Design Methodologies for

Low-Power Asynchronous-Logic Digital Systems

Law Chong Fatt

School of Electrical and Electronic Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of
Doctor of Philosophy

2008
Acknowledgements

I would like to thank my supervisors Associate Professor Joseph Chang and Associate Professor Gwee Bah Hwee for their guidance and encouragements. I would also like to thank my wife Annie for her patience and support throughout my doctoral studies.
# Contents

Acknowledgements ................................................................. ii
Author's Publications ............................................................ vi
Abstract ................................................................................... vii
List of Figures ........................................................................... ix
List of Tables ............................................................................ xiii
Abbreviations ........................................................................... xv

1. Introduction ................................................................. 1
   1.1. Motivation ................................................................. 1
   1.2. Objectives .................................................................. 12
   1.3. Contributions of the Thesis ............................................ 13
   1.4. Organization of the Thesis .............................................. 18

2. Modeling and Synthesis of Asynchronous Systems ........... 21
   2.1. Introduction ................................................................. 21
   2.2. Literature Review ........................................................ 22
      2.2.1. Syntax-Directed Translation ..................................... 22
      2.2.2. Logic Synthesis Approach ....................................... 23
      2.2.3. Desynchronization .................................................. 24
      2.2.4. Spatial Computation ................................................. 25
   2.3. Salient Features of Proposed Synthesis Method .............. 25
   2.4. Comparisons of Proposed Method with Reported Methods 29
   2.5. Design Modeling .......................................................... 32
      2.5.1. Compliance Issues with IEEE Standard .................... 32
      2.5.2. Initialization ............................................................ 33
      2.5.3. Asynchronous Communication Channels .................. 34
         2.5.3.1. Overview ........................................................ 34
         2.5.3.2. Push and Pull Channels ..................................... 36
         2.5.3.3. Data and Control Channels ............................... 38
         2.5.3.4. Guarded Channels ............................................. 40
         2.5.3.5. Channel Merging ............................................... 41
      2.5.4. Looping Statements ............................................... 41
      2.5.5. Storage Devices ..................................................... 42
         2.5.5.1. Level-Sensitive Storage Devices (Latches) .......... 42

Author’s Publications


Abstract

Asynchronous design has been an active area of research since the 1950s, but has hitherto yet to achieve widespread use or acceptance. This is largely because several major problems continue to persist that inhibit its acceptance in the very large-scale integration industry as a viable alternative to the prevalent synchronous design. This thesis addresses one such problem: how to reduce the circuit area and power dissipation of asynchronous control networks.

Three main contributions are made in this thesis to the design of asynchronous pipelines with low asynchronous control overheads. First, a synthesis method for asynchronous pipelines is proposed that adopts a coarse-grain approach to the synthesis of asynchronous control networks, thereby leading to low asynchronous control overhead pipelines. It also has the advantages of offering a largely transparent modeling style for asynchronous communication and ease for integration into the conventional synchronous design flow. The efficacy of the proposed synthesis method is demonstrated through the design of a Reed-Solomon error detector and an interpolated finite-impulse-response filterbank – the simulated circuit implementations based on the proposed synthesis method dissipate on the average 31% less energy than those implemented by comparable reported synthesis methods.

Second, two optimization methods are proposed for reducing the circuit area and power dissipation of asynchronous control networks while satisfying pipeline throughput constraints. The first proposed optimization method – handshake component fusion – is a form of peephole optimization that iteratively selects a pair of handshake components that share input channel sources or output channel destinations and replaces them with a single component. The second proposed optimization method – optimal decoupling – searches for the optimal mix of handshake components of different degree of concurrency in a given asynchronous control network with the objective of incurring the least circuit area and power dissipation for the control network, while satisfying a throughput constraint. The proposed optimization methods are applied on three designs and are found to reduce the asynchronous control networks’ transistor count and energy dissipation by, on the average, 28% and 36%, respectively. A fundamental requirement of the proposed optimization methods is that the minimal-support S-invariants for the Petri net models of the asynchronous control networks are recomputed at each optimization iteration.
Arising from the fundamental requirement mentioned above, the third main contribution of the thesis is a fast and memory-efficient algorithm for computing all minimal-support S-invariants for ordinary Petri nets. Based on a large number of test problems, the proposed algorithm is demonstrated to be at least 2.2x faster and 1.8x more memory efficient than reported algorithms.

The proposed synthesis method, optimization techniques, and invariant-computation algorithm are developed into computer-aided-design tools that integrate easily into the conventional synchronous design flow. This allows existing commercial simulation and logic synthesis tools to be leveraged for asynchronous design.
List of Figures

Fig. 1.1 Basic asynchronous pipeline structure based on delay matching................. 8
Fig. 1.2 Handshake protocols: (a) four-phase and (b) two-phase................................. 9
Fig. 1.3 The AND-gate chain for implementing asymmetric channel delays............ 11
Fig. 1.4 The integration of the proposed synthesis and optimization methods into the conventional design flow for synchronous circuits.......................... 15
Fig. 2.1 The request signal (req) and data travel (a) in the same direction for push channels, and (b) in the opposite direction for pull channels............. 36
Fig. 2.2 Example of push channel: Two-stage shift register........................................ 37
Fig. 2.3 The circuit inferred for the code in Example 2.10............................................. 45
Fig. 2.4 Constructing directed graph $D_1$: (a) assignments; (b) conditional statements; (c) case statements and case items; and (d) procedural timing control by named events................................................. 47
Fig. 2.5 The directed graphs constructed for the ALU model of Example 2.11: (a) $D_1$ and (b) $D_2$................................................................. 49
Fig. 2.6 The circuit inferred for the code in Example 2.2............................................. 51
Fig. 2.7 The Sync handshake component: (a) symbol, (b) STG specification, and (c) circuit implementation.......................................................... 52
Fig. 2.8 The C-gate: (a) symbol and (b) CMOS implementation................................. 53
Fig. 2.9 The four-phase normally-closed broad handshake protocol.......................... 54
Fig. 2.10 The SyncPassiveOut handshake component: (a) symbol and (b) STG specification................................................................. 56
Fig. 2.11 The SyncPassiveOutArb handshake component consists of a channel arbiter and a SyncPassiveOutCore.................................................. 57
Fig. 2.12 The channel arbiter [1][3]........................................................................ 57
Fig. 2.13 The mutual-exclusion element (CMOS implementation) [1][3].................. 57
Fig. 2.14 The SyncActiveIn handshake component: (a) symbol, (b) STG specification, and (c) circuit implementation................................................. 58
Fig. 2.15 The circuit inferred for the code in Example 2.4............................................. 59
Fig. 2.16 The SyncGuard handshake component: (a) symbol, (b) STG specification, and (c) circuit implementation

Fig. 2.17 The circuit inferred for the code in Example 2.7

Fig. 2.18 The ChMerge handshake component: (a) symbol, (b) STG specification, and (c) circuit implementation

Fig. 2.19 The circuit inferred for the code in Example 2.8

Fig. 2.20 The circuit inferred for the code in Example 2.9

Fig. 2.21 The LoopFront handshake component: (a) symbol and (b) STG specification

Fig. 2.22 The LoopEnd handshake component: (a) symbol and (b) STG specification

Fig. 2.23 Circuit implementations of: (a) LoopFront and (b) LoopEnd

Fig. 2.24 STG specification for Sync with active output channel upon initialization

Fig. 2.25 Feedback loop with different initial states: (a) X and Y have idle output channels (deadlocked), (b) X has active output channels (live), and (c) Y has active output channels (live)

Fig. 2.26 Modular PN models of: (a) Sync, (b) SyncPassiveOut, (c) SyncActiveIn, (d) SyncGuard, (e) ChMerge, and (f) LoopFront and LoopEnd combined

Fig. 2.27 (a) An asynchronous control network comprising two Sync handshake components and (b) the corresponding PN model

Fig. 2.28 Timing diagram analysis to ensure correct data transfers across pipeline stages

Fig. 2.29 Asynchronous pipeline with receiving pipeline stage implemented as: (a) latches and (b) flip-flops

Fig. 2.30 Block diagram of the Reed-Solomon error detector for the compact disc player

Fig. 2.31 The asynchronous control network synthesized by the proposed synthesis method for the Reed-Solomon error detector

Fig. 2.32 Waveforms obtained from HDL simulation of the compiled design for the syndrome computation phase
Fig. 2.33 Waveforms obtained from HDL simulation of the compiled design for the error detection phase

Fig. 3.1 STG specifications and circuit implementations of Sync: (a) minimally-concurrent, (b) semi-concurrent, and (c) maximally-concurrent

Fig. 3.2 A flow chart showing the execution of the proposed handshake component fusion method

Fig. 3.3 A simple example of handshake component fusion: (a) original control network; and (b) control network after the fusion of the components $S_2$ and $S_3$

Fig. 3.4 A simple example to illustrate the proposed optimization target selection algorithm

Fig. 3.6 (a) The PN model of a fork where $t_A$ and $t_B$ represent the optimization targets $A$ and $B$, respectively, for handshake component fusion and $t_L$ represents a channel link from $A$ to $B$. (b) The fusion of $A$ and $B$ forms a channel loop involving $t_{AB}$ and $t_L$

Fig. 3.7 16-input asynchronous pipelined parallel prefix trees for the addition operation: (a) without pipelining; and (b) with pipelining (asynchronous)

Fig. 3.8 The original control network of the asynchronous pipelined parallel prefix tree (delay-matching elements are not shown)

Fig. 3.9 The optimized control network (OPT_10%) of the asynchronous pipelined parallel prefix tree (delay-matching elements not shown) after handshake component fusion

Fig. 3.11 The control network of the four-bit asynchronous cross-pipelined array multiplier (delay-matching elements not shown)

Fig. 3.12 The optimized control network of the four-bit asynchronous cross-pipelined array multiplier after handshake component fusion (delay-matching elements are not shown)

Fig. 3.13 The optimized control network of the asynchronous Reed-Solomon error detector after handshake component fusion

Fig. 3.14 The PN models of three-stage linear pipelines: (a) minimally-concurrent, (b) maximally-concurrent, and (c) optimally-decoupled
Fig. 3.15  A branching operation in the branch-and-bound algorithm for the proposed optimal decoupling method ............................................. 148

Fig. 3.16  The optimally-decoupled control network of the 16-input asynchronous pipelined parallel prefix tree ....................................................... 153

Fig. 3.17  The optimally-decoupled control network of the four-bit asynchronous cross-pipelined array multiplier ................................................. 155

Fig. 4.1  An “exponential net” with $a$ transitions and $b$ places in the pre-set and post-set of each transition ................................................................. 167

Fig. 4.2  An example of parallel places: $p_1$ and $p_2$ are parallel to each other .... 174

Fig. 4.3  (a) The given net; (b) the creation of new places $mp_1$ and $mp_2$ during the annihilation of $t_2$ and $t_3$, respectively; and (c) the replacement of $mp_1$ and $mp_2$ with representative parallel place $pp_1$ ............................................. 175

Fig. 4.4  PN $N'$ corresponding to PN $N$ of Fig. 4.3(a) ............................................. 178

Fig. 4.5  Substructures of PN $G$ associated with (a) a representative parallel place and (b) a macro place ................................................................. 182

Fig. 4.6  PN $G$ representing the relationships between the places in Fig. 4.3 ...... 182

Fig. 4.7  Parallel enumeration: (a) PN dynamics and (b) state graph representation ................................................................................................................. 185

Fig. 4.8  Macro refinement: (a) PN dynamics and (b) state graph representation. 187

Fig. 4.9  The PN $G$ for the illustration of the second heuristic rule proposed to reduce the state space of $G$ ................................................................. 192

Fig. 4.10  The state spaces generated from $M_0$ for the PN $G$ of Fig. 4.9: (a) $pp_1$ is selected for parallel enumeration before $pp_2$; and (b) $pp_2$ is selected for parallel enumeration before $pp_1$ ............................................. 193

Fig. 4.11  The PN related to the computation of $r$ ..................................................... 198

Fig. 4.12  S-invariant computation example: transformation of PN $N$ during net transformation phase ........................................................................... 202

Fig. 4.13  S-invariant computation example: PN $G$ ............................................. 205

Fig. 4.14  S-invariant computation example: part of the state space of $G$ generated from the initial marking that corresponds to $mp_{15}$ ........ 206
List of Tables

Table 2.1 Comparisons of Reed-Solomon Error Detectors Realized Using Different Implementation Styles........................................... 99

Table 2.2 Comparisons of Synchronous and Asynchronous Reed-Solomon Error Detectors........................................................................ 102

Table 2.3 Comparisons of IFIR Filter Banks Realized Using Different Implementation Styles.......................................................... 104

Table 3.1 Handshake Components Fused During Optimization of Pipelined Parallel Prefix Tree’s Asynchronous Control Network........... 133

Table 3.2 Comparisons of Pipelined Parallel Prefix Tree Before and After Handshake Component Fusion................................................. 135

Table 3.3 Handshake Components Fused During Optimization of Cross-Pipelined Array Multiplier’s Asynchronous Control Network........ 140

Table 3.4 Comparisons of Four-Bit Asynchronous Cross-Pipelined Array Multiplier Before and After Handshake Component Fusion........ 142

Table 3.5 Comparisons of Reed-Solomon Error Detector Before and After Handshake Component Fusion................................................. 143

Table 3.6 Comparisons Of 16-Input Pipelined Parallel Prefix Trees Implemented Using Different Handshake Component Configurations........ 154

Table 3.7 Comparisons of 16-Input Pipelined Parallel Prefix Trees Implemented Using Different Handshake Component Configurations........ 156

Table 3.8 Comparisons of Reed-Solomon Error Detectors Implemented Using Different Handshake Component Configurations........... 157

Table 4.1 S-Invariant Computation Example: Lookup Table $R$ at End of Phase 1................................................................. 203

Table 4.2 S-Invariant Computation Example: Lookup Table $U$ at End of Phase 1................................................................. 203

Table 4.3 S-Invariant Computation Example: Invariance Matrix $D$ at End of Phase 1................................................................. 204

Table 4.4 Experimental Results For Proposed S-Invariant Computation Algorithm................................................................. 207

Table 4.5 Proportion Of Parallel Places ($R$) Created During Phase 1 of Proposed Algorithm................................................................. 208
Table 4.6 Comparisons Between Proposed, FM1, FM2, and D’Anna Algorithms
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACN</td>
<td>asymmetric choice net</td>
</tr>
<tr>
<td>CAD</td>
<td>computer-aided design</td>
</tr>
<tr>
<td>CSP</td>
<td>communicating sequential processes</td>
</tr>
<tr>
<td>HDL</td>
<td>hardware description language</td>
</tr>
<tr>
<td>PN</td>
<td>Petri net</td>
</tr>
<tr>
<td>RTL</td>
<td>register-transfer level</td>
</tr>
<tr>
<td>RTZ</td>
<td>return-to-zero</td>
</tr>
<tr>
<td>STG</td>
<td>signal transition graph</td>
</tr>
<tr>
<td>VLSI</td>
<td>very large scale integration</td>
</tr>
</tbody>
</table>
1

Introduction

1.1. Motivation

Asynchronous design has been an active area of research since the 1950s [1]. The following is a very brief history of asynchronous design [2][3].

The ILLIAC and ILLIAC II, two of the most powerful mainframe computers at the time built at the University of Illinois in 1952 and 1962, respectively, are the earliest examples of electronic systems that contained both synchronous and asynchronous parts [4]. The 1960 PDP-6 computer from Digital Equipment Corp. (DEC) was mostly asynchronous [5]. In the work on macromodular computer systems in the 1960s, asynchronous building blocks, such as registers, adders, memories, and control devices, were proposed that, in spirit, are very similar to a modern system-level approach to asynchronous design [6]. Important theoretical contributions to asynchronous design made in this period, particularly in the area of metastability, include the works of Huffman [7], Muller [8], and Unger [9]. In the 1980s, the first “modern” synthesis methods were developed for designing asynchronous controllers [10] and asynchronous
systems [11]. This was soon followed by works based on similar methodologies [12][13][14].

Practical very large-scale integration (VLSI) asynchronous design, particularly in the field of microprocessors, was demonstrated in 1988, when the world's first asynchronous microprocessor was developed at the California Institute of Technology [15]. This was followed by the first Amulet (a family of asynchronous processors compatible with the ARM processor) from the University of Manchester in 1993 [16], the TITAC, an 8-bit microprocessor from the Tokyo Institute of Technology in 1994 [17], the Amulet2e [18] and TITAC-2 [19] in 1997, the latter being a 32-bit processor, the programmable digital signal processor developed at Cogency in 1998 [20], the asynchronous 8051 microcontrollers developed by Philips Research Laboratories in 1998 [21] and by the California Institute of Technology in 2003 [22], and the ARM processor from Handshake Solutions in 2005 [23].

Compared with synchronous circuits, i.e., those that rely on a global clock to sequence operations, asynchronous circuits are advantageous in the following ways.

First, lower power dissipation. An asynchronous circuit only dissipates power when and where active. Any subcircuit is quiescent until it is activated. After the completion of its task, it returns to a quiescent, almost nondissipating state (except for the leakage power) until the next activation. This makes asynchronous design techniques particularly suitable for low-power circuits and systems that frequently enter the idle or standby mode either in parts or in entirety. Applications that involve algorithms with loops whose number of iterations is data dependent are also good candidates for asynchronous implementation. Excellent real-life examples of low-power asynchronous design include the Reed-Solomon error detector operating at audio rates [24], the 80C51 microcontroller [21], and the standby circuits for a low-power pager [25], all developed...
at the Philips Research Laboratories, the IFIR filter bank for digital hearing aids developed at the Technical University of Denmark in cooperation with Oticon Inc [26], and the programmable digital signal processor from Cogency [20]. All of these designs were reported to have significantly lower power dissipation compared with their synchronous counterparts, with power savings ranging from 50% to several times less.

Opponents of asynchronous design point out that a similar low-power strategy can be employed in synchronous circuits through clock gating, a technique that involves shutting down the clocks to subcircuits which are not operational at a particular time. However, in many cases, shutting down an entire synchronous system is not possible because clocks must still be continuously supplied to components that monitor the environment for the next call to action. This means that power will be consumed even during idle periods. Furthermore, clock gates exacerbate the global clock distribution problem and might increase the clock skew, which degrades the performance of the circuit.

Second, better performance. A synchronous circuit always exhibits worst-case performance because its clock frequency must be set to accommodate the slowest computation block under the worst-case conditions. On the other hand, computation blocks in an asynchronous circuit are allowed to operate at their highest possible speeds. Furthermore, in an asynchronous circuit, a computation block can start operating immediately after it has received all its inputs and, thus, there is no need to wait for a transition of the clock signal. This means that asynchronous circuits can, potentially, provide average-case performance instead of fixed (i.e., worst-case) performance, thus giving them a fundamental performance advantage over synchronous circuits. Circuits that benefit most from the average-case performance provided by asynchronous design techniques are those whose computation blocks have large variability in delays. A good
example of such circuits are arithmetic modules, such as adders [27][28][29]. Other examples where asynchronous design techniques have provided better performances include a software decompression engine for embedded processors [30], where the large variations in delays typical for Huffman decoders were exploited using asynchronous design techniques, instruction decoders for microprocessors [31], and microprogrammed control systems [32].

Third, lower noise and lower emission. In synchronous circuits, all clock drivers and storage devices switch simultaneously at each active edge of the global clock, which causes voltage glitches on the power supply lines or induces currents in the silicon substrate. This noise may affect the performance of an analog component (for example, an analog-to-digital converter) that is drawing power from the same source or integrated on the same substrate as the synchronous circuits. A synchronous circuit also emits electromagnetic radiation at its clock frequency (and the higher harmonic frequencies), which might be mistaken by a radio receiver in the same system as a radio signal.

Since activities in asynchronous circuits are not synchronized to a common clock, they tend to have lower noise and better electromagnetic compatibility properties than synchronous ones [18][33]. For this reason, asynchronous circuits are suitable for applications that require the use of sensitive analog and radio components, as well as those that contain electronic security circuits, such as encryption engines [34]. For example, Philips Semiconductors has developed a family of entirely asynchronous pager baseband controller integrated circuits, based on its asynchronous 8051 microcontroller [21].

Fourth, absence of the global clock skew problem. In synchronous circuits, large clock skews due to an unevenly distributed global clock might lead to a malfunctioning
of the circuit. Since asynchronous circuits do not have global clocks, they do not suffer from the global clock skew problem.

With the advent of the nano-scale era for VLSI systems, the advantages of asynchronous circuits are becoming increasingly valid and important. Take, for example, the global clock skew problem in synchronous circuits. It is now generally accepted that the parameter variations across a chip that contains an entire system will make it prohibitively expensive to synthesize a global clock tree that is evenly distributed [3]. As a result, such a chip is very unlikely to operate under the control of a single clock but will require asynchronous techniques.

Despite its advantages and long history of development, asynchronous design has remained largely an academic endeavor where the practitioners are mainly researchers and students in academic institutions. Although there are commercial designs that are based on asynchronous techniques (such as the ARM processor by Handshake Solutions [23], ethernet switches by Fulcrum Microsystems [35], and the DCC player error detector by Philips [24]), modern VLSI systems are overwhelmingly designed using synchronous techniques.

Several problems with asynchronous design have been suggested in the literature that account for the predominance of synchronous circuits over asynchronous ones. One such problem is that it has traditionally been much more difficult to resolve the problem of hazards (i.e., undesired signal transitions or glitches) in an asynchronous circuit than a synchronous one, prior to the recent development of synthesis tools for hazard-free asynchronous controllers.

In synchronous designs, the problem of hazards is, conceptually at least, easy to solve – glitches are allowed at all times except near the active edge of the clock, i.e., the
clock period is set to be long enough such that the signals are stable before they are latched into registers at the active edge of the clock.

In contrast, the lack of a global clock for synchronization in asynchronous designs means that an asynchronous controller will respond to input signal transitions at any time and will malfunction if there are glitches at its inputs. Thus, hazards must either be removed from the controller or not introduced into the design in the first place. Many reported methods in the past for asynchronous controllers, in particular asynchronous finite state machines, avoid hazards by imposing harsh restrictions on the range of behaviors that can be specified, such as allowing only single-input changes \[9\][36][37] (i.e., once an input changes, no new input change can occur until the system is stable; this is also referred to as the fundamental mode timing assumption \[9\]), or restricted multiple-input changes which impose timing constraints on inputs \[9\][38]. In other reported methods, hazards in asynchronous controllers are eliminated by adding delay elements to ensure that the outputs are generated only after the inputs are stable \[39\]. Unfortunately, this technique severely degrades the speed performances of the controllers. Many reported methods for asynchronous finite state machines also require special state encodings to handle state changes correctly \[38\][40][41][42]. These difficulties have made asynchronous circuits largely intractable for practical system design in the past.

In summary, synchronous techniques offered an easier way of dealing with timing issues and hiding hazards than asynchronous techniques (at least in the past), which has contributed to the former's widespread adoption in the industry.

However, substantial progress has been made in recent work in overcoming the difficulties of asynchronous controller design. In particular, the problem of hazards in asynchronous controllers has recently been addressed with reasonable success through
the development of computer-aided design (CAD) tools for the automated synthesis of asynchronous speed-independent circuits [43][44][45] and asynchronous burst-mode machines [46][47][48][49], both of which guarantee hazard-free implementations. Although these tools have greatly improved the attractiveness of asynchronous design, they were developed fairly recently in the 1990s, by when the synchronous design style had already been the dominant practice in the industry for several decades.

Another reason for the industry's lack of interest in the asynchronous design style in the past is that the most significant feature of asynchronous design – the absence of the global clock – has not been compelling enough for the industry to adopt the asynchronous design style. This is because the problem of global clock distribution in synchronous design, though challenging, has been manageable until recent times.

However, the design of global clock distribution networks has become significantly more challenging in recent years due to increasing die sizes and clock frequencies [50]. Intra-die processing variations have compounded the difficulties involved in the design of a clock distribution network. Although clock distribution and de-skewing methods are abundant (see, for example, [51]), they share the common characteristic of being expensive in either power or area, and they become prohibitively so as clock speeds increase. In a modern microprocessor design, the clock distribution network typically dissipates more than 20% of the total power and can occupy as much as 50% of the chip area [52]. In addition, clock skew can significantly deteriorate the performance of a high-speed integrated circuit because the minimum clock period is partly determined by the maximum clock skew in the circuit [53].

Thus, it is becoming increasingly clear that chips of the near future are unlikely to operate under the control of a single clock but will require asynchronous design techniques.
Yet another often cited reason for the industry's lack of interest in asynchronous design is the lack of commercially-available and technically well-supported CAD tools for asynchronous design. This does not imply that there are no CAD tools for asynchronous design. On the contrary, there exists a plethora of asynchronous design tools, some of which are distributed freely [54][55][56][43][57]. The problem is that these tools, with a few exceptions (such as [58][59]), are developed for research purposes and, therefore, are usually not well documented and little technical support, if any at all, is provided.

This thesis addresses another important problem with asynchronous design: reducing the circuit area and power dissipation of *asynchronous control networks* in asynchronous systems. Before this problem can be further elaborated, it is necessary to explain clearly the structure and purpose of control networks in asynchronous systems. This will now be reviewed in the following paragraphs.

Consider the basic structure of the *asynchronous pipeline* (or *micropipeline*) [60], as depicted in Fig. 1.1. Unlike in synchronous pipelines, where the operation of all the stages are synchronized to a common clock, each stage in the asynchronous pipeline operates at its own pace, using control information from its adjacent stages. More
specifically, the storage devices in each stage are controlled by an asynchronous controller, termed *handshake component*, which communicates with the asynchronous controllers of the adjacent stages to exchange control information.

Communication between handshake components is carried out using two control wires, request (*req*) and acknowledge (*ack*), which are commonly referred to as *handshake signals*, and is governed by a *handshake protocol*, which is a set of rules that defines the order of handshake signal transitions and their meanings. In asynchronous pipelines, two types of handshake protocols are common: *four-phase* and *two-phase*.

Fig. 1.2(a) shows the four-phase handshake protocol. In each data transfer operation, the sender puts valid data on the data wires and then produces a rising transition on *req* to indicate to the receiver that a new data is available. If the data that is held by the receiver has already been taken by its successor stage, then it consumes the new data and produces a rising transition on *ack* to indicate to the sender that the new data has been taken. Subsequently, the sender produces a falling transition on *req*, which is followed by the receiver producing a falling transition on *ack*. Note that the period during which the falling transitions on *req* and *ack* occur is commonly referred to as the *return-to-zero* (RTZ) phase. The two-phase handshake protocol, as depicted in Fig. 1.2(b), differs from its four-phase counterpart in that the handshake signals do not
return to their initial states after each data transfer. Thus, for the two-phase handshake protocol, only the transitions of the handshake signals are meaningful, not their levels.

To prevent the receiver from latching in invalid data, one has to ensure that the request signal does not reach the receiver before the data. This can be achieved by a technique termed delay matching, which involves delaying the request signal by placing on the request wire a delay-matching element. To ensure correct data transfer, the latency of the delay-matching element must match (or, in practice, be greater than) the critical path delay from the storage device-enable output port of the sender’s handshake component to the data input port of the receiver’s storage device. As shown in Fig. 1.1 where the critical path is depicted as a dotted arc, the critical path delay typically includes the delay across the storage device-enable signal buffers, the delay across the storage device of the sender, and the critical path delay across the combinational logic block between the sender and the receiver.

For the four-phase handshake protocol described above, two delay-matching schemes are possible: symmetric channel delays and asymmetric channel delays. The symmetric channel delay scheme uses inverter chains as delay-matching elements and, thus, imposes the same delay on the request signal regardless of whether the request signal is undergoing a rising or falling transition. On the other hand, the asymmetric channel delay scheme exploits the redundancy (in terms of data transfer) of the RTZ phase by reducing the RTZ-phase delay to the latency of a single gate. In this scheme, AND-gate chains are used as delay-matching elements (see Fig. 1.3).

The handshake components, handshake signals, and delay-matching elements are collectively referred to as the asynchronous control network of the asynchronous pipeline.
For datapath-dominated asynchronous pipelines, such as finite impulse response filters, the asynchronous control networks are likely to be relatively small. However, for asynchronous pipelines that are control dominated, where the data-processing parts play a relatively minor role, and those that are synthesized from high-level programming languages, there is strong evidence that the asynchronous control networks are relatively large [61][62][63][64][66][67]. This is due in part to the syntax-directed translation method that is used by many design compilers [54][55][56][58][59][68][69][70][71][72][73][74][14][76][78] that synthesize asynchronous circuits from high-level programs. The syntax-directed translation method is very efficient in terms of compilation time because it simply maps every language construct in a program to a corresponding handshake component and completely avoids logic synthesis. Such a direct approach, however, can potentially introduce significant control redundancies into the design. For example, it is reported in [67] that for an asynchronous Reed-Solomon error detector compiled using the syntax-directed translation method, the control network occupies 35% of the total circuit area.

In some asynchronous pipelines, the power dissipation overhead of the control network is higher than the circuit area overhead due to the higher signal switching factor in the control network relative to the datapaths. For the extreme case, each control signal in the control network always switches twice in each data transfer operation (this is to allow the control signal to return to its initial state), whereas the data signals in the single-rail datapaths have a switching factor of only 0.5. In such a scenario, the switching factor of the control signals is 4x higher than the data signals and one can
expect the power dissipation overhead of the asynchronous control network to be higher than the circuit area overhead. For example, the control network of the asynchronous pipelined processor reported in [61] occupies 18% of the total circuit area, but dissipates 35% of the total power.

Such large asynchronous control overheads are undesirable because they might offset the advantages that asynchronous circuits enjoy over synchronous ones, in particular the advantage of lower power dissipation.

Reducing the size of asynchronous control networks is also desirable from the perspective of enhancing the speed performance of asynchronous pipelines [34][61]. There are two reasons for this. First, by decreasing the complexities of individual handshake components in the control network, the internal delays within the handshake components can be reduced. This would potentially reduce the delay overheads that are imposed by the control network on the pipeline [61]. Second, by decreasing the number of components in the control network, the capacitive loads that are driven by the control signals will be reduced. This would allow the control signals to switch faster and, thus, improve the speed performance of the pipeline.

Based on the arguments presented above, it is clear that in order for the asynchronous design style to gain wider acceptance within the VLSI design industry, one of the problems with asynchronous design that must be addressed is the large control overheads in asynchronous circuits.

1.2. Objectives

The main objectives of the work described in this thesis are to research on methods that facilitate the design of asynchronous pipelines with low asynchronous control
overheads, and develop the methods into useful computer-aided-design (CAD) tools. More specifically, the aim of the work is threefold:

i) To research on and develop a synthesis method for asynchronous pipelines that avoids introducing redundancies that are attributed to language constructs into asynchronous control networks. The efficacy of this synthesis method is evaluated by comparing the circuit area and power dissipation of the asynchronous control networks that it synthesizes with that of the control networks synthesized by comparable reported methods.

ii) To research on and develop optimization methods that reduce the circuit area and power dissipation of asynchronous control networks. An important consideration that is taken when developing these optimization methods is that they should preserve the performance, or at least not violate the timing constraints, of the asynchronous pipelines.

iii) To develop the proposed synthesis and optimization methods into CAD tools that can be integrated easily into the conventional synchronous design flow. This is in view of the recent trend in asynchronous design research of designing asynchronous circuits using commercial CAD tools (that were originally developed for synchronous design) so as to leverage on the capabilities of and technical support provided for these tools [61][80][81][82][83][84][85][86].

1.3. Contributions of the Thesis

This thesis makes three main contributions to asynchronous design methods. They are summarized as follows.
First, a new synthesis method for asynchronous pipelines is proposed that has the following salient features:

i) **Low asynchronous control overheads.** The proposed synthesis method adopts a coarse-grain approach to the synthesis of asynchronous control networks to keep their circuit area and power dissipation low. This means that unlike syntax-directed translation, the proposed synthesis method does not translate every language construct in the program code (that describes the behavior of the asynchronous pipeline) into a corresponding asynchronous component. Instead, it reserves asynchronous control to the implementation of essential asynchronous operations, such as:
   
   a) the implementation of channel joins, channel forks, channel merges, and channel guards;
   b) the enabling and disabling of storage devices; and
   c) the implementation of functional loops.

ii) **Largely-transparent asynchronous communication modeling.** Standard hardware description languages (HDLs), such as Verilog HDL, offer a well-established way to describe digital circuits but lack constructs that can model asynchronous events in an abstract manner. The proposed synthesis method overcomes this deficiency in HDLs by imposing on conventional HDL constructs additional semantics that infer asynchronous communication and implicitly govern the flow of data. These additional semantics are extracted from the constructs during the parsing of the HDL code. This approach enables the modeling of different kinds of asynchronous operations, yet allows asynchronous communication to be largely transparent.
iii) *Easy integration into conventional synchronous design flow.* Fig. 1.4 shows how the proposed synthesis method fits into the conventional synchronous design flow. The proposed synthesis method accepts design specifications written in Verilog HDL and generates intermediate Verilog HDL models. When supplemented by a library that contains the behavioral models of handshake components, these models are suitable for functional simulation and register-transfer level (RTL) synthesis using existing HDL simulators and RTL synthesis tools.

![Diagram](image)

**Fig. 1.4.** The integration of the proposed synthesis and optimization methods into the conventional design flow for synchronous circuits.
Second, two new optimization methods for reducing the circuit area and power dissipation of asynchronous control networks are proposed. As shown in Fig. 1.4, the proposed optimization methods are applied to the Petri net (PN) [87] model of the asynchronous control network of the compiled design (PNs are a mathematical and graphical tool for modeling and studying information-processing systems that can be characterized as being asynchronous, concurrent, and/or distributed; see Chapter 2, Section 2.3.3.1 for a basic introduction to PNs). They are:

i) **Handshake component fusion.** An optimization method for asynchronous control networks, termed *handshake component fusion*, is proposed. Handshake component fusion is a peephole optimization technique that locally restructures the control network of an asynchronous pipeline to reduce the pipeline’s control overheads. In essence, a proposed heuristic algorithm is used to iteratively select two handshake components of the same type, called *optimization targets*, that share at least one input channel source (fork) or output channel destination (join). The optimization targets are then replaced by (or *fused* into) a single component of the same type provided that the replacement preserves the behavior and throughput performance of the asynchronous pipeline. A salient feature of the proposed optimization target selection algorithm is that it performs the selection on the basis that the fusion of the optimization targets is least likely, amongst other possible targets, to degrade the throughput performance of the pipeline.

ii) **Optimal decoupling.** Optimal decoupling is a proposed optimization method that resolves the dilemma between using small handshake components to reduce asynchronous control overheads and satisfying timing constraints for pipeline throughput. Conventionally, asynchronous control networks are
designed using handshake components of the same degree of concurrency. By abandoning the conventional approach, the designer gains the freedom to use handshake components of different degree of concurrency in an asynchronous control network. The proposed method exploits this freedom through a branch-and-bound algorithm that searches for the optimal mix of handshake components of different degree of concurrency, i.e., one that occupies the smallest circuit area and dissipates the least power while meeting the pipeline throughput timing constraint. The main idea behind the proposed method is to use as many low concurrency handshake components as possible, so as to reduce the circuit area and power dissipation overheads incurred by the asynchronous control network, and to attempt to satisfy the throughput specification of the asynchronous system by selecting handshake components of higher concurrency where necessary.

Third, a fast and memory-efficient algorithm for the computation of all minimal-support S-invariants of PNs is proposed (see Chapter 2, Section 2.3.3.1 for an introduction to the concepts of PN S-invariants). As it shall be explained in Chapter 3, a fundamental requirement of the optimization methods for asynchronous control networks proposed in Chapter 3 is that the minimal-support S-invariants for the PN models of the control networks are recomputed at each optimization iteration. This implies that the feasibility of the proposed optimization methods depend critically, in terms of time and memory requirements, on the method that is employed to compute the minimal-support S-invariants. Unfortunately, although many algorithms for S-invariant computation have been reported, they tend to have long execution time and large memory requirement. Thus, there is a strong motivation for developing an S-invariant computation algorithm that is faster and more memory efficient than existing ones.
It is of significance to note that although the work on the proposed S-invariant computation algorithm has been motivated by the need to improve the feasibility of the proposed optimization methods, the proposed algorithm is independent of the optimization methods. More specifically, the proposed algorithm provides an efficient solution to the general problem of finding all minimal-support S-invariants of PNs.

1.4. Organization of the Thesis

This thesis is organized as follows. Chapter 1 describes the motivation behind and the objectives of the work described in this thesis. It also summarizes the main contributions reported in this thesis.

Chapter 2 describes the proposed synthesis method for asynchronous pipelines. To describe the formulation of the synthesis problem, a review of existing compilers for asynchronous systems and comparisons between the existing compilers and the proposed synthesis method are first given.

This is followed by a description of the coding style and modeling rules that are supported by the proposed synthesis method for the description of various types of asynchronous communication channels commonly found in asynchronous pipelines. An detailed explanation is then given of the synthesis process, which consists of three main tasks: the extraction of asynchronous communication channels, the inference of handshake components, and the computation, when necessary, of an initial state for the asynchronous control network. The proposed initial state computation method guarantees deadlock-free operation in the control network and a formal proof that the proposed method preserves the nondeadlock behavior of the control network is given.

The efficacy of the proposed synthesis method is demonstrated through the design of a Reed-Solomon error detector and an interpolated finite-impulse-response (IFIR)
filterbank. Comparisons, in terms of transistor count, energy dissipation, and throughput performance, are made between the designs implemented by the proposed synthesis method and comparable reported synthesis methods.

Chapter 3 describes the two proposed optimization methods (optimal decoupling and handshake component fusion) for asynchronous control networks. To describe the formulation of the optimization problem, the chapter starts by reviewing the existing optimization methods for asynchronous circuits and comparing the existing methods with the proposed methods.

For the proposed optimization method of handshake component fusion, a fork structure is used as a simple example to demonstrate the main ideas of the method. This is followed by a description of the proposed heuristic algorithm for optimization target selection. Finally, the issues of satisfying the throughput constraint and preserving the behavior of the asynchronous control network under optimization are discussed.

For the proposed optimization method of optimal decoupling, a simple example consisting of a three-stage linear asynchronous pipeline is used to illustrate the main ideas of the method. This is followed by a description of the proposed branch-and-bound algorithm that searches for the optimum mix of handshake components of different degree of concurrency that satisfies a throughput constraint.

The effectiveness of the proposed optimization methods are demonstrated by applying them on the asynchronous control networks of three designs: a 16-input pipelined parallel prefix tree, a four-bit cross-pipelined array multiplier, and a Reed-Solomon error detector.

Chapter 4 describes the proposed algorithm for minimal-support S-invariant computation of PNs. To describe the formulation of the S-invariant computation problem, reported algorithms for S-invariant computation are first introduced and their
problems relating to execution time and memory requirement are discussed. This is followed by a detailed description of the proposed algorithm, including formal proofs of its correctness and an analysis on its time complexity. An application example is then provided to illustrate the main ideas behind the algorithm. Finally, the efficacy of the proposed S-invariant computation algorithm is demonstrated by means of experimental results on the execution time and memory requirement of the proposed algorithm and comparisons against reported algorithms.

Chapter 5 concludes the thesis and recommends some directions for further research.
2
Modeling and Synthesis of Asynchronous Systems

2.1. Introduction

This chapter describes the proposed synthesis method for asynchronous pipelines and is organized as follows.

As a preamble to the formulation of the synthesis problem, a review of existing compilers for asynchronous systems and comparisons between the existing compilers and the proposed synthesis method are first given.

This is followed by a description of the coding style and modeling rules that are supported by the proposed synthesis method for the description of various types of asynchronous communication channels commonly found in asynchronous pipelines. A detailed explanation is then given of the synthesis process, which consists of three main tasks: the extraction of asynchronous communication channels, the inference of handshake components, and the computation, when necessary, of an initial state for the asynchronous control network. The proposed initial state computation method guarantees deadlock-free operation in the control network and a formal proof that the proposed method preserves the nondeadlock behavior of the control network is given.

Finally, the efficacy of the proposed synthesis method is demonstrated through the design of a Reed-Solomon error detector and an interpolated finite-impulse-response
(IFIR) filterbank. Comparisons, in terms of transistor count, energy dissipation, and throughput performance, are made between the designs implemented by the proposed synthesis method and comparable reported synthesis methods.

2.2. Literature Review

Synthesis methods for asynchronous systems can be broadly classified into four categories: syntax-directed translation, logic synthesis, desynchronization, and spatial computation. These methods will be reviewed in turn.

2.2.1. Syntax-Directed Translation

Many synthesis methods for asynchronous systems are based on translating high-level concurrent programs into hardware. Typically, the designer uses a high-level concurrent language that is based on the concepts of communicating sequential processes (CSP) [88] to describe the behavior of the hardware. CSP is a general-purpose programming language whose key characteristics include the support for concurrent processes, sequential and concurrent composition of statements within a process, and synchronized message passing over channels. A synthesis process, termed syntax-directed translation, is then utilized to convert the program into hardware, with each language construct mapped to a corresponding handshake component.

CSP and syntax-directed translation form the basis of many asynchronous synthesis systems: Haste/TiDE [58][59], Balsa [54][55][56], Rainbow [69][70], Tangram [72][73], an OCCAM-based design system [14][74][75], SHILPA [76][77], and Communicating Hardware Processes [78]. Asynchronous synthesis systems that rely on syntax-directed translation but are based on languages that are not related to CSP, such as C [68] and VHDL [71], have also been reported.
Using high-level programming languages for VLSI design has the advantage of masking the actual circuit implementation complexity and abstracting away the details of handshake protocols and data computation. This allows a designer to concentrate on the algorithmic and architectural aspects of the design and even enables designers without in-depth knowledge of VLSI circuits and IC technology to approach the design of complex VLSI systems as a programming task. Furthermore, syntax-directed translation is an inherently transparent process, in which the resulting hardware has a one-to-one correspondence with its source program, thereby allowing designs that are correct by construction.

However, an important consequence of this one-to-one mapping of language constructs to handshake components is that the designer needs to be aware of the cost of language constructs and even about the details of the compilation steps from programs to netlists. Otherwise, significant redundancies might be introduced into the asynchronous control networks [59]. Even with careful programming, there is substantial evidence indicating that asynchronous systems, particularly the control-dominated ones, that are translated from high-level programs tend to have large asynchronous control overheads [61][63][66][67].

2.2.2. Logic Synthesis Approach

Another way of synthesizing asynchronous systems is the asynchronous logic synthesis approach [64][65][89][90]. In this approach, a high-level description of the design is partitioned into its control and datapath components. Typically, the datapath component is synthesized using conventional register transfer-level (RTL) synthesis tools. On the other hand, the control component is translated into a graph-based specification, such as a signal transition graph (STG) [10], or a state diagram, such as a burst-mode
specification [48], which is then synthesized using low-level asynchronous logic synthesis tools [43][57]. The logic synthesis approach allows logic and timing optimization at the global level, a feature that is lacking in synthesis methods based on syntax-directed translation.

However, for nontrivial asynchronous systems, the logic synthesis approach can lead to large and complex STG or burst-mode specifications for the control component. This is undesirable because it can be difficult and, thus, time consuming to synthesize large specifications into corresponding hardware. In particular, the synthesis of an STG into a circuit requires enumerating the STG’s entire state space (the state space explosion problem) and uniquely encoding every reachable state of the controller (the complete state coding problem [91]). Thus, for large STGs, the synthesis time can be prohibitively long [64][89][92]. While a work-around to this problem is to partition a complex circuit into several parts and synthesize them separately, no tools are currently available to perform the partitioning automatically.

2.2.3. Desynchronization

Yet another method of synthesizing asynchronous systems is to convert synchronous design specifications into asynchronous circuits [80][81][83][84][86][93], commonly referred to as desynchronization. The basic idea of desynchronization is to start from a synchronous circuit and replace the globally-clocked control of the data registers with a set of communicating handshake components. This approach greatly simplifies the design of asynchronous circuits because it allows the reuse of existing synchronous specifications, can be supported by existing CAD tools for synchronous design, and does not require the designer to be knowledgeable in asynchronous design.
However, a desynchronized circuit contains only one type of asynchronous communication channel: push channels. A push channel always contains a datapath and, thus, is also a data channel. Data transfers from a sender to a receiver across a push channel are always initiated by the sender. Since desynchronized circuits contain only push channels, they are not able to support certain asynchronous design techniques that are useful for reducing power dissipation. For example, conditional communication, which ensures that only useful channel communications are propagated down pipelines, thereby preventing power wastage, cannot be implemented in a desynchronized circuit.

2.2.4. Spatial Computation

Spatial computation [94] [95] is yet another approach for compiling high-level programs into asynchronous circuits. It involves translating a C program into a dataflow intermediate representation, which is then synthesized into a hardware dataflow machine that directly executes the input program.

Spatial computation suffers from two drawbacks. First, in spatial computation, every logical and arithmetic operation in the input program is implemented as a pipeline stage. This can lead to unnecessarily large asynchronous control overheads in the design implementation. Second, similar to the desynchronization approach, spatial computation supports only push channels.

2.3. Salient Features of Proposed Synthesis Method

In this chapter, a synthesis method for asynchronous pipelines is proposed that has several salient features which address the problems and limitations of the reported synthesis methods reviewed in Section 2.2.
First, the proposed synthesis method facilitates the design of asynchronous systems with low asynchronous control overheads. It does so by adopting a coarse-grain approach to the synthesis of asynchronous control networks. This means that unlike syntax-directed translation, the proposed synthesis method does not translate every language construct in the program code (that describes the behavior of the asynchronous system) into a corresponding asynchronous component. Instead, it reserves asynchronous control to the implementation of essential asynchronous operations, such as:

i) the implementation of channel joins, channel forks, channel merging, and channel guards (conditional communication);

ii) the enabling and disabling of storage devices; and

iii) the implementation of functional loops.

Second, the proposed synthesis method offers a largely-transparent modeling style for asynchronous communication. The proposed coarse-grain approach to design compilation requires designs to be specified such that it is possible to distinguish between the language constructs that model combinational logic, which do not require asynchronous control (data multiplexing is an exception), and the language constructs that model sequential logic and asynchronous events, which do. A convenient way of modeling combinational logic is to specify designs at the RTL using standard HDLs, such as Verilog HDL [96]. The use of standard HDLs also brings forth other benefits, such as their wide-spread usage within the VLSI design industry and allowing semantic continuity between all levels of abstraction, from the first behavioral-level specification to the final gate-level netlist. One of the implications of this semantic continuity is the possibility of using the same test program throughout the entire design flow.
However, standard HDLs lack constructs that can model asynchronous events in an abstract manner. While it is possible for asynchronous communication signals to be described explicitly in HDLs, such a low-level modeling approach is unlikely to be acceptable to designers since it would represent a tremendous overhead in design time and effort when compared to the design style of synchronous circuits. Solutions reported in the literature to this problem all use specially-developed packages and subroutines that provide abstract models for asynchronous communication [82][85][90][97][98], such as subroutines for sending and receiving data. Although these abstract models ease to a certain extent the difficulties in describing asynchronous activities using HDLs, the resulting design style still leaves much to be desired because the subroutines must be called in the code whenever and wherever asynchronous communication is required.

In contrast, the proposed synthesis method overcomes the deficiency of HDLs in modeling asynchronous communication by imposing on conventional HDL constructs additional semantics that infer asynchronous communication and implicitly govern the flow of data. These additional semantics are extracted from the constructs during the parsing of the design specification. This approach enables the designer to model different kinds of asynchronous operations and, therefore, exploit the full advantages of the asynchronous design style, yet allows asynchronous communication to be largely transparent to the designer.

Third, the proposed synthesis method integrates easily into the conventional design flow for synchronous circuits. As shown in Fig. 1.4, which illustrates the proposed design flow for asynchronous circuits that is based on the conventional design flow for synchronous circuits, the proposed synthesis method performs the compilation task, which is the step after the creation of the Verilog HDL model and before logic
synthesis. At the compilation step, the compiler parses the design specification (written in Verilog HDL, well-known by all VLSI designers), synthesizes the asynchronous control network of the design, and establishes the control of the synthesized network on the datapath components, such as storage devices and multiplexers. The compiled design is written as an RTL model in Verilog HDL. When supplemented by a library that contains the behavioral models of the handshake components in the asynchronous control network, this model is suitable for functional simulation using any HDL simulator of choice.

It is important to note that the compiled design (written at the RTL) is a semi-structural model in that it contains instantiations of handshake components that are explicitly interconnected but largely preserves the original datapath specification. Modifications to the datapath specification are made only when necessary, such as the conversion of behavioral-level constructs into RTL ones and the control of datapath components by handshake components. This is significant because it allows designers to leverage state-of-the-art logic synthesis tools for datapath logic optimization at the RTL, which is the level at which the most improvement can be obtained during logic optimization.

In contrast, the reported synthesis methods reviewed in Section 2.2, with the exception of the desynchronization approach, rely on programming languages and CAD tools (for functional simulation and logic synthesis) that are unfamiliar to most VLSI designers [54][56][78][73][14]. In many cases, commercial tools (originally developed for synchronous design) are relegated to the back-end tasks, such as place and route. This not only means that significant relearning effort is required on the part of the (synchronous) designers, but also that the capabilities of, and technical support provided
for, commercial tools cannot be leveraged. A case in point is the logic synthesis of
datapath components for which state-of-the-art commercial tools exist.

2.4. Comparisons of Proposed Method with Reported Methods

The proposed synthesis method is now compared with the reported methods. The
greatest advantage that the proposed method has over the reported methods based on
CSP and syntax-directed translation is its coarse-grain approach towards the synthesis
of asynchronous control networks, which allows asynchronous control overheads to be
kept relatively low.

In particular, sequencers, a fundamental component in circuits generated by CSP-
based methods, are not required by the proposed method. Sequencers are necessary in
CSP-based methods to ensure that sequential operations are executed in the order
specified in the input program. In the proposed method, sequencers are not required
because sequential statements (called blocking statements in Verilog HDL) are
restricted to the following two areas. First, the initialization of variables upon system
reset and the initialization of loop variables prior to the execution of loops. Since order
has no relevance to the initialization of variables, it is clear that sequencers are not
necessary here. Second, the modeling of combinational logic. Sequencers are not
necessary here because combinational logic specifications are synthesized into a
monolithic logic block (as is the case in synchronous RTL synthesis) that is not directly
under any control from the asynchronous control circuitry.

The main disadvantage of the proposed synthesis method compared with the CSP-
based methods is its relatively limited support for behavioral constructs due to its
compliance with the synthesizable subset [99] of Verilog HDL. For example, probe
expressions in Haste [58][59], which are used to provide information on the status of
channels, are not supported by the proposed method. In addition, the proposed method supports only procedure calls (such as task and function calls in Verilog HDL) that model combinational logic because most RTL synthesis tools support only such procedure calls [100].

Compared with the desynchronization methods, the proposed method offers the advantage of support for various asynchronous communication channels commonly found in asynchronous circuits, other than push channels. These communication channels include pull channels, guarded channels (or conditional communication), and control channels (see Section 2.5.3). The proposed method also supports behavioral loops and arbitrated read, write, and read-write access to variables, features that are lacking in desynchronized circuits.

In particular, the support for various asynchronous communication channels allows the designer to prevent unnecessary handshakes (during which no useful data transfers are carried out) in asynchronous control networks by exploiting the characteristics in the algorithm and architecture of the application in question. For asynchronous pipelines that are control dominated and have narrow datapaths, the blocking of unnecessary handshakes can potentially be important in reducing power dissipation.

Two examples where unnecessary handshakes in asynchronous control networks can be prevented are now discussed. The first example involves conditional communication. Conditional communication ensures that only useful channel communications are propagated down pipelines, thereby preventing power wastage. It is similar to clock gating in synchronous circuits, which can be automatically supported by desynchronization [79]. However, clock gating in desynchronized circuits prevents only the propagation of useless data, not the corresponding handshakes. This is because handshakes in desynchronized circuits are performed by basic latch controllers that
always activate their output channels whenever an input handshake has occurred. On the other hand, conditional communication in asynchronous circuits prevent the propagation of useless data and handshakes. It requires latch controllers that decide whether to activate their output channels or not after input handshakes have occurred by sampling signals in datapaths. Such latch controllers are lacking in desynchronized circuits.

The second example involves pull channels. In a desynchronized circuit, a read operation on a memory variable has to be initiated by the variable itself and, thus, must be preceded by a write operation on the variable, even if the value stored in the variable is not being changed. With pull channels, read operations on memory variables are initiated by relevant receivers and, thus, write channels to the variables are not exercised.

The disadvantages of the proposed synthesis method compared with the desynchronization approach are as follows. First, the proposed method does not accept existing synchronous RTL specifications. New codes that comply with the proposed modeling rules must be written. Second, designers have to acquire an understanding of the asynchronous communication channels supported by the proposed approach. Third, the proposed method tends to lead to circuits that are slower than desynchronized circuits because it requires handshake components that are more complex than basic latch controllers.

Compared with spatial computation, the proposed method is advantageous in two key aspects. First, as mentioned earlier, spatial computation implements every logical and arithmetic operation in the input program as a pipeline stage, which can lead to unnecessarily large asynchronous control overheads in the design implementation. In contrast, the proposed approach gives the designer full control over where the pipeline stages are and, thus, can potentially lead to more efficient asynchronous control
networks. This flexibility in specifying pipeline stages also allows the modeling of combinational logic blocks that execute more than one operation. This is advantageous because the designer can then leverage commercial logic synthesis tools to optimize the logic blocks.

Second, spatial computation supports only push channels. On the other hand, the proposed approach supports a variety of communication channels commonly found in asynchronous pipelines. As explained earlier, this is useful for preventing unnecessary handshakes in the asynchronous control networks.

However, spatial computation offers a wealth of optimizations that operate on its dataflow intermediate representations, including scalar, memory, and Boolean optimizations. These optimizations are lacking in the proposed synthesis method.

2.5. Design Modeling

This section describes the proposed modeling rules with respect to Verilog HDL that are supported by the proposed synthesis method.

2.5.1. Compliance Issues with IEEE Standard

RTL synthesis is heavily used in the VLSI design industry and has gained such wide acceptance that RTL synthesis based on Verilog HDL has been formalized as an IEEE Standard [99]. The Standard defines a set of modeling rules for writing Verilog HDL descriptions suitable for RTL synthesis.

The design (in Verilog HDL) that is generated by the proposed synthesis method complies fully with the Standard and, thus, can be synthesized using any existing RTL synthesis tool that also complies with the Standard.
The design that is submitted for compilation to the proposed synthesis method must comply with the Standard, but with several exceptions. Some of these exceptions are related to certain data types and behavioral constructs which are not suitable for RTL synthesis, but are supported by the proposed synthesis method because they can be easily implemented using asynchronous design techniques. They are:

i) Named events (a Verilog HDL data type). By associating a named event with an input port or a variable, its triggering is used to represent activations of the port’s channel or the variable’s output channel.

ii) The event trigger $\rightarrow$, which is used to trigger named events.

iii) The while construct, which is useful for describing iterative computations whose number of iterations cannot be determined at the onset of the computation.

The proposed synthesis method does not support edge descriptors (posedge and negedge) because they are intended for the modeling of edge-sensitive sequential logic. This does not, however, exclude the use of edge-sensitive storage devices from designs that are compiled by the proposed synthesis method because edge-sensitive (as well as level-sensitive) storage devices are inferred from the design specification by the proposed synthesis method based on certain modeling rules (see Section 2.5.5).

2.5.2. Initialization

Variables in the design specification are initialized using the initial construct. For example, the following code specifies that variable x is initialized to zero.

Example 2.1:

```
initial
    x = 0;
```
For the variable initialization to be compiled correctly, exactly one input port in the design specification must be designated as the system reset signal using the `async_set_reset` attribute (in Verilog HDL, an attribute is a property about an object or a statement in a code; the attribute controls the operation of tools that act on the code [96]), as recommended by [96]. This reset signal is used to initialize not just the variables, but also the handshake components in the asynchronous control network.

### 2.5.3. Asynchronous Communication Channels

This subsection describes the modeling rules for different types of asynchronous communication channels.

#### 2.5.3.1. Overview

In this work, the modeling of asynchronous communication channels is largely transparent to the designer. No explicit handshake signals or abstract channel communication models (in the form of packages or subroutines) are used in the design specification. Instead, additional semantics are imposed on standard Verilog HDL constructs to infer asynchronous communication and implicitly govern the flow of data.

For example, the following code models a one-place buffer.

Example 2.2:

```verilog
always
  y <= x;
```

Based on the notion that data transfers occur on channels, the above code implicitly infers an asynchronous communication channel from variable `x` to variable `y`, inferred by the nonblocking procedural assignment `y <= x`. During compilation, this implicit channel is elaborated into explicit handshake signals which, together with the inferred
handshake component (in this example, a basic latch controller is inferred to control y),
allow the compiled model to be simulated and synthesized.

Note that there is a fundamental difference between our interpretation of the code
and its original semantics. Originally, the code simply describes a wire connecting x to
y. Furthermore, due to the repetitive nature of the always construct, the lack of any
timing controls in it prevents the simulation time to advance, causing a simulation
deadlock. In contrast, the code, as interpreted by the proposed synthesis method, has
additional semantics imposed on it to support channel communication between x and y,
and to govern the flow of data from x to y. Specifically, the assignment y <= x is
interpreted such that its execution is delayed until: i) x has made a request to transfer its
data to y; and ii) the data held by y has been taken by its successor.

In the example above, channel communication is transparent to the designer. The
proposed synthesis method also supports a more explicit way of modeling channel
communication. Specifically, a named event (a Verilog HDL data type) can be
associated with a variable such that communication events on the variable’s output
channel implicitly trigger the named event. The triggering of the named event is implicit
because it is merely an interpretation of the code by the compiler and the event trigger
--> is not used. Such an association is useful when the communication between the
sender and the receiver involves only handshaking and no data transfer.

Example 2.3:

(* async, channel = "x" *) event x_event;
always
  x <= w;
always @ x_event
  z <= y;

In the example above, named event x_event is associated with the output channel
of variable x using an attribute instance that is headed by the proposed keyword async,
and that assigns the value “x” to the proposed channel attribute. The code specifies that \texttt{x_event} is triggered by communication events on the output channel of \texttt{x} and that the triggering of \texttt{x_event} initiates the transfer of data from variable \texttt{y} to variable \texttt{z} (provided that \texttt{y} has data to transfer).

### 2.5.3.2. Push and Pull Channels

The channels modeled in the examples thus far are all push channels. A \textit{push channel} is one where data transfers are initiated by the sender, i.e., the request signal (\texttt{req}) and data flow in the same direction (from the sender to the receiver), as depicted in Fig. 2.1(a).

Put differently, the push channel is an \textit{active} output channel for the sender and a \textit{passive} input channel for the receiver. Unless they have been specified otherwise, channels modeled in the design are assumed, by default, to be push channels by the proposed synthesis method.
A two-stage shift register as shown in Fig. 2.2 is used as an example to explain more clearly the concept of push channels in asynchronous systems. Note that Stage 1 of the shift register is the *sender* and Stage 2 the *receiver*. The channel between Stage 1 and Stage 2 is a push channel. This means that it is Stage 1, the sender, that is initiating the data transfers between the two stages. For this reason, the request signal (*req*) in the channel is directed from Stage 1 to Stage 2, whereas the acknowledge signal (*ack*) in the channel is directed from Stage 2 to Stage 1. In each data transfer operation, Stage 1 puts valid data on the data wires and then produces a rising transition on *req* to indicate to Stage 2 that a new data is available. If the data that is held by Stage 2 has already been taken by its successor stage, then it consumes the new data and produces a rising transition on *ack* to indicate to Stage 1 that the new data has been taken. Subsequently, Stage 1 produces a falling transition on *req*, which is followed by Stage 2 producing a falling transition on *ack*. The form of communication between Stage 1 and Stage 2 described above is called the four-phase handshake protocol, as illustrated in Fig. 1.2(a).

A *pull channel* is different from a push channel in that its data transfers are initiated by the receiver and, therefore, the request signal flows from the receiver to the sender. Consequently, the pull channel is a passive output channel for the sender and an active input channel for the receiver. The proposed *pull* attribute is used to model pull
channels and can be attributed to an input port or a variable (for variables, the attribute applies only to their output channels).

For example, the following code infers a pull channel whose data direction is from variable $y$ to variable $z$. Note that the pull channel is activated by the triggering of named event $x$.

Example 2.4:

```vhdl
(* async, pull *) reg y;
always @(x)
  z <= y;
```

### 2.5.3.3. Data and Control Channels

A **data channel** is for transferring data from one pipeline stage to another and, therefore, always has a corresponding datapath. All channels discussed thus far in this section are data channels.

In contrast, a **control channel** consists of only handshake signals and no datapath. In this work, control channels are modeled using named events, the event trigger $\rightarrow$, and the event-control symbol $\@$.

More specifically, the activation of a control channel $x$ (a channel is said to be *active* when handshake is occurring on it) is modeled by the activation of an event-triggering statement, such as $\rightarrow \text{event}_x$, where $\text{event}_x$ is the named event that represents $x$. The effect of the activation of channel $x$ is modeled by an event-controlled statement, such as $@ (\text{event}_x) \ z \ <= \ y$, in which the assignment $\ z \ <= \ y$ is delayed until $\text{event}_x$ is triggered.

For example, the following code models a control channel $x$ identified by named event $\text{event}_x$ (note that $x$ does not explicitly appear in the code):

```vhdl
38
```
Example 2.5:

```vhdl
event eventx;
always
  if (expression) -> eventx;
always @(eventx)
  z <= y;
```

The if statement specifies that `eventx` is triggered when `expression` is true. Once `eventx` is triggered, the assignment "`z <= y`" is executed.

Named events can also be used to model communication events on data channels. More specifically, a named event can be associated with an input port, an output port, or a reg variable such that communication events on the port’s channel or the variable’s output channel trigger the named event. Unlike for control channels, the triggering of the named event associated with a data channel is implicit because it is merely an interpretation of the code by the compiler and no event triggering statements are used. Such an association, for which the channel attribute is proposed, is useful when the communication between the sender and the receiver involves only handshaking and no data transfer.

For example, the following code associates named event `eventx` to the output channel of reg variable `x`:

Example 2.6:

```vhdl
(* async, channel = "x" *) event eventx;
always @(eventx)
  z <= y;
```

The association between `eventx` and `x` is declared in an attribute instance that assigns the value "`x"" to the channel attribute. The above code specifies that `eventx` is triggered by the activation of the output channel of `x` and that the execution of "`z <= y\" is delayed until `eventx` is triggered."
2.5.3.4. Guarded Channels

Guarded channels are those whose activations are dependent on conditions, termed guard expressions, that are explicitly defined in the design specification. Guarded channels (conditional communication) provide a convenient means of reducing unnecessary power dissipation because they permit only useful channel communication to propagate down a pipeline. Guarded channels are modeled using if statements that have no else-if and else clauses.

As explained in Section 2.1, conditional communication is different from clock gating in desynchronized circuits in that it not only prevents the propagation of useless data down a pipeline, but also the propagation of unnecessary handshakes. For control-dominated asynchronous pipelines with narrow datapaths, the blocking of unnecessary handshakes can potentially be important in reducing power dissipation.

In Example 2.5, the control channel x is guarded because it is activated only when expression is true. To model data channels that are guarded, the proposed guard attribute must be attributed to the corresponding if construct, as shown in the following Example 2.7.

Example 2.7:

```plaintext
always (* async, guard *) if (x == 3) z <= y;
```

The above code specifies that the output channel of variable z is guarded by the expression “x == 3”, i.e., the data transfer from variable y to z is carried out only if “x == 3” is true.
2.5.3.5. Channel Merging

Two incoming channels are said to be *merged* into a single outgoing channel if the activation of the outgoing channel requires only *one* of the incoming channels to be active (in contrast, a channel *join* requires *all* incoming channels to be active before the activation of the outgoing channel). A variable's input channels are inferred by the compiler to be merged if it has assignments in *more than one* *always* statements. Channel merging is equivalent to the *select* statement in Balsa, Tangram, and Haste.

In the following example, channel $x$ and channel $y$ are merged into a single channel $z$: if channel $x$ is active, then the value of variable $x$ is passed to variable $z$; on the other hand, if channel $y$ is active, then the value of variable $y$ is passed to variable $z$.

Example 2.8:

```plaintext
always
  z <= x;
always
  z <= y;
```

Note that by default the proposed synthesis method assumes that the channels to be merged are *not* mutually exclusive, i.e., it is possible for both channels to be active simultaneously. In such a case, *arbitration* between the channels, which requires additional hardware (see Section 2.6.2.1), is necessary. To avoid channel arbitration, channel pairs can be specified to be mutually exclusive using the proposed *mutually_exclusive* attribute:

```plaintext
(* async, mutually_exclusive = "x, y" *)
```

2.5.4. Looping Statements

The proposed synthesis method supports the *for* and *while* looping constructs. Looping statements can appear *only* in *always* statements and must be preceded by the
initialization of its *loop variables* (defined as variables that appear on the left-hand sides of the assignment statements in the looping statement).

Example 2.9:

```plaintext
always begin
  x = xin; y = yin;
  while (x != y) begin
    if (x > y)
      x <= x - y;
    if (y > x)
      y <= y - x;
  end
end
```

In the example above, a *while* loop is used to implement the greatest-common-divisor algorithm. The loop variables are *x* and *y*. Note that the assignments used to initialize the loop variables ("x = xin; y = yin;") are not allowed to have any data dependences on each other. As a result, the order of the assignments is irrelevant to the compilation of looping statements. This implies that sequencers are not necessary for the correct execution of the assignments.

### 2.5.5. Storage Devices

This subsection describes the modeling rules for level-sensitive and edge-sensitive storage devices.

#### 2.5.5.1. Level-Sensitive Storage Devices (Latches)

According to [99], a level-sensitive storage device (i.e., a latch) is inferred for a variable when: i) it is assigned a value in an *always* statement that has no edge events in its event list; and ii) there are executions in the *always* statement in which there is no explicit assignment to the variable, i.e., the variable is incompletely assigned.
The second condition, which differentiates the modeling of combinational logic from that of latches in RTL synthesis, poses a problem for this work. According to the proposed modeling rules, neither the handshake components nor their outputs for controlling latches are present in the design specification. Thus, for the proposed synthesis method, a variable that is completely assigned and intended by the designer to be a latch that is controlled by a handshake component is indistinguishable from a variable that is to be implemented as the output of a combinational logic block.

To solve this problem, it is proposed that: i) the nonblocking procedural assignment (\(\leq\)) be reserved for the inference of latches, i.e., the variables on the left-hand side of the operator \(\leq\) are to be implemented as latches; and ii) the modeling of combinational logic use only blocking procedural assignments (=). Indeed, this is a recommendation in [99] to avoid race conditions during RTL simulations.

2.5.5.2. Edge-Sensitive Storage Devices (Flip-Flops)

Most asynchronous designs use latches instead of edge-sensitive storage devices (hereafter referred to as flip-flops) to store data. This is because latches are smaller and dissipate less power compared to flip-flops.

However, there are situations where it is desirable to use flip-flops. An example is the storage of a self-dependent variable, i.e., one whose next value depends in part on its current value. Using latches directly is not possible since combinational logic loops are created when the latches are transparent. Instead, two sets of latches must be connected to form a sequential logic loop and care is taken to ensure that only one set is transparent at any one time. Such a loop would require handshake components for both sets of latches and handshake signaling between the handshake components, incurring area and power overheads.
These overheads can be avoided if flip-flops are used. Since a flip-flop is edge triggered, a combinational logic path from its output to its input does not pose a problem. Consequently, only one set of flip-flops is required to implement a self-looping variable instead of two sets of latches. Although a D-type flip-flop is composed of a master and a slave latch, it is likely to be smaller than the combined areas of two discrete latches due to the integration of its master and slave latches into a single cell. Furthermore, only one handshake component is required when flip-flops are used.

In this work, the choice between latches or flip-flops is transparent to the designer. The proposed synthesis method automatically detects self-dependent variables and generates the RTL descriptions in the compiled design such that flip-flops are inferred for them.

It is worth noting that although a design consisting of only self-dependent variables would have all of its pipeline stages implemented by flip-flops, it does not mean that the design has been rendered synchronous. This is because the handshake protocol (see Section 2.6.2.1) used in this work that governs data transfers across pipeline stages is applicable to pipeline stages implemented by latches or flip-flops. The only difference is that the latch-enable signal \( \text{en} \) generated by a handshake component, which is normally used to control latches, is now used as a clock signal to trigger flip-flops. Thus, each pipeline stage is controlled by its own local clock signal and no global clock is used to synchronize data transfers across pipeline stages.

Example 2.10:

```verbatim
always begin
   x <= x + y;
   y <= y + z;
   z <= z + x;
end
```

In the example above, the variables \( x, y, \) and \( z \) are all self-dependent and, thus, are implemented by flip-flops (see Fig. 2.3). It can be shown that the circuit in Fig. 2.3
operates correctly provided that at least one channel is active upon reset (if all channels are active upon reset, then at least one buffering pipeline stage must be inserted into the loop) and that the relevant timing constraints are satisfied. Proper initialization of asynchronous control networks is discussed in Section 2.6.3. Timing constraints that ensure correct data transfers across pipeline stages (implemented by latches or flip-flops) are described in Section 2.7.

2.6. Design Compilation

This section explains the proposed synthesis process, i.e., how the proposed synthesis method synthesizes the asynchronous control network of the design specification and establishes the control of the asynchronous control network over the design’s datapath. The proposed synthesis process comprises two main tasks: extraction of asynchronous communication channels and inference of handshake components. The proposed synthesis method is written in Perl (Practical Extraction and Report Language), a language suitable for text analysis and report writing.

2.6.1. Asynchronous Communication Channel Extraction

The proposed synthesis method uses directed graphs [101] as intermediate representations of asynchronous control networks. A directed graph $D$ consists of a set...
$V(D)$ of points called vertices, and a set $A(D)$ of ordered pairs of these vertices called arcs.

To extract the asynchronous communication channels modeled in the design specification, the proposed synthesis method first constructs a directed graph $D_1$ that captures all data and event dependences between the relevant entities (i.e., ports, nets, variables, and named events) declared in the specification. More specifically, the vertices of $D_1$ represent the relevant entities and an arc is drawn from an entity $x$ to another entity $y$ if $y$ is data- or event-dependent on $x$. An entity $y$ is said to be data-dependent on an entity $x$ if $x$ and $y$ are ports, nets, or variables, and the value of $y$ depends on that of $x$. On the other hand, $y$ is said to be event-dependent on $x$ if $x$ is a named event, and a triggering of $x$ causes an assignment to $y$ (if $y$ is a variable) or a triggering of $y$ (if $y$ is a named event).

Formally, an arc $(x, y)$ is drawn in $D_1$ from a vertex $x$ to a vertex $y$, where $x$ and $y$ are relevant entities in the Verilog specification, if one of the following conditions is true:

i) Assignments: $x$ is an operand of the right-hand side expression of a continuous or procedural assignment; $y$ is a net or variable on the left-hand side of the assignment (see Fig. 2.4(a)).

ii) Conditional statements: $x$ is an operand of an if or else-if expression; $y$ is either: a) a variable on the left-hand side of an assignment whose execution depends on the evaluation result of the if or else-if expression; or b) a named event whose triggering depends on the evaluation result of the if or else-if expression (see Fig. 2.4(b)).
iii) case statements: $x$ is an operand of a case expression; $y$ is either: a) a variable on the left-hand side of an assignment in the case statement; or b) a named event that is triggered in the case statement (see Fig. 2.4(c)).

iv) case items: $x$ is an operand of a case item expression; $y$ is either: a) a variable on the left-hand side of an assignment in the same case item or in a case item of a lower priority; or b) a named event that is triggered in the same case item or in a case item of a lower priority (see Fig. 2.4(c)).

v) Procedural timing control by named events: $x$ is a named event; $y$ is either: a) a variable on the left-hand side of an assignment that is executed when $x$ is triggered; or b) a named event that is triggered when $x$ is triggered (see Fig. 2.4(d)).

\[
y = x; \quad (a)
\]

\[
\text{if (v)} \quad \begin{array}{l}
x = w; \\
\text{else if (y)} \\
\quad \rightarrow z;
\end{array} \quad (b)
\]

\[
\text{case (u)} \\
v: x = w; \\
y: \rightarrow z; \\
\text{endcase} \quad (c)
\]

\[
\text{always @ w begin} \\
y = x; \\
\rightarrow z; \\
\text{end} \quad (d)
\]

Fig. 2.4. Constructing directed graph $D$: (a) assignments; (b) conditional statements; (c) case statements and case items; and (d) procedural timing control by named events.
An arithmetic-logic unit (ALU) model is used as a simple example to illustrate the construction of $D_1$.

Example 2.11:

(* async, channel = "ov" *) event ov_event;

always begin
  sum[4:0] = {op1[3], op1} + {op2[3], op2};
  diff[4:0] = {op1[3], op1} - {op2[3], op2};
  case (sel)
    0: x = sum;
    1: x = diff;
  endcase
end

always
  if (abs & x < 0)
    result <= -x[3:0];
  else
    result <= x[3:0];

always
  if (x > 7 | x < -8)
    -> ov_event;

The ALU accepts two four-bit signed (in two’s-complement) operands op1 and op2, and, depending on the value of sel, generates either the sum or difference of the operands. The absolute value of the operation’s result is generated if abs is set. If the operation causes an overflow in result, which is also four bits long, then the control channel ov is activated.

Fig. 2.5(a) shows $D_1$ of the ALU, in which the vertex set comprises all the entities used in the code, i.e., $V(D_1) = \{op1, op2, sel, abs, sum, diff, x, ov_event, result, ov, result_out\}$ (note that the suffix “_out” distinguishes output port result from variable result), and the arcs are drawn according to the conditions described earlier. For instance, the case statement in the code leads to three edges. The first two edges (sum, x) and (diff, x) are due to the procedural assignments $x = sum$ and $x = diff$, respectively, in the case items (see Condition i). The third edge (sel,
Fig. 2.5. The directed graphs constructed for the ALU model of Example 2.11: (a) $D_1$ and (b) $D_2$. The dotted arcs illustrate the extraction of the channels from $op_1$ to $ov_{event}$ and result.

$x$) represents the effect of $sel$ (an operand of the case expression) on the selection of the case item to execute and, therefore, the value assigned to $x$ (see Condition iii).

Using $D_1$, the proposed synthesis method constructs another intermediate representation $D_2$ also in the form of a directed graph. The vertices $V(D_2)$ of $D_2$ are the same as $V(D_1)$ but exclude all nets, as well as variables that are not controlled by handshake components. The arcs $A(D_2)$ of $D_2$ represent the asynchronous communication channels in the design and are extracted from $D_1$ using the following three steps.

First, $V(D_2)$ is decomposed into two (intersecting) subsets $V_D$ and $V_D$. $V_D$ is the set of all channel origins, i.e., the input ports, named events, variables that are to be controlled by handshake components, and outputs of instantiated modules. $V_D$ is the set of all channel destinations, i.e., the output ports, named events, variables that are to be controlled by handshake components, and inputs of instantiated modules.
Second, the channel destinations $V_{v_0}$ of each channel origin $v_o \in V_O$ are found by traversing in $D_1$ all paths leading away from $v_o$. In directed graphs, a path is a sequence of vertices and arcs of the form $v_1, a_1, v_2, a_2, \ldots, a_n, v_n$ such that all the vertices and arcs are different. When a path reaches a vertex $v_d \in V_D$, $v_d$ is added to $V_{v_0}$.

Third, arcs are drawn in $D_2$ from each $v_o \in V_O$ to each $v_d \in V_{v_0}$. Each arc $(v_o, v_d)$ represents an asynchronous communication channel in the design.

The ALU model of Example 2.11 is used to demonstrate the construction of $D_2$. From the model, it can be deduced that $V_O = \{opl, op2, sel, abs, result, ov\textunderscore event\}$ and $V_D = \{result, ov\textunderscore event, result\textunderscore out, ov\}$. Note that the nets sum and diff are not members of $V_O$ or $V_D$. Variable $x$ is also excluded from $V_O$ and $V_D$ because the model does not infer any storage devices for $x$. To find the channel destinations of, say, opl, all paths in $D_1$ leading away from opl are traversed, terminating at any vertex $v_d \in V_D$. These paths are shown as dotted arcs in Fig. 2.5(a). Thus, $V_{opl} = \{result, ov\textunderscore event\}$ and the corresponding channels are (opl, result) and (opl, ov\textunderscore event). Fig. 2.5(b) shows $D_2$ of the ALU.

In summary, $D_2$, which is the end product of the asynchronous communication channel extraction procedure described above, represents the underlying structure of the asynchronous control network of the design. The next step in the synthesis process is to refine each vertex of $D_2$ that represents a named event or a variable into a particular handshake component. This is discussed in the following section.

2.6.2. Handshake Component Inference

Three classes (synchronization, channel merging, and looping) of handshake components are proposed to support the proposed synthesis method in the design of asynchronous pipelines. The type of handshake component inferred for an entity
Fig. 2.6. The circuit inferred for the code in Example 2.2.

(variable or named event) in the design specification depends on the entity’s purpose and the characteristics of its input and output channels.

2.6.2.1. Synchronization Class

All members of the synchronization class perform synchronization (i.e., fork and join) operations on channels and provide control over storage devices. They differ in the type of channels that they serve. Members of the synchronization class are \texttt{Sync\{} , \texttt{PassiveOut\{} , \texttt{Arb\}}, \texttt{ActiveIn\}}, \texttt{Guard\}} , \texttt{ActiveIn\}}\}.

The \texttt{Sync} handshake component is a basic controller for storage devices that joins and forks to push channels and, thus, is used to implement a simple pipeline stage. It has passive input channels and active output channels. A \texttt{Sync} is inferred by the proposed synthesis method for a variable that satisfies the following conditions: i) the variable appears on the left-hand side of a nonblocking procedural assignment; and ii) the variable has passive input and active output channels. For example, the code in Example 2.2 infers a \texttt{Sync}, denoted as \texttt{S}, for variable \texttt{y}, as shown in Fig. 2.6. The input channel of \texttt{S} consists of the request signal \texttt{x_req}, the acknowledge signal \texttt{x_ack}, and the data \texttt{x}, and the output channel consists of the request signal \texttt{y_req}, the acknowledge signal \texttt{y_ack}, and the data \texttt{y}. The latch-enable signal \texttt{en_y} generated by \texttt{S} controls the storage device that stores \texttt{y}.
Fig. 2.7. The Sync handshake component: (a) symbol, (b) STG specification, and (c) circuit implementation.

Fig. 2.7(a) shows the symbol of a one-input, one-output Sync. The input channel handshake ports consist of the input request $ri$ and input acknowledge $ai$. The output channel handshake ports consist of the output request $ro$ and output acknowledge $ao$. The output port $en$ controls the storage devices. A Sync with multiple input channels joins the multiple input requests from its predecessors into a single signal and forks its input acknowledge to its predecessors. Likewise, a Sync with multiple output channels forks its output request to its successors and joins the output acknowledges from its successors into a single signal.

In asynchronous circuits, the "joining" of asynchronous signals is typically accomplished using Muller C-gates (or, simply, C-gate). A C-gate is a state-holding component that maintains the logic level of its output when its set and reset equations, whose operands are the C-gate's inputs, are false. It changes its output logic level to "1" when its set equation is true and to "0" when its reset equation is true. C-gates can be
implemented using standard library cells, such as set-reset flip-flops, or customized at the transistor level. For example, Fig. 2.8 shows the symbol and CMOS implementation of a three-input C-gate. Note that the notation used in Fig. 2.8 is such that: i) an input that is connected to the main body of the gate controls both the rising and falling transitions of the output; ii) an input that is connected to the extension marked as “+” controls only the rising transition of the output; and iii) an input that is connected to the extension marked as “−” controls only the falling transition of the output.

The design of basic latch controllers is well reported in the literature [72][80][102][103]. In this work, Sync (and the other handshake components) is designed to comply with the four-phase normally-closed broad handshake protocol [72][80][102][103] (see Fig. 2.9). Under this protocol, the latches operate in the normally-closed mode, where they remain closed (opaque) until valid input data has arrived, after which they open (become transparent) to register the input data. Note that this protocol ensures that data on a channel is valid from the assertion of the request signal req to the deassertion of the acknowledge signal ack.

Fig. 2.7(b) depicts one possible STG specification of Sync that complies with the handshake protocol. The STG is similar to those reported in [72][80][102][103], differing only in the operation of the latches. Intuitively, an STG is a kind of directed graph that consists of two types of nodes, called places and (signal) transitions, where
arcs are either from a place to a signal transition, or from a signal transition to a place (see Section 2.6.3.1 for a formal introduction to STGs). In graphical representation, places are drawn as circles. In STGs, it is very common for a place $p$ to have exactly one input transition $t_1$ (i.e., $p$ has exactly one incoming arc $(t_1, p)$ from $t_1$), and exactly one output transition $t_2$ (i.e., $p$ has exactly one outgoing arc $(p, t_2)$ to $t_2$). Such a structure, consisting of $(t_1, p)$, $(p, t_2)$, and $p$, is commonly replaced with a single arc $(t_1, t_2)$. According to the dynamics of STGs, a signal transition, such as $en^+$ (note that the polarities $\{+, -\}$ represent rising and falling transition, respectively) is enabled to occur (or fire) if all its incoming arcs or input places are marked. For example, $en^+$ is enabled if both its input arcs $(ri^+, en^+)$ and $(ao^-, en^+)$ are marked. An arc or place is said to be marked if it holds a token (which is depicted as a black dot beside the arc or inside the circle). The marking of the graph indicates which arcs are marked and which are not, and, therefore, represents the state that the controller is in. The initial marking of the graph refers to the marking immediately after the controller has been initialized. Once a signal transition has fired, the tokens held by its incoming arcs are removed and each of its outgoing arcs are marked with a token. For example, after $en^+$ has fired, its input arcs $(ri^+, en^+)$ and $(ao^-, en^+)$ become unmarked, and its output arc $(en^+, ro^+)$ becomes marked.

Based on Fig. 2.7(b), the behavior of Sync is as follows. Upon initialization, Sync waits for a request on its input channel $(ri^+)$, after which it enables its latches $(en^+)$ and simultaneously sends out an acknowledgement on its input channel $(at^+)$ and a request
on its output channel \((ro+)\). It then waits for the input request to clear \((ri-)\) and the acknowledgement on its output channel \((ao+)\). Subsequently, Sync disables its latches \((en-)\) and simultaneously clears its input acknowledge \((ai-)\) and output request \((ri-)\). The clearing of the input acknowledge completes the handshake cycle on the input channel and allows a new cycle to commence. Note that to prevent data overwrite, a new input request is not be serviced until the handshake cycle on the output channel is also complete \((ao-)\). This is enforced by joining the arcs \((ri+, en+)\) and \((ao-, en+)\) at \(en+\).

Fig. 2.7(c) shows the standard-C implementation of Sync synthesized by Petrify using the \(-gcm\) (generalized-C with monotonous cover) option (note that the circle with a “C” inside it depicts a standard two-input C-gate, where the term “standard” means that the inputs control both the rising and falling transitions of the gate’s output). This implementation is guaranteed by Sync to be speed-independent (i.e., its functionality is independent of gate delays) [8] and hazard-free.

The SyncPassiveOut handshake component (see Fig. 2.10) is a variant of Sync that is inferred for a variable that has a passive output channel, i.e. the output channel of the variable is a pull channel on which data transfers are initiated by the channel destination. As such, the input and output channels of a SyncPassiveOut can be considered as write and read ports, respectively. Note that the input and output channels of SyncPassiveOut are mutually exclusive. This implies that its behavior can be modeled as an STG with a free-choice between read and write operations (see Fig. 2.10 (b)). A free-choice in an STG is a place \(p\) (depicted as a circle) that has two or more output signal transitions such that the signal transitions have \(p\) as their only input place. When a free-choice is marked with a token, all its output signal transitions are enabled,
but only one can fire because the firing of any one of the transitions removes the token at the free-choice. Note that SyncPassiveOut can be implemented using only wires.

In the event that the input and output channels of the variable are not specified to be mutually exclusive (i.e., a write and a read can occur on the variable simultaneously), a SyncPassiveOutArb is inferred for the variable instead. A SyncPassiveOutArb consists of a SyncPassiveOut core and a channel arbiter [1][3] to arbitrate between concurrent read and write requests (see Fig. 2.11).

Fig. 2.12 shows the circuit implementation of the channel arbiter [1][3]. It consists of a mutual-exclusion element, which selects between its two possibly concurrent inputs ($r_1$ and $r_2$) and produces mutually-exclusive outputs ($g_1$ and $g_2$), and a pair of NAND-
Fig. 2.11. The SyncPassiveOutArb handshake component consists of a channel arbiter and a SyncPassiveOutCore.

Fig. 2.12. The channel arbiter [1][3].

Fig. 2.13. The mutual-exclusion element (CMOS implementation) [1][3].

gates, which prevent the second of two requests from proceeding before the handshake of the first request is completed.

Fig. 2.13 shows the CMOS implementation of the mutual-exclusion element [1][3], which consists of two cross-coupled NAND-gates and a filter. If both inputs $r_1$ and $r_2$ of the mutual-exclusion element go high at exactly the same time, then the NAND-gates outputs $q_1$ and $q_2$ may enter the metastable state, during which $V(q_1) = V(q_2) = V_T$, where $V_T$ is the logic threshold voltage of the NAND-gates. The filter eliminates the spurious values of $q_1$ and $q_2$ during the metastable state and ensures that the outputs of the
The SyncActiveIn handshake component (see Fig. 2.14) is a variant of Sync that is inferred for a variable that has both passive and active input channels. SyncActiveIn initiates a data transfer on its active input channel, which is always connected to a SyncPassiveOut component, after it has received a request on its passive input channel.

For example, the code in Example 2.4 infers a SyncPassiveOut, denoted as $P$, and a SyncActiveIn, denoted as $A$, to control the variables $y$ and $z$, respectively, as shown in Fig. 2.15. Upon receiving a request on its passive input channel ($x_{req}, x_{ack}$), $A$ sends a request on its active input channel ($y_{req}, y_{ack}$) to $P$ to read $y$ (note that the channel between $P$ and $A$ is a pull channel). If a write operation on $y$ is not in progress (i.e., $y$ is mutual-exclusion element $g_1$ and $g_2$ are at logic “0” during the metastable state. Mutual-exclusion elements can also be implemented using standard library cells [104].
Fig. 2.15. The circuit inferred for the code in Example 2.4.

stable), then $P$ acknowledges $A$. Subsequently, $A$ enables its storage devices to store $y$ and sends out a request on its output channel ($z_{req}, z_{ack}$).

The SyncGuard handshake component (see Fig. 2.16) is a variant of Sync that is inferred for a variable or named event whose output channel is guarded. Fig. 2.16(b) shows the STG specification for SyncGuard (recall that a circle depicts a place and a circle with a black dot depicts a place marked with a token; note that a bidirectional arc between a place $p$ and a signal transition $t$ depicts the presence of two arcs, one from $p$ to $t$ and the other from $t$ to $p$). After receiving an input request, SyncGuard’s input $g$ samples the evaluation result of the corresponding guard expression. If the result is “1”, then SyncGuard enables its storage devices to register the new data and activates its output channel. Otherwise, SyncGuard merely completes its input handshake.

Note that the channels that carry the operands of the guard expression are always inferred by the proposed synthesis method to be input channels of the corresponding SyncGuard. This implies that the evaluation result $g$ of the guard expression is guaranteed to be valid after the arrival of the (joined) input requests and, thus, can be safely sampled by the SyncGuard. For example, the code in Example 2.7 infers a SyncGuard for variable $z$ (see Fig. 2.17). The evaluation result $g$ of the guard
expression "x == 3" is guaranteed to be valid when it is sampled by the SyncGuard because the sampling occurs only after the request \texttt{x\_req} on channel \texttt{x} has arrived.

The \texttt{SyncGuardActiveIn} handshake component is a variant of \texttt{Sync} that combines the features of \texttt{SyncGuard} and \texttt{SyncActiveIn}, i.e., like the former, \texttt{SyncGuardActiveIn} has guarded output channels, and like the latter, it has active input channels. \texttt{SyncGuardActiveIn} samples the evaluation result of its guard expression \texttt{after} receiving an acknowledgement on its active input channel.
2.6.2.2. Channel Merging Class

Members of the *channel merging class* perform merging operations on channels. They include ChMerge{, {Arb}, Cont}.

The ChMerge handshake component (see Fig. 2.18) is inferred for a variable whose input channels need to be merged. The input channels of ChMerge are assumed to be mutually exclusive. This allows the behavior of ChMerge to be modeled using an STG (see Fig. 2.18(b)) that contains a free choice between executing the handshake for the first or second input channel. ChMerge generates a signal \( s \) that controls a data multiplexer in the datapath so that the data whose channel is active is selected by the data multiplexer. Note that the latching of the data is separately controlled by a Sync that is automatically instantiated and connected to the ChMerge by the proposed synthesis method. As such, it is necessary for the input handshake of a ChMerge to *enclose* its output handshake (i.e., the input handshake can only complete after the completion of the output handshake) to ensure correct data transfers.

For example, the code in Example 2.8 infers a ChMerge and a Sync for variable \( z \), as shown in Fig. 2.19. Using a data multiplexer, the signal \( s_z \) selects either variable \( x \) or variable \( y \) as the input to \( z \), depending on which input channel is active.
Fig. 2.18. The ChMerge handshake component: (a) symbol, (b) STG specification, and (c) circuit implementation.

If the input channels of the variable are not specified to be mutually exclusive, then a \textit{ChMergeArb}, which consists of a ChMerge and a channel arbiter to arbitrate between concurrent input requests, is inferred for the variable.

The \textit{ChMergeCont} handshake component is a variant of ChMerge that merges control channels. Since control channels do not include datapaths, a ChMergeCont does not generate the data-select signal.
Fig. 2.19. The circuit inferred for the code in Example 2.8.

Fig. 2.20. The circuit inferred for the code in Example 2.9.

2.6.2.3. Looping Class

Members of the looping class (LoopFront and LoopEnd) implement the looping statements (for and while) in the specification. Looping statements are compiled into the structure shown in Fig. 2.20, the operation of which is governed by the initiation, evaluation, and iteration channels. The initiation channel initiates looping operation by sending a request to LoopFront, which then activates the evaluation channel \{eval\_req, eval\_ack\}, during which the loop body is executed and the looping condition evaluated (the proposed synthesis method treats the loop body as a datapath
Fig. 2.21. The LoopFront handshake component: (a) symbol and (b) STG specification.

specification and, thus, does not synthesize it). Once the computation in the loop body is complete, the evaluation result $g$ of the looping condition is sampled by both LoopFront and LoopEnd. If $g$ is "1", then iteration is required and LoopEnd activates the iteration channel, which in turn causes LoopFront to activate the evaluation channel again. If, on the other hand, $g$ is "0", then iteration is not required, in which case LoopFront
Fig. 2.22. The LoopEnd handshake component: (a) symbol and (b) STG specification.

Completes the handshake on the initiation channel and LoopEnd activates the output channel. Fig. 2.21 and Fig. 2.22 show the symbols and STG specifications for LoopFront and LoopEnd, respectively. Fig. 2.23 shows the circuit implementations of LoopFront and LoopEnd.

In the compiled structure, two sets of latches are inferred for the loop variables. The first set of latches are controlled by LoopFront and act as buffers for the loop variables. They receive as input either the initial or iterated values of the loop variables, via the initiation or iteration channel, respectively. The buffers provide inputs to the computations in the loop body. The second set of latches are controlled by LoopEnd and...
serve as storage devices for the loop variables. They receive as input either the results of the computations in the loop body when there is to be another iteration of the loop or the values stored in the buffers when the loop is to be exited.

The compiled structure handles only one set of data at any one time and does not accept new data when it is in operation. This is ensured by specifying the behavior of LoopFront such that a new request on the initiation channel is acknowledged only after the last handshake on the iteration channel is completed.

2.6.3. Initialization of Asynchronous Control Networks

In this work, a pipeline stage can be either idle or active upon initialization. A pipeline stage that is idle upon initialization (the default for all pipeline stages) remains idle until an input request arrives. In contrast, a pipeline stage that is active upon initialization (specified using the active_upon_reset attribute) holds valid data that needs to be consumed by its succeeding stages. This flexibility requires that a handshake component has two selectable initial states with respect to whether it has idle or active output channels upon initialization. It is of interest to note that the two selectable initial states of a handshake component in this work are similar to the initial states of the odd and even latch controllers in [80].
For example, in Fig. 2.7(b), the initial marking of the STG specification for Sync specifies that it has idle output channels upon initialization. This means that upon initialization, all handshake signals \((ri, ai, ro, and ao)\) of Sync are at logic low and that Sync is waiting for the input request to arrive \((ri+)\). On the other hand, Fig. 2.24 shows the same STG specification for Sync but with a different initial marking such that Sync has active output channels upon initialization. This means that upon initialization, all handshake signals of Sync are at logic low, except for \(ro\), which is at logic high. Thus, Sync is concurrently waiting for the arrivals of the input request \((ri+)\) and output acknowledge \((ao+)\). Note that these two STGs can be implemented using the same circuit. The desired initial state of the circuit can be selected using two one-hot coded reset wires.

Proper initialization of an asynchronous control network is necessary to ensure that it does not enter a deadlock situation during its operation. This is particularly important for asynchronous pipelines that contain feedback loops [105]. For example, Fig. 2.25 shows a two-stage feedback loop consisting of two basic latch controllers \((X\) and \(Y)\) connected back-to-back. Clearly, if neither \(X\) nor \(Y\) are specified by the designer to have active output channels upon initialization (see Fig. 2.25(a)), then the feedback loop is in a deadlocked. To resolve the deadlock, either \(X\) or \(Y\) must have active output channels upon initialization (see Fig. 2.25(b) and Fig. 2.25(c), respectively).
In this subsection, a method is proposed for computing an initial state for an asynchronous control network if it is found to have a deadlock problem due to an improper initialization specified by the designer. As it shall be shown, the computed initial state ensures that the control network is free of deadlocks and preserves the nondeadlock behavior of the control network.

2.6.3.1. Basic Petri Nets Concepts

The proposed method for computing a deadlock-free initial state for asynchronous control networks is based on the theory of PNs (Petri nets). This subsection formally introduces some basic concepts of PNs [87].

PNs are a graphical and mathematical modeling tool that is widely used to describe and study information processing systems that are characterized as being concurrent, asynchronous, distributed, or parallel. Formally, a PN $N$ is a 5-tuple, $N = (P, T, F, W, M_0)$, where $P(N) = \{p_1, p_2, \ldots, p_n\}$ is a finite set of $n$ places, $T(N) = \{t_1, t_2, \ldots, t_m\}$ is a finite set of $m$ transitions, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (or flow relation), $W : F \rightarrow \{1, 2, 3, \ldots\}$ is a weight function, $M_0 : P \rightarrow \{0, 1, 2, \ldots\}$ is the initial marking, $P \cap T = \emptyset$, and $P \cup T \neq \emptyset$. Typically, places are depicted pictorially as circles and transitions as bars or boxes.
A PN is said to be ordinary if its arc weights are all 1's. STGs, informally introduced in Section 2.6.2.1, are a special case of ordinary PNs in which each place can contain no more than one token.

In order to simulate the dynamic behavior of a system, a state or marking in a PN is changed according to the following transition (or firing) rule:

i) 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 \( t \).

ii) An enabled transition may or may not fire (depending on whether or not the event actually takes place).

iii) 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 \( p \).

The sets of input and output places of a transition \( t \) are called the pre-set and post-set of \( t \), respectively, and are denoted as \( \bullet t \) and \( t \bullet \), respectively. Similarly, the sets of input and output transitions of a place \( p \) are called the pre-set and post-set of \( p \), respectively, and are denoted as \( \bullet p \) and \( p \bullet \), respectively.

The incidence matrix \( A = [a_{ij}] \) of a PN is an \( n \times m \) matrix of integers and its typical entry is given by \( a_{ij} = a_{i}^{+} - a_{i}^{-} \) where \( a_{i}^{+} = w(j, i) \) is the weight of the arc from transition \( j \) to its output place \( i \), and \( a_{i}^{-} = w(i, j) \) is the weight of the arc from place \( i \) to its output transition \( j \). It is easy to see from the transition rule that \( a_{i}^{+}, a_{i}^{-}, \) and \( a_{ij} \) represent the number of tokens added, removed, and changed, respectively, in place \( i \) when transition \( j \) fires once.

An \( S \)- or place invariant \( y \) (\( T \)- or transition invariant \( x \)) is an \( n \)-vector (\( m \)-vector) of integers such that \( A^{T}y = 0 \) (\( Ax = 0 \)). Intuitively, an \( S \)-invariant can be viewed as a set of
places whose weighted sum of tokens remains unchanged for all reachable markings of
the PN. On the other hand, a T-invariant indicates the number of times a transition fires
in a firing sequence from a marking $M$ back to $M$.

The set of places (transitions) corresponding to nonzero entries in an S-invariant $y \geq
0$ (T-invariant $x \geq 0$) is termed the support of an invariant and is denoted by $|y|$ ($|x|$).
A support is said to be minimal if no proper nonempty subset of the support is also a
support. An invariant $y$ is said to be minimal if there is no other invariant $y_1$ such that
$y_1(p) \leq y(p)$ for all $p$. Given a minimal support of an invariant, there is a unique
minimal invariant corresponding to the minimal support. Such an invariant is termed a
minimal-support invariant. The set of all possible minimal-support invariants serves as
a generator of invariants, i.e., any invariant can be written as a linear combination of
minimal-support invariants.

A siphon $S$ is a subset of places such that every transition that has an output place in
$S$ also has an input place in $S$. A siphon has a behavioral property that if it is token-free
under some marking, then it remains token-free under each successor marking. A trap $Q$
is a subset of places such that every transition that has an input place in $Q$ also has an
output place in $Q$. A trap has a behavioral property that if it is marked (i.e., it has at least
one token) under some marking, then it remains marked under each successor marking.
A siphon is said to be minimal if it does not contain any other siphon. A trap is said to
be maximal if it is not contained in any other trap.

A timed PN is one whose transitions or places are associated with delays. In a
periodically-operated timed PN, a period $\tau$ is defined as the time to complete a firing
sequence in the net leading back to the starting marking after firing each transition at
least once. $\tau$ is termed a cycle time. If the delays are associated with the places of the PN
and are deterministic, then it has been shown [87] that the lower bound of \( \tau \), termed the \textit{minimum cycle time} and denoted as \( \tau_{\text{min}} \), is given by

\[
\tau_{\text{min}} = \max_k \left\{ y_k^T D \left( A^+ \right) x / y_k^T M_0 \right\}
\]  

(2.1)

where the maximum is taken over all minimal-support S-invariants \( y_k \geq 0 \), \( D \) is the diagonal matrix of place delay \( d_i \) for \( i = 1, 2, \ldots, n \), \( A^+ = \left[ a_{ij}^+ \right] \), \( x > 0 \) is a T-invariant, and \( M_0 \) is the starting marking.

The method for computing \( \tau_{\text{min}} \) can be explained intuitively as follows. First, all minimal-support S-invariants of the PN model are computed (see Chapter 4 for a detailed discussion on algorithms for computing minimal-support S-invariants). Second, each minimal-support S-invariant \( y_k \) is associated with a "cycle time" \( \lambda_k \), which is given by the expression \( \lambda_k = \left\{ y_k^T D \left( A^+ \right) x / y_k^T M_0 \right\} \). Put simply, the numerator \( \left( y_k^T D \left( A^+ \right) x \right) \) is the weighted sum of the delays that are associated with the support of \( y_k \) and the denominator \( \left( y_k M_0 \right) \) is the number of tokens held by the support of \( y_k \). Third, the maximum \( \lambda_k \) is taken over all \( y_k \) to produce \( \tau_{\text{min}} \).

2.6.3.2. Deadlock Detection

The presence of deadlocks in an asynchronous control network can be detected by first composing a PN model of the control network and then checking if the model is \textit{live}. A PN is said to be live if it is possible to ultimately fire any transition in the net by progressing through some further firing sequence, independent of the marking that has been reached from the initial marking. Thus, a live PN model guarantees that the corresponding control network is free of deadlocks.

The composition of the PN model of the control network is based on abutting the modular PN models of handshake components in the control network that are adjacent.
to each other. The modular PN model of a handshake component is derived in two steps. First, the STG specification for the handshake component is reduced by projecting it on the handshake and latch-enable signals, thus abstracting away its behaviors related to the other signals. This can be accomplished by Petrify using the hide option, which specifies the signals to be hidden. Note that the original and reduced STGs are observationally equivalent with respect to the handshake and latch-enable signals. Second, each external arc \((t_1, t_2)\) and the associated token (if any) is replaced with a sink place that has \(t_1\) as its only input transition and a source place that has \(t_2\) as its only output transition. External arcs are those that model the handshakes between adjacent handshake components. For example, the external arcs in the STG for Sync (see Fig. 2.7(b)) are \((a_i-, r_i^+), (a_i^+, r_i^-), (r_o^+, a_o^+),\) and \((r_o^-, a_o^-)\). Fig. 2.26 shows the modular PN models of the handshake components used in this work.

Two modular PN models are abutted together by fusing the corresponding source and sink places. For example, Fig. 2.27(b) shows the PN model of the control network depicted in Fig. 2.27(a). The PN model is composed by fusing the sink places of \(r_o^+\) and \(r_o^-\) with the source places of \(r_{i2}^+\) and \(r_{i2}^-\), respectively, and by fusing the source places of \(a_o^+\) and \(a_o^-\) with the sink places of \(a_{i2}^+\) and \(a_{i2}^-\), respectively. For convenience, an arc that is incident from \(r_o^+\) and incident to \(r_{i1}^+\) is called an active-request place, one that is incident from \(a_o^-\) and incident to \(e_{n1}^+\) is called an end-of-output-RTZ place, and an arc that is incident from \(e_{n1}^-\) and incident to \(r_o^-\) is called a start-of-output-RTZ place.

Once the PN model of the control network has been composed, its liveness can be investigated. Algorithmically speaking, reachability analysis [87] is the simplest method for liveness investigation. However, it involves computing the entire state space of the PN and, thus, suffers from the state explosion problem.
To avoid exhaustive reachability analyses, deadlock detection methods based on PN unfolding have been reported [106][107][108]. In essence, PN unfolding involves first constructing the finite prefix of an occurrence net (an acyclic net where all places have no more than one input transition) that is equivalent to the PN of interest, and then analyzing the occurrence net for the presence of deadlocks. However, it is known that the problem of deadlock detection by unfolding is NP-complete [106][107][108].
Fig. 2.27. (a) An asynchronous control network comprising two Sync handshake components and (b) the corresponding PN model.

In particular, deadlock detection algorithms that work on occurrence nets are known to have exponential complexity relative to the sizes of the nets.

Yet another method that is commonly used to check the liveness of PNs is to determine whether they satisfy the siphon-trap property. A PN $N$ satisfies the siphon-trap property if every siphon in $N$ contains a marked trap. It is well known that an asymmetric choice net (ACN) is live if it satisfies the siphon-trap property [87]. An ACN is an ordinary PN such that for all pairs of places $\{p_1, p_2\}$ in the net that have common output transitions, either the output transition set of $p_1$ is a subset of that of $p_2$, or vice versa. As it shall be shown in the following section, a PN that is composed of the modular PN models of the handshake components used in this work is an ACN. Thus, the siphon-trap property method for determining liveness of PNs is relevant to this work. Like the unfolding method, the siphon-trap property method also suffers from the exponential complexity problem because the number of siphons and traps in a PN can increase exponentially with the size of the net.

In this work, the siphon-trap property method for deadlock prevention is favored over the unfolding method because the former leads directly to a simple technique for initializing a control network such that it is free of deadlocks (see following subsection). While it is true that the feasibility of the siphon-trap property method is affected by the potentially large number of siphons a PN can have, it is argued that this should not be
viewed as a disadvantage compared with the unfolding method because the latter also involves exponentially complex deadlock detection algorithms [106][107][108]. Furthermore, efficient algorithms for the computation of siphons have been reported [109].

2.6.3.3. Deadlock Prevention By Proper Initialization

The problem of finding a new initial state for an asynchronous control network such that the control network is free of deadlocks and that the nondeadlock behavior of the control network resulting from the specified initial state is preserved is formulated as follows.

**Problem:** Given a nonlive PN \((N, M_0)\), where \(N\) is a PN model of the control network and \(M_0\) is the initial marking in \(N\) that corresponds to the specified initial state of the control network, find a new initial marking \(M_0'\) for \(N\) such that

i) \((N, M_0')\) is live; and

ii) Any transition firing sequence \(\sigma\) in \((N, M_0)\) is also feasible in \((N, M_0')\).

Since the proposed solution to the problem is based on the siphon-trap property method, it is necessary to first show that a PN that is composed of the modular PN models of the handshake components used in this work is an ACN. Formally, an ACN is an ordinary PN such that

\[
p_1 \cap p_2 = \emptyset \Rightarrow p_1 \subseteq p_2 \text{ or } p_2 \subseteq p_1 \quad \text{for all } p_1, p_2 \in P
\]

(2.2)

The following theorem states the conditions under which the composition of two ACNs is also an ACN.

**Theorem 2.1:** Let \(N_1\) and \(N_2\) be ACNs. Let \(P_1\) and \(P_2\) be the sets of places in \(N_1\) and \(N_2\), respectively. Let \(P_{\text{sink}1} \subset P_1\) and \(P_{\text{sink}2} \subset P_2\) be sets of sink places, i.e., \(|p\ast| = 0\) for every \(p \in P_{\text{sink}1} \cup P_{\text{sink}2}\). Let \(P_{\text{source}1} \subset P_1\) and \(P_{\text{source}2} \subset P_2\) be sets of source places, i.e.,
|p| = 0 for every p ∈ PS1 ∪ PS2. For every p ∈ PS1 ∪ PS2, let p have exactly one output transition, i.e., |p*| = 1. If N3 is composed using N1 and N2 through a one-to-one fusion of the places of PS1 and PS2, and a one-to-one fusion of the places of PS1 and PS2, then N3 is also an ACN.

Proof: Suppose that the fusion of the places is performed one pair at a time. Let p1 ∈ PS1 and p2 ∈ PS2. Let p3 be the place that is created by the fusion of p1 and p2. To prove that N3 is an ACN after the fusion of p1 and p2, it is sufficient to show that p3 satisfies condition (2.2). We know that p3 = p1 ∪ p2 = p1* and p3* = p1* ∪ p2* = p2*.

Let p4 be a place in N2 that shares a common output transition with p2. Thus, p4 shares a common output transition with p3 after the fusion of p1 and p2, i.e., p4* ∩ p3* = ∅. Since |p3*| = |p2*| = 1, it follows that p3* ⊆ p4*, i.e., p3 satisfies condition (2.2). The same argument can be applied to any one-to-one fusion of the places of PS1 and PS2, and to any one-to-one fusion of the places of PS1 and PS2. Thus, N3 is an ACN.

From Fig. 2.26, it can be seen that the modular PN models of the handshake components relevant to this work are all ACNs because all places in the models satisfy condition (2.2). Furthermore, it can be seen that every source place in the modular PN models has exactly one output transition. Thus, it follows from Theorem 2.1 that if a PN model N3 is composed using any two modular PN models N1 and N2 through a one-to-one fusion of the source places of N1 and the sink places of N2, and a one-to-one fusion of the sink places of N1 and the source places of N2, then N3 is also an ACN.

The basic idea in the proposed method for computing a live initial marking in a PN (N, M0) is to modify M0 into M0' such that (N, M0') satisfies the siphon-trap property. This means that if a siphon S in N is found to be clean (i.e., it contains no tokens) at M0, then a trap Q in S is identified and a place in Q is selected to be assigned a token at M0'. Note that it is sufficient to consider only minimal siphons and maximal traps. The place
p in Q that is selected to be assigned a token must be an active-request place. This is because assigning a token to p means the corresponding channel, which is idle at the specified initial state of the control network, is now being respecified to be an active channel at the new initial state. Once p is assigned a token, it is necessary to also assign tokens to the other active-request places that share the same input transition with p and to the corresponding start-of-output-RTZ place. It is also necessary to remove the token at the corresponding end-of-output-RTZ place. In terms of handshake component initialization, this reassignment of the initial marking is equivalent to changing the initial state of a handshake component from one where the output channels are idle to one where the output channels are active.

The algorithm for computing a live initial marking in a composed PN model N is formally stated as follows:

1) Label the first initial marking in the composed PN model N as the root node and tag it as “new”.

2) While a “live” initial marking has not yet been found and there exists at least one “new” initial marking, do the following:

   2.1) Select a “new” initial marking M.

   2.2) If (N, M) satisfies the siphon-trap property, then tag M as “live” and return to Step 2.

   2.3) If M is identical to an initial marking on the path from the root node to M, then tag M as “old” and return to Step 2.

   2.4) Choose a clean trap Q in (N, M) (do not choose the same trap twice).

   2.5) For each active-request place p in Q, do the following:

      2.5.1) Compute a new initial marking M by:
2.5.1.1) Assigning tokens to \( p \) and the other active-request places that share the same input transition with \( p \);

2.5.1.2) Assigning a token to the corresponding start-of-output-RTZ place; and

2.5.1.3) Removing the token at the corresponding end-of-output-RTZ place.

2.5.2) Introduce \( M' \) as a "new" node and draw an arc from \( M \) to \( M' \).

In essence, the algorithm constructs a tree, in which a node represents an initial marking in \( N \) (the root node represents the specified initial marking \( M_0 \)) and an arc represents a change in the initial state of a handshake component from one where the output channels are idle to one where they are active. For an initial marking \( M \) to have child nodes, it must not be live and must not be identical to any markings along the path from the root node to \( M \). Once the tree has been constructed, the compiler parses each live initial marking in the tree and selects the one that represents an initial state that is most similar to the specified initial state.

2.6.3.4. Preservation of Nondeadlock Behavior

This subsection formally proves that the new initial state computed by the proposed algorithm for a control network preserves the nondeadlock behavior of the control network resulting from the specified initial state.

In the following, let \( N \) be the PN model of the control network, \( P \) be the set of places in \( N \), \( T \) be the set of transitions in \( N \), \( M_0 \) be the initial marking in \( N \) that corresponds to the specified initial state of the control network, \( M'_0 \) be the new initial marking in \( N \) that corresponds to the new initial state of the control network, and \( R(M) \) be the set of all markings in \( N \) reachable from \( M \).
The notions of active-request places, start-of-output-RTZ places, and end-of-output-RTZ places are formally defined as follows.

**Definition 2.1:** A place in $N$ is called an active-request place if it is incident from a signal transition $ro_i^+$ and incident to a signal transition $ri^+$.  

**Definition 2.2:** A place in $N$ is called a start-of-output-RTZ place if it is incident from a signal transition $en_i^-$ and incident to a signal transition $ro_i^-$.  

**Definition 2.3:** A place in $N$ is called an end-of-output-RTZ place if it is incident from a signal transition $ao_i^-$ and incident to a signal transition $en_i^+$.  

Let $X$, $Y$, and $Z$ be the sets of active-request places, start-of-output-RTZ places, and end-of-output-RTZ places, respectively, of $N$. Since the difference between the specified and new initial states of the control network lies only in the initial state of one or more handshake components, it is clear that the only difference between $M_0$ and $M_0'$ is in their assignments of tokens to a subset $U$ of $X \cup Y \cup Z$. From the following definition of $U$, it is clear that $U$ completely describes the difference between $M_0$ and $M_0'$.

**Definition 2.4:** $U$ is a subset of $X \cup Y \cup Z$ such that: 1) for each place $p \in U \cap (X \cup Y)$, $M_0(p) = 0$ and $M_0'(p) = 1$; and 2) for each place $p \in U \cap Z$, $M_0(p) = 1$ and $M_0'(p) = 0$.  

The following lemma states that for each place $p_1 \in U$, $p_1$ and $p_1^*$ are dead, i.e., $p_1$ and $p_1^*$ are not enabled at each marking $M \in R(M_0)$ and, thus, can never fire. This implies that for each place $p_1 \in U$, $M(p_1) = M_0(p_1)$ for each marking $M \in R(M_0)$.  

**Lemma 2.1:** For each place $p_1 \in U$, $p_1$ and $p_1^*$ are dead (i.e., $p_1$ and $p_1^*$ are not enabled at each marking $M \in R(M_0)$ and, thus, can never fire).

**Proof:** There are exactly three cases to be considered: $p_1 \in U \cap X$, $p_1 \in U \cap Y$, and
\( p_1 \in U \cap Z \). In the following, let \( S \) be a clean siphon in \((N, M_0)\).

Case 1 \((p_1 \in U \cap X)\): If \( p_1 \in U \cap X \), then \( p_1 \in S \). This is because the proposed algorithm for computing \( M'_0 \) in \( N \) only selects active-phase request places from clean siphons in \((N, M_0)\). Since \( S \) is clean at \( M_0 \), it is clean at each marking \( M \in R(M_0) \). This implies that \( S \) are dead. Since \( p_1 \in S \subset S^* \) and \( p_1 \in S^* \), it follows that \( p_1 \) and \( p_1^* \) are dead.

Case 2 \((p_1 \in U \cap Z)\): According to the proposed algorithm for computing \( M'_0 \) in \( N \), if \( p_1 \in U \cap Z \), then there exists another place \( p_2 \) such that \( p_2 \in U \cap X \), \( p_2 \in S \), and \( p_2 \) corresponds to \( p_1 \) (i.e., the input transitions of \( p_1 \) and \( p_2 \) (\( ao_{-} \) and \( ro_{+} \), respectively) have the same subscript). Using \( p_1 \in S \subset S^* \), we know that \( p_1 \in S \in S^* \), i.e., \( p_1 \) is dead. Furthermore, it can be deduced from the STG models (see Fig. 2.7 and Fig. 2.26) of the handshake components used in this work that if \( (ro_{+})^* \) are clean at \( M_0 \), then the first firing of \( ao_{-} \) can only occur after the first firing of \( ro_{+} \). Since \( \{ro_{+}\} = p_2 \subset S^* \), it follows that \( ro_{+} \) is dead. This implies that \( ao_{-} \) (i.e., \( p_1 \)) is dead.

Case 3 \((p_1 \in U \cap Y)\): According to the proposed algorithm for computing \( M'_0 \) in \( N \), if \( p_1 \in U \cap Y \), then there exists another place \( p_2 \) such that \( p_2 \in U \cap X \), \( p_2 \in S \), and \( p_2 \) corresponds to \( p_1 \) (i.e., the input transitions of \( p_1 \) and \( p_2 \) (\( en_{-} \) and \( ro_{+} \), respectively) have the same subscript). It can be deduced from the STG models (see Fig. 2.7 and Fig. 2.26) of the handshake components used in this work that the first firing of \( en_{-} \) can only occur after the first firing of \( ro_{+} \). Since \( \{ro_{+}\} = p_2 \subset S^* \), it follows that \( ro_{+} \) is dead. This implies that \( en_{-} \) (i.e., \( p_1 \)) is dead. Furthermore, the output transition of \( p_1 \) (i.e., \( ro_{-} \)) has \( p_1 \) as its only input place. Since \( p_1 \) is dead, \( p_1 \) is clean at each \( M \in R(M_0) \). Thus, \( p_1 \) is dead.
The notion of equivalent markings is now introduced.

**Definition 2.5:** Two markings $M \in R(M_0)$ and $M' \in R(M_0')$ are said to be equivalent if: 1) for each place $p \in U \cap (X \cup Y), M(p) = 0$ and $M'(p) = 1$; 2) for each place $p \in U \cap Z, M(p) = 1$ and $M'(p) = 0$; and 3) for each place $p \in P \setminus U, M(p) = M'(p)$.

Note that $M_0$ and $M_0'$ are equivalent markings. The following lemmas lead to Theorem 2, which states that any firing sequence $\sigma$ in $(N, M_0)$ is also feasible in $(N, M_0')$.

**Lemma 2.2:** Let $M \in R(M_0)$ and $M' \in R(M_0')$ be equivalent markings. If a transition $t \in T$ is enabled at $M$, then $t$ is enabled at $M'$.

**Proof:** Consider the contrapositive of Lemma 1, i.e., for each place $p \in P$, if $\bullet p$ or $p\bullet$ are not dead, then $p \notin U$. This implies that if a transition $t \in T$ is not dead, then $\bullet t \subset U$. Thus, for each place $p \in \bullet t, M(p) = M'(p)$. It therefore follows that if $t$ is enabled at $M$, then $t$ is enabled at $M'$.

**Lemma 3:** Let $M_1 \in R(M_0)$ and $M_1' \in R(M_0')$ be equivalent markings. If a transition $t \in T$ is enabled at $M_1$ and $M_1'$, then the respective succeeding markings, $M_2$ and $M_2'$, after firing $t$ are equivalent.

**Proof:** From the contrapositive of Lemma 1, we know that if a transition $t \in T$ is enabled at $M_1$, then $t \bullet \subset U$. This implies that for each place $p \in U, M_1(p) = M_2(p)$ and $M_1'(p) = M_2'(p)$. For each place $p \in P \setminus U$, it is obvious that $M_2(p) = M_2'(p)$. Thus, $M_2$ and $M_2'$ are equivalent markings.

Lemma 2 and 3 lead to the following theorem.

**Theorem 2:** Any transition firing sequence $\sigma$ in $(N, M_0)$ is also feasible in $(N, M_0')$.

**Proof:** Since $M_0$ and $M_0'$ are equivalent, it follows that for each transition $t \in T$ that is enabled at $M_1$, $t$ is also enabled at $M_0'$. Furthermore, from Lemma 3, we know that the
succeeding markings $M_1$ and $M'_1$ of $M_0$ and $M'_0$, respectively, after the firing of $t$ are equivalent. Since $M_1$ and $M'_1$ are equivalent, the above argument can be applied to $M_1$ and $M'_1$, and their succeeding markings. Thus, it can be deduced that any firing sequence in $N$ originating from $M_0$ or any marking $M \in R(M_0)$ can be replicated by starting from $M_0'$ or any marking $M' \in R(M'_0)$, where $M$ and $M'$ are equivalent.

### 2.7. Timing Constraints

This section describes the timing constraints that must be satisfied to ensure correct data transfers across pipeline stages.

The case where the receiving pipeline stage is implemented by latches is first considered. The timing constraints, with reference to Fig. 2.28, are as follows. First, valid data must arrive at the receiver before the assertion of the receiver’s latch-enable signal $en_r$ (note that the latch-enable signals are active-high), i.e.,

$$T_a \geq T_{max} - T_1 - T_2$$  \hspace{1cm} (2.3)

where $T_a$, called the active request delay of the channel, is the rising transition delay from the sender’s output request pin $ro_s$ to the receiver’s input request pin $ri_r$, $T_1$ and $T_2$ are internal delays within the handshake components, and $T_{max}$ is the maximum delay of the path $Z_1$ (see Fig. 2.29(a)), which originates at the sender’s latch-enable pin $en_s$, goes through the sender’s latch-enable buffer tree, the sender’s registers, and the combinational logic between the sender and receiver, and terminates at the input data pins of the receiver’s latches (see Fig. 2.29(a)).

As shown in Fig. 2.29, $T_a$ is implemented by placing a delay element $D_a$ on the request wire of the channel between the sender and the receiver. Note that to take into account the variations in circuit delays due to process and temperature variations, some safety margin should be added to the constraint by overestimating $T_{max}$ (deciding the
Fig. 2.28. Timing diagram analysis to ensure correct data transfers across pipeline stages.

amount of safety margin involves finding the “right” balance between performance and robustness; a safety margin of 100% is not uncommon [67][81]).

It is worth noting that (2.3) is conservative because for the receiver to register its input data correctly, it is not necessary that valid data arrives at the receiver before the assertion of \( en_r \), but only that it arrives \( T_{su1} \) before the deassertion of \( en_r \), where \( T_{su1} \) is the setup time of latches. However, the arrival of valid data after the assertion of \( en_r \) complicates the determination of \( T_{max1} \) for the succeeding pipeline stage because \( T_{max1} \) would now have to include the latency between the assertion of \( en_r \) and the arrival of valid data. This complication can be avoided by satisfying (2.3).

Second, the deassertion of \( en_r \) must occur \( T_{su1} \) after the arrival of valid data, where \( T_{su1} \) is the setup time of latches. Since valid data is guaranteed to arrive before the assertion of \( en_r \), this constraint is equivalent to the requirement that \( T_{su1} \) be longer than the duration of the assertion of \( en_r \), i.e.,

\[
T_3 + T_4 + T_5 + T_{re} + T_6 \geq T_{su1}
\]  

(2.4)

where \( T_{re} \) is the falling transition delay from the pin \( ro_s \) to the pin \( ri_r \), \( T_4 \) is the rising transition delay from the receiver’s input acknowledge pin \( ai_r \) to the sender’s output.
The acknowledge pin $a_o$, and $T_3$, $T_5$, and $T_6$ are internal delays within the handshake components. The satisfaction of (2.4) is trivial because $T_{inl}$ is typically no more than one logic gate delay.

Third, data at the receiver’s input must remain stable for at least $T_{hl}$ after the deassertion of $en_r$, i.e.,

$$T_7 + T_8 + T_9 + T_{min} \geq T_{hl}$$

(2.5)

where $T_{hl}$ is the hold time of latches, $T_{min}$ is the minimum delay of the path $Z_1$, $T_8$ is the
falling transition delay from the pin $a_i$, to the pin $a_o$, and $T_7$ and $T_9$ are internal delays within the handshake components. This constraint is easily satisfied because $T_{h1}$ is typically no more than one logic gate delay.

We now consider the case where the receiving pipeline stage is implemented by positive-edge triggered flip-flops. The timing constraints are as follows. First, valid data must arrive at the receiver $T_{su2}$ before the assertion of $en_r$, i.e.,

$$T_a \geq \max\{(T_{max1} + T_{su2} - T_1 - T_2), (T_{max2} + T_{su2} - T_3 - T_4 - T_5 - T_{rz} - T_6 - T_7 - T_8 - T_9 - T_{x2})\} \quad (2.6)$$

where $T_{su2}$ is the setup time of flip-flops, and $T_{max2}$ is the maximum delay of the path $Z_2$ (see Fig. 2.28(b)), which originates at the receiver’s latch-enable pin $en_r$, goes through the receiver’s clock tree, the receiver’s flip-flops, and the combinational logic between the output and input data pins of the flip-flops, and terminates at the input data pins of the flip-flops.

The second operand of the maximum function in (2.6) ensures that the latency between two consecutive data registrations by the flip-flops is longer than the maximum delay in the feedback path from the outputs to the inputs of the flip-flops. It is of interest to note that this constraint is somewhat similar to the fundamental mode timing assumption of burst-mode machines [46], where inputs are assumed to be stable until a circuit has stabilized. However, in this work, (2.6) is not merely an assumption, but a constraint that can be readily satisfied by choosing an appropriate value for $T_a$.

Second, the receiver’s input data must remain stable for at least $T_{h2}$ after the assertion of $en_r$, i.e.,

$$T_{h2} \leq \min\{(T_{min2}), (T_3 + T_4 + T_5 + T_{rz} + T_6 + T_7 + T_8 + T_9 + T_{min})\} \quad (2.7)$$
where $T_{h2}$ is the hold time of flip-flops, and $T_{min2}$ is the minimum delay of the path $Z_2$. This constraint is easily satisfied because $T_{h2}$ is typically no more than one logic gate delay.

Finally, implementing a pipeline stage by flip-flops requires that the local clock signal be properly distributed so that

\[ T_{skew} < T_{min3} \]  

(2.8)

where $T_{skew}$ is the local clock skew and $T_{min3}$ is the minimum delay of the receiver’s feedback path, which originates at the clock pins of the receiver’s flip-flops, goes through the combinational logic between the output and input data pins of the flip-flops, and terminates at the input data pins of the flip-flops. Unlike the global clock skew problem in synchronous circuits, the satisfaction of (2.8) is not difficult because the clock is local to the pipeline stage and, thus, drives only the flip-flops within the pipeline stage.

2.8. Testing

The testing of asynchronous circuits is not within the scope of this thesis. However, given the importance of this subject, it will be helpful to the reader if a brief introduction to the testing of asynchronous circuits is provided. The focus in this section is on testing methods for asynchronous pipelines based on the bundled-data convention because the resulting architecture of the proposed synthesis method is in effect that of a bundled-data asynchronous pipeline.

The testing of bundled-data asynchronous pipelines can be classified into two areas – the testing of the control network and the testing of the datapath. A good treatment of this subject is given in [110][111].
As discussed in Chapter 2 of the thesis, the control network of an asynchronous pipeline is an interconnection of handshake components, which can be implemented using standard logic gates and standard two-input C-gates. While conventional automatic test pattern generation methods designed for combinational logic circuits can be used to generate test patterns for the standard logic gates in the control network, the same is not true for the C-gates, which are sequential elements whose states depend on a potentially unknown number of previous inputs. It is suggested in [110][111] that a C-gate be treated as a finite state machine so that its behavior can be captured in the form of a state diagram, from which a checking sequence can be constructed [112]. By applying the checking sequence on the C-gate, its functionality can be verified to be correct. This method for testing a single C-gate can be extended to test a group of C-gates that are interconnected in a tree or cascade structure. To test an entire control network, it is necessary to insert scan latches (the same as those used in synchronous circuits) into the network such that the output of every C-gate in the network is controllable. Typically, this involves inserting scan latches on the feedback paths in the control network.

Once controllability has been established in the control network through the insertion of scan latches, the testing of the datapath becomes relatively straightforward. This is because the pipeline latches in the datapath are all controlled by the latching signals from the control network. Thus, without adding additional hardware to the datapath itself, it is possible to perform tests on the datapath by controlling the latching of the pipeline latches using the control mechanism already established within the control network. Note also that the datapath of a bundled-data asynchronous pipeline is not substantially different from that of a synchronous pipelined circuit, thus allowing the testing methods developed for synchronous circuits to be applied directly to the
asynchronous datapath. For example, it is possible to apply the testing method reported in [113] for synchronous circuits on the datapaths of bundled-data asynchronous pipelines [110].

2.9. Design Examples

This section describes the implementation of an asynchronous Reed-Solomon error detector for the compact-disc player [24][67][90][114] and an asynchronous IFIR filter bank for digital hearing aids [26] using the proposed synthesis method. It also provides quantitative comparisons between the circuits implemented by the proposed synthesis method and those realized using desynchronization (specifically, the approach reported in [80]) and Pipefitter [64]. These reported methods are comparable to the proposed approach because they also follow the pipelined/dataflow approach and use existing RTL synthesis tools for datapath synthesis.

The circuits were implemented using the Austria Mikro System (AMS) 0.35 \textmu m CMOS standard cell library and standard-C elements. The datapaths were synthesized using Synopsys Design Compiler, and the energy dissipation and speed performances of the circuits (at the supply voltage of 3.3V) were analyzed using Synopsys Nanosim.

2.9.1. Reed-Solomon Error Detector

This section describes the modeling of an asynchronous Reed-Solomon error detector for the compact disc player [24][67][90][114] using the proposed modeling rules. It also describes the synthesis of the error detector’s asynchronous control network using the proposed synthesis method and the implementation of the error detector using the design flow as shown in Fig. 1.4. The Reed-Solomon error detector has been chosen as a design example for its appropriate size. On the one hand, it is large enough to
Fig. 2.30. Block diagram of the Reed-Solomon error detector for the compact disc player.

demonstrate many of the features in the proposed synthesis method. On the other hand, it is small enough to allow a detailed presentation of its design and implementation process.

The Reed-Solomon error detector for the compact disc player accepts codewords consisting of 28 or 32 eight-bit symbols, including four parity symbols. As shown in Fig. 2.30, the error detector comprises two main blocks. The syndrome computation block consists of a functional loop that inputs a codeword symbol-by-symbol and computes the four syndromes for the codeword. Once the syndromes are computed, they are transferred to the error detection block which determines the error values, if any, and their locations. The error detector is able to provide the error value and location if there is exactly one error in the codeword. It raises a flag if there are two or more errors in the codeword. Note that the two blocks operate concurrently, i.e., when the error detection block is computing the error in the $n$-th codeword, the syndrome computation block is computing the syndromes for the $(n + 1)$-th codeword. A detailed theoretical background on Reed-Solomon error detection is given in [114].

2.9.1.1. Modeling

The error detector is described in Verilog HDL using the proposed modeling rules as follows. The global reset signal is provided by input $rst$, which is assigned to the `async_set_reset` attribute:

\[
\text{(* synthesis, async_set_reset = "rst" *)}
\]
The above attribute is applied to module rserrdet, the top-level module, to indicate that rst shall be used to initialize the handshake components and variables that have an initial value. One such variable is sym_cnt, which is initialized to zero using the initial construct:

\[
\text{initial} \\
\quad \text{sym\_cnt} = 0;
\]

The size of the codeword is specified by input size, where "0" indicates a codeword size of 28 and "1" a codeword size of 32. Note that during syndrome computation, the value of size is stored in variable size_buf1, which has a passive output channel specified using the pull attribute:

\[
\text{(* async, pull *) reg size\_buf1;}
\]

This allows the codeword size to be read by computation blocks in the module.

The codeword is read one symbol at a time through input sym. The arrival of each symbol is implicitly represented by named event sym_event using the channel attribute:

\[
\text{(* async, channel = "sym" *) event sym\_event;}
\]

To keep track of the number of symbols received, sym_cnt is incremented each time sym_event is triggered:

\[
\text{always @(sym\_event)} \\
\quad \text{if ( (size\_buf1 \& sym\_cnt == 31) | (\neg size\_buf1 \& sym\_cnt == 27))} \\
\qquad \text{sym\_cnt <= 0;}
\]
\[
\text{else} \\
\qquad \text{sym\_cnt <= sym\_cnt + 1;}
\]

Note that sym_cnt is reset to zero once an entire codeword is read.

To compute the four syndromes \(\text{syn}[0]\), \(\text{syn}[1]\), \(\text{syn}[2]\), and \(\text{syn}[3]\) for the codeword, the syndrome variables are initialized to the value of the first symbol in the codeword, and Galois field additions and multiplications (described using the functions
gfadd and alpha, respectively) are applied on the syndrome variables and incoming symbols accumulatively:

```vhdl
always
    if (sym_cnt == 1)
        for (i = 0; i <= 3; i = i + 1)
            syn[i] <= sym;
    else begin
        syn[0] <= gfadd(alpha(syn[0]), sym);
        syn[1] <= gfadd(alpha(alpha(syn[1])), sym);
        syn[2] <= gfadd(alpha(alpha(alpha(syn[2]))), sym);
        syn[3] <=
            gfadd(alpha(alpha(alpha(alpha(syn[3])))), sym);
    end
```

Once the syndromes have been computed, they are transferred to the variables syn_buf[0], syn_buf[1], syn_buf[2], and syn_buf[3]. These variables serve as a buffering stage for the computed syndromes between the syndrome computation and error detection block. Having a buffering stage for the current set of computed syndromes is beneficial for the throughput performance of the error detector because it allows syndrome computation for the next codeword to proceed even if the error detection block is still working on the previous set of syndromes. The conditional transfer of data from syn to syn_buf is realized using a guarded channel:

```vhdl
always
    (* async, guard *) if (sym_cnt == 0)
        for (i = 0; i <= 3; i = i + 1)
            syn_buf[i] <= syn[i];
```

A while loop is used to determine if there are any errors in the codeword (note that the loop variables loopsyn[0], loopsyn[1], loopsyn[2], and loopsyn[3] are initialized to the computed syndromes):

```vhdl
for (i = 0; i <= 3; i = i + 1)
    loopsyn[i] = syn_buf[i];

while (~loopcnt[5] &
    loopcnt <= loopcnt-1;
    loopsyn[0] <= alpha(alpha(alpha(loopsyn[0])));
```
If the codeword contains no errors, which implies that all computed syndromes are zeros, then the while loop is not entered and zero is sent out through the error output err. If there is exactly one error in the codeword, then the while loop iterates $m - n$ times before terminating, where $m$ is the codeword size and $n$ is the location of the error. Note that the loop variable loopcnt, which is decremented in each iteration of the loop, holds the error location after the loop has terminated, and the error value is given by syn_buf0. The error location and value are output through the output ports errloc and err, respectively. If the codeword contains more than one error, then the while loop iterates $m$ times before terminating and a flag is raised by communicating a logic “1” through the output port stat.

The Verilog HDL model of the error detector in its entirety is as follows.

Example 2.12:

(* synthesis, async_set_reset = "rst" *)
module rserrdet (rst, size, sym, stat, err, errloc);

input rst;
input size;
input [7:0] sym;

output stat;
output [7:0] err;
output [4:0] errloc;

(* async, pull *) reg size_buf1;
reg size_buf2;
reg [4:0] sym_cnt;
reg [7:0] syn [3:0];
reg [7:0] syn_buf [3:0];
reg [7:0] loopsyn [3:0];
reg [5:0] loopcnt;
reg [7:0] err;

integer i;

(* async, channel = "sym" *) event sym_event;
// Galois field addition

function [7:0] gfadd;
  input [7:0] in0, in1;
  gfadd = in0^in1;
endfunction

// Galois field multiplication

function [7:0] alpha;
  input [7:0] in;
  alpha[7] = in[6];
  alpha[6] = in[5];
  alpha[5] = in[4];
  alpha[2] = in[1] \land in[7];
  alpha[1] = in[0];
  alpha[0] = in[7];
endfunction

initial
  sym_cnt = 0;

always begin
  size_buf1 <= size;
  size_buf2 <= size;
end

always @ sym_event
  if ((size_buf1 & sym_cnt == 31) | (~size_buf1 & sym_cnt == 27))
    sym_cnt <= 0;
  else
    sym_cnt <= sym_cnt + 1;

always
  if (sym_cnt == 1)
    for (i = 0; i <= 3; i = i + 1)
      syn[i] <= sym;
  else begin
    syn[0] <= gfadd(alpha(syn[0]), sym);
    syn[1] <= gfadd(alpha(alpha(syn[1])), sym);
    syn[2] <= gfadd(alpha(alpha(alpha(syn[2]))), sym);
    syn[3] <= gfadd(alpha(alpha(alpha(alpha(syn[3])))), sym);
  end

always
  /* async, guard */ if (sym_cnt == 0)
    for (i = 0; i <= 3; i = i + 1)
      syn_buf1[i] <= syn[i];

always begin
  if (~size_buf2)
    loopcnt = 27;
  else
    loopcnt = 31;
for (i = 0; i <= 3; i = i + 1)
    loopsyn[i] = syn_buf[i];

while (~loopcnt[5] &
    loopcnt = loopcnt-1;
    loopsyn[0] = alpha(alpha(alpha(loopsyn[0])));
    loopsyn[1] = alpha(alpha(loopsyn[1]));
    loopsyn[2] = alpha(loopsyn[2]);
end

stat = loopcnt[5];
errloc = loopcnt;
end

always
err <= syn_buf[0];
endmodule

2.9.1.2. Asynchronous Control Network

The proposed synthesis method is used to synthesize the asynchronous control network of the error detector and establish the network's control over its datapath. Fig. 2.31 shows the asynchronous control network synthesized by the proposed synthesis method. The operation of the control network is explained as two phases: syndrome computation and error detection.

In the syndrome computation phase, the codeword size is first sent to the error detector on the size channel, whose handshake signals are size_req and size_ack. The codeword size is stored in variable size_buf1, whose data storage operations are controlled by the SyncPassiveOut handshake component P6. A SyncPassiveOut is used because size_buf1 has a passive output channel. The codeword is sent to the error detector one symbol at a time on the sym channel, whose handshake signals are sym_req and sym_ack. As shown in Fig. 2.31, sym_req is broadcast to the handshake components A0, S0, S1, S2, and S3. A0 is a SyncActiveIn that controls the storage of
sym_cnt. As shown in Fig. 2.32, $A_0$, upon receiving a request on the sym channel (sym_req↑), communicates with $P_0$ (size_buf1_req↑, size_buf1_ack↑, size_buf1_req↓, size_buf1_ack↓) so that the value of size_buf1 can be accessed. It then asserts its latch-enable output (sym_cnt_en↑) to increment sym_cnt. Subsequently, $A_0$ broadcasts a request (sym_cnt_req↑) to $S_0$, $S_1$, $S_2$, and $S_3$, the Sync handshake components that control the storage of syn[0], syn[1], syn[2], and syn[3], respectively.

---

Fig. 2.31. The asynchronous control network synthesized by the proposed synthesis method for the Reed-Solomon error detector.
Fig. 2.32. Waveforms obtained from HDL simulation of the compiled design for the syndrome computation phase.

Once $S_0$, $S_1$, $S_2$, and $S_3$ receive a request from $A_0$, they assert their respective latch-enable outputs ($syn0\_en^\uparrow$, $syn1\_en^\uparrow$, $syn2\_en^\uparrow$, and $syn3\_en^\uparrow$, respectively) to register the new values of $syn[0]$, $syn[1]$, $syn[2]$, and $syn[3]$. $A_0$ also broadcasts its output request to $G_0$, $G_1$, $G_2$, and $G_3$, the SyncGuard handshake components that control the storage of $syn\_buf[0]$, $syn\_buf[1]$, $syn\_buf[2]$, and $syn\_buf[3]$, respectively.

For each request that $A_0$ generates, $G_0$, $G_1$, $G_2$, and $G_3$ sample the evaluation result $sym\_buf0\_g$ of the guard expression "$sym\_cnt == 0$" to determine if an entire codeword has been read. If so (i.e., $sym\_buf0\_g$ is "$1$"), then $G_0$, $G_1$, $G_2$, and $G_3$ assert their respective latch-enable outputs ($sym\_buf0\_en^\uparrow$, $sym\_buf1\_en^\uparrow$, $sym\_buf2\_en^\uparrow$, and $sym\_buf3\_en^\uparrow$, respectively) to register the computed syndromes. Otherwise (i.e., $sym\_buf0\_g$ is "$0$"), $G_0$, $G_1$, $G_2$, and $G_3$ merely complete the handshakes on their input.
channels. The above steps are repeated for each symbol of the codeword, except for the communication on the size channel, which is performed just once for each codeword.

The error detection phase of the control network’s operation is initiated by the requests sent by $G_0$, $G_1$, $G_2$, and $G_3$ (syn_buf0_req↑, syn_buf1_req↑, syn_buf2_req↑, and syn_buf3_req↑, respectively) to $LF_0$, the LoopFront handshake component that implements the while loop for error detection together with the LoopEnd handshake component $LE_0$.

As shown in Fig. 2.33, Upon receiving requests from $G_0$, $G_1$, $G_2$, and $G_3$, $LF_0$ asserts its latch-enable output (loopcnt_buf_en↑) to initialize the loop buffers loopsyn0_buf, loopsyn1_buf, and loopsyn2_buf to the first three computed syndromes, and loopcnt_buf to 27 (note that the data-select output loopcnt_buf_s of $LF_0$ is “0” at this time). $LF_0$ then sends a request (loopcnt_buf_req↑) on the loop evaluation channel.
to $LE_0$. If another iteration is required (i.e., the looping condition $\text{whileloop0\_condition}$ is evaluated to be "1"), then $LE_0$ sends a request ($\text{loopcnt\_it\_req}$) on the loop iteration channel to $LF_0$. Otherwise (i.e., $\text{whileloop0\_condition}$ is evaluated to be "0"), $LE_0$ sends a request to the output ports $\text{stat\_req}$ and $\text{errloc\_req}$ to indicate to the external environment that the error detector has finished processing a codeword. Note that in both cases, $LE_0$ will assert its latch-enable output ($\text{loopcnt\_en\uparrow}$) to register the new values of the loop variables $\text{loopsyn[0]}$, $\text{loopsyn[1]}$, $\text{loopsyn[2]}$, and $\text{loopcnt}$. Note also that if an iteration is required in the loop, then the values stored in the loop variables are transferred on the iteration channel to the loop buffers.

### 2.9.1.3. Comparisons with Other Methods

To validate the efficacy of the proposed approach, the error detector implemented by the proposed approach is compared with those implemented by desynchronization (specifically, the method reported in [80]) and Pipefitter [64].

Since the desynchronization method reported in [80] is not freely available, a synchronous (i.e., clocked) RTL version of the error detector was first created and compiled by Synopsys Design Compiler into a gate-level netlist. The gate-level netlist was then manually transformed into the desynchronized version (RS2) of the error detector. In essence, this involved splitting each flip-flop into two latches, instantiating a handshake component to control each of the latches, interconnecting the handshake components with handshake signals, and removing the global clock. The Pipefitter tools, on the other hand, are freely available and were used in the implementation of the Pipefitter version (RS3) of the error detector. The Pipefitter tools were used to generate the STG specifications for the asynchronous controllers and the RTL models of the datapaths. The STG specifications were then fed into Petrify to generate the circuit


**Table 2.1**
Comparisons of Reed-Solomon Error Detectors Realized Using Different Asynchronous Implementation Styles

<table>
<thead>
<tr>
<th>$N_T$ Syndrome Computation</th>
<th>HC</th>
<th>LEB</th>
<th>DE</th>
<th>Total</th>
<th>Datapath</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Network</td>
<td>390</td>
<td>762</td>
<td>785</td>
<td></td>
<td>4,523</td>
</tr>
<tr>
<td>LEB</td>
<td>16</td>
<td>28</td>
<td>8</td>
<td></td>
<td>4,382</td>
</tr>
<tr>
<td>DE</td>
<td>56</td>
<td>80</td>
<td>88</td>
<td></td>
<td>4,420</td>
</tr>
<tr>
<td>Total</td>
<td>462</td>
<td>870</td>
<td>881</td>
<td></td>
<td>4,420</td>
</tr>
<tr>
<td>Datapath</td>
<td>4,523</td>
<td>4,382</td>
<td>4,420</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$E (nJ)$ Error Detection</th>
<th>HC</th>
<th>LEB</th>
<th>DE</th>
<th>Total</th>
<th>Datapath</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Network</td>
<td>258</td>
<td>224</td>
<td>410</td>
<td></td>
<td>1,645</td>
</tr>
<tr>
<td>LEB</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td></td>
<td>1,766</td>
</tr>
<tr>
<td>DE</td>
<td>16</td>
<td>16</td>
<td>56</td>
<td></td>
<td>1,852</td>
</tr>
<tr>
<td>Total</td>
<td>278</td>
<td>248</td>
<td>472</td>
<td></td>
<td>1,852</td>
</tr>
<tr>
<td>Datapath</td>
<td>1,645</td>
<td>1,766</td>
<td>1,852</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$E (nJ)$ Syndrome Computation</th>
<th>HC</th>
<th>LEB</th>
<th>DE</th>
<th>Total</th>
<th>Datapath</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Network</td>
<td>0.67</td>
<td>1.30</td>
<td>1.87</td>
<td></td>
<td>6,908</td>
</tr>
<tr>
<td>LEB</td>
<td>0.09</td>
<td>0.19</td>
<td>0.17</td>
<td></td>
<td>7,266</td>
</tr>
<tr>
<td>DE</td>
<td>0.11</td>
<td>0.24</td>
<td>0.19</td>
<td></td>
<td>7,625</td>
</tr>
<tr>
<td>Total</td>
<td>0.87</td>
<td>1.73</td>
<td>2.23</td>
<td></td>
<td>7,625</td>
</tr>
<tr>
<td>Datapath</td>
<td>6,908</td>
<td>7,266</td>
<td>7,625</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$E (nJ)$ Error Detection</th>
<th>HC</th>
<th>LEB</th>
<th>DE</th>
<th>Total</th>
<th>Datapath</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Network</td>
<td>0.15</td>
<td>0.66</td>
<td>0.40</td>
<td></td>
<td>2.87</td>
</tr>
<tr>
<td>LEB</td>
<td>0.07</td>
<td>0.08</td>
<td>0.06</td>
<td></td>
<td>4.27</td>
</tr>
<tr>
<td>DE</td>
<td>0.02</td>
<td>0.09</td>
<td>0.08</td>
<td></td>
<td>4.73</td>
</tr>
<tr>
<td>Total</td>
<td>0.23</td>
<td>0.83</td>
<td>0.54</td>
<td></td>
<td>4.73</td>
</tr>
<tr>
<td>Datapath</td>
<td>2.87</td>
<td>4.27</td>
<td>4.73</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Codewords per Sec ($10^6$) | 4.72| 5.18| 2.08|       |
| $E \tau^2 (10^{-22} J s^2)$ | 1.29| 1.59| 10.9|       |
| $N_T \tau (10^3 s)$        | 1.46| 1.40| 3.67|       |

HC: handshake components  
LEB: latch-enable buffers  
DE: delay elements  
$N_T$: transistor count  
$E$: energy dissipation per code word  
$\tau$: processing time per codeword

Implementations for the asynchronous controllers, and the RTL models were compiled by Synopsys Design Compiler to generate the circuit implementations for the datapaths.

Table 2.1 compares the circuits implemented using the proposed approach (RS1), desynchronization (RS2), and Pipefitter (RS3). The energy dissipation estimations are based on randomly generated codewords, each having with one erroneous symbol. The errors are uniformly distributed among the symbols.
RS1 is first compared with RS2. Despite having only 5% less transistors than RS2, RS1 dissipates 33% less energy than RS2 per codeword. This is due to a smaller control network in its syndrome computation block and a more energy efficient control network in the error detection block.

More specifically, the control network of RS1’s syndrome computation block has 47% less transistors than that of RS2 due mainly to its smaller number of handshake components – RS1 uses 11 handshake components to control the 11 variables in its syndrome computation block, whereas RS2 uses 24 handshake components to control 12 variables (note that in desynchronization, a variable is always implemented as two sets of latches [80], and in RS2 one latch controller is used to control each set of latches (see [62] for more sophisticated latch-clustering methods)). As a result, the control network of RS1’s syndrome computation block dissipates 50% less energy than that of RS2.

The control network of RS1’s error detection block is more energy efficient than that of RS2 due to conditional communication – it operates only when error detection is ongoing and becomes idle once the error location has been found (in the case of correct codewords, it is completely idle). In contrast, the control network of RS2’s error detection block operates continuously (however, clock gating is used to prevent unnecessary energy dissipation in the datapath). Consequently, the control network of RS1’s error detection block dissipates 72% less energy than that of RS2, despite having 12% more transistors (the higher transistor count is due to the use of the LoopFront and LoopEnd handshake components, which are significantly more complex than basic latch controllers, to implement a while loop).

In terms of number of codewords per second, RS1 is 9% slower than RS2. This is due to the higher complexity in some of the handshake components in RS1 compared
with basic latch controllers in RS2. However, the energy-delay complexity $E^2$ \cite{115,118}, where $E$ is the average energy dissipation per codeword and $\tau$ is the time taken to process one codeword, of RS1 is 19% lower (i.e., better) than that of RS2. Note that $E^2\tau$ is, in the first order of approximation, independent of the supply voltage. A better $E^2\tau$ for RS1 indicates that if the supply voltage of RS1 is raised to improve its speed (i.e., to decrease $\tau$) so as to match the speed of RS2, the energy dissipation of RS1, though also increased, would remain less than that of RS2. When the area-delay figure of merit $NT\tau$ is considered, RS1 and RS2 are comparable.

RS1 is now compared with RS3. RS1 has 9% less transistors, dissipates 39% less energy, and is 2.3x faster than RS3. The improvements are due to RS1’s smaller (45%) control networks, which dissipates 61% less energy than those of RS3. The significant differences in transistor count and energy dissipation between the asynchronous control networks in RS1 and RS3 are due in part to the large number of internal state signals inserted in the latter’s control networks to resolve the complete-state-coding problem. This highlights the potential disadvantages in using monolithic asynchronous controllers as opposed to networks of distributed controllers.

A synchronous version of the error detector was also implemented and compared with RS1. As shown in Table 2.2, RS1 has 8% more transistors than the synchronous detector, due to its control network. This compares favorably with the circuit area overheads of asynchronous circuits that are reported in the literature, which are typically at least 20% \cite{21,26,67}. In addition, as it shall be shown in Chapter 3 (see Section 3.5.4.3), RS1’s control network can be reduced through optimization by 34%, bringing the circuit overhead down to 4%. In terms of energy dissipation, the synchronous detector is 18% better than RS1. This is not unexpected due to three reasons. First, the error detection block of the synchronous detector is clock gated so that computations
within the block are terminated once the location of the error in the word has been computed (similar to the technique used in RS1). This means that the power dissipation of the synchronous circuit would be very close to that of RS1’s datapath, if its clock distribution network is not taken into account. Second, the control network of RS1 is unoptimized and contributes 38% to the total energy dissipation. However, as it shall be discussed in Section 3.5.4.3, the energy dissipation of RS1’s control network can be reduced by 32% through optimization, bringing its energy overhead with respect to the synchronous detector down to 7%. Third, with only 75 flip-flops in the synchronous detector, its clock distribution network is fairly small and contributes only 6% to the total energy dissipation. For large designs, however, the clock distribution networks would result in much larger power overheads, typically dissipating more than 20% of the total power [52][116][117].

2.9.2. IFIR Filter Bank

The purpose of the IFIR filter bank in digital hearing aids is to split the input signal into seven frequency bands on which further signal processing, such as amplification, is performed [26]. The filter bank accounts for approximately half of the signal processing circuitry in digital hearing aids. As shown in Fig. 2.34, the filter bank consists of a tree

---

Table 2.2
Comparisons of Synchronous and Asynchronous Reed-Solomon Error Detectors

<table>
<thead>
<tr>
<th></th>
<th>(N_T(k))</th>
<th>(E(\text{nJ}))</th>
<th>(\tau(\text{ns}))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Control</td>
<td>Total</td>
<td>Control</td>
</tr>
<tr>
<td>Synchronous</td>
<td>-</td>
<td>6.38</td>
<td>-</td>
</tr>
<tr>
<td>RS1</td>
<td>0.74</td>
<td>6.91</td>
<td>1.10</td>
</tr>
</tbody>
</table>

\(N_T\): transistor count
\(E\): av. energy per codeword
\(\tau\): processing time per codeword
The architecture of an IFIR filter bank for digital hearing aids is illustrated in Fig. 2.34. Each filter (see Fig. 2.35) consists of an add-multiply-accumulate (AMA) unit, a RAM for the input samples, a ROM for the filter coefficients, and an address generator whose main responsibility is to generate the memory addresses and coordinate the activities within the filter. The order of the IFIR filters is as follows: H1 - 96, H2 - 48, H3 - 56, H4 - 24, H5 - 20, H6 - 12, H7 - 60, H8 - 4, and H9 - 4.

Table 2.3 compares the circuits implemented using the proposed approach (FB1), desynchronization (FB2), and Pipefitter (FB3). The energy estimations are based on 2,000 randomly generated input samples. Note that the filter banks process only one input sample at any point in time due to the low input sampling rate of 20,000 samples per second, which is equivalent to an interval of 50 μs between the arrival of two input samples. Given that the worst latency of the three filter banks implemented is 203 ns (see Table 2.3), it is clear that in a real-life situation only one input sample would be processed in the filter banks at any point in time.

FB1 is first compared with FB2. Although FB1 and FB2 have similar transistor counts, FB1’s control network dissipates 66% less energy than that of FB2. This is
Table 2.3
Comparisons of IFIR Filter Banks Realized Using Different Implementation Styles

<table>
<thead>
<tr>
<th>$N_T$ (k)</th>
<th>FB1</th>
<th>FB2</th>
<th>FB3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Network</td>
<td>HC</td>
<td>6.79</td>
<td>6.44</td>
</tr>
<tr>
<td></td>
<td>LEB</td>
<td>0.38</td>
<td>0.41</td>
</tr>
<tr>
<td></td>
<td>DE</td>
<td>0.79</td>
<td>0.86</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>7.96</td>
<td>7.71</td>
</tr>
<tr>
<td>Execution AMA/Addr. Gen.</td>
<td>36.3</td>
<td>40.2</td>
<td>41.5</td>
</tr>
<tr>
<td></td>
<td>RAM/ROM</td>
<td>236</td>
<td>236</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>272</td>
<td>276</td>
</tr>
<tr>
<td>Total</td>
<td>280</td>
<td>284</td>
<td>289</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$E$ (nJ)</th>
<th>FB1</th>
<th>FB2</th>
<th>FB3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Network</td>
<td>HC</td>
<td>1.38</td>
<td>4.41</td>
</tr>
<tr>
<td></td>
<td>LEB</td>
<td>0.18</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>DE</td>
<td>0.20</td>
<td>0.62</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>1.76</td>
<td>5.22</td>
</tr>
<tr>
<td>Execution AMA/Addr. Gen.</td>
<td>2.84</td>
<td>3.03</td>
<td>3.05</td>
</tr>
<tr>
<td></td>
<td>RAM/ROM</td>
<td>1.24</td>
<td>1.18</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>4.08</td>
<td>4.21</td>
</tr>
<tr>
<td>Total</td>
<td>5.83</td>
<td>9.43</td>
<td>6.80</td>
</tr>
</tbody>
</table>

| Latency per Sample (ns) | 125 | 107 | 203 |
| $E \tau^2 (10^{22} \text{Js}^2)$ | 0.91 | 1.08 | 2.80 |
| $N_T \tau (10^3 \text{s})$ | 35 | 30 | 59 |

HC: handshake components
LEB: latch-enable buffers
DE: delay elements
$E$: energy dissipation per computation
$\tau$: processing time per computation
$N_T$: transistor count

because in FB1 the control network of a filter is idle when the filter is not computing. On the other hand, in FB2, a filter’s control network continues to operate even when the filter is not computing (the datapath is, however, idle through clock gating). Since each sample goes through four “levels” of processing in the filter bank (see Fig. 2.35), it can be seen that only 25% of the energy dissipated by FB2’s control network is useful. Overall, FB1 dissipates 38% less energy than FB2. However, FB2 is faster than FB1 due to the less complex handshake components in its control network – its latency per sample is 14% shorter than that of FB1. Nevertheless, the $E \tau^2$ of FB1 is 15% lower than that of FB2.
FB1 is now compared with FB3. FB1's control network has 31% less transistors and dissipates 28% less energy than that of FB3, due to the large number of internal state signals inserted in the latter to resolve the CSC problem. Overall, FB1 is only marginally (3%) smaller than FB3 but still enjoys a 14% advantage in energy dissipation. In addition, the latency of FB1 is 38% shorter than that of FB3.

2.10. Summary

A synthesis method that facilitates the design of asynchronous pipelines with low asynchronous control overheads has been proposed, developed, and automated. More specifically, the following work on the modeling and synthesis of asynchronous pipelines has been described.

First, a coarse-grain approach that is employed by the proposed synthesis method to synthesize asynchronous control networks has been described. The approach reserves asynchronous control to the implementation of essential asynchronous operations, thus facilitating the design of asynchronous pipelines with low asynchronous control overheads in terms of circuit area and power dissipation.

Second, a set of modeling rules for asynchronous communication that is supported by the proposed synthesis method has been described. The modeling rules are based on conventional Verilog HDL constructs, on which additional semantics are imposed during synthesis to infer asynchronous communication. This means that the use of special packages or subroutines for asynchronous communication modeling is avoided.

Third, the process of synthesizing the asynchronous control network of the design specification and establishing the control of the asynchronous control network over the design's datapath has been described. In particular, the three main tasks performed during the synthesis process – asynchronous communication channel extraction,
handshake component inference, and initial state computation for asynchronous control networks – have been described in detail.

Fourth, a method for computing a live initial state for an asynchronous control network has been proposed. The proposed method has been formally proven to preserve the nondeadlock behavior of the control network resulting from its specified initial state.

Fifth, the efficacy of the proposed synthesis method has been demonstrated through the design of an asynchronous Reed-Solomon error detector and an asynchronous IFIR filter bank.

Compared with the error detector implemented by desynchronization, that implemented by the proposed synthesis method is found to dissipate 33% less energy. Although the error detector implemented by the proposed synthesis method is 9% slower than the desynchronized circuit, it is 19% better in $E_t^2$. Compared with the error detector implemented by Pipefitter, the circuit implemented by the proposed synthesis method has 9% less transistors, dissipates 39% less energy, and is 2.3x faster.

Compared with the filter bank implemented by desynchronization, the circuit implemented by the proposed synthesis method dissipates 38% less energy and is 15% better in $E_t^2$. Compared with the filter bank implemented by Pipefitter, the circuit implemented by the proposed synthesis method enjoys a 14% advantage in energy dissipation and has a latency that is 38% shorter.
3
Optimization of Asynchronous Control Networks

3.1. Introduction
In Chapter 2, a synthesis method was proposed that facilitates the design of asynchronous pipelines with low control overheads relative to circuits implemented using the CSP-based methods. The proposed synthesis method is able to generate relatively smaller asynchronous control networks largely because it avoids syntax-directed translation and adopts a coarse-grain approach towards the implementation of asynchronous control.

However, even with such a minimalist approach towards asynchronous control network synthesis, there remain ample opportunities for optimizing the asynchronous control networks generated by the proposed synthesis method.

For example, meeting the objectives of clarity and readability when writing high-level (including RTL) descriptions for VLSI systems often leads to the introduction of redundancies into the descriptions. This can lead to asynchronous control networks that are larger and dissipate more power than necessary.
For instance, consider the Reed-Solomon error detector model described in Chapter 2. For reasons of clarity and readability, it is preferable to use four eight-bit variables, as in \text{syn[0]}, \text{syn[1]}, \text{syn[2]}, and \text{syn[3]}, rather than to use one 32-bit variable to store the four syndromes that are computed for each codeword. However, the latter has the advantage of requiring just one handshake component whereas the former requires four, one to control each variable. Furthermore, the simultaneous computation of the four syndromes suggests that using a single handshake component to control the computation would not incur a loss in throughput for the error detector.

In this chapter, two optimization methods — \textit{handshake component fusion} and \textit{optimal decoupling} — for reducing the circuit areas and power dissipation of asynchronous control networks while satisfying given pipeline throughput constraints are proposed (this work has been accepted for publication in \textit{IEEE Trans. Computer-Aided Design} [120]). This chapter is organized as follows.

This chapter starts by reviewing the existing optimization methods for asynchronous circuits and comparing the existing methods with the proposed methods.

For the proposed optimization method of handshake component fusion, a fork structure is used as a simple example to demonstrate the main ideas of the method. This is followed by a description of the proposed heuristic algorithm for optimization target selection. Finally, the issues of satisfying the throughput constraint and preserving the behavior of the asynchronous control network under optimization are discussed.

For the proposed optimization method of optimal decoupling, a simple example consisting of a three-stage linear asynchronous pipeline is used to illustrate the main ideas of the method. This is followed by a description of the proposed branch-and-bound algorithm that searches for the optimum mix of handshake components of different degree of concurrency that satisfies a throughput constraint.
The effectiveness of the proposed optimization methods are demonstrated by applying them on the asynchronous control networks of three designs: a 16-input pipelined parallel prefix tree, a four-bit cross-pipelined array multiplier, and a Reed-Solomon error detector.

3.2. Literature Review

This section reviews the reported methods for optimizing asynchronous control networks.

3.2.1. Control Resynthesis

Control resynthesis involves grouping interconnected handshake components into blocks, composing the behavior of each block into a graph-based specification, such as an STG or a burst-mode specification, and resynthesizing the specification into a monolithic control component [66][121][122]. By removing the redundant internal communication channels and handshake components within a block, control resynthesis can lead to one or several macro components that are smaller, less power dissipating, or faster than their network counterpart.

However, most reported algorithms for grouping handshake components either combine all handshake components into a single control entity [66][122] or attempt to form as large blocks as possible [121], resulting in large controller specifications that are difficult and time consuming to resynthesize. In particular, the synthesis of an STG into a circuit requires the enumeration of its entire state space (the state-explosion problem [64][90][92][123][124][125]) and uniquely encoding every reachable state of the controller (the complete-state-coding problem [92][123]). This means that even for advanced synthesis tools, such as Petrify [43], the time taken to synthesize large circuits
may be prohibitively long [64][92]. Another problem with control resynthesis is that the steps involved for composing the block specification are computationally expensive [66].

3.2.2. Peephole Optimization

Peephole optimization (also referred to by some authors as *clustering*) [61][63][72][126], in contrast, does not involve any resynthesis of control components. A technique that is similar to optimization schemes used in software compilers, peephole optimization involves iteratively looking at small regions of a circuit and replacing a predefined pattern of handshake components by another configuration of handshake components such that the resulting circuit is smaller, faster, or dissipates less power.

3.2.3. Designing Small Handshake Components

Another way to reduce the circuit areas and power dissipation of asynchronous control networks is to keep the handshake components small.

To date, the smallest handshake component reported is a basic latch controller consisting of just one XOR-gate [127]. However, this latch controller has timing assumptions that affect its robustness. Furthermore, its applicability is limited as it can only be used to build pipelines with forks and joins, whereas nontrivial asynchronous systems usually has a variety of handshake components that support their functions.

GasP is another way of designing small and fast handshake components [128][129]. However, GasP cannot be implemented by standard cell libraries because the correct operation of GasP handshake components relies on equalizing the delays in some logic gates through careful choice of transistor widths. Furthermore, GasP has mainly been
designed to control first-in-first-outs (FIFOS) and is not suitable for information-processing pipelines.

A well-established way of designing robust handshake components (and asynchronous circuits in general) is to specify their behaviors using STGs (see Chapter 2, Section 2.3.2.1) and synthesize the STGs into speed-independent circuits. A speed-independent circuit \[8\] is one whose functionality is independent of logic gate delays but dependent on the assumption of negligible wire delays. To keep the handshake components small, one can impose on their STG specifications strong dependencies between the input and output handshake signal transitions.

For example, Fig. 3.1(a) shows the STG specification of a Sync handshake component (a basic latch controller) whose signal transitions are \textit{minimally concurrent}. Note that some of the signal transition dependencies are not necessary for correct functionality, such as that between \textit{ri-} and \textit{ro+} (implied by the transition chain \textit{ri-} → \textit{en-} → \textit{ro-}), but are imposed to reduce the state space of the STG and, thus, simplify the circuit implementation of the handshake component. Fig. 3.1(b) shows the circuit implementation of the minimally-concurrent Sync, which consists of only one C-gate.

It is, however, well known that using only minimally-concurrent handshake components in asynchronous pipelines leads to poor throughput and pipeline utilization \[102\].

For better pipeline throughput and utilization, one can use handshake components whose signal transitions are more concurrent, such as the \textit{semi-concurrent} and \textit{maximally-concurrent} Syncs (the latter is similar to the fully-decoupled long hold latch controller reported in \[102\]), as shown in Fig. 3.1(b) and Fig. 3.1(c), respectively. These circuits, however, are larger and dissipate more power. There is, therefore, a tradeoff between the concurrency (and, thus, size) of handshake components and pipeline utili-
Fig. 3.1. STG specifications and circuit implementations of Sync: (a) minimally-concurrent, (b) semi-concurrent, and (c) maximally-concurrent.

throughput performance. On the one hand, using minimally-concurrent handshake components reduces asynchronous control overheads but tends to lead to poor throughput. On the other hand, using handshake components of higher concurrency tends to improve the throughput but incurs larger asynchronous control overheads.

### 3.2.4. Other Methods

Other reported optimization methods for asynchronous pipelines include *minimal pipelining* [129] and *slack matching* [130][131][132].
Minimal pipelining addresses the problem of identifying the minimum number of pipeline stages required to satisfy a given throughput constraint, thereby implicitly minimizing area and power for a given performance.

Slack matching is a timing optimization method that determines the amount of buffering pipelining stages that must be added to each communication channel of a slack-elastic asynchronous system in order to reduce its minimum cycle time to a specified target. A slack-elastic asynchronous system is one such that an arbitrary amount of buffering can be added to any communication channel without affecting the correctness of the system.

### 3.3. Comparisons of Proposed Methods with Reported Methods

This section compares the proposed optimization methods with the reported methods for optimizing asynchronous circuits.

The proposed handshake component fusion method is a form of peephole optimization that iteratively selects two handshake components of the same type, termed optimization targets, that share input channel sources or output channel destinations and replaces them with a single component of the same type (in short, the optimization targets are said to be fused).

The fusion of the optimization targets not only reduces the number of handshake components in the asynchronous control network, but also merges the optimization targets’ separate input (output) communication channels of the same channel source (destination), thereby reducing the number of handshake signals.

The fusion is subject to the conditions that the restructured pipeline is behaviorally equivalent to the original one and has a minimum cycle time that satisfies the given timing constraint.
A heuristic algorithm is proposed for the selection of the optimization targets. The selected optimization targets at each optimization iteration are those that are judged by the algorithm to be least likely to cause a change in the minimum cycle time of the pipeline.

The key difference between the proposed handshake component fusion method and other reported peephole optimization methods is that it incorporates minimum cycle time analysis (see Chapter 4), which is beneficial for two reasons.

First, the selection of the optimization targets is not restricted by the need to preserve performance because minimum cycle time analysis is used to predict the effect of the fusion on performance. If the fusion of the optimization targets is predicted to violate the pipeline throughput constraint, then the fusion is abandoned. Thus, a greater degree of freedom in optimization target selection is allowed, which potentially leads to a better quality of result.

Second, a trade-off is now possible between the pipeline’s throughput, power dissipation, and circuit area during the optimization process. That is, the pipeline’s throughput performance can be sacrificed, to the extent of not violating design specification, to reduce the power dissipation and circuit area of its control network.

The proposed optimal decoupling method is a new optimization method that resolves the dilemma when designing asynchronous pipelines between using small handshake components to reduce asynchronous control overheads and satisfying throughput constraints.

Conventionally, asynchronous control networks are designed using handshake components of the same degree of concurrency. Abandoning this conventional approach gives the designer the freedom to mix handshake components of different degree of concurrency in an asynchronous control network.
The term decoupling refers to the process of identifying and eliminating certain signal transition dependencies in the STG of a handshake component, thereby increasing the STG's concurrency, in order to improve the pipeline throughput. In practice, decoupling is realized by replacing a handshake component with another (of the same type) that has a higher level of concurrency.

The proposed method produces an optimal solution by using a branch-and-bound algorithm that searches for the optimal mix of handshake components of different degree of concurrency, i.e., one that occupies the smallest circuit area and dissipates the least power while meeting the pipeline throughput timing constraint.

In short, the main idea behind the proposed method is to use as many low concurrency handshake components as possible, so as to reduce the circuit area and power dissipation overheads incurred by the asynchronous control network, while attempting to satisfy the throughput specification by selecting, where necessary, handshake components of higher concurrency.

The proposed optimal decoupling method is different from reported handshake protocol decoupling techniques [80][102][103] in two ways.

First, the reported techniques only consider asynchronous control networks that comprise handshake components of the same degree of concurrency. In contrast, the proposed method mixes handshake components of different degree of concurrency in one control network.

Second, the reported techniques only target throughput improvement. The proposed method, on the other hand, tries to minimize the circuit area and power dissipation penalties that are incurred when it is optimizing the pipeline throughput. Although the reported minimal pipelining method [129] also reduces circuit area and power
dissipation as much as possible for a given performance, it does not consider mixing handshake components of different degree of concurrency.

Finally, the reported minimal pipelining and slack matching methods [129][130][131][132] model asynchronous systems using marked graphs [87], which are a subclass of PNs that do not model choices. This implies that these methods support only deterministic asynchronous pipelines, i.e., those that do not make real-time decisions which govern the flow of data. In contrast, the proposed optimal decoupling method (and the handshake component fusion method) uses STGs to model asynchronous systems. Since STGs do support the modeling of choices, the proposed methods are applicable to nondeterministic pipelines.

### 3.4. Timed Petri Net Models

The steps for composing the PN model of an asynchronous control network are described in Section 2.6.3.2. To create the corresponding timed PN model, delays are associated with the places of the model.

In general, there are three types of delays that need to be modeled: active-request delays, RTZ-request delays, and handshake component internal delays.

The active-request delay of a channel refers to the time taken for a rising transition of the request signal of the channel to propagate across the channel and is given by the latency of the delay-matching element placed on the request wire. For example, the delay that is associated with the active-request place \((ro_1^+, ri_2^+)\) in Fig. 2.27(b) is given by the latency of the delay-matching element \(D\) in Fig. 2.27(a).

On the other hand, the RTZ-request delay of a channel is the time taken for a falling transition of the request signal to propagate across the channel. It is either the same as the active-request delay (for channels with symmetric delays) or the delay of a single
gate (for channels with asymmetric delays) (see Chapter 1, Section 1.1 for discussion on delay matching issues).

3.5. Handshake Component Fusion

This section describes the proposed handshake component fusion method. The method is summarized as follows (see Fig. 3.2).

At each iteration of the optimization process, a proposed optimization target selection algorithm is used to select two handshake components of the same type that share at least one input channel source (fork) or output channel destination (join).

The optimization targets are replaced by a single component of the same type (in short, the optimization targets are said to be fused) such that:

i) the replacement component has as its input channel sources all the input channel sources of the optimization targets;

ii) the replacement component has as its output channel destinations all the output channel destinations of the optimization targets;

iii) the replacement component controls all the storage devices of the optimization targets; and

iv) for each pair of channels that are fused, the delay of the resulting channel is given by the larger channel delay of the pair.

The fusion of the optimization targets is subject to the conditions of flow equivalence between the original and restructured pipelines, and minimum cycle time constraint satisfaction by the restructured pipeline. The optimization process terminates when no more fusion can be carried out.
3.5.1. Demonstration

A simple example is used to demonstrate the main ideas behind the proposed handshake component fusion method. Consider the asynchronous control network of Fig. 3.3(a), where the handshake components $S_2$ and $S_3$ share a common channel source at
component $S_1$. Fusing $S_2$ and $S_3$ leads to the following changes to the control network to yield the new control network shown in Fig. 3.3(b):

i) $S_2$ and $S_3$ are replaced with $S_6$;

ii) the output request $ro_1$ of $S_1$, which originally forked into two signals driving the C-gates $C_1$ and $C_2$, now drives only the C-gate $C_4$;

iii) the C-gate $C_3$, which joins the input acknowledges $ai_2$ and $ai_3$ of $S_2$ and $S_3$, respectively, is removed;

iv) the input requests $ri_4$ and $ri_5$ of $S_4$ and $S_5$, respectively, which originally were generated by $S_2$ and $S_3$, respectively, now originate from the same source, the output request $ro_6$ of $S_6$;

v) the input acknowledges $ai_4$ and $ai_5$ of $S_4$ and $S_5$, respectively, which originally drive different handshake components ($S_2$ and $S_3$), now join at the C-gate $C_5$ to drive the output acknowledge $ao_6$ of $S_6$;
vi) the delay elements $D_1$ and $D_2$ are fused into a single delay element $D_3$ such that $T_3 = T_1 + T_2$, where $T_i$ is the latency of $D_i$;

vii) the delay elements $D_3$ and $D_4$ are replaced with $D_5$ and $D_7$ such that $T_6 = T_4$ and $T_7 = T_3 - T_4$ (assuming $T_3 \geq T_4$); and

viii) the storage devices originally controlled separately by $S_2$ and $S_3$ are now all controlled by $S_6$.

### 3.5.2. Optimization Target Selection

This subsection describes the proposed heuristic algorithm for the selection of optimization targets. The main ideas behind the proposed algorithm are as follows.

It is known in PNs theory that the support of an $S$-invariant $y_k$ of a PN $N$ is both a siphon and a trap (referred to as a siphon-trap). A siphon is a nonempty subset of places $S$ in $N$ such that every transition in $N$ that has an output place in $S$ also has an input place in $S$. A trap is a (possibly empty) subset of places $Q$ in $N$ such that every transition in $N$ that has an input place in $Q$ also has an output place in $Q$.

The implication of $\| y_k \|$ being a siphon-trap is that it can be informally viewed as forming one or more directed loops in $N$ that “traverse” the modular PN models of various handshake components of the corresponding asynchronous control network. For brevity, these handshake components are said to be associated with $y_k$. From (2.1), it is intuitive to see that if a minimal-support $S$-invariant, say $y_1$, has a “cycle time” $\lambda_1$ that is equal to the minimum cycle time $\tau_{\text{min}}$ of $N$, then any fusion that is performed on the handshake components associated with $y_1$ would likely have an effect on $\tau_{\text{min}}$ because the fusion will result in changes in $N$, which may in turn modify $y_1$. On the other hand, for a minimal-support $S$-invariant, say $y_2$, that has a “cycle time” $\lambda_2$ that is much less than $\tau_{\text{min}}$, then it is reasonable to argue that it is less likely, compared with the former...
case, for a fusion that is performed on the handshake components associated with $y_2$ to have an effect on $\tau_{\text{min}}$.

The essence of the proposed algorithm is, therefore, to select as optimization targets handshake components that are associated with minimal-support S-invariants whose "cycle times" are the lowest amongst all minimal-support S-invariants of $N$.

The proposed algorithm is described in detail as follows:

1) Group the minimal-support S-invariants of $N$ according to their respective "cycle times" $\lambda_k$, where $\lambda_k = y_k^T D(A^*) x / y_k^T M_0$.

2) Rank the S-invariant groups according to their respective values of $\lambda_k$. That is, the group with the highest $\lambda_k$ is ranked 1, the group with the second-highest $\lambda_k$ is ranked 2, and so forth.

3) Arbitrarily select an S-invariant $y_k$ in the highest-ranked group.

4) While there exist handshake components in the asynchronous control network that have not been assigned scores, do the following:

4.1) For each place $p_i \in \|y_k\|$, assign the rank of the current group as the score to any handshake component whose modular PN model contains $p_i$, if the handshake component has not already been assigned a score.

4.2) Select another S-invariant in the current group. If all S-invariants in the current group have been selected, move on to the next highest-ranked group.

5) Compute the score for each fork and join in the pipeline by summing the scores of its components.

6) Select as optimization targets the fork or join that has the highest score.
In the above description of the algorithm, the score of a handshake component indicates the magnitude of the difference between $\lambda$ of the highest-ranked $S$-invariant and that of the $S$-invariant which is associated with the handshake component. Since the fusion of the optimization targets will modify only the invariants associated with the optimization targets, it can be argued that if the fork or join with the highest combined handshake component score is selected as optimization targets, then the fusion of the optimization targets is less likely to have an effect on the minimum cycle time of the pipeline, compared with a fork or join whose combined handshake component score is lower.

A simple example is used to illustrate the proposed algorithm. Consider a timed PN model as shown in Fig. 3.4 that is composed by abutting the modular PN models of six minimally-concurrent Sync handshake components, denoted as $A$, $B$, $C$, $D$, $E$, and $F$ (for clarity, only the places connecting the modular PN models are shown). The active and RTZ request delays (i.e., the delays associated with the arcs $(ro_A^+, ri_B^+)$ and $(ro_A^-, ri_B^-)$, respectively).
ri}_\text{g-}), respectively) from A to B are 10 and 0.2 ns, respectively, and all places internal to the modular PN models are associated with a delay of 0.2 ns. All the other places have no delay. The PN model has a total of 469 minimal-support S-invariants. These S-invariants are divided into 38 groups according to their respective $\lambda$ values, which range from 12.2 to 0.8 ns. The first group has exactly one S-invariant $y_1$ and it “traverses” the modular PN models of A and B (see Fig. 3.4). Thus, A and B are assigned a score of 1. The handshake components C, D, and E are associated with S-invariants in the second group and, therefore, are assigned a score of 2. An S-invariant $y_2$ in the fourth group “traverses” the modular PN models of A, B, D, E, and F (see Fig. 3.4). However, only F is assigned a score of 4 because the other handshake components already have a score. Since the join involving the handshake components C, E, and F has the highest combined score, it is selected by the algorithm as the optimization targets.

3.5.3. Conditions for Fusion

The fusion of the optimization targets selected in each optimization iteration is subject to two conditions: satisfaction of minimum cycle time constraint by the restructured pipeline and flow equivalence between the original and restructured pipelines.

3.5.3.1. Minimum Cycle Time Constraint Satisfaction

The first condition that must be met in order for the fusion of the optimization targets to be accepted by the optimization process is that the restructured pipeline must satisfy the given minimum cycle time constraint.

There are two reasons why the minimum cycle time of the pipeline must be recomputed after the fusion of the optimization targets. The first is that the PN model of the restructured pipeline is different (albeit only slightly) from that of the pipeline.
before the fusion of the optimization targets. Given that the new PN model would not have exactly the same minimal-support S-invariants as the old one, and that the computation of minimum cycle time is based on minimal-support S-invariants (see (2.1)), it is obvious that the fusion of the optimization targets can potentially modify the minimum cycle time of the pipeline.

The second reason that necessitates the computation of the restructured pipeline’s minimum cycle time is that when the optimization targets are replaced with a single component, the two sets of storage devices that were originally controlled by the optimization targets are now collectively controlled by the replacement component. This means that the delay of the enable signal, denoted as $en_r$, generated by the replacement component to control its storage devices is likely to be longer (due to a higher capacitive load) than the delays of the enable signals generated by the optimization targets to control their respective storage devices. Consequently, the speed performance of the pipeline might be affected.

The longer delay of $en_r$ is taken into consideration by appropriately increasing, if necessary, the active-request delays associated with the output channels of the replacement component. The procedure for doing so is as follows.

First, the capacitive load of $en_r$ is computed by summing the respective capacitive loads of the enable signals generated by the optimization targets. This is facilitated by a lookup table that is initialized, at the beginning of the optimization, with the capacitive load (associated with the enable/clock inputs of the storage devices) of each storage device-enable signal in the original pipeline. During the optimization, when two handshake components are fused, a new entry that corresponds to $en_r$ is created in the table to record the capacitive load of $en_r$. 
Second, the optimization process consults another lookup table to determine the delay of \( en \) (note that the table is prepared by the designer prior to the optimization). More specifically, each entry in the table corresponds to a particular range of capacitive load that a storage device-enable signal is driving and a delay is associated with each range. The delays are obtained through transistor-level simulations of distribution networks (typically consisting of signal buffers arranged in a tree-like structure) for the enable signals. In general, the size and, thus, delay, of a distribution network is proportional to the total capacitive load that it has been designed to drive (note that the design of distribution networks for signals with large capacitive loads, such as global clock signals, is a well-understood topic in VLSI design (see, for example, [53])).

Third, for each output channel, denoted as \( C \), of the replacement component, the associated active-request delay, denoted as \( t_{req} \), is updated:

\[
t'_{req} = t_{req} + t_{enr} - t_{eno}
\]

(3.1)

where \( t'_{req} \) denotes the updated active-request delay, \( t_{enr} \) denotes the delay of \( enr \), and \( t_{eno} \) denotes the delay of the enable signal that is generated by the optimization target which has \( C \) as its output channel before the fusion.

### 3.5.3.2. Flow Equivalence

The second condition that must be met in order for the fusion of the optimization targets to be accepted by the optimization process is that the original and restructured pipelines must be equivalent in behavior.

The notion of flow equivalence [80][134] is used to determine whether the original and restructured pipelines are behaviorally equivalent. Intuitively, two pipelines are said to be flow equivalent if their behaviors cannot be distinguished by observing the sequence of values stored in each storage device. Formally, two pipelines are flow
equivalent if: i) they have the same set of storage devices in the datapaths; and ii) for
each storage device $L$, the projections of the traces onto $L$ (i.e., the sequence of values
stored in $L$) are the same in both pipelines. The following theorem is used when
determining if flow equivalence is observed by the restructuring of the pipeline.

**Theorem 3.1:** The original and restructured pipelines are flow equivalent if there is
no channel link between the optimization targets.

**Proof:** The first condition of flow equivalence, that the original and restructured
pipelines have the same set of storage devices in their datapaths, is immediately
satisfied because the datapath of the original pipeline is not modified by the fusion of
the optimization targets. The rest of the proof is concerned with the satisfaction of the
second condition of flow equivalence, that for each storage device $L$, the projections of
traces onto $L$ are the same in both pipelines.

The trace projection onto a storage device depends on the enabling conditions for
the storage device and the data that is at the input of the storage device when it is
enabled. The latter is a function of the combinational logic block driving the storage
device, as well as the inputs that are fed into the logic block. Since combinational logic
blocks are not affected by handshake component fusion, and their inputs are values held
by storage devices, it is sufficient, when testing the second condition of flow
equivalence, to analyze the effect of fusing the optimization targets on the \textit{data-
registering conditions} for the storage devices controlled by the optimization targets. The
data-registering conditions for a storage device $L$ refer to those under which $L$ is poised
Consider a general fork-join structure as shown in Fig. 3.5(a), where the handshake components $A$ and $B$ are the optimization targets. After the fusion of $A$ and $B$, they are replaced with the handshake component $C$, as shown in Fig. 3.5(b). Note that the channel sources and destinations of $C$ are the combined channel sources and destinations, respectively, of $A$ and $B$.

Now consider a PN model of the fork-join structure shown in Fig. 3.6(a), where the transitions $t_A$ and $t_B$ model the optimization targets $A$ and $B$, respectively, and each to register its input data.
communication channel is modeled as a place $p$ and its complementary place $p'$ such that $\bullet p = p' \bullet$ and $p' = \bullet p$ (see Fig. 3.6(a); note that, for clarity, the places are drawn as arcs). The following points should be noted with regard to the model:

i) A noncomplementary place $p$ that is marked indicates that the corresponding communication channel is active, i.e., a handshake is in progress.

ii) A complementary place $p'$ that is marked indicates that the corresponding communication channel is not active.

iii) The data-registering conditions for the set of storage devices $L_A$ ($L_B$) controlled by $A$ ($B$) are equivalent to the conditions under which $t_A$ ($t_B$) is enabled. For example, in Fig. 3.6(a), when $p_1, p_2, p_3'$, and $p_4'$ are marked, $t_A$ is enabled and $L_A$ are poised to register their new input data.

iv) Each firing of $t_A$ ($t_B$) indicates that $L_A$ ($L_B$) have registered their input data.

v) If it existed, a channel link from $A$ to $B$ ($B$ to $A$) would be modeled as one or more pairs of \{p_i, p_i'\} such that there is a directed path $p_1t_1p_2t_2...p_n$ from $A$ to $B$ ($B$ to $A$) where $p_1, p_2, ..., p_n$ are all noncomplementary places.

The fusion of $A$ and $B$ is modeled by the fusion of $t_A$ and $t_B$ to form $t_C$, where $t_C$ models the handshake component $C$ that has replaced $A$ and $B$ (see Fig. 3.6(b)). Note that the input places of $t_C$ are those of $t_A$ and $t_B$:

$$\bullet t_C = \bullet t_A \cup \bullet t_B$$

(3.2)

Since $C$ controls $L_A$ and $L_B$, it is clear from (3.2) that the original (i.e., before the fusion of $A$ and $B$) data-registering conditions for $L_A$ are added to the those for $L_B$, and vice versa. In the absence of channel links between $A$ and $B$, it can be seen that the satisfaction of the data-registering conditions for $L_A$ and $L_B$ after the fusion of $A$ and $B$ are not causally related. Consequently, it can be deduced that the additional data-registering conditions imposed on $L_A$ ($L_B$) after the fusion of $A$ and $B$ only have the
potential effect of delaying the registering of input data by $L_a (L_B)$, but do not alter the sequence of values stored in $L_a (L_B)$.

Using Theorem 3.1, a simple algorithm is developed that inspects the PN model of the given asynchronous control network for the existence of channel links between the optimization targets. If channel links are found, then the fusion of the optimization targets is abandoned. Otherwise, the optimization targets are fused (provided that the restructured pipeline satisfies the given minimum cycle time constraint).

3.5.4. Optimization Examples

This subsection describes the application of the proposed handshake component fusion method on the control networks of three asynchronous designs: a 16-input pipelined parallel prefix tree (for the addition operation), a four-bit cross-pipelined array multiplier, and a Reed-Solomon error detector for the compact-disc player. The speed and energy dissipations of the circuits are obtained using transistor-level SPICE simulations (Synopsys Nanosim) at the supply voltage of 3.3 V using the AMS 0.35 µm CMOS standard cell library.

3.5.4.1. Pipelined Parallel Prefix Tree

This subsection describes the implementation of a 16-input asynchronous parallel prefix tree for the addition operation with pipelining and demonstrates the optimization of its control network using the proposed handshake component fusion method.

The prefix problem [135] is as follows. Given $x_1, x_2, \ldots, x_n$, compute the prefixes $y_1, y_2, \ldots, y_m$ where $y_k = x_1 \otimes x_2 \otimes \cdots \otimes x_k$ for $1 \leq k \leq n$ and $\otimes$ is an associative operation. An operation on a set $S$ is called associative if it satisfies the associative law:
\[(x \otimes y) \otimes z = x \otimes (y \otimes z) \text{ for all } x, y, z \in S \tag{3.3}\]

Intuitively, this means that the evaluation order of the associative operations within an expression does not affect the value of the expression. Examples of associative operations include addition and multiplication of real numbers. A simplistic way of solving the prefix problem is to compute the results serially, i.e., one would use the expression \(y_k = y_{k-1} \otimes x_k\) to compute \(y_1\) first, followed by \(y_2\), and so on. This method, however, has a computational depth of \(n\) and is, therefore, slow. A much faster way is to exploit the associativity of the operation of interest and compute all the prefixes in parallel. A parallel prefix tree serves such a purpose.

Fig. 3.7(a) shows the architecture of the parallel prefix tree for \(n = 16\) [136]. The tree consists of two substructures: the associative reduction tree and the prefix tree. The purpose of the associative reduction tree is to generate not only some of the final results, namely \(y_1, y_3, y_7,\) and \(y_{15}\), but also the intermediate results, such as \(a[4,5]\) and \(a[8,9]\), where \(a[4,5] = x_4 \otimes x_5 \otimes \cdots \otimes x_j\). These intermediate and final results are then used by the prefix tree to generate all the other final results. For example, \(a[4,5]\) is combined with \(y_3\) to generate \(y_5\). Note that the computational depth of the 16-input parallel prefix tree is only six, compared with 15 if the computations were performed serially. In general, the computational depth is less than \(2 \log(n)\).

To further increase the speed of prefix computations, one could pipeline the parallel prefix tree into multiple stages. Pipelined implementations of the parallel prefix tree using synchronous [136] and asynchronous [137] design techniques have been reported.
Fig. 3.7. 16-input asynchronous pipelined parallel prefix trees for the addition operation: (a) without pipelining; and (b) with pipelining (asynchronous).

Fig. 3.7(b) shows a 16-input asynchronous pipelined parallel prefix tree. It consists of two kinds of cells: adder and buffer. An adder cell, labeled as $A_{i,j}$, adds its two inputs and provides pipelining by storing the result $\left( \sum_{m=1}^{n} x_m \right)$ in latches, whereas a buffer
cell only provides pipelining by buffering its input in latches. The pipelining of the tree has been arrived at in two steps. First, the tree is maximally pipelined, i.e., the tree is pipelined at every level. For example, the intermediate result $s[4,5]$ is pipelined not only at the first level ($k = 1$) within the adder stage, but also at the second ($k = 2$), third ($k = 3$), fourth ($k = 4$), and fifth ($k = 5$) level within the respective buffer stages. Second, the pipelined buffer stages in the maximally-pipelined tree are incrementally removed while preserving its throughput performance. This means that a pipelined buffer stage is not removed if doing so is detrimental to the throughput of the tree.

Each of the cells in the tree contains a handshake component (in this work, the handshake component is a Sync) that communicates with the handshake components in its neighboring cells and controls the latches in the cell. Fig. 3.8 shows the asynchronous control network of the tree (for clarity, the delay-matching elements are not shown).
Table 3.1
Handshake Components Fused During Optimization of Pipelined Parallel Prefix Tree’s
Asynchronous Control Network

<table>
<thead>
<tr>
<th>Fusion #</th>
<th>Targets</th>
<th>Replacement</th>
<th>Fusion #</th>
<th>Targets</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S_{42,3}, S_{82,6}</td>
<td>S_1</td>
<td>16</td>
<td>S_{14}, S_{13,5}</td>
<td>S_{16}</td>
</tr>
<tr>
<td>2</td>
<td>S_{30,9}, S_{810,6}</td>
<td>S_2</td>
<td>17</td>
<td>S_{10,10}, S_{87,7}</td>
<td>S_{17}</td>
</tr>
<tr>
<td>3</td>
<td>S_{44,5}, S_{84,5}</td>
<td>S_3</td>
<td>18</td>
<td>S_{B12,5}, S_{B14,4}</td>
<td>S_{18}</td>
</tr>
<tr>
<td>4</td>
<td>S_{40,13}, S_{B14,6}</td>
<td>S_4</td>
<td>19</td>
<td>S_{17}, S_{10,8}</td>
<td>S_{19}</td>
</tr>
<tr>
<td>5</td>
<td>S_{48,5}, S_{88,4}</td>
<td>S_5</td>
<td>20</td>
<td>S_{10,11}, S_{60,15}</td>
<td>S_{20}</td>
</tr>
<tr>
<td>6</td>
<td>S_{412,15}, S_{48,11}</td>
<td>S_6</td>
<td>21</td>
<td>S_{8,6}, S_{10,11}</td>
<td>S_{21}</td>
</tr>
<tr>
<td>7</td>
<td>S_{412,13}, S_{B12,4}</td>
<td>S_7</td>
<td>22</td>
<td>S_{15}, S_{84,6}</td>
<td>S_{22}</td>
</tr>
<tr>
<td>8</td>
<td>S_{7}, S_{414,15}</td>
<td>S_8</td>
<td>23</td>
<td>S_{8}, S_{89,4}</td>
<td>S_{23}</td>
</tr>
<tr>
<td>9</td>
<td>S_{40,4}, S_{10,5}</td>
<td>S_9</td>
<td>24</td>
<td>S_{18}, S_{13,4}</td>
<td>S_{24}</td>
</tr>
<tr>
<td>10</td>
<td>S_{1}, S_{40,1}</td>
<td>S_{10}</td>
<td>25</td>
<td>S_{13,3}, S_{B14,3}</td>
<td>S_{25}</td>
</tr>
<tr>
<td>11</td>
<td>S_{2}, S_{87,6}</td>
<td>S_{11}</td>
<td>26</td>
<td>S_{89,5}, S_{B10,5}</td>
<td>S_{26}</td>
</tr>
<tr>
<td>12</td>
<td>S_{40,3}, S_{44,7}</td>
<td>S_{12}</td>
<td>27</td>
<td>S_{B11,4}, S_{48,15}</td>
<td>S_{27}</td>
</tr>
<tr>
<td>13</td>
<td>S_{4}, S_{40,12}</td>
<td>S_{13}</td>
<td>28</td>
<td>S_{12}, S_{55,5}</td>
<td>S_{28}</td>
</tr>
<tr>
<td>14</td>
<td>S_{82,6}, S_{B14,5}</td>
<td>S_{14}</td>
<td>29</td>
<td>S_{3}, S_{46,7}</td>
<td>S_{29}</td>
</tr>
<tr>
<td>15</td>
<td>S_{12}, S_{40,2}</td>
<td>S_{15}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Only successful fusions are included.

The asynchronous control network of the parallel prefix tree was optimized using the proposed handshake component fusion method. The minimum cycle time constraint for the optimization was set at 10% higher than the minimum cycle time of the original tree. This provides some degree of freedom for the optimization process without compromising too much on the speed performance of the design. Table 3.1 shows the handshake components that were successfully fused during the optimization. For example, in the first successful fusion, the handshake components $S_{42,3}$ and $S_{82,6}$ are replaced with $S_1$ (note that the latches controlled by $S_{42,3}$ and $S_{82,6}$ are controlled by $S_1$ after the fusion). Fig. 3.9 shows the optimized control network, called OPT_10%.
Comparisons between the original parallel prefix tree and the optimized one (OPT_10%) indicate that significant reductions in transistor count and energy dissipation in the control network have been achieved by the optimization (see Table 3.2). Compared with the original network, the control network of OPT_10% has 40% fewer transistors and dissipates 43% less energy, with OPT_10% sacrificing only 6% of its throughput. As a whole, OPT_10% has 12% fewer transistors and dissipates 20% less energy than the original tree.

To investigate the effect of the degree of relaxation of the minimum cycle time constraint on the optimization process, two additional optimizations were performed. In the first additional optimization, the minimum cycle time constraint was relaxed by 0%,

Fig. 3.9. The optimized control network (OPT_10%) of the asynchronous pipelined parallel prefix tree (delay-matching elements not shown) after handshake component fusion.
### Table 3.2
Comparisons of Pipelined Parallel Prefix Tree Before and After Handshake Component Fusion

<table>
<thead>
<tr>
<th></th>
<th>$N_T (k)$</th>
<th>$E$ (pJ)</th>
<th>$\tau$ (ns)</th>
<th>$E_r$ $E_r$ (10^{-26}$J$s$^2$)</th>
<th>$N_T\tau$ (10^{-4}s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>3.71</td>
<td>12.5</td>
<td>185</td>
<td>336</td>
<td>9.24</td>
</tr>
<tr>
<td>OPT_0%</td>
<td>2.87</td>
<td>11.6</td>
<td>135</td>
<td>290</td>
<td>8.97</td>
</tr>
<tr>
<td>OPT_10%</td>
<td>2.23</td>
<td>11.0</td>
<td>106</td>
<td>268</td>
<td>9.86</td>
</tr>
<tr>
<td>OPT_20%</td>
<td>1.70</td>
<td>10.5</td>
<td>75</td>
<td>241</td>
<td>10.91</td>
</tr>
</tbody>
</table>

$N_T$: transistor count  
$E$: av. energy per computation  
$\tau$: minimum cycle time

...
OPT_20% has 16% fewer transistors and dissipates 28% less energy than the original tree. However, OPT_20% is 18% slower than the original tree.

In terms of the energy-delay figure of merit ($E^2$), OPT_0% is the most favorable circuit, with a 19% improvement over the original unoptimized tree.

To further investigate the benefits of the proposed handshake component fusion method for asynchronous designs, a synchronous version of the parallel prefix tree was implemented and subject to the same Synopsys Nanosim simulations as the asynchronous trees. When running at a clock frequency of 100 MHz (i.e., the clock period is 10 ns, which is close to the minimum cycle time of the asynchronous trees), the energy dissipation of the synchronous tree is 303 pJ per computation. This is 10% less than the original unoptimized asynchronous tree, but 13% more than OPT_10%

This result supports the argument made in this thesis that reducing the power of control networks in asynchronous circuits is important in order for asynchronous circuits to be viable alternatives to synchronous circuits.

The maximum clock frequency of the synchronous tree is approximately 125 MHz, which is at least 11% faster than the asynchronous trees. The slower speeds of the asynchronous trees are due to the delays incurred during handshaking within the control network. However, it is worth noting that the combinational logic blocks in the parallel prefix tree are all adders, which makes the asynchronous parallel prefix tree a prime target for speed improvement through asynchronous design techniques that exploit average-case performance [27][28][29].

The transistor count for the synchronous tree 14.2 k, which is at least 14% higher than the asynchronous trees, despite the overheads caused by the asynchronous control networks. The higher transistor count for the synchronous tree can be largely attributed to the use of flip-flops instead of latches (the latter are used in the asynchronous trees).
The computation time for the optimization of the parallel prefix tree using the proposed handshake component fusion method (for the case of OPT_10%) on a personal computer (CPU: Pentium IV 3.4 GHz; OS: Windows XP; Memory: 1.0 GB) is 1157 s, of which 41 s is spent on target selection and 1116 s on minimum cycle time analysis.

3.5.4.2. Cross-Pipelined Array Multiplier

Previous work [138][139][140] has indicated that asynchronous design techniques are well suited for the implementation of pipeline arrays. In this subsection, an asynchronous four-bit cross-pipelined array multiplier [140] is considered and the optimization of the multiplier’s asynchronous control network using the proposed handshake component fusion method is demonstrated.

As shown in Fig. 3.10, the multiplier has a highly-regular structure consisting of interconnected cells, of which there are three types, denoted as L, M, and A.

The L cells serve as buffering stages for the operands (x[3:0] and y[3:0]) and result (p[7:0]) of the multiplication. Note that the buffering stages for the operands are necessary because in order for the operands to be received correctly by the multiplier, the individual bits of the operands for each multiplication must be received by the multiplier not simultaneously but in sequence. For example, each operand y has its most significant bit (MSB) y[3] transferred to cell M50 first. This is followed by the transfer of y[2] to M20, y[1] to M10, and finally y[0] to M00 in that order. Similarly, buffering stages are needed to correctly output the result of the multiplier because the individual bits of the result are generated in sequence, starting at p[0] and ending at p[7].
Fig. 3.10. The architecture of a four-bit asynchronous cross-pipelined array multiplier.

The \( M \) cells performs two functions. First, each \( M_y \) cell acts as a buffering stage for the operand bits \( y[i] \) and \( x[j] \), transferring \( y[i] \) from left to right and \( x[j] \) from bottom to top. Second, each \( M_y \) cell computes the bit product \( b_y = y[i] \cdot x[j] \) and performs the full addition \( \{c_y, s_y\} = b_y + c_{(i-1)} + s_{(i-1)(j-1)} \), where \( c_{(i-1)} \) and \( s_{(i-1)(j-1)} \) are the carry and sum.
Fig. 3.11. The control network of the four-bit asynchronous cross-pipelined array multiplier (delay-matching elements not shown).

inputs, respectively, and $c_y$ and $s_y$ the carry and sum outputs, respectively. Note that $c_y$ and $s_y$ are buffered in $M_y$.

The $A$ cells are pipelined half-adders. More specifically, each $A_y$ cell performs the half addition $\{c_y, s_y\} = c_{y(-1)} + s_{y(-1)}$, where $c_{y(-1)}$ and $s_{y(-1)}$ are the carry and sum inputs, respectively, and $c_y$ and $s_y$ the carry and sum outputs, respectively. Note that $c_y$ and $s_y$ are buffered in $A_y$.

Each of the cells contains a handshake component (in this work, the handshake component is a Sync) that communicates with the handshake components in its neighboring cells and controls the latches in the cell. Fig. 3.11 shows the asynchronous control network of the multiplier (for clarity, the delay-matching elements are not shown).
The asynchronous control network of the multiplier was optimized using the proposed handshake component fusion method. The minimum cycle time constraint for the optimization was set at 10% higher than the minimum cycle time of the original multiplier. Due to the highly-regular structure of the control network, many optimization targets were identified in it and successfully fused during the optimization without impacting the multiplier’s throughput. Table 3.3 shows the handshake components that were successfully fused during the optimization. For example, in the first successful fusion, the handshake components $S_{A04}$ and $S_{p3}$ were replaced with the handshake component $S_1$. Fig. 3.12 shows the optimized control network.

Comparisons between the original and optimized multipliers indicate that significant reductions in transistor count and energy dissipation in the control network
Fig. 3.12. The optimized control network of the four-bit asynchronous cross-pipelined array multiplier after handshake component fusion (delay-matching elements are not shown).

have been achieved by the optimization (see Table 3.4). When compared with the original control network, the optimized control network has 67% fewer transistors and dissipates, on the average, 66% less energy per multiplication. As a whole, the optimized multiplier has 30% fewer transistors and dissipates, on the average, 56% less energy per multiplication than the original multiplier, and yet is marginally faster.

The computation time for the optimization of the array multiplier using the proposed handshake component fusion method on a personal computer (CPU: Pentium IV 3.4 GHz; OS: Windows XP; Memory: 1.0 GB) is 682 s, of which 28 s is spent on target selection and 654 s on minimum cycle time analysis.
### Table 3.4
Comparisons of four-bit asynchronous cross-pipelined array multiplier Before and After Handshake Component Fusion

<table>
<thead>
<tr>
<th></th>
<th>(N_T) (k)</th>
<th>(E) (pJ)</th>
<th>(\tau) (ns)</th>
<th>(E\tau^2 (10^{-26} J_s^2))</th>
<th>(N_T\tau (10^{-5}s))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>2.42</td>
<td>5.49</td>
<td>179</td>
<td>207</td>
<td>9.19</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimized</td>
<td>0.79</td>
<td>3.86</td>
<td>60.7</td>
<td>91.2</td>
<td>9.04</td>
</tr>
</tbody>
</table>

\(N_T\): transistor count  
\(E\): av. energy per computation  
\(\tau\): minimum cycle time

### 3.5.4.3. Reed-Solomon Error Detector

This subsection describes the optimization of the asynchronous control network (see Fig. 2.21) of the asynchronous Reed-Solomon error detector (see Chapter 2 for a detailed description of the design of the error detector) using the proposed handshake component fusion method.

Fig. 3.13 shows the optimized control network of the error detector (the delay-matching elements are not shown) after handshake component fusion. Specifically, the Sync handshake components \(\{S_0, S_1, S_2, S_3\}\) and the SyncGuard handshake components \(\{G_0, G_1, G_2, G_3\}\) have been replaced with Sync \(S_0\) and SyncGuard \(G_4\), respectively. Compared with the original control network, the optimized control network has 34% fewer transistors, dissipates, on the average, 32% less energy per codeword, and is marginally faster (see Table 3.5).

The computation time for the optimization of the array multiplier using the proposed optimal decoupling method on a personal computer (CPU: Pentium IV 3.4 GHz; OS: Windows XP; Memory: 1.0 GB) is 193 ms, of which 42 ms is spent on target selection and 151 ms on minimum cycle time analysis.
Fig. 3.13. The optimized control network of the asynchronous Reed-Solomon error detector after handshake component fusion.

Table 3.5
Comparisons of Reed-Solomon Error Detector Before and After Handshake Component Fusion

<table>
<thead>
<tr>
<th></th>
<th>(N_T) (k)</th>
<th>(E) (pJ)</th>
<th>(\tau) (ns)</th>
<th>(E\tau^2) (\times 10^{25}\text{Js}^2)</th>
<th>(N_T\tau) (\times 10^{-3}) s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>0.74</td>
<td>6.91</td>
<td>1.10</td>
<td>2.87</td>
<td>213</td>
</tr>
<tr>
<td>Optimized</td>
<td>0.49</td>
<td>6.64</td>
<td>0.75</td>
<td>2.51</td>
<td>205</td>
</tr>
</tbody>
</table>

\(N_T\): transistor count
\(E\): av. energy per codeword
\(\tau\): processing time per codeword

3.6. Optimal Decoupling

This section describes the proposed optimal decoupling method that resolves the dilemma when designing asynchronous pipelines between using small handshake components to reduce asynchronous control overheads and satisfying throughput constraints. At the core of the method is a proposed branch-and-bound algorithm that searches for the optimal mix of handshake components of different degree of concurrency in a given asynchronous control network, with the objective of incurring
the least circuit area and power dissipation for the control network, while satisfying a given minimum cycle time constraint.

3.6.1. Demonstration

A simple example is used to demonstrate the main ideas behind the proposed optimal decoupling method.

Consider the PN model of a three-stage linear pipeline with asymmetric channel delays based on minimally-concurrent latch controllers (or Sync handshake components, in the context of this work) as shown in Fig. 3.14(a).

To compute the minimum cycle time \( \tau_1 \) of the pipeline, all minimal-support S-invariants of the model is first extracted using an existing algorithm (see Chapter 4). Note that for the PN models in Fig. 3.14, the support of a minimal-support S-invariant is given by the places in a corresponding simple directed loop. A directed loop is said to be simple if it does not contain another directed loop. For example, the dotted arcs in Fig. 3.14(a) correspond to a simple directed loop, labeled as \( A \). The “cycle time” \( \lambda \) of each simple directed loop is computed using (2.1). It can be shown that for the PN model in Fig. 3.14(a), \( A \) has the longest “cycle time” among all simple directed loops, i.e., \( \tau_1 = \lambda_A \), where \( \lambda_A \) is the “cycle time” of \( A \). Note that \( \lambda_A \) is given by \( \lambda_A = d_1 + d_3 + d_A \), where \( d_1 \) and \( d_3 \) are the active request delays between Stage 1 and Stage 2, and Stage 2 and Stage 3, respectively, and \( d_A \) is the total handshake component delay associated with \( A \).

To improve the pipeline’s throughput, one can replace all the minimally-concurrent latch controllers with maximally-concurrent ones (see Fig. 3.14(b)). Now, the minimum cycle time is \( \tau_2 = \max \{ \lambda_B, \lambda_C \} \), where \( \lambda_B = d_1 + d_2 + d_B \) and \( \lambda_C = d_3 + d_4 + d_C \) are the
"cycle times" of the simple directed loops $B$ and $C$, respectively, and $d_2$ and $d_4$ are the RTZ request delays between Stage 1 and Stage 2, and Stage 2 and Stage 3, respectively. Given that the request delays are asymmetric (i.e., $d_2$ and $d_4$ are no more than $d_1$ and $d_3$) and assuming that each pipeline stage is performing a nontrivial computation (i.e., $d_A$, $d_B$, and $d_C$ are small compared with the $d_1$ and $d_3$), it can be deduced that $r_2 \leq r_1$.

Alternatively, the minimum cycle time of the pipeline can be improved by eliminating the directed loop $A$ through the following handshake protocol decoupling
procedure: i) replace the arc $ro_2\rightarrow ai_1$ with $en_2\rightarrow ai_1$; and ii) replace the arc $ao_2^+ \rightarrow en_2^-$ with $ao_2^+ \rightarrow ro_2^-$. This procedure has the effect of allowing Stage 2’s new input handshake to start before it receives an acknowledgement ($ao_2^+$) from Stage 3.

The result (see Fig. 3.14(c)) is a semi-concurrent latch controller at Stage 2 while the latch controllers at Stage 1 and 3 remain minimally-concurrent. Thus, one now has a pipeline whose minimum cycle time is $\tau_3 = \max\{\lambda_D, \lambda_E\}$, where $\lambda_D = d_1 + d_2 + d_D$ and $\lambda_E = d_3 + d_4 + d_E$ are the “cycle times” of the simple directed loops $D$ and $E$, respectively. Since the minimally- and semi-concurrent latch controllers are less complex than the maximally-concurrent ones, $d_D$ and $d_E$ are likely to be less than $d_B$ and $d_C$. Thus, it is reasonable to expect $\tau_3 \leq \tau_2 \leq \tau_1$.

In summary, maximum throughput for the three-stage pipeline can be attained simply by partially decoupling Stage 2 of the pipeline, without making any changes to Stage 1 and Stage 3. Furthermore, the control network of the optimally-decoupled pipeline is expected to dissipate less power than that of the maximally-concurrent one due to its smaller latch controllers.

### 3.6.2. Branch-and-bound Algorithm

The most straightforward approach to solving the optimal decoupling problem is to enumerate all possible ways of mixing the concurrency level of the handshake components in the given asynchronous control network, and then pick out the least concurrent configuration that satisfies the throughput constraint. Such a simple approach would, however, have a time complexity of $O(r^n)$, where $r$ is the number of concurrency levels that a handshake component can have (note that in this work, $r = 3$) and $n$ is the number of handshake components in the control network. This renders the
search space for the solution impractically large even for moderate-size control networks.

This subsection describes the proposed algorithm for solving the optimal decoupling problem using branch and bound [141]. Branch and bound is a general algorithmic method for finding optimal solutions of various optimization problems. In essence, it is an enumeration approach that reduces the search space by pruning unpromising search space.

A core tool of any branch-and-bound procedure is branching, where a region of candidate solutions is split into several subregions. Since branching is applied iteratively on subregions that are not pruned, all the subregions created in a branch-and-bound procedure form a tree, termed a branch-and-bound tree, with each node of the tree representing a subregion.

In the proposed branch-and-bound algorithm, each node of the branch-and-bound tree corresponds to a particular configuration of the given control network. The configuration of the control network is a mapping of each handshake component in the control network to one of four states: unconstrained, minimally-concurrent, semi-concurrent, and maximally-concurrent. An unconstrained handshake component is identical to a minimally-concurrent one except that it can be selected as a branching variable during a branching operation, whereas the latter cannot. The other three states are constrained states representing different degree of concurrency for a handshake component. At the root node of the branch-and-bound tree, all handshake components in the control network are unconstrained.

During a branching operation, an unconstrained handshake component is selected from the configuration of the parent node as the branching variable and a constrained state is assigned to it. The constrained state assignment to the branching variable is
called the branching constraint. During a branching operation, the configuration of the parent node is altered by the branching constraint before it is passed to the child node. Since there are three constrained states, there are three possible assignments to the branching variable and, thus, three child nodes for each parent node (see Fig. 3.15).

For the first branch, the branching variable, denoted as $v$, is assigned to be minimally-concurrent ($v = \text{"minimally-concurrent"}$). Note that since an unconstrained handshake component is physically the same as its minimally-concurrent counterpart, there is no difference between the control networks that correspond to the configurations of the parent node and the first child node, i.e., the nodes $n_1$ and $n_2$ in Fig. 3.15 correspond to identical control networks.

For the second and third branch leading to the nodes $n_3$ and $n_4$ in Fig. 3.15, respectively, the branching variable is assigned to be semi-concurrent ($v = \text{"semi-concurrent"}$) and maximally-concurrent ($v = \text{"maximally-concurrent"}$), respectively. Unlike the first child node $n_2$, the second and third child nodes, $n_3$ and $n_4$, correspond to control networks that are different from that of the parent node $n_1$. This means that the
PN models of the control networks that correspond to $n_3$ and $n_4$ are different from that of the control network that corresponds to $n_1$. Thus, the minimal-support S-invariants of the new PN models must be recomputed so as to determine the minimum cycle times of the new configurations (see Chapter 4 for a detailed discussion on algorithms for computing minimal-support S-invariants).

The minimum cycle time of each new configuration is compared against the overall upper bound, denoted as $u$, i.e., the best minimum cycle time amongst the configurations examined thus far by the algorithm. If the new minimum cycle time is less than $u$, then it replaces $u$ as the upper bound (note that at the beginning of the algorithm, $u$ is initialized as the minimum cycle time of the root node’s configuration).

The minimum cycle time of each new configuration is also compared against the minimum cycle time constraint specified by the designer. If the former is found to be not more than the latter, then the associated node is pruned, i.e., the branching operation is not performed on it. This is because any improvement in minimum cycle time due to further handshake component decouplings in the configuration is redundant given that the minimum cycle time constraint has already been satisfied. In addition, further decouplings in the configuration will lead to control networks that are larger and dissipate more power. The configuration of such a node is considered to be a candidate solution to the optimal decoupling problem.

A node is also pruned if the lower bound, denoted as $l$, for the minimum cycle time of its configuration $c$ is greater than $u$. To compute $l$, one needs to predict how far the minimum cycle time of $c$ can be improved through further handshake component decoupling. The prediction is performed using the following steps. First, the minimal-support S-invariants ($y_k$) of the PN model of $c$ are grouped into sets based on their “cycle times” $\lambda_k$, where $\lambda_k = y_k^* D(A^*) x / y_k^T M_0$ (see (2.1)). Second, the sets are ranked
in decreasing order of $\lambda_k$, i.e., the first set has the largest $\lambda_k$, the second set has the second largest $\lambda_k$, and so forth. Third, the decoupling expression, denoted as $E_i$, of a set $S_i$ (starting with the highest ranking set) is composed as follows. The support of each invariant $y_k$ in $S_i$ is searched to detect the presence of certain predefined ordered sets of places. These sets of places identify which handshake components should be decoupled to eliminate $y_k$ and, therefore, potentially improve the minimum cycle time of $c$.

For example, a minimally-concurrent Sync (see Fig. 3.1(a)) is identified for decoupling if the places $\{(ao^+, en^-), (en^-, ro^-), (ro^-, ai^-)\}$ (equivalent to the path $ao^+ \rightarrow en^- \rightarrow ro^- \rightarrow ai^-$) in its modular PN model are found in $\|y_k\|$. Thus, if $y_k$ is the minimal-support $S$-invariant that is associated with the directed loop $A$ in Fig. 3.14(a), then the minimally-concurrent Sync at Stage 2 would be identified for decoupling because the places $\{(ao_2^+, en_2^-), (en_2^-, ro_2^-), (ro_2^-, ai_2^-)\}$ are found in $\|y_k\|$. Likewise, a semi-concurrent Sync (see Fig. 3.1(b)) is identified for decoupling if the places $\{(ri^-, en^-), (en^-, ro^-)\}$ in its modular PN model are found in $\|y_k\|$.

The handshake components identified for decoupling in $y_k$ are represented as Boolean equation literals, which are joined together using an OR operator to form a clause. The OR operator is used because only one of the handshake components needs to be decoupled in order to eliminate $y_k$. Since all minimal-support $S$-invariants in $S_i$ must be eliminated to potentially improve the minimum cycle time, the clauses extracted from all minimal-support $S$-invariants in $S_i$ are joined together using the AND operator, resulting in $E_i$. In the event that no handshake component has been identified in $y_k$, then $y_k$ cannot be eliminated through decoupling and the corresponding “clause” is set to FALSE. Furthermore, to reflect the fact that a handshake component assigned with a constrained state cannot be further decoupled, the constraints of the current
configuration are imposed on $E_i$ by setting the literals of $E_i$ that represent handshake components with constrained states to FALSE.

At the end of the above procedure, if $E_i$ is FALSE, then it implies that the minimal-support S-invariants in $S_i$ cannot all be eliminated through further decoupling. In this case, $l$ is given by the $\lambda_k$ value of any $y_k \in S_i$ (recall that all $y_k$ in $S_i$ have the same $\lambda_k$ value). On the other hand, if $E_i$ has at least one literal, then it implies that all minimal-support S-invariants in $S_i$ can be eliminated through further decoupling. In this case, the above procedure is repeated for the next highest ranking set of S-invariants.

For example, let $S_i$ consists of two minimal-support S-invariants $y_1$ and $y_2$, and let the identified handshake components that can be decoupled to eliminate $y_1$ and $y_2$ be $\{A, B\}$ and $\{C, D\}$, respectively. Thus, the decoupling expression corresponding to $S_i$ is given by

$$E_i = (A | B) \cdot (C | D) \quad (3.4)$$

Furthermore, let handshake component $A$ be assigned with a constrained state, i.e., the literal $A$ in (3.4) is set to FALSE, leading to

$$E_i = B \cdot (C | D) \quad (3.5)$$

The above equation means that handshake component $B$, together with either handshake component $C$ or $D$ can be decoupled to eliminate both $y_1$ and $y_2$. Thus, the above procedure must be repeated for the next highest ranking set of minimal-support S-invariants.

Yet another situation where a node in the branch-and-bound tree is pruned is when the minimum cycle time of its associated configuration can no longer be improved through further handshake component decoupling, i.e., when the minimum cycle time is equal to $l$. 

151
If a node is not pruned, then it is branched into three child nodes, the procedure for which has been described earlier in this section. The branching variable for the branching operation is arbitrarily selected from the literals of the decoupling expression of the minimal-support S-invariant set that has the largest corresponding $\lambda_k$ value.

At the end of the search, if there is one or more configurations whose minimum cycle times satisfy the minimum cycle time constraint, then the configuration with the least amount of decoupling is selected as the solution to the optimal decoupling problem. Otherwise, the configuration with the best minimum cycle time is returned as the solution.

### 3.6.3. Optimization Examples

This subsection describes the application of the proposed optimal decoupling method on the control networks of three asynchronous designs: a 16-input pipelined parallel prefix tree (for the addition operation), a four-bit cross-pipelined array multiplier, and a Reed-Solomon error detector for the compact-disc player. The minimum cycle time constraint was set to be zero during the optimization. This means that the algorithm would search for the control network configuration that provides the best minimum cycle time. The speed and energy dissipations of the circuits are obtained using transistor-level SPICE simulations (Synopsys Nanosim) at the supply voltage of 3.3 V using the AMS 0.35 µm CMOS standard cell library.

#### 3.6.3.1. Pipelined Parallel Prefix Tree

The asynchronous control network of the 16-input pipelined parallel prefix tree for the addition operation was optimized using the proposed optimal decoupling method (see Section 3.5.4.1 for a description of parallel prefix computation).
Fig. 3.16. The optimally-decoupled control network of the 16-input asynchronous pipelined parallel prefix tree.

Fig. 3.16 shows the optimally-decoupled (OPTD) control network (the delay-matching elements are not shown) of the parallel prefix tree, which consists of 28% minimally-concurrent, 60% semi-concurrent, and 12% maximally-concurrent handshake components.
Table 3.6
Comparisons of 16-Input Pipelined Parallel Prefix Trees Implemented Using Different Handshake Component Configurations

<table>
<thead>
<tr>
<th></th>
<th>$N_T$ (k)</th>
<th>$E$ (pJ)</th>
<th>$\tau$ (ns)</th>
<th>$E\tau^2$ $(10^{-26}Js^2)$</th>
<th>$N_T\tau$ $(10^4s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Total</td>
<td>Control</td>
<td>Total</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FALLD [80]</td>
<td>5.17</td>
<td>13.9</td>
<td>264</td>
<td>402</td>
<td>8.07</td>
</tr>
<tr>
<td>DESYNC [80]</td>
<td>5.86</td>
<td>14.6</td>
<td>353</td>
<td>497</td>
<td>7.59</td>
</tr>
<tr>
<td>MINC</td>
<td>3.71</td>
<td>12.5</td>
<td>185</td>
<td>336</td>
<td>9.24</td>
</tr>
<tr>
<td>SEMID [102]</td>
<td>5.04</td>
<td>13.8</td>
<td>246</td>
<td>388</td>
<td>9.98</td>
</tr>
<tr>
<td>FULLYD [102]</td>
<td>5.83</td>
<td>14.6</td>
<td>375</td>
<td>515</td>
<td>7.83</td>
</tr>
<tr>
<td>OPTD</td>
<td>4.65</td>
<td>13.4</td>
<td>219</td>
<td>360</td>
<td>7.01</td>
</tr>
</tbody>
</table>

$N_T$: transistor count  
$E$: av. energy per computation  
$\tau$: minimum cycle time

To compare the OPTD control network with control networks based on uniform handshake component concurrency, five other control networks were implemented using the following handshake component design styles: fall-decoupled (FALLD) [80], desynchronization control (DESYNC) [80], minimally-concurrent (MINC), semi-decoupled (SEMID) [102], and fully-decoupled (FULLYD) [102]. Compared with these control networks (except the MINC control network), the OPTD control network, on the average, has 15% fewer transistors and dissipates 27% less energy per computation, and yet is one of the fastest (see Table 3.6).

Although the OPTD control network is larger and dissipates more energy than the MINC control network, it is 24% faster. In addition, the parallel prefix tree with the OPTD control network is 38% better in $E\tau^2$ than that with the MINC control network.

The computation time for the optimization of the parallel prefix tree using the proposed optimal decoupling method on a personal computer (CPU: Pentium IV 3.4 GHz; OS: Windows XP; Memory: 1.0 GB) is 846 s, of which 114 s is spent on lower bound computation and 732 on minimum cycle time analysis.
Fig. 3.17. The optimally-decoupled control network of the four-bit asynchronous cross-pipelined array multiplier.

### 3.6.3.2. Cross-Pipelined Array Multiplier

The asynchronous control network of the four-bit cross-pipelined array multiplier was optimized using the proposed optimal decoupling method (see Section 3.5.4.2 for a description of the multiplier).

Fig. 3.17 shows the optimally-decoupled (OPTD) control network (the delay-matching elements are not shown) of the multiplier, which consists of 22% minimally-concurrent and 78% semi-concurrent handshake components, but has no maximally-concurrent handshake components.

Compared with the control networks based on uniform handshake component concurrency (except the MINC control network), the OPTD control network, on the average, has 26% fewer transistors and dissipates 34% less energy per multiplication,
Table 3.7
Comparisons of 16-Input Cross-Pipelined Array Multipliers Implemented Using Different Handshake Component Configurations

<table>
<thead>
<tr>
<th></th>
<th>(N_f) (k)</th>
<th>(E) (pJ)</th>
<th>(\tau) (ns)</th>
<th>(E\tau^2(10^{-28}Js^2))</th>
<th>(N_f\tau(10^{-5}))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Control</td>
<td>Total</td>
<td>Control</td>
<td>Total</td>
<td></td>
</tr>
<tr>
<td>FALLD [80]</td>
<td>3.01</td>
<td>6.08</td>
<td>273</td>
<td>301</td>
<td>7.82</td>
</tr>
<tr>
<td>DESYNC [80]</td>
<td>3.78</td>
<td>6.85</td>
<td>332</td>
<td>361</td>
<td>7.26</td>
</tr>
<tr>
<td>MINC</td>
<td>2.42</td>
<td>5.49</td>
<td>179</td>
<td>207</td>
<td>9.19</td>
</tr>
<tr>
<td>SEMID [102]</td>
<td>3.30</td>
<td>6.37</td>
<td>265</td>
<td>296</td>
<td>9.64</td>
</tr>
<tr>
<td>FULLYD [102]</td>
<td>3.71</td>
<td>6.78</td>
<td>320</td>
<td>348</td>
<td>7.62</td>
</tr>
<tr>
<td>OPTD</td>
<td>2.55</td>
<td>5.62</td>
<td>195</td>
<td>224</td>
<td>6.69</td>
</tr>
</tbody>
</table>

\(N_f\): transistor count  
\(E\): av. energy per computation  
\(\tau\): minimum cycle time

and yet is the fastest (see Table 3.7). Although the OPTD control network is larger and dissipates more energy than the MINC control network, it is 27% faster. In addition, the multiplier with the OPTD control network is 43% better in \(E\tau^2\) than that with the MINC control network.

The computation time for the optimization of the array multiplier using the proposed optimal decoupling method on a personal computer (CPU: Pentium IV 3.4 GHz; OS: Windows XP; Memory: 1.0 GB) is about 502 s, of which 64 s is spent on lower bound computation and 438 s on minimum cycle time analysis.

3.6.3.3. Reed-Solomon Error Detector

The asynchronous control network of the Reed-Solomon error detector was optimized using the proposed optimal decoupling method (see Chapter 2, Section 2.5 for a description of the error detector).

The solution returned by the optimal decoupling algorithm is a control network that consists of only minimally-concurrent (MINC) handshake components. Compared with
### Table 3.8
Comparisons of Reed-Solomon Error Detectors Implemented Using Different Handshake Component Configurations

<table>
<thead>
<tr>
<th></th>
<th>$N_T$ (k)</th>
<th>$E$ (nJ)</th>
<th>$\tau$ (ns)</th>
<th>$E \tau^2$ ($10^{-22} J s^2$)</th>
<th>$N_T \tau$ ($10^3$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Control</td>
<td>Total</td>
<td>Control</td>
<td>Total</td>
<td></td>
</tr>
<tr>
<td>MINC</td>
<td>0.74</td>
<td>6.91</td>
<td>1.10</td>
<td>2.87</td>
<td>213</td>
</tr>
<tr>
<td>SEMIC</td>
<td>1.01</td>
<td>7.17</td>
<td>1.58</td>
<td>3.39</td>
<td>221</td>
</tr>
<tr>
<td>MAXC</td>
<td>1.16</td>
<td>7.32</td>
<td>1.97</td>
<td>3.78</td>
<td>227</td>
</tr>
</tbody>
</table>

$N_T$: transistor count  
$E$: av. energy per codeword  
$\tau$: processing time per codeword

The control networks based on only semi-concurrent (SEMIC) and maximally-concurrent (MAXC) handshake components, the MINC control network has at least 27% fewer transistors, dissipates, on the average, 30% less energy per codeword, and is marginally faster (see Table 3.8).

The computation time for the optimization of the error detector using the proposed optimal decoupling method on a personal computer (CPU: Pentium IV 3.4 GHz; OS: Windows XP; Memory: 1.0 GB) is 48 ms, of which 11 ms is spent on lower bound computation and 37 ms on minimum cycle time analysis.

### 3.7. Overall Optimization Flow

The two optimization methods – handshake component fusion and optimal decoupling – proposed in this chapter have, thus far, been discussed separately. This section discusses the feasibility of applying both optimization methods on a single design and proposes an optimization flow for doing so.

As described earlier in this chapter, handshake component fusion and optimal decoupling have both been proposed to facilitate the design of low-power asynchronous circuits. However, their specific objectives are different.
The objective of the handshake component fusion method is to reduce the size and power dissipation of the control network by attempting to combine as many pairs of handshake components as possible, while ensuring that the minimum cycle time constraint is satisfied. In most cases, handshake component fusion will not lead to an improvement in the minimum cycle time of the control network, but make it worse (within the specified limit). An exception of this is when the minimum cycle time constraint is set to be the same as the minimum cycle time of the original network, i.e., the optimization is performed with 0% minimum cycle time relaxation. In this case, the resulting minimum cycle time of the network might be marginally better than its original minimum cycle time. For example, in the optimization of the parallel prefix tree reported in Section 3.5.4.1, only OPT_0% (with 0% minimum cycle time relaxation) has a slightly better minimum cycle time than the original network. Both OPT_10% and OPT_20% (with 10% and 20%, respectively, minimum cycle time relaxation) have poorer minimum cycle times than the original network (by less than 10% and 20%, respectively).

On the other hand, the objective of the optimal decoupling method is to compute the smallest possible configuration of the control network that is structurally the same as the original network but with a mix of handshake components of different degree of concurrency that satisfies the minimum cycle time constraint. Given that the starting point of the optimization process is always the original network consisting of only minimally-concurrent handshake components, the minimum cycle time that is set as the optimization goal should be shorter than the minimum cycle time of the original network (if the minimum cycle time constraint is set to be the same as or longer than the minimum cycle time of the original network, then the optimization process would simply return the original network as the solution).
In summary, the handshake component fusion method can be viewed as a timing-aware power optimization method, whereas the optimal decoupling method can be viewed as a power-aware timing optimization method. Given that the specific objectives of the optimization methods are different, integrating the two methods does not appear to be feasible.

However, it is certainly possible to apply the two proposed methods in sequence on a single design. The following is a discussion on a proposed optimization flow for doing so.

In most real-life design problems, the most important specification that needs to be satisfied is usually the timing constraint – there is no point in having a low-power design if it does even meet its timing specification. Bearing this in mind, and given that the proposed optimal decoupling method is in essence a timing optimization method, it is intuitive that the optimal decoupling method should be applied on the given design first, followed by the proposed handshake component fusion method. A proposed optimization flow of applying both methods on a single design is shown in Fig. 3.18.

As shown in Fig. 3.18, the given control network (with annotated timing information) is first parsed to see if it satisfies the specified minimum cycle time constraint. If the minimum cycle time constraint is satisfied, then the optimal decoupling step is skipped and handshake component fusion is directly applied on the control network. If the minimum cycle time constraint is violated, then optimal decoupling is applied on the control network to improve its timing performance.

In the event that the optimal decoupling step does not return a control network that satisfies the minimum cycle time constraint, designer intervention would be required to carry out further architectural or datapath optimization on the design, such as increasing the number of pipeline stages or reducing the delays across combinational logic blocks.
These optimizations can be guided by the information generated by the timing analysis performed on the control network, such as the composition of the critical cycle in the control network. The improved design is then passed back to the optimization flow.

Once the minimum cycle time constraint is met by optimal decoupling, the resulting control network is passed on to the handshake component fusion tool so as to reduce its size and power dissipation, while still satisfying the minimum cycle time constraint.

As a demonstration of the proposed optimization flow, consider the parallel prefix tree discussed in Section 3.5.4.1 and Section 3.6.3.1. According to the optimization flow, the control network of the tree would first be subject to a timing analysis to determine if the minimum cycle time constraint is met. Assuming that the minimum cycle time constraint is set at 7.5 ns, the original control network, which consists of only minimally-concurrent handshake components and has a minimum cycle time of 9.24 ns
(see Table 3.6), would fail the timing analysis. The control network would therefore be subject to optimal decoupling to improve its minimum cycle time. As reported in Section 3.6.3.1, the optimally-decoupled control network has a minimum cycle time of 7.01 ns and therefore satisfies the minimum cycle time constraint.

The optimally-decoupled control network is then passed on for handshake component fusion to reduce its size and power dissipation, with a minimum cycle time constraint of 7.5 ns (a 7% relaxation in minimum cycle time). The resulting control network after handshake component fusion is shown in Fig. 3.19.
Table 3.9
Comparisons of 16-Input Pipelined Parallel Prefix Tree at Various Stages of Proposed Optimization Flow

<table>
<thead>
<tr>
<th></th>
<th>( N_T(k) )</th>
<th>( E ) (pJ)</th>
<th>( \tau ) (ns)</th>
<th>( E\tau^2 ) ( \times 10^{-26} ) J s^2</th>
<th>( N_T\tau ) ( \times 10^{-4} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (MINC)</td>
<td>3.71</td>
<td>12.5</td>
<td>Control 185</td>
<td>Total 336</td>
<td>9.24</td>
</tr>
<tr>
<td>OPTD</td>
<td>4.65</td>
<td>13.4</td>
<td>Control 219</td>
<td>Total 360</td>
<td>7.01</td>
</tr>
<tr>
<td>Final</td>
<td>3.15</td>
<td>11.9</td>
<td>Control 145</td>
<td>Total 283</td>
<td>7.14</td>
</tr>
</tbody>
</table>

\( N_T \): transistor count
\( E \): av. energy per computation
\( \tau \): minimum cycle time

Table 3.9 compares the parallel prefix tree when it is at various stages of the optimization flow. As shown in Table 3.9, the optimally-decoupled (OPTD) tree is 7% higher in transistor count and dissipates 7% more energy than the original (MINC) tree. However, the former is 24% faster and 38% better in \( E\tau^2 \) than the original tree (whether the increases in transistor count and energy dissipation are worthwhile or not depends on the minimum cycle time constraint of the design; in this particular example, the increases are necessary in order to satisfy the minimum cycle time constraint).

After handshake component fusion, the Final tree has its transistor count and energy dissipation reduced by 11% and 21%, respectively, compared with the OPTD tree, but lost only 2% of its speed. When compared with the original tree, the Final tree is 23% faster, dissipates 16% less energy (resulting in a better \( E\tau^2 \) figure by 51%), and has 5% fewer transistors.

3.8. Summary

In this chapter, two optimization methods – handshake component fusion and optimal decoupling – for reducing the circuit areas and power dissipation of asynchronous
control networks while satisfying given pipeline throughput constraints have been proposed, developed, and automated.

The proposed handshake component fusion method is a form of peephole optimization that iteratively selects a pair of optimization targets that share input channel sources or output channel destinations and replaces them with a single component of the same type. More specifically, the following work on the proposed handshake component fusion method has been described.

First, a heuristic algorithm for the selection of the optimization targets has been described. In essence, the selected optimization targets at each optimization iteration are those that are judged by the algorithm to be least likely to cause a change in the minimum cycle time of the pipeline.

Second, the conditions that must be satisfied in order for the fusion of the optimization targets to be accepted by the optimization process have been described. The first condition — satisfaction of minimum cycle time constraint — involves the computation of the minimum cycle time of the restructured pipeline. In particular, a procedure has been described for taking into consideration the effect of additional capacitive loading taken on by latch enable signals due to handshake component fusion. The second condition — flow equivalence between the original and restructured pipelines — is related to preserving the behavior of the pipeline under optimization. In particular, it has been formally proven that the original and restructured pipelines are flow equivalent if there is no channel link between the optimization targets.

Third, the application of the proposed handshake component fusion method on the control networks of three asynchronous designs have been described. The designs are a 16-input pipelined parallel prefix tree (for the addition operation), a four-bit cross-pipelined array multiplier, and a Reed-Solomon error detector for the compact-disc
player. For the three optimization examples, the results attained by the proposed handshake component method have been encouraging. Compared with the original asynchronous control networks, the optimized control networks, on the average, have 49% fewer transistors and dissipate 48% less energy, while sacrificing at most 6% of their throughputs.

The optimal decoupling method has been proposed to resolve the dilemma when designing asynchronous pipelines between using small handshake components to reduce asynchronous control overheads and satisfying throughput constraints. More specifically, the following work on the proposed optimal decoupling method has been described.

First, a branch-and-bound algorithm has been described that searches for the optimal mix of handshake components of different degree of concurrency in a given asynchronous control network. The objective of the algorithm is to incur the least circuit area and power dissipation for the control network, while satisfying a given minimum cycle time constraint.

Second, the efficacy of the proposed optimal decoupling method has been demonstrated through three optimization examples: a 16-input pipelined parallel prefix tree (for the addition operation), a four-bit cross-pipelined array multiplier, and a Reed-Solomon error detector for the compact-disc player. For the three optimization examples, the optimally-decoupled asynchronous control networks, on the average, have 22% fewer transistors and dissipate 32% less energy when compared with the other control networks based on uniform handshake component concurrency (except the minimally-concurrent control network). Although the optimally-decoupled control networks are larger and dissipate more energy than their minimally-concurrent counterparts, they are at least 24% faster. In addition, the circuits with optimally-
decoupled control networks are at least 38% better in $E^2$ than those with minimally-concurrent control networks.
4

Petri Net Invariant Computation

4.1. Introduction

The optimization methods for asynchronous control networks proposed in Chapter 3 operate on the timed PN models of the control networks. To be more specific, the proposed optimization methods rely heavily on the analysis of the minimal-support S-invariants of the PN models. For the proposed handshake component fusion method, minimal-support S-invariants are used by the optimization target selection algorithm to select the handshake components to fuse. For the proposed optimal decoupling method, minimal-support S-invariants are used by the branch-and-bound algorithm to select the branching variable (i.e., the handshake component to decouple). Minimal-support S-invariants are also required by both optimization methods to compute the minimum cycle time of the PN models.

As explained in Chapter 3, the proposed optimization methods make incremental changes to the control network under optimization at each iteration of the optimization
Fig. 4.1. An “exponential net” with $a$ transitions and $b$ places in the pre-set and post-set of each transition.

process. This means that at each optimization iteration, the PN model of the control network is modified and, consequently, all of its minimal-support S-invariants need to be recomputed.

This poses a significant challenge for the feasibility of the proposed optimization methods in terms of their time and memory requirements because it is known that, in the worst case, the number of minimal-support S-invariants for a PN is exponential in the number of places in the net [142]. More specifically, the number of minimal-support S-invariants for a PN is upper bounded by the combinatory number $\binom{n}{\lceil n/2 \rceil}$ [142], where $n$ denotes the number of places in the net and $\lceil n/2 \rceil$ denotes rounding $n/2$ up to an integer. This implies that PNs, even small ones, can have disproportionately large number of minimal-support S-invariants. An example of such a net is the so-called “exponential net”, as shown in Fig. 4.1. An “exponential net” has $a$ transitions and $b$ places in the pre-set and post-set of each transition. For such a net, the number of minimal-support S-invariants is $b^{a+1}$. Thus, even for a small “exponential net” with, say, $a = 10$ and $b = 3$, there are close to 60,000 minimal-support S-invariants. Given that the time and memory requirements for computing the minimal-support S-invariants of PNs can be substantial, it is clear that the feasibility of the proposed optimization methods is
critically dependent on employing an efficient S-invariant computation method for the purpose of minimum cycle time analysis.

In this chapter, a fast and memory-efficient algorithm for computing all minimal-support S-invariants of ordinary PNs is proposed (this work has been published in *IET Proc. Computers and Digital Techniques* [143]). The proposed algorithm is based on the Fourier Motzkin (FM) algorithm, a widely-known algorithm for computing the S-invariants of general PNs. Ordinary PNs are a special case of PNs whose arc weights are all 1’s (note that the optimization methods proposed in Chapter 3 operate on ordinary Petri nets).

This chapter is organized as follows. Reported algorithms for S-invariant computation are first reviewed and their problems relating to execution time and memory requirement discussed. This is followed by a detailed description of the proposed algorithm, including formal proofs of its correctness and an analysis on its time complexity. An application example is then provided to illustrate the main ideas behind the algorithm. Finally, experimental results on the time and memory requirement of the proposed algorithm are reported and compared with those of comparable reported algorithms.

### 4.2. Literature Review

Many methods for S-invariant computation have been reported in the literature [142][144][145][146][147][148][149][150][151]. In [144], an algorithm based on linear programming is reported. However, as the algorithm does not compute all minimal-support S-invariants, it is not suitable for the present purpose.

In [145], the system of homogenous equations based on which the S-invariants are defined are reduced to an equivalent form and solutions in different spaces are
computed. Although this method has low computational requirements, it does not guarantee a solution.

In [146], a parallel method is reported that decomposes the incidence matrix of the PN into several subsystems, each of which is then computed separately to generate the invariants. This method, however, can only be implemented on a multiprocessor system.

By far the most popular method for computing the S-invariants of PNs is the Fourier-Motzkin (FM) algorithm [142][147][148][149][150][151] (the FM algorithm is discussed in detail in Section 4.4). When executed with sufficient memory, the FM algorithm guarantees the computation of all minimal-support S-invariants of the given PN. However, the FM algorithm, in its original form, also generates nonminimal-support S-invariants, which are unnecessary because any S-invariant of a PN can always be written as a linear combination of the PN's minimal-support S-invariants [87]. As a result, various optimizations of the FM algorithm have been reported that reduce the number of nonminimal-support S-invariants generated [142][147][148][149][150][151].

In [150], the first two phases of the FM algorithm's implementation are restricted to polynomial complexity, with only the third and last phase having exponential complexity. This implementation, however, still generates a substantially large number of nonminimal-support S-invariants and, thus, is disadvantageous in execution time and memory usage.

In [147][148][149], several algorithms were reported for extracting minimal siphon-traps, which are used as a basis for computing S-invariants. This significantly reduces the number of candidate vectors since S-invariant computation is restricted to considering only place sets that are minimal siphon-traps. However, since the support of a minimal-support S-invariant is not necessarily a minimal siphon-trap, this
optimization technique does not guarantee the computation of all minimal-support S-invariants.

In [142][151], several optimizations for the FM algorithm were reported: i) choosing to annihilate the transition that results in the smallest increase in candidate vectors; ii) deleting all nonminimal-support candidate vectors as they are generated; iii) deleting the coverable candidate vectors; and iv) partially deleting the nonminimal-support invariants. In the rest of this chapter, the algorithm incorporating the first and second optimization techniques is denoted as FM1, and that incorporating the first, third, and fourth optimization techniques is denoted as FM2. It is of interest to note that, on the average, FM1 is reportedly the fastest variant of the FM algorithm prior to this work.

4.3. Salient Features of Proposed Algorithm

In this chapter, it is argued that FM1 can be further improved by exploiting the presence of parallel places that are created in the PN during the execution of the algorithm. Parallel places are those that have the same input and output transitions, and have the same corresponding arc weights.

The argument is primarily based on the observation that for a set \( S \) of parallel places, if there exists an S-invariant \( y_1 \) whose support contains \( p_1 \in S \), then there also exist \( k - 1 \) invariants \( y_i \) for \( i = 2, 3, \ldots, k \), where \( k = |S| \), such that the support of \( y_i \) contains a place \( p_i \in S \). This observation is formalized and proven in Section 4.5. It can be deduced from this observation that one merely needs to compute \( y_1 \) and then obtain the other corresponding invariants through an enumerative process. By doing so, the invariant computation process can potentially be expedited.
However, in order for this idea to work, a significant proportion of the new places that are created during the execution of the FM algorithm has to be parallel to each other. As shall be discussed in Section 4.7 later, tests on a large number of randomly-generated ordinary PNs indicate that it is indeed the case that a significant proportion (close to 50%) of the new places created are parallel to each other.

An improved implementation of the FM algorithm is proposed in this chapter that exploits the presence of parallel places in the PN during the execution of the algorithm. The proposed algorithm computes all minimal-support S-invariants of PNs, thus making it suitable for incorporation into the optimization methods proposed in Chapter 3.

It is of significance to note that although the work on the proposed algorithm has been motivated by the need to improve the feasibility of the proposed optimization methods, the proposed algorithm is in itself of value to the analysis of PNs in general. More specifically, the proposed algorithm provides a fast and memory-efficient solution to the general problem of finding all minimal-support S-invariants of ordinary PNs. Invariant computation is an important problem in PNs because they are valuable tools in studying the structural properties of PNs. Structural properties of PNs are those that are dependent on the topological structures of the nets, but are independent of the initial states of the nets, such as: i) structural liveness, which indicates whether there exists a live initial state for the net; and ii) repetitiveness, which indicates whether there exists for the net an initial state $M_0$ and a transition firing sequence $\sigma$ from $M_0$ such that every transition occurs infinitely in $\sigma$.

4.4. Conventional Fourier-Motzkin Algorithm

This section provides an explanation of the conventional FM algorithm. Intuitively, the FM algorithm can be viewed as a PN transformation algorithm, in which places are
created and deleted, while preserving the set of S-invariants of the original net. Each new place accumulates a portion of the original net and the composition of the new place is recorded using an invariance matrix which augments the incidence matrix.

The conventional FM algorithm is formally stated as follows, where $A$ is the incidence matrix of the net, $I_n$ is the $n$-dimensional identity matrix, $n$ is number of places in the net, $D$ is termed the invariance matrix, and $m$ is the number of transitions in the net.

1) Initializations: $C = A; D = I_n$.

2) For $j = 1$ to $m$, do the following:
   1.1) Append to $B = [C : D]$ the rows resulting from positive linear combinations of row pairs in $B$ that annihilate column $j$ of $B$.
   1.2) Delete any row $i$ of $B$ if its $j$-th element is nonzero.

In the algorithm, $C$ denotes the incidence matrix of the net as it is being transformed. $D$ denotes the invariance matrix, which records the "composition" (or, formally, the support), in terms of the places of the original net, of the new places that are created after each transition annihilation. More specifically, each column of $D$ corresponds to a place of the original net and each row of $D$ corresponds to either a place of the original net or a new place (note that $D$ is initialized at the beginning of the algorithm as the $n$-dimensional identity matrix $I_n$, where $n$ is the number of places in the original net). The row of $D$ (let it be $\text{Row}_j(D)$) that corresponds to a new place is given by the positive linear combination of a pair of existing rows of $D$. The support of the new place is the set of places (of the original net) corresponding to the positive elements of $\text{Row}_j(D)$. The rows remaining in $D$ at the end of the algorithm are the S-invariants of the net. For each step in the algorithm, a transition $t$ with at least one input place and one output place is selected. A new place $p_t$ is then added for each pairing of an input
place $p_j$ of $t$ with an output place $p_k$ of $t$ such that the input transition of $p_i$ is that of $p_j$ and the output transition of $p_i$ is that of $p_k$. Finally, the input and output places of the transition are deleted. A transition is annihilated after all its input and output places have been deleted. The algorithm completes after the annihilation of all transitions.

4.5. Proposed Algorithm

This section describes the proposed algorithm, which consists of the net transformation phase and the invariant computation phase.

4.5.1. Phase 1: Net Transformation

The net transformation phase of the proposed algorithm is the same as the conventional FM algorithm except for the additional steps of detecting parallel places in the net after each transition has been annihilated, and the replacement of each set of parallel places with a representative parallel place. It is important to note that these additional steps are not just performed on the original net. Instead, they are performed on the transformed net every time a transition is annihilated (i.e., after each iteration of the conventional FM algorithm). Thus, the parallel places that are of interest include not just the places of the original net, but also the new places created by transition annihilations. At the end of the net transformation phase, the remaining candidate vectors are passed to the second phase.

In the following, $N$ denotes a PN, $n$ and $m$ denote the number of places and transitions, respectively, in $N$, $\text{Row}_i(A)$ and $\text{Col}_j(A)$ denote row $i$ and column $j$, respectively, of the matrix $A$, $C$ and $D$ denote the incidence matrix and invariance matrix, respectively, of the net undergoing transformation, and the matrix $B=[C:D]$. 
Definition 4.1: Two places $p_i$ and $p_j$ are said to be parallel if their corresponding rows in the incidence matrix $C$ are identical, i.e., $\text{Row}_i(C) = \text{Row}_j(C)$, where $\text{Row}_i(C)$ and $\text{Row}_j(C)$ are the rows of $C$ corresponding to $p_i$ and $p_j$, respectively.

Intuitively, parallel places have the same input and output transitions, and the weights on their corresponding input and output arcs are identical. For an example of parallel places, consider the PN fragment of Fig. 4.2. The places $p_1$ and $p_2$ are parallel to each other because they have the same input transition $t_1$, the same output transition $t_2$, and the weights on their corresponding input and output arcs are the same, i.e., $w(t_1, p_1) = w(t_1, p_2) = 1$ and $w(p_1, t_2) = w(p_2, t_2) = 1$.

As specified in the formal description of the conventional FM algorithm in Section 4.4, the incidence matrix $C$ is modified every time a transition is annihilated. Thus, it should be clear from Fig. 4.2 that the detection of parallel places and their replacement with representative parallel places are not merely preprocessing steps that are performed only on the places of the original net $N$. Instead, the additional steps are performed on the transformed net every time a transition $t$ is annihilated and takes into consideration the new places that are created by the linear combinations of the input and output places of $t$.

For an example of new places that are parallel to each other, consider the PN fragments of Fig. 4.3. The new places $mp_1$ and $mp_2$ of Fig. 4.3(b) are created by the fusion of the places $\{p_1, p_2\}$ and $\{p_3, p_4\}$, respectively, of Fig. 4.3(a). $mp_1$ and $mp_2$ are
parallel to each other.

Each set of parallel places that are detected after each transition annihilation are replaced with a new place, termed the representative parallel place, which has the same input and output transitions as the parallel places. For each representative parallel place \( pp \) that is created, a new column that corresponds to \( pp \) is appended to the invariance matrix \( D \) (see Section 4.4 for both formal and intuitive descriptions of \( D \)). The new column contains all 0's except for the element that corresponds to \( pp \), which is set to 1.

**Definition 4.2 :** A representative parallel place, denoted as \( pp \), is a new place that is created to replace a set of parallel places. The row vector that is appended to the incidence matrix \( C \) to represent the representative parallel place is given by a row vector of \( C \) that corresponds to any of the parallel places. The row vector that is appended to the invariance matrix \( D \) to represent the representative parallel place is given by \( Row_i(D)[k]=0 \) for \( k = 1, 2, \ldots, n + n_{pp} - 1 \) and \( Row_i(D)[n+n_{pp}]=1 \), where \( n_{pp} \) is the number of representative parallel places that have been created.

For an example of how parallel places are replaced with a representative parallel place, consider the PN fragments of Fig. 4.3. The parallel places \( mp_1 \) and \( mp_2 \) are replaced with the representative parallel place \( pp_1 \). Note that \( pp_1 \) has the same input and output transitions as \( mp_1 \) and \( mp_2 \).
Definition 4.3: $Q_p$ denotes the set of all representative parallel places introduced during the net transformation phase.

For each representative parallel place that is created, an entry is made in the lookup table $R$. The lookup table $R$ provides a record of the representative parallel places that have been created and the parallel places that they represent.

Definition 4.4: $R$ denotes a lookup table for representative parallel places such that the keys for $R$ are the representative parallel places (i.e., $\text{key}(R) = Q_p$) and the value that is associated with each key $pp$ is the set of parallel places $S$ that have been replaced with $pp$ (i.e., $R(pp) = S$).

For an example of making an entry in $R$, consider the PN fragments of Fig. 4.3. After the creation of the representative parallel place $pp_1$ to replace the parallel places $mp_1$ and $mp_2$, the following entry is made in $R$: $R(pp_1) = \{mp_1, mp_2\}$.

For each set of parallel places $S$, an entry is made in the lookup table $U$ for each macro place that is found in $S$. A macro place refers to a new place that is created during transition annihilation.

Definition 4.5: A macro place, denoted as $mp$, is a place that corresponds to a row $\text{Row}_h(B)$ of the matrix $B$ that results from the positive linear combination of a row pair $\{\text{Row}_i(B), \text{Row}_j(B)\}$ in $B$, i.e., $\text{Row}_h(B)[k] = \text{Row}_i(B)[k] + \text{Row}_j(B)[k]$ for $k = 1, 2, \ldots, l$, where $l$ is the number of columns in $B$. Its support $\|mp\|$ comprises representative parallel places and places of the original net, and is given by the positive elements of the row of the invariance matrix $D$ corresponding to the macro place, i.e., $\text{Row}_h(D)[k] > 0 \Rightarrow p_k \in \|mp\|$, where $p_k$ is a representative parallel place or a place of the original net that corresponds to the index $k$. 

176
For an example of the creation of macro places, consider the PN fragments of Fig. 4.3. Since the macro place \$mp_1\$ is created by the fusion of the places \$p_1\$ and \$p_2\$, its corresponding row \$Row_h(B)\$ in the matrix \$B\$ is given by the summation of \$Row_1(B)\$ and \$Row_2(B)\$, which correspond to \$p_1\$ and \$p_2\$, respectively. The row representing \$mp_1\$ in the invariance matrix \$D\$ is given by \$Row_h(D) = [1 1 0 0 ... 0]\$, where the first (and only) two positive entries correspond to \$p_1\$ and \$p_2\$, respectively. The support of \$mp_1\$ is given by \$\|mp_1\| = \{p_1, p_2\}\$.

**Definition 4.6:** \$Q_m\$ denotes the set of macro places that have been replaced with representative parallel places.

**Definition 4.7:** \$U\$ denotes a lookup table for macro places such that the keys for \$U\$ are the macro places that have been replaced with representative parallel places (i.e., \$key(U) = Q_m\$) and the value that is associated with each key \$mp\$ is the row \$Row_h(D)\$ of the invariance matrix \$D\$ that corresponds to \$mp\$ (i.e., \$U(mp) = Row_h(D)\$).

For an example of making an entry in the lookup table \$U\$, consider the PN fragments of Fig. 4.2. The parallel place set \$\{mp_1, mp_2\}\$ are replaced with the representative parallel place \$pp_1\$. Since \$mp_1\$ and \$mp_2\$ are macro places, the following entries are made in \$U\$: \$U(mp_1) = Row_1(D) = [1 1 0 0 ... 0]\$, where \$Row_1(D)\$ corresponds to \$mp_1\$ and its first two entries correspond to \$p_1\$ and \$p_2\$, respectively, and \$U(mp_2) = Row_j(D) = [0 0 1 1 ... 0]\$, where \$Row_j(D)\$ corresponds to \$mp_2\$, and its third and fourth entries correspond to \$p_3\$ and \$p_4\$, respectively.

The net \$N'\$ is obtained from \$N\$ by adding to the latter the representative parallel places and macro places that are created during the net transformation phase. \$N'\$ is formally defined through its incidence matrix \$A'\$ (note that it is \$A'\$, not \$N'\$).
explicitly created during the execution of the algorithm).

**Definition 4.8**: $A'$ denotes the incidence matrix of $N'$ and is constructed as follows.

At the beginning of the algorithm, $A'$ is initialized as $A$, the incidence matrix of $N$. Subsequently, for each representative parallel place $pp$ or macro place $mp$ that is created, the row in the incidence matrix $C$ that corresponds to $pp$ or $mp$ is appended to $A'$.

For an example of the construction of $N'$, consider the PN fragments of Fig. 4.3. The macro places that are created are $mp_1$ and $mp_2$. The representative parallel place that is created is $pp_1$. If Fig. 4.3(a) is denoted as $N$, then the corresponding $N'$ is constructed by adding $pp_1$, $mp_1$, and $mp_2$ to $N$ such that $N'$ is as shown in Fig. 4.4.

At the end of the net transformation phase, the remaining row vectors in the invariance matrix $D$ are passed to the second phase of the algorithm.

The net transformation phase of the proposed algorithm is formally stated as follows:

/* Phase 1: Net Transformation */

1) Initializations: $C = A$, $D = I_m$, $A' = A$.

2) For $i = 1$ to $m$, do the following:
/* Replace each parallel place set with a representative parallel place */

2.1) For each set \( S_j \) of parallel places in \( C \), do the following:

2.1.1) If there exists a parallel place \( pp_k \) in \( S_j \), then add \( \{ S_j \backslash pp_k \} \) to \( R(pp_k) \); otherwise, do the following:

2.1.1.1) Add a representative parallel place \( pp_k \) to \( Q_p \) and set \( R(pp_k) = S_j \). Arbitrarily select a place \( p_l \in S_j \) to be \( pp_k \).

2.1.1.2) Set the row of \( D \) corresponding to \( p_l \) to all 0's, append to \( D \) a zero column, and set the entry that corresponds to \( p_l \) of the new column to 1.

2.1.1.3) Append the row of \( C \) corresponding \( p_l \) to \( A' \).

2.1.2) Append to \( U \): i) a zero column if a representative parallel place has been created; and ii) the rows of \( D \) that correspond to the macro places in \( S_j \).

2.1.3) Delete the rows of \( B \) that correspond to \( \{ S_j \backslash pp_k \} \).

/* Execute one iteration of FM1 */

2.2) Select \( Col_k(C) \) such that its annihilation will lead to the smallest increase in the number of rows in \( C \), i.e., \( Col_k(C) \) is selected such that

\[ f_k = \min_j \{ f_j \} \]

where \( f_j = p_j \cdot n_j - p_l \cdot n_l \) is the maximum increase in the number of rows in \( C \) after annihilating \( Col_j(C) \), and \( p_j \) and \( n_j \) are the number of positive and negative elements, respectively, in \( Col_j(C) \).

2.3) Append to \( B \) the rows resulting from positive linear combinations of row pairs in \( B \) that annihilate \( Col_k(C) \).
2.4) Delete any \( \text{Row}_j \) if its \( k \)-th element is nonzero or if \( \text{Row}_j(D) \) has a nonminimal support (i.e., \( \text{check\_support}(\|\text{Row}_j(D)\|, A') > 1 \), where \( A' \) denotes the matrix comprising the columns of \( A' \) that correspond to the transitions annihilated thus far).

3) Append \( |Q_m| \) zero columns to \( D \).

Note that in the above algorithm, a method reported in [151] is used to determine if the support of a candidate vector \( \text{Row}_j(D) \) is minimal. The method involves constructing a matrix \( H \) whose rows are given by \( \text{Row}_i(A') \) such that the index \( k \) satisfies \( \text{Row}_j(D)[k] > 0 \) for \( k = 1, 2, \ldots, \alpha \), where \( \alpha \) denotes the size of \( \text{Row}_j(D) \), and then determining if \( q = \text{rank}(H) + 1 \), where \( q \) denotes the size of \( \|y\| \) and \( \text{rank}(H) \) denotes the rank of \( H \). If so, then \( \text{Row}_j(D) \) has a minimal support. If, on the other hand, \( q > \text{rank}(H) + 1 \), then \( \text{Row}_j(D) \) is deemed to have a nonminimal support. In the above algorithm, the method is implemented as the subroutine \( \text{check\_support}(\|\text{Row}_j(D)\|, A') \) (see Step 2.4), which returns the value of \( q - \text{rank}(H) \). Thus, if \( \text{check\_support}(\|\text{Row}_j(D)\|, A') > 1 \), then \( \text{Row}_j(D) \) is deemed to have a nonminimal support and is deleted.

4.5.2. Phase 2: Invariant Enumeration

In the second phase of the proposed algorithm, termed the invariant enumeration phase, the row vectors of the invariance matrix \( D \) remaining at the end of Phase 1 of the algorithm are used to enumerate all minimal-support invariants of the given net \( N \). In essence, the second phase is an enumerative process in which each candidate vector is
used to spawn, with respect to the representative parallel places in its support, other candidate vectors. These vectors are in turn used to generate more vectors based on the same procedure if their supports contain representative parallel places. A vector that is generated during the enumerative process whose support contains only the places of the original net is a minimal-support invariant of the net.

The first step in the enumeration process is to express, in a mathematical form, the relations between the representative parallel places, the macro places (both created in Phase 1 of the algorithm), and the places of the original net. For this purpose, a PN, denoted as $G$, is defined as follows (note that in the algorithm, it is not $G$ that is actually derived, but its incidence matrix, denoted as $E$).

**Definition 4.9**: $G$ is a PN such that: i) $P(G) = P(N) \cup Q_p \cup Q_m$, where $P(G)$ and $P(N)$ are the places of $G$ and $N$, respectively; ii) for each representative parallel place $pp_i \in Q_p$ and each place $p_j \in R(pp_i)$, a transition $t_{ij}$ (the subscripts $i$ and $j$ correspond to $pp_i$ and $p_j$, respectively) is introduced such that $\bullet t_{ij} = \{pp_i\}$, $\bullet t_{ij} = \{p_j\}$, and $w(pp_i, t_{ij}) = w(t_{ij}, p_j) = 1$ (see Fig. 4.5(a)); and iii) for each macro place $mp_h \in Q_m$ and for each place $p_k \in U(mp_h)$, a transition $t_{hk}$ (the subscripts $h$ and $k$ correspond to $mp_h$ and $p_k$, respectively) is introduced such that $\bullet t_{hk} = \{mp_h\}$, $\bullet t_{hk} = U(mp_h)[l]$, $w(mp_h, t_{hk}) = 1$, and $w(t_{hk}, p_k) = U(mp_h)[l]$, where $l$ is the index that corresponds to $p_k$ (see Fig. 4.5(b)).

**Definition 4.10**: $E$ denotes the incidence matrix of the PN $G$.

For an example of the construction of the net $G$, consider the PN fragments in Fig. 4.3. Let $N$ denote the net in Fig. 4.3(a). The net $G$, as shown in Fig. 4.6, that corresponds to $N$ is constructed as follows. It is easy to see that $P(N) = \{p_1, p_2, p_3, p_4\}$,
Fig. 4.5. Substructures of PN $G$ associated with (a) a representative parallel place and (b) a macro place.

Fig. 4.6. PN $G$ representing the relationships between the places in Fig. 4.3.

$Q_m = \{mp_1, mp_2\}$, and $Q_p = \{pp\}$. Thus, $P(G) = \{p_1, p_2, p_3, p_4, mp_1, mp_2, pp\}$. Given that $R(pp) = \{mp_1, mp_2\}$, two transitions $t_1$ and $t_2$ are introduced in $G$ and the arcs $(pp, t_1)$, $(t_1, mp_1)$, $(pp, t_2)$, and $(t_2, mp_2)$ are drawn (note that despite the same labels, the transitions of $G$ are not related to those in Fig. 4.3). Given that $\|U(mp_1)\| = \{p_1, p_2\}$, a transition $t_3$ is introduced in $G$ and the arcs $(mp_1, t_3)$, $(t_3, p_3)$, and $(t_3, p_2)$ are drawn.

Likewise, given that $\|U(mp_2)\| = \{p_3, p_4\}$, a transition $t_4$ is introduced in $G$ and the arcs $(mp_2, t_4)$, $(t_4, p_3)$, and $(t_4, p_4)$ are drawn.

In essence, the invariant enumeration process is a partial generation of the state space of net $G$, using the row vectors of the invariance matrix $D$ remaining at the end of the net transformation phase of the algorithm as the initial markings. The state space of
$G$ is only partially generated due to the special transition firing rule (see below) employed by the algorithm, as well as the heuristic rules (see below) proposed to reduce the state space. Note that all markings on $G$ are invariants of the net $N'$ (see Theorem 4.1 and 4.2 later). Note also that a dead-end marking $M$ on $G$ (i.e., one that has no enabled transitions) is an invariant of the net $N$ because the support of $M$ comprises only the places of $N$.

Definition 4.11: An intermediate invariant of $N'$, denoted as $y'$, is a marking on $G$ whose support contains at least one macro place $mp \in Q_m$ or representative parallel place $pp \in Q_p$.

The conventional transition firing rules of PNs (see Chapter 2, Section 2.3.3.1) are observed when constructing the state space of $G$, but with the following exception: when an enabled transition is selected to fire, it does so consecutively until it becomes disabled. Put differently, the firing count of an enabled transition that has been selected to fire is given by the number of tokens at its input place in the marking that immediately precedes its firing.

The recursive subroutine $\text{generate_invariants}(\ )$ is the main engine for the invariant enumeration process. It comprises two main tasks: parallel enumeration and macro refinement.

Parallel enumeration is considered first. Consider a marking $M$ on $G$ that has in its support a representative parallel place $pp$ such that the weight of $pp$ in $M$ is $w$. To generate the succeeding markings of $M$ (the number of succeeding markings is given by $|R(pp)|$), each output transition of $pp$ in $G$ is fired $w$ times consecutively. This means that in the support of each succeeding marking, $pp$ is replaced with a place $p, \in R(pp)$. Based on the following theorem, the succeeding markings remain as invariants of $N'$.
Theorem 4.1: If a PN $N$ has two places $p_h$ and $p_k$ that are parallel to each other, and an S-invariant $y_h$ such that $y_h[h] = \alpha$ and $y_h[k] = 0$, then there exists another S-invariant $y_k$ of $N$ such that $y_k[h] = 0$, $y_k[k] = \alpha$, and $y_k[l] = y_h[l]$ for $l = 1, 2, \ldots, n$, excluding $h$ and $k$, where $n$ denotes the number of places in $N$.

Proof: Let $h = 1$ and $k = 2$ (note that there is no loss of generality here because the index of a place in a PN is arbitrary), i.e., $y_1^T = [\alpha \ 0 \ \ldots]$. Let $A = [a_{ij}]$ denote the incidence matrix of $N$. Since $y_1$ is an S-invariant of $N$, we have

$$a_{ij} \alpha + \sum_{i=1}^{m} a_{ij} y_1[i] = 0, \quad j = 1, 2, \ldots, m$$

(4.1)

where $m$ denotes the number of transitions in $N$. Given that $p_1$ and $p_2$ are parallel to each other, it follows that $a_{ij} = a_{ij}$ for $j = 1, 2, \ldots, m$. Consequently, (4.1) can be rewritten as

$$a_{ij} \alpha + \sum_{i=1}^{m} a_{ij} y_1[i] = 0, \quad j = 1, 2, \ldots, m$$

(4.2)

It can be seen from (4.2) that there exists an S-invariant $y_2^T = [0 \ \alpha \ y_1[3] \ y_1[4] \ \ldots \ y_1[m]]$ of $N$.

For an example of parallel enumeration, consider a marking $M_1$ on $G$ that assigns a token to the representative parallel place $pp_1$, as shown in Fig. 4.7(a). During the parallel enumeration of $pp_1$, the transition $t_1$ is fired so that the token at $pp_1$ is removed and a token is assigned to $mp_1$, resulting in the marking $M_2$. This is followed by the firing of $t_2$, which removes the token at $pp_1$ and assigns a token to $mp_2$, resulting in the marking $M_2$. The corresponding state graph is shown in Fig. 4.7(b).

The task of macro refinement is now discussed. During macro refinement, a macro place $mp$ in $G$ marked with $w$ tokens is refined into its constituents, given by $\|U(mp)\|$,
through the firing of $mp$'s output transition $w$ times consecutively. This means that in the support of the succeeding marking, $mp$ is replaced with the places given by $\lfloor U(mp) \rfloor$. Based on the following theorem, the succeeding marking remains as an invariant of $N'$.  

Fig. 4.7. Parallel enumeration: (a) PN dynamics and (b) state graph representation.
Theorem 4.2: Let $N$ be a PN with $n$ places and $m$ transitions. Let $A = [a_{ij}]$ denote the incidence matrix of $N$. Let $mp$ be a macro place of $N$, i.e., there exists a vector $v$ of size $n$ such that $v[k] = -1$ and $\sum_{j=1}^{n} a_{ij} v[j] = 0$ for $j = 1, 2, \ldots, m$, where $k$ is the index of $mp$ in $N$. If $N$ has an $S$-invariant $y_1$ such that $y_1[k] = w$, where $w$ is the weight of $mp$ in $y_1$, then there exists another $S$-invariant $y_2 = y_1 + wv$ of $N$.

**Proof:** Let $k = 1$ (note that there is no loss of generality here because the index of a place in a PN is arbitrary). Given that $y_1$ is an $S$-invariant of $N$, we have

$$a_{ij}w + \sum_{i=2}^{n} a_{ij} y_1[i] = 0, \quad j = 1, 2, \ldots, m \quad (4.3)$$

Putting $a_{ij} = \sum_{i=2}^{n} a_{ij} v[i]$ for $j = 1, 2, \ldots, m$, into (4.3) leads to

$$\sum_{i=2}^{n} a_{ij} (y_1[i] + wv[i]) = 0, \quad j = 1, 2, \ldots, m \quad (4.4)$$

Since $y_1[1] + wv[1] = 0$, (4.4) can be extended to

$$\sum_{i=1}^{n} a_{ij} (y_1[i] + wv[i]) = 0, \quad j = 1, 2, \ldots, m \quad (4.5)$$

Thus, $y_2 = y_1 + wv$ is an $S$-invariant of $N$. 

For an example of macro refinement, consider a marking $M_1$ on $G$ that assigns a token to the macro place $mp_1$, as shown in Fig. 4.8(a). During the refinement of $mp_1$, the transition $t_3$ is fired so that the token at $mp_1$ is removed and a token is assigned to each of $p_1$ and $p_2$. Fig. 4.8 (b) shows the corresponding state graph, in which $M_2$ denotes the marking on $G$ after the macro refinement.

To reduce the execution time and memory usage of the invariant enumeration phase, two heuristic rules are proposed to reduce the state space of $G$ that is generated.
Fig. 4.8. Macro refinement: (a) PN dynamics and (b) state graph representation.

First, an intermediate invariant $y'$ is discarded if: i) it has a nonminimal support; and ii) it is projected to generate only nonminimal-support invariants. The rationale behind this heuristic rule is that we are interested only in finding the minimal-support invariants, and not those with nonminimal-supports. This is because only minimal-support invariants are required when computing the minimum cycle time of a timed Petri net model (see (2.1)).

The first condition associated with the heuristic rule is tested by the subroutine check_support( ), which has been discussed in Section 4.5.1. To test the second condition, it is necessary to examine the linearly related place sets $L$, if any, in $\|y'\|$.

Definition 4.12: $L = \{p_1, p_2, \ldots, p_k\}$, termed a linearly-related place set, denotes a subset of $\|y'\|$ ($L \neq \|y'\|$) such that its members are linearly related, i.e., there exists a set of integers $\{\alpha_1, \alpha_2, \ldots, \alpha_k\}$, not all zeros, whereby $\sum_{i=1}^{k} \alpha_i \text{Row}_i(A') = 0$, where $\text{Row}_i(A')$ corresponds to $p_i \in L$. 

187
Two observations can be made with regard to \( L \). First, the number of linearly-related place sets in \( \| y' \| \) is given by \( q - \text{rank}(H) - 1 \), where \( q \) is the size of \( \| y' \| \) and \( H \) is the matrix whose rows are given by the rows of \( A' \) that correspond to \( \| y' \| \). Second, if \( y' \) has a nonminimal support, then it has at least one linearly-related place set \( L \) in its support. This is because the value of \( q - \text{rank}(H) \) for a nonminimal-support \( y' \) is always greater than one (see the discussion on the subroutine check_support( ) in Section 4.5.1).

Based on the observation that a nonminimal-support \( y' \) (or, in general, an \( S \)-invariant) has at least one linearly-related place set \( L \), the question of whether \( y' \) will generate only nonminimal-support invariants can be reformulated as follows: given a linearly-related place set \( L \) that exists in \( \| y' \| \), does \( L \) exist in the supports of all invariants generated from \( y' \)? If the answer is affirmative, then all the invariants generated from \( y' \) have nonminimal supports and, therefore, \( y' \) can be discarded. Otherwise, \( y' \) might generate a minimal-support invariant and, therefore, cannot be discarded.

In determining if \( L \) exist in the supports of all invariants generated from \( y' \), it is useful to group \( L \) into three types. For Type I, \( L \) consists of only the original places of net \( N \) (i.e., \( L \) does not contain any macro places and representative parallel places). It is clear from the definition of the net \( G \) (see Definition 4.10) that the original places of \( N \) do not have any output transitions in \( G \). This means that the original places of \( N \) are not subject to refinement or enumeration and, thus, it can be deduced that if an \( L \) of Type I is present in \( \| y' \| \), then \( L \) will be present in the supports of all invariants that are
generated from \( y' \). As a result, it can be concluded that \( y' \) generates only nonminimal-support invariants.

For Type II, \( L \) contains a representative parallel place \( pp \) such that: i) there exists a set of nonnegative integers \( \{\beta_2, \beta_3, \ldots, \beta_k\} \) (not all 0's) such that

\[
\text{Row}_i(H) = \sum_{i=2}^{k} \beta_i \text{Row}_i(H), \quad \text{where} \quad \text{Row}_i(H) \quad \text{corresponds to} \quad pp; \quad \text{and} \quad \text{ii)}
\]

\( L \setminus \{pp\} \subset X(pp) \), where \( X \) denotes a lookup table defined as follows.

**Definition 4.13:** \( X \) denotes a lookup table such that its keys are given by \( Q_p \) (i.e., \( \text{key}(X) = Q_p \)) and the value that is associated with each key \( pp \) is given by

\[
X(pp) = \bigcup_{i=1}^{k} P(Z_i), \quad \text{where} \quad Z_i \quad \text{is a directed path that originates from} \quad pp \quad \text{in net} \quad G, \quad k \quad \text{is the number of such directed paths, and} \quad P(Z_i) \quad \text{is the set of places in} \quad Z_i.
\]

Intuitively, \( X \) is a record of all the places in net \( G \) that can be reached from a particular representative parallel place when traversing in the direction of the arcs.

For an example of making an entry in the lookup table \( X \), consider the PN \( G \) of Fig. 4.6. There are four directed paths originating from the representative parallel place \( pp_1 \):

- a) \( Z_1 \), from \( pp_1 \) to \( p_1 \) (\( pp_1 \rightarrow t_1 \rightarrow mp_1 \rightarrow t_3 \rightarrow p_1 \)),
- b) \( Z_2 \), from \( pp_1 \) to \( p_2 \) (\( pp_1 \rightarrow t_1 \rightarrow mp_1 \rightarrow t_3 \rightarrow p_2 \)),
- c) \( Z_3 \), from \( pp_1 \) to \( p_3 \) (\( pp_1 \rightarrow t_2 \rightarrow mp_2 \rightarrow t_4 \rightarrow p_3 \)), and
- d) \( Z_4 \), from \( pp_1 \) to \( p_4 \) (\( pp_1 \rightarrow t_2 \rightarrow mp_2 \rightarrow t_3 \rightarrow p_4 \)).

Thus, the entry in \( X \) for \( pp_1 \) is given by

\[
X(pp_1) = \{mp_1, mp_2, p_1, p_2, p_3, p_4\}.
\]

By considering the dynamic behavior of net \( G \), it can be seen that if an \( L \) of Type II is present in \( \|y'\| \), then \( L \) will be present (though in different forms due to macro refinement and parallel enumeration) in the supports of all invariants that are generated from \( y' \). This means that \( y' \) generates only nonminimal-support invariants.
An $L$ of Type III is similar to that of Type II except that $L \setminus \{pp\} \subset X(pp)$. By considering the dynamic behavior of net $G$, it can be deduced that an $L$ of Type III that is present in $\|y\|$ will eventually cease to exist in at least one invariant generated from $y$.

For an example of how an $L$ of Type III ceases to exist, consider a marking $y'$ on $G$ in Fig. 4.6 such that $L = \{pp_1, p_1, p_2\} \subset \|y\|$. It can be seen that one of the markings $y''$ generated from $y'$ due to the parallel enumeration of $pp_1$ has in its support a transformed version of $L$ that comprises $mp_1, p_1,$ and $p_2$ (i.e., $pp_1$ has been replaced with $mp_1$). By taking a step further and considering the refinement of $mp_1$ into $p_1$ and $p_2$, we see that the support of the succeeding marking of $y''$ will not contain $L$ in any form.

In summary, the first heuristic rule is proposed as follows. Given an intermediate invariant $y'$ of $N'$ whose support is nonminimal, if $\|y\|$ contains linearly-related place sets of Type III only, then $y'$ may generate one or more invariant whose support is minimal. In this case, $y$ is not discarded. On the other hand, if $\|y\|$ contains at least one linearly-related place set of Type I or II, then it is certain that $y'$ will generate only nonminimal-support invariants. In this case, $y'$ can be discarded without affecting the correctness of the invariant enumeration process.

The second heuristic rule proposed to reduce the state space of net $G$ involves deciding which representative parallel place $pp$ (when there is more than one) within $\|y\|$ for parallel enumeration. It is proposed that $pp$ be selected such that $|X(pp)| = \max_k \{|X(pp_k)|\}$, where the maximum is taken over all $pp_k \in \|y\|$. The rationale behind the rule is as follows.
Let \( \{ pp_1, pp_2 \} \subseteq \| y' \| \) and let there exists a directed path from \( pp_2 \) to \( pp_1 \) in net \( G \), i.e., \( X(pp_1) \subseteq X(pp_2) \). Let us first consider the case where \( pp_1 \) is selected for parallel enumeration before \( pp_2 \). It can be deduced that if one starts at the marking \( y' \) and transverses along a directed path in the state space of \( G \), one eventually reaches a point where \( pp_1 \) is selected again for parallel enumeration. This is because the parallel enumeration of \( pp_2 \) always leads to at least one marking whose support contains \( pp_1 \).

Now let us consider the other case where \( pp_2 \) is selected before \( pp_1 \). Again, if one starts at the marking \( y' \) and transverses along a directed path in the state space of \( G \), one would also eventually reach a point where \( pp_1 \) is selected for parallel enumeration. However, unlike the first case, this will be the first time that \( pp_1 \) is selected along the directed path. In short, the proposed rule can potentially reduce the size of the state space of \( G \) because it decreases the number of times a representative parallel place is selected for parallel enumeration.

For example, consider the PN \( G \) shown in Fig. 4.9. Let \( M_0 \) be an initial marking on \( G \) such that \( \| M_0 \| = \{ pp_1, pp_2 \} \). If \( pp_1 \) is selected for parallel enumeration before \( pp_2 \), then the state space generated from \( M_0 \) is as shown in Fig. 4.10(a) (note that the label associated with each arc indicates the transition that is fired to generate a marking from another). Observe that along each of the directed paths \( (M_0 \rightarrow M_1 \rightarrow M_3 \rightarrow M_5 \rightarrow M_9 \rightarrow M_{11}) \), \( (M_0 \rightarrow M_4 \rightarrow M_5 \rightarrow M_9 \rightarrow M_{12}) \), \( (M_0 \rightarrow M_2 \rightarrow M_4 \rightarrow M_7 \rightarrow M_{10} \rightarrow M_{13}) \), and \( (M_0 \rightarrow M_2 \rightarrow M_4 \rightarrow M_7 \rightarrow M_{10} \rightarrow M_{14}) \), \( pp_1 \) is selected twice for parallel enumeration. For instance, along the directed path \( (M_0 \rightarrow M_1 \rightarrow M_3 \rightarrow M_5 \rightarrow M_9 \rightarrow M_{11}) \), \( pp_1 \) is selected at \( M_0 \) and \( M_9 \).

If, on the other hand, \( pp_2 \) is selected for parallel enumeration before \( pp_1 \), then the state space generated from \( M_0 \) is as shown in Fig. 4.10(b). Observe now that \( pp_1 \) is
selected for parallel enumeration only once in any of the directed paths originating from $M_0$. The difference in the state space size of $G$ generated from $M_0$ for the two cases is apparent: for the first case, the state space consists of 19 markings; for the second case, the state space consists of 13 markings.

The invariant enumeration phase of the proposed algorithm is formally stated as follows:

/* Phase 2: Invariant Enumeration */

1) Construct $E$, the incidence matrix of the net $G$, and use it to compute the lookup table $X$.

2) For each $y'_i = Row_i(D)$, execute the recursive subroutine `generate_invariants(y'_i)` to generate invariants whose supports are minimal and comprise only places of the original net.
Fig. 4.10. The state spaces generated from $M_0$ for the PN $G$ of Fig. 4.9: (a) $pp_1$ is selected for parallel enumeration before $pp_2$; and (b) $pp_2$ is selected for parallel enumeration before $pp_1$. 

(a)

(b)
subroutine generate_invariants\( (y'_i) \)

1) If there is a macro place \( mp \) in \( \|y'_i\| \), then do the following:

/* Macro refinement */

1.1) Compute the succeeding marking of \( y'_i \) by firing \( w \) times consecutively the output transition \( t \) of \( mp \) in \( G \), where \( w \) is the weight of \( mp \) in \( y'_i \).

Specifically, execute \( y'_i = y'_i + Eu \), where \( u \) is the firing count vector that corresponds to the firing of \( t \) and is given by \( u[k] = w \), where \( k \) is the index of \( t \) in \( G \), and \( u[i] = 0 \) for \( i = 1, 2, \ldots, \varphi \), excluding \( k \), where \( \varphi \) is the number of transitions in \( G \).

/* Heuristic rule #1 */

1.2) Initialize \( Y = \|y'_i\| \) and \( h = 0 \).

1.3) Execute \( k = \text{check support}(\|y'_i\|, A') \). If \( k > 1 \), then \( y'_i \) has a nonminimal support. In this case, do the following:

1.3.1) Compute \( h \), the number of linearly-related place sets \( L \) of Type III, by doing the following for each parallel place \( pp_j \in \|y'_i\| \):

1.3.1.1) Execute \( h = h + \text{check support}(pp_j \cup S, A') \), where

\[
S = Y \cap X(pp_j).
\]

1.3.1.2) Delete \( S \) in \( Y \).

1.3.2) If \( h < k - 1 \), then there is at least one \( L \) of Type I or II in \( \|y'_i\| \). In this case, discard \( y'_i \) by exiting the subroutine.

2) If there is no representative parallel place in \( \|y'_i\| \), then \( y'_i \) is a minimal-support invariant of the original net \( N \); otherwise, do the following:
/* Heuristic rule #2 */

2.1) Select a representative parallel place \( pp_j \in \| y' \| \) such that

\[
|X(pp_j)| = \max \{ |X(pp_k)| \}, \quad \text{where the maximum is taken over all}
\]

\[
pp_k \in \| y' \| .
\]

/* Parallel enumeration */

2.2) For each place \( p_h \in R \left( pp_j \right) \), do the following:

2.2.1) Compute the succeeding marking of \( y'_j \) by firing \( w \) times consecutively the input transition \( t \) of \( p_h \) in \( G \), where \( w \) is the weight of \( p_h \) in \( y'_j \). Specifically, execute \( y'_j = y'_j + Eu \), where \( u \) is the firing count vector that corresponds to the firing of \( t \) and is given by \( u[k] = w \), where \( k \) is the index of \( t \) in \( G \), and \( u[i] = 0 \) for \( i = 1, 2, \ldots, \varphi \), excluding \( k \), where \( \varphi \) is the number of transitions in \( G \).

2.2.2) If there is a macro place or representative parallel place in \( \| y'_j \| \), then call the subroutine generate_invariants \( (y'_j) \); otherwise, \( y'_j \) is an invariant of the original net \( N \).

2.2.3) If \( y'_j \) has a minimal support (i.e., if check_support \( (\| y'_j \| , A) = 1 \)), then \( y'_j \) is a minimal-support invariant of the original net \( N \).

4.5.3. Computational Complexity

The computational complexity of the net transformation phase (Phase 1) of the proposed algorithm is an upper-bound estimate of the number of candidate vectors that are
The worst-case scenario for Phase 1 is when no parallel places are found throughout its execution. In such a situation, Phase 1 will generate as many candidate vectors as the conventional FM algorithm. Thus, the complexity of Phase 1 is the same as that of the conventional FM algorithm, which is given by $O(2^n)$, where $n$ is the number of places in the given net [147][148][149].

The computational complexity of the invariant enumeration phase (Phase 2) of the proposed algorithm is an upper-bound estimate of the total state space size of the net $G$ (recall that in Phase 2, a state space of $G$ is (partially) generated from each of its initial markings). The following estimator [152] is used to estimate the state space size, denoted as $s$, of $G$ generated from one initial marking:

$$s = \frac{(r + \alpha)!}{\tau!\alpha!}$$  \hspace{1cm} (4.6)

where $r$ is the maximum number of tokens that $G$ can have throughout the state space and $\alpha = |P(N)| + |Q_p| + |Q_m|$ is the number of places in $G$. Thus, the total state space size, denoted as $s_T$, of $G$ is

$$s_T = \sum_{i=1}^{k} s_i = \sum_{i=1}^{k} \frac{(r_i + \alpha)!}{\tau_i!\alpha!}$$  \hspace{1cm} (4.7)

where $k$ is the number of initial markings used to generate the state space of $G$ and $r_i$ is the maximum number of tokens that $G$ can have throughout the state space that is generated from the $i$-th initial marking $M'_i$.

To estimate $\tau_n$, an estimation must be performed on the number of tokens that will be gained by $G$ during its state space generation from $M'_0$. To do this, we need to consider the structure of $G$, which consists of two substructures. The first is associated with representative parallel places, as shown in Fig. 4.5(a). Given that each output transition $t_i$ of a representative parallel place $pp$ has $pp$ as its only input place and has a
single output place $p_i$, and that the weights of the arcs $(pp, t_i)$ and $(t_i, p_i)$ are ones, we can deduce that the firing of $t_i$ is not going to change the total number of tokens in $G$. Thus, the substructures associated with the representative parallel places in $G$ can be ignored when estimating $r_i$.

The second substructure of $G$ is associated with macro places, as shown in Fig. 4.5(b). Note that the output transition $t$ of a macro place $mp$ has $mp$ as its only input place and $\{U(mp)\}$ as its output places, that the weight of the arc $(mp, t)$ is one, and that the weight of the arc $(t, p_i)$ is given by the element of $U(mp)$ that corresponds to $p_i$. Hence, it can be deduced that the net gain in the number of tokens, denoted as $\delta$, in $G$ by firing $t$ once is

$$\delta = \left( \sum_{i=1}^{\beta} U(mp)[i] \right) - 1$$

(4.8)

where $\beta = |P(N)| + |Q_p|$ is the number of elements in $U(mp)$.

Since we are interested in computing the maximum possible increase in the number of tokens in $G$ during its state space generation, the following conservative assumption is made: it is assumed that every token associated with $M'_0$ leads to a transition firing sequence in which the output transition of every macro place in $X(pp_1)$, where $pp_1$ is the representative parallel place that holds the token at $M'_0$, is fired once. The lengths of the directed paths in $G$ from $pp_1$ to each macro place in $X(pp_1)$ is used to determine the order in which the output transitions of the macro places are fired (these lengths can be computed using $E$, the incidence matrix of $G$).

More specifically, the transition firing sequence $t_1t_2...t_h$, where $t_i$ is the output transition of $mp_i$ and $h$ is the number of macro places in $X(pp_1)$, implies that the path
length from \( pp_1 \) to \( mp_1 \) is the shortest compared to all other macro places in \( X(pp_1) \),
the path length from \( pp_1 \) to \( mp_2 \) is the second shortest compared to all other macro
places in \( X(pp_1) \), and so on. Fig. 4.11 shows the PN that corresponds to the transition
firing sequence described above, where \( \mu_i = \max_j (U(mp_j)[j]) \) is the maximum weight
of the arcs incident from \( t_i \) (note that the maximum arc weight is chosen so as to
maximize \( \tau_i \)).

From (4.8), we see that the net gain in the number of tokens in \( G \) after \( w_i \)
consecutive firings of \( t_i \) is

\[
\delta_i = w_i \left( \left( \sum_{j=1}^{\theta} U(mp_j)[j] \right) - 1 \right)
\]

where \( w_i \) is the number of tokens held by \( mp_i \) prior to the firing of \( t_i \). From Fig. 4.11, we
know that \( w_i = \mu_{i-1}w_{i-1} \), and \( w_{i-1} \) is, in turn, given by \( w_{i-1} = \mu_{i-2}w_{i-2} \), and so on. Thus, it
can be deduced that

\[
w_i = \lambda \left( \prod_{j=1}^{i-1} \mu_j \right) \text{, for } i > 1
\]

where \( \lambda \) is the number of tokens held by \( pp_1 \) at the initial marking on \( G \). Note that
\( w_1 = \lambda \). Consequently, we obtain the maximum total net gain in the number of tokens in
\( G \) with respect to \( pp_1 \), denoted as \( \gamma_1 \):

\[
\gamma_1 = \sum_{i=1}^{h} \delta_i
\]
Finally, by repeating the above procedure for the other representative parallel places \( \{pp_2, pp_3, \ldots, pp_l\} \) that are marked at \( M_0' \), we get

\[ \tau_i = \sum_{j=1}^{l} \gamma_j \]  

(4.12)

In summary, the above analysis shows that the proposed algorithm, like the other reported variants of the FM algorithm that compute all minimal-support invariants, has exponential computational complexity. However, two points need to be taken into account when considering the performance of the proposed algorithm. First, as shall be discussed in Section 4.7 later, tests based on a large number of randomly generated ordinary PNs indicate that, on the average, close to 50% of the new places created during Phase 1 of the algorithm are parallel to each other. This means that, in practice, the actual number of candidate vectors generated during Phase 1 is typically much lower than the conventional FM algorithm. Second, the actual state space of \( G \) that is generated during Phase 2 of the algorithm is likely to be smaller than that predicted by (4.7) due to the proposed heuristic rules.

4.6. S-Invariant Computation Example

This section provides a step-by-step illustration of how the proposed algorithm computes all minimal-support S-invariants of a given net. The PN \( N \) of Fig. 4.12(a) is used as an example. The execution of the net transformation phase of the proposed algorithm is summarized as follows:

1) Create the representative parallel places \( pp_1, pp_2, \) and \( pp_3 \) to replace the parallel place sets \( \{p_1, p_{10}\}, \{p_{13}, p_{18}\}, \) and \( \{p_5, p_{20}\} \), respectively. Create the
following entries in the lookup table $R$: $R(pp_1) = \{p_4, p_{17}\}$, $R(pp_2) = \{p_{12}, p_{18}\}$, and $R(pp_3) = \{p_5, p_{20}\}$ (see Table 4.1).

2) Annihilate transition $t_1$ by fusing $\{p_2, pp_1\}$ and $\{p_{11}, pp_1\}$ to create the macro places $mp_1$ and $mp_2$, respectively, and deleting $\{p_2, p_{11}, pp_1\}$ (see Fig. 4.12(b)).

3) Create $pp_4$ to replace the parallel places $\{p_{14}, mp_1\}$. Create the following entry in $R$: $R(pp_4) = \{p_{14}, mp_1\}$. Create an entry in the lookup table $U$ for $mp_1$ such that $\|U(mp_1)\| = \{p_2, pp_4\}$ (see Table 4.2).

4) Annihilate $t_2$ by fusing $\{p_3, pp_4\}$ and $\{p_{15}, pp_4\}$ to create $mp_3$ and $mp_4$, respectively, and deleting $\{p_3, p_{15}, pp_4\}$ (see Fig. 4.12(c)).

5) Create $pp_5$ to replace the parallel places $\{mp_2, mp_3\}$. Create the following entry in $R$: $R(pp_5) = \{mp_2, mp_3\}$ (see Table 4.1). Create entries in $U$ for $mp_2$ and $mp_3$ such that $\|U(mp_2)\| = \{p_{11}, pp_2\}$ and $\|U(mp_3)\| = \{p_5, pp_4\}$.

6) Annihilate $t_3$ by fusing $\{p_4, mp_4\}$ to create $mp_5$, and deleting $\{p_4, mp_4\}$ (see Fig. 4.12(d)).

7) Create $pp_6$ to replace the parallel places $\{p_{19}, mp_5\}$. Create the following entry in $R$: $R(pp_6) = \{p_{19}, mp_5\}$. Create an entry in $U$ for $mp_5$ such that $\|U(mp_5)\| = \{p_4, mp_4\}$.

8) Annihilate $t_4$ by fusing $\{pp_3, pp_6\}$ to create $mp_6$, and deleting $\{pp_3, pp_6\}$.

9) Annihilate $t_5$ by fusing $\{p_6, mp_6\}$ and $\{p_{21}, mp_6\}$ to create $mp_7$ and $mp_8$, respectively, and deleting $\{p_6, p_{21}, mp_6\}$.
10) Since $mp_8$ is parallel to $pp_5$, add $mp_8$ to $R(pp_5)$ and create an entry in $U$ for $mp_8$ such that $\|U(mp_8)\| = \{p_{21}, pp_5, pp_6\}$.

11) Annihilate $t_6$ by fusing $\{p_1, mp_3\}$ to create $mp_9$, and deleting $\{p_1, mp_3\}$. 

12) Since $mp_9$ is parallel to $pp_5$, add $mp_9$ to $R(pp_5)$ and create an entry in $U$ for $mp_9$ such that $\|U(mp_9)\| = \{p_6, pp_5, pp_6\}$.

13) Annihilate $t_7$ by fusing $\{p_8, pp_5\}$, $\{p_{12}, pp_5\}$, $\{p_{16}, pp_5\}$, and $\{p_{22}, pp_5\}$ to create $mp_{10}$, $mp_{11}$, $mp_{12}$, and $mp_{13}$, respectively, and deleting $\{p_8, p_{12}, p_{16}, p_{22}, pp_5\}$.

14) Annihilate $t_8$ by fusing $\{p_{17}, mp_{12}\}$ to create $mp_{14}$, and deleting $\{p_{17}, mp_{12}\}$.

15) Create $pp_7$ to replace the parallel places $\{mp_{11}, mp_{14}\}$. Create the following entry in $R$: $R(pp_7) = \{mp_{11}, mp_{14}\}$. Create entries in $U$ for $mp_{11}$ and $mp_{14}$ such that $\|U(mp_{11})\| = \{p_{12}, pp_5\}$ and $\|U(mp_{14})\| = \{p_{16}, p_{17}, pp_5\}$.

16) Annihilate $t_9$ by fusing $\{pp_2, pp_5\}$ to create $mp_{15}$, and deleting $\{pp_2, pp_5\}$.

17) Annihilate $t_{10}$ by fusing $\{p_{23}, mp_{13}\}$ to create $mp_{16}$, and deleting $\{p_{23}, mp_{13}\}$.

18) Create $pp_8$ to replace the parallel places $\{mp_{10}, mp_{16}\}$. Create the following entry in $R$: $R(pp_8) = \{mp_{10}, mp_{16}\}$. Create entries in $U$ for $mp_{10}$ and $mp_{16}$ such that $\|U(mp_{10})\| = \{p_8, pp_5\}$ and $\|U(mp_{16})\| = \{p_{22}, p_{23}, pp_5\}$.

19) Annihilate $t_{11}$ by fusing $\{p_9, pp_8\}$ and $\{p_{24}, pp_8\}$ to create $mp_{17}$ and $mp_{18}$, respectively, and deleting $\{p_9, p_{24}, pp_8\}$.

20) Annihilate $t_{11}$ by fusing $\{p_{25}, mp_{18}\}$ to create $mp_{19}$, and deleting $\{p_{25}, mp_{18}\}$.
Fig. 4.12. S-invariant computation example: transformation of PN $N$ during net transformation phase.
### Table 4.1
S-Invariant Computation Example: Lookup Table $R$ at End of Phase 1

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$PP_1$</td>
<td>${P_1, P_0}$</td>
</tr>
<tr>
<td>$PP_2$</td>
<td>${P_{13}, P_{18}}$</td>
</tr>
<tr>
<td>$PP_3$</td>
<td>${P_3, P_{20}}$</td>
</tr>
<tr>
<td>$PP_4$</td>
<td>${P_{14}, mp_1}$</td>
</tr>
<tr>
<td>$PP_5$</td>
<td>${mp_2, mp_3, mp_4, mp_5}$</td>
</tr>
<tr>
<td>$PP_6$</td>
<td>${P_{19}, mp_3}$</td>
</tr>
<tr>
<td>$PP_7$</td>
<td>${mp_11, mp_{14}}$</td>
</tr>
<tr>
<td>$PP_8$</td>
<td>${mp_{10}, mp_{16}}$</td>
</tr>
</tbody>
</table>

### Table 4.2
S-Invariant Computation Example: Lookup Table $U$ at End of Phase 1

|        | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ | $P_7$ | $P_8$ | $P_9$ | $P_{10}$ | $P_{11}$ | $P_{12}$ | $P_{13}$ | $P_{14}$ | $P_{15}$ | $P_{16}$ | $P_{17}$ | $P_{18}$ | $P_{19}$ | $P_{20}$ | $P_{21}$ | $P_{22}$ | $P_{23}$ | $P_{24}$ | $P_{25}$ | $P_{26}$ | $P_{27}$ | $P_{28}$ |
|--------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| $mp_1$ | 0     | 1     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        |
| $mp_2$ | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0        | 1        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        |
| $mp_3$ | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        |
| $mp_5$ | 0     | 0     | 1     | 0     | 0     | 0     | 0     | 0     | 0     | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        |
| $mp_8$ | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        |
| $mp_9$ | 0     | 0     | 0     | 0     | 0     | 0     | 1     | 1     | 0     | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        |
| $mp_{10}$ | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 1     | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        |
| $mp_{11}$ | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0        | 0        | 0        | 0        | 1        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        |
| $mp_{14}$ | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 1        | 1        | 0        | 0        | 0        | 0        | 0        |
| $mp_{16}$ | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        | 0        |

<table>
<thead>
<tr>
<th></th>
<th>$P_{18}$</th>
<th>$P_{19}$</th>
<th>$P_{20}$</th>
<th>$P_{21}$</th>
<th>$P_{22}$</th>
<th>$P_{23}$</th>
<th>$P_{24}$</th>
<th>$P_{25}$</th>
<th>$P_{26}$</th>
<th>$P_{27}$</th>
<th>$P_{28}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$mp_1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$mp_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$mp_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$mp_5$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$mp_8$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$mp_9$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$mp_{10}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$mp_{11}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$mp_{14}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$mp_{16}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

203
At the end of the net transformation phase, the lookup tables $U$ and $R$, and the invariance matrix $D$ are as shown in Table 4.1, Table 4.2, and Fig. 4.12, respectively. Note that the rows of $D$ correspond to the macro places $mp_{15}$, $mp_{17}$, and $mp_{19}$.

In the invariant enumeration phase, the incidence matrix $E$ of the net $G$ is derived from the lookup tables $R$ and $U$. Fig. 4.13 shows the structure of $G$. The state space of $G$ is then partially constructed using the rows of $D$ as initial markings to compute all minimal-support S-invariants of $N$.

For brevity, only part of the state space of $G$ generated from the initial marking that corresponds to $mp_{15}$ is described (see Fig. 4.14). The initial marking is labeled as $M_0$. The invariant enumeration process is as follows:

1) Perform parallel enumeration on $M_0$ with respect to $pp_7$ by firing $t_1$ and $t_2$, leading to $M_1$ and $M_2$, respectively.

2) Refine $mp_{11}$ of $M_1$ by firing $t_3$, leading to $M_3$.

3) Perform parallel enumeration on $M_3$ with respect to $pp_5$ by firing $t_9$, $t_{10}$, $t_{11}$ and $t_{12}$, leading to $M_4$, $M_5$, $M_6$, and $M_7$, respectively.

4) Refine $mp_{2}$ of $M_4$ by firing $t_{13}$, leading to $M_8$.

5) Perform parallel enumeration on $M_8$ with respect to $pp_1$ by firing $t_{23}$ and $t_{24}$, leading to $M_9$ and $M_{10}$, respectively.
Fig. 4.13. S-invariant computation example: PN $G$. 
6) Perform parallel enumeration on $M_9$ with respect to $p_{p_2}$ by firing $t_22$ and $t_26$, leading to $M_{11}$ and $M_{12}$, respectively. $M_{11}$ and $M_{12}$ are dead-end markings and minimal-support S-invariants of $N$.

4.7. Experimental Results

Experiments were carried out to evaluate the performance of the proposed algorithm and compare it against other reported FM-based algorithms. Ordinary PNs serving as test problems were randomly generated using the procedure reported in [148]. The tests are summarized as follows:

i) number of test problems (ordinary PNs): 1,527;

ii) number of places, $n$: 30–400; number of transitions, $m$: 30–400; number of arcs: 40–1600;

iii) maximum matrix size: (number of rows) × (number of columns) = $10^9$;
Table 4.4
Experimental Results for Proposed S-Invariant Computation Algorithm\(^a\)

<table>
<thead>
<tr>
<th>Phase 1</th>
<th>Phase 2</th>
<th>Improvement(^b)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>None</td>
<td>OPT 1</td>
<td>OPT 2</td>
</tr>
<tr>
<td></td>
<td>CPU Time (s)</td>
<td>0.007</td>
<td>8.053</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>0.015</td>
<td>5.059</td>
</tr>
<tr>
<td>I</td>
<td>CPU Time (s)</td>
<td>0.007</td>
<td>46.08</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>0.015</td>
<td>3.148</td>
</tr>
<tr>
<td>II</td>
<td>CPU Time (s)</td>
<td>0.008</td>
<td>113.7</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>0.020</td>
<td>5.164</td>
</tr>
<tr>
<td>III</td>
<td>CPU Time (s)</td>
<td>0.006</td>
<td>269.3</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>0.022</td>
<td>6.902</td>
</tr>
<tr>
<td>IV</td>
<td>CPU Time (s)</td>
<td>0.005</td>
<td>497.5</td>
</tr>
<tr>
<td></td>
<td>Memory</td>
<td>0.015</td>
<td>8.505</td>
</tr>
</tbody>
</table>

\(^a\)All CPU time and memory requirement results are average values.
\(^b\)Improvement provided by configuration 4 (OPT 1 & 2) over configuration 1 (None).
\(^c\)Number of 32-bit floating-point variables (x10^6).

iv) number of computed minimal-support invariants: 163–115,358; and
v) algorithms written in C and executed on a personal computer (CPU: Pentium IV 3.4 GHz; OS: Windows XP; Memory: 1.0 GB).

It is worth noting that the total number of test problems was actually 2000, a number that is close to what is typically cited in the literature. However, not all of these problems were successfully solved by the reported D'Anna algorithm [150], which tend to generate a large number of nonminimal-support invariants and, as a result, cause memory overflow (note that the available matrix size is limited to: (number of rows) x (number of columns) = 10^8; this figure is typical in the literature). The number of test problems that were successfully solved by the D'Anna algorithm (and also by the proposed algorithm and the other reported algorithms, namely FM1 and FM2) is 1527.
Table 4.5
Proportion of Parallel Places (r) Created During Phase 1 of Proposed Algorithm

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Av</td>
<td>0.51</td>
<td>0.68</td>
<td>0.55</td>
<td>0.70</td>
<td>0.56</td>
</tr>
<tr>
<td>Min.</td>
<td>0.21</td>
<td>0.37</td>
<td>0.56</td>
<td>0.44</td>
<td>0.57</td>
</tr>
<tr>
<td>Max.</td>
<td>0.68</td>
<td>0.44</td>
<td>0.68</td>
<td>0.67</td>
<td>0.59</td>
</tr>
</tbody>
</table>

The performance of the proposed algorithm, in particular the effectiveness of the proposed heuristic rules for the invariant enumeration phase, are first discussed. Table 4.4 shows the experimental results for four configurations of the proposed algorithm: i) without the heuristic rules (None); ii) with heuristic rule #1 (OPT 1) only; iii) with heuristic rule #2 (OPT 2) only; and iv) with heuristic rule #1 and #2 (OPT1 & 2).

The results are categorized into five groups based on the number of minimal-support S-invariants computed (I: < 20,000, II: 20,000–39,999, III: 40,000–59,999, IV: 60,000–79,999, V: > 80,000). The different configurations are compared based on average CPU time and memory requirements. The latter is measured in terms of the number of 32-bit floating-point variables used for the storage of matrix $B$, the lookup tables $R$ and $U$, and the intermediate invariants generated during the invariant enumeration phase (i.e., the state space of $G$). Both measurements are broken down into the net transformation phase (Phase 1) and invariant enumeration phase (Phase 2). The proportion $r$ (see Table 4.5) of parallel places among the places created during Phase 1 was also calculated using

$$r = \frac{\text{total number of parallel places}}{n + \text{total number of places created}}$$

where $n$ is the number of places originally in the net.
Table 4.6
Comparisons Between Proposed, FM1, FM2, and D'Anna Algorithms

<table>
<thead>
<tr>
<th></th>
<th>D'Anna</th>
<th>FM2</th>
<th>FM1</th>
<th>Proposed</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Time (s)</td>
<td>18.32</td>
<td>4.086</td>
<td>3.234</td>
<td>1.458</td>
<td>2.22x</td>
</tr>
<tr>
<td>Memory</td>
<td>6.717</td>
<td>0.787</td>
<td>0.713</td>
<td>0.396</td>
<td>1.80x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>CPU Time (s)</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Time (s)</td>
<td>122.4</td>
<td>45.86</td>
</tr>
<tr>
<td>Memory</td>
<td>62.71</td>
<td>7.267</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>CPU Time (s)</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Time (s)</td>
<td>243.5</td>
<td>71.01</td>
</tr>
<tr>
<td>Memory</td>
<td>62.71</td>
<td>7.267</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>CPU Time (s)</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Time (s)</td>
<td>625.6</td>
<td>106.1</td>
</tr>
<tr>
<td>Memory</td>
<td>174.7</td>
<td>10.17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>CPU Time (s)</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Time (s)</td>
<td>869.2</td>
<td>124.8</td>
</tr>
<tr>
<td>Memory</td>
<td>338.8</td>
<td>12.07</td>
</tr>
</tbody>
</table>

aAll CPU time and memory requirement results are average values.
bImprovement provided by proposed algorithm over FM1.
cNumber of 32-bit floating-point variables (×10^6).

Several observations are made from the results shown in Table 4.4 and Table 4.5. First, the proportion of parallel places among the places created during Phase 1 is high: the average value of \( r \) ranges from 0.51 to 0.59 across the five test groups. Second, the computation time and memory requirement of the proposed algorithm are largely dominated by Phase 2 (close to 99%), thus highlighting the necessity of the proposed heuristic rules. Third, the proposed heuristic rules (especially heuristic rule #1) are highly effective in reducing the Phase 2 computation time, offering an improvement of at least 5.2x. In terms of memory requirement, the improvement is at least 1.2x.

Table 4.6 compares the computation time and memory requirement of the proposed algorithm (implemented with heuristic rules #1 and #2) against other reported algorithms that generate all minimal-support invariants, namely the FM1, FM2 [5, 16], and D’Anna [150] algorithms. Note that FM1 is reportedly [147][148][149] the fastest
implementation of the FM algorithm. For the FM2 and D’Anna algorithms, the computation time includes the time taken to delete the nonminimal-support invariants. For the reported algorithms, the memory requirement is defined as the maximum size of matrices used.

Among the reported algorithms, FM1 has the best performance in terms of both computation time and memory requirement, whereas the D’Anna algorithm has the worst performance due mainly to the large number of nonminimal-support invariants that it generates. As shown in Table 4.6, the proposed algorithm is significantly better than the reported ones. In terms of computation time, the proposed algorithm is at least 2.2x faster than the rest. It is also more efficient in memory usage as it requires at least 1.8x less memory compared to the reported algorithms.

### 4.8. Parallelizing the Invariant Enumeration Phase

This section describes how the proposed algorithm can be parallelized to take advantage of parallel computing systems (i.e., grid-based computing systems or multi-core computers) so as to improve its speed performance.

As explained earlier in this chapter, the net transformation phase is the same as the conventional FM algorithm except for the additional steps of detecting parallel places in the net after each transition has been annihilated, and the replacement of each set of parallel places with a representative parallel place. This phase cannot be parallelized and therefore cannot take advantage of parallel computing systems. However, speeding up the net transformation phase is not critical because, as reported in Section 4.7, it typically takes up less than 1% of the total computation time.

On the other hand, the invariant enumeration phase is able to take advantage of parallel computing systems. To recap, the invariant enumeration phase is an
Table 4.7
Computation Time Improvements Provided by Parallelizing Invariant Enumeration Phase of Proposed Algorithm

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>1.458</td>
<td>8.800</td>
<td>20.83</td>
<td>43.58</td>
<td>65.77</td>
</tr>
<tr>
<td>Parallel</td>
<td>1.074</td>
<td>6.852</td>
<td>14.53</td>
<td>34.73</td>
<td>50.35</td>
</tr>
<tr>
<td>Improvement</td>
<td>1.36X</td>
<td>1.28X</td>
<td>1.43X</td>
<td>1.25X</td>
<td>1.31X</td>
</tr>
</tbody>
</table>

enumerative process in which each candidate vector resulting from the net transformation phase is used to generate, with respect to the representative parallel places in its support, other candidate vectors, which are in turn used to generate more vectors based on the same procedure. A vector that is generated during the enumerative process whose support contains only the places of the original net is a minimal-support invariant of the net.

To take advantage of parallel computing systems, the invariant enumeration algorithm can be revised such that the candidate vectors resulting from the net transformation phase are distributed to the various processors in the system, so that the enumeration process for the candidate vectors can be executed in parallel. It is worth noting that communications between the processors are not necessary when the enumeration processes are executed in parallel because the enumeration process for each candidate vector can be executed independently. This is significant because interprocessor communications can be expensive in terms of time [153].

Given the dominance in computation time of the invariant enumeration phase (close to 99% of the total computation time) and the independence between the enumeration processes for the candidate vectors, the improvement in computation time of the proposed algorithm by parallelizing its invariant enumeration phase is expected to be significant.
To get an indication of how much the improvement would be, the test problems were recomputed using the proposed algorithm with a modification on the way the computation times are measured. For each test problem, instead of simply measuring the total time taken to execute the enumeration processes for all the candidate vectors, the candidate vectors are randomly divided into two groups (assuming that the algorithm is running on a dual-core system) and the time taken to execute the enumeration processes for the candidate vectors in each group is measured. The larger of the two measured computation times is then taken as the computation time of the invariant enumeration phase for that particular test problem.

Table 4.7 compares the simulated computation times of the parallel algorithm with the computation times of the original algorithm for the different test problem categories. As shown in Table 4.7, parallelizing the invariant enumeration phase of the proposed algorithm can provide a speed improvement that ranges from 1.25x to 1.43x.

4.9. Summary

In this chapter, a fast and memory-efficient algorithm for the computation of all minimal-support S-invariants of ordinary PNs has been proposed. This work was motivated by the proposed optimization methods' requirement that the minimum cycle time of the pipeline under optimization be recomputed at each optimization iteration.

The algorithm has been fully developed and its correctness has been proven formally and rigorously. The efficacy of the proposed algorithm has been demonstrated through a large number of test problems (randomly generated ordinary PNs). For the tests that were conducted, the proposed algorithm is at least 2.2x faster than the other comparable algorithms. It is also more efficient in memory usage as it requires at least 1.8x less memory compared to the reported algorithms.
It is of significance to note that although the proposed algorithm has been motivated by the need to improve the feasibility of the proposed optimization methods, the proposed algorithm is in fact independent of the optimization methods and is in itself of value to the analysis of PNs in general.
5
Conclusions and Recommendations

5.1. Conclusions
This thesis has proposed several methods that facilitate the design of asynchronous pipelines with low asynchronous control overheads. The development of the methods have been formal and rigorous. The methods have been automated and shown to be effective through several nontrivial design examples in reducing the circuit areas and power dissipation of asynchronous control networks. The work that has been described in this thesis are as follows.

First, a synthesis method for asynchronous pipelines with low asynchronous control overheads has been proposed, developed, and automated. More specifically, the following work on the modeling and synthesis of asynchronous pipelines has been described:

i) A coarse-grain approach that is employed by the proposed synthesis method to synthesize asynchronous control networks has been described. This approach reserves asynchronous control to the implementation of essential asynchronous
operations, thus facilitating the design of asynchronous pipelines with low asynchronous control overheads in terms of circuit area and power dissipation.

ii) A set of modeling rules for asynchronous communication that is supported by the proposed synthesis method has been described. The modeling rules are based on conventional Verilog HDL constructs, on which additional semantics are imposed during synthesis to infer asynchronous communication. This means that the use of special packages or subroutines for asynchronous communication modeling is avoided.

iii) The process of synthesizing the asynchronous control network of the design specification and establishing the control of the asynchronous control network over the design’s datapath has been described. In particular, the three main tasks performed during the synthesis process – asynchronous communication channel extraction, handshake component inference, and initial state computation for asynchronous control networks – have been described in detail.

iv) A method for computing a live initial state for an asynchronous control network has been proposed. The proposed method has been formally proven to preserve the nondeadlock behavior of the control network resulting from its specified initial state.

v) The efficacy of the proposed synthesis method has been demonstrated through the design of an asynchronous Reed-Solomon error detector and an asynchronous IFIR filter bank. Compared with the error detector implemented by desynchronization, that implemented by the proposed synthesis method is found to dissipate 33% less energy. Although the error detector implemented by the proposed synthesis method is 9% slower than the desynchronized
circuit, it is 19% better in $E^2$. Compared with the error detector implemented by Pipefitter, the circuit implemented by the proposed synthesis method has 9% less transistors, dissipates 39% less energy, and is 2.3x faster. The filter bank implemented by the proposed synthesis method dissipates 38% less energy and is 15% better in $E^2$ than that implemented by desynchronization. Compared with the filter bank implemented by Pipefitter, the circuit implemented by the proposed synthesis method enjoys a 14% advantage in energy dissipation and has a latency that is 38% shorter.

Second, two optimization methods – handshake component fusion and optimal decoupling – for reducing the circuit areas and power dissipation of asynchronous control networks while satisfying given pipeline throughput constraints have been proposed, developed, and automated.

The first proposed optimization method – handshake component fusion – is a form of peephole optimization that iteratively selects a pair of optimization targets that share input channel sources or output channel destinations and replaces them with a single component of the same type. More specifically, the following work on the proposed handshake component fusion method has been described:

i) A heuristic algorithm for the selection of the optimization targets has been described. In essence, the selected optimization targets at each optimization iteration are those that are judged by the algorithm to be least likely to cause a change in the minimum cycle time of the pipeline.

ii) The conditions that must be satisfied in order for the fusion of the optimization targets to be accepted by the optimization process have been described. The first condition – satisfaction of minimum cycle time constraint – involves the computation of the minimum cycle time of the restructured pipeline. In
particular, a procedure has been described for taking into consideration the effect of additional capacitive loading taken on by latch enable signals due to handshake component fusion. The second condition – flow equivalence between the original and restructured pipelines – is related to preserving the behavior of the pipeline under optimization. In particular, it has been formally proven that the original and restructured pipelines are flow equivalent if there is no channel link between the optimization targets.

iii) The application of the proposed handshake component fusion method on the control networks of three asynchronous designs have been described. The designs are a 16-input pipelined parallel prefix tree (for the addition operation), a four-bit cross-pipelined array multiplier, and a Reed-Solomon error detector for the compact-disc player. For the three optimization examples, the results attained by the proposed handshake component method have been encouraging. Compared with the original asynchronous control networks, the optimized control networks, on the average, have 49% fewer transistors and dissipate 48% less energy, while sacrificing at most 6% of their throughputs.

The second proposed optimization method – optimal decoupling – has been proposed to resolve the dilemma when designing asynchronous pipelines between using small handshake components to reduce asynchronous control overheads and satisfying throughput constraints. More specifically, the following work on the proposed optimal decoupling method has been described:

i) A branch-and-bound algorithm has been described that searches for the optimal mix of handshake components of different degree of concurrency in a given asynchronous control network. The objective of the algorithm is to incur the
least circuit area and power dissipation for the control network, while satisfying a given minimum cycle time constraint.

ii) The efficacy of the proposed optimal decoupling method has been demonstrated through three optimization examples: a 16-input pipelined parallel prefix tree (for the addition operation), a four-bit cross-pipelined array multiplier, and a Reed-Solomon error detector for the compact-disc player. For the three optimization examples, the optimally-decoupled asynchronous control networks, on the average, have 22% fewer transistors and dissipate 32% less energy when compared with the other control networks based on uniform handshake component concurrency (except the minimally-concurrent control network). Although the optimally-decoupled control networks are larger and dissipate more energy than their minimally-concurrent counterparts, they are at least 24% faster. In addition, the circuits with optimally-decoupled control networks are at least 38% better in $E^2$ than those with minimally-concurrent control networks.

Third, a fast and memory-efficient algorithm for the computation of all minimal-support S-invariants of ordinary PNs has been described. This work was motivated by the proposed optimization methods’ requirement that the minimum cycle time of the pipeline under optimization is recomputed at each optimization iteration. The algorithm has been fully developed and its correctness has been proven formally and rigorously. The efficacy of the proposed algorithm has been demonstrated through a large number of test problems (randomly generated ordinary PNs). For the tests that were conducted, the proposed algorithm is at least 2.2x faster than the other comparable algorithms. It is also more efficient in memory usage as it requires at least 1.8x less memory compared to the reported algorithms. It is of significance to note that although the proposed
algorithm has been motivated by the need to improve the feasibility of the proposed optimization methods, the proposed algorithm is in fact independent of the optimization methods and is in itself of value to the analysis of PNs in general.

Fourth, the proposed synthesis method for asynchronous pipelines and the proposed optimization methods for asynchronous control networks have been automated and augmented with additional scripts such that they are integrated easily into the conventional synchronous design flow. The proposed algorithm for PN invariant computation has been automated and incorporated into the proposed optimization methods to expedite minimum cycle time analyses.

5.2. Recommendations

This section recommends two improvements to the methods proposed in this thesis and two topics for further research. It also suggests some real-world circuits that are suitable for implementation using the methods proposed in this thesis.

The first recommended improvement to the work described in this thesis concerns the proposed modeling and synthesis method. As described in Chapter 2, a Verilog HDL model that is written based on the proposed modeling rules must go through a compilation step using the proposed synthesis method before it can be simulated to verify its functionality. It would be desirable if the Verilog models can be simulated directly without going through the compilation step.

One possible way of achieving this is to make use of some of the features available in SystemVerilog that facilitate high-level communication modeling. SystemVerilog (IEEE Standard 1800) is built on top of the existing Verilog language (IEEE Standard 1364) and its features are now increasingly supported by most advanced HDL
simulators, including Synopsys VCS, Mentor Graphics ModelSim, and Cadence NC-Sim.

For example, it might be worth investigating the possibility of modeling asynchronous communication channels using the SystemVerilog mailbox class and the related methods (in particular, put() and get()), and the SystemVerilog interface construct (for inter-module communication). The SystemVerilog mailbox is a communication mechanism that allows messages to be exchanged between processes. Data can be sent to a mailbox by one process and retrieved by another. Thus, the put() method, which places a message in a mailbox, can be used to model the initiation of a handshake on a channel and the sending of data on the input end of the channel. On the other hand, the get() method, which retrieves a message from a mailbox, can be used to model the completion of a handshake on a channel and the receiving of data on the output end of the channel.

Note that adopting this recommendation would not only require revising the proposed modeling rules, but also the proposed synthesis method.

The second recommended improvement to the work described in this thesis concerns the heuristic algorithm for optimization target selection proposed for the handshake component fusion method. As described in Section 3.5.2, the algorithm only considers handshake component pairs that form a fork or join as optimization targets. To increase the search space for optimization targets, one could extend the algorithm such that all handshake component pairs of the same type in the control network, instead of just the pairs in fork or join configurations, are considered as potential optimization targets.

Such an extension of the algorithm would involve modifying Step 4 and 5 of the original algorithm (see Section 3.5.2) such that the score for each handshake component
pair of the same type is computed and such that the handshake component pair with the
highest score is selected as the optimization targets. If there are two or more candidate
pairs with the same highest score, then priority is given to those that form forks or joins.

The proposed extension increases the search space for optimization targets and can,
potentially, lead to better optimization results.

Two topics are now recommended for further research. While these topics are
directly related to the work described in this thesis, they are also relevant to the wider
field of asynchronous design methodology.

The first recommended topic for further research is concerned with the optimization
of asynchronous control networks based on nonspeed-independent handshake
components. This thesis has focused on speed-independent handshake components due
to their timing robustness and the availability of well-established CAD tools for their
synthesis. It has not considered nonspeed-independent handshake components. Some of
these handshake components (see, for example, [127][128][129]) are designed under
certain timing assumptions (besides the speed-independent timing assumption) which
allow them to be capable of delivering better pipeline throughput than speed-
independent handshake components.

It is, therefore, of interest to investigate whether the proposed optimization methods
can be applied to nonspeed-independent handshake components. One way of doing so is
to establish the conditions that nonspeed-independent handshake components must
satisfy in order for the optimization methods to be applicable. One such condition would
be that it must be possible to incorporate the timing assumptions made by the handshake
component into its STG specification. This is simply because the proposed optimization
methods operate on PN models that are composed of STGs. Other conditions might be
less obvious and detailed analysis of the proposed optimization methods will be
required to determine them. If the conditions turned out to be so stringent that no or very few nonspeed-independent handshake components can satisfy them, then one would need to study ways of extending the proposed optimization methods to improve their applicability.

The second recommended topic for further research is related to the computation of S-invariants for PNs. It has been argued in Chapter 4 that the proposed algorithm for PN S-invariant computation is able to provide significant reductions in the time and memory requirements of the optimization methods proposed in Chapter 3. Nevertheless, S-invariant computation for PNs that are composed of STGs remains a task that consumes significant computer time and memory because such PNs are intrinsically complex.

It is, therefore, of interest to develop alternative PN models of asynchronous control networks that are simpler than their STG-based counterparts. Two methods for doing so will now be suggested.

First, PN models of handshake components could be conceived that are simpler (i.e., fewer transitions, places, and arcs) than the corresponding STGs. Global PN models composed of such simpler models would inherently be less complex than their counterparts based on STGs. The challenge in this approach is that the simpler models need to be at least as expressive as STGs in modeling the behaviors of various handshake components and in representing various handshake protocols. For example, the full buffer channel net (FBCN) [130], which models a leaf cell instance as a single transition, is considerably simpler than PNs composed of STGs. However, FBCNs cannot model four-phase handshake protocols, which are ubiquitous in asynchronous designs.
Second, some existing or novel transformation methods could be used to reduce the complexity of the STG models of handshake components. A necessary property of such transformation methods is that they must guarantee that the minimum cycle time of a PN model composed of the reduced STG models is the same as that composed of the original STG models. Furthermore, the S-invariants that are computed for a PN model composed of the reduced STG models must contain sufficient information to be useful to the proposed optimization methods.

Finally, a brief discussion on the real-world circuits that are suitable for implementation using the methods proposed in this thesis. The proposed modeling, synthesis, and optimization methods are most suitable for the design of asynchronous pipelines with distributed control. The Reed-Solomon error detector and the IFIR filterbank reported in Chapter 2 are good examples of real-world circuits that are suitable for implementation using the proposed methods. Other real-world circuits that are suitable applications for the proposed methods include microprocessors, encryption engines (such as Data Encryption Standard (DES) circuits), Huffman decoders, and First-In-First-Outs (FIFOs).
Bibliography


[29] S. M. Nowick, K. Y. Yun, and P. A. Beerel, “Speculative completion for the

decompression circuit for embedded processors,” *Adv. Res. VLSI*, pp. 219–236,

[31] W. Chou, P. A. Beerel, R. Ginosar, R. Kol, C. J. Myers, S. Rotem, K. Stevens,
and K. Y. Yun, “Average-case optimized technology mapping of one-hot

control for high-performance asynchronous circuits,” *IEEE Proc.*, vol. 87, no. 2,

[33] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij,
“Asynchronous circuits for low power: A DCC error corrector,” *IEEE Design


[36] J. S. Chiang and D. Radhakrishnan, “Hazard-free design of mixed operating
23–37, 1990.


[38] G. Mago, “Realization methods for asynchronous sequential circuits,” *IEEE

asynchronous machines using controlled excitation and flip-flops,” *IEEE Trans.

[40] L. A. Hollaar, “Direct implementation of asynchronous control units,” *IEEE

[41] J. H. Tracey, “Internal state assignments for asynchronous sequential machines,”


[58] Timeless Design Environment (TiDE) by Handshake Solutions (http://www.handshakesolutions.com)


598.


[84] A. Branover, R. Kol, and R. Ginosar, “Asynchronous design by conversion:
Converting synchronous circuits into asynchronous ones,” in Proc. DATE, 2004,
pp. 870–875.

framework for describing and modeling asynchronous circuits at all levels of

CAD tools,” IEEE Design & Test of Computers, vol. 19, no. 4, pp. 107–117, July
2002.


[89] T. Yoneda, A. Matsumoto, M. Kato, and C. Myers, “High level synthesis of
timed asynchronous circuits,” in Proc. Int. Symp. Asynchronous Circuits and

asynchronous system design using the ACK framework,” in Proc. Advanced

Complete State Coding for Signal Transition Graphs”, in Proc. Int. Conf.

[92] I. Blunno and L. Lavagno, “Automated synthesis of micropipelines from
behavioral Verilog HDL,” in Proc. Advanced Research in Asynchronous Circuits
and Systems, 2000, pp. 84–92.

pipelining using domino style asynchronous library,” in Proc. Int. Conf.


