Fall 2020

An Efficient Design Methodology for Complex Sequential Asynchronous Digital Circuits

Tomasz Chadzynski
San Jose State University

AN EFFICIENT DESIGN METHODOLOGY FOR COMPLEX SEQUENTIAL ASYNCHRONOUS DIGITAL CIRCUITS.

A Thesis

Presented to

The Faculty of the Department of Electrical Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Tomasz Chądzyński

December 2020

The Designated Thesis Committee Approves the Thesis Titled

AN EFFICIENT DESIGN METHODOLOGY FOR COMPLEX SEQUENTIAL ASYNCHRONOUS DIGITAL CIRCUITS.

by

Tomasz Chądzyński

APPROVED FOR THE DEPARTMENT OF ELECTRICAL ENGINEERING

SAN JOSÉ STATE UNIVERSITY

December 2020

Tri Caohuu, Ph.D.	Department of Electrical Engineering
Chang Choo, Ph.D.	Department of Electrical Engineering
Morris Jones, MSEE	Department of Electrical Engineering

ABSTRACT

AN EFFICIENT DESIGN METHODOLOGY FOR COMPLEX SEQUENTIAL ASYNCHRONOUS DIGITAL CIRCUITS.

by Tomasz Chądzyński

Asynchronous digital logic as a design alternative offers a smaller circuit area and lower power consumption but suffers from increased complexity and difficulties related to logic hazards and elements synchronization. The presented work proposes a design methodology based on the speed-independent sequential logic theory, oriented toward asynchronous hardware implementation of complex multi-step algorithms. Targeting controller-centric devices that perform data-driven non-linear execution, the methodology offers a CSP language-based controller workflow description approach and the specification of a project implementation template supported by a two-stage design process. First, the CSP layer describes complex speed-independent controller behavior offering better scalability and maintainability than the STG model. Second, the component-oriented design template specifies functional elements' structural organization and emphasizes the divide-and-conquer philosophy, streamlining large and complex devices' design and maintenance. Finally, the implementation process is divided into two stages: a rapid development and functional verification stage and a synthesizable codebase stage. Additionally, a case study design of a split-transaction MESI cache coherency controller and its analysis are presented to validate the proposed methodology. The testing phase compares synthesized and routed gate-level asynchronous and synchronous implementations. For models synthesized to work with the same speed, the asynchronous circuit area is 20% smaller with lower power consumption at approximately 18% of the synchronous reference. The synchronous version synthesized for performance is 3.5 times faster, at the cost of a large increase in area and power usage. The results prove the methodology's ability to deliver working complex asynchronous circuits competitive in the chip area and power characteristics.

To my beloved wife Joanna and our daughter Emilia.

ACKNOWLEDGMENTS

I want to thank Dr. Tri Caohuu for his continuous guidance and support. Professor Caohuu dedicated an enormous amount of time to help me with the research, and thanks to his guidance, this work came into the shape it is today. I would also like to thank Dr. David Parent, Prof. Morris Jones and Audrey Leong for their time spent helping me with the Lab Equipment and Practical portion of the thesis. I am incredibly grateful to Professor Parent for his time spent, ensuring that I have all the required CAD software available to finish this thesis. Additionally, I would like to thank Magdalena Krajewska, Mikayla Hutchinson, Attapol Rutherford, Keith Mueller, Greg Boyd, Bruce Rakes, Curt Kineast and Adam Molzahn, whom in past years, inspired and helped me get to this point.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACK	Acknowledge
ASIC	Application Specific Integrated Circuit
CAS	Compare And Swap
CPU	Central Processing Unit
CSC	Complete State Coding
CSP	Communicating Sequential Processes
DI	Delay Insensitive Circuit
DSP	Digital Signal Processing
DSPL	Direct Signal Pull Low
DUT	Design Under Test
ECO	Engineering Change Order
FSE	Flow Support Element
GALS	Globally Asynchronous Locally Synchronous
HDL	Hardware Description Language
IOT	Internet Of Things
ISA	Instruction Set Architecture
ISPL	Inverted Signal Pull Low
LR	Load Reserved
MESI	Modified Exclusive Shared Invalid
MIC	Multiple Input Change
NCL	Null Convention Logic
NOC	Network On Chip
PDK	Process Design Kit
PLOG	Pending transaction Log
QDI	Quasi Delay Insensitive Circuit
REQ	Request
SC	Store Conditional
SG	State Graph
SI	Speed Independent Circuit
SIC	Single Input Change
SOC	System On Chip
STG	Signal Transition Graph

1 INTRODUCTION

Asynchronous logic is an alternative digital design approach that offers tangible advantages in the form of a smaller circuit area, lower power requirements, and improved fault tolerance [1], [2] compared to the widely used synchronous methodology. A major differentiating factor here is the lack of the clock signal in the asynchronous circuits. Eliminating the need for global clock distribution or even localized clocking removes the need for the additional clock tree in the circuit that consumes a significant portion of the chip area [3]. The absence of the clock signal also implies less switching activity in the device as the elements do not receive an unconditional signal to which they have to react. In asynchronous logic design, any unnecessary activity virtually does not exist as the components are active only when taking when executing their part of the workflow [4], [5]. Therefore, removing the clock signal opens the door to the synthesis of digital circuits exhibiting lower power consumption and smaller chip area than their synchronous counterparts [6], [7]. In the era of IoT and growing popularity for small embedded custom cyber-physical systems [8], the use of asynchronous logic in a design can benefit projects for which low power and small size are crucial design goals.

Despite the advantages, asynchronous logic suffers a significant drawback due to the level of difficulty in design. Removing the clock increases the impact of logic hazards on the entire device and removes the primary synchronization signal. Asynchronous logic must be designed under strict control over any unexpected logic fluctuations as they can set off incorrect device behavior. Despite the existence of well-established methods in the asynchronous design, modeling of a larger, non-trivial circuit still presents a significant challenge.

This work proposes a methodology that specifies an approach for designing complex sequential asynchronous circuits targeted for implementing digital modules that execute

multi-step algorithmic workflows. The focus is on the type of circuits that execute data-driven tasks composed of multiple consecutive operations. The methodology builds on top of the theory of speed-independent circuits [4], [5], [9]. The proposed framework introduces an HDL template that defines the model's codebase structure and adds a CSP-based description layer on top of STG graphs that simplifies the modeling of speed-independent asynchronous controller modules. Also, an emphasis is placed on the divide-and-conquer philosophy in which the large and complex design splits into simpler and easier to maintain elements.

In asynchronous logic, the state machine construct is replaced by the Petri-net derivative model called Signal Transition Graph (STG) as the high-level representation of the working algorithm. Unlike the state graph used in the synchronous design, the STG model represents both: the sequence of events and a description of the concurrency in the system. In the STG flow model, events can happen in parallel and at an unspecified time but in a known order. Correct derivation of the STG graph is critical for obtaining a working asynchronous controller. Consequentially, asynchronous design modeling using STG becomes a much more complex and error-prone task than the sequential model equivalent.

This research proposes a methodology based on the proven STG based synthesis approach [4], [9] and builds on top of it to improve the design and verification process of non-trivial systems. The base STG-based speed-independent workflow is extended to streamline the design of complex controller behavior. An additional Communicating Sequential Processes (CSP) language-derived abstraction layer is added on top of the STG that offers more flexibility and improves model maintainability, especially in large designs. While the STG graph grows with the complexity of the design and becomes more challenging to maintain, the CSP representation offers an imperative-like programming language representation that allows for decomposition into smaller

submodules. The CSP model is then translated into the STG graph from which the target circuit next-state Boolean equations are derived. The proposed methodology presents a reference implementation of a CSP to STG translator using a subset of CSP language with custom extensions that allow for an automated translation.

The second element introduced by the proposed methodology next to the CSP translator is the HDL template that specifies the project implementation structure. The proposed methodology follows the divide-and-conquer philosophy, which favors breaking a complex problem into a set of smaller ones that are easier to deal with. The specified design template dictates dividing a design into smaller functional blocks called Components. The Component is the basic building block that covers a subset of functionality and contains a methodology-specified set of modules. A single instance of the Component contains at minimum one or more speed-independent controllers to implement its workflow and a set of supporting modules. As the speed-independent controller works only with control signals but not the data, the proposed methodology introduces templates for a set of datapath elements called Flow Support Elements (FSE). One or more FSEs work in tandem with the controller in a passive-active setup and execute various tasks such as conditional resolution or data manipulation. In addition to the controller and FSE modules, the Component can contain miscellaneous elements such as registers, mutexes, or support modules that perform simple signal manipulation tasks. Going upward in the template hierarchy, the Component modules then connect together within the Top module. The Top module is the primary element that groups functional elements into the designed module and provides the external interface.

With the state of current computer technology, nearly every system is composed of more than one processing core. Using a multi-core system and parallel computing model became the dominant way to achieve increased performance while mitigating the power wall effect. Using multiple cores to compute in parallel creates additional architectural

challenges related to memory subsystems, including cache memory. When more than one core operates on shared address space, keeping a multi-layer memory architecture in a stable and coherent state is crucial for error-free system operation.

When analyzing the execution pattern of software, two characteristics become apparent with the majority of computer programs exhibit a temporal and spatial locality behavior [11]. As a result, separate programs or execution threads do not often compete for access to the main memory. Instead, they operate on a subset of the memory space stored as a local copy in their cache memory for most of their execution time. However, an application memory access pattern itself is unpredictable and depends on its state, inputs and executed algorithm. Few examples of such unpredictable input sources are the user input or a variety of environmental sensors.

In the global system perspective, such asynchronous and unpredictable memory access behavior in conjunction with multi-layered cache memory hierarchies can be a good fit for implementation using asynchronous logic. Asynchronous logic is not constrained by the clock time and can react to an event as soon as the hardware resource is ready, potentially providing a shorter time to completion [5]. Another advantage of an asynchronous circuit is the absence of a clock. The clock tree logic used to distribute the clock signal throughout the circuit consumes a significant portion of the chip area. The elimination of the clock tree lowers the required transistor count within the chip leading to smaller area requirements. Asynchronicity, lower power and area requirements could be particularly appealing when applied in small, low footprint embedded systems.

The presented work provides a case study in the form of an implementation of a cache coherence controller to evaluate the proposed methodology. The provided controller implements a MESI protocol, one of the most widely adopted cache coherency algorithms used in general-purpose processors, and communicates using the split-transaction bus [10], [12]. The motivation behind selecting the cache coherence controller is to

provide a sufficiently complex design problem to demonstrate and evaluate the methodology's practical use. Its implementation structure splits into four components: Sender which processes the requests from the CPU core and places requests on the bus, Receiver which processes messages incoming from the bus, Pending Request Log that serves as a journal for any outstanding data request transactions across the system and the Responder answering requests from other cores. All four components together form an implementation that realizes the MESI cache coherence protocol with a split-transaction bus interface.

2 LITERATURE SURVEY

Though not currently widespread in the mainstream industry, the asynchronous logic design is a topic of active and extensive research. The theory of speed-independent controller circuit is widely documented by a sizable selection of books and research papers. Many books discuss various implementations of logic elements starting from the asynchronous pipelines through asynchronous arbiters, QDI circuit templates, and many more. The asynchronous logic also saw a number of practical implementations of working circuits such as microprocessors and DSP units. This section surveys through selected literature positions relevant to the presented work.

2.1 General Introductory Literature

The asynchronous logic design introduces a large variety of new design concepts non-existent in the synchronous approach. Simultaneously, most techniques widely used in the synchronous methodology, such as the state machine or pipelines, are fundamentally changed and require a completely different design approach. Among the available literature, there exists a number of publications that could serve as good introductory material and beyond. The two-part article [1], [13] by Steven M. Nowick and Montek Singh in part 1 lays out the history of asynchronous logic design techniques, presents a series of commercial projects, and discusses additional emerging areas of asynchronous logic applications such as ultra-low power or extreme environments. Finally, part one presents a fundamentals overview, primarily discussing handshaking protocols and pipelining. In part 2, the authors focus on synthesis methods; the paper starts from the discussion about logic hazards and their significance in asynchronous circuits. Next, the paper introduces design techniques for asynchronous controller circuits then finishes with a discussion about high-level synthesis from languages like CSP.

For a more in-depth introduction, the report titled "An Introduction to Asynchronous Circuit Design" [14] offers a general overview of essential concepts in asynchronous logic design. Aside from general motivation and basics, the paper is divided into four sections. Initially, the discussion starts with communication protocols for control and data. The 4-phase and 2-phase handshaking protocols are introduced as well as bundle data delay matching and dual-rail data encoding. Then the section finishes with an analysis of the important concept of completion detection. Next, the work discusses topics related to hazards and delay models used to classify asynchronous circuits. Classes of Single Input Change (SIC) and Multiple Input Change (MIC) hazards are presented along with logic synthesis methods that avoid them. Next, the text discusses the topic of asynchronous arbitration. The arbitration in asynchronous circuits is an analog problem involving expected metastability and metastability filtering. The arbitration in asynchronous logic is an essential concept; in some cases, it is the only way to synchronize communication between the device's separate components. Lastly, the report lays out the topics of the design of sequential components and asynchronous datapath. Different design techniques are discussed for sequential circuits, such as the Huffman asynchronous state machine and the STG-based input-output speed-independent controller. The asynchronous datapath segment focuses on pipeline design. Overall the report serves as an excellent introductory material with a sufficient level of technical discussion.

The final position is the "Asynchronous Circuit Design, A Tutorial" [5], which goes beyond the asynchronous design topics' general introduction. The book's leading theme focuses on constructing asynchronous circuits as a series of functional blocks organized into complex pipelines or ring structures. The material starts with introducing the fundamentals, and similar to other positions, discusses the communication protocols, the C-element and the theoretical basis. Throughout the next two chapters, three and four, the book takes on the topic of circuit design and performance. Specifically, chapter three

introduces basic functional building blocks and design patterns such as the conditional if-statement or while loop design using the introduced functional blocks. The following sections go from an abstract block-based design view into a specific implementation of the functional blocks. The fifth chapter discusses the circuit-level implementation of functional blocks starting from elements such as fork-and-join, then discussing bundle data and dual-rail encoding and finishing with topics related to mutual exclusion and arbitration. Chapter 6, which is the last section in the book that focuses on low-level design-related material, is entirely dedicated to the synthesis of the speed-independent controllers. The section discusses derivation techniques that start from Signal Transition Graphs (STG) through the synthesis procedure into gate-level representation. The work presents a step-by-step approach to obtain the set of next-state equations both for the complex gate and standard C-element controller structures. The chapter finishes with the introduction of the Petrify tool [15], an automated synthesizer of asynchronous controller circuits. Lastly, the final two chapters discuss more advanced topics in channel-based communication and high-level synthesis from languages like CSP. The book serves as an excellent resource for a reader who is new to asynchronous design but is looking for an introduction and a guide to starting custom asynchronous design projects.

2.2 Asynchronous Controller Design and Synthesis

The topic of asynchronous controller circuits is one of the most important parts of asynchronous logic design and also one of if not the most complex one. The book "Asynchronous Circuit Design" [4] by Chris J. Myers provides an in-depth discussion in asynchronous controller design. The book follows a rigorous mathematical and definition-based approach in introducing the material. The text's main portion is dedicated to two types of asynchronous controllers: Huffman Circuit, also called the fundamental mode machine, and the Mueller Circuit that is the input-output mode speed-independent

design. Also, the fundamentals such as communication protocols, Petri-nets, and basic introduction are covered as well. The book provides a complete mathematical definition of the STG graph model and walks in-depth through the entire synthesis process of the Mueller Circuit. Advanced synthesis concepts and their algorithmic specification are presented for every step of the process, including Complete State Coding (CSC) and Hazard-Free Logic Decomposition. The book delivers a solid, well-defined basis for further research and CAD tool development.

In the category of input-output mode asynchronous circuits, two positions, the "Logic Synthesis for Asynchronous Controllers and Interfaces" [9] and "Synthesis of asynchronous hardware from Petri nets" [16] provide dedicated coverage into all topic starting from STG specification until hazard-free decomposition into logic gates with limited fan-in. The first position is a book dedicated entirely to the speed-independent controller synthesis through the State-based model. The book thoroughly examines every stage of the synthesis; it presents an entire chapter dedicated to correct STG representation topics, implementability of STG and SG, and the derivation of the next-state equations for the complex gate and the standard C-element. Also, the book presents algorithms for CSC generation and methods of analysis of the SG structures. Finally, the book discusses hazard-free logic decomposition, including technology mapping to a logic gate library. The second position [16] tackles the problem of state explosion known from the State-based synthesis approach. The paper discusses alternative ways of directly deriving next-state equations from the STG representation bypassing the SG graph processing phase.

2.3 Circuit Level Element Design

The asynchronous logic design introduces many circuit-level techniques not used in the synchronous discipline. The paper titled "Asynchronous Techniques for

System-on-Chip Design" [17] presents circuit-level constructs of many asynchronous functional elements. The text introduces an approach to the specification of asynchronous circuits functionality in a CHP language, a CSP derivative. The work discusses many templates for asynchronous components, inter-component communication, and pipelines. One of the most notable topics is the simple arbiter circuit, which is also called the mutex element; then, the paper surveys its more complex variations. Then the article discusses an asynchronous register design with write completion detection mechanisms, pipeline techniques, and interfacing in GALS circuits.

Asynchronous pipelines are an essential concept right next to asynchronous controllers. Unlike in synchronous methodology, there exists a large variety of asynchronous pipeline designs that vary with complexity and speed. Article [18] presents several pipeline constructs dividing those into two categories. The first category is a static logic pipeline that uses latches, and the second is the latchless dynamic logic. Among the categories, the text shows different completion detection techniques such as bundle data or dual-rail encoding and different communication protocols such as 2-phase or 4-phase. On the static logic side, the paper presents the base Sutherland pipeline and high-performance constructs such as the Mousetrap and GasP pipelines. Then in the dynamic logic section, the PS0 pipeline model is presented. The paper delivers a useful review of different types of asynchronous pipelines presenting options of different complexity and performance-power characteristics.

2.4 CSP Language Applications in Asynchronous Logic Design

The CSP language and its derivatives are widely used in the asynchronous circuit design specification. The language itself was designed to model concurrent systems. Its syntax can represent sequences of events occurring in different flows such as serial or parallel order as well as the dependencies between them. The language uses a concept of

a process, in which a single process is an entity that interacts with its environment. The positions [19]–[21] extensively cover the language, its constructs, use cases, and examples. The CSP language does not have one formal specification and exists in many variations; thus, its description, as expected, varies from text to text.

One of the most notable works in asynchronous logic design using CSP is the California Institute of Technology's effort to design a method of compiling CSP defined processes into delay-insensitive VLSI circuits. The series of texts [22]–[24] presents the proposed methodology and implementation of a working automated compiler. The approach translates a concurrent program written in CSP language to a set of predefined elements mapped to VLSI primitives. The result is a delay insensitive asynchronous circuit.

2.5 Multi-Core Systems and Cache Coherency

There exist two excellent texts on the topics of computer architecture, cache coherency, and multicore systems. A well known "Computer Architecture A Quantitative Approach" [10] is the primary reference material for general knowledge about computer architecture. The book surveys through a large variety of topics such as basic ISA and CPU pipeline design, memory hierarchy, including cache and virtual memory. Then the book focuses on the current design trends in computers, primarily parallel execution at multiple levels of abstraction from instructions to computer cluster systems. The text discusses instruction, data, and thread-level parallelism and how to take advantage of it to improve computing performance. Finally, the book surveys the topics of warehouse-scale computers and domain-specific architectures. Overall the position is the source for a broad spectrum of computer architecture topics.

Second position the "Parallel Computer Architecture" [12]. The book is entirely focused on parallel processing systems design and provides in-depth discussion in topics

related to shared memory multiprocessor systems or cache coherency. The book begins with an introduction of parallelization and related topics; then, the next three chapters discuss the structure of parallel programs and computing performance. In the following chapters, the book focuses on multiprocessor systems, bus communication, and cache coherency. Among many, the text covers topics such as snoop and directory-based cache coherence protocols, split-transaction bus, and scalable systems. In the final chapters, the book presents network-related topics in multiprocessor architectures.

2.6 Practical Applications of Asynchronous Logic Design

The asynchronous logic design is not only confined to academia. Over the years, many practical applications emerged, including a significant number of projects focused on constructing asynchronous CPUs. The AMULET3 is a fully asynchronous microprocessor implementing the ARM instruction set [25] and a successor of the AMULET2e [26] project that proven realistic gains from using the asynchronous logic design approach. The project carried at the University of Manchester resulted in a manufactured circuit and led to the development of multiple techniques useful in designing asynchronous CPU type of chip. While the goal of AMULET2e is to prove that asynchronous logic design can be used in practice and provide benefits over synchronous design, the AMULET3 goal is to demonstrate asynchronous logic as a commercially viable digital design method. Another project, the ARM996HS [27], is the second example of an ARM ISA based processor designed as an asynchronous circuit. The design used the TiDE framework specializing in generating handshake circuits that use the Haste (variant of CSP) language as a design entry. As expected from asynchronous designs, the ARM995HS features lower power consumption than the synchronous counterpart and delivers higher reliability against variable environmental conditions such as temperature and power supply voltage. The core is designed to connect into SOC through the AHB-Lite bus.

The asynchronous logic design was also successfully leveraged as a base for frameworks that perform high-level synthesis. The Philips Research Laboratory created the Tangram [28] language and a supporting framework for creating asynchronous handshake circuits using CSP based description language. The work is targeted toward a tool that allows the designer without specialized knowledge of VLSI design to specify an asynchronous circuit's behavior and obtain the gate-level representation. The Department of Computer Science at the University of Manchester conducted research into the development of the Balsa [29] system. The system focus is the rapid development and synthesis of the asynchronous circuits. The Balsa language is a procedural language focused on the description of algorithms, which are then synthesized into working asynchronous logic constructs.

2.7 Null Convention Logic as an Alternative Approach in Asynchronous Logic Design

The Null Convention Logic is another distinctive approach in asynchronous logic design. Compared to the methodology based on speed-independent circuits or Huffman asynchronous state machine, the NCL does not use traditional combinational logic paired with C-elements. Instead, NCL relies on a set of multi-input threshold activated state-holding elements. The NCL elements activate after a certain number of inputs becomes high and deactivates when all inputs go back low. When it comes to data representation, the NCL gate's output does not represent Boolean value; instead, high output means data present, and low means no data. Therefore it is common to use dual-rail encoding in conjunction with NCL for the datapath. The paper [30] serves as a good introduction to NCL. The text includes basic concepts and presents the elementary NCL gates. Furthermore, the paper talks about forming pipelines with NCL and compares NCL to the other asynchronous logic design techniques.

The book "Designing asynchronous circuits using NULL convention logic (NCL)" [31] serves as more in-depth coverage of the topic. Similarly, the book introduces the basics and then shows transistor level implementation of the threshold gates. Furthermore, the book dedicates a chapter to combinational logic design in NCL, discussing completeness detection and data encoding as a dual and quad rail. Then the text covers sequential circuit design in the form of state machines in NCL.

3 THEORY OF ASYNCHRONOUS LOGIC DESIGN REVIEW

The methodology presented in chapter 4 relies on a series of well established fundamental concepts from asynchronous circuit design [1], [13], [14]. Most notable is the use of a speed-independent controller as the central control point for the design's Component modules. Then the methodology uses a 4-phase handshake protocol and the bundled-data delay-matching line model for communication between elements. Finally, a CSP derived language is used to describe speed-independent controllers' behavior as an additional layer on top of the STG representation. This section serves as a brief overview of the fundamental concepts and points to proper literature for more in-depth discussion.

3.1 The input-output mode asynchronous sequential system

Used as the central core element by the methodology, the controller element design is modeled as sequential, speed-independent, input-output mode asynchronous circuits [4], [5], [9]. The input-output is one of the operating modes [9] that allows for signals to occur asynchronously without a timing restriction. Unlike the fundamental mode, the input-output circuit does not require a certain time period to pass between the signal changes. Relying on the input-output mode allows for greater flexibility in the design. Any signal can occur at any time as long as the specified order of transitions is preserved.

Specification of the speed-independent asynchronous controller in input-output mode is done using the Signal Transition Graph (STG) [4], [16]. The STG is a special form of a Petri-net [32], [33] with additional imposed assumptions required for the model to be implementable as a speed-independent asynchronous circuit. An example of an STG is shown in Fig. 1; it is a uni-directed graph in which the nodes represent either transitions or places holding a token that enables a transition. Each transition node represents a change, either a rising or falling of a single signal in the system. The represented signal

could be an input or an output line and sometimes an internal connection that connects to the circuit feedback, but it is not a part of the element's interface.

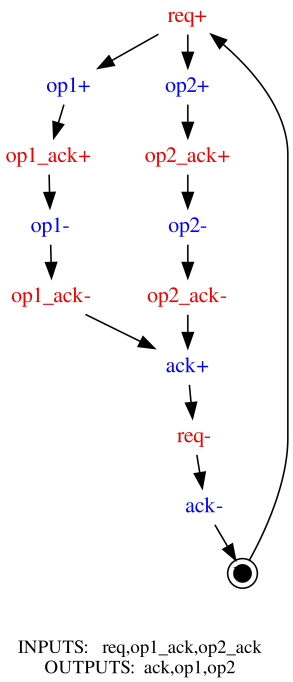


Fig. 1. Sample signal transition graph.

The second element next to the transition within the STG model is the place node. A place node's purpose is to indicate a token's existence at a specific execution point that enables the following transitions to fire. Depending on how a place node is positioned, the STG fragment might be a choice or a parallel construct and similar [4], [32]. For example in the Fig. 1 a place between ack- and req+ carries a single token indicating that the expected next transition is exclusively the req+. As a second example, the post-set of req+ which are the op1+ and op2+ are connected through two implicit and empty place nodes indicating the execution splits into two parallel paths. An implicit place exists when its description within the STG is unique, meaning there is only one possible way to specify

the place at a given position in the graph. As a result, the implicit place is not drawn, and instead, a direct line follows from one transition to the other.

Altogether, the STG graph models a speed-independent system's behavior by graphing the flow of signals that consist of the asynchronous circuit workflow. The STG model can capture the system at any given state. However, the most common is the initial state shown in Fig. 1, which serves as the model state-space entry point used for synthesis. However, it is possible to show the system in an arbitrary state by explicitly drawing the corresponding set of place nodes that hold the tokens at a given moment. The STG representation strength is the ability to represent sequences of events capturing the causality and parallelism but without any information about timing.

3.1.1 State Graph representation and Complete State Coding

Continuing, the next step in the synthesis of the asynchronous speed-independent controller involves translating the STG into the State Graph (SG) [4], [9]. The SG model serves as a bridge between abstract STG and the final Boolean next-state equations describing the system's behavior. In the SG model, the system is represented as a series of nodes in which each node holds information about the system's current state and the next transition(s). When converting to SG, all the system's signals, meaning all inputs, all outputs, and all internal lines, are bundled together into a state vector in which one vector element represents one signal. Each element in the vector is then described using a four-value number system. The value of 1 or 0 represents a stable signal that does not change due to any immediately following transition. Then the value of R represents the current state of 0, but the signal is excited and can potentially rise during the next transition. Finally, the state value of F is a counterpart of R; for the F state, the current value is 1, and the signal is excited to potentially fall as a result of the next transition.

An example is shown in Fig. 2 in which (a) is the STG of the system and (b) its corresponding SG. The SG state vector contains, starting from the leftmost element, the inputs to the system, then the outputs in order as they appear in the legend at the bottom of the figure. The edge between the place with a token and req+ shown in (a) corresponds to the s1_R000 in (b). At this state, the STG shows that the next transition possible will be the rising req. The existing token within the place node and destination transition of req+ indicate state value R for the req signal within the state vector. Consequentially, the distribution of tokens within the STG and the post-set transitions out of the place nodes determines the state vector's value. Therefore, a single state in SG represents a single state of STG and its current marking assignment to the place nodes. Also, note that the s1 state is marked as the SG initial state corresponding to the initial marking assignment in STG.

While describing the speed-independent system's state, the SG model also introduces potential inconsistencies in the model. Ultimately the 4-value representation collapses to binary data when the system is implemented in hardware, during which a single element in state vector becomes a single bit. Looking at states s2 and s6 in the segment (c) of Fig. 2, both have the same value. However, when referring back to (b) of the same figure, these two states represent the system in two completely different moments with different upcoming transitions. Therefore, the 4-value data representation collapsing into binary form can result in a loss of information and an invalid model.

An additional step must be performed to restore the model to its correct state. This additional step involves introducing new signals that result in a model regain the property called Complete State Coding (CSC) [4], [9]. Adding additional CSC states split the model into regions in which the added CSC signal has different values. The goal is to divide the model into regions such that one of the previously overlapping states lies within one region while the second ends up situated in another. Adding a single CSC signal extends the state vector by one element that serves as the differentiating factor when the

representation is collapsed to the binary number system. The Fig. 3 shows a system after the CSC is achieved. The previously overlapping states, now having names s7 and s6, have a 5th element in the state vector that contains a different value for each state, allowing to distinguish between them. The fifth value comes from adding the csc0 signal that changes to 1 before s6 occurs and changes back to 0 past s6 and before s7. With the csc0 signal, the model can uniquely represent each state, thus eliminating inconsistencies.

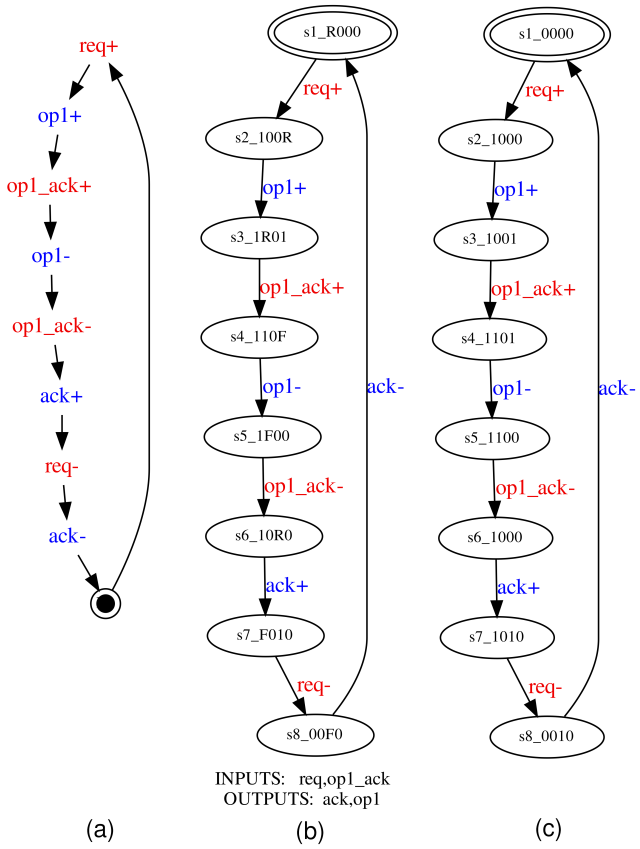


Fig. 2. Simplified model (a) STG (b) SG 4-state (c) SG binary.

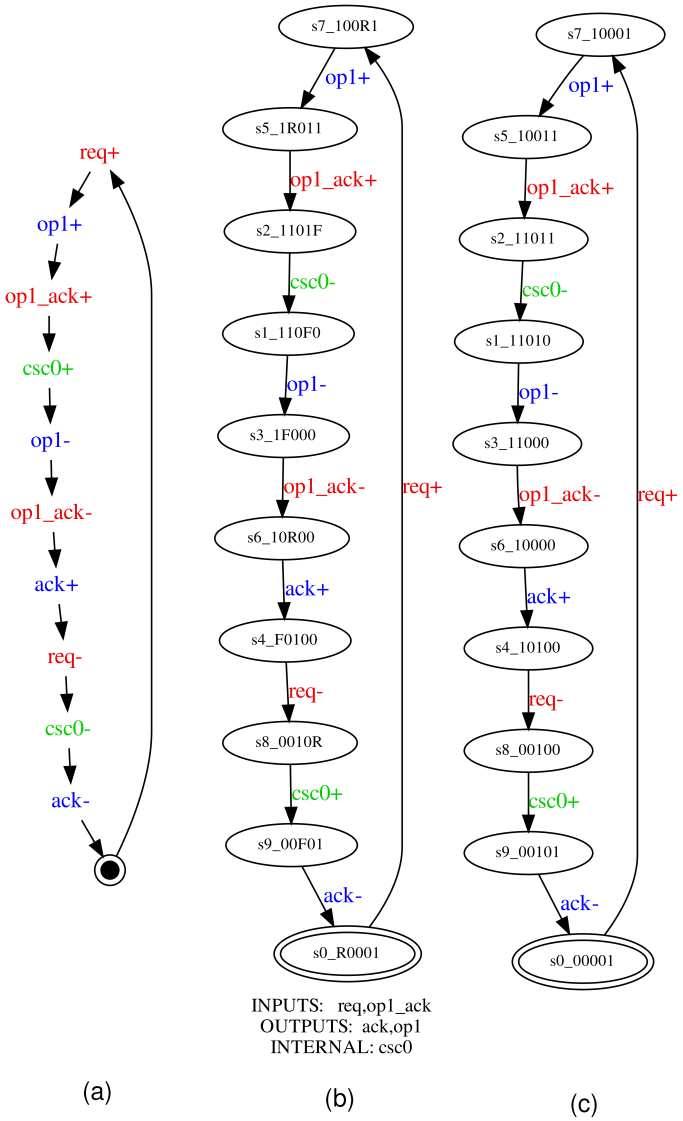


Fig. 3. Simplified model with CSC (a) STG (b) SG 4-state (c) SG binary.

3.1.2 Synthesis to gate level representation.

After obtaining the SG with CSC comes the last step of the synthesis that results in a set of Boolean next-step equations that implement the system. Then the final mapping to gate primitives. The synthesis procedure aims to obtain a Boolean function for every output and every internal signal within the controller. The input signals are controlled by

the external environment and are considered only as variables in the output and internal signals' equations. However, the model assumes that the input signals behave according to the implementable STG model's specification and follow the rules of the speed-independent model.

For each signal of interest, the first step in the process is dividing the state space into four distinct regions relative to the processed signal. First are two excitation regions (ER(s+) and ER(s-)) one for an excited rising and the other for an excited falling signal. An excitation region covers all states in which a given signal is excited, meaning it can rise/fall next. Two separate excitation regions exist, one for rising and one for falling. The second pair of regions is the quiescent regions (QR(s+) and QR(s-)) that contain all the states for which the given signal is stable high or stable low. An example region division is shown in Fig. 4 and applies to the signal op1, for which state value is situated at the state vector's second rightmost position.

The obtained regions provide sets of cubes from which cover functions are derived. The derivation method varies with the implementation approach taken; the two main approaches are the complex gate and the C-element [4], [5], [9]. The complex gate variant targets implementation through custom CMOS pull-up and pull-down networks treating any Boolean function as a single entity. The C-element approach uses the Mueller C-element [34] as a memory element reducing the complexity of the cover functions simplifying Boolean logic, and provides better resistance to logic hazards. The proposed methodology focuses on implementing the speed-independent controllers using C-elements exclusively.

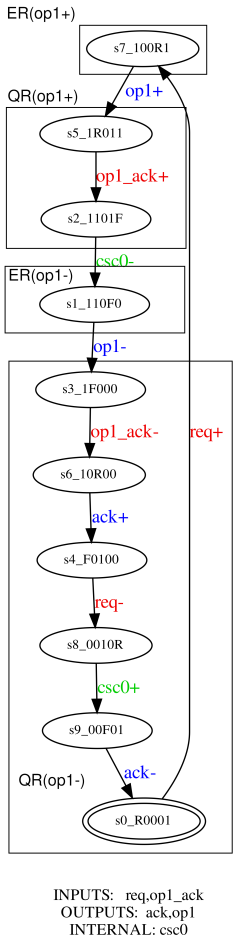


Fig. 4. Regions for signal op1.

$$C_{next} = AB + C(A + B) \tag{1}$$

The C-element is a type of memory element in which the state change occurs when both inputs are either high or low. The Equation 1 and truth table in Table 1 show the function of the C-element. When all the device inputs reach a low state, the output also becomes low. Similarly, when all the inputs become high, then the output switches to high. At any given time, when the inputs have different logic states, the element preserves the last set value.

Table 1
C-element Truth Table.

a	***b***	***c***	***c****
0	0	–	0
0	1	0/1	c
1	0	0/1	c
1	1	–	1

The Boolean next-state functions' derivation using the C-element focuses on obtaining a pair of a set and reset equations per C-element. The set equation covers all the cubes (states) in the rising excitation (ER(s+)) region for a given signal with the option to use cubes from the quiescent stable high region to simplify the formula. On the opposite end, the reset equation must cover all the cubes in the falling excitation region (ER(s-)) also with an option to use additional cubes form the quiescent region for stable low [4], [9].

The set and reset functions are duals of each other. When one is high, the second is guaranteed to be low. An output of the set function connects directly with one of the C-element inputs, while the reset equation first goes through an inverter. When the circuit enters a rising excitation region for a given output or an internal signal, the set equation becomes high, and the reset becomes low. However, because the reset feeds through an inverter, both functions will exert a high signal at the C-element inputs causing the element to transition to high itself and preserve its state through the quiescent region. The same behavior occurs in the falling excitation region, but the values are inverted; the set function becomes low, and the reset high resulting in both feeding low values to the C-element, causing it to transition to low.

The described model, in its simplest form, is called the generalized C-element implementation (gC). Even though the gC approach alleviates some logic hazards, the set and reset equations are still expected to be implemented as complex gates. The realization

of the gC using elementary logic gates can introduce hazards leading to circuit malfunction due to the delays present in the logic gates and the interconnections. However, an implementation using discrete elementary gates is possible by following the Standard C-implementation approach, in which Boolean function synthesis satisfies the monotonous cover constraint [4], [9], [35]. Unfortunately, the Standard C-implementation alone with monotonous cover constraint introduces an AND-OR logic layer, which might require large fan-in gates not available in the standard set. However, additional hazard-free decomposition methods exist that generate fully working next-state equations built using a realistic and implementable set of logic gates [4], [9].

3.2 Handshake protocols and communication between asynchronous modules

A complex system usually extends into multiple components that need to communicate with each other. The asynchronous approach uses a handshake protocol to facilitate communication between modules [4]. One such protocol is the 4-phase handshake that is widely used by the proposed methodology. Handshake based communication defines a communicating element either as an active or a passive actor. The active entity is the one that initiates the communication while the passive reacts to the communication request. A module can also be an active-passive component, in which case it fills the active role on one interface and passive on another.

The 4-phase handshake involves four stages of the information exchange cycle. The first stage when the active side communicates the request for an exchange, followed by the second stage when the passive component acknowledges the request. When both components are done with processing the message, the third stage occurs when the active side withdraws the request, followed by the passive element withdrawing the acknowledge that indicates the end of the handshake.

An example of 4-phase setup is shown in Fig. 5. The data lines are optional, but when they exist, the data flow will go either direction. The important difference lies within the time the data lines must remain valid. In the case of d1 that travels from active to passive element, the data lines must become valid before the active element requests the transaction. An ack signal from the passive component indicates that it received the data and no longer needs it. After the ack, the active module is free to release the data lines from their valid state at any time. In the opposite case of d2, when the data travels from the passive element to the active, the data lines must remain valid for a potentially longer period. In the second case, the active side requests the data, and the passive module delivers it. Therefore, the data lines must become valid before the ack occurs. When the active module finishes with the message data, it signals it by lowering the req line, at which point in time the passive element of free to release the data lines and withdraw the ack signal.

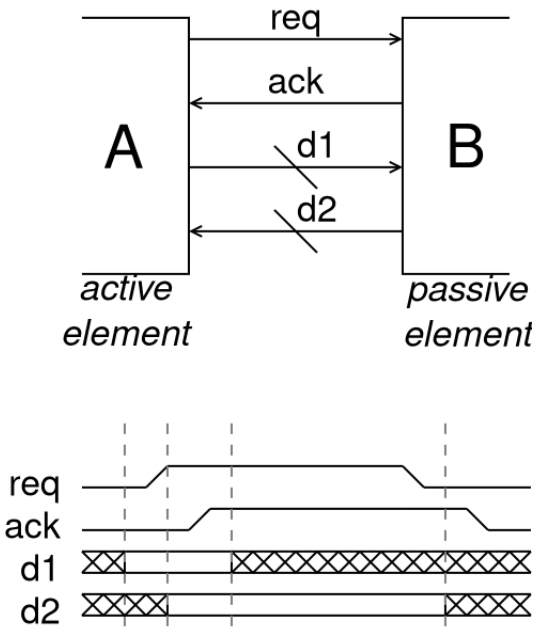


Fig. 5. The 4-phase protocol and data line validity.

Another aspect of complex systems is the existence of the data-path in parallel to the control path. The speed-independent controller alone does not have the ability to analyze and process data. The system needs additional combinational logic modules for this task. However, the data processing logic itself introduces a logic delay of which the controller must be aware to properly synchronize the events in the system. To cover the data-path components delay, a class of circuits called completion detection logic [36] exist. The simplest completion detection mechanism is the bundle delay matching approach [4], [36] that introduces a control line existing in parallel to the data-path. The control line delay must also be longer than the combinational logic long-path delay. An example of a bundle delay is shown in Fig. 6. The req signal goes high at the time when the combinational logic receives valid data. The delayed ack signal is then guaranteed to go high after the combinational function is resolved and produced stable output. The simple bundle delay model allows incorporating complex data-path logic into the speed independent asynchronous controller workflow.

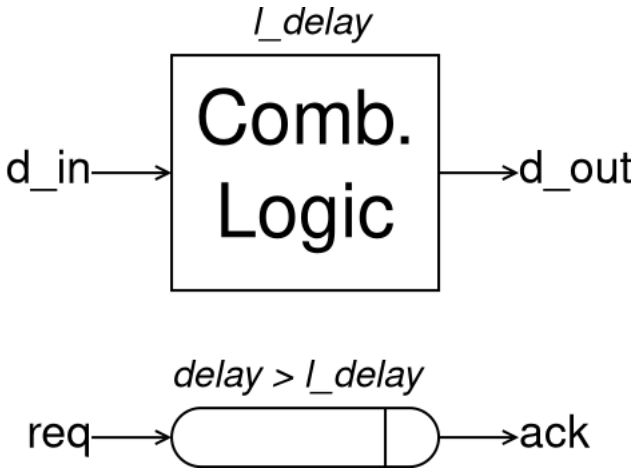


Fig. 6. Bundle delay model.

3.3 CSP notation in describing asynchronous sequential transition system

The Communicating Sequential Processes (CSP) language is a notation that specializes in modeling entities as concurrent processes that exchange information with each other through communication channels [17], [19]–[21], [37]. The syntax of CSP is widely used as a high-level description for the functionality of a variety of asynchronous circuits [17] as well as an input model for the synthesis of some classes of asynchronous circuits [22]–[24]. The language provides a variety of constructs that allow describing sequences of events similar to STG as well as parallelism and, in some cases, a notation that directly translates to STG constructs such as confusion block.

The presented work uses a subset of the CSP notation as a base for a description layer that translates directly to an STG. An example shown in Listing 1 represents the STG presented earlier in Fig. 2 segment (a). The goal of using the CSP-based modeling layer is to provide a tool allowing for a clear description of the complex execution flow of controllers, which syntax is then parsed to STG representation.

```
*[ [req+]; op1+; [op1_ack+]; op1-; [op1_ack-]; ack+; [req-]; ack- ]
```

Listing 1. Simplified STG in CSP notation.

4 COMPLEX SEQUENTIAL ASYNCHRONOUS LOGIC DESIGN METHODOLOGY

The presented methodology specifies a workflow for the design of complex asynchronous sequential circuits. The primary focus is the class of circuits in which the device executes a multi-step algorithm. An example case of a multi-step algorithm is a search algorithm that involves iteration over the data-set, comparing two items with each other, then, if applicable, performing a memory swap. In traditional synchronous circuits, such multi-step functionality can be implemented either by using a pipeline or a state machine, or a combination of both.

The pipeline approach provides a structurally simpler model and higher throughput but suffers from a lack of flexibility. The processed data must pass through every stage, even when that stage might not execute any actions, such as a conditional execution that flow depends on the data content. Also, a complex multi-step algorithm might require a potentially long pipeline resulting in a large design. The state machine-based model offers a different set of characteristics from the pipeline oriented approach. The state machine that drives the execution is free to use only the components necessary for the specific path and skip idle steps during conditional execution. The model structure is more flexible, allowing for potential reuse of components at different processing stages, resulting in a smaller circuit. However, the state machine model does not favor throughput, making it challenging to parallelize multiple tasks due to shared resource availability.

The proposed methodology targets the state-machine-oriented approach to provide a framework for implementing the multi-step sequential behavior using asynchronous logic in a form that scales well with the rising complexity of the design. The methodology breaks down the design process into a two-stage model. Stage number one delivers a high-level behavioral representation of the device for initial design capture and functional verification. Stage number two reuses parts of the code from the previous stage and

replaces certain constructs with ASIC technology-dependent structural modules that deliver a model ready for synthesis.

4.1 Model template organization

The approach builds on top of the input-output, speed-independent sequential asynchronous circuit concept [9]. The sequential asynchronous circuit delivers the central controller component. The methodology then introduces extensions on top of the speed-independent core creating a complete ecosystem for design specification.

The first addition is the model template specifying the structure of the functional elements and the building blocks' interoperability rules. The template introduces a structure in which the core speed-independent controller works with a set of data processing components supporting the execution. The methodology specifies an implementation structure template for each module and defines the operating rules for all elements. The Fig. 7 presents a high-level overview of the structure of component templates, which form the complete description of the design. The hierarchical layout follows the divide-and-conquer philosophy favoring decomposition into elements with a clearly defined scope. The approach makes it easier to write unit tests with clearly stated expectations from each component and enhances each building block's testability as a separate entity. The specified unit templates are:

1. The C-Layer and the next-state elements both forming the speed-independent asynchronous controller. The two modules are the final output of the CSP model's compilation, which is first translated to STG then processed by Petrify. The next-state module contains a set of the next-state Boolean equations derived by Petrify for each controller output signal. The C-Layer provides the "standard-C Architecture" [15] layer implementation and consists of multiple instances of C-elements and pass-through gate configurations that correspond to its next-state equations.

2. Flow support elements which are working in tandem with the controller module and perform data manipulation and data analysis tasks. A flow support element consists of the combinational logic designed for a specific task and an asynchronous handshake layer for control. Communication with the controller occurs through a 4-phase handshake protocol. Flow support elements execute various tasks, such as producing data or condition resolution that steers the processing path.
3. The collection of additional modules such as Mutex and an Asynchronous Register all supporting the controller and flow support elements' work.
4. The Component modules and the Top module. They are two module types that tie together the entire design. The Component module encloses one or more controllers, a set of flow support elements, and any additional modules such as registers and mutexes. The Component element models a part of the circuit that performs a specific task, representing some logical portion of functionality. The Top module ties together one or more components into the final full structure. Aside from connecting lower-level elements, the Components and the Top module can provide additional signal routing and manipulation logic.
5. The support module that exists exclusively in the stage two model. Its purpose is to isolate any signal routing and manipulation logic present within the Components and the Top module for the logic synthesis purposes. The reason behind introducing the support module only in stage two is to provide a more flexible design environment at earlier stages without introducing additional baggage of required modules.

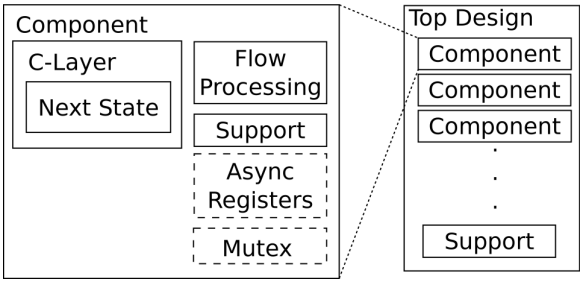


Fig. 7. The implementation template overview.

The second addition is the CSP [17], [19] to the STG translation layer. As the complexity of the controller grows, the standard STG graph representation becomes challenging to maintain. The methodology proposes an approach to specify the speed-independent controller functionality using a CSP based language. The CSP language approach offers cleaner syntax and emphasis on modularization. The notation scales with the model's growth in complexity and size better than the STG graph representation. The presented work features a reference implementation of a parser converting CSP to the STG model accepted by the Petrify tool. The Petrify [15] is used to provide the implementation algorithms that synthesize the STG model of the speed-independent circuit into the set of next-state equations and the layer of C-elements [16].

4.2 Part 1: Stage 1 Model for behavioral design and verification.

Stage one model is a behavioral representation that encompasses all the device functionality and allows for functional verification. This stage provides an environment that assumes the current design is a proper speed-independent circuit. A proper speed-independent circuit means that all the critical delays and the C-Layer exist in the behavioral form with timing adjusted such that the circuit meets speed-independent constraints. All the components that contain combinational logic, including the controller, are fully implemented. The delays are arbitrary but assume the correct working scenario;

for example, the delay-matching line of a flow support element must exceed the delay of its internal combinational logic. Stage one primarily allows for the development and execution of functional tests to verify if the circuit behaves as expected when delay-matching is assumed correct. Tests developed for stage one are then reused during the gate-level simulation, with accurate delays from the ASIC library.

For the stage one model, the methodology specifies three design steps. An overview of the workflow and inter-dependencies between each element is shown in Fig. 8. The first step involves specifying every controller workflow in CSP notation and the HDL implementation of the flow support elements. The controllers and flow support elements are interdependent. For maintainability purposes, the signal names in both models should be the same if they are connected. For example, if the controller expects to invoke specific functionality in a flow support element using an arbitrary fnc_req signal, then the flow support element interface should provide this signal line under the same label.

The second step involves generating the speed-independent circuit from the CSP model. During this stage, the CSP code is parsed by the CSP to STG translator, and the output STG is then synthesized by the Petrify tool producing circuit model describing the set of next-state equations and the C-Layer. Although step two is automated, this phase might fail and require returning to step one to perform corrections. The first possible reason for step two to fail are errors in CSP description, which could be a syntax error, or the implementation might not reflect the design intended. The use of CSP does not eliminate the need for STG in the design process. The generated STG should still provide an additional view of the design. A second potential point of failure is related to the speed-independent circuit derivation process's properties and limitations. The currently used algorithms provided by Petrify are not guaranteed to succeed in generating the circuit. Standard problems of state explosion or Complete State Coding task complexity can prevent the tools from obtaining the solution [16]. In this case, returning to the initial

design documentation is necessary to modify the controller's execution flow or breaking the controller implementation further into several smaller controllers.

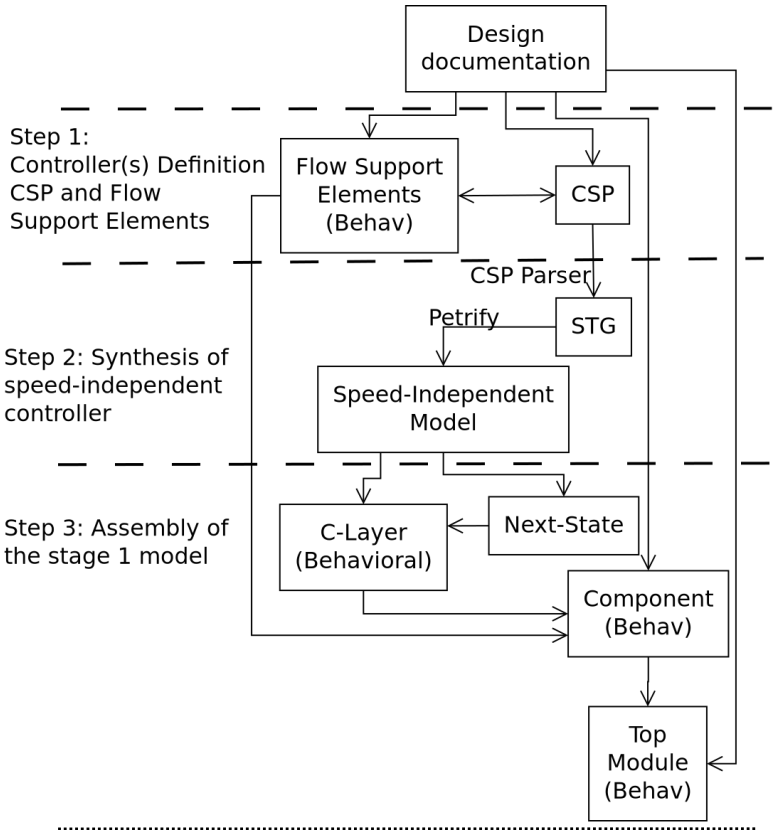


Fig. 8. Methodology steps leading to stage 1 model.

The third step is the final step that ends with a complete codebase for stage one. During step three, the controllers' HDL implementation, Components' definition, and the Top module are derived and then connected. Step three produces an HDL representation of the design, which is ready to perform functional tests. Stage one model is technology-independent; all the delays are arbitrarily defined such that the circuit behaves like a proper speed-independent logic. In stage one model, the flow support elements consist of two artificially assigned delays: one for behavioral combinational logic and

another for the delay-matching line if the bundled data delay model is used. The delay-line delay must be greater than the delay of combinational logic.

The controller itself at this stage provides two layers: the next-state equations treated as complex gate elements with a single delay and a behavioral model of the C-layer with behavioral C-element and passthrough elements. The stage one model provides the baseline for developing and executing functional tests without going through the technology-dependent synthesis phase yet. Such an approach allows for the disconnect of the process from potentially lengthy synthesis, allowing quick design iterations. The behavioral HDL model is also much easier to debug than the post-synthesis gate-level representation.

4.2.1 Using CSP in modeling sequential behavior of the controller

The speed-independent controller circuit plays a central role within the design methodology, in which it drives the execution of the working algorithm. The input-output speed-independent controller circuits are modeled using a from of Petri-net called Signal Transition Graph (STG) [4]. The STG is a bipartite directed graph [38] composed of two types of elements, the transitions and the places that together reflect the state changes occurring within the circuit throughout its operation. The synthesis process of an input-output type of asynchronous controller based on the STG graph carries inherent drawbacks. The most notable is the state explosion, caused primarily by parallelism in STG, which results in a very large state graph (SG). Eventually, the state graph size might grow large enough such that it is infeasible from the point of computational processing resources to synthesize the model and perform the complete state coding step.

The proposed methodology favors the divide-and-conquer approach in dealing with complex models to help overcome the "State-based" [16] synthesis limitations. Instead of one large controller, the component element can host multiple smaller ones working

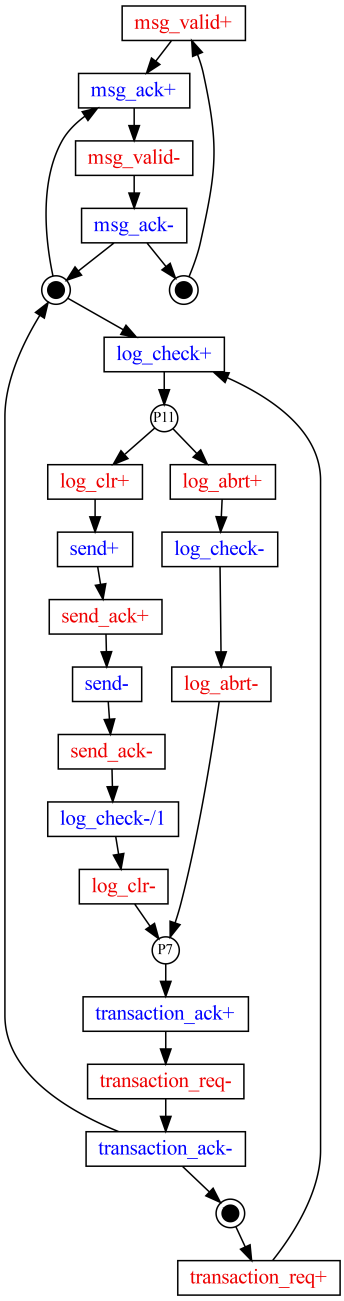
together. For example, a scenario in which the central main controller and smaller utility controllers work together in the Active-Passive configuration. When a single controller unit becomes too large, and there exists an STG fragment within the original STG graph that can be separated, then it is possible to break down the design into two speed-independent controller units given that input and output signals are not shared between the two. Some signals might need additional gating within the containing component module, and an additional communication sequence must be established between the controllers.

The second problem with the speed-independent controller is the maintainability of the STG model as the complexity of a circuit grows. Large STG graphs become difficult to understand and maintain. The proposed methodology uses a language based on the subset of CSP with additional extensions to describe the speed-independent controller replacing the direct description in STG. The CSP representation is more compact and easier to maintain at the source code level; it scales better with model size by allowing for representation of STG fragments separately and favors reusability. The presented work features reference implementation of a CSP to STG parser. An example CSP code is shown in Listing 2.

The CSP code expressiveness is superior to the STG, with better modularity and the ability to express large and complex transition flows. The source code resembles a well know imperative programming paradigm. However, the CSP model is not a sequential program but a structural model of transition flow in a speed-independent asynchronous controller. The syntax follows standard CSP [17], [19], [20], [24] with custom additions. The Fig. 9 shows the STG graph which is a result of executing the parser on the code in Listing 2. The current version of the parser outputs the STG file in the format accepted by Petrify. The Petrify is then used to synthesize the STG into a final set of next-state equations and corresponding C-layer.

```
module controller;

inputs msg_valid transaction_req log_clr log_abrt send_ack;

outputs msg_ack transaction_ack log_check send;

explicit_place P0;

loop_place [msg_ack-]->[msg_valid+];
loop_place [transaction_ack-]->[transaction_req+];

marking [msg_ack-]->[msg_valid+];
marking [transaction_ack-]->[transaction_req+];
marking [msg_ack- transaction_ack-]->[msg_ack+ log_check+];

main sndr_main;

fragment frag_msg_ack_flow;
fragment frag_transaction_flow;

endinterface

sndr_main:
    frag_msg_ack_flow : frag_transaction_flow

frag_msg_ack_flow:
    *[
        [ msg_valid+   ];
      P0=>msg_ack+      ;
        [ msg_valid-   ];
          msg_ack-=>P0  ;
    ]

frag_transaction_flow:
    *[
        [ transaction_req+ ];
      P0=>log_check+       ;
        [(
            [ log_abrt+ ];
              log_check- ;
            [ log_abrt- ];
         ) | (
            [ log_clr+  ];
              send+      ;
            [ send_ack+ ];
              send-      ;
            [ send_ack- ];
              log_check- ;
            [ log_clr-  ];
         )
        ];
          transaction_ack+       ;
        [ transaction_req-      ];
          transaction_ack- =>P0  ;
    ]
```

Listing 2. Example CSP model.



INPUTS: msg_valid,transaction_req,log_clr,log_abrt,send_ack
OUTPUTS: msg_ack,transaction_ack,log_check,send

Fig. 9. Example STG model.

4.2.2 The Controller

After being processed by the translator and then by the Petrify tool, the CSP model results in a set of next-state equations describing the speed-independent controller's behavior. The Petrify tool outputs its description in form shown on the Listing 3. The model consists of the combinational logic implementing the next-state equations and a layer of C-elements that serve as memory elements and prevent some logic hazards. The proposed methodology specifies the template in SystemVerilog, which specification spans over two design stages dictating the approach to implementation of the asynchronous circuit controller in a way that favors verification and maintainability of the design.

```
SET(log_check')   = send_ack' csc0'
RESET(log_check') = msg_ack' transaction_req csc0
[log_check] = log_check'        (output inverter)
  > triggers(SET):     csc0- -> log_check-   send_ack- -> log_check-/1
  > triggers(RESET):   (msg_ack-,transaction_req+) -> log_check+
  > 5 transistors (3 n, 2 p) + 2 inverters
  > Estimated delay: rising = 32.96, falling = 24.12

send' = csc0' + log_clr'
[send] = send'        (output inverter)
  > triggers(SET):     csc0- -> send-
  > triggers(RESET):   log_clr+ -> send+
  > 4 transistors (2 n, 2 p) + 1 inverters
  > Estimated delay: rising = 24.21, falling = 16.62
```

Listing 3. Example output from Petrify.

The Listing 3 shows a representative example of an output from the Petrify tool for two cases. The first case for the log_check signal implemented using the C-element, and the second case the send signal, which is a direct signal. The direct signal scenario occurs when the "combinational optimization" [9] is applied for which the C-element becomes unnecessary. For each case, exist two variations, with or without the inverter at the output.

The HDL implementation of the controller consists of two modules the next-state and the C-layer module. The next-state module provides the implementation of the combinational logic describing all next-state equations in the given controller. It is a

purely combinational element without any capabilities to store information. An example template for the next-state module is shown in Listing 4.

```
module ctrl_nextstate (
    input logic in1,
    input logic in2,
    input logic out1,
    input logic out2,
    input logic csc0,

    output logic out1_on,
    output logic out1_off,
    output logic out2_sig,
    output logic csc0_on,
    output logic csc0_off
);
    timeunit        1ps;
    timeprecision   1ps;

    import conf::CD; // Combinational delay

    always_comb begin
        out1_on     <= #CD <nextstate expr>;
        out1_off    <= #CD <nextstate expr>;
        out2_sig    <= #CD <nextstate expr>;
        csc0_on     <= #CD <nextstate expr>;
        csc0_off    <= #CD <nextstate expr>;
    end

endmodule: ctrl_nextstate
```

Listing 4. NextState module template.

The module requires the specification of the inputs, outputs, and the set of Boolean expressions. The inputs list consists of:

- All input signals to the controller.
- All feedback output signals coming out of the controller.
- All internal signals, particularly the CSC lines generated as a result of the complete state coding step during STG synthesis.

The input signals introduce a set of constraints preventing inconsistencies and potential errors in the implementation. First, all the input signals are always single bit data lines. Secondly, the next-state module must not accept any inputs other than the signals

used in the CSP model. Finally, no additional logic except the next-state Boolean expression is allowed within the module.

The second group consists of the output signals. There are two types of output signals. The first output type is the pair of signals that implement set and reset functions and connect to the C-element. Every output pair leading to the C-element has the naming convention signame_[on|off] which leads to a signal name followed by either on or off suffix. The second output category is the signals leading directly out of the module without the C-element. The direct signals are always a single line with the name in the format <signame>_sig. The Fig. 10 shows an example controller structure with signal naming convention and their connections.

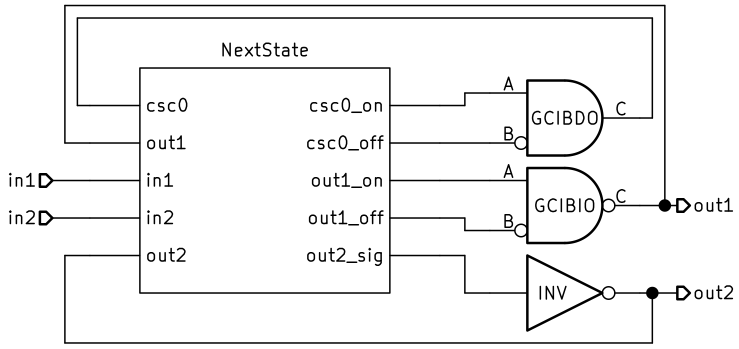


Fig. 10. Stage 1 asynchronous controller structure example.

The final third group within the next-state module is the set of the next-state equations. It is a standard SystemVerilog syntax implementation of combinational logic. However, to correctly simulate the circuit delays, the behavioral logic delay must be modeled as a transport delay, as seen in Listing 4. The use of the inertial delay model is incorrect and can cause the simulation to skip some short hazardous signal spikes leading to potential undetected errors in the design at the stage 1 level model.

The second and final element defining the controller implementation is the C-layer module. The C-layer module models the layer of C-elements and direct connections; it

also contains the next-state component within itself. The Listing 5 shows the example template for the implementation codebase. The module defines all the inputs, outputs and internal signals of the controller. It instantiates the next-state component, the set of C-elements and provides feedback signal lines.

```
module ctrl (
    input   logic rst,
    input   logic in1,
    input   logic in2,
    output  logic out1,
    output  logic out2
);
    timeunit        1ps;
    timeprecision   1ps;

    //internal signals
    logic csc0;

    //signals coming out of the comb module
    logic out1_on,
    logic out1_off,
    logic out2_sig,
    logic csc0_on,
    logic csc0_off

    //module output signals
        logic out1_out;
        logic out2_out;

    ctrl_nextstate
    NS  (.in1(in1), .in2(in2), .out1(out1), .out2(out2), .csc0(csc0),
         .out1_on(out1_on), .out1_off(out1_off), .out2_sig(out2_sig),
         .csc0_on(csc0_on), .csc0_off(csc0_off)
    );

    //C-Layer connections
    c_elem #(.DELAY(conf::C_DELAY), .RESET_VAL(1'b0),
             .INV_OUT(1'b1), .INV_A(1'b0), .INV_B(1'b1))
    c0      (.a(out1_on), .b(out1_off), .c(out1_out), .rst(rst));

    assign out2_out = ~rst * ~out2_sig;

    c_elem #(.DELAY(conf::C_DELAY), .RESET_VAL(1'b1),
             .INV_OUT(1'b0), .INV_A(1'b0), .INV_B(1'b1))
    c1      (.a(csc0_on), .b(csc0_off), .c(csc0), .rst(rst));

    //Output connections
        assign out1 = out1_out;
        assign out2 = out2_out;
endmodule: sndr_main_ctrl
```

Listing 5. Stage 1 C-Layer template.

Consistently with the next-state module, the C-layer provides behavioral C-element instances or assigns the signal directly if the synthesized output from the Petrify tool defines it as such. If the synthesized output generated by Petrify specifies an output inverter, then it is the C-layer that provides the inverting component at the output signal.

The speed-independent standard C-element model generated by Petrify results in four possible signal configuration:

- Element that resets to 0 with direct output.
- Element that resets to 0 with inverted output.
- Element that resets to 1 with direct output.
- Element that resets to 1 with inverted output.

The behavioral C-layer implements all the scenarios involving C-element at the level of stage 1 model through a parametrizable C-element module shown in the Listing 6. The module upon instantiation is set to the specific configuration using parametrization. Similarly, for the direct signals, the initial state is achieved through negating the output signal from the next-state module and correct use of the reset functionality. If a direct signal resets to 0, then the "AND" operation joins the next-state signal and the inverted rst line. If the direct signal resets to 1, then the OR function is used, and the rst line is not inverted.

The output log provided by the Petrify tool shown in the Listing 3 is missing information about the initial state of the controller circuit. The initial state is a critical piece of information that determines the starting point of the system. When using Petrify, the synthesized system's initial state must be obtained from the SG, in which it is explicitly indicated. Therefore, the SG graph must also be generated for the same STG for which the next-state equations were obtained, specifically the post-CSC STG, which potentially contains generated by Petrify internal signals.

```
module c_elem #(
    parameter           DELAY       = 100ps,
    parameter   logic   RESET_VAL   = 1'b0,
    parameter   logic   INV_OUT     = 1'b0,
    parameter   logic   INV_A       = 1'b0,
    parameter   logic   INV_B       = 1'b0
)(
    input   logic   a,
    input   logic   b,
    output  logic   c,
    input   logic   rst
);
    timeunit        1ps;
    timeprecision   1ps;

    logic   i_c;

    always_latch begin: MAIN
        logic   ret;
        logic   v_a;
        logic   v_b;

        v_a = a ^ INV_A;
        v_b = b ^ INV_B;

        if(rst == 1'b1) begin
            ret = RESET_VAL;
        end else if((v_a == 1'b0) && (v_b == 1'b0)) begin
            ret = 1'b0 ^ INV_OUT;
        end else if((v_a == 1'b1) && (v_b == 1'b1)) begin
            ret = 1'b1 ^ INV_OUT;
        end else begin
            ret = c;
        end

        i_c = ret;
    end: MAIN

    always_comb begin: OUTPUT
        c <= #DELAY (i_c);
    end: OUTPUT


endmodule: c_elem
```

Listing 6. Behavioral C-element.

The next-state and the C-layer modules' implementation provides a working controller design ready to use in the stage 1 model. The controller circuit specifies the flow and order of events occurring in the system, but it operates only on the control signals. The controller is unable by itself to operate on the data processed by the device. As such, the

controller operation must be supported by the flow support elements (FSE), which form the system's datapath.

4.2.3 Flow Support Elements

Flow support elements are the next elementary building block type proposed by the methodology that forms the data path and operates in tandem with the controller in the Active-Passive configuration. The controller is the Active side that requests an action while the flow support elements are Passive and operate on the data lines, thus bridging the data path with the control path. Structurally the flow support elements vary depending on their function, but all variants share some similarities, like communication through handshake protocol. The Fig. 11 shows example types of the elements.

The methodology specifies two elementary FSE types the data type and the decision type. Also, variations are allowed as long as they conform to the FSE's operating rules. The data variant of the flow support element focuses on data manipulation tasks. The element receives some input, computes and outputs the result at the controller's request. The controller initiates the action using a handshake protocol. The data modules serve as data processing elements, for example, an element that formats message packet placed on the bus. The Listing 7 and Listing 8 shows HDL template for implementation of the datatype flow support element.

Similar to the speed-independent controller, the flow support element also consists of two building blocks, the function module and the handshake layer. The function module shown in Listing 7 is a purely combinational module implementing the data path function. The function module's interface contains all the combinational inputs, the combinational outputs, and the handshake request signal. The function module does not store any state and must be composed only of the combinational logic.

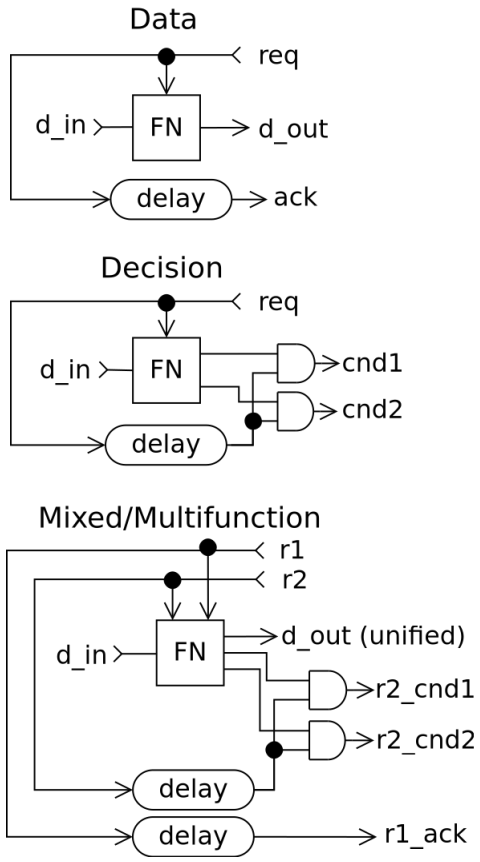


Fig. 11. Example types of flow support elements

The second module is the handshake module. It is a wrapper layer over the function module that introduces the 4-phase handshake communication mechanism used to communicate with the controller. The handshake layer's primary purpose is to signal FSE's readiness to the controller and use the handshaking to filter out any hazards occurring when the function module internal logic settles. The version presented in the Listing 8 is specific to the stage 1 model and uses the bundle-delay completion detection mechanism. The data flow support element introduces a single bit unconditional acknowledge signal ack that always appears following the request from the controller. The ack signal in the bundle-delay model is delayed for the time period exceeding the long path within the function module. The stage 1 simulation model implements a specific

```
module fse_t1_fn (
    //Data
    input   logic   [3:0]   d_in,
    output  logic   [3:0]   d_out,

    //Control
    input   logic           req
);

    timeunit        1ps;
    timeprecision   1ps;

    always_comb begin
        if(req == 1'b1) begin
            if(d_in == 4'hF) begin
                d_out = 1;
            end
            else begin
                d_out = d_in + 1;
            end
        end
        else begin
            d_out = 0;
        end
    end
endmodule: fse_t1_fn
```

Listing 7. FSE data-type combinational module.

```
module fse_t1 (
    //Data
    input   logic   [3:0]   d_in,
    output  logic   [3:0]   d_out,

    //Control
    input   logic           req,
    output  logic           ack
);

    timeunit        1ps;
    timeprecision   1ps;

    logic   [4:0]   d_out_next;

    fse_t1_fn
    FN  (.d_in(d_in), .d_out(d_out_next), .req(req));

    always_comb begin
        d_out   <= #conf::COMB_DELAY d_out_next;
        ack     <= #conf::FNC_D1 req;
    end
endmodule: fse_t1
```

Listing 8. FSE data-type handshake module.

delay configuration under the assumption that the logic specifies the correct circuit behavior. Therefore the combinational outputs are delayed together with the ack signal, but the ack delay must always be longer.

The second type of flow support element is the decision element. These elements provide the ability for the controller to perform branched execution based on the data. Instead of data path signals, the decision element outputs two or more single-bit lines that end up back as inputs to the speed-independent controller. This way, the controller can perform conditional execution implemented as an STG confusion block construct. The request signal initiates the operation, after which exactly one of the conditional lines become high, steering the controller's signal transition in a specific direction.

Similarly to the data FSE, the decision FSE type splits into function and handshake modules. However, in the case of the decision element, the output is always a set of single line signals representing the final resolution of the combinational logic function. For example the Listing 9 shows a template for a decision module which receives 4-bit vector and outputs two lines sel_a and sel_b. Based on the vector's bit pattern, the module selects one of the output lines.

The output signals are then gated and delayed within the handshake layer. As shown in the Listing 10, the handshake layer for the decision element employs a different structure than the data variant. The ack signal appearance is now conditional on the output from the function module. The handshake module now has two ack signals: sel_a and sel_b, in this specific case. Both signals are delayed according to the bundle data delay model and gated by the request signal sel_req. The gating is essential to avoid any potential hazards caused by signal transitions at the combinational inputs while the module is not in use and could potentially trigger an unexpected transition in one of the ack signals causing the circuit to malfunction.

```
module fse_t2_fn (
    //Data
    input   logic   [3:0]   d_in,

    //Control
    output  logic           sel_a,
    output  logic           sel_b,
);

    timeunit        1ps;
    timeprecision   1ps;

    always_comb begin
        if(d_in == 4'b0101) begin
            sel_a = 1;
            sel_b = 0;
        end
        else begin
            sel_a = 0;
            sel_b = 1;
        end
    end
endmodule: fse_t2_fn
```

Listing 9. FSE decision-type combinational module.

```
module fse_t2 (
    //Data
    input   logic   [3:0]   d_in,

    //Control
    input   logic           sel_req,
    output  logic           sel_a,
    output  logic           sel_b
);

    timeunit        1ps;
    timeprecision   1ps;

    logic   sel_a_out;
    logic   sel_b_out;

    fse_t2_fn
    FN  (.d_in(d_in), .sel_a(sel_a_out), .sel_b(sel_b_out));

    always_comb begin
        sel_a <= #conf::FNC_D1 sel_a_out & sel_req;
        sel_b <= #conf::FNC_D2 sel_b_out & sel_req;
    end
 endmodule: fse_t2
```

Listing 10. FSE decision-type handshake module.

The flow support elements can also take different forms. A good example is the third mixed, multi-function module type. A single multi-function module can service more than one request, given that the module is performing only one of its functions at a time. The use of multi-function modules also provides a solution to a use case when the design needs to access some resources, such as a write to a register, in different scenarios, such as when the data comes from different sources at different stages of the algorithm. Instead of merging multiple FSEs by putting merge blocks then connecting to the resource, the design can provide a single multi-function module. The multi-function module can also be a mix of data and decision modules. A good scenario for using a multi-function module is when a shared resource is accessed by different entities that modify its state, or multiple blocks rely on similar or same logic used multiple times in different parts of the circuit. The second use case allows for the reuse of the combinational logic, potentially reducing the model's size.

The flow support elements in order to work together must conform to a set of rules. The first rule states that if multiple modules are accessing the same resources, they must not collide with each other. Not only their activation must not overlap, but every FSE must return all its output data lines to the default state when finished handshake. An example of the first rule is shown in Listing 7 and Listing 10 in which the data module logic unconditionally outputs value of 0 always when req signal is low and the decision module output lines are AND-ed with the sel_req signal. The first rule prevents the manifestation of unexpected signals affecting the device. The second rule is that while the flow support element is in the middle of its handshake cycle, its inputs must remain stable, valid, and do not change. If the surrounding logic could not provide stability for inputs, then the asynchronous register must be used instead of the direct connection for at least the unstable subset of inputs. The third rule is that the flow support elements are not storage elements and are stateless. All the output signals of the module must go to default

when the handshake finishes. If an output from the flow support element needs to be preserved beyond its handshake cycle, then the register must be used.

Considering the stage 1 model specifically and the accuracy of the behavioral simulation. The delay model definition within the handshake module must follow specific guidelines. The stage 1 model's goal is to simulate the circuit operation under the assumption that it is a correct speed-independent model. This assumption applies to both the controller and the flow support elements. In the case of flow support elements, the following rules must be met:

1. The assigned delay of combinational outputs must be smaller than every delay-matching line delay of the handshake components. Example in the Listing 8 the COMB_DELAY value must be smaller than the FNC_D1.
2. If a combinational output signal is combined with a handshake protocol signal, then the line receives a bundle data delay. Example the sel_a signal in Listing 10
3. There are no delays specified inside the functional module itself. All the behavioral delay definition resides within the handshake module exclusively.
4. Each line with an assigned bundle-data delay should have a different value assigned to this delay. An FSE input hazard or error within the design could cause an unintended switch of the controller, particularly for the decision module. Using exact delay values might cause difficult to observe error conditions. The correct signal activity could hide away glitch transitions, which later cause a malfunction in the gate-level model. Using different values of the delays can make glitchy spike transitions more visible, increasing the probability of identifying the problem during stage 1.
5. All the delays should be specified as transport delays for the same reason as in point 4. Unlike inertial delays, the transport delay model does not filter transition spikes [39].

The set of FSE components, along with the speed-independent controller working together, form the central construct allowing for the implementation of complex data-driven multi-step algorithms. The FSE modules support the controller by providing data-path-based analysis, decision making, and data manipulation capabilities. However, the FSE lacks the memory and state holding element functionality, and the speed-independent controller is incapable of handling contending input signals that require arbitration. Therefore, the set of additional asynchronous elements is necessary to fill the mentioned gaps in functionality.

4.2.4 Additional components

The set of components to be complete and to allow for modeling a broad set of designs needs to be expanded beyond asynchronous controller and flow support elements. The methodology must have the ability to provide memory capabilities and include components that perform asynchronous arbitration on signals that are beyond the control of the speed-independent controller. In many cases of more intricate designs, it is useful or even necessary to have the ability to provide a short term internal memory to preserve portions of data or to provide stable input into FSE modules. The necessity for an asynchronous register arises from the constraints of flow support elements. By design, the FSE modules require that their input remains unchanged, valid, and stable throughout the handshake cycle. However, a need for an FSE may occur with output lines that must be accessible and valid for a long duration throughout the algorithm execution. It can quickly become a difficult task to satisfy the input stability requirement for the FSE, especially when the environment changes in a way that could alter the FSE inputs before its handshake exchange ends. The asynchronous register solves the problem by preserving the data allowing to finalize handshake with given FSE early. For the asynchronous register, the write cycle operates based on the handshake protocol, preferably 4-phase.

One version of such a register could be built from the D-latches with asynchronous reset functionality and a delay line covering the D-latch setup and hold requirements to provide handshake, as shown in Fig. 12.

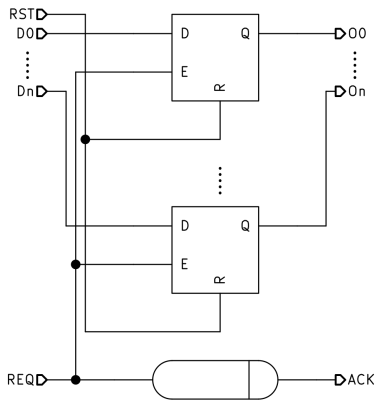


Fig. 12. Asynchronous register example.

Another use case for asynchronous register in the proposed methodology serves the purpose of performance optimization. For example, an external environment resource is required by multiple FSE modules. The controller reserves the resource for the FSE and holds it to satisfy the input stability rule for some time until the FSE handshake protocol finalizes. Reservation of the shared resource is causing other consumers to wait, effectively stalling parts of the system. The throughput could improve by storing the data needed by the FSE in an intermediate register rather than directly reading the shared element data-lines. When using the intermediate memory, the shared resource can be released as soon as the register captures the data. However, using the registers is a tradeoff. Instantiating large numbers of storage cells increases the design's area requirements and must be used with caution.

The second essential element is the asynchronous mutex. In some cases, a necessity comes to arbitrate between multiple signals coming from different device elements that do not closely cooperate. Such contention cannot be resolved by the controller alone; if

passed unchecked to the controller, the contending signals can trigger metastable behavior within the device, causing malfunction. The mutex element provides the necessary capabilities to arbitrate the access and provides a base for a variety of higher-level asynchronous arbiter circuits [14]. The most common uses of the mutex element are a gateway for which only one signal can pass at a time. The second use case is a base for an arbiter circuit in which multiple modules can request the resource, then one of the modules is selected, and the request passes forward [17], then the response is guaranteed to return only to the single selected module.

The asynchronous register and the mutex element provide the necessary minimal set of components, allowing specifying a broad set of asynchronous designs while supporting the divide-and-conquer philosophy. The asynchronous register provides means for latching data as stable input as required by the FSE input stability rule as well as temporary data storage and supports performance optimizations. The mutex element serves as an element resolving signal contention and as a base for a variety of arbiter modules. All the elements together must then be organized in a systematic means to support the maintainability of larger designs.

4.2.5 Putting the model together

The last two types of elements introduced by the methodology are the Component and the Top module. The purpose of both is to organize other elements in a consistent and easy to maintain manner. Both serve the purpose of containers to bring together other modules within the same functional group. As seen in Fig. 7, the design consists of a set of components and a single top-level module. The Component element is the elementary functional unit that contains the controllers and modules supporting the controllers, including FSE, registers, mutexes, and others. The Top module then connects all components forming the final stage 1 design. The Fig. 13 shows an example of a

Component that follows the methodology. The example is a receiver component, part of the cache coherence controller design. Its function is to process any incoming messages from the bus, potentially affecting the local cache's content, depending on the information placed on the bus. The receiver component contains one controller, two flow support elements, and a single asynchronous register.

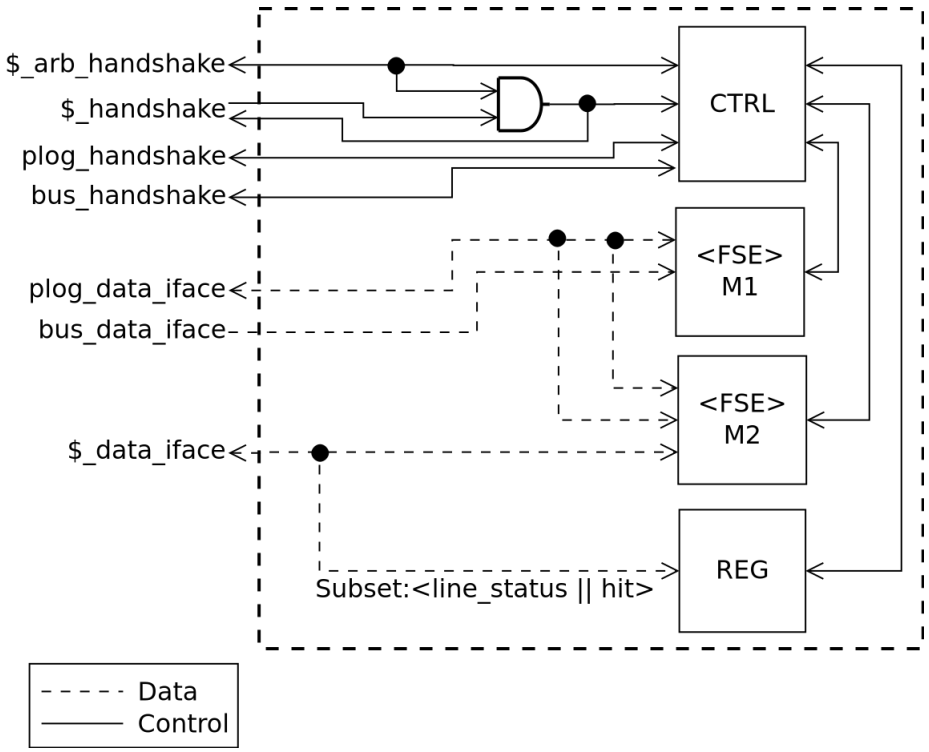


Fig. 13. Receiver component overview.

Aside from providing a platform for complex problem decomposition and functional elements grouping, the Component element handles internal signal management. The signal redirection logic within a Component must be simple and not violate logic timing in a way that would render the delay-matched lines incorrect. Proper signal management can significantly simplify the complexity of the speed-independent controller design. For example, Fig. 13, shows a signal from $_handshake interface coming to controller fed through the AND gate along with the replicated signal from $_arb_handshake. On the

global level, the receiver is one of four components that share access to the core's cache memory. The cache memory interface lines split and go to each sharing module. As a result, access to the shared cache memory is guarded by arbitration, but the same cache ack 4-phase handshake signal line travels to each sharing controller. Regardless of which component currently talks to the memory, the ack signal still propagates to all. An unexpected cache ack signal can turn the receiver controller meta-stable. It is possible to design a controller that handles unexpected transitions on the cache ack line, but it will result in a significantly more complicated STG model. Also, the STG fragment responsible for handling the unexpected ack must be operating in parallel with the controllers' regular function, significantly increasing State Graph's size. Alternatively, simple gating of the signal provides a much cleaner and less complicated solution. For this reason, the Component module is tasked with delivering, when necessary, some level of signal manipulation such as signal gating to offload complexity from the speed-independent controller. Moreover, the Component module can also provide simple signal manipulation such as splitting, merging and redirection to allow for simpler data-path interfaces among its internal functional elements.

The Component and the Top module types fulfill the function of providing an interconnect platform and support signal management. In relation to this topic, the example in Fig. 13 also shows why the flow support elements must reset to the default state upon finishing their handshake cycle. The M2 module controls cache data lines through $_data_iface interface, shared among all the modules talking to the cache memory. If the M2 element would not go to the default state, it could unexpectedly alter the cache memory component inputs affecting other modules currently working with the memory module. All FSEs need to adhere to this rule because it would be highly inefficient to add extensive signal gating within the scope of Component element signal management. Therefore, keeping the balance between participating blocks' functional scope is essential.

Finally, the Top module serves as a container for all Component elements. The top module, just like the Component, gathers together and organizes modules but at a higher level. Instead of containing controllers, FSEs and other functional elements, the Top module holds other Components providing a global container and internal connectivity. The Top element should not contain any other instances except the Component modules and necessary connectivity logic. Although this rule somewhat flexible and putting black box elements such as arbiter in Top module might be justified, elements like a speed-independent controller or FSE are not allowed. The scope of the Top element is only to interconnect components and provide the device interface. Any functional implementation should be delegated to the scope of the Component module. All the described component types form the set of building blocks necessary to express a broad set of algorithmic type asynchronous designs and lead to the Stage 1 model for behavioral simulation.

4.3 Design of the CSP to STG parser

The CSP language is a formalization of the process algebra [40] that specializes in modeling concurrent systems [19]–[21]. One of the CSP language applications is the ability to describe the functional specification of an asynchronous circuit [17]. The goal of the CSP to STG parser is to provide an automated translation of a model defining the behavior of a speed-independent controller circuit written in CSP language into the STG graph. The presented work provides an algorithm that processes a model in the CSP language and generates STG representation in a format acceptable by the Petrify tool. The presented work delivers a reference implementation of the CSP to STG parser. The reference implementation performs all the functions from pre-processing the input code through parsing to STG and generating output targeted for the Petrify tool. The proof of concept reference implementation is available at the web address given in Appendix C.

The subset of CSP used in the translation is based on the language defined in the following three publications [17], [19], [24]. The derived CSP language features a relatively simple syntax allowing for direct decomposition into tokens, which then are parsed one by one, gradually building the STG graph. The parser primarily operates based on the concept of the source code fragments; its main processing loop starts from the entry point fragment then runs through tokens contained within. The algorithm processes the simple tokens immediately, but delegates nested items such as parenthesis enclosed segments and references to other fragments for later. Finally, the parser engine iterates over the entire set with a loop until there are no unprocessed CSP fragment elements present. This breadth-first parsing approach gradually builds the graph translating nested fragments into transitions and places and injecting them into the STG structure.

To illustrate the parser's working principle, this section presents a simple waiting room ticket machine model shown in Fig. 14. The example design shows four components, two clients requesting their queue number, a number generator, and a controller overseeing the process. All components are asynchronous elements. The clients and the generator are part of the simulated test bench, and the controller module is the DUT speed-independent asynchronous controller circuit.

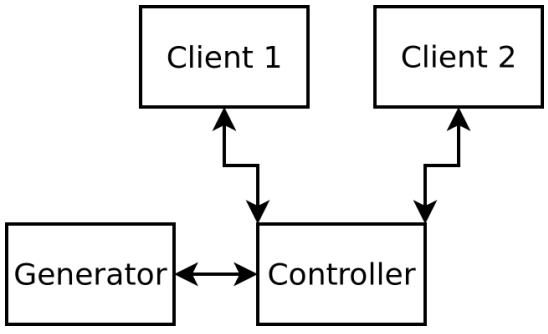


Fig. 14. Wait room system overview.

This simple model demonstrates the design problems and their solutions efficiently with a sufficient level of complexity for the discussion about CSP representation of concurrent behavior, arbitration, and general transition sequencing. The controller circuit takes the form of a passive-active device. In the presented example configuration, the client module initiates a sequence of operations in the controller. Then the controller circuit activates the transaction with the queue number generator. A next number is provided over a shared line to the requesting client. Finally, the controller completes the transaction with the generator and the client.

The CSP language defines a concept of a process, which is a single entity that performs an action. When applied to asynchronous circuit design, the CSP process models either an entire controller circuit or its fragment. Effectively the CSP process represents an entire STG diagram or an STG fragment. The semantics of CSP language is capable of expressing the fundamental Petri-net constructs such as the sequential and concurrent flow of transitions as well as conflict and confusion [4] segments. The presented methodology uses a language based on a subset of the CSP notation syntax strongly influenced by the works titled "Asynchronous techniques for system-on-chip design" [17] and "Programming in VLSI: From communicating processes to delay-insensitive circuits" [23] also adding a set of language extensions specific to the CSP to STG conversion task.

Analyzing the system in Fig. 15 and its corresponding STG model shown in Fig. 16. The interface signals layout of the modeled controller circuit is as follows. The system's inputs are the $r1$, $r2$ and *ack* signals that connect the controller to the clients. The next input signal is the *rdy* signal coming from the generator. The outputs of the controller are $r1ack$, $r2ack$, *resp*, *req* and the $done$. The working sequence starts with one of the clients. This example assumes that the initiating client is the Client 1 module. The sequence starts with the client sending the request signal by asserting the $r1$ line high ($r1+$). If the controller is not busy with other requests, it immediately responds with $r1ack+$ and

begins the procedure. At this stage, the client module can bring the $r1$ signal down at any time ($r1-$) while the controller begins the communication with the generator. First, the controller puts the signal $req+$ to obtain the next number from the generator. When the new number is ready, the generator responds with $rdy+$. At this point, a valid number is present on the data bus line, and the controller performs two operations. First, the controller brings its req down ($req-$), then the controller responds to the current client with the $resp+$ signal on the shared line. When the client finishes with the data, it responds with $ack+$, and at this point, all the participating elements finish the handshaking protocol. First, the controller communicates a complete transaction to the generator through a $done+$ signal. Followed by the $rdy-$ signal from the generator, the controller sends $done-$, and $resp-$ signals then await for $ack-$ from the client. After the client responds with $ack-$ and the $r1-$, the controller emits $r1ack-$, and the system goes back to its initial state awaiting further requests.

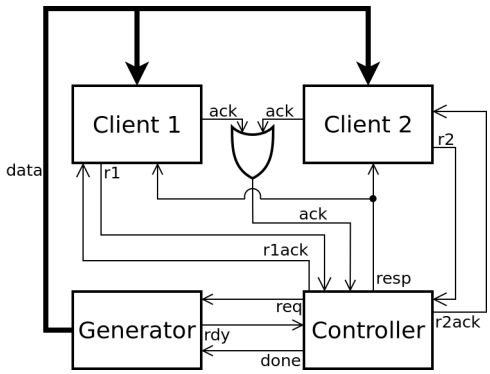


Fig. 15. Wait room system signal connections.

The STG model shown in Fig. 16 is not the only possible implementation. Alternatively, the $req-$ signal can appear in parallel with the rest of the flow up until the $rdy-$ before which both paths $req-$ and $done+$ must merge. It is also possible to put $done-$ and $resp-$ in parallel. Situations like that are common and show one of the main strengths of asynchronous logic, an inherent capability to synthesize parallel behavior in

the digital logic. However, massive parallelization has its drawbacks during the STG synthesis process. Increased concurrency contributes to phenomena called state explosion, which occurs during the state graph synthesis stage. The state explosion causes a significant increase in the size of the computational problem of solving CSC and logic synthesis [16]. The state explosion is one of the major open problems in the theory of asynchronous logic design for the State-based synthesis method.

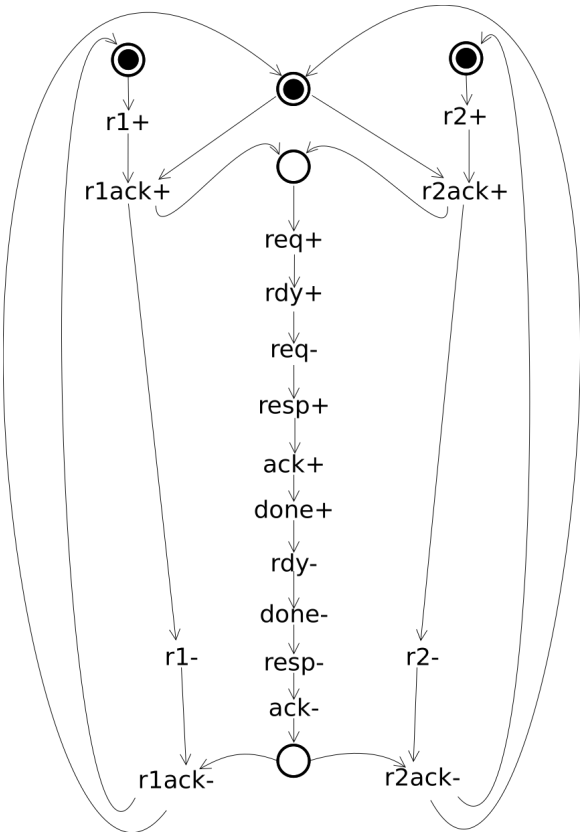


Fig. 16. Controller: the expected generated STG output.

Another critical concept present in the STG diagram is the two gated-confusion blocks. The first occurrence is the controller reaction on incoming $r1+$ or $r2+$ signals. The controller branches to either $r1ack+$ or $r2ack+$. The $r1+$ and $r2+$ are external signals that can occur at any time. Moreover, it is possible that both request signals could

arrive within a close enough timespan that the controller circuit will not have enough time to settle after the first signal transition resulting in an unstable state. In undecided cases like this, the circuit needs support from an additional external arbiter unit.

An alternative case exists in the second confusion block, which models the controller finishing a transaction handshake with one of the clients. The guard place receives token after occurrence $ack-$. The confusion is resolved by signal $r1-$ or $r2-$ depends on which client currently participates in the exchange. The confusion resolves in either $r1ack-$ or $r2ack-$. The second confusion case guarantees that only one client will respond with the handshake closing signal $r1-$ or $r2-$ because only one client gets accepted by the controller for the duration of the transaction. As such, there is no need for additional arbitration in the second confusion block.

When modeling the circuits in STG, it is important to closely track the proper positioning of the transitions in the graph. For example, considering the STG fragment in Fig. 17. The circuit can experience faulty behavior due to a hazard related to the concurrency between both clients. From the perspective of the controller described by the STG fragment in Fig. 17 the $r1$ and $r2$ are input signals each coming from a client. The STG places a shared token, a confusion block between the two inputs expecting only one occurring at a time. However, the environment might not be aware of this particular module's inner workings, and the configuration might lead to controller malfunction if both clients send colliding requests. Specifically, colliding means that the second input would appear while the previous one did not clear yet. The order of appearance of the signals will lead to an unexpected state of the controller and likely an error. Creating robust designs is important, and in this case, setting the input signals as targets for gated confusion is incorrect. The CSP model shows such errors more clearly in the syntax and makes it easier to catch by the pre-processor.

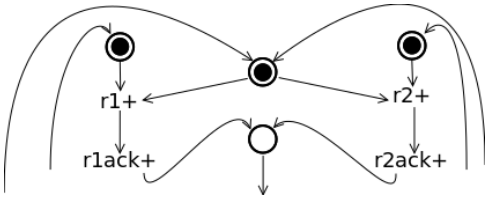


Fig. 17. Controller: an unsafe STG construct.

4.3.1 Model representation in CSP

The CSP source code splits into two major sections. The interface section specifies metadata used within the code, and the code section defining the signal flow. The header section contains information that complements the CSP notation allowing for a full description of the circuit. The following Listing 11 shows the proposed CSP for the controller model and its translation results in the STG graph shown in Fig. 16. A controller modeling language is based on the CSP language with some modifications and additions to the syntax, allowing automated synthesis.

The first element is the *module* keyword indicating the type of the model description. In the current version, the *module* occurs in two forms, the *controller* that indicates translatable implementation to STG and the *behav* form that serves as an informative overview not suitable for automated parsing. The next two fields in the header are the definitions of inputs and outputs of the circuit. The elements listed as inputs and outputs are necessary for the translator to recognize their tokens in the code. If an input or output is not listed, then the translation fails with an error stating an unrecognized element detected. The input and output list is global and shared by all the fragments defined within the code section.

```
1  module controller;
2
3  inputs r1 r2 rdy ack;
4  outputs r1ack r2ack req resp done;
5
6  main ctrl;
7
8  explicit_place P0;
9  explicit_place P1;
10
11 loop_place [r1ack-]->[r1+];
12 loop_place [r2ack-]->[r2+];
13 loop_place [r1ack- r2ack-]->[r1ack+ r2ack+];
14
15 marking [r1ack-]->[r1+];
16 marking [r2ack-]->[r2+];
17 marking [r1ack- r2ack-]->[r1ack+ r2ack+];
18
19 endinterface
20
21 ctrl: *[
22     [(r1+)->(r1ack+ => P0) | (r2+)->(r2ack+ => P1)];
23     req+;
24     [rdy+];
25     req-;
26     resp+;
27     [ack+];
28     done+;
29     [rdy-];
30     done-;
31     resp-;
32     [ack-];
33     [(P0; r1-)->r1ack- [] (P1; r2-)->r2ack-];
34 ]
```

Listing 11. Controller: CSP implementation.

Another set of elements in the header section are the *loop_place* and *marking*. The code section can contain a loop statement causing the model to execute the defined workflow continuously. However, the pure CSP language lacks the syntax to describe the loopback transitions for the synthesis of STG connections. The *loop_place* specifies a set of Petri-net places that connect transitions from the beginning and the end of the loop. The single loop place is specified by a pair of sets containing the input and output transitions. When constructing the loop place, the translator must be able to find all the participating transitions. Otherwise, the loop place insertion fails, which indicates an error in the CSP implementation.

The second field token named *marking* describes a set of places that contain the initial marking. The target places are identified the same as $loop_place$ by two sets, one for pre-set and the other for post-set. First, the translator looks for places that match exactly the description. If the first search fails, then the translator attempts to create the matching place. At this stage, all places with more than one element in the post-set or pre-set or both are expected to exist in the graph and do not need to be created by the marking process. If a search for a place with multiple inputs and outputs fails, the marking process fails immediately. An exception to the rule is a place with exactly one element in pre-set and one in the post-set. Single input and output places are not explicitly present within the STG representation. In the case of missing single input single output place, the translator attempts to find a matching connection between two transitions. When the connection is found, the translator inserts a marked place; otherwise, the marking process fails if no such connection exists.

The remaining three keywords in the header segment are the *main*, $fragment$, and $explicit\_place$. The *main* keyword indicates the entry point fragment from which the translation starts. The $fragment$ shown in Fig. 12 identifies a label of a CSP process fragment. The final $explicit\_place$ is a supporting CSP language extension. Usually, the place elements in the STG are inferred based on the flow of the CSP code. However, the designer can create a custom place that connects to arbitrary transitions. The explicit_place keyword defines all the names of such custom places. An explicit place construct allows for a custom definition of a named Petri-net place without affecting the rest of the translation process. Finally, the header section finishes with the $endinterface$ keyword.

After the header, the remaining content in the file is the code section. The code section is organized into labels, and each label indicates a code fragment. A fragment starts with its label name, after which the CSP code begins. In terms of CSP terminology,

a single source file describes a single process, and a single process represents a single controller. Additionally, the CSP process can be partitioned into multiple fragments to improve code readability and introduce modularization. The code in Listing 12 shows an example of using the modularized approach. The *data*!*curr_count* label implements 4-phase data send from the *curr_count* variable through the data channel. The code fragments can be referenced by the other fragments, including the main entry.

```
module behav;

inputs r1 r2 ack;
outputs r1ack r2ack resp;

main ticket;

fragment data!curr_count;

marking r1+;
marking r2+;

endinterface

ticket: *[
    [(r1+)->r1ack+ | (r2+)->r2ack+];
    data!curr_count;
    [(r1-)->r1ack- | (r2-)->r2ack-];
]

data!curr_count:
    data:=curr_count;
    resp+;
    [ack+];
    resp-;
    [ack-];
```

Listing 12. Use of model fragments.

A single process can contain multiple CSP constructs that implement the algorithm workflow. The used CSP notation represents a standard semantics used to model asynchronous logic [17], [19], [20] with custom additions. The two CSP extensions are the detach operator and the explicit place syntax. The detach ":" syntax function is to split the sequence of CSP transitions into two disconnected STG fragments. By default, each consecutive signal transition in CSP is tied with its predecessor, and the parser infers,

based on the operator, the proper connection. However, when the detach (":") operator appears, it causes a hard split, after which any further tokens end up in a separated STG. No inferred connections are made. Any transition between the separated STGs' must be specified by using an explicit place.

The second language extension is the explicit place construct used to manually specify a place within the STG graph connecting two transitions manually. The explicit place $P0$ and $P1$ shown in Listing 11 connect the $r1ack-$ with $r1-$ and $r2ack-$ with $r2-$. The explicit place creates a connection between two transitions outside of the regular flow. The explicit place construct serves, among others, an important role when defining gated confusion blocks. By default, the confusion block at its end implies a single place that merges the paths and connects them to the first occurring transition after the block. Sometimes, however, it is desirable to make additional paths leading to other transitions that are not automatically inferred. Use of the explicit place results in the $r1ack-$ and $r2ack-$ to be connected to corresponding transitions $r1-$ and $r2-$ through manual specification. Then $r1ack-$ and $r2ack-$ connection to $req+$ through a single implicit place is inferred automatically by the parser. Every explicit place has its unique name through which it is referenced from anywhere within the controller. If more than one explicit place exists, both must have different names.

A connection to an explicit place can be instantiated in two ways, as the in-flow element or using the "=>" operator. The in-flow instantiation occurs when the explicit place appears in the code as it would be a transition in a sequential flow. For example, an explicit place can be defined as "tran1+; P0; tran2+" in which $P0$ is an explicit place between two transitions. The second option is the use of the "=>" operator in conjunction with the explicit place. The operator will attach the place element to a transition either as an element of its pre-set or post-set. If the explicit place occurs on the operator's left-hand side, it is assigned to the pre-set of the next transition. Similarly, if the place occurs at the

right-hand side, it is assigned to the preceding transition's post-set. Assignment by the "=>" operator creates a connection between the place and the transition but does not add the explicit place into the sequence of transitions, unlike the in-flow addition.

The detach operator and explicit place are essential in specifying a controller construct in which two sets of transitions are loosely coupled and cannot be described using the default CSP constructs. The Fig. 18 along with the code in Listing 13 shows an example scenario. Two STG fragments exist; the first is a loop of *req* and *ack* providing a 4-phase handshake, and the second is a block for transactions. The controller is capable of servicing only one of the segments at a time; therefore, a guard place is present; it is the place that leads either to $ack+$ or $choice_r eq+$. In regular flow, it would be difficult to specify such a connection using the regular confusion block. As such, the detach operator is used first to separate two STG fragments, then using the "=>" operator, which places the two input two output explicit place $P0$ connecting the two fragments.

In addition to the base syntax and syntax extensions, the CSP to STG processing algorithm implements a set of features that understanding is critical for constructing working controller circuits. The first feature is the parenthesis rule. When the parser processes the CSP transitions, it traverses the tokens one by one, constructing a graph fragment. For example a sequence "tran1+; tran2+; tran3+; tran4+;" results in an immediate creation of four-transition STG fragment. However, when some transitions exist within parenthesis such as "(tran1+; tran2+;) tran3+; tran4+;" the parser interprets it first as a single fragment. The CSP code segment inside the parenthesis is placed within a newly created fragment element and deferred for processing in the next iteration. What comes from the first iteration is an STG fragment with a nested fragment element "(tran1+; tran2+;)" and two transitions "tran3+; tran4+;". More importantly, any specific operations related to the segment are applied immediately regardless of the content within

the parenthesis. This rule is important because it influences the parser behavior for explicit places and confusion blocks.

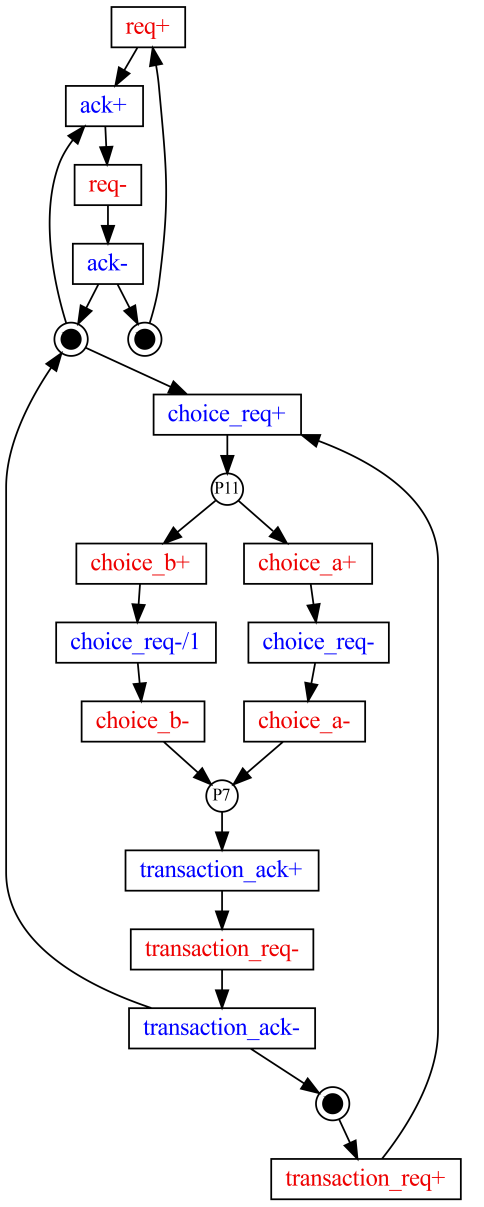


Fig. 18. Example STG model, concurrent events flow.

```
module controller;

inputs req transaction_req choice_b choice_a send_ack;
outputs ack transaction_ack choice_req;

explicit_place P0;
loop_place [ack-]->[req+];
loop_place [transaction_ack-]->[transaction_req+];

marking [ack-]->[req+];
marking [transaction_ack-]->[transaction_req+];
marking [ack- transaction_ack-]->[ack+ choice_req+];

main stg_main;
fragment frag_task_a;
fragment frag_task_b;
endinterface

stg_main:
    frag_task_a : frag_task_b

frag_task_a:
    *[
        [ req+      ];
      P0=>ack+      ;
        [ req-      ];
          ack-=>P0  ;
    ]

frag_task_b:
    *[
        [ transaction_req+ ];
      P0=>choice_req+      ;
        [( [ choice_a+ ]; choice_req-; [ choice_a- ]; ) |
         ( [ choice_b+ ]; choice_req-; [ choice_b- ]; )
        ];
          transaction_ack+      ;
        [ transaction_req-     ];
          transaction_ack-=>P0  ;
    ]
```

Listing 13. Example CSP model concurrent events flow.

The use of explicit place as an in-flow element in conjunction with the parenthesis rule results in a detached segment. Unlike the "=>" operator, the in-flow place is injected into the sub-graphs transition chain. After processing each code fragment, the parser returns a set of heads and tails of the created STG fragment for injection into the existing main graph. However, if a head or tail happens to be an explicit place, the parser interprets it as a case in which the designer opted for manually specifying the path,

causing the parser not to infer connection by itself. As a result, the inferred connection to the main graph is not made for a sequence with an explicit place at the beginning or end; for example, for a sequence "(P0; tran2+; tran3+;)" the parser will return a tail pointing to $tran3+$, but the head will be NULL. A similar case applies for an in-flow explicit place at the end of the transition sequence, in which case the tail will then be NULL. The same scenario holds true for constructs with more than one path, such as the parallel and confusion block. The detached segment concept allows for the specification of more complex event sequences, for example, a parallel or a confusion block in which one or more paths do not merge to the same default point with the rest. Instead, the detached path further flow is specified using an explicit place connection.

The parenthesis rule directly determines how the confusion block processes its content. The different behaviors are necessary to synthesize constructs such as the gated confusion shown in Fig. 17 and regular confusion as shown in Fig. 18 which begins from place $P11$. The difference lies in how the transitions are specified within each segment. The base rule of operation for the confusion segment syntax is that the operator receives sequences of transitions in blocks separated by the "|" or "[]" token. The operator returns a collection of heads and tails. For tails, it creates an implicit place that connects all the outgoing paths unless there is a detached segment with an explicit place at the end. For heads, the confusion block creates one implicit place, which becomes a part of the pre-set for every finishing transition in each block unless that last token is an explicit place. Moreover, the list of heads contains references to every beginning transition in each block unless it is an explicit place, or there is only a single transition within the specific block. An example of valid constructs is shown in the Listing 14.

```
[ t1; t2; t3 | (P0; t7)-> t8 | fragment_name | (t9; t10) ];
```

Listing 14. Confusion block variations.

Beginning from the leftmost block. The first block will result in t1 in heads, the initial and final implicit places pointing to t3. In the second segment, an implicit fragment will be created containing the content of the parenthesis. At first, the heads point to the implicit fragment and the initial and final implicit place to t8. When the implicit fragment gets parsed, it will result in a detached block, effectively clearing any implicit connection leading to P0. In the third case, a single fragment is treated as a single transition; thus, the heads will only point to the implicit initial place, then the implicit initial and final places point to the fragment. The fourth case is equivalent to the third by parenthesis rule, which will create an implicit fragment during the initial encounter with the parenthesis.

The third and fourth cases are used to specify regular confusion, such as in Fig. 18, in which the initial place is the $P11$ and final $P7$. The second case is useful in modeling constructs such as the bottom confusion block from Fig. 16 where confusion occurs between $r1ack-$ and $r2ack-$ but both are gated by $r1-$ and $r2-$ respectively. Finally, the first case exists to work in conjunction with the loop construct. If a confusion block appears right away after the loop opening "*[" as in Listing 11, the confusion block returns the $r1+$, $r2+$, and the implicit initial place in heads which are tied to the loop node. Later, when the parser closes the loop, it searches for specific transitions indicated by the loop_place and with connections to the loop node specifically instead of in the graph's global node space, preventing unwanted name collision.

The presented CSP approach allows describing asynchronous control circuits in a form that is easier to maintain for the designer and also translatable to the STG representation. The concise and modularized approach delivers better capabilities in terms of analyzing and debugging the implementation.

4.3.2 Translation from CSP to STG

The translation process starts with the pre-processing of the input CSP code. The first stage splits the header and code sections. The header segments starts from the *module* declaration and ends with the *endinterface* keyword (Listing 11). The entire header segment is processed and organized into the *Meta* classes shown in Listing 15.

```
class MetaModelType(Enum):
    CONTROLLER = 1
    BEHAV = 2
    UNDEFINED = 0

class MetaPlace():
    def __init__(self):
        self.input = []
        self.output = []

class Meta():
    def __init__(self):
        self.model_type = MetaModelType.UNDEFINED
        self.inputs = []
        self.outputs = []
        self.main = ''
        self.fragments = []
        self.loop_place = []
        self.explicit_place = []
        self.markings = []
```

Listing 15. Meta classes.

Next, the parser moves to the process definition. The segment is divided by the fragment labels such that sections beginning with labels like "main:" or "data!curr_count" become separate entities. After initial pre-processing the CSP code is divided into segments represented by a *CodeSegment* class shown in Listing 16. The class contains a label that identifies the given fragment and its code in the form of an array of tokens.

```
class CodeFragment():
    def __init__(self, l, s):
        self.label = l
        self.src = s
```

Listing 16. The code segment class.

Every code fragment is broken into the array of tokens during the pre-processing stage. For example the fragment from Listing 11 between lines 25 and 28 would become an array composed of "req-, ;, resp+, [, ack+,], ;, done+, ;" and assigned to the *src* field of the *CodeFragment* class. The pre-processing stage finishes when all the fragments are processed and placed in the collection of *CodeFragment* classes along with the meta-data description.

The primary goal of the translation from CSP to STG representation is to produce the STG graph used to derive the input-output type asynchronous sequential controller circuit. The STG representation is a uni-directed bipartite graph with nodes representing either the transitions of the circuit or Petri-net places. The translation algorithm shown in Algorithm 1 consists of four main steps:

1. Derivation of the initial graph from the CSP representation.
2. Removal of redundant place elements.
3. Insertion of places corresponding to the CSP loop(Loop closing).
4. Insertion of markings to indicate the initial state of the model.

Algorithm 1 CSP to STG translation main algorithm.

```
nodes ← Fragment(main_label)
while exists any Fragment do
    for all e in nodes: do
        if e is Fragment then
            ProcessFragment(e)
RemoveRedundantPlaces(nodes)
ProcessLoopPlaces(nodes)
RemoveRedundantPlaces(nodes)
ApplyMarkings(nodes)
```

The algorithm defines four types of nodes: *Place* node, *Transition* node, *Loop* node and *Fragment* node. The *Place* and *Transition* nodes are direct representations of STG graph elements. The *Loop* node represents the loop construct and is processed in the third

stage of the algorithm. The final *Fragment* node is a meta-node that exists in two states. It contains either a CSP code fragment or a handle to a specific CSP code fragment.

The STG graph's derivation is an iterative process starting from the initial node that contains a handle to the primary process. All the graph elements are organized in an array that is traversed until all of the places of the type *Fragment* are processed into either *Loop*, *Place*, or *Transition*. Any time the *Fragment* element is encountered by the parser, one of two scenarios happens. If the *Fragment* contains a label pointing to another code segment, then the actual code from that segment is copied into the element then the label is cleared. The second scenario occurs when the encountered *Fragment* element contains the actual code of the fragment. Then the code is processed, creating a new set of nodes that replaces the current *Fragment* element. Processing code segment occurs by traversing the tokens array from the beginning to the end. The resulting code segment can occur in three variations:

- Sequential segment
- Parallel segment
- Confusion segment

Each segment represents a different possible outcome of a code fragment parsing. The sequential segment represents the most straightforward construct in which transitions happen one after another. The detached sequential segment is a variation in which the entering or exiting node does not implicitly connect to any path within the graph. The detached segment exists when an explicit place occurs at the beginning or end of the transition chain. The explicit place defines a custom connection and is not inferred automatically by the algorithm. The parallel segment and confusion segment are two special cases; both allow for the synthesis of branching execution paths, which are then further interpreted according to each segment's rules.

The classification into segments serves the purpose of defining the final graph connection points after processing of a given code block finishes. When the parsing is done, the created STG fragment is re-attached into the main graph replacing the initial Fragment node. The Segments specification defines the heads and tail nodes collections for the generated graph segment. During the re-attachment, all the nodes leading into the processed fragment are connected to the head nodes in the generated graph segment. Similarly, all the tail nodes from the fragment are added as the destination nodes from the outgoing nodes in the generated fragment.

During the CSP code processing, the parser traverses through the array of tokens generating corresponding graph elements. The generating engine behavior shown in Algorithm 2 is dictated by the currently processed token. There are eight primary classification categories of a token the loop element, grouping token, confusion, sequential, explicit place assignment, parallel, detach, and unknown type of token. When the parser encounters a loop or grouping token, it creates a new Fragment element. Then the parser traverses the tokens list forward, looking for the corresponding closing parenthesis. Nesting is allowed, and the parser counts occurrences of opening and closing grouping tokens. The process only finishes when the parser encounters an equal number of closing brackets to opening brackets. After extracting the segment, the Fragment element src field is populated with the extracted code, and its processing is deferred to the next iteration. In the case of the loop token, an additional Loop node appears at the beginning to allow for loop synthesis in later stages.

The second group of tokens is the parallel and the confusion block. When the parallel token or one of the confusion tokens is encountered, a change from the Sequential segment to Parallel or Confusion segment type occurs. In all cases of this type, an already populated sequential path exists. The existing path becomes a first parallel path. The parser finishes the first parallel path and starts a new one to which any upcoming tokens

Algorithm 2 Processing the code fragment.

```
Initially use segment type Sequential
for all t in tokens: do
    if t is *[ then
        1.Insert loop node
        2.Insert Fragment with loop internals
    else if t is [ or ( then
        1.Extract all tokens up to ] or )
        2.Insert fragment with nested segment code
    else if t is | or || then
        1.Convert from Sequential to Confusion segment
        2.Set existing sequential as first confusion path and begin next confusion path
    else if t is ; or -> then
        Insert single STG place
    else if t is => then
        Do nothing, behavior depends on existence of explicit place in syntax
    else if t is , then
        1.Convert from Sequential to Parallel segment type
        2.Set existing sequential as first parallel path and begin next parallel path
    else if t is : then
        1.Create a detached source fragment without any parent child connections
        2.Absorb the remaining of tokens for the new fragment
    else
        if t is explicit place then
            Create place object for this name if one does not exist
            if t_{i-1} is => then
                Add created place as a child of preceding transition
                Do not register in the transition tree
            if t_{i+1} is => then
                1.Fetch transition at t_{i+2}
                2.Insert Transition node
                3.Add the transition as child of the place
            else if t_{i-1} is not => then
                Add the place as regular element into transition chain
        else if t is transition then
            Insert Transition node
        else if t is fragment label then
            Insert Fragment node
Prepare output structure and return head and tail nodes
```

are assigned. If more than two parallel paths exist, the parser keeps adding the next paths on every occurrence of the parallel or confusion token. The parallel or confusion blocks are placed in the grouping parenthesis to allow the parser to process the beginning and end of the parallel paths upfront.

The confusion block is a specialized construct that models a one-of-many choice of the transition path. One form of confusion is a regular split into many execution paths in which the only one is expected to be taken. In the regular form, as seen in Fig. 18, the confusion block starts from place $P11$, and each of the upcoming transitions is an input signal. The external environment is responsible for making the next step and is expected to emit only one signal out of the two. The confusion block also exists in the gated form, as seen in Fig. 16. In the gated version, the execution branches begin with an output signal controlled by the modeled device. However, the firing of such a branch transition is gated first by the shared token, ensuring one-of-many execution second by a gating transition that is usually an input coming from the external environment. For example, the $r1ack+$ to fire requires the central token to be present and the $r1+$ to trigger beforehand. In the gated confusion setup, it is not uncommon to connect inputs of confusion triggers and their outputs to explicit places. The explicit place construct allows for a definition of an execution path outside of the main flow inferred by the translator. Looking back at Fig. 16 the two explicit places are used to connect $r1ack+$ to $r1+$ and $r1-$.

The third group is the set of tokens that are extensions to the standard CSP language. The new items are the arrow token for explicit place assignment and the detach token. The explicit place assignment token is used in conjunction with the explicit place to tie the place with a transition without inserting the place element into the transition flow. In the explicit place assignment using the arrow operator, a simple reference is created from a transition to the place or from place to transition depending on which side of the operator both tokens appear. When traversing the CSP implementation, the explicit place

assignment token's occurrence does not trigger any immediate action from the parser. However, the position of the assignment token matters when the parser encounters an explicit place. The algorithm determines based on the order of elements on how to perform the reference assignment. Depending on whether the explicit place assignment token precedes or tails the explicit place, the parser assigns the place as a part of the post-set or pre-set of some transition.

The second CSP extension is the detach token. When the parser encounters the detach token, it immediately stops adding new elements to the current STG segment. Everything that appears after the detach token is combined into a new Fragment and placed in the main array without any connections to the previous portion of the graph. Then the existing segment is finalized and returned for placement. The detach token element effectively finishes processing any elements that exist up to this point and spawns an entirely new STG fragment that is not implicitly connected to the current one.

The last two classes are the sequential token and the unknown token class. The sequential token causes addition of an STG place to the graph that indicates a single sequential path of execution. The unknown token indicates that the current element is not within the previous seven categories and triggers an additional check. When the header metadata is processed, the translator obtains the list of all signals, code fragments, and explicit places. If the unknown token is encountered, the translator checks against these lists. In the case of a transition match, the transition element is inserted. Alternatively, the fragment label triggers the addition of a Fragment element.

For the explicit place, a series of additional checks occur. First, the algorithm looks for the explicit place assignment token next to the explicit place token position. If the assignment token exists at the position $i-1$ before the place on the left side, then the current place object is assigned as a child of a transition at the $i-2$ position. If the assignment token exists at the $i+1$ position, then the transition at $i+2$ becomes a child of

the explicit place. Finally, if there is no assignment token, this means that it is an in-flow explicit place. In the in-flow case, the algorithm adds the place sequentially to the current transition chain segment. Additionally, if the in-flow place happens to be the first or last element in the chain, it will later form a detached segment.

After processing the entire array, the translator using the Segment definition prepares a graph segment, then exports two collections heads and tails that are respectively entry and exit points from the generated segment. A special case exists when an explicit place occurs as the first or the last element in the current chain. In this case, the reference to the place is added to neither heads nor tails, and the parser expects that the designer specifies all required connections to the explicit place manually. The segment is then attached in place of its source fragment element; the returned heads become post-set for any parent nodes, and tails would become pre-set for every child node of the replaced fragment. After attachment, the translator proceeds further with traversing the element array. The cycle repeats until there are no Fragment type nodes present, indicating all source code got translated to the nodal representation, and the next stage of parsing begins.

The next stage is the removal of redundant Place nodes from the graph. During its operation, the translator deletes out Place elements that are unnecessary or, in some cases, lead to an incorrect description of STG. For example, one Place pointing to another Place. The order in which redundant places are identified and removed from the graph matters and must not be altered. The redundant nodes must match the description exactly, cannot have markings, and are removed in the following order:

1 Places with exactly one input Place and multiple output Places
2 Places with exactly one input Transition and exactly one output Transition
3 Places with exactly one input Place and exactly one output Place or exactly one output Transition
4 Places with one input Transition and one output Place

5 Places with one or more parents that have exactly one child that is a place and the child have one or more parents

6 Places with one or more with one or more parents that are transitions, one or more parents that are places and exactly one child that is a transition

7 Places with one or more parents that have exactly one child that is a place and has exactly one parent

As an exception, the redundant place removal process does not affect places with empty post-set or empty pre-set. A place with an empty pre-set or post-set indicates a loop element that belongs to either beginning or end of the loop and is processed in the next step. After the removal of the redundant nodes, the tree is ready to connect any existing loop places.

The third step creates connections for any STG loop existing. The Listing 11 shows three existing loop places in the circuit. The loop connectivity algorithm looks for an existing Loop element in the graph. During the CSP code parsing, when a loop token is encountered, a Loop element is instantiated along with the Fragment element. The Loop element indicates the beginning of the loop, and the Fragment contains all the CSP code that consists of the loop body. Eventually, the Fragment element gets processed, and all its input places, unless they belong to the detached segment, are connected as children of the Loop element. During the loop closing step, the parser uses the Loop node children array and any non-explicit places with an empty post-set to match the loop closing combinations specified in the header. When such a combination is found, a corresponding loop closing Place is inserted into the STG. When all the loops are processed, the algorithm moves to the next step that adds markings to places.

Adding the markings occurs in two phases. The first phase searches for places that have matched the marking description and have multiple inputs or outputs. Such a place must exist in the graph already. If the place is not found, it means an error in translation.

The second stage searches for places that are candidates for markings and have a single input and single output transition. A one input one output place might not exist because STG does not require an explicitly defined place element between two transitions. If such a place is missing, the algorithm attempts to find a matching transition to transition connection. After the connection is found, the algorithm inserts a marked place between the two transition elements. If the connection is not present, it indicates an error.

As the last step, the removal of redundant places executes again. The loop closing algorithm can potentially leave some redundant place markers, which should be removed. With the final step finished, the STG model is completed and used to print out into the Petrify format for synthesis. The printout step uses the metadata section to provide lists of controller inputs, outputs, and markings. Then it traverses through the graph elements for each extracting its label and target transitions.

The parser looks for repeating occurrences of transitions with the same signature but in different locations within the graph. When a duplicate exists, its name is converted to format, allowing Petrify to distinguish between the transitions which would otherwise treat as one. For example, if a repeated definition of $r1+$ exists, then the second occurrence is renamed to $r1+/1$ and so on if more repetitions are present. The final output is then saved for further processing in Petrify. The Listing 17 shows an example of the STG graph model in the format accepted by the Petrify tool.

```
 1: .model ctrl
 2: .inputs r1 r2 rdy ack
 3: .outputs r1ack r2ack req resp done
 4: .graph
 5: P2 req+
 6: req+ rdy+
 7: P10 r1ack+ r2ack+
 8: rdy+ req- resp+
 9: req- rdy-
10: rdy- done- resp-
11: done- ack-
12: resp- ack-
13: ack- P21
14: r1ack- P10 P4
15: r2ack- P10 P3
16: P21 r1ack- r2ack-
17: r1+ r1ack+
18: r1ack+ P2 r1-
19: r2+ r2ack+
20: r2ack+ P2 r2-
21: resp+ ack+
22: done+ rdy-
23: r1- r1ack-
24: r2- r2ack-
25: ack+ done+
26: P3 r2+
27: P4 r1+
28: .marking { P10 P3 P4 }
29: .end
```

Listing 17. Final STG form.

4.3.3 Synthesis from the STG model using Petrify

The Petrify tool is a framework designed to perform the synthesis of an asynchronous controller circuit defined as the STG representation [15], [16]. The tool allows for the synthesis of the STG model and produces the results in the form of next-state Boolean equations with additional information of any required use of the C-element. An example of the Petrify output is shown in the Listing 18. The excerpt from the Petrify output log file shows the set and reset functions' equations and the corresponding C-element configuration. In this case, the C-element requires the input from the reset function to be inverted and an additional inverter at the C-element output.

```
SET(r1ack')   = r1ack r1' csc0 ack' rdy'
RESET(r1ack') = r2ack' r1 csc0'
[r1ack] = r1ack'          (output inverter)
  > triggers(SET):      (r1-,ack-) -> r1ack-
  > triggers(RESET):    ([r1+,r1+/1],[csc0-,csc0-/1]) -> r1ack+
  > 8 transistors (3 n, 5 p) + 4 inverters
  > Estimated delay: rising = 35.96, falling = 54.75
```

Listing 18. Petrify output.

Petrify performs the synthesis from the STG description of an input-output asynchronous sequential circuit [9]. The initial STG model is translated into the State Graph (SG). Then the software performs optimizations on the state assignment and solves the Complete State Coding problem [4]. Then the tool derives the next-state equations for the selected representation. For the final synthesized form, the user can select the complex gate representation or one of the solutions based on the C-element [4], [5]. The C-element based solutions generate the generalized C-element or the standard C-element with monotonous cover constraint.

Although the Petrify tool provides a complete set allowing for the synthesis of the asynchronous controller circuit, the tool itself has certain limitations arising from the computational complexity of the synthesis method [9]. Circuit properties like parallel transition flow or technology-aware logic decomposition add to the overall complexity of the intermediate SG model. In the case of parallel behavior, the SG must cover all the possible permutations of states that can occur during the parallel changes of the circuit's inputs and outputs. The result is a state explosion in which a seemingly small parallel optimization in the STG model can unfold into hundreds of extra states. Also, the technology-aware logic decomposition can introduce additional internal signals to the design in an attempt to generate hazard-free gate-level representation, further expanding the SG model size. In addition to the SG size, there comes the Complete State Coding (CSC) problem. The tool heuristically attempts to find a set of internal states to remove all

the inconsistencies in SG description in the case when two states with the same encoding represent two different SG elements. The limitations prevent Petrify from producing large models within a reasonable time.

4.4 Part 2: Stage 2 Model for logic synthesis

The Stage 2 model introduces the next step in the methodology that leads from behavioral representation to a codebase ready for model synthesis that ends as the final gate-level representation. The Fig. 19 shows remaining steps of the process. The stage 2 model builds on top of the previous stage 1 and reuses some of the components while redefining others. As the primary purpose of stage 2 is to deliver a synthesizable model, the stage becomes a technology-dependent and uses the ASIC component library elements directly within the model.

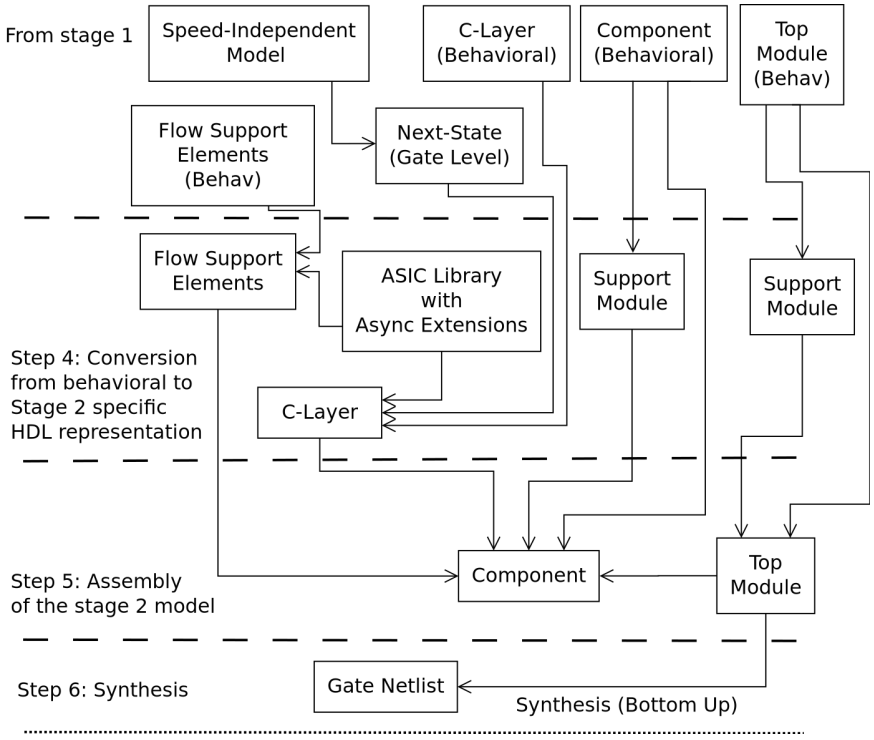


Fig. 19. Methodology steps leading to stage 2 model.

The design from stage 1 cannot be directly fed to the synthesizer because the optimization would automatically reduce some of the critical constructs like the delay lines producing an incorrect implementation. Also, the first stage model contains behavioral elements like the C-element or mutex that need to be manually mapped to corresponding cells in the ASIC technology library. Finally, the correct technology mapping of the controller's next-state equations requires a specialized approach to produce hazard-free implementation. In general, stage 2 reuses some of the components from stage 1 for direct synthesis, while others must be reimplemented. The directly reused elements are the internal combinational functional modules from the flow support elements. The collection of modules that must be redefined using the primitives from the ASIC technology library are the C-Layer, the handshake layer of FSE, and the Component and the Top module. Also, components like arbiters or asynchronous registers must be rewritten in structural form using the ASIC primitives. The presented methodology introduces a template for stage 2 specific modules showing the reference implementation approach. The stage 2 components might vary from the template, especially when a different set of ASIC primitives or a different STG synthesis tool is used. The shown examples of the controller's C-layer are based and optimized for the output generated by the Petrify tool.

For the speed-independent controller side, the behavioral C-layer from stage 1 is completely replaced by a stage 2 implementation and becomes a structural gate-level model composed entirely of primitives available in the ASIC library. The reference template for stage 2 maps the behavioral constructs from stage 1 with stage 2 primitives directly in the one-to-one relation. Shown in Listing 19 is the stage 2 implementation equivalent the the behavioral stage 1 model presented in Listing 5.

```
module ctrl (
    input   logic rst,
    input   logic in1,
    input   logic in2,
    output  logic out1,
    output  logic out2
);

    //internal signals
    logic csc0;

    //signals coming out of the comb module
    logic out1_on,
    logic out1_off,
    logic out2_sig,
    logic csc0_on,
    logic csc0_off

    ctrl_nextstate
    NS  (.in1(in1), .in2(in2), .out1(out1), .out2(out2), .csc0(csc0),
         .out1_on(out1_on), .out1_off(out1_off), .out2_sig(out2_sig),
         .csc0_on(csc0_on), .csc0_off(csc0_off)
    );

    //C-Layer connections
    //Type: inv B, inv out, reset low
    GC_IBIORL
    c0  (.O(out1), .A(out1_on), .B(out1_off), .R(rst));

    //Inverted signal, pull low
    ctrl_sig_ispl
    s1 (.rst(rst), .sig(send_sig), .o(send));

    //Type: inv B, direct out, reset high
    GC_IBDORH
    c3  (.O(csc0), .A(csc0_on), .B(csc0_off), .R(rst));

endmodule: ctrl
```

Listing 19. Stage 2 C-Layer template.

In the stage 2 version of the C-layer, all the behavioral constructs get replaced by their structural counterparts. Same as in the behavioral version from stage 1, the model contains a direct path connection and a C-element, both in multiple configurations. The proposed reference structure for the direct path connection is shown in Listing 20. The template implements two types of a direct path component, the DSPL (Direct Signal Pull Low), which is a signal buffer with a reset capability that forces the signal to state low and ISPL (Inverted Signal Pull Low), a signal inverter with reset capability also forcing

the signal to state low. A pull-high variant might also be required depending on the design when an initial state of a direct path is high instead of low. The reference implementation shown uses the Open Cell Library [41] and the FreePDK15 [42] as the base for custom cells. Regardless of the technology library used, stage 2 elements that are not part of automated synthesis must provide a valid structural gate-level design with no behavioral constructs that require translation.

```
module ctrl_sig_dspl (
    input   logic   rst,
    input   logic   sig,
    output  logic   o
);
    INV_X1 I1  ( .I(rst), .ZN(i1rst));
    AND2_X1 A1  ( .A1(sig), .A2(i1rst), .Z(o));
endmodule: ctrl_sig_dspl

module ctrl_sig_ispl (
    input   logic   rst,
    input   logic   sig,
    output  logic   o
);
    /*
    * (~rst * ~sig) -> ~(rst + sig)
    */

    NOR2_X1 NO1 ( .A1(rst), .A2(sig), .ZN(o));
endmodule: ctrl_sig_ispl
```

Listing 20. Direct signal line with pull-low.

The stage 2 model reflects all the four combinations of the C-element produced by the Petrify tool. Unlike the behavioral model, every C-element configuration in stage 2 is represented by a unique structural model rather than a parametrizable module. The mentioned four variations are:

- GC_IBDORL - Element that resets to 0 with direct output.
- GC_IBIORL - Element that resets to 0 with inverted output.
- GC_IBDORH - Element that resets to 1 with direct output.
- GC_IBIORH - Element that resets to 1 with inverted output.

The implementation of the C-element configurations can be approached either by providing a structural model composed of the ASIC library elements similar to the direct signal module or by providing custom ASIC primitives implementing an entire configuration as one indivisible cell. Both ways have their advantages and shortcomings. The structural method requires less work on the transistor level design and layout of an ASIC component. In fact, it is possible to implement all the configurations using standard combinational gates since a C-element variant exists that has its representation composed entirely of NAND gates [36]. However, the structural approach is not area efficient, requires interconnecting of multiple gates during the place and route, and can produce a circuit containing a hazard that breaks speed-independence constraints. On the other hand, the full custom ASIC approach allows for more compact and predictable elements but requires more work upfront to bring up a set of custom elements.

The second type of stage 2 specific elements are the handshake layers for the FSE modules. This currently presented variant uses delay matching as the completion detection mechanism. The structural stage 2 model replaces behavioral delays placed on signals with a specialized module providing a delay line composed of the ASIC primitives. The Listing 21 shows an equivalent to the stage 1 model, data-type handshake module. All the connections remain unchanged at the design level, but the model introduces a bundle_delay element, a parametrizable model that generates a chain of delay elements. In the case of the data-type FSE, the delay applies only to the handshake ack signal while the combinational element implementing the module's logic connects to its environment. The delay line length parameter N is adjusted during the synthesis process to exceed the combinational element's long path.

In the case of the decision-type element shown in Listing 22, the delay line is applied in conjunction with the decision signal line coming out of the combinational part. Every output line at its end is merged with a delay line through the AND gate. Merging allows

for the suppression of any hazardous switching on the decision line while the combinational logic resolves. Similar to the data type, the decision-type FSE also uses the bundle_delay module and adjusts its length during the synthesis.

```
module fse_t1 (
    //Data
    input   logic   [3:0]   d_in,
    output  logic   [3:0]   d_out,

    //Control
    input   logic           req,
    output  logic           ack
);

    import delay_def::*;

    logic   [4:0]   d_out_next;

    fse_t1_fn
    FN  (.d_in(d_in), .d_out(d_out_next), .req(req));

    assign d_out = d_out_next;

    bundle_delay #(.N(DM_FSE_T1), .S(1))
    DELAY  (.I(idx_req), .O(idx_ack));
endmodule: fse_t1
```

Listing 21. FSE synthesizable data-type handshake module.

Both types of data and the decision FSE must meet certain conditions during the design verification. In the data-type FSE, all the output lines must be analyzed during the synthesis, and the handshake signal must have the longest delay for both rising and falling edge. Testing for both edges ensures that, in the case of a rising edge, the valid data is available at the outputs by the time the module acknowledges the request. For the falling edge, the delay requirement ensures the module is in its default state, which prevents potential unwanted signals on the data path. For the decision-type element, to ensure correct behavior, every output signal path must be analyzed as well. The delay matching is correct when the long path within the output signal logic occurs only through the delay-matching path. The requirement ensures that the final AND gate is guaranteed to

activate after the internal combinational logic is resolved, and a stable result is available at the output.

```
module fse_t2 (
    //Data
    input   logic   [3:0]   d_in,
    //Control
    input   logic           sel_req,
    output  logic           sel_a,
    output  logic           sel_b
);

    import delay_def::*;

    logic   sel_a_out;
    logic   sel_b_out;
    logic   sel_req_delay;

    fse_t2_fn
    FN  (.d_in(d_in), .sel_a(sel_a_out), .sel_b(sel_b_out));

    bundle_delay #(.N(DM_FSE_T2), .S(1))
    DELAY  (.I(sel_req), .O(sel_req_delay));

    AND2_X1
    A1 (.A1(sel_a_out), .A2(sel_req_delay), .Z(sel_a));

    AND2_X1
    A2 (.A1(sel_b_out), .A2(sel_req_delay), .Z(sel_b));
endmodule: fse_t2
```

Listing 22. FSE synthesizable decision-type handshake module.

Stage two also introduces the support module that encapsulates all the signal redirecting and management code present within the Component and Top modules. Each instance of a Component and the Top module contains its version of the support module unless all the connections are direct and trivial. Trivial means a wire only and without any logic in the connection path within the scope of the component. The motivation behind introducing the support module is to provide the ability to perform automatic synthesis on the signal management logic. In some cases, the signal management logic could become difficult to implement by hand using ASIC primitives. A good example is the address line coming into the cache memory from multiple sources. All three components share a single path; thus, all three sources must be OR-ed for every bit of the address line within

the Top module. Modeling the merged memory bus connection by hand using primitives could become quite a challenge and introduce bugs. The Support module covers the task of isolating the signal redirecting management logic and allows it to be automatically synthesized. Also, the support element can contain a delay path if necessary and when the logic synthesizer allows for the exclusion of individual HDL modules from optimization [43].

4.5 Asynchronous Extension to standard set of ASIC primitives

To support the asynchronous constructs presented by the methodology, the corresponding ASIC library must be extended beyond the standard set of primitives [41]. At the very minimum, the library must provide a C-element [34] and the mutex [5], [17] element. Even though it is possible to implement the C-element using NAND gates [36], it is not a practical approach, thus making the custom C-element cell an essential addition. The Fig. 20 shows static implementation of the C-element. The circuit realizes the Boolean Equation 2 which describes two input C-element functionality.

$$C_{next} = AB + C(A + B) \tag{2}$$

It is also possible to extend the static C-element to have more inputs as seen in Fig. 21 showing a 3-input version. Every new input requires the addition of 4 transistors. Although the methodology does not use any other than a two-input C-element within the controller, the gate is also useful as a merging point to combine multiple passive handshake elements, which number might exceed two. For example, two or more passive elements feed their ack signal through the C-element. The gate's output becomes high only when all ack signals are high, showing the passive components' readiness. Similarly, the output goes back to low only when all the ack signals go low. This way, the C-element can become a simple mechanism of completion detection from multiple sources. However,

increasing fan-in creates a large gate in CMOS technology. The size of a CMOS gate is limited in practice by transistor-level effects such as charge sharing problem [3], [44].

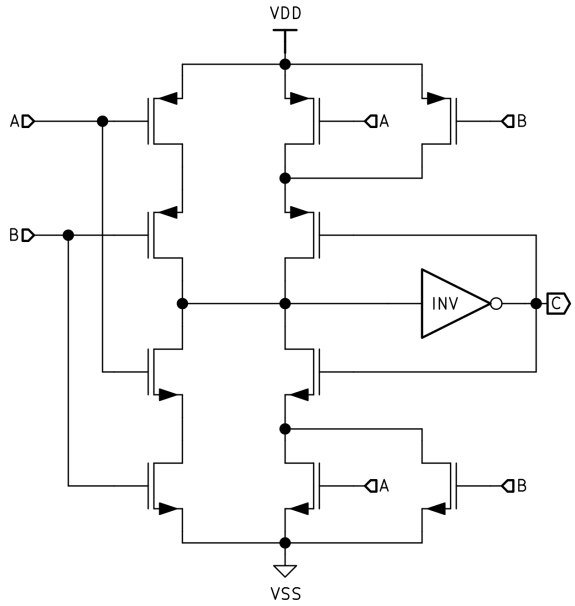


Fig. 20. Generic C-element.

The second essential ASIC primitive is the mutex element, which resolves the problem of two contending input signals going into a speed-independent controller that could appear at an unknown time. It is common within the methodology for a gated STG confusion block to exist that implements conditional execution steering the flow in a specific direction. Such flow case by default assumes that the appearance of one signal within the choice disables all others until the STG reaches again the same choice place or another place that allows for the occurrence of the remaining signals. The speed independence assumes the unknown time of occurrence but does specify the order of events. If the mentioned input signals come from external modules that do not know others' states, it is possible to put the receiving controller in an invalid state, causing it to malfunction. The mutex element solves the problem by allowing only one signal to come through and preserves its state until the selected line goes back to low, releasing the lock.

Also, the mutex element is an essential building block for a variety of arbiter circuits [17]. The mutex element and the derived arbiter circuits provide irreplaceable functionality that allows for the synchronization of independent modules making the mutex element an essential circuit among the asynchronous extensions of the ASIC library.

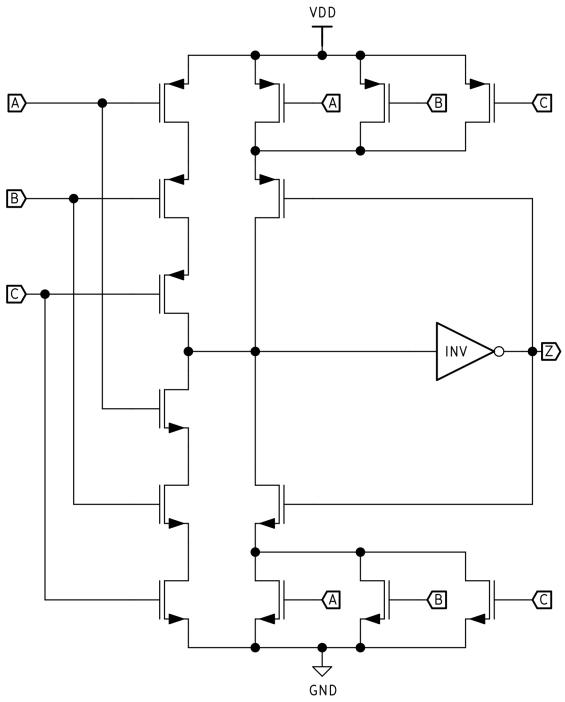


Fig. 21. Generic C-element with 3 inputs.

Also, the asynchronous arbitration operation in the asynchronous domain becomes an analog problem with the expected occurrence of meta-stability that is filtered out by specific circuit configuration. Therefore, it is impossible to construct the mutex element from the standard set of Boolean logic gates, and the mutex primitive element must be constructed at the transistor level. The Fig. 22 shows an implementation of the mutex element, which consists of two stages, the selector and the meta-stability filter. The NAND gates' cross-connected configuration provides the selector circuit that ensures that only a single line is eventually chosen, becoming a logic state low. If both request signals appear

close enough, the selector might become meta-stable, but the filtering segment inhibits the transitional fluctuations at the output. The filtering configuration is essentially an inverter gate with VDD connected to an output of another gate. The selected line feeds the input signal to its corresponding filtering configuration while the other selector output provides power. The circuit only emits a signal high when both outputs are in a settled state.

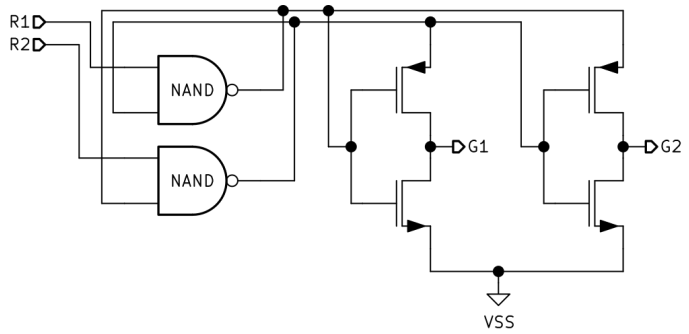


Fig. 22. Asynchronous MUTEX.

The mutex element can be extended to handle more than two inputs. The Fig. 23 shows a three input mutex element. Extending the mutex element occurs in two steps. First, the selector stage adds one more NAND gate. All the gates are cross-connected; therefore, every gate gains one more input. Then the filtering block gets one more filtering element for the additional output. The current delivery is resolved by assigning VDD to one gate other than the gate providing the signal. Then an additional PMOS with an inverter is added that serves as the second current delivery gate. Extending the mutex allows for more inputs and avoids the necessity to cascade connect multiple elements. However, the extending requires significant growth in transistor count and higher fan-in NAND gates.

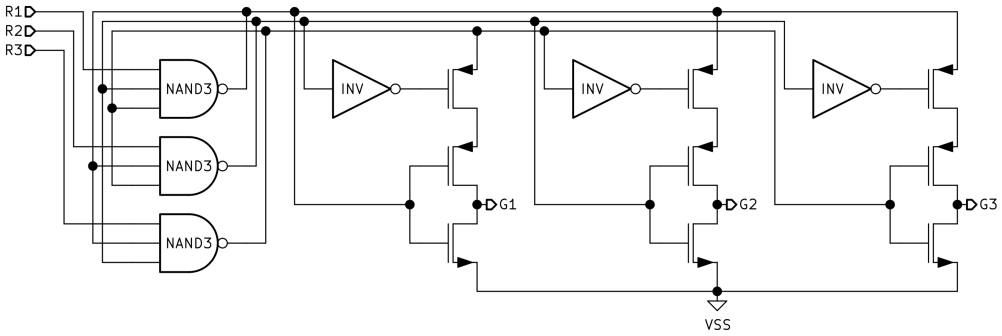


Fig. 23. Asynchronous MUTEX with three lines.

The C-element and the mutex are essential components that must be present as primitives in the ASIC library to successfully implement asynchronous designs in the proposed methodology. However, a set of additional elements exists that, while possible to build from the essential components, it is highly recommended to include as primitives to optimize area usage. The first of the optional recommended elements is the D-Latch with an asynchronous reset. A D-latch is common in almost any ASIC standard component library; however, the standard latch might not be equipped with the asynchronous reset signal. The reset signal's importance stems from the ability to provide a hazard-free reset functionality to the asynchronous circuit by ensuring a clean unconditional reset of the registers. While it is possible to design an asynchronous reset logic from essential gates and the simple D-latch without a reset pin, the solution introduces significant overhead in terms of additional logic and gate fan-out related problems for larger registers. Since the D-Latch primary use is to provide memory for asynchronous registers, any additional logic per bit replicated multiple times would quickly introduce significant growth in the area requirement. The reset logic must ensure that the latch would not accidentally capture any hazardous input from the environment if the EN signal is still high during the reset signal release. The asynchronous reset solves both the area and the reset cycle correctness by adding an independent signal. As shown in Fig. 24 adding reset requires only two

transistors, and during the reset, the EN signal is separately gated by the handshake logic preventing any unwanted input from being accidentally latched into the memory.

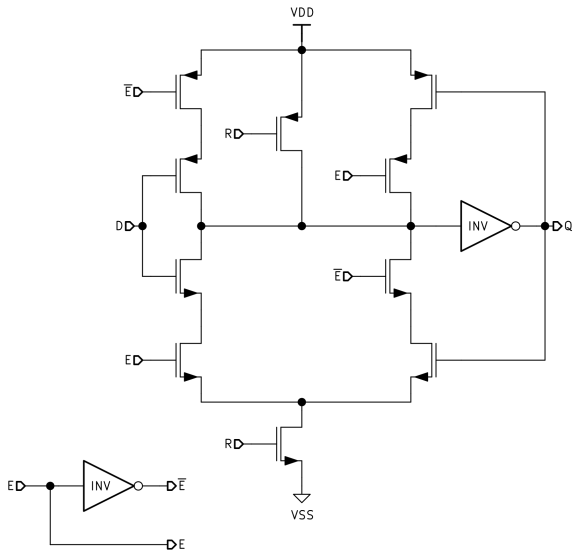


Fig. 24. D-Latch with reset.

The last set of optional components comes from adapting the design methodology to the output generated by the Petrify tool. The generated output produces four configurations of the C-Element with possible inverted output and the reset value being either 1 or 0. Like the D-latch case, the reset functionality implementation plays a significant role in reducing the area requirement. Two intermediate forms of a modified C-element are introduced with a reset capability. Shown in Fig. 25 is a version of the C-Element which a pull-hi signal. The transistor at the input of feedback inverter inhibits any signals coming from the regular gate inputs while the transistor tied to VSS forces the inverter input to become low, effectively setting the gate state to high.

Similar configuration exists for the pull-low variant shown in Fig. 26. The difference in this configuration is the NMOS connected to VDD. Similarly, the input of the feedback inverter is forced to high while all regular inputs are inhibited.

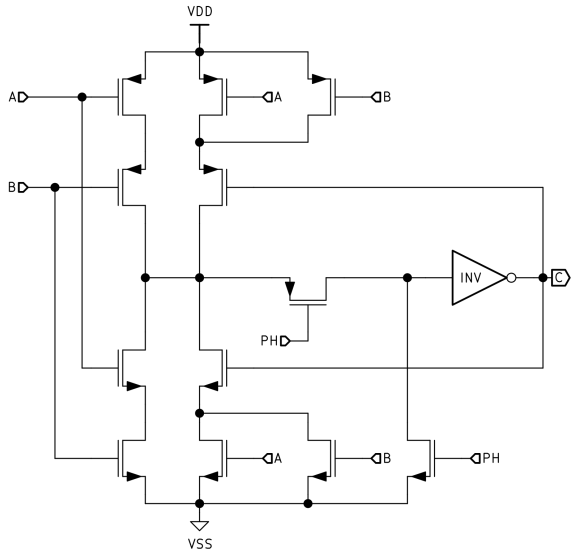


Fig. 25. Generic C-element with pull high reset.

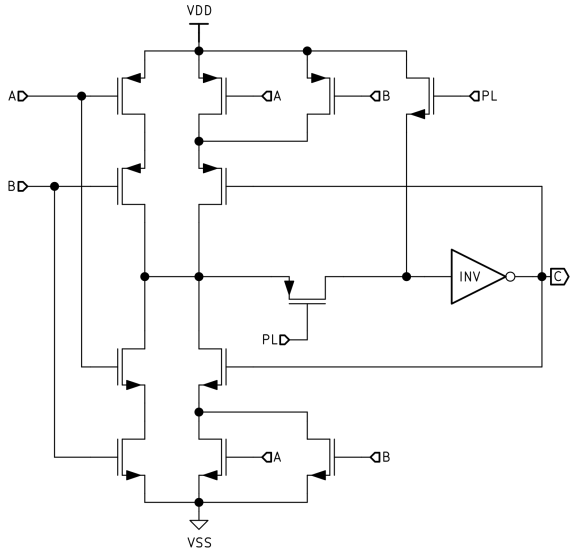


Fig. 26. Generic C-element with pull low reset.

Using the intermediate C-elements with pull signals allows for implementing the four configurations produced by the Petrify tool. The Fig. 27 show the diagrams of all variants, which then could be implemented as transistor-level ASIC primitives. Because of the common occurrence of the C-element-based configurations in the C-Layer, the

implementation of all four configurations as primitives is used to improve the design's area requirements.

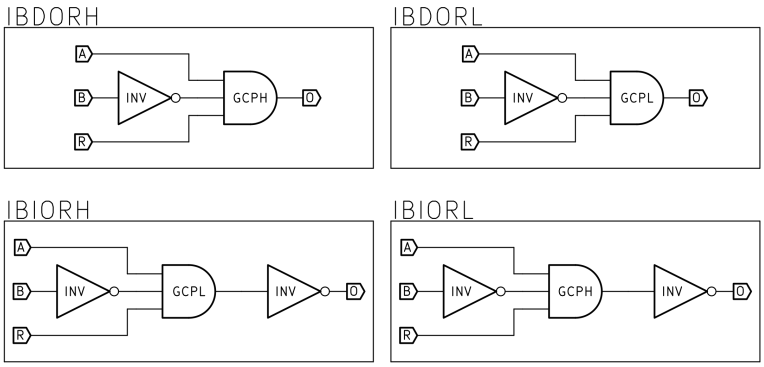


Fig. 27. Petrify output-compatible C-Layer components.

4.6 Model Synthesis and delay matching

The stage two codebase, due to its structure, requires a specialized approach to synthesis. The code consists of elements divided into two types, modules that go directly to synthesis and modules that already contain gate-level specification and will be incorrectly altered by the optimizer. Thus, the traditional top-down synthesis approach in which the entire model is fed at once to the synthesizer and compiled as a whole will not work. Instead, the bottom-up [43], [45] approach is used. The technique uses the synthesizer capabilities to isolate a single module instance and perform synthesis only on that module. An essential aspect of bottom-up synthesis in the asynchronous case is to ensure correct timing such that delay-matched lines would provide a correct delay. For this purpose, the synthesized module inputs and outputs must be constrained by the actual load to which a given element is connected within the chip. The constrain is achieved with the logic synthesizer feature called characterization [43].

4.6.1 Bottom-up selective module synthesis

The synthesis algorithm operates based on the bottom-up approach. The implementations rely on synthesizer capabilities to isolate specific components in the design and invoke the compilation only within the isolated scope. Moreover, the synthesis approach is based on the iterative workflow; the modules are incrementally recompiled while the model is changing. The need for incremental compilation stems from the situation that during the first build, not all the components are yet defined in terms of elements from the technology library; thus, providing an accurate specification of the surrounding environment of the compiled module is impossible. However, after a few loops, all the compiled components become their final gate-level representation allowing the synthesizer to optimize compiled elements correctly. The pseudo-code in Algorithm 3 shows a high-level flow of the synthesis algorithm.

Algorithm 3 Selective bottom up synthesis algorithm.

```
1:  Load global definitions and constants.
2:  Elaborate generic modules for selected parameters.
3:  Load project-specific modules.
4:  Select top module.
5:  Assign external environment I/O loads.
6:  Uniquify
7:
8:  #Initial compile
9:  for all e in (CTRL_COMB_MODULES and FNC_SUPP_MODULES) do
10:     characterize(e)
11:     compile(e)
12:
13: #Iterative compile
14: while i < NUM_ITERATIONS do
15:     for all e in (CTRL_COMB_MODULES and FNC_SUPP_MODULES) do
16:         characterize(e)
17:         compile_incremental(e)
18: Write netlist and SDF
```

The algorithm operates, expecting a specific format of the model source code. Primarily, the name of a file must be the same as the module name it contains. Secondly, every instance of a module must be assigned an identifier. All the design specific modules are then listed and grouped into six categories, as shown in Listing 28. The breakdown differentiates between the components based on their type. Among the categories, the CTRL_COMB_MODULES and FNC_SUPP_MODULES are the ones that will go through automated synthesis while the other categories contain elements already implemented as structural modules.

In the initial stage of the algorithm, the entire design is loaded first. The script loads common source files like types definitions and constants then use the modules lists to load the modules. Meanwhile, the elaboration resolves the Verilog generate construct, which allows for the introduction of parameterized [46], [47] dynamically created asynchronous registers and delay lines. The Listing 24 shows how different versions of delay lines with different lengths and different variations of sizes of the registers are generated. It is necessary to establish specified instances of generic components for the linking process.

After loading all modules, the algorithm first sets the Top module as the current design. Then instantiates default IO load on the Top module using the set_driving_cell and set_load commands. After setting the cell load, the uniquify command is executed, generating multiple copies of any modules reused multiple times. The uniquify is necessary, for example, in a case when the same FSE is instantiated more than once. Since the combinational module within FSE is synthesized, uniquification ensures that every instance of the same module is compiled specifically to its surrounding environment.

```
set CTRL_COMB_MODULES {
    plog_m1_fn plog_m2_fn plog_m3_fn plog_m4_fn plog_m5_fn
    plog_m6_fn recv_m1_fn recv_m2_fn resp_m1_fn resp_m2_fn
    resp_m3_fn resp_m4_fn resp_m5_fn sndr_m1_fn sndr_m2_fn
    sndr_m3_fn sndr_m5_fn sndr_m6_fn
}

set CTRL_HANDSHAKE_MODULES {
    plog_m1 plog_m2 plog_m3 plog_m4 plog_m5 plog_m6
    recv_m1 recv_m2
    resp_m1 resp_m2 resp_m3 resp_m4 resp_m5
    sndr_m1 sndr_m2 sndr_m3 sndr_m5 sndr_m6
}

set CTRL_CLAYER_MODULES {
    plog_ctrl plog_ctrl_nextstate
    recv_ctrl recv_ctrl_nextstate
    resp_ctrl resp_ctrl_nextstate
    sndr_bus_ctrl sndr_bus_ctrl_nextstate
    sndr_core_if_ctrl sndr_core_if_ctrl_nextstate
    sndr_logcheck_ctrl sndr_logcheck_ctrl_nextstate
    sndr_main_ctrl sndr_main_ctrl_nextstate
}

set FNC_SUPP_MODULES {
    plog_supp1
    resp_supp1
    sndr_supp1
    recv_supp1
    ctrl_supp1
}

set FNC_MODULES {
    plog
    resp
    sndr
    recv
}

set TOP ctrl

set TIMING_VERF_NAMES { RP/R1_REG RP/R2_REG RP/R3_REG RP/RT_REG
```

Listing 23. Synthesis elements grouping.

The next step is the compilation during which all the elements from CTRL_COMB_MODULES and FNC_SUPP_MODULES are translated from the behavioral form to the gate-level representation. For every selected module, the synthesizer performs the surrounding environment's characterization to establish input and output load associated with the compiled module. Then the synthesizer compiles the

selected element providing an optimized combinational circuit. The process happens in more than one iteration as the functional blocks might be interconnected, and their internal structure changes after compilation, altering the environment for other modules.

```
#Elaborate parametrized delay and registers
for { set i 1 } { $i <= $MAX_DELAY } { incr i } {
    analyze -format sverilog -define N=$i,S=1 ../../rtl/ctrl_synth/
        support/delay.sv
}

for { set j 1 } { $j <= $MAX_REG_DELAY } { incr j } {
    foreach i $REG_SZ {
        analyze -format sverilog -define WIDTH=$i,DELAY_STEPS=$j ../../
            rtl/ctrl_synth/support/async_reg.sv
    }
}
```

Listing 24. Elaboration of parametrized modules.

The process executes the initial compilation once and then performs incremental iterations through the entire list of components selected for synthesis and finally circles back. The Listing 25 shows the iterative loop in which the algorithm processes the FSE combinational and the Component support modules. An important thing to note is the two internal loops that iterate over the result returned by get_designs. Even though all the modules that require synthesis are already listed in the input data shown in Listing 28, as a result of the uniquify command, some FSE can get replicated, creating newly generated instances which also must be included in the synthesis process. Calling get_designs with parameter composed of the module name concatenated with a post-fix wildcard allows catching all the generated modules.

```
for {set i 0} { $i < $SYNTH_ITER } {incr i} {
    foreach n $CTRL_COMB_MODULES {
        foreach_in_collection m [get_designs ${n}*] {
            characterize_comb_module $m
            compile_comb_module_incr $m
        }
    }

    foreach n $FNC_SUPP_MODULES {
        foreach_in_collection m [get_designs ${n}*] {
            characterize_comb_module $m
            compile_comb_module_incr $m
        }
    }

    current_design $TOP
}
```

Listing 25. Compilation main loop.

```
proc characterize_comb_module {object} {
    global TOP
    current_design $TOP
    set obj_name [get_object_name $object]

    set query_str_list "ref_name == "
    lappend query_str_list $obj_name
    set query_str [join $query_str_list]

#NOTE: Because of uniquify we expect always a single instance
    set query_cell [get_cells -filter $query_str -hierarchical]

    characterize -constraints $query_cell
}

proc compile_comb_module_incr {object} {
    set obj_name [get_object_name $object]

    current_design $obj_name
    set_max_area -ignore_tns 0
    set_dont_touch bundle_delay*

    compile_ultra -incremental
}
```

Listing 26. Characterization and compilation.

The synthesis of a module happens using two steps, module characterization and then compilation. The first action performs a characterization of the synthesized modules. The characterization extracts capacitive loads placed on modules input-output pins by its environment and setups as parameters for the processed component. The second step is

the actual compilation, during which the characterized module is translated to its gate-level representation, or the logic is improved if an incremental compilation occurs. The Listing 26 shows the characterization and compilation procedures used by the script.

After several passes over the modules list, the entire design stabilizes, and the synthesis finishes. The last step is to save the results of synthesis. Two essential files are created, the gate net-list and the SDF file that contains delays extracted from the ASIC cell descriptions, which are then used to simulate the gate-level model correctly.

4.6.2 Post-synthesis timing analysis and delay matching

After the synthesis, the gate-level model must be examined for correctness related to timing. All the flow support elements and the asynchronous register delay-matching lines must be checked. The verification procedure relies on the timing analysis of the synthesized model. A sample of the data format used for verification is presented in Listing 27, showing an example for the data and decision FSE.

```
#Data FSE timing
ELEM_TIMING PL/M3:
  crit:
    set_out_reg_ack: r(38.53)/f(38.53) DELAY/DELAY
  all:
    set_out_reg_ack: r(38.53) DELAY
    out_valid: f(35.44) SIGNAL
    out_this_unit: f(28.79) SIGNAL
    out_rdx: f(28.79) SIGNAL

#Decision FSE timing
ELEM_TIMING RP/M2:
  crit:
    p_log_respond: r(46.48)/f(45.33) DELAY/DELAY
    p_log_skip: r(47.21)/f(45.96) DELAY/DELAY
  all:
    p_log_skip: r(47.21) DELAY
    p_log_respond: r(46.48) DELAY
```

Listing 27. Timing analysis output.

Every record contains two sections. First, the critical section shows delays for lines that are affected by the delay-matching logic. Each line shows the signal's name, rising

and falling delay, and whether the printed signal long-path results from the signal traveling through delay logic or regular logic. When a signal is marked DELAY, it means that the printed delay is due to the delay logic, and when the SIGNAL marking appears, this means the long path is due to the regular module logic and exceeds the matched-delay path. The critical section provides a pair of markings, one for rising and one for the falling delay.

The second section, titled "all", is the list of signals starting with the longest delay. Here the ten or less slowest signals are listed for verification, including the critical lines. Unlike the critical section, the all section displays either rising or falling delay, whichever is longer. An important point to note is that if a signal displays a rising delay, then the falling delay is always shorter. The same is true for the opposite; if a falling signal delay is displayed, then the rising is shorter.

Condition for correctness varies between the types of elements. For the data-type FSE and asynchronous registers, the critical line must have marking DELAY, and the critical line delays both for raising and falling must be longer than any other line within the module. If a module is a mixed-type FSE, for example, combining two data-type functions, then the delays of both critical lines must exceed any other signal. Even if the data signal within the FSE applies only to one function, not to the other, still, both critical handshake lines must exceed its delay. The requirement for the critical lines to be the slowest avoids any unexpected hazards. Unexpected transitions can occur on any line within the module while the combinational logic stabilizes. If the element is the Support module, the critical line must have marking DELAY with a delay value that exceeds only the corresponding signals, not all the signals within the module.

In the decision-type FSE, the analysis shows only the handshake signals, which at the same time are the only outputs from the module. For decision-module timing to be correct, all the signals must have a marking DELAY. The same rules apply for the

mixed-type module in which one FSE contains data and decision-type functionality or other variations. All the critical signals must have a delay longer than any data line and the DELAY marking for both rising and falling transition.

The implementation of the timing extraction algorithm works similarly to the synthesis. The model must use names for every instantiated module. The analysis code requires a list of analyzed elements and a list of critical signals for each record. The Listing 28 shows a portion of the file describing the list of modules for timing analysis. The primary requirement for the codebase implementing the model is that all the instantiated modules have assigned identifiers. The identifiers are then listed as a full path starting from the instances in the top module. For example, the TIMING_VERF_NAMES list, which contains the full list of all analyzed elements, has a record PL_/M1 which points to an instance called M1 within the RP element then the PL is instantiated within the top module. Then this instance, M1 is a subject for timing analysis. The second piece of information is the list of critical signals within the tested module. The set TIMING_VERF contains a collection of lists of one record for each tested element.

```
    SD/LINE_REG RV/LINE_REG PL/M1 PL/M2 PL/M3 PL/M4 PL/M5 PL/M6
    RV/M1 RV/M2 RP/M1 RP/M2 RP/M3 RP/M4 RP/M5 SD/M1 SD/M2 SD/M3 SD/M4
    SD/M5 SD/M6 SUPP1 }

set TIMING_VERF(RP/R1_REG) { "ack" }
set TIMING_VERF(RP/R2_REG) { "ack" }
set TIMING_VERF(RP/R3_REG) { "ack" }

set TIMING_VERF(PL/M1) { "recv_proc_skip" "recv_proc_mark" "
   recv_proc_clr" }
set TIMING_VERF(PL/M2) { "sel_storage_ack" }
set TIMING_VERF(PL/M3) { "set_out_reg_ack" }
set TIMING_VERF(PL/M4) { "set_storage_ack" }
set TIMING_VERF(PL/M5) { "sndr_proc_ack" }
set TIMING_VERF(PL/M6) { "resp_proc_ack" }

set TIMING_VERF(RV/M1) { "stp1_ack" }
set TIMING_VERF(RV/M2) { "stp2_ack" "stp3_skip" "stp3_write" }
```

Listing 28. Synthesis elements grouping.

With the information provided, the algorithm keeps iterating over the list of modules. The Listing 29 shows the central part of the analysis algorithm. When a module is selected for analysis, the first step is to perform characterization to obtain accurate timing results. However, since the communication is based on handshaking, it is safe to assume that all the module's data inputs are stable before the module itself is triggered to work. The standard synthesizer for sequential logic not aware of the delay-matched FSE working principle; therefore, all the delays on input lines must be cleared manually by the analysis algorithm.

```
        characterize_comb_module $r
        current_design $obj_name

        remove_input_delay [all_inputs]

        echo "\n\nCRIT-CRIT-CRIT-CRIT-CRIT-CRIT-CRIT\n\n" > build/
            synth_timing/$file_name.timing


        foreach sig $TIMING_VERF($query_cell_name) {
             report_timing -nosplit -rise_to $sig >> build/synth_timing/
                $file_name.timing
             report_timing -nosplit -fall_to $sig >> build/synth_timing/
                $file_name.timing
        }

        echo "\n\nALL-ALL-ALL-ALL-ALL-ALL-ALL\n\n" >> build/
            synth_timing/$file_name.timing
        report_timing -nosplit -max_paths 10 >> build/synth_timing/
            $file_name.timing
    }
```

Listing 29. Extraction of signals for analysis.

After characterizing the module and clearing input delays, the algorithm then iterates over the critical signals list and obtains timing reports for each rising and falling transition. Finally, the algorithm prints a timing analysis report for the entire module with ten or fewer slowest lines. The synthesizer's raw output is then processed and compiled into the format shown in Listing 27. The DELAY and SIGNAL markings are obtained by inspecting the entire path. If the path contains a module with the name matching the

delay-line element, then the long path must be going through the delay element; otherwise, it is a SIGNAL marking. The final delay becomes the total path delay for the specific signal.

During the analysis, if the timing is not met, the delay-matching line must be extended by additional segments, then the synthesis repeats until the correct solution is present. The timing results might oscillate across the modules between correct and incorrect, but after a number of delay-tuning iterations, the delays stabilize and arrive at the correct result. Obtained gate-level representation is then ready for back-end processing that leads to the final form in GDSII format.

4.6.3 Post Place and Route delay matching through ECO

During the place and route process, a more accurate model of the design emerges. Aside from the ASIC primitives delays, the additional interconnect delays show up as the tools can determine the actual connectivity and have insight into the fabrication process. The added path delays have a high probability of altering the circuit's timing to the point when delay-matched lines become incorrect. In such a case, it is necessary to employ the Engineering Change Order (ECO) approach to fix the invalid delay paths. The ECO revolves around making small "last-minute" changes to the nearly finished design, allowing to avoid returning to the earlier design stages.

One of the useful properties of asynchronous design is that there is no clock tree synthesis, and as such, the place and route tool do not alter the underlying netlist. Because the netlist remains unaltered, the same timing analysis tools from the synthesis stage can be used to determine the circuit's correctness after Place and Route. The place and route tool export an updated SDF file, which contains both cell and net delays. The updated timing is then used to output the timing report.[1]

1. Tested on Cadence Innovus

For the paths that fail the timing due to the added path delay, it is necessary to extend the matching delay. The extension is done as a part of the ECO workflow on existing gate-level representation. It is impossible to use any of the Verilog behavioral constructs at this stage, and all changes must be done strictly at the structural level. The following procedure is verified for the Innovus suite but applies to any other tool of equal functionality. The existing netlist first must be exported from the tool by using the saveNetlist command. The specific instance of the delay line corresponding to the failing element must be found, and additional delay elements are injected into it. Finally, the updated netlist is loaded using the ecoDesign command. An additional placement and routing might be required if the ecoDesign is invoked with parameters noEcoPlace or noEcoRoute. For the additional place end route, the ecoPlace and ecoRoute commands are used. Finally, the checkPlace and verifyConnectivity commands validate the design. The delay matching in the last stage might require multiple iterations similar to the synthesis process. Eventually, the design stabilizes, producing the correct result.

4.6.4 Synthesis of the Controller Circuit

The speed-independent controller circuit requires special treatment when it comes to synthesis into ASIC library elements. Standard combinational logic synthesis algorithms are insufficient when it comes to the decomposition of the next-state logic equations. Due to the gate and interconnect delays, synchronous model-oriented synthesis could produce a result that contains hazards [4], causing malfunction of the controller and generally breaking the speed-independent assumptions. Specialized [4], [9] synthesis and hazard free logic decomposition methods must be applied separately to the controller. Stage 2 must reexamine the STG and synthesize it into gate-level representation. Then the gate level structural next-state modules are plugged directly into the synthesis process as black boxes.

The necessity to synthesize STG again, specifically for stage 2, can invalidate the stage 1 model. The invalidation can happen because of the signal insertion algorithm [4], [9] that simplifies the set and reset functions by adding additional intermediate internal signals into the model, which splits an output line function into smaller subsets. The new signals alter the SG and, as a result, the controller structure. In some cases, it might be beneficial to re-run stage 1 tests on the set of next-state equations obtained after stage 2 specific synthesis. Note that stage 2 synthesis, in this case, produces two representations of the same result. One is the structural gate-level next-state module, and the second is the next-state Boolean equations. Stage 1 model still operates under the single complex gate delay assumption but with the new set of signals and potentially different CSC.

In some cases, hazardous behavior can manifest after the Place and Route. Introduced interconnect delays could potentially slow down certain paths causing hazardous behavior in high-speed circuits. Though it is unlikely because the asynchronous synthesis algorithms attempt to construct circuits that behave correctly within the set of possible SG states, there still exists a chance that added interconnect path delay could violate speed independence. Thus, it might be necessary to perform P&R layout using a version of the placement algorithm that is aware of potential hazards and perform delay matching on interconnect lines. In this case, the P&R is done on the controller circuit separately. Then, when finished, the resulting block is inserted into the entire design as a black box around which the automated standard P&R happens.

Although not recommended, it is possible to synthesize the controller's next-state functions using synchronous logic algorithms. For that purpose, the stage 1 modules containing the set and reset logic need to be included as COMB modules, as shown in Listing 30. This way, it is possible to end up with a result for which either the hazardous behavior does not exist or the hazardous behavior does not manifest at the controller outputs, or there is no potential sequence of states that would cause the hazardous

behavior. However, using a standard synthesizer for synchronous logic, in general, is still incorrect and must be used with caution as it cannot guarantee a hazard-free result.

```
set CTRL_COMB_MODULES {
    plog_ctrl_nextstate plog_m1_fn plog_m2_fn plog_m3_fn plog_m4_fn
    plog_m5_fn plog_m6_fn recv_ctrl_nextstate recv_m1_fn recv_m2_fn
    resp_ctrl_nextstate resp_m1_fn resp_m2_fn resp_m3_fn resp_m4_fn
    resp_m5_fn sndr_main_ctrl_nextstate sndr_bus_ctrl_nextstate
    sndr_core_if_ctrl_nextstate sndr_logcheck_ctrl_nextstate
    sndr_m1_fn sndr_m2_fn sndr_m3_fn sndr_m5_fn sndr_m6_fn
}

set CTRL_HANDSHAKE_MODULES {
    plog_m1 plog_m2 plog_m3 plog_m4 plog_m5 plog_m6
    recv_m1 recv_m2
    resp_m1 resp_m2 resp_m3 resp_m4 resp_m5
    sndr_m1 sndr_m2 sndr_m3 sndr_m5 sndr_m6
}

set CTRL_CLAYER_MODULES {
    plog_ctrl
    recv_ctrl
    resp_ctrl
    sndr_bus_ctrl
    sndr_core_if_ctrl
    sndr_logcheck_ctrl
    sndr_main_ctrl
}

set FNC_SUPP_MODULES {
    plog_supp1
    resp_supp1
    sndr_supp1
    recv_supp1
    ctrl_supp1
}

set FNC_MODULES {
    plog
    resp
    sndr
    recv
}

set TOP ctrl
```

Listing 30. Synthesis elements grouping with controller next-state equations.

5 CASE STUDY: ASYNCHRONOUS MESI CACHE COHERENCE CONTROLLER WITH SPLIT TRANSACTION BUS

The presented case study's goal is to evaluate and validate the proposed methodology. Serving as the proof of concept is a cache coherency controller element that implements the MESI memory coherency algorithm and communicates through a split transaction bus. The design provides sufficient complexity to show the capabilities and prove or disprove the methodology's usefulness. Featuring the MESI algorithm as a non-linear and data-driven flow makes the controller a good fit within the covered class of devices. The execution is non-deterministic as it depends on multiple competing entities in the system and involves arbitration and complex decision making based on the data. The algorithm implementation provides sufficient complexity to show the advantages of the proposed methodology and analyze its effectiveness.

The presented case study features two implementations of the controller, the asynchronous version that follows the proposed methodology and a synchronous reference baseline. The goal of having the synchronous reference design is to evaluate the methodology and the resulting asynchronous version compared to a design created in a well known and widely used approach. Both implementations follow the same algorithm to avoid discrepancies in the results coming from the difference in the working principle. However, some implementation differences exist, which are caused by the use of different design methodologies that are expected and intend to show strong and weak sides of both synchronous and asynchronous design.

5.1 Cache Coherence MESI Algorithm and Split Transaction Bus Review

The multi-level cache memory model in multiprocessor systems brings significant processing performance improvements but introduces additional memory coherency problems. The locally cached subset of memory space for every processor must be kept

consistent among all CPU units despite the disconnect. One among a number of the solutions to the problem is the snooping MESI cache coherency algorithm. In a snooping protocol, the cache controller maintains the local cache state by listening to the information flow on the bus modifying the applicable local cache accordingly [10].

With the MESI algorithm, a cache line, in addition to its regular structure [11], gains an additional field that reflects the MESI state. The MESI state describes the cache line status with reference to its coherent state spanning all the processors working on the shared memory space. It is a four-state [12] value that specifies the cache line status as:

- (**M**)odified - The local cache memory contains a unique copy of cache line data that was modified in previous cycles. If other cores have a copy of this line, then its state in the other cores memory must indicate invalid with an outdated value. The main memory also contains an outdated value.
- (**E**)xclusive - The local cache memory contains an exclusive valid copy. If any other cores have the same cache line, then its status in other cores' local memory must indicate invalid with illegal to use value regardless of whether it is outdated or not. The main memory contains the current value.
- (**S**)hared - The local cache memory contains a shared copy between other cores and the main memory.
- (**I**)nvalid - The local cache memory contains an invalid copy which unreliable value.

The cache line's MESI state determines what steps must be taken when performing an arbitrary memory operation. For example, if a core writes to a cache line that holds an exclusive state, then the data is modified immediately, and the MESI state changes to modified. However, if the MESI state is shared instead of exclusive, then the core must first send an upgrade (Upg) signal to all other cores, so they invalidate their copies while promoting its local version from shared to exclusive. Then the actual write can occur.

The MESI cache coherence algorithm allows for preserving memory consistency in multiprocessor systems with multi-level memory hierarchies. Separate CPU cores, despite having isolated snapshot subsets of the entire memory space, still see the whole address space as a single consistent entity. The MESI is a snooping algorithm that listens to all bus communication and adjusts the local memory state accordingly.

Two-way communication on a shared bus in a multiprocessor system can become a performance bottleneck, for example, when considering an atomic BusRd [12] message, which is a request for data sent by a core. The Sender locks the access to the bus and sends the request. Then the entire system waits until one of the other units sends a response. In an atomic transaction model, the system is effectively locked until the transaction finishes. The locked state can cause a significant system slowdown, especially if the response comes from the main memory directly. Holding the bus inhibits not only other data requests but also some seemingly internal and unrelated actions happening within other cores. As seen previously, a core must send the Upg message to perform a write into the cache if the line is in shared status. If the bus is locked, that core must wait to send Upg even though it executes an unrelated operation to the currently ongoing bus-read request. In effect, an unrelated core is stalled, and the effect could spread to more CPUs' overtime, significantly degrading the system performance.

The solution to the atomic message bottleneck is the split-transaction bus [12]. In the split-transaction model, the request for data and the response are broken down into two separate bus messages. The requesting unit puts a request message then immediately releases the bus. While other units prepare the response, other communication can go through. Finally, the responding entity locks the bus access and sends the response. The split-transaction model allows for more efficient bus utilization but places a heavier burden on the communicating elements by imposing the need to track outstanding transactions. For example, core A requests a cache line, which can be delivered as a

response either by core B, core C, or main memory. All the three elements now keep working on retrieving the data from local storage and finally compete with each other for the bus to deliver the response. In the end, only one unit will deliver the response, while all the others must cancel the procedure and switch to another task. The improvement comes from the fact that while the response is prepared, the bus is free and other communication can occur. The split transaction model allows for improved utilization of a shared bus and prevents bottleneck caused by two-way atomic messages but at the cost of more complex implementation.

5.2 Cache coherency controller design goals

The designed cache coherency controller is a module managing the L1-level cache within a single CPU core. The design is intended to work in a multi-core environment in which every CPU core contains its local controller module. The controller maintains all the bus communication and implements the CPU core memory interface. Effectively, the controller's working environment is the CPU core, the cache memory, and the system bus, as shown in Fig. 28.

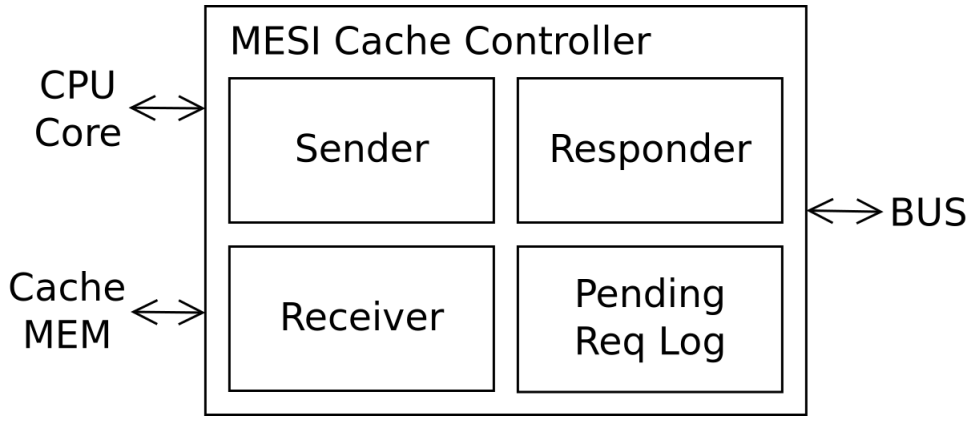


Fig. 28. The MESI cache controller overview.

Following the divide-and-conquer approach, the MESI controller breaks down its operation into four components. The four components are: Sender, Receiver, Responder,

and the Pending Request Log (PLOG) component. The Sender component is the primary module servicing requests from the CPU core. Sender reads and writes the data to the local cache as requested by the core or performs bus communication if necessary in cases such as requesting a cache line or signaling a MESI status change. The Receiver component, which task is to snoop on the bus communication and act on relevant messages. When a message on the bus is relevant to a specific controller, its Receiver acts on it by modifying the local cache content and, if applicable, updating the pending request log records. For example, the Receiver is the entity that updates the internal cache when a message with a requested cache line comes through the bus or invalidates a cache line if a message comes through the bus that indicates that another CPU core modified a shared cache line.

The used bus type is a split-transaction bus, which requires additional logic to track outstanding requests from the system CPU cores. The split-transaction operation is implemented through the combination of the Pending Request Log and the Responder modules. The Pending Request Log is a storage element that keeps track of the outstanding requests. The log module content is modified by the Receiver when it encounters a request or a response message on the bus. When the request appears on the bus, the Receiver places a new record in the PLOG with the corresponding address. Then when the response follows, the record is cleared. The final Responder component is responsible for answering the cache-line requests from other cores; it crawls the Pending Request Log records, and if the Responder determines that the current core entity can supply data to another core, it attempts to deliver it. It is possible that multiple cores have valid data in their local storage, and both can respond to the same request. The contention is resolved on a first-come, first-served basis. The core that obtains access to the bus first sends the response while the second one cancels the operation as it is no longer needed.

The designed controller works within a system with the structure shown in Fig. 29. It is a multi-core system with all the elements connected through the parallel bi-directional bus [48]. The bus arbiter resolves the contention for access to the bus. Presented proof of concept assumes the existence of three cores but also discusses the approach to extend the core count.

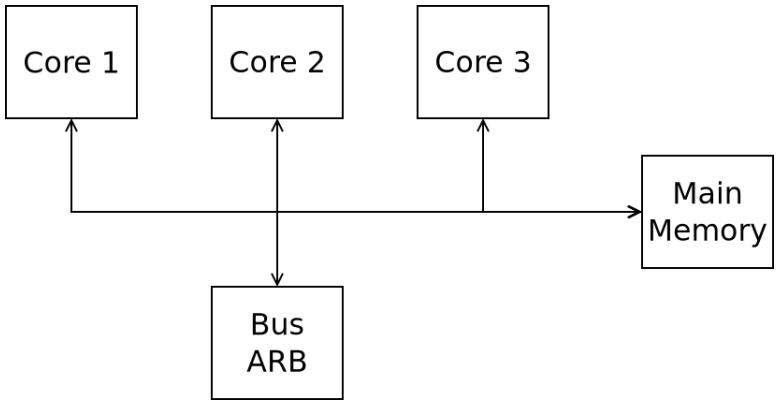


Fig. 29. System block diagram.

The communication within the system happens through the split-transaction bus. The communication bus uses a single message format shown in Fig. 30. The message format contains four fields: memory address, data field, message origin flag, and the operation type. The memory address always exists and must be valid, but the valid data field is not always required. For example, the BusRd message is only a request for data containing the memory address but not the actual content. The origin mm_sndr field indicates whether the message originated from the main memory or a CPU core. In certain cases, the receiver must know the origin of the message to correctly assign cache line status within the local memory. The last field, named "op", indicates the type of operation related to the message and is always valid. The operation field carries one of the bus message type identifiers, such as BusRd or BusRdX [12]. This version of the controller also implements an optimization on top of the standard MESI algorithm in the form of the

Upgrade (Upg) message that allows for a core to obtain exclusive access to the cache line without sending the two-way BusRdX message if it already contains the valid cache line.

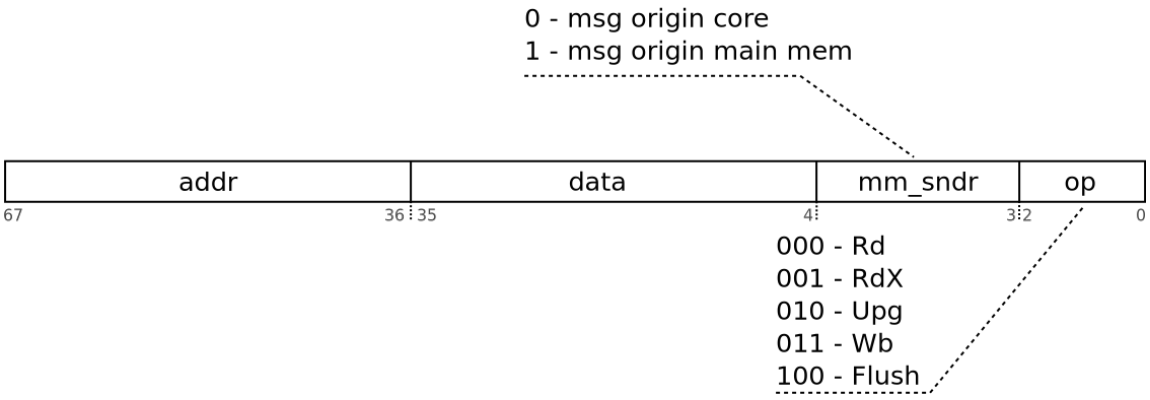


Fig. 30. Bus message packet structure.

5.3 Cache coherency controller design assumptions

While aiming to introduce a sufficiently complex model, the controller design still makes some assumptions to prevent overly complicated implementation. While the presented implementation realizes the MESI algorithm to evaluate the proposed methodology, it still lacks certain features to become a full-featured solution in a realistic computer system. For the purposes of the presented work, the controller design makes a few simplifying assumptions. The first assumption made in the design is that regardless of the system's state, the CPU cores will always respond faster than the main memory. Under this assumption, the CPU core is guaranteed to respond faster than the main memory regardless of the delay under the condition that any core contains the requested cache line. The first assumption simplifies the problem of CPU cores racing with memory when responding to a data request. For example, such a situation occurs when a cache line in a modified state exists in one of the cores. The core that contains the cache line in a modified state is the only entity that contains the most recent data, thus being the only one

that can respond to this request. Simultaneously, the main memory does not keep track of every core's internal cache state. Therefore a potential scenario exists when the responding core is in a state that is delayed long enough so that the main memory would respond first and deliver an outdated value causing memory decoherence. The presented design assumes that such a situation will never occur, and the cores would always be quicker than the main memory.

The second assumption relates to different memory regions with different characteristics. Realistic computer systems contain memory regions that cannot be cached, such as MMIO, for which an address in memory space effectively points to a register of a peripheral device instead of general-purpose memory. During regular operation, when reading and writing to a memory, the Sender module initially contacts the internal L1 cache memory, but if a region could not be cached, the Sender must, by default, ignore the internal memory and issue a bus message. To implement this functionality, the controller must contain additional logic used to resolve such mapping. This implementation assumes that all the memory space is a general random access memory without regions that cannot be cached.

The third assumption is the number of cores present in the system. This implementation assumes three cores. The number of cores determines the size and implementation approach of the Pending Request Log and the Sender module's workflow. The implications are further discussed in the section 6.2.

The final assumption is that the cache line size is a single word (4 bytes). The cache line of this size is intended to cause increased bus communication, thus providing more data points for the power usage tests. In real life systems, the cache line size is normally larger than a single word. Implementing multi-word cache line size requires additional logic and approach discussed in section 6.2.

5.4 Asynchronous Cache Coherence Controller Design

The presented cache coherency controller design is an asynchronous MESI implementation working over a split-transaction bus. The discussed controller design interfaces with three other entities, the CPU core, the local L1 cache memory, and the system bus. An overview of the interface connections is shown in Fig. 31. Each interface consists of the data-path lines and control signals that use a 4-phase handshake protocol for synchronization. The CPU core does not exchange any information directly with the internal cache memory neither with the bus. This setup allows for a simple interface for the CPU core while the controller facilitates all the communication.

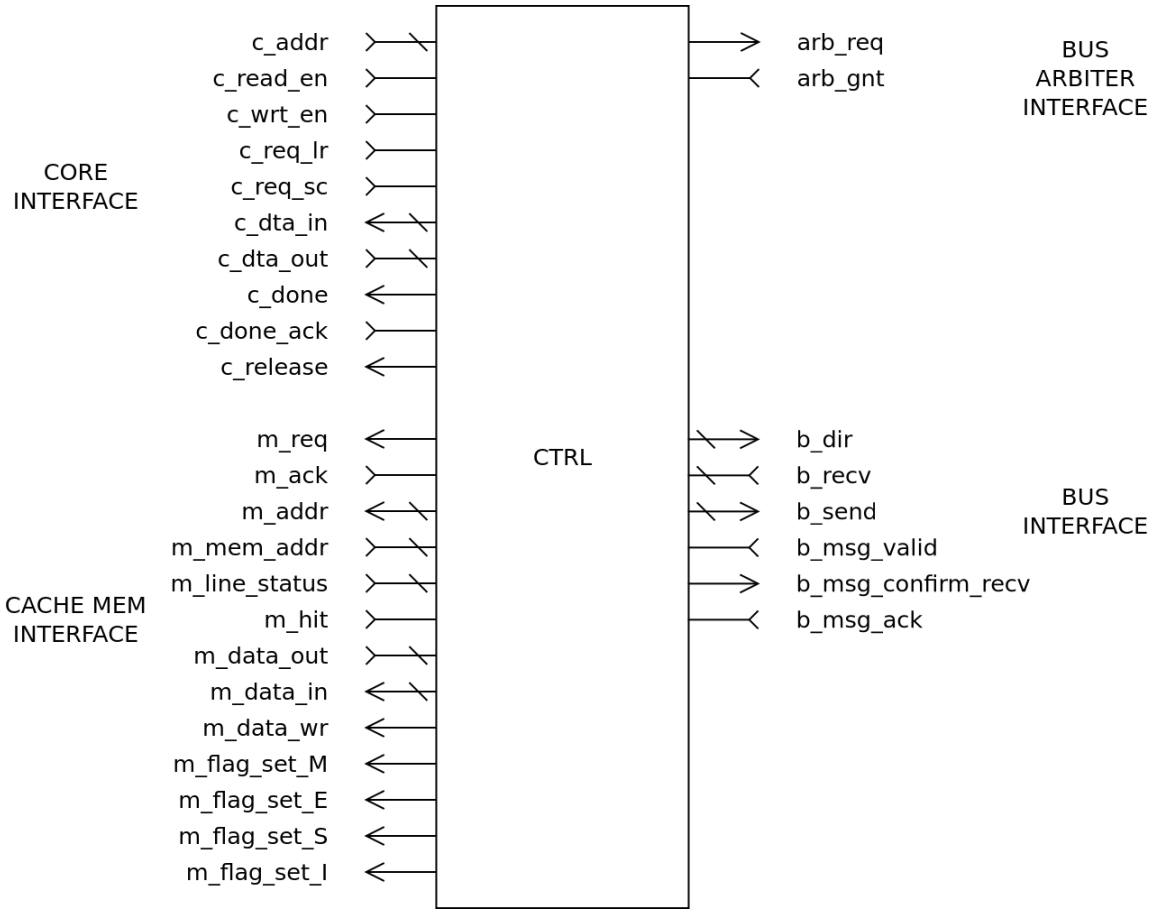


Fig. 31. Controller interface.

The interface to the CPU core module is an implementation of the RISC-V type of core. Both read and write signals are supported and communicated through the c_read_en and c_wrt_en signals. Also, the core supports the atomic pair used to support multi-core parallel applications. The atomic pair [49] is a set of two instructions, load reserved (LR) and store conditional (SC), that guarantees an atomic read and write to a memory address that forms a basis for the compare-and-swap (CAS) paradigm used to implement spinlock mechanism [10]. One way to implement the atomic pair functionality is through the L1 cache. The load reserved (c_req_lr) operation obtains an exclusive state to the cache line region of interest. Then the store conditional (c_req_sc) writes to this region if the core still retains the exclusive state. If the core loses the cache line's exclusive MESI state at any time between the two operations, the store-conditional instruction fails, the data is not written, and failure indicating status is returned to the CPU core.

At the interface between the CPU core and the controller, the handshake occurs at two levels. The CPU core acts as the active end and initiates the transaction. The transaction begins with the assertion of one of the control signals; either read, write, load reserved, or store conditional line is brought high. When the controller finishes, it responds with c_done signal. At this point, the CPU core processes the data from the controller and, when finished, responds with c_done_ack after which the internal handshake finishes but not the transaction between core and the controller. Finally, when the controller is ready to finish the transaction, it emits the c_release signal, after which the core withdraws its control line asserted at the beginning, which finishes the entire transaction process. The double-nested handshake setup is necessary to ensure the controller completes its operation. In some scenarios, even when the core is done with the transaction data, the controller still works through additional steps for which it needs continuous stable input from the core. The c_release signal allows the retention of this stable input state until the

controller completes any additional operations simultaneously, allowing for the CPU core to finish its internal tasks as soon as possible.

Next to the control signals, the CPU-controller interface also features a set of three data lines. The two outgoing lines out of the CPU core are the address and data. Depending on the transaction type, the data line could either carry information or hold its default state with all bits set to zero. In the other direction, from the controller to the core, only the data line exists. Similarly, the incoming data path either caries information or is zeroed out. The address path goes only in one direction, and the controller never dictates the value of the memory address.

The second interface is the connection to the L1 cache memory element. As in all interfaces, the cache interface is controlled by the 4-phase handshake protocol. Unlike the CPU core interface, the memory interface uses only a single-level handshake composed of signals m_req and m_ack. In the controller-memory setup, the controller is the active element initiating transactions.

The controller-memory interface data-path splits into three semi-independent parts. When the controller initiates the transaction, it must provide a valid address through the m_addr line and the address is used by all the parts of the interface. The first subset of the controller-memory interface is the read subset. Regardless of the operation invoked on the memory, the module always returns the current data under the index computed from the m_addr field. The subset returns the data currently in memory using the m_data_out line and the current address in memory m_mem_addr. The returned address can be different from the one given by the controller, for example, when a new data overwrites current. The subset also returns the current cache line status through the m_line_status line and whether the cache hit occurred signaled by m_hit. When new data is written to the cache, the read subset will still return the current content and its corresponding address that

occupies the memory before writing. An additional transaction is necessary after write, to read back the newly written information.

The second subset is the data write subset. The subset consist of the m_data_in line and m_data_wr. When the m_data_wr is asserted high the data under index computed using m_addr is overwritten by the data provided through m_data_in. An important thing to note is that the cache line flag does not change automatically with the write. To modify the cache line flag, the controller uses the third subset composed of four signals m_flag_[MESI]. Asserting a line corresponding to the specified MESI cache line state causes the cache line state to change. The third subset is independent of the write subset allowing for modification of a cache line state without executing a data write to the memory. Depending on the situation, the controller might modify a cache line state along with a write, not modify the cache line state during the write or modify the cache line state separately without an actual write.

The third and final interface is the bus interface. Further down, the bus interface breaks down into the communication and arbitration sub-interfaces. The arbiter section connects to the global bus arbiter deciding the bus access between all CPU units. From the controller's perspective, the arbiter interface looks like a 4-phase handshake pair for which the controller is the active side. The only exception from the regular 4-phase behavior is that the controller can withdraw the arb_req signal at any time, even without being first confirmed by the arbiter.

The second sub-interface of the bus interface is the communication sub-interface. The interface structure is designed to be connected through a module that converts the uni-directional b_send and b_recv into a bidirectional bus data line. The bus-communication control cycle for a controller that is granted by the arbiter is as follows. First, the controller asserts the b_dir line indicating that this unit sends a message packet over the bus. At the time when b_dir asserts to the SEND state, the b_send line

must be valid and stable. As a response, the bus circuitry sends back b_msg_valid to all cores, including the message sender, indicating that the message propagated through the bus. When a participating core, including the sending core, finishes processing the bus message, it emits the b_msg_confirm_recv. The sending controller must also confirm its own messages resulting from the internal design split into four subsystems. Additionally to external messages, the local Receiver component processes messages sent by the local Sender or Responder units. For example, a given controller emits the Upg bus message, which elevates a cache line's status for the sending core to exclusive while all other cores that have the matching cache line in their local memories must invalidate. The Upg message is placed on the bus by the Sender submodule while the Receiver submodule from the same unit handles the flag change in local cache memory exclusive. Simultaneously, the Receiver components from all other cores test if their local memory contains a matching cache line and, if true, invalidate it. Therefore, the sending controller unit waits for message processing confirmation from every unit, including itself. All the b_msg_confirm_recv are and-ed together, and when all are high, the b_msg_ack goes high and is propagated to the sending controller. After which, the handshake unfolds; first, all the b_msg_confirm_recv go back low, resulting in b_msg_ack to go low with last falling transition. Then b_dir is withdrawn resulting in b_msg_valid to eventually go low. Finally, the controller withdraws its bus access request from the arbiter.

All three interfaces compose of a connectivity set for the cache coherency controller. Internally the controller itself consists of four sub-modules working together. The following sections first describe the portion of the algorithm driving each sub-module and its internal structure. Then the discussion finishes by presenting how all the sub-modules interconnect together into the full controller module.

5.4.1 Pending Request Log component design

The pending request log (PLOG) is the first of the four primary components within the controller module. The PLOG component serves a supporting role for the rest of the elements and is a key part of the split-transaction bus functionality implementation. When a cache line's request appears on the bus, its corresponding response does not come in the same bus access cycle. Instead, the bus control is released by the requesting component. The response that originates from another core or the main memory comes later as a separate transaction, thus effectively splitting the data request Rd and RdX into operations composed of two transactions.

This implementation delivers the snooping coherency protocol model in which the cores react to communication on the bus but do not store information about memory state other units. However, each core does keep track of outstanding split transactions and uses the PLOG for this purpose. When any core puts an Rd or RdX request on the bus, all cores, including the sender, snoop the event and place a record about it in their internal PLOG instances. The existence of an outstanding transaction is critical information for all other internal components of a controller resulting in the PLOG interfacing with Receiver, Sender, and the Responder components as shown in Fig. 32.

The Sender component interface provides data for the Sender module functionality that prevents a controller from issuing colliding data requests. Before posting any transaction, the Sender module queries its local PLOG component. If a matching address is present, meaning there is already an outstanding transaction related to the address in question, then the Sender pauses until the currently pending transaction clears. The Sender interface uses a 4-phase protocol for handshaking in which the Sender acts as the active side. The Sender puts an address onto the query_addr, and if a matching record marked as ongoing exists within the PLOG, then the query_match becomes high.

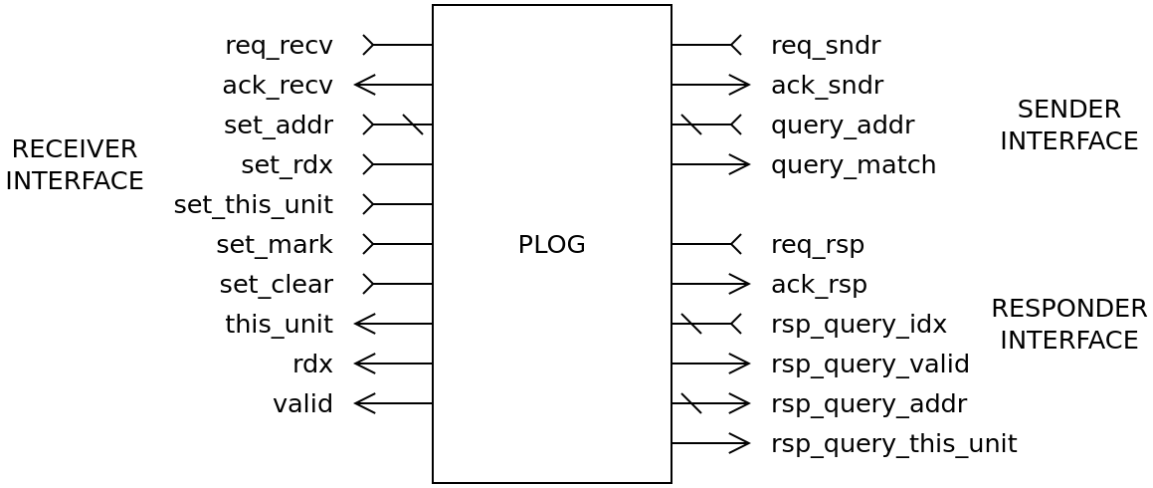


Fig. 32. Pending log component interface.

The second part of the PLOG interface is the Responder interface. The Responder component, in its idle state, iterates over the records within the PLOG. Per every iteration, each query is matched against the current content of the local cache memory. If a match is found and the matching cache line is in a valid state, the Responder attempts to respond to the request. The Responder crawls through the PLOG records infinitely circulating through the content. Similar to the Sender's interface, the Responder interface works based on the 4-phase handshake protocol. The Responder puts an index value on the rsp_query_idx line. The index is incremented and recirculated when it reaches the last record. The PLOG component upon being queried returns three pieces of information, a flag indicating whether the currently queried record is an ongoing valid transaction (rsp_query_valid) a memory address on the record (rsp_query_addr) and whether the queried request originates from this specific core (rsp_query_this_unit). All the information is then used by the Responder to query the local cache memory or to abandon this record and move to the next one.

The final third segment of the PLOG interface is the Receiver interface. Unlike the two previous parts, the Receiver segment is a read-write interface. The Receiver component is the only submodule in the controller that modifies the PLOG content based on the transactions observed on the bus. Similar to the previous two, this is also a 4-phase handshake type interface. The Receiver, when observes a data request Rd[X] or a response (Flush), places or deletes a record within the PLOG, respectively. When adding a new record, the Receiver sets the corresponding memory address (set_addr), a flag indicating whether it is an Rd or RdX request (set_rdx) with the value high for the later. Additionally, the Receiver records whether the request comes locally from the same controller unit where the component resides (set_this_unit). Only the unit that sends the request sets the set_this_unit flag. The writing occurs when the set_mark flag is set to high. To clear a record, the Receiver raises only the set_clear flag and gives a valid address on the set_addr line. When a record is removed the PLOG component populates the output signals this_unit, rdx and valid. The outputs stay until the next record clearing transaction happens and are used by the Receiver in its workflow.

The provided implementation is designed for a system with three cores and relies on the following assumptions. The key assumption is that the Sender module, when requesting data, pauses until a response comes in. Therefore a core can send only a single Rd[X] message at a time defining the required PLOG record count to the number of cores within the system, which in this case is three. In such a setup, a PLOG overflow cannot occur. The blocking behavior of the Sender also fits an in-order CPU core architecture. However, if the CPU core would be an out-of-order, superscalar core, additional measures and modification to the algorithm and the PLOG element become necessary and are discussed in section 6.2.

The PLOG component's internal design requires local arbitration between the Receiver, Responder, and the Sender components. Each component might require access

to the PLOG at an undetermined moment. It is impossible to predict at the level of STG design the access order and time of access to the PLOG. Requests from the Sender are the result of program execution, and requests from the Receiver are determined by the bus's communication pattern formed by the local Responder activity and other cores. The 3-way asynchronous mutex is used that lets through only one of the three components at a time to provide reliable non-deterministic arbitration. The arbitrated signals are the req_* signals initiating the 4-phase handshake.

The PLOG element is an internal utility component used by other units locally within the controller. The PLOG keeps track of any outstanding split-transaction requests on the bus allowing the cores to preserve information on any ongoing transactions. The PLOG component is crucial in implementing a coherent state of the system that uses the split-transaction bus model.

5.4.2 Receiver component design

The second component in the system is the Receiver module. The Receiver's role is to listen to the communication on the bus and adjust the local cache's content accordingly. Every time a message is placed on the bus, it is the Receiver that processes it. A bus message processed by the Receiver can originate from multiple different sources: messages coming from other cores, messages that originate from the same controller and messages coming from the main memory controller.

The interface of the Receiver, shown in Fig. 33 breaks down into five sections. The first segment, connecting to the local cache memory. The second segment, the arbiter interface to resolve access contention to the local cache memory between the components. The third segment is the PLOG interface of the Receiver. The fourth segment consists of a subset of signals necessary to listen and acknowledge messages on the bus. Finally, an utility interface that taps into the internal bus arbiter grant signals to determine whether a

bus message originates from the same core. When a grant signal from the internal bus arbiter is present for one of the internal components, it indicates that this component is currently accessing the bus. The internal bus arbiter then connects to the external system-wide arbiter such that the local arbiter will never grant access unless the external arbiter grants bus access first to this core.

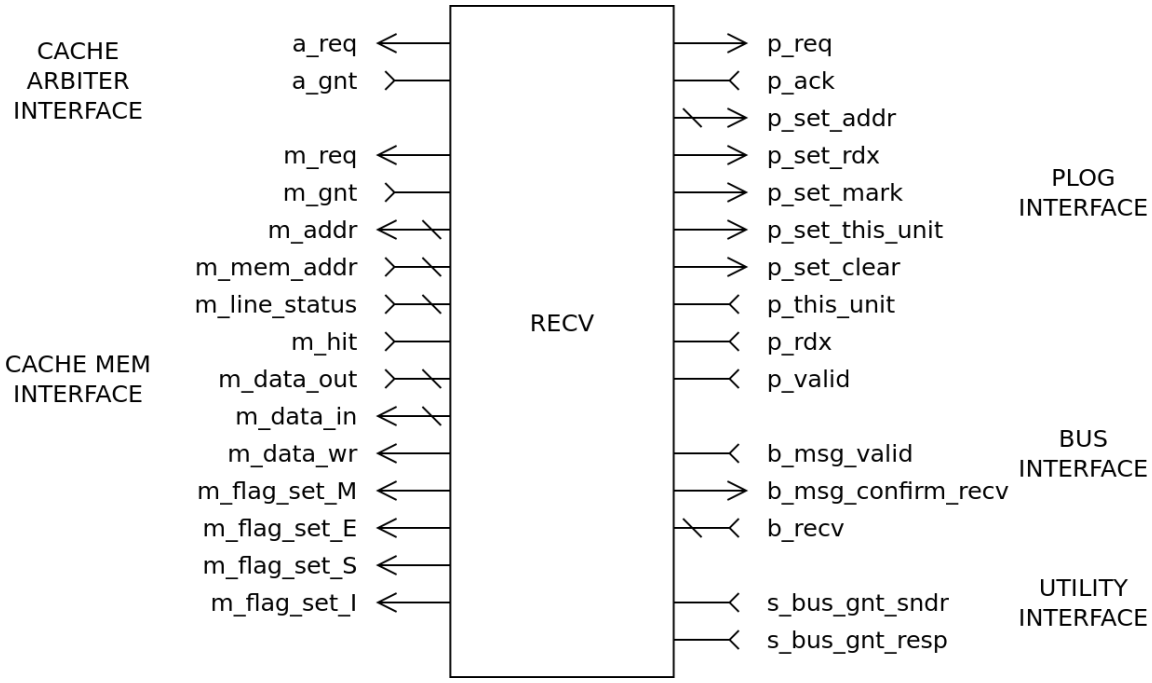


Fig. 33. Receiver component interface.

The Receiver component is a passive element type reacting to incoming messages on the bus. The component's main loop, shown in Algorithm 4, initially waits until a valid message appears on the bus. After a valid message shows, the Receiver begins to query the local cache using the address from the bus message. At the same time, the Receiver engages in interaction with the Pending Transaction Log element. After the initial local cacheQuery and the plogCycle operations finish, the Receiver examines all data and executes the writeCacheCycle operation that might conditionally write to the local cache

depending on the local memory state, the data within the PLOG element and bus message content.

Algorithm 4 Receiver module operation algorithm: main procedure.

```
1: procedure RECVMAIN
2:     loop
3:         waitOn(Valid message on the bus)
4:         plogCycle()
5:         lineStatus ← cacheQuery(msg.addr)
6:         writeCacheCycle()
7:         finalizeBusTransaction()
```

Looking closer at the plogCycle procedure shown in Algorithm 5. When a valid message appears on the bus, the Receiver component goes through the interaction with the PLOG component. State changes in the PLOG element are only caused by the Rd, RdX on Flush types of messages. The Receiver module generates a plogRecord dataset using the FSE module. The algorithm checks the type of operation, and if it is a read, it sets the PLOG record address to the value from the request. Then sets the RdXFlag to low for the Rd message. If the utility interface line indicates that the message comes from the same controller unit, it sets the thisUnit flag. Finally, the Receiver asserts the p_set_mark line and initiates a 4-phase transaction. The same flow occurs for the RdX request with the difference the RdXFlag is then set to high.

Different behavior occurs when the message is of a Flush type. The Flush operation indicates data coming as a response to a previous outstanding request. When the Receiver detects a Flush response, it removes the outstanding record from the PLOG as the transaction completes. The PLOG contains a complete record of outstanding transactions; therefore, it is guaranteed that a matching record exists and no additional checks are necessary. The Receiver sets the address and asserts the p_set_clear signal. After removal, the record's data remains at the outputs of the Receiver's section of the PLOG interface

and remains unchanged and stable until another clear operation. After the initial cache read cycle and the PLOG cycle, the Receiver has all the data necessary to enter the final step that could involve writing new information into the local cache memory.

Algorithm 5 Receiver module operation algorithm: pending log cycle subroutine.

```
1:  procedure PLOGCYCLE
2:      if msg.op is Rd then
3:          plogRecord.addr ← msg.addr, .RdXFlag(0)
4:          if msg.origin is this unit then
5:              plogRecord.thisUnit ← 1
6:
7:          plogInsertRecord(plogRecord)
8:      else if msg.op is RdX then
9:          plogRecord ← .addr(msg.addr), .RdXFlag(1)
10:
11:         if msg.origin is this unit then
12:             plogRecord.thisUnit ← 1
13:
14:         plogInsertRecord(plogRecord)
15:     else if msg.op is Flush then
16:         plogClearRecord(msg.addr)
17:     else
18:         Do nothing
```

The Receiver algorithm's final step is to analyze data obtained in previous steps, then determine if an update of local cache memory is necessary and perform it. The algorithm of the writeCacheCycle uses information from three sources, the message on the system bus, data from the initial cache read, and the data from PLOG output. The primary focus at this stage is to determine whether the message present on the bus affects the current state of the local cache memory. If it does, then to determine what in the local memory needs to be changed. The algorithm driving the writeCacheCycle is shown on Algorithm 6.

Algorithm 6 Receiver module operation algorithm: cache write subroutine.

```
1:  procedure WRITECACHECYCLE
2:      write ← false
3:      if msg.op is Rd or RdX then
4:          pass
5:      else if msg.op is Upg then
6:          if msg.origin is this unit, sender module then
7:              newLine ← .addr(msg.addr), .flag(Exclusive )
8:              write ← true
9:          else if lineStatus.hit is true then
10:             newLine ← .addr(msg.addr), .flag(Invalid )
11:             write ← true
12:     else if msg.op is Wb then
13:         if msg.origin is this unit, sender module then
14:             newLine ← .addr(msg.addr), .flag(Invalid )
15:             write ← true
16:     else if msg.op is Flush then
17:         if plogRecord.thisUnit is 1 then
18:             newLine ← .addr(msg.addr), data(msg.data)
19:             write ← true
20:             if plogRecord.flagRdX is 1 or msg.source is main memory then
21:                 newLine.flag ← Exclusive
22:             else
23:                 newLine.flag ← Shared
24:         else
25:             if lineStatus.hit is true then
26:                 if plogRecord.flagRdX is 1 then
27:                     newLine.addr ← .addr(msg.addr), .flag(Invalid )
28:                     write ← true
29:                 else if lineStatus.flag is Modified or Exclusive then
30:                     newLine.addr ← .addr(msg.addr), .flag(Shared )
31:                     write ← true
32:     if write is true then
33:         cacheWrite(newLine)
```

The primary piece of information involved in decision making is the operation type assigned to the bus message. In the case of a read or read-exclusive request, the Receiver alters the cache memory content. However, no change in the cache line status flag occurs

immediately when the Rd or RdX request is present. Instead, the Receiver at each core adjusts the flags and data during the response (Flush) message processing. The next case is the upgrade (Upg) message; it is an optimization that allows a core to promote its cache line from Shared status to Exclusive without the need for the RdX transaction. The condition is that the core already contains the line in its local cache. The Upg case shows how the Receiver uses the utility section of the interface. From the Receiver's perspective, if the origin of the Upg message is the same core where this instance of Receiver is located, then the local cache write begins during which the flag is set to Exclusive. Otherwise, if the message does not originates from the same core and the initial cache request shows cache hit, then a cache write occurs, and the corresponding line status becomes invalid.

The next case is the writeback (Wb) message type. Writeback occurs when a controller action results in new data being placed in the local cache under an index that is already occupied by a different address, for which the cache line is in the modified state. To prevent decoherence, the controller first sends the line in local cache memory out to the bus with the opcode Wb which is picked by the main memory. The Sender module sends the writeback message, while the Receiver captures the event and modifies the local cache accordingly. When the Receiver detects that the writeback message comes from the same unit, it invalidates the corresponding line in the local cache, freeing it for the next transaction. Otherwise, if the message comes from another core, it is ignored. It is guaranteed that a cache line owned by one core with status exclusive or modified implies the rest of the units if they have a matching copy, then that copy status must be invalid.

Finally, the fourth case is the Flush message type. Flush carries valid data in the data field, and it is a response for an outstanding data request, either Rd or RdX. Here the Receiver uses PLOG output data present after clearing the record. If the PLOG thisUnit flag indicates that this was the core that requested the data, then the message's origin field

is tested. If the Flush message is a response to an RdX request or comes from the main memory, then the new data is written to the local memory with an Exclusive status flag. Otherwise, if the incoming message is a response to Rd, and the origin is another core, then the data is written with the Shared flag. If the requesting core was not this core, then the result of the initial cacheQuery is tested. If there is a cache hit and the request was RdX, then the line is invalidated. Otherwise, if there were a cache hit and the request was not RdX, but the current local line status is modified or exclusive, then the flag is switched to shared. Finally, in other cases, nothing happens. Using the presented algorithm, the controller implements a portion of its functionality responsible for processing incoming bus messages.

The Fig. 34 shows an overview block diagram for the Receiver. The structure of the component is a relatively simple construct compared to the other key modules. It contains single speed-independent controller which CSP specification is shown in Appendix A. The component contains two FSE elements. The M1 FSE is responsible for the plogCycle function that assembles inputs to the PLOG element. Second is the M2 FSE, which is a multifunction type block. The M2's first task is to generate inputs to the local cache for the initial query, and the second task is the implementation of writeCacheCycle. The component uses an asynchronous register to preserve relevant data from the initial query beyond the first handshake with local cache memory.

The Receiver also uses a simple support module. The support module formats a subset of data-path coming from the local cache memory and is placed in the asynchronous register. The data consist consists of the hit flag and cache line MESI flag. Another part of the support element is the implementation of the signal gating. In this instance, the element gates the ack signal originating from the local cache. The gating signal is the grant signal from the internal local cache access arbiter. The gating, in this case, allows simplifying the speed-independent controller. The m_ack cache acknowledge line is

shared between all components, but the only receiving component is the one that initiated the 4-phase handshake with the local cache memory. A component must first be selected by the arbiter to be allowed to initialize the handshake; therefore, if the a_gnt signal is low, then the Receiver is not the recipient, and the m_ack must be prevented from reaching the speed-independent controller. The gating allows for simplification of the speed-independent controller taking out the need to handle the unexpected occurrence of m_ack. The Receiver element is one of four components working within the cache controller and is responsible for processing incoming messages on the bus, including messages originating from the same core. It is a passive element reacting to ongoing bus communication and the only entity that modifies the data in the PLOG element.

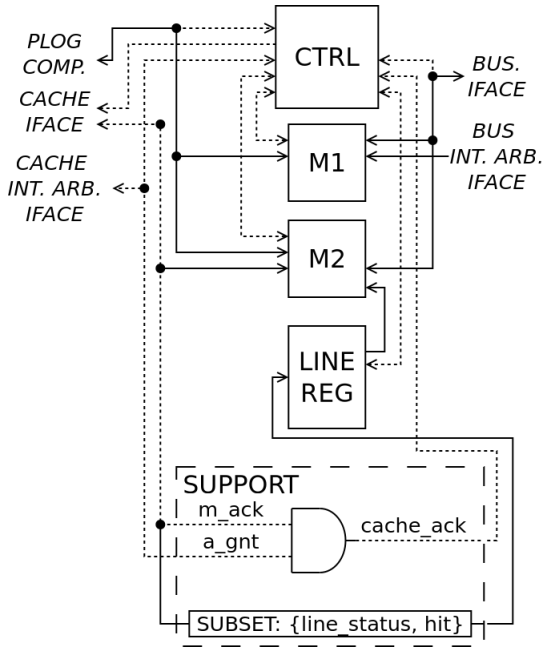


Fig. 34. Receiver internal block structure.

5.4.3 Responder component design

The Responder is the next key component in the cache coherency controller module. The Responder task is to respond to other requests for data coming from other cores.

When a core records an outstanding request advertised as the Rd or RdX operation, the request is logged in every core's PLOG element. The Responder constantly iterates over its local PLOG content and checks if it can respond to any active requests.

The Responder is an active type of asynchronous element. It works driven by its internal circuit that constantly increments and recirculates the indexing counter for lookup in the PLOG element. Depending on the returned PLOG record, the Responder initiates an exchange with the local cache and, in the end, the system bus. Responder's interface shown in Fig. 35 consists of three subsegments, PLOG portion that queries data under given index, bus interface, including the interface to the internal bus arbiter, and the cache memory interface also including internal cache arbiter.

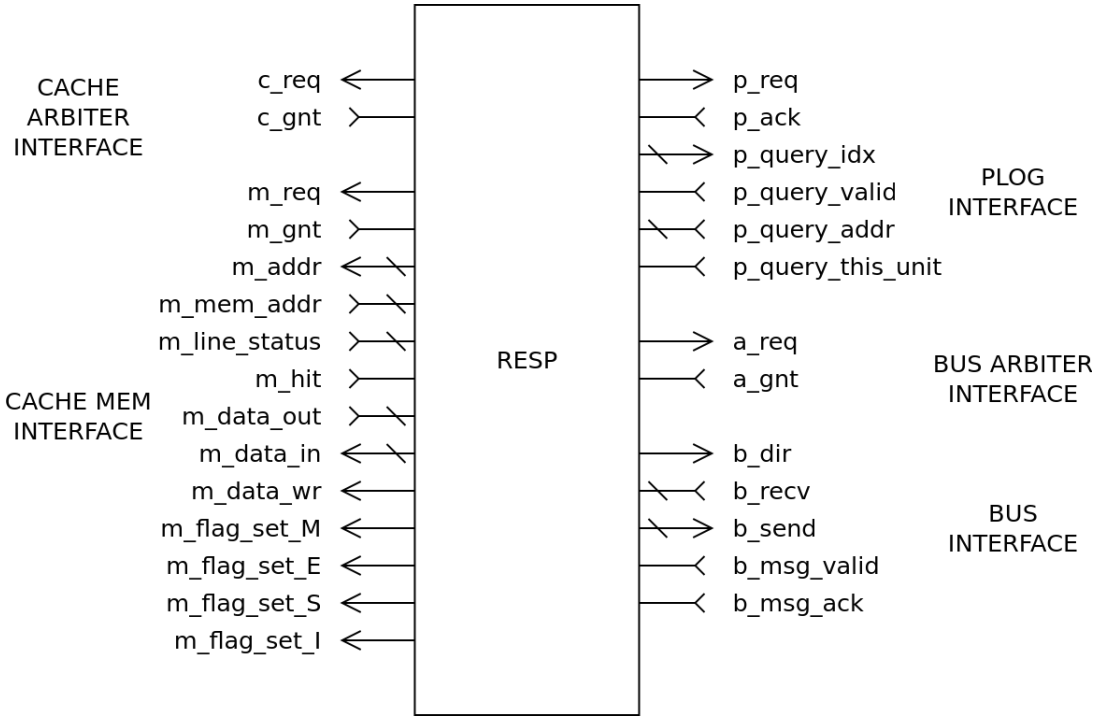


Fig. 35. Responder component interface.

The Responder's working principle as shown in Algorithm 7, revolves around crawling through the PLOG dataset in an infinite loop and testing whether an outstanding

transaction is ongoing and if this unit can respond to it. The entire workflow starts with the Responder generating the next index. Then the PLOG component is queried using this generated index. If the PLOG returns record that shows q_query_valid high and p_query_this_unit low the Responder heads to second step. Otherwise, the workflow starts again from the beginning, a new index is generated, and the process repeats.

Algorithm 7 Responder module operation algorithm.

```
1:  loop
2:      plogIndex ← nextPlogIndex()
3:      plogRecord ← plogAtIndex(plogIndex)
4:
5:      if plogRecord.active is 1 and plogRecord.thisUnit is 0 then
6:          cacheLine ← cacheQuery(plogRecord.addr)
7:
8:          if cacheLine.hit is 1 and cacheLine.flag not Invalid  then
9:              requestBusAccess()
10:             waitOn(Bus access granted)
    //Comment: Verify again the ability to respond by this unit in case the content
    changed while waiting on bus grant.
11:             plogRecord ← plogAtIndex(plogIndex)
12:             cacheLine ← cacheQuery(plogRecord.addr)
13:
14:             if plogRecord.active is 1 and plogRecord.thisUnit is 0 and
15:                 cacheLine.hit is 1 and cacheLine.flag not Invalid  then
16:                 prepareBusMessage()
17:                 placeMessageOnBus()
18:                 finalizeBusTransaction()
```

If the inspected PLOG record turns out to be valid, meaning it indicates a currently ongoing transaction and does not originate from this unit, then the Responder queries the local cache using memory address from the inspected PLOG record. The data returned from the cache is then analyzed; if there is a hit (m_hit is 1) and the cache line status is not invalid, this unit can respond to the request. Otherwise, the algorithm recirculates to the beginning.

If the Responder determines that it can respond to the request, it initiates the bus transaction by first requesting the bus access from the arbiter. After bus access is granted, the Responder again queries the PLOG, and the local cache then repeats previous checks. A repeated process is necessary for performance reasons and to prevent lockup. On the performance side, if the Responder would hold onto cache memory until the bus access grant comes, it would create a memory access-bound bottleneck for the local CPU core. The CPU core would be unable to perform any memory operation through the Sender component on the local cache memory and would have to stall even if the CPU operation applies to a completely different cache record. Also, holding onto the cache memory and PLOG prevents the Receiver from processing and confirming messages coming from the bus. For example, in a situation when the Responder managed to lock cache access, and it now waits for bus access grant. At the same time, another core gets the bus access instead and places a bus message. The core with the Responder module holding the cache would not be unable to confirm the current bus message, effectively locking up the system.

To avoid system lockdown and a performance bottleneck, the Responder duplicates the checking process. The Responder at first performs the preliminary checks before attempting the time-costly process of requesting the bus access. When obtained the bus access, the Receiver performs both checks on PLOG and local cache again to verify that the request is still pending and can still respond to it. It is possible that while the Responder awaits the bus grant, some other core already responded to the request, or the locally held cache line intended as response got replaced. If the second check fails, the Responder releases the bus and gets back to its idle operation. It is also possible that while Responder awaits the bus grant, the initial request got fulfilled and replaced by another valid request residing at the same record in PLOG. In this case, the workflow still performs correctly. Responder will correctly determine whether it can respond to the new request and perform the transaction if the test comes out positive.

After performing the final check, if the result from PLOG is a valid request originating from the different core and the result from the local cache is a memory hit with a MESI flag other than invalid, the Responder begins the final steps to send the message over the bus. Before the actual bus communication occurs, the Responder releases access to PLOG and the local cache. It is necessary to release the two resources before placing the message on the bus because the local Receiver must be able to process the message when it appears on the bus communication lines. First, all the necessary message data is latched inside Responder's internal memory registers. The message is then formed conforming to the bus message structure shown in Fig. 30. The message contains the corresponding memory address, the valid data content, and the cache line flag set to Flush.

Finally, the Responder places the message on the bus, and the transmission process begins. After all, cores confirm the message the transaction finishes. The Responder then releases the bus and withdraws the bus access request from the arbiter. The Responder partially delegates part of the processing of the placed response to the Receiver. The local Receiver handles the local side effects on the cache memory content. For example, a Responder responds to an RdX request. When the data is passed to another core, at the same time, in the case of RdX, the locally held cache line must be invalidated. The local Receiver handles this while the Responder waits for message confirmation. For other cores, the message is also handled by their Receivers, such as any other bus message. As the last step, the algorithm reiterates to the beginning and generates a new index, then the loop continues.

The Responder unit features a similar but more complex structure compared to the Receiver. An overview block diagram is shown in Fig. 36. The Responder module contains a single speed-independent controller. The controller CSP code is shown in Appendix A.3. First, the Responder's support module performs signal gating on two lines, the m_ack and the b_msg_ack. As in the Receiver case, the m_ack gating ensures that the

cache handshake confirmation goes to Responder's controller only when it is the Responder that talks to the L1 memory. Similarly, the b_msg_ack is only visible to the Responder's speed-independent controller if the Responder unit has an active bus grant. The support module also provides gating for the shared m_addr line, which travels from the Responder to the local cache. Since the m_addr is coming directly from the internal register, it is not guaranteed that the register's content would be zeroed-out to bring the m_addr to the default state. Because the m_addr merges coming from all major three components, a non zero idle value will interfere with the Receiver or Sender while communicating with the local cache memory.

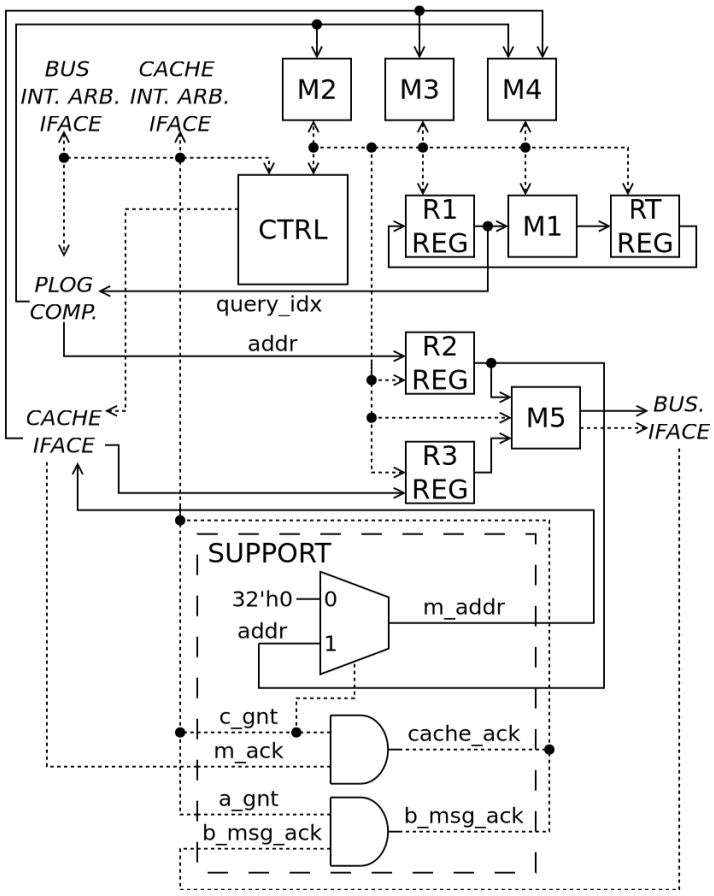


Fig. 36. Responder internal block structure.

The Responder component uses multiple FSE modules along with internal asynchronous registers. The first is the combination of M1 FSE and a pair of R1 and RT registers. The three units together are responsible for generating the indexes used to query the PLOG. The generation process is driven by the speed independent controller. The three components assemble into a feedback loop. The M1 FSE takes a stable signal from R1, which holds the current index value. Then the M1 FSE calculates the next index. The indexes go from 0 to 2 because there are a total of three cores in the system, and because of the assumed system configuration, there is only one outstanding split transaction request possible per core. The M1 performs the $n\_next = MOD(n+1, N)$ operation. The n is the current index, and N is the total number of cores in the system. Unlike in synchronous systems based on flip flops triggered on a clock edge, the M1 cannot feed directly back to R1. Doing so would destabilize R1 output during the 4-phase handshake, causing unstable input for M1, resulting in the system's malfunction. The temporary register RT is used to ensure the feedback loop stability. The M1 feeds to RT, and when RT latches the new value, the controller initiates write back to R1. The R1 also provides stable input to the PLOG part of the Responder's interface responsible for setting the query index.

Then the Responder component contains a set of decision type FSEs, the M2, M3, and M4. The M2 FSE performs the initial PLOG record check for the currently indexed record being valid and not originating from this core. The M3 verifies the local cache's output, testing if there is a cache hit, and the cache line flag is of value other than invalid. Finally, the M4 is a combination of M2 and M3 and is used after the Responder gets the bus access. All three FSE modules direct the processing flow for the speed-independent controller.

Finally, the component contains the M5 data-type FSE working with R2 and R3 registers. The R2 and R3 registers are used to latch relevant data from the PLOG and

local cache after the final check. The R2 register preserves the split-transaction request's memory address, and the R3 holds the response data content. Both registers feed to the M5 that generates the bus message.

Together all the elements consist of the Responder component that works as a part of the cache controller. The Responder is responsible for answering the data requests coming from other cores. Based on the data in the PLOG component and the local cache, the component assesses whether this core can respond to the given request, and if it can, the Responder attempts to send such a response. Responder units from different cores work in a competitive environment based on first-comes-first-served philosophy. If two or more cores can respond to the same request, the one who gets the bus access first sends the message. The other will identify an outdated request during the second check after it gets the bus access approved, which will cause it to abandon the process and loop back to the beginning.

5.4.4 Sender component design

The Sender component is the third and final component in the cache coherency controller. Sender fulfills two main tasks; it handles requests from the CPU core and posts messages on the system bus. Whenever there is a MESI message that is a result of CPU core interaction, it is up to the Sender to deliver it to the system. Sender handles posting request messages such as Rd and RdX, as well as Upg and Wb. On the CPU core end, the Sender implements the four request types coming from the core and serves as a bridge for the flow of the data between the CPU core and the local cache memory. The Sender is the central unit bridging the CPU core with the internal cache memory and the bus. It facilitates data movement while fulfilling requests coming from the CPU core itself. When necessary, the Sender posts messages on the bus, either to request data for the core or to advertise relevant data activity conforming to the MESI cache coherence algorithm.

The Sender interface shown in Fig. 37 breaks down into four groups: the local cache memory interface, including the internal cache arbiter, the bus interface also including the local bus arbiter interface, the PLOG interface, and the CPU core interface. The local cache interface is the standard type that covers all the signals, inputs, and outputs plus additional two lines to connect to the local cache access arbiter used by the internal components to resolve who is accessing the local cache memory. The PLOG interface is specific for the Sender module; Sender puts a memory address, and the PLOG component responds whether a valid record with a matching address is currently present. Then there is the core interface, which contains an address field for core requests, two data lines, one in each direction, and core request control lines. Finally, the bus interface which provides a connection to the system bus.

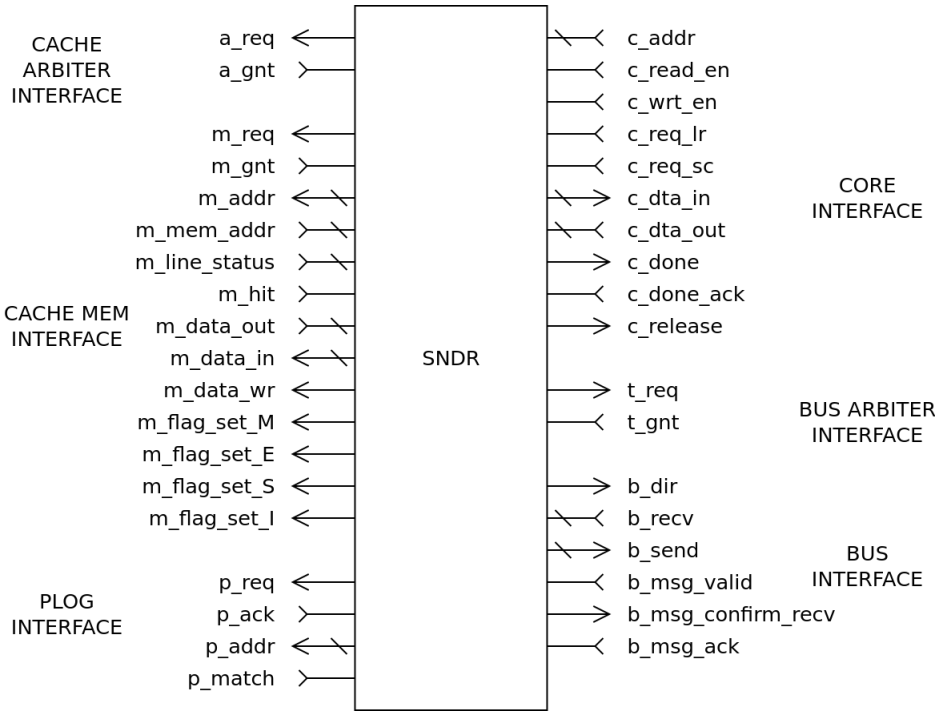


Fig. 37. Sender component interface.

The Sender is the most complex of all four components within the controller design as it bridges two workflows together; the CPU core side and the bus side. Due to its complexity, the Sender component uses multiple speed-independent controller circuits instead of one and extensively utilizes signal gating through the support element. The multi speed-independent controller setup specifies one controller that handles the CPU core requests and three controllers for bus communication.

After receiving a request, the CPU core interface controller first accesses the local cache to determine if it can finish the transaction immediately. Then the path split between the read operation, write operation, and bus transaction. The read and write actions can be completed immediately, but the bus transaction requires communication over the system bus. If a bus transaction is necessary, the CPU interface controller delegates the workflow to the bus main-transaction controller that handles the bus communication workflow. The main controller further delegates tasks to two utility controllers, the PLOG-check controller and the bus-communication controller. The PLOG-check utility controller ensures memory coherency discussed later while the bus-communication controller handles all the steps involved in sending a message through the bus.

The presented implementation of the cache controller supports the basic RISC-V in-order pipeline. In this design, when a memory stage is active, the core stalls until data is available again. Following the design choice, the cache coherency controller's portion that delivers data to the CPU core must implement the stalling functionality. Excluding the situation when the Sender awaits an arbiter bus access grant, the stall can occur in two cases. The first case happens when the message could not be posted on the bus at this time, while the second case is when the Sender awaits data arrival.

The case when data could not be posted on the bus at this time occurs when a conflicting request transaction is currently pending. The Sender component detects the situation by querying the PLOG component, and if the p_match line becomes high as a

result of the query, this means that another conflicting split-transaction read request is already taking place, and the Sender must wait until it completes. The use of a PLOG check in the Sender component workflow prevents multiple data requests from existing simultaneously in the system.

One of the requirements for memory coherency in the multi-core system states that all the cores must observe the same order of transactions on the bus [12]. Allowing multiple requests to stack up into a single one breaks this rule. Stacking makes it indistinguishable, which request came first and which later. Knowing the order of transactions is essential in the case of Read-Exclusive (RdX). When RdX completes, all cores except the requesting must invalidate the corresponding cache line. Compressing two such requests together prevents the core from determining which core has the Exclusive state and invalidates its copy of the cache line. A similar scenario applies to an Rd request that is satisfied by the main memory. In this case, the flag also becomes Exclusive since only the main memory and a single core contains the cache line. However, if two cores would request the data simultaneously, the correct flag should have been Shared. To preserve the system's coherent state, the Sender queries the PLOG component first, and if there is an address matching record, the Sender pauses until such transaction clears.

The second case takes place when the Sender stalls as the controller itself await a response for Rd or RdX message. When the transaction completes, the Sender resumes operation, eventually delivering data to the CPU core. The necessity to wait for a response implies that the Sender must actively react to the incoming message events and confirm them. In asynchronous systems, all signal transitions must be confirmed to avoid race conditions [9]. The Sender, together with the Receiver, must participate in the bus transactions and handshake. Unlike the Responder circuit, which did not need to know about external incoming messages because of the bus arbitration and second check that guaranteed correctness, the Sender component actively receives b_msg_valid and

confirms bus messages regardless of the state it is in. The need for signal confirmation results in increased complexity of the speed-independent controllers and the supporting signal-gating logic. The Sender's design approach shows how the divide-and-conquer strategy helps distribute the complexity among multiple elements and keep the model maintainable. The Sender component's main execution loop splits between the core interface controller and the bus-transaction main controller. In its idle state, the bus controller confirms incoming bus messages without any other action in an infinite loop unless a request from the core-interface controller interrupts it.

The CPU core interface is the initial point of contact for the CPU core. It directly receives and processes the memory access requests. It is a passive type of asynchronous device initially waiting idly until a request from the CPU core comes in. The algorithm for the CPU core interface is shown in Algorithm 8 and Algorithm 9. When a request comes, the Sender queries the local cache memory using the request's address. Next, the Sender begins to analyze the request. The first decision point is based on the type of request. Depending on the request type, the Sender performs specialized checks and executes response action. Suppose the request is a memory-read request; the component tests if there is a cache hit, and the flag is not invalid. When both cases are true, this indicates valid data to read. The Sender then prepares a response for the CPU core and executes the read action. Otherwise, the mmTransaction action executes that performs the bus transaction. In the second case, if the operation is the memory write, then the core tests whether there is a cache hit and the cache line flag is either Exclusive or Modified. If the condition check passes, new cache line data is formed, and the write action executes. However, if the condition test fails, the core must first obtain exclusive access to the memory region in question, which results in scheduling the mmTransaction action.

Algorithm 8 Sender operation algorithm: core interface routine (pt.1/2).

```
1:  procedure COREINTERFACE
2:      loop
3:          waitOn(New request from the core)
4:          coreReq ← coreRequest()
5:          EXEC_P1: cacheLine ← cacheQuery(coreReq.addr)
6:
7:          if coreReq.op is Read then
8:              if cacheLine.hit is 1 and cacheLine.flag not Invalid then
9:                  coreResp.data ← cacheLine.data
10:                 action ← read
11:             else
12:                 action ← mmTransaction
13:         else if coreReq.op is Write then
14:             if cacheLine.hit is 1 and (cacheLine.flag is Exclusive or Modified ) then
15:                 newLine ← addr(coreReq.addr), .data(coreReq.data), .flag(Modified )
16:                 action ← write
17:             else
18:                 action ← mmTransaction
19:         else if coreReq.op is LoadReserved then
20:             if cacheLine.hit is 1 and (cacheLine.flag is Exclusive or Modified ) then
21:                 coreResp.data ← cacheLine.data
22:                 action ← read
23:             else
24:                 action ← mmTransaction
25:         else //Comment: coreReq.op is StoreConditional
26:             if cacheLine.hit is 1 and (cacheLine.flag is Exclusive or Modified ) then
27:                 newLine ← addr(coreReq.addr), .data(coreReq.data), .flag(Modified )
28:                 coreResp.data ← SC_SUCCESS
29:                 action ← write
30:             else
31:                 coreResp.data ← SC_FAILED
32:                 action ← read
```

The third case is the load reserved request. The atomic pair LR and SC is implemented using the cache coherency mechanism. An Exclusive or Modified flag indicates implicitly that no other core altered the data under the given address. Effectively, the presence of an Exclusive or Modified state triggered by LR and preserved until SC

execution implies an atomic, unaltered memory state between the two instructions. During the load reserved request, the Sender component tests whether there is a hit, and the cache line has either an exclusive or modified state. If the test succeeds, the current data under the address returns to the CPU core; the controller schedules the read action. Otherwise, if the test fails, the controller must first obtain exclusive access to the data resulting in the mmTransaction action.

Algorithm 9 Sender operation algorithm: core interface routine (pt.2/2).

```
33:     if action is read then
34:         respondToCore(coreResp)
35:         finalizeCoreTransaction()
36:     else if action is write then
37:         cacheWrite(newLine)
38:         //Comment: If op is SC then sends transaction status back to the core
39:         respondToCore(coreResp)
40:         finalizeCoreTransaction()
41:     else //Comment: action is mmTransaction
42:         busMsg ← createBusMessage()
43:         transactionMain(busMsg)
44:         goto EXEC_P1
```

Finally, if the operation is store conditional, the controller tests for a cache hit and whether the cache line is in an Exclusive or Modified state. On success, new data is written to the local cache memory, and the controller prepares response SC_SUCCESS for the CPU core. Otherwise, if the test fails, the controller delivers the SC_FAILED response instead. The store conditional operation failing the conditional check means some other core took away the exclusive access and potentially modified the data effectively, interrupting the atomic exchange.

The first part of the CPU core interface functionality revolves around receiving the request from the CPU core, analyzing it, and preparing either: a response to the core, new data for the local cache, or both together in the case of store conditional operation. In the

second part, the segment responsible for communication with the CPU core schedules an action: either a read, a write, or mmTransaction. The action segment modifies and writes the data. First, the workflow recognizes which action takes place. If it is the read action, then the controller initiates the response transaction with the CPU core then finalizes the exchange. During the write action and additional local cache write cycle occurs then the controller responds to the core. During write action, the response functionality is needed to implement the store-conditional workflow, in which case the instruction returns its status of execution. Finally, if the mmTransaction is requested, the Sender component prepares the bus message; early preparation allows for early release of the cache memory lock that improves the performance. Then the core interface subsystem delegates the bus-send operation to the bus-transaction controller and stalls until the transaction completes. After the transaction finishes, the local cache memory contains the updated state.

When finished with the workflow cycle, the algorithm loops back to its idle state for the read and write operations. However, for the mmTransaction, the flow jumps back to the initial cache query step, and all the data is re-evaluated. The repeated evaluation is necessary to cover the remaining steps required to complete a CPU request, which at first resulted in a bus transaction. It is also possible that while the Sender awaits its turn on the bus, an external communication invalidated the currently generated bus message, and the entire request must be repeated. For example, the Sender attempts to send an Upg message as a part of the load-reserved operation. The current cache-line status is shared, but the controller still needs an Exclusive status to satisfy the LR condition. Before the Sender's turn on the bus, another RdX came from another core and was completed. Now the local cache line status is in the invalid state, and the most current version most likely changed in the other core. Now in order for LR to complete and deliver the most current data to the CPU core, the Sender must abort the current Upg operation and retry, but this

time with the RdX instead. Another corner case is the writeback case. If a controller requests data for a cache line that placement in the local cache collides with another line that on top of it has the Modified flag, the Sender must first write the modified data back to the main memory to preserve consistency. In this case, the Sender first performs a bus transaction using the Wb operation. After Wb finishes, the line modified status changes to invalid. Now the Sender can put the Rd or RdX request to get the initially requested data.

During the mmTransaction action, the Sender generates the bus message, which content strongly depends on the type of request and current state of the system. An algorithm synthesizing the bus message is shown in Algorithm 10. The algorithm assumes previous conditions shown in Algorithm 8 failed and led to the mmTransaction action. Therefore the message generating procedure must be treated as an extension to the core-interface controller workflow, not a separable component. The order of execution of the conditional tests is also essential and must not be changed.

The first test states that if mmTransaction was invoked and there were a cache miss and the flag value under the cache line that caused the miss represents a Modified state, then whatever the original request is, it will replace this line. Therefore a writeback operation is first necessary to preserve the most recent value. The bus message is set with the address from the cache line in the memory instead of the address in the request; also, the data is taken from the local cache line, and the operation becomes Wb indicating writeback. In any subsequent case, if the first check fails, it means that data in the local cache memory at this particular record is safe to overwrite. In the second case, if the operation was a memory read, this directly leads to forming an Rd bus request with the address from the CPU core request and Rd operation type.

Algorithm 10 Sender operation algorithm: routine generating bus message.

```
1:  procedure CREATEBUSMESSAGE
    //Comment: Line that is modified and a miss at requested address.
2:      if cacheLine.hit is 0 and cacheLine.flag is Modified then
3:          busMsg ← .addr(cacheLine.addr), .data(cacheLine.data)
4:          busMsg ← .srcMM(0), .op(Wb)
5:      else if coreReq.op is Read then
6:          busMsg ← .addr(coreReq.addr), .data(X)
7:          busMsg ← .srcMM(0), .op(Rd)
8:      else if coreReq.op is Write then
9:          if cacheLine.hit is 1 then
10:             busMsg ← .addr(coreReq.addr), .data(X), .srcMM(0)
11:             if cacheLine.flag is Invalid then
12:                 busMsg.op ← RdX
13:             else //Comment: Must be cacheLine.flag is Shared
14:                 busMsg.op ← Upg
15:         else
16:             busMsg ← .addr(coreReq.addr), .data(X)
17:             busMsg ← .srcMM(0), .op(RdX)
18:     else //Comment: Must be coreReq.op is LoadReserved
19:         if cacheLine.hit is 1 then
20:             busMsg ← .addr(coreReq.addr), .data(X), .srcMM(0)
21:             if cacheLine.flag is Invalid then
22:                 busMsg.op ← RdX
23:             else //Comment: Must be cacheLine.flag is Shared
24:                 busMsg.op ← Upg
25:         else
26:             busMsg ← .addr(coreReq.addr), .data(X)
27:             busMsg ← .srcMM(0), .op(RdX)
28:     return busMsg
```

In the third case, if the operation is a memory write and it failed, resulting in mmTransaction, then it means that the current record available to this core does not have exclusive or modified status; therefore must be upgraded, causing invalidation in other cores to preserve coherency. If the requested current cache line is a hit, the correct data is there but marked with a flag that does not indicate an exclusive copy; then the flag must

be tested further. If the flag value is Shared, then the optimized Upg message can be sent to invalidate other shared copies and obtain Exclusive status. However, if the flag of the local copy indicates Invalid, then following the algorithm [12], an RdX must be sent to obtain an Exclusive and up-to-date copy. Similarly, when there was a cache miss, an RdX message must be sent as well. In the case of the RdX request, the bus message is composed of the address coming from the CPU core request and the RdX flag. Note that for this specific implementation that uses a single-word cache line size, it is redundant to send RdX at all to obtain a copy of the data since the CPU core will overwrite the entire line regardless. However, in realistic systems, the cache line size is more than one word; the actual write would modify just a portion of the data. Therefore an exclusive copy must be obtained, and this implementation mimics the MESI behavior in realistic systems.

The final fourth case occurs when the request is the load reserved operation. From the perspective of generating the bus message, the memory write and load reserved cases are identical. If mmTransaction got triggered during LR, then either there was a cache miss or the data is there, but its MESI status is not exclusive. Same for the memory write case, the cache line access can be upgraded with Upg request if the current flag value indicates a shared state; otherwise, an exclusive copy must be obtained through split-transaction exclusive data request RdX.

After generating the bus message, the CPU core interface-controller of the Sender component delegates the task of sending the bus message to the Sender's main-transaction controller and stalls then itself. From the conceptual perspective, the design of the main transaction controller is trivial. The conceptual algorithm shown in Algorithm 11 shows the overview and the Fig. 38 the STG of the main-transaction controller's operation. When called, the main-transaction controller right away delegates the task to the PLOG-check controller. After the PLOG check finishes, there is a decision step based on the result of the check. If the PLOG task clears the main to proceed, then another

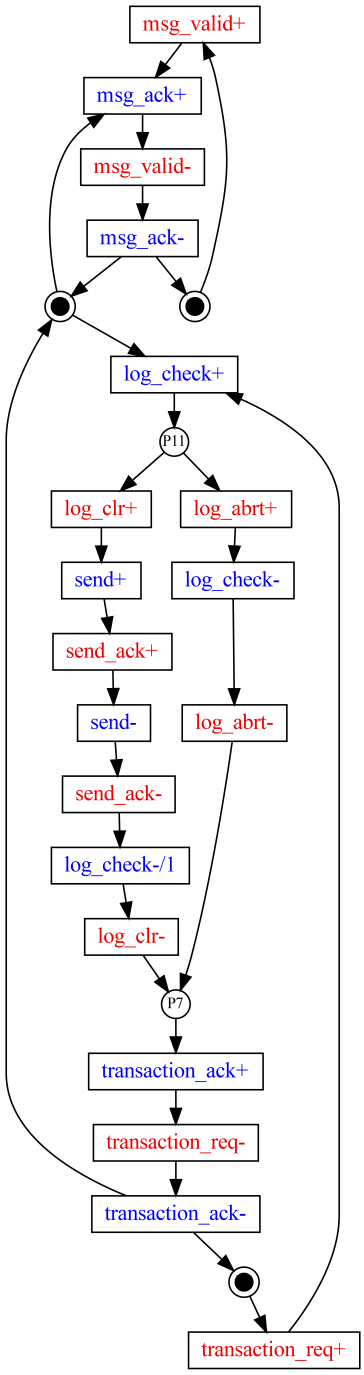
delegation occurs, this time to the bus-communication controller. Otherwise, if the PLOG task indicates procedure abort, then the controller finishes its task, and the execution goes back to the core-interface controller. The abort case could occur when there was a message collision detected.

Algorithm 11 Sender operation algorithm: main bus transaction routine.

```
1: procedure TRANSACTIONMAIN
2:     plogCycleResult ← plogCycle()
3:
4:     if plogCycleResult is clear then
5:         transactionBusCycle()
6:     else //Comment: plogCycleResult is abort
7:         Do nothing, Main algorithm will loop and retry.
```

The main-transaction controller's function splits into two roles: cover the ability of the Sender component to confirm bus messages in an idle state and supervise the message transmission process. The main transaction controller confirms messages that are not of interest to the Sender component at this time. The component must react to the bus communication regardless of whether the module currently takes an active part in incoming communication or not. The Sender module heavily relies on signal gating, enabling and blocking control signals depending on its internal state and in a manner that will not cause a race condition.

In the main-transaction controller case, there are two competing signals: the transaction_req and msg_valid that must be treated in a mutually exclusive manner. In the idle state, when any external origin message appears on the bus, the controller would see msg_valid transition, which it acknowledges through emitting msg_ack without any additional actions. However, when the transaction_req comes in, the whole message sending process starts. If any external message appears after transaction_req then instead of the main-transaction controller, the message will be acknowledged by either the



INPUTS: msg_valid,transaction_req,log_clr,log_abrt,send_ack
OUTPUTS: msg_ack,transaction_ack,log_check,send

Fig. 38. The main transaction controller STG.

PLOG-check or the bus-communication controller. Both competing transitions could not appear at the same time as it will lead to a metastable state. The log_check signal can take away token, preventing the msg_ack from appearing. Simultaneously, the external message gets confirmed by the other controller causing msg_valid to disappear without confirmation causing undefined behavior in the main-transaction controller's logic. To prevent the hazard condition, both the msg_valid and the transaction_req are delivered first through the asynchronous mutex component ensuring only one line will ever be active at the time. The msg_valid is a synonym of the b_msg_valid input after it passes Sender's internal mutex. The Sender component significantly relies on signal gating to implement a proper incoming message confirmation mechanism.

Continuing with the message sending procedure flow. The transaction request is first delegated to the PLOG-check controller. Because of the requirement of coherent systems and the design assumptions of this implementation, it is forbidden for two transactions with the same address to occur simultaneously. Such a situation can happen when, for example, there is an outstanding Rd request from one core, and some other core tries to send another message with the same address. To prevent colliding messages, the Sender module checks for any outstanding requests with a matching address, using the PLOG module. The workflow of the PLOG-checking procedure is shown in Algorithm 12 and Fig. 39.

In the simplest case, with no matching record, the PLOG-check controller sends back the clear signal to the main-transaction controller allowing it to proceed. However, if a matching record exists, the Sender module must stall until the previous outstanding transaction concludes. In this case, the controller listens to any incoming bus messages. When a message arrives, the address is compared; if incoming the message's address does not match the colliding transaction's address, the PLOG-check controller acknowledges the message, loops back, and continues to wait. Eventually, when the addresses finally

match, the PLOG-check controller sends back the abort signal to the main-transaction controller and finishes its workflow. Aborting the operation is necessary as the previous transaction might have changed the cache-line state to the point at which it must be analyzed again, potentially generating an updated bus message.

Algorithm 12 Sender algorithm: bus transaction routine, pending log cycle.

```
1:  procedure PLOGCYCLE
2:      //Comment: Present in log means outstanding transaction
3:      if coreReq.addr in pending log then
4:          //Comment: Wait until matching response arrives finishing transaction
5:          loop
6:              waitOn(Valid message on bus)
7:
8:              if msg.addr equals busMsg.addr then
9:                  //Comment: Transaction finalized, force sender to retry
10:                 finalizeBusTransaction()
11:                 return abort
12:             else
13:                 finalizeBusTransaction()
14:     else
15:         return clear
```

Since the PLOG-check controller inspects and acknowledges bus messages, it takes an active part in handling the message acknowledging process. For this controller, the gated version of the b_msg_valid is the gmv signal. The gmv signal results from an AND operation on b_msg_valid indicating a new message on the bus and the transaction_req after it leaves the MUTEX. Therefore when the mutex lets through the transaction_req, it is guaranteed that msg_valid will no longer appear in the main-transaction controller. Instead, when b_msg_valid shows, it will get through the AND gate and be visible to the PLOG-check controller as a signal named gmv.

Looking at Fig. 39 raises another question about the hazard safety of the circuit. Note that there is a certain amount of time between the occurrence of the in_log_n+ signal that indicates no message collision and the future access grant of the bus access for this unit.

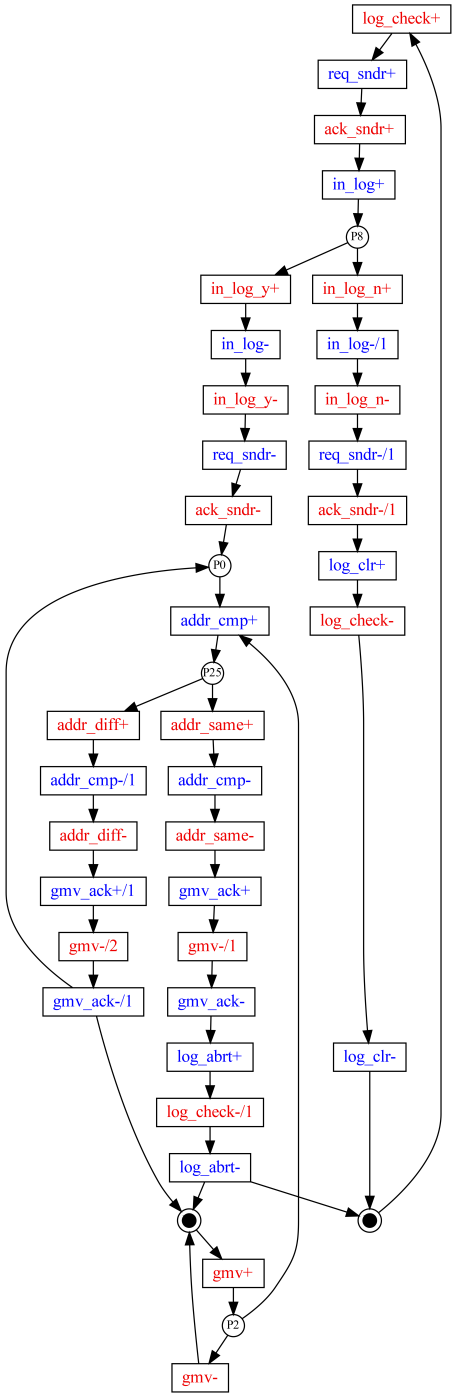


Fig. 39. The STG for the PLOG-check controller.

What happens if, after no collision determination, there will arrive an external Rd request with a colliding address? The answer is not immediately obvious from the diagram, but the circuit is secured against such a case. The PLOG-check controller only acknowledges messages if there is a collision detected, and as a result, the controller waits for the colliding transaction to end so it can reset the sending process. In any other case, the PLOG-check controller contains a mechanism allowing it to withdraw itself from the message acknowledgment process. The mechanism is implemented by the loop at the shown bottom of Fig. 39 that circulates between gmv+ and gmv-. The loop allows the speed-independent circuit to work correctly while the actual acknowledgment is done elsewhere. Focusing back on a potentially hazardous Rd signal, if the PLOG-check controller does not confirm the message, the transaction is held until the bus-communication controller takes over. Moreover, the bus-communication controller contains functionality that analyzes any incoming foreign messages for collisions and aborts if such one occurs.

The bus-communication controller is the final component driving the Sender's bus message communication functionality with workflow shown in Algorithm 13, and it is responsible for obtaining access to the bus and message delivery tasks. The bus-communication controller performs three main functions. First, it delivers the outgoing message to the system; second, it waits for the response if it was a split-transaction request; and third, identifies any colliding requests that would invalidate the message it is tasked to send. As the first step after activation, the controller places the bus access request to the arbiter. Then it waits for either the bus grant or an incoming message from the bus. When a foreign-origin bus message comes first, its address is tested. If the incoming bus message collides with the awaiting outgoing one, the controller acknowledges the incoming message then withdraws the bus-access request and aborts. Then the core interface controller restarts the process. Otherwise, if the bus grant

comes indicating the controller wins access to the bus, the controller sends the message through. After delivery, if the message op is Rd or RdX, the controller stalls until the response comes in; otherwise, the process finishes and goes back to the core-interface controller. If stalled, the controller awaits and acknowledges any incoming messages, and when the address matches, it finishes the process.

Algorithm 13 Sender algorithm: bus transaction routine, bus transmission.

```
1:  procedure TRANSACTIONBUSCYCLE
2:      requestBusAccess()
3:      EXEC_P2: waitOn(Bus Grant or Valid Bus Message)
4:
5:      if Bus Grant then
6:          placeMessageOnBus()
7:          finalizeBusTransaction()
8:          if busMsg.flag is Rd or RdX then
9:              //Comment: Stall sender module if split transaction occurs
10:             EXEC_P3: waitOn(Valid Bus Message)
11:             finalizeBusTransaction()
12:
13:             //Comment: Loop until matching response arrives
14:             if msg.addr not equals busMsg.addr then
15:                 goto EXEC_P3
16:     else //Comment: Valid Bus Message
17:         if msg.addr equals busMsg.addr then
18:             //Comment: Message invalidated by overriding transaction, restart
19:             finalizeBusTransaction()
20:         else
21:             finalizeBusTransaction()
22:             goto EXEC_P2
```

The Sender module is the most complex of all four components. As seen in Fig. 40, It contains four separate controllers that work together and are heavily supported by the support element that delivers signal gating. The support module includes logic that formats input to the line register that preserves the queried cache line for the purpose of generating a bus message. The support module also provides merging of all the bus ack

signals into a single output b_msg_confirm_recv and all the core request lines into a single mem_op that serves as a control signal to the core-interface controller. Additionally, the support module provides standard signal gating for common signals such as b_msg_ack that is gated by the transaction_req and the cache_ack that feeds to the m_ack input of local cache memory. Finally, the module provides the gated (gmv) versions of b_msg_valid. The gmv feeding to the bus-communication controller only activates when there is a valid message on the bus (b_msg_valid), and the Sender requested a bus transaction (transaction_req) and the bus access is requested (t_req) but not granted (~t_gnt) by the bus arbiter. This gmv signal is used to process and acknowledge any bus messages that appear before the unit is granted bus access. The second gmv2 also feeds to the bus-communication controller, and it is used in awaiting a response in the case when a split-transaction message was sent. The gmv2 activates when there is a valid message on the bus (b_msg_valid), and Sender requested a bus transaction (transaction req), and the req_type_y transition is active, indicating controller awaiting messages. The other version of gmv that feeds to the PLOG-check controller activates only when there is a valid message on the bus (b_msg_valid), and Sender requested a bus transaction (transaction req).

The Sender component uses six FSE units. The M1 FSE formats the bus message. Then the M2 used by the PLOG-check controller resolves the branch deciding whether a record is present in PLOG. Next, the M3 compares addresses from CPU-core requests and bus messages while the PLOG-check controller awaits split-transaction completion. Then, the M4 performs the same function as M3 but for the bus-communication controller. The M5 handles the output bus interface, and M6 performs condition resolution checking whether the sent bus message is a split-transaction request or not. The Sender component also uses a line register to hold the cache line extracted from the local cache memory using the CPU-core request address. Then, there is a mutex element present that resolves

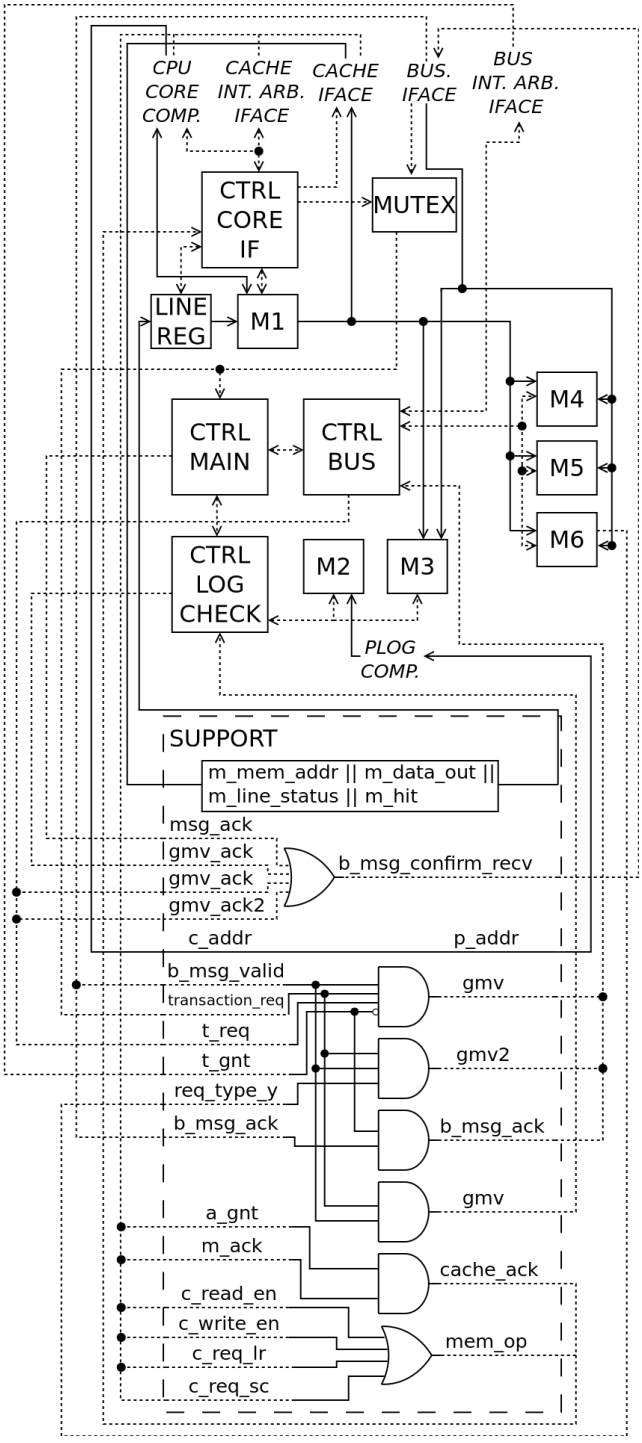


Fig. 40. Sender component internal block structure.

contention between b_msg_valid coming from the outside environment and the transaction_req that enables the sending functionality, including cutting off the direct b_msg_valid replacing it with the gated versions.

5.4.5 Controller module

The Controller module is the top element connecting all functional components. As shown in Fig. 41, the module provides internal connections as well as the support logic. The Controller module instantiates all four primary components: PLOG, Receiver, Responder, and the Sender. Also, the module contains two arbiters. The internal cache arbiter decides which one Sender, Receiver, or Responder gets access to the local cache memory. The internal cache arbiter is implemented as three input three output asynchronous mutex. Whichever element request passes through the mutex that one is granted the access. The second is the internal bus arbiter. Its structure is presented in the publication [17]. The local bus arbiter is, in a way, an intermediate element involved in the bus arbitration. It takes requests from Sender and Responder circuits, but it must be first selected by the external bus arbiter to emit a grant signal. When receiving a request, the local bus arbiter sends a request itself to the external bus arbiter. The external bus arbiter is the central system arbiter that decides which core can access the bus. Then when selected by the external bus arbiter, the local version emits a grant access signal. If it is the case that Responder and Sender compete with each other for the bus, then the internal arbiter decides which one gets the access.

The support element provides signal merging and additional support logic. The entire local cache-memory input interface signals coming out of all three main components are merged. This situation shows why it is important for the FSE modules to return to their default 0 value at the end of a handshake. A faulty unit can forcefully override valid signals coming from another component. Merged are also the b_send lines carrying the

data from the cache Sender and the Responder components to the bus and the b_dir signals that serve as control signals. When the b_dir signal has value BUS_SEND it causes the b_msg_valid to activate which initiates bus transfer.

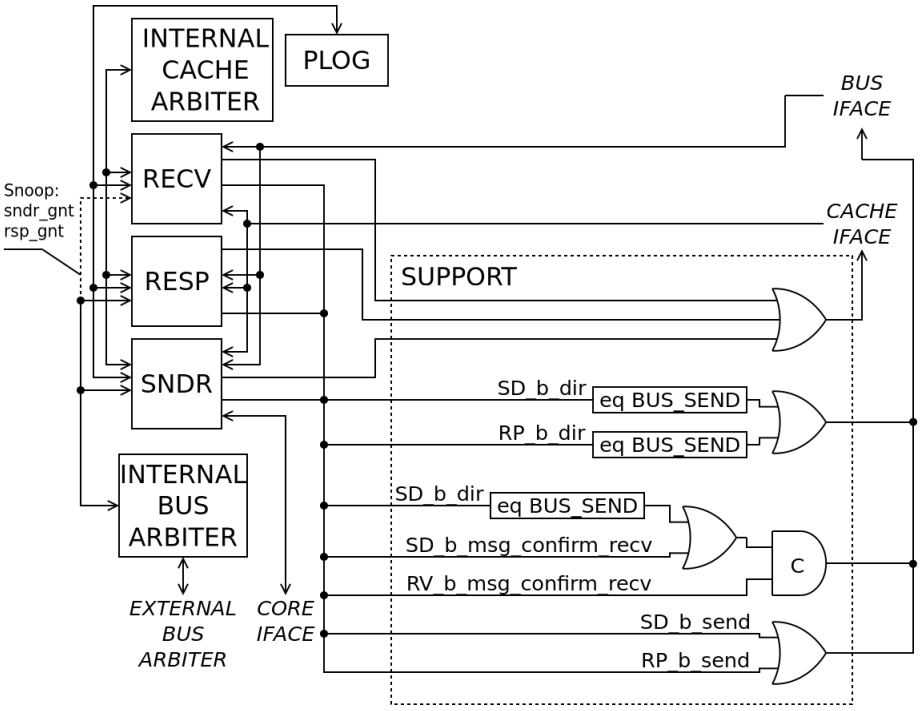


Fig. 41. Controller internal block structure.

Finally, the support module provides combined acknowledge functionality. The Sender component acknowledges a bus message in two ways. The b_msg_confirm_recv is placed high when the Sender acknowledges the message of external origin. When the message comes from this Sender, the b_dir of a value BUS_SEND is used to satisfy multiple-place confirmation. The multiple source confirmation for a small number of elements is most efficiently implemented by the C-element. The output's acknowledge signal goes high only when all components acknowledge the message and then goes back low when all components are ready to finish the handshake putting their ack signals low.

The Controller is the top module connecting all functional components and the interface connections to the environment. The top module provides an additional element in the form of arbiters to support the components. Finally, the Controller module provides the support element handling the signal management and supplying support logic. All elements together implement the cache coherency controller circuit providing the MESI algorithm and split-transaction bus support.

5.5 Reference synchronous design

The presented work introduces an additional reference synchronous implementation of the controller to evaluate the methodology's effectiveness. The purpose of the synchronous design is to provide a reliable baseline that allows for obtaining meaningful results. The comparison aims primarily to evaluate the proposed methodology's ability to generate circuits that offer competitive properties that could justify its use in practical applications.

The synchronous version to provide a reliable reference point follows the same design structure and executes the same algorithm as the asynchronous model. The same as the asynchronous, the synchronous design contains the four primary components: PLOG, Receiver, Responder, and Sender, all executing the same high-level algorithm as the asynchronous counterpart. The design also uses two internal arbiters, one for internal cache access and the other for the bus exactly as the asynchronous counterpart. Moreover, the synchronous version avoids aggressive optimizations at the algorithmic level to prevent potential unwanted skew of the results. However, the synchronous model introduces implementation level optimizations to prevent results bias in the other direction. The goal is for both models to closely resemble each other to allow for the evaluation of the proposed methodology at the same time to retain properties characteristic to the methodology they are designed in. Unlike the asynchronous design,

the synchronous version uses a standard state machine-based design that executes the algorithm. Each algorithm step is executed on a clock signal, but the workflow is optimized to execute multiple parallel steps when possible. Same as in the asynchronous version, each component contains its own state machine that executes the algorithm.

Effectively the synchronous design requires, in general, fewer clock cycles compared to the number of transitions of the speed-independent controller in the asynchronous version to execute the same task. The synchronous version does not require as many internal registers that are necessary for the asynchronous version to provide stable inputs to the FSE elements. However, the D-latch-based asynchronous registers are now replaced by D-flip-flops that increase chip area requirements. Most of the D-flip-flops are dedicated to the state machine state holding memory. Differently in asynchronous design, the state holding task is realized by the C-element, which implementation uses fewer transistors than the D-flip-flop.

The synchronous design dictates the speed of the surrounding simulated environment for analysis purposes. All external to the controller elements, which consist of the local cache memory and the main memory, respond within a single clock cycle. The one clock cycle delay in conjunction with the clock time is then used to specify the same environment delay in the asynchronous version. The approach allows for the unification of the delay-impact caused by the simulated environment surrounding DUT on test results.

The synchronous design delivers a reliable reference implementation that provides an equivalent functionally module compared to the asynchronous version. Using two closely related devices allows for an accurate evaluation of the circuits produced by the proposed methodology with reference to widely practiced synchronous design methods. Both controllers execute the same algorithm and feature the same structure with differences only related to the use of different design methodologies.

5.6 Note on synthesis approach of the Controller circuit

Due to the lack of tools that would directly use the CCS library model of ASIC technology library to synthesize the speed-independent controller, the synthesis was performed using standard algorithms available in the Synopsys DC Compiler tools for synchronous logic. Functional tests performed on the gate-level representation indicated no apparent hazardous behavior or circuit malfunction. All the tests give correct results; therefore, it is justifiable to assume that the generated model is stable enough to be considered correct for the presented work's evaluation purposes.

6 FUTURE IMPROVEMENTS.

6.1 Improvements to the methodology and tools

The presented methodology allows for the design of asynchronous circuits implementing complex algorithms and provides a streamlined approach in the specification, testing, and maintenance of non-trivial designs. However, there are scenarios in which the current state could be improved to further extend the methodology's scope, optimize the design process, or provide a clearer and concise design specification approach. The following section discusses some improvements that could help solve some of the existing problems, extend the functionality, and make the design process easier.

6.1.1 The FSE completion detection

In its current version, the methodology proposes delay matching lines as the only mechanism to ensure correct timing for the FSE modules. Delay matching lines are simple in their design but have drawbacks, such as being dependent on the used ASIC technology, and they could grow long when covering slow logic delays. Because of the technology dependence, the delay matching lines require tuning with the change of PDK or change of the surrounding environment of the FSE. Effectively, the use of delay matching lines makes the design less portable and requires repeated adjustment effort with any change of the circuit environment or the process technology.

The methodology can benefit from additional research into applications of other completion detection circuits [36] and their uses in specific types of FSE. Compatible structures of FSE modules could be isolated as candidates resulting in a recommendation of best practices in FSE module design. The completion detection circuits might not completely replace any use of delay matching lines, but reducing the number of critical points that require technology-dependent delay matching would simplify the synthesis process and make the design itself more robust.

6.1.2 Difficult syntax of the confusion block

In its current version, the syntax representing the confusion block introduces a challenge from the expressiveness standpoint. Especially in the case of gated confusion, the syntax variations make it hard to determine the entry and exit connections to the choice transitions. The choice transitions are the set of transitions that the event-flow branches into. The CSP parser could introduce an additional syntax to differentiate specific types of confusion block.

The biggest challenge comes with the gated confusion block construct. In the gated confusion case, the choice transition's pre-set consists of one or more preceding transitions and places that lead to the choice transition. Similarly, at the exit point from a choice transition, the post-set specifies one or more places or transitions that the choice transition leads to. When a choice transition or a choice transition fragment does not merge into the default output point, it requires an explicit place to break the default transition chain. Similarly, if the choice place post-set includes the default exit point and some other transition, then that branching to non-default transition must be specified using the explicit place construct. The current syntax structure makes it difficult to ascertain from the code which transitions are the choice transitions and which belong to their pre-sets and post-sets.

The CSP parser could extend the confusion block syntax to make it possible to explicitly specify the entry and exit points. Such a syntax can be a triplet that specifies pre-set, post-set, and the choice transition itself. Also, an additional notation is necessary to indicate a situation when a path does not merge back to the default point. Having the extended syntax would allow for more readable code and better syntax verification.

6.1.3 Passing CSP fragments as arguments to other CSP fragments

There are many scenarios in the CSP flow in which one set of transitions can be embedded within another repeatedly. An example is some STG fragments that occur during a 4-phase handshake. First the req+ and ack+ transitions occur for some element, then the embedded part and at the end req- and ack-. Such scenarios are common, and they can repeatedly occur for a certain element but with different embedded transition sets. In its current state, each transition string must be specified separately. It is possible to group repeating parts into fragments, but it is not an optimal approach from the syntax readability perspective.

The CSP parser would benefit from a syntax extension in which a CSP fragment receives another fragment as an argument and embeds it in itself. An example would be a CSP fragment that introduces the following chain req+; ack; f_embd req- ack-. The f_embd is the argument that expands into the CSP fragment containing an arbitrary set of transitions. Providing the embedding of CSP fragments would allow for easy specification of a common external transition environment that can be reused with different internal transition sets.

6.1.4 Full support of standard C-implementation synthesis of set and reset Boolean functions

Despite the next-state equations are being generated by the Petrify tool with the "gcm" flag, which generates Boolean equations that satisfy the monotonous cover constraint, using standard synchronous logic synthesis algorithms still introduces a possibility that the resulting gate-level representation might suffer from hazards that could affect the correct circuit behavior. To prevent the scenario that results in a faulty circuit being generated, the stage 2 model should synthesize the controller's next-state equations using algorithms presented in [4] and [9].

A specialized synthesis module is required for the speed-independent controller synthesis that generates gate-level, process technology aware representation of the controller in stage 2. Ideally, the functionality should extend to stage 1 because of the possibility of added internal signals resulting from a hazard-free decomposition process. Potentially beneficial is also adding a mechanism for Place and Route that performs placement and interconnect routing while aware of speed-independent constraints of the circuit.

6.2 Improvements to the cache coherency controller design

The presented cache coherency controller design introduces a sufficient complexity level to serve as the proposed methodology verification and evaluation platform. However, realistic systems must account for additional scenarios like the memory space regions that are not cacheable or larger than three number of cores in the system. This section discusses some of the additional features, proposed solutions and implementation concepts related to the cache controller design.

6.2.1 Handling main memory regions with different properties

One of the characteristics of a realistic computer system's memory space allocation is the non-uniformity. Different address regions within the systems can have different access properties and not necessarily represent the random access general-purpose memory. Moreover, some memory regions could be set up so that they could not be cached locally. A good example is the MMIO regions, in which a memory address represents some peripheral device's operating registers. If such a region would be cached locally, then writing under its address would incorrectly only alter local cache memory content instead of immediately sending a command to the device. The second scenario is the existence of read-only memory regions, in which case, a core must not alter a specific memory region's content.

A hybrid system based on lookup tables and additional flags in the cache line could be introduced to deal with the memory region access properties. Such a system would be a utility element for the Sender component. One of its parts is the addition of extra flag bits into the cache line structure. The flags indicate the memory line's access level under the given address and whether the cache line resides in a non-cacheable region. If in a non-cacheable region, the Sender seeing the non-cacheable flag, omits cache access and resorts immediately to bus communication. However, such a solution introduces a tradeoff between speed and resource usage as the memory line is unused as long as the non-cacheable record is held in the local memory. The second part of the system would be a lookup element. Such an element would contain information mapping records about the entire memory space; when queried with a given address, it would return all the access properties. However, depending on the information's complexity and granularity, such an item could introduce an additional performance hit as it would take time to perform the query and increase the chip area to hold the records.

Most of the changes would involve the Sender component; however, the Receiver and Responder might require additional data indicating whether to react on certain bus messages based on the memory region properties. Finally, an exception reporting system would be required that notifies the CPU core of the failed transaction due to memory restrictions. Such a system would most likely rely on the CPU interrupt system to deliver the error-related information and activate proper service subroutine.

6.2.2 Implementing cache line size larger than a single word

The presented design introduced an assumption that the cache line size is of a one machine word, in this case, 4 bytes. However, realistic systems deal with larger than single word cache line sizes. It is not uncommon to have the cache line size from two to even four words and larger. Multiple word cache line introduces an additional complexity

related to bus communication. Depending on the cache line's size, two proposed techniques could be applied to add the hardware support.

The first approach is to widen the bus data line. Instead of transmitting a single word per transaction, the bus data path could be extended to transmit multiple words. It is a simple solution that does not increase the complexity of the send and receive operations but also has some limitations. For the cache line spanning multiple words like four or more, extending the bus width also increases the circuit size. At some point, it is impractical to apply bus widening alone, especially for small embedded circuits.

The second approach is to extend the transmission operation across multiple transactions. As a result, all the system components would have to be modified to support processing data transmissions that span multiple bus messages. The Sender module must add a mechanism that detects when all the chunks of data came through during the Rd or RdX response. In its simplest form, it could be a counter which, when reached a certain count, would indicate completeness. Also, the Sender and Responder must implement a sending mechanism capable of splitting the cache line into portions that fit the bus width and send in multiple transactions. The Receiver would also need to change to identify and place the data portions in their designated places within the cache line.

Lastly, the multi-message transaction introduces a design tradeoff in how the subsequent messages are delivered. One option is to lock the bus and transmit everything in a burst. The burst approach guarantees the fastest time to complete data transmission but prevents any other communication and negatively impacts system response time. For example, a third core does not participate in receiving data but tries to write to an unrelated memory region. This third core would not send the Upg message until the currently ongoing Flush transaction finishes. The second approach is to obtain the bus access and immediately release it after single transmission, repeating the process for every chunk of data. The bus becomes more accessible, but the transmission speed decreases as

the sending core need to go through arbitration every time it sends the data portion. There is also a middle ground possible such that a core sends bursts, which each burst covers a portion of the whole transaction. After a single burst, the responding core releases the bus to allow for other communication and locks the bus again to send the next burst sometime after.

The best approach is to combine all the techniques based on system design goals. Widening the bus lowers the number of messages needed to transmit; splitting a transaction among multiple bus transmissions allows to send large multiword cache lines without overly increasing the design area but at the cost of the increased complexity of the transmission logic.

6.2.3 Removing the assumption that memory is always slower than the cores

The presented controller design makes an assumption regarding the data delivery, stating that no matter what state the circuit is in, if there is an outstanding request for data and at least one core is capable of responding, then this core will always be faster to obtain the bus access than the main memory. The assumption is made to lower the controller design complexity level and to clearly demonstrate the methodology and not overly complicate the design for this thesis's purposes. However, in a realistic asynchronous system, it is unknown when any core will be capable of responding. The delay might be caused by arbitration, both for the internal resources and the system-wide access to the bus. There is also an inherent delay introduced by the Responder operation caused by the need to traverse the PLOG component to eventually reach the request of interest. In a realistic system, the delays can sum up to the point when it would be the main memory that is faster to respond.

The race condition between the main memory and the cores causes data consistency problems. The presented controller uses the write-back scheme, meaning when the core

changes the local memory's data, the newest value is not sent outside of the core unless necessary. In the write-back based system, the main memory can contain an outdated value unless a core sends the data out in a write-back (Wb) bus transmission. If the core and main memory race each other for the bus access, a situation is possible when the main memory wins and accidentally delivers outdated data overwriting the most recent version causing incorrect system behavior.

An additional confirmation system is necessary to protect the asynchronous system from a race condition between the main memory and the CPU cores. One proposed solution is to split the Rd and RdX requests into separate operations, one set for the core to core communication and the other for core and memory data exchange. If, for example, a core to core Rd or RdX is sent, the main memory will ignore it and never respond. However, when a core sends a hypothetical RdMM or RdXMM, then it is the other cores that will not respond, but the main memory will eventually deliver the data.

Splitting the request operation allows to avoid the race but leaves the question of when to ask which unit for the data. The answer is first to query the cores, and if none can deliver the data, then ask the main memory. The system must add another bus message type to implement the race-free functionality that indicates no data (NDta). Then the algorithm requires every core to respond to Rd or RdX either with Flush or with NDta. The requesting core counts the NDta responses, and knowing how many cores are present in the system will determine the condition when it must ask the main memory instead. When the count reaches $N-1$ for which N is the number of cores in the system, then the Rd or RdX cancels, and the requesting core sends RdMM. However, if an actual Flush comes from any core, then the count cancels, and data is received.

An alternative approach could be taken in which the main memory controller performs the counting of NDta messages, and when the count reaches its threshold, the main memory controller sends the requested data. The alternative approach puts an

additional burden on the main memory controller to actively keep track of the bus communication listening for NDta messages but also allows for optimization. The main memory can pre-fetch the data while waiting for all CPU cores and respond without delay if the NDta count threshold is reached.

Solving the race condition between the main memory and the cores is necessary to ensure memory coherency in a realistic system. The speed independent approach to asynchronous logic design places a general assumption on the system expecting an unspecified time frame for each component to deliver data. However, the unspecified response time assumption does not allow for an unspecified order in which all system components can respond, placing a burden on the design to sequence the communication flow properly.

6.2.4 The PLOG bottleneck

The cache coherency controller's current design allows for a resource contention-free implementation of the PLOG component. The setup is currently designed to support a single issue in-order pipeline that has to stall if waiting for the memory stage. Also, the number of cores in the system is relatively small, with only three cores. Both scenarios result in a PLOG design in which only one record is needed per core, keeping the memory requirement small.

However, in some classes of systems, it is not uncommon to see a larger number of cores and superscalar out-of-order CPU design. The multiple issue CPU with the out-of-order execution delivers an architecture capable of executing multiple instructions in parallel beyond the simple pipeline approach. Therefore, a superscalar CPU could issue multiple parallel memory requests in an environment in which the execution of instructions does not have to finish in the order in which they were scheduled. Allowing multiple outstanding data requests per core increases the number of required records in

the PLOG element. Similarly, increasing the number of cores multiplies the required PLOG memory capacity. Both factors can cause the PLOG memory register to grow significantly enough to make the asynchronous register-based approach for holding records impractical.

One approach to solve the issue is to modify the PLOG to use a static memory block, increasing the memory capacity. Adding additional external memory rises the design logic complexity and introduces longer delays that involve the dataset query and write operations. However, the approach based on memory capacity extension keeps the PLOG a contention-free, given that enough memory capacity is available, ensuring that the CPU cores do not have to compete for PLOG space to put the data in.

Another approach is to limit the number of available records at a quantity being below the theoretical maximum generated by the system. The PLOG would then add a full signal to its interface that, when high, prevents the Sender from going forward with transmitting the data request. The Sender then stalls until a free record becomes available. This approach allows the module recordset size to remain at a controlled level, but the PLOG module becomes a contention point between all the system cores. The PLOG in each core keeps track of all outstanding split-transaction requests, even those unrelated to this core. Thus, if the PLOG size is too small, competing cores could have problems obtaining a spot to place the record of its transaction, leading to decreased system performance under significant load for which a large number of bus messages are of Rd or RdX type. Both proposed approaches offer tradeoffs in terms of performance and resource requirements. However, improving the PLOG component's ability to handle many outstanding data request transactions is necessary when supporting more advanced CPU architectures and systems with a larger number of processing cores.

6.2.5 Extending the number of cores, asynchronous arbiter bottleneck

Extending the number of CPU cores in an arbitrated system makes the asynchronous arbiter a bottleneck and a potential limitation in scaling up to a many-core architecture. There is a significant growth in transistor count between a mutex that handles two inputs and the version with three lines or more. Increasing the number of inputs further causes the transistor count to grow, eventually making the element impractical to implement.

To allow the system to handle arbitration of more cores requires additional design techniques. One proposed approach is to introduce clustering and multilevel arbitration [17]. The clustering approach resembles the current design of arbitration deciding access to the bus between Responder and Sender, in which a local arbiter within the MESI controller requests bus access from the top-level systemwide bus arbiter. Only when the top-level arbiter sends the grant signal can the local arbiter allow bus access to either Responder or the Sender. A similar approach is possible to handle many-core applications. The total number of cores is divided into smaller clusters. Each cluster has its local arbiter that then contacts the global unit. The approach allows for a reduction in the number of contending inputs per arbiter device, also reducing its size.

An additional starvation problem arises from the asynchronous arbitration of many cores in a non-deterministic way. The many-core systems suffer from starvation problems that are subject to ongoing research [10]. A simple non-deterministic asynchronous arbiter might turn out to be insufficient and introduce a non-uniform or, more generally, non-optimal arbitration process. One potential solution is to extend a regular arbiter with a controller circuit that executes the arbitration process based on an algorithm that introduces systemwide load balancing.

7 ANALYSIS AND RESULTS

7.1 Practical considerations for efficient asynchronous design

The proposed methodology allows for the design of complex circuits that can compete with synchronous counterparts in the chip area requirements and power consumption. However, the proposed methodology leaves the designer the task to specify elementary device composition within the Component modules. The designer has much freedom in deciding which elements and their specific parameters to use, but the methodology requires the designer to be aware of the design goals and be mindful of unnecessary overuse of the resources. It is essential to be aware of the trade-offs using specific techniques and know when and how to apply them. This section briefly discusses some of the design techniques tied to the proposed methodology, their advantages, and their limitations.

The first and most influential element in the proposed design methodology is the use of asynchronous registers. In some cases, using a register is necessary and the only way to provide stable input to the FSE module. The register also serves the role of a local general-purpose memory element that allows preservation of essential data for later use and early release of other resources. However, the use of the register element results in an increased area footprint. Uncontrolled overuse could lead to a substantial growth of the chip invalidating the area advantage over the synchronous design.

To avoid uncontrolled register usage following proposed techniques can be applied. The primary way to save space is to store only essential data. Suppose a piece of information is held in a portion of the register and is never used, then that portion of the register should not be instantiated. Depending on the difficulty of filtering out the relevant data, an FSE or Support element can be used to provide inputs to the register. The second approach is to re-use a register. A register should be re-used when it can support multiple

tasks that do not collide with each other at any point in time. Finally, the designer should look at opportunities to provide stable input to FSE modules without over-relying on registers.

Another proposed technique is related to the inherent problems with the design and synthesis of speed-independent controllers. Complex functionality translates directly to increased complexity in the STG model, effectively putting a bigger strain on the synthesis tools. At some point, the STG synthesis could suffer from state explosion or the CSC step's inability to converge, preventing the STG synthesis from generating the final set of next-state Boolean expressions. It is possible to reduce the complexity of a speed-independent controller by breaking it into multiple smaller circuits that execute their sub-tasks and cooperate with each other. The divide and conquer strategy allows for better maintainability and ability to overcome the synthesis tools computing limitations. However, splitting a large controller into a set of simpler controller units shifts the complexity from the STG into the component's internal design. As seen in the Sender component case, the module needed a complex support logic for gating and distributing the b_msg_valid line that signaled incoming bus messages. However, the controller breakdown greatly simplified the sequential controller circuit, which otherwise would have to support long chains of parallel transitions to properly acknowledge the incoming bus messages, thus resulting in a very large SG during the synthesis.

Increased complexity of the support element is not always an undesirable side effect. In fact, the support module's primary task is to take away some of the complexity from the sequential speed-independent controller design by providing the signal gating, combining signal lines, and other data and control signals manipulations. Specifically, the speed-independent controller design can be significantly simplified by signal gating. The signal gating ensures that the controller would not have to deal with transitions coming at a time when they are being a side effect from other components execution and are of no

interest to the local controller at this time. The support module also provides a simple mechanism for signal combining, such as the message acknowledge signals that combine using the C-element in the top controller module or an input signal formatting. However, parts of the support module that perform input formatting can introduce additional delay in the combinational logic that breaks the delay matching in the FSE components. Therefore, the support modules must sometimes extend the delay matched control lines to preserve hazard-free behavior.

7.2 Approach to system verification

One of the most prominent features of the proposed methodology is the emphasis on design verification. The functional verification process is expected to be performed for both model stages and allows for different insight into the device's behavior. This section discusses some proposed techniques in verification for the asynchronous design.

The first three techniques apply to the stage 1 model, for which the behavioral model of combinational elements works together with a simulated model of primitives such as C-element or the MUTEX. At the stage 1 level, the delays embedded into the device are tuned in such a way that the device is assumed a correct asynchronous logic construct. Assuming a correctly working logic delay leaves the verification to focus on testing the functional aspects. The first technique allows for verification of the speed-independent controller device. The resulting Boolean next-state expressions and the C-layer's behavioral model is placed into a test bench that runs through every expected path in the STG. The test bench exerts the input signals on DUT and listens for the correct responses on the outputs. If an unexpected output shows up at any time, then there is a fault in the circuits at the STG specification level. Otherwise, we can expect correct behavior, and the test passes.

Stepping through the controller workflow is also necessary for the stage 2 model after hazard-free decomposition, synthesis and P&R step. The test allows for examining all execution paths under different environment delays to detect any hazards in the circuit's gate-level implementation. In the stage 2 model scenario, instead of the behavioral model, the gate-level representation is used with realistic ASIC primitives delay. Additionally, after the P&R phase, the realistic interconnect delays are taken into account, increasing the test accuracy.

The second technique is the unit testing of the FSE. FSE's are essentially combinational circuits with added delay matching lines or other completion detection mechanism. Therefore an FSE can be wrapped in a test bench that tests output correctness for any expected stable input. For every possible input configuration, the test bench first sets the data signals then executes the handshake. If the output is incorrect, then there is an error in the logic, and the test fails. The third and last technique is to test the top module and, if necessary, the internal components separately. The third technique is a typical functional test bench that runs expected scenarios and checks the circuit's response.

All three techniques allow for functional verification, starting from their basic building blocks and ending in the entire device. Passing the tests concludes that the functional model of the device is correct. However, to ensure correct implementation, stage 2 tests performed on the gate netlist with realistic interconnect delays are necessary. The functional tests written for the third technique from stage 1 are particularly important as they can be reused on the synthesized model from stage 2. The gate-level series of tests is best executed after the P&R phase with the delay matching adjustments completed. The place and route tool can export a structural representation of the device and the interconnect delays. Then the series of tests written previously for the top model are applied to the device to check if the gate-level model passes them.

Although not guaranteed for all P&R the tools, the Innovus [45] suite does not alter the gate-level model from the synthesis phase. The gate-level structure is not flattened or, in any other way, altered because there is no automated addition of any primitives, unlike in the synchronous model when adding the clock tree. The effect is that the gate-level SystemVerilog module structure remains unchanged. Retaining the human-readable model allows, in the case of an error at the gate-level, to inspect specific interfaces signal transition log to identify bugs. Alternatively, suppose the P&R tool does alter the model structure. In that case, there usually exists a functionality within the tool to generate signal mappings between the high-level and the generated gate-level representation, which preserves the ability to identify and inspect critical signals.

7.3 Results

The presented case study implementation of a MESI cache controller evaluates the usability of the proposed methodology. For functional verification, the controller circuit is wrapped in a test bench that mimics its expected working environment. The test bench provides modules to simulate the CPU core, the bus, and the cache memory. The stage one model passes the functional test that checks the correct operation in different conditions, such as when a CPU core requests data not currently present in the local cache or when a request for data comes from the bus. Then the stage two gate-level representation executes and passed the same tests during simulation with extracted realistic interconnect and primitive delays from the ASIC library. Implementation of the custom asynchronous elements and the synthesized design uses the FreePDK15 PDK [42]. The custom set of asynchronous elements supplements the Open Cell Library 15nm [50] and follows its design rules.

The methodology evaluation phase uses both model variants: the asynchronous and synchronous reference design. Both models execute test tasks equivalent from the

functional perspective but employ interface communication models characteristic to each design approach. The test tasks intend to simulate expected execution tasks performed by the cache controller. Full documentation of the tests performed and the test configurations and their results are shown in Appendix B while a subset of representative results is included in Table 2.

Table 2
Comparison of Synchronous and Asynchronous Design Models.

Async vs Sync comparison			
	Asynchronous	***Synchronous Fast*** $(CT = 200ps)$	***Synchronous Slow*** $(CT = 700ps)$
Area $[\mu m^2]$	759.890	1053.082	944.849
Exec. Duration $[ns]$	4709.420	1349.901	4724.651
Avg. Power $[mW]$	0.3245	8.851	1.784

The comparison between asynchronous design and the synchronous reference happens in two configurations. Configuration 1 measures the speed of the asynchronous circuit with respect to the synchronous reference. The asynchronous model follows the default synthesis procedure, while the synthesis of the synchronous model tries to obtain the highest achievable speed. The resulting achievable clock time for the fast synchronous circuit becomes 200 picoseconds. Following the test environment assumptions, the test-bench response time is of a single clock cycle in the synchronous design. Therefore the response delay for the asynchronous circuit test harness is also set to one clock cycle equivalent time, which is 200ps.

The configuration one that tests maximum possible synchronous design speed shows a significant speedup compared to asynchronous design. The synchronous version is nearly 3.5 times faster but at the cost of increased area and significantly higher power usage. Such a result is not unexpected; the synchronous design can finish multiple tasks such as

data calculation, resource contention, or algorithm branch resolution within a single clock cycle. The situation looks different for the asynchronous design; the STG must perform multiple transitions in a sequence. For example, request local memory access from the arbiter, then set the input lines when the access is granted, and finally execute a 4-phase handshake to exchange data. The asynchronous design's execution speed result can also be improved by potentially reducing the delay of FSE elements, increasing parallelization of tasks, or other optimization procedures. However, the performance improvement might come at the cost of increased area.

When considering the Power, Area, and Speed properties, it is apparent that all three attributes depend on each other; tuning for one causes changes in the other two. Therefore, comparing power usage and area requirements of the asynchronous design versus the fast synchronous version does not give an accurate result because of the already significant bias toward the fast clock speed. In short, the fast synchronous design sacrifices all other parameters for maximum speed.

The synchronous design must be slowed down to the point when its test execution time matches with the asynchronous version to perform a more objective analysis of the area and power usage. However, the speed matching process does not end up merely slowing the simulation clock. Instead, the synchronous design is incrementally re-synthesized with lower clock speed. The synthesis under less stringent timing constraints allows the tools to optimize the area usage of the circuit. Also, the process of speed matching happens seamlessly for both synchronous and asynchronous designs. During the process, both the synchronous design clock is slowed down, and the asynchronous test harness delay is adjusted using the new clock time. When both designs show the same or very close execution times for a target test case, the test can be performed.

$$Area\ ratio(async/sync):\ \frac{759.890}{944.849} = 8.04224 * 10^{-1} \approx 80.42\% \tag{3}$$

For configuration number two, the speed of both synchronous and asynchronous circuits are matched. Removing the speed difference takes one variable attribute out of the equation allowing for a more reliable look at the power usage and area requirements. The asynchronous design exhibits a smaller area compared to the synchronous design. In the tested configurations, the asynchronous circuit area is approximately twenty percent smaller than the synchronous design. The obtained percentage result comes from taking a ratio, as in Equation 3, of the asynchronous to synchronous final area usage. The difference is due to multiple factors, which first is the existence of a clock tree in the synchronous design. For configuration 2, the clock tree consists of approximately 20.4% of the synchronous circuit total cell area. The asynchronous design also adds the area usage during P&R due to delay matching elements, but this additional logic accounts only for about a 6.5% area increase.

The second cause of the area difference is the use of memory elements. The synchronous design uses flip-flops while the asynchronous design has D-latches and C-elements as memory elements, which in both cases are smaller than a flip-flop element. The synchronous design does not use as many internal registers, but the flip-flop elements are used for holding the state and serve as record memory in the PLOG component.

Finally, the asynchronous design presents a significant improvement in power usage. The power measurements performed for both designs were done using the time-based approach offered by the PrimeTime PX [51] tool. The timing-based power analysis is necessary to estimate the accurate power usage by the asynchronous system. The commonly used power estimation methods based on the switching rate of signals do not give accurate results because of the lack of the reference clock signal in the asynchronous

logic switching data-set and inconsistent components' behavior. The timing-based power analysis method requires a representative data-set of signal activity obtained by simulating a test case to estimate power usage accurately. Next, a set of data points is generated in which each point represents current power use at a specific point in time. The final result is a collection of data points that can be plotted to visualize the activity and used for further computations.

The Fig. 42 and Fig. 43 show a subset of the power data acquired during the analysis of the waveforms generated by execution of the test cases. The Fig. 42 shows the power usage for the asynchronous device. We can observe nearly random spikes representing the activity of the internal components as they execute their tasks. The second Fig. 43 shows the behavior of the synchronous model. A clear pattern is visible in the synchronous version, representing transitions occurring at the clock edges with the tallest spikes showing the electric current flow during the flip-flop switching. Also, every other transition shows smaller activity dying out before the next clock cycle; that is the combinational logic settling in reaction to the input change. The main power spikes are farther apart than in the asynchronous version and carry a much higher value. Both plots show expected patterns, which proves a valid base for power calculations.

$$P_{avg} = \frac{1}{T} \int_0^T P(t)dt \tag{4}$$

$$P_{avg} = \frac{1}{T} \sum_{t=0}^{T} (P[t] * \triangle t) \tag{5}$$

When calculating the average power usage, it is necessary to compute the integral over the obtained power dataset based on the formula in Equation 4. The integration provides the total energy transferred through the circuit. Next, the calculated energy is divided by

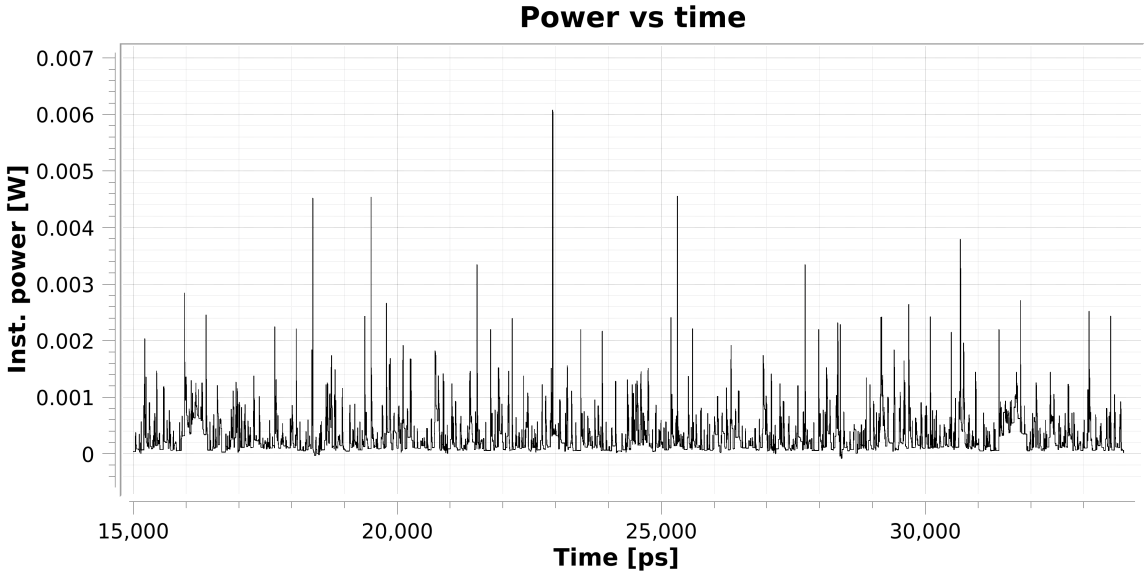


Fig. 42. Asynchronous design dynamic power measurement data.

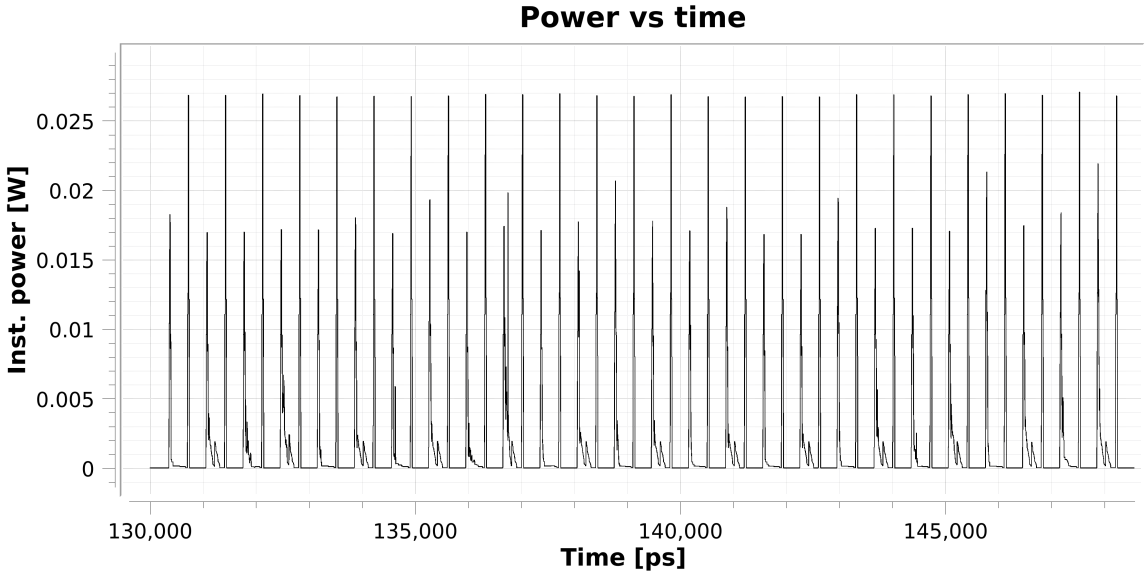


Fig. 43. Synchronous design dynamic power measurement data.

the test duration time to obtain the average power in Watts. Since the power dataset is discrete, the integral transforms into a sum in which each data point is multiplied by the minimal simulation time step, as shown in Equation 5. The time difference between two data points occurring is one picosecond ($\triangle t = 1ps$). Finally, the sum of the points is then divided by the test's total duration time, the same as in the continuous-time case.

$$Power\ ratio(async/sync):\ \frac{0.3245}{1.784} = 1.819 * 10^{-1} \approx 18.19\% \tag{6}$$

In configuration number 1, the resulting power calculations show a significant decrease in power usage for the asynchronous circuit with reference to the synchronous counterpart. For the same test in configuration number 2 with speed matching, the power usage of the asynchronous device is at 20% of the corresponding synchronous version as seen in Equation 6. The power consumption improvement can be attributed to the lack of unnecessary switching activity. In the synchronous design, all the registers in the system contribute to the total activity regardless of whether the data contained by them changes. For the asynchronous variant, the logic activates only the circuit elements currently participating in the device activity.

The obtained results show improvements in the area usage and power consumption of the asynchronous design relative to the equivalent synchronous version. However, the asynchronous model in its current form cannot reach the speeds of the synchronous version. The results prove that the methodology can produce asynchronous circuits that do offer advantages over synchronous models. The primary drawback is the speed of the device. The asynchronous model must perform more steps than the synchronous version to execute the same algorithm resulting in slower operation. However, if the asynchronous circuit can meet the design goals related to the device's speed, then using the proposed methodology would result in a smaller device that uses less power.

8 CONCLUSIONS

The presented work introduces a methodology that provides a framework for creating complex sequential asynchronous circuits. Introduced methodology specializes in the design of sequential digital devices based on the input-output speed-independent controller model. The target type of circuit is a design that executes a multi-step, data-driven, non-linear complex algorithm. An expected outcome is a non-pipelined, sequential controller-centric circuit focused on small circuit area and low power properties. The presented approach emphasizes the divide-and-conquer philosophy and the ability to perform tests early in the design, allowing for an efficient design and debug process.

The methodology introduces a code template and a series of steps that form the design process. With the emphasis on the divide-and-conquer approach, the template structure introduces split into Component modules to split the design into smaller, easier to maintain functional blocks. The Component design then breaks into controller circuits defined in CSP derived syntax and a set of flow support elements that form the datapath and are driven by the speed-independent controller. The methodology offers a two-stage design approach. Stage one, a technology-independent stage, provides a behavioral codebase that allows early functional tests. Stage two specifies the implementation structure that leads to a gate-level representation of the circuit. Synthesizing the model requires a special approach. A bottom-up synthesis flow must be taken to synthesize specific parts of the design while protecting others from unwanted optimizations.

An implementation of a cache coherence controller is used to provide a case study to evaluate the proposed methodology. The controller implements the MESI cache coherency algorithm with split-transaction bus communication. The presented study provides two implementations: asynchronous implementation developed using the proposed methodology and a synchronous counterpart for the reference. Both models execute the

same algorithm to provide a reliable testing environment and result. The design passes functional tests both on behavioral stage one and gate-level stage two, and the developed MESI cache coherence controller successfully proves the usefulness of the proposed methodology.

Testing and evaluation of the methodology rely on three categories: area, speed, and power usage. The synthesis outcome demonstrates reduced area requirements of the asynchronous model by 20% compared to the synchronous speed-matched reference design. Performed simulation tests show the synchronous design to achieve speed higher than the asynchronous counterpart at the cost of increased circuit area and power usage. Finally, the power usage tests executed on speed-matched implementations show the asynchronous design exhibits significantly lower power consumption at the level of approximately 18% of the synchronous reference.

The proposed methodology introduces improvement in the design process of the asynchronous circuits implementing multi-step complex algorithms. Using the CSP inspired abstraction layer to specify speed-independent controller behavior improves the model's readability and maintainability. The introduced design template provides a component-oriented structure, allowing to break down complex models into easier to maintain and test elements. Qualitative analysis of the design process using the methodology shown achieved systematic workflow, which helps avoid design errors and assists the debug process. The comparison tests with the reference design indicate improvements in the form of decreased circuit area and lower power consumption. In conclusion, the methodology shows a viable potential for practical applications in generating complex sequential asynchronous circuits with a focus on low power and small chip area.

Literature Cited

[1] S. M. Nowick and M. Singh, "Asynchronous design—part 1: Overview and recent advances," *IEEE Design & Test*, vol. 32, no. 3, pp. 5–18, 2015.

[2] B. Rahbaran and A. Steininger, "Is asynchronous logic more robust than synchronous logic?" *IEEE Transactions on dependable and secure computing*, vol. 6, no. 4, pp. 282–294, 2009.

[3] N. H. Weste and D. M. Harris, "Cmos vlsi design: a circuits and systems perspective," 2011.

[4] C. J. Myers, *Asynchronous circuit design*. John Wiley & Sons, 2001.

[5] J. Sparsø, *Asynchronous circuit design - A tutorial*. Boston / Dordrecht / London: Kluwer Academic Publishers, dec 2001. [Online]. Available: http://www2.imm.dtu.dk/pubdb/p.php?855

[6] P. A. Beerel and M. E. Roncken, "Low power and energy efficient asynchronous design," *Journal of Low Power Electronics*, vol. 3, no. 3, pp. 234–253, 2007.

[7] N. Karaki, T. Nanmoto, H. Ebihara, S. Inoue, and T. Shimoda, "43.1: A flexible 8-bit asynchronous microprocessor based on low-temperature poly-silicon (ltps) tft technology," in *SID Symposium Digest of Technical Papers*, vol. 36, no. 1. Wiley Online Library, 2005, pp. 1430–1433.

[8] M. Wolf, *High performance embedded computing : architectures, applications, and methodologies*, 2014.

[9] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, *Logic Synthesis for Asynchronous Controllers and Interfaces: With 146 Figures*. Springer Science & Business Media, 2002, vol. 8.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[11] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[12] D. E. Culler, A. Gupta, and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[13] S. M. Nowick and M. Singh, "Asynchronous design—part 2: Systems and methodologies," *IEEE Design & Test*, vol. 32, no. 3, pp. 19–28, 2015.

[14] A. Davis and S. M. Nowick, "An introduction to asynchronous circuit design," *The Encyclopedia of Computer Science and Technology*, vol. 38, pp. 1–58, 1997.

[15] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on information and Systems*, vol. 80, no. 3, pp. 315–325, 1997.

[16] J. Carmona, J. Cortadella, V. Khomenko, and A. Yakovlev, "Synthesis of asynchronous hardware from petri nets," in *Advanced Course on Petri Nets*. Springer, 2003, pp. 345–401.

[17] A. J. Martin and M. Nystrom, "Asynchronous techniques for system-on-chip design," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1089–1120, June 2006.

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

[19] C. Hoare, "Communicating sequential processes, electronic version based on the 1985 edition," 2015.

[20] S. Brookes, "Retracing the semantics of csp," in *Communicating Sequential Processes. The First 25 Years*. Springer, 2005, pp. 1–14.

[21] A. Roscoe, C. Hoare, and R. Bird, "The theory and practice of concurrency. 2005," *Revised edition. Only available online*.

[22] A. J. Martin, "Compiling communicating processes into delay-insensitive vlsi circuits," *Distributed computing*, vol. 1, no. 4, pp. 226–234, 1986.

[23] ——, "Programming in vlsi: From communicating processes to delay-insensitive circuits," CALIFORNIA INST OF TECH PASADENA DEPT OF COMPUTER SCIENCE, Tech. Rep., 1989.

[24] S. M. Burns, “Automated compilation of concurrent programs into self-timed circuits,” 1988.

[25] J. D. Garside, S. B. Furber, and S.-H. Chung, “Amulet3 revealed,” in *Proceedings. Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE, 1999, pp. 51–59.

[26] S. B. Furber, J. D. Garside, P. Riocreux, S. Temple, P. Day, J. Liu, and N. C. Paver, “Amulet2e: An asynchronous embedded controller,” *Proceedings of the IEEE*, vol. 87, no. 2, pp. 243–256, 1999.

[27] A. Bink and R. York, “Arm996hs: The first licensable, clockless 32-bit processor core,” *Ieee micro*, vol. 27, no. 2, 2007.

[28] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, “The vlsi-programming language tangram and its translation into handshake circuits,” in *Proceedings of the Conference on European Design Automation*, ser. EURO-DAC ’91. Los Alamitos, CA, USA: IEEE Computer Society Press, 1991, pp. 384–389. [Online]. Available: http://dl.acm.org/citation.cfm?id=951513.951597

[29] A. Bardsley and D. Edwards, “Balsa: An Asynchronous Hardware Synthesis Language,” *The Computer Journal*, vol. 45, no. 1, pp. 12–18, 01 2002. [Online]. Available: https://doi.org/10.1093/comjnl/45.1.12

[30] L. D. Tran, G. I. Matthews, P. Beckett, and A. Stojcevski, “Null convention logic (ncl) based asynchronous design - fundamentals and recent advances,” in *2017 International Conference on Recent Advances in Signal Processing, Telecommunications Computing (SigTelCom)*, Jan 2017, pp. 158–163.

[31] S. C. Smith, *Designing asynchronous circuits using NULL convention logic (NCL)*, ser. Synthesis lectures on digital circuits and systems (Online), 23. San Rafael, Calif.: Morgan & Claypool Publishers, 2009.

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

[33] A. Spiteri Staines, “Representing petri net structures as directed graphs,” 01 2011.

[34] T. Caohuu and J. Edwards, "Implementation of an efficient library for asynchronous circuit design with synopsys," in *Progress in Systems Engineering*. Springer, 2015, pp. 465–471.

[35] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanbekbergen, and A. Yakovlev, "Basic gate implementation of speed-independent circuits," in *Proceedings of the 31st annual Design Automation Conference*, 1994, pp. 56–62.

[36] P. A. Beerel, *A designer's guide to asynchronous VLSI*. Cambridge ; New York: Cambridge University Press, 2010.

[37] A. E. Abdallah, *Communicating Sequential Processes. The First 25 Years: Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004. Revised Invited Papers*. Springer Science & Business Media, 2005, vol. 3525.

[38] K. H. Rosen, *Discrete mathematics and its applications*, 7th ed. Boston: McGraw-Hill Higher Education, 2011.

[39] V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995.

[40] J. C. Baeten, "A brief history of process algebra," *Theoretical Computer Science*, vol. 335, no. 2-3, pp. 131–146, 2005.

[41] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open cell library in 15nm freepdk technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, ser. ISPD '15. New York, NY, USA: ACM, 2015, pp. 171–178. [Online]. Available: http://doi.acm.org/10.1145/2717764.2717783

[42] K. Bhanushali and W. R. Davis, "Freepdk15: An open-source predictive process design kit for 15nm finfet technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, ser. ISPD '15. New York, NY, USA: ACM, 2015, pp. 165–170. [Online]. Available: http://doi.acm.org/10.1145/2717764.2717782

[43] *Design Compiler User Guide*. Synopsys, 06 2009.

[44] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolić, *Digital integrated circuits: a design perspective*. Pearson Education Upper Saddle River, NJ, 2003, vol. 7.

[45] *Innovus User Guide*. Cadence, 06 2019.

[46] S. Sutherland and D. Mills, "Synthesizing systemverilog busting the myth that systemverilog is only for verification," *SNUG Silicon Valley*, p. 24, 2013.

[47] S. Sutherland, "Verilog® hdl, quick reference guide," 09 2007.

[48] A. Bindal, *Fundamentals of Computer Architecture and Design*, 2017.

[49] D. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017.

[50] A. J. Martin, A. Lines, R. Manohar, M. Nystroem, P. I. Pénzes, R. Southworth, U. Cummings, and T. K. Lee, "The design of an asynchronous mips r3000 microprocessor." in *ARVLSI*, vol. 97, 1997, pp. 1–18.

[51] *PrimeTime PX Users Guide*. Synopsys, 12 2011.

[52] A. J. Martin, "Asynchronous datapaths and the design of an asynchronous adder," *Formal Methods in System Design*, vol. 1, no. 1, pp. 117–137, 1992.

[53] R. Manohar and A. J. Martin, "Quasi-delay-insensitive circuits are turing-complete," Pasadena, CA, USA, Tech. Rep., 1995.

[54] J. Sparsø, "Current trends in high-level synthesis of asynchronous circuits," in *Electronics, Circuits, and Systems, 2009. ICECS 2009. 16th IEEE International Conference on*. IEEE, 2009, pp. 347–350.

[55] E. Brunvand, S. Nowick, and K. Yun, "Practical advances in asynchronous design and in asynchronous/synchronous interfaces," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, 1999, pp. 104–109.

[56] G. B. Randy H Katz, *Contemporary Logic Design, 2nd ed.* Pearson Education, Inc., 2005.

[57] D. Givone, *Digital Principles and Design with CD-ROM*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2003.

[58] K. N. Bhanushali *et al.*, "Design rule development for freepdk15: An open source predictive process design kit for 15nm finfet devices." 2014.

[59] S. M. Sze, *Semiconductor devices: physics and technology*. John wiley & sons, 2008.

[60] Predictive technology model (ptm). Arizona State University. [Online]. Available: http://ptm.asu.edu/

Appendix A
THE CSP SOURCE CODE FOR THE CONTROLLERS

A.1 Pending-Log component controller.

```
module controller;

inputs recv_req sndr_req resp_req r1_ack stwr_ack sndr_proc_ack
   resp_proc_ack recv_proc_skip recv_proc_mark recv_proc_clr
   set_out_reg_ack r_idx_ack set_storage_ack sel_storage_ack;

outputs recv_ack sndr_ack resp_ack r1_req stwr_req sndr_proc resp_proc
   recv_proc set_out_reg r_idx set_storage sel_storage;

explicit_place P0;

loop_place [recv_ack-]->[recv_req+];
loop_place [sndr_ack-]->[sndr_req+];
loop_place [resp_ack-]->[resp_req+];
loop_place [recv_ack- sndr_ack- resp_ack-]->[recv_proc+ sndr_proc+
   resp_proc+];

marking [recv_ack-]->[recv_req+];
marking [sndr_ack-]->[sndr_req+];
marking [resp_ack-]->[resp_req+];
marking [recv_ack- sndr_ack- resp_ack-]->[recv_proc+ sndr_proc+
   resp_proc+];

main plog;

fragment frag_recv_proc;
fragment frag_sndr_proc;
fragment frag_resp_proc;

fragment frag_proc_skip;
fragment frag_proc_mark;
fragment frag_proc_clr;

endinterface

plog:
    *[
        [   [ recv_req+ ]; frag_recv_proc |
            [ sndr_req+ ]; frag_sndr_proc |
            [ resp_req+ ]; frag_resp_proc
        ];
    ]

frag_recv_proc:
      recv_proc+;
    [ frag_proc_skip | frag_proc_mark | frag_proc_clr ];
      recv_ack+  ;
    [ recv_req- ];
      recv_ack-  ;

frag_proc_skip:
    [ recv_proc_skip+ ];
      recv_proc-      ;
    [ recv_proc_skip- ];

frag_proc_mark:
    [ recv_proc_mark+  ];
      sel_storage+     ;
    [ sel_storage_ack+ ];
      r_idx+           ;
```

```
    [ r_idx_ack+        ];
      r_idx-             ;
    [ r_idx_ack-        ];
      sel_storage-       ;
    [ sel_storage_ack-  ];
      recv_proc-         ;
    [ recv_proc_mark-   ];
      P0                 ;
      set_storage+       ;
    [ set_storage_ack+  ];
      stwr_req+          ;
    [ stwr_ack+         ];
      stwr_req-          ;
    [ stwr_ack-         ];
      set_storage-       ;
    [ set_storage_ack-  ];

frag_proc_clr:
    [ recv_proc_clr+    ];
      sel_storage+       ;
    [ sel_storage_ack+  ];
      r_idx+             ;
    [ r_idx_ack+        ];
      r_idx-             ;
    [ r_idx_ack-        ];
      sel_storage-       ;
    [ sel_storage_ack-  ];
      set_out_reg+       ;
    [ set_out_reg_ack+  ];
      r1_req+            ;
    [ r1_ack+           ];
      r1_req-            ;
    [ r1_ack-           ];
      set_out_reg-       ;
    [ set_out_reg_ack-  ];
      recv_proc-         ;
    [ recv_proc_clr-    ];
      P0

frag_sndr_proc:
      sndr_proc+       ;
    [ sndr_proc_ack+  ];
      sndr_ack+        ;
    [ sndr_req-      ];
      sndr_proc-       ;
    [ sndr_proc_ack-  ];
      sndr_ack-        ;

frag_resp_proc:
      resp_proc+       ;
    [ resp_proc_ack+  ];
      resp_ack+        ;
    [ resp_req-      ];
      resp_proc-       ;
    [ resp_proc_ack-  ];
      resp_ack-        ;
```

Listing 31. CSP implementation: plog.

A.2 Receiver component controller.

```
module controller;

inputs msg_valid cache_gnt cache_ack p_ack stp1_ack stp2_ack stp3_write
     stp3_skip r1_ack;
outputs msg_ack cache_arb cache_req p_req stp1_req stp2_req stp3_req
    r1_req;

loop_place [msg_ack-]->[msg_valid+];

marking [msg_ack-]->[msg_valid+];

main recv;

fragment frag_cache_req;
fragment frag_stp1_req;
fragment frag_stp3_write;
fragment frag_stp3_skip;

endinterface

recv:
     *[
         [ msg_valid+ ];
         (frag_cache_req , frag_stp1_req)
           stp2_req+   ;
         [ stp2_ack+  ];
           cache_req+  ;
         [ cache_ack+ ];
           r1_req+     ;
         [ r1_ack+    ];
           r1_req-     ;
         [ r1_ack-    ];
           cache_req-  ;
         [ cache_ack- ];
           stp2_req-   ;
         [ stp2_ack-  ];
           stp3_req+   ;
         [ frag_stp3_write | frag_stp3_skip ];
           cache_arb-  ;
         [ cache_gnt- ];
           msg_ack+    ;
         [ msg_valid- ];
           msg_ack-    ;
     ]

frag_cache_req:
       cache_arb+  ;
     [ cache_gnt+ ];

frag_stp1_req:
       stp1_req+  ;
     [ stp1_ack+ ];
       p_req+     ;
     [ p_ack+    ];
       p_req-     ;
     [ p_ack-    ];
       stp1_req-  ;
     [ stp1_ack- ];

frag_stp3_write:
     [ stp3_write+ ];
       cache_req+   ;
     [ cache_ack+  ];
       cache_req-   ;
     [ cache_ack-  ];
```

```
        stp3_req-      ;
    [   stp3_write- ];

frag_stp3_skip:
    [   stp3_skip+ ];
        stp3_req-     ;
    [   stp3_skip- ];
```

Listing 32. CSP implementation: recv.

A.3 Responder component controller.

```
module controller;

inputs cache_gnt cache_ack p_ack rt_ack r1_ack r2_ack r3_ack idx_ack
    p_log_respond p_log_skip cache_respond cache_skip check_final_ok
    check_final_abrt bus_gnt bus_ack;

outputs cache_arb cache_req p_req rt_req r1_req r2_req r3_req idx_req
    p_log_check cache_check check_final bus_req place_msg;


loop_place [bus_gnt- cache_gnt- p_ack-]->[idx_req+];

marking [bus_gnt- cache_gnt- p_ack-]->[idx_req+];


main resp;

fragment frag_log_respond;
fragment frag_log_skip;
fragment frag_cache_respond;
fragment frag_cache_skip;
fragment frag_check_final_ok;
fragment frag_check_final_abrt;


endinterface

resp:
    *[
          idx_req+  ;
        [ idx_ack+ ];
          rt_req+   ;
        [ rt_ack+  ];
          rt_req-   ;
        [ rt_ack-  ];
          idx_req-  ;
        [ idx_ack- ];
          r1_req+   ;
        [ r1_ack+  ];
          r1_req-   ;
        [ r1_ack-  ];
          p_req+    ;
        [ p_ack+   ];
          p_log_check+;
        [ frag_log_respond | frag_log_skip ];
    ]


frag_log_skip:
    [ p_log_skip+ ];
      p_log_check- ;
    [ p_log_skip- ];
      p_req-       ;
    [ p_ack-      ];


frag_log_respond:
    [ p_log_respond+ ];
      r2_req+         ;
    [ r2_ack+        ];
      r2_req-         ;
    [ r2_ack-        ];
      p_log_check-    ;
    [ p_log_respond- ];
      p_req-          ;
    [ p_ack-         ];
```

```
      cache_arb+      ;
    [ cache_gnt+      ];
      cache_req+      ;
    [ cache_ack+      ];
      cache_check+    ;
    [ frag_cache_respond | frag_cache_skip ];


frag_cache_skip:
    [ cache_skip+ ];
      cache_check- ;
    [ cache_skip- ];
      cache_req-   ;
    [ cache_ack-  ];
      cache_arb-   ;
    [ cache_gnt-  ];


frag_cache_respond:
    [ cache_respond+ ];
      cache_check-    ;
    [ cache_respond- ];
      cache_req-      ;
    [ cache_ack-      ];
      cache_arb-      ;
    [ cache_gnt-      ];
      bus_req+        ;
    [ bus_gnt+        ];
    (
          p_req+  ;
        [ p_ack+ ],

          cache_arb+  ;
        [ cache_gnt+ ];
          cache_req+  ;
        [ cache_ack+ ]
    )
    check_final+;
    [ frag_check_final_ok | frag_check_final_abrt ];
    [ bus_gnt- ];


frag_check_final_ok:
    [ check_final_ok+ ];
      check_final-     ;
    [ check_final_ok- ];
      r3_req+          ;
    [ r3_ack+          ];
      r3_req-          ;
    [ r3_ack-          ];
    (
          place_msg+ ;
        [ bus_ack+   ];
          place_msg- ;
        [ bus_ack-   ],

          cache_req-  ;
        [ cache_ack- ];
          cache_arb-  ;
        [ cache_gnt- ],

          p_req-  ;
        [ p_ack- ]
    )
    bus_req-;
```

```
frag_check_final_abrt:
    [ check_final_abrt+ ];
      check_final-      ;
    [ check_final_abrt- ];
    (
          cache_req-  ;
        [ cache_ack- ];
          cache_arb-  ;
        [ cache_gnt- ],

          p_req-  ;
        [ p_ack- ]
    )
    bus_req-;
```

Listing 33. CSP implementation: resp.

A.4 Sender component controller handling bus transmission.

```
module controller;

inputs send gmv gmv2 bus_gnt addr_match addr_diff bus_ack req_type_y
    req_type_n;

outputs send_ack gmv_ack gmv_ack2 bus_req addr_cmp place_msg req_type;

explicit_place P0;
explicit_place P1;

loop_place [send_ack-]->[send+];

marking [send_ack-]->[send+];

main sndr_bus;

fragment frag_bus_gnt;
fragment frag_req_type_y;
fragment frag_req_type_n;
fragment frag_gmv;
fragment frag_addr_match;
fragment frag_addr_diff;
fragment frag_addr_match2;
fragment frag_addr_diff2;

endinterface

sndr_bus:
     *[
         [ send+     ];
           bus_req+  ;
           P0        ;
         [ frag_bus_gnt | frag_gmv ];
           send_ack+ ;
         [ send-     ];
           send_ack- ;
     ]

frag_bus_gnt:
     [ bus_gnt+  ];
       place_msg+ ;
     [ bus_ack+  ];
       place_msg- ;
     [ bus_ack-  ];
       req_type+  ;
     [ frag_req_type_y | frag_req_type_n ];

frag_req_type_y:
     [ req_type_y+ ];
       bus_req-    ;
     [ bus_gnt-    ];
       P1          ;
     [ gmv2+       ];
       addr_cmp+   ;
     [ frag_addr_match2 | frag_addr_diff2 ];

frag_addr_match2:
     [ addr_match+ ];
       addr_cmp-   ;
     [ addr_match- ];
       gmv_ack2+   ;
     [ gmv2-       ];
       req_type-   ;
     [ req_type_y- ];
       gmv_ack2-   ;
```

```
frag_addr_diff2:
     [ addr_diff+ ];
       addr_cmp-   ;
     [ addr_diff- ];
       gmv_ack2+   ;
     [ gmv2-      ];
       gmv_ack2-   ;
       P1

frag_req_type_n:
     [ req_type_n+ ];
       bus_req-    ;
     [ bus_gnt-    ];
       req_type-   ;
     [ req_type_n- ];

frag_gmv:
     [ gmv+      ];
       addr_cmp+ ;
     [ frag_addr_match | frag_addr_diff ];

frag_addr_match:
     [ addr_match+ ];
       addr_cmp-   ;
     [ addr_match- ];
       gmv_ack+    ;
     [ gmv-        ];
       bus_req-    ;
       gmv_ack-    ;

frag_addr_diff:
     [ addr_diff+ ];
       addr_cmp-   ;
     [ addr_diff- ];
       gmv_ack+    ;
     [ gmv-        ];
       gmv_ack-    ;
       P0
```

Listing 34. CSP implementation: sndr bus.

A.5 Sender component controller handling interfacing with CPU core.

```
module controller;

inputs mem_op cache_gnt cache_ack reg_ack cache_op1_ack cache_op2_read
   cache_op2_write cache_op2_mm transaction_ack done_ack;

outputs mem_op_ack cache_arb cache_req reg_req cache_op1 cache_op2
   transaction_req done;

explicit_place P0;

loop_place [mem_op_ack-]->[mem_op+];

marking [mem_op_ack-]->[mem_op+];

main sndr_core_if;

fragment frag_op2_write;
fragment frag_op2_mm;
fragment frag_op2_read;

endinterface

sndr_core_if:
    *[
        [ mem_op+          ];
          P0               ;
          cache_arb+       ;
        [ cache_gnt+       ];
          cache_op1+       ;
        [ cache_op1_ack+   ];
          cache_req+       ;
        [ cache_ack+       ];
          reg_req+         ;
        [ reg_ack+         ];
          reg_req-         ;
        [ reg_ack-         ];
          cache_req-       ;
        [ cache_ack-       ];
          cache_op1-       ;
        [ cache_op1_ack-   ];
          cache_op2+       ;
        [ frag_op2_write | frag_op2_mm | frag_op2_read ];
          done-            ;
        [ done_ack-        ];
          mem_op_ack+      ;
        [ mem_op-          ];
          mem_op_ack-      ;
    ]

frag_op2_write:
    [ cache_op2_write+ ];
      cache_req+       ;
    [ cache_ack+       ];
      cache_req-       ;
    [ cache_ack-       ];
      cache_arb-       ;
    [ cache_gnt-       ];
      done+            ;
    [ done_ack+        ];
      cache_op2-       ;
    [ cache_op2_write- ];

frag_op2_mm:
    [ cache_op2_mm+    ];
      cache_arb-       ;
```

```
    [ cache_gnt-        ];
      transaction_req+  ;
    [ transaction_ack+  ];
      transaction_req-  ;
    [ transaction_ack-  ];
      cache_op2-        ;
    [ cache_op2_mm-     ];
      P0

frag_op2_read:
    [ cache_op2_read+ ];
      cache_arb-      ;
    [ cache_gnt-      ];
      done+           ;
    [ done_ack+       ];
      cache_op2-      ;
    [ cache_op2_read- ];
```

Listing 35. CSP implementation: sndr core iface.

A.6 Sender component controller handling message collision detection.

```
module controller;

inputs log_check gmv ack_sndr in_log_y in_log_n addr_same addr_diff;

outputs log_clr log_abrt gmv_ack req_sndr in_log addr_cmp;

explicit_place P0;
explicit_place P1;
explicit_place P2;

main sndr_logcheck;

fragment frag_log_check_segment;
fragment frag_gmv_segment;

fragment frag_in_log_y;
fragment frag_in_log_n;
fragment frag_addr_same;
fragment frag_addr_diff;

loop_place [log_abrt- log_clr-]->[log_check+];

marking [log_abrt- log_clr-]->[log_check+];
marking [log_abrt- gmv_ack- gmv-]->[gmv+];

endinterface

sndr_logcheck:
    frag_log_check_segment : frag_gmv_segment

frag_log_check_segment:
    *[
        [ log_check+ ];
          req_sndr+   ;
        [ ack_sndr+  ];
          in_log+     ;
        [ frag_in_log_y | frag_in_log_n ];
    ]

frag_in_log_y:
    [ in_log_y+    ];
      in_log-      ;
    [ in_log_y-    ];
      req_sndr-    ;
    [ ack_sndr-    ];
      P0           ;
  P2=>addr_cmp+    ;
    [ frag_addr_same | frag_addr_diff ];

frag_in_log_n:
    [ in_log_n+  ];
      in_log-    ;
    [ in_log_n-  ];
      req_sndr-  ;
    [ ack_sndr-  ];
      log_clr+   ;
    [ log_check- ];
      log_clr-   ;

frag_addr_same:
    [ addr_same+    ];
      addr_cmp-     ;
    [ addr_same-    ];
      gmv_ack+      ;
    [ gmv-          ];
```

```
        gmv_ack-          ;
      [ log_abrt+       ];
        log_check-        ;
      [ log_abrt-=>P1   ];

frag_addr_diff:
      [ addr_diff+      ];
        addr_cmp-         ;
      [ addr_diff-      ];
        gmv_ack+          ;
      [ gmv-            ];
        gmv_ack- =>P0     ;
        P1

frag_gmv_segment:
        P1
      [ gmv+ ];
        P2
      [ gmv- ];
        P1
```

Listing 36. CSP implementation: sndr logcheck.

A.7 Sender component main controller

```
module controller;

inputs msg_valid transaction_req log_clr log_abrt send_ack;

outputs msg_ack transaction_ack log_check send;

explicit_place P0;

loop_place [msg_ack-]->[msg_valid+];
loop_place [transaction_ack-]->[transaction_req+];

marking [msg_ack-]->[msg_valid+];
marking [transaction_ack-]->[transaction_req+];
marking [msg_ack- transaction_ack-]->[msg_ack+ log_check+];

main sndr_main;

fragment frag_msg_ack_flow;
fragment frag_transaction_flow;

endinterface

sndr_main:
    frag_msg_ack_flow : frag_transaction_flow

frag_msg_ack_flow:
    *[
        [ msg_valid+   ];
      P0=>msg_ack+     ;
        [ msg_valid-   ];
          msg_ack-=>P0 ;
    ]

frag_transaction_flow:
    *[
        [ transaction_req+ ];
      P0=>log_check+       ;
        [(
            [ log_abrt+ ];
              log_check- ;
            [ log_abrt- ];
         ) | (
            [ log_clr+  ];
              send+      ;
            [ send_ack+ ];
              send-      ;
            [ send_ack- ];
              log_check- ;
            [ log_clr-  ];
         )
        ];
          transaction_ack+       ;
        [ transaction_req-     ];
          transaction_ack- =>P0  ;
    ]
```

Listing 37. CSP implementation: sndr main.

Appendix B
COMPLETE TEST RESULTS

Table 3
Full Test Results

Model	Test	Duration [ns]	Pwr PTPX [mW]	Pwr Calc. [mW]	Energy [J]	Area [μm^2]
async1	case00_t1	10	0.1548	0.1522	$1.522 * 10^{-12}$	**759.890**
	case00_t2	**18.758**	0.2974	0.2997	$5.621 * 10^{-12}$	
	case00_t3	**15.393**	0.2123	0.2096	$3.226 * 10^{-12}$	
	pwr_t1	1000	0.1542	0.1528	$1.526 * 10^{-10}$	
	pwr_t2	**4709.420**	**0.3245**	0.3273	$1.542 * 10^{-9}$	
	pwr_t3	**7974.899**	0.2936	0.2970	$2.369 * 10^{-9}$	
async2	case00_t1	10	0.1548	0.1522	$1.522 * 10^{-12}$	759.890
	case00_t2	15.247	0.3298	0.3320	$5.061 * 10^{-12}$	
	case00_t3	13.449	0.2397	0.2363	$3.178 * 10^{-12}$	
	pwr_t1	1000	0.1542	0.1528	$1.528 * 10^{-10}$	
	pwr_t2	3592.381	0.3647	0.372	$1.336 * 10^{-9}$	
	pwr_t3	6386.365	0.3337	0.3379	$2.158 * 10 * -9$	
sync1	case00_t1	10	5.253	5.126	$5.125 * 10^{-11}$	1053.082
	case00_t2	**5.301**	5.618	5.550	$2.942 * 10^{-11}$	
	case00_t3	**12.340**	5.357	5.243	$6.470 * 10^{-11}$	
	pwr_t1	1000	5.311	5.159	$5.159 * 10^{-9}$	
	pwr_t2	**1349.901**	8.851	5.742	$7.752 * 10^{-9}$	
	pwr_t3	**2199.141**	5.786	5.881	$1.293 * 10^{-8}$	
sync2	case00_t1	10	1.619	1.628	$1.602 * 10^{-11}$	**944.849**
	case00_t2	18.551	1.738	1.734	$3.217 * 10^{-11}$	
	case00_t3	18.094	1.660	1.655	$2.995 * 10^{-11}$	
	pwr_t1	1000	1.651	1.628	$1.628 * 10^{-9}$	
	pwr_t2	4724.651	**1.784**	1.805	$8.528 * 10^{-9}$	
	pwr_t3	7696.845	1.757	1.793	$1.380 * 10^{-8}$	

List of the MESI controller device build configurations:

1 **async1:** Asynchronous design with environment response delay matched single clock cycle time of design sync1.

2 **async2:** Asynchronous design with environment response delay of 1ps to prevent environment interfering with power calculation.

3 **sync1:** Synchronous design with clock cycle of 200ps (5GHz).

4 **sync2:** Synchronous design with total time required to execute pwr_t2 test case matched async1 through extending the clock time. The sync2 design is used to obtain power results by equating the timeframe in which the energy is counted for both designs.

List of the specific test runs performed on the evaluated model of MESI controller:

1 **case00_t1:** Idle test case, no bus or cpu transaction only responder crawling through pending log, fixed duration 10ns.

2 **case00_t2:** Test case where core requests memory address not available in local cache thus triggering entire request transaction over the bus with response.

3 **case00_t3:** Test case where incoming bus message requests address present in local cache thus triggering response.

4 **pwr_t1:** Same as case00_t1 but with duration of 1000ns.

5 **pwr_t2:** Same as case00_t2 times 250.

6 **pwr_t3:** Same as case00_t3 times 250.

Cell area sizes of sync2 design:

- Before P&R (no clock tree): 751.927μm^2
- After P&R (with added clock tree): 944.849μm^2
- Percentage of the area as clock tree: 20.4183%

Cell area sizes of async design:

- Without delay matching elements: 710.816μm^2
- After delay matching elements added: 759.899μm^2
- Percentage of the area as delay matching elements: 6.45804%

Appendix C
REFERENCE CSP TO STG PARSER SOURCE CODE

The reference concept preview implementation of the CSP to STG parser is available under the URL: https://github.com/tchad/CSP_to_STG_parser.