Tampere University

Roni Hämäläinen

SYNTHESIS OF DIGITAL QUASI-DELAY-INSENSITIVE GREATEST COMMON DIVISOR CIRCUIT

Bachelors Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Sakari Lahti
September 2024

ABSTRACT

Roni Hämäläinen: Synthesis of Digital
Quasi-Delay-Insensitive Greatest
Common Divisor Circuit
Bachelors Thesis
Tampere University
Electrical Engineering
September 2024

This work investigates the design of digital asynchronous quasi-delay-insensitive circuits with a focus on the methodology developed by Alain Martin [22]. While synchronous circuits have a global clock signal that drives the state machines forward, asynchronous counterparts use handshaking protocols. The lack of a global clock can offer multiple benefits, including lower power consumption, higher performance, plug-and-play modularity and reduced electromagnetic emissions. Even though multiple asynchronous processors and other circuits have been designed over the years, large-scale adoption by industry has been lacking due to lack of tools.

A high-level overview of quasi-delay-insensitive circuits and Martin's synthesis methodology is presented. The circuit is designed in multiple phases, starting from creating a specification via requirements gathering. Then the specification is implemented using a language called the Communicating Hardware Processes. A series of semantics preserving model transformations are executed until a CMOS netlist is obtained. Methodology is applied by synthesizing a greatest common divisor circuit by hand until a component netlist is obtained.

Keywords: asynchronous, digital, circuit, design, quasi-delay-insensitive

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Roni Hämäläinen: Kvasiviivesietoisen suurimman yhteisen tekijän ratkaisevan piirin synteesi
Kandidaatintyö
Tampereen yliopisto
Sähkötekniikka
Syyskuu 2024

Työssä tarkastellaan digitaalisten asynkronisten kvasiviivesietoisten piirien suunnittelua keskittymällä Alain Martinin kehittämään metodologiaan [22]. Kun synkronisissa piireissä globaali kellosignaali ajaa tilakoneiden tilamuutoksia eteenpäin, asynkronisissa piireissä käytetään kättelyprotokollia. Globaalin kellosignaalin puuttuminen voi tarjota useita etuja, kuten matalamman tehonkulutuksen, korkeamman suorituskyvyn, plug-and-play -modulaarisuuden ja vaimeammat elektromagneettiset emissiot. Vaikka vuosien aikana on suunniteltu useita asynkronisia prosessoreita ja muita piirejä, teollisuus ei ole ottanut käyttöön asynkronista suunnittelua laajassa mittakaavassa.

Työssä esitellään korkean tason kuvaus kvasiviivesietoisista piireistä sekä Martinin synteesimetodologia. Piiri suunnitellaan useassa vaiheessa, aloittaen vaatimusten keräämisestä spesifikaatiota varten. Seuraavaksi spesifikaatio kirjoitetaan Communicating Hardware Processes -nimisellä mallinnuskielellä. Lopulta suoritetaan sarja semantiikan säilyttäviä mallimuunnoksia, kunnes saavutetaan CMOS-taso. Metodologiaa sovelletaan suorittamalla suurimman yhteisen tekijän ratkaisevan piirin synteesi käsin, kunnes saavutetaan komponenttitaso.

Avainsanat: asynkroninen, digitaalinen, piiri, suunnittelu, kvasiviivesietoisuus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

CONTENTS

1. INTRODUCTION

After George Boole introduced the Boolean algebra and Claude Shannon proved that digital circuits can execute Boolean logic, the information age started [28]. Over the years, the synchronous design paradigm, where a global clock signals is distributed throughout the circuit to synchronize the state transitions, has consolidated its position as the dominant way to design digital state machines [1, 2, 5]. The synchronous model discretizes the time domain which significantly simplifies the design process [33]. At least to a point.

Careful timing is required to ensure that the clock signal reaches all flip-flops at approximately the same time to avoid glitches and metastability [35]. Sometimes the clock frequency is the limiting factor for the total execution time of an algorithm [5]. Multiple factors limit the maximum achievable clock frequency. The clock signal's period is determined with the worst-case delay between two registers [5]. Also, driving a clock signal distribution network is power intensive which heats the circuit due to resistive losses. If this heat is not efficiently dissipated, a too high temperature can lead to material degradation which can reduce the reliability of the device. As it turns out, the power consumption increases with clock frequency [36]. Since the state transitions are synchronized, the majority of electromagnetic noise is emitted at the clock frequency and its harmonics which can disrupt electrical devices [2] and opens a vector for side-channel attacks [29]. In systems with multiple clock domains with different clock frequencies and phases, the communication over these domain crossings require special precautions to ensure that the data is synchronized with the receiving domain's frequency and phase [33].

Asynchronous circuits offer an alternative for the "tyranny of the clock" [33]. As Sutherland puts it, "instead of making all logic 'march to an external drum beat,' let us allow each logic element to proceed at its own pace" [33]. In asynchronous circuits the state transitions happen in a self-timed manner. State-holding elements communicate via channels using handshaking protocols which ensures that the transaction happens only if the sender is ready to transmit and the receiver is ready to receive. The lack of global clock signal promises and demonstratably can offer lower power consumption [3, 6], higher performance [6], plug-and-play modularity [11] and reduced electromagnetic emissions [3, 6]. For example, an asynchronous equivalent of synchronous ARM996HS processor consumes one-third of the power with lower electromagnetic emissions [3]. Asynchronous equivalent of synchronous 80C51 microcontroller consumes one-third of the power, is capable to provide 2,5 times the performance, has five times smaller current peaks and is able to operate correctly with wider range of supply voltage [6]. Even though asynchronous circuits have been studied since the 1950s and multiple chips have been realized throughout the years, the lack of tools has prevented their large-scale entry to industry.

The broad, guiding research question of this work is "how to design and implement a digital asynchronous circuit that can execute computation". This work simplifies the problem by focusing on the quasi-delay-insentitive synthesis methodology developed by Martin in [22]. Using this methodology, a greatest common divisor circuit is synthesized from specification to component netlist by hand.

First the theory of quasi-delay-insentitive circuits is presented in chapter 2. Then the synthesis methodology is presented in chapter 3. The synthesis of the greatest common divisor circuit is presented in chapter 4. The work is concluded in chapter 5.

2. QUASI-DELAY-INSENSITIVE ASYNCHRONOUS DIGITAL CIRCUITS

This section first presents the motivation to study quasi-delay-insensitive (QDI) asynchronous digital circuits (QDIADC) and then how QDIADCs are modeled. This work focuses to the conceptual framework developed by Alain Martin and Rajit Manohar et. al. The framework is extensive and this work only presents a small subset of it. A comprehensive description is found in [21, 22].

2.1 Motivation

ADCs have potential for lower and steadier power consumption compared to synchronous digital circuits (SDC). In SDC the total time required to execute an algorithm is $T_{total} = KT_{clock}$ where K is the amount of clock cycles required by the algorithm and T_{clock} is the clock period. Thus, the total execution time is inversely proportional to the clock frequency. [5] The total power consumption of a SDC can be modeled as $P_{total} = P_{dynamic} + P_{static}$ where $P_{dynamic} = P_{switching} + P_{short\ circuit}$ where $P_{switching} = \alpha C V_{DD}^2 f$ where α is the activity factor[1], C is the switching capacitance, V_{DD} is the voltage source voltage and f is the clock frequency. Thus, the total power consumption is directly proportional to the clock frequency. [36] Since the global clock signal is distributed to the whole circuit, also the inactive parts of the circuit consume power. One way to reduce power consumption is clock gating which reduces the activity factor by blocking the clock signal from selected parts of the circuit. Even if the inactive parts are excluded with clock gating, the clock signal driver must still be powerful enough to provide a clock signal that reaches all parts of the circuit. While the subcircuits in SDCs are clock-driven, in ADCs they are demand-driven which means that inactive subcircuits stop switching. In SDCs the switching happens in precise frequency and a majority of energy is concentrated around the clock frequency and its harmonics. Electromagnetic noise which is concentrated on these frequencies can affect nearby analog circuits. Switching in ADCs is uncorrelated which results in more distributed noise spectrum and lower peak noise. [2, 32]

ADCs have potential advantage in performance and in modularity. Latency of an ADC can be lower than a SDC counterpart since the operating speed is determined by local

[1]Activity factor is the proportion of gates that switch.

latencies of the components instead of a global worst-case latency which is used to determine the clock signal's frequency. [2, 27, 32] ADC can stop functioning if one element stops functioning. Faulty element can be located by observing the state of the circuit. [27] Since the different components of an ADC communicate with handshakes and take care of their own internal timings, the ADCs can be more modular and composable compared to SDCs which require equal clock frequencies between components. [32] QDIADCs demonstrate robustness against variations in supply voltage, temperature and fabrication process parameters. [15]

2.2 Circuit as a network of concurrent processes

A QDIADC can be treated as a concurrent and distributed system [17] and can be modeled using a derivative of the Communicating Sequential Processes (CSP) [8] called the **Communicating Hardware Processes** (CHP). This section presents a subset of the CHP. A comprehensive description is presented in [21, 22].

A circuit is modeled as a set of concurrently executing processes $P_1 \parallel P_2 \parallel ... \parallel P_n$ where the || is the **parallel composition** operator. A **process** P executes a sequence of statements $P \equiv S_1; S_2; ...; S_n$ where the semicolon is the **sequential composition** operator. A **statement** either manipulates intra-process variables or controls the execution flow. CHP restricts CSP by only allowing Boolean data types. Other data types are derived from Booleans. Intra-process variables usually can not be shared with other processes. Intra-process variable assignment is performed with the $:=$-operator. Execution flow can be controlled with conditional branches. The **selection** command can be either deterministic $\left[G_1 \rightarrow S_1 \,[\!]\, G_2 \rightarrow S_2 \,[\!]\, ...\right]$ or non-deterministic $\left[G_1 \rightarrow S_1 | G_2 \rightarrow S_2 | ...\right]$ both of which contain **guarded commands** $G_i \rightarrow S_i$. Execution of selection blocks until any guard G_i is true and then continues to execute the corresponding statement S_i. Deterministic selection assumes that only one guard can be true at the same time while non-deterministic variant allows multiple guards to be true at the same time. Execution of $[G]$ is equal to $[G \rightarrow skip]$ which waits until G is true. The **repetition** command $*\ [S]$ is equivalent to $*\ [true \rightarrow S]$ which repeats the statement S forever. [20, 21, 22]

Two processes P_1 and P_2 can interact via a one-to-one **channel** (A, Q) with two **ports** A and Q. One-to-many and many-to-many channels are excluded from this work. Assume that P_1 owns A and P_2 owns Q. If A is **active** and Q is **passive**, then P_1 can initiate an interaction with A. The interaction is **pending** until P_2 completes it with Q. P_2 can **probe** Q with $\overline{Q}$ to check if an interaction is currently pending on A. Active port can not be probed. If the channel can carry data, then the interaction is a **communication action**. Otherwise, the interaction is a **synchronization action**. A communication action is an inter-process variable assignment where the sender P_1 writes a local variable x to A with $A!x$ and the receiver P_2 reads the value from Q to a local variable y with $Q?y$. Since a synchronization action does not deliver data, the direction of the action can sometimes be omitted. In this case the synchronization action is initiated with A and completed with Q. A **straight-line program** contains only a sequence of simple assignments, communication actions and synchronization actions. [15, 20, 24]

2.3 Circuit as a network of operators

An **operator** implements the Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ of n inputs and one output and can store state. All inputs and outputs of operators form the set of **variables** V. A single variable $x \in V$ is output of one operator O_x and input to one or many operators. An operator A with inputs x and y and output z can also be marked with $(x, y)Az$. An output of an operator must not be its own input, in other words, **self-looping** is not allowed. A **circuit** consists of $|V|$ operators. A **configuration** $c : V \rightarrow \mathbb{B}$ associates each variable $x \in V$ with a Boolean value $\mathbb{B} = \{0, 1\}$. The initial configuration of a circuit is C_0. A circuit communicates with surrounding environment by reading inputs and writing outputs. The **environment** is also considered to be a circuit, and it can react to the outputs by changing inputs. [17, 18, 21, 23]

The operating mode is a contract between the circuit and the environment. If a circuit operates in the **fundamental mode**, then if the environment changes one input signal, then it must wait until the circuit has stabilized. The circuit is stabilized when no internal switching happens and its outputs can no longer change. After the circuit is stabilized, then the environment can change one input signal again. If a circuit operates in the **input-output mode**, then the environment can change input signals at any time. [32] This work focuses on circuits which operate in the input-output mode.

The delay model is a set of timing assumptions about operator delays internal to the circuit. The circuit must preserve its correctness when these assumptions hold. The circuit is correct when it is hazard-free [9]. If a circuit uses the **bounded-delay**-model, then all operator delays have known upper bounds. If a circuit is **speed-independent**, then operator delays are unbounded, except wire delays are zero [27]. If a circuit is **delay-insensitive** (DI), then operator delays are unbounded. [32] This work focuses on DI circuits.

2.4 Operator as a production rule set

A **production rule** (PR) $B \rightarrow t$ contains a guard B and a transition t. A **guard** is a Boolean function which uses a subset of V. A **transition** is a simple assignment of a variable. A **simple assignment** of variable $x \in V$ is either $x\uparrow$ which sets the value of x to true and $x\downarrow$ which sets the value of x to false. The semantics of assignment can be expressed with Hoare triples $\{\neg x\}\ x\uparrow\ \{\diamond x\}$ and $\{x\}\ x\downarrow\ \{\diamond\neg x\}$ where $\diamond$-operator means that the next statement will hold eventually. If the guard B of a PR evaluates to true in configuration c, then the PR is **enabled** $c \vDash B$. Enabled PR is eventually **fired** when the transition is executed. Transition has a positive non-zero duration and is **executed** when the output variable value is updated. Firing is **effective** if transition changes the value of the output variable and **vacuous** if not. [17, 18, 23] If a transition t_1 makes guard of transition t_2 true, then t_2 is a **successor** of t_1. Transition t_2 **acknowledges** t_1 when t_2 is executed. [23] A PR can be modeled with CHP. For example, a PR $G \rightarrow x \uparrow$ can be modeled with a process $*\,[[G \rightarrow x \uparrow]]$.

The behavior of an operator can be specified with two **complementary** PRs (CPRs) $B_u \rightarrow z \uparrow$ and $B_d \rightarrow z \downarrow$ which modify the same variable with opposite effects. Operator is **combinational** if $B_u = \neg B_d$ holds and **state-holding** if $\neg B_u \wedge \neg B_d$ can hold. A set of PRs form a **production rule set** (PRS). [10, 17, 18, 23] If each PR is stable and CPRs are non-interfering[2], then the concurrent execution of PRS is equivalent to the following sequential execution: $*$ [select enabled PR; fire PR]. [21] For example, a PRS with $B_u \rightarrow x \uparrow$ and $B_d \rightarrow x \downarrow$ can be modeled with a process $*\,[[Bu \rightarrow x \uparrow]] \parallel *\,[[Bd \rightarrow x \downarrow]]$.

All circuits can be constructed from a set of **elementary operators** which contains the wire, the fork, the AND, the OR, the C-element, the arbiter and the synchronizer. The **wire** with $B_u \equiv x \rightarrow y \uparrow$ and $B_d \equiv \neg x \rightarrow y \downarrow$ is a one input and one output operator that connects two variables. The **fork** with $B_u \equiv x \rightarrow y \uparrow, z \uparrow$ and $B_d \equiv \neg x \rightarrow y \downarrow, z \downarrow$ is a one input and, exceptionally, a two output operator that connects three variables. The **AND** with $B_u \equiv x \wedge y \rightarrow z \uparrow$ and $B_d \equiv \neg x \vee \neg y \rightarrow z \downarrow$ is a two input and one output operator. The **OR** with $B_u \equiv x \vee y \rightarrow z \uparrow$ and $B_d \equiv \neg x \wedge \neg y \rightarrow z \downarrow$ is a two input and one output operator. The **C-element** with $B_u \equiv x \wedge y \rightarrow z \uparrow$ and $B_d \equiv \neg x \wedge \neg y \rightarrow z \downarrow$ is a two input and one output operator that can hold state. [20] If an operator is not a wire or fork, then it is a **gate** [9].

The **arbiter** provides non-deterministic choice between two true guards [21]. A simple example of a non-fair arbiter is a process P which communicates with the environment via two channels (A, A') and (B, B') is presented in eq. (2.1). A more fair variant is presented in eq. (2.2). [19] The more fair arbiter presented previously uses negated probes. While a probe $\overline{A}$ is stable since $\overline{A}$ stays true until A is completed, a negated probe $\neg\overline{A}$ is unstable since it can become true at any time. [24] The **synchronizer**, presented in eq. (2.3), accepts unstable guards. Variable b can change from false to true at any time. Both b and z must remain true until u or v has changed. First guard is stable and second guard is unstable. [21]

[2]Stability and non-interference are explained in section 2.5.

$$ARB_{simple} \equiv * [[\overline{A} \rightarrow A | \overline{B} \rightarrow B]] \tag{2.1}$$

$$ARB_{fairer} \equiv * [[\overline{A} \rightarrow A | \neg\overline{A} \rightarrow skip]; [\overline{B} \rightarrow B | \neg\overline{B} \rightarrow skip]] \tag{2.2}$$

$$SYNC \equiv * [[b \wedge z \rightarrow u \uparrow; [\neg z]; u \downarrow \ \ [] \ \neg b \wedge z \rightarrow v \uparrow; [\neg z]; v \downarrow]] \tag{2.3}$$

2.5 Hazard-free behavior with stability and non-interference

The failure modes of asynchronous circuits include interference and instability. Consider a gate O_x with CPRS guards B_u and B_d. If $B_u \wedge B_d$, then the gate experiences **inference**. Consider also present configuration $c_{present}$ and next configuration c_{next}. If (1) O_x is being pulled up in $c_{present}$, $c_{next}(x) \neq 1$ and O_x is not being pulled down in c_{next}, or (2) O_x is being pulled down in $c_{present}$, $c_{next}(x) \neq 0$ and O_x is not being pulled up in c_{next}, then the gate O_x is **unstable** in configurations $c_{present}$ and c_{next}. [9]

As explained in section 2.3, a DI circuit operates hazard-free assuming unbounded operator delays. If a circuit is stable and non-interfering, then it is a DI circuit [23]. Non-interference is a property of a CPRS. In some cases it is possible to prove that a CPRS is non-interfering by verifying that $B_u \wedge B_d \equiv \perp$ at all times. If all CPRS of a circuit are non-interfering, then the circuit is non-interfering. [18] Stability is a property of a circuit. If a circuit implements the **unique-successor-set theorem** (USS), then it is stable and DI. In short, the USS requires that each non-final transition of the same variable has the same successor set in all possible computations. [23]

The set of DI circuits is limited to inverters, wires and C-elements. For example, an AND-gate does not fulfill the USS. Assume that both inputs i_0 and i_1 are low. If input i_0 is set high, the output does not change. Assume that both inputs are high. If input i_0 is set low, the output changes. Thus, the successor set of transition of the input i_0 is not the same with all possible computations. This limitation is bypassed by weakening the requirement of unbounded operator delays by allowing **isochronic forks**. Assume a fork with input x and outputs y_1 and y_2. If the fork is not isochronic, then the delay between acknowledgents of y_1 and y_2 is unbounded. If the fork is isochronic, then if y_1 has acknowledged x, then y_2 is also assumed to have acknowledged x. A circuit which operates hazard-free assuming unbounded operator delays, but allowing isochronic forks, is **quasi-delay-insensitive** (QDI). [23] A QDI circuit can implement any Turing-computable function [16]. This work focuses on QDI circuits.

The **adversary path timing assumption** (APTA) is the weakest necessary and sufficient assumption to guarantee hazard-free operation of a QDI circuit. APTA requires that for each isochronic fork, the delay of the isochronic branch is shorter than the delay of the corresponding adversary path. [9, 26] Further elaboration is omitted from this work.

2.6 Implementing operator with CMOS

A combinational operator with CPRs $B_u \rightarrow z\uparrow$ and $B_d \rightarrow z\downarrow$ can be implemented with CMOS. The PR $B_u \rightarrow z\uparrow$ is implemented with the pull-up network and the PR $B_d \rightarrow z\downarrow$ with the pull-down network. Since the pull-up network of CMOS is implemented with PMOS, it requires negated inputs. Thus, inverters might be needed to the inputs or to the output. The combination of the pull-up and the pull-down networks is called the **restoring circuit**. The wire and the fork are only elementary operators implemented without a restoring circuit. The interference of CPRs manifests as a short circuit in CMOS where both the pull-up and the pull-down networks conduct at the same time. If a circuit is proven to be non-interfering, then a short circuit is avoided. [21]

A state-holding operator can be implemented as a dynamic or static circuit. Dynamic circuit assumes that the charge stored in the wires is refreshed often enough so that it does not completely fade due to leakage. Static circuit stores the output variable's state to a storage element, e.g. two cross-coupled inverters, called the **staticizer**, where the feedback inverter is weak. Since the feedback inverter maintains the output's current value, the current through the pull-up and the pull-down networks must be stronger than the current through the feedback inverter. This can be achieved by implementing the transistors of the feedback inverter with high resistance. The weak current must be strong enough to maintain the output value. This reliance on a two-sided equality requirement can be risky if the resistance value of the inverter has large variance in the manufacturing process. Other solutions for a staticizer exist. [21, 26] CMOS implementations for elementary operators are presented in [21].

3. CIRCUIT SYNTHESIS

A CHP program can be transformed into QDIAD CMOS [20]. This section briefly presents the major steps of this transformation. The details for each step can be found in [22]. First, the design entry is given with CHP. The communication actions contained in the CHP are then replaced with handshaking protocols during the **handshaking expansion** (HSE). The result is then transformed into PRS during the **production-rule expansion** which is finally transformed into CMOS. [21]

Three basic model transformations are CHP-to-HSE, HSE-to-PRS and PRS-to-CMOS as presented in eq. (3.1). Other transformations include e.g. CHP-to-CHP. [7] A transformation between two models is marked with the ⊳-operator. Since each model transformation preserves the semantics of the previous model, the final model should be correct in respect to the specification of the first model.

$$CHP \rhd HSE \rhd PRS \rhd CMOS \tag{3.1}$$

Other synthesis methods for ADCs exist [30]. One method synthesizes SI circuit from a signal transition graph (STG) which is a certain type of Petri net. STG is built from a set of separate STG fragments that describe signal transitions of the circuit. STG is transformed into a state graph which is then transformed into a set of Boolean equations which are mapped into transistors. [32] Other notable method uses asynchronous FSMs [37]. This work focuses on CHP-to-CMOS synthesis.

3.1 Design entry

The first step for the designer is to gather functional and non-functional requirements and restrictions into a specification. Specification is then implemented as a concurrent program with CHP. CHP program is then behaviorally verified using a CHP simulator to check its correctness. A correct program fulfills its specification. After verification succeeds, the next step is to synthesize a circuit from CHP.

3.2 CHP-to-CHP transformations

The synthesis starts with the **process decomposition** where complex processes are transformed into simpler processes. The process decomposition is performed using the **decomposition rule** [4]

$$S_1; S_2; S_3 \ \rhd_{decompose} \ S_1; A_2; S_3 \parallel (A_2/S_2) \tag{3.2}$$

where a process with arbitrary statements S_1, S_2 and S_3 is transformed into a set of two processes which communicate via a channel (A_2, Q_2). The execution of the active synchronization action A_2 can be interpreted as a **function call** since it triggers the execution of the new process. The callee is implemented with probe [4]

$$(A/S) \rhd \ *[[\overline{Q} \rightarrow S; Q]]. \tag{3.3}$$

Process decomposition can be used to simplify **sequencing** of statements into separate processes with transformation rule [22]

$$S_1; S_2 \rhd A_1 \parallel (A_1/A_{1-1}; A_2) \parallel (A_{1-1}/S_1) \parallel (A_2/S_2). \tag{3.4}$$

Sequencing introduces processes with form $(L/A; R)$ where L, A and R are active synchronization actions. This form can be implemented with wire $(l_i)w(a_o)$ and D-element $(a_i, l_o)D(r_i, r_o)$. [22] Decomposition is applied until right-hand side of each guarded command is a straight-line program. [20] Various semantics-preserving optimizations can be applied to the CHP program at this stage. [22] Presentation of these optimizations is omitted from this work.

3.3 CHP-to-HSE transformation

The **4-phase handshaking protocol** is used to implement synchronization and communication actions between sender and receiver. Assuming a synchronization channel, the channel between the sender and the receiver contains two wires, the request-wire and the acknowledge-wire. A transaction begins when the sender sets the request-wire to high. After the receiver detects the request-signal it sets the acknowledge-wire to high. The sender detects the acknowledge-signal and sets the request-wire to low. Finally, the receiver detects low request-signal and sets the acknowledge-wire to low. Both parties are now ready for next transaction. [22]

Each communication and synchronization action is transformed into 4-phase handshaking protocol implementation during **handshaking expansion** (HSE). Transformation rules for an active action A, a lazy-active action A_{lazy}, a passive action Q and a probe $\overline{Q}$ are [22]

$$\begin{aligned}
A &\rhd_{chp-to-hse} \; a_o\uparrow; [a_i]; a_o\downarrow; [\neg a_i] && (3.5)\\
A_{lazy} &\rhd_{chp-to-hse} \; [\neg a_i]; a_o\uparrow; [a_i]; a_o\downarrow && (3.6)\\
Q &\rhd_{chp-to-hse} \; [q_i]; q_o\uparrow; [\neg q_i]; q_o\downarrow; && (3.7)\\
\overline{Q} &\rhd_{chp-to-hse} \; [q_i]. && (3.8)
\end{aligned}$$

Transformation rules for communication actions are more complex and omitted from this work. The HSE form can be optimized for throughput, latency or size by **reshuffling** which means changing the order of port actions. For example, a process 3.9 can be reshuffled into 3.10 while preserving its semantics. [22]

$$\begin{aligned}
(A_1/A_2) \equiv &\quad * [[q_{1-i}]; a_{2-o}\uparrow; [a_{2-i}]; a_{2-o}\downarrow; [\neg a_{2-i}]; q_{1-o}\uparrow; [\neg q_{1-i}]; q_{1-o}\downarrow] && (3.9)\\
\rhd_{reshuffle} &\quad * [[q_{1-i}]; a_{2-o}\uparrow; q_{1-o}\uparrow; [\neg q_{1-i}]; [a_{2-i}]; a_{2-o}\downarrow; [\neg a_{2-i}]; q_{1-o}\downarrow] && (3.10)
\end{aligned}$$

Presenting the rules for reshuffling are omitted from this work. An analysis of correctness and performance of various reshufflings is presented in [13]. An analysis of different reshufflings for a SISO buffer $* \; [L; R]$ is presented in [12]. Another transformation is **process factorization** which decomposes processes on synchronization actions to multiple processes where each process manages one output variable. Presenting the rules for factorization are omitted from this work. For example, a process 3.11 can be factorized into 3.13 while preserving its semantics. [22]

$$\begin{aligned}
* [A_1; A_2] \equiv &\quad * [a_{1-o}\uparrow; [a_{1-i}]; a_{1-o}\downarrow; [\neg a_{1-i}]; a_{2-o}\uparrow; [a_{2-i}]; a_{2-o}\downarrow; [\neg a_{2-i}]] && (3.11)\\
\rhd_{reshuffle} &\quad * [a_{1-o}\uparrow; [a_{1-i}]; a_{2-o}\uparrow; [a_{2-i}]; a_{1-o}\downarrow; [\neg a_{1-i}]; a_{2-o}\downarrow; [\neg a_{2-i}]] && (3.12)\\
\rhd_{factorize} &\quad * [a_{1-o}\uparrow; [a_{2-i}]; a_{1-o}\downarrow; [\neg a_{2-i}]] \parallel * [[a_{1-i}]; a_{2-o}\uparrow; [\neg a_{1-i}]; a_{2-o}\downarrow;] && (3.13)
\end{aligned}$$

3.4 HSE-to-PRS transformation

HSE form is transformed into PRS during **production rule expansion** using rule [22]

$$* [[w_1]; t_1; \; \ldots \; [w_n]; t_n] \rhd_{hse-to-prs} \{b_1 \to t_1, \; \ldots \; b_n \to t_n\}. \qquad (3.14)$$

The resulting PRS must fulfill three properties. If Boolean expression b_i evaluates true, then the corresponding wait-condition w_i must also evaluate true. Only one PR can be enabled at any time. This property is known as **sequential execution**. If wait-condition w_{i+1} holds after t_i, then b_{i+1} must hold after t_i. This property is known as **program-order execution**.

These three properties are also expressed with eqs. (3.15) to (3.17). [22]

$$b_i \Rightarrow w_i \tag{3.15}$$

$$b_i \Rightarrow \neg b_j, i \neq j \tag{3.16}$$

$$\{\neg w_{i+1}\}t_i\{w_{i+1}\} \Rightarrow \{\neg b_{i+1}\}t_i\{b_{i+1}\} \tag{3.17}$$

If two or more configurations are equal in different parts of the program, then the circuit can not distinguish between them and multiple PRs can become enabled at the same time, violating both the sequential execution and the program-order execution properties. Consider an active-active buffer in HSE form in eq. (3.19). Hoare triples for transitions $l_o \downarrow$ and $r_o \downarrow$ are $\{\neg l_o \wedge l_i \wedge \neg r_o \wedge \neg r_i\} l_o \downarrow \{\neg l_o \wedge \diamond \neg l_i \wedge \neg r_o \wedge \neg r_i\}$ and $\{\neg l_o \wedge \neg l_i \wedge \neg r_o \wedge r_i\} r_o \downarrow \{\neg l_o \wedge \neg l_i \wedge \neg r_o \wedge \diamond \neg r_i\}$. Both postconditions are eventually equal. This means that existing variables can not be used to distinguish these postconditions as separate states. States can be separated by adding a **state variable** during **state assignment** as is done in eq. (3.20). Resulting triples are now $\{x \wedge \neg l_o \wedge l_i \wedge \neg r_o \wedge \neg r_i\} l_o \downarrow \{x \wedge \neg l_o \wedge \diamond \neg l_i \wedge \neg r_o \wedge \neg r_i\}$ and $\{\neg x \wedge \neg l_o \wedge \neg l_i \wedge \neg r_o \wedge r_i\} r_o \downarrow \{\neg x \wedge \neg l_o \wedge \neg l_i \wedge \neg r_o \wedge \diamond \neg r_i\}$. [22, 25]

After state assignment, each configuration can be distinguished from each other using program variables. Next step is **guard strengthening** where the guards' predicates are expanded until properties given in eqs. (3.16) and (3.17) are fulfilled as is done in eq. (3.21). [22, 25]

$$* [L; R] \tag{3.18}$$

$$\triangleright_{chp-to-prs} \quad * [l_o \uparrow; [l_i]; l_o \downarrow; [\neg l_i]; r_o \uparrow; [r_i]; r_o \downarrow; [\neg r_i]] \tag{3.19}$$

$$\triangleright_{state-assignment} \quad * [l_o \uparrow; [l_i]; x \uparrow; l_o \downarrow; [\neg l_i]; r_o \uparrow; [r_i]; x \downarrow; r_o \downarrow; [\neg r_i]] \tag{3.20}$$

$$\triangleright_{strengthening} \quad * [l_o \uparrow; [l_i]; x \uparrow; [x]; l_o \downarrow; [x \wedge \neg l_i]; r_o \uparrow; [r_i]; x \downarrow; [\neg x]; r_o \downarrow; [\neg x \wedge \neg r_i]] \tag{3.21}$$

3.5 PRS-to-CMOS transformation

PRS is transformed into a set of operators during the **operator reduction**. PRs which assign to same variable are identified, grouped and replaced with elementary operators. [22] Sometimes it is possible to transform state-holding operators into combinational ones via **symmetrization**. Consider a CPRS given in eq. (3.22). If e.g. guard b_1 can be expressed using a complement of guard b_2 and an arbitrary Boolean x, then the CPRS can be redefined into eq. (3.23) where $B \equiv b_2$. If $B \Rightarrow \neg x$, then the second guard can be replaced with $\neg x \vee B$. If invariant $x \vee B \vee \neg z$ holds, then no new effective firings have been introduced. [14, 22]

$$\{b_1 \rightarrow z\uparrow,\ b_2 \rightarrow z\downarrow\} \tag{3.22}$$

$$\equiv \quad \{x \wedge \neg\boldsymbol{B} \rightarrow z\uparrow,\ \boldsymbol{B} \rightarrow z\downarrow\} \tag{3.23}$$

$$\vartriangleright_{symmetrize} \{x \wedge \neg\boldsymbol{B} \rightarrow z\uparrow,\ \neg x \vee \boldsymbol{B} \rightarrow z\downarrow\} \tag{3.24}$$

4. SYNTHESIS OF GCD CIRCUIT

The CHP program presented in listing 1 solves the greatest common divisor of two integers x and y using the Euclidean algorithm. The program communicates with environment via two passive input ports Q_{l-x} and Q_{l-y} and one active output port A_{r-x}. Local variables x and y are encoded with binary encoding. This program is expressed as a process P in 4.1.

```
*[
    Q_l_x?x;
    Q_l_y?y;
    *[
            x > y -> x := x - y
        [] x < y -> y := y - x
    ];
    A_r_x!x
]
```

Listing 1. *CHP program for solving the greatest common divisor*

Left environment is formed by processes 4.2 and 4.3 and right by process 4.4. Previously mentioned ports are connected to environment via channels (A_{l-x}, Q_{l-x}), (A_{l-y}, Q_{l-y}) and (A_{r-x}, Q_{r-x}). Expressions is_valid and is_empty are explained in section 4.2. The syntax $A \Uparrow$ means that the data path of port A is set to a valid value and $A \Downarrow$ means that the data path is cleared.

$$P \equiv *\,[Q_{l-x}?x; Q_{l-y}?y; *\,[x > y \rightarrow x := x - y \,[]\, x < y \rightarrow y := y - x]; A_{r-x}!x] \quad (4.1)$$

$$P_{l-x} \equiv *\,[A_{l-x-o} \Uparrow; [A_{l-x-i}]; A_{l-x-o} \Downarrow; [\neg A_{l-x-i}]] \quad (4.2)$$

$$P_{l-y} \equiv *\,[A_{l-y-o} \Uparrow; [A_{l-y-i}]; A_{l-y-o} \Downarrow; [\neg A_{l-y-i}]] \quad (4.3)$$

$$P_{r} \equiv *\,[[is\_valid(Q_{r-x-i})]; Q_{r-x-o} \uparrow; [is\_empty(Q_{r-x-i})]; Q_{r-x-o} \downarrow] \quad (4.4)$$

P with its environment, signals and connections is presented in fig. 4.1. Both input ports and the output port communicate data encoded with one-hot encoding. Data path signals and variables are named using syntax $name[index]_{value}$, where $name$ is the name of the

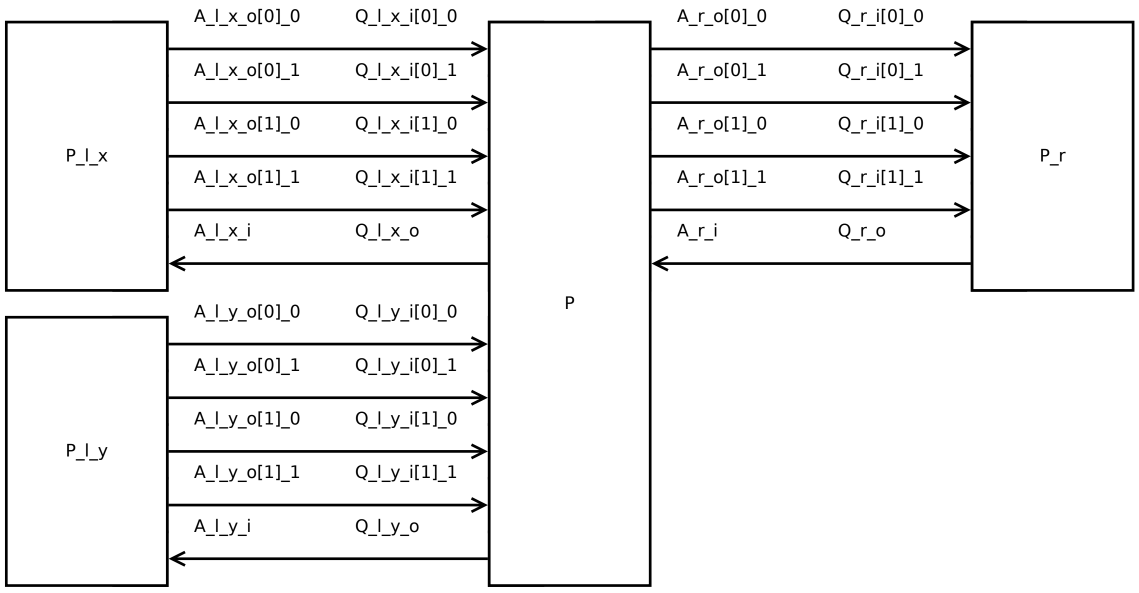

Figure 4.1. *The process P with its environment*

signal or variable, $index$ is the index of the bit and $value$ is either 0 or 1 indicating the one-hot encoded value. For example, consider a one-hot encoded variable x with width of two bits. Now x is represented by $x[0]_0$, $x[0]_1$, $x[1]_0$ and $x[1]_1$. Indexed variable $x[0]$ represents the set of $x[0]_0$ and $x[0]_1$.

4.1 Verification with ACT

The program was simulated and verified using the Asynchronous Circuit Toolkit (ACT) which is an open source electronic design automation toolkit developed by Manohar and his asynchronous VLSI and architecture research group at Yale. The design under test is presented in listing 2 in appendix A. The model was verified using a testbench which is also written using CHP as presented in listing 3 in appendix B. A process called `test` contains instances of the GCD, a source and a sink. The source feeds test data and its channels `X` and `Y` are connected to the GCD's corresponding channels. The GCD's result channel is connected to sink which consumes the results. The testbench was simulated using the `actsim`-tool. Sample output is presented in listing 4 and a generated waveform in figure B.1 in appendix B. The design works as expected based on the results. [1, 31, 34]

4.2 Useful definitions

In this section a set of useful expressions is defined. Also, an important construct $(L/A; R)$ is expanded. Following expressions take an input and return either $true$ or $false$. The type of input x is marked with $x : type$. If $type$ is one-hot oh, then x is composed of x_0 and x_1. If $type$ is binary bin, then x is composed of just x. An array can be marked with

$x : type[n]$ where n is the amount of inputs with given type.

$$is\_valid(x : oh) \equiv x_0 \oplus x_1 \tag{4.5}$$

$$is\_empty(x : oh) \equiv \neg x_0 \wedge \neg x_1 \tag{4.6}$$

$$is\_low(x : oh) \equiv x_0 \wedge \neg x_1 \tag{4.7}$$

$$is\_high(x : oh) \equiv \neg x_0 \wedge x_1 \tag{4.8}$$

$$are\_valid(X : oh[2]) \equiv is\_valid(X[0]) \wedge is\_valid(X[1]) \tag{4.9}$$

$$are\_empty(X : oh[2]) \equiv is\_empty(X[0]) \wedge is\_empty(X[1]) \tag{4.10}$$

A common process produced by process decomposition is $(L/A; R)$ where A and R are active synchronization actions. A transformation chain from CHP to operators is given in eqs. (4.11) to (4.16). [22]

$$(L/A; R) \tag{4.11}$$

$$\rhd_{chp-to-hse} \ * [[l_i]; U_a; D_a; U_r; D_r; l_o \uparrow; [\neg l_i]; l_o \downarrow] \tag{4.12}$$

$$\rhd_{reshuffle} \ * [[l_i]; U_a; D_a; l_o \uparrow; [\neg l_i]; U_r; D_r; l_o \downarrow] \tag{4.13}$$

$$\rhd_{factorize} \ * [[l_i]; a_o \uparrow; [\neg l_i]; a_o \downarrow] \parallel * [[a_i]; l_o \uparrow; [\neg a_i]; R; l_o \downarrow] \tag{4.14}$$

$$\rhd_{hse-to-prs} \ \{l_i \to a_o \uparrow; \neg l_i \to a_o \downarrow; a_i \to l_o \uparrow; \neg a_i \to r_o \uparrow; r_i \to r_o \downarrow; \neg r_i \to l_o \downarrow\} \tag{4.15}$$

$$\rhd_{prs-to-ops} \ \{(l_i)w(a_o), (a_i, l_o)D(r_i, r_o)\} \tag{4.16}$$

4.3 Analysis of process P

Process decomposition is applied to process P to obtain following set of processes.

$$P \equiv * [A_1] \tag{4.17}$$

$$P_1 \equiv (A_1/A_2; A_3) \tag{4.18}$$

$$P_2 \equiv (A_2/Q_{l-x}?x) \tag{4.19}$$

$$P_3 \equiv (A_3/A_4; A_5) \tag{4.20}$$

$$P_4 \equiv (A_4/Q_{l-y}?y) \tag{4.21}$$

$$P_5 \equiv (A_5/A_6; A_7) \tag{4.22}$$

$$P_6 \equiv (A_6/ * [x > y \to x := x - y \,[\!]\, x < y \to y := y - x]) \tag{4.23}$$

$$P_7 \equiv (A_7/A_{r-x}!x) \tag{4.24}$$

Using temporary signal renames $A_{1-i} \equiv a_i$ and $A_{1-o} \equiv a_o$, the process P can be transformed into an inverter in following transformation chain.

$$
\begin{aligned}
P \rhd_{chp-to-hse} &\ * [a_o \uparrow; [a_i]; a_o \downarrow; [\neg a_i]] && (4.25)\\
\rhd_{reshuffle} &\ * [[\neg a_i]; a_o \uparrow; [a_i]; a_o \downarrow] && (4.26)\\
\rhd_{hse-to-prs} &\ \{a_i \rightarrow a_o \downarrow, \neg a_i \rightarrow a_o \uparrow\} && (4.27)\\
\rhd_{prs-to-ops} &\ \{(a_i) inv(a_o)\} && (4.28)
\end{aligned}
$$

Processes P_1, P_3 and P_5 have the form $(L/A; R)$ and are implemented using the same flow as presented in section 4.2. Rest of the processes are expanded in following sections.

4.4 Analysis of processes P_2 and P_4

In this section processes P_2 and P_4 are expanded. Both processes are symmetric in structure, but P_2 reads from channel (A_{l-x}, Q_{l-x}) and writes to local variable x while P_4 reads from (A_{l-y}, Q_{l-y}) and writes to y. To simplify the analysis, processes and ports are temporarily renamed with definitions $P_2 \equiv P_x$, $P_4 \equiv P_y$, $A_2 \equiv A_x$, $A_4 \equiv A_y$.

Following high-level explanation applies to both processes. Input data path width is two bits with one-hot encoding. The process must first verify that the input provided by left environment is valid. After valid input is received, the next step is to assign the input to the local variable. Since two bits are delivered, the process can read and store these inputs in sequential order or in parallel. After input has been stored to the local variable, the process must acknowledge the left environment. The left environment then clears the input and the process finishes the transaction. Finally, control is returned to the call site.

The previous high-level explanation is implemented for P_x in following pseudocode. Functions and expressions are_valid, are_empty, is_low and is_high are shortened into av, ae, lo and hi to save space. Statements $store[0]$ and $store[1]$ are elaborated later.

$$P_x \equiv (A_1/[av(Q_{l-x-i})]; store[0]; store[1]; Q_{l-x-o} \uparrow; [ae(Q_{l-x-i})]; Q_{l-x-o} \downarrow) \qquad (4.29)$$

P_x can be decomposed into following processes. Process P_{x-6} was not decomposed further since its implementation is already simple as is.

$$P_x \equiv (A_x / A_{x-1}; A_{x-2}) \tag{4.30}$$
$$P_{x-1} \equiv (A_{x-1} / [av(Q_{l-x-i})]) \tag{4.31}$$
$$P_{x-2} \equiv (A_{x-2} / A_{x-3}; A_{x-4}) \tag{4.32}$$
$$P_{x-3} \equiv (A_{x-3} / store[0]) \tag{4.33}$$
$$P_{x-4} \equiv (A_{x-4} / A_{x-5}; A_{x-6}) \tag{4.34}$$
$$P_{x-5} \equiv (A_{x-5} / store[1]) \tag{4.35}$$
$$P_{x-6} \equiv (A_{x-6} / Q_{l-x-o} \uparrow; [ae(Q_{l-x-i})]; Q_{l-x-o} \downarrow) \tag{4.36}$$

Processes P_x, P_{x-2} and P_{x-4} have the form $(L/A;R)$ and are implemented using the same flow as presented in section 4.2. Transformation chain for P_{x-1} is given next with temporary renames $Q_{x-1-i} \equiv q_i$, $Q_{x-1-o} \equiv q_o$ and $av(Q_{l-x-i}) \equiv B$. Symmetrization is possible since invariant $B \vee q_i \vee \neg q_o$ holds.

$$P_{x-1} \rhd_{chp-to-hse} \quad *[[[q_i];[B];q_o \uparrow;[\neg q_i];q_o \downarrow]] \tag{4.37}$$
$$\rhd_{simplify} \quad *[[[q_i \wedge B];q_o \uparrow;[\neg q_i];q_o \downarrow]] \tag{4.38}$$
$$\rhd_{hse-to-prs} \quad \{q_i \wedge B \to q_o \uparrow, \neg q_i \to q_o \downarrow\} \tag{4.39}$$
$$\rhd_{symmetrization} \quad \{q_i \wedge B \to q_o \uparrow, \neg q_i \vee \neg B \to q_o \downarrow\} \tag{4.40}$$
$$\rhd_{prs-to-ops} \quad \{(q_i, B)and(q_o)\} \tag{4.41}$$

Next process to be expanded is P_{x-3} with temporary renames $Q_{x-3-i} \equiv q_i$, $Q_{x-3-o} \equiv q_o$, $lo(Q_{l-x-i}[0]) \equiv B_0$ and $hi(Q_{l-x-i}[0]) \equiv B_1$. Because invariant $B_0 \oplus B_1$ holds during execution of the process, it can be defined that $B_0 \equiv \neg B_1$.

$$P_{x-3} \equiv (A_{x-3}/[lo(Q_{l-x-i}[0]) \to x[0]\downarrow \,[]\, hi(Q_{l-x-i}[0]) \to x[0]\uparrow]) \tag{4.42}$$
$$\equiv (A_{x-3}/[B_0 \to x[0]\downarrow \,[]\, B_1 \to x[0]\uparrow]) \tag{4.43}$$
$$\equiv (A_{x-3}/[\neg B_1 \to x[0]\downarrow \,[]\, B_1 \to x[0]\uparrow]) \tag{4.44}$$
$$\rhd_{chp-to-hse} \quad *[[[q_i];[\neg B_1 \to x[0]\downarrow \,[]\, B_1 \to x[0]\uparrow];q_o \uparrow;[\neg q_i];q_o \downarrow]] \tag{4.45}$$
$$\rhd_{simplify} \quad *[[branch_1 \,[]\, branch_2]] \tag{4.46}$$
$$branch_1 \equiv q_i \wedge \neg B_1 \to x[0]\downarrow; q_o \uparrow; [\neg q_i]; q_o \downarrow$$
$$branch_2 \equiv q_i \wedge B_1 \to x[0]\uparrow; q_o \uparrow; [\neg q_i]; q_o \downarrow$$

After HSE-to-PRS we get PRS

$$q_i \wedge B_1 \rightarrow x[0] \uparrow, q_o \uparrow \tag{4.47}$$

$$q_i \wedge \neg B_1 \rightarrow x[0] \downarrow, q_o \uparrow \tag{4.48}$$

$$\neg q_i \rightarrow q_o \downarrow \tag{4.49}$$

which is transformed into operator set $\{(B_1, q_i)sw(x[0]), (q_i)w(q_o)\}$. Process P_{x-5} has similar structure with different signals.

Next process to be expanded is P_{x-6} with temporary renames $Q_{x-6-i} \equiv q_i$, $Q_{x-6-o} \equiv q_o$, $ae(Q_{l-x-i}) \equiv B$ and $Q_{l-x-o} \equiv x$. Since $\{q_i \wedge \neg B\}\ x \uparrow\ \{q_i \wedge \diamond B\}$, both guards $[q_i]$ and $[B]$ can be true at the same time causing interference with x. Since $\{q_i \wedge B\}\ q_o \uparrow\ \{\diamond\neg q_i \wedge B\}$, both guards $[B]$ and $[\neg q_i]$ can be true at the same time causing interference with q_o. Both cases of interference can be solved by reshuffling. Process P_y has similar structure with P_x with different signals.

$$P_{x-6} \rhd_{chp-to-hse} \quad * [[[q_i]; x \uparrow; [B]; x \downarrow; q_o \uparrow; [\neg q_i]; q_o \downarrow]] \tag{4.50}$$

$$\rhd_{reshuffle} \quad * [[[q_i]; x \uparrow; q_o \uparrow; [\neg q_i]; [B]; x \downarrow; q_o \downarrow]] \tag{4.51}$$

$$\rhd_{simplify} \quad * [[[q_i]; x \uparrow; q_o \uparrow; [\neg q_i \wedge B]; x \downarrow; q_o \downarrow]] \tag{4.52}$$

$$\rhd_{strengthening} \quad * [[[q_i]; x \uparrow; [x]; q_o \uparrow; [\neg q_i \wedge B]; x \downarrow; [\neg x]; q_o \downarrow]] \tag{4.53}$$

$$\rhd_{hse-to-prs} \quad \{q_i \rightarrow x \uparrow; x \rightarrow q_o \uparrow; \neg q_i \wedge B \rightarrow x \downarrow; \neg x \rightarrow q_o \downarrow\} \tag{4.54}$$

$$\rhd_{prs-to-ops} \quad \{(\neg q_i, B)aC(x), (x)w(q_o)\} \tag{4.55}$$

4.5 Analysis of process P_6

In this section the process P_6 is expanded. Comparator expressions $x > y$ and $y > x$ are implemented as Boolean expressions with signature $>: \mathbb{B}^2 \times \mathbb{B}^2 \mapsto \mathbb{B}^1$. Subtractor expressions $x - y$ and $y - x$ are implemented as Boolean expressions $SUB(x, y)$ and $SUB(y, x)$ with signature $-\ : \mathbb{B}^2 \times \mathbb{B}^2 \mapsto \mathbb{B}^2$. Further elaboration of comparators and subtractors is trivial and omitted from this work.

$$P_6 \equiv (A_6 / *[x > y \to A_{6-1} \talloblong y > x \to A_{6-2}]) \tag{4.56}$$

$$P_{6-1} \equiv (A_{6-1}/x := x - y) \tag{4.57}$$

$$\equiv (A_{6-1}/A_{6-1-1}; A_{6-1-2}) \tag{4.58}$$

$$P_{6-1-1} \equiv (A_{6-1-1}/[SUB(x,y)[0] \to x[0] \uparrow \talloblong \neg SUB(x,y)[0] \to x[0] \downarrow]) \tag{4.59}$$

$$P_{6-1-2} \equiv (A_{6-1-2}/[SUB(x,y)[1] \to x[1] \uparrow \talloblong \neg SUB(x,y)[1] \to x[1] \downarrow]) \tag{4.60}$$

$$P_{6-2} \equiv (A_{6-2}/y := y - x) \tag{4.61}$$

$$\equiv (A_{6-2}/A_{6-2-1}; A_{6-2-2}) \tag{4.62}$$

$$P_{6-2-1} \equiv (A_{6-2-1}/[SUB(y,x)[0] \to y[0] \uparrow \talloblong \neg SUB(y,x)[0] \to y[0] \downarrow]) \tag{4.63}$$

$$P_{6-2-2} \equiv (A_{6-2-2}/[SUB(y,x)[1] \to y[1] \uparrow \talloblong \neg SUB(y,x)[1] \to y[1] \downarrow]) \tag{4.64}$$

Processes P_{6-1} and P_{6-2} have the form $(L/A; R)$ and are implemented using the same flow as presented in section 4.2. Transformation chain for P_6 is given next with temporary renames $Q_{6-i} \equiv q_i$, $Q_{6-o} \equiv q_o$, $A_{6-1-i} \equiv a_i$, $A_{6-1-o} \equiv a_o$, $A_{6-2-i} \equiv b_i$, $A_{6-2-o} \equiv b_o$, $x > y \equiv B_0$ and $y > x \equiv B_1$.

Branches are mutually exclusive due to invariant $\neg B_0 \vee \neg B_1$. Consider a scenario where $q_i \wedge B_0 \wedge \neg a_i$ holds initially. Since initially $\neg a_i$, the guard $[\neg a_i]$ is true causing effective firing of $q_o \uparrow$ which violates the program-order. Effective firings can be removed by reshuffling which restores program-order.

$$P_6 \rhd_{chp-to-hse} \quad *[[[q_i]; *[br_1 \talloblong br_2]; q_o \uparrow; [\neg q_i]; q_o \downarrow]] \tag{4.65}$$

$$br_1 \equiv \quad B_0 \to a_o \uparrow; [a_i]; a_o \downarrow; [\neg a_i]$$

$$br_2 \equiv \quad B_1 \to b_o \uparrow; [b_i]; b_o \downarrow; [\neg b_i]$$

$$P_6 \rhd_{simplify} \quad *[*[br_1 \talloblong br_2]] \tag{4.66}$$

$$br_1 \equiv \quad q_i \wedge B_0 \to a_o \uparrow; [a_i]; a_o \downarrow; [\neg a_i]; q_o \uparrow; [\neg q_i]; q_o \downarrow$$

$$br_2 \equiv \quad q_i \wedge B_1 \to b_o \uparrow; [b_i]; b_o \downarrow; [\neg b_i]; q_o \uparrow; [\neg q_i]; q_o \downarrow$$

$$P_6 \rhd_{reshuffling} \quad *[*[br_1 \talloblong br_2]] \tag{4.67}$$

$$br_1 \equiv \quad q_i \wedge B_0 \to a_o \uparrow; [a_i]; q_o \uparrow; [\neg q_i]; a_o \downarrow; [\neg a_i]; q_o \downarrow$$

$$br_2 \equiv \quad q_i \wedge B_1 \to b_o \uparrow; [b_i]; q_o \uparrow; [\neg q_i]; b_o \downarrow; [\neg b_i]; q_o \downarrow$$

Resulting PRS after HSE-to-PRS is

$$q_i \wedge B_0 \rightarrow a_o \uparrow \tag{4.68}$$

$$q_i \wedge B_1 \rightarrow b_o \uparrow \tag{4.69}$$

$$a_i \vee b_i \rightarrow q_o \uparrow \tag{4.70}$$

$$\neg q_i \rightarrow a_o \downarrow, b_o \downarrow \tag{4.71}$$

$$\neg a_i \vee \neg b_i \rightarrow q_o \downarrow . \tag{4.72}$$

Because branches are mutually exclusive, $\neg a_i \vee \neg b_i$ is invariant. Resulting component set after PRS-to-OPS is $\{(q_i, B_0)aC(a_o), (q_i, B_1)aC(b_o), (a_i, b_i)or(q_o)\}$.

Next process to be expanded is P_{6-1-1} with temporary renames $Q_{6-1-1-i} \equiv q_i$, $Q_{6-1-1-o} \equiv q_o$ and $SUB(x, y)[0] \equiv B$. Processes P_{6-1-2}, P_{6-2-1} and P_{6-2-2} have similar structure with different signals.

$$P_{6-1-1} \rhd_{chp-to-hse} \ * [[[qi]; [B \rightarrow x[0] \uparrow [] \neg B \rightarrow x[0] \downarrow]; q_o \uparrow; [\neg q_i]; q_o \downarrow]] \tag{4.73}$$

$$\rhd_{simplify} \ * [[branch_1 [] branch_2]] \tag{4.74}$$

$$branch_1 \equiv \quad q_i \wedge B \rightarrow x[0] \uparrow; q_o \uparrow; [\neg q_i]; q_o \downarrow$$

$$branch_2 \equiv \quad q_i \wedge \neg B \rightarrow x[0] \downarrow; q_o \uparrow; [\neg q_i]; q_o \downarrow$$

$$\rhd_{hse-to-prs} \ \{q_i \wedge B \rightarrow x[0] \uparrow, q_o \uparrow; q_i \wedge \neg B \rightarrow x[0] \downarrow, q_o \uparrow; \neg q_i \rightarrow q_o \downarrow\} \tag{4.75}$$

$$\rhd_{prs-to-ops} \ \{(B, q_i)sw(x[0]), (q_i)w(q_o)\} \tag{4.76}$$

4.6 Analysis of process P_7

In this section the process P_7 is expanded. The process sets output signals to valid values based on local variable x. Output signals can be written to sequentially or in parallel. In this work the signals are written sequentially. This triggers a transaction in the right environment which responds with acknowledgement after which the output signals are cleared. Clearing the output signals can be done in one function call, but this optimization is not done in this work. Transformation chain for P_7 is given next with temporary renames

$A_{r-o}[0]_0 \equiv y_0$, $A_{r-o}[0]_1 \equiv y_1$, $A_{r-o}[1]_0 \equiv z_0$, $A_{r-o}[1]_1 \equiv z_1$ and $A_{r-i} \equiv B$.

$$\begin{aligned}
P_7 &\equiv && (A_7/set_0; set_1; [B]; y_0\downarrow; y_1\downarrow; z_0\downarrow; z_1\downarrow; [\neg B]) && (4.77)\\
set_0 &\equiv && [\neg x[0] \rightarrow y_0\uparrow, y_1\downarrow \,[]\, x[0] \rightarrow y_0\downarrow, y_1\uparrow] &&\\
set_1 &\equiv && [\neg x[1] \rightarrow z_0\uparrow, z_1\downarrow \,[]\, x[1] \rightarrow z_0\downarrow, z_1\uparrow] &&\\
&\rhd_{decompose} && (A_7/A_{7-1}; A_{7-2}) && (4.78)\\
P_{7-1} &\equiv && (A_{7-1}/set_0) && (4.79)\\
P_{7-2} &\equiv && (A_{7-2}/A_{7-3}; A_{7-4}) && (4.80)\\
P_{7-3} &\equiv && (A_{7-3}/set_1) && (4.81)\\
P_{7-4} &\equiv && (A_{7-4}/A_{7-5}; A_{7-6}) && (4.82)\\
P_{7-5} &\equiv && (A_{7-5}/[B]) && (4.83)\\
P_{7-6} &\equiv && (A_{7-6}/A_{7-7}; A_{7-8}) && (4.84)\\
P_{7-7} &\equiv && (A_{7-7}/y_0\downarrow) && (4.85)\\
P_{7-8} &\equiv && (A_{7-8}/A_{7-9}; A_{7-10}) && (4.86)\\
P_{7-9} &\equiv && (A_{7-9}/y_1\downarrow) && (4.87)\\
P_{7-10} &\equiv && (A_{7-10}/A_{7-11}; A_{7-12}) && (4.88)\\
P_{7-11} &\equiv && (A_{7-11}/z_0\downarrow) && (4.89)\\
P_{7-12} &\equiv && (A_{7-12}/A_{7-13}; A_{7-14}) && (4.90)\\
P_{7-13} &\equiv && (A_{7-13}/z_1\downarrow) && (4.91)\\
P_{7-14} &\equiv && (A_{7-14}/[\neg B]) && (4.92)
\end{aligned}$$

Processes P_7, P_{7-2}, P_{7-4}, P_{7-6}, P_{7-8}, P_{7-10} and P_{7-12} have the form $(L/A; R)$ and are implemented using the same flow as presented in section 4.2. Transformation chain for P_{7-1} is given next with temporary renames $Q_{7-1-i} \equiv q_i$ and $Q_{7-1-o} \equiv q_o$. Process P_{7-3} has similar structure with different signals.

$$\begin{aligned}
P_{7-1} \rhd_{chp-to-hse} &&& *[[[q_i]; [branch_1 \,[]\, branch_2]; q_o\uparrow; [\neg q_i]; q_o\downarrow]] && (4.93)\\
branch_1 &\equiv && \neg x[0] \rightarrow y_0\uparrow, y_1\downarrow &&\\
branch_1 &\equiv && x[0] \rightarrow y_0\downarrow, y_1\uparrow &&\\
&\rhd_{simplify} && *[[branch_1 \,[]\, branch_2]] && (4.94)\\
branch_1 &\equiv && q_i \wedge \neg x[0] \rightarrow y_0\uparrow, y_1\downarrow; q_o\uparrow; [\neg q_i]; q_o\downarrow &&\\
branch_1 &\equiv && q_i \wedge x[0] \rightarrow y_0\downarrow, y_1\uparrow; q_o\uparrow; [\neg q_i]; q_o\downarrow &&
\end{aligned}$$

Resulting PRS after HSE-to-PRS is

$$q_i \wedge \neg x[0] \rightarrow y_0 \uparrow, y_1 \downarrow, q_o \uparrow \tag{4.95}$$

$$q_i \wedge x[0] \rightarrow y_0 \downarrow, y_1 \uparrow, q_o \uparrow \tag{4.96}$$

$$\neg q_i \rightarrow q_o \downarrow . \tag{4.97}$$

Resulting component set is $\{(\neg x[0], q_i)sw(y_0), (x[0], q_i)sw(y_1), (q_i)w(q_o)\}$.

Transformation chain for P_{7-5} is given next with temporary renames $Q_{7-5-i} \equiv q_i$ and $Q_{7-5-o} \equiv q_o$. Process P_{7-14} has similar structure with different signals.

$$P_{7-5} \rhd_{chp-to-hse} \ * [[[q_i]; [B]; q_o \uparrow; [\neg q_i]; q_o \downarrow]] \tag{4.98}$$

$$\rhd_{simplify} \ * [[[q_i \wedge B]; q_o \uparrow; [\neg q_i]; q_o \downarrow]] \tag{4.99}$$

$$\rhd_{hse-to-prs} \ \{q_i \wedge B \rightarrow q_o \uparrow, \neg q_i \rightarrow q_o \downarrow\} \tag{4.100}$$

$$\rhd_{symmetrize} \ \{q_i \wedge B \rightarrow q_o \uparrow, \neg q_i \vee \neg B \rightarrow q_o \downarrow\} \tag{4.101}$$

$$\rhd_{prs-to-ops} \ \{(q_i, B)and(q_o)\} \tag{4.102}$$

Transformation chain for P_{7-7} is given next with temporary renames $Q_{7-7-i} \equiv q_i$ and $Q_{7-7-o} \equiv q_o$. Processes P_{7-9}, P_{7-11} and P_{7-13} have similar structure with different signals.

$$P_{7-7} \rhd_{chp-to-hse} \ * [[[q_i]; y_0 \downarrow; q_o \uparrow; [\neg q_i]; q_o \downarrow]] \tag{4.103}$$

$$\rhd_{hse-to-prs} \ \{q_i \rightarrow y_0 \downarrow, q_o \uparrow; \neg q_i \rightarrow q_o \downarrow\} \tag{4.104}$$

$$\rhd_{prs-to-ops} \ \{(0, \neg q_i)ff(y_0), (q_i)w(q_o)\} \tag{4.105}$$

4.7 Conclusion of analysis

The final netlist with all components and connections is presented next starting from eq. (C.1) and ending with eq. (C.40) in appendix C. Excluding environment, the netlist contains 40 processes, 42 channels including channels to environment, 1 inverter, 34 wires, 18 D-elements, 4 AND-gates, 12 switch-elements, 4 asymmetric C-elements, 1 OR-gate and 4 flip-flops.

5. CONCLUSION

Asynchronous circuits offer an alternative for synchronous circuits, expanding the design space for circuit designers. Instead of using a global clock signal to control state transitions of state-holding memory elements, asynchronous circuits utilize handshaking protocols between registers. Even though the industry has focused more effort to synchronous tools, also asynchronous alternatives exist. One of these alternatives is the Asynchronous Circuit Toolkit (ACT) developed by Rajit Manohar and his research group in Yale. The ACT or its predecessors have already been applied in designing functioning quasi-delay-insensitive circuits based on synthesis methodology developed by Alain Martin in California Institute of Technology. In this work, the synthesis methodology was presented and an example circuit which solves the greatest common divisor was synthesized by hand using the methodology. Even though ACT was only used for behavioral verification of the circuit in this work, author also tested and succeeded to convert the design to semirouted and semifinished ASIC just using the tools provided with ACT. The flow to ASIC produced template layouts for non-standard cells which require manual work to finish them.

Next some ideas for future research. One idea would be to implement the hand synthesized circuit with SPICE and simulate and verify it. This was omitted from this work due to page limit. Second idea would be to implement the verified circuit on real chip and physically verify it. Another high-level idea would be to implement two chips, one synchronous and one asynchronous, and then comparing their power consumption, performance and electromagnetic emissions. This could pave way for more research questions. Can asynchronous circuits beat synchronous circuits in terms of raw computation performance? If not, then what is the bottleneck? Can asynchronous circuits beat synchronous circuits in power consumption? If so, then this would expand the design space for energy constrained systems. Are asynchronous circuits more suitable for applications that require smaller electromagnetic emissions? If not, why? If emissions truly are smaller and more evenly distributed, is the success rate for side-channel attacks reduced? Also, since this work focused to the Martin's translation flow from CHP to CMOS, other design methods were not covered. What other design methods there are, and how do they work? What are the advantages and disadvantages of each methodology? Could asynchronous design become the next paradigm shift in electrical circuit design?

REFERENCES

[1] Ataei, Samira and Hua, Wenmian and Yang, Yihang and Manohar, Rajit and Lu, Yi-Shan and He, Jiayuan and Maleki, Sepideh and Pingali, Keshav. “An Open-Source EDA Flow for Asynchronous Logic”. In: *IEEE Design & Test* 38.2 (2021), pp. 27–37. DOI: 10.1109/MDAT.2021.3051334.

[2] Peter A. Beerel, Recep O. Ozdag, and Marcos Ferretti. *A Designer’s Guide to Asynchronous VLSI*. Cambridge University Press 2010, 2010.

[3] Arjan Bink and Richard York. “ARM996HS: The First Licensable, Clockless 32-Bit Processor Core”. In: *IEEE Micro* 27.2 (2007), pp. 58–68. DOI: 10.1109/MM.2007.28.

[4] Steven M. Burns and Alain J. Martin. *Synthesis of Self-Timed Circuits by Program Transformation*. California Institute of Technology, 1987.

[5] Pong P. Chu. *RTL Hardware Design Using VHDL*. Wiley-Interscience, 2006. 669 pp.

[6] H. van Gageldonk et al. “An asynchronous low-power 80C51 microcontroller”. In: *Proceedings Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 1998, pp. 96–107. DOI: 10.1109/ASYNC.1998.666497.

[7] Marcel Rene van der Goot. “Semantics of VLSI Synthesis”. PhD thesis. California Institute of Technology, 1995.

[8] C. A. R. Hoare. “Communicating Sequential Processes”. In: *Communications of the ACM* 21.8 (1978).

[9] Sean Keller. “Robust Near-Threshold QDI Circuit Analysis and Design”. PhD thesis. California Institute of Technology, 2014.

[10] Sean Keller, Michael Katelman, and Alain J. Martin. “A Necessary and Sufficient Timing Assumption for Speed-Independent Circuits”. In: *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*. 2009, pp. 65–76. DOI: 10.1109/ASYNC.2009.27.

[11] J. Kessels and A. Peeters. “The Tangram framework: asynchronous circuits for low power”. In: *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001 (Cat. No.01EX455)*. 2001, pp. 255–260. DOI: 10.1109/ASPDAC.2001.913315.

[12] Andrew Matthew Lines. “Pipelined Asynchronous Circuits”. MA thesis. California Institute of Technology, 1998.

[13] Rajit Manohar. “An analysis of reshuffled handshaking expansions”. In: *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001*. 2001, pp. 96–105. DOI: 10.1109/ASYNC.2001.914073.

[14] Rajit Manohar. *Asynchronous VLSI Design Lecture Notes*. 2018.

[15] Rajit Manohar. “The Impact of Asynchrony on Computer Architecture”. PhD thesis. California Institute of Technology, 1998.

[16] Rajit Manohar and Alain J. Martin. “Quasi-delay-insensitive circuits are Turing-complete”. In: (1995).

[17] Rajit Manohar and Yoram Moses. “Analyzing Isochronic Forks with Potential Causality”. In: *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*. 2015, pp. 69–76. DOI: 10.1109/ASYNC.2015.19.

[18] Rajit Manohar and Yoram Moses. “The Eventual C-Element Theorem for Delay-Insensitive Asynchronous Circuits”. In: *2017 23rd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. 2017, pp. 102–109. DOI: 10.1109/ASYNC.2017.15.

[19] Alain J. Martin. *A Delay-insensitive Fair Arbiter*. California Institute of Technology, 1985.

[20] Alain J. Martin. “Compiling communicating processes into delay-insensitive VLSI circuits”. eng. In: *Distributed computing* 1.4 (1986), pp. 226–234. ISSN: 0178-2770.

[21] Alain J. Martin. *Programming in VLSI: From Communicating Processes To Delay-Insensitive Circuits*. California Institute of Technology, 1989.

[22] Alain J. Martin. *Synthesis of Asynchronous VLSI Circuits*. California Institute of Technology, 1991.

[23] Alain J. Martin. *The Limitations to Delay-Insensitivity in Asynchronous Circuits*. California Institute of Technology, 1990.

[24] Alain J. Martin. *The Probe: An Addition To Communication Primitives*. California Institute of Technology, 1984.

[25] Alain J. Martin and Mika Nyström. “Asynchronous Techniques for System-on-Chip Design”. In: *Proceedings of the IEEE* 94.6 (2006), pp. 1089–1120. DOI: 10.1109/JPROC.2006.875789.

[26] Alain J. Martin and Piyush Prakash. “Asynchronous Nano-Electronics: Preliminary Investigation”. In: *2008 14th IEEE International Symposium on Asynchronous Circuits and Systems*. 2008, pp. 58–68. DOI: 10.1109/ASYNC.2008.22.

[27] David E. Muller. *Theory of asynchronous circuits*. University of Illinois, Graduate College, Digital Computer Laboratory, 1955.

[28] Paul J. Nahin. *The Logician and the Engineer: How George Boole and Claude Shannon Created the Information Age*. Princeton University Press, 2013.

[29] Joshua Jaffe Paul Kocher and Benjamin Jun. “Differential Power Analysis”. In: *Advances in Cryptology - Crypto 99 Proceedings* 1666 (1999).

[30] Nigel Charles Paver. “The Design and Implementation of an Asynchronous Microprocessor”. PhD thesis. The University of Manchester, 1994.

[31] Rajit Manohar. “An Open-Source Design Flow for Asynchronous Circuits”. In: (2019).

[32] Jens Sparsø and Steve Furber. *Principles of Asynchronous Circuit Design, A Systems Perspective*. Springer Science+Business Media Dordrecht, 2001.

[33] Ivan Sutherland. “The Tyranny of the Clock”. In: *Communications of the ACM* 55.10 (2012).

[34] *The ACT VLSI Design Tools*. URL: https://avlsi.csl.yale.edu/act/doku.php (visited on 11/11/2023).

[35] John F. Wakerly. *Digital Design Principles and Practices*. 4th ed. Pearson Education, Inc., 2006.

[36] Neil H. E. Weste and David Money Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. 4th ed. Addison Wesley, 2010.

[37] K. Y. Yun and D. L. Dill. “Automatic synthesis of extended burst-mode circuits. I. (Specification and hazard-free implementations)”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18.2 (1999).

APPENDIX A: GREATEST COMMON DIVISOR ACT PROGRAM

```
defproc gcd(chan?(int<3>) X; chan?(int<3>) Y; chan!(int<3>) R) {
    int<3> x;
    int<3> y;
    chp {
        *[
            // Read operands 1 and 2 from channels X and Y
            X?x; Y?y;
            log("x=", x, ", y=", y);

            // Solve GCD and store it to x
            *[
                x > y ->
                    x := x - y;
                    log("x=", x, ", y=", y)
                [] x < y ->
                    y := y - x;
                    log("x=", x, ", y=", y)
            ];

            // Write result to channel R
            log("gcd is ", x);
            R!x
        ]
    }
}
```

Listing 2. *GCD implemented with ACT's CHP*

APPENDIX B: TESTBENCH

```
import "gcd.act";

defproc test_source(chan!(int<3>) X; chan!(int<3>) Y) {
    chp {
        // Test cases
        X!1, Y!1;
        X!5, Y!2;
        X!2, Y!5;
        X!6, Y!2;
        X!2, Y!6
    }
}

defproc test_sink(chan?(int<3>) X) {
    int<3> x;
    chp {
        *[
            X?x;
            log("sink ", x)
        ]
    }
}

defproc test() {
    gcd g;
    test_source so;
    test_sink si;
    g.X = so.X;
    g.Y = so.Y;
    g.R = si.X;
}
```

Listing 3. *Testbench*

```
$ actsim test-gcd.act test
WARNING: test_sink<>: substituting chp model (requested prs, not found)
WARNING: test_source<>: substituting chp model (requested prs, not found)
WARNING: gcd<>: substituting chp model (requested prs, not found)
actsim> cycle
[                   20] <g>  x=1, y=1
[                   20] <g>  gcd is 1
[                   30] <si>  sink 1
[                   50] <g>  x=5, y=2
[                   60] <g>  x=3, y=2
[                   70] <g>  x=1, y=2
[                   80] <g>  x=1, y=1
[                   80] <g>  gcd is 1
[                   90] <si>  sink 1
[                  110] <g>  x=2, y=5
[                  120] <g>  x=2, y=3
[                  130] <g>  x=2, y=1
[                  140] <g>  x=1, y=1
[                  140] <g>  gcd is 1
[                  150] <si>  sink 1
[                  170] <g>  x=6, y=2
[                  180] <g>  x=4, y=2
[                  190] <g>  x=2, y=2
[                  190] <g>  gcd is 2
[                  200] <si>  sink 2
[                  220] <g>  x=2, y=6
[                  230] <g>  x=2, y=4
[                  240] <g>  x=2, y=2
[                  240] <g>  gcd is 2
[                  250] <si>  sink 2
actsim> quit
```

Listing 4. *Testbench output*

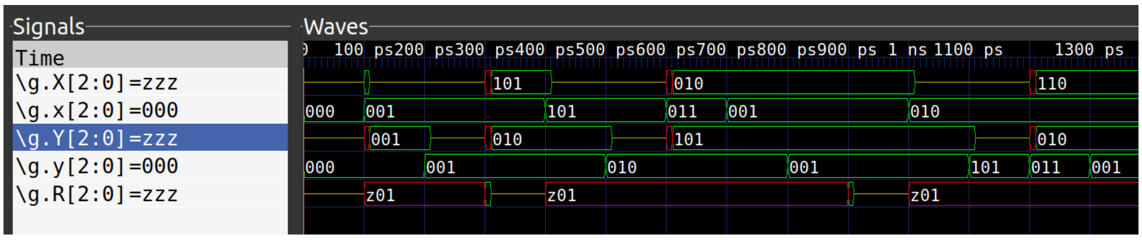

Figure B.1. *Testbench output sample as a waveform*

APPENDIX C: FINAL COMPONENT NETLIST

$$P \equiv (A_{1-i})inv(A_{1-o}) \tag{C.1}$$

$$P_1 \equiv (Q_{1-i})w(A_{2-o}),\ (A_{2-i}, Q_{1-o})D(A_{3-i}, A_{3-o}) \tag{C.2}$$

$$P_2 \equiv (Q_{2-i})w(A_{2-1-o}),\ (A_{2-1-i}, Q_{2-o})D(A_{2-2-i}, A_{2-2-o}) \tag{C.3}$$

$$P_{2-1} \equiv (Q_{2-1-i}, av(Q_{l-x-i}))and(Q_{2-1-o}) \tag{C.4}$$

$$P_{2-2} \equiv (Q_{2-2-i})w(A_{2-3-o}),\ (A_{2-3-i}, Q_{2-2-o})D(A_{2-4-i}, A_{2-4-o}) \tag{C.5}$$

$$P_{2-3} \equiv (hi(Q_{l-x-i}[0]), Q_{2-3-i})sw(x[0]),\ (Q_{2-3-i})w(Q_{2-3-o}) \tag{C.6}$$

$$P_{2-4} \equiv (Q_{2-4-i})w(A_{2-5-o}),\ (A_{2-5-i}, Q_{2-4-o})D(A_{2-6-i}, A_{2-6-o}) \tag{C.7}$$

$$P_{2-5} \equiv (hi(Q_{l-x-i}[1]), Q_{2-5-i})sw(x[1]),\ (Q_{2-5-i})w(Q_{2-5-o}) \tag{C.8}$$

$$P_{2-6} \equiv (\neg Q_{2-6-i}, ae(Q_{l-x-i}))aC(Q_{l-x-o}),\ (Q_{l-x-o})w(Q_{2-6-o}) \tag{C.9}$$

$$P_3 \equiv (Q_{3-i})w(A_{4-o}),\ (A_{4-i}, Q_{3-o})D(A_{5-i}, A_{5-o}) \tag{C.10}$$

$$P_4 \equiv (Q_{4-i})w(A_{4-1-o}),\ (A_{4-1-i}, Q_{4-o})D(A_{4-2-i}, A_{4-2-o}) \tag{C.11}$$

$$P_{4-1} \equiv (Q_{4-1-i}, av(Q_{l-y-i}))and(Q_{4-1-o}) \tag{C.12}$$

$$P_{4-2} \equiv (Q_{4-2-1})w(A_{4-3-o}),\ (A_{4-3-i}, Q_{4-2-o})D(A_{4-4-i}, A_{4-4-o}) \tag{C.13}$$

$$P_{4-3} \equiv (hi(Q_{l-y-i}[0]), Q_{4-3-i})sw(y[0]),\ (Q_{4-3-i})w(Q_{4-3-o}) \tag{C.14}$$

$$P_{4-4} \equiv (Q_{4-4-i})w(A_{4-5-o}),\ (A_{4-5-i}, Q_{4-4-o})D(A_{4-6-i}, A_{4-6-o}) \tag{C.15}$$

$$P_{4-5} \equiv (hi(Q_{l-y-i}[1]), Q_{4-5-i})sw(y[1]),\ (Q_{4-5-i})w(Q_{4-5-o}) \tag{C.16}$$

$$P_{4-6} \equiv (\neg Q_{4-6-i}, ae(Q_{l-y-i}))aC(Q_{l-y-o}),\ (Q_{l-y-o})w(Q_{4-6-o}) \tag{C.17}$$

$$P_5 \equiv (Q_{5-i})w(A_{6-o}),\ (A_{6-i}, Q_{5-o})D(A_{7-i}, A_{7-o}) \tag{C.18}$$

$$P_6 \equiv (Q_{6-i}, x > y)aC(A_{6-1-o}),\ (Q_{6-i}, y > x)aC(A_{6-2-o}),\ (A_{6-1-i}, A_{6-2-i})or(Q_{6-o}) \tag{C.19}$$

$$P_{6-1} \equiv (Q_{6-1-i})w(A_{6-1-1-o}),\ (A_{6-1-1-i}, Q_{6-1-o})D(A_{6-1-2-i}, A_{6-1-2-o}) \tag{C.20}$$

$$P_{6-1-1} \equiv (SUB(x, y)[0], Q_{6-1-1-i})sw(x[0]),\ (Q_{6-1-1-i})w(Q_{6-1-1-o}) \tag{C.21}$$

$$P_{6-1-2} \equiv (SUB(x, y)[1], Q_{6-1-2-i})sw(x[1]),\ (Q_{6-1-2-i})w(Q_{6-1-2-o}) \tag{C.22}$$

$$P_{6-2} \equiv (Q_{6-2-i})w(A_{6-2-1-o}),\ (A_{6-2-1-i}, Q_{6-2-o})D(A_{6-2-2-i}, A_{6-2-2-o}) \tag{C.23}$$

$$P_{6-2-1} \equiv (SUB(y, x)[0], Q_{6-2-1-i})sw(y[0]),\ (Q_{6-2-1-i})w(Q_{6-2-1-o}) \tag{C.24}$$

$$P_{6-2-2} \equiv (SUB(y, x)[1], Q_{6-2-2-i})sw(y[1]),\ (Q_{6-2-2-i})w(Q_{6-2-2-o}) \tag{C.25}$$

$$P_7 \equiv (Q_{7-i})w(A_{7-1-o}),\ (A_{7-1-i}, Q_{7-o})D(A_{7-2-i}, A_{7-2-o}) \tag{C.26}$$

$$P_{7-1} \equiv (\neg x[0], Q_{7-1-i})sw(A_{r-o}[0]_0),\ (x[0], Q_{7-1-i})sw(A_{r-o}[0]_1),\ (Q_{7-1-i})w(Q_{7-1-o}) \tag{C.27}$$

$$P_{7-2} \equiv (Q_{7-2-i})w(A_{7-3-o}),\ (A_{7-3-i}, Q_{7-2-o})D(A_{7-4-i}, A_{7-4-o}) \tag{C.28}$$

$$P_{7-3} \equiv (\neg x[1], Q_{7-3-i})sw(A_{r-o}[1]_0),\ (x[1], Q_{7-3-i})sw(A_{r-o}[1]_1),\ (Q_{7-3-i})w(Q_{7-3-o}) \tag{C.29}$$

$$P_{7-4} \equiv (Q_{7-4-i})w(A_{7-5-o}),\ (A_{7-5-i}, Q_{7-4-o})D(A_{7-6-i}, A_{7-6-o}) \tag{C.30}$$

$$P_{7-5} \equiv (Q_{7-5-i}, A_{r-i})and(Q_{7-5-o}) \tag{C.31}$$

$$P_{7-6} \equiv (Q_{7-6-i})w(A_{7-7-o}),\ (A_{7-7-i}, Q_{7-6-o})D(A_{7-8-i}, A_{7-8-o}) \tag{C.32}$$

$$P_{7-7} \equiv (0, \neg Q_{7-7-i})ff(A_{r-o}[0]_0),\ (Q_{7-7-i})w(Q_{7-7-o}) \tag{C.33}$$

$$P_{7-8} \equiv (Q_{7-8-i})w(A_{7-9-o}),\ (A_{7-9-i}, Q_{7-8-o})D(A_{7-10-i}, A_{7-10-o}) \tag{C.34}$$

$$P_{7-9} \equiv (0, \neg Q_{7-9-i})ff(A_{r-o}[0]_1),\ (Q_{7-9-i})w(Q_{7-9-o}) \tag{C.35}$$

$$P_{7-10} \equiv (Q_{7-10-i})w(A_{7-11-o}),\ (A_{7-11-i}, Q_{7-10-o})D(A_{7-12-i}, A_{7-12-o}) \tag{C.36}$$

$$P_{7-11} \equiv (0, \neg Q_{7-11-i})ff(A_{r-o}[1]_0),\ (Q_{7-11-i})w(Q_{7-11-o}) \tag{C.37}$$

$$P_{7-12} \equiv (Q_{7-12-i})w(A_{7-13-o}),\ (A_{7-13-i}, Q_{7-12-o})D(A_{7-14-i}, A_{7-14-o}) \tag{C.38}$$

$$P_{7-13} \equiv (0, \neg Q_{7-13-i})ff(A_{r-o}[1]_1),\ (Q_{7-13-i})w(Q_{7-13-o}) \tag{C.39}$$

$$P_{7-14} \equiv (Q_{7-14-i}, \neg A_{r-i})and(Q_{7-14-o}) \tag{C.40}$$