PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

PULSAR: TOWARDS A SYNTHESIS FLOW FOR QDI CIRCUITS

MARCOS LUIGGI LEMOS SARTORI

Dissertation submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fullfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Ney Laert Vilar Calazans
Co-Advisor: Dr. Matheus Trevisan Moreira

Porto Alegre
2019

REPLACE THIS PAGE WITH THE LIBRARY CATALOG PAGE

REPLACE THIS PAGE WITH
THE COMMITTEE FORMS

To those I love,

“Creativity Is Intelligence Having Fun.”
(Albert Einstein)

PULSAR: EM DIREÇÃO A UM FLUXO DE SÍNTESE PARA CIRCUITOS QDI

RESUMO

Circuitos assíncronos quase-insensíveis a atrasos ou QDI são conhecidos pela robustez a variações de PVT. Isso os torna bons candidatos para uso de técnicas de projeto agressivas de redução da tensão de alimentação. No entanto, a adoção do projeto QDI em escala industrial é dificultada: (i) pela dependência de ferramentas de projeto especializadas para circuitos QDI; (ii) pela falta de integração com fluxos de projeto ASIC tradicionais. Esta Dissertação propõe Pulsar, um novo fluxo de síntese para o projeto QDI. Pulsar emprega ferramentas comerciais de automação de projeto eletrônico (EDA) para capturar o projeto, expandir descrições para uso de códigos insensíveis a atrasos, e realizar o mapeamento tecnológico e a otimização de circuitos QDI. Ferramentas EDA comerciais habilitam projetistas a definir objetivos de desempenho e equilibrar características de energia e área. Esta Dissertação traz seis contribuições originais: (i) um fluxo pseudo-síncrono estendido, que agrega novos modelos de registradores; (ii) o fluxo SDDS-NCL sequencial, para lidar com descrições de projeto genéricas (combinacionais e/ou sequenciais); (iii) o modelo *rede de canais meio-buffer* ou HBCN, que viabiliza a análise de temporização de circuitos assíncronos QDI meio-buffer; (iv) uma formulação de programação linear para restringir projetos a operar com um tempo de ciclo assíncrono alvo. (v) uma técnica de captura de projeto similar à RTL para circuitos síncronos e uma técnica associada para a expansão de descrições de circuitos para versões em trilha dupla; (vi) uma ferramenta de cálculo automatizado de restrições de síntese para circuitos QDI. Experimentos mostram que Pulsar permite o projeto de circuitos assíncronos a partir de descrições similares a RTL sujeitas a restrições de tempos de ciclo. Pulsar permite designar tempos de ciclo alvo para circuitos QDI usando ferramentas comerciais, o que constitui um avanço para projetistas QDI, que agora podem vincular com segurança métricas de desempenho de pior caso a projetos. Além disso, Pulsar permite que projetistas definam o desempenho para otimização de energia ou de área, capitalizando nas folgas dos orçamentos temporais do circuito.

Palavras-Chave: Circuitos assíncronos, QDI, EDA, NCL, modelagem, síntese, projeto.

PULSAR: TOWARDS A SYNTHESIS FLOW FOR QDI CIRCUITS

ABSTRACT

Asynchronous quasi-delay-insensitive (QDI) circuits are known for their robustness against PVT variations. This makes them good candidates for enabling aggressive voltage scaling design techniques. However, the adoption of QDI design by industries is hindered by: (i) the dependency on specialised design tools for QDI circuits; (ii) the lack of integration with traditional ASIC design flows. This Dissertation presents Pulsar, a new synthesis flow for QDI circuit design. Pulsar leverages commercial EDA tools for design capture, dual-rail expansion, technology mapping and optimisation of QDI circuits. Commercial EDA tools enable designers to define performance targets and naturally balance power and area optimisation. The Dissertation brings six main original contributions: (i) an extended pseudo-synchronous flow, with new register models; (ii) a generalised SDDS-NCL flow to deal with both combinational and sequential circuits; (iii) the proposition of half-buffer channel network (HBCN), a new model for timing analysis of half-buffer asynchronous circuits; (iv) a linear programming formulation to constrain a design to a target asynchronous cycle time. (v) an RTL-like design capture technique and an associated dual-rail expansion technique; (vi) a tool that automatically extracts the HBCN model of a circuit and computes its synthesis constraints. Results show that Pulsar enables the design of asynchronous circuits from an RTL-like description under cycle-time constraints. Pulsar enables the sign-off of target cycle times for QDI circuits using commercial EDA tools. This is a breakthrough for QDI designers, as they can now safely bound worst case performance metrics for applications. Moreover, Pulsar enables designers to naturally trade performance for power or area optimisations, whenever there is slack in timing budgets.

Keywords: Asynchronous circuits, QDI, EDA, NCL, modelling, synthesis, design.

LIST OF FIGURES

2.1 Examples of handshake protocols: (a) 2-phase; (b) 4-phase. 31
2.2 The basic transition diagram for transmitting binary data in a QDI channel, where S stands for the spacer. Adapted from [MBSC18]. 33
2.3 RTZ dual-rail channel operation: (a) encoding; (b) example waveforms. Adapted from [MBSC18]. 34
2.4 RTO dual-rail channel operation: (a) encoding; (b) example waveforms. Adapted from [MBSC18]. 35
2.5 Timing models used by the STA engine: (a) gate timing arcs; (b) timing paths. 36
3.1 Characteristics of NCL gates: (a) generic symbol; (a) generic symbol; (b) specific symbol example; (c) example truth table behaviour. 41
3.2 Example of an NCL implementation: the generate path of part of a Kogge-Stone adder. Adapted from [MTMC14]. 42
3.3 Characteristics of NCL+ gates: (a) generic symbol; (a) generic symbol; (b) specific symbol example; (c) example truth table behaviour. 43
3.4 Example of an NCL+ implementation: the generate path of part of a Kogge-Stone adder. Adapted from [MTMC14]. 43
3.5 Example of an SDDS-NCL circuit: the generate path for a Kogge-Stone adder. Adapted from [MTMC14]. 45
3.6 Two arc models for a resetable C-element: (a) conventional description; (b) pseudo-synchronous description [TBV12]. 48
3.7 Simplified view of a 1-bit data channel WCHB pipeline, showing the inner cycles controlled by the pseudo-clock. 49
3.8 Proposed characterisation models for a resetable C-element with a dummy clock pin, G: (a) pseudo-flop; (2) pseudo-latch. 50
3.9 Structure of a sequential SDDS-NCL pipeline. 51
3.10 The SDDS-NCL sequential synthesis flow. 52
3.11 Illustration of the RTO and RTZ gate classification process. 53
4.1 Observed cycle time distribution at the input and output of an integer, 6-stage, multiply-and-accumulate circuit. 59
4.2 FBCN modelling a 2-phase, half-buffer pipeline. Vertical lines are transitions, circles are forward-propagation places, squares are backward-propagation places and black dots are tokens. 60
4.3 A 2-stage dual-rail linear buffer, implemented using a WCHB half-buffer 4-stage RTZ pipeline. 63

4.4 Example HBCN modelling a 4-phase, half-buffer circuit. Blue lines are valid data transitions; red lines are spacer transitions; squares are backward propagation places; circles are forward propagation places. 64
4.5 HBCN of a 3-stage, circular half-buffer circuit. 67
5.1 The Pulsar synthesis flow. 75
5.2 Expansion of the `nand2` gate from: (a) a single-rail Boolean truth table to (b) a 3NCL code table. In the tables, (-) indicates hysteretic behaviour. 81
5.3 Map between codes. Only valid codewords are depicted. 81
5.4 Dual-rail RTZ code table of the `NAND2` gate with equivalent 3NCL codewords. Here, (-) means hold the previous value and (-/0) means that either: hold the previous value, or set it low. 82
5.5 Truth tables for the output rail functions: (a) false rail; (b) true rail. Here, (-) represents hysteretic behaviour and (?) represents that the value is a “don’t-care”. 83
5.6 Truth tables for the output rail virtual functions: (a) false rail; (b) true rail. Here, (?) represents that the input value is a “don’t-care”. 84
5.7 The `dff` half-buffer dual-rail register: (a) schematic; (b) HBCn model. 87
5.8 The two full-buffer components: (a) DFF with set ; (b) DFF with reset ; (c) the HBCN model. 88
5.9 The HBCN extracted from the structural graph depicted in Listing 5.8. 95
6.1 Results for the 3-stage MAC. 106
6.2 Results for the 4-stage MAC. 107
6.3 Results for the 5-stage MAC. 108
6.4 Results for the 6-stage MAC. 109
6.5 Results for the 6-stage MAC synthesised with timing path exceptions on free-slack. 110

LIST OF TABLES

4.1 Solution of the LP formulation. 68
4.2 Solutions to the LP problem in Algorithm 4.2 71
4.3 Solutions to the LP problem in Algorithm 4.3. 72
5.1 Combinational gates of the component library. 85
6.1 Characteristics of the virtual netlists for each MAC version. 104

LIST OF ALGORITHMS

3.1 The Fix X-Netlist algorithm. 54

4.1 LP formulation to compute the cycle time and free slack for the example HBCN. 68

4.2 LP formulation to constrain the cycle time of the HBCN in Figure 4.5 to 2ns, using the pseudo-clock method with fixed delays. 71

4.3 LP formulation to constrain the cycle time of the HBCN in Figure 4.5 to 2ns, using the pseudo-clock method. 72

CONTENTS

1 INTRODUCTION AND MOTIVATION **23**
1.1 ASYNCHRONOUS CIRCUITS AS AN ALTERNATIVE 24
1.2 CHALLENGES TO QDI ADOPTION 24
1.3 CONTRIBUTIONS OF THIS WORK 25

2 BACKGROUND **27**
2.1 GRAPHS AND PETRI NETS 27
2.1.1 PETRI NETS 28
2.1.2 PETRI NET PROPERTIES 29
2.1.3 MARKED GRAPHS 30
2.2 HANDSHAKE PROTOCOLS 31
2.3 QUASI-DELAY-INSENSITIVE DESIGN 32
2.4 STATIC TIMING ANALYSIS 35

3 THE PSEUDO-SYNCHRONOUS SDDS-NCL DESIGN TEMPLATE **39**
3.1 THE SDDS-NCL ASYNCHRONOUS DESIGN TEMPLATE (RELATED WORK) . 39
3.2 PSEUDO-SYNCHRONOUS WEAK-CONDITIONED HALF BUFFER (RELATED WORK) 47
3.3 THE PSEUDO-SYNCHRONOUS SDDS-NCL SYNTHESIS FLOW 50

4 A NEW MODEL FOR CYCLE TIME COMPUTATION **57**
4.1 CYCLE TIME OF ASYNCHRONOUS CIRCUITS 57
4.2 THE FBCN TIMING MODEL (RELATED WORK) 58
4.3 THE HBCN TIMING MODEL 61
4.4 CALCULATING THE MAXIMUM CYCLE TIME 65
4.5 CONSTRAINING THE MAXIMUM CYCLE TIME 69

5 THE PULSAR SYNTHESIS FLOW **75**
5.1 QDI SYNTHESIS SYSTEMS (RELATED WORK) 76
5.2 DUAL-RAIL CHANNELS 77
5.3 A LIBRARY OF COMPONENTS 79
5.3.1 COMBINATIONAL COMPONENTS 80
5.3.2 SEQUENTIAL COMPONENTS 86

5.4 A CASE STUDY: APPLYING THE DUAL RAIL EXPANSION FLOW 89
5.5 THE HBCN CONSTRUCTION AND THE CYCLE TIME CONSTRAINING 92
5.6 COMPLETING THE CASE STUDY SYNTHESIS 98

6 EXPERIMENTS AND RESULTS 101

7 CONCLUSION AND FUTURE WORK 111

REFERENCES 113

APPENDIX A – The Single-Rail Synthesis Scripts **119**

APPENDIX B – The Dual-Rail Expander **133**

APPENDIX C – The Cycle Time Constrainer **139**

APPENDIX D – The Pseudo-Synchronous Synthesis Scripts **147**

1. INTRODUCTION AND MOTIVATION

Advances in semiconductor fabrication technologies allow higher integration, but impose design challenges. Some of the challenges faced in newer technologies are: (i) higher sensitivity to process variations; (ii) higher static power; and (iii) longer wire delays. Process variations are consequence of imperfections in the fabrication process and manifest themselves as changes in some circuit electrical properties. Static power is the power dissipated by the circuit when idle, that is, when no switching activity is taking place. Higher static power can be attributed to higher leakage current, due to thinner oxide layers between the transistor channel and the transistor gate. It is roughly proportional to the number of transistors in the die, each providing a contribution to the overall leakage power. Larger wire delays are attributable to smaller wire cross sections and relatively longer wire length, which cause reduced wire current capacity, higher parasitic capacitance and (relatively) increased effects of coupling with neighbour wires. Wire delays in recent technologies make it unfeasible to route global signals in large circuits without using buffers to "repeat" the signal during its propagation.

They pose a challenge to synchronous circuits, as these employ a global "clock" signal to provide a discrete time reference for synchronisation. The clock signal distribution is assumed to be ideal, meaning that the clock arrives to all locations in the circuit (where it is used) at nominally the same time. Meeting this clock distribution criteria requires a clock distribution network composed of buffers and other signal distribution components. This clock distribution network can have a high cost in area and power. The power consumption associated to the clock distribution can be a significant percentage of the overall power dissipated by complex circuits. Taking 40% of the total power consumption is not unusual [DMM04]. Also, as ideal clock distribution networks with no skew are difficult to achieve or even impossible to obtain, it becomes necessary to compensate the skew and uncertainties by introducing margins in the clock period. This of course impacts overall circuit performance.

Modern designs mitigate these issues by dividing the circuit in clock domains and using synchronisers to transfer signals between clock domain boundaries. The approach only helps solving the problem locally. However, logic spread across a large area (e.g. interconnects) still suffers from clock distribution problems. Furthermore, the use of multiple clock domains can result in significant synchronisation overheads, as different clock domains are possibly working at different clock phases and/or operating frequencies. A possible solution is the overall elimination of global or semi-global clock signals. Digital circuits without any global or semi-global clock signals are known as *asynchronous circuits*.

1.1 Asynchronous Circuits as an Alternative

In asynchronous circuits the delay of individual pipeline stages is not constrained by a global clock signal. The overall performance of the circuit is constrained only by the actual delay of data processing in its pipelines. This creates opportunities to design innovative pipeline architectures [NS15a, NS15b]. The elimination of the constraints imposed by clock distribution allows the design of fine-grained pipelines. It also allows on-demand operation analogous to very fine-grained clock gating without the associated clock gating costs.

There are two major classes of asynchronous circuits: (i) Bundled Data (BD) is a class of asynchronous circuits where correct functionality depends on assumptions about the propagation delay of individual pipeline stages; (ii) Quasi Delay Insensitive (QDI) is a class of asynchronous circuits that use Delay Insensitive (DI) codes to better tolerate gate and wire delays and their variations. Besides the advantages provided by clock elimination, QDI circuits are naturally more resilient to delay variations.

A common source of delay variations are PVT variations, ageing and other challenges faced by recent sub-micron technologies. For example, intra-die process variation causes the same gate in different parts of the circuit to present different switching delays, what can possibly lead to faulty circuit behaviour and lower production yield. This problem affects interconnect circuits more acutely, as they span over a large silicon area. QDI circuits are thus excellent candidates to tolerate the conditions imposed by intrinsic intra-die process variations. Circuits that use QDI interconnects between synchronous components can take advantage of this aspect of QDI circuits. Furthermore, the mixed use of asynchronous circuits as interconnect for synchronous components solve some of the synchronisation problems between clock domains.

1.2 Challenges to QDI Adoption

The design of QDI circuits often relies on specialised infrastructures, which can frequently hinder the adoption of QDI circuit design. This infrastructure often includes: (i) specific gate libraries, containing e.g. C-Elements, NCL gates or PCHB logic cells; (ii) specific synthesis tools; and (iii) specific design capture languages.

QDI circuits normally require the use of gates with hysteretic behaviour to facilitate or enable handshake synchronisation. A hysteretic gate holds the output stable until certain criteria are met. These special gates, are not usually available in conventional cell libraries designed for synchronous semi-custom ASIC flows.

Specialised tools like Uncle [RST12] and Balsa [EB02], both briefly covered in Section 5.1, can be used to produce asynchronous circuits. However, these specialised tools

do not integrate well with semi-custom ASIC flows. They also lack the power and flexibility provided by commercial EDA tools.

Another challenge to design asynchronous circuits is guaranteeing some minimal throughput operating point. This throughput is constrained by the maximum cycle time of the circuit. However, on complex concurrent asynchronous systems cycle time is not trivial to capture. Synchronous circuits typically rely on register transfer level (RTL) models, where the maximum throughput is limited by a clock period. This not only makes design capture simpler, but also eases the task of optimising a netlist, as every timing path has a same, fixed maximum delay constraint, the clock period. In fact, synchronous RTL models drove decades of development on commercial EDA tools, which provide strong means for designers to explore power, performance and area optimisation in modern technologies. These means are nonetheless very specific, and efforts to abandon the synchronous paradigm in exchange for more powerful design techniques can easily make commercial tools not applicable. Accordingly, the support for asynchronous design lags behind and, as technologies get less predictable and wire dominated, there is a particular need for new solutions that allow asynchronous circuit optimisation after technology mapping and during physical design.

An alternative that allows the use of commercial EDA tools started to be explored by works that proposed the WHCB pseudo-synchronous design flow [TBV12] and the SDDS-NCL design flow [MTMC14, MNM+14, MBSC18]. These approaches demonstrated advantages over specialised tools. They are further described respectively in Section 3.2 and in Section 3.1.

1.3 Contributions of this Work

This dissertation presents Pulsar, an innovative synthesis flow for QDI circuits. Pulsar enables the design of asynchronous circuits from a RTL-alike description under cycle-time constraints. Pulsar integrates the pseudo-synchronous and the SDDS-NCL design flows to enable technology mapping and optimisation of sequential SDDS-NCL circuits with commercial EDA tools. It proposes a timing model that enables the cycle time analysis and constraining using standard STA tools. Pulsar also leverages EDA tools for the design capture and dual-rail expansion of QDI circuits.

The Pulsar Flow enables the sign-off of target cycle times for QDI circuits using commercial EDA tools. This is a major breakthrough for QDI designers, as they can now safely bound worst case performance metrics for their target applications. Moreover the flow enables designers to naturally trade performance for power or area optimisations, whenever there is slack in timing budgets.

This work has six main original contributions:

1. Extends the pseudo-synchronous design technique, introducing new registers models.
2. Generalises the SDDS-NCL flow to deal with both, combinational and sequential designs.
3. Proposes the half-buffer channel network (HBCN), a new model for timing analysis of half-buffer asynchronous circuits.
4. Devises linear programming formulations to constrain an asynchronous design to a target cycle time.
5. Introduces an RTL-like design capture technique, associated dual-rail expansion and synthesis flow that leverages commercial EDA tools.
6. Creates a tool that automatically extracts the HBCN model of a circuit and computes synthesis constraints that meet a target cycle time.

Chapter 3 describes in some detail contributions 1 and 2. Contributions 3 and 4 are the target of Chapter 4. The last two original contributions (Contributions 5 and 6) are explored as part of the Pulsar flow description, in Chapter 5.

2. BACKGROUND

This Chapter provides an introduction to some topics required to read this dissertation. It also establishes some of the terminology used throughout the work. Sections 2.1 provide formal definitions for Petri nets and marked graphs used throughout this work, which are the basis for the HBCN model proposed herein. Sections 2.2 and 2.3 provide an overview of asynchronous circuits, especially QDI circuits. They also provide definitions for the RTO and RTZ handshake protocols, extensively employed in this Dissertation. Finally, Section 2.4 provides and overview of Static Timing Analysis, a fundamental concept for dealing with (asynchronous) cycle time constraints and their computation.

2.1 Graphs and Petri Nets

Most definitions used in this work rely on or derive from the fundamental concept of *graphs*, or more specifically on *directed graphs*. Accordingly, a precise definition of this concept is provided here, based on classical texts definitions such as the one provided by Cormen et al. [CLRS09].

Definition 2.1.1 (Directed Graph) *A* **directed graph (or digraph)** *G is a pair $G = (V, E)$, where V is a finite set and E is a binary relation on V. The set V is called the* **vertex set** *of G, and its elements are called* **vertices** *(singular:* **vertex***). The set E is called the* **edge set** *of G, and its elements are called* **edges***.*

Given a vertex $v \in V$ of a graph $G = (V, E)$, the subset of V with the form $\{w | w \in V \wedge (w, v) \in E\}$ is called the **preset** *of vertex v. Accordingly, given a vertex $v \in V$ of a graph $G = (V, E)$, the subset of V with the form $\{w | w \in V \wedge (v, w) \in E\}$ is called the* **postset** *of vertex v.*

Unless otherwise noted, in this work all references to graphs refer to directed graphs and the word *directed* is omitted. Note that the previous definition includes describing the predecessor and successor vertex sets in graphs, a concept very important for more elaborate structures used herein. Graphs are generic structures that can be specialised to address more specific modelling needs. One such specialisation relevant here is that of *bipartite graphs*.

Definition 2.1.2 (Bipartite Graph) *A* **bipartite graph** *is a directed graph $G = (V, E)$ where the set V is in the union of two sets, $V = W \cup X$ and where E is formed by edges having exactly one element from W and one element from X, i.e. $E \subseteq \{(a, b) | ((a \in W) \wedge (b \in X)) \vee ((a \in X) \wedge (b \in W))\}$.*

2.1.1 Petri Nets

Often, the modelling of asynchronous circuits relies on Petri nets, whose static structure can be captured by graphs. The next definitions formalise the general concept of Petri nets and particularise it to more specific forms useful in asynchronous circuit modelling.

Note that a Petri net has a static structure, an initial marking and a marking evolution behaviour, the two later ones encompassing the dynamics of the net. The definition covers all parts of the concept, and is based on [Mur89].

Definition 2.1.3 (Petri Net (PN)) *A* **Petri net** *is a* 5*-tuple* $PN = (P, T, F, W, M_0)$ *where:* $P = \{p_1, p_2, \dots, p_m\}$ *is a finite set of* **places***,* $T = \{t_1, t_2, \dots, t_n\}$ *is a finite set of* **transitions***,* $F \subseteq (P \times T) \cup (T \times P)$ *is a set of* **arcs***, collectively called the* **Petri net flow relation***,* $W : F \longrightarrow \mathbb{N}^*$ *is the* **weight** *function, and* $M_0 : P \rightarrow \mathbb{N}$ *is the* **initial marking***, with* $P \cap T = \emptyset$ *and* $P \cup T \neq \emptyset$*. The* **Petri net structure** *is the 4-tuple* $N = (P, T, F, W)$ *with no consideration of marking. A Petri net with a given initial marking can be alternatively denoted by* (N, M_0)*.*

The **behaviour** *of a Petri net relies on a set of rules that dictate how a* **marking** *or* **state** *evolves into another state, according to the following set of* **firing rules***:*

1. *A transition* t *is said to be enabled if each input place* p *of* t *is marked with at least* $w(p, t)$ *tokens, where* $w(p, t)$ *is the weight of the arc from* p *to* t*;*
2. *An enabled transition may or may not fire, depending on whether or not the event actually takes place;*
3. *A firing of an enabled transition* t *removes* $w(p, t)$ *tokens from each input place* p *of* t*, and adds* $w(t, p)$ *tokens to each output place* p *of* t*, where* $w(t, p)$ *is the weight of the arc from* t *to* p*.*

A transition without any input place is called a **source transition***, and one without any output place is called a* **sink transition***. Note that a source transition is unconditionally enabled, and that the firing of a sink transition consumes tokens, but does not produce any.*

A pair of a place p *and a transition* t *is called a* **self-loop** *if* p *is both an input and output place of* t*. A Petri net is said to be* **pure** *if it has no self-loops. A Petri net is said to be* **ordinary** *if all of its arc weights are* 1*.*

It should be clear from the PN definition and from Definition 2.1.2 that the structure N of a PN can be represented by a bipartite graph where the vertex set V of the graph is the union of the set of places and of the set of transitions of the Petri net, i.e. $V = P \cup T$. Because of this, it is common and practical to informally state that PNs *are* bipartite graphs, ignoring the underlying marking and behaviour concepts. A big advantage of treating a PN

as a graph is inheriting to PNs all graph concepts, e.g. vertex degrees, vertices (places or transitions) presets and postsets, etc. Where precision is not compromised, this document adopts this little abuse.

2.1.2 Petri Net Properties

A large set of behavioural properties derives from the definition of a PN; Techniques to analyse PN instances for such properties abound in the literature. This Section explores PN properties specifically relevant to this work. The interested reader can refer to [Mur89] or to PN books such as [Rei13] for a more complete discussion of PN properties.

According to Murata [Mur89], there are two types of PN properties: those that depend on the initial marking M_0, called **behavioural properties**, and those independent of M_0, called **structural properties**. This works addresses only some of the behavioural properties.

A first important property is **reachability**. This is a fundamental property to study the dynamic properties of any system described by PNs. Reachability relies on the PN´s *firing rules* and on the initial marking M_0 of a PN. A marking M_n is said to be reachable from marking M_0 if there exists a sequence of firings that transforms M_0 into M_n. A firing or occurrence sequence is denoted by $\sigma = M_0 t_1 M_1 t_1 M_2 \ldots t_n M_n$, or simply $\sigma = t_1 t_1 t_2 \ldots t_n$. In this case, M_n is reachable from M_0 by σ and we write $M_0[\sigma > M_n$. The set of all possible markings reachable from M_0 in a PN (N, M_0) is denoted by $R(N, M_0)$, or simply $R(M_0)$. The set of all possible firing sequences from M_0 in a PN (N, M_0) is denoted by $L(N, M_0)$, or simply $L(M_0)$. The **reachability problem** for PNs is the problem of finding, for a given marking M_n, if $M_n \in R(M_0)$ in a PN (N, M_0). Sometimes it is interesting to define the **submarking reachability problem**, where instead of a PN marking M_n attentions is restricted to M'_n, a marking limited to just some subset of places of P.

A second property worth defining here is **boundedness**, related to the maximum amount of tokens a place of some PN holds. A PN (N, M_0) is said to be k-bounded or simply bounded if the number of tokens in each place does not exceed a finite number k for any marking reachable from M_0, i.e. $M(p) \leq k$ for every place p and every marking $M \in R(Mo)$. A PN (N, M_0) is said to be **safe** if it is 1-bounded. In (asynchronous) hardware design places of a PN are often used to represent buffers and registers for storing intermediate data. By verifying that the net is bounded or safe, it is guaranteed that there will be no overflows in buffers or registers, no matter what firing sequence is taken.

A last property needed to define in this work is **liveness**, associated to concepts in system design like absence of deadlocks. A PN (N, M_0) is said to be **live** (or equivalently M_0 is said to be a live marking for N) if, no matter what marking has been reached from M_0, it is possible to ultimately fire any transition of the PN by progressing through some firing

sequence. This means that a live PN guarantees deadlock-free operation, no matter what firing sequence is chosen.

2.1.3 Marked Graphs

Marked graphs constitute a limited class of PNs that allows modelling concurrency, but not choice (to avoid e.g. non-determinism). This enables capturing the behaviour of handshaking circuits. A timed marked graph can capture not only the inter-dependency and concurrency in asynchronous circuits, but also the timing of certain events occurring within the circuit.

As a type of Petri net, marked graphs are bipartite graphs. Informally, a marked graph is a Petri net where each place has exactly one transition in its preset and exactly one transition in its postset. Also, a marked graph is guaranteed to be a safe Petri net, meaning that places can hold at most one token at any moment in time. These characteristics enable deriving a simple definition for a marked graph.

Definition 2.1.4 (Marked Graph) *A marked graph is a 3-tuple* $MG = (T, P, M_0)$*, where* T *is the set of transitions,* $P \subset \{(u, v) : u, v \in T\}$ *is the set of edges connecting transitions, and* $M_0 \subset P$ *is the subset of edges initially marked (the initial marking). The marking* $M_i \subseteq P$ *corresponds to the the subset of places holding tokens at some given instant* $i \in \mathbb{N}$*. Of course,* $i = 0$ *corresponds to the initial state of MG, where the initial marking* M_0 *in in place.* M_i *represents the state of MG at instant* i*.*

Compared to a regular PN, MGs suppress the representation of places but marks still occupy their position, between transitions. In a marked graph, token movements (i.e. state changes) obey a deterministic causality relation formally defined, in this work called *token flow*.

Definition 2.1.5 (Token Flow) *Let* M_i *be the marking of a marked graph MG at instant* $i \in \mathbb{N}$*. Let* $\bullet t = \{(u, v) \in P | v = t\}$ *be the preset of transition* t *and let* $t\bullet = \{(u, v) \in P | u = t\}$ *be the postset of transition* t*.*

Then, it is true that $\exists t \in T : \bullet t \subseteq M_i \implies \exists n \in \mathbb{N}_* : t\bullet \subseteq M_{i+n} \wedge \bullet t \nsubseteq M_{i+n}$*. This means that if all elements in the preset of a transition (*$\bullet t$*) are marked, there is a moment in the future (*$i + n$*) where all elements in the postset of that transition (*$t\bullet$*) will be marked and its preset* $\bullet t$ *will be unmarked.*

Transitions control the flow of tokens in a network through the firing process. As defined for any PN (MGs obviously included), when a transition fires, it removes tokens from all places in its preset and deposits tokens to all places in its postset. Said otherwise, when a

transition fires it simultaneously marks its postset and unmarks its preset. An MG transition can only fire after all edges in its preset are marked. Further limitations can be imposed as to when a transition is allowed to fire (including time counting and other conditions).

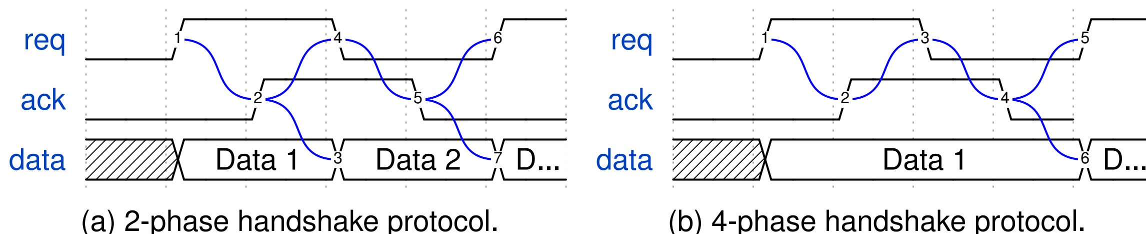
An useful extension of the MG concept is adding labels to account for time in either places and/or transitions, giving rise to the *timed marked graphs* or TMGs. The former is relevant to this work and is accordingly precisely defined.

Definition 2.1.6 (Place-Delayed Marked Graph) *Given a marked graph MG, it is possible to define a* **place-delayed marked graph** *as a 3-tuple PDMG* $= (T, P, M_0)$*, where* T *and* M_0 *are defined as in the corresponding MG and* $P \subset \{(u, v, d)|u, v \in T, d \in \mathbb{R}+\}$*. The edges, as in an MG, connect transition* u *to transition* v*, with a label* d*, representing the delay assigned to each edge.*

A token flowing into a PDMG edge (u, v, d) experiences a delay d before enabling the firing of a transition. That is, once receiving a token, an edge (u, v, d) must remain marked at least for the duration d before the token is removed. As in any PN, a transition can only fire in a PDMG after all its predecessor edges are holding tokens. When a transition fires it removes the tokens from its predecessor edges and deposits tokens in all of its succeeding edges.

2.2 Handshake Protocols

Handshake protocols are characterised by two distinct steps: (i) **request**, when a transmitter announces data availability; (ii) data **acknowledgement**, when a receiver acknowledges data reception, allowing the transmission of new data. The implementation of these steps depends on a series of choices for specific actions on control signals. Figure 2.1 displays two possible handshake protocols performing complete handshake cycles.



(a) 2-phase handshake protocol. (b) 4-phase handshake protocol.

Figure 2.1: Examples of handshake protocols: (a) 2-phase; (b) 4-phase.

Figure 2.1a presents a 2-phase, also known as edge-sensitive handshake protocol. In this protocol data availability and acknowledgement are determined by transitions ("edges") in control signals. It implements the request and acknowledgement steps in a cycle with 2 phases: (i) Data are placed in the *data* lines and, at the same time or immediately

afterwards a transition occurs on the *req*, announcing data availability; (ii) The transition of the **req** line triggers the consumer to capture the data, later acknowledging reception by transitioning the **ack** line.

Figure 2.1b presents an instance of a 4-phase, also known as level-sensitive, handshake protocol. In this protocol data availability and acknowledgement are indicated by logic levels in control signals. Handshakes take place in a cycle with two parts: (i) evaluation, when handshake signals are asserted; and (ii) reset, when the signals return to their inactive state before the next evaluation part starts. This results in data-exchange cycle with 4 phases: (i) data is placed in the *data* lines and, at the same time or immediately afterwards, the *req* line is raised to announce data availability; (ii) the consumer acknowledges the data arrival by raising the *ack* line; (iii) after recognising the rise of the *ack* signal, the producer resets the *req* signal; (iv) the consumer recognises the fall of the *req* signal by lowering the *ack* signal, thus completing the protocol reset stage and allowing a new transmission to take place.

The use of dedicated request/acknowledge signals separate from the data lines characterises what is traditionally called **Bundled Data** (BD) design style. The BD style allows simpler data path implementations, at the expense of timing constraints complexity. Since combinational logic transforming data must be transparent to the local handshake protocol [SF01], the request line must arrive to the consumer only after all computations on channel data is concluded, otherwise the consumer sequential barrier may capture incorrect data.

This poses a design challenge, as the request line may be required to be delayed with regard to data availability, to guarantee correct synchronisation. Such synchronisation is often achieved in the BD design style using delay-lines that match the delay of combinational logic in the data lines to the *req* logic generation circuit.

As an alternative, the request line can be encoded within the data itself, using some delay-insensitive encoding for data. This approach reduces the complexity of timing constraints and provides a more robust circuit implementation, at the cost of additional area. The next Section further explores this class of circuits, which will be the main target of this work.

2.3 Quasi-Delay-Insensitive Design

Synchronous design relies on the assumption that the value on the inputs of all its registers will only be sampled at the rising (and/or falling) edge of the clock signal. This assumption enables designers to define timing constraints for the maximum delay in logic paths, which must always be lower than the clock period. This allows ignoring gate and wire delays, as long as constraints are respected. In other words, combinational logic is allowed

to switch as it computes data during, say, the interval between two consecutive rising clock edges, but it must be stable and correct at each such edge. Having such a simple model for circuit design is possible only because the clock is a periodic signal, *i.e.* its edges only occur at specific and known points in time. Hence, in synchronous circuits, events will only take place at specific moments; time can thus be treated as a discrete variable.

However, in asynchronous circuits there is no such thing as a single clock to signal data validity on the inputs of all registers. In these, events can happen at any moment, and time must be regarded as a continuous variable. Thus, asynchronous designers rely on local handshake protocols for communication and synchronisation, and on different design templates to build circuits, each with its own specific assumptions about gate and wire delays [BOF10]. These templates can be classified in two main families: bundled-data (BD) [Sut89] and QDI [MN06]. The design of a BD circuit is similar to a synchronous one, the difference is that BD relies on carefully designed delay elements for matching the timing of logic paths and controlling registers, rather than having a clock signal. Communication and synchronisation are accomplished through handshake protocols. QDI, on the other hand, uses special data codes that allow data to carry their own validity information, enabling receivers to compute the presence or absence of data at inputs/outputs, and rendering possible the local exchange of information. Because of this characteristic QDI circuits can adapt more gracefully to wire and gate delay variations.

QDI design relies on delay-insensitive (DI) codes [Ver88], which include codewords for valid data, and a special codeword representing the absence of data. The latter codeword is usually called a **spacer**. Figure 2.2 shows a basic transition diagram for transmitting data on a 1-bit QDI channel. Assume transmission always starts with a spacer (S). A transition from the S codeword to 1 (0) characterises the transmission of a valid 1 (0) and a transition from 1 (0) to S characterises the removal of data. Thus, there is always a spacer between any pair of consecutive data values.

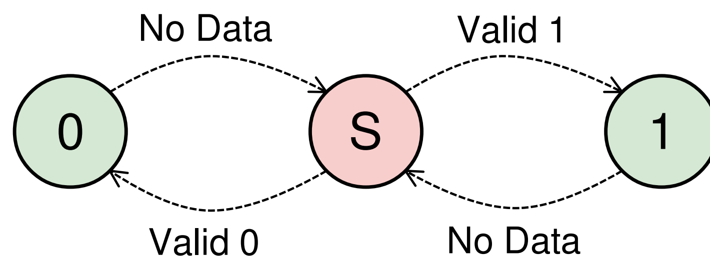
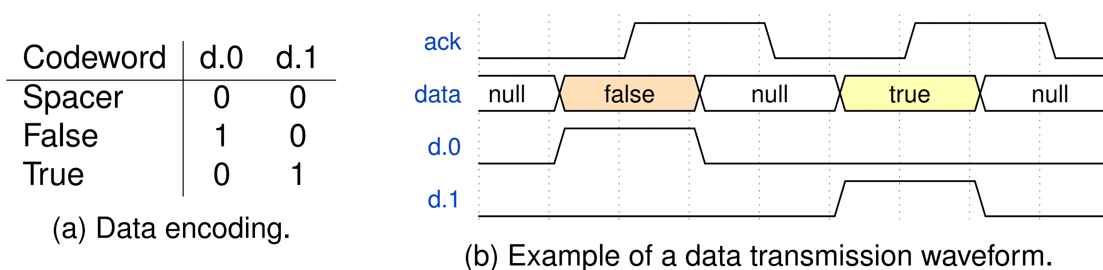


Figure 2.2: The basic transition diagram for transmitting binary data in a QDI channel, where S stands for the spacer. Adapted from [MBSC18].

In circuit design, often used examples of DI codes are the k-of-n codes, where **n** is the number of wires used to represent data (or its absence) and **k** is the number of wires that must be at a given logic value for the codeword to represent valid data. Note that an acknowledge signal is typically required to control the injection of spacers and valid data in a QDI channel. Albeit different codes are available in the contemporary literature (see

e.g. [Ver88]), according to Martin and Nyström [MN06], the most practical class of DI codes is the **1-of-n** (or one-hot), and more specifically the **1-of-2** code. The latter is the basis to form codes to represent any n-bit information using two wires to denote each of the n bits, producing the so-called **dual-rail** code. Besides data encoding QDI design usually implies a choice of either a 2-phase or a 4-phase communication protocol [MN06]. The majority of QDI designs available in the state-of-the-art, from networks-on-chip [BCV+05, PMMC10], to general purpose processors [MNW03], and network switches [DLD+14] primarily rely on a 4-phase protocol and dual-rail or 1-of-4 codes. This work focus attention on 4-phase, dual-rail QDI design.

In a 4-phase dual-rail DI channel *D*, a single bit datum is represented using two signals, *D.1* and *D.0* that carry the datum value, and one signal *ack* to control data flow. For the data portion of a channel, as Figure 2.3a depicts, a spacer is classically encoded as a codeword with all signals at 0. Valid data are encoded using exactly one signal at 1, *D.1*=1 for a logic 1 and *D.0*=1 for a logic 0. In this case, both signals at 1 is a codeword that does not correspond to any valid datum and is not used. Figure 2.3b shows an example of data transmission using this convention to demonstrate the control flow allowed by the *ack* signal combined to codewords represented in signals *D.1* and *D.0*. In this example a sender provides dual-rail data in *D.1* and *D.0* to a receiver that acknowledges received data through *ack*. Communication starts with a spacer, all signals at 0. Note that the *ack* signal also starts at 0, as the receiving side is ready to receive new data. Next, the sender puts a valid 0 bit in the channel, by raising the logic value of *D.0*, which is acknowledged by the receiver raising the *ack* signal. After the sender receives *ack*, it produces a spacer to end communication, bringing all data signals in the channel back to 0. The receiver then lowers its *ack* signal, after which another communication can take place. Due to its nature, which requires all signals to go to 0 before each data transmission, this 4-phase protocol is also known as return-to-zero (RTZ).



Codeword	d.0	d.1
Spacer	0	0
False	1	0
True	0	1

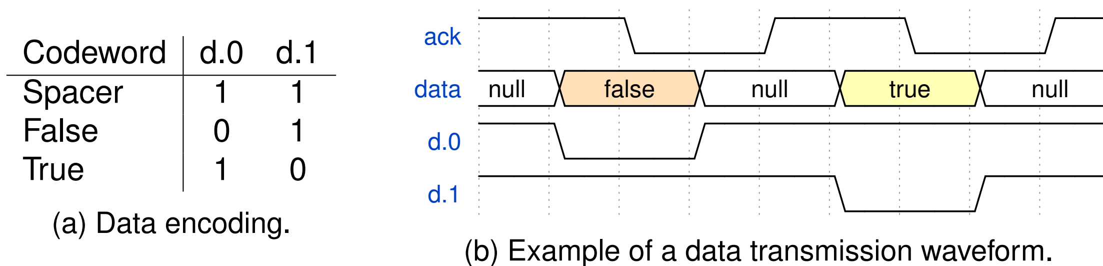
(a) Data encoding.

(b) Example of a data transmission waveform.

Figure 2.3: RTZ dual-rail channel operation: (a) encoding; (b) example waveforms. Adapted from [MBSC18].

Another 4-phase protocol for dual-rail QDI design is the return-to-one (RTO) protocol [MGC12]. RTO employs the same amount of valid codewords as an equivalent RTZ, but its data values are inverted compared to the latter. As Figure 2.4a shows, a spacer here is the codeword with all signals at 1 and valid data is represented by one signal at 0, *D.1*=0 for a logic 1 and *D.0*=0 for a logic 0. Figure 2.4b depicts an example RTO data transmission,

which starts with all signals at 1 in the data channel. As soon as the sender puts valid data in the channel, the receiver may acknowledge it by lowering *ack*. Next, all data signals must return to 1 to denote a spacer, ending transmission. When the spacer is detected by the receiver, it raises the *ack* signal and new data can follow. The idea behind the RTO protocol is simple but powerful and allows a better design space exploration for QDI circuits, enabling optimisations in power [MGC12] and robustness [MGHC14]. Furthermore, as demonstrated in [MPC14], RTZ and RTO can be mixed in a same QDI design and the conversion of values between them requires only an inverter per wire. According to Martin and Nyström, in [MN06], such conversion is DI and does not compromise the robust functionality of a QDI circuit. This article refers to signals operating under the RTZ (RTO) protocol as RTZ (RTO) signals.



Codeword	d.0	d.1
Spacer	1	1
False	0	1
True	1	0

(a) Data encoding.

(b) Example of a data transmission waveform.

Figure 2.4: RTO dual-rail channel operation: (a) encoding; (b) example waveforms. Adapted from [MBSC18].

2.4 Static Timing Analysis

Timing analysis is an important step in digital circuit design, since establishing the circuit performance under certain operating conditions is often one of the main project requirements. Historically timing is analysed using electrical simulation, this is called Dynamic Timing Analysis (DTA). It relies on stimulating every possible path of electrical-level model of the circuit being tested and measuring the time elapsed until a stable output is given. However, this type of analysis is computationally expensive and relies on a good source of stimuli to cover every possible path.

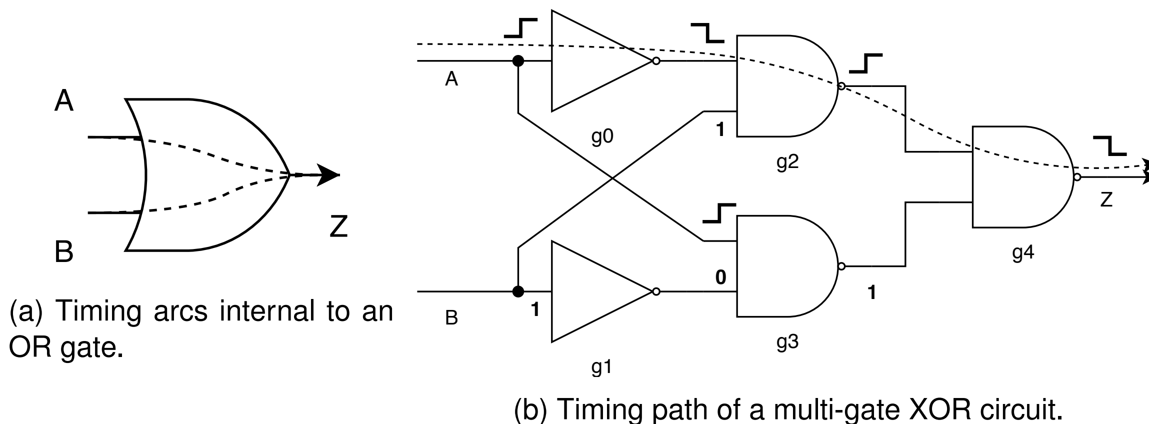
Alternatively, Static Timing Analysis (STA) provides a cheaper alternative to analysing the timing of the circuit globally through simulation. It relies on delay models of small components of the circuit, e.g. logic gates. These delays models are used to estimate the delay of every possible timing path in the circuit. This avoids the coverage problem and is less computational intensive, allowing the use of STA on much larger circuits.

The delay models of components are created using electrical simulation in a process called characterisation. A delay model of a component captures characteristics of arcs. An arc is a timing path internal to a component, it captures how changes in a input affects

the circuit. There are three types of arcs: (i) hidden arcs, where a change in input does not affect any output; (ii) transition arcs, where a change in input causes a transition in the output; and (iii) constraint arcs, which are limitations of when an input can change. As an example, the propagation arcs of an OR gate are shown in Figure 2.5a. Each dashed line represent two propagation arcs, one for the rise and another for the fall of the output Z.

A possible model is the non-linear delay model (NLDM), which is composed of tables that capture characteristics for each arc based on the input slew and output capacitance. Some characteristics captured by the NLDM tables are: (i) slew rate, the time an output takes to finish transitioning once it starts; (ii) propagation delay, the time a transition in an input propagates to the output ; (iii) power consumption, the energy consumed by a transition in the arc. The delay model of a component also captures the capacitance of pins.

The timing paths are acyclic with well defined start and end points. A timing path is composed by a chain of arcs. Transitions propagate in these paths thru arcs, due to this characteristic a timing path is also called a **propagation path** throughout this work. Figure 2.5b shows a circuit implementing a XOR gate using two inverters and three NAND gates. The timing arc relating the fall of the output Z when the input A rises is highlighted. A rise transition in A activates an arc of $g0$ which produce a fall transition in its output, in turn this transition activates an arc of "g2" that causes a rise in its output, which finally activate an arc of $g4$ that cause a fall transition in the primary output Z.



(a) Timing arcs internal to an OR gate.

(b) Timing path of a multi-gate XOR circuit.

Figure 2.5: Timing models used by the STA engine: (a) gate timing arcs; (b) timing paths.

From the information contained in the delay model, it is possible to analyse the timing paths of circuits. The STA engine computes the delay of the timing paths with the following sequence of steps: First the capacitance of each net is computed based on the components connected to it, the capacitance of primary outputs are provided as a parameter. The STA engine then creates an acyclic directed graph containing all possible timing paths in the circuit from the circuit netlist and propagation arcs. For each arc in the graph, the STA engine uses the slew information in the delay model to compute the slew rate of every possible transition, the slew of the primary inputs are provided as a parameter. Once

the capacitance and slew rate are computed, the STA engine is able to use the propagation arcs of the delay models to compute the time each transition occurs in the timing path.

On synchronous sequential circuits, the timing paths are bounded by clock events. The start point of each path is either a primary input or the output of a clocked sequential element. Also, the end point is either an primary output or the input of a clocked sequential element.

The clock provides a periodic time reference which can be used to compute the timing of the circuit. Primary inputs are assumed to be synchronous to clock events and sequential elements produce new values on clock events. This implies that a clock event triggers a sequence of transitions in propagation paths. The time between the first and the last transition in the propagation path is called its propagation delay of that path.

A more comprehensive background on STA lays beyond the scope of this Dissertation. The interested reader may consult works like that of Harris “Skew-Tolerant Circuit Design” [Har01].

3. THE PSEUDO-SYNCHRONOUS SDDS-NCL DESIGN TEMPLATE

The timing of propagation paths in a digital circuit can be computed using STA. Commercial EDA tools attempt to find and optimal solution that respects the timing budget while balancing power and area consumption. Also, during physical implementation, STA enables an optimal placement of gates in the layout, guaranteeing that wire delays do not compromise the circuit functionality. On synchronous flop-based circuits, the clock period bounds the delay of propagation paths between two registers. The last transition in each path must occur a certain time prior to the clock edge for the circuit to work correctly.

The pseudo-synchronous flow enables us to define a pseudo-clock period, analogous to a clock period of a synchronous flop-based circuit. This pseudo-clock can then be used to constrain the delay of propagation paths between two registers. The SDDS-NCL design template leverages standard EDA to synthesise and optimise combinational QDI circuits. This Chapter introduces the Pseudo-Synchronous SDDS-NCL, a QDI template that combines and extends both SDDC-NCL and the Pseudo-Synchronous flows. This enables the synthesis and optimisation of sequential asynchronous circuits with constrained propagation paths using commercial EDA tools.

3.1 The SDDS-NCL Asynchronous Design Template (Related Work)

NCL is an asynchronous design template to construct QDI circuits – Nowick and Singh present a good summary of this and other templates in [NS15a, NS15b]. Fant and Brandt proposed NCL in [FB96], targeting RTZ 1-of-n schemes. It was the technology used for industrial designs produced by Theseus Logic in the late 90s. Since then, NCL has been explored by different research groups and employed for different applications, typically focusing on low power and robust design [JSL+10, CCGC10, SCH+11, ZSD10, GLY10]. NCL assumes a semi-custom design method, and relies on a basic set of components designed at the cell level, called *NCL gates*. As Fant and Brandt discuss in [FB96], these gates allow the construction of logic blocks that ensure the *completeness of input* criterion, which enforces that a logic block only produces a spacer (valid data) in its outputs when all its inputs have a spacer (valid data). The authors state that this is the key to ensure DI operation.

NCL gates are often called *threshold gates*, but this is imprecise, because they do not exactly implement threshold logic functions (TLFs) as defined, e.g. in [Hur69]. Rather, these gates implement modifications of such TLFs coupled to specific mechanisms to ensure the completeness of input criterion [FB96]. Before defining TLFs, a useful concept is that of unate functions [BSVMH84], as all TLFs are unate [Hur69]:

Definition 3.1.1 *A Boolean function $f(x_1, x_2, ..., x_n)$ is said to be* **positive unate** *in x_i ($1 \leq i \leq n$), if $\forall x_j$, $i \neq j$, $f(x_1, \ldots, x_{i-1}, 1, \ldots, x_n) \geq f(x_1, \ldots, x_{i-1}, 0, \ldots, x_n)$. Similarly, f is called* **negative unate** *in x_i if $\forall x_j$, $i \neq j$, $f(x_1, \ldots, x_{i-1}, 0, \ldots, x_n) \geq f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, ..., x_n)$. If a function is neither positive unate nor negative unate in x_i, it is said to be* **binate** *in x_i. Moreover, if a function is positive or negative unate in all its variables it is called a* **unate function***. A unate function that is positive (negative) unate in all its variables is called a* **positive** *(***negative***)* **unate function***. Also, this dissertation calls* **positive** *(***negative***)* **unate gates** *those that implement positive (negative) unate functions.*

Definition 3.1.2 *An n-variable* **threshold logic function (TLF)** *τ is an n-variable unate function defined by a* **threshold value** *$T \in N^*$ and* **weights** *$w_i \in N^*$ assigned to each of its variables x_i such that:*

$$\tau = \begin{cases} 1, & \sum_{i=0}^{n-1} w_i x_i \geq T \\ 0, & \sum_{i=0}^{n-1} w_i x_i < T \end{cases} \tag{3.1}$$

Hurst [Hur69] defines threshold gates with the threshold and weights being real numbers. This work restricts attention to non-zero, natural threshold and weights. TLFs can be either negative or positive unate in a given input – but not binate. However, to ensure the completeness of input criterion, NCL gates must be positive unate in all their inputs, as they target RTZ templates, where data validity is given by wires at 1. The OFF-set of a logic gate is the set of input patterns that force its output to 0. Analogously, the ON-set is the set of input patterns that force the output of a logic gate to 1. Because the spacer corresponds to all wires at 0, the OFF-set of an NCL gate includes just the condition where all inputs are 0.

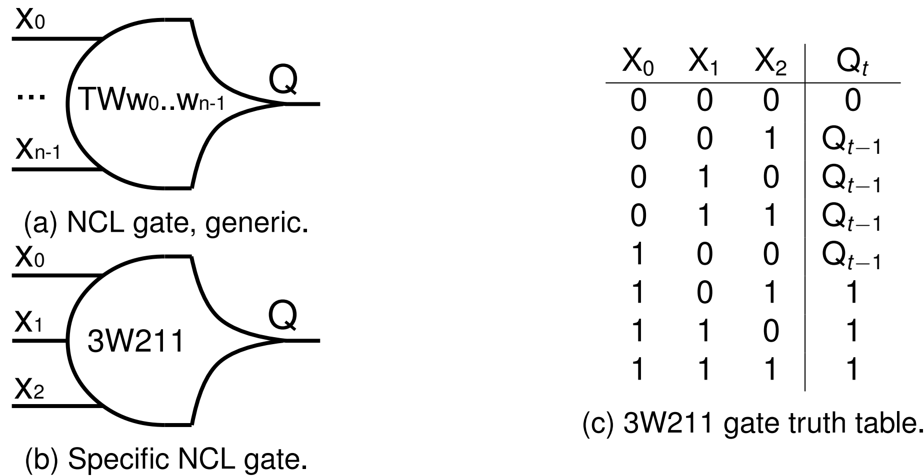
As Fant and Brandt discuss in [FB96], while a QDI logic block is transitioning between a spacer and valid data, output values of the block should be either a spacer or valid data. Therefore, NCL gates must also account for situations where an input combination is neither in the ON-set nor in the OFF-set. In these no gate output should transition. This leads to the definition of the correct behaviour for NCL gates.

Definition 3.1.3 *An n-input* **NCL gate** *is a logic gate with a threshold value $T \in N^*$, a weight $w_i \in N^*$ assigned to each variable x_i ($i = 1, \ldots, n$), and a hysteresis mechanism such that the gate output Q at each instant of time t is given by:*

$$Q_t = \begin{cases} 1, & \sum_{i=1}^{n} w_i x_i \geq T \\ 0, & \sum_{i=1}^{n} x_i = 0 \\ Q_{t-1}, & 0 < \sum_{i=1}^{n} w_i x_i < T \end{cases} \tag{3.2}$$

Figure 3.1a shows a generic NCL gate symbol, where *n* is the number of inputs of the gate and T is the threshold of the TLF it implements, for which each input has a weight

w_i. If a weight w_i is omitted, w_i = 1 is assumed. Weights come after the W specifier. As an example, Figure 3.1b shows the symbol of a 3-input NCL gate with threshold 3 and weights 2, 1 and 1 (in the order from the topmost input down). Figure 3.1c shows the truth table for this latter example, computed from Equation (3.2). Accordingly, the output of the gate only switches to 0 when all inputs are at 0. Also, because x_1 has weight 2, x_2 and x_3 have weight 1 and the threshold is 3, the gate only switches to 1 when x_1 is at 1 and at least one of the other inputs is at 1. In all other cases the output remains unchanged.



(a) NCL gate, generic.

(b) Specific NCL gate.

X_0	X_1	X_2	Q_t
0	0	0	0
0	0	1	Q_{t-1}
0	1	0	Q_{t-1}
0	1	1	Q_{t-1}
1	0	0	Q_{t-1}
1	0	1	1
1	1	0	1
1	1	1	1

(c) 3W211 gate truth table.

Figure 3.1: Characteristics of NCL gates: (a) generic symbol; (a) generic symbol; (b) specific symbol example; (c) example truth table behaviour.

It is possible to build classic Boolean logic blocks with NCL gates, as Figure 3.2 illustrates. This schematic shows a generate path circuit fragment for an RTZ dual-rail Kogge-Stone adder, taken from [MTMC14]. Note that this circuit only generates a spacer output when all dual-rail inputs (G_i, P_i, G_p) have spacers. Recalling Figure 3.1, the output of an NCL gate only goes to 0 when all its inputs are at 0, which means that gates G0-G3 in Figure 3.2 only write a 0 in their outputs when all primary inputs have spacers. Also note that the outputs become valid data only when all inputs display valid data. Accordingly, $n0$ is 1 only if P_i and G_p are valid 1s (i.e. $P_i.1$ = 1 and $G_p.1$ = 1) and $n1$ is 1 only if P_i has a valid 0 and G_p has a valid 1, P_i has a valid 1 and G_p has a valid 0, or both P_i and G_p have a valid 0. In this way, $n0$ and $n1$ are valid only if the primary inputs P_i and G_p are valid. Similarly, gates G2 and G3, which generate the outputs of the logic block, have their inputs connected to $n0$, $n1$ and primary input G_i, and these gates only generate valid data when their inputs are valid.

Even though the classic definition of an NCL gate relies on TLFs for their ON-set, different works (e.g. [JN08, RST12]) report the usage of non-TLF positive unate functions. This work also considers such functions, but as exceptions to the definition. Since these do not agree with the given NCL gate definition, they are called *hysteretic functions*. Hysteretic functions can be described using three minterm sets: (i) the ON-set contain all minterms that

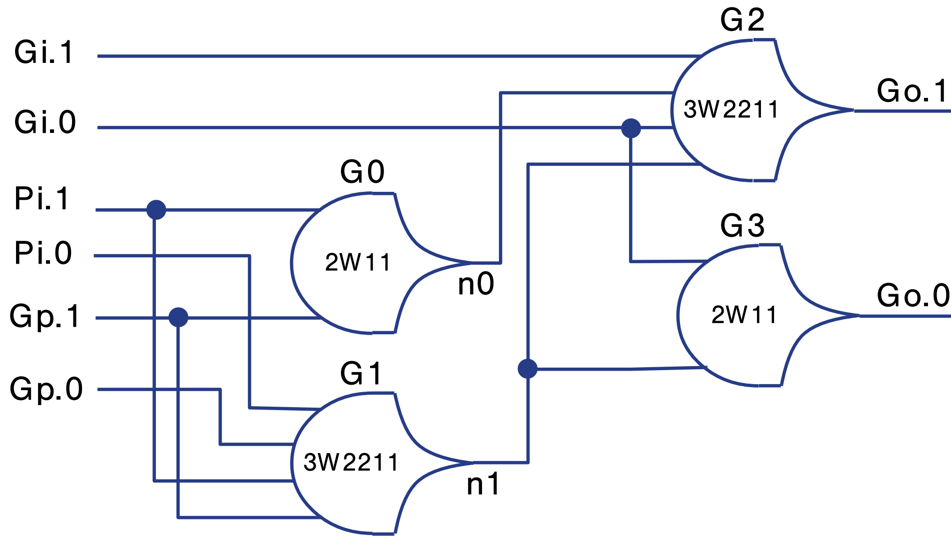


Figure 3.2: Example of an NCL implementation: the generate path of part of a Kogge-Stone adder. Adapted from [MTMC14].

force the output high; (ii) the OFF-set contains all minterms that force the output low;and (iii) the HOLD-set contain all minterms that hold the current output value.

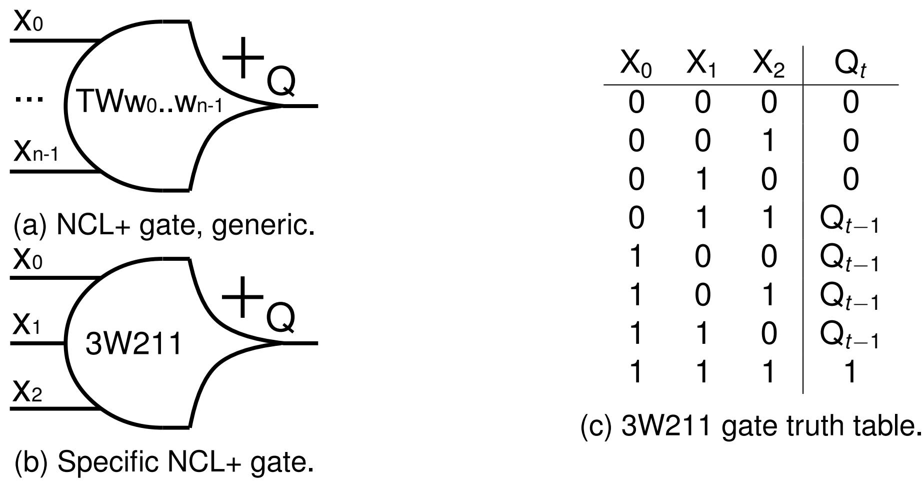
NCL supports the design of RTZ QDI circuits and, to meet the requirements of associated templates, NCL gates are typically restricted to implement only positive unate functions. This limits the potential logic optimisations on NCL circuits and complicates achieving compatibility with conventional EDA tools. However, the introduction of RTO [MGC12] and the NCL+ gates described in [MOPC13] change this situation. NCL+ gates are similar to NCL gates, but, because they target RTO templates, they need to synchronise spacers encoded by all wires at 1 and compute valid data signalled by logic 0s, as Section 2.3 described. The definition of an NCL+ gate is straightforward.

Definition 3.1.4 *An n-input* **NCL+ gate** *is a logic gate with a threshold value $T \in N^*$, specific weights $w_i \in N^*$ assigned to each variable x_i ($i = 1, \dots, n$), and a hysteresis mechanism such that the gate output Q at each instant of time t is given by:*

$$Q_t = \begin{cases} 1, & \sum_{i=1}^{n} \overline{x_i} = 0 \\ 0, & \sum_{i=1}^{n} w_i \overline{x_i} \geq T \\ Q_{t-1}, & 0 < \sum_{i=1}^{n} w_i \overline{x_i} < T \end{cases} \tag{3.3}$$

The symbol of an NCL+ gate is similar to that of an NCL gate, but with a + sign on its top right corner, as Figure 3.3a shows. Figure 3.3b shows an example of an NCL+ gate with 3 inputs, threshold 3 and weights 2, 1 and 1. The truth table of this gate appears in Figure 3.3c. As its truth table shows, this gate output only switches to 1 when all inputs are at 1. Also, its output only switches to 0 when x_1, which has a weight of 2, and at least one of

the other inputs, both with weight 1, are at 0. For all other combinations of inputs, the gate keeps its output value. In these cases, the output only switches to 1 (0) when all inputs are at 1 (0).



(a) NCL+ gate, generic.

(b) Specific NCL+ gate.

X_0	X_1	X_2	Q_t
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	Q_{t-1}
1	0	0	Q_{t-1}
1	0	1	Q_{t-1}
1	1	0	Q_{t-1}
1	1	1	1

(c) 3W211 gate truth table.

Figure 3.3: Characteristics of NCL+ gates: (a) generic symbol; (a) generic symbol; (b) specific symbol example; (c) example truth table behaviour.

As in NCL, NCL+ gates are also useful to build logic blocks. For instance, Figure 3.4 shows the NCL+ version of the same generate path of the Kogge-Stone adder from Figure 3.2. The circuit topology is exactly the same as that of the NCL version. In fact, the gate characteristics are the same. The only difference is that NCL+ gates replace NCL gates with the same number of inputs, thresholds and weights. Hence, all internal nodes and primary inputs and outputs follow the RTO protocol, instead of RTZ.

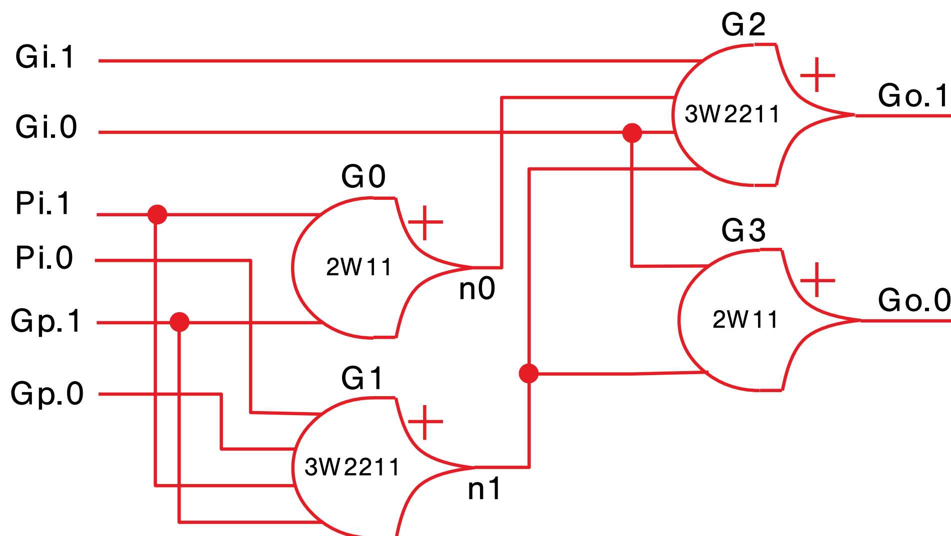


Figure 3.4: Example of an NCL+ implementation: the generate path of part of a Kogge-Stone adder. Adapted from [MTMC14].

The advent of NCL+ created the possibility of mixing NCL and NCL+ in a single circuit, because RTO protocol signals can be translated to RTZ protocol signals and vice-versa

[MPC14]. Recalling Section 2.3, the conversion of a signal from RTO to RTZ, or the other way around, only requires inverters, each of which implements a negative unate function. Thus, as discussed in [MTMC14], the mix of NCL and NCL+ enables the use of negative unate functions, which allows further optimisation opportunities and expands the design space of QDI circuits. For each positive unate NCL (NCL+) gate, a negative unate gate can be defined, where the latter has its OFF-SET defined as the ON-SET of the former. To differentiate negative unate gates from positive unate ones from Definitions 3.1.3 and 3.1.4, these are named *Inverted* NCL and NCL+ gates (or INCL and INCL+, respectively) and defined accordingly.

Definition 3.1.5 *An n-input* **INCL gate** *is a logic gate with a threshold value $T \in N^*$, specific weights $w_i \in N^*$ ($i = 1, \dots, n$), assigned to each variable x_i and a hysteresis mechanism such that the gate output Q at each instant of time t is given by:*

$$Q_t = \begin{cases} 1, & \sum_{i=1}^{n} x_i = 0 \\ 0, & \sum_{i=1}^{n} w_i x_i \geq T \\ Q_{t-1}, & 0 < \sum_{i=1}^{n} w_i x_i < T \end{cases} \quad (3.4)$$

Definition 3.1.6 *An n-input* **INCL+ gate** *is a logic gate with a threshold value $T \in N^*$, specific weights $w_i \in N^*$ ($i = 1, \dots, n$), assigned to each variable x_i and a hysteresis mechanism such that the gate output Q at each instant of time t is given by:*

$$Q_t = \begin{cases} 1, & \sum_{i=1}^{n} w_i \overline{x_i} \geq T \\ 0, & \sum_{i=1}^{n} \overline{x_i} = 0 \\ Q_{t-1}, & 0 < \sum_{i=1}^{n} w_i \overline{x_i} < T \end{cases} \quad (3.5)$$

From a functional point of view, the only difference between an NCL (NCL+) and an INCL (INCL+) gate is that their ON- and OFF-sets are swapped. However, they all still rely on a hysteresis behaviour to ensure the respect of QDI properties. To produce INCL/INCL+ gate symbols, a bubble is added to the output of the respective non-inverted gate symbol. Using inverted gates it is possible to convert signals from RTZ to RTO, and vice versa. Because each time an inverted gate is used the protocol changes (including the codeword to representing spacers), circuits using (I)NCL and (I)NCL+ gates are called *spatially distributed dual spacer NCL* (or SDDS-NCL) [MTMC14].

Figure 3.5 shows an example of SDDS-NCL circuit equivalent to the ones in Figures 3.2 and 3.4. As the figure shows, when the circuit inputs present spacers, it issues a spacer in its output. In this figure, all INCL gates have 1 in their outputs, which means that

the INCL+ gates have 1 in all their inputs and a 0 in their outputs. In other words, all first level wires (in blue) will be at 0 and all second level wires (in red) will be at 1. From this state, whenever the inputs become valid dual-rail data, exactly two of the INCL gates fire, setting their outputs to 0, which causes exactly one of the INCL+ gates to fire, setting its output to 1.

The two inverters *G4* and *G5* (1W1 gates) are required to ensure that all the inputs of gates *G2* and *G3* are in the same domain (RTO). Also, because there are only two levels of logic and all gates are negative unate, the inputs and outputs of this circuit must be RTZ. If the output was expected to be RTO, inverters should be added after gates *G2* and *G3*, or these gates could be mapped to NCL+ ones. In fact, different combinations of NCL, INCL, NCL+ and INCL+ gates can be explored, depending on the requirements for the circuit input and output channels.

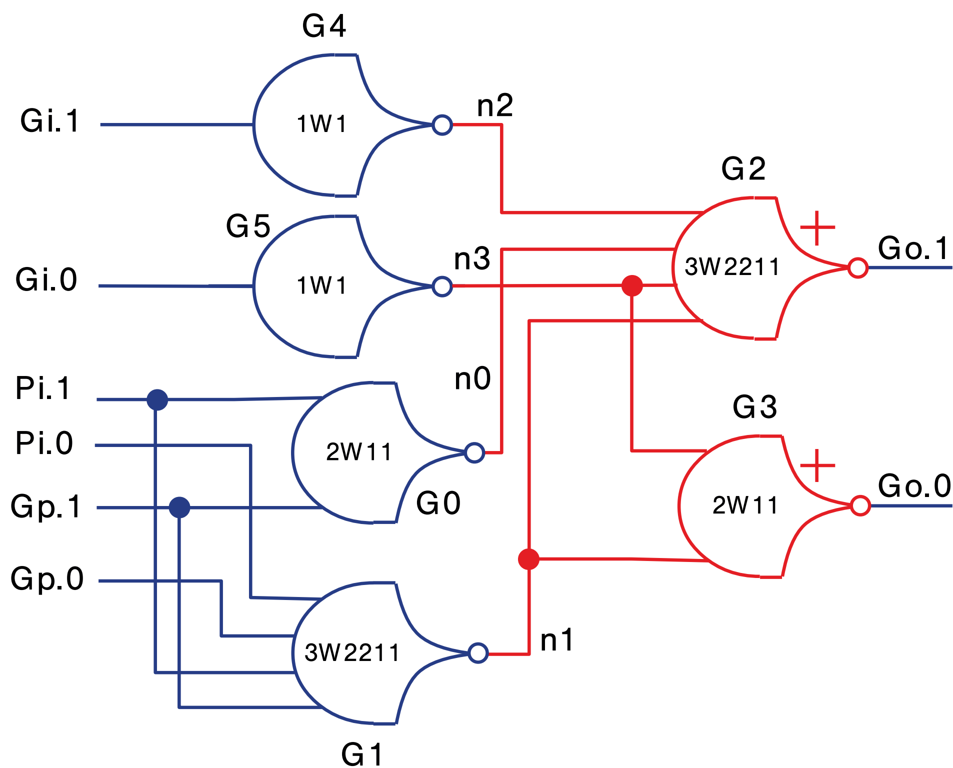


Figure 3.5: Example of an SDDS-NCL circuit: the generate path for a Kogge-Stone adder. Adapted from [MTMC14].

Having available negative and positive unate cells to compose circuits enables the use of commercial tools to perform logic and physical synthesis steps. These tools allow exploring new automation degrees in technology mapping and further logic optimisations. However, conventional CAD tools cannot explicitly handle NCL and NCL+ gates, as these present the hysteresis behaviour, which cannot be naturally handled by such tools. To overcome this problem and leverage the optimisation and automation degrees allowed by EDA tools, [MBSC18] proposes a design flow for circuits based on the SDDS-NCL template. The flow presented by Moreira et al. supports logic blocks only, requiring the designer to split

sequential from combinational blocks and synthesise the latter in isolation. This Chapter later further extends the SDDS-NCL and the pseudo-synchronous [TBV12] design flows to support the fully automated synthesis of both combinational and sequential blocks.

The SDDS-NCL design flow comprises two steps: logic and physical synthesis. Both are based in the usage of *virtual libraries*, a concept similar to that of image libraries proposed for the Proteus asynchronous design flow [BDL11]. The virtual library (VL) concept builds upon the definition of virtual function (VF), presented herein.

Definition 3.1.7 *A* **virtual function (VF)** *is an n-input Boolean function associated with an n-input NCL, NCL+, INCL or INCL+ gate, called its* support gate*. The truth table of a virtual function f is defined as follows:*

1. *if the support gate of f is an NCL gate θ, the ON-set of f is the same as the ON-set of θ. The OFF-set of f comprises all other n-input patterns;*
2. *if the support gate of f is an NCL+ gate ϕ, the OFF-set of f is the same as the OFF-set of ϕ. The ON-set of f comprises all other n-input patterns;*
3. *if the support gate of f is an INCL gate ψ, the OFF-set of f is the same as the OFF-set of ψ. The ON-set of f comprises all other n-input patterns;*
4. *if the support gate of f is an INCL+ gate υ, the ON-set of f is the same as the ON-set of υ. The OFF-set of f comprises all other n-input patterns.*

If the support gate of f is of types NCL or NCL+, f is a positive VF*. Otherwise, f is a* negative VF*.*

Note that each VF always has exactly one NCL and one NCL+ support gates or one INCL and one INCL+ support gates. Reference [MNM+14] provides the detailed description of a method to compute the support gates for VFs. As another consequence of Definition 3.1.7, all positive VFs are positive unate functions, but not all positive unate functions are positive VFs. Furthermore, all negative VFs are negative unate functions, but not all negative unate functions are negative VFs. Hence, all VFs are unate functions, but not all unate functions are VFs. As an example, consider the 3-input NCL gate 3W211 depicted in Figure 3.1. The reader can verify that the virtual function f_1 for this gate can be expressed by $f_1 = x_0.(x_1 + x_2)$. Another example is the 3-input NCL+ gate 3W211 depicted in Figure 3.3 has a virtual function f_2 expressible by $f_2 = x_0 + x_1.x_2$. As expected and can be verified, these VFs are positive unate.

Definition 3.1.8 *A* **virtual library (VL)** *is a library of cells such that their functions are modelled exclusively using VFs. In this way, it is guaranteed that synthesis tools will be able to handle a VL, as all VFs are unate functions. Two types of VLs exist, NCL VLs and NCL+ VLs.*

1. *An* **NCL VL** *is a VL composed exclusively by NCL and INCL gates modelled using VFs.*
2. *An* **NCL+ VL** *is a VL composed exclusively by NCL+ and INCL+ gates modelled using VFs.*

3.2 Pseudo-Synchronous Weak-Conditioned Half Buffer (Related Work)

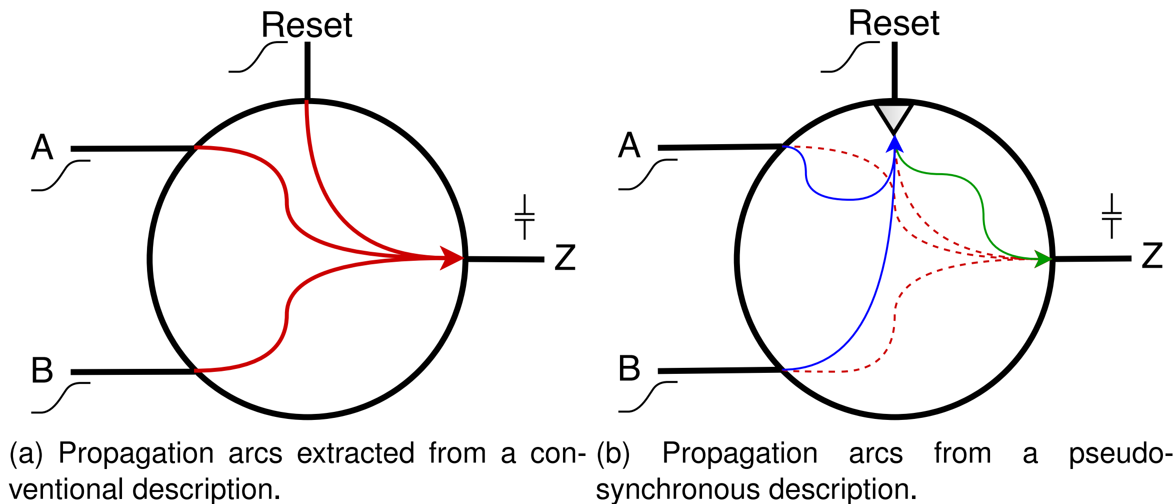
Weak-Conditioned Half Buffer or WCHB is a classical asynchronous QDI template used to build sequential asynchronous circuits [BOF10]. It assumes the use of a DI code (often, the dual-rail one) and allows constructing asynchronous pipelines where stages containing data are surrounded by stages containing spacers and vice-versa (originating the *half buffer* denomination, meaning a data token at each two stages).

In [TBV12] Thonnart et al. compare the timing arcs of a resetable C-element, a basic building block of WCHB pipeline registers, to those of edge-triggered flip-flops, a basic building block of synchronous pipelines. From this analysis arises the proposal of pseudo-synchronous WCHB design technique, which relies on a clever modelling of asynchronous components and on standard static timing analysis (STA) tools to optimise sequential logic specifically for the WCHB QDI design template. They propose a modified, pseudo-synchronous (flop-like) model to represent resetable C-elements, building timing tables in the Open Liberty standard supported by conventional STA tools. The underlying reasoning is that conventional tools can analyse timing paths that start and end only at flip-flops, primary inputs, or primary outputs.

The Thonnart et al. model allows the analysis of paths that start and/or end in pseudo-synchronous flops without adding significant error to the actual delay of these C-elements. This work also details the specification of a pseudo-clock that guides the synthesis decision making process and constrains each logic stage during synthesis. Included is also a methodology to characterise pseudo-synchronous C-elements.

The original timing arcs of a C-element appear in Figure 3.6a, arcs from A, B to Z, and the arc from Reset to Z. These arcs are modelled as a function of the slew from an incoming transition in an input pin and the capacitance being driven by the output pin. In the model devised for use in the pseudo-synchronous flow, the arcs are modified to mimic those equivalent to a flip-flop (as in Figure 3.6b) [TBV12].

Here, the C-element Reset is transformed into a pseudo-clock pin and each of its original propagation arcs are split in two parts: (i) a clock propagation delay (in green), from Reset to Z (a function of the slew in Reset and of the capacitance in Z); and (ii) setup constraints (in blue), from A to Reset (a function of the slew in A and of the slew in Reset) and from B to Reset (a function of the slew in B and of the slew in Reset). Note that, since the sequential nature of the Reset pin is only a mechanism to create the pseudo-clock, its



(a) Propagation arcs extracted from a conventional description.

(b) Propagation arcs from a pseudo-synchronous description.

Figure 3.6: Two arc models for a resetable C-element: (a) conventional description; (b) pseudo-synchronous description [TBV12].

slew is usually ignored. This way, the generated delay tables are a single row (or column) that depends only on the slew of A or B and on the capacitance driven by Z.

The reasoning behind this is that the original arcs can be constructed from the sum of a clock propagation delay and from a setup constraint of the Thonnart et al. model. For example, the original A to Z arc is represented by the propagation delay from Reset to Z added to the setup constraint from A to Reset.

This way, every time the STA tool encounters a pseudo-synchronous flop model of a C-element, it identifies a new start point for paths that begin at output Z and a new end point for paths finishing at inputs A or B. This is crucial to enable STA in WCHB pipelines, as every pipeline stage is marked by the presence of a group of C-elements (e.g. 2 for a 1-of-2, 4-phase encoding, 1-bit data channel). The logic paths between these C-elements are optimised to meet the defined period (say λ) of the pseudo-clock connected to the Reset pin.

For example, Figure 3.7 shows how these paths are analysed in a WCHB pipeline. It shows a single C-element composing the memory of each pipeline stage (hardly true in practice, due to the use of DI codes) and two types of logic paths: (i) the forward logic (usually where useful computation happens); and (ii) the backward logic (usually used for flow control). As the Figure shows, between every pair of C-elements in a cycle there are forward and backward logic paths that are constrained by a single pseudo-clock, with period λ, entering at the Reset pin (not shown). Furthermore, there are also paths between the input channel and the first C-element and the last C-element and the output channel.

One of the major drawbacks of the Thonnart et al. method is that it can cause errors when computing delays for the C-elements, as a result of the regeneration of the original arc from the sum of the two new arcs. This takes place because the new arcs do not measure the delay of the cell as an explicit function of the slew in its inputs and the capacitance in its

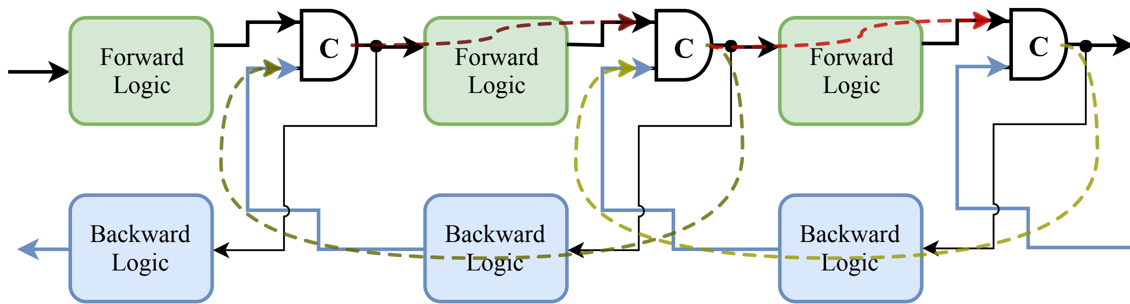


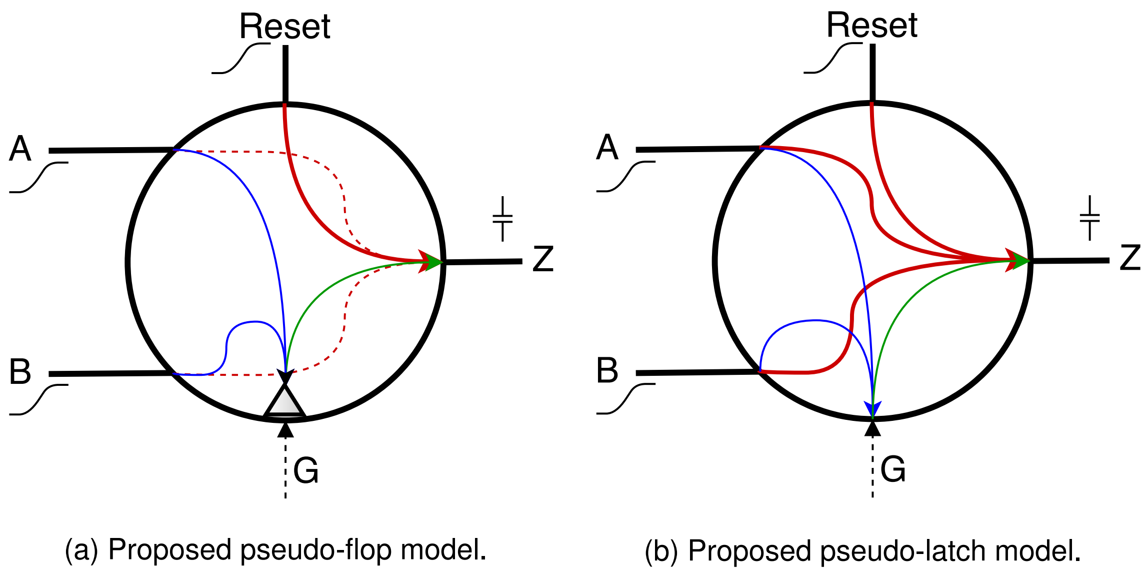
Figure 3.7: Simplified view of a 1-bit data channel WCHB pipeline, showing the inner cycles controlled by the pseudo-clock.

output. It rather builds two independent delay values that are then added, one as a function of the capacitance driven by Z and another as a function of the slew in A or B.

Another drawback is that the reset network is tied to a (pseudo-)clock network and optimising these signals independently becomes challenging, since balancing of reset signals conflicts with the tool trying to synthesise a clock tree for the pseudo-clock. Also, the annotation of delays for post-implementation simulations in this model is tricky. The created arcs will be in accordance to the pseudo-synchronous model, not fitting the original C-element arcs, and will not reflect the real delay of the circuit. Thonnart et al. suggest that the original models be provided to the tool for delay annotation. However, this causes the insertion of loop breakers, which complicates analysis.

To deal with these issues, this work proposes creating a fictitious clock pin (G) and two new models for characterising resetable C-elements, as Figure 3.8 details: the pseudo-flop model (Figure 3.8a), used during synthesis; and the pseudo-latch model (Figure 3.8b) that preserves all original propagation arcs and is only used to extract delay annotations for post-synthesis simulation. The fictitious pin G allows preserving the original Reset pin timing arcs. This enables using STA to design a reset network, meeting the constraints extracted from the cell characterisation process.

The pseudo-latch model relies on the fundamental behaviour of latches. A latch is a sequential element that has two distinctive operation modes, transparent and opaque. During the transparent mode operation, a latch has a direct arc from its data input to its data output and during the opaque mode it holds data. Transitions between these modes are governed by a clock signal, which imposes sequential arcs on the latch. Therefore, a latch timing model typically includes a setup constraint and a clock to output delay (as in flip-flops), and a combinational arc from its input to its output (to serve in those cases when the latch is transparent). This model respectively creates valid start and end points for STA at the outputs and inputs of latches. During the pseudo-latch characterisation, this feature is leveraged to preserve the original arcs of the C-element, while avoiding the insertion of loop breakers. In simulation, it is enough that the behavioural model consider the cell as always in transparent mode to reflect these delays, ignoring any sequential arc.



(a) Proposed pseudo-flop model.

(b) Proposed pseudo-latch model.

Figure 3.8: Proposed characterisation models for a resetable C-element with a dummy clock pin, G: (a) pseudo-flop; (2) pseudo-latch.

Although this work only explores the use of pseudo-synchronous C-elements with reset, the use of a fictitious pin for the pseudo-clock allows any NCL(NCL+) gate to be characterised both using pseudo-flop or pseudo-latch models. Gates characterised using this technique can be modelled as *sequential combo cells*, composed of a combinational gate with the virtual function (VF) of the original gate followed by the sequential element. This allows EDA tools to differentiate pseudo-flop functions during synthesis.

3.3 The Pseudo-Synchronous SDDS-NCL Synthesis Flow

The SDDS-NCL combinational synthesis flow, originally proposed in [Mor16] was significantly extended and modified to produce the SDDS-NCL sequential synthesis flow. Yet the flow described in this Section bears some resemblance to the original one. For example, the use of commercial tools also produces netlists that do not correspond to the intended behaviour of the final circuit (the X-Netlists). As in the original combinational flow, these can be corrected by using a low complexity *Fix X-Netlist* procedure, but the new flow uses a quite distinct approach to implement this procedure, and it is used in more places in the flow.

Figure 3.9 depicts the pipeline structure for a sequential SDDS-NCL circuit. Both forward (in green) and backward (in blue) propagation logic modules are SDDS-NCL. Registers are pseudo-synchronous resetable C-elements. Forward-propagation logic is always positive unate, meaning it preserves the logic protocol of the inputs. Conversely, backward-propagation (completion detection) logic is negative unate, meaning that it inverts the protocol of its inputs. Hence, sequential elements have their inputs at opposite protocol domains.

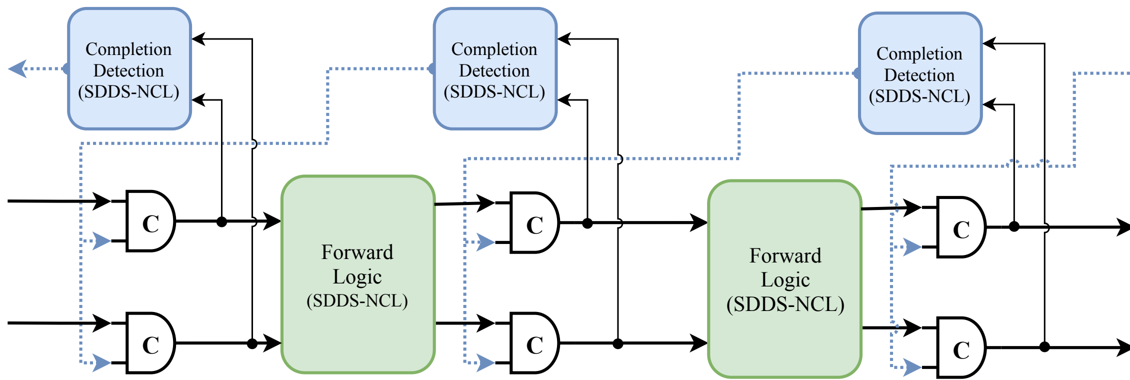


Figure 3.9: Structure of a sequential SDDS-NCL pipeline.

Figure 3.10 depicts the pseudo-synchronous SDDS-NCL synthesis flow. The flow is substantiated as a set of TCL scripts driving tools of the Cadence™EDA framework. Logic synthesis and optimisation employ Genus™ 18.1. Placement and routing are achieved with Innovus™ 18.1.

The flow takes a pre-expanded dual-rail netlist as input. The process of dual-rail expansion, part of the Pulsar flow, is addressed in Section 5.4. The dual-rail input netlist contains both combinational and sequential elements. Pseudo-flops are explicitly instantiated as sequential elements in pipeline stages. Boolean expressions are used to describe the behaviour of combinational logic as VFs for both forward and backward propagation between sequential elements.

SDDS-NCL requires that the logic modules be strictly unate, meaning that all inputs of a gate must reside on the same domain. Since SDDS-NCL is only used for combinational logic, there is no problem in having sequential gates with their inputs at different domains. To solve this, a “size_only” constraint is set on all sequential elements to preserve their logic function.

Genus synthesises and optimises the combinational logic in both backward and forward propagation paths. Synthesis and optimisation must meet some requirements to generate a fixable X-Netlist: (i) they must not generate intermediate circuits with binate functions, as gates in these circuits have inputs at different domains (RTZ and RTO), which would break the circuit behaviour; (ii) they must not produce circuits with gate inputs tied to a constant, as a constant at gate inputs may inhibit the set or reset behaviour of NCL (NCL+) gates.

In the new design flow the input always implements unate functions, all internal nodes should be unate in path start points, i.e. primary inputs and register outputs. However, according to Das et al. [DCB93] it is possible that the synthesis tool to perform a binate realisation of a unate input function. To overcome this issue Moreira [Mor16] proposed using design for testability (DFT) techniques, guaranteeing that requirement (i) above is always meet. Requirement (ii) is fulfilled setting “iopt_force_constant_removal=true” on Genus.

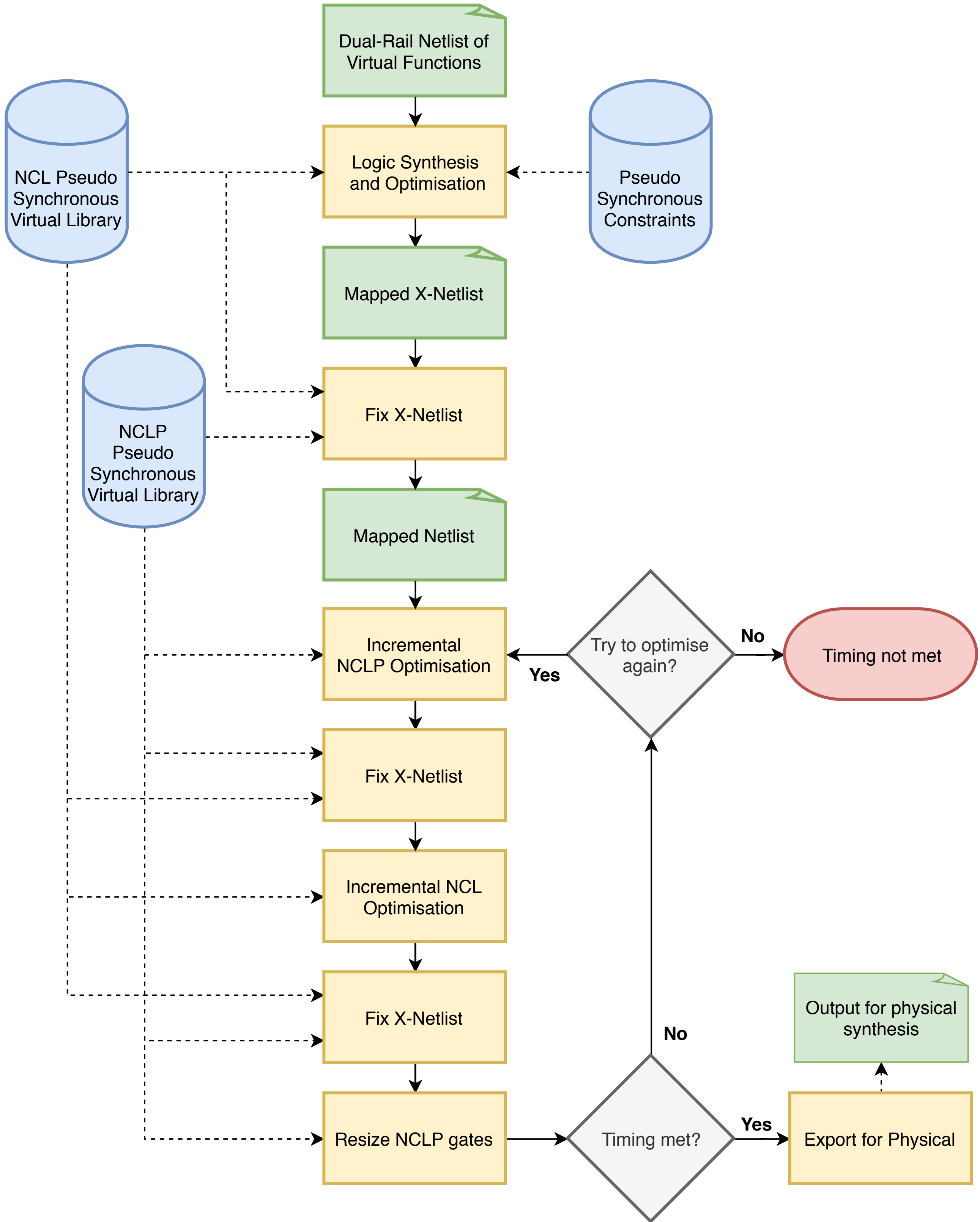


Figure 3.10: The SDDS-NCL sequential synthesis flow.

Assuming the initial circuit description is NCL (i.e. an RTZ description), the output of sequential gates are guaranteed to be RTZ, since initial synthesis and optimisation steps preserve this characteristic. It is also safe to assume from this condition that all primary inputs are also RTZ.

To produce a circuit with the correct behaviour, gates which are in the RTO domain must be NCL+ and gates in the RTZ domain must be NCL. The combinational logic in both forward and backward propagation paths are processed by Algorithm 3.1, which identifies the protocol of each gate and employs the correct gate type.

In the Fix X-Netlist algorithm, a gate is considered in RTZ if its non-inverting fan-ins are RTZ and its inverting fan-ins are RTO. Conversely, a gate is considered in RTO if its non-inverting fan-ins are RTO and its inverting fan-ins are RTZ. Figure 3.11 illustrates this.

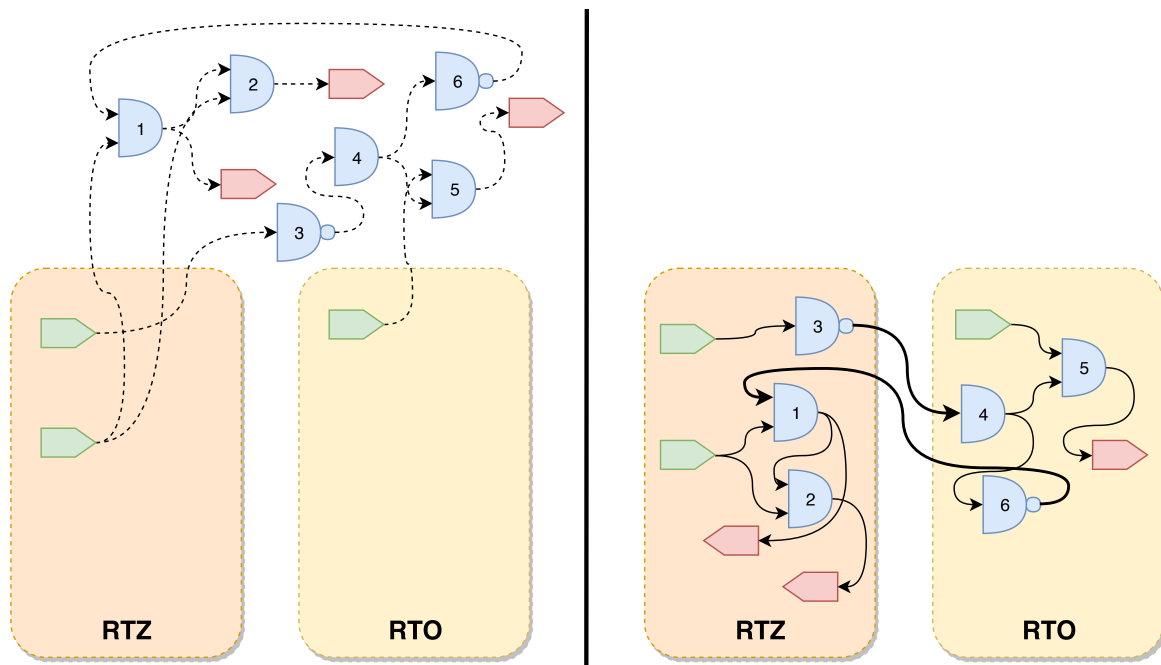


Figure 3.11: Illustration of the RTO and RTZ gate classification process.

If a combinational gate is classified as both RTO and RTZ, the X-Netlist is deemed unsuitable for correction and the synthesis is invalid. The only gates allowed to be in both RTO and RTZ domains are the registers, but these do not need to be processed by the Fix X-Netlist algorithm.

The initial netlist correction may result in a circuit that no longer meets the design constraints. In this case, further optimisation steps are needed to meet them. These optimisation steps are run on each set of gates independently, to avoid undoing the initial netlist correction. After optimising a set of gates, the Fix X-Netlist correction algorithm is re-run to correct any domain changes introduced by the optimisation. These steps are iteratively run until the timing constraints are fulfilled.

Require: Mapped X-Netlist
Ensure: Fixed SDDS-NCL Netlist

```
{Breath first gate classification}
Add all primary inputs to rtz set;
Add all registers to rtz set;
while exist unexplored gates in the X-Netlist do
   for g in rtz do
      if g is inverting gate then
         Add combinational fanout of g to the rto set;
      else
         Add combinational fanout of g to the rtz set;
      end if
   end for
   for g in rto do
      if g is inverting gate then
         Add combinational fanout of g to the rtz set;
      else
         Add combinational fanout of g to the rto set;
      end if
   end for
end while

{Correct gates in RTZ}
for g in rtz do
   if g is NCL+ then
      Replace g with NCL gate of same vFunction;
   end if
end for

{Correct gates in RTO}
for g in rto do
   if g is NCL then
      Replace g with NCL+ gate of same vFunction;
   end if
end for
```

Algorithm 3.1: The Fix X-Netlist algorithm.

After logic synthesis, optimisation and correction steps, the final netlist can be placed and routed by a physical synthesis tool. However, meeting timing constraints during physical synthesis may require some additional optimisation steps. If the logic is changed during these steps, logical correctness of the circuit may be compromised.

Prior to place and route, NCL+ gates are marked with “dont_touch” and “dont_use”, NCL gates are marked “size_only” and buffers are left free for the physical synthesis tool to use. This guarantees the logic will not suffer any changes during physical synthesis and gives the physical synthesis tool freedom to place buffers and resize gates to meet timing.

4. A NEW MODEL FOR CYCLE TIME COMPUTATION

The throughput of asynchronous circuits is not directly dictated by its slowest propagation path. It is rather defined as the inverse of the system *cycle time*, i.e. the mean time between two results. The cycle time depends on complex interactions between multiple pipeline stages and the involved delays. The definition of (asynchronous) cycle time is explored in Section 4.1.

Analysis of the cycle time is also not straightforward. It requires complex models that capture the interactions internal to asynchronous circuits. Section 4.2 reviews the full-buffer channel network (FBCN) timing model and Section 4.3 extends this model to support half-buffer pipelines. These models can be used in conjunction with STA to compute the maximum cycle time of an asynchronous circuit. This is further explored in Section 4.4, where we present a methodology based on linear programming to compute the maximum cycle time of circuits modelled using both FBCN and HBCN.

By the end of this chapter it is expected that the reader is provided with a methodology to analyse the maximum cycle time of asynchronous circuits using global timing models and traditional STA techniques. Thus, estimating the minimal throughput of the circuit during its design.

4.1 Cycle Time of Asynchronous Circuits

According to Beerel, Ozdag, Ferretti [BOF10] the *cycle time* of an asynchronous circuit is the time between two consecutive computation results. On a synchronous circuit this is dictated by the clock period, i.e. a synchronous circuit (most often) produces a result every clock cycle. However, for asynchronous circuits this is defined as the time between two consecutive tokens.

The cycle time is limited by the internal delays and token placement in the circuit. It is best observed under an *ideal null-delay environment*, i.e. an environment that consumes tokens as soon as they arrive and produces new tokens as soon as the previous is acknowledged. This ideal of null-delay environment is used here to isolate the circuit from external influences and data dependent delay variations. Another reason for using an ideal environment model is that the cycle time is observed when the pipeline is at *full capacity*. When the pipeline is empty or partially empty, the time between results at the output is not dependent on internal factors only, but also on the availability of tokens coming from the environment and propagating in the pipeline.

Also, the cycle times observed in an asynchronous circuit can present data dependent variations. The data flow within the asynchronous system may be data dependent,

because they may present choice behaviour due to characteristics such as arbitration, splits and merges. Systems with such behaviour are called *non-deterministic*. On such systems, the tokens propagate through different paths with different delays, thus affecting the cycle time.

Furthermore, even deterministic systems, which present no data-dependent token flow, can show variations in the cycle time observed at inputs or outputs. For instance, the delays of an individual pipeline stage can be data dependent, e.g. an adder unit may complete faster if all inputs are zero. This can lead to situations that a token advances faster through the pipeline, arriving earlier at the output, but leaving part of the pipeline idle waiting for data from a slower part. This idle waiting time will appear as an increase in the cycle time of the output for the next token. Also, temporarily faster delays near the input may accept tokens into the pipeline faster than they can be processed by later pipeline stages. This can cause tokens to queue up in the pipeline entrance. This queuing causes a delay to the admission of the next token, which translates as increases in the observed cycle time in the input.

On average, these variations may cancel each other, i.e. a token with a faster cycle time can be followed by a token with a slower cycle time. The observed cycle time forms a distribution around the average cycle time. On a system that generates an output for each input, the observed average cycle time must be the same on the system inputs and outputs. This average cycle time is the *system cycle time*, i.e. disregarding transient variations, on average the pipeline is capable of producing new results after each cycle time has elapsed.

To demonstrate this effect, the observed cycle time of the input and output of 6-stage multiply and accumulate (MAC) unit is shown in Figure 4.1. The measurements were extracted using delay annotated post-synthesis simulation and a zero-delay external environment to produce and consume data.

The circuit presents a wider cycle time distribution at the output, indicating that sometimes the pipelines stages near the output (namely the accumulator) complete faster than the multiplier (at the beginning of the pipe) is capable of providing new data.

4.2 The FBCN Timing Model (Related Work)

Beerel, Ozdag, Ferretti [BOF10] proposed in their book the Full-Buffer channel network (FBCN), a timed marked graph model that captures the behaviour of full-buffer asynchronous pipelines. FBCN allows modelling the global timing of an asynchronous circuit from individual circuit timing paths (extractable using STA tools).

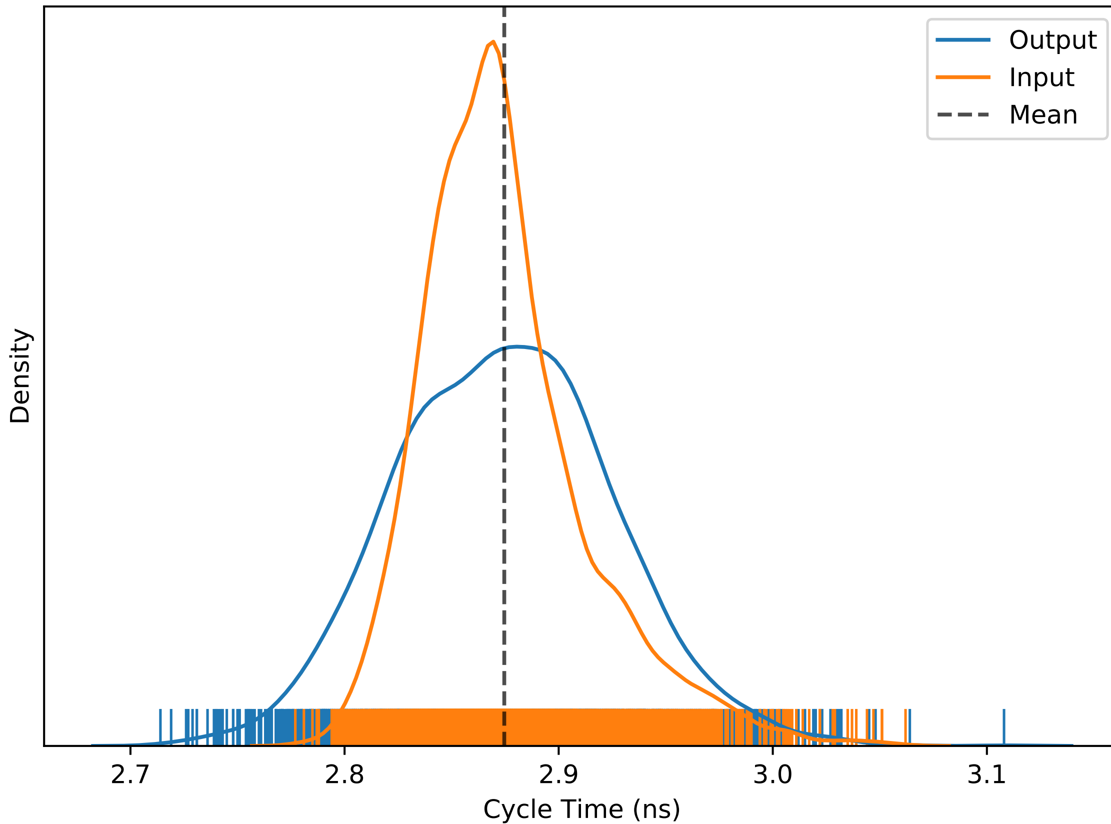


Figure 4.1: Observed cycle time distribution at the input and output of an integer, 6-stage, multiply-and-accumulate circuit.

The formal definition in [BOF10] derives from a comprehensive Petri net definition. However, since FBCN is a marked graph, it is possible to use the definition in Section 2.1.3 to formally redefine FBCN as follows:

Definition 4.2.1 (Full-Buffer Channel Network - FBCN) *A Full-Buffer Channel Network is defined as the 3-tuple $N = (T, C, I_0)$, where T is a set of transitions, $C = \{(u, v) : u, v \in T, u \neq v\}$ is a set of channels, and $I_0 : C \mapsto \{ack, req\}$ is a function defining each channel initial state.*

An FBCN is equivalently represented as a marked graph defined as the 3-tuple $N = (T, P, M_0)$, where T is a set of transitions; $P = \{(u, v) : u, v \in T\}$ is a set of places and M_0 is a set of initially marked places. The equivalence between N and G is defined by the following set of rules:

$$\forall (u, v) \in P : \exists \{(u, v), (v, u)\} \subseteq P \tag{4.1}$$

$$\forall (u, v) | I_0((u, v)) = ack, \exists (v, u) \in M_0 \tag{4.2}$$

$$\forall (u, v) | I_0((u, v)) = req, \exists (u, v) \in M_0 \tag{4.3}$$

Equation 4.1 states that every channel is represented by a pair of places, a forward propagation place and a backward propagation place; Equation 4.2 states that the channel

backward propagation place is marked when the channel is initialised to the ack state; and Equation 4.3 states that the channel forward propagation place is marked when the channel is initialised to the req state.

These definitions introduce the structure of the FBCN. Figure 4.2 depicts a graphical representation of an example FBCN. The token propagation behaviour in an FBCN is governed by the timed marked graph process, as defined in Section 2.1.3. From this, it is viable to define correlations to describe how an FBCN models handshake channels of full-buffer asynchronous circuits.

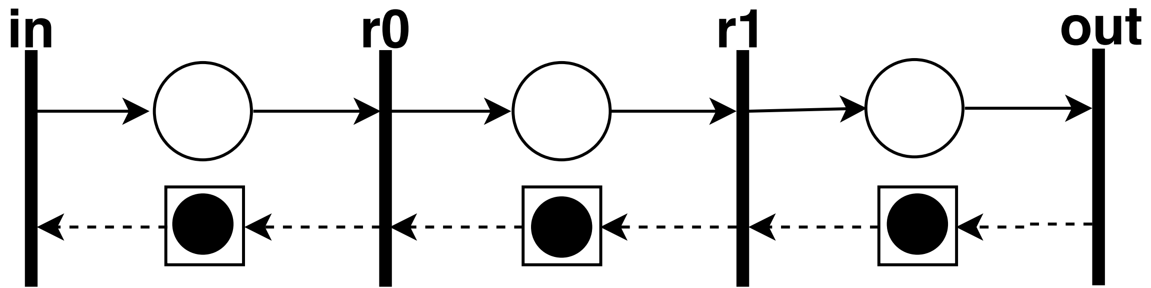


Figure 4.2: FBCN modelling a 2-phase, half-buffer pipeline. Vertical lines are transitions, circles are forward-propagation places, squares are backward-propagation places and black dots are tokens.

Each pair of places ($\{(u,v),(v,u)\} \subset P$) models a channel. Each channel is composed of a forward-propagation place and a backward-propagation place. The forward-propagation place models the forward-propagation path that transport data, and the backward-propagation place models the backward-propagation path that carries control signals. The arrangement of places and transitions creates cycles that model the cyclic paths of handshake channel circuits.

Places are marked with tokens to indicate the state of the handshake channel. When a token is marking the backward-propagation place in a channel, it indicates that the channel is acknowledging the reception of data. When a token is marking the forward-propagation place in a channel, it indicates that the channel is currently carrying valid data. These states are mutually exclusive, a channel cannot be simultaneously at both states, which is formalised as:

Definition 4.2.2 (Handshake state) *Let $M_i \subseteq P$ be the set of marked places at instant $i \in \mathbb{N}$. The following predicates are true for all instants $i \in \mathbb{N}$:*

$$\forall (u,v) \in M_i : \nexists (v,u) \in M_i$$

$$\forall (u,v) \in P : \exists (u,v) \in M_i \vee \exists (v,u) \in M_i$$

That is, a pair of places taking part on a channel cannot be simultaneously marked and at least one place of a channel must be marked at any time.

Transitions capture the handshaking behaviour of registers or of the environment. A transition fires when all of its inbound places are holding tokens, that is, it synchronises the acknowledgement from the outbound channel with the data from the inbound channel producing an acknowledgement in the inbound channel and propagating data to the outbound channel. This effectively models the behaviour of a 2-phase handshake protocol.

Each place is annotated with the delay from the propagation path it represents. This delay can be extracted from the circuit using STA tools. For instance, assuming a 2-phase bundled-data design, the place indicating the forward propagation place in a channel is annotated with the propagation delay of the request signal on that channel. Conversely, the backward propagation place is annotated with the delay of the acknowledge signal on the channel.

4.3 The HBCN Timing Model

As the name states, an FBCN is a model suitable to describe full-buffer pipelines. In a full-buffer pipeline there are only valid data being propagated and acknowledged. Each channel in some FBCN can be in one of two states, either acknowledging or transmitting valid data.

Conversely, in half-buffer pipelines valid data tokens are always followed by spacer tokens. Spacer tokens are characteristic of half-buffer circuits such as WCHB pipelines. They need to be represented in the model to correctly capture the behaviour of a half-buffer pipeline. In a half-buffer pipeline each channel can be in one of four states: propagating data, acknowledging data, propagating a spacer and acknowledging a spacer.

Since FBCN is not capable of modelling this behaviour naturally, this work proposes the Half Buffer Channel Network (HBCN), a modification of FBCN to explicitly model half-buffer pipelines. HBCNs are also timed marked graphs, and all properties of FBCN regarding cycle time analysis hold true for them. An HBCN models the behaviour of 4-phase half-buffer pipelines, by representing the four possible states of each half-buffer channel. These states are captured using four places per channel. A pair of transitions is required at each end of a channel to synchronise the propagation of valid data and spacers. The structure of the HBCN can be formalised as follows.

Definition 4.3.1 (HBCN) *A Half-Buffer Channel Network (HBCN) is defined by the 4-tuple* $N = (T, \Gamma, C, I_0)$*, where* T *is a set of transitions,* $\Gamma = \{(t, t') : t, t' \in T, t \neq t'\}$ *is a set of transition pairs,* $C = \{(u, v) : u, v \in \Gamma, u \neq v\}$ *is a set of channels, and* $I_0 : C \mapsto \{ack_{null}, req_{data}, ack_{data}, req_{null}\}$ *is a function defining each channel initial state. An HBCN has a* characteristic marked graph *defined as the 3-tuple* $G = (T, P, M_0)$ *where* T *is a the set of transitions of the HBCN,* $P = \{(u, v) : u, v \in T\}$ *is a set of places, and* $M_0 \subset P$ *is a set*

of initially marked places. The relation between N and G is defined by the following set of rules:

$$(u,v) \in \Gamma \implies u,v \in T \tag{4.4}$$

$$((u,u'),(v,v')) \in C \implies \{(u,v),(u',v'),(v',u),(v,u')\} \subseteq P \tag{4.5}$$

$$I_0(((u,u'),(v,v'))) = ack_{null} \implies (v',u) \in M_0 \tag{4.6}$$

$$I_0(((u,u'),(v,v'))) = req_{data} \implies (u,v) \in M_0 \tag{4.7}$$

$$I_0(((u,u'),(v,v'))) = ack_{data} \implies (v,u') \in M_0 \tag{4.8}$$

$$I_0(((u,u'),(v,v'))) = req_{null} \implies (u',v') \in M_0 \tag{4.9}$$

Implication 4.4 states that every transition taking part on a transition pair must exist on the transition set; Implication 4.5 states that each channel is expanded to four places, two forward propagation places (for data and spacer) and two cross-connected backward propagation places; Implications 4.6 to 4.9 define which places are initially marked to set channels to their respective initial states.

A possible intuition to understand the HBCN model is to see it as two parallel FBCNs cross-connected by the backward propagation places at each channel. One network captures the propagation of spacers and the other valid data. The handshake protocol is captured by cross-connecting the null backward propagation place of a stage to the data transition of the previous stage and by connecting the data back-propagation place to the previous stage null transition. This enforces that data propagation depends on the propagation of spacers and vice-versa.

Delays can be attributed to each place of the HBCN to capture the propagation time of data and spacers in the network. This allows using HBCN to analyse the timing characteristics of a half-buffer pipelines. The cycle time is defined as the time a token takes to return to a place after being removed, that is, the time it takes for a channel to transitions to between all its states back to its initial state.

To illustrate concepts an example HBCN is now derived from the Weak-Conditioned Half Buffer (WCHB) pipeline in Figure 4.3. This WCHB is a simple Quasi-Delay Insensitive (QDI) half-buffer pipeline template that employs Delay Insensitive (DI) codes, completion detectors and a 4-phase RTZ handshake protocol. C-elements are used as registers to latch DI codes in each stage. This example uses a 1-of-2 code, thus requiring 2 C-elements (one for each rail), the employed completion detector is a NOR gate. A spacer is represented by having the two rails at zero and valid data is represented by having one of the rails at one.

The example circuit comprises two registers (`r0` and `r1`), one input (`in`) and one output (`out`). There are 3 channels in this circuit: (i) from the input to `r0`; (ii) from `r0` to `r1`; and (iii) from `r1` to the output. For each channel, the forward propagation path is the pair of rails transporting data and spacers, and the backward propagation place is the comple-

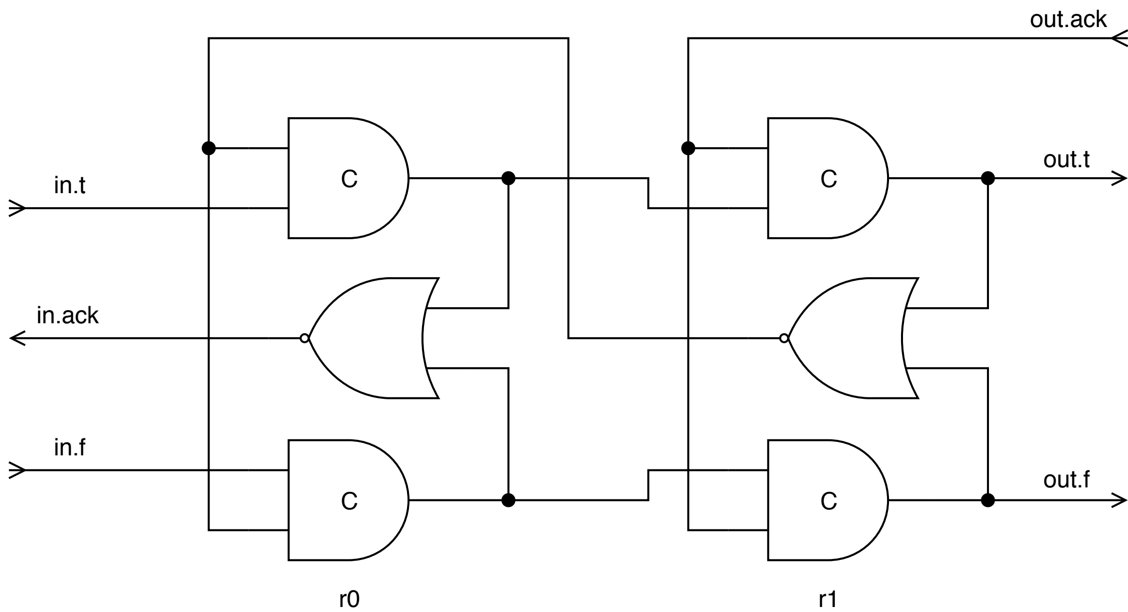


Figure 4.3: A 2-stage dual-rail linear buffer, implemented using a WCHB half-buffer 4-stage RTZ pipeline.

tion detection circuit and its output acknowledgement signal. All channels in this circuit are initialised to spacers.

From the example circuit, we construct the example HBCN in Figure 4.4. Each register (`r0` and `r1`) is modelled as pair of transitions: the transition in blue captures the arrival of valid data and the transition in red captures the arrival of a spacer. The environment (`in` and `out`) is also modelled as a pair of transitions to capture the behaviour of an ideal environment. Each propagation path is modelled as a pair of places: (i) one indicating the propagation of a data token; and (ii) another indicating the propagation of spacers. Square places are backward propagation places representing backward propagation paths. Round places are forward propagation places representing forward propagation paths.

The example HBCN can be described using the channel network formulation presented in Definition 4.3.1 as:

$$N = (T, \Gamma, C, I_0) \tag{4.10}$$

$$T = \{in, in', r0, r0', r1, r1', out, out'\} \tag{4.11}$$

$$\Gamma = \{(in, in'), (r0, r0'), (r1, r1'), (out, out')\} \tag{4.12}$$

$$C = \{((in, in'), (r0, r0')), ((r0, r0'), (r1, r1')), ((r1, r1'), (out, out'))\} \tag{4.13}$$

$$\forall c \in C : I_0(c) = ack_{null} \tag{4.14}$$

In this formulation, Equation 4.10 declares the channel network; Equation 4.12 declares the transition pairs that represent the input, the two registers (`r0` and `r1`) and the output; Equation 4.13 declares the channels connecting the registers; and Equation 4.14 declares the initial state of all channels as ack_{null}. The channel network formulation is equivalent to a

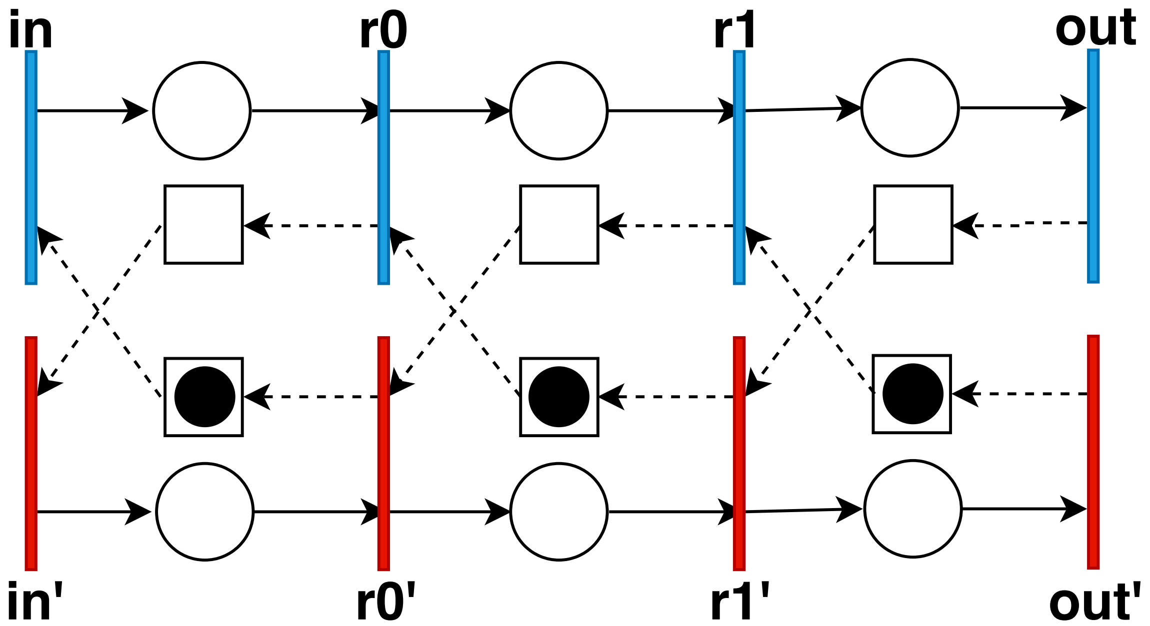


Figure 4.4: Example HBCN modelling a 4-phase, half-buffer circuit. Blue lines are valid data transitions; red lines are spacer transitions; squares are backward propagation places; circles are forward propagation places.

marked graph expansion defined as:

$$G = (T, P, M_0) \tag{4.15}$$

$$T = \{in, in', r0, r0', r1, r1', out, out'\} \tag{4.16}$$

$$P = \begin{Bmatrix} (in, r0), (r0, in'), (r0', in), (in', r0') \\ (r0, r1), (r1, r0'), (r1', r0), (r0', r1') \\ (r1, out), (out, r1'), (out', r1), (r1', out') \end{Bmatrix} \tag{4.17}$$

$$M_0 = \{(r0', in), (r1', r0), (out', r1)\} \tag{4.18}$$

Here, Equation 4.15 declares this HBCN marked graph; Equation 4.16 declares the set of transitions extracted from the transition pairs Γ; Equation 4.17 declares the set of places of the marked graph according to the expansion rule from Equation 4.5; and Equation 4.18 is the set of places marked to initialise all channels to ack_{null}, according to the expansion rule from Equation 4.6.

The marked graph expansion is important, since this representation will later be seen to enable asynchronous cycle time computations. The next Section explores the delay model associated with places in a timed marked graph and introduces a linear programming formulation that aids in the computation of the maximum cycle-time from the timed marked graph.

4.4 Calculating the Maximum Cycle Time

Both HBCN and FBCN are timed marked graphs that contain cycles. Cycles are defined as a cyclic sequence of places where tokens repeatedly flow. On a timed marked graph, token propagation is ruled by the place delays. On these models, the cycle time is defined as the sum of the delays of places taking part in a cycle divided by the number of tokens residing in the cycle.

The maximum cycle time is the largest cycle time in the graph. Ultimately, due to complex interactions in the circuit, this largest cycle time dominates the overall asynchronous pipeline performance. Determining the maximum cycle time is important to analyse pipeline performance.

The maximum cycle time is an upper limit for the circuit cycle time. This is an important performance measurement, as it bounds circuit performance floor. On a marked graph model (such as the HBCN) the maximum cycle time is defined as the greatest cycle time in the circuit. This value is not always intuitively found, especially on large marked graphs with multiple interacting cycles such as the HBCN of a large circuit.

Magott [Mag84] has devised a methodology using linear programming to compute the maximum cycle time in a Petri net. It works by estimating the time each transition fires based on a system of constraints.

The time a transition fires is called the *transition arrival time*. A transition arrival time is limited by its incoming places. The transition can only fire after all of its incoming places allows it. A place only allows its outgoing transition to fire after it has been holding a token for the duration of its delay.

Any place receives a token after its incoming transition fires. If a place holds a token at the initial time it is assumed that it has received that token at the previous cycle time. From these statements it is possible to formalise an arrival time constraint for each place.

Definition 4.4.1 (Arrival time constraint) *Inequation 4.19 is the arrival time constraint defined for a place $p = (u, v)$, where a_u is the arrival time for the incoming transition u; a_v is the arrival time for the outgoing transition v; $d_p \geq 0$ is the delay associated with place p; $m_0(p)$ is the number of initial tokens in p (1, if initially marked, else 0); and $\phi \geq 0$ is the cycle time of the marked graph.*

$$a_v \geq a_u + d_p - m_0(p)\phi \tag{4.19}$$

Inequation 4.19 states that v fires d_p time units after u has fired, unless p is initially marked, than it is assumed that v has fired one cycle before.

Since the delay attributed to each place is the maximum delay of the propagation path, it is safe to assume that the transition will fire as soon as all arrival constraints are satisfied. Following this assumption, the maximum cycle-time is the minimal value of ϕ that satisfies the set of constraints. This can be computed using a linear programming (LP) solver, by providing the arrival time constraints with the place delays and setting the objective function as minimising ϕ.

The LP solver solution returns values for each variable in the input formulation. This includes the value of ϕ, which is the maximum cycle time. But it also includes values for the arrival time of each transition, byproduct values with no concrete meaning, one solution of many possible ones. The LP solver is free to attribute values to these variables, only the distance and ordering of these values are important to the solution process.

However, more useful information could be extracted from the solution. For instance, the free slack is the amount of time a token stays in a place after its delay has expired. This information is useful for optimising the pipeline performance. Beerel et al. [BLDK06] has extended the arrival time constraint formulation to extract the free slack of each place, creating the arrival time equation.

Definition 4.4.2 (Arrival Time Equation) *Equation 4.20 is the arrival time equation defined for each place* $p = (u, v)$*, where* a_u *is the arrival time for the incoming transition* u*;* a_v *is the arrival time for the outgoing transition* v*;* $d_p \geq 0$ *is the delay associated with place* p*;* $\rho_p \geq 0$ *is the free slack of place* p*,* $m_0(p)$ *is the number of initial tokens in* p *(*1 *if initially marked, else* 0*); and* $\phi \geq 0$ *is the cycle time of the marked graph.*

$$a_v = a_u + d_p + \rho_p - m_0(p)\phi \tag{4.20}$$

The arrival time equation can replace the arrival time constraint in the LP solver formulation. The solution of the augmented formulation yields the maximum cycle time, the arbitrary transition arrival times and the free slack of each place.

An example to demonstrate this methodology is to compute the maximum cycle time of a 3-stage circular half-buffer circuit. The example circuit is modelled as an HBCN, depicted in Figure 4.5, with the formulation below.

$$N = (\Gamma, C, I_0) \tag{4.21}$$

$$\Gamma = \{r0, r1, r2\} \tag{4.22}$$

$$C = \{(r0, r1), (r1, r2), (r2, r0)\} \tag{4.23}$$

$$I_0(c) = \begin{cases} req_{data} & c = (r0, r1) \\ ack_{null} & c = (r1, r2) \\ req_{null} & c = (r2, r0) \end{cases} \tag{4.24}$$

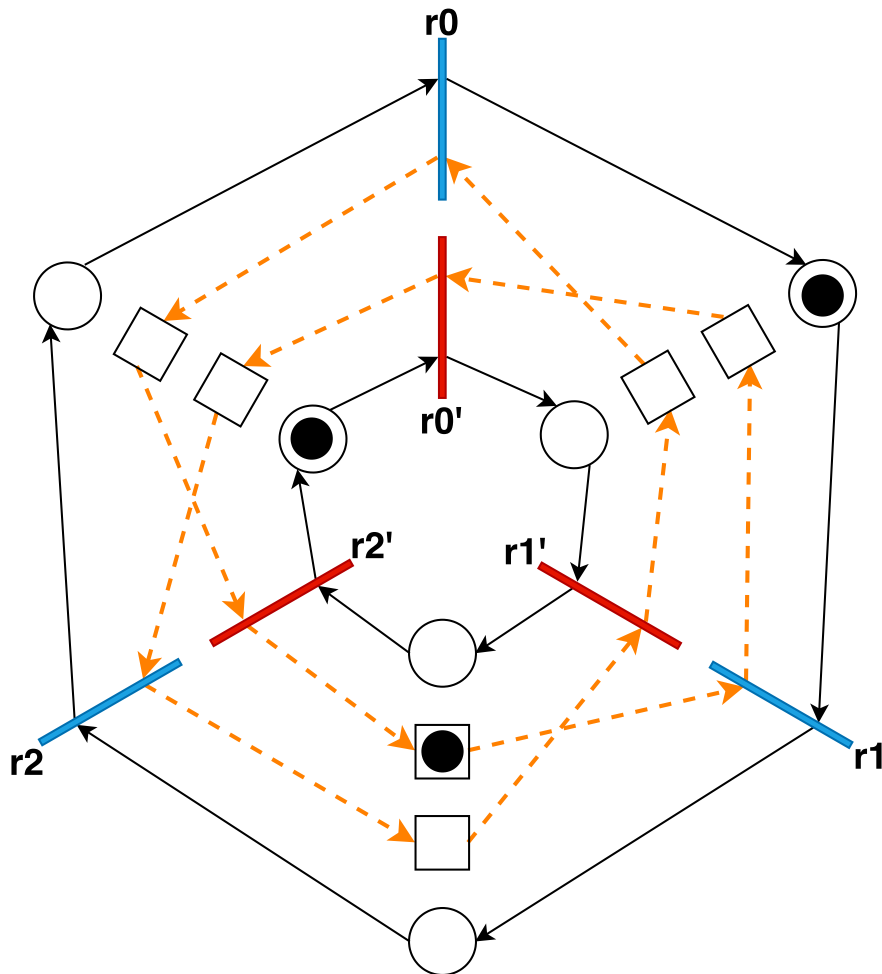


Figure 4.5: HBCN of a 3-stage, circular half-buffer circuit.

Using Definition 4.3.1 it is possible to find the HBCN equivalent marked graph, which is as follows.

$$G = (T, P, M_0) \tag{4.25}$$

$$T = \{r0, r0', r1, r1', r2, r2'\} \tag{4.26}$$

$$P = \begin{Bmatrix} (r0, r1), (r0', r1'), (r1', r0), (r1, r0') \\ (r1, r2), (r1', r2'), (r2', r1), (r2, r1') \\ (r2, r0), (r2', r0'), (r0', r2), (r0, r2') \end{Bmatrix} \tag{4.27}$$

$$M_0 = \{(r0, r1), (r2', r1), (r2', r0')\} \tag{4.28}$$

On this example, assume every forward propagating place is assigned a delay of 1.5 time units and that every backward propagating place is assigned a delay of 1 time unit. Using the arrival equation from Definition 4.4.2, it is possible to obtain an LP problem formulation to compute the maximum cycle time and the free slacks for the Example HBCN. Algorithm 4.1 shows the LP formulation, and its solution, obtained from an LP solver appears in Table 4.1. All variables are bounded to be positive real values.

Minimise

ϕ

Subject To

$a_{r1} = a_{r0} + 1.5 + \rho_{(r0,r1)} - \phi$

$a_{r1'} = a_{r0'} + 1.5 + \rho_{(r0',r1')}$

$a_{r0} = a_{r1'} + 1 + \rho_{(r1',r0)}$

$a_{r0'} = a_{r1} + 1 + \rho_{(r1,r0')}$

$a_{r2} = a_{r1} + 1.5 + \rho_{(r1,r2)}$

$a_{r2'} = a_{r1'} + 1.5 + \rho_{(r1',r2')}$

$a_{r1} = a_{r2'} + 1 + \rho_{(r2',r1)} - \phi$

$a_{r1'} = a_{r2} + 1 + \rho_{(r2,r1')}$

$a_{r0} = a_{r2} + 1.5 + \rho_{(r2,r0)}$

$a_{r0'} = a_{r2'} + 1.5 + \rho_{(r2',r0')} - \phi$

$a_{r2} = a_{r0'} + 1 + \rho_{(r0',r2)}$

$a_{r2'} = a_{r0} + 1 + \rho_{(r0,r2')}$

End

Algorithm 4.1: LP formulation to compute the cycle time and free slack for the example HBCN.

Variable	Value
ϕ	6
a_{r1}	0
$a_{r0'}$	1
a_{r2}	2
$a_{r1'}$	3
a_{r0}	4
$a_{r2'}$	5
$\rho_{(r0,r1)}$	0.5
$\rho_{(r0',r1')}$	0.5
$\rho_{(r1,r2)}$	0.5
$\rho_{(r1',r2')}$	0.5
$\rho_{(r2,r0)}$	0.5
$\rho_{(r2',r0')}$	0.5
$\rho_{(r1,r0')}$	0
$\rho_{(r1',r0)}$	0
$\rho_{(r2,r1')}$	0
$\rho_{(r2',r1)}$	0
$\rho_{(r0,r2')}$	0
$\rho_{(r0',r2)}$	0

Table 4.1: Solution of the LP formulation.

The solution of ϕ shows that the maximum cycle time of the Example circuit is 6 time units. The solutions of the arrival variables from a_{r1} to $a_{r2'}$ are the estimated arrival

times, they show the order in which transitions happen in a cycle. The solution of variables $\rho_{(r0,r1)}$ to $\rho_{(r2',r0')}$ show that the forward propagation places have a free slack of 0.5 time units, which means that data transitioning in that path have to wait 0.5 time units to propagate after they are ready. The solution of variables $\rho_{(r1,r0')}$ to $\rho_{(r0',r2)}$ indicates that the backward propagation paths are part of the critical cycle, data propagating in these paths evolve to the next stage as soon as they arrive to their registers. The existence of free slack in this circuit shows that the pipeline is not optimised.

4.5 Constraining the Maximum Cycle Time

The last Section proposed and explored a technique to analyse the cycle time of HBCNs. However, this technique assumes a pre-synthesised circuit. This Section extends the linear programming (LP) formulations used for cycle time analysis to derive the individual path delays that satisfy a target cycle time.

Commercial EDA tools heavily rely on Static Timing Analysis (STA). Tools that perform STA are timing driven, they operate with the concept of a timing budget to guide their synthesis effort. For a circuit to work correctly, the timing of the combinational logic in these paths must be less or equal to their timing budget. Logic synthesis tools employ STA to elaborate logic circuits that respect the defined timing budget. Usually, for synchronous flop-based circuits, the timing budget is constrained using the target clock period, but timing exceptions can also be set to constrain individual paths. Chapter 3 introduced the pseudo-synchronous SDDS-NCL, a novel QDI design template that enables commercial EDA tools to synthesise and optimise sequential QDI circuits with constrained propagation paths. The pseudo-synchronous SDDS-NCL template allows defining a pseudo-clock and some timing exceptions to constraint the delay of individual propagation paths.

The previous Section developments relied on annotating an HBCN model with delays of individual propagation paths. From such HBCN it was then possible to derive a system of arrival equations $a_v = a_u + d_p + \rho_p - m_0(p)\phi$. These were then used in an LP formulation to compute the maximum cycle time of the circuit modelled by the HBCN. Recall that d_p is the place delay corresponding to a timing path and ϕ is the maximum cycle time.

The cycle time analysis method is now extended to a synthetic approach. Instead of analysing the maximum cycle time of a given circuit, the HBCN and arrival equation are used to compute the delay constraints of places required to meet a desired maximum cycle time (ϕ). For that, a single delay λ is assigned to every place corresponding to timing paths which are to be constrained. A system of arrival time equations is then extracted from the HBCN model, the maximum cycle time value ϕ is set to the desired value and the value of λ is maximised using an LP solver. The computed value of λ is the pseudo-clock used to constrain the timing paths of the original circuit. The intuition behind this method is to

discover "how slow" the constrained timing paths can be, without violating the maximum cycle time constraint. Since the pseudo-synchronous SDDS-NCL template allows defining a pseudo-clock to constrain the delay of individual paths, it is fair to assume that the delay of these paths will be limited by λ.

Notice that the method proposed here allows the designer to optionally specify a fixed delay for any place p by attributing a numeric value to d_p. This is useful when a timing path has a known delay or is its delay is fine-tuned by the designer using timing exceptions. These fixed delays are taken in consideration during the pseudo-clock computation, thus guaranteeing that the pseudo-clock constraint still bounds the maximum cycle time. The use of fixed delays on timing paths known to be fast in the critical cycle may allow the pseudo-clock constraint to be more relaxed.

To demonstrate the method, consider an example of constraining the 3-stage circular buffer whose HBCN appears in Figure 4.5, to the maximum cycle time of 2 ns. In this example, assume that the places corresponding to the channel connecting the transition pair $(r0, r0')$ to $(r1, r1')$ have a fixed delay of 100 ps. The LP formulation used to compute the pseudo-clock constraint λ is depicted in Algorithm 4.2. It states that the value of λ should be maximised, subject to $\phi = 2$ and the system of arrival equations. Each arrival equation corresponds to a place $p = (u, v)$ in the HBCN. These equations state that the transition on the postset of a place should occur at least λ time units after the transition on the preset of the same place, except on places with tokens where the transition can occur ϕ time units ahead. Here, the free slack ρ_p is a positive real value that captures the amount of time a token has to wait idling in a place after its propagation delay. Running this model on an LP solver results in the solutions for the variables presented in Table 4.2.

Maximise

λ

Subject To

$\phi = 2$

$a_{r1} = a_{r0} + 0.1 + \rho_{(r0,r1)} - \phi$

$a_{r1'} = a_{r0'} + 0.1 + \rho_{(r0',r1')}$

$a_{r0} = a_{r1'} + 0.1 + \rho_{(r1',r0)}$

$a_{r0'} = a_{r1} + 0.1 + \rho_{(r1,r0')}$

$a_{r2} = a_{r1} + \lambda + \rho_{(r1,r2)}$

$a_{r2'} = a_{r1'} + \lambda + \rho_{(r1',r2')}$

$a_{r1} = a_{r2'} + \lambda + \rho_{(r2',r1)} - \phi$

$a_{r1'} = a_{r2} + \lambda + \rho_{(r2,r1')}$

$a_{r0} = a_{r2} + \lambda + \rho_{(r2,r0)}$

$a_{r0'} = a_{r2'} + \lambda + \rho_{(r2',r0')} - \phi$

$a_{r2} = a_{r0'} + \lambda + \rho_{(r0',r2)}$

$a_{r2'} = a_{r0} + \lambda + \rho_{(r0,r2')}$

End

Algorithm 4.2: LP formulation to constrain the cycle time of the HBCN in Figure 4.5 to 2ns, using the pseudo-clock method with fixed delays.

Variable	**Value**
λ	0.45 ns
a_{r1}	0 ns
$a_{r0'}$	0.1 ns
a_{r2}	0.55 ns
$a_{r1'}$	1 ns
a_{r0}	1.1 ns
$a_{r2'}$	1.55 ns
$\rho_{(r0,r1)}$	0.8 ns
$\rho_{(r0',r1')}$	0.8 ns
$\rho_{(r1,r2)}$	0.1 ns
$\rho_{(r1',r2')}$	0.1 ns
$\rho_{(r2,r0)}$	0.1 ns
$\rho_{(r2',r0')}$	0.1 ns
$\rho_{(r1,r0')}$	0 ns
$\rho_{(r1',r0)}$	0 ns
$\rho_{(r2,r1')}$	0 ns
$\rho_{(r2',r1)}$	0 ns
$\rho_{(r0,r2')}$	0 ns
$\rho_{(r0',r2)}$	0 ns

Table 4.2: Solutions to the LP problem in Algorithm 4.2

In this example, the fixed delays take part on the critical cycle and helped relaxing the pseudo-clock constraint. To understand their impact, the experiment is repeated without fixing any place delays. Here, every timing path is equally constrained with the pseudo-clock value. The LP formulation is depicted in Algorithm 4.3 and its solution is presented in Table 4.3. Notice that value of λ is much lower in comparison. Note also the free slack is equally distributed along all forward-propagating places.

Maximise

λ

Subject To

$\phi = 2$

$a_{r1} = a_{r0} + \lambda + \rho_{(r0,r1)} - \phi$

$a_{r1'} = a_{r0'} + \lambda + \rho_{(r0',r1')}$

$a_{r0} = a_{r1'} + \lambda + \rho_{(r1',r0)}$

$a_{r0'} = a_{r1} + \lambda + \rho_{(r1,r0')}$

$a_{r2} = a_{r1} + \lambda + \rho_{(r1,r2)}$

$a_{r2'} = a_{r1'} + \lambda + \rho_{(r1',r2')}$

$a_{r1} = a_{r2'} + \lambda + \rho_{(r2',r1)} - \phi$

$a_{r1'} = a_{r2} + \lambda + \rho_{(r2,r1')}$

$a_{r0} = a_{r2} + \lambda + \rho_{(r2,r0)}$

$a_{r0'} = a_{r2'} + \lambda + \rho_{(r2',r0')} - \phi$

$a_{r2} = a_{r0'} + \lambda + \rho_{(r0',r2)}$

$a_{r2'} = a_{r0} + \lambda + \rho_{(r0,r2')}$

End

Algorithm 4.3: LP formulation to constrain the cycle time of the HBCN in Figure 4.5 to 2ns, using the pseudo-clock method.

Variable	Value
λ	0.333 ns
a_{r1}	0 ns
$a_{r0'}$	0.333 ns
a_{r2}	0.667 ns
$a_{r1'}$	1 ns
a_{r0}	1.333 ns
$a_{r2'}$	1.667 ns
$\rho_{(r0,r1)}$	0.333 ns
$\rho_{(r0',r1')}$	0.333 ns
$\rho_{(r1,r2)}$	0.333 ns
$\rho_{(r1',r2')}$	0.333 ns
$\rho_{(r2,r0)}$	0.333 ns
$\rho_{(r2',r0')}$	0.333 ns
$\rho_{(r1,r0')}$	0 ns
$\rho_{(r1',r0)}$	0 ns
$\rho_{(r2,r1')}$	0 ns
$\rho_{(r2',r1)}$	0 ns
$\rho_{(r0,r2')}$	0 ns
$\rho_{(r0',r2)}$	0 ns

Table 4.3: Solutions to the LP problem in Algorithm 4.3.

The maximum delay λ is used as a pseudo-clock during synthesis, constraining all paths in the circuit. This solution is simple and scales very well to large designs, but it limits the maximum cycle time at the expense of over-constraining some paths. As a consequence, this unnecessarily reduces the time budget of some paths, possibly causing negative impact on the quality of the synthesis on designs with free slack. The result can be the production of circuits with higher than minimum area and power consumption. To understand the issues with the free slack and the pseudo-clock period, take the 3-stage loop example HBCN shown in Figure 4.5. On this example, the longest cycle, i.e. the cycle with most places per token, is marked with a dashed orange line. It has 6 places for a single token, while all other cycles have 4 places per token. If every place is to be constrained equally by the pseudo-clock period, places not taking part in this longest cycle would be overly constrained. This is further evidenced in the LP formulation, showing that the places taking part in the highlighted cycle have nought free slack while all other places have some free slack. The paths corresponding to these places could be made slower without negatively impacting the maximum cycle time constraint. The maximum cycle time ultimately limits the throughput of the circuit. The free slack could be incorporated in the corresponding path constraint to relax

the synthesis of these paths without affecting the cycle time. This could be solved by setting timing exceptions on these paths.

5. THE PULSAR SYNTHESIS FLOW

The Pulsar flow is the main contribution of this dissertation. It is a complete flow for synthesising a constrained QDI circuit from an RTL description. Pulsar comprises the Pseudo-Synchronous SDDS-NCL Template and the HBCN timing model to leverage commercial EDA tools. The Pulsar Flow is depicted in Figure 5.1. Here, the synthesis flow begins with a single-rail synthesis, which produces a single-rail netlist. This netlist is processed and transformed into a virtual netlist comprising virtual functions and pseudo-flops. The virtual netlist is the input to the Pseudo-Synchronous SDDS-NCL synthesis flow. The Pulsar flow also constructs the HBCN and creates the cycle time constraints automatically. Readers should notice that the Sequential SDDS-NCL Synthesis Flow, detailed in Section 3.3 is at the heart of the Pulsar flow.

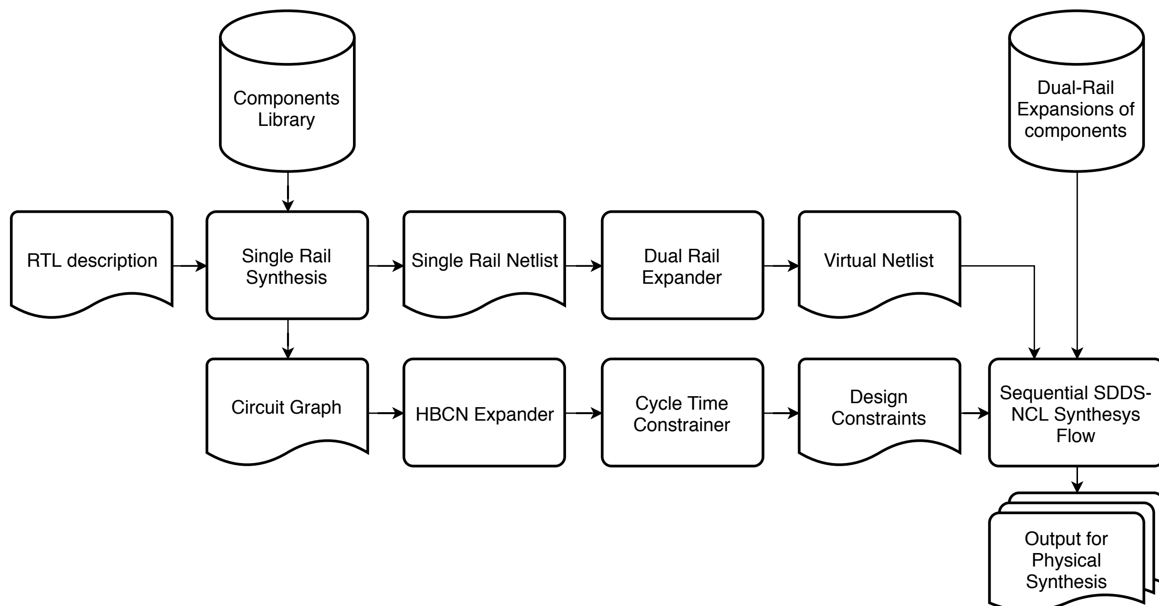


Figure 5.1: The Pulsar synthesis flow.

The design capture methodology shares similarities with Uncle [RST12], as it uses an especially crafted RTL description and traditional EDA tools to synthesise a single-rail netlist. However, Uncle relies on its own specialised tool for technology mapping and optimisation of the NCL netlist, whereas Pulsar flow relies on SystemVerilog constructs and the Sequential SDDS-NCL Synthesis Flow. The RTL netlist is synthesised using a traditional EDA tool to produce a single-rail netlist of *components*. These components are defined in the *components library* using the Synopsys Liberty format. This library contains the combinational and sequential components that can be used by the EDA tool during the synthesis of the *single-rail netlist*. Each component has an equivalent SystemVerilog module defining its dual-rail expansion.

The dual-rail expansion takes advantage of SystemVerilog interfaces [SDF06] to represent dual-rail four-phase RTZ channels. These channels interconnect modules implementing the dual-rail expansion of components. The SystemVerilog interface is also used for constructing the acknowledgement network for channels. A simple tool replaces every wire in the single-rail netlist with a channel to create the *virtual netlist*. The virtual netlist and the modules implementing the dual-rail expansion of components are input to the synthesis flow described in Section 3.3.

Concurrent to the creation of the virtual netlist, cycle time constraints are computed. The scripts used for the single-rail synthesis produce a *structural graph* describing the pipeline topology. This circuit graph is used to model the HBCN of the expanded circuit. The same tool that constructs the HBCN also computes the cycle-time constraint using the linear programming technique discussed in 4.5.

Each topic taking part on this flow is explored in specific Sections herein. Section 5.2 covers details on the channel expansion process. The component library and its components are explored in Section 5.3. The Pulsar flow is depicted using a simple example, in Sections 5.4 and 5.6. The automatic construction of the HBCN and the implementation of the cycle time constrainer are the subject of Section 5.5

5.1 QDI Synthesis Systems (Related Work)

Balsa [EB02] is a CSP-derived [Hoa78, Hoa85] language and framework for implementing asynchronous circuits. The Balsa language comprises primitives for explicit parallelism of sequential statements and communication channels. Balsa produces *handshake circuits* using a process called *syntax-directed translation*. Handshake circuits [Ber93] implement control-driven asynchronous circuits. They are composed of *handshake components* connected by channels. Each handshake component is designed to handshake with their neighbours sequentially, conditionally or in parallel. The behaviour of handshake components are directly related to the behaviour of CSP constructs. A handshake circuit can be seen as direct hardware equivalent of a CSP program, controlling the data flow in a datapath. In syntax-directed translation handshake components are selected from a small predefined set that maps directly onto Balsa language constructs. Each handshake component has a parameterisable gate-level implementation. The circuit gate-level netlist is generated by applying the parameters to each component instance and composing the resulting component gate-level netlists. This approach gives control to the designer, providing one-to-one relation between language constructs and handshake components. However, the control driven nature of handshake circuits imposes overheads that hurt performance sensitive applications [PTE05, BTE09]. Data-driven asynchronous circuits provide an interesting lower-overhead alternative to control-driven circuits.

Uncle [RST12] is a design flow for producing NCL circuits from RTL descriptions. The first step in the Uncle data-driven design flow is to use a commercial synthesis tool to produce a single-rail synchronous netlist from the RTL description, using a simple library of gates composed of 2-input AND/OR/XOR gates, NOT gates, edge sensitive flip-flops and level sensitive latches. The single-rail netlist is next expanded to a dual-rail netlist in the following four steps: (i) the clock network is removed; (ii) every net (except the global reset) is replaced by two nets (for dual-rail encoding) and logic gates are replaced by dual-rail NCL equivalents using predefined templates of NCL gates; (iii) single-rail latches are replaced by a single dual-rail latch and flip-flops are substituted by a sequence of two dual-rail latches; Uncle dual-rail latches are implemented using a resetable NCL2W11OF2 gate (a resetable C-Element) and an NOR, 1-of-2 completion detection gate; (iv) the acknowledge network (backward propagation path) is generated between latches. The generated dual-rail netlist passes through optional optimisation steps that improve performance: (i) cell merging, where a subset of simple connected NCL gates is replaced by a more complex NCL gate; (ii) retiming, a process that attempts to balance the propagation delay of the forward and backward path by moving logic across latches; and (iii) relaxation, a process in which NCL gates are replaced by simple combinational gates without compromising the indication principle.

5.2 Dual-rail Channels

As discussed in Section 2.2, handshake channels allow the communication of tokens between entities in an asynchronous circuit. A producer sends tokens through the channel and consumers receive it. Once a consumer receives the token, it acknowledges the reception. A producer must wait until all consumers connected to it acknowledge the reception of token prior to starting a new transmission.

Each channel is a interconnection between components in an asynchronous circuit. They are analogous to wires in a single-rail netlist, i.e. a single element sends tokens to multiple elements. But in contrast to a wire, a channel provides synchronisation between these elements. Pulsar dual-rail expansion replaces all wires from the single-rail netlist with channels. However, a channel is not a simple wire. A channel implementation may require multiple wires and some logic.

To this end, this works takes advantage of SystemVerilog *interfaces* [SDF06]. An interface is a SystemVerilog construct that allows abstracting interconnections on a system-level description. They are instantiated and bound to module instances much like wires. However, each interface bundles multiple wires that are part of the same interconnect, e.g. bundling the tokens, address and control lines of a processor bus. Besides wires, interfaces include *modports*, that allow defining how modules connect the interface, e.g. bus master or

slave. These modports define the direction that each wire takes when the interface is bound to a module.

SystemVerilog interface and modport are supported by current commercial synthesis tools. Since an interface binds to module instances like a wire, it is straightforward to replace a wire with an interface instance. It only requires modifying the Verilog file of the single-rail netlist, replacing every wire declaration with an instance of the desired interface.

To exploit this, we define the `drwire` interface, depicted in Listing 5.1. This interface bundles three wires that together implement a dual-rail four-phase RTZ channel: wires `t` and `f` encode the forward-propagating token; and the `ack` wire carries the back-propagating acknowledgement.

Listing 5.1: The `drwire` interface.

```
1 interface drwire();
2    wire t, f;
3    wand ack;
4
5    modport in (input t, input f, output ack);
6    modport out (input ack, output t, output f);
7 endinterface // drwire
```

The `drwire` interface also features two modports. These are used by SystemVerilog modules implementing the dual-rail expansion of components to bind with the channel: (i) modport `in` is used by modules as a replacement for input ports, binding to the channel as a consumer; and (ii) modport `out` is used by modules as a replacement for output ports, binding to the channel as a producer.

Since channels here are replacements for wires, they must support multiple consumers. At a given time the producer may send tokens, starting a handshake on the channel. The token reception must be acknowledge by all consumers listening to the channel before the producer is allowed to send more tokens. This means that the producer must receive the acknowledgement only after all consumers have acknowledged the reception.

Sparsø and Furber [SF01] describe in their book this situation where a channel feeds two or more consumers, as in a fork. A fork is implemented by merging the concurrent acknowledgement coming from different consumers with n-input C-elements, i.e. the channel acknowledgement only raises (lowers) when every consumer has raised (lowered) its acknowledgements. An n-input C-element is commonly implemented using a tree of smaller C-elements. This tree of C-elements is also know as the *acknowledgement network*, because it propagates the acknowledgement from consumers to producers in the channel.

To implement forks transparently in channels, this work uses the Verilog *wand* wire type to create the acknowledgement network. Wire declared as wand is a wire that and-reduces multiple assignments. The synthesis tool implements wand wires as n-input AND gates, where every assignment creates a new input. Reading from a wand wire yields

the output of the associated n-input AND gate. During synthesis, this n-input AND gate is interpreted as an and-reduction virtual function.

As discussed in Section 3.1, virtual functions are used to select NCL or NCLP gates during synthesis according to required protocol. By design, all channels in the virtual netlist implement the RTZ protocol. Therefore, the virtual functions associated with the channel are interpreted as virtual functions of NCL gates. C-elements are valid NCL gates and they can be selected by their v-function during synthesis. The virtual function of NCL gates is equal to their activation function.

The activation function of an n-input C-element is the and-reduction of its input, i.e. its output changes to one when all inputs are one. Therefore, the and-reduction virtual function is realised by an n-input C-element on RTZ. This allows creating the acknowledgement network for the channel by setting the `ack` wire type to wand in the `drwire` interface definition.

The use of `drwire` allows the construction of fine-grain channel networks, where each channel represents a single dual-rail “bit”. Combinational components bind to channels passively, they manipulate tokens and propagate acknowledgements between their inbound and outbound channels. Conversely, sequential components bind to channels as active producers and consumers. They consume tokens from their inbound channels and commence handshaking on their outbound channels.

The construction of this fine-grain channel network allows handshakes to be performed only where data dependency exists. Parallel independent channels in a bus can complete their handshakes concurrently. This allows for instance that individual bits in an adder complete handshaking as soon as their computation is ready, regardless the computation status of other bits.

5.3 A Library of Components

The synthesis of a sequential SDDS-NCL circuit with the Pulsar flows starts with the synthesis of a single-rail netlist. This single-rail netlist comprises sequential and combinational components from the component library. During single-rail synthesis, these components are presented to the synthesis tool as cell models using a Synopsys Liberty file. However, these bear no physical layout. Instead each component is expanded by a SystemVerilog module that binds to channels implemented by the `drwire` interface.

Combinational components binds passively to their input and output channels. This means that they do not complete handshakes, instead they act as a passive consumers and producers. Combinational components combine tokens from their input channels on the output channel. They also propagate the acknowledgement between their output and input

channels. Conversely, sequential components are active handshaking elements that control the propagation of tokens in the pipeline. They complete handshakes between their input and output channels.

These SystemVerilog modules are instantiated by the virtual netlist. The content of these modules are used during the synthesis process to implement the circuit using the Pseudo-Synchronous SDDS-NCL template. Combinational components are expanded using virtual functions and sequential components are expanded to pseudo-synchronous WCHB registers. The following Sections cover the expansion of each component type.

5.3.1 Combinational Components

To understand how the expansion of a combinational element is constructed, consider the expansion of a `nand2` gate. This is a passive element that binds to `drwire` channels. Since the `drwire` interface implements dual-rail RTZ encoded channels, a component expansion needs to implement adequate delay insensitive logic to manipulate dual-rail RTZ codewords. The component corresponds to a 2-input NAND gate. The truth table for its single-rail equivalent is depicted in Figure 5.2a. It implements the function $y = \neg(a \wedge b)$. The implementation is not delay insensitive, as it does not differentiate intermediary computations from final computation results.

Remember the discussion about delay insensitive codes in Section 2.3. To make it delay insensitive, it is necessary to encode the gate input and output with a delay insensitive (DI) code. For this purpose, it is possible to use the 3NCL code presented by Fant in [Fan05]. This code introduces a null (N) codeword, alongside the traditional true (T) and false (F) valid Boolean values. Each codeword encodes a token that propagates in channels.

The 3NCL code table of the gate is depicted in Figure 5.2b. In this expansion, the values from the single-rail truth table are mapped to codewords of 3NCL: 1 maps to T and 0 maps to F. As a requirement of QDI circuits, logic must be transparent to the channel handshake. This requirement implies that the state of the output channel of a combinational element must indicate the state of its input channels. This is guaranteed by implementing a *strong-indicating* component, meaning that the component output must only become valid (T or F) when all inputs are valid and must become null (N) only when all inputs are null. To implement a strong-indicating component, it is possible to introduce a hysteresis behaviour to the output. This means that the component output holds its previous value unless all input are null or all are valid. This guarantees that when a valid (null) output is detected, all inputs are valid (null).

This 3-valued logic is not implementable using CMOS conventional gates. However, it is possible to encode 3NCL codewords using a dual-rail code. Section 2.3 reviewed two possible dual-rail codes, each suitable to a matching protocol. The return-to-one (RTO)

	y	1	0
b	1	0	1
	0	1	1

(a) Single-rail Boolean truth table.

	y	*T*	*F*	*N*
	T	*F*	*T*	-
b	*F*	*T*	*T*	-
	N	-	-	*N*

(b) 3NCL code table.

Figure 5.2: Expansion of the `nand2` gate from: (a) a single-rail Boolean truth table to (b) a 3NCL code table. In the tables, (-) indicates hysteretic behaviour.

dual-rail code is employed when the channel implements the RTO protocol and the return-to-zero (RTZ) protocol is used when the channel implements the RTZ protocol. Figure 5.3 depicts the mapping between Boolean, 3NCL and dual-rail RTZ and RTO codes.

Boolean	3NCL	Dual-Rail RTZ t	Dual-Rail RTZ f	Dual-Rail RTO t	Dual-Rail RTO f
1	*T*	1	0	0	1
0	*F*	0	1	1	0
	N	0	0	1	1

Figure 5.3: Map between codes. Only valid codewords are depicted.

A dual-rail code uses two wires to encode tokens. Of course there are 4 possible combination of Boolean values that can be encoded with 2 wires. However, DI codes require that no codeword is contained in another [Ver88], which limits possible DI codewords to 2. Dual-rail RTZ is a bitwise one-hot code (each bit is one-hot coded), where each valid codeword is represented by rising its respective rail and the null codeword is represented by setting both rails low. Rising both rails is considered an invalid codeword, this invalid codeword can be use during the expansion as a “don’t-care” to minimise logic.

The code table for the dual-rail RTZ encoding of the `NAND2` gate is depicted in Figure 5.4. This table establishes how output *y* behaves as a function of inputs *a* and *b*. On the left-hand side it depicts 3NCL codewords and on the right-hand side appears their dual-rail RTZ equivalents. Each line establishes possible output values for each input minterm. There are three possibilities contemplated for each output rail: (-) hold the previous value; (1) the output is high; (0) the output is low. Note that it requires 16 minterms to cover the 4 input rails, but there are only 9 valid codewords combinations. The missing minterms can be associated to any value on the output rails. This is not a problem since invalid codewords are not expected to occur during normal circuit operation and this allows us to simplify the circuit implementing this code table.

It is interesting to analyse each line type in the code table. The first line captures the reset phase of the component, when both inputs are null the output rails are driven low. The next four lines state that if any of the inputs are null the output must hold its previous value. In the next four lines all inputs are valid. These lines capture the behaviour of the

3NCL			Dual-rail RTZ					
a	b	y	a.t	a.f	b.t	b.f	y.t	y.f
N	*N*	*N*	0	0	0	0	0	0
N	*F*	-	0	0	0	1	-	-/0
N	*T*	-	0	0	1	0	-	-
F	*N*	-	0	1	0	0	-	-/0
T	*N*	-	1	0	0	0	-	-
F	*F*	*T*	0	1	0	1	1	-/0
F	*T*	*T*	0	1	1	0	1	-/0
T	*F*	*T*	1	0	0	1	1	-/0
T	*T*	*F*	1	0	1	0	-/0	1

Figure 5.4: Dual-rail RTZ code table of the `NAND2` gate with equivalent 3NCL codewords. Here, (-) means hold the previous value and (-/0) means that either: hold the previous value, or set it low.

component evaluation phase. When the component is expected to set the output to *T* it drives the true rail high and when evaluating to *F* it drives the false rail high.

Note that some lines indicate that one of the output rails can either hold its previous value or go to a certain specific value. This is allowed because the protocol alternates evaluation and reset phases, thus guaranteeing that under correct operating condition the value of some output rails can be assumed. For instance, the reset phase always resets the rails to a known null value prior to the evaluation phase. Therefore, driving a single rail and holding the state of the other during evaluation is enough to produce a valid dual-rail codeword on the output. A similar situation occurs on the reset phase, when the evaluation is known to not have changed a certain output rail. For instance, in the gate in question, the false rail is only risen when both input codewords are *T*, therefore it must only hold its state on the reset phase if one of the inputs are *T*. If one input is *F* and the other is *N*, it is safe to assume that the false output rail is 0.

From this code table it is possible to establish two hysteretic functions, each implementing the behaviour of an output rail. Remembering from Section 3.1, an hysteretic function can be described using three minterm sets: (i) the ON-set contain all minterms that force the output high (1); (ii) the OFF-set contains all minterms that force the output low (0); and (iii) the HOLD-set contain all minterms that keep the previous value in the output (-). The *output rail functions* are depicted in Figure 5.5. Each of them captures the behaviour of the output rails in the code table. These functions are minimised, but they still respect the values on the code table. For instance, the function of the output rail, depicted in Figure 5.5a, depends only on two variables but it satisfies all values associated with the false rail on Figure 5.4.

Functions operating on the channel must by strictly unate. A strictly unate hysteretic function must change its output in response only to one direction of stimuli on its input. Seeing that during the reset phase the rails of a channel are driven in one direction and

a.t	b.t	y.f
0	0	0
0	1	-
1	0	-
1	1	1

(a) False rail.

a.t	a.f	b.t	b.f	y.t
0	0	0	0	0
0	0	?	1	-
0	0	1	?	-
?	1	0	0	-
1	?	0	0	-
?	1	?	1	1
?	1	1	?	1
1	?	?	1	1

(b) True rail.

Figure 5.5: Truth tables for the output rail functions: (a) false rail; (b) true rail. Here, (-) represents hysteretic behaviour and (?) represents that the value is a “don’t-care”.

during the evaluation phase they are driven in the opposite direction. For instance, when the channel implements a dual-rail RTZ code, during the reset phase all rails go low to encode a spacer and on the evaluation phase a single rail goes high to encode valid data. Both our output rail functions satisfy this requirement.

The logic described by the output rail functions can be realised using any QDI template that supports the required protocol. However, the Pulsar Flow uses standard EDA tools to implement SDDS-NCL circuits. Although a hysteretic function describes the behaviour of a SDDS-NCL circuit at both reset and evaluation phases, they are not supported by standard EDA tools. Remembering Section 3.1, SDDS-NCL uses virtual functions which allow standard EDA tools to select NCL and NCL+ gates. To implement the logic it is necessary to map hysteretic functions to virtual functions.

A virtual function is described by only two sets of minterms: the ON-set is the set of minterms that drive the output high; and the OFF-set is the set of minterms that drive the output low. We say that a virtual function preserves the ON-set (OFF-set) of a hysteretic function when their ON-sets (OFF-sets) are equivalent. Contrary to a hysteretic function, a virtual function does not have a HOLD-set. If the HOLD-set is not empty, the virtual function cannot preserve both ON-set and OFF-set. Therefore, when mapping an hysteretic function to a virtual function one must choose to preserve either the ON-set or the OFF-set and merge the HOLD-set to the set chosen not to be preserved.

A virtual function must capture the distinctive behaviour of the circuit evaluation phase. When using the RTO protocol the circuit reset phase is characterised by its hysteretic function ON-set and its evaluation phase by the OFF-set. Conversely, when using the RTZ protocol the circuit reset phase is characterised by its hysteretic function OFF-set and the evaluation phase by its ON-set. Therefore, the virtual function of a circuit implementing the RTZ protocol preserves the ON-set of its hysteretic function and the virtual function of a circuit implementing the RTO protocol preserves its OFF-set.

The example output rail functions are hysteretic functions that describe a circuit implementing the RTZ protocol. Therefore, the virtual functions for its output rails must preserve the ON-set of the hysteretic function and merge the HOLD-set with the OFF-set. The virtual functions for these output rails are depicted in Figure 5.6. Here, each function rises its output under the same input as the hysteresis functions. However, under the conditions that the hysteresis function holds its output, the virtual function drives its output low. This is not an issue, since the fix X-Netlist algorithm guarantees that the reset phase is correctly implemented.

a.t	b.t	y.f
0	?	0
?	0	0
1	1	1

(a) False rail.

a.t	a.f	b.t	b.f	y.t
?	0	?	0	0
0	0	?	1	0
?	1	0	0	0
?	1	?	1	1
?	1	1	?	1
1	?	?	1	1

(b) True rail.

Figure 5.6: Truth tables for the output rail virtual functions: (a) false rail; (b) true rail. Here, (?) represents that the input value is a “don’t-care”.

To map virtual functions during synthesis, these are represented as Boolean expressions. Each of these expressions are assigned to an output rail in the expansion module. This module, depicted in Listing 5.2, is implemented using SystemVerilog. The ports for the module are channels implemented using `drwire` interfaces. Input ports are declared using the `drwire.in` modport and outputs are declared using `drwire.out` modport. The acknowledgement of the output channel is connected directly to the acknowledgement of input channels. This module is instantiated by the virtual netlist during the sequential SDDS-NCL synthesis.

Listing 5.2: The `nand2` expanded module.

```
1  module nand2
2    (drwire.in a,
3     drwire.in b,
4     drwire.out y);
5
6     assign y.t = a.f & b.f |
7                  a.f & b.t |
8                  a.t & b.f;
9
10    assign y.f = a.t & b.t;
11
12    assign a.ack = y.ack;
13    assign b.ack = y.ack;
14 endmodule // nand2
```

However, to construct the virtual netlist, it is first necessary to synthesise a single-rail netlist. This single-rail netlist instantiates standard cells by name. The virtual netlist is a simple textual transformation of the single-rail netlist that preserves the name of the instantiated cells. Therefore, for the virtual netlist to instantiate components, a cell with the same name must be selected during the single-rail synthesis. For this reason, components are modelled as cells in the *component library*. This component library is presented to the synthesis tool as a Liberty file. Table 5.1 presents the list of available components in the library. `nand2` and `nor2` are the most fundamental components, paired by the `inv` they can generate any combinational logic. `xor2` is provided to allow efficient implementations of arithmetic logic and `buff` is provided to allow forking a channel.

Component	Rail	Virtual Function	Transition	Virtual Delay
`nand2`	True	$y.t = (a.f \wedge b.f) \vee (a.f \wedge b.t) \vee (a.t \wedge b.f)$	Rise	20 ps
	False	$y.f = a.t \wedge b.t$	Fall	10 ps
`nor2`	True	$y.t = a.f \wedge b.f$	Rise	10 ps
	False	$y.f = (a.t \wedge b.t) \vee (a.t \wedge b.f) \vee (a.f \wedge b.t)$	Fall	20 ps
`xor2`	True	$y.t = (a.t \wedge b.f) \vee (a.f \wedge b.t)$	Rise	15 ps
	False	$y.f = (a.t \wedge b.t) \vee (a.f \wedge b.f)$	Fall	15 ps
`inv`	True	$y.t = a.f$	Rise	0 ps
	False	$y.f = a.t$	Fall	0 ps
`buff`	True	$y.t = a.t$	Rise	0 ps
	False	$y.f = a.f$	Fall	0 ps

Table 5.1: Combinational gates of the component library.

The synthesis tool is driven by timing, it attempts to synthesise the circuit with the smallest area that meets the timing constraint. This work takes advantage of this to guide the single-rail synthesis flow to produce a virtual netlist that is simpler to synthesise. For that, virtual delay and area values are assigned to combinational elements of the component library, according to the complexity of their virtual function expansions. This heuristics attempts to produce an expanded dual-rail netlist with simpler virtual functions for each rail.

Table 5.1 also presents the virtual delay of combinational components in the library. The virtual delay corresponding to the true rail virtual function is expressed in the rise transition. Conversely, the virtual delay corresponding to the false rail is expressed in the fall transition. The timing estimations are based on the minimal number of 2-input gates required to activate the virtual function in a disjunctive normal form, for each OR gate is attributed a 5 ps delay and for each AND gate is attributed a 10 ps delay. For instance, consider that the `nand2` gate, activating its true rail virtual function requires activating one AND gate and two OR gates, totalling 20 ps. In contrast, the `inv` component, when using dual rail code, is basically two crossed wires, it does not introduce any additional logic to channels, therefore it has 0-delay.

5.3.2 Sequential Components

Another class of components used during the dual-rail expansion are sequential components. These implement registers in asynchronous pipelines. They are modelled as flip-flops on the single-rail synthesis, allowing standard synthesis tools to instantiate them from canonical RTL constructions. The sequential components expand to dual-rail pseudo-synchronous WCHB registers, providing endpoints to be used during the sequential SDDS-NCL synthesis. Sequential elements bind to `drwire` channels actively, performing handshake on their input and output channels. There are three sequential components in the Library: (i) the `dff` half-buffer dual-rail RTZ register; (ii) the `dffr` resetable full-buffer component; and (iii) the `dffs` settable full-buffer component. These components are explored in the next paragraphs.

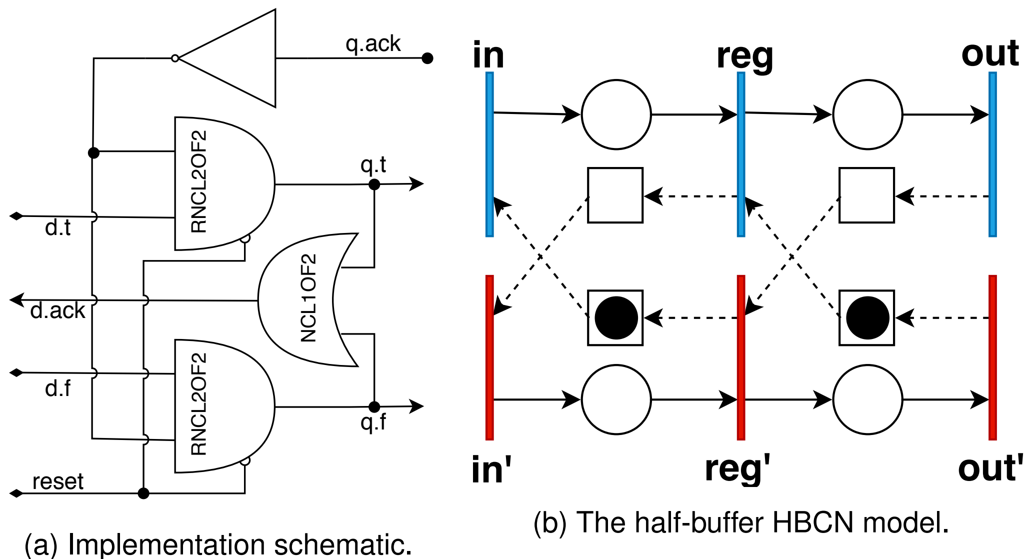
From a high-level perspective, registers allow sequential parallelism. They control the propagation of tokens in the pipeline, allowing subsequent stages to perform handshake concurrently. A half-buffer register achieves this by allowing its input and output channels to be in opposite handshake phases. The half-buffer register latches tokens, either valid or null, until both channels finish their handshake phase. This is controlled by incoming output acknowledgement and data availability in the inputs. When the register latches a new token on its output channel, it acknowledges the reception to the channel, thus allowing a new handshake phase to occur.

The implementation of a half-buffer dual-rail RTZ register is depicted in Figure 5.7a. The half-buffer register is the simplest sequential component and is the base for constructing pipelines. It connects to two channels, an input channel (d) and an output channel (q). The half-buffer consists of: two resetable C-Elements, here named using the NCL-style threshold gate denomination RNCL2OF2; one OR gate (NCL1OF2); and an inverter. When the output channel acknowledges the reception of a null (valid) token, the resetable C-Elements can latch an incoming valid (null) token. This register operates on RTZ channels, valid token codewords are one-hot and null token codewords are presented by all bits in 0. Therefore, an OR gate acknowledges when either a valid or null token has been latched in the input channel. A null token is acknowledged by lowering the ack wire of the input channel and a valid token is acknowledged by rising the same signal. The inverter on the output ack wire enables the alternation between reset and evaluation phases.

During initialisation it is important to place the circuit in a known state. This is due the fact that all components in the circuit consist of gates with hysteresis, which start at unknown initial states. A QDI circuit with an unknown state may not operate properly, as it may contain invalid codewords and possibly inconsistent handshake signal values on its channels. Both RTZ and RTO protocols require that combinational components outputs are

null prior to entering the evaluation phase. Thus, it is important to initialise all combinational components in the circuit by propagating the null codeword.

For the reason stated above, half-buffer registers employ resetable C-Elements that initialise their output channels to null codewords. Null codewords propagate through combinational components in cascade, placing the forward propagation logic in a well-known state. Similarly, the backward propagation logic on the input channel must also be initialised. When the register initiates the output rails low, the OR gate sets the acknowledgement signal of the input channel low. This signals that the channel is ready to receive new data, an information which cascades in the backward propagation of the inbound channel, initialising it. This is evident when the HBCN model depicted in Figure 5.7b is analysed. Here, the inbound channel is represented by four places preceding the register and the outbound by four places succeeding the register. The initial marking represents the initial state of the channel, it marks that both channels are ready to accept new data tokens.



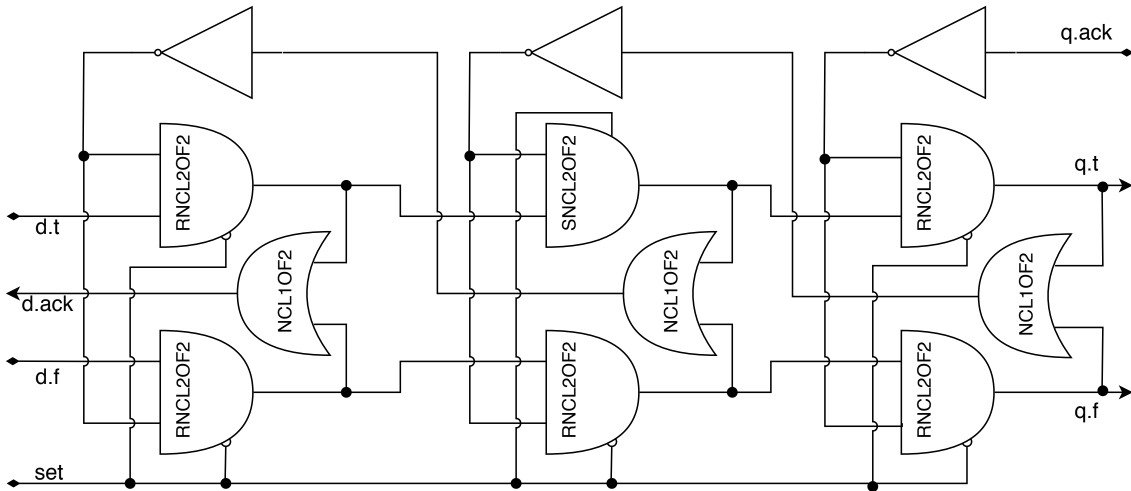
(a) Implementation schematic.

(b) The half-buffer HBCN model.

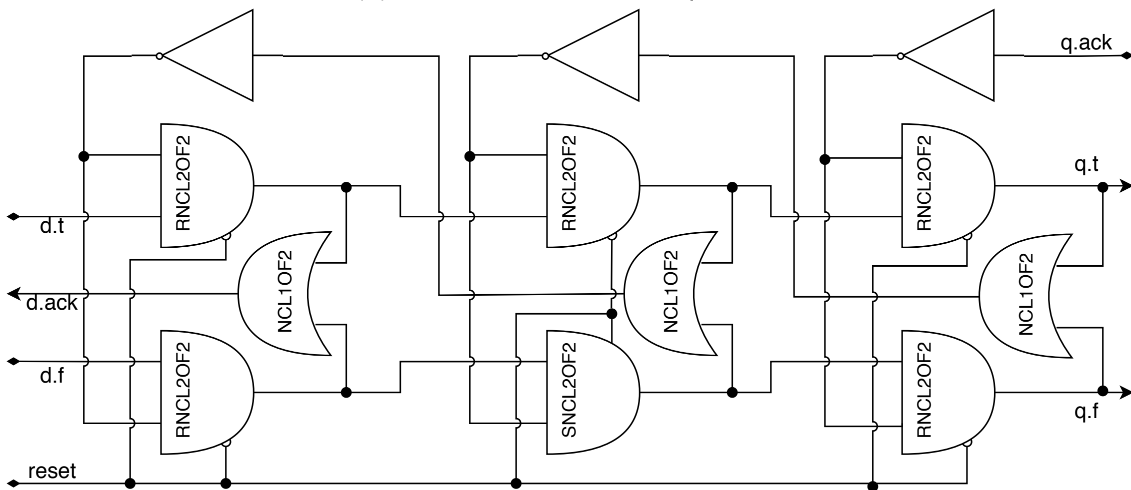
Figure 5.7: The `dff` half-buffer dual-rail register: (a) schematic; (b) HBCn model.

Sometimes it is necessary to initialise a circuit with data. For example, a counter must initialise to a known data value. This implies initialising some channels with valid tokens. This can be achieved using the full-buffer components depicted in Figure 5.8. The settable full-buffer component, depicted in Figure 5.8a, places a valid true data token in the circuit. The resetable full-buffer component, depicted in Figure 5.8b, places a valid false data token in the circuit. A full-buffer component can simultaneously hold a valid and a null token. This separates their inbound and outbound channels by a full handshake cycle.

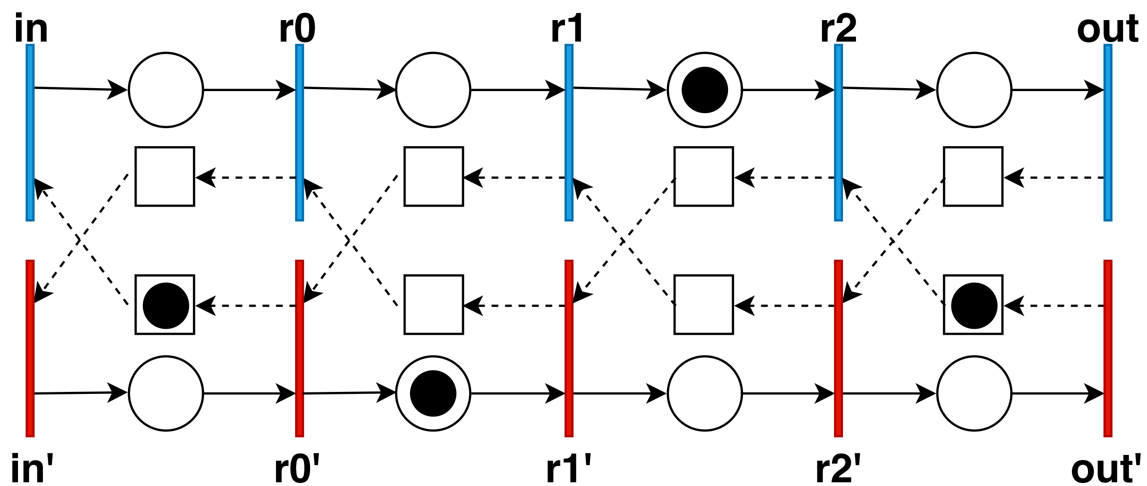
The full-buffer component comprises three half-buffer registers in sequence. These are required to place valid and a null token in the pipeline while correctly initialising the inbound and outbound channels. Propagating valid tokens in a circuit at an unknown state would yield invalid results. Therefore, the first and the last are regular half-buffer registers that reset to null. These two registers are responsible for initialising the inbound and out-



(a) The `dffs` settable component.



(b) The `dffr` resettable component.



(c) The full-buffer HBCN model.

Figure 5.8: The two full-buffer components: (a) DFF with set ; (b) DFF with reset ; (c) the HBCN model.

bound channels. Due to the provided isolation, the middle register can safely reset to a valid token without compromising circuit initialisation. This is implemented by instantiating a settable C-Element (SNCL2OF2) for either the true or the false rail depending on the component.

The behaviour of full-buffer components is further evidenced by an analysis of its HBCN model, depicted in Figure 5.8c. Here, the inbound and outbound channels are initialised to a state where both are ready to accept new tokens, similar to the initial state of the half-buffer register. However, two channels internal to the component are initialised to a valid and a null tokens. These internal channels contain no logic, thus they do not need to be initiated by a null token.

The settable and resettable C-elements employed in full-buffer components are pseudo-flop instances. Remembering Section 3.2, a pseudo-flop allows breaking the cycles of WCHB pipelines and using STA to analyse the forward and backward propagation paths. This is important during the sequential SDDS-NCL synthesis. However, a commercial EDA tool cannot infer these gates from virtual functions, as it does with NCL and NCLP gates employed in combinational component expansions. Therefore, these pseudo-flops are manually instantiated in the SystemVerilog module implementing the component expansion. This makes the implementation of sequential component expansions technology dependent.

Just as in the case of combinational components, the sequential components are also instantiated by name during the single-rail synthesis. They are modelled as D-type flip-flops in the component library Liberty file. The settable and resettable full-buffer components are modelled as flops with preset and reset, respectively. The half-buffer register is modelled as a D-type flip-flop with neither reset nor preset control signals. This approach contrasts with Uncle [RST12], where half-buffer registers are modelled as latches. Note that according to the author's experience, standard EDA tools do not support retiming latch circuits. Therefore, modelling sequential components as flops additionally enables performing retiming during single-rail synthesis. This balances the amount of components employed in each pipeline stage and opens new opportunities for optimisation in early synthesis steps.

5.4 A Case Study: Applying the Dual Rail Expansion Flow

After presenting the building blocks of the design capture and dual-rail expansion processes, it is now possible demonstrating the dual rail expansion flow with a simple example. This example assumes the synthesis of a 2-bit 2-stage adder, an example simple enough for the reader to follow through all steps of the expansion.

The starting point for synthesis is the Verilog RTL description. Listing 5.3 depicts the example RTL. The description must contain a clock port named `clk`. This clock is used

to infer rise-sensitive flops that translate to sequential components. Optionally, an active-low reset port named `reset` can be used to infer full-buffer components.

Listing 5.3: Input RTL example.

```
1  module adder(a, b, out, clk);
2      input wire [1:0] a, b;
3      output reg [1:0] out;
4      input wire       clk;
5
6      always @(posedge clk)
7        out <= a + b;
8  endmodule // adder
```

The RTL code is synthesised using Genus with a nought period clock constraint and retiming. The script used for this synthesis is listed in Appendix A. The output of the single-rail synthesis is the netlist depicted in Listing 5.4. It instantiates the components by name. Note the number of dff instances, there should be enough to have only a two-bit register. However, Genus retiming engine considered advantageous to break one of the flops in two to balance logic between the pipeline stages (see the code lines starting with **dff retime_**).

Listing 5.4: Single-rail netlist example, reordered for clarity.

```
1  module adder(a, b, out, clk);
2    input [1:0] a, b;
3    input clk;
4    output [1:0] out;
5    wire [1:0] a, b;
6    wire clk;
7    wire [1:0] out;
8    wire n_0, n_1, n_2, n_3, n_4, n_5;
9
10   xor2 g176__4296(.a (a[0]), .b (b[0]), .y (n_3));
11   nand2 g178__1474(.a (a[0]), .b (b[0]), .y (n_1));
12   dff out_reg_0_(.ck (clk), .d (n_3), .q (out[0]));
13   dff retime_s1_2_reg(.ck (clk), .d (n_1), .q (n_4));
14   inv g179(.a (a[1]), .y (n_0));
15   xor2 g177__3772(.a (n_0), .b (b[1]), .y (n_2));
16   dff retime_s1_1_reg(.ck (clk), .d (n_2), .q (n_5));
17   xor2 g172__8780(.a (n_4), .b (n_5), .y (out[1]));
18 endmodule
```

From the single-rail netlist, the *DRExpander* program constructs a virtual netlist. Appendix B lists the Haskell source code for this program. DRExpander parses the Verilog netlist, replaces every wire in the netlist by instances of the `drwire` interface. Buses of wires, such as `a`, `b` and `out` are broken in multiple `drwire` instances.

DRExpander is also responsible for recreating the module ports. Each port of the single-rail netlist is expanded to three ports suffixed by `_t`, `_f` and `_ack`, which are respectively the true, false and acknowledgement wires of the channel. The DRExpander connects

these ports to the internal wires of the respective drwire instance. This provides a consistent interface for the asynchronous module to connect with the external world.

If the single-rail netlist does not include a reset port, the DRExpander creates one and connects it to all sequential components in the virtual netlist. This is important to reset half-buffer registers. Remember that half-buffer registers are modelled as D-flip-fops with no reset. On the single-rail netlist, the dff instances do not contain a reset pin. However, their expanded module requires a reset to function properly.

Finally, the virtual netlist contains a clock port clk that connects to all its sequential components. This is the pseudo-clock signal, used to guide the synthesis during the sequential pseudo-synchronous SDDS-NCL flow. This signal is not implemented in the final circuit, it is only present to guide EDA tools to perform the QDI synthesis correctly.

Listing 5.5: Virtual netlist example, edited for clarity.

```
1  module adder
2     (a_t, a_f, a_ack,
3      b_t, b_f, b_ack,
4      out_t, out_f, out_ack,
5      clk, reset);
6     input [1:0] a_t, a_f;
7     output [1:0] a_ack;
8     input [1:0]  b_t, b_f;
9     output [1:0] b_ack;
10    output [1:0] out_t, out_f;
11    input [1:0]  out_ack;
12    input  clk, reset;
13
14    // Instances of drwire for internal channels
15    drwire n_0 ();
16    drwire n_1 ();
17    drwire n_2 ();
18    drwire n_3 ();
19    drwire n_4 ();
20    drwire n_5 ();
21    drwire a_0 ();
22    drwire a_1 ();
23    drwire b_0 ();
24    drwire b_1 ();
25    drwire out_0 ();
26    drwire out_1 ();
27
28    // Connections between external ports and drwire channels
29    assign a_0.t = a_t[0];
30    assign a_0.f = a_f[0];
31    assign a_ack[0] = a_0.ack;
32    assign a_1.t = a_t[1];
```

```
33      assign a_1.f = a_f[1];
34      assign a_ack[1] = a_1.ack;
35      assign b_0.t = b_t[0];
36      assign b_0.f = b_f[0];
37      assign b_ack[0] = b_0.ack;
38      assign b_1.t = b_t[1];
39      assign b_1.f = b_f[1];
40      assign b_ack[1] = b_1.ack;
41      assign out_t[0] = out_0.t;
42      assign out_f[0] = out_0.f;
43      assign out_0.ack = out_ack[0];
44      assign out_t[1] = out_1.t;
45      assign out_f[1] = out_1.f;
46      assign out_1.ack = out_ack[1];
47
48      // Component Instances
49      xor2 g176__4296 (.a(a_0), .b(b_0), .y(n_3));
50      nand2 g178__1474 (.a(a_0), .b(b_0), .y(n_1));
51      dff out_reg_0_ (.reset(reset), .ck(clk), .d(n_3), .q(out_0));
52      dff retime_s1_2_reg (.reset(reset), .ck(clk), .d(n_1), .q(n_4));
53      inv g179 (.a(a_1), .y(n_0));
54      xor2 g177__3772 (.a(n_0), .b(b_1), .y(n_2));
55      dff retime_s1_1_reg (.reset(reset), .ck(clk), .d(n_2), .q(n_5));
56      xor2 g172__8780 (.a(n_4), .b(n_5), .y(out_1));
57  endmodule // adder
```

5.5 The HBCN Construction and the Cycle Time Constraining

The Pulsar flow provides a tool that computes the pseudo-synchronous constraints used during the sequential pseudo-synchronous synthesis. This tool computes the HBCN model that is instrumental to automatically produce the constraints to use during the QDI synthesis flow.

The HBCN model is calculated from a structural graph extracted from the single-rail netlist. The structural graph is a directed graph that describes the single-rail netlist timing paths. In this graph, vertices represent sequential components or ports, and each edge in the graph represents a channel connecting two registers. Since combinational components are transparent to the handshaking process, they are abstracted in this representation. Each vertex has a name, identifying the component it represents. There are three types of vertices: (i) `Port`, as the name implies, used to identify ports; (ii) `NullReg`, used to identify half-buffer components; and (iii) `DataReg`, used to identify full-buffer components. The structural graph adjacency list is exported to a file during single-rail synthesis. Each line of this file represents a vertex containing its type, name and a list of successor vertices names.

Recalling Section 4.3, an HBCN has a formal representation and a marked graph representation. Using the HBCN formal representation, each register or port corresponds to a transition pair; each channel is a relation between two transition pairs; and each channel has a initial state. The rules governing the relationship between the HBCN formal representation and its characteristic marked graph is covered in Definition 4.3.1.

Since the expansion process is well-defined, it is possible to use the structural graph to construct the HBCN. To this end, it suffices to traverse the structural graph, constructing the HBCN based on the expansion of the components represented by the vertices. For each `Port` vertex a single transition pair is created, which models the (ideal) environment[1]. Each `NullReg` vertex represents a half-buffer register, which is modelled by a transition pair. Each `DataReg` vertex represents a full-buffer component. These comprise three registers in sequence. This is modelled with a sequence of three transitions pairs connected by two channels, one channel initialising to req_{null} and the others initialising to req_{data}. For each edge, a channel is created from the last transition pair corresponding to the source vertex to the first transition pair corresponding to the target vertex.

To understand this process, it is possible to rely on a simple example. The RTL for the example circuit is depicted in Listing 5.6. This is a simple circuit, a 3-stage full-buffer loop, an xor gate and a half-buffer output register. The virtual netlist for this circuit is depicted in Listing 5.7. This was generated using Genus and the DRExpander program, as discussed in the previous Section. The associated structural graph is depicted in Listing 5.8. This was automatically generated by a TCL script running within Genus at the end of the single-rail synthesis. The TCL script code is depicted in Appendix A.

Listing 5.6: RTL for the simple loop example circuit.

```
1  module xorexp(in, out, clk, reset);
2     input wire in;
3     output reg out;
4     input wire clk, reset;
5     reg r;
6
7     always @(posedge clk or negedge reset)
8       if (!reset)
9         r <= 0;
10      else
11        r <= in^r;
12
13    always @(posedge clk)
14      out <= r;
15 endmodule // xorexp
```

Here, it is possible to notice the relationship between a virtual netlist and the associated structural graph. In the virtual netlist, the channel from the `in` port connects to the

[1] An *ideal* environment provides a input token immediately upon request by any circuit input, and immediately consumes every token produced at any circuit output.

Listing 5.7: Virtual netlist for the simple loop example circuit.

```
1  module xorexp(in_t, in_f, in_ack, out_t, out_f, out_ack, clk, reset);
2      input  clk, reset;
3      input  in_t, in_f;
4      output in_ack;
5      input  out_t, out_f;
6      output out_ack;
7      drwire in ();
8      drwire n_0 ();
9      drwire out ();
10     drwire r ();
11     assign in.t = in_t;
12     assign in.f = in_f;
13     assign in_ack = in.ack;
14     assign out.t = out_t;
15     assign out.f = out_f;
16     assign out_ack = out.ack;
17     dffr r_reg (.rb(reset), .ck(clk), .d(n_0), .q(r));
18     xor2 g19__8780 (.a(r), .b(in), .y(n_0));
19     dff out_reg (.reset(reset), .ck(clk), .d(r), .q(out));
20 endmodule // xorexp
```

`xor2` combinational component, the output channel of which connects to the `r_reg` full-buffer sequential component. This path is covered by in first line of the structural graph, where the `in` port is declared and `r_reg` is assigned as its successor. `r_reg` is a full-buffer sequential component. The output channel feeds both the `out_reg` half-buffer register and the `xor2` combinational component. The `xor2` component in turn feeds back the `r_reg` forming a loop. This is expressed in the third line of the structural graph, where `out_reg` is declared as a full-buffer sequential component and lists `out_reg` and `r_reg` itself as its successors. The fourth line declares `out_reg` as a half-buffer register and lists the output port `out` as its sole successor. Finally, the last line declares the output port `out`. Since it is an output port, this is a leaf node in the graph. Accordingly, its successor list is empty.

Listing 5.8: Structural graph for the simple loop example circuit.

```
1 Port "port:xorexp/in" ["inst:xorexp/r_reg"]
2 DataReg "inst:xorexp/r_reg" ["inst:xorexp/out_reg","inst:xorexp/r_reg"]
3 NullReg "inst:xorexp/out_reg" ["port:xorexp/out"]
4 Port "port:xorexp/out" []
```

Figure 5.9 depicts the HBCN constructed from this structural graph. This HBCN explicitly shows the 3-stage full-buffer component forming a loop, the input and output transition pairs and the half-buffer register transition pair between the loop and the output. Each channel is represented by four places and the marking indicates the initial state of the channels. A merge between two channels, such as the one occurring inside the `xor2` component, is captured by the arrival of multiple channels to the same transition pair. This occurs e.g. between `in`, `r_reg/sout` and `r_reg`. A fork happens when a channel feeds multiple compo-

nents, which is captured in the HBCN as multiple channels leaving a transition pair. This is depicted between `r_reg/sout`, `out_reg` and `r_reg`.

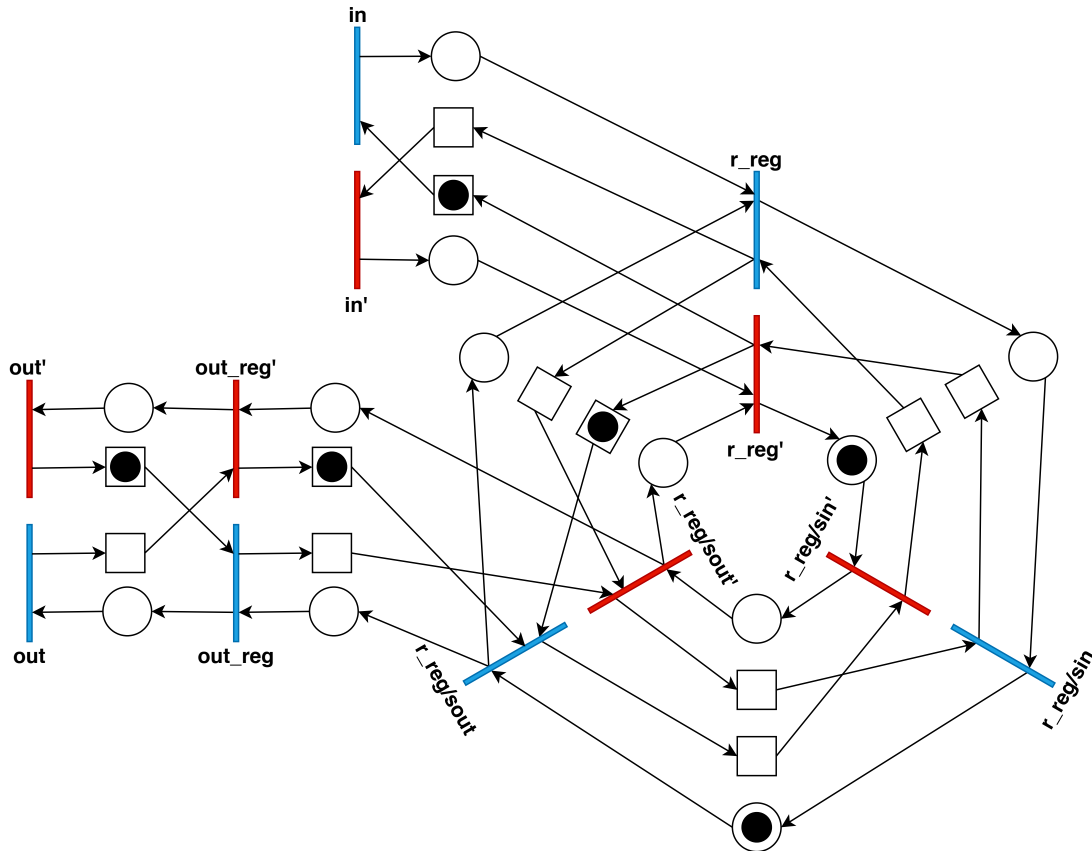


Figure 5.9: The HBCN extracted from the structural graph depicted in Listing 5.8.

After constructing the HBCN, it is possible to apply the linear programming (LP) technique presented in Section 4.5 to compute the pseudo-clock and timing exceptions. It is desirable to use the pseudo-clock to constrain the delays of combinational components in the circuit. However, the full-buffer component contains two internal channels comprising no logic. These internal channels can be individually constrained to a minimal delay value. This minimal delay is parameterisable and depends on the target technology, but it must be enough to cover the delay of a C-element and a NOR gate. The constraining of these paths to a minimal delay affects the computation of the pseudo-clock. As discussed in Section 4.5, if these take part in a critical cycle, they can allow the pseudo-clock constraint to assume a more relaxed value.

The process of computing the timing constraints is handled by the *hbcnConstrainer program*. The Haskell source code for this program is listed in Appendix C. This program computes the HBCN from the structural graph. It uses the HBCN to define the system of arrival equations constraining the cycle time to a specified target cycle time constraint. The program invokes the GLPK [Fre12] LP solver to solve the system of arrival equations. From the solution provided by GLPK, it produces a Synopsys design constraints (SDC) file. This SDC file contains the pseudo-clock constraint used during the synthesis of the virtual netlist.

Listing 5.9 depicts the SDC file that constrains the example circuit to a cycle time of 2 ns. This file was generated automatically by the hbcnConstrainer program from the structural graph in Listing 5.8 The first three lines define the pseudo-clock constraint. The remaining lines constrain the internal channels of the full-buffer components. The program was parameterised to constrain these paths to 100 ps.

Listing 5.9: SDC file to constrain the simple loop example cycle time to 2 ns.

```
1  create_clock -period 0.500 [get_port {clk}]
2  set_input_delay -clock {clk} 0 [all_inputs]
3  set_output_delay -clock {clk} 0 [all_outputs]
4  # Forward Delay from inst:xorexp/r_reg to inst:xorexp/r_reg/sin
5  set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/t} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/t}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.100
6  set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/f} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/f}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.100
7  set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/t} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/f}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.100
8  set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/f} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/t}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.100
9  # Forward Delay from inst:xorexp/r_reg/sin to inst:xorexp/r_reg/sout
10 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/t} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sout/t}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.100
11 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/f} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sout/f}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.100
12 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/t} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sout/f}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.100
13 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/f} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sout/t}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.100
14 # Backward Delay from inst:xorexp/r_reg/sin to inst:xorexp/r_reg
15 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/t} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/t} -filter
       {(is_clock_pin==false) && (direction==in)}] 0.100
16 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/f} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/f} -filter
       {(is_clock_pin==false) && (direction==in)}] 0.100
17 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/t} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/f} -filter
       {(is_clock_pin==false) && (direction==in)}] 0.100
```

```
18 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/f} -filter
     {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/t} -filter
     {(is_clock_pin==false) && (direction==in)}] 0.100
19 # Backward Delay from inst:xorexp/r_reg/sout to inst:xorexp/r_reg/sin
20 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sout/t} -filter
     {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/t}
     -filter {(is_clock_pin==false) && (direction==in)}] 0.100
21 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sout/f} -filter
     {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/f}
     -filter {(is_clock_pin==false) && (direction==in)}] 0.100
22 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sout/t} -filter
     {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/f}
     -filter {(is_clock_pin==false) && (direction==in)}] 0.100
23 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sout/f} -filter
     {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/t}
     -filter {(is_clock_pin==false) && (direction==in)}] 0.100
```

The hbcnConstrainer program can optionally generate timing exceptions for paths with free slack, allowing more relaxed timing constraints on paths containing free slack. The previous example was generated with the option to generate these timing exceptions disabled. Listing 5.10 depicted the SDC generated by hbcnConstrainer to constraint the same circuit to the cycle time of 2 ns with the option to relax the paths with free slack enabled. Notice that the pseudo-clock constraint does not change when enabling this option. Here, the maximum delay of each propagation path containing free slack is set by a `set_max_delay` timing exception. In this particular case, only the propagation paths internal to the full-buffer components have free slack. This is not very useful, but in more complex circuits this free slack could relax some logic stages.

Listing 5.10: SDC file to constrain the simple loop example cycle time to 2 ns.

```
1 create_clock -period 0.500 [get_port {clk}]
2 set_input_delay -clock {clk} 0 [all_inputs]
3 set_output_delay -clock {clk} 0 [all_outputs]
4 # forward delay from inst:xorexp/r_reg to inst:xorexp/r_reg/sin
5 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/t} -filter
     {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/t}
     -filter {(is_clock_pin==false) && (direction==in)}] 0.600
6 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/f} -filter
     {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/f}
     -filter {(is_clock_pin==false) && (direction==in)}] 0.600
7 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/t} -filter
     {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/f}
     -filter {(is_clock_pin==false) && (direction==in)}] 0.600
8 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/f} -filter
     {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/t}
     -filter {(is_clock_pin==false) && (direction==in)}] 0.600
9 # forward delay from inst:xorexp/r_reg/sin to inst:xorexp/r_reg/sout
```

```
10 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/t} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sout/t}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.900
11 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/f} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sout/f}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.900
12 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/t} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sout/f}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.900
13 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/f} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sout/t}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.900
14 # backward delay from inst:xorexp/r_reg/sin to inst:xorexp/r_reg
15 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/t} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/t} -filter
       {(is_clock_pin==false) && (direction==in)}] 0.400
16 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/f} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/f} -filter
       {(is_clock_pin==false) && (direction==in)}] 0.400
17 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/t} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/f} -filter
       {(is_clock_pin==false) && (direction==in)}] 0.400
18 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sin/f} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/t} -filter
       {(is_clock_pin==false) && (direction==in)}] 0.400
19 # backward delay from inst:xorexp/r_reg/sout to inst:xorexp/r_reg/sin
20 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sout/t} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/t}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.100
21 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sout/f} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/f}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.100
22 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sout/t} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/f}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.100
23 set_max_delay -from [get_pin -of_objects {inst:xorexp/r_reg/sout/f} -filter
       {is_clock_pin==true}] -to [get_pin -of_objects {inst:xorexp/r_reg/sin/t}
       -filter {(is_clock_pin==false) && (direction==in)}] 0.100
```

5.6 Completing the Case Study Synthesis

After constructing the virtual netlist, it is then possible to synthesise the virtual netlist using the pseudo-synchronous SDDS-NCL synthesis flow described in Section 3.3. This Section continues the synthesis of the example circuit explored in Section 5.4. For

that, the first step is to compute the constraints using the hbcnConstrainer from the previous Section. The structural graph of our example circuit is depicted in Listing 5.11

Listing 5.11: Structural graph of virtual netlist depicted in Listing 5.5.

```
1 Port "port:adder/a[1]" ["inst:adder/retime_s1_1_reg"]
2 Port "port:adder/a[0]" ["inst:adder/out_reg_0_","inst:adder/retime_s1_2_reg"]
3 Port "port:adder/b[1]" ["inst:adder/retime_s1_1_reg"]
4 Port "port:adder/b[0]" ["inst:adder/out_reg_0_","inst:adder/retime_s1_2_reg"]
5 NullReg "inst:adder/retime_s1_1_reg" ["port:adder/out[1]"]
6 NullReg "inst:adder/out_reg_0_" ["port:adder/out[0]"]
7 NullReg "inst:adder/retime_s1_2_reg" ["port:adder/out[1]"]
8 Port "port:adder/out[0]" []
9 Port "port:adder/out[1]" []
```

The hbcnConstrainer program creates an SDC file that constrains this circuit to a cycle time of 4 ns. This SDC file is depicted in Listing 5.12. The adder circuit is a very simple linear pipeline, it contains no free slack when constrained with a pseudo-clock constraint. Enabling the free slack relaxation options does not affect the produced SDC file.

Listing 5.12: SDC file constraining the cycle time of the pipeline adder to 4 ns.

```
1 create_clock -period 1.000 [get_port {clk}]
2 set_input_delay -clock {clk} 0 [all_inputs]
3 set_output_delay -clock {clk} 0 [all_outputs]
```

The produced SDC, along with the virtual netlist depicted in Listing 5.5 are used as input to the pseudo-synchronous SDDS-NCL synthesis flow. The virtual netlist is the primary netlist input to the synthesis flow. Another netlist input is the component expansion library. The SystemVerilog file containing the component expansions and the pseudo-synchronous synthesis scripts are depicted in Appendix D. The example circuit was synthesised using the ASCEND-ST65NCL Library [MOPC11], a library of NCL and NCLP gates targeting STMicro 65 nm technology node. The resulting netlist is depicted in Listing 5.13. This netlist was reordered and commented to highlight its equivalence to the virtual netlist.

Listing 5.13: Sequential SDDS-NCL Netlist, edited for clarity.

```
1  module adder (a_t, a_f, a_ack, b_t, b_f, b_ack,
2                out_t, out_f, out_ack, clk, reset);
3    input  wire [1:0] a_t, a_f, b_t, b_f, out_ack;
4    input  wire       clk, reset;
5    output wire [1:0] a_ack, b_ack, out_t, out_f;
6    wire          n_1, n_1_f, n_1_t, n_2_f, n_2_t, n_3, n_3_f, n_4;
7    wire          n_4_f, n_4_t, n_5, n_5_f, n_5_t, n_6, n_7, n_8;
8    wire          n_9, n_10, n_14;
9
10   // backward propagation logic
11   assign b_ack[0] = a_ack[0];
12   assign b_ack[1] = a_ack[1];
```

```
13     HS65_GS_IVX9 g43(.A (out_ack[0]), .Z (n_10));
14     HS65_GS_IVX9 g44(.A (out_ack[1]), .Z (n_9));
15     SY_INCLP2W11OF2X9 g104(.A (n_5), .B (n_4), .Q (a_ack[0]));
16
17     // xor2 g176__4296 & nand2 g178__1474 (logic sharing)
18     ST_NCLAO22OF4X9 g2(.A (a_f[0]), .B (b_t[0]), .C (a_t[0]),
19                        .D (b_f[0]), .Q (n_14));
20     ST_NCL2W11OF2X4 g97(.A (a_t[0]), .B (b_t[0]), .Q (n_1_f));
21     ST_NCL1W11OF2X9 g93(.A (n_1), .B (n_14), .Q (n_1_t));
22     ST_NCL2W11OF2X4 g99(.A (a_f[0]), .B (b_f[0]), .Q (n_1));
23     ST_NCL1W11OF2X9 g94(.A (n_1_f), .B (n_1), .Q (n_3_f));
24
25     // dff out_reg_0_
26     ST_RNCL2W11OF2X9 out_reg_0__f(.RN (reset), .G (clk), .A (n_3_f),
27                                   .B (n_10), .Q (out_f[0]));
28     ST_RNCL2W11OF2X9 out_reg_0__t(.RN (reset), .G (clk), .A (n_14),
29                                   .B (n_10), .Q (out_t[0]));
30     ST_INCL1W11OF2X9 g110(.A (out_t[0]), .B (out_f[0]), .Q (n_5));
31
32     // dff retime_s1_2_reg
33     ST_RNCL2W11OF2X9 retime_s1_2_reg_f(.RN (reset), .G (clk), .A (n_1_f),
34                                        .B (n_9), .Q (n_4_f));
35     ST_RNCL2W11OF2X9 retime_s1_2_reg_t(.RN (reset), .G (clk), .A (n_1_t),
36                                        .B (n_9), .Q (n_4_t));
37     ST_INCL1W11OF2X9 g111(.A (n_4_t), .B (n_4_f), .Q (n_4));
38
39     // inv g179 & xor2 g177_3772
40     ST_NCLAO22OF4X7 g78(.A (b_f[1]), .B (a_f[1]), .C (a_t[1]),
41                         .D (b_t[1]), .Q (n_2_t));
42     ST_NCLAO22OF4X7 g79(.A (b_f[1]), .B (a_t[1]), .C (a_f[1]),
43                         .D (b_t[1]), .Q (n_2_f));
44
45     // dff retime_s1_1_reg
46     ST_RNCL2W11OF2X9 retime_s1_1_reg_f(.RN (reset), .G (clk), .A (n_2_f),
47                                        .B (n_9), .Q (n_5_f));
48     ST_RNCL2W11OF2X9 retime_s1_1_reg_t(.RN (reset), .G (clk), .A (n_2_t),
49                                        .B (n_9), .Q (n_5_t));
50     ST_NCL1W11OF2X9 g113(.A (n_5_f), .B (n_5_t), .Q (a_ack[1]));
51
52     // xor2 g172__8780
53     SY_INCL2W11OF2X9 g107(.A (n_4_f), .B (n_5_f), .Q (n_8));
54     SY_INCL2W11OF2X9 g108(.A (n_4_t), .B (n_5_t), .Q (n_7));
55     ST_INCLP1W11OF2X4 g106(.A (n_8), .B (n_7), .Q (out_f[1]));
56     SY_INCL2W11OF2X9 g109(.A (n_4_f), .B (n_5_t), .Q (n_6));
57     SY_INCL2W11OF2X9 g112(.A (n_4_t), .B (n_5_f), .Q (n_3));
58     ST_INCLP1W11OF2X4 g105(.A (n_3), .B (n_6), .Q (out_t[1]));
59 endmodule // adder
```

6. EXPERIMENTS AND RESULTS

The Pulsar flow was initially validated by synthesising and simulating a set of multiply and accumulate (MAC) units under a range of timing constraints. MACs were chosen as example circuits due to their logic complexity and non-linear pipeline structure. Non-linear pipelines are good case studies for evaluating the HBCN timing constraining capabilities, because they present non-trivial maximum cycle times. The accumulator of a MAC comprises a circular buffer. This buffer can be implemented with different number of pipeline stages. Four different MAC architectures were thus designed:

- A 3-stage MAC, comprising a 3-stage accumulator.
- A 4-stage MAC, comprising a 4-stage accumulator.
- A 5-stage MAC, comprising a 5-stage accumulator.
- A 6-stage MAC, comprising a 6-stage accumulator.

These MACs present the same interface and behaviour, they multiply two 16-bit numbers from the input and add the 32-bit result in an accumulation loop. The new accumulator value is presented to the output after every computation cycle. This allow using the same testbench for simulating all MACs.

MACs were synthesised using the Pulsar flow from RTL descriptions. The RTL for the 3-stage MAC is depicted in Listing 6.1. Both inputs and outputs are buffered with half-buffer registers. Here, the accumulator loop has the minimal number of stages required to correctly operate. Listing 6.2 presents the RTL for the 4-stage MAC. Here, a `result` half-buffer register is introduced between the input and output registers. This additional register also takes part in the accumulator loop. The RTL for 5-stage and 6-stage MACs are respectively depicted in Listing 6.3 and in Listing 6.4. Each of these respectively include a second and a third `result` registers in the pipeline. Notice that on the RTL description most pipeline stages are empty. The retiming engine takes care of distributing the logic among the empty pipeline stages during single-rail synthesis.

Single-rail synthesis was performed using the Cadence Genus 18.1 tool with retiming enabled. The optimisation effort was set to extreme and the circuit was synthesised with a nought clock period constraint. Genus was also configured to optimise the total negative slack. These settings were used to minimise the logical depth of each pipeline stage. The single-rail synthesis of the RTL descriptions resulted in virtual netlists with the characteristics summarised in Table 6.1. Here, the worst virtual delays indicate the complexity of the virtual function composition for the longest path in the circuit.

Listing 6.1: RTL code for the 3-stage MAC.

```
1  module mac #(WIDTH=32)
2     (input logic [(WIDTH/2)-1:0] a, b,
3      input logic                 clk, reset,
4      output logic [WIDTH-1:0]    out);
5     logic [(WIDTH/2)-1:0]        reg_a;
6     logic [(WIDTH/2)-1:0]        reg_b;
7     logic [WIDTH-1:0]            result;
8     logic [WIDTH-1:0]            acc;
9
10    assign result = (reg_a * reg_b) + acc;
11
12    always @(posedge clk) begin
13       reg_a <= a;
14       reg_b <= b;
15       out <= result;
16    end
17
18    always @(posedge clk or negedge reset)
19       if (!reset)
20         acc <= '0;
21       else
22         acc <= result;
23 endmodule // mac
```

Listing 6.2: RTL code for the 4-stage MAC.

```
1  module mac #(WIDTH=32)
2     (input logic [(WIDTH/2)-1:0] a, b,
3      input logic                 clk, reset,
4      output logic [WIDTH-1:0]    out);
5     logic [(WIDTH/2)-1:0]        reg_a;
6     logic [(WIDTH/2)-1:0]        reg_b;
7     logic [WIDTH-1:0]            result;
8     logic [WIDTH-1:0]            acc;
9
10    always @(posedge clk) begin
11       reg_a <= a;
12       reg_b <= b;
13       result <= (reg_a * reg_b) + acc;
14       out <= result;
15    end
16
17    always @(posedge clk or negedge reset)
18       if (!reset)
19         acc <= '0;
20       else
21         acc <= result;
22 endmodule // mac
```

Listing 6.3: RTL code for the 5-stage MAC.

```
1  module mac #(WIDTH=32)
2     (input logic [(WIDTH/2)-1:0] a, b,
3      input logic                 clk, reset,
4      output logic [WIDTH-1:0]    out);
5     logic [(WIDTH/2)-1:0]        reg_a;
6     logic [(WIDTH/2)-1:0]        reg_b;
7     logic [WIDTH-1:0]            result[2];
8     logic [WIDTH-1:0]            acc;
9
10    always @(posedge clk) begin
11       reg_a <= a;
12       reg_b <= b;
13       result[0] <= (reg_a * reg_b) + acc;
14       result[1] <= result[0];
15       out <= result[1];
16    end
17
18    always @(posedge clk or negedge reset)
19       if (!reset)
20         acc <= '0;
21       else
22         acc <= result[1];
23 endmodule // mac
```

Listing 6.4: RTL for the 6-stage MAC

```
1  module mac #(WIDTH=32)
2     (input logic [(WIDTH/2)-1:0] a, b,
3      input logic                 clk, reset,
4      output logic [WIDTH-1:0]    out);
5     logic [(WIDTH/2)-1:0]        reg_a;
6     logic [(WIDTH/2)-1:0]        reg_b;
7     logic [WIDTH-1:0]            result[3];
8     logic [WIDTH-1:0]            acc;
9
10    always @(posedge clk) begin
11       reg_a <= a;
12       reg_b <= b;
13       result[0] <= (reg_a * reg_b) + acc;
14       result[1] <= result[0];
15       result[2] <= result[1];
16       out <= result[2];
17    end
18
19    always @(posedge clk or negedge reset)
20       if (!reset)
21         acc <= '0;
22       else
23         acc <= result[2];
24 endmodule // mac
```

Circuit	Worst Virtual Delay	Component Count		
		Combinational	Sequential	Total
3-stage MAC	335 ps	2094	206	2300
4-stage MAC	205 ps	2088	267	2355
5-stage MAC	150 ps	2121	344	2465
6-stage MAC	130 ps	2086	442	2508

Table 6.1: Characteristics of the virtual netlists for each MAC version.

The final pseudo-synchronous SDDS-NCL netlists were implemented with NCL and NCLP gates from the ASCEnD [MOPC11] library for STMicroelectronics 65nm technology node. This library is characterised at three PVT corners:

- The *worst corner* has slow transistors operating under 0.9 V and 125 °C.
- The *nominal corner* has typical transistors operating under 1.0 V and 25 °C.
- The *best corner* has fast transistors operating under 1.1 V and -40 °C.

Dual-rail synthesis was performed using the worst corner. As discussed in Section 3.2, the synthesis is performed with a library that models settable and resettable C-Elements as pseudo-flops. The pseudo-flop model introduces a small error on the delay model, and a clock uncertainty of 5 ps was used to compensate for this error. Each circuit was synthesised under a range of target cycle time constraints, from 2 ns to 6 ns in steps of 250 ps. The minimal delay was set to 200 ps during the cycle time constraint computation. Synthesis were performed using Genus with the effort set to high. Physical-aware optimisation was performed with the effort set to extreme. After running the Fix X-netlist algorithm, physical-aware optimisation was performed on each set of gates iteratively, until the timing was met or a maximum number of 10 iterations was reached. For each synthesis the following dataset is collected:

- The *worst pseudo-synchronous timing slack* – a zero or positive value here guarantees that the cycle time constraint is met.
- The *gate area* – this is the area occupied by the gates selected during the synthesis.
- The *resulting pseudo-synchronous SDDS-NCL netlist* – which implements the circuit.

Each resulting pseudo-synchronous SDDS-NCL netlist was delay-annotated using the three corners of the sign-off library. The sign-off library models the settable and resettable C-elements as pseudo-latches. As discussed in Section 3.2, this library allows annotating the timing arcs with the delays extracted from the characterisation. Thus, the annotated delays are not affected by the error introduced by the pseudo-flop model. Each delay-annotated netlists was simulated with a testbench that emulates an ideal environment.

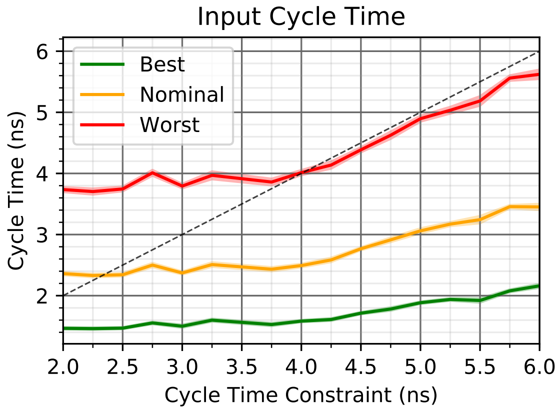
An ideal environment has no delay of itself, it feeds tokens into the pipeline as soon as an acknowledgement is received and it also acknowledges the reception of tokens instantaneously, thus isolating the circuit from external delays. Under these conditions, for each corner it was possible measuring:

- The *output cycle time* – this is the time between the reception of two data tokens at the pipeline output.
- The *input cycle time* – this is the time between the insertion of two successive data tokens in the pipeline input.
- The *latency*, – this is the time between a the insertion of a data token at the input and the reception of a corresponding data token with the result at the output.
- The averaged switching activity of the circuit – this is used in conjunction with the circuit netlist and the sign-off library as input for static power analysis (SPA).

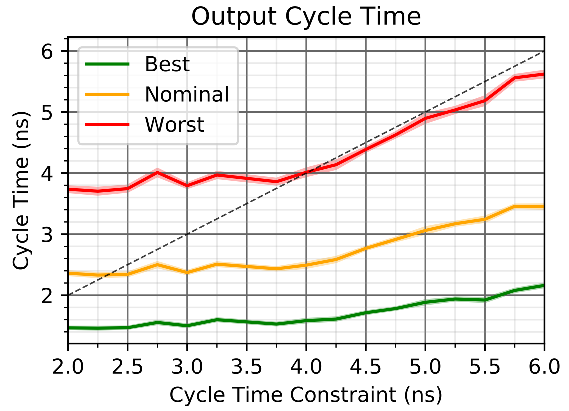
Timing measurements are subject to data-dependent delay variations, as discussed in Section 4.1. These values are presented here as an average. The standard deviation of the measured data points is presented as a range around the former average.

Results for the 3-stage MAC are depicted in Figure 6.1. They show the timing extracted through simulation and the data collected from synthesis reports. The worst pseudo-synchronous timing slack for this circuit is depicted in Figure 6.1d. It presents negative slack values for cycle time constraints under 5 ns. According to the models, a negative slack value indicates that it is not possible to guarantee that the cycle time constraint is met. This is made evident in the results of the 4-stage MAC, depicted in Figure 6.2. Here, the first negative slack value is observed on the cycle time constraint of 3 ns. At the same point, it is possible to observe a violation of the cycle time constraint for the worst corner in the timing results collected from the simulations in Figure 6.2a and 6.2b. Another interesting aspect observable when comparing the results from the 3-stage MAC and the 4-stage MAC is how distributing logic among an extra pipeline allows achieving a faster cycle time. This is consistently observable in their worst virtual delay, worst timing slacks, input and output cycle times. This increase in performance however comes at the cost of an increase in area and power. Taking this approach of distributing the logic amount pipeline stages, results for the 5-stage and 6-stage MACs are respectively depicted in Figure 6.3 and Figure 6.4.

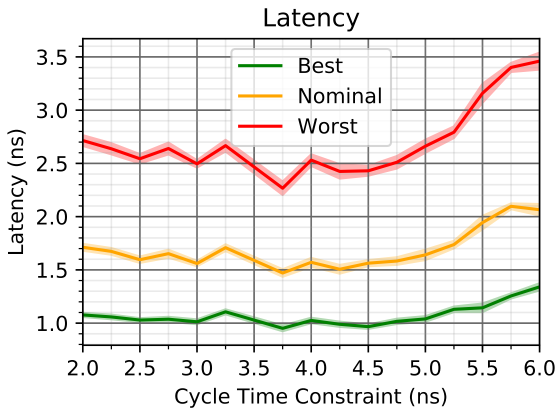
As it is possible to observe in Figure 6.4a, the cycle time observed in simulation is consistently below the cycle time constraint. This indicates that these circuits might be over-constrained. A possible solution is to enable the generation of timing exceptions that relax the timing constraints on free slack. The results for the 6-stage MAC synthesised with the relaxed timing constraints is presented in Figure 6.5. Here, observe that the measured cycle time matches more closely the cycle time constraint. This leaded to a reduction in power and area, while still matching the target cycle time.



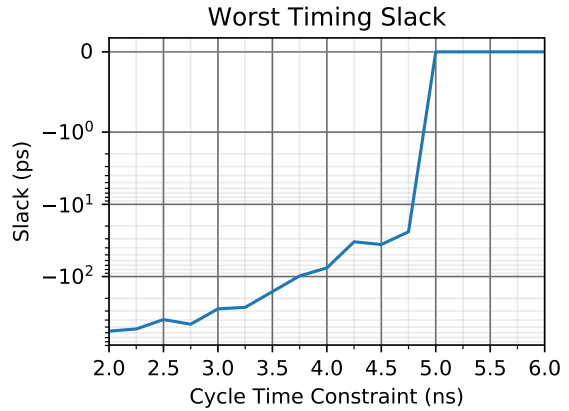
(a) Input cycle time observed in simulation, the black dashed line is the cycle time constraint.



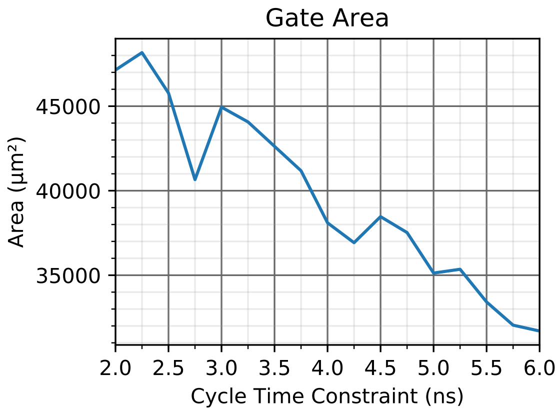
(b) Output cycle time observed in simulation, the black dashed line is the cycle time constraint.



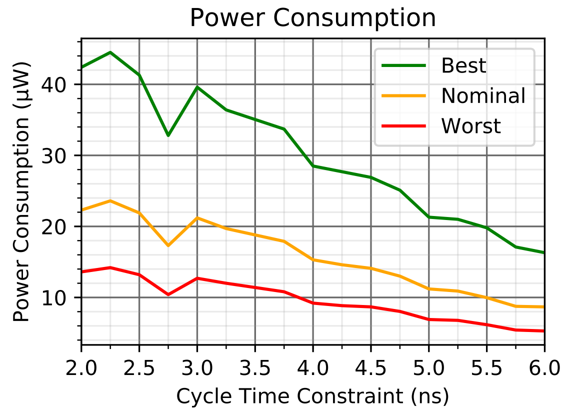
(c) Latency observed in simulation.



(d) Worst pseudo-synchronous timing slack.

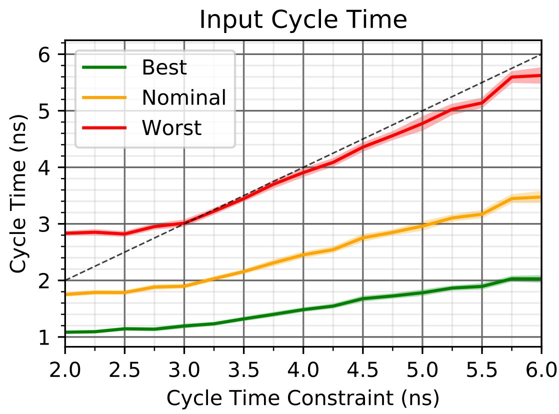


(e) Gate area extracted from synthesis reports.

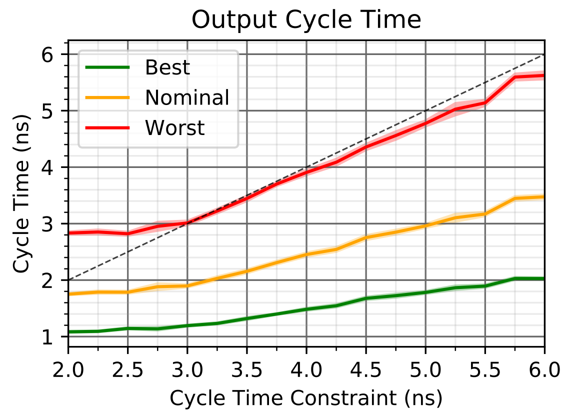


(f) Power consumption extracted using static power analysis (SPA) with switching activity from simulation.

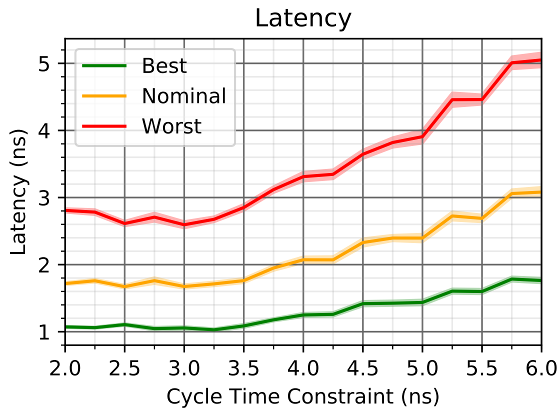
Figure 6.1: Results for the 3-stage MAC.



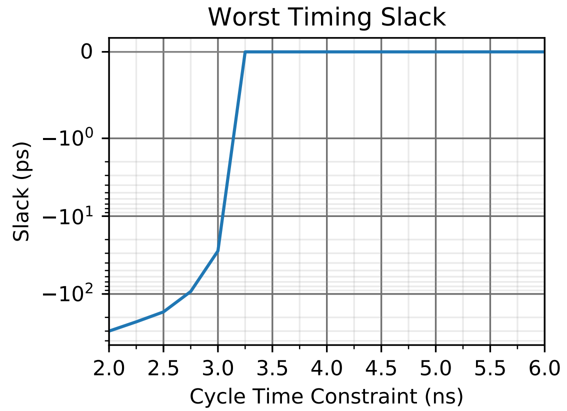
(a) Input cycle time observed in simulation. The black dashed line is the cycle time constraint.



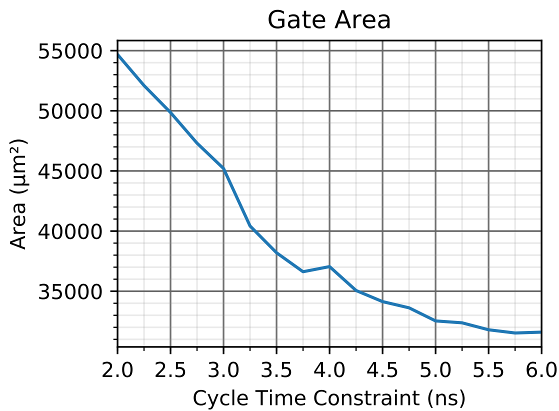
(b) Output cycle time observed in simulation. The black dashed line is the cycle time constraint.



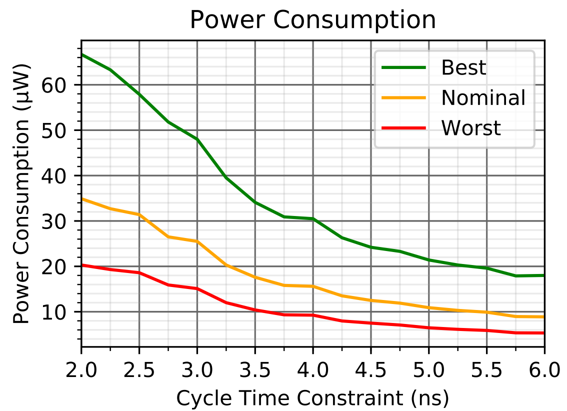
(c) Latency observed in simulation.



(d) Worst pseudo-synchronous timing slack.

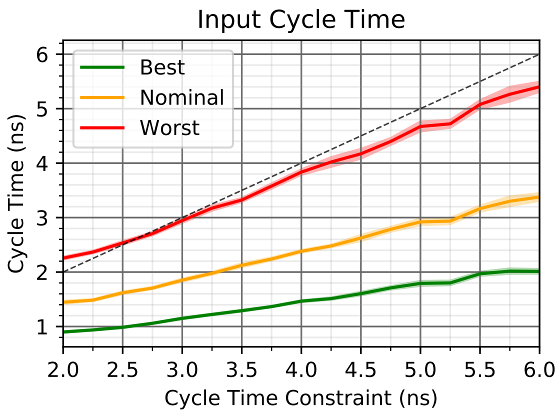


(e) Gate area extracted from synthesis reports.

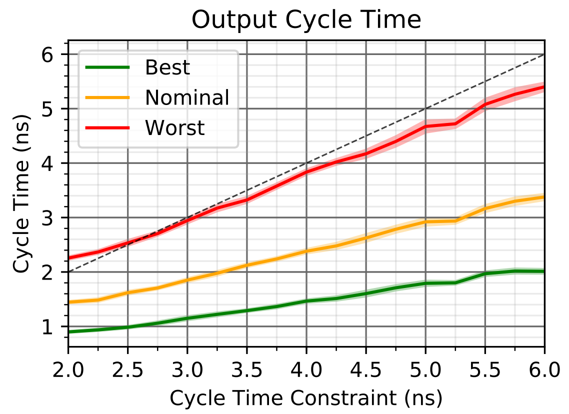


(f) Power consumption extracted using static power analysis (SPA) with switching activity from simulation.

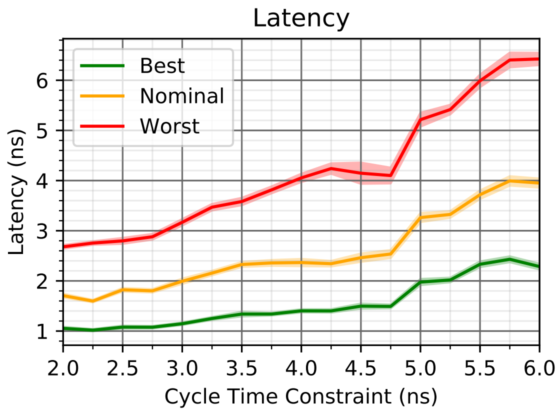
Figure 6.2: Results for the 4-stage MAC.



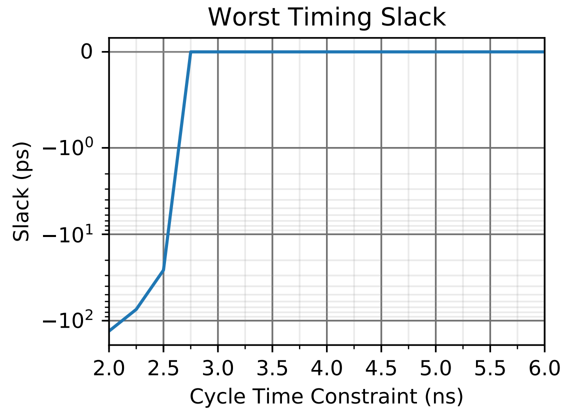
(a) Input cycle time observed in simulation, the black dashed line is the cycle time constraint.



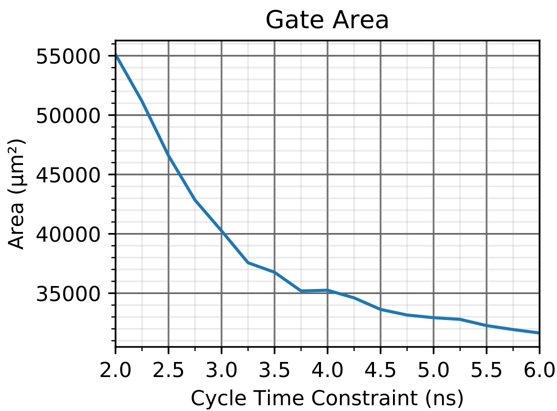
(b) Output cycle time observed in simulation, the black dashed line is the cycle time constraint.



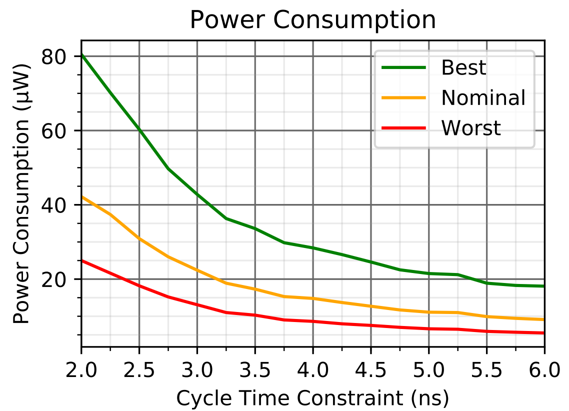
(c) Latency observed in simulation.



(d) Worst pseudo-synchronous timing slack.

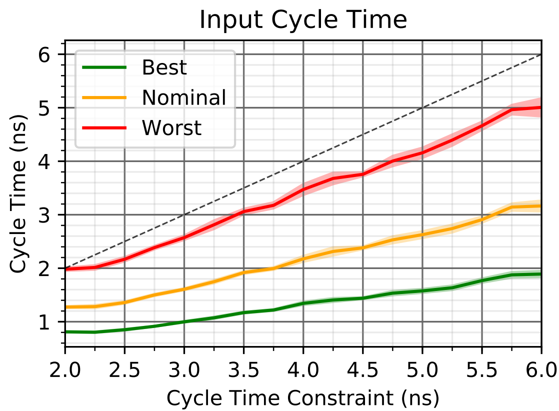


(e) Gate area extracted from synthesis reports.

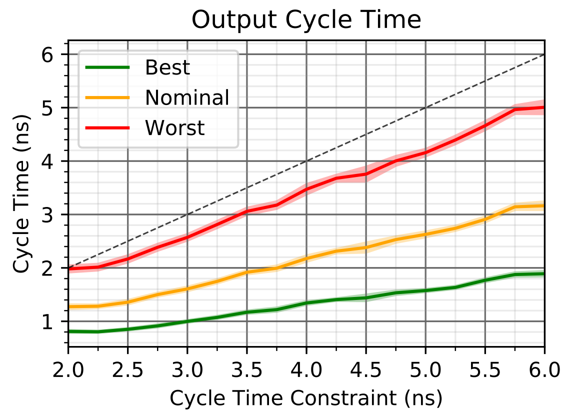


(f) Power consumption extracted using static power analysis (SPA) with switching activity from simulation.

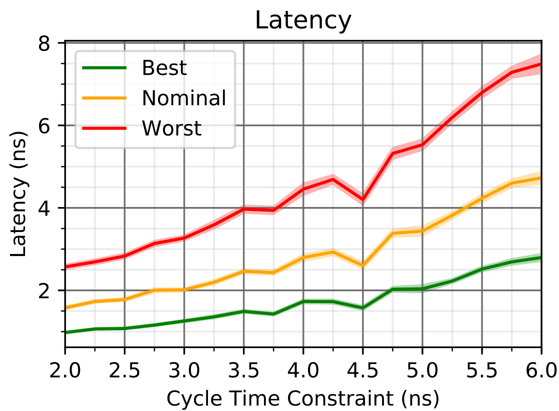
Figure 6.3: Results for the 5-stage MAC.



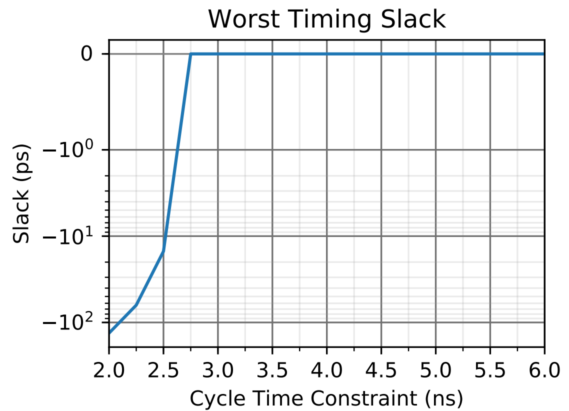
(a) Input cycle time observed in simulation. The black dashed line is the cycle time constraint.



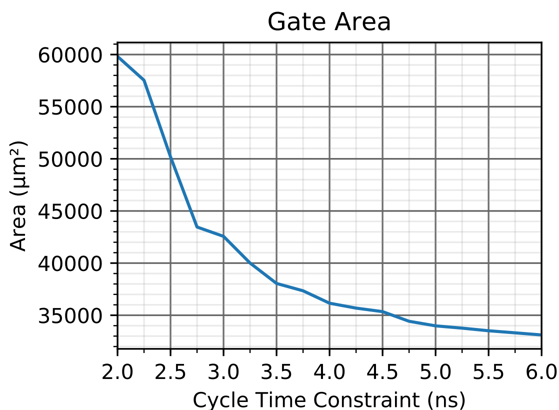
(b) Output cycle time observed in simulation, The black dashed line is the cycle time constraint.



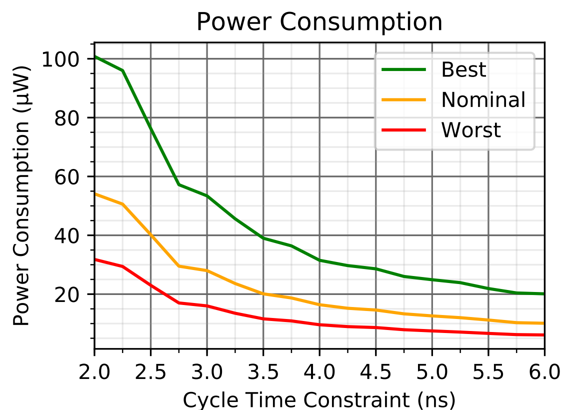
(c) Latency observed in simulation.



(d) Worst pseudo-synchronous timing slack.

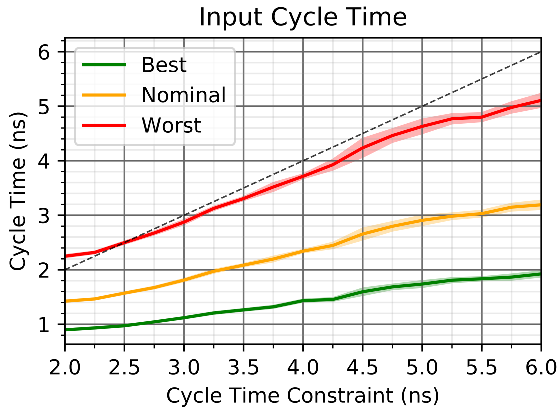


(e) Gate area extracted from synthesis reports.

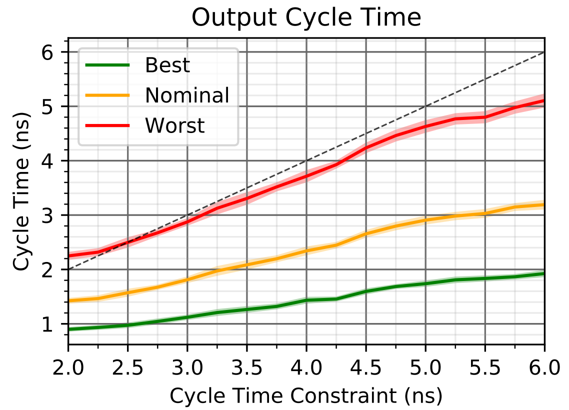


(f) Power consumption extracted using static power analysis (SPA) with switching activity from simulation.

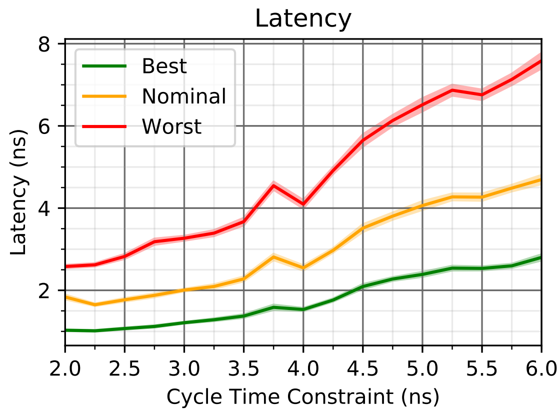
Figure 6.4: Results for the 6-stage MAC.



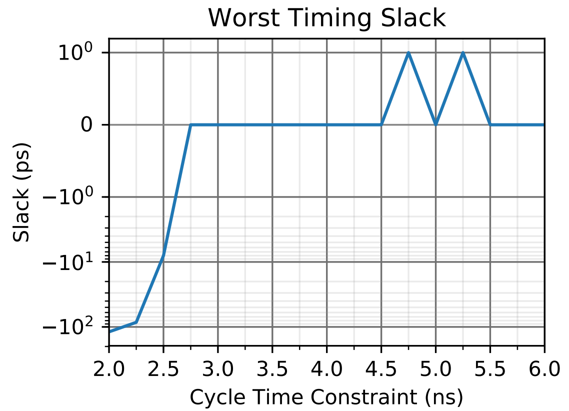
(a) Input cycle time observed in simulation. The black dashed line is the cycle time constraint.



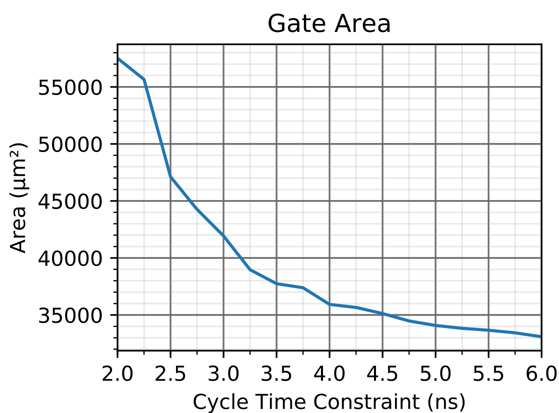
(b) Output cycle time observed in simulation. The black dashed line is the cycle time constraint.



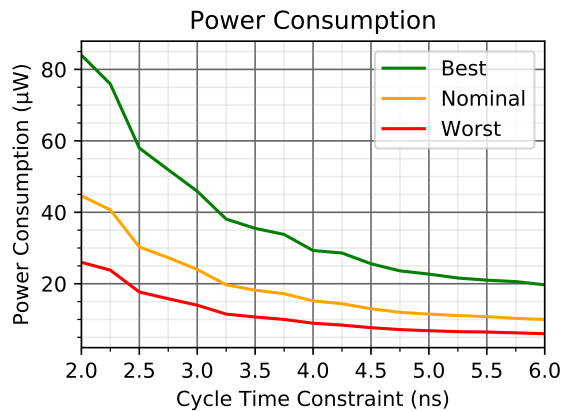
(c) Latency observed in simulation.



(d) Worst pseudo-synchronous timing slack.



(e) Gate area extracted from synthesis reports.



(f) Power consumption extracted using static power analysis (SPA) with switching activity from simulation.

Figure 6.5: Results for the 6-stage MAC synthesised with timing path exceptions on free-slack.

7. CONCLUSION AND FUTURE WORK

This Dissertation has presented Pulsar, a innovative synthesis flow for QDI circuits. Pulsar leverages commercial EDA tools for the design capture, dual-rail expansion, technology mapping and optimisation of QDI circuits. It also enables the sign-off of target cycle times for QDI circuits using commercial EDA tools. This is a major breakthrough for QDI designers, as they can now safely bound worst case performance metrics for their target applications. Moreover, the flow enables designers to naturally trade performance, power and area optimisations, whenever there is slack in timing budgets.

The use of commercial EDA tools facilitates its adoption and integration in industrial flows. First, Pulsar uses an RTL-like description for design capture, a description form familiar to most designers. In this aspect, Pulsar is very similar to Uncle [RST12], whereas both synthesise single-rail netlists that are latter expanded to a QDI circuit. The approach proposed here allows the use of commercial tools optimisation and retiming algorithms for logic, data-path, state machine synthesis. However, RTL descriptions are not fully adequate for capturing the behaviour of asynchronous circuits. They often rely on the global clock signal to synchronise the merging of data flows, making it not suitable to capture the inter-dependencies of data flows in the circuit. A common and well established approach is to use descriptions based on Hoare communicating sequential processes (CSP) [Hoa85]. For instance, Concurrent Hardware Processes (CHP) [MM11], based on CSP, is used for design capture in Beerel et al. Proteus [BDL11], Manohar ACT [Man19] and others. Another classical CSP-based language is Balsa [EB02], which has its own synthesis tool, but it is also used as a language on Teak [BTE09]. Another interesting approach is the use of functional programming languages, this is explored for synchronous circuits on works such as Lava [BCSS99, Gil11], CλasH (pronounced Clash) [BKK+10] and Chisel [BVR+12]. Also, the current design capture and dual-rail expansion technique is limited to deterministic pipelines, which limits the application of Pulsar to more complex circuits. Future work comprises exploring alternative design capture methodologies that better capture the behaviour of a broader range of asynchronous circuits.

The use of standard EDA also tools helps integrating QDI circuits in existing design flows. This makes Pulsar attractive for designing globally asynchronous locally synchronous (GALS) circuits. A GALS configuration consists of synchronous circuits interfacing with an asynchronous circuit. A GALS circuit uses an asynchronous circuit to overcome the limitations of clock distribution on synchronous circuits. It enables different modules in a circuit to operate with different clocks with possible distinct frequencies and/or phases. An asynchronous circuit acts as a interconnection between different circuit modules. For optimal performance, such an asynchronous circuit must withstand the throughput of the synchronous modules. A synchronous circuit throughput is dictated by its clock frequency. The

HBCN and the cycle time constraining methodology presented in this Dissertation enables the construction of QDI circuits with bounded throughput, thus enabling the construction of interconnection circuits for GALS design, which are capable of handling the required design throughput.

The cycle time constraining technique presented here uses a pseudo-clock to constrain the cycle time. This assumes that the logic of every pipeline stage has the same delay. This assumption is only true if the pipeline is well balanced, as it disregards the logic complexity of individual pipeline stages. Furthermore, it also disregards the delay disparity between forward and backward propagation paths. In this work, empty propagation paths internal to the full-buffer component expansion are attributed a fixed minimal delay. Future work will target the automatic identification of pipeline disparities and constraining of each propagation path in accordance to their respective logic complexity.

Another aspect of the Pulsar flow where further optimisations are expected to be beneficial is the dual-rail expansion. Currently the dual-rail expansion relies on EDA tools ability to optimise combinations of virtual functions. However, commercial EDA tools do not recognise optimisation opportunities inherent to the use of DI codes. For instance, DI codes expressed in binary contain a large amount of invalid binary combinations, which can used as "don't-cares" during logic optimisation and technology mapping. Fant [FB96] has demonstrated that when composing *image functions*, which are analogous to the virtual functions defined in this work, minterms containing these invalid codes do arise. This opens opportunities for exploring alternative dual-rail expansion and optimisations and technology mapping techniques that take advantage of specific characteristics of DI codes. Another optimisation that could be explored during the dual-rail expansion and technology mapping is relaxation [LM09]. This technique allows producing faster and smaller QDI circuits by enabling parts of the circuit to violate the indication principle locally, while globally maintaining strong indication. Future work will explore the construction of an alternative dual-rail expansion technique and possibly a technology mapper to allow these and other optimisations.

The physical synthesis, albeit possible with the current flow requires further exploration. This is especially interesting when it comes to the introduction of adversarial paths [KKM09] that hinder the indication principle. Currently, this is verified through simulation. However, a formal methodology for detecting these paths is highly desirable. Such a methodology would possibly not only detect potential hazards, but could also enable avoiding them entirely, by using e.g. relative timing constraints [MS16].

Simulation and formal verification is another aspect that requires further attention. Currently, the behaviour of the synthesised circuit is not captured in the simulation of the RTL description. Also, the current flow does not allow equivalence checking between the input RTL description, the virtual netlist and the final synthesised netlist. Solutions to allow equivalence checking and high-level simulation shall be addressed in further work.

REFERENCES

[BCSS99] Bjesse, P.; Claessen, K.; Sheeran, M.; Singh, S. "Lava: Hardware Design in Haskell", *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, vol. 34–1, 1999, pp. 174–184.

[BCV+05] Beigné, E.; Clermidy, F.; Vivet, P.; Clouard, A.; Renaudin, M. "An Asynchronous NoC Architecture Providing Low Latency Service and Its Multi-Level Design Framework". In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2005, pp. 54–63.

[BDL11] Beerel, P. A.; Dimou, G. D.; Lines, A. M. "Proteus: An ASIC Flow for GHz Asynchronous Designs", *IEEE Design & Test of Computers*, vol. 28–5, 2011, pp. 36–50.

[Ber93] van Berkel, K. "Handshake Circuits: An Asynchronous Architecture for VLSI Programming". New York, NY, USA: Cambridge University Press, 1993, 225p.

[BKK+10] Baaij, C. P. R.; Kooijman, M.; Kuper, J.; Boeijink, W. A.; Gerards, M. E. T. "CλasH: Structural Descriptions of Synchronous Hardware using Haskell". In: EUROMICRO Conference on Digital System Design (DSD), 2010, pp. 714–721.

[BLDK06] Beerel, P. A.; Lines, A. M.; Davies, M.; Kim, N.-H. K. "Slack Matching Asynchronous Designs". In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2006, pp. 184–194.

[BOF10] Beerel, P. A.; Ozdag, R. O.; Ferretti, M. "A Designer´s Guide to Asynchronous VLSI". Cambridge University Press, 2010, 353p.

[BSVMH84] Brayton, R. K.; Sangiovanni-Vincentelli, A.; McMullen, C.; Hachtel, G. "Logic Minimization Algorithms for VLSI Synthesis". Norwell, MA, USA: Kluwer Academic Publishers, 1984, 174p.

[BTE09] Bardsley, A.; Tarazona, L.; Edwards, D. "Teak: A Token-Flow Implementation for the Balsa Language". In: 2009 Ninth International Conference on Application of Concurrency to System Design, 2009, pp. 23–31.

[BVR+12] Bachrach, J.; Vo, H.; Richards, B.; Lee, Y.; Waterman, A.; Avižienis, R.; Wawrzynek, J.; Asanović, K. "Chisel: Constructing hardware in a Scala embedded language". In: Design Automation Conference (DAC), 2012, pp. 1212–1221.

[CCGC10] Chen, J.; Chong, K.-S.; Gwee, B.-H.; Chang, J. S. “An ultra-low power asynchronous quasi-delay-insensitive (QDI) sub-threshold memory with bit-interleaving and completion detection”. In: 8th IEEE International NEWCAS Conference (NEWCAS), 2010, pp. 117–120.

[CLRS09] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. “Introduction to Algorithms”. The MIT Press, 2009, 3rd ed., 1292p.

[DCB93] Das, D. K.; Chacraborty, S.; Bhattacharya, B. B. “Irredundant binate realizations of unate functions”, *International Journal of Electronics Theoretical and Experimental*, vol. 75–1, 1993, pp. 65–73.

[DLD+14] Davies, M.; Lines, A.; Dama, J.; Gravel, A.; Southworth, R.; Dimou, G.; Beerel, P. A. “A 72-Port 10G Ethernet Switch/Router Using Quasi-Delay-Insensitive Asynchronous Design”. In: 20th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), 2014, pp. 103–104.

[DMM04] Donno, M.; Macii, E.; Mazzoni, L. “Power-aware clock tree planning”. In: International Symposium on Physical Design (ISPD), 2004, pp. 138–147.

[EB02] Edwards, D.; Bardsley, A. “Balsa: An Asynchronous Hardware Synthesis Language”, *The Computer Journal*, vol. 45–1, 2002, pp. 12–18.

[Fan05] Fant, K. M. “Logically Determined Design: Clockless System Design with NULL Convention Logic”. Wiley, 2005, 292p.

[FB96] Fant, K. M.; Brandt, S. A. “NULL Convention Logic^{TM}: A complete and consistent logic for asynchronous digital circuit synthesis”. In: International Conference on Application Specific Systems, Architectures and Processors (ASAP), 1996, pp. 261–273.

[Fre12] Free Software Foundation, Inc. “GNU Linear Programming Kit (GLPK)”. Source: https://www.gnu.org/software/glpk/glpk.html, 2012.

[Gil11] Gill, A. “Declarative FPGA Circuit Synthesis using Kansas Lava”. In: International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), 2011, pp. 55–64.

[GLY10] Guan, X.; Liu, Y.; Yang, Y. “Performance Analysis of Low Power Null Convention Logic Units with Power Cutoff”. In: Asia-Pacific Conference on Wearable Computing Systems (APWCS), 2010, pp. 55–58.

[Har01] Harris, D. “Skew-tolerant Circuit Design”. Morgan Kaufmann, 2001, 300p.

[Hoa78] Hoare, C. A. R. “Communicating Sequential Processes”, *Communications of the ACM*, vol. 21–8, Aug 1978, pp. 666–677.

[Hoa85] Hoare, C. A. R. “Communicating Sequential Processes”. Prentice Hall International, 1985, 260p.

[Hur69] Hurst, S. “An Introduction to Threshold Logic: A survey of present theory and practice”, *The Radio and Electronic Engineer*, vol. 37–6, 1969, pp. 339–351.

[JN08] Jeong, C.; Nowick, S. “Technology Mapping and Cell Merger for Asynchronous Threshold Networks”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27–4, 2008, pp. 659–672.

[JSL+10] Jorgenson, R.; Sorensen, L.; Leet, D.; Hagedorn, M.; Lamb, D.; Friddell, T.; Snapp, W. “Ultralow-Power Operation in Subthreshold Regimes Applying Clockless Logic”, *Proceedings of the IEEE*, vol. 98–2, Feb 2010, pp. 299–314.

[KKM09] Keller, S.; Katelman, M.; Martin, A. J. “A Necessary and Sufficient Timing Assumption for Speed-Independent Circuits”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2009, pp. 65–76.

[LM09] LaFrieda, C.; Manohar, R. “Reducing Power Consumption with Relaxed Quasi Delay-Insensitive Circuits”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2009, pp. 217–226.

[Mag84] Magott, J. “Performance Evaluation of Concurrent Systems using Petri Nets”, *Information Processing Letters*, vol. 18–1, 1984, pp. 7–13.

[Man19] Manohar, R. “An Open-Source Design Flow for Asynchronous Circuits”. In: Government Microcircuit Applications And Critical Technology Conference (GOMACTECH), 2019, pp. 1–5.

[MBSC18] Moreira, M, T.; Beerel, P. A.; Sartori, M. L. L.; Calazans, N. L. V. “NCL Synthesis With Conventional EDA Tools: Technology Mapping and Optimization”, *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65–6, 2018, pp. 1981–1993.

[MGC12] Moreira, M. T.; Guazzelli, R. A.; Calazans, N. L. V. “Return-to-One Protocol for Reducing Static Power in QDI Circuits Employing m-of-n Codes”. In: Symposium on Integrated Circuits and Systems Design (SBCCI), 2012, pp. 1–6.

[MGHC14] Moreira, M. T.; Guazzelli, R. A.; Heck, G.; Calazans, N. L. V. “Hardening QDI Circuits Against Transient Faults Using Delay-Insensitive Maxterm Synthesis”. In: ACM Great Lakes Symposium on VLSI (GLSVLSI), 2014, pp. 3–8.

[MM11] Martin, A. J.; Moore, C. D. "CHP and CHPsim: A Language and Simulator for Fine-grain Distributed Computation", Technical Report, CS Dept., Caltech, Pasadena, CA, USA, Tech. Rep. CS-TR-1-2011, 2011, 12p.

[MN06] Martin, A.; Nyström, M. "Asynchronous Techniques for System-on-Chip Design", *Proceedings of the IEEE*, vol. 94–6, Jun 2006, pp. 1089–1120.

[MNM+14] Moreira, M. T.; Neutzling, A.; Martins, M.; Reis, A.; Ribas, R.; Calazans, N. L. V. "Semi-custom NCL Design with Commercial EDA Frameworks: Is it possible?" In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2014, pp. 53–60.

[MNW03] Martin, A. J.; Nyström, M.; Wong, C. G. "Three Generations of Asynchronous Microprocessors", *IEEE Design & Test of Computers*, vol. 20–6, 2003, pp. 9–17.

[MOPC11] Moreira, M. T.; Oliveira, B. S.; Pontes, J. J. H.; Calazans, N. L. V. "A 65nm Standard Cell Set and Flow Dedicated to Automated Asynchronous Circuits Design". In: IEEE International System on Chip Conference (SoCC), 2011, pp. 99–104.

[MOPC13] Moreira, M. T.; Oliveira, C. H. M.; Porto, R. C.; Calazans, N. L. V. "NCL+: Return-to-one Null Convention Logic". In: IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), 2013, pp. 836–839.

[Mor16] Moreira, M. T. "Asynchronous Circuits: Innovations in Components, Cell Libraries and Design Templates", Ph.D. Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, FACIN-PPGCC, 2016, 276p.

[MPC14] Moreira, M. T.; Pontes, J. H.; Calazans, N. L. V. "Tradeoffs between RTO and RTZ in WCHB QDI Asynchronous Design". In: International Symposium on Quality Electronic Design (ISQED), 2014, pp. 692–699.

[MS16] Manoranjan, J. V.; Stevens, K. S. "Qualifying relative timing constraints for asynchronous circuits". In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2016, pp. 91–98.

[MTMC14] Moreira, M. T.; Trojan, G.; Moraes, F. G.; Calazans, N. L. V. "Spatially Distributed Dual-Spacer Null Convention Logic Design", *Journal of Low Power Electronics*, vol. 10–3, 2014, pp. 313–320.

[Mur89] Murata, T. "Petri nets: Properties, analysis and applications", *Proceedings of the IEEE*, vol. 77–4, 1989, pp. 541–580.

[NS15a] Nowick, S. M.; Singh, M. “Asynchronous Design - Part 1: Overview and Recent Advances”, *IEEE Design & Test of Computers*, vol. 32–3, Jun 2015, pp. 5–18.

[NS15b] Nowick, S. M.; Singh, M. “Asynchronous Design - Part 2: Systems and Methodologies”, *IEEE Design & Test of Computers*, vol. 32–3, Jun 2015, pp. 19–28.

[PMMC10] Pontes, J.; Moreira, M. T.; Moraes, F. G.; Calazans, N. L. V. “Hermes-AA: A 65nm asynchronous NoC router with adaptive routing”. In: IEEE International System-on-Chip Conference (SOCC), 2010, pp. 493–498.

[PTE05] Plana, L. A.; Taylor, S.; Edwards, D. “Attacking Control Overhead to Improve Synthesised Asynchronous Circuit Performance”. In: IEEE International Conference on Computer Design (ICCD), 2005, pp. 703–710.

[Rei13] Reisig, W. “Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies”. Springer, 2013, 230p.

[RST12] Reese, R. B.; Smith, S. C.; Thornton, M. A. “Uncle - An RTL Approach to Asynchronous Design”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2012, pp. 65–72.

[SCH+11] Seok, M.; Chen, G.; Hanson, S.; Wieckowski, M.; Blaauw, D.; Sylvester, D. “Mitigating Variability in Near-Threshold Computing”, *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1–1, Mar 2011, pp. 42–49.

[SDF06] Sutherland, S.; Davidmann, S.; Flake, P. “SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling”. Springer Science & Business Media, 2006, 2nd ed., 418p.

[SF01] Sparsø, J.; Furber, S. “Principles of Asynchronous Circuit Design – A Systems Perspective”. Springer, 2001, 356p.

[Sut89] Sutherland, I. E. “Micropipelines”, *Communications of the ACM*, vol. 32–6, Jun 1989, pp. 720–738.

[Tan18] Tange, O. “GNU Parallel 2018”. Ole Tange, 2018, 112p.

[TBV12] Thonnart, Y.; Beigné, E.; Vivet, P. “A pseudo-synchronous Implementation Flow for WCHB QDI Asynchronous Circuits”. In: IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC), 2012, pp. 73–80.

[Ver88] Verhoeff, T. “Delay-Insensitive Codes - An Overview”, *Distributed Computing*, vol. 3–1, Mar 1988, pp. 1–8.

[ZSD10] Zhou, L.; Smith, S.; Di, J. “Bit-Wise MTNCL: An ultra-low power bit-wise pipelined asynchronous circuit design methodology”. In: 53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), 2010, pp. 217–220.

APPENDIX A – THE SINGLE-RAIL SYNTHESIS SCRIPTS

Listing A.1: circuits/design/syn_rtl.tcl

```
1  set DESIGN $::env(DESIGN)
2
3  set_db syn_global_effort high
4  set_db syn_opt_effort extreme
5  set_db retime_effort_level high
6  set_db tns_opto true
7  source ../../scripts/analysis.tcl
8  read_libs ../../tech/nand2.lib
9  read_hdl -sv rtl_${DESIGN}.sv
10 elaborate
11 set_db [current_design] .retime true
12 create_clock -period 0 [get_port clk]
13 set_input_delay -clock clk 0 [all_inputs]
14 set_output_delay -clock clk 0 [all_outputs]
15
16 set_db iopt_force_constant_removal true
17 set_db remove_assigns true
18 syn_generic
19 syn_map
20
21 ungroup -all
22 syn_opt
23 syn_opt -incremental
24
25 set_db write_vlog_wor_wand false
26 update_names -restricted { [ ] } -system_verilog
27
28 report_gates > rtl_gates.rpt
29 report_timing > rtl_timing.rpt
30 write_hdl > ${DESIGN}.v
31 export_graph ${DESIGN}.graph
32 shell drexpander ${DESIGN}.v > ncl_${DESIGN}.v
```

Listing A.2: scripts/analysis.tcl

```
1  proc export_graph {file_name} {
2    variable graph
3    variable regs_data
4    variable regs_null
5    variable in_ports
6    variable out_ports
7    variable fp
8    set fp [open $file_name w]
```

```
9
10    set regs_data [concat [get_db [all::all_seqs] -if {.base_cell == dffr}]
          [get_db [all::all_seqs] -if {.base_cell == dffs}]]
11    set regs_null [concat [get_db [all::all_seqs] -if {.base_cell == dff}]
          [get_db [all::all_seqs] -if {.base_cell == dffen}]]
12
13    set in_ports [get_db [all_inputs -no_clock] -if {.name != reset}]
14    set out_ports [get_db [all_outputs]]
15
16    foreach {i} $in_ports {
17      foreach {o} [get_db [get_fanout -endpoints $i] -if {.obj_type == pin}] {
18        lappend graph($i) [vdirname $o]
19      }
20      foreach {o} [get_db [get_fanout -endpoints $i] -if {.obj_type == port}] {
21        lappend graph($i) $o
22      }
23    }
24
25    foreach {i} [all::all_seqs -output_pins] {
26      foreach {o} [get_db [get_fanout -endpoints $i] -if {.obj_type == pin}] {
27        lappend graph([vdirname $i]) [vdirname $o]
28      }
29      foreach {o} [get_db [get_fanout -endpoints $i] -if {.obj_type == port}] {
30        lappend graph([vdirname $i]) $o
31      }
32    }
33
34    foreach {i} ${in_ports} {
35      variable adjacent
36      if [info exists graph($i)] {
37        set adjacent $graph($i)
38      } else {
39        set adjacent [list]
40      }
41      puts $fp [format {Port "%s" [%s]} ${i} [join [lmap x ${adjacent} {format
            {"%s"} $x}] {,}]]
42    }
43
44    foreach {i} ${out_ports} {
45      variable adjacent
46      if [info exists graph($i)] {
47        set adjacent $graph($i)
48      } else {
49        set adjacent [list]
50      }
51      puts $fp [format {Port "%s" [%s]} ${i} [join [lmap x ${adjacent} {format
            {"%s"} $x}] {,}]]
52    }
```

```
53
54    foreach {i} ${regs_data} {
55      variable adjacent
56      if [info exists graph($i)] {
57        set adjacent $graph($i)
58      } else {
59        set adjacent [list]
60      }
61      puts $fp [format {DataReg "%s" [%s]} ${i} [join [lmap x ${adjacent} {format
          {"%s"} $x}] {,}]]
62    }
63
64    foreach {i} ${regs_null} {
65      variable adjacent
66      if [info exists graph($i)] {
67        set adjacent $graph($i)
68      } else {
69        set adjacent [list]
70      }
71      puts $fp [format {NullReg "%s" [%s]} ${i} [join [lmap x ${adjacent} {format
          {"%s"} $x}] {,}]]
72    }
73
74    close $fp
75    unset -nocomplain graph
76    unset -nocomplain regs_data
77    unset -nocomplain regs_null
78    unset -nocomplain in_ports
79    unset -nocomplain out_ports
80  }
81
82  # Local Variables:
83  # tcl-indent-level: 2
84  # indent-tabs-mode: nil
85  # End:
```

Listing A.3: tech/nand2.lib – The component library Liberty file.

```
1   library (PULSAR_NAND2) {
2     delay_model : table_lookup;
3
4
5     /* unit attributes */
6     time_unit : "1ns";
7     voltage_unit : "1V";
8     current_unit : "1mA";
9     pulling_resistance_unit : "1kohm";
10    leakage_power_unit : "1pW";
```

```
11    capacitive_load_unit (1.0,pf);
12
13    /* operation conditions */
14    nom_process      : 1;
15    nom_temperature  : 25;
16    nom_voltage      : 1.2;
17    operating_conditions(typical) {
18      process : 1;
19      temperature : 25;
20      voltage : 1.2;
21      tree_type : balanced_tree
22    }
23    default_operating_conditions : typical;
24
25      /* threshold definitions */
26    slew_lower_threshold_pct_fall : 30.0;
27    slew_upper_threshold_pct_fall : 70.0;
28    slew_lower_threshold_pct_rise : 30.0;
29    slew_upper_threshold_pct_rise : 70.0;
30    input_threshold_pct_fall      : 50.0;
31    input_threshold_pct_rise      : 50.0;
32    output_threshold_pct_fall     : 50.0;
33    output_threshold_pct_rise     : 50.0;
34    slew_derate_from_library      : 0.5;
35
36    /* default attributes */
37    default_leakage_power_density : 0.0;
38    default_cell_leakage_power : 0.0;
39    default_fanout_load : 1.0;
40    default_output_pin_cap : 0.0;
41    default_inout_pin_cap : 0.00158;
42    default_input_pin_cap : 0.00158;
43    default_max_transition : 1.02;
44
45
46  cell (nor2) {
47    area : 31.0;
48    pin(a) {
49      direction : input;
50      capacitance : 0.002;
51    }
52    pin(b) {
53      direction : input;
54      capacitance : 0.002;
55    }
56    pin(y) {
57      direction : output;
58      capacitance : 0.0;
```

```
59        function : "!(a|b)";
60        timing() {
61          related_pin : "a";
62          timing_sense : negative_unate;
63          cell_rise(scalar) {
64          values ("0.01");
65          }
66          rise_transition(scalar) {
67          values ("0.0");
68          }
69          cell_fall(scalar) {
70          values ("0.02");
71          }
72          fall_transition(scalar) {
73          values ("0.0");
74          }
75        }
76        timing() {
77          related_pin : "b";
78          timing_sense : negative_unate;
79          cell_rise(scalar) {
80          values ("0.01");
81          }
82          rise_transition(scalar) {
83          values ("0.0");
84          }
85          cell_fall(scalar) {
86          values ("0.02");
87          }
88          fall_transition(scalar) {
89          values ("0.0");
90          }
91        }
92      }
93    }
94
95    cell (nand2) {
96      area : 31.0;
97      pin(a) {
98        direction : input;
99        capacitance : 0.002;
100     }
101     pin(b) {
102       direction : input;
103       capacitance : 0.002;
104     }
105     pin(y) {
106       direction : output;
```

```
107       capacitance : 0.0;
108       function : "!(a*b)";
109       timing() {
110         related_pin : "a";
111         timing_sense : negative_unate;
112         cell_rise(scalar) {
113         values ("0.02");
114         }
115         rise_transition(scalar) {
116         values ("0.0");
117         }
118         cell_fall(scalar) {
119         values ("0.01");
120         }
121         fall_transition(scalar) {
122         values ("0.0");
123         }
124       }
125       timing() {
126         related_pin : "b";
127         timing_sense : negative_unate;
128         cell_rise(scalar) {
129         values ("0.02");
130         }
131         rise_transition(scalar) {
132         values ("0.0");
133         }
134         cell_fall(scalar) {
135         values ("0.01");
136         }
137         fall_transition(scalar) {
138         values ("0.0");
139         }
140       }
141     }
142   }
143
144   cell (inv) {
145     area : 1.0;
146     pin(a) {
147       direction : input;
148       capacitance : 0.001;
149     }
150     pin(y) {
151       direction : output;
152       capacitance : 0.0;
153       function : "!a";
154       timing() {
```

```
155         related_pin : "a";
156         timing_sense : negative_unate;
157         cell_rise(scalar) {
158         values ("0.0");
159         }
160         rise_transition(scalar) {
161         values ("0.0");
162         }
163         cell_fall(scalar) {
164         values ("0.0");
165         }
166         fall_transition(scalar) {
167         values ("0.0");
168         }
169       }
170     }
171   }
172
173   cell (buff) {
174     area : 1.0;
175     pin(a) {
176       direction : input;
177       capacitance : 0.001;
178     }
179     pin(y) {
180       direction : output;
181       capacitance : 0.0;
182       function : "a";
183       timing() {
184         related_pin : "a";
185         timing_sense : positive_unate;
186         cell_rise(scalar) {
187         values ("0.0");
188         }
189         rise_transition(scalar) {
190         values ("0.0");
191         }
192         cell_fall(scalar) {
193         values ("0.0");
194         }
195         fall_transition(scalar) {
196         values ("0.0");
197         }
198       }
199     }
200   }
201
202   cell (xor2) {
```

```
203     area : 38.0;
204     pin(a) {
205       direction : input;
206       capacitance : 0.002;
207     }
208     pin(b) {
209       direction : input;
210       capacitance : 0.002;
211     }
212     pin(y) {
213       direction : output;
214       capacitance : 0.0;
215       function : "a^b";
216       timing() {
217         related_pin : "a";
218         timing_sense : positive_unate;
219         cell_rise(scalar) {
220         values ("0.015");
221         }
222         rise_transition(scalar) {
223         values ("0.0");
224         }
225         cell_fall(scalar) {
226         values ("0.015");
227         }
228         fall_transition(scalar) {
229         values ("0.0");
230         }
231       }
232       timing() {
233         related_pin : "b";
234         timing_sense : positive_unate;
235         cell_rise(scalar) {
236         values ("0.015");
237         }
238         rise_transition(scalar) {
239         values ("0.0");
240         }
241         cell_fall(scalar) {
242         values ("0.015");
243         }
244         fall_transition(scalar) {
245         values ("0.0");
246         }
247       }
248       timing() {
249         related_pin : "a";
250         timing_sense : negative_unate;
```

```
251          cell_rise(scalar) {
252          values ("0.015");
253          }
254          rise_transition(scalar) {
255          values ("0.0");
256          }
257          cell_fall(scalar) {
258          values ("0.015");
259          }
260          fall_transition(scalar) {
261          values ("0.0");
262          }
263        }
264        timing() {
265          related_pin : "b";
266          timing_sense : negative_unate;
267          cell_rise(scalar) {
268          values ("0.015");
269          }
270          rise_transition(scalar) {
271          values ("0.0");
272          }
273          cell_fall(scalar) {
274          values ("0.015");
275          }
276          fall_transition(scalar) {
277          values ("0.0");
278          }
279        }
280      }
281    }
282
283    cell (dff) {
284      area : 32.0;
285      pin(d) {
286        direction : input;
287        capacitance : 0.001;
288        timing() {
289          related_pin : "ck";
290          timing_type : setup_rising;
291          rise_constraint(scalar) {
292          values ("0.0");
293          }
294          fall_constraint(scalar) {
295          values ("0.0");
296          }
297        }
298        timing() {
```

```
299         related_pin : "ck";
300         timing_type : hold_rising;
301         rise_constraint(scalar) {
302         values ("0.0");
303         }
304         fall_constraint(scalar) {
305         values ("0.0");
306         }
307       }
308     }
309     pin(ck) {
310       direction : input;
311       clock : true;
312       capacitance : 0.001;
313     }
314     ff(IQ,IQN) {
315       clocked_on : "ck";
316       next_state : "d";
317     }
318     pin(q) {
319       direction : output;
320       capacitance : 0.0;
321       function : "IQ";
322       timing() {
323         related_pin : "ck";
324         timing_type : rising_edge;
325         timing_sense : non_unate;
326         cell_rise(scalar) {
327         values ("0.01");
328         }
329         rise_transition(scalar) {
330         values ("0.0");
331         }
332         cell_fall(scalar) {
333         values ("0.01");
334         }
335         fall_transition(scalar) {
336         values ("0.0");
337         }
338       }
339     }
340   }
341
342   cell (dffr) {
343     area : 64.0;
344     pin(d) {
345       direction : input;
346       capacitance : 0.001;
```

```
347       timing() {
348         related_pin : "ck";
349         timing_type : setup_rising;
350         rise_constraint(scalar) {
351         values ("0.0");
352         }
353         fall_constraint(scalar) {
354         values ("0.0");
355         }
356       }
357       timing() {
358         related_pin : "ck";
359         timing_type : hold_rising;
360         rise_constraint(scalar) {
361         values ("0.0");
362         }
363         fall_constraint(scalar) {
364         values ("0.0");
365         }
366       }
367     }
368     pin(ck) {
369       direction : input;
370       clock : true;
371       capacitance : 0.001;
372     }
373     pin(rb) {
374       direction : input;
375       capacitance : 0.001;
376       timing() {
377         related_pin : "ck";
378         timing_type : setup_rising;
379         rise_constraint(scalar) {
380         values ("0.0");
381         }
382         fall_constraint(scalar) {
383         values ("0.0");
384         }
385       }
386       timing() {
387         related_pin : "ck";
388         timing_type : hold_rising;
389         rise_constraint(scalar) {
390         values ("0.0");
391         }
392         fall_constraint(scalar) {
393         values ("0.0");
394         }
```

```
395       }
396     }
397     ff(IQ,IQN) {
398       clocked_on : "ck";
399       next_state : "d";
400       clear   : "(!rb)";
401     }
402     pin(q) {
403       direction : output;
404       capacitance : 0.0;
405       function : "IQ";
406       timing() {
407         related_pin : "ck";
408         timing_type : rising_edge;
409         timing_sense : non_unate;
410         cell_rise(scalar) {
411         values ("0.01");
412         }
413         rise_transition(scalar) {
414         values ("0.0");
415         }
416         cell_fall(scalar) {
417         values ("0.01");
418         }
419         fall_transition(scalar) {
420         values ("0.0");
421         }
422       }
423       timing() {
424         related_pin : "rb";
425         timing_type : clear;
426         timing_sense : positive_unate;
427         cell_fall(scalar) {
428         values ("0.01");
429         }
430         fall_transition(scalar) {
431         values ("0.0");
432         }
433       }
434     }
435   }
436 
437   cell (dffs) {
438     area : 64.0;
439     pin(d) {
440       direction : input;
441       capacitance : 0.001;
442       timing() {
```

```
443         related_pin : "ck";
444         timing_type : setup_rising;
445         rise_constraint(scalar) {
446         values ("0.0");
447         }
448         fall_constraint(scalar) {
449         values ("0.0");
450         }
451       }
452       timing() {
453         related_pin : "ck";
454         timing_type : hold_rising;
455         rise_constraint(scalar) {
456         values ("0.0");
457         }
458         fall_constraint(scalar) {
459         values ("0.0");
460         }
461       }
462     }
463     pin(ck) {
464       direction : input;
465       clock : true;
466       capacitance : 0.001;
467     }
468     pin(sb) {
469       direction : input;
470       capacitance : 0.001;
471       timing() {
472         related_pin : "ck";
473         timing_type : setup_rising;
474         rise_constraint(scalar) {
475         values ("0.0");
476         }
477         fall_constraint(scalar) {
478         values ("0.0");
479         }
480       }
481       timing() {
482         related_pin : "ck";
483         timing_type : hold_rising;
484         rise_constraint(scalar) {
485         values ("0.0");
486         }
487         fall_constraint(scalar) {
488         values ("0.0");
489         }
490       }
```

```
491     }
492     ff(IQ,IQN) {
493       clocked_on : "ck";
494       next_state : "d";
495       preset   : "(!sb)";
496     }
497     pin(q) {
498       direction : output;
499       capacitance : 0.0;
500       function : "IQ";
501       timing() {
502         related_pin : "ck";
503         timing_type : rising_edge;
504         timing_sense : non_unate;
505         cell_rise(scalar) {
506         values ("0.01");
507         }
508         rise_transition(scalar) {
509         values ("0.0");
510         }
511         cell_fall(scalar) {
512         values ("0.01");
513         }
514         fall_transition(scalar) {
515         values ("0.0");
516         }
517       }
518       timing() {
519         related_pin : "sb";
520         timing_type : preset;
521         timing_sense : negative_unate;
522         cell_rise(scalar) {
523         values ("0.01");
524         }
525         rise_transition(scalar) {
526         values ("0.0");
527         }
528         cell_fall(scalar) {
529         values ("0.01");
530         }
531         fall_transition(scalar) {
532         values ("0.0");
533         }
534       }
535     }
536   }
537   }
```

APPENDIX B – THE DUAL-RAIL EXPANDER

Listing B.1: expander.hs – Haskell source code for the dual-rail expander.

```
1  import             Control.Monad.Reader
2  import             DRExpander
3  import             Options.Applicative
4  import Data.Monoid
5
6  prgOptions :: Parser PrgOptions
7  prgOptions = PrgOptions
8              <$> some (argument str (metavar "FILES"
9                                  <> help "Input File Name"))
10             <*> strOption (long "reset"
11                            <> short 'r'
12                            <> value "reset"
13                            <> help "Reset port name")
14             <*> strOption (long "clock"
15                           <> short 'c'
16                           <> value "clk"
17                           <> help "Clock port name")
18
19
20 main :: IO ()
21 main = do
22   let opts = info (prgOptions <**> helper)
23              ( fullDesc
24                <> progDesc "Prepares a netlist for dual-rail expansion"
25                <> header "drexpand - Pulsar's dual-rail expansor")
26   options <- execParser opts
27   runReaderT prgMain options
28
29 prgMain :: ReaderT PrgOptions IO ()
30 prgMain = do
31   env <- ask
32   modules <- processVerilogFiles $ verilogFiles env
33   liftIO . putStr . unlines $ show <$> modules
```

Listing B.2: src/DRExpander.hs – Haskell source code for the dual-rail expander

```
1 {-# LANGUAGE FlexibleContexts #-}
2 {-# LANGUAGE OverloadedLists  #-}
3 module DRExpander where
4
5 import             Control.Monad.IO.Class
6 import             Control.Monad.Reader
7 import             Data.BitVec
```

```
8  import           Data.Set            (Set)
9  import qualified Data.Set            as Set
10 import qualified Language.Verilog    as Verilog
11
12 data Wire = Wire String
13           | Bus Integer Integer String
14           deriving (Eq, Ord)
15
16 data PrgOptions = PrgOptions
17   { verilogFiles :: [FilePath]
18   , resetName    :: String
19   , clkName      :: String
20   } deriving (Show)
21
22 bitBlastWire :: Wire -> [Wire]
23 bitBlastWire (Bus x y name) = map (Wire . expandBusWireName name) [x'..y'] where
24   x' = min x y
25   y' = max x y
26 bitBlastWire x              = [x]
27
28 expandBusWireName :: String -> Integer -> String
29 expandBusWireName name idx = name ++ "_" ++ show idx
30
31 readVerilogFile :: (MonadIO m) => FilePath -> m [Verilog.Module]
32 readVerilogFile path = do
33   s <- liftIO $ readFile path
34   return $ Verilog.parseFile [] path s
35
36 readVerilogFiles :: (MonadIO m) => [FilePath] -> m [Verilog.Module]
37 readVerilogFiles = fmap concat . mapM readVerilogFile
38
39 vlogModuleInputs :: Verilog.Module -> Set Wire
40 vlogModuleInputs (Verilog.Module _ _ items) = Set.fromList $ concatMap go items
41   where
42     go (Verilog.Input Nothing xs)       = map Wire xs
43     go (Verilog.Input (Just (Verilog.Number x, Verilog.Number y)) xs) = map
        (Bus x' y') xs where
44       y' = value y
45       x' = value x
46     go _                                = []
47
48 vlogModuleOutputs :: Verilog.Module -> Set Wire
49 vlogModuleOutputs (Verilog.Module _ _ items) = Set.fromList $ concatMap go items
50   where
51     go (Verilog.Output Nothing xs)       = map Wire xs
52     go (Verilog.Output (Just (Verilog.Number x, Verilog.Number y)) xs) = map
        (Bus x' y') xs where
53       y' = value y
```

```
54          x' = value x
55        go _                                = []
56
57  vlogModuleWires :: Verilog.Module -> Set Wire
58  vlogModuleWires (Verilog.Module _ _ items) = Set.fromList $ concatMap go items
59    where
60        go (Verilog.Wire Nothing xs)        = map (Wire . fst) xs
61        go (Verilog.Wire (Just (Verilog.Number x, Verilog.Number y)) xs) = map (Bus
            x' y') xs' where
62          y' = value y
63          x' = value x
64          xs' = map fst xs
65        go _                                = []
66
67  vlogModuleIntWires :: Verilog.Module -> Set Wire
68  vlogModuleIntWires mod = wires Set.\\ Set.union inputs outputs
69    where
70        wires = vlogModuleWires mod
71        inputs = vlogModuleInputs mod
72        outputs = vlogModuleOutputs mod
73
74  vlogModuleAllWires :: Verilog.Module -> Set Wire
75  vlogModuleAllWires mod = Set.unions [wires, inputs, outputs]
76    where
77        wires = vlogModuleWires mod
78        inputs = vlogModuleInputs mod
79        outputs = vlogModuleOutputs mod
80
81  vlogModuleWithoutWires :: Verilog.Module -> Verilog.Module
82  vlogModuleWithoutWires (Verilog.Module name _ items) =
83    Verilog.Module name [] $ filter p items where
84    p (Verilog.Input _ _)  = False
85    p (Verilog.Output _ _) = False
86    p (Verilog.Wire _ _)   = False
87    p _                    = True
88
89  vlogDRWireInstance :: Wire -> [Verilog.ModuleItem]
90  vlogDRWireInstance (Wire name) = [Verilog.Instance "drwire" [] name []]
91  vlogDRWireInstance bus = concatMap vlogDRWireInstance $ bitBlastWire bus
92
93  vlogDRWirePort :: Verilog.Identifier -> [Verilog.Identifier]
94  vlogDRWirePort name = map (name ++) ["_t", "_f", "_ack"]
95
96  vlogDRWireInputInst :: Wire -> [Verilog.ModuleItem]
97  vlogDRWireInputInst (Wire name) =
98    [Verilog.Input Nothing $ map (name ++) ["_t", "_f"]
99    ,Verilog.Output Nothing [name ++ "_ack"]
100   ,Verilog.Assign (Verilog.LHS $ name ++ ".t") (Verilog.Ident $ name ++ "_t")
```

```
101    ,Verilog.Assign (Verilog.LHS $ name ++ ".f") (Verilog.Ident $ name ++ "_f")
102    ,Verilog.Assign (Verilog.LHS $ name ++ "_ack") (Verilog.Ident $ name ++
         ".ack")]
103  vlogDRWireInputInst (Bus x y name) =
104      [Verilog.Input range $ map (name ++) ["_t", "_f"]
105      ,Verilog.Output range [name ++ "_ack"]] ++
106      concatMap go values
107    where
108      range = Just (Verilog.Number $ fromInteger x, Verilog.Number $ fromInteger
           y)
109      values = [x'..y'] :: [Integer]
110      x' = min x y
111      y' = max x y
112      go :: Integer -> [Verilog.ModuleItem]
113      go i =
114        let name' = expandBusWireName name i in
115          [Verilog.Assign (Verilog.LHS $ name' ++ ".t") (Verilog.IdentBit (name
               ++ "_t") . Verilog.Number $ fromInteger i)
116          ,Verilog.Assign (Verilog.LHS $ name' ++ ".f") (Verilog.IdentBit (name
               ++ "_f") . Verilog.Number $ fromInteger i)
117          ,Verilog.Assign (Verilog.LHSBit (name ++ "_ack") (Verilog.Number $
               fromInteger i)) (Verilog.Ident $ name' ++ ".ack")]
118
119
120  vlogDRWireOutputInst :: Wire -> [Verilog.ModuleItem]
121  vlogDRWireOutputInst (Wire name) =
122    [Verilog.Input Nothing $ map (name ++) ["_t", "_f"]
123    ,Verilog.Output Nothing [name ++ "_ack"]
124    ,Verilog.Assign (Verilog.LHS $ name ++ ".t") (Verilog.Ident $ name ++ "_t")
125    ,Verilog.Assign (Verilog.LHS $ name ++ ".f") (Verilog.Ident $ name ++ "_f")
126    ,Verilog.Assign (Verilog.LHS $ name ++ "_ack") (Verilog.Ident $ name ++
         ".ack")]
127  vlogDRWireOutputInst (Bus x y name) =
128      [Verilog.Output range $ map (name ++) ["_t", "_f"]
129      ,Verilog.Input range [name ++ "_ack"]] ++
130      concatMap go values
131    where
132      range = Just (Verilog.Number $ fromInteger x, Verilog.Number $ fromInteger
           y)
133      values = [x'..y'] :: [Integer]
134      x' = min x y
135      y' = max x y
136      go :: Integer -> [Verilog.ModuleItem]
137      go i =
138        let name' = expandBusWireName name i in
139          [Verilog.Assign (Verilog.LHSBit (name ++ "_t") (Verilog.Number $
               fromInteger i)) (Verilog.Ident $ name' ++ ".t")
```

```
140           ,Verilog.Assign (Verilog.LHSBit (name ++ "_f") (Verilog.Number $
                fromInteger i)) (Verilog.Ident $ name' ++ ".f")
141           ,Verilog.Assign (Verilog.LHS (name' ++ ".ack") ) (Verilog.IdentBit
                (name ++ "_ack") . Verilog.Number $ fromInteger i)]
142
143 fixDffReset :: (MonadReader PrgOptions m) => Verilog.ModuleItem -> m
      Verilog.ModuleItem
144 fixDffReset inst@(Verilog.Instance mname parms name portmap)
145   | mname == "dff" || mname == "dffen" = do
146       env <- ask
147       let resetPin = (Just "reset", Just . Verilog.Ident $ resetName env)
148       return $ Verilog.Instance mname parms name (resetPin : portmap)
149   | otherwise = return inst
150 fixDffReset x = return x
151
152 fixInstancesBitBlast :: Verilog.ModuleItem -> Verilog.ModuleItem
153 fixInstancesBitBlast (Verilog.Instance mname parms name portmap) =
154   Verilog.Instance mname parms name $ map go portmap where
155   go (x, Just (Verilog.IdentBit name (Verilog.Number idx))) = (x, Just .
        Verilog.Ident . expandBusWireName name $ value idx)
156   go z = z
157 fixInstancesBitBlast x = x
158
159 vlogInputInstance :: [Verilog.Identifier] -> Verilog.ModuleItem
160 vlogInputInstance = Verilog.Input Nothing
161
162 wireName :: Wire -> String
163 wireName (Wire n)    = n
164 wireName (Bus _ _ n) = n
165
166 processModule :: (MonadReader PrgOptions m) => Verilog.Module -> m
      Verilog.Module
167 processModule mod = do
168   options <- ask
169   let clkAndResetNames = [clkName options, resetName options]
170   let clkAndReset = Set.fromList $ map Wire clkAndResetNames
171   let inputs = vlogModuleInputs mod Set.\\ clkAndReset
172   let outputs = vlogModuleOutputs mod
173   let wires = vlogModuleAllWires mod Set.\\ clkAndReset
174   let (Verilog.Module mname _ mitems) = vlogModuleWithoutWires mod
175   let clkrstInst = vlogInputInstance clkAndResetNames
176   let drWires = concatMap vlogDRWireInstance $ Set.elems wires
177   let drInputs = concatMap vlogDRWireInputInst $ Set.elems inputs
178   let drOutputs = concatMap vlogDRWireOutputInst $ Set.elems outputs
179   let margs = concatMap (vlogDRWirePort . wireName) (Set.elems $ Set.union
        inputs outputs) ++ clkAndResetNames
180   instances <- mapM (fixDffReset . fixInstancesBitBlast) mitems
181   let insts' = [clkrstInst] ++ drWires ++ drInputs ++ drOutputs ++ instances
```

```
182     return $ Verilog.Module mname margs insts'
183 
184   processVerilogFile :: (MonadReader PrgOptions m, MonadIO m) => FilePath -> m
        [Verilog.Module]
185   processVerilogFile path = readVerilogFile path >>= mapM processModule
186 
187   processVerilogFiles :: (MonadReader PrgOptions m, MonadIO m) => [FilePath] -> m
        [Verilog.Module]
188   processVerilogFiles = fmap concat . mapM processVerilogFile
```

APPENDIX C – THE CYCLE TIME CONSTRAINER

Listing C.1: constrainer.hs – Main Haskell source

```
1  {-# LANGUAGE FlexibleContexts #-}
2  import           Control.Monad.Reader
3  import           Data.LinearProgram.GLPK
4  import           Data.Map                (Map)
5  import qualified Data.Map                as Map
6  import           Data.Monoid
7  import           HBCN
8  import           Options.Applicative
9  import           Text.Printf
10 import           Text.Regex.TDFA
11
12 type LPRet = (ReturnCode, Maybe (Double, Map LPVar Double))
13
14 data PrgOptions = PrgOptions
15   { inputFiles      :: [FilePath]
16   , targetCycleTime :: Double
17   , minimalDelay    :: Double
18   , clockName       :: String
19   , outputFile      :: FilePath
20   , pathExceptions  :: Bool
21   , relaxEnable     :: Bool
22   , debugSol        :: Bool
23   } deriving (Show)
24
25 prgOptions :: Parser PrgOptions
26 prgOptions = PrgOptions
27              <$> some (argument str
28                         (metavar "input"
29                          <> help "List of Structural Graph Files"))
30              <*> option auto (long "cycletime"
31                                <> metavar "time"
32                                <> short 't'
33                                <> help "Target Cycle Time Constraint")
34              <*> option auto (long "mindelay"
35                                <> short 'm'
36                                <> metavar "time"
37                                <> value (-1)
38                                <> help "Minimum Path Delay, defaults to 10% of
                                      cycle time constraint")
39              <*> strOption (long "clock"
40                              <> metavar "portname"
41                              <> short 'c'
42                              <> value "clk"
```

```
43                              <> help "Clock port name")
44              <*> option str (long "output"
45                               <> metavar "filename"
46                               <> short 'o'
47                               <> value "ncl_constraints.sdc"
48                               <> help "Output SDC File, defaults to
                                    ncl_constraints.sdc")
49              <*> flag True False (long "no-path-exceptions"
50                                   <> help "Don't construct mindelay path
                                        exceptions")
51              <*> flag False True (long "relax"
52                                   <> help "Use free slack to relax timing
                                        constraints")
53              <*> flag False True (long "debug"
54                                   <> help "Print LP Variables Solution and
                                        export lp problem")
55
56
57
58  main :: IO ()
59  main = do
60    let opts = info (prgOptions <**> helper)
61              ( fullDesc
62                <> progDesc "Calculates the pseudo-clock constraints for a given
                    circuit"
63                <> header "hbcnConstrainer - Pulsar Linear Programming HBCN
                    constrainer")
64    options <- execParser opts
65    runReaderT prgMain options
66
67  hbcnFromFiles :: [FilePath] -> ReaderT PrgOptions IO HBCN
68  hbcnFromFiles files = do
69    text <- mapM (liftIO . readFile) files
70    let structure = map read $ (lines . concat) text
71    return $ createHBCNFromStructure structure
72
73  sdcContent :: (MonadReader PrgOptions m) => LPRet -> m String
74  sdcContent (Data.LinearProgram.GLPK.Success, Just (_, vars)) = do
75    opts <- ask
76    let clkPeriod = vars Map.! PseudoClock
77    let individual = pathExceptions opts
78    return $ printf "create_clock -period %.3f [get_port {%s}]\n" clkPeriod
        (clockName opts) ++
79      printf "set_input_delay -clock {%s} 0 [all_inputs]\n"   (clockName opts) ++
80      printf "set_output_delay -clock {%s} 0 [all_outputs]\n" (clockName opts) ++
81      if individual then
82        concatMap maxDelay (filter (\(_, v) -> (v > clkPeriod + 0.001) || (v <
            clkPeriod - 0.001)) $ Map.toList vars)
```

```
 83      else []
 84    where
 85      maxDelay (FwDelay src dst, val)
 86        | (src =~ "port:") && (dst =~ "port:") =
 87          printf "# Forward Delay from %s to %s\n" src dst ++
 88          printf "set_max_delay -from {%s} -to {%s} %.3f\n" (trueRail  src)
               (trueRail  dst) val ++
 89          printf "set_max_delay -from {%s} -to {%s} %.3f\n" (falseRail src)
               (falseRail dst) val ++
 90          printf "set_max_delay -from {%s} -to {%s} %.3f\n" (trueRail  src)
               (falseRail dst) val ++
 91          printf "set_max_delay -from {%s} -to {%s} %.3f\n" (falseRail src)
               (trueRail  dst) val
 92        | src =~ "port:" =
 93          printf "# Forward Delay from %s to %s\n" src dst ++
 94          printf "set_max_delay -from {%s} -to [get_pin -of_objects {%s} -filter
               {(is_clock_pin==false) && (direction==in)}] %.3f\n" (trueRail  src)
               (trueRail  dst) val ++
 95          printf "set_max_delay -from {%s} -to [get_pin -of_objects {%s} -filter
               {(is_clock_pin==false) && (direction==in)}] %.3f\n" (falseRail src)
               (falseRail dst) val ++
 96          printf "set_max_delay -from {%s} -to [get_pin -of_objects {%s} -filter
               {(is_clock_pin==false) && (direction==in)}] %.3f\n" (trueRail  src)
               (falseRail dst) val ++
 97          printf "set_max_delay -from {%s} -to [get_pin -of_objects {%s} -filter
               {(is_clock_pin==false) && (direction==in)}] %.3f\n" (falseRail src)
               (trueRail  dst) val
 98        | otherwise =
 99          printf "# Forward Delay from %s to %s\n" src dst ++
100          printf "set_max_delay -from [get_pin -of_objects {%s} -filter
               {is_clock_pin==true}] -to [get_pin -of_objects {%s} -filter
               {(is_clock_pin==false) && (direction==in)}] %.3f\n" (trueRail  src)
               (trueRail  dst) val ++
101          printf "set_max_delay -from [get_pin -of_objects {%s} -filter
               {is_clock_pin==true}] -to [get_pin -of_objects {%s} -filter
               {(is_clock_pin==false) && (direction==in)}] %.3f\n" (falseRail src)
               (falseRail dst) val ++
102          printf "set_max_delay -from [get_pin -of_objects {%s} -filter
               {is_clock_pin==true}] -to [get_pin -of_objects {%s} -filter
               {(is_clock_pin==false) && (direction==in)}] %.3f\n" (trueRail  src)
               (falseRail dst) val ++
103          printf "set_max_delay -from [get_pin -of_objects {%s} -filter
               {is_clock_pin==true}] -to [get_pin -of_objects {%s} -filter
               {(is_clock_pin==false) && (direction==in)}] %.3f\n" (falseRail src)
               (trueRail  dst) val
104      maxDelay (BwDelay src dst, val)
105        | (src =~ "port:") && (dst =~ "port:") =
106          printf "# Backward Delay from %s to %s\n" src dst ++
```

```
107          printf "set_max_delay -from {%s} -to {%s} %.3f\n" (ackRail src)
                 (ackRail dst) val
108        | src =~ "port:" =
109          printf "# Backward Delay from %s to %s\n" src dst ++
110          printf "set_max_delay -from {%s} -to [get_pin -of_objects {%s} -filter
                 {(is_clock_pin==false) && (direction==in)}] %.3f\n" (ackRail src)
                 (trueRail dst) val ++
111          printf "set_max_delay -from {%s} -to [get_pin -of_objects {%s} -filter
                 {(is_clock_pin==false) && (direction==in)}] %.3f\n" (ackRail src)
                 (falseRail dst) val
112        | dst =~ "port:" =
113          printf "# Backward Delay from %s to %s\n" src dst ++
114          printf "set_max_delay -from [get_pin -of_objects {%s} -filter
                 {is_clock_pin==true}] -to {%s} %.3f\n" (trueRail  src) (ackRail dst)
                 val ++
115          printf "set_max_delay -from [get_pin -of_objects {%s} -filter
                 {is_clock_pin==true}] -to {%s} %.3f\n" (falseRail src) (ackRail dst)
                 val
116        | otherwise =
117          printf "# Backward Delay from %s to %s\n" src dst ++
118          printf "set_max_delay -from [get_pin -of_objects {%s} -filter
                 {is_clock_pin==true}] -to [get_pin -of_objects {%s} -filter
                 {(is_clock_pin==false) && (direction==in)}] %.3f\n" (trueRail  src)
                 (trueRail  dst) val ++
119          printf "set_max_delay -from [get_pin -of_objects {%s} -filter
                 {is_clock_pin==true}] -to [get_pin -of_objects {%s} -filter
                 {(is_clock_pin==false) && (direction==in)}] %.3f\n" (falseRail src)
                 (falseRail dst) val ++
120          printf "set_max_delay -from [get_pin -of_objects {%s} -filter
                 {is_clock_pin==true}] -to [get_pin -of_objects {%s} -filter
                 {(is_clock_pin==false) && (direction==in)}] %.3f\n" (trueRail  src)
                 (falseRail dst) val ++
121          printf "set_max_delay -from [get_pin -of_objects {%s} -filter
                 {is_clock_pin==true}] -to [get_pin -of_objects {%s} -filter
                 {(is_clock_pin==false) && (direction==in)}] %.3f\n" (falseRail src)
                 (trueRail  dst) val
122      maxDelay _ = []
123      separateBus :: String -> (String, String, String)
124      separateBus = (=~ "\\[[0-9]+\\]")
125      ackRail s = let (n, b, _) = separateBus s in
126            n ++ "_ack" ++ b
127      trueRail s
128        | s =~ "port:" = let (n, b, _) = separateBus s in
129            n ++ "_t" ++ b
130        | otherwise = s ++ "/t"
131      falseRail s
132        | s =~ "port:" = let (n, b, _) = separateBus s in
133            n ++ "_f" ++ b
```

```
134         | otherwise = s ++ "/f"
135
136  sdcContent err = errorWithoutStackTrace . printf "Could not solve LP: %s" $
        show err
137
138  printSolution :: (MonadIO m) => LPRet -> m ()
139  printSolution (Data.LinearProgram.GLPK.Success, Just (_, vars)) = liftIO $
140    mapM_ print $ Map.toList vars
141  printSolution err = errorWithoutStackTrace . printf "Could not solve LP: %s" $
        show err
142
143  lpObjective :: LPRet -> Double
144  lpObjective (Data.LinearProgram.GLPK.Success, Just (o, _)) = o
145  lpObjective err = errorWithoutStackTrace . printf "Could not solve LP: %s" $
        show err
146
147  prgMain :: ReaderT PrgOptions IO ()
148  prgMain = do
149    opts <- ask
150    hbcn <- hbcnFromFiles $ inputFiles opts
151    let cycleTime = targetCycleTime opts
152    let relax = relaxEnable opts
153    let minDelay = case minimalDelay opts of
154          x | x < 0 -> cycleTime/10
155            | otherwise -> x
156    let lp = constraintCycleTime hbcn cycleTime minDelay relax
157    result <- liftIO $ glpSolveVars simplexDefaults lp
158    sdc <- sdcContent result
159    when (debugSol opts) $ do
160      let lpfile = outputFile opts ++ ".lp"
161      printSolution result
162      liftIO $ writeLP lpfile lp
163    liftIO $ if lpObjective result > 0.0005 then do
164      printf "Writing constraints to %s\n" (outputFile opts)
165      writeFile (outputFile opts) sdc
166    else
167      errorWithoutStackTrace "Deadlock Found in the Design, not writing
           constraints file"
```

Listing C.2: src/Data/HBCN.hs

```
1  module HBCN
2    ( module HBCN.Timing
3    , module HBCN.Internal
4    ) where
5
6  import           HBCN.Internal
7  import           HBCN.Timing
```

Listing C.3: src/Data/HBCN/Internal.hs – Structural graph to HBCN

```haskell
module HBCN.Internal where

import           Algebra.Graph.Labelled
import           Data.Semigroup         as SG

data StructuralElement = Port String [String]
                       | DataReg String [String]
                       | NullReg String [String]
                       deriving (Show, Read, Eq, Ord)

data Transition = DataTrans {nodeName :: String}
                | NullTrans {nodeName :: String}
                deriving (Show, Read, Eq, Ord)

data Place = Unconnected
           | Place {hasToken :: Bool}
           | MindelayPlace {hasToken :: Bool}
           deriving (Show, Read, Eq, Ord)

type HBCN = Graph Place Transition

instance Semigroup Place where
  Unconnected <> a = a
  a <> _ = a

instance Monoid Place where
  mempty = Unconnected
  mappend = (SG.<>)

createHBCNFromStructure :: [StructuralElement] -> HBCN
createHBCNFromStructure = edges . concatMap go where
  go (Port src dst) = concatMap
    (\x -> [(Place False, DataTrans src, DataTrans x)
           ,(Place False, NullTrans src, NullTrans x)
           ,(Place False, DataTrans x,   NullTrans src)
           ,(Place True,  NullTrans x,   DataTrans src)]) dst
  go (NullReg src dst) = concatMap
    (\x -> [(Place False, DataTrans src, DataTrans x)
           ,(Place False, NullTrans src, NullTrans x)
           ,(Place False, DataTrans x,   NullTrans src)
           ,(Place True,  NullTrans x,   DataTrans src)]) dst
  go (DataReg src dst) =
    let
      sout = src ++ "/sout"
      sin  = src ++ "/sin"
    in -- Input Slave
```

```
47        [(MindelayPlace False, DataTrans src,  DataTrans sin)
48        ,(MindelayPlace True,  NullTrans src,  NullTrans sin)
49        ,(MindelayPlace False, DataTrans sin,  NullTrans src)
50        ,(MindelayPlace False, NullTrans sin,  DataTrans src)
51        -- Data stage
52        ,(MindelayPlace True,  DataTrans sin,  DataTrans sout)
53        ,(MindelayPlace False, NullTrans sin,  NullTrans sout)
54        ,(MindelayPlace False, DataTrans sout, NullTrans sin)
55        ,(MindelayPlace False, NullTrans sout, DataTrans sin)] ++
56        -- Output slave
57     concatMap (\x -> [(Place False, DataTrans sout, DataTrans x)
58                      ,(Place False, NullTrans sout, NullTrans x)
59                      ,(Place False, DataTrans x,    NullTrans sout)
60                      ,(Place True,  NullTrans x,    DataTrans sout)]) dst
```

Listing C.4: src/Data/HBCN/Timing.hs – HBCN cycle time constrainer

```
1  {-# LANGUAGE FlexibleContexts #-}
2  module HBCN.Timing
3    (LPVar (..)
4    ,TimingLP
5    ,constraintCycleTime) where
6
7  import           Algebra.Graph.Labelled
8  import           Control.Monad
9  import           Data.LinearProgram
10 import           HBCN.Internal
11 import           Prelude                hiding (Num (..))
12
13 data LPVar = Arrival Transition
14            | FwDelay String String
15            | BwDelay String String
16            | FwSlack String String
17            | BwSlack String String
18            | PseudoClock
19            deriving (Show, Read, Eq, Ord)
20
21 type TimingLP = LP LPVar Double
22
23 arrivalTimeEq cycleTime minDelay relax (place, src, dst) = do
24   let src' = Arrival src
25   let dst' = Arrival dst
26   let ct = if hasToken place then cycleTime else 0
27   let slack = case (src, dst) of
28         (DataTrans s, DataTrans d) -> FwSlack s d
29         (NullTrans s, NullTrans d) -> FwSlack s d
30         (DataTrans s, NullTrans d) -> BwSlack s d
31         (NullTrans s, DataTrans d) -> BwSlack s d
```

```
32     let delay = case (src, dst) of
33           (DataTrans s, DataTrans d) -> FwDelay s d
34           (NullTrans s, NullTrans d) -> FwDelay s d
35           (DataTrans s, NullTrans d) -> BwDelay s d
36           (NullTrans s, DataTrans d) -> BwDelay s d
37     setVarBounds delay $ LBound minDelay
38     setVarBounds slack $ LBound 0
39     setVarBounds src' $ LBound 0
40     setVarBounds dst' $ LBound 0
41     if relax
42       then do
43       linCombination [(1, src'), (-1, dst'), (1, delay)] `equalTo` ct
44       case place of
45         MindelayPlace _ -> linCombination [(1, delay), (-1, slack)] `equalTo`
               minDelay
46         _ -> linCombination [(1, delay)] `equal` linCombination [(1,
               PseudoClock), (1, slack)]
47       else do
48       linCombination [(1, src'), (-1, dst'), (1, delay), (1, slack)] `equalTo` ct
49       case place of
50         MindelayPlace _ -> linCombination [(1, delay)] `equalTo` minDelay
51         _ -> linCombination [(1, delay)] `equal` linCombination [(1, PseudoClock)]
52
53   constraintCycleTime :: HBCN -> Double -> Double -> Bool -> TimingLP
54   constraintCycleTime hbcn cycleTime minDelay relax = execLPM $ do
55     setDirection Max
56     setObjective $ linCombination [(1, PseudoClock)]
57     setVarBounds PseudoClock $ LBound minDelay
58     mapM_ (arrivalTimeEq cycleTime minDelay relax) $ edgeList hbcn
```

APPENDIX D – THE PSEUDO-SYNCHRONOUS SYNTHESIS SCRIPTS

Listing D.1: circuits/design/syn_st65.tcl

```
1  source ../../scripts/fixnetlist.tcl
2
3  set PERIOD $::env(PERIOD)
4
5  set DESIGN $::env(DESIGN)
6  set OUTDIR output/${PERIOD}
7
8  file mkdir ${OUTDIR}
9
10 shell hbcnConstrainer ${DESIGN}.graph -m 0.2 -t ${PERIOD} -o
       ${OUTDIR}/ncl_constraints.sdc
11
12 set_db auto_super_thread true
13 set_db syn_global_effort high
14 #set_db leakage_power_effort medium
15 set_db syn_opt_effort extreme
16 set_db tns_opto true
17 set_db auto_partition true
18 set_db avoid_tied_inputs true
19
20 read_mmmc st65_mmmc.tcl
21 set_db lef_library {
22      ../../tech/CORE65GPSVT_soc.lef
23      ../../tech/PRHS65_soc.lef
24      ../../tech/cmos065_7m4x0y2z_AP_Worst.lef
25      ../../tech/ASCEND_NCLP65GPSVT.lef
26      ../../tech/ASCEND_NCL65GPSVT.lef
27 }
28
29 source ../../tech/st65.tcl
30
31 read_hdl -sv ../../tech/drexpansion.sv
32 read_hdl -sv ncl_${DESIGN}.v
33 elaborate ${DESIGN}
34
35 init_design
36
37 set insts [all::all_seqs]
38 if {[llength $insts] > 0} {
39      set_size_only $insts true
40 }
41
42 set_db iopt_ultra_optimization true
```

```
43 set_dont_use [nclp_cell_list] true
44 syn_generic
45 syn_map
46
47 syn_opt
48 syn_opt -physical
49 syn_opt -incremental -physical
50
51 fix_xnetlist
52
53 opt_sdds 10
54
55 prepare_for_physical
56
57 set_db write_vlog_wor_wand false
58 report_timing -nworst 1000 > ${OUTDIR}/timing.rpt
59 report_gates > ${OUTDIR}/gates.rpt
60 report_power > ${OUTDIR}/power.rpt
61 write_hdl > ${OUTDIR}/logical.v
62 write_snapshot -innovus -outdir ${OUTDIR}/snapshot -tag logical
63
64 set_analysis_view -setup worst_latch_view
65 write_sdf > ${OUTDIR}/worst.sdf
66
67 set_analysis_view -setup nominal_latch_view
68 write_sdf > ${OUTDIR}/nominal.sdf
69
70 set_analysis_view -setup best_latch_view
71 write_sdf > ${OUTDIR}/best.sdf
```

Listing D.2: scripts/fixnetlist.tcl

```
1  proc get_adjacent_nodes {node} {
2    variable ret
3    set ret [list]
4
5    foreach {a} [get_db -if {.obj_type == pin} \
6                     [get_fanout -max_pin_depth 1 $node]] {
7      if [get_db [vdirname $a] .is_combinational] {
8        lappend ret $a
9      }
10   }
11
12   return $ret
13 }
14
15 proc get_instance_of_pin {pin} {
16   return [vdirname $pin]
```

```
17  }
18
19  proc get_inst_cell_name {inst} {
20    return [vbasename [get_db $inst .lib_cell]]
21  }
22
23  proc replace_inst_gate {inst gate} {
24    change_link -instances $inst -lib_cell [lindex [vfind / -lib_cell $gate] 0]
25  }
26
27  proc get_barriers_output {} {
28    return [all::all_seqs -output_pins]
29  }
30
31  proc is_nclp {instance} {
32    global nclp_cell
33    return [info exists nclp_cell([get_inst_cell_name ${instance}])]
34  }
35
36  proc is_ncl {instance} {
37    global ncl_cell
38    return [info exists ncl_cell([get_inst_cell_name ${instance}])]
39  }
40
41  proc ncl_cell_list {} {
42    global ncl_cell
43    return [array names ncl_cell]
44  }
45
46  proc nclp_cell_list {} {
47    global nclp_cell
48    return [array names nclp_cell]
49  }
50
51  proc orphan_cell_list {} {
52    global orphan_cell
53    return [array names orphan_cell]
54  }
55
56  proc get_domain_pins {rtzpinsvar rtopinsvar} {
57    global rtz_startpoints
58    global rto_startpoints
59    upvar 1 $rtzpinsvar rtzpinslist
60    upvar 1 $rtopinsvar rtopinslist
61
62    variable rtzpins
63    variable rtopins
64    variable visited
```

```
65
66   if {![info exists rtz_startpoints] || ![info exists rto_startpoints]} {
67     set rtz_startpoints [concat [vfind / -port *] [all::all_seqs -output_pins]]
68   }
69
70   if [info exists rtz_startpoints] {
71     foreach {pin} $rtz_startpoints {
72       foreach {a} [get_adjacent_nodes $pin] {
73         incr rtzpins($a)
74       }
75     }
76   }
77
78   if [info exists rto_startpoints] {
79     foreach {pin} $rto_startpoints {
80       foreach {a} [get_adjacent_nodes $pin] {
81         incr rtopins($a)
82       }
83     }
84   }
85
86   variable done
87   set done 0
88   while {!$done} {
89     set done 1
90     #breath first RTZ pins processing
91     foreach {a} [array names rtzpins] {
92       if {![info exists visited($a)] || !$visited($a)} {
93         if [is_inverting_output $a] {
94           #inversion found
95           foreach {b} [get_adjacent_nodes $a] {
96             incr rtopins($b)
97           }
98         } else {
99           #no domain inversion
100          foreach {b} [get_adjacent_nodes $a] {
101            incr rtzpins($b)
102          }
103        }
104        incr visited($a)
105        set done 0
106      }
107    }
108
109    # breath first RTO pins processing
110    foreach {a} [array names rtopins] {
111      if {![info exists visited($a)] || !$visited($a)} {
112        if [is_inverting_output $a] {
```

```
113             #inversion found
114             foreach {b} [get_adjacent_nodes $a] {
115               incr rtzpins($b)
116             }
117           } else {
118             #no domain inversion
119             foreach {b} [get_adjacent_nodes $a] {
120               incr rtopins($b)
121             }
122           }
123           incr visited($a)
124           set done 0
125         }
126       }
127     }
128 
129     unset -nocomplain rtopinslist
130     unset -nocomplain rtopinslist
131 
132     set rtopinslist [lsort -unique [array names rtopins]]
133     set rtzpinslist [lsort -unique [array names rtzpins]]
134 
135     unset -nocomplain rtzpins
136     unset -nocomplain rtopins
137     unset -nocomplain visited
138 
139     return 1
140   }
141 
142   proc get_domain_instances {rtzinstsvar rtoinstsvar} {
143     upvar 1 $rtzinstsvar rtzinsts
144     upvar 1 $rtoinstsvar rtoinsts
145 
146     variable rtop
147     variable rtzp
148 
149     get_domain_pins rtzp rtop
150 
151     # get instance names from pin names
152     set rtzinsts [list]
153     foreach {a} ${rtzp} {
154       lappend rtzinsts [get_instance_of_pin $a]
155     }
156     set rtzinsts [lsort -unique $rtzinsts]
157 
158     set rtoinsts [list]
159     foreach {a} ${rtop} {
160       lappend rtoinsts [get_instance_of_pin $a]
```

```
161     }
162     set rtoinsts [lsort -unique $rtoinsts]
163
164     unset -nocomplain rtzpins
165     unset -nocomplain rtopins
166
167     return 1
168   }
169
170   proc fix_xnetlist {} {
171     variable rtz
172     variable rto
173     variable count
174     global equivalent_cell
175
176     set count 0
177
178     get_domain_instances rtz rto
179
180     foreach {i} $rto {
181       if {[lsearch -sorted $rtz $i] >= 0} {
182         return -code error "${i} is in both RTO and RTZ domains"
183       }
184     }
185
186     foreach {i} $rtz {
187       if {[is_nclp $i]} {
188         variable ln
189         variable nln
190         set ln [get_inst_cell_name $i]
191         set nln $equivalent_cell($ln)
192         puts "Replacing ${i} from ${ln} to ${nln}"
193         replace_inst_gate $i $nln
194         incr count
195       }
196     }
197
198     foreach {i} $rto {
199       if {[is_ncl $i]} {
200         variable ln
201         variable nln
202         set ln [get_inst_cell_name $i]
203         set nln $equivalent_cell($ln)
204         puts "Replacing ${i} from ${ln} to ${nln}"
205         replace_inst_gate $i $nln
206         incr count
207       }
208     }
```

```
209
210   return $count
211 }
212
213 proc check_sdds_consistency {} {
214   variable rtz
215   variable rto
216   variable ret
217
218   get_domain_instances rtz rto
219
220   set ret 1
221
222   foreach {i} $rto {
223     if {[lsearch -sorted $rtz $i] >= 0} {
224       puts "${i} is in both RTO and RTZ domains"
225       set ret 0
226     }
227   }
228
229   foreach {i} $rtz {
230     if {[is_nclp $i]} {
231       puts "${i} is a NCLP cell in the RTZ domain"
232       set ret 0
233     }
234   }
235
236   foreach {i} $rto {
237     if {[is_ncl $i]} {
238       puts "${i} is a NCL cell in the RTO domain"
239       set ret 0
240     }
241   }
242
243   return $ret
244 }
245
246 proc opt_rtz {} {
247
248   puts "Optimising RTZ Gates"
249
250   set_dont_use [ncl_cell_list] false
251   set_dont_touch [ncl_cell_list] false
252   set_dont_use [nclp_cell_list] true
253   set_dont_touch [nclp_cell_list] true
254   syn_opt -incremental -physical
255
256   set_dont_use [nclp_cell_list] false
```

```
257   set_dont_touch [nclp_cell_list] false
258
259   return [fix_xnetlist]
260 }
261
262 proc opt_rto {} {
263
264   puts "Optimising RTO Gates"
265
266   set_dont_use [ncl_cell_list] true
267   set_dont_touch [ncl_cell_list] true
268   set_dont_use [nclp_cell_list] false
269   set_dont_touch [nclp_cell_list] false
270   syn_opt -incremental -physical
271
272   set_dont_use [ncl_cell_list] false
273   set_dont_touch [ncl_cell_list] false
274
275   return [fix_xnetlist]
276 }
277
278 proc opt_sdds {maxit} {
279   variable it
280
281   set it 0
282   phys_opt_rto
283   phys_opt_rtz
284
285   while {$it < $maxit} {
286     puts "SDDS Optimisation iteration ${it}"
287     incr it
288
289     if [opt_rto] {
290       phys_opt_rtz
291     }
292
293     if [opt_rtz] {
294       phys_opt_rto
295     }
296
297     if {[get_db [current_design] .slack] >= 0} {
298       break
299     }
300   }
301
302   return $it
303 }
304
```

```
305 proc phys_opt_rtz {} {
306
307   puts "Physically Optimising RTZ Gates"
308
309   set_dont_touch [all::all_insts] false
310   set sequential_insts [all::all_seqs]
311   if {[llength $sequential_insts] > 0} {
312     set_size_only true $sequential_insts
313   }
314
315   set_dont_use [ncl_cell_list] false
316   set_dont_touch [ncl_cell_list] false
317   set_dont_use [nclp_cell_list] true
318   set_dont_touch [nclp_cell_list] true
319
320   foreach {i} [all::all_insts] {
321     if [is_ncl $i] {
322       set_size_only $i true
323     }
324   }
325
326   syn_opt -physical
327
328   set_dont_use [nclp_cell_list] false
329   set_dont_touch [nclp_cell_list] false
330
331   set_dont_touch [all::all_insts] false
332   set sequential_insts [all::all_seqs]
333   if {[llength $sequential_insts] > 0} {
334     set_size_only true $sequential_insts
335   }
336
337   return [fix_xnetlist]
338 }
339
340 proc phys_opt_rto {} {
341
342   puts "Physically Optimising RTO Gates"
343
344   set_dont_touch [all::all_insts] false
345   set sequential_insts [all::all_seqs]
346   if {[llength $sequential_insts] > 0} {
347     set_size_only true $sequential_insts
348   }
349
350   set_dont_use [ncl_cell_list] true
351   set_dont_touch [ncl_cell_list] true
352   set_dont_use [nclp_cell_list] false
```

```
353   set_dont_touch [nclp_cell_list] false
354
355   foreach {i} [all::all_insts] {
356     if [is_nclp $i] {
357       set_size_only $i true
358     }
359   }
360
361   syn_opt -physical
362
363   set_dont_use [ncl_cell_list] false
364   set_dont_touch [ncl_cell_list] false
365
366   set_dont_touch [all::all_insts] false
367   set sequential_insts [all::all_seqs]
368   if {[llength $sequential_insts] > 0} {
369     set_size_only true $sequential_insts
370   }
371
372   return [fix_xnetlist]
373 }
374
375 proc phys_opt_sdds {maxit} {
376   variable done
377   variable it
378
379   set done 0
380   set it 0
381
382   while {!$done && $it < $maxit} {
383     puts "SDDS Physical Optimisation iteration ${it}"
384     set done 1
385     incr it
386
387     if [phys_opt_rto] {
388       set done 0
389     }
390
391     if [phys_opt_rtz] {
392       set done 0
393     }
394   }
395
396   return $it
397 }
398
399 proc prepare_for_physical {} {
400   set_dont_touch [all::all_insts] false
```

```
401     set sequential_insts [all::all_seqs]
402     if {[llength $sequential_insts] > 0} {
403       set_size_only true $sequential_insts
404     }
405 
406     set_dont_touch [nclp_cell_list] true
407     set_dont_use [nclp_cell_list] true
408 
409     set_dont_touch [ncl_cell_list] false
410     set_dont_use [ncl_cell_list] false
411 
412     foreach {i} [all::all_insts] {
413       if [is_ncl $i] {
414         set_size_only $i true
415       }
416     }
417 
418     return 1
419   }
420 
421   #forces optimisation to not use constants
422   set_db iopt_force_constant_removal true
423 
424   # Local Variables:
425   # mode: tcl
426   # tcl-indent-level: 2
427   # indent-tabs-mode: nil
428   # End:
```

Listing D.3: circuits/design/st65_mmmc.tcl

```
 1  ## library_sets
 2  create_library_set -name worst_flop_libset \
 3      -timing { ../../tech/CORE65GPSVT_wc_0.90V_125C.lib \
 4                ../../tech/ASCEND_NCL65GPSVT_SS_0.90V_125C_letiflop.lib \
 5                ../../tech/ASCEND_NCLP65GPSVT_SS_0.90V_125C.lib }
 6 
 7  create_library_set -name nominal_flop_libset \
 8      -timing { ../../tech/CORE65GPSVT_nom_1.00V_25C.lib \
 9                ../../tech/ASCEND_NCL65GPSVT_TT_1.00V_25C_letiflop.lib \
10                ../../tech/ASCEND_NCLP65GPSVT_TT_1.00V_25C.lib }
11 
12  create_library_set -name best_flop_libset \
13      -timing { ../../tech/CORE65GPSVT_bc_1.10V_m40C.lib \
14                ../../tech/ASCEND_NCL65GPSVT_FF_1.10V_m40C_letiflop.lib \
15                ../../tech/ASCEND_NCLP65GPSVT_FF_1.10V_m40C.lib }
16 
17  create_library_set -name worst_latch_libset \
```

```
18      -timing {  ../../tech/CORE65GPSVT_wc_0.90V_125C.lib \
19                 ../../tech/ASCEND_NCL65GPSVT_SS_0.90V_125C_letilatch.lib \
20                 ../../tech/ASCEND_NCLP65GPSVT_SS_0.90V_125C.lib }
21
22  create_library_set -name nominal_latch_libset \
23      -timing {  ../../tech/CORE65GPSVT_nom_1.00V_25C.lib \
24                 ../../tech/ASCEND_NCL65GPSVT_TT_1.00V_25C_letilatch.lib \
25                 ../../tech/ASCEND_NCLP65GPSVT_TT_1.00V_25C.lib }
26
27  create_library_set -name best_latch_libset \
28      -timing {  ../../tech/CORE65GPSVT_bc_1.10V_m40C.lib \
29                 ../../tech/ASCEND_NCL65GPSVT_FF_1.10V_m40C_letilatch.lib \
30                 ../../tech/ASCEND_NCLP65GPSVT_FF_1.10V_m40C.lib }
31
32  ## opcond
33  create_opcond -name worst_opcond -process 1.2 -voltage 0.9 -temperature 125.0
34  create_opcond -name nominal_opcond -process 1.0 -voltage 1.0 -temperature 25.0
35  create_opcond -name best_opcond -process 0.8 -voltage 1.1 -temperature -40.0
36
37  ## timing_condition
38  create_timing_condition -name worst_flop_timing_cond \
39      -opcond worst_opcond \
40      -library_sets { worst_flop_libset }
41
42  create_timing_condition -name nominal_flop_timing_cond \
43      -opcond nominal_opcond \
44      -library_sets { nominal_flop_libset }
45
46  create_timing_condition -name best_flop_timing_cond \
47      -opcond best_opcond \
48      -library_sets { best_flop_libset }
49
50  create_timing_condition -name worst_latch_timing_cond \
51      -opcond worst_opcond \
52      -library_sets { worst_latch_libset }
53
54  create_timing_condition -name nominal_latch_timing_cond \
55      -opcond nominal_opcond \
56      -library_sets { nominal_latch_libset }
57
58  create_timing_condition -name best_latch_timing_cond \
59      -opcond best_opcond \
60      -library_sets { best_latch_libset }
61
62  ## rc_corner
63  create_rc_corner -name worst_rc_corner \
64      -temperature 125.0 \
65      -cap_table ../../tech/cmos065_7m4x0y2z_AP_Worst.captable
```

```
66
67 create_rc_corner -name nominal_rc_corner \
68     -temperature 25.0 \
69     -cap_table ../../tech/cmos065_7m4x0y2z_AP_Worst.captable
70
71 create_rc_corner -name best_rc_corner \
72     -temperature -40.0 \
73     -cap_table ../../tech/cmos065_7m4x0y2z_AP_Best.captable
74
75 ## delay_corner
76 create_delay_corner -name worst_flop_delay_corner \
77     -timing_condition worst_flop_timing_cond \
78     -rc_corner worst_rc_corner
79
80 create_delay_corner -name nominal_flop_delay_corner \
81     -timing_condition nominal_flop_timing_cond \
82     -rc_corner nominal_rc_corner
83
84 create_delay_corner -name best_flop_delay_corner \
85     -timing_condition best_flop_timing_cond \
86     -rc_corner best_rc_corner
87
88 create_delay_corner -name worst_latch_delay_corner \
89     -timing_condition worst_latch_timing_cond \
90     -rc_corner worst_rc_corner
91
92 create_delay_corner -name nominal_latch_delay_corner \
93     -timing_condition nominal_latch_timing_cond \
94     -rc_corner nominal_rc_corner
95
96 create_delay_corner -name best_latch_delay_corner \
97     -timing_condition best_latch_timing_cond \
98     -rc_corner best_rc_corner
99
100 ## constraint_mode
101 create_constraint_mode -name default_constraints \
102     -sdc_files { constraints.sdc }
103
104 ## analysis_view
105 create_analysis_view -name worst_flop_view \
106     -constraint_mode default_constraints \
107     -delay_corner worst_flop_delay_corner
108
109 create_analysis_view -name nominal_flop_view \
110     -constraint_mode default_constraints \
111     -delay_corner nominal_flop_delay_corner
112
113 create_analysis_view -name best_flop_view \
```

```
114     -constraint_mode default_constraints \
115     -delay_corner best_flop_delay_corner
116
117 create_analysis_view -name worst_latch_view \
118     -constraint_mode default_constraints \
119     -delay_corner worst_latch_delay_corner
120
121 create_analysis_view -name nominal_latch_view \
122     -constraint_mode default_constraints \
123     -delay_corner nominal_latch_delay_corner
124
125 create_analysis_view -name best_latch_view \
126     -constraint_mode default_constraints \
127     -delay_corner best_latch_delay_corner
128
129 ## set_analysis_view
130 set_analysis_view -setup { worst_flop_view }
```

Listing D.4: circuits/design/constraints.sdc

```
1 set_load_unit -picofarads 1
2
3 source ${OUTDIR}/ncl_constraints.sdc
4
5 set_load 0.005 [all_outputs]
6 set_input_transition 0.1 [all_inputs]
7
8 set_ideal_network [get_port clk]
9 set_clock_uncertainty 0.02 [get_clocks]
```

Listing D.5: tech/drexpansion.sv – The component Library elements

```
1  interface drwire();
2     wor t, f;
3     wand ack;
4
5     modport in (input t, f,
6                 output ack);
7     modport out (input ack,
8                  output t, f);
9  endinterface // drwire
10
11 module dff
12   (drwire.in d,
13    drwire.out q,
14    input wire ck,
15    input wire reset);
16    wire       q_t, q_f, ack_in;
17
```

```
18     SY_RNCL2W11OF2X9 f (.A(d.f), .B(ack_in), .Q(q_f), .RN(reset), .G(ck));
19     SY_RNCL2W11OF2X9 t (.A(d.t), .B(ack_in), .Q(q_t), .RN(reset), .G(ck));
20
21     assign ack_in = ~(q.ack);
22     assign d.ack = q_t | q_f;
23     assign q.t = q_t;
24     assign q.f = q_f;
25  endmodule // dff
26
27  module dffs_slave
28    (drwire.in d,
29     drwire.out q,
30     input wire ck,
31     input wire reset);
32
33     wire        ack_in, set, int_t, int_f;
34
35     ST_SNCL2W11OF2X9 t (.A(d.t), .B(ack_in), .Q(int_t), .S(set), .G(ck));
36     SY_RNCL2W11OF2X9 f (.A(d.f), .B(ack_in), .Q(int_f), .RN(reset), .G(ck));
37
38     assign set = ~reset;
39     assign ack_in = ~(q.ack);
40     assign q.t = int_t;
41     assign q.f = int_f;
42     assign d.ack = int_t | int_f;
43  endmodule // dffs_slave
44
45  module dffs
46    (drwire.in d,
47     drwire.out q,
48     input wire ck,
49     input wire sb);
50     wire        int_t, int_f, ack_in;
51
52     drwire out();
53     drwire in();
54
55     SY_RNCL2W11OF2X9 t (.A(d.t), .B(ack_in), .Q(int_t), .RN(sb), .G(ck));
56     SY_RNCL2W11OF2X9 f (.A(d.f), .B(ack_in), .Q(int_f), .RN(sb), .G(ck));
57
58     dff sout (.d(out), .q(q), .reset(sb), .ck(ck));
59     dffs_slave sin (.d(in), .q(out), .reset(sb), .ck(ck));
60
61     assign in.t = int_t;
62     assign in.f = int_f;
63     assign ack_in = ~(in.ack);
64     assign d.ack = int_t | int_f;
65  endmodule // dffr
```

```
66
67 module dffr_slave
68   (drwire.in d,
69    drwire.out q,
70    input wire ck,
71    input wire reset);
72
73    wire       ack_in, set, int_t, int_f;
74
75    SY_RNCL2W11OF2X9 t (.A(d.t), .B(ack_in), .Q(int_t), .RN(reset), .G(ck));
76    ST_SNCL2W11OF2X9 f (.A(d.f), .B(ack_in), .Q(int_f), .S(set), .G(ck));
77
78    assign set = ~reset;
79    assign ack_in = ~(q.ack);
80    assign q.t = int_t;
81    assign q.f = int_f;
82    assign d.ack = int_t | int_f;
83 endmodule; // dffr_slave
84
85 module dffr
86   (drwire.in d,
87    drwire.out q,
88    input wire ck,
89    input wire rb);
90    wire       int_t, int_f, ack_in;
91
92    drwire out();
93    drwire in();
94
95    SY_RNCL2W11OF2X9 t (.A(d.t), .B(ack_in), .Q(int_t), .RN(rb), .G(ck));
96    SY_RNCL2W11OF2X9 f (.A(d.f), .B(ack_in), .Q(int_f), .RN(rb), .G(ck));
97
98    dff sout (.d(out), .q(q), .reset(rb), .ck(ck));
99    dffr_slave sin (.d(in), .q(out), .reset(rb), .ck(ck));
100
101    assign in.t = int_t;
102    assign in.f = int_f;
103    assign ack_in = ~(in.ack);
104    assign d.ack = int_t | int_f;
105 endmodule // dffr
106
107 module nand2
108   (drwire.in a,
109    drwire.in b,
110    drwire.out y);
111
112    assign y.t = a.f & b.f |
113                 a.f & b.t |
```

```
114                 a.t & b.f;
115
116     assign y.f = a.t & b.t;
117     assign a.ack = y.ack;
118     assign b.ack = y.ack;
119 endmodule // nand2
120
121 module nor2
122    (drwire.in a,
123     drwire.in b,
124     drwire.out y);
125
126     assign y.f = a.t & b.t |
127                 a.t & b.f |
128                 a.f & b.t;
129
130     assign y.t = a.f & b.f;
131     assign a.ack = y.ack;
132     assign b.ack = y.ack;
133 endmodule // nor2
134
135 module xor2
136    (drwire.in a,
137     drwire.in b,
138     drwire.out y);
139
140     assign y.t = a.t & b.f |
141                 a.f & b.t;
142
143     assign y.f = a.f & b.f |
144                 a.t & b.t;
145
146     assign a.ack = y.ack;
147     assign b.ack = y.ack;
148 endmodule // xor2
149
150 module buff
151    (drwire.in a,
152     drwire.out y);
153
154     assign y.t = a.t;
155     assign y.f = a.f;
156     assign a.ack = y.ack;
157 endmodule // buff
158
159 module inv
160    (drwire.in a,
161     drwire.out y);
```

```
162
163     assign y.t = a.f;
164     assign y.f = a.t;
165     assign a.ack = y.ack;
166 endmodule // inv
```

Listing D.6: tech/st65.tcl

```
 1 set equivalent_cell(ST_NCL1W111OF3X2) ST_NCLP3W111OF3X2
 2 set equivalent_cell(ST_NCLP3W111OF3X2) ST_NCL1W111OF3X2
 3 set equivalent_cell(ST_NCL1W111OF3X4) ST_NCLP3W111OF3X4
 4 set equivalent_cell(ST_NCLP3W111OF3X4) ST_NCL1W111OF3X4
 5 set equivalent_cell(ST_NCL1W111OF3X7) ST_NCLP3W111OF3X7
 6 set equivalent_cell(ST_NCLP3W111OF3X7) ST_NCL1W111OF3X7
 7 set equivalent_cell(ST_NCL1W111OF3X9) ST_NCLP3W111OF3X9
 8 set equivalent_cell(ST_NCLP3W111OF3X9) ST_NCL1W111OF3X9
 9 set equivalent_cell(ST_NCL1W111OF3X13) ST_NCLP3W111OF3X13
10 set equivalent_cell(ST_NCLP3W111OF3X13) ST_NCL1W111OF3X13
11 set equivalent_cell(ST_NCL1W111OF3X18) ST_NCLP3W111OF3X18
12 set equivalent_cell(ST_NCLP3W111OF3X18) ST_NCL1W111OF3X18
13 set equivalent_cell(ST_INCL3W111OF3X2) ST_INCLP1W111OF3X2
14 set equivalent_cell(ST_INCLP1W111OF3X2) ST_INCL3W111OF3X2
15 set equivalent_cell(ST_INCL3W111OF3X4) ST_INCLP1W111OF3X4
16 set equivalent_cell(ST_INCLP1W111OF3X4) ST_INCL3W111OF3X4
17 set equivalent_cell(ST_INCL3W111OF3X7) ST_INCLP1W111OF3X7
18 set equivalent_cell(ST_INCLP1W111OF3X7) ST_INCL3W111OF3X7
19 set equivalent_cell(ST_INCL3W111OF3X9) ST_INCLP1W111OF3X9
20 set equivalent_cell(ST_INCLP1W111OF3X9) ST_INCL3W111OF3X9
21 set equivalent_cell(ST_INCL3W111OF3X13) ST_INCLP1W111OF3X13
22 set equivalent_cell(ST_INCLP1W111OF3X13) ST_INCL3W111OF3X13
23 set equivalent_cell(ST_INCL3W111OF3X18) ST_INCLP1W111OF3X18
24 set equivalent_cell(ST_INCLP1W111OF3X18) ST_INCL3W111OF3X18
25 set equivalent_cell(ST_INCL3W111OF3X31) ST_INCLP1W111OF3X18
26 set equivalent_cell(ST_INCL4W2211OF4X2) ST_INCLP3W2211OF4X2
27 set equivalent_cell(ST_INCLP3W2211OF4X2) ST_INCL4W2211OF4X2
28 set equivalent_cell(ST_INCL4W2211OF4X4) ST_INCLP3W2211OF4X4
29 set equivalent_cell(ST_INCLP3W2211OF4X4) ST_INCL4W2211OF4X4
30 set equivalent_cell(ST_INCL4W2211OF4X7) ST_INCLP3W2211OF4X7
31 set equivalent_cell(ST_INCLP3W2211OF4X7) ST_INCL4W2211OF4X7
32 set equivalent_cell(ST_INCL4W2211OF4X9) ST_INCLP3W2211OF4X13
33 set equivalent_cell(ST_INCLP3W2211OF4X13) ST_INCL4W2211OF4X9
34 set equivalent_cell(ST_INCL3W2111OF4X2) ST_INCLP3W2111OF4X2
35 set equivalent_cell(ST_INCLP3W2111OF4X2) ST_INCL3W2111OF4X2
36 set equivalent_cell(ST_INCL3W2111OF4X4) ST_INCLP3W2111OF4X4
37 set equivalent_cell(ST_INCLP3W2111OF4X4) ST_INCL3W2111OF4X4
38 set equivalent_cell(ST_INCL3W2111OF4X7) ST_INCLP3W2111OF4X7
39 set equivalent_cell(ST_INCLP3W2111OF4X7) ST_INCL3W2111OF4X7
40 set equivalent_cell(ST_INCL3W2111OF4X9) ST_INCLP3W2111OF4X9
```

```
41  set equivalent_cell(ST_INCLP3W2111OF4X9) ST_INCL3W2111OF4X9
42  set equivalent_cell(ST_INCL3W2111OF4X13) ST_INCLP3W2111OF4X13
43  set equivalent_cell(ST_INCLP3W2111OF4X13) ST_INCL3W2111OF4X13
44  set equivalent_cell(ST_NCLAO21O2OF4X2) ST_NCLPAO2O21OF4X2
45  set equivalent_cell(ST_NCLPAO2O21OF4X2) ST_NCLAO21O2OF4X2
46  set equivalent_cell(ST_NCLAO21O2OF4X4) ST_NCLPAO2O21OF4X4
47  set equivalent_cell(ST_NCLPAO2O21OF4X4) ST_NCLAO21O2OF4X4
48  set equivalent_cell(ST_NCLAO21O2OF4X7) ST_NCLPAO2O21OF4X7
49  set equivalent_cell(ST_NCLPAO2O21OF4X7) ST_NCLAO21O2OF4X7
50  set equivalent_cell(ST_NCLAO21O2OF4X9) ST_NCLPAO2O21OF4X9
51  set equivalent_cell(ST_NCLPAO2O21OF4X9) ST_NCLAO21O2OF4X9
52  set equivalent_cell(ST_NCLAO21O2OF4X13) ST_NCLPAO2O21OF4X13
53  set equivalent_cell(ST_NCLPAO2O21OF4X13) ST_NCLAO21O2OF4X13
54  set equivalent_cell(ST_NCL4W3221OF4X2) ST_NCLP5W3221OF4X2
55  set equivalent_cell(ST_NCLP5W3221OF4X2) ST_NCL4W3221OF4X2
56  set equivalent_cell(ST_NCL4W3221OF4X4) ST_NCLP5W3221OF4X4
57  set equivalent_cell(ST_NCLP5W3221OF4X4) ST_NCL4W3221OF4X4
58  set equivalent_cell(ST_NCLP5W3221OF4X7) ST_NCL4W3221OF4X9
59  set equivalent_cell(ST_NCL4W3221OF4X9) ST_NCLP5W3221OF4X9
60  set equivalent_cell(ST_NCLP5W3221OF4X9) ST_NCL4W3221OF4X9
61  set equivalent_cell(ST_NCL4W3221OF4X13) ST_NCLP5W3221OF4X13
62  set equivalent_cell(ST_NCLP5W3221OF4X13) ST_NCL4W3221OF4X13
63  set equivalent_cell(ST_INCL3W2211OF4X2) ST_INCLP4W2211OF4X4
64  set equivalent_cell(ST_INCL3W2211OF4X4) ST_INCLP4W2211OF4X4
65  set equivalent_cell(ST_INCLP4W2211OF4X4) ST_INCL3W2211OF4X4
66  set equivalent_cell(ST_INCL3W2211OF4X7) ST_INCLP4W2211OF4X7
67  set equivalent_cell(ST_INCLP4W2211OF4X7) ST_INCL3W2211OF4X7
68  set equivalent_cell(ST_INCL3W2211OF4X9) ST_INCLP4W2211OF4X9
69  set equivalent_cell(ST_INCLP4W2211OF4X9) ST_INCL3W2211OF4X9
70  set equivalent_cell(ST_INCLP4W2211OF4X13) ST_INCL3W2211OF4X9
71  set equivalent_cell(ST_INCL1W1111OF4X2) ST_INCLP4W1111OF4X2
72  set equivalent_cell(ST_INCLP4W1111OF4X2) ST_INCL1W1111OF4X2
73  set equivalent_cell(ST_INCL1W1111OF4X4) ST_INCLP4W1111OF4X4
74  set equivalent_cell(ST_INCLP4W1111OF4X4) ST_INCL1W1111OF4X4
75  set equivalent_cell(ST_INCL1W1111OF4X7) ST_INCLP4W1111OF4X7
76  set equivalent_cell(ST_INCLP4W1111OF4X7) ST_INCL1W1111OF4X7
77  set equivalent_cell(ST_INCL1W1111OF4X9) ST_INCLP4W1111OF4X9
78  set equivalent_cell(ST_INCLP4W1111OF4X9) ST_INCL1W1111OF4X9
79  set equivalent_cell(ST_INCL1W1111OF4X13) ST_INCLP4W1111OF4X9
80  set equivalent_cell(ST_INCL1W1111OF4X18) ST_INCLP4W1111OF4X9
81  set equivalent_cell(ST_INCL1W1111OF4X31) ST_INCLP4W1111OF4X9
82  set equivalent_cell(ST_NCL2W1111OF4X2) ST_NCLP3W1111OF4X2
83  set equivalent_cell(ST_NCLP3W1111OF4X2) ST_NCL2W1111OF4X2
84  set equivalent_cell(ST_NCL2W1111OF4X4) ST_NCLP3W1111OF4X4
85  set equivalent_cell(ST_NCLP3W1111OF4X4) ST_NCL2W1111OF4X4
86  set equivalent_cell(ST_NCL2W1111OF4X7) ST_NCLP3W1111OF4X7
87  set equivalent_cell(ST_NCLP3W1111OF4X7) ST_NCL2W1111OF4X7
88  set equivalent_cell(ST_NCL2W1111OF4X9) ST_NCLP3W1111OF4X9
```

```
 89  set equivalent_cell(ST_NCLP3W1111OF4X9) ST_NCL2W1111OF4X9
 90  set equivalent_cell(ST_NCL2W1111OF4X13) ST_NCLP3W1111OF4X13
 91  set equivalent_cell(ST_NCLP3W1111OF4X13) ST_NCL2W1111OF4X13
 92  set equivalent_cell(ST_NCL5W3221OF4X2) ST_NCLP4W3221OF4X2
 93  set equivalent_cell(ST_NCLP4W3221OF4X2) ST_NCL5W3221OF4X2
 94  set equivalent_cell(ST_NCL5W3221OF4X4) ST_NCLP4W3221OF4X4
 95  set equivalent_cell(ST_NCLP4W3221OF4X4) ST_NCL5W3221OF4X4
 96  set equivalent_cell(ST_NCL5W3221OF4X7) ST_NCLP4W3221OF4X7
 97  set equivalent_cell(ST_NCLP4W3221OF4X7) ST_NCL5W3221OF4X7
 98  set equivalent_cell(ST_NCL5W3221OF4X9) ST_NCLP4W3221OF4X9
 99  set equivalent_cell(ST_NCLP4W3221OF4X9) ST_NCL5W3221OF4X9
100  set equivalent_cell(ST_NCL5W3221OF4X13) ST_NCLP4W3221OF4X13
101  set equivalent_cell(ST_NCLP4W3221OF4X13) ST_NCL5W3221OF4X13
102  set equivalent_cell(ST_NCL3W3111OF4X2) ST_NCLP4W3111OF4X2
103  set equivalent_cell(ST_NCLP4W3111OF4X2) ST_NCL3W3111OF4X2
104  set equivalent_cell(ST_NCL3W3111OF4X4) ST_NCLP4W3111OF4X4
105  set equivalent_cell(ST_NCLP4W3111OF4X4) ST_NCL3W3111OF4X4
106  set equivalent_cell(ST_NCL3W3111OF4X7) ST_NCLP4W3111OF4X9
107  set equivalent_cell(ST_NCL3W3111OF4X9) ST_NCLP4W3111OF4X9
108  set equivalent_cell(ST_NCLP4W3111OF4X9) ST_NCL3W3111OF4X9
109  set equivalent_cell(ST_NCL3W3111OF4X13) ST_NCLP4W3111OF4X13
110  set equivalent_cell(ST_NCLP4W3111OF4X13) ST_NCL3W3111OF4X13
111  set equivalent_cell(ST_INCL4W1111OF4X2) ST_INCLP1W1111OF4X2
112  set equivalent_cell(ST_INCLP1W1111OF4X2) ST_INCL4W1111OF4X2
113  set equivalent_cell(ST_INCL4W1111OF4X4) ST_INCLP1W1111OF4X4
114  set equivalent_cell(ST_INCLP1W1111OF4X4) ST_INCL4W1111OF4X4
115  set equivalent_cell(ST_INCL4W1111OF4X7) ST_INCLP1W1111OF4X7
116  set equivalent_cell(ST_INCLP1W1111OF4X7) ST_INCL4W1111OF4X7
117  set equivalent_cell(ST_INCL4W1111OF4X9) ST_INCLP1W1111OF4X9
118  set equivalent_cell(ST_INCLP1W1111OF4X9) ST_INCL4W1111OF4X9
119  set equivalent_cell(ST_INCLP1W1111OF4X13) ST_INCL4W1111OF4X9
120  set equivalent_cell(ST_INCLP1W1111OF4X18) ST_INCL4W1111OF4X9
121  set equivalent_cell(ST_NCL3W211OF3X2) ST_NCLP2W211OF3X2
122  set equivalent_cell(ST_NCLP2W211OF3X2) ST_NCL3W211OF3X2
123  set equivalent_cell(ST_NCL3W211OF3X4) ST_NCLP2W211OF3X4
124  set equivalent_cell(ST_NCLP2W211OF3X4) ST_NCL3W211OF3X4
125  set equivalent_cell(ST_NCL3W211OF3X7) ST_NCLP2W211OF3X7
126  set equivalent_cell(ST_NCLP2W211OF3X7) ST_NCL3W211OF3X7
127  set equivalent_cell(ST_NCL3W211OF3X9) ST_NCLP2W211OF3X9
128  set equivalent_cell(ST_NCLP2W211OF3X9) ST_NCL3W211OF3X9
129  set equivalent_cell(ST_NCL3W211OF3X13) ST_NCLP2W211OF3X13
130  set equivalent_cell(ST_NCLP2W211OF3X13) ST_NCL3W211OF3X13
131  set equivalent_cell(ST_NCL3W211OF3X18) ST_NCLP2W211OF3X18
132  set equivalent_cell(ST_NCLP2W211OF3X18) ST_NCL3W211OF3X18
133  set equivalent_cell(ST_NCL2W211OF3X2) ST_NCLP3W211OF3X2
134  set equivalent_cell(ST_NCLP3W211OF3X2) ST_NCL2W211OF3X2
135  set equivalent_cell(ST_NCL2W211OF3X4) ST_NCLP3W211OF3X4
136  set equivalent_cell(ST_NCLP3W211OF3X4) ST_NCL2W211OF3X4
```

```
137 set equivalent_cell(ST_NCL2W211OF3X7) ST_NCLP3W211OF3X7
138 set equivalent_cell(ST_NCLP3W211OF3X7) ST_NCL2W211OF3X7
139 set equivalent_cell(ST_NCL2W211OF3X9) ST_NCLP3W211OF3X9
140 set equivalent_cell(ST_NCLP3W211OF3X9) ST_NCL2W211OF3X9
141 set equivalent_cell(ST_NCL2W211OF3X13) ST_NCLP3W211OF3X13
142 set equivalent_cell(ST_NCLP3W211OF3X13) ST_NCL2W211OF3X13
143 set equivalent_cell(ST_NCLP3W211OF3X18) ST_NCL2W211OF3X13
144 set equivalent_cell(ST_INCL1W11OF2X2) SY_INCLP2W11OF2X2
145 set equivalent_cell(SY_INCLP2W11OF2X2) ST_INCL1W11OF2X4
146 set equivalent_cell(ST_INCL1W11OF2X4) ST_INCLP2W11OF2X4
147 set equivalent_cell(ST_INCLP2W11OF2X4) ST_INCL1W11OF2X4
148 set equivalent_cell(SY_INCLP2W11OF2X4) ST_INCL1W11OF2X7
149 set equivalent_cell(ST_INCL1W11OF2X7) ST_INCLP2W11OF2X7
150 set equivalent_cell(ST_INCLP2W11OF2X7) ST_INCL1W11OF2X7
151 set equivalent_cell(SY_INCLP2W11OF2X7) ST_INCL1W11OF2X9
152 set equivalent_cell(ST_INCL1W11OF2X9) SY_INCLP2W11OF2X9
153 set equivalent_cell(SY_INCLP2W11OF2X9) ST_INCL1W11OF2X13
154 set equivalent_cell(ST_INCL1W11OF2X13) ST_INCLP2W11OF2X13
155 set equivalent_cell(ST_INCLP2W11OF2X13) ST_INCL1W11OF2X13
156 set equivalent_cell(SY_INCLP2W11OF2X13) ST_INCL1W11OF2X18
157 set equivalent_cell(ST_INCL1W11OF2X18) ST_INCLP2W11OF2X18
158 set equivalent_cell(ST_INCLP2W11OF2X18) ST_INCL1W11OF2X18
159 set equivalent_cell(SY_INCLP2W11OF2X18) ST_INCL1W11OF2X18
160 set equivalent_cell(ST_INCLP2W11OF2X22) ST_INCL1W11OF2X18
161 set equivalent_cell(ST_INCLP2W11OF2X31) ST_INCL1W11OF2X18
162 set equivalent_cell(ST_INCL3W211OF3X2) ST_INCLP2W211OF3X2
163 set equivalent_cell(ST_INCLP2W211OF3X2) ST_INCL3W211OF3X2
164 set equivalent_cell(ST_INCL3W211OF3X4) ST_INCLP2W211OF3X4
165 set equivalent_cell(ST_INCLP2W211OF3X4) ST_INCL3W211OF3X4
166 set equivalent_cell(ST_INCL3W211OF3X7) ST_INCLP2W211OF3X7
167 set equivalent_cell(ST_INCLP2W211OF3X7) ST_INCL3W211OF3X7
168 set equivalent_cell(ST_INCL3W211OF3X9) ST_INCLP2W211OF3X9
169 set equivalent_cell(ST_INCLP2W211OF3X9) ST_INCL3W211OF3X9
170 set equivalent_cell(ST_INCL3W211OF3X13) ST_INCLP2W211OF3X18
171 set equivalent_cell(ST_INCL3W211OF3X18) ST_INCLP2W211OF3X18
172 set equivalent_cell(ST_INCLP2W211OF3X18) ST_INCL3W211OF3X18
173 set equivalent_cell(ST_INCLP2W211OF3X22) ST_INCL3W211OF3X31
174 set equivalent_cell(ST_INCLP2W211OF3X27) ST_INCL3W211OF3X31
175 set equivalent_cell(ST_INCL3W211OF3X31) ST_INCLP2W211OF3X27
176 set equivalent_cell(ST_NCL2W11OF2X2) ST_NCLP1W11OF2X2
177 set equivalent_cell(ST_NCLP1W11OF2X2) ST_NCL2W11OF2X2
178 set equivalent_cell(SY_NCL2W11OF2X2) ST_NCLP1W11OF2X4
179 set equivalent_cell(ST_NCL2W11OF2X4) ST_NCLP1W11OF2X4
180 set equivalent_cell(ST_NCLP1W11OF2X4) ST_NCL2W11OF2X4
181 set equivalent_cell(SY_NCL2W11OF2X4) ST_NCLP1W11OF2X7
182 set equivalent_cell(ST_NCL2W11OF2X7) ST_NCLP1W11OF2X7
183 set equivalent_cell(ST_NCLP1W11OF2X7) ST_NCL2W11OF2X7
184 set equivalent_cell(ST_NCL2W11OF2X9) ST_NCLP1W11OF2X9
```

```
185 set equivalent_cell(ST_NCLP1W11OF2X9) ST_NCL2W11OF2X9
186 set equivalent_cell(SY_NCL2W11OF2X9) ST_NCLP1W11OF2X13
187 set equivalent_cell(ST_NCL2W11OF2X13) ST_NCLP1W11OF2X13
188 set equivalent_cell(ST_NCLP1W11OF2X13) ST_NCL2W11OF2X13
189 set equivalent_cell(SY_NCL2W11OF2X13) ST_NCLP1W11OF2X18
190 set equivalent_cell(ST_NCLP1W11OF2X18) SY_NCL2W11OF2X18
191 set equivalent_cell(SY_NCL2W11OF2X18) ST_NCLP1W11OF2X18
192 set equivalent_cell(ST_NCL4W1111OF4X2) ST_NCLP1W1111OF4X2
193 set equivalent_cell(ST_NCLP1W1111OF4X2) ST_NCL4W1111OF4X2
194 set equivalent_cell(ST_NCL4W1111OF4X4) ST_NCLP1W1111OF4X4
195 set equivalent_cell(ST_NCLP1W1111OF4X4) ST_NCL4W1111OF4X4
196 set equivalent_cell(ST_NCL4W1111OF4X7) ST_NCLP1W1111OF4X7
197 set equivalent_cell(ST_NCLP1W1111OF4X7) ST_NCL4W1111OF4X7
198 set equivalent_cell(ST_NCL4W1111OF4X9) ST_NCLP1W1111OF4X13
199 set equivalent_cell(ST_NCL4W1111OF4X13) ST_NCLP1W1111OF4X13
200 set equivalent_cell(ST_NCLP1W1111OF4X13) ST_NCL4W1111OF4X13
201 set equivalent_cell(ST_NCLP1W1111OF4X18) ST_NCL4W1111OF4X13
202 set equivalent_cell(ST_INCL5W2211OF4X2) ST_INCLP2W2211OF4X2
203 set equivalent_cell(ST_INCLP2W2211OF4X2) ST_INCL5W2211OF4X2
204 set equivalent_cell(ST_INCL5W2211OF4X4) ST_INCLP2W2211OF4X4
205 set equivalent_cell(ST_INCLP2W2211OF4X4) ST_INCL5W2211OF4X4
206 set equivalent_cell(ST_INCL5W2211OF4X7) ST_INCLP2W2211OF4X7
207 set equivalent_cell(ST_INCLP2W2211OF4X7) ST_INCL5W2211OF4X7
208 set equivalent_cell(ST_INCL5W2211OF4X9) ST_INCLP2W2211OF4X9
209 set equivalent_cell(ST_INCLP2W2211OF4X9) ST_INCL5W2211OF4X9
210 set equivalent_cell(ST_INCL5W2211OF4X13) ST_INCLP2W2211OF4X13
211 set equivalent_cell(ST_INCLP2W2211OF4X13) ST_INCL5W2211OF4X13
212 set equivalent_cell(ST_INCL5W2211OF4X18) ST_INCLP2W2211OF4X18
213 set equivalent_cell(ST_INCLP2W2211OF4X18) ST_INCL5W2211OF4X18
214 set equivalent_cell(ST_INCL5W2211OF4X22) ST_INCLP2W2211OF4X18
215 set equivalent_cell(ST_INCL2W2211OF4X2) ST_INCLP5W2211OF4X2
216 set equivalent_cell(ST_INCLP5W2211OF4X2) ST_INCL2W2211OF4X2
217 set equivalent_cell(ST_INCL2W2211OF4X4) ST_INCLP5W2211OF4X4
218 set equivalent_cell(ST_INCLP5W2211OF4X4) ST_INCL2W2211OF4X4
219 set equivalent_cell(ST_INCL2W2211OF4X7) ST_INCLP5W2211OF4X7
220 set equivalent_cell(ST_INCLP5W2211OF4X7) ST_INCL2W2211OF4X7
221 set equivalent_cell(ST_INCL2W2211OF4X9) ST_INCLP5W2211OF4X9
222 set equivalent_cell(ST_INCLP5W2211OF4X9) ST_INCL2W2211OF4X9
223 set equivalent_cell(ST_INCL2W2211OF4X13) ST_INCLP5W2211OF4X13
224 set equivalent_cell(ST_INCLP5W2211OF4X13) ST_INCL2W2211OF4X13
225 set equivalent_cell(ST_INCL2W2211OF4X18) ST_INCLP5W2211OF4X13
226 set equivalent_cell(ST_INCL2W2211OF4X31) ST_INCLP5W2211OF4X13
227 set equivalent_cell(ST_INCL5W3221OF4X2) ST_INCLP4W3221OF4X2
228 set equivalent_cell(ST_INCLP4W3221OF4X2) ST_INCL5W3221OF4X2
229 set equivalent_cell(ST_INCL5W3221OF4X4) ST_INCLP4W3221OF4X4
230 set equivalent_cell(ST_INCLP4W3221OF4X4) ST_INCL5W3221OF4X4
231 set equivalent_cell(ST_INCL5W3221OF4X7) ST_INCLP4W3221OF4X7
232 set equivalent_cell(ST_INCLP4W3221OF4X7) ST_INCL5W3221OF4X7
```

```
233 set equivalent_cell(ST_INCLP4W3221OF4X9) ST_INCL5W3221OF4X7
234 set equivalent_cell(ST_INCLP4W3221OF4X13) ST_INCL5W3221OF4X7
235 set equivalent_cell(ST_NCLAO22OF4X2) ST_NCLPOA22OF4X4
236 set equivalent_cell(ST_NCLAO22OF4X4) ST_NCLPOA22OF4X4
237 set equivalent_cell(ST_NCLPOA22OF4X4) ST_NCLAO22OF4X4
238 set equivalent_cell(ST_NCLAO22OF4X7) ST_NCLPOA22OF4X7
239 set equivalent_cell(ST_NCLPOA22OF4X7) ST_NCLAO22OF4X7
240 set equivalent_cell(ST_NCLAO22OF4X9) ST_NCLPOA22OF4X9
241 set equivalent_cell(ST_NCLPOA22OF4X9) ST_NCLAO22OF4X9
242 set equivalent_cell(ST_NCLAO22OF4X13) ST_NCLPOA22OF4X13
243 set equivalent_cell(ST_NCLPOA22OF4X13) ST_NCLAO22OF4X13
244 set equivalent_cell(ST_INCL4W2111OF4X2) ST_INCLP2W2111OF4X2
245 set equivalent_cell(ST_INCLP2W2111OF4X2) ST_INCL4W2111OF4X2
246 set equivalent_cell(ST_INCL4W2111OF4X4) ST_INCLP2W2111OF4X4
247 set equivalent_cell(ST_INCLP2W2111OF4X4) ST_INCL4W2111OF4X4
248 set equivalent_cell(ST_INCL4W2111OF4X7) ST_INCLP2W2111OF4X7
249 set equivalent_cell(ST_INCLP2W2111OF4X7) ST_INCL4W2111OF4X7
250 set equivalent_cell(ST_INCL4W2111OF4X9) ST_INCLP2W2111OF4X9
251 set equivalent_cell(ST_INCLP2W2111OF4X9) ST_INCL4W2111OF4X9
252 set equivalent_cell(ST_INCL4W2111OF4X13) ST_INCLP2W2111OF4X9
253 set equivalent_cell(ST_NCL4W2211OF4X2) ST_NCLP3W2211OF4X2
254 set equivalent_cell(ST_NCLP3W2211OF4X2) ST_NCL4W2211OF4X2
255 set equivalent_cell(ST_NCLP3W2211OF4X4) ST_NCL4W2211OF4X7
256 set equivalent_cell(ST_NCL4W2211OF4X7) ST_NCLP3W2211OF4X4
257 set equivalent_cell(ST_NCL4W2211OF4X9) ST_NCLP3W2211OF4X13
258 set equivalent_cell(ST_NCL4W2211OF4X13) ST_NCLP3W2211OF4X13
259 set equivalent_cell(ST_NCLP3W2211OF4X13) ST_NCL4W2211OF4X13
260 set equivalent_cell(ST_INCL3W3111OF4X2) ST_INCLP4W3111OF4X2
261 set equivalent_cell(ST_INCLP4W3111OF4X2) ST_INCL3W3111OF4X2
262 set equivalent_cell(ST_INCL3W3111OF4X4) ST_INCLP4W3111OF4X4
263 set equivalent_cell(ST_INCLP4W3111OF4X4) ST_INCL3W3111OF4X4
264 set equivalent_cell(ST_INCL3W3111OF4X7) ST_INCLP4W3111OF4X7
265 set equivalent_cell(ST_INCLP4W3111OF4X7) ST_INCL3W3111OF4X7
266 set equivalent_cell(ST_INCL3W3111OF4X9) ST_INCLP4W3111OF4X9
267 set equivalent_cell(ST_INCLP4W3111OF4X9) ST_INCL3W3111OF4X9
268 set equivalent_cell(ST_INCL3W3111OF4X13) ST_INCLP4W3111OF4X9
269 set equivalent_cell(ST_INCL3W3111OF4X18) ST_INCLP4W3111OF4X9
270 set equivalent_cell(ST_INCL3W3111OF4X31) ST_INCLP4W3111OF4X9
271 set equivalent_cell(ST_NCL2W2211OF4X2) ST_NCLP5W2211OF4X2
272 set equivalent_cell(ST_NCLP5W2211OF4X2) ST_NCL2W2211OF4X2
273 set equivalent_cell(ST_NCL2W2211OF4X4) ST_NCLP5W2211OF4X4
274 set equivalent_cell(ST_NCLP5W2211OF4X4) ST_NCL2W2211OF4X4
275 set equivalent_cell(ST_NCL2W2211OF4X7) ST_NCLP5W2211OF4X7
276 set equivalent_cell(ST_NCLP5W2211OF4X7) ST_NCL2W2211OF4X7
277 set equivalent_cell(ST_NCL2W2211OF4X9) ST_NCLP5W2211OF4X9
278 set equivalent_cell(ST_NCLP5W2211OF4X9) ST_NCL2W2211OF4X9
279 set equivalent_cell(ST_NCL2W2211OF4X13) ST_NCLP5W2211OF4X13
280 set equivalent_cell(ST_NCLP5W2211OF4X13) ST_NCL2W2211OF4X13
```

```
281 set equivalent_cell(ST_INCL2W1111OF4X2) ST_INCLP3W1111OF4X2
282 set equivalent_cell(ST_INCLP3W1111OF4X2) ST_INCL2W1111OF4X2
283 set equivalent_cell(ST_INCL2W1111OF4X4) ST_INCLP3W1111OF4X4
284 set equivalent_cell(ST_INCLP3W1111OF4X4) ST_INCL2W1111OF4X4
285 set equivalent_cell(ST_INCL2W1111OF4X7) ST_INCLP3W1111OF4X9
286 set equivalent_cell(ST_INCLP3W1111OF4X9) ST_INCL2W1111OF4X13
287 set equivalent_cell(ST_INCL2W1111OF4X13) ST_INCLP3W1111OF4X9
288 set equivalent_cell(ST_NCL3W111OF3X2) ST_NCLP1W111OF3X2
289 set equivalent_cell(ST_NCLP1W111OF3X2) ST_NCL3W111OF3X2
290 set equivalent_cell(ST_NCL3W111OF3X4) ST_NCLP1W111OF3X4
291 set equivalent_cell(ST_NCLP1W111OF3X4) ST_NCL3W111OF3X4
292 set equivalent_cell(ST_NCL3W111OF3X7) ST_NCLP1W111OF3X7
293 set equivalent_cell(ST_NCLP1W111OF3X7) ST_NCL3W111OF3X7
294 set equivalent_cell(ST_NCL3W111OF3X9) ST_NCLP1W111OF3X9
295 set equivalent_cell(ST_NCLP1W111OF3X9) ST_NCL3W111OF3X9
296 set equivalent_cell(ST_NCL3W111OF3X13) ST_NCLP1W111OF3X13
297 set equivalent_cell(ST_NCLP1W111OF3X13) ST_NCL3W111OF3X13
298 set equivalent_cell(ST_NCL3W111OF3X18) ST_NCLP1W111OF3X18
299 set equivalent_cell(ST_NCLP1W111OF3X18) ST_NCL3W111OF3X18
300 set equivalent_cell(ST_NCL3W2211OF4X2) ST_NCLP4W2211OF4X2
301 set equivalent_cell(ST_NCLP4W2211OF4X2) ST_NCL3W2211OF4X2
302 set equivalent_cell(ST_NCL3W2211OF4X4) ST_NCLP4W2211OF4X4
303 set equivalent_cell(ST_NCLP4W2211OF4X4) ST_NCL3W2211OF4X4
304 set equivalent_cell(ST_NCL3W2211OF4X7) ST_NCLP4W2211OF4X7
305 set equivalent_cell(ST_NCLP4W2211OF4X7) ST_NCL3W2211OF4X7
306 set equivalent_cell(ST_NCL3W2211OF4X9) ST_NCLP4W2211OF4X13
307 set equivalent_cell(ST_NCL3W2211OF4X13) ST_NCLP4W2211OF4X13
308 set equivalent_cell(ST_NCLP4W2211OF4X13) ST_NCL3W2211OF4X13
309 set equivalent_cell(ST_NCL3W2111OF4X2) ST_NCLP3W2111OF4X2
310 set equivalent_cell(ST_NCLP3W2111OF4X2) ST_NCL3W2111OF4X2
311 set equivalent_cell(ST_NCL3W2111OF4X4) ST_NCLP3W2111OF4X4
312 set equivalent_cell(ST_NCLP3W2111OF4X4) ST_NCL3W2111OF4X4
313 set equivalent_cell(ST_NCL3W2111OF4X7) ST_NCLP3W2111OF4X7
314 set equivalent_cell(ST_NCLP3W2111OF4X7) ST_NCL3W2111OF4X7
315 set equivalent_cell(ST_NCLP3W2111OF4X9) ST_NCL3W2111OF4X13
316 set equivalent_cell(ST_NCL3W2111OF4X13) ST_NCLP3W2111OF4X13
317 set equivalent_cell(ST_NCLP3W2111OF4X13) ST_NCL3W2111OF4X13
318 set equivalent_cell(ST_NCL2W111OF3X2) ST_NCLP2W111OF3X2
319 set equivalent_cell(ST_NCLP2W111OF3X2) ST_NCL2W111OF3X2
320 set equivalent_cell(ST_NCL2W111OF3X4) ST_NCLP2W111OF3X4
321 set equivalent_cell(ST_NCLP2W111OF3X4) ST_NCL2W111OF3X4
322 set equivalent_cell(ST_NCL2W111OF3X7) ST_NCLP2W111OF3X7
323 set equivalent_cell(ST_NCLP2W111OF3X7) ST_NCL2W111OF3X7
324 set equivalent_cell(ST_NCL2W111OF3X9) ST_NCLP2W111OF3X9
325 set equivalent_cell(ST_NCLP2W111OF3X9) ST_NCL2W111OF3X9
326 set equivalent_cell(ST_NCL2W111OF3X13) ST_NCLP2W111OF3X9
327 set equivalent_cell(ST_NCL1W1111OF4X2) ST_NCLP4W1111OF4X2
328 set equivalent_cell(ST_NCLP4W1111OF4X2) ST_NCL1W1111OF4X2
```

```
329 set equivalent_cell(ST_NCL1W1111OF4X4) ST_NCLP4W1111OF4X4
330 set equivalent_cell(ST_NCLP4W1111OF4X4) ST_NCL1W1111OF4X4
331 set equivalent_cell(ST_NCL1W1111OF4X7) ST_NCLP4W1111OF4X7
332 set equivalent_cell(ST_NCLP4W1111OF4X7) ST_NCL1W1111OF4X7
333 set equivalent_cell(ST_NCL1W1111OF4X9) ST_NCLP4W1111OF4X9
334 set equivalent_cell(ST_NCLP4W1111OF4X9) ST_NCL1W1111OF4X9
335 set equivalent_cell(ST_NCL1W1111OF4X13) ST_NCLP4W1111OF4X13
336 set equivalent_cell(ST_NCLP4W1111OF4X13) ST_NCL1W1111OF4X13
337 set equivalent_cell(ST_NCL1W1111OF4X18) ST_NCLP4W1111OF4X13
338 set equivalent_cell(ST_NCL3W3211OF4X2) ST_NCLP5W3211OF4X2
339 set equivalent_cell(ST_NCLP5W3211OF4X2) ST_NCL3W3211OF4X2
340 set equivalent_cell(ST_NCL3W3211OF4X4) ST_NCLP5W3211OF4X4
341 set equivalent_cell(ST_NCLP5W3211OF4X4) ST_NCL3W3211OF4X4
342 set equivalent_cell(ST_NCL3W3211OF4X7) ST_NCLP5W3211OF4X7
343 set equivalent_cell(ST_NCLP5W3211OF4X7) ST_NCL3W3211OF4X7
344 set equivalent_cell(ST_NCL3W3211OF4X9) ST_NCLP5W3211OF4X9
345 set equivalent_cell(ST_NCLP5W3211OF4X9) ST_NCL3W3211OF4X9
346 set equivalent_cell(ST_NCL3W3211OF4X13) ST_NCLP5W3211OF4X13
347 set equivalent_cell(ST_NCLP5W3211OF4X13) ST_NCL3W3211OF4X13
348 set equivalent_cell(ST_INCL4W2321OF4X2) ST_INCLP5W2321OF4X2
349 set equivalent_cell(ST_INCLP5W2321OF4X2) ST_INCL4W2321OF4X2
350 set equivalent_cell(ST_INCL4W2321OF4X4) ST_INCLP5W2321OF4X7
351 set equivalent_cell(ST_INCL4W2321OF4X7) ST_INCLP5W2321OF4X7
352 set equivalent_cell(ST_INCLP5W2321OF4X7) ST_INCL4W2321OF4X7
353 set equivalent_cell(ST_INCL4W2321OF4X9) ST_INCLP5W2321OF4X7
354 set equivalent_cell(ST_INCL4W2321OF4X13) ST_INCLP5W2321OF4X7
355 set equivalent_cell(ST_INCL3W3211OF4X2) ST_INCLP5W3211OF4X2
356 set equivalent_cell(ST_INCLP5W3211OF4X2) ST_INCL3W3211OF4X2
357 set equivalent_cell(ST_INCL3W3211OF4X4) ST_INCLP5W3211OF4X4
358 set equivalent_cell(ST_INCLP5W3211OF4X4) ST_INCL3W3211OF4X4
359 set equivalent_cell(ST_INCL3W3211OF4X7) ST_INCLP5W3211OF4X7
360 set equivalent_cell(ST_INCLP5W3211OF4X7) ST_INCL3W3211OF4X7
361 set equivalent_cell(ST_INCL3W3211OF4X9) ST_INCLP5W3211OF4X9
362 set equivalent_cell(ST_INCLP5W3211OF4X9) ST_INCL3W3211OF4X9
363 set equivalent_cell(ST_INCL3W3211OF4X18) ST_INCLP5W3211OF4X9
364 set equivalent_cell(ST_INCL3W3211OF4X22) ST_INCLP5W3211OF4X9
365 set equivalent_cell(ST_INCL3W3211OF4X31) ST_INCLP5W3211OF4X9
366 set equivalent_cell(ST_INCLAO22OF4X2) ST_INCLPOA22OF4X2
367 set equivalent_cell(ST_INCLPOA22OF4X2) ST_INCLAO22OF4X2
368 set equivalent_cell(ST_INCLAO22OF4X4) ST_INCLPOA22OF4X4
369 set equivalent_cell(ST_INCLPOA22OF4X4) ST_INCLAO22OF4X4
370 set equivalent_cell(ST_INCLAO22OF4X7) ST_INCLPOA22OF4X7
371 set equivalent_cell(ST_INCLPOA22OF4X7) ST_INCLAO22OF4X7
372 set equivalent_cell(ST_INCLAO22OF4X9) ST_INCLPOA22OF4X9
373 set equivalent_cell(ST_INCLPOA22OF4X9) ST_INCLAO22OF4X9
374 set equivalent_cell(ST_INCLAO22OF4X13) ST_INCLPOA22OF4X13
375 set equivalent_cell(ST_INCLPOA22OF4X13) ST_INCLAO22OF4X13
376 set equivalent_cell(ST_INCLAO22OF4X18) ST_INCLPOA22OF4X13
```

```
377 set equivalent_cell (ST_INCLP1W11OF2X2) SY_INCL2W11OF2X2
378 set equivalent_cell (SY_INCL2W11OF2X2) ST_INCLP1W11OF2X4
379 set equivalent_cell (ST_INCL2W11OF2X4) ST_INCLP1W11OF2X4
380 set equivalent_cell (ST_INCLP1W11OF2X4) ST_INCL2W11OF2X4
381 set equivalent_cell (SY_INCL2W11OF2X4) ST_INCLP1W11OF2X4
382 set equivalent_cell (ST_INCL2W11OF2X7) ST_INCLP1W11OF2X9
383 set equivalent_cell (SY_INCL2W11OF2X7) ST_INCLP1W11OF2X9
384 set equivalent_cell (ST_INCLP1W11OF2X9) SY_INCL2W11OF2X9
385 set equivalent_cell (SY_INCL2W11OF2X9) ST_INCLP1W11OF2X13
386 set equivalent_cell (ST_INCL2W11OF2X13) ST_INCLP1W11OF2X13
387 set equivalent_cell (ST_INCLP1W11OF2X13) ST_INCL2W11OF2X13
388 set equivalent_cell (SY_INCL2W11OF2X13) ST_INCLP1W11OF2X18
389 set equivalent_cell (ST_INCL2W11OF2X18) ST_INCLP1W11OF2X18
390 set equivalent_cell (ST_INCLP1W11OF2X18) ST_INCL2W11OF2X18
391 set equivalent_cell (SY_INCL2W11OF2X18) ST_INCLP1W11OF2X18
392 set equivalent_cell (ST_INCL2W11OF2X22) ST_INCLP1W11OF2X27
393 set equivalent_cell (ST_INCLP1W11OF2X27) ST_INCL2W11OF2X31
394 set equivalent_cell (ST_INCL2W11OF2X31) ST_INCLP1W11OF2X27
395 set equivalent_cell (ST_NCLP5W2321OF4X2) ST_NCL4W2321OF4X4
396 set equivalent_cell (ST_NCL4W2321OF4X4) ST_NCLP5W2321OF4X4
397 set equivalent_cell (ST_NCLP5W2321OF4X4) ST_NCL4W2321OF4X4
398 set equivalent_cell (ST_NCL4W2321OF4X7) ST_NCLP5W2321OF4X7
399 set equivalent_cell (ST_NCLP5W2321OF4X7) ST_NCL4W2321OF4X7
400 set equivalent_cell (ST_NCL4W2321OF4X9) ST_NCLP5W2321OF4X9
401 set equivalent_cell (ST_NCLP5W2321OF4X9) ST_NCL4W2321OF4X9
402 set equivalent_cell (ST_NCL4W2321OF4X13) ST_NCLP5W2321OF4X13
403 set equivalent_cell (ST_NCLP5W2321OF4X13) ST_NCL4W2321OF4X13
404 set equivalent_cell (ST_INCL4W3111OF4X2) ST_INCLP3W3111OF4X2
405 set equivalent_cell (ST_INCLP3W3111OF4X2) ST_INCL4W3111OF4X2
406 set equivalent_cell (ST_INCL4W3111OF4X4) ST_INCLP3W3111OF4X7
407 set equivalent_cell (ST_INCL4W3111OF4X7) ST_INCLP3W3111OF4X7
408 set equivalent_cell (ST_INCLP3W3111OF4X7) ST_INCL4W3111OF4X7
409 set equivalent_cell (ST_INCL4W3111OF4X9) ST_INCLP3W3111OF4X9
410 set equivalent_cell (ST_INCLP3W3111OF4X9) ST_INCL4W3111OF4X9
411 set equivalent_cell (ST_INCL4W3111OF4X13) ST_INCLP3W3111OF4X13
412 set equivalent_cell (ST_INCLP3W3111OF4X13) ST_INCL4W3111OF4X13
413 set equivalent_cell (ST_INCL4W3111OF4X18) ST_INCLP3W3111OF4X18
414 set equivalent_cell (ST_INCLP3W3111OF4X18) ST_INCL4W3111OF4X18
415 set equivalent_cell (ST_INCL4W3111OF4X22) ST_INCLP3W3111OF4X18
416 set equivalent_cell (ST_INCL2W2111OF4X2) ST_INCLP4W2111OF4X2
417 set equivalent_cell (ST_INCLP4W2111OF4X2) ST_INCL2W2111OF4X2
418 set equivalent_cell (ST_INCL2W2111OF4X4) ST_INCLP4W2111OF4X4
419 set equivalent_cell (ST_INCLP4W2111OF4X4) ST_INCL2W2111OF4X4
420 set equivalent_cell (ST_INCL2W2111OF4X7) ST_INCLP4W2111OF4X9
421 set equivalent_cell (ST_INCL2W2111OF4X9) ST_INCLP4W2111OF4X9
422 set equivalent_cell (ST_INCLP4W2111OF4X9) ST_INCL2W2111OF4X9
423 set equivalent_cell (ST_INCL2W2111OF4X13) ST_INCLP4W2111OF4X13
424 set equivalent_cell (ST_INCLP4W2111OF4X13) ST_INCL2W2111OF4X13
```

```
425 set equivalent_cell(ST_INCL2W111OF3X2) ST_INCLP2W111OF3X2
426 set equivalent_cell(ST_INCLP2W111OF3X2) ST_INCL2W111OF3X2
427 set equivalent_cell(ST_INCL2W111OF3X4) ST_INCLP2W111OF3X4
428 set equivalent_cell(ST_INCLP2W111OF3X4) ST_INCL2W111OF3X4
429 set equivalent_cell(ST_INCL2W111OF3X7) ST_INCLP2W111OF3X7
430 set equivalent_cell(ST_INCLP2W111OF3X7) ST_INCL2W111OF3X7
431 set equivalent_cell(ST_INCL2W111OF3X9) ST_INCLP2W111OF3X9
432 set equivalent_cell(ST_INCLP2W111OF3X9) ST_INCL2W111OF3X9
433 set equivalent_cell(ST_INCL2W111OF3X13) ST_INCLP2W111OF3X13
434 set equivalent_cell(ST_INCLP2W111OF3X13) ST_INCL2W111OF3X13
435 set equivalent_cell(ST_INCL2W111OF3X18) ST_INCLP2W111OF3X18
436 set equivalent_cell(ST_INCLP2W111OF3X18) ST_INCL2W111OF3X18
437 set equivalent_cell(ST_INCLP2W111OF3X22) ST_INCL2W111OF3X18
438 set equivalent_cell(ST_INCL4W3221OF4X2) ST_INCLP5W3221OF4X2
439 set equivalent_cell(ST_INCLP5W3221OF4X2) ST_INCL4W3221OF4X2
440 set equivalent_cell(ST_INCL4W3221OF4X4) ST_INCLP5W3221OF4X4
441 set equivalent_cell(ST_INCLP5W3221OF4X4) ST_INCL4W3221OF4X4
442 set equivalent_cell(ST_INCL4W3221OF4X7) ST_INCLP5W3221OF4X7
443 set equivalent_cell(ST_INCLP5W3221OF4X7) ST_INCL4W3221OF4X7
444 set equivalent_cell(ST_INCL4W3221OF4X9) ST_INCLP5W3221OF4X9
445 set equivalent_cell(ST_INCLP5W3221OF4X9) ST_INCL4W3221OF4X9
446 set equivalent_cell(ST_INCL4W3221OF4X13) ST_INCLP5W3221OF4X13
447 set equivalent_cell(ST_INCLP5W3221OF4X13) ST_INCL4W3221OF4X13
448 set equivalent_cell(ST_NCL1W11OF2X2) ST_NCLP2W11OF2X2
449 set equivalent_cell(ST_NCLP2W11OF2X2) ST_NCL1W11OF2X2
450 set equivalent_cell(SY_NCLP2W11OF2X2) ST_NCL1W11OF2X4
451 set equivalent_cell(ST_NCL1W11OF2X4) ST_NCLP2W11OF2X4
452 set equivalent_cell(ST_NCLP2W11OF2X4) ST_NCL1W11OF2X4
453 set equivalent_cell(SY_NCLP2W11OF2X4) ST_NCL1W11OF2X7
454 set equivalent_cell(ST_NCL1W11OF2X7) ST_NCLP2W11OF2X7
455 set equivalent_cell(ST_NCLP2W11OF2X7) ST_NCL1W11OF2X7
456 set equivalent_cell(ST_NCL1W11OF2X9) ST_NCLP2W11OF2X9
457 set equivalent_cell(ST_NCLP2W11OF2X9) ST_NCL1W11OF2X9
458 set equivalent_cell(SY_NCLP2W11OF2X9) ST_NCL1W11OF2X13
459 set equivalent_cell(ST_NCL1W11OF2X13) ST_NCLP2W11OF2X13
460 set equivalent_cell(ST_NCLP2W11OF2X13) ST_NCL1W11OF2X13
461 set equivalent_cell(SY_NCLP2W11OF2X13) ST_NCL1W11OF2X13
462 set equivalent_cell(SY_NCLP2W11OF2X18) ST_NCL1W11OF2X13
463 set equivalent_cell(ST_NCL2W2111OF4X2) ST_NCLP4W2111OF4X4
464 set equivalent_cell(ST_NCL2W2111OF4X4) ST_NCLP4W2111OF4X4
465 set equivalent_cell(ST_NCLP4W2111OF4X4) ST_NCL2W2111OF4X4
466 set equivalent_cell(ST_NCL2W2111OF4X7) ST_NCLP4W2111OF4X7
467 set equivalent_cell(ST_NCLP4W2111OF4X7) ST_NCL2W2111OF4X7
468 set equivalent_cell(ST_NCL2W2111OF4X9) ST_NCLP4W2111OF4X9
469 set equivalent_cell(ST_NCLP4W2111OF4X9) ST_NCL2W2111OF4X9
470 set equivalent_cell(ST_NCL2W2111OF4X13) ST_NCLP4W2111OF4X13
471 set equivalent_cell(ST_NCLP4W2111OF4X13) ST_NCL2W2111OF4X13
472 set equivalent_cell(ST_INCLOA22OF4X2) ST_INCLPAO22OF4X2
```

```
473 set equivalent_cell (ST_INCLPAO22OF4X2) ST_INCLOA22OF4X2
474 set equivalent_cell (ST_INCLOA22OF4X4) ST_INCLPAO22OF4X4
475 set equivalent_cell (ST_INCLPAO22OF4X4) ST_INCLOA22OF4X4
476 set equivalent_cell (ST_INCLOA22OF4X7) ST_INCLPAO22OF4X7
477 set equivalent_cell (ST_INCLPAO22OF4X7) ST_INCLOA22OF4X7
478 set equivalent_cell (ST_INCLOA22OF4X9) ST_INCLPAO22OF4X9
479 set equivalent_cell (ST_INCLPAO22OF4X9) ST_INCLOA22OF4X9
480 set equivalent_cell (ST_INCLOA22OF4X13) ST_INCLPAO22OF4X13
481 set equivalent_cell (ST_INCLPAO22OF4X13) ST_INCLOA22OF4X13
482 set equivalent_cell (ST_INCLOA22OF4X18) ST_INCLPAO22OF4X13
483 set equivalent_cell (ST_INCL1W111OF3X2) ST_INCLP3W111OF3X2
484 set equivalent_cell (ST_INCLP3W111OF3X2) ST_INCL1W111OF3X2
485 set equivalent_cell (ST_INCL1W111OF3X4) ST_INCLP3W111OF3X4
486 set equivalent_cell (ST_INCLP3W111OF3X4) ST_INCL1W111OF3X4
487 set equivalent_cell (ST_INCL1W111OF3X7) ST_INCLP3W111OF3X7
488 set equivalent_cell (ST_INCLP3W111OF3X7) ST_INCL1W111OF3X7
489 set equivalent_cell (ST_INCL1W111OF3X9) ST_INCLP3W111OF3X9
490 set equivalent_cell (ST_INCLP3W111OF3X9) ST_INCL1W111OF3X9
491 set equivalent_cell (ST_INCL1W111OF3X13) ST_INCLP3W111OF3X13
492 set equivalent_cell (ST_INCLP3W111OF3X13) ST_INCL1W111OF3X13
493 set equivalent_cell (ST_INCL1W111OF3X18) ST_INCLP3W111OF3X18
494 set equivalent_cell (ST_INCLP3W111OF3X18) ST_INCL1W111OF3X18
495 set equivalent_cell (ST_INCL1W111OF3X31) ST_INCLP3W111OF3X31
496 set equivalent_cell (ST_INCLP3W111OF3X31) ST_INCL1W111OF3X31
497 set equivalent_cell (ST_NCL5W2211OF4X2) ST_NCLP2W2211OF4X2
498 set equivalent_cell (ST_NCLP2W2211OF4X2) ST_NCL5W2211OF4X2
499 set equivalent_cell (ST_NCL5W2211OF4X4) ST_NCLP2W2211OF4X4
500 set equivalent_cell (ST_NCLP2W2211OF4X4) ST_NCL5W2211OF4X4
501 set equivalent_cell (ST_NCL5W2211OF4X7) ST_NCLP2W2211OF4X7
502 set equivalent_cell (ST_NCLP2W2211OF4X7) ST_NCL5W2211OF4X7
503 set equivalent_cell (ST_NCL5W2211OF4X9) ST_NCLP2W2211OF4X9
504 set equivalent_cell (ST_NCLP2W2211OF4X9) ST_NCL5W2211OF4X9
505 set equivalent_cell (ST_NCLOA22OF4X2) ST_NCLPAO22OF4X2
506 set equivalent_cell (ST_NCLPAO22OF4X2) ST_NCLOA22OF4X2
507 set equivalent_cell (ST_NCLOA22OF4X4) ST_NCLPAO22OF4X4
508 set equivalent_cell (ST_NCLPAO22OF4X4) ST_NCLOA22OF4X4
509 set equivalent_cell (ST_NCLOA22OF4X7) ST_NCLPAO22OF4X7
510 set equivalent_cell (ST_NCLPAO22OF4X7) ST_NCLOA22OF4X7
511 set equivalent_cell (ST_NCLOA22OF4X9) ST_NCLPAO22OF4X9
512 set equivalent_cell (ST_NCLPAO22OF4X9) ST_NCLOA22OF4X9
513 set equivalent_cell (ST_INCL3W1111OF4X2) ST_INCLP2W1111OF4X2
514 set equivalent_cell (ST_INCLP2W1111OF4X2) ST_INCL3W1111OF4X2
515 set equivalent_cell (ST_INCL3W1111OF4X4) ST_INCLP2W1111OF4X4
516 set equivalent_cell (ST_INCLP2W1111OF4X4) ST_INCL3W1111OF4X4
517 set equivalent_cell (ST_INCL3W1111OF4X7) ST_INCLP2W1111OF4X7
518 set equivalent_cell (ST_INCLP2W1111OF4X7) ST_INCL3W1111OF4X7
519 set equivalent_cell (ST_INCL3W1111OF4X9) ST_INCLP2W1111OF4X9
520 set equivalent_cell (ST_INCLP2W1111OF4X9) ST_INCL3W1111OF4X9
```

```
521 set equivalent_cell(ST_INCLP2W1111OF4X13) ST_INCL3W1111OF4X9
522 set equivalent_cell(ST_NCL3W1111OF4X2) ST_NCLP2W1111OF4X2
523 set equivalent_cell(ST_NCLP2W1111OF4X2) ST_NCL3W1111OF4X2
524 set equivalent_cell(ST_NCL3W1111OF4X4) ST_NCLP2W1111OF4X4
525 set equivalent_cell(ST_NCLP2W1111OF4X4) ST_NCL3W1111OF4X4
526 set equivalent_cell(ST_NCL3W1111OF4X7) ST_NCLP2W1111OF4X7
527 set equivalent_cell(ST_NCLP2W1111OF4X7) ST_NCL3W1111OF4X7
528 set equivalent_cell(ST_NCL3W1111OF4X9) ST_NCLP2W1111OF4X9
529 set equivalent_cell(ST_NCLP2W1111OF4X9) ST_NCL3W1111OF4X9
530 set equivalent_cell(ST_NCL3W1111OF4X13) ST_NCLP2W1111OF4X13
531 set equivalent_cell(ST_NCLP2W1111OF4X13) ST_NCL3W1111OF4X13
532 set equivalent_cell(ST_NCL4W2111OF4X2) ST_NCLP2W2111OF4X2
533 set equivalent_cell(ST_NCLP2W2111OF4X2) ST_NCL4W2111OF4X2
534 set equivalent_cell(ST_NCL4W2111OF4X4) ST_NCLP2W2111OF4X4
535 set equivalent_cell(ST_NCLP2W2111OF4X4) ST_NCL4W2111OF4X4
536 set equivalent_cell(ST_NCL4W2111OF4X7) ST_NCLP2W2111OF4X7
537 set equivalent_cell(ST_NCLP2W2111OF4X7) ST_NCL4W2111OF4X7
538 set equivalent_cell(ST_NCL4W2111OF4X9) ST_NCLP2W2111OF4X9
539 set equivalent_cell(ST_NCLP2W2111OF4X9) ST_NCL4W2111OF4X9
540 set equivalent_cell(ST_NCL4W2111OF4X13) ST_NCLP2W2111OF4X13
541 set equivalent_cell(ST_NCLP2W2111OF4X13) ST_NCL4W2111OF4X13
542 set equivalent_cell(ST_NCL4W3111OF4X2) ST_NCLP3W3111OF4X2
543 set equivalent_cell(ST_NCLP3W3111OF4X2) ST_NCL4W3111OF4X2
544 set equivalent_cell(ST_NCL4W3111OF4X4) ST_NCLP3W3111OF4X4
545 set equivalent_cell(ST_NCLP3W3111OF4X4) ST_NCL4W3111OF4X4
546 set equivalent_cell(ST_NCL4W3111OF4X7) ST_NCLP3W3111OF4X7
547 set equivalent_cell(ST_NCLP3W3111OF4X7) ST_NCL4W3111OF4X7
548 set equivalent_cell(ST_NCL4W3111OF4X9) ST_NCLP3W3111OF4X9
549 set equivalent_cell(ST_NCLP3W3111OF4X9) ST_NCL4W3111OF4X9
550 set equivalent_cell(ST_NCL4W3111OF4X13) ST_NCLP3W3111OF4X9
551 set equivalent_cell(ST_INCL5W3211OF4X2) ST_INCLP3W3211OF4X2
552 set equivalent_cell(ST_INCLP3W3211OF4X2) ST_INCL5W3211OF4X2
553 set equivalent_cell(ST_INCL5W3211OF4X4) ST_INCLP3W3211OF4X4
554 set equivalent_cell(ST_INCLP3W3211OF4X4) ST_INCL5W3211OF4X4
555 set equivalent_cell(ST_INCL5W3211OF4X7) ST_INCLP3W3211OF4X7
556 set equivalent_cell(ST_INCLP3W3211OF4X7) ST_INCL5W3211OF4X7
557 set equivalent_cell(ST_INCL5W3211OF4X9) ST_INCLP3W3211OF4X9
558 set equivalent_cell(ST_INCLP3W3211OF4X9) ST_INCL5W3211OF4X9
559 set equivalent_cell(ST_INCL5W3211OF4X13) ST_INCLP3W3211OF4X13
560 set equivalent_cell(ST_INCLP3W3211OF4X13) ST_INCL5W3211OF4X13
561 set equivalent_cell(ST_INCL5W3211OF4X18) ST_INCLP3W3211OF4X18
562 set equivalent_cell(ST_INCLP3W3211OF4X18) ST_INCL5W3211OF4X18
563 set equivalent_cell(ST_INCL5W3211OF4X22) ST_INCLP3W3211OF4X18
564 set equivalent_cell(ST_INCL2W211OF3X2) ST_INCLP3W211OF3X2
565 set equivalent_cell(ST_INCLP3W211OF3X2) ST_INCL2W211OF3X2
566 set equivalent_cell(ST_INCLP3W211OF3X4) ST_INCL2W211OF3X7
567 set equivalent_cell(ST_INCL2W211OF3X7) ST_INCLP3W211OF3X7
568 set equivalent_cell(ST_INCLP3W211OF3X7) ST_INCL2W211OF3X7
```

```
569  set equivalent_cell(ST_INCL2W211OF3X9) ST_INCLP3W211OF3X9
570  set equivalent_cell(ST_INCLP3W211OF3X9) ST_INCL2W211OF3X9
571  set equivalent_cell(ST_INCL2W211OF3X13) ST_INCLP3W211OF3X13
572  set equivalent_cell(ST_INCLP3W211OF3X13) ST_INCL2W211OF3X13
573  set equivalent_cell(ST_INCL2W211OF3X18) ST_INCLP3W211OF3X18
574  set equivalent_cell(ST_INCLP3W211OF3X18) ST_INCL2W211OF3X18
575  set equivalent_cell(ST_INCL2W211OF3X22) ST_INCLP3W211OF3X27
576  set equivalent_cell(ST_INCLP3W211OF3X27) ST_INCL2W211OF3X31
577  set equivalent_cell(ST_INCL2W211OF3X31) ST_INCLP3W211OF3X27
578  set equivalent_cell(ST_NCL5W3211OF4X2) ST_NCLP3W3211OF4X2
579  set equivalent_cell(ST_NCLP3W3211OF4X2) ST_NCL5W3211OF4X2
580  set equivalent_cell(ST_NCL5W3211OF4X4) ST_NCLP3W3211OF4X7
581  set equivalent_cell(ST_NCL5W3211OF4X7) ST_NCLP3W3211OF4X7
582  set equivalent_cell(ST_NCLP3W3211OF4X7) ST_NCL5W3211OF4X7
583  set equivalent_cell(ST_NCL5W3211OF4X9) ST_NCLP3W3211OF4X9
584  set equivalent_cell(ST_NCLP3W3211OF4X9) ST_NCL5W3211OF4X9
585  set equivalent_cell(ST_NCL5W3211OF4X13) ST_NCLP3W3211OF4X9
586  incr nclp_cell(ST_NCLPOA22OF4X9)
587  incr nclp_cell(ST_NCLP2W111OF3X2)
588  incr nclp_cell(ST_INCLP2W111OF3X2)
589  incr nclp_cell(ST_INCLP2W211OF3X2)
590  incr nclp_cell(ST_INCLP4W2211OF4X4)
591  incr nclp_cell(ST_NCLP5W3221OF4X9)
592  incr nclp_cell(SY_NCLP2W11OF2X18)
593  incr nclp_cell(ST_NCLPAO2O21OF4X9)
594  incr nclp_cell(ST_NCLP2W111OF3X9)
595  incr nclp_cell(ST_INCLP5W3211OF4X7)
596  incr nclp_cell(ST_NCLP4W1111OF4X9)
597  incr nclp_cell(ST_INCLP1W1111OF4X4)
598  incr nclp_cell(ST_INCLP4W1111OF4X9)
599  incr nclp_cell(ST_NCLP1W111OF3X2)
600  incr nclp_cell(ST_INCLP2W2111OF4X4)
601  incr nclp_cell(ST_NCLP4W1111OF4X7)
602  incr nclp_cell(ST_INCLP1W111OF3X18)
603  incr nclp_cell(ST_NCLP4W1111OF4X4)
604  incr nclp_cell(ST_INCLP2W1111OF4X7)
605  incr nclp_cell(ST_INCLP2W11OF2X31)
606  incr nclp_cell(ST_INCLP2W2211OF4X18)
607  incr nclp_cell(ST_INCLP1W11OF2X18)
608  incr nclp_cell(ST_NCLP2W211OF3X4)
609  incr nclp_cell(ST_INCLP4W2111OF4X9)
610  incr nclp_cell(ST_INCLP1W111OF3X13)
611  incr nclp_cell(SY_INCLP2W11OF2X2)
612  incr nclp_cell(ST_INCLP3W2211OF4X13)
613  incr nclp_cell(ST_INCLP2W111OF3X4)
614  incr nclp_cell(ST_INCLP3W111OF3X31)
615  incr nclp_cell(ST_INCLP3W2111OF4X2)
616  incr nclp_cell(ST_NCLP3W2111OF4X2)
```

```
617 incr nclp_cell(ST_INCLP5W3211OF4X4)
618 incr nclp_cell(ST_INCLP1W1111OF4X9)
619 incr nclp_cell(ST_INCLP2W1111OF4X9)
620 incr nclp_cell(ST_NCLPAO2O21OF4X7)
621 incr nclp_cell(ST_INCLP3W111OF3X2)
622 incr nclp_cell(ST_INCLP2W2111OF4X7)
623 incr nclp_cell(ST_INCLP3W211OF3X13)
624 incr nclp_cell(ST_INCLP5W2211OF4X7)
625 incr nclp_cell(ST_NCLP1W11OF2X7)
626 incr nclp_cell(ST_NCLP3W111OF3X9)
627 incr nclp_cell(ST_INCLPAO22OF4X13)
628 incr nclp_cell(ST_INCLP1W11OF2X13)
629 incr nclp_cell(ST_NCLP4W2111OF4X9)
630 incr nclp_cell(ST_NCLP2W2111OF4X9)
631 incr nclp_cell(ST_NCLP2W11OF2X9)
632 incr nclp_cell(ST_NCLP2W211OF3X9)
633 incr nclp_cell(ST_NCLP2W2211OF4X7)
634 incr nclp_cell(ST_INCLP2W11OF2X7)
635 incr nclp_cell(ST_NCLP2W2111OF4X2)
636 incr nclp_cell(ST_NCLP3W3111OF4X7)
637 incr nclp_cell(ST_NCLP3W3111OF4X9)
638 incr nclp_cell(ST_INCLP4W3221OF4X4)
639 incr nclp_cell(ST_INCLP5W2211OF4X9)
640 incr nclp_cell(ST_INCLP5W3221OF4X4)
641 incr nclp_cell(ST_INCLP1W11OF2X27)
642 incr nclp_cell(ST_NCLP1W111OF3X13)
643 incr nclp_cell(ST_NCLP3W211OF3X18)
644 incr nclp_cell(ST_NCLP2W2211OF4X2)
645 incr nclp_cell(ST_INCLP3W211OF3X27)
646 incr nclp_cell(ST_NCLP5W3221OF4X2)
647 incr nclp_cell(ST_INCLP5W3221OF4X2)
648 incr nclp_cell(ST_INCLP5W2211OF4X4)
649 incr nclp_cell(ST_INCLP4W3221OF4X9)
650 incr nclp_cell(ST_INCLP4W1111OF4X7)
651 incr nclp_cell(ST_NCLP5W2321OF4X7)
652 incr nclp_cell(ST_NCLP2W2111OF4X7)
653 incr nclp_cell(ST_NCLP1W11OF2X13)
654 incr nclp_cell(ST_INCLP3W111OF3X9)
655 incr nclp_cell(ST_INCLPAO22OF4X9)
656 incr nclp_cell(ST_NCLP2W2111OF4X4)
657 incr nclp_cell(ST_NCLP5W2211OF4X9)
658 incr nclp_cell(ST_INCLP3W1111OF4X4)
659 incr nclp_cell(ST_INCLP3W2111OF4X9)
660 incr nclp_cell(ST_INCLP4W3111OF4X2)
661 incr nclp_cell(ST_INCLP3W2211OF4X4)
662 incr nclp_cell(ST_INCLP5W2211OF4X13)
663 incr nclp_cell(ST_NCLP1W11OF2X9)
664 incr nclp_cell(ST_INCLP3W3211OF4X18)
```

```
665 incr nclp_cell (ST_INCLP3W3211OF4X4)
666 incr nclp_cell (ST_NCLPOA22OF4X13)
667 incr nclp_cell (ST_INCLP4W3111OF4X9)
668 incr nclp_cell (ST_INCLP4W2111OF4X13)
669 incr nclp_cell (ST_NCLP1W11OF2X2)
670 incr nclp_cell (ST_NCLP4W2111OF4X4)
671 incr nclp_cell (ST_NCLP5W2211OF4X4)
672 incr nclp_cell (ST_NCLP2W111OF3X4)
673 incr nclp_cell (ST_NCLPAO22OF4X2)
674 incr nclp_cell (ST_INCLP3W111OF3X7)
675 incr nclp_cell (ST_NCLP3W3211OF4X7)
676 incr nclp_cell (ST_NCLP3W111OF3X4)
677 incr nclp_cell (ST_INCLP5W2211OF4X2)
678 incr nclp_cell (ST_NCLP5W3211OF4X13)
679 incr nclp_cell (ST_NCLP1W1111OF4X2)
680 incr nclp_cell (ST_NCLP3W1111OF4X2)
681 incr nclp_cell (ST_NCLP3W1111OF4X4)
682 incr nclp_cell (ST_NCLP4W3111OF4X13)
683 incr nclp_cell (ST_INCLP2W2111OF4X2)
684 incr nclp_cell (ST_INCLP2W211OF3X27)
685 incr nclp_cell (ST_INCLP5W2321OF4X7)
686 incr nclp_cell (ST_NCLP5W3221OF4X13)
687 incr nclp_cell (SY_INCLP2W11OF2X18)
688 incr nclp_cell (ST_NCLP5W3211OF4X7)
689 incr nclp_cell (ST_INCLP3W111OF3X18)
690 incr nclp_cell (ST_INCLP2W1111OF4X4)
691 incr nclp_cell (ST_INCLP3W2111OF4X4)
692 incr nclp_cell (ST_INCLP4W2211OF4X13)
693 incr nclp_cell (ST_NCLP1W11OF2X18)
694 incr nclp_cell (ST_INCLPOA22OF4X7)
695 incr nclp_cell (ST_INCLP3W3211OF4X2)
696 incr nclp_cell (ST_NCLP2W11OF2X7)
697 incr nclp_cell (ST_NCLP2W211OF3X7)
698 incr nclp_cell (ST_NCLP3W3211OF4X9)
699 incr nclp_cell (ST_INCLP1W11OF2X9)
700 incr nclp_cell (ST_INCLP3W3211OF4X7)
701 incr nclp_cell (ST_INCLP1W11OF2X4)
702 incr nclp_cell (ST_INCLP1W1111OF4X7)
703 incr nclp_cell (ST_INCLP2W11OF2X22)
704 incr nclp_cell (ST_INCLP1W111OF3X4)
705 incr nclp_cell (ST_INCLP4W1111OF4X2)
706 incr nclp_cell (ST_INCLP3W1111OF4X2)
707 incr nclp_cell (ST_NCLP2W11OF2X4)
708 incr nclp_cell (ST_NCLP4W2111OF4X13)
709 incr nclp_cell (ST_NCLP4W3221OF4X2)
710 incr nclp_cell (ST_NCLP3W111OF3X13)
711 incr nclp_cell (SY_NCLP2W11OF2X4)
712 incr nclp_cell (ST_NCLP2W111OF3X7)
```

```
713 incr nclp_cell (ST_NCLP1W111OF3X7)
714 incr nclp_cell (ST_NCLP4W2211OF4X7)
715 incr nclp_cell (ST_INCLP2W211OF3X18)
716 incr nclp_cell (ST_NCLPAO22OF4X4)
717 incr nclp_cell (ST_NCLP4W3221OF4X7)
718 incr nclp_cell (ST_INCLP5W3221OF4X13)
719 incr nclp_cell (ST_INCLP3W2111OF4X13)
720 incr nclp_cell (ST_INCLP3W3111OF4X2)
721 incr nclp_cell (ST_NCLP5W2321OF4X13)
722 incr nclp_cell (ST_INCLPOA22OF4X9)
723 incr nclp_cell (ST_INCLP1W111OF3X7)
724 incr nclp_cell (ST_NCLP4W2211OF4X2)
725 incr nclp_cell (ST_NCLP5W2321OF4X9)
726 incr nclp_cell (ST_INCLPAO22OF4X4)
727 incr nclp_cell (ST_NCLP3W3111OF4X4)
728 incr nclp_cell (ST_INCLP2W1111OF4X13)
729 incr nclp_cell (ST_INCLP5W3221OF4X9)
730 incr nclp_cell (ST_NCLP1W1111OF4X7)
731 incr nclp_cell (ST_NCLP4W1111OF4X13)
732 incr nclp_cell (SY_INCLP2W11OF2X13)
733 incr nclp_cell (ST_INCLPAO22OF4X2)
734 incr nclp_cell (ST_NCLP2W1111OF4X9)
735 incr nclp_cell (ST_NCLP3W2211OF4X4)
736 incr nclp_cell (ST_NCLP4W3111OF4X4)
737 incr nclp_cell (ST_INCLP4W2111OF4X4)
738 incr nclp_cell (ST_NCLP3W111OF3X7)
739 incr nclp_cell (ST_INCLPAO22OF4X7)
740 incr nclp_cell (ST_NCLP4W2211OF4X13)
741 incr nclp_cell (ST_NCLP2W11OF2X13)
742 incr nclp_cell (ST_INCLP3W211OF3X2)
743 incr nclp_cell (ST_INCLP2W111OF3X9)
744 incr nclp_cell (ST_INCLP4W2211OF4X7)
745 incr nclp_cell (ST_INCLP4W3221OF4X2)
746 incr nclp_cell (ST_NCLP3W211OF3X9)
747 incr nclp_cell (ST_INCLP4W3221OF4X7)
748 incr nclp_cell (ST_NCLP3W2111OF4X7)
749 incr nclp_cell (ST_INCLP2W111OF3X13)
750 incr nclp_cell (ST_NCLP4W3111OF4X9)
751 incr nclp_cell (ST_INCLP2W2211OF4X9)
752 incr nclp_cell (SY_INCLP2W11OF2X9)
753 incr nclp_cell (ST_INCLP1W111OF3X9)
754 incr nclp_cell (ST_NCLP1W11OF2X4)
755 incr nclp_cell (ST_NCLPOA22OF4X4)
756 incr nclp_cell (ST_INCLP1W1111OF4X18)
757 incr nclp_cell (ST_INCLPOA22OF4X13)
758 incr nclp_cell (ST_INCLP2W2211OF4X13)
759 incr nclp_cell (ST_NCLP1W1111OF4X4)
760 incr nclp_cell (ST_NCLP1W111OF3X18)
```

```
761 incr nclp_cell (ST_INCLPOA22OF4X4)
762 incr nclp_cell (ST_NCLPAO2O21OF4X2)
763 incr nclp_cell (ST_NCLP2W2211OF4X4)
764 incr nclp_cell (ST_NCLP3W1111OF4X9)
765 incr nclp_cell (ST_NCLP3W2111OF4X13)
766 incr nclp_cell (ST_INCLP2W11OF2X18)
767 incr nclp_cell (ST_NCLP5W3211OF4X9)
768 incr nclp_cell (ST_INCLP5W3211OF4X9)
769 incr nclp_cell (ST_INCLP3W211OF3X18)
770 incr nclp_cell (ST_NCLP5W2211OF4X2)
771 incr nclp_cell (ST_INCLP3W2211OF4X2)
772 incr nclp_cell (ST_NCLP4W3221OF4X13)
773 incr nclp_cell (ST_INCLP2W2211OF4X2)
774 incr nclp_cell (ST_NCLPAO22OF4X7)
775 incr nclp_cell (ST_NCLP2W1111OF4X13)
776 incr nclp_cell (ST_INCLP3W3111OF4X7)
777 incr nclp_cell (ST_NCLP3W1111OF4X13)
778 incr nclp_cell (ST_NCLP3W2211OF4X13)
779 incr nclp_cell (ST_NCLPOA22OF4X7)
780 incr nclp_cell (SY_INCLP2W11OF2X4)
781 incr nclp_cell (ST_NCLP1W1111OF4X18)
782 incr nclp_cell (ST_NCLP2W211OF3X13)
783 incr nclp_cell (ST_NCLP1W111OF3X4)
784 incr nclp_cell (ST_NCLP2W11OF2X2)
785 incr nclp_cell (ST_INCLP1W1111OF4X13)
786 incr nclp_cell (ST_INCLP3W2211OF4X7)
787 incr nclp_cell (ST_NCLP1W111OF3X9)
788 incr nclp_cell (ST_INCLP4W3111OF4X7)
789 incr nclp_cell (ST_NCLP5W2321OF4X2)
790 incr nclp_cell (ST_INCLP2W111OF3X22)
791 incr nclp_cell (ST_INCLP2W211OF3X9)
792 incr nclp_cell (ST_INCLP4W1111OF4X4)
793 incr nclp_cell (ST_INCLP3W3111OF4X9)
794 incr nclp_cell (ST_NCLP5W2321OF4X4)
795 incr nclp_cell (ST_INCLP5W2321OF4X2)
796 incr nclp_cell (ST_INCLP2W211OF3X7)
797 incr nclp_cell (ST_NCLP4W1111OF4X2)
798 incr nclp_cell (ST_NCLP2W1111OF4X2)
799 incr nclp_cell (ST_INCLP3W3211OF4X9)
800 incr nclp_cell (ST_INCLP3W3111OF4X18)
801 incr nclp_cell (ST_NCLP4W2111OF4X7)
802 incr nclp_cell (ST_NCLP4W3221OF4X9)
803 incr nclp_cell (ST_INCLP2W111OF3X7)
804 incr nclp_cell (ST_INCLP1W11OF2X2)
805 incr nclp_cell (ST_NCLP5W3221OF4X4)
806 incr nclp_cell (ST_NCLPAO22OF4X9)
807 incr nclp_cell (ST_INCLP2W11OF2X4)
808 incr nclp_cell (ST_NCLP5W3221OF4X7)
```

```
809 incr nclp_cell (ST_NCLP2W2211OF4X9)
810 incr nclp_cell (ST_INCLP3W1111OF4X9)
811 incr nclp_cell (ST_NCLP3W111OF3X18)
812 incr nclp_cell (SY_INCLP2W11OF2X7)
813 incr nclp_cell (ST_NCLP3W2111OF4X4)
814 incr nclp_cell (ST_NCLP5W3211OF4X2)
815 incr nclp_cell (ST_NCLP3W3211OF4X2)
816 incr nclp_cell (ST_NCLP5W3211OF4X4)
817 incr nclp_cell (ST_NCLP3W1111OF4X7)
818 incr nclp_cell (ST_NCLP3W211OF3X13)
819 incr nclp_cell (ST_NCLP3W211OF3X4)
820 incr nclp_cell (ST_INCLP3W3211OF4X13)
821 incr nclp_cell (ST_INCLP1W111OF3X2)
822 incr nclp_cell (ST_NCLP4W3111OF4X2)
823 incr nclp_cell (ST_INCLP3W3111OF4X13)
824 incr nclp_cell (ST_NCLPAO2O21OF4X4)
825 incr nclp_cell (ST_INCLP2W11OF2X13)
826 incr nclp_cell (ST_INCLP3W111OF3X4)
827 incr nclp_cell (ST_INCLP3W211OF3X4)
828 incr nclp_cell (ST_NCLP5W2211OF4X7)
829 incr nclp_cell (ST_INCLP4W3221OF4X13)
830 incr nclp_cell (ST_INCLP5W3221OF4X7)
831 incr nclp_cell (ST_INCLP2W2211OF4X4)
832 incr nclp_cell (SY_NCLP2W11OF2X9)
833 incr nclp_cell (ST_INCLPOA22OF4X2)
834 incr nclp_cell (ST_NCLP3W111OF3X2)
835 incr nclp_cell (ST_NCLP2W1111OF4X4)
836 incr nclp_cell (ST_NCLPAO2O21OF4X13)
837 incr nclp_cell (SY_NCLP2W11OF2X13)
838 incr nclp_cell (ST_INCLP2W211OF3X4)
839 incr nclp_cell (ST_NCLP3W2111OF4X9)
840 incr nclp_cell (ST_INCLP5W3211OF4X2)
841 incr nclp_cell (ST_INCLP3W211OF3X7)
842 incr nclp_cell (ST_NCLP4W2211OF4X4)
843 incr nclp_cell (ST_INCLP3W2111OF4X7)
844 incr nclp_cell (ST_NCLP1W1111OF4X13)
845 incr nclp_cell (ST_NCLP2W1111OF4X7)
846 incr nclp_cell (ST_NCLP2W211OF3X2)
847 incr nclp_cell (ST_NCLP3W211OF3X2)
848 incr nclp_cell (ST_NCLP3W3111OF4X2)
849 incr nclp_cell (ST_INCLP4W3111OF4X4)
850 incr nclp_cell (ST_INCLP1W1111OF4X2)
851 incr nclp_cell (ST_INCLP2W2211OF4X7)
852 incr nclp_cell (ST_INCLP4W2111OF4X2)
853 incr nclp_cell (ST_NCLP3W211OF3X7)
854 incr nclp_cell (ST_NCLP5W2211OF4X13)
855 incr nclp_cell (SY_NCLP2W11OF2X2)
856 incr nclp_cell (ST_INCLP2W211OF3X22)
```

```
857  incr  nclp_cell (ST_INCLP3W111OF3X13)
858  incr  nclp_cell (ST_INCLP2W2111OF4X9)
859  incr  nclp_cell (ST_NCLP2W2111OF4X13)
860  incr  nclp_cell (ST_INCLP2W1111OF4X2)
861  incr  nclp_cell (ST_INCLP2W111OF3X18)
862  incr  nclp_cell (ST_NCLP2W211OF3X18)
863  incr  nclp_cell (ST_INCLP3W211OF3X9)
864  incr  nclp_cell (ST_NCLP3W2211OF4X2)
865  incr  nclp_cell (ST_NCLP4W3221OF4X4)
866  incr  nclp_cell (ST_INCLP4W2211OF4X9)
867  incr  ncl_cell (ST_INCL3W111OF3X4)
868  incr  ncl_cell (ST_INCL2W211OF3X31)
869  incr  ncl_cell (ST_INCL4W2321OF4X9)
870  incr  ncl_cell (ST_NCL2W11OF2X13)
871  incr  ncl_cell (ST_NCL2W111OF3X13)
872  incr  ncl_cell (ST_NCL3W2211OF4X13)
873  incr  ncl_cell (ST_INCL5W2211OF4X4)
874  incr  ncl_cell (ST_INCL3W1111OF4X9)
875  incr  ncl_cell (ST_INCL1W11OF2X4)
876  incr  ncl_cell (ST_NCL3W2211OF4X2)
877  incr  ncl_cell (ST_NCL3W3111OF4X4)
878  incr  ncl_cell (ST_INCL2W2211OF4X18)
879  incr  ncl_cell (ST_INCL3W2211OF4X9)
880  incr  ncl_cell (ST_NCLOA22OF4X7)
881  incr  ncl_cell (ST_NCL2W2111OF4X9)
882  incr  ncl_cell (ST_INCL3W1111OF4X4)
883  incr  ncl_cell (ST_NCLAO22OF4X4)
884  incr  ncl_cell (SY_INCL2W11OF2X13)
885  incr  ncl_cell (ST_NCL5W3211OF4X7)
886  incr  ncl_cell (ST_INCL3W111OF3X9)
887  incr  ncl_cell (SY_NCL2W11OF2X2)
888  incr  ncl_cell (ST_INCL2W111OF3X4)
889  incr  ncl_cell (ST_NCL4W2211OF4X2)
890  incr  ncl_cell (ST_NCL3W2111OF4X2)
891  incr  ncl_cell (ST_NCL4W2321OF4X7)
892  incr  ncl_cell (ST_INCL4W3111OF4X4)
893  incr  ncl_cell (ST_NCL3W2211OF4X9)
894  incr  ncl_cell (ST_INCL3W3211OF4X7)
895  incr  ncl_cell (ST_INCL5W3211OF4X13)
896  incr  ncl_cell (ST_NCL1W1111OF4X2)
897  incr  ncl_cell (ST_NCL3W111OF3X2)
898  incr  ncl_cell (ST_NCL3W3211OF4X4)
899  incr  ncl_cell (ST_INCL4W2111OF4X9)
900  incr  ncl_cell (ST_INCL2W111OF3X13)
901  incr  ncl_cell (ST_INCL4W1111OF4X4)
902  incr  ncl_cell (ST_NCL4W2111OF4X13)
903  incr  ncl_cell (ST_INCL2W2211OF4X7)
904  incr  ncl_cell (ST_INCL5W2211OF4X18)
```

```
905  incr ncl_cell(ST_INCL2W211OF3X9)
906  incr ncl_cell(ST_INCL2W2211OF4X4)
907  incr ncl_cell(ST_NCL1W11OF2X2)
908  incr ncl_cell(ST_NCL3W2211OF4X7)
909  incr ncl_cell(ST_NCLOA22OF4X4)
910  incr ncl_cell(ST_INCL3W2211OF4X2)
911  incr ncl_cell(ST_NCL3W3111OF4X13)
912  incr ncl_cell(ST_NCL4W1111OF4X2)
913  incr ncl_cell(ST_NCL5W3221OF4X4)
914  incr ncl_cell(ST_INCL5W3221OF4X7)
915  incr ncl_cell(ST_INCL4W3111OF4X13)
916  incr ncl_cell(ST_INCL4W3111OF4X18)
917  incr ncl_cell(ST_NCL3W1111OF4X13)
918  incr ncl_cell(ST_INCL3W211OF3X18)
919  incr ncl_cell(ST_INCL3W211OF3X2)
920  incr ncl_cell(ST_INCL2W2211OF4X13)
921  incr ncl_cell(ST_INCL3W2211OF4X4)
922  incr ncl_cell(ST_NCL3W211OF3X18)
923  incr ncl_cell(ST_NCL2W211OF3X13)
924  incr ncl_cell(ST_NCL3W3111OF4X9)
925  incr ncl_cell(ST_INCL4W3221OF4X7)
926  incr ncl_cell(SY_NCL2W11OF2X13)
927  incr ncl_cell(ST_INCL4W2321OF4X13)
928  incr ncl_cell(ST_INCL2W2111OF4X13)
929  incr ncl_cell(ST_INCLOA22OF4X18)
930  incr ncl_cell(ST_INCL3W3111OF4X13)
931  incr ncl_cell(ST_INCL3W111OF3X13)
932  incr ncl_cell(ST_INCL4W2111OF4X2)
933  incr ncl_cell(ST_INCL2W2211OF4X31)
934  incr ncl_cell(ST_INCL4W3111OF4X7)
935  incr ncl_cell(ST_NCL1W111OF3X4)
936  incr ncl_cell(ST_NCL5W3211OF4X4)
937  incr ncl_cell(ST_INCLAO22OF4X18)
938  incr ncl_cell(ST_NCL2W211OF3X2)
939  incr ncl_cell(ST_INCL2W2111OF4X4)
940  incr ncl_cell(ST_NCL2W211OF3X9)
941  incr ncl_cell(ST_INCL2W111OF3X2)
942  incr ncl_cell(ST_INCL4W3221OF4X4)
943  incr ncl_cell(ST_NCLAO21O2OF4X13)
944  incr ncl_cell(SY_INCL2W11OF2X9)
945  incr ncl_cell(ST_INCL3W111OF3X2)
946  incr ncl_cell(ST_INCL2W2211OF4X9)
947  incr ncl_cell(ST_NCL4W2111OF4X7)
948  incr ncl_cell(ST_INCL3W1111OF4X7)
949  incr ncl_cell(ST_INCL4W3111OF4X22)
950  incr ncl_cell(ST_INCL1W1111OF4X2)
951  incr ncl_cell(ST_INCL2W11OF2X22)
952  incr ncl_cell(ST_INCL5W2211OF4X7)
```

```
953  incr  ncl_cell(ST_INCL5W3211OF4X4)
954  incr  ncl_cell(ST_NCL2W11OF2X4)
955  incr  ncl_cell(ST_INCL3W2211OF4X7)
956  incr  ncl_cell(ST_INCLOA22OF4X9)
957  incr  ncl_cell(ST_NCL4W3111OF4X13)
958  incr  ncl_cell(ST_INCL3W3211OF4X31)
959  incr  ncl_cell(ST_NCL4W3111OF4X4)
960  incr  ncl_cell(ST_INCL3W2111OF4X13)
961  incr  ncl_cell(ST_NCL4W3221OF4X9)
962  incr  ncl_cell(ST_INCL2W1111OF4X13)
963  incr  ncl_cell(ST_INCL3W3211OF4X22)
964  incr  ncl_cell(ST_NCL2W11OF2X7)
965  incr  ncl_cell(ST_NCL3W111OF3X13)
966  incr  ncl_cell(SY_INCL2W11OF2X7)
967  incr  ncl_cell(ST_NCL2W1111OF4X9)
968  incr  ncl_cell(ST_INCL3W111OF3X31)
969  incr  ncl_cell(ST_NCLOA22OF4X2)
970  incr  ncl_cell(ST_INCL3W3111OF4X7)
971  incr  ncl_cell(ST_INCL2W2211OF4X2)
972  incr  ncl_cell(ST_NCL5W3221OF4X9)
973  incr  ncl_cell(ST_INCL5W2211OF4X2)
974  incr  ncl_cell(ST_NCL2W2111OF4X4)
975  incr  ncl_cell(ST_NCL3W3211OF4X13)
976  incr  ncl_cell(ST_NCL4W2111OF4X9)
977  incr  ncl_cell(ST_NCL4W3111OF4X2)
978  incr  ncl_cell(ST_NCL1W1111OF4X18)
979  incr  ncl_cell(ST_INCLAO22OF4X13)
980  incr  ncl_cell(ST_INCLOA22OF4X4)
981  incr  ncl_cell(ST_NCL3W211OF3X13)
982  incr  ncl_cell(ST_INCL4W2211OF4X4)
983  incr  ncl_cell(ST_INCL1W111OF3X2)
984  incr  ncl_cell(ST_INCL4W3221OF4X13)
985  incr  ncl_cell(ST_NCL3W3111OF4X2)
986  incr  ncl_cell(ST_NCL3W211OF3X9)
987  incr  ncl_cell(ST_NCL4W2211OF4X7)
988  incr  ncl_cell(ST_NCL5W3221OF4X13)
989  incr  ncl_cell(ST_NCL4W1111OF4X9)
990  incr  ncl_cell(ST_INCL4W2211OF4X7)
991  incr  ncl_cell(ST_NCL5W3211OF4X2)
992  incr  ncl_cell(ST_INCL3W2111OF4X4)
993  incr  ncl_cell(ST_NCLAO22OF4X2)
994  incr  ncl_cell(ST_INCL1W111OF3X9)
995  incr  ncl_cell(ST_INCL5W2211OF4X13)
996  incr  ncl_cell(ST_INCL3W3111OF4X18)
997  incr  ncl_cell(ST_NCL3W1111OF4X7)
998  incr  ncl_cell(SY_NCL2W11OF2X18)
999  incr  ncl_cell(ST_INCL2W11OF2X13)
1000 incr  ncl_cell(ST_INCL3W211OF3X4)
```

```
1001 incr ncl_cell(ST_INCL3W3111OF4X31)
1002 incr ncl_cell(ST_INCL4W3221OF4X9)
1003 incr ncl_cell(ST_INCL1W111OF3X18)
1004 incr ncl_cell(ST_INCL1W11OF2X13)
1005 incr ncl_cell(ST_NCL1W111OF3X13)
1006 incr ncl_cell(ST_INCL3W211OF3X7)
1007 incr ncl_cell(ST_NCL1W11OF2X7)
1008 incr ncl_cell(ST_NCL2W2211OF4X7)
1009 incr ncl_cell(ST_NCL2W2211OF4X9)
1010 incr ncl_cell(ST_NCL4W1111OF4X4)
1011 incr ncl_cell(ST_NCL4W3221OF4X4)
1012 incr ncl_cell(ST_INCL3W3211OF4X9)
1013 incr ncl_cell(ST_NCL3W3111OF4X7)
1014 incr ncl_cell(ST_INCL2W1111OF4X4)
1015 incr ncl_cell(ST_NCL3W2111OF4X13)
1016 incr ncl_cell(ST_INCL5W3211OF4X7)
1017 incr ncl_cell(ST_NCL3W111OF3X18)
1018 incr ncl_cell(ST_NCL1W111OF3X7)
1019 incr ncl_cell(ST_NCLAO21O2OF4X4)
1020 incr ncl_cell(ST_INCL2W11OF2X31)
1021 incr ncl_cell(ST_INCL3W2111OF4X7)
1022 incr ncl_cell(ST_INCL1W1111OF4X4)
1023 incr ncl_cell(ST_INCL2W2111OF4X9)
1024 incr ncl_cell(ST_INCL2W111OF3X9)
1025 incr ncl_cell(ST_INCL5W3221OF4X2)
1026 incr ncl_cell(ST_INCLAO22OF4X4)
1027 incr ncl_cell(ST_INCLOA22OF4X2)
1028 incr ncl_cell(ST_NCL5W3211OF4X9)
1029 incr ncl_cell(ST_NCL3W211OF3X2)
1030 incr ncl_cell(ST_INCL3W3211OF4X4)
1031 incr ncl_cell(ST_INCL3W211OF3X9)
1032 incr ncl_cell(ST_NCL2W111OF3X7)
1033 incr ncl_cell(ST_INCL1W111OF3X7)
1034 incr ncl_cell(ST_INCL4W2321OF4X7)
1035 incr ncl_cell(ST_NCL1W1111OF4X7)
1036 incr ncl_cell(ST_NCL2W2111OF4X13)
1037 incr ncl_cell(ST_INCL5W2211OF4X9)
1038 incr ncl_cell(ST_INCL4W2211OF4X9)
1039 incr ncl_cell(ST_NCL4W2321OF4X9)
1040 incr ncl_cell(ST_INCL1W11OF2X18)
1041 incr ncl_cell(ST_NCL4W3221OF4X13)
1042 incr ncl_cell(ST_NCL3W111OF3X7)
1043 incr ncl_cell(ST_INCL4W2321OF4X2)
1044 incr ncl_cell(ST_NCL4W2211OF4X9)
1045 incr ncl_cell(ST_INCL4W2111OF4X4)
1046 incr ncl_cell(ST_NCL1W111OF3X18)
1047 incr ncl_cell(ST_INCL3W3211OF4X2)
1048 incr ncl_cell(ST_INCL2W1111OF4X2)
```

```
1049 incr ncl_cell(ST_NCL4W3111OF4X9)
1050 incr ncl_cell(ST_INCL1W11OF2X9)
1051 incr ncl_cell(ST_NCLAO21O2OF4X9)
1052 incr ncl_cell(ST_NCL3W111OF3X4)
1053 incr ncl_cell(ST_INCL4W2111OF4X13)
1054 incr ncl_cell(ST_INCL1W1111OF4X9)
1055 incr ncl_cell(ST_NCL2W2211OF4X2)
1056 incr ncl_cell(ST_NCL3W2111OF4X7)
1057 incr ncl_cell(ST_INCLAO22OF4X2)
1058 incr ncl_cell(ST_INCL3W3211OF4X18)
1059 incr ncl_cell(ST_INCL1W111OF3X13)
1060 incr ncl_cell(ST_NCL3W1111OF4X9)
1061 incr ncl_cell(ST_NCL4W3221OF4X2)
1062 incr ncl_cell(ST_NCL4W3111OF4X7)
1063 incr ncl_cell(SY_INCL2W11OF2X18)
1064 incr ncl_cell(ST_INCL5W3211OF4X22)
1065 incr ncl_cell(ST_INCL2W2111OF4X7)
1066 incr ncl_cell(ST_INCL2W111OF3X18)
1067 incr ncl_cell(ST_INCL3W211OF3X13)
1068 incr ncl_cell(ST_NCL3W3211OF4X7)
1069 incr ncl_cell(ST_NCL2W11OF2X9)
1070 incr ncl_cell(ST_INCL2W1111OF4X7)
1071 incr ncl_cell(ST_INCLAO22OF4X7)
1072 incr ncl_cell(ST_INCL2W111OF3X7)
1073 incr ncl_cell(ST_INCL5W3211OF4X2)
1074 incr ncl_cell(ST_NCL1W111OF3X9)
1075 incr ncl_cell(ST_INCL3W2111OF4X2)
1076 incr ncl_cell(ST_NCL3W2111OF4X4)
1077 incr ncl_cell(ST_INCL4W3221OF4X2)
1078 incr ncl_cell(ST_INCL5W3211OF4X9)
1079 incr ncl_cell(ST_NCL3W211OF3X7)
1080 incr ncl_cell(ST_INCL2W211OF3X13)
1081 incr ncl_cell(ST_INCLOA22OF4X13)
1082 incr ncl_cell(ST_INCL1W1111OF4X18)
1083 incr ncl_cell(ST_INCLAO22OF4X9)
1084 incr ncl_cell(ST_NCL3W2211OF4X4)
1085 incr ncl_cell(ST_NCLAO22OF4X13)
1086 incr ncl_cell(ST_NCL2W2111OF4X7)
1087 incr ncl_cell(ST_NCL1W11OF2X13)
1088 incr ncl_cell(ST_INCL1W1111OF4X13)
1089 incr ncl_cell(ST_INCL3W111OF3X18)
1090 incr ncl_cell(ST_INCL5W3221OF4X4)
1091 incr ncl_cell(ST_NCL3W1111OF4X4)
1092 incr ncl_cell(SY_NCL2W11OF2X4)
1093 incr ncl_cell(ST_INCL5W2211OF4X22)
1094 incr ncl_cell(ST_INCL3W111OF3X7)
1095 incr ncl_cell(ST_NCLAO21O2OF4X2)
1096 incr ncl_cell(SY_INCL2W11OF2X2)
```

```
1097 incr ncl_cell(ST_INCL3W3111OF4X2)
1098 incr ncl_cell(ST_NCLOA22OF4X9)
1099 incr ncl_cell(ST_INCL4W1111OF4X2)
1100 incr ncl_cell(ST_NCL5W2211OF4X9)
1101 incr ncl_cell(ST_NCL5W3221OF4X7)
1102 incr ncl_cell(ST_NCL2W111OF3X2)
1103 incr ncl_cell(ST_NCL4W1111OF4X13)
1104 incr ncl_cell(ST_INCL4W2111OF4X7)
1105 incr ncl_cell(ST_INCL4W2211OF4X2)
1106 incr ncl_cell(ST_INCL1W1111OF4X31)
1107 incr ncl_cell(ST_NCL3W111OF3X9)
1108 incr ncl_cell(ST_INCL1W11OF2X2)
1109 incr ncl_cell(ST_NCLAO22OF4X9)
1110 incr ncl_cell(ST_NCL1W1111OF4X9)
1111 incr ncl_cell(ST_INCL2W211OF3X2)
1112 incr ncl_cell(ST_INCL2W211OF3X22)
1113 incr ncl_cell(ST_NCL2W211OF3X4)
1114 incr ncl_cell(ST_NCL4W2111OF4X2)
1115 incr ncl_cell(ST_NCL4W2111OF4X4)
1116 incr ncl_cell(ST_INCL1W1111OF4X7)
1117 incr ncl_cell(ST_INCL2W11OF2X18)
1118 incr ncl_cell(ST_INCL2W2111OF4X2)
1119 incr ncl_cell(ST_NCL2W11OF2X2)
1120 incr ncl_cell(ST_NCL4W2211OF4X13)
1121 incr ncl_cell(ST_NCL2W211OF3X7)
1122 incr ncl_cell(ST_NCL3W1111OF4X2)
1123 incr ncl_cell(ST_NCL1W11OF2X4)
1124 incr ncl_cell(ST_NCL1W11OF2X9)
1125 incr ncl_cell(ST_NCL4W2321OF4X4)
1126 incr ncl_cell(ST_NCL3W3211OF4X9)
1127 incr ncl_cell(ST_NCLAO21O2OF4X7)
1128 incr ncl_cell(ST_INCL1W111OF3X31)
1129 incr ncl_cell(SY_NCL2W11OF2X9)
1130 incr ncl_cell(ST_NCL5W2211OF4X2)
1131 incr ncl_cell(ST_INCL1W11OF2X7)
1132 incr ncl_cell(ST_INCL3W2111OF4X9)
1133 incr ncl_cell(ST_INCL3W211OF3X31)
1134 incr ncl_cell(ST_NCL2W2111OF4X2)
1135 incr ncl_cell(ST_INCL4W1111OF4X7)
1136 incr ncl_cell(ST_INCL2W11OF2X7)
1137 incr ncl_cell(ST_NCL2W1111OF4X13)
1138 incr ncl_cell(ST_NCL2W111OF3X9)
1139 incr ncl_cell(ST_INCL2W211OF3X18)
1140 incr ncl_cell(ST_INCL3W3111OF4X4)
1141 incr ncl_cell(ST_INCL5W3211OF4X18)
1142 incr ncl_cell(ST_INCLOA22OF4X7)
1143 incr ncl_cell(ST_INCL4W3111OF4X2)
1144 incr ncl_cell(ST_NCL2W1111OF4X2)
```

```
1145 incr ncl_cell(ST_NCL2W2211OF4X4)
1146 incr ncl_cell(ST_INCL2W211OF3X7)
1147 incr ncl_cell(ST_INCL4W1111OF4X9)
1148 incr ncl_cell(ST_NCL3W211OF3X4)
1149 incr ncl_cell(ST_INCL3W1111OF4X2)
1150 incr ncl_cell(ST_NCL5W2211OF4X7)
1151 incr ncl_cell(SY_INCL2W11OF2X4)
1152 incr ncl_cell(ST_INCL4W3111OF4X9)
1153 incr ncl_cell(ST_INCL3W3111OF4X9)
1154 incr ncl_cell(ST_NCL2W2211OF4X13)
1155 incr ncl_cell(ST_NCL5W3211OF4X13)
1156 incr ncl_cell(ST_NCL2W1111OF4X7)
1157 incr ncl_cell(ST_NCL2W111OF3X4)
1158 incr ncl_cell(ST_NCL4W1111OF4X7)
1159 incr ncl_cell(ST_NCL1W1111OF4X13)
1160 incr ncl_cell(ST_NCL5W2211OF4X4)
1161 incr ncl_cell(ST_NCL5W3221OF4X2)
1162 incr ncl_cell(ST_INCL2W11OF2X4)
1163 incr ncl_cell(ST_INCL4W2321OF4X4)
1164 incr ncl_cell(ST_NCL1W111OF3X2)
1165 incr ncl_cell(ST_NCLAO22OF4X7)
1166 incr ncl_cell(ST_NCL3W3211OF4X2)
1167 incr ncl_cell(ST_NCL1W1111OF4X4)
1168 incr ncl_cell(ST_INCL1W111OF3X4)
1169 incr ncl_cell(ST_NCL2W1111OF4X4)
1170 incr ncl_cell(ST_NCL4W2321OF4X13)
1171 incr orphan_cell(ST_INCLPAO2O21OF4X7)
1172 incr orphan_cell(ST_INCLPAO2O21OF4X9)
1173 incr orphan_cell(ST_INCLPAO2O21OF4X4)
1174 incr orphan_cell(ST_INCLP3W3111OF4X4)
1175 incr inverter_cell(ST_INCL1W1111OF4X13)
1176 incr inverter_cell(ST_INCL1W1111OF4X18)
1177 incr inverter_cell(ST_INCL1W1111OF4X2)
1178 incr inverter_cell(ST_INCL1W1111OF4X31)
1179 incr inverter_cell(ST_INCL1W1111OF4X4)
1180 incr inverter_cell(ST_INCL1W1111OF4X7)
1181 incr inverter_cell(ST_INCL1W1111OF4X9)
1182 incr inverter_cell(ST_INCL1W111OF3X13)
1183 incr inverter_cell(ST_INCL1W111OF3X18)
1184 incr inverter_cell(ST_INCL1W111OF3X2)
1185 incr inverter_cell(ST_INCL1W111OF3X31)
1186 incr inverter_cell(ST_INCL1W111OF3X4)
1187 incr inverter_cell(ST_INCL1W111OF3X7)
1188 incr inverter_cell(ST_INCL1W111OF3X9)
1189 incr inverter_cell(ST_INCL1W11OF2X13)
1190 incr inverter_cell(ST_INCL1W11OF2X18)
1191 incr inverter_cell(ST_INCL1W11OF2X2)
1192 incr inverter_cell(ST_INCL1W11OF2X4)
```

```
1193 incr inverter_cell(ST_INCL1W11OF2X7)
1194 incr inverter_cell(ST_INCL1W11OF2X9)
1195 incr inverter_cell(ST_INCL2W1111OF4X13)
1196 incr inverter_cell(ST_INCL2W1111OF4X2)
1197 incr inverter_cell(ST_INCL2W1111OF4X4)
1198 incr inverter_cell(ST_INCL2W1111OF4X7)
1199 incr inverter_cell(ST_INCL2W111OF3X13)
1200 incr inverter_cell(ST_INCL2W111OF3X18)
1201 incr inverter_cell(ST_INCL2W111OF3X2)
1202 incr inverter_cell(ST_INCL2W111OF3X4)
1203 incr inverter_cell(ST_INCL2W111OF3X7)
1204 incr inverter_cell(ST_INCL2W111OF3X9)
1205 incr inverter_cell(ST_INCL2W11OF2X13)
1206 incr inverter_cell(ST_INCL2W11OF2X18)
1207 incr inverter_cell(ST_INCL2W11OF2X22)
1208 incr inverter_cell(ST_INCL2W11OF2X31)
1209 incr inverter_cell(ST_INCL2W11OF2X4)
1210 incr inverter_cell(ST_INCL2W11OF2X7)
1211 incr inverter_cell(ST_INCL2W2111OF4X13)
1212 incr inverter_cell(ST_INCL2W2111OF4X2)
1213 incr inverter_cell(ST_INCL2W2111OF4X4)
1214 incr inverter_cell(ST_INCL2W2111OF4X7)
1215 incr inverter_cell(ST_INCL2W2111OF4X9)
1216 incr inverter_cell(ST_INCL2W211OF3X13)
1217 incr inverter_cell(ST_INCL2W211OF3X18)
1218 incr inverter_cell(ST_INCL2W211OF3X2)
1219 incr inverter_cell(ST_INCL2W211OF3X22)
1220 incr inverter_cell(ST_INCL2W211OF3X31)
1221 incr inverter_cell(ST_INCL2W211OF3X7)
1222 incr inverter_cell(ST_INCL2W211OF3X9)
1223 incr inverter_cell(ST_INCL2W2211OF4X13)
1224 incr inverter_cell(ST_INCL2W2211OF4X18)
1225 incr inverter_cell(ST_INCL2W2211OF4X2)
1226 incr inverter_cell(ST_INCL2W2211OF4X31)
1227 incr inverter_cell(ST_INCL2W2211OF4X4)
1228 incr inverter_cell(ST_INCL2W2211OF4X7)
1229 incr inverter_cell(ST_INCL2W2211OF4X9)
1230 incr inverter_cell(ST_INCL3W1111OF4X2)
1231 incr inverter_cell(ST_INCL3W1111OF4X4)
1232 incr inverter_cell(ST_INCL3W1111OF4X7)
1233 incr inverter_cell(ST_INCL3W1111OF4X9)
1234 incr inverter_cell(ST_INCL3W111OF3X13)
1235 incr inverter_cell(ST_INCL3W111OF3X18)
1236 incr inverter_cell(ST_INCL3W111OF3X2)
1237 incr inverter_cell(ST_INCL3W111OF3X31)
1238 incr inverter_cell(ST_INCL3W111OF3X4)
1239 incr inverter_cell(ST_INCL3W111OF3X7)
1240 incr inverter_cell(ST_INCL3W111OF3X9)
```

```
1241 incr inverter_cell(ST_INCL3W2111OF4X13)
1242 incr inverter_cell(ST_INCL3W2111OF4X2)
1243 incr inverter_cell(ST_INCL3W2111OF4X4)
1244 incr inverter_cell(ST_INCL3W2111OF4X7)
1245 incr inverter_cell(ST_INCL3W2111OF4X9)
1246 incr inverter_cell(ST_INCL3W211OF3X13)
1247 incr inverter_cell(ST_INCL3W211OF3X18)
1248 incr inverter_cell(ST_INCL3W211OF3X2)
1249 incr inverter_cell(ST_INCL3W211OF3X31)
1250 incr inverter_cell(ST_INCL3W211OF3X4)
1251 incr inverter_cell(ST_INCL3W211OF3X7)
1252 incr inverter_cell(ST_INCL3W211OF3X9)
1253 incr inverter_cell(ST_INCL3W2211OF4X2)
1254 incr inverter_cell(ST_INCL3W2211OF4X4)
1255 incr inverter_cell(ST_INCL3W2211OF4X7)
1256 incr inverter_cell(ST_INCL3W2211OF4X9)
1257 incr inverter_cell(ST_INCL3W3111OF4X13)
1258 incr inverter_cell(ST_INCL3W3111OF4X18)
1259 incr inverter_cell(ST_INCL3W3111OF4X2)
1260 incr inverter_cell(ST_INCL3W3111OF4X31)
1261 incr inverter_cell(ST_INCL3W3111OF4X4)
1262 incr inverter_cell(ST_INCL3W3111OF4X7)
1263 incr inverter_cell(ST_INCL3W3111OF4X9)
1264 incr inverter_cell(ST_INCL3W3211OF4X18)
1265 incr inverter_cell(ST_INCL3W3211OF4X2)
1266 incr inverter_cell(ST_INCL3W3211OF4X22)
1267 incr inverter_cell(ST_INCL3W3211OF4X31)
1268 incr inverter_cell(ST_INCL3W3211OF4X4)
1269 incr inverter_cell(ST_INCL3W3211OF4X7)
1270 incr inverter_cell(ST_INCL3W3211OF4X9)
1271 incr inverter_cell(ST_INCL4W1111OF4X2)
1272 incr inverter_cell(ST_INCL4W1111OF4X4)
1273 incr inverter_cell(ST_INCL4W1111OF4X7)
1274 incr inverter_cell(ST_INCL4W1111OF4X9)
1275 incr inverter_cell(ST_INCL4W2111OF4X13)
1276 incr inverter_cell(ST_INCL4W2111OF4X2)
1277 incr inverter_cell(ST_INCL4W2111OF4X4)
1278 incr inverter_cell(ST_INCL4W2111OF4X7)
1279 incr inverter_cell(ST_INCL4W2111OF4X9)
1280 incr inverter_cell(ST_INCL4W2211OF4X2)
1281 incr inverter_cell(ST_INCL4W2211OF4X4)
1282 incr inverter_cell(ST_INCL4W2211OF4X7)
1283 incr inverter_cell(ST_INCL4W2211OF4X9)
1284 incr inverter_cell(ST_INCL4W2321OF4X13)
1285 incr inverter_cell(ST_INCL4W2321OF4X2)
1286 incr inverter_cell(ST_INCL4W2321OF4X4)
1287 incr inverter_cell(ST_INCL4W2321OF4X7)
1288 incr inverter_cell(ST_INCL4W2321OF4X9)
```

```
1289 incr inverter_cell(ST_INCL4W3111OF4X13)
1290 incr inverter_cell(ST_INCL4W3111OF4X18)
1291 incr inverter_cell(ST_INCL4W3111OF4X2)
1292 incr inverter_cell(ST_INCL4W3111OF4X22)
1293 incr inverter_cell(ST_INCL4W3111OF4X4)
1294 incr inverter_cell(ST_INCL4W3111OF4X7)
1295 incr inverter_cell(ST_INCL4W3111OF4X9)
1296 incr inverter_cell(ST_INCL4W3221OF4X13)
1297 incr inverter_cell(ST_INCL4W3221OF4X2)
1298 incr inverter_cell(ST_INCL4W3221OF4X4)
1299 incr inverter_cell(ST_INCL4W3221OF4X7)
1300 incr inverter_cell(ST_INCL4W3221OF4X9)
1301 incr inverter_cell(ST_INCL5W2211OF4X13)
1302 incr inverter_cell(ST_INCL5W2211OF4X18)
1303 incr inverter_cell(ST_INCL5W2211OF4X2)
1304 incr inverter_cell(ST_INCL5W2211OF4X22)
1305 incr inverter_cell(ST_INCL5W2211OF4X4)
1306 incr inverter_cell(ST_INCL5W2211OF4X7)
1307 incr inverter_cell(ST_INCL5W2211OF4X9)
1308 incr inverter_cell(ST_INCL5W3211OF4X13)
1309 incr inverter_cell(ST_INCL5W3211OF4X18)
1310 incr inverter_cell(ST_INCL5W3211OF4X2)
1311 incr inverter_cell(ST_INCL5W3211OF4X22)
1312 incr inverter_cell(ST_INCL5W3211OF4X4)
1313 incr inverter_cell(ST_INCL5W3211OF4X7)
1314 incr inverter_cell(ST_INCL5W3211OF4X9)
1315 incr inverter_cell(ST_INCL5W3221OF4X2)
1316 incr inverter_cell(ST_INCL5W3221OF4X4)
1317 incr inverter_cell(ST_INCL5W3221OF4X7)
1318 incr inverter_cell(ST_INCLAO22OF4X13)
1319 incr inverter_cell(ST_INCLAO22OF4X18)
1320 incr inverter_cell(ST_INCLAO22OF4X2)
1321 incr inverter_cell(ST_INCLAO22OF4X4)
1322 incr inverter_cell(ST_INCLAO22OF4X7)
1323 incr inverter_cell(ST_INCLAO22OF4X9)
1324 incr inverter_cell(ST_INCLOA22OF4X13)
1325 incr inverter_cell(ST_INCLOA22OF4X18)
1326 incr inverter_cell(ST_INCLOA22OF4X2)
1327 incr inverter_cell(ST_INCLOA22OF4X4)
1328 incr inverter_cell(ST_INCLOA22OF4X7)
1329 incr inverter_cell(ST_INCLOA22OF4X9)
1330 incr inverter_cell(SY_INCL2W11OF2X13)
1331 incr inverter_cell(SY_INCL2W11OF2X18)
1332 incr inverter_cell(SY_INCL2W11OF2X2)
1333 incr inverter_cell(SY_INCL2W11OF2X4)
1334 incr inverter_cell(SY_INCL2W11OF2X7)
1335 incr inverter_cell(SY_INCL2W11OF2X9)
1336 incr inverter_cell(ST_INCLP1W1111OF4X13)
```

```
1337 incr inverter_cell(ST_INCLP1W1111OF4X18)
1338 incr inverter_cell(ST_INCLP1W1111OF4X2)
1339 incr inverter_cell(ST_INCLP1W1111OF4X4)
1340 incr inverter_cell(ST_INCLP1W1111OF4X7)
1341 incr inverter_cell(ST_INCLP1W1111OF4X9)
1342 incr inverter_cell(ST_INCLP1W111OF3X13)
1343 incr inverter_cell(ST_INCLP1W111OF3X18)
1344 incr inverter_cell(ST_INCLP1W111OF3X2)
1345 incr inverter_cell(ST_INCLP1W111OF3X4)
1346 incr inverter_cell(ST_INCLP1W111OF3X7)
1347 incr inverter_cell(ST_INCLP1W111OF3X9)
1348 incr inverter_cell(ST_INCLP1W11OF2X13)
1349 incr inverter_cell(ST_INCLP1W11OF2X18)
1350 incr inverter_cell(ST_INCLP1W11OF2X2)
1351 incr inverter_cell(ST_INCLP1W11OF2X27)
1352 incr inverter_cell(ST_INCLP1W11OF2X4)
1353 incr inverter_cell(ST_INCLP1W11OF2X9)
1354 incr inverter_cell(ST_INCLP2W1111OF4X13)
1355 incr inverter_cell(ST_INCLP2W1111OF4X2)
1356 incr inverter_cell(ST_INCLP2W1111OF4X4)
1357 incr inverter_cell(ST_INCLP2W1111OF4X7)
1358 incr inverter_cell(ST_INCLP2W1111OF4X9)
1359 incr inverter_cell(ST_INCLP2W111OF3X13)
1360 incr inverter_cell(ST_INCLP2W111OF3X18)
1361 incr inverter_cell(ST_INCLP2W111OF3X2)
1362 incr inverter_cell(ST_INCLP2W111OF3X22)
1363 incr inverter_cell(ST_INCLP2W111OF3X4)
1364 incr inverter_cell(ST_INCLP2W111OF3X7)
1365 incr inverter_cell(ST_INCLP2W111OF3X9)
1366 incr inverter_cell(ST_INCLP2W11OF2X13)
1367 incr inverter_cell(ST_INCLP2W11OF2X18)
1368 incr inverter_cell(ST_INCLP2W11OF2X22)
1369 incr inverter_cell(ST_INCLP2W11OF2X31)
1370 incr inverter_cell(ST_INCLP2W11OF2X4)
1371 incr inverter_cell(ST_INCLP2W11OF2X7)
1372 incr inverter_cell(ST_INCLP2W2111OF4X2)
1373 incr inverter_cell(ST_INCLP2W2111OF4X4)
1374 incr inverter_cell(ST_INCLP2W2111OF4X7)
1375 incr inverter_cell(ST_INCLP2W2111OF4X9)
1376 incr inverter_cell(ST_INCLP2W211OF3X18)
1377 incr inverter_cell(ST_INCLP2W211OF3X2)
1378 incr inverter_cell(ST_INCLP2W211OF3X22)
1379 incr inverter_cell(ST_INCLP2W211OF3X27)
1380 incr inverter_cell(ST_INCLP2W211OF3X4)
1381 incr inverter_cell(ST_INCLP2W211OF3X7)
1382 incr inverter_cell(ST_INCLP2W211OF3X9)
1383 incr inverter_cell(ST_INCLP2W2211OF4X13)
1384 incr inverter_cell(ST_INCLP2W2211OF4X18)
```

```
1385 incr inverter_cell(ST_INCLP2W2211OF4X2)
1386 incr inverter_cell(ST_INCLP2W2211OF4X4)
1387 incr inverter_cell(ST_INCLP2W2211OF4X7)
1388 incr inverter_cell(ST_INCLP2W2211OF4X9)
1389 incr inverter_cell(ST_INCLP3W1111OF4X2)
1390 incr inverter_cell(ST_INCLP3W1111OF4X4)
1391 incr inverter_cell(ST_INCLP3W1111OF4X9)
1392 incr inverter_cell(ST_INCLP3W111OF3X13)
1393 incr inverter_cell(ST_INCLP3W111OF3X18)
1394 incr inverter_cell(ST_INCLP3W111OF3X2)
1395 incr inverter_cell(ST_INCLP3W111OF3X31)
1396 incr inverter_cell(ST_INCLP3W111OF3X4)
1397 incr inverter_cell(ST_INCLP3W111OF3X7)
1398 incr inverter_cell(ST_INCLP3W111OF3X9)
1399 incr inverter_cell(ST_INCLP3W2111OF4X13)
1400 incr inverter_cell(ST_INCLP3W2111OF4X2)
1401 incr inverter_cell(ST_INCLP3W2111OF4X4)
1402 incr inverter_cell(ST_INCLP3W2111OF4X7)
1403 incr inverter_cell(ST_INCLP3W2111OF4X9)
1404 incr inverter_cell(ST_INCLP3W211OF3X13)
1405 incr inverter_cell(ST_INCLP3W211OF3X18)
1406 incr inverter_cell(ST_INCLP3W211OF3X2)
1407 incr inverter_cell(ST_INCLP3W211OF3X27)
1408 incr inverter_cell(ST_INCLP3W211OF3X4)
1409 incr inverter_cell(ST_INCLP3W211OF3X7)
1410 incr inverter_cell(ST_INCLP3W211OF3X9)
1411 incr inverter_cell(ST_INCLP3W2211OF4X13)
1412 incr inverter_cell(ST_INCLP3W2211OF4X2)
1413 incr inverter_cell(ST_INCLP3W2211OF4X4)
1414 incr inverter_cell(ST_INCLP3W2211OF4X7)
1415 incr inverter_cell(ST_INCLP3W3111OF4X13)
1416 incr inverter_cell(ST_INCLP3W3111OF4X18)
1417 incr inverter_cell(ST_INCLP3W3111OF4X2)
1418 incr inverter_cell(ST_INCLP3W3111OF4X7)
1419 incr inverter_cell(ST_INCLP3W3111OF4X9)
1420 incr inverter_cell(ST_INCLP3W3211OF4X13)
1421 incr inverter_cell(ST_INCLP3W3211OF4X18)
1422 incr inverter_cell(ST_INCLP3W3211OF4X2)
1423 incr inverter_cell(ST_INCLP3W3211OF4X4)
1424 incr inverter_cell(ST_INCLP3W3211OF4X7)
1425 incr inverter_cell(ST_INCLP3W3211OF4X9)
1426 incr inverter_cell(ST_INCLP4W1111OF4X2)
1427 incr inverter_cell(ST_INCLP4W1111OF4X4)
1428 incr inverter_cell(ST_INCLP4W1111OF4X7)
1429 incr inverter_cell(ST_INCLP4W1111OF4X9)
1430 incr inverter_cell(ST_INCLP4W2111OF4X13)
1431 incr inverter_cell(ST_INCLP4W2111OF4X2)
1432 incr inverter_cell(ST_INCLP4W2111OF4X4)
```

```
1433 incr inverter_cell(ST_INCLP4W2111OF4X9)
1434 incr inverter_cell(ST_INCLP4W2211OF4X13)
1435 incr inverter_cell(ST_INCLP4W2211OF4X4)
1436 incr inverter_cell(ST_INCLP4W2211OF4X7)
1437 incr inverter_cell(ST_INCLP4W2211OF4X9)
1438 incr inverter_cell(ST_INCLP4W3111OF4X2)
1439 incr inverter_cell(ST_INCLP4W3111OF4X4)
1440 incr inverter_cell(ST_INCLP4W3111OF4X7)
1441 incr inverter_cell(ST_INCLP4W3111OF4X9)
1442 incr inverter_cell(ST_INCLP4W3221OF4X13)
1443 incr inverter_cell(ST_INCLP4W3221OF4X2)
1444 incr inverter_cell(ST_INCLP4W3221OF4X4)
1445 incr inverter_cell(ST_INCLP4W3221OF4X7)
1446 incr inverter_cell(ST_INCLP4W3221OF4X9)
1447 incr inverter_cell(ST_INCLP5W2211OF4X13)
1448 incr inverter_cell(ST_INCLP5W2211OF4X2)
1449 incr inverter_cell(ST_INCLP5W2211OF4X4)
1450 incr inverter_cell(ST_INCLP5W2211OF4X7)
1451 incr inverter_cell(ST_INCLP5W2211OF4X9)
1452 incr inverter_cell(ST_INCLP5W2321OF4X2)
1453 incr inverter_cell(ST_INCLP5W2321OF4X7)
1454 incr inverter_cell(ST_INCLP5W3211OF4X2)
1455 incr inverter_cell(ST_INCLP5W3211OF4X4)
1456 incr inverter_cell(ST_INCLP5W3211OF4X7)
1457 incr inverter_cell(ST_INCLP5W3211OF4X9)
1458 incr inverter_cell(ST_INCLP5W3221OF4X13)
1459 incr inverter_cell(ST_INCLP5W3221OF4X2)
1460 incr inverter_cell(ST_INCLP5W3221OF4X4)
1461 incr inverter_cell(ST_INCLP5W3221OF4X7)
1462 incr inverter_cell(ST_INCLP5W3221OF4X9)
1463 incr inverter_cell(ST_INCLPAO22OF4X13)
1464 incr inverter_cell(ST_INCLPAO22OF4X2)
1465 incr inverter_cell(ST_INCLPAO22OF4X4)
1466 incr inverter_cell(ST_INCLPAO22OF4X7)
1467 incr inverter_cell(ST_INCLPAO22OF4X9)
1468 incr inverter_cell(ST_INCLPAO2O21OF4X4)
1469 incr inverter_cell(ST_INCLPAO2O21OF4X7)
1470 incr inverter_cell(ST_INCLPAO2O21OF4X9)
1471 incr inverter_cell(ST_INCLPOA22OF4X13)
1472 incr inverter_cell(ST_INCLPOA22OF4X2)
1473 incr inverter_cell(ST_INCLPOA22OF4X4)
1474 incr inverter_cell(ST_INCLPOA22OF4X7)
1475 incr inverter_cell(ST_INCLPOA22OF4X9)
1476 incr inverter_cell(SY_INCLP2W11OF2X13)
1477 incr inverter_cell(SY_INCLP2W11OF2X18)
1478 incr inverter_cell(SY_INCLP2W11OF2X2)
1479 incr inverter_cell(SY_INCLP2W11OF2X4)
1480 incr inverter_cell(SY_INCLP2W11OF2X7)
```

```
1481 incr inverter_cell(SY_INCLP2W11OF2X9)
1482 incr inverter_cell(HS65_GS_IVX106)
1483 incr inverter_cell(HS65_GS_IVX13)
1484 incr inverter_cell(HS65_GS_IVX142)
1485 incr inverter_cell(HS65_GS_IVX18)
1486 incr inverter_cell(HS65_GS_IVX2)
1487 incr inverter_cell(HS65_GS_IVX213)
1488 incr inverter_cell(HS65_GS_IVX22)
1489 incr inverter_cell(HS65_GS_IVX27)
1490 incr inverter_cell(HS65_GS_IVX284)
1491 incr inverter_cell(HS65_GS_IVX31)
1492 incr inverter_cell(HS65_GS_IVX35)
1493 incr inverter_cell(HS65_GS_IVX4)
1494 incr inverter_cell(HS65_GS_IVX40)
1495 incr inverter_cell(HS65_GS_IVX44)
1496 incr inverter_cell(HS65_GS_IVX49)
1497 incr inverter_cell(HS65_GS_IVX53)
1498 incr inverter_cell(HS65_GS_IVX62)
1499 incr inverter_cell(HS65_GS_IVX7)
1500 incr inverter_cell(HS65_GS_IVX71)
1501 incr inverter_cell(HS65_GS_IVX9)
1502
1503 proc is_inverting_output {pin} {
1504   global inverter_cell
1505   return [info exists inverter_cell([get_inst_cell_name [vdirname [get_db -if
         {.direction == out} $pin]]])]
1506 }
1507
1508 set_dont_use [vfind CORE65GPSVT -lib_cell *] true
1509 set_dont_use [vfind CORE65GPSVT -lib_cell *BFX*] false
1510 set_dont_use [vfind CORE65GPSVT -lib_cell *IVX*] false
1511 set_dont_use [array names inverter_cell] false
1512 set_dont_use [orphan_cell_list] true
```