104) );
BUFX1 U105 ( .I(n106), .O(n105) );
BUFX1 U106 ( .I(n107), .O(n106) );
BUFX1 U107 ( .I(n108), .O(n107) );
BUFX1 U108 ( .I(n109), .O(n108) );
BUFX1 U109 ( .I(n110), .O(n109) );
BUFX1 U110 ( .I(n111), .O(n110) );
BUFX1 U111 ( .I(n112), .O(n111) );
BUFX1 U112 ( .I(n113), .O(n112) );
BUFX1 U113 ( .I(n114), .O(n113) );
BUFX1 U114 ( .I(n115), .O(n114) );
BUFX1 U115 ( .I(n116), .O(n115) );
BUFX1 U116 ( .I(n117), .O(n116) );
INVX1 U117 ( .I(in), .O(n117) );
endmodule
module delayele_inv_1 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_70 ( in, out );
input in;
output out;
wire t1;
```

```
delayele_inv_1 d1 ( .in(in), .out(t1) );
delayele_inv_0 d2 ( .in(t1), .out(out) );
endmodule
module delayele_inv_2 ( in, out );
input in;
output out;
wire n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14, n15, n16,
n17, n18, n19, n20, n21, n22, n23, n24, n25, n26, n27, n28, n29, n30,
n31, n32, n33, n34, n35, n36, n37, n38, n39, n40, n41, n42, n43, n44,
n45, n46, n47, n48, n49, n50, n51, n52, n53, n54, n55, n56, n57, n58,
n59, n60, n61, n62, n63, n64, n65, n66, n67, n68, n69, n70, n71, n72,
n73, n74, n75, n76, n77, n78, n79, n80, n81, n82, n83, n84, n85, n86,
n87, n88, n89, n90, n91, n92, n93, n94, n95, n96, n97, n98, n99, n100,
n101, n102, n103, n104, n105, n106, n107, n108, n109, n110, n111,
n112, n113, n114, n115, n116, n117;
BUFX1 U1 ( .I(n2), .O(out) );
BUFX1 U2 ( .I(n3), .O(n2) );
BUFX1 U3 ( .I(n4), .O(n3) );
BUFX1 U4 ( .I(n5), .O(n4) );
BUFX1 U5 ( .I(n6), .O(n5) );
BUFX1 U6 ( .I(n7), .O(n6) );
BUFX1 U7 ( .I(n8), .O(n7) );
BUFX1 U8 ( .I(n9), .O(n8) );
BUFX1 U9 ( .I(n10), .O(n9) );
BUFX1 U10 ( .I(n11), .O(n10) );
BUFX1 U11 ( .I(n12), .O(n11) );
BUFX1 U12 ( .I(n13), .O(n12) );
BUFX1 U13 ( .I(n14), .O(n13) );
BUFX1 U14 ( .I(n15), .O(n14) );
BUFX1 U15 ( .I(n16), .O(n15) );
BUFX1 U16 ( .I(n17), .O(n16) );
BUFX1 U17 ( .I(n18), .O(n17) );
BUFX1 U18 ( .I(n19), .O(n18) );
BUFX1 U19 ( .I(n20), .O(n19) );
BUFX1 U20 ( .I(n21), .O(n20) );
BUFX1 U21 ( .I(n22), .O(n21) );
BUFX1 U22 ( .I(n23), .O(n22) );
BUFX1 U23 ( .I(n24), .O(n23) );
BUFX1 U24 ( .I(n25), .O(n24) );
BUFX1 U25 ( .I(n26), .O(n25) );
BUFX1 U26 ( .I(n27), .O(n26) );
BUFX1 U27 ( .I(n28), .O(n27) );
BUFX1 U28 ( .I(n29), .O(n28) );
BUFX1 U29 ( .I(n30), .O(n29) );
BUFX1 U30 ( .I(n31), .O(n30) );
BUFX1 U31 ( .I(n32), .O(n31) );
```

```
BUFX1 U32 ( .I(n33), .O(n32) );
BUFX1 U33 ( .I(n34), .O(n33) );
BUFX1 U34 ( .I(n35), .O(n34) );
BUFX1 U35 ( .I(n36), .O(n35) );
BUFX1 U36 ( .I(n37), .O(n36) );
BUFX1 U37 ( .I(n38), .O(n37) );
BUFX1 U38 ( .I(n39), .O(n38) );
BUFX1 U39 ( .I(n40), .O(n39) );
BUFX1 U40 ( .I(n41), .O(n40) );
BUFX1 U41 ( .I(n42), .O(n41) );
BUFX1 U42 ( .I(n43), .O(n42) );
BUFX1 U43 ( .I(n44), .O(n43) );
BUFX1 U44 ( .I(n45), .O(n44) );
BUFX1 U45 ( .I(n46), .O(n45) );
BUFX1 U46 ( .I(n47), .O(n46) );
BUFX1 U47 ( .I(n48), .O(n47) );
BUFX1 U48 ( .I(n49), .O(n48) );
BUFX1 U49 ( .I(n50), .O(n49) );
BUFX1 U50 ( .I(n51), .O(n50) );
BUFX1 U51 ( .I(n52), .O(n51) );
BUFX1 U52 ( .I(n53), .O(n52) );
BUFX1 U53 ( .I(n54), .O(n53) );
BUFX1 U54 ( .I(n55), .O(n54) );
BUFX1 U55 ( .I(n56), .O(n55) );
BUFX1 U56 ( .I(n57), .O(n56) );
BUFX1 U57 ( .I(n58), .O(n57) );
BUFX1 U58 ( .I(n59), .O(n58) );
BUFX1 U59 ( .I(n60), .O(n59) );
BUFX1 U60 ( .I(n61), .O(n60) );
BUFX1 U61 ( .I(n62), .O(n61) );
BUFX1 U62 ( .I(n63), .O(n62) );
BUFX1 U63 ( .I(n64), .O(n63) );
BUFX1 U64 ( .I(n65), .O(n64) );
BUFX1 U65 ( .I(n66), .O(n65) );
BUFX1 U66 ( .I(n67), .O(n66) );
BUFX1 U67 ( .I(n68), .O(n67) );
BUFX1 U68 ( .I(n69), .O(n68) );
BUFX1 U69 ( .I(n70), .O(n69) );
BUFX1 U70 ( .I(n71), .O(n70) );
BUFX1 U71 ( .I(n72), .O(n71) );
BUFX1 U72 ( .I(n73), .O(n72) );
BUFX1 U73 ( .I(n74), .O(n73) );
BUFX1 U74 ( .I(n75), .O(n74) );
BUFX1 U75 ( .I(n76), .O(n75) );
BUFX1 U76 ( .I(n77), .O(n76) );
BUFX1 U77 ( .I(n78), .O(n77) );
```

```
BUFX1 U78 ( .I(n79), .O(n78) );
BUFX1 U79 ( .I(n80), .O(n79) );
BUFX1 U80 ( .I(n81), .O(n80) );
BUFX1 U81 ( .I(n82), .O(n81) );
BUFX1 U82 ( .I(n83), .O(n82) );
BUFX1 U83 ( .I(n84), .O(n83) );
BUFX1 U84 ( .I(n85), .O(n84) );
BUFX1 U85 ( .I(n86), .O(n85) );
BUFX1 U86 ( .I(n87), .O(n86) );
BUFX1 U87 ( .I(n88), .O(n87) );
BUFX1 U88 ( .I(n89), .O(n88) );
BUFX1 U89 ( .I(n90), .O(n89) );
BUFX1 U90 ( .I(n91), .O(n90) );
BUFX1 U91 ( .I(n92), .O(n91) );
BUFX1 U92 ( .I(n93), .O(n92) );
BUFX1 U93 ( .I(n94), .O(n93) );
BUFX1 U94 ( .I(n95), .O(n94) );
BUFX1 U95 ( .I(n96), .O(n95) );
BUFX1 U96 ( .I(n97), .O(n96) );
BUFX1 U97 ( .I(n98), .O(n97) );
BUFX1 U98 ( .I(n99), .O(n98) );
BUFX1 U99 ( .I(n100), .O(n99) );
BUFX1 U100 ( .I(n101), .O(n100) );
BUFX1 U101 ( .I(n102), .O(n101) );
BUFX1 U102 ( .I(n103), .O(n102) );
BUFX1 U103 ( .I(n104), .O(n103) );
BUFX1 U104 ( .I(n105), .O(n104) );
BUFX1 U105 ( .I(n106), .O(n105) );
BUFX1 U106 ( .I(n107), .O(n106) );
BUFX1 U107 ( .I(n108), .O(n107) );
BUFX1 U108 ( .I(n109), .O(n108) );
BUFX1 U109 ( .I(n110), .O(n109) );
BUFX1 U110 ( .I(n111), .O(n110) );
BUFX1 U111 ( .I(n112), .O(n111) );
BUFX1 U112 ( .I(n113), .O(n112) );
BUFX1 U113 ( .I(n114), .O(n113) );
BUFX1 U114 ( .I(n115), .O(n114) );
BUFX1 U115 ( .I(n116), .O(n115) );
BUFX1 U116 ( .I(n117), .O(n116) );
INVX1 U117 ( .I(in), .O(n117) );
endmodule
module delayele_inv_3 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
```

```
module DelayElement_71 ( in, out );
input in;
output out;
wire t1;
delayele_inv_3 d1 ( .in(in), .out(t1) );
delayele_inv_2 d2 ( .in(t1), .out(out) );
endmodule
module delayele_inv_4 ( in, out );
input in;
output out;
wire n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14, n15, n16,
n17, n18, n19, n20, n21, n22, n23, n24, n25, n26, n27, n28, n29, n30,
n31, n32, n33, n34, n35, n36, n37, n38, n39, n40, n41, n42, n43, n44,
n45, n46, n47, n48, n49, n50, n51, n52, n53, n54, n55, n56, n57, n58,
n59, n60, n61, n62, n63, n64, n65, n66, n67, n68, n69, n70, n71, n72,
n73, n74, n75, n76, n77, n78, n79, n80, n81, n82, n83, n84, n85, n86,
n87, n88, n89, n90, n91, n92, n93, n94, n95, n96, n97, n98, n99, n100,
n101, n102, n103, n104, n105, n106, n107, n108, n109, n110, n111,
n112, n113, n114, n115, n116, n117;
BUFX1 U1 ( .I(n2), .O(out) );
BUFX1 U2 ( .I(n3), .O(n2) );
BUFX1 U3 ( .I(n4), .O(n3) );
BUFX1 U4 ( .I(n5), .O(n4) );
BUFX1 U5 ( .I(n6), .O(n5) );
BUFX1 U6 ( .I(n7), .O(n6) );
BUFX1 U7 ( .I(n8), .O(n7) );
BUFX1 U8 ( .I(n9), .O(n8) );
BUFX1 U9 ( .I(n10), .O(n9) );
BUFX1 U10 ( .I(n11), .O(n10) );
BUFX1 U11 ( .I(n12), .O(n11) );
BUFX1 U12 ( .I(n13), .O(n12) );
BUFX1 U13 ( .I(n14), .O(n13) );
BUFX1 U14 ( .I(n15), .O(n14) );
BUFX1 U15 ( .I(n16), .O(n15) );
BUFX1 U16 ( .I(n17), .O(n16) );
BUFX1 U17 ( .I(n18), .O(n17) );
BUFX1 U18 ( .I(n19), .O(n18) );
BUFX1 U19 ( .I(n20), .O(n19) );
BUFX1 U20 ( .I(n21), .O(n20) );
BUFX1 U21 ( .I(n22), .O(n21) );
BUFX1 U22 ( .I(n23), .O(n22) );
BUFX1 U23 ( .I(n24), .O(n23) );
BUFX1 U24 ( .I(n25), .O(n24) );
BUFX1 U25 ( .I(n26), .O(n25) );
BUFX1 U26 ( .I(n27), .O(n26) );
BUFX1 U27 ( .I(n28), .O(n27) );
```

```
BUFX1 U28 ( .I(n29), .O(n28) );
BUFX1 U29 ( .I(n30), .O(n29) );
BUFX1 U30 ( .I(n31), .O(n30) );
BUFX1 U31 ( .I(n32), .O(n31) );
BUFX1 U32 ( .I(n33), .O(n32) );
BUFX1 U33 ( .I(n34), .O(n33) );
BUFX1 U34 ( .I(n35), .O(n34) );
BUFX1 U35 ( .I(n36), .O(n35) );
BUFX1 U36 ( .I(n37), .O(n36) );
BUFX1 U37 ( .I(n38), .O(n37) );
BUFX1 U38 ( .I(n39), .O(n38) );
BUFX1 U39 ( .I(n40), .O(n39) );
BUFX1 U40 ( .I(n41), .O(n40) );
BUFX1 U41 ( .I(n42), .O(n41) );
BUFX1 U42 ( .I(n43), .O(n42) );
BUFX1 U43 ( .I(n44), .O(n43) );
BUFX1 U44 ( .I(n45), .O(n44) );
BUFX1 U45 ( .I(n46), .O(n45) );
BUFX1 U46 ( .I(n47), .O(n46) );
BUFX1 U47 ( .I(n48), .O(n47) );
BUFX1 U48 ( .I(n49), .O(n48) );
BUFX1 U49 ( .I(n50), .O(n49) );
BUFX1 U50 ( .I(n51), .O(n50) );
BUFX1 U51 ( .I(n52), .O(n51) );
BUFX1 U52 ( .I(n53), .O(n52) );
BUFX1 U53 ( .I(n54), .O(n53) );
BUFX1 U54 ( .I(n55), .O(n54) );
BUFX1 U55 ( .I(n56), .O(n55) );
BUFX1 U56 ( .I(n57), .O(n56) );
BUFX1 U57 ( .I(n58), .O(n57) );
BUFX1 U58 ( .I(n59), .O(n58) );
BUFX1 U59 ( .I(n60), .O(n59) );
BUFX1 U60 ( .I(n61), .O(n60) );
BUFX1 U61 ( .I(n62), .O(n61) );
BUFX1 U62 ( .I(n63), .O(n62) );
BUFX1 U63 ( .I(n64), .O(n63) );
BUFX1 U64 ( .I(n65), .O(n64) );
BUFX1 U65 ( .I(n66), .O(n65) );
BUFX1 U66 ( .I(n67), .O(n66) );
BUFX1 U67 ( .I(n68), .O(n67) );
BUFX1 U68 ( .I(n69), .O(n68) );
BUFX1 U69 ( .I(n70), .O(n69) );
BUFX1 U70 ( .I(n71), .O(n70) );
BUFX1 U71 ( .I(n72), .O(n71) );
BUFX1 U72 ( .I(n73), .O(n72) );
BUFX1 U73 ( .I(n74), .O(n73) );
```

```
BUFX1 U74 ( .I(n75), .O(n74) );
BUFX1 U75 ( .I(n76), .O(n75) );
BUFX1 U76 ( .I(n77), .O(n76) );
BUFX1 U77 ( .I(n78), .O(n77) );
BUFX1 U78 ( .I(n79), .O(n78) );
BUFX1 U79 ( .I(n80), .O(n79) );
BUFX1 U80 ( .I(n81), .O(n80) );
BUFX1 U81 ( .I(n82), .O(n81) );
BUFX1 U82 ( .I(n83), .O(n82) );
BUFX1 U83 ( .I(n84), .O(n83) );
BUFX1 U84 ( .I(n85), .O(n84) );
BUFX1 U85 ( .I(n86), .O(n85) );
BUFX1 U86 ( .I(n87), .O(n86) );
BUFX1 U87 ( .I(n88), .O(n87) );
BUFX1 U88 ( .I(n89), .O(n88) );
BUFX1 U89 ( .I(n90), .O(n89) );
BUFX1 U90 ( .I(n91), .O(n90) );
BUFX1 U91 ( .I(n92), .O(n91) );
BUFX1 U92 ( .I(n93), .O(n92) );
BUFX1 U93 ( .I(n94), .O(n93) );
BUFX1 U94 ( .I(n95), .O(n94) );
BUFX1 U95 ( .I(n96), .O(n95) );
BUFX1 U96 ( .I(n97), .O(n96) );
BUFX1 U97 ( .I(n98), .O(n97) );
BUFX1 U98 ( .I(n99), .O(n98) );
BUFX1 U99 ( .I(n100), .O(n99) );
BUFX1 U100 ( .I(n101), .O(n100) );
BUFX1 U101 ( .I(n102), .O(n101) );
BUFX1 U102 ( .I(n103), .O(n102) );
BUFX1 U103 ( .I(n104), .O(n103) );
BUFX1 U104 ( .I(n105), .O(n104) );
BUFX1 U105 ( .I(n106), .O(n105) );
BUFX1 U106 ( .I(n107), .O(n106) );
BUFX1 U107 ( .I(n108), .O(n107) );
BUFX1 U108 ( .I(n109), .O(n108) );
BUFX1 U109 ( .I(n110), .O(n109) );
BUFX1 U110 ( .I(n111), .O(n110) );
BUFX1 U111 ( .I(n112), .O(n111) );
BUFX1 U112 ( .I(n113), .O(n112) );
BUFX1 U113 ( .I(n114), .O(n113) );
BUFX1 U114 ( .I(n115), .O(n114) );
BUFX1 U115 ( .I(n116), .O(n115) );
BUFX1 U116 ( .I(n117), .O(n116) );
INVX1 U117 ( .I(in), .O(n117) );
endmodule
module delayele_inv_5 ( in, out );
```

```
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_72 ( in, out );
input in;
output out;
wire t1;
delayele_inv_5 d1 ( .in(in), .out(t1) );
delayele_inv_4 d2 ( .in(t1), .out(out) );
endmodule
module FFT16_control_async(
input wire reset,
input wire FFT4_1_rout,
input wire FFT4_2_rout,
input wire FFT4_3_rout,
input wire FFT4_4_rout,
input wire M1_aout,
input wire M2_aout,
input wire M3_aout,
input wire M4_aout,
output wire FFT4_1_aout,
output wire FFT4_2_aout,
output wire FFT4_3_aout,
output wire FFT4_4_aout,
output wire c_M1,
output wire c_M2,
output wire c_M3,
output wire c_M4,
output wire M1_rout,
output wire M2_rout,
output wire M3_rout,
output wire M4_rout
);
wire FFT4_1_rout_delay, FFT4_2_rout_delay, FFT4_3_rout_delay, FFT4_4_rout_delay;
//Multiplier delay element should include Clock to Q delay as well
//Delay element named changed here as it clashed with FFT4 control
DelayElement_73 d1( .in(FFT4_1_rout), .out(FFT4_1_rout_delay) );
DelayElement_72 d2( .in(FFT4_2_rout), .out(FFT4_2_rout_delay) );
DelayElement_71 d3( .in(FFT4_3_rout), .out(FFT4_3_rout_delay) );
DelayElement_70 d4( .in(FFT4_4_rout), .out(FFT4_4_rout_delay) );
handshake_ctl HC1(.lr(FFT4_1_rout_delay),.la(FFT4_1_aout),.rr(M1_rout),.ra(M1_aout),.ck(c_M1),.reset(reset));
handshake_ctl HC2(.lr(FFT4_2_rout_delay),.la(FFT4_2_aout),.rr(M2_rout),.ra(M2_aout),.ck(c_M2),.reset(reset));
handshake_ctl HC3(.lr(FFT4_3_rout_delay),.la(FFT4_3_aout),.rr(M3_rout),.ra(M3_aout),.ck(c_M3),.reset(reset));
handshake_ctl HC4(.lr(FFT4_4_rout_delay),.la(FFT4_4_aout),.rr(M4_rout),.ra(M4_aout),.ck(c_M4),.reset(reset));
endmodule
```

4-point FFT Datapath Verilog Code

```
module FFT4_data path_async(
    input wire reset,
    input wire [31:0] x,
    input wire ci1,
    input wire ci2,
    input wire ci3,
    input wire ci4,
    input wire c1,
    input wire c2,
    input wire c3,
    input wire c4,
    input wire c5,
    input wire c6,
    input wire c7,
    input wire c8,
    input wire c9,
    input wire c10,
    input wire c11,
    input wire c12,
    input wire c13,
    input wire c14,
    input wire c15,
    input wire c16,
    output wire [31:0] X1,
    output wire [31:0] X2,
    output wire [31:0] X3,
    output wire [31:0] X4
    );
    wire [31:0] x1,x2,x3,x4;
    wire [15:0] ra,rb,rc,rd,ia,ib,ic,id;
    wire signed [15:0] reala,realb,realc,reald;
    wire signed [15:0] imaga,imagb,imagc,imagd;
    wire signed [15:0] realx1,realx2,realx3,realx4;
    wire signed [15:0] imagx1,imagx2,imagx3,imagx4;
    wire signed [15:0] ADD1_in1,ADD1_in2,ADD2_in1,ADD2_in2,SUB1_in1;
    wire signed [15:0] SUB1_in2,SUB2_in1,SUB2_in2,ADD1_out,ADD2_out,SUB1_out,SUB2_out;
    wire signed [15:0] ADD3_in1,ADD3_in2,ADD4_in1,ADD4_in2,SUB3_in1;
    wire signed [15:0] SUB3_in2,SUB4_in1,SUB4_in2,ADD3_out,ADD4_out,SUB3_out,SUB4_out;
    wire signed [15:0] ADD5_in1,ADD5_in2,ADD6_in1,ADD6_in2,SUB5_in1;
    wire signed [15:0] SUB5_in2,SUB6_in1,SUB6_in2,ADD5_out,ADD6_out,SUB5_out,SUB6_out;
    wire signed [15:0] ADD7_in1,ADD7_in2,ADD8_in1,ADD8_in2,SUB7_in1;
    wire signed [15:0] SUB7_in2,SUB8_in1,SUB8_in2,ADD7_out,ADD8_out,SUB7_out,SUB8_out;
    reg32_async R1(.D(x),.Q(x1),.ck(ci1),.reset(reset));
    reg32_async R2(.D(x),.Q(x2),.ck(ci2),.reset(reset));
```

```
reg32_async R3(.D(x),.Q(x3),.ck(ci3),.reset(reset));
reg32_async R4(.D(x),.Q(x4),.ck(ci4),.reset(reset));
assign realx1 = $signed(x1[31:16]);
assign imagx1 = $signed(x1[15:0]);
assign realx2 = $signed(x2[31:16]);
assign imagx2 = $signed(x2[15:0]);
assign realx3 = $signed(x3[31:16]);
assign imagx3 = $signed(x3[15:0]);
assign realx4 = $signed(x4[31:16]);
assign imagx4 = $signed(x4[15:0]);
assign ADD1_in1 = realx1;
assign ADD1_in2 = realx3;
assign ADD2_in1 = imagx1;
assign ADD2_in2 = imagx3;
assign SUB1_in1 = realx1;
assign SUB1_in2 = realx3;
assign SUB2_in1 = imagx1;
assign SUB2_in2 = imagx3;
assign ADD3_in1 = realx2;
assign ADD3_in2 = realx4;
assign ADD4_in1 = imagx2;
assign ADD4_in2 = imagx4;
assign SUB3_in1 = realx2;
assign SUB3_in2 = realx4;
assign SUB4_in1 = imagx2;
assign SUB4_in2 = imagx4;
assign ADD1_out = (ADD1_in1 >>> 1) + (ADD1_in2 >>> 1); //reala
assign ADD2_out = (ADD2_in1 >>> 1) + (ADD2_in2 >>> 1); //imaga
assign SUB1_out = (SUB1_in1 >>> 1) - (SUB1_in2 >>> 1); //realc
assign SUB2_out = (SUB2_in1 >>> 1) - (SUB2_in2 >>> 1); //imagc
assign ADD3_out = (ADD3_in1 >>> 1) + (ADD3_in2 >>> 1); //realb
assign ADD4_out = (ADD4_in1 >>> 1) + (ADD4_in2 >>> 1); //imagb
assign SUB3_out = (SUB3_in1 >>> 1) - (SUB3_in2 >>> 1); //reald
assign SUB4_out = (SUB4_in1 >>> 1) - (SUB4_in2 >>> 1); //imagd
reg16_async R5(.D($unsigned(ADD1_out)),.Q(ra),.ck(c1),.reset(reset));
reg16_async R6(.D($unsigned(ADD2_out)),.Q(ia),.ck(c2),.reset(reset));
reg16_async R7(.D($unsigned(ADD3_out)),.Q(rb),.ck(c3),.reset(reset));
reg16_async R8(.D($unsigned(ADD4_out)),.Q(ib),.ck(c4),.reset(reset));
reg16_async R9(.D($unsigned(SUB1_out)),.Q(rc),.ck(c5),.reset(reset));
reg16_async R10(.D($unsigned(SUB2_out)),.Q(ic),.ck(c6),.reset(reset));
reg16_async R11(.D($unsigned(SUB3_out)),.Q(rd),.ck(c7),.reset(reset));
reg16_async R12(.D($unsigned(SUB4_out)),.Q(id),.ck(c8),.reset(reset));
assign reala = $signed(ra);
assign imaga = $signed(ia);
assign realb = $signed(rb);
assign imagb = $signed(ib);
```

```
    assign realc = $signed(rc);
    assign imagc = $signed(ic);
    assign reald = $signed(rd);
    assign imagd = $signed(id);
    assign ADD5_in1 = reala; //r1 control
    assign ADD5_in2 = realb; //r3 control
    assign ADD6_in1 = imaga; //r2 control
    assign ADD6_in2 = imagb; //r4 control
    assign SUB5_in1 = reala; //r1 control
    assign SUB5_in2 = realb; //r3 control
    assign SUB6_in1 = imaga; //r2 control
    assign SUB6_in2 = imagb; //r4 control
    assign ADD7_in1 = realc; //r5 control
    assign ADD7_in2 = imagd; //r8 control
    assign ADD8_in1 = imagc; //r6 control
    assign ADD8_in2 = reald; //r7 control
    assign SUB7_in1 = imagc; //r6 control
    assign SUB7_in2 = reald; //r7 control
    assign SUB8_in1 = realc; //r5 control
    assign SUB8_in2 = imagd; //r8 control
    assign ADD5_out = (ADD5_in1 >>> 1) + (ADD5_in2 >>> 1); //X1[31:16]
    assign ADD6_out = (ADD6_in1 >>> 1) + (ADD6_in2 >>> 1); //X1[15:0]
    assign SUB5_out = (SUB5_in1 >>> 1) - (SUB5_in2 >>> 1); //X3[31:16]
    assign SUB6_out = (SUB6_in1 >>> 1) - (SUB6_in2 >>> 1); //X3[15:0]
    assign ADD7_out = (ADD7_in1 >>> 1) + (ADD7_in2 >>> 1); //X2[31:16]
    assign ADD8_out = (ADD8_in1 >>> 1) + (ADD8_in2 >>> 1); //X4[15:0]
    assign SUB7_out = (SUB7_in1 >>> 1) - (SUB7_in2 >>> 1); //X2[15:0]
    assign SUB8_out = (SUB8_in1 >>> 1) - (SUB8_in2 >>> 1); //X4[31:16]
    reg16_async R13(.D($unsigned(ADD5_out)),.Q(X1[31:16]),.ck(c9),.reset(reset));
    reg16_async R14(.D($unsigned(ADD6_out)),.Q(X1[15:0]),.ck(c10),.reset(reset));
    reg16_async R15(.D($unsigned(ADD7_out)),.Q(X2[31:16]),.ck(c11),.reset(reset));
    reg16_async R16(.D($unsigned(ADD8_out)),.Q(X4[15:0]),.ck(c12),.reset(reset));
    reg16_async R17(.D($unsigned(SUB5_out)),.Q(X3[31:16]),.ck(c13),.reset(reset));
    reg16_async R18(.D($unsigned(SUB6_out)),.Q(X3[15:0]),.ck(c14),.reset(reset));
    reg16_async R19(.D($unsigned(SUB7_out)),.Q(X2[15:0]),.ck(c15),.reset(reset));
    reg16_async R20(.D($unsigned(SUB8_out)),.Q(X4[31:16]),.ck(c16),.reset(reset));
    endmodule
```

4-point FFT Bundled-Data Control Verilog Code

```
module delayele_adder16_31 ( in, out );
    input in;
    output out;
    INVX1 U1 ( .I(in), .O(out) );
    endmodule
```

```
module delayele_adder16_30 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module DelayElement_15 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_31 d1 ( .in(in), .out(t1) );
delayele_adder16_30 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_0 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_1 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_0 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_1 d1 ( .in(in), .out(t1) );
delayele_adder16_0 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_2 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
```

```
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_3 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_1 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_3 d1 ( .in(in), .out(t1) );
delayele_adder16_2 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_4 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_5 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_2 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_5 d1 ( .in(in), .out(t1) );
delayele_adder16_4 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_6 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
```

```
endmodule
module delayele_adder16_7 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_3 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_7 d1 ( .in(in), .out(t1) );
delayele_adder16_6 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_8 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_9 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_4 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_9 d1 ( .in(in), .out(t1) );
delayele_adder16_8 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_10 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_11 ( in, out );
```

```
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_5 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_11 d1 ( .in(in), .out(t1) );
delayele_adder16_10 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_12 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_13 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_6 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_13 d1 ( .in(in), .out(t1) );
delayele_adder16_12 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_14 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_15 ( in, out );
input in;
output out;
```

```
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_7 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_15 d1 ( .in(in), .out(t1) );
delayele_adder16_14 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_16 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_17 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_8 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_17 d1 ( .in(in), .out(t1) );
delayele_adder16_16 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_18 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_19 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
```

```
module DelayElement_9 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_19 d1 ( .in(in), .out(t1) );
delayele_adder16_18 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_20 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_21 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_10 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_21 d1 ( .in(in), .out(t1) );
delayele_adder16_20 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_22 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_23 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_11 ( in, out );
input in;
```

```
output out;
wire t1;
delayele_adder16_23 d1 ( .in(in), .out(t1) );
delayele_adder16_22 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_24 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_25 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_12 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_25 d1 ( .in(in), .out(t1) );
delayele_adder16_24 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_26 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_27 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_13 ( in, out );
input in;
output out;
wire t1;
```

```
delayele_adder16_27 d1 ( .in(in), .out(t1) );
delayele_adder16_26 d2 ( .in(t1), .out(out) );
endmodule
module delayele_adder16_28 ( in, out );
input in;
output out;
wire n2, n3, n4, n5;
DELAX3 U1 ( .I(n5), .O(n4) );
INVCKXLP U2 ( .I(in), .O(n5) );
BUFX1 U3 ( .I(n2), .O(out) );
BUFX1 U4 ( .I(n3), .O(n2) );
BUFX1 U5 ( .I(n4), .O(n3) );
endmodule
module delayele_adder16_29 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_14 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_29 d1 ( .in(in), .out(t1) );
delayele_adder16_28 d2 ( .in(t1), .out(out) );
endmodule
module FFT4_control_async(
input wire reset,
input wire rin1,
input wire rin2,
input wire rin3,
input wire rin4,
input wire ao1,
input wire ao2,
input wire ao3,
input wire ao4,
output wire ro1,
output wire ro2,
output wire ro3,
output wire ro4,
output wire ain1,
output wire ain2,
output wire ain3,
output wire ain4,
output wire ci1,
output wire ci2,
output wire ci3,
output wire ci4,
output wire c1,
output wire c2,
output wire c3,
output wire c4,
output wire c5,
output wire c6,
output wire c7,
output wire c8,
output wire c9,
output wire c10,
output wire c11,
output wire c12,
output wire c13,
output wire c14,
output wire c15,
output wire c16
);
wire ri_pre,ri1,ri2,ri3,ri4,ai1,ai2,ai3,ai4;
wire reqa1_1,reqa1_2,reqa2_1,reqa2_2,reqa3_1,reqa3_2,reqa4_1,reqa4_2,reqa5_1,re
qa5_2,reqa6_1,reqa6_2;
wire reqa7_1,reqa7_2,reqa8_1,reqa8_2;
wire reqb1_1,reqb1_2,reqb2_1,reqb2_2,reqb3_1,reqb3_2,reqb4_1,reqb4_2,reqb5_1,re
qb5_2,reqb6_1,reqb6_2;
wire reqb7_1,reqb7_2,reqb8_1,reqb8_2;
wire acka1_1,acka1_2,acka2_1,acka2_2,acka3_1,acka3_2,acka4_1,acka4_2;
wire ackb1_1,ackb1_2,ackb2_1,ackb2_2,ackb3_1,ackb3_2,ackb4_1,ackb4_2;
wire ackre1,ackre2,ackre3,ackre4,ackim1,ackim2,ackim3,ackim4;
wire acka5_1,acka5_2,acka6_1,acka6_2,acka7_1,acka7_2,acka8_1,acka8_2;
wire ackb5_1,ackb5_2,ackb6_1,ackb6_2,ackb7_1,ackb7_2,ackb8_1,ackb8_2;
wire req1,req2,req3,req4,req5,req6,req7,req8;
wire req1_pre,req2_pre,req3_pre,req4_pre,req5_pre,req6_pre,req7_pre,req8_pre;
wire ack1,ack2,ack3,ack4,ack5,ack6,ack7,ack8;
wire r1,r2,r3,r4,r5,r6,r7,r8;
wire a1,a2,a3,a4,a5,a6,a7,a8;
wire req9,req10,req11,req12,req13,req14,req15,req16;
wire req9_pre,req10_pre,req11_pre,req12_pre,req13_pre,req14_pre,req15_pre,req16
_pre;
wire ack9,ack10,ack11,ack12,ack13,ack14,ack15,ack16;
wire r9,r10,r11,r12,r13,r14,r15,r16;
wire a9,a10,a11,a12,a13,a14,a15,a16;
wire n1, n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13;
wire n14, n15, n16, n17, n18, n19, n20, n21, n22, n23, n24, n25, n26, n27;
wire n28, n29, n30, n31, n32n33, n34, n35, n36, n37, n38, n39, n40, n41;
wire n42, n43, n44, n45, n46, n47, n48, n49, n50, n51, n52, n53, n54, n55;
wire n56, n57, n58, n59, n60, n61, n62, n63, n64, n65, n66, n67, n68, n69;
wire n70, n71, n72, n73, n74, n75, n76, n77, n78, n79, n80, n81, n82, n83;
wire n84, n85, n86, n87, n88, n89, n90, n91, n92, n93, n94, n95, n96;
handshake_ctl HC1(.lr(rin1),.la(ain1),.rr(ri1),.ra(ai1),.ck(ci1),.reset(reset))
;
handshake_ctl HC2(.lr(rin2),.la(ain2),.rr(ri2),.ra(ai2),.ck(ci2),.reset(reset))
;
handshake_ctl HC3(.lr(rin3),.la(ain3),.rr(ri3),.ra(ai3),.ck(ci3),.reset(reset))
;
handshake_ctl HC4(.lr(rin4),.la(ain4),.rr(ri4),.ra(ai4),.ck(ci4),.reset(reset))
;
//Control Stage 1 (Fork/Join)
//Fork
assign reqa1_1 = ri1;
assign reqa2_1 = ri1;
assign reqb1_1 = ri1;
assign reqb2_1 = ri1;
assign reqa3_1 = ri2;
assign reqa4_1 = ri2;
assign reqb3_1 = ri2;
assign reqb4_1 = ri2;
assign reqa1_2 = ri3;
assign reqa2_2 = ri3;
assign reqb1_2 = ri3;
assign reqb2_2 = ri3;
assign reqa3_2 = ri4;
assign reqa4_2 = ri4;
assign reqb3_2 = ri4;
assign reqb4_2 = ri4;
//Join
//Substage 1
Celement j1(.in1(ackre1),.in2(ackim1),.out(ai1));
Celement j2(.in1(ackre2),.in2(ackim2),.out(ai2));
Celement j3(.in1(ackre3),.in2(ackim3),.out(ai3));
Celement j4(.in1(ackre4),.in2(ackim4),.out(ai4));
//Substage 2
Celement j5(.in1(acka1_1),.in2(ackb1_1),.out(ackre1));
Celement j6(.in1(acka2_1),.in2(ackb2_1),.out(ackim1));
Celement j7(.in1(acka3_1),.in2(ackb3_1),.out(ackre2));
Celement j8(.in1(acka4_1),.in2(ackb4_1),.out(ackim2));
Celement j9(.in1(acka1_2),.in2(ackb1_2),.out(ackre3));
Celement j10(.in1(acka2_2),.in2(ackb2_2),.out(ackim3));
Celement j11(.in1(acka3_2),.in2(ackb3_2),.out(ackre4));
Celement j12(.in1(acka4_2),.in2(ackb4_2),.out(ackim4));
//Control Stage 2 (Fork/Join)
//Join
Celement j13(.in1(reqa1_1),.in2(reqa1_2),.out(req1_pre)); //reala
Celement j14(.in1(reqa2_1),.in2(reqa2_2),.out(req2_pre)); //imaga
Celement j15(.in1(reqa3_1),.in2(reqa3_2),.out(req3_pre)); //realb
Celement j16(.in1(reqa4_1),.in2(reqa4_2),.out(req4_pre)); //imagb
Celement j17(.in1(reqb1_1),.in2(reqb1_2),.out(req5_pre)); //realc
Celement j18(.in1(reqb2_1),.in2(reqb2_2),.out(req6_pre)); //imagc
Celement j19(.in1(reqb3_1),.in2(reqb3_2),.out(req7_pre)); //reald
Celement j20(.in1(reqb4_1),.in2(reqb4_2),.out(req8_pre)); //imagd
//delay elements for 16 bit adder pipeline
DelayElement_15 d1 ( .in(n6), .out(req1) );
DelayElement_14 d2 ( .in(n12), .out(req2) );
DelayElement_13 d3 ( .in(n30), .out(req3) );
DelayElement_12 d4 ( .in(n36), .out(req4) );
DelayElement_11 d5 ( .in(n18), .out(req5) );
DelayElement_10 d6 ( .in(n24), .out(req6) );
DelayElement_9 d7 ( .in(n42), .out(req7) );
DelayElement_8 d8 ( .in(n48), .out(req8) );
//Fork
assign acka1_1 = ack1;
assign acka1_2 = ack1;
assign acka2_1 = ack2;
assign acka2_2 = ack2;
assign acka3_1 = ack3;
assign acka3_2 = ack3;
assign acka4_1 = ack4;
assign acka4_2 = ack4;
assign ackb1_1 = ack5;
assign ackb1_2 = ack5;
assign ackb2_1 = ack6;
assign ackb2_2 = ack6;
assign ackb3_1 = ack7;
assign ackb3_2 = ack7;
assign ackb4_1 = ack8;
assign ackb4_2 = ack8;
handshake_ctl HC5(.lr(req1),.la(ack1),.rr(r1),.ra(a1),.ck(c1),.reset(reset));
handshake_ctl HC6(.lr(req2),.la(ack2),.rr(r2),.ra(a2),.ck(c2),.reset(reset));
handshake_ctl HC7(.lr(req3),.la(ack3),.rr(r3),.ra(a3),.ck(c3),.reset(reset));
handshake_ctl HC8(.lr(req4),.la(ack4),.rr(r4),.ra(a4),.ck(c4),.reset(reset));
handshake_ctl HC9(.lr(req5),.la(ack5),.rr(r5),.ra(a5),.ck(c5),.reset(reset));
handshake_ctl HC10(.lr(req6),.la(ack6),.rr(r6),.ra(a6),.ck(c6),.reset(reset));
handshake_ctl HC11(.lr(req7),.la(ack7),.rr(r7),.ra(a7),.ck(c7),.reset(reset));
handshake_ctl HC12(.lr(req8),.la(ack8),.rr(r8),.ra(a8),.ck(c8),.reset(reset));
//Control Stage 3 (Fork/Join)
//Fork
assign reqa5_1 = r1;
assign reqb5_1 = r1;
assign reqa6_1 = r2;
assign reqb6_1 = r2;
assign reqa5_2 = r3;
assign reqb5_2 = r3;
assign reqa6_2 = r4;
assign reqb6_2 = r4;
assign reqa7_1 = r5;
assign reqb8_1 = r5;
assign reqa8_1 = r6;
assign reqb7_1 = r6;
assign reqa8_2 = r7;
assign reqb7_2 = r7;
assign reqa7_2 = r8;
assign reqb8_2 = r8;
//Join
Celement j21(.in1(acka5_1),.in2(ackb5_1),.out(a1));
Celement j22(.in1(acka6_1),.in2(ackb6_1),.out(a2));
Celement j23(.in1(acka5_2),.in2(ackb5_2),.out(a3));
Celement j24(.in1(acka6_2),.in2(ackb6_2),.out(a4));
Celement j25(.in1(acka7_1),.in2(ackb8_1),.out(a5));
Celement j26(.in1(acka8_1),.in2(ackb7_1),.out(a6));
Celement j27(.in1(acka8_2),.in2(ackb7_2),.out(a7));
Celement j28(.in1(acka7_2),.in2(ackb8_2),.out(a8));
//Control Stage 4 (Fork/Join)
//Join
Celement j29(.in1(reqa5_1),.in2(reqa5_2),.out(req9_pre)); //X1[31:16]
Celement j30(.in1(reqa6_1),.in2(reqa6_2),.out(req10_pre)); //X1[15:0]
Celement j31(.in1(reqa7_1),.in2(reqa7_2),.out(req11_pre)); //X2[31:16]
Celement j32(.in1(reqa8_1),.in2(reqa8_2),.out(req12_pre)); //X4[15:0]
Celement j33(.in1(reqb5_1),.in2(reqb5_2),.out(req13_pre)); //X3[31:16]
Celement j34(.in1(reqb6_1),.in2(reqb6_2),.out(req14_pre)); //X3[15:0]
Celement j35(.in1(reqb7_1),.in2(reqb7_2),.out(req15_pre)); //X2[15:0]
Celement j36(.in1(reqb8_1),.in2(reqb8_2),.out(req16_pre)); //X4[31:16]
//delay elements for 16 bit adder pipeline
DelayElement_7 d9 ( .in(n54), .out(req9) );
DelayElement_6 d10 ( .in(n66), .out(req10) );
DelayElement_5 d11 ( .in(n78), .out(req11) );
DelayElement_4 d12 ( .in(n90), .out(req12) );
DelayElement_3 d13 ( .in(n60), .out(req13) );
DelayElement_2 d14 ( .in(n72), .out(req14) );
DelayElement_1 d15 ( .in(n96), .out(req15) );
DelayElement_0 d16 ( .in(n84), .out(req16) );
//Fork
assign acka5_1 = ack9;
assign acka5_2 = ack9;
assign acka6_1 = ack10;
assign acka6_2 = ack10;
assign acka7_1 = ack11;
assign acka7_2 = ack11;
assign acka8_1 = ack12;
assign acka8_2 = ack12;
assign ackb5_1 = ack13;
assign ackb5_2 = ack13;
assign ackb6_1 = ack14;
assign ackb6_2 = ack14;
assign ackb7_1 = ack15;
assign ackb7_2 = ack15;
assign ackb8_1 = ack16;
assign ackb8_2 = ack16;
handshake_ctl HC13(.lr(req9),.la(ack9),.rr(r9),.ra(a9),.ck(c9),.reset(reset));
handshake_ctl HC14(.lr(req10),.la(ack10),.rr(r10),.ra(a10),.ck(c10),.reset(rese
t));
handshake_ctl HC15(.lr(req11),.la(ack11),.rr(r11),.ra(a11),.ck(c11),.reset(rese
t));
handshake_ctl HC16(.lr(req12),.la(ack12),.rr(r12),.ra(a12),.ck(c12),.reset(rese
t));
handshake_ctl HC17(.lr(req13),.la(ack13),.rr(r13),.ra(a13),.ck(c13),.reset(rese
t));
handshake_ctl HC18(.lr(req14),.la(ack14),.rr(r14),.ra(a14),.ck(c14),.reset(rese
t));
handshake_ctl HC19(.lr(req15),.la(ack15),.rr(r15),.ra(a15),.ck(c15),.reset(rese
t));
handshake_ctl HC20(.lr(req16),.la(ack16),.rr(r16),.ra(a16),.ck(c16),.reset(rese
t));
//In/Out Fork-Join
//OUTPUT Join
Celement j37(.in1(r9),.in2(r10),.out(ro1)); //X1
Celement j38(.in1(r11),.in2(r15),.out(ro2)); //X2
Celement j39(.in1(r13),.in2(r14),.out(ro3)); //X3
Celement j40(.in1(r12),.in2(r16),.out(ro4)); //X4
//INPUT Fork
assign a9 = ao1;
assign a10 = ao1;
assign a11 = ao2;
assign a15 = ao2;
assign a13 = ao3;
assign a14 = ao3;
assign a12 = ao4;
assign a16 = ao4;
DELAX3 U1 ( .I(req1_pre), .O(n1) );
DELAX3 U2 ( .I(n1), .O(n2) );
DELAX3 U3 ( .I(n2), .O(n3) );
DELAX3 U4 ( .I(n3), .O(n4) );
```

```
DELAX3 U5 ( .I(n4), .O(n5) );
DELAX3 U6 ( .I(n5), .O(n6) );
DELAX3 U7 ( .I(req2_pre), .O(n7) );
DELAX3 U8 ( .I(n7), .O(n8) );
DELAX3 U9 ( .I(n8), .O(n9) );
DELAX3 U10 ( .I(n9), .O(n10) );
DELAX3 U11 ( .I(n10), .O(n11) );
DELAX3 U12 ( .I(n11), .O(n12) );
DELAX3 U13 ( .I(req5_pre), .O(n13) );
DELAX3 U14 ( .I(n13), .O(n14) );
DELAX3 U15 ( .I(n14), .O(n15) );
DELAX3 U16 ( .I(n15), .O(n16) );
DELAX3 U17 ( .I(n16), .O(n17) );
DELAX3 U18 ( .I(n17), .O(n18) );
DELAX3 U19 ( .I(req6_pre), .O(n19) );
DELAX3 U20 ( .I(n19), .O(n20) );
DELAX3 U21 ( .I(n20), .O(n21) );
DELAX3 U22 ( .I(n21), .O(n22) );
DELAX3 U23 ( .I(n22), .O(n23) );
DELAX3 U24 ( .I(n23), .O(n24) );
DELAX3 U25 ( .I(req3_pre), .O(n25) );
DELAX3 U26 ( .I(n25), .O(n26) );
DELAX3 U27 ( .I(n26), .O(n27) );
DELAX3 U28 ( .I(n27), .O(n28) );
DELAX3 U29 ( .I(n28), .O(n29) );
DELAX3 U30 ( .I(n29), .O(n30) );
DELAX3 U31 ( .I(req4_pre), .O(n31) );
DELAX3 U32 ( .I(n31), .O(n32) );
DELAX3 U33 ( .I(n32), .O(n33) );
DELAX3 U34 ( .I(n33), .O(n34) );
DELAX3 U35 ( .I(n34), .O(n35) );
DELAX3 U36 ( .I(n35), .O(n36) );
DELAX3 U37 ( .I(req7_pre), .O(n37) );
DELAX3 U38 ( .I(n37), .O(n38) );
DELAX3 U39 ( .I(n38), .O(n39) );
DELAX3 U40 ( .I(n39), .O(n40) );
DELAX3 U41 ( .I(n40), .O(n41) );
DELAX3 U42 ( .I(n41), .O(n42) );
DELAX3 U43 ( .I(req8_pre), .O(n43) );
DELAX3 U44 ( .I(n43), .O(n44) );
DELAX3 U45 ( .I(n44), .O(n45) );
DELAX3 U46 ( .I(n45), .O(n46) );
DELAX3 U47 ( .I(n46), .O(n47) );
DELAX3 U48 ( .I(n47), .O(n48) );
DELAX3 U49 ( .I(req9_pre), .O(n49) );
DELAX3 U50 ( .I(n49), .O(n50) );
```

```
DELAX3 U51 ( .I(n50), .O(n51) );
DELAX3 U52 ( .I(n51), .O(n52) );
DELAX3 U53 ( .I(n52), .O(n53) );
DELAX3 U54 ( .I(n53), .O(n54) );
DELAX3 U55 ( .I(req13_pre), .O(n55) );
DELAX3 U56 ( .I(n55), .O(n56) );
DELAX3 U57 ( .I(n56), .O(n57) );
DELAX3 U58 ( .I(n57), .O(n58) );
DELAX3 U59 ( .I(n58), .O(n59) );
DELAX3 U60 ( .I(n59), .O(n60) );
DELAX3 U61 ( .I(req10_pre), .O(n61) );
DELAX3 U62 ( .I(n61), .O(n62) );
DELAX3 U63 ( .I(n62), .O(n63) );
DELAX3 U64 ( .I(n63), .O(n64) );
DELAX3 U65 ( .I(n64), .O(n65) );
DELAX3 U66 ( .I(n65), .O(n66) );
DELAX3 U67 ( .I(req14_pre), .O(n67) );
DELAX3 U68 ( .I(n67), .O(n68) );
DELAX3 U69 ( .I(n68), .O(n69) );
DELAX3 U70 ( .I(n69), .O(n70) );
DELAX3 U71 ( .I(n70), .O(n71) );
DELAX3 U72 ( .I(n71), .O(n72) );
DELAX3 U73 ( .I(req11_pre), .O(n73) );
DELAX3 U74 ( .I(n73), .O(n74) );
DELAX3 U75 ( .I(n74), .O(n75) );
DELAX3 U76 ( .I(n75), .O(n76) );
DELAX3 U77 ( .I(n76), .O(n77) );
DELAX3 U78 ( .I(n77), .O(n78) );
DELAX3 U79 ( .I(req16_pre), .O(n79) );
DELAX3 U80 ( .I(n79), .O(n80) );
DELAX3 U81 ( .I(n80), .O(n81) );
DELAX3 U82 ( .I(n81), .O(n82) );
DELAX3 U83 ( .I(n82), .O(n83) );
DELAX3 U84 ( .I(n83), .O(n84) );
DELAX3 U85 ( .I(req12_pre), .O(n85) );
DELAX3 U86 ( .I(n85), .O(n86) );
DELAX3 U87 ( .I(n86), .O(n87) );
DELAX3 U88 ( .I(n87), .O(n88) );
DELAX3 U89 ( .I(n88), .O(n89) );
DELAX3 U90 ( .I(n89), .O(n90) );
DELAX3 U91 ( .I(req15_pre), .O(n91) );
DELAX3 U92 ( .I(n91), .O(n92) );
DELAX3 U93 ( .I(n92), .O(n93) );
DELAX3 U94 ( .I(n93), .O(n94) );
DELAX3 U95 ( .I(n94), .O(n95) );
DELAX3 U96 ( .I(n95), .O(n96) );
```

```
    endmodule
```

4-point FFT EB Maximal Control Verilog Code

```
module delayele_adder16_3 ( in, out );
    input in;
    output out;
    INVX1 U1 ( .I(in), .O(out) );
    endmodule
    module delayele_adder16_2 ( in, out );
    input in;
    output out;
    wire n2, n3;
    DELAX3 U1 ( .I(n3), .O(n2) );
    INVCKXLP U2 ( .I(in), .O(n3) );
    BUFX1 U3 ( .I(n2), .O(out) );
    endmodule
    module DelayElement_1 ( in, out );
    input in;
    output out;
    wire t1;
    delayele_adder16_3 d1 ( .in(in), .out(t1) );
    delayele_adder16_2 d2 ( .in(t1), .out(out) );
    endmodule
    module delayele_adder16_0 ( in, out );
    input in;
    output out;
    wire n2, n3, n4, n5, n6, n7, n8, n9, n10, n11, n12, n13, n14, n15, n16,
    n17, n18, n19, n20, n21, n22, n23, n24, n25, n26, n27, n28, n29, n30,
    n31, n32, n33, n34, n35, n36, n37, n38, n39, n40;
    BUFX1 U1 ( .I(n2), .O(out) );
    BUFX1 U2 ( .I(n3), .O(n2) );
    BUFX1 U3 ( .I(n4), .O(n3) );
    BUFX1 U4 ( .I(n5), .O(n4) );
    BUFX1 U5 ( .I(n6), .O(n5) );
    BUFX1 U6 ( .I(n7), .O(n6) );
    BUFX1 U7 ( .I(n8), .O(n7) );
    BUFX1 U8 ( .I(n9), .O(n8) );
    BUFX1 U9 ( .I(n10), .O(n9) );
    BUFX1 U10 ( .I(n11), .O(n10) );
    BUFX1 U11 ( .I(n12), .O(n11) );
    BUFX1 U12 ( .I(n13), .O(n12) );
    BUFX1 U13 ( .I(n14), .O(n13) );
    BUFX1 U14 ( .I(n15), .O(n14) );
    BUFX1 U15 ( .I(n16), .O(n15) );
```

```
BUFX1 U16 ( .I(n17), .O(n16) );
BUFX1 U17 ( .I(n18), .O(n17) );
BUFX1 U18 ( .I(n19), .O(n18) );
BUFX1 U19 ( .I(n20), .O(n19) );
BUFX1 U20 ( .I(n21), .O(n20) );
BUFX1 U21 ( .I(n22), .O(n21) );
BUFX1 U22 ( .I(n23), .O(n22) );
BUFX1 U23 ( .I(n24), .O(n23) );
BUFX1 U24 ( .I(n25), .O(n24) );
BUFX1 U25 ( .I(n26), .O(n25) );
BUFX1 U26 ( .I(n27), .O(n26) );
BUFX1 U27 ( .I(n28), .O(n27) );
BUFX1 U28 ( .I(n29), .O(n28) );
BUFX1 U29 ( .I(n30), .O(n29) );
BUFX1 U30 ( .I(n31), .O(n30) );
BUFX1 U31 ( .I(n32), .O(n31) );
BUFX1 U32 ( .I(n33), .O(n32) );
BUFX1 U33 ( .I(n34), .O(n33) );
BUFX1 U34 ( .I(n35), .O(n34) );
BUFX1 U35 ( .I(n36), .O(n35) );
BUFX1 U36 ( .I(n37), .O(n36) );
BUFX1 U37 ( .I(n38), .O(n37) );
BUFX1 U38 ( .I(n39), .O(n38) );
BUFX1 U39 ( .I(n40), .O(n39) );
INVX1 U40 ( .I(in), .O(n40) );
endmodule
module delayele_adder16_1 ( in, out );
input in;
output out;
INVX1 U1 ( .I(in), .O(out) );
endmodule
module DelayElement_0 ( in, out );
input in;
output out;
wire t1;
delayele_adder16_1 d1 ( .in(in), .out(t1) );
delayele_adder16_0 d2 ( .in(t1), .out(out) );
endmodule
module FFT4_control_async(
input wire reset,
input wire rin1,
input wire rin2,
input wire rin3,
input wire rin4,
input wire ao1,
input wire ao2,
input wire ao3,
input wire ao4,
output wire ro1,
output wire ro2,
output wire ro3,
output wire ro4,
output wire ain1,
output wire ain2,
output wire ain3,
output wire ain4,
output wire ci1,
output wire ci2,
output wire ci3,
output wire ci4,
output wire c1,
output wire c2,
output wire c3,
output wire c4,
output wire c5,
output wire c6,
output wire c7,
output wire c8,
output wire c9,
output wire c10,
output wire c11,
output wire c12,
output wire c13,
output wire c14,
output wire c15,
output wire c16
);
wire ri1,ri2,ri3,ri4,ai1,ai2,ai3,ai4;
wire req_a,req_b,req_c,req_d,req1,req2;
wire ack_a,ack_b,ack_c,ack_d;
wire req_bundle1,req_bundle2;
wire req_bundle1_pre;
wire ack_bundle1,ack_bundle2;
wire r_bundle1,r_bundle2;
wire a_bundle1,a_bundle2;
wire c_bundle1,c_bundle2;
wire n1, n2, n3, n4, n5, n6;
handshake_ctl HC1(.lr(rin1),.la(ain1),.rr(ri1),.ra(ai1),.ck(ci1),.reset(reset));
handshake_ctl HC2(.lr(rin2),.la(ain2),.rr(ri2),.ra(ai2),.ck(ci2),.reset(reset));
handshake_ctl HC3(.lr(rin3),.la(ain3),.rr(ri3),.ra(ai3),.ck(ci3),.reset(reset));
handshake_ctl HC4(.lr(rin4),.la(ain4),.rr(ri4),.ra(ai4),.ck(ci4),.reset(reset));
//Control Stage 1 (Fork/Join)
```

```
//Fork
assign ai1 = ack_bundle1;
assign ai2 = ack_bundle1;
assign ai3 = ack_bundle1;
assign ai4 = ack_bundle1;
//Bundle1 <x1 x2 x3 x4 > => <reala imaga realb imagb realc imag c reald imag d>
//Join
//Substage 1
Celement j1(.in1(ri1),.in2(ri3),.out(req1));
Celement j2(.in1(ri2),.in2(ri4),.out(req2));
//Substage 2
Celement j3(.in1(req1),.in2(req2),.out(req_bundle1_pre));
//delay elements for 16 bit adder pipeline
DelayElement_1 d1 ( .in(n6), .out(req_bundle1) );
handshake_ctl HC5(.lr(req_bundle1),.la(ack_bundle1),.rr(r_bundle1),
    .ra(a_bundle1),.ck(c_bundle1),.reset(reset));
//Bundle FORK/JOIN
//Fork
assign c1 = c_bundle1;
assign c2 = c_bundle1;
assign c5 = c_bundle1;
assign c6 = c_bundle1;
assign c3 = c_bundle1;
assign c4 = c_bundle1;
assign c7 = c_bundle1;
assign c8 = c_bundle1;
//Control Stage 2 (Fork/Join)
assign a_bundle1 = ack_bundle2;
//Bundle2 <a b c d> => <X1 X2 X3 X4>
//delay elements for 16 bit adder pipeline
DelayElement_0 d2 ( .in(r_bundle1), .out(req_bundle2) );
handshake_ctl HC6(.lr(req_bundle2),.la(ack_bundle2),.rr(r_bundle2),
    .ra(a_bundle2),.ck(c_bundle2),.reset(reset));
//Bundle FORK/JOIN
//Fork
assign c9 = c_bundle2;
assign c10 = c_bundle2;
assign c11 = c_bundle2;
assign c12 = c_bundle2;
assign c13 = c_bundle2;
assign c14 = c_bundle2;
assign c15 = c_bundle2;
assign c16 = c_bundle2;
//Not a join. Since we know the order that ao1 -> ao2 -> ao3 -> ao4, ack ao4 will indicate when bundle <X1 X2 X3
X4> should be generated. hence below
assign a_bundle2 = ao4;
```

```
handshake_ctl HC7(.lr(r_bundle2),.rr(ro1),.ra(ao1),.reset(reset));
handshake_ctl HC8(.lr(r_bundle2),.rr(ro2),.ra(ao2),.reset(reset));
handshake_ctl HC9(.lr(r_bundle2),.rr(ro3),.ra(ao3),.reset(reset));
handshake_ctl HC10(.lr(r_bundle2),.rr(ro4),.ra(ao4),.reset(reset));
DELAX3 U1 ( .I(req_bundle1_pre), .O(n1) );
DELAX3 U2 ( .I(n1), .O(n2) );
DELAX3 U3 ( .I(n2), .O(n3) );
DELAX3 U4 ( .I(n3), .O(n4) );
DELAX3 U5 ( .I(n4), .O(n5) );
DELAX3 U6 ( .I(n5), .O(n6) );
endmodule
```

4-point FFT Top-Level Verilog Code

```
module FFT4async(
    input wire [31:0] D_in,
    output wire [31:0] D_out,
    input wire reqi,
    output wire acki,
    output wire reqo,
    input wire acko,
    input wire reset
    );
    wire [31:0] x1,x2,x3,x4,Din,Dout;
    wire [1:0] sel,sel_nxt,sel_temp,select,select_nxt,select_temp;
    wire [31:0] X1,X2,X3,X4;
    wire r1,r2,r3,r4,a1,a2,a3,a4;
    wire ri,ai,ro,ao,ri_delay;
    wire ci,co,t1,t2,t3,t4,t5,t6;
    wire req1,req2,req3,req4,ack1,ack2,ack3,ack4;
    wire req1_delay,req2_delay,req3_delay,req4_delay;
    wire ci1,ci2,ci3,ci4,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16;
    handshake_ctl HC1(.lr(reqi),.la(acki),.rr(ri),.ra(ai),.ck(ci),.reset(~reset));
    reg32_async R1(.D(D_in),.Q(Din),.ck(ci),.reset(reset));
    //delay element for Clock to Q delay
    delayele_Reg2Reg d1(.in(ri),.out(ri_delay));
    //First stage at ack ai
    reg_counter2_async R2(.D(sel_nxt),.Q(sel_temp),.ck(ai),.reset(reset));
    //Second stage at ack ~ai
    //Time between ai and ~ai is about 400 ps, CtoQ delay is not violated
    reg_counter2_async R3(.D(sel_temp),.Q(sel),.ck(~ai),.reset(reset));
    //There is enough delay between consecutive ai pulses. Delay element not require
    d . (actually satisfied by delay d1)
    counter2_async count1(.in(sel),.out(sel_nxt));
    //no delay element required here since request is automatically delayed by logic
```

```
decimator4_async decimator(
.R(ri_delay),
.sel(sel),
.a1(a1),
.a2(a2),
.a3(a3),
.a4(a4),
.r1(r1),
.r2(r2),
.r3(r3),
.r4(r4),
.A(ai)
);
FFT4_control_async control(
.reset(~reset),
.rin1(r1),.rin2(r2),.rin3(r3),.rin4(r4),
.ao1(ack1),.ao2(ack2),.ao3(ack3),.ao4(ack4),
.ro1(req1),.ro2(req2),.ro3(req3),.ro4(req4),
.ain1(a1),.ain2(a2),.ain3(a3),.ain4(a4),
.ci1(ci1),.ci2(ci2),.ci3(ci3),.ci4(ci4),
.c1(c1),.c2(c2),.c3(c3),.c4(c4),.c5(c5),.c6(c6),.c7(c7),.c8(c8),
.c9(c9),.c10(c10),.c11(c11),.c12(c12),.c13(c13),.c14(c14),.c15(c15),.c16(c16
)
);
FFT4_data path_async data path(
.reset(reset),
.x(Din),
.ci1(ci1),.ci2(ci2),.ci3(ci3),.ci4(ci4),
.c1(c1),.c2(c2),.c3(c3),.c4(c4),.c5(c5),.c6(c6),.c7(c7),.c8(c8),
.c9(c9),.c10(c10),.c11(c11),.c12(c12),.c13(c13),.c14(c14),.c15(c15),.c16(c16
),
.X1(X1),.X2(X2),.X3(X3),.X4(X4)
);
//delay element for Clock to Q delay from data path output
delayele_Reg2Reg d2(.in(req1),.out(req1_delay));
delayele_Reg2Reg d3(.in(req2),.out(req2_delay));
delayele_Reg2Reg d4(.in(req3),.out(req3_delay));
delayele_Reg2Reg d5(.in(req4),.out(req4_delay));
//First stage at ack a0
reg_counter2_async R4(.D(select_nxt),.Q(select_temp),.ck(ao),.reset(reset));
//Second stage at ack falls ~ao
//Time between ao and ~ao is 400 ps, CtoQ delay is not violated
reg_counter2_async R5(.D(select_temp),.Q(select),.ck(~ao),.reset(reset));
counter2_async count2(.in(select),.out(select_nxt));
expander4_async expander(
.sel(select),
.r1(req1_delay),
.r2(req2_delay),
.r3(req3_delay),
.r4(req4_delay),
.A(ao),
.D1(X1),
.D2(X2),
.D3(X3),
.D4(X4),
.R(ro),
.a1(ack1),
.a2(ack2),
.a3(ack3),
.a4(ack4),
.Dout(Dout)
);
handshake_ctl HC2(.lr(ro),.la(ao),.rr(reqo),.ra(acko),.ck(co),.reset(~reset));
reg32_async R6(.D(Dout),.Q(D_out),.ck(co),.reset(reset));
endmodule
```

Appendix C

Timing Assumption Scripts for EB Maximal 4-point FFT

This section presents the synthesis timing constraints and assumptions used to synthesise the 4-point FFT EB Maximal circuit of Chapter 5.

setup.tcl

This script sets up the design to technology library and gives basic timing assumptions for design synthesis.

```
set search_path {. /eda/designkit/Faraday/L90_SP_RVT/
    fsd0a_a/2009Q2v2.0/GENERIC_CORE_1D0V/FrontEnd/synopsys}
set target_library {fsd0a_a_generic_core_ss0p9v125c.db
    fsd0a_a_generic_core_ff1p1vm40c.db}
set symbol_library {fsd0a_a_generic_core.sdb}
set link_library [concat * $target_library]
set design FFT4async
set testbench FFT4async_tb
set top_level FFT4async
set clk clock
set clk_period 2
set expander4_period 0.175
set counter2_period 0.03
set latch_ckq 0.3
set req_del_min 2.3
set req_del_max 2.5
set lib_name "fsd0a_a_generic_core_ss0p9v125c"
set dff_cell "DFFRBX1"
set lib_dff_d "$lib_name/$dff_cell/D"
set dff_setup 0.1
set dff_ckq 0.1
set rst reset
set race_margin 0.1
```

delays.sdc

This script provides the timing constraints for handshake control logic and datapath control logic. The script is responsible for sizing and synthesis of delay elements.

```
set_max_delay $expander4_period -from expander/D1 -to R6/D
set_max_delay $expander4_period -from expander/D2 -to R6/D
set_max_delay $expander4_period -from expander/D3 -to R6/D
set_max_delay $expander4_period -from expander/D4 -to R6/D
set_min_delay $expander4_period -from expander/r1 -to expander/R
set_min_delay $expander4_period -from expander/r2 -to expander/R
set_min_delay $expander4_period -from expander/r3 -to expander/R
set_min_delay $expander4_period -from expander/r4 -to expander/R
set_max_delay $counter2_period -from count1/in -to count1/out
set_max_delay $counter2_period -from count2/in -to count2/out
set_max_delay $clk_period -from data path/R1/Q -to data path/R5/D
set_max_delay $clk_period -from data path/R1/Q -to data path/R6/D
set_max_delay $clk_period -from data path/R1/Q -to data path/R9/D
set_max_delay $clk_period -from data path/R1/Q -to data path/R10/D
set_max_delay $clk_period -from data path/R3/Q -to data path/R5/D
set_max_delay $clk_period -from data path/R3/Q -to data path/R6/D
set_max_delay $clk_period -from data path/R3/Q -to data path/R9/D
set_max_delay $clk_period -from data path/R3/Q -to data path/R10/D
set_max_delay $clk_period -from data path/R2/Q -to data path/R7/D
set_max_delay $clk_period -from data path/R2/Q -to data path/R8/D
set_max_delay $clk_period -from data path/R2/Q -to data path/R11/D
set_max_delay $clk_period -from data path/R2/Q -to data path/R12/D
set_max_delay $clk_period -from data path/R4/Q -to data path/R7/D
set_max_delay $clk_period -from data path/R4/Q -to data path/R8/D
set_max_delay $clk_period -from data path/R4/Q -to data path/R11/D
set_max_delay $clk_period -from data path/R4/Q -to data path/R12/D
set_max_delay $clk_period -from data path/R5/Q -to data path/R13/D
set_max_delay $clk_period -from data path/R7/Q -to data path/R13/D
set_max_delay $clk_period -from data path/R6/Q -to data path/R14/D
set_max_delay $clk_period -from data path/R8/Q -to data path/R14/D
set_max_delay $clk_period -from data path/R5/Q -to data path/R17/D
set_max_delay $clk_period -from data path/R7/Q -to data path/R17/D
set_max_delay $clk_period -from data path/R6/Q -to data path/R18/D
set_max_delay $clk_period -from data path/R8/Q -to data path/R18/D
set_max_delay $clk_period -from data path/R9/Q -to data path/R15/D
set_max_delay $clk_period -from data path/R12/Q -to data path/R15/D
set_max_delay $clk_period -from data path/R10/Q -to data path/R16/D
set_max_delay $clk_period -from data path/R11/Q -to data path/R16/D
set_max_delay $clk_period -from data path/R10/Q -to data path/R19/D
set_max_delay $clk_period -from data path/R11/Q -to data path/R19/D
set_max_delay $clk_period -from data path/R9/Q -to data path/R20/D
set_max_delay $clk_period -from data path/R12/Q -to data path/R20/D
```

```
#Datapath Control constraints
set_min_delay $latch_ckq -rise_from HC1/rr -rise_to decimator/R
set_min_delay $latch_ckq -rise_from control/ro1 -rise_to expander/r1
set_min_delay $latch_ckq -rise_from control/ro2 -rise_to expander/r2
set_min_delay $latch_ckq -rise_from control/ro3 -rise_to expander/r3
set_min_delay $latch_ckq -rise_from control/ro4 -rise_to expander/r4
#control constraints
set_min_delay $req_del_min -rise_from control/HC1/rr -rise_to control/HC5/lr
set_min_delay $req_del_min -rise_from control/HC3/rr -rise_to control/HC5/lr
set_min_delay $req_del_min -rise_from control/HC2/rr -rise_to control/HC5/lr
set_min_delay $req_del_min -rise_from control/HC4/rr -rise_to control/HC5/lr
set_min_delay $req_del_min -rise_from control/HC5/rr -rise_to control/HC6/lr
```

constraints.tcl

This script provides the constraints for timing-driven synthesis of asynchronous control logic and control logic optimisation.

```
#Latch timing constraints (critical path optimisation and delay element)
source "delays.sdc"
#Prevent tool from optimising gates
set_size_only -all_instances { */lc1 }
set_size_only -all_instances { */lc3 }
set_size_only -all_instances { */lc4 }
set_size_only -all_instances { */lc5 }
set_size_only -all_instances { */lc6 }
set_size_only -all_instances { */*/lc1 }
set_size_only -all_instances { */*/lc3 }
set_size_only -all_instances { */*/lc4 }
set_size_only -all_instances { */*/lc5 }
set_size_only -all_instances { */*/lc6 }
set_dont_touch [get_cells { */j1 */j2 */j3 }]
#set_size_only -all_instances { */*/Cele1 }
#break local cycles
set_disable_timing -from B3 -to O [find -hier cell *lc1]
set_disable_timing -from A2 -to O [find -hier cell *lc1]
set_disable_timing -from B3 -to O [find -hier cell *lc3]
set_disable_timing -from A2 -to O [find -hier cell *lc3]
set_disable_timing -from B1 -to O [find -hier cell *lc5]
set_disable_timing -from B1 -to O [find -hier cell *Cele1]
#break handshake protocol cycles
set_disable_timing -from B2 -to O [find -hier cell *lc1]
set_disable_timing -from B2 -to O [find -hier cell *lc3]
set_disable_timing -from A1 -to O [find -hier cell *lc3]
set_disable_timing -from B1 -to O [find -hier cell *lc3]
```

Appendix D

In this section, the CDFG of the 4-point FFT architectures used in the several implementations of the 16-point FFT case study is depicted. Note that the EB Reuse Bundle implementation involves additional control circuit requirement due to additional control overhead of switching the muxes and demuxes for resource sharing of ADDs and SUBs.

4-point FFT Asynchronous CDFG

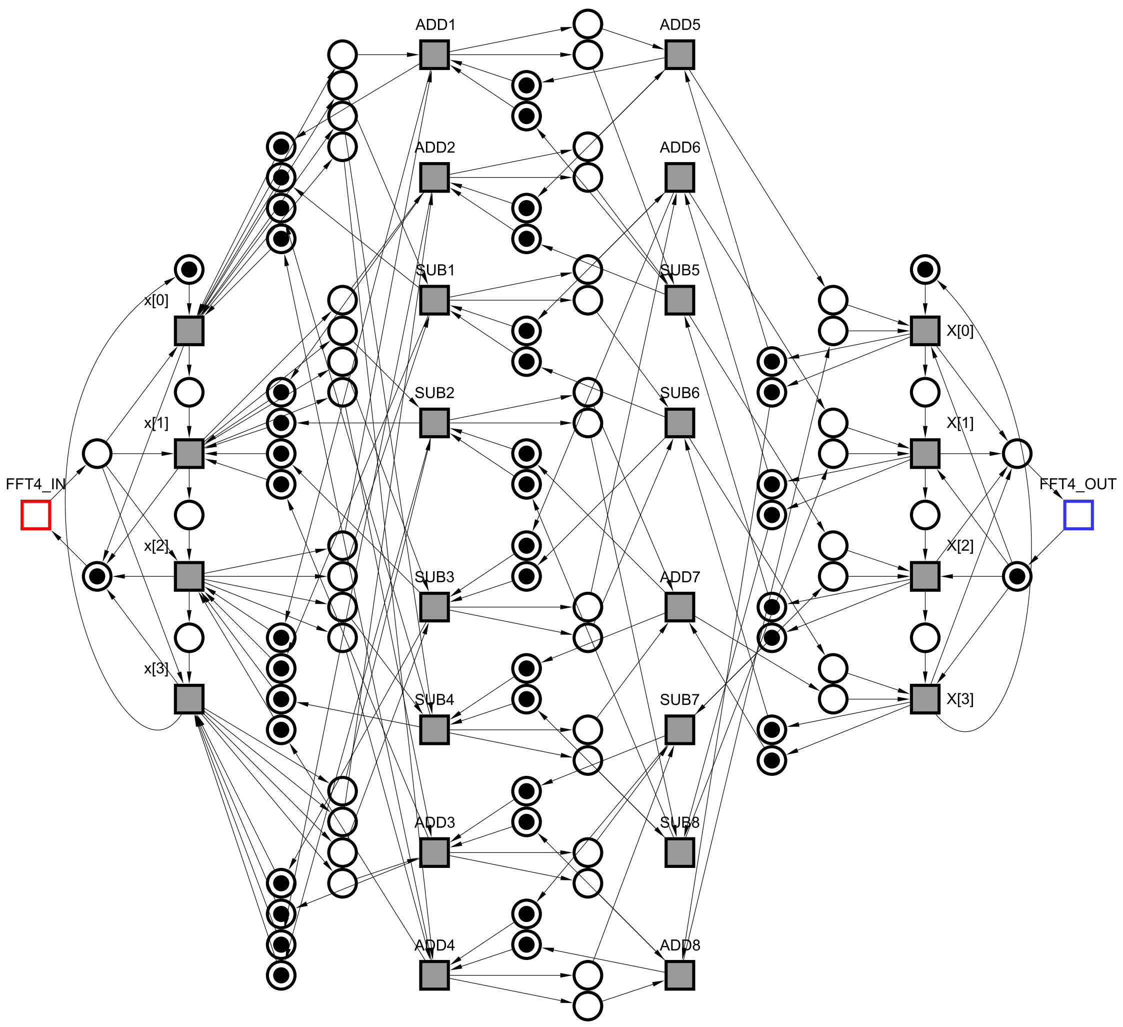


Figure 6.1: Petri net of asynchronous 4-point FFT

4-point FFT Temporal Bundle CDFG

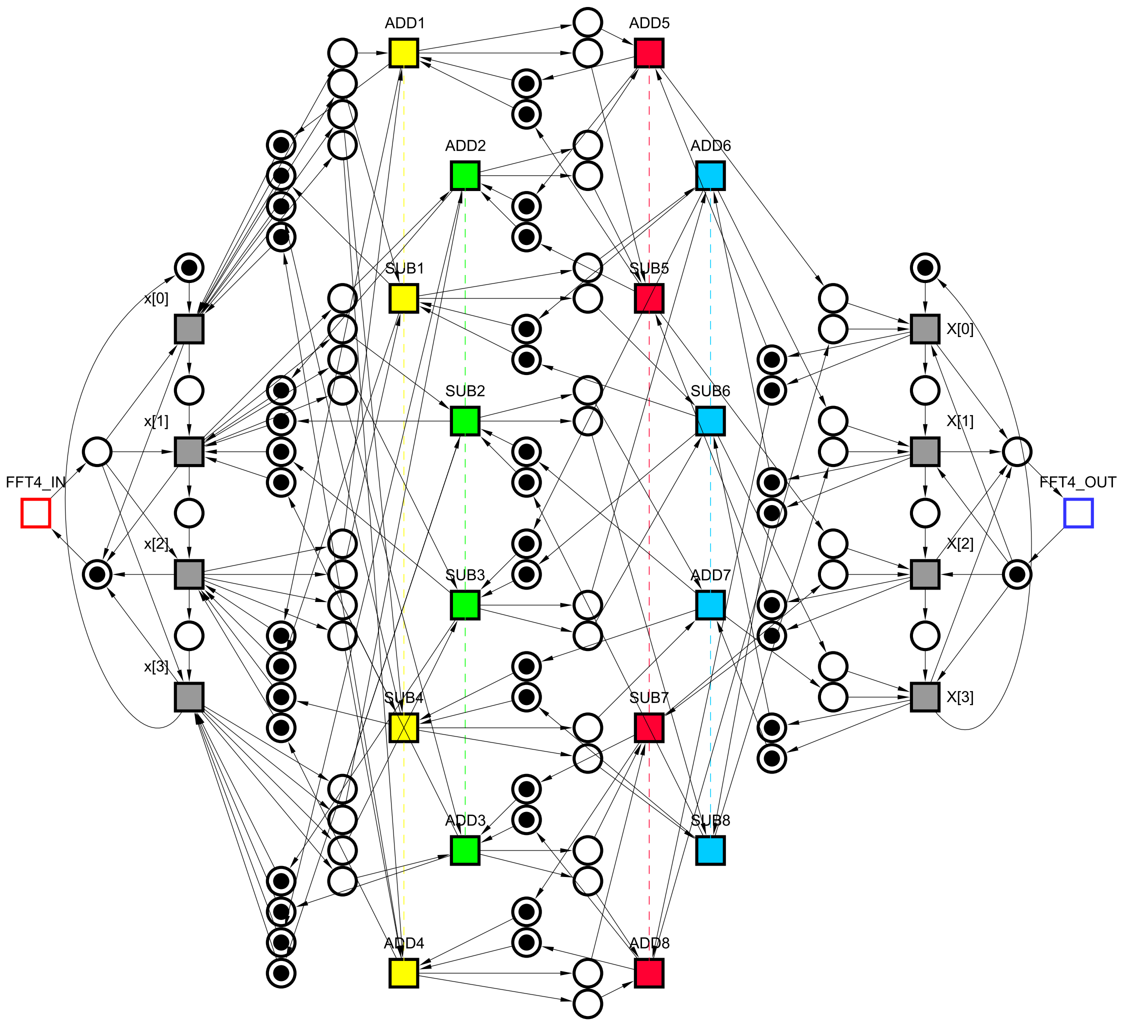


Figure 6.2: Policy net of 4-point FFT with temporal partitioning

4-point FFT Maximal Bundle CDFG

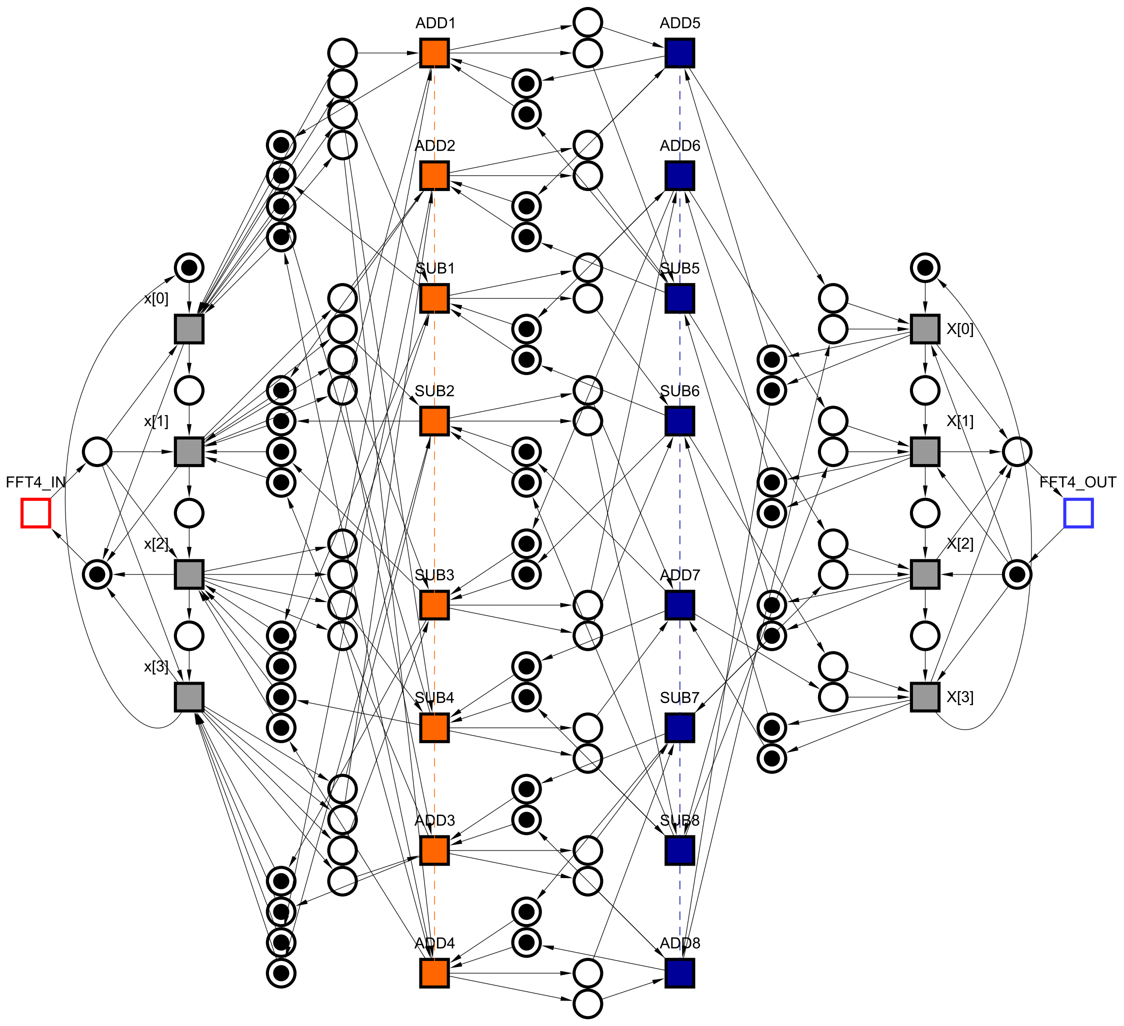


Figure 6.3: Policy net of 4-point FFT with maximal partitioning

4-point FFT Reuse Bundle CDFG and Control Circuits

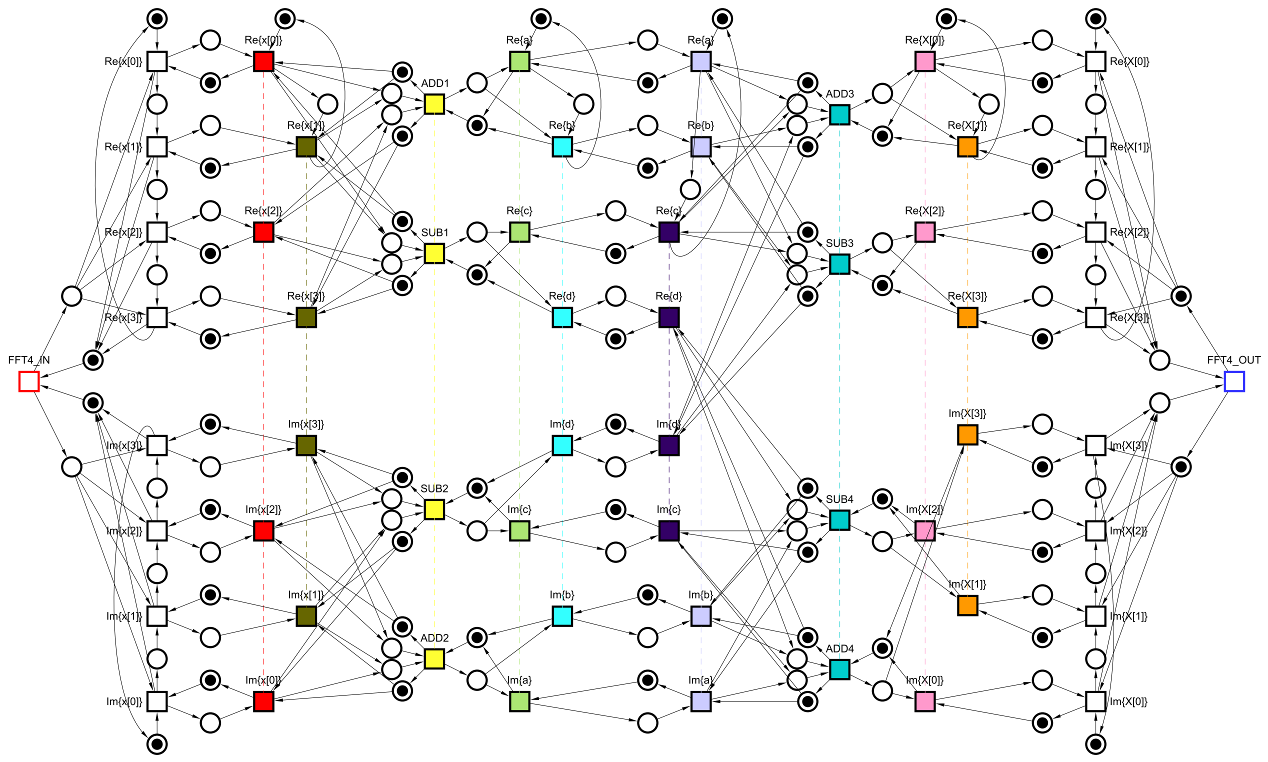
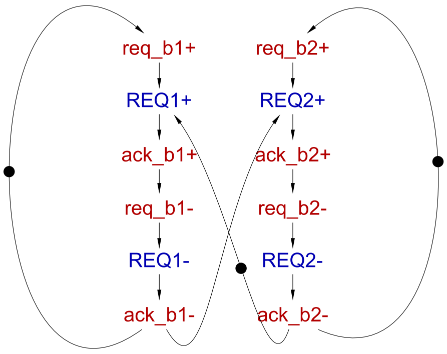
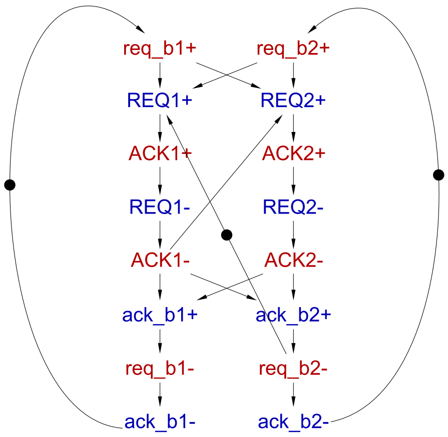


Figure 6.4: Policy net of 4-point FFT with ADD/SUB reuse partitioning



(a) Mux control for pipeline stage 1



(b) Mux control for pipeline stage 2

Figure 6.5: Signal transition graph (STG) for 4-point FFT reuse bundle control

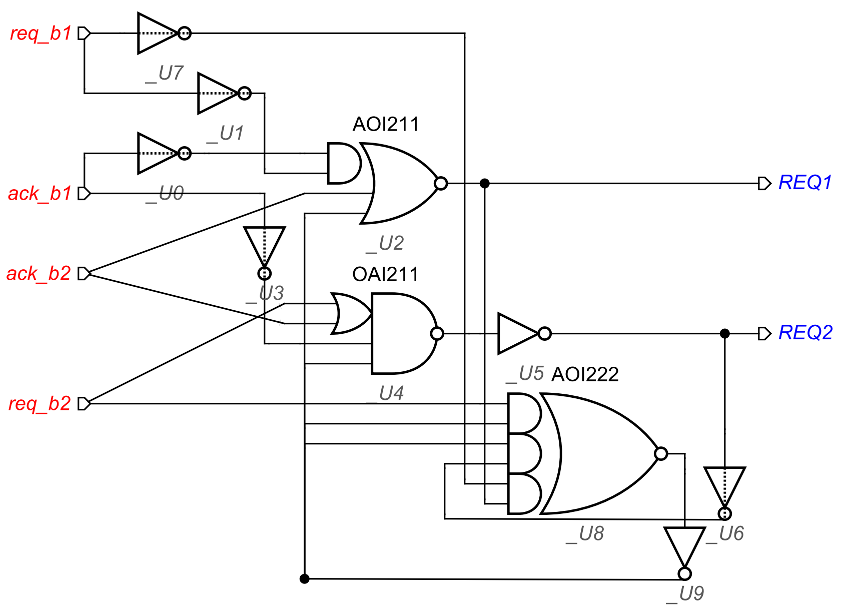


Figure 6.6: 4-point FFT pipeline stage 1 bundle control circuit

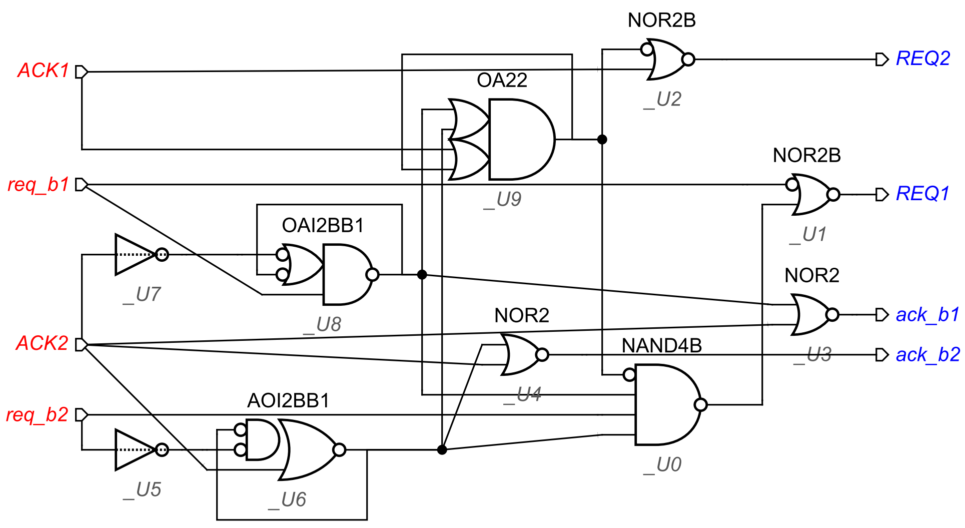


Figure 6.7: 4-point FFT pipeline stage 2 bundle control circuit