Mixed-Level Emulation of Asynchronous Circuits on Synchronous FPGAs

Ruslan Dashkin and Rajit Manohar
Computer Systems Lab, Yale University, New Haven, CT 06520
{ruslan.dashkin, rajit.manohar} at yale.edu

***Abstract*—Delivering an FPGA-based emulation model to the software and hardware development teams is one of the crucial steps in the chip design process. The parallelism available on the FPGA gives a performance boost necessary to speed up development and verification processes and benefits both hardware and software engineers. However, this step is challenging in the asynchronous circuits design flow due to the limitations of the commercially available FPGA platforms and Electronic Design Automation (EDA) tools.**

We present a comprehensive solution to the problem of asynchronous design emulation on synchronous FPGAs by extending prior work for emulating gate-level asynchronous designs [1]. Our framework supports asynchronous designs described at the behavioral level in the Communicating Hardware Processes language, the gate level, and hybrid designs that combine the two. We also support designs where parts of the system use natively synchronous logic. We show that our model for behavioral emulation is up to 3×10^5 faster than CPU-based simulation and up to $1.96\times$ faster than the gate-level emulation model of the same design.

We evaluate our toolchain using three real-world asynchronous design examples. We present a case study where we use our flow to emulate an asynchronous CPU on the synchronous FPGA and use this hardware to boot a real-time operating system, Zephyr RTOS. In this example, FPGA I/O interfaces use synchronous Verilog IP, and the RISC-V core model is generated from the asynchronous design.

***Index Terms*—FPGA, Asynchronous VLSI, Prototyping**

I. Introduction

A typical asynchronous design flow starts with a behavioral description in one of the languages based on Communicating Sequential Processes (CSP) [2]. CSP is a message-passing language suitable for describing highly parallel systems. Multiple extensions of CSP were developed to accommodate the needs of digital circuit designers, e.g., Tangram [3], Occam [4], Balsa [5], Haste [6] or Communicating Hardware Processes (CHP) [7]. After behavioral specification, circuits are synthesized into a gate-level representation using various techniques, such as Process Decomposition [8], Syntax-Directed Translation [9], through the State Transition Graph [10], or through the Control-Data Flow Graph [11], [12]. In the context of this paper, we target asynchronous designs that use the synthesis approach presented in [8]. In this methodology, the CHP description is first synthesized into an intermediate representation called Handshaking Expansion (HSE) and then into the gate-level netlist described in Production Rules (PRS). It is worth noting that behavioral to gate-level transition does not necessarily happen for the entire project at once, and the case when parts of the projects are at different levels of abstraction is possible and valid. We call such step-by-step transition an *incremental* design approach.

A behavioral level of abstraction does not require circuit-specific details necessary for the gate level, which makes the design process faster. A rapid transition from an idea to an implementation makes it possible to start software development and extensive functional verification early in the design process. However, these steps require a high-performance hardware prototype or an emulation system. FPGAs are the primary candidates to satisfy this requirement and are broadly adopted in the synchronous design flow. In the asynchronous domain, this option is unavailable due to the absence of asynchronous FPGAs on the market and the lack of support for asynchronous designs in the commercially available synchronous EDA flows and corresponding hardware. The only way to resolve this issue is to run asynchronous designs on the hardware originally developed for the synchronous domain. To implement this idea, a CSP-like description can be converted into a functionally equivalent model described in the languages understood by the standard synchronous EDA tools, such as Verilog HDL or VHDL, and in a way that would make this model synthesizable.

Various teams in the industry and academia explored the possibility of an automated generation of synchronous circuits from a behavioral CSP-like description. The most common approach is to use a Syntax-Directed Translation(SDT). Solutions from [13], [14] demonstrated an approach to a synthesizable synchronous circuit generation from a Tangram description. A method from [13] used a quick return to zero communication protocol and clock gating. It targeted mapping to the standard cell netlist for the synchronous ASIC flow. A work in [14] presented a pulse-based communication and targeted Xilinx FPGA implementation. Multiple papers presented methods for translating circuits described in Occam [15]–[19]. The paper [19] presented formal proof of the compilation of Communicating Processes into clocked circuits. An Occam subset mapping to OAL language and Algotronix FPGA implementation are described in [15]. Occam compilation into a proprietary netlist description and Xilinx FPGA implementation are presented in [16], [17]. In [18], authors synthesized synchronous circuits using proprietary INMOS languages. The CHP-based framework presented in [20] generated a non-synthesizable VHDL with an option for synchronous-asynchronous co-simulation. In a recent work [21], authors used a Cλash framework to describe CSP con-

structs, which were converted into VHDL and mapped onto the FPGA. Finally, [22], [23] presented a Verilog library of non-synthesizable modules called VerilogCSP for asynchronous circuit simulation in the synchronous environment.

Although all mentioned works demonstrate excellent results in compiling CSP-like circuit descriptions into synchronous circuits, they have certain limitations. For example, some of the generated circuits are not synthesizable and cannot be implemented on FPGAs. Some circuits are described in proprietary formats which are not publicly available. The major drawback in all cases is that the input design must be fully specified at the behavioral level. In other words, these solutions assume a one-step synthesis from behavioral to the gate level and prohibit an incremental transition. Such functionality can benefit the asynchronous flow due to the high flexibility in circuit design choices compared to the synchronous design methodology.

This paper presents a unified approach to asynchronous circuits mapping onto synchronous FPGAs for high-performance emulation. This solution is based on the Asynchronous Circuits Toolkit (ACT) [24], [25] and extends our previous work [1] on gate-level emulation. The new solution works with both behavioral and gate-level circuit descriptions in CHP and PRS languages, respectively. It also supports an intermediate circuit representation in HSE, as it uses a subset of CHP. However, in the rest of the paper, we will only mention CHP, implying support for HSE.

We present ACT2FPGA, an integrated toolchain that translates an asynchronous design in the ACT framework into a synthesizable synchronous Verilog model, which functionally replicates the original design. Using information in the ACT input, it automatically generates interfaces between different levels of abstraction to support an incremental design approach. The synchronous nature makes it easy to map the generated model onto commercially available FPGAs using standard EDA tools and to co-emulate it with the synchronous IP libraries, for example, provided by FPGA vendors. To analyze circuit behavior, one can use a built-in FPGA logic analyzer, such as Xilinx ILA or Intel Signal Tap.

We note that the presented translation strategy does not target a high-performance and energy-efficient synchronous circuit implementation but instead focuses on the functional equivalence and the possibility of a vendor-independent FPGA implementation with minimal manual effort. Our models have only one clock signal and do not require extensive timing constraints to go through the place and route flow without errors.

The rest of this paper is organized as follows: Section II describes CHP language constructs and how they are used to describe circuits. Section III dives into the CHP to Verilog translation details. Section IV shows how we generate logic for behavioral and gate-level communication. Section V summarizes the functionality of the presented framework. Section VI demonstrates an architecture of the generated model and a case of a complex circuit translation and implementation on the FPGA. Section VII presents Verilog synthesis results for various designs and comparisons against the gate-level emulation. Finally, Section VIII summarizes our work and describes our plans.

II. Specifying Asynchronous Circuits

This section provides a detailed description of the CHP language constructs. We explain the syntax and the expected behavior of each language statement.

A. CHP Syntax

- **Sequential composition** defines a strict order in which statements separated by ';' must be executed and completed. Each statement starts its execution only when its predecessor is complete.

$$...; S_1; S_2; ...; S_n; ...$$

- **Parallel composition** defines a group of statements separated by ',' which execute in parallel and complete in an arbitrary order.

$$...; S_1, S_2, ..., S_n; ...$$

- **Assignment** stores the result of the expression E evaluation into the variable v.

$$v := E$$

- **Send** transmits the result of an expression E evaluation over the channel C.

$$C!E$$

- **Receive** stores the value transmitted over the channel C into the variable v.

$$C?v$$

- **Probe** detects whether the channel C is ready to communicate.

$$\overline{C}$$

- **Selection** provides a conditional execution to the program. Selection statement **waits** until one of the conditions $G_1, G_2, ..., Gn$(guards) is $true$ and executes all the statements in the corresponding branch.

$$\begin{array}{l}[\, G_1 \rightarrow S_1; ... \\ []G_2 \rightarrow S_2, ... \\ []... \\ []G_n \rightarrow S_n; ... \\ ]\end{array}$$

  If guards are mutually exclusive, the selection is called *Deterministic*; if guards are independent, the selection is called *Non-Determinstc*. To resolve non-determinism among multiple $true$ guards, an extra circuit responsible for fair arbitration is added.
- **While loop** executes a corresponding sequence of statements $S_{i_{th}}$ while one of the condition $G_1, G_2, ..., G_n$ (guards) is $true$. All guards are reevaluated at the beginning of each iteration. If all guards are $false$, the loop terminates. Guards must be mutually exclusive.

TABLE I: CHP Expressions

Type	Operation
Arithmetic	Addition: $+$, Subtraction: $-$, Division: $\div$, Multiplication: $\times$ Modulo Operation: $\%$
Shift	Logic shift left: $\ll$, Logic shift right: $\gg$, Arithmetic shift right: $\ggg$
Boolean	Logic AND: &, Logic OR: $\mid$, Exclusive OR: $\oplus$, Negation: $\sim$
Compare	Less: $<$, Less or Equal: $\leq$, Greater: $>$, Greater or Equal: $\geq$, Equal: $==$, Not Equal: $!=$
Concatenation	$\{a, b, c\}$
Bit Extraction	Single bit: $a\{i\}$, Range: $a\{i..j\}$

$$
\begin{array}{l}
*[\ G_1 \rightarrow S_1; ... \\
\ []G_2 \rightarrow S_2, ... \\
\ []... \\
\ []G_n \rightarrow S_n; ... \\
\ ]
\end{array}
$$

- **Do-while loop** executes a sequence of statement S while guard G is $true$. Condition is checked at the end of each iteration, and at least one iteration is always executed. A "do-while" loop can have only one branch.

$$*[S \leftarrow G]$$

B. Expressions

Table I demonstrates all arithmetic, logic, and bit-manipulation operations supported in CHP.

C. Data Types

CHP supports two basic synthesizable types: int and $bool$. Basic types can be used to compose complex user-defined data types similar to the $struct$ type in C language. Nested user-defined types are legal as well. Along with the synthesizable types, CHP supports $pint$, $pbool$, and $preal$ parameters for constructing parameterized circuits and non-synthesizable processes.

D. Channel definition

CHP supports a $chan$ type representing channels for communication between processes. Channels are declared in the following way:

$$\textbf{\textit{chan}}(type)$$

where $chan$ is a reserved word defining a channel instance, $type$ is a type of data transmitted over the channel. Data type can be $integer$, $boolean$, or $user-defined$ structure.

At the behavioral level, communication over a channel is an abstract synchronization process, and a detailed handshaking protocol specification is not necessary. One $chan$ can be connected to another $chan$ of the same type, and correct synchronization is always guaranteed. However, while transitioning to the gate-level implementation, one would want to specify a precise communication protocol for each channel.

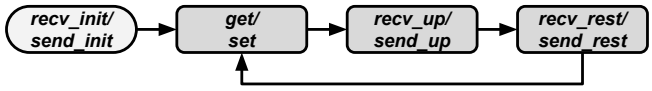

Fig. 1: A flowchart diagram of a channel communication protocol specification in ACT language.

It is necessary for the correct simulation when two processes are specified at different levels of abstraction. ACT framework supports syntax, which allows users to describe each phase of the desired handshaking protocol. The syntax is shown below:

```
defchan my_chan (bool var_in, ...;
                 bool var_out; ...){
  methods {
    recv_init{...}  send_init{...}
    get{...}        set{...}
    recv_up{...}    send_up{...}
    recv_rest{...}  send_rest{...}
  }
}
```

where the $methods$ section contains two groups of methods – one for the receiving channel and one for the sending channel. $recv\_init$ and $send\_init$ are used to initialize the channel after the global reset. get, $recv\_up$, and $recv\_rest$ define three steps of the receive operation: 1. Getting transmitted data; 2. Completing synchronization; 3. Completing the rest of the protocol. set, $send\_up$, and $send\_rest$ define three steps of the send operation: 1. Setting data to transmit; 2. Completing synchronization; 3. Completing the rest of the protocol. Figure 1 visualizes the above behavior in a flowchart diagram.

These methods are specified using a subset of CHP where only selections and assignments are allowed.

III. CHP Emulation

This section presents our automated approach to converting CHP programs into synthesizable and synchronous Verilog models. We start with a general description of our approach. Then, we demonstrate how each language structure is emulated in Verilog.

A. General Idea

Our solution is based on the Syntax-Directed Translation (SDT). SDT is a simple yet powerful compilation technique driven by the program's syntax. The idea is to construct an output using pre-defined building blocks while traversing the parse tree of the design description.

We use the ACT framework to compile input projects into C++ data structures. These structures are used to traverse and analyze the design. Each process in the design hierarchy is checked for having a behavioral or gate-level description in CHP or PRS languages, respectively. If a process is defined in PRS, it is skipped. We assume the entire sub-tree with a PRS process at the root contains only PRS processes. This assumption significantly simplifies the software but covers a

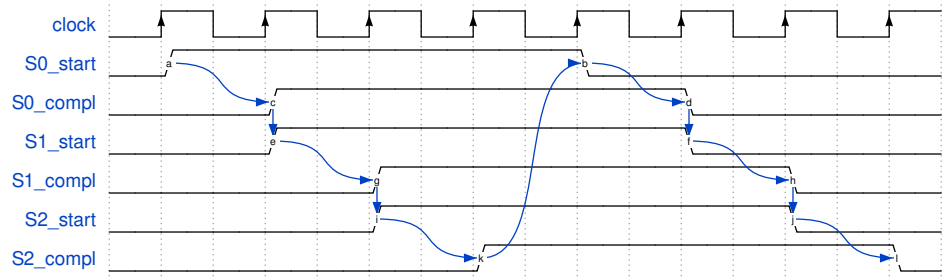

Fig. 2: A waveform demonstrating *start/complete* conditions propagation in the synchronous model emulating a sequence of events in the original asynchronous circuit.

very large subset of designs. Processes described in CHP, on the other hand, are our main target. To apply SDT, we specify a set of building blocks, each corresponding to a specific CHP structure. While traversing a program tree, we replace each original CHP statement with a corresponding building block described in Verilog. This process results in a tree data structure preserving an original architecture and nodes replaced with functional equivalents in Verilog.

Each building block includes a small state machine for the local control and a data path logic such as guard expressions or assignments. All state machines share a similar behavioral model and operate in the synchronous domain, where each state transition takes one clock cycle. First, in the initial state, they wait for the *start* condition to begin execution. The initial state depends on the type of the CHP statement (See following subsections). Once execution completes, they switch to the *DONE* state and generate a *complete* condition. In the general case, a *start* condition for each state machine is either a signal from a parent node or a *complete* condition from a predecessor. However, the root node of each process tree does not have either and requires a special circuit to generate an initial *start* condition. Finally, since we work with the hardware, we expect programs to run infinitely, which requires all state machines to return to the initial state when an execution is over. In our model, once in the *DONE* state, each state machine waits for the *return* condition to switch back to the corresponding initial state. Figure 2 shows a waveform demonstrating start/complete behavior. It is important to note that the time between the start and complete signals depends on the type of the statement, and in the example, it is shown as one cycle just for simplicity.

An execution model described above matches an execution model of CHP language and thus can be used for emulation. This model allows a user to analyze the internal state and interfaces of each process in the project.

In the following subsections, we provide more details about how each CHP structure is implemented in our model.

B. Communication Channel

As mentioned in Section II, communication in asynchronous circuits happens over the channels. At the behavioral level, a communication action is an abstract event not tied to a specific communication protocol. The only requirement is the presence of a synchronization event. This model allows us to use a simple valid/ready protocol, where *valid* is a sender's signal and *ready* is a receiver's signal. A state when both signals are $high$ represents the synchronization event. This state only lasts one clock cycle and must be reset after each synchronization. This behavior prevents unwanted repetitive transactions. Along with the valid/ready signals, such a channel can have a bus for data transmission.

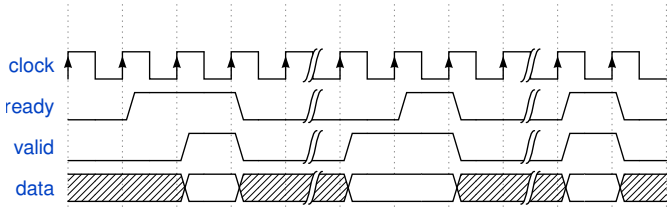

Fig. 3: A waveform demonstrating a cycle-accurate timing of the *valid/ready* protocol emulating synchronization events in the original asynchronous circuit.

A waveform diagram showing valid timings of the implemented protocol is shown in Figure 3. Both *valid* and *ready* signals are assigned independently and can be set $high$ at any time.

We note that the described valid/ready protocol differs from the one commonly used in the synchronous domain, as our version emulates synchronization events with a handshake while the synchronous version uses a clock signal for the same purpose. Although our protocol has lower performance and does not transfer data at every clock cycle, its behavior is correct and suits our model well.

C. Switching Condition Types

In this section, we define three types of conditions state machines use to transition between states and to generate *start* and *complete* signals.

1) **State condition** signals when a state machine is in a specific state. This condition is used in selection and loop statements when a parent node needs to signal to a child node that it can start execution or in a sequence to notify the successor to start execution.

$$state\_condition = state == STATE\_N$$

2) **Guard condition** signals when a guard expression in a selection or loop statement is $true/false$. A guard can be defined with a boolean expression or a probe.

$$guard\_condition = boolean\_expression\,/\,probe$$

3) **Communication completion condition** signals when both *valid* and *ready* signal are $high$. This condition is used in send and receive state machines.

$$commun\_condition = C\_valid\ \&\ C\_ready$$

The basic types can be used to compose more complex conditions. Complex conditions are arbitrary combinations of logically ANDed, ORed, and NANDed conditions. Complex conditions are necessary to improve performance and guarantee correct event scheduling.

D. Composition

In our solution, we view sequential and parallel compositions as a set of rules on how *start* and *complete* signals are propagated among the state machines.

Sequential composition requires each statement to terminate before its successor can start execution. To satisfy this requirement, each *complete* condition is wired to all the subsequent *start* conditions.

Parallel composition requires all statements in the parallel group to terminate before the successor can start execution. For that, we combine all *complete* conditions with a logic AND. To guarantee synchronization and to prevent a repetitive false execution, each state machine has the *DONE* state. This extra state stops execution until the *return* condition is received.

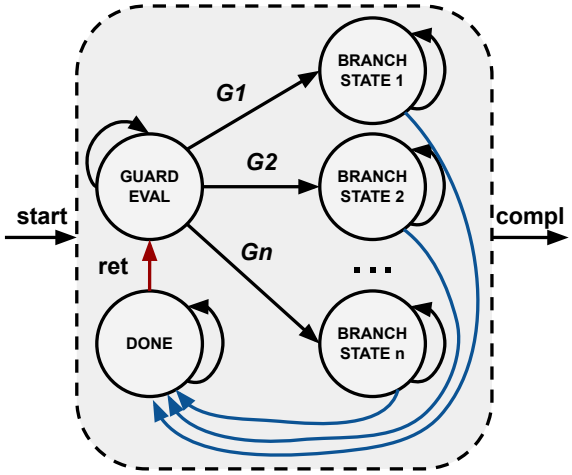

Fig. 4: A graph representation of the synchronous state machine emulating the behavior of the CHP Selection statement. It consists of a *GUARD EVAL* state for the guard evaluation, *BRANCH STATEs* to control child nodes and a *DONE* state to indicate execution completion.

E. Selection

A general structure of the selection state machine is shown in Figure 4. Initially, this machine remains in the *GUARD EVALUATION* state. Once *start* condition is received, all guards $G_1, ..., G_n$ are evaluated until one of them is $true$. When a guard is $true$, a corresponding branch is taken by switching to one of the *BRANCH STATEs*. Branch states are

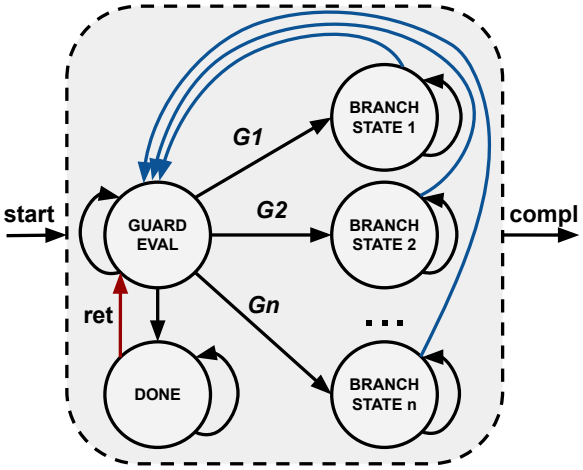

Fig. 5: A graph representation of the synchronous state machine emulating the behavior of the CHP "While" loop. It consists of a *GUARD EVAL* state for guard evaluation, *BRANCH STATEs* to control child nodes and a *DONE* state to indicate execution completion.

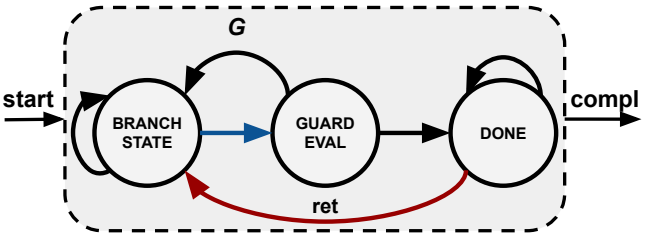

Fig. 6: A graph representation of the synchronous state machine emulating the behavior of the CHP "Do-while" loop. It consists of a *GUARD EVAL* state for the guard evaluation, a single *BRANCH STATE* for the child node control, and a *DONE* state to indicate execution completion.

used to generate a *start* conditions for the child sequences. The machine remains in the selected state until the last statement in the branch completes execution. When this statement sends back a *complete* condition, the selection machine switches to the *DONE* state.

In the non-deterministic selection, when a mutual exclusion of guards is not guaranteed, we instantiate a round-robin arbiter module to match the behavior described in Section II. It takes original guards as inputs and fairly picks only one output to be set $true$. The Arbiter module was originally synchronous and was implemented in Verilog.

F. Loop

Loops emulating state machines are shown in Figure 5 and Figure 6 for the "while" and "do-while" loops respectively.

Similarly to the selection statement, a "while" loop starts in the *GUARD EVALUATION* state. Once it receives the *start* condition, all guards $G_1, ..., G_n$ are evaluated. If one guard is $true$, the machine switches to the corresponding *BRANCH STATE* to initiate the execution of a child sequence. When the last statement in the branch generates a *complete*

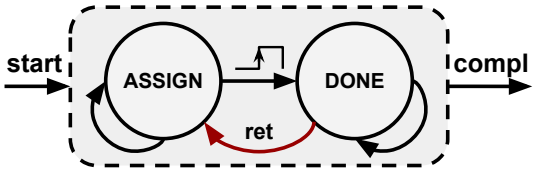

Fig. 7: A graph representation of a synchronous state machine emulating the behavior of the CHP Assignment statement. It consists of an *ASSIGN* state for the expression evaluation phase and a *DONE* state to indicate completion. A square waveform in the middle demonstrates that the assignment takes a single clock cycle.

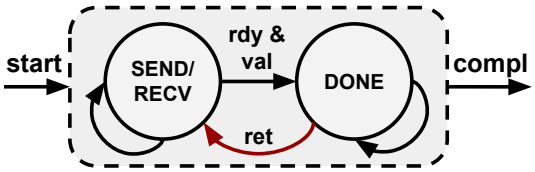

Fig. 8: A graph representation of a synchronous state machine emulating the behavior of the CHP Send and Receive statements. consisting of a *SEND/RECV* state for the synchronization step and *DONE* state to indicate completion. A $rdy\ \&\ val$ expression in the middle indicates a valid/ready protocol synchronization event and a switching condition when both signals are $high$.

condition, the "while" loop machine switches back to the *GUARD EVALUATION* state to reevaluate guards. This cycle repeats until all guards become $false$. The loop terminates by switching to the *DONE* state.

A "do-while" loop, on the other hand, guarantees an execution of at least one iteration. It is initialized in the only available *BRANCH STATE*. Once *start* condition is received, a child sequence starts execution. When the last statement in the sequence generates a *complete* condition, the "do-while" machine switches to the *GUARD EVALUATION* state. If the guard *G* is $true$, the loop returns to the *BRANCH STATE*; if the guard is $false$, the loop terminates by switching to the *DONE* state.

G. Assignment

An assignment state machine is shown in Figure 7. Initially, it starts in the *ASSIGN* state. Upon receiving the *start* condition, this state machine immediately evaluates an expression on the right-hand side of the assignment operator. It completes the assignment by storing a new value to the variable on the left-hand side and switching to the *DONE* state. Once in the *DONE* state, a new value is available at the corresponding flip-flop output. The assignment operation takes only one clock cycle for all arithmetic and logic operations.

CHP assignment is equivalent to the non-blocking assignment in Verilog.

H. Communication

A state machine emulating send and receive statements is shown in Figure 8. When the "send" state machine receives a *start* condition, it puts an expression evaluation result to the data line and sets a *valid* signal to $high$. Both values become available at the next clock cycle after the *start* condition is $high$. When the *start* condition arrives at the "receive" machine, it sets *ready* signal $high$. Once *valid* and *ready* are $high$, communication is considered over, and both machines switch to the *DONE* state, set *valid* and *ready* signals low and a new data value becomes available in the receiver's variable. A Verilog equivalent of the data transfer is a non-blocking assignment of the sender's data value to the corresponding receiver's channel variable.

It is worth noting that the Probe statement does not have a complex behavioral model and only acts as a guard condition. In our channel model, Probe checks for the *valid* signal to be $high$.

I. Data Types

Non-synthesizable data types are resolved at the ACT framework compilation stage. Synthesizable int and $bool$ are replaced with Verilog $reg/wire$ types depending on the context, e.g., instance interconnects are $wire$ type and an internal variable storing assignment value is reg type.

IV. Behavioral and Gate-level Co-emulation

This section describes how we generate glue logic for CHP and PRS communication using SDT and a user-defined channel specification.

Glue logic generation is one of the key features of our solution. As mentioned in Section II, the ACT framework provides a very flexible environment for the channel definition. Users can specify any protocol they want using a CHP language subset. Since our goal was to automate the process as much as possible and not to use any extra files but those required by the ACT framework, we use this specification to generate a Verilog module connecting a valid/ready protocol described in the previous section with a user-defined protocol.

All methods in the channel definition represent a sequence of events described in CHP and can be put together separated by ';'. Thus, rewriting all methods in the way shown in the listing below is a valid interpretation of channel behavior:

$$
\begin{aligned}
&\textbf{\textit{SEND}} \equiv \\
&\quad SEND\_INIT; \\
&\quad *[\, SET;\ SEND\_UP;\ SEND\_REST\,] \\
&\\
&\textbf{\textit{RECEIVE}} \equiv \\
&\quad RECV\_INIT; \\
&\quad *[\, GET;\ RECV\_UP;\ RECV\_REST\,]
\end{aligned}
$$

First, send and receive channels execute $SEND\_INIT$ and $RECV\_INIT$ sections. These methods are executed only once. After that, both channels fall into the infinite loops of three main parts of the synchronization procedure.

Figures 9a and 9b show how we would like glue logic to be placed in the generated Verilog model. This architecture can be realized by simple modification in the code above. A listing

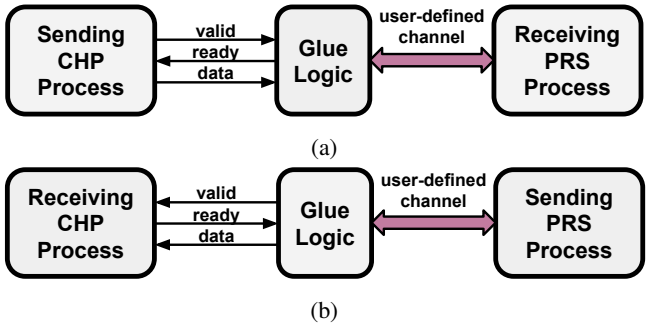

Fig. 9: A glue logic for the behavioral and gate-level co-emulation connecting communication channels at different levels of abstraction. The diagram showing signals' names and directions for (a) the CHP sending and PRS receiving case, (b) the PRS sending and CHP receiving case.

below demonstrates the final versions of both send and receive processes.

```
SEND ≡
    SEND_INIT;
    *[ [CHP_RECV];
       SET; SEND_UP; SEND_REST;
       CHP_RECV
     ]

RECEIVE ≡
    RECV_INIT;
    *[ GET;
       CHP_SEND;
       RECV_UP; RECV_REST;
     ]
```

CHP_SEND and CHP_RECV communication actions are added to complete synchronization on the CHP side of the glue process. The former executes communication when a CHP process sends a value to a PRS process, and the latter when a CHP process receives a value from a PRS process.

Send implementation starts with a Probe statement $[\overline{CHP\_SEND}]$, which is interpreted as "wait for the CHP_SEND channel to be ready to communicate." At the physical level, this is equivalent to waiting for the *valid* signal to be $high$ and the data bus to hold a valid value. When this condition is satisfied, the glue process executes communication with the PRS side and only then completes communication with CHP. This sequence represents a valid execution order, satisfying slack elasticity condition [26].

Receive implementation starts with the GET method, which guarantees that the PRS process executes the first phase of the communication protocol by providing a valid data value on the data bus. The valid signal on the PRS side can be either implicit, as in dualrail QDI circuits, or explicit, as in bundled-data circuits. Once the GET method is finished, we can transmit data to the CHP process. The rest of the PRS protocol is executed when CHP communication is over. Again, this order of statements is necessary to satisfy the slack elasticity condition.

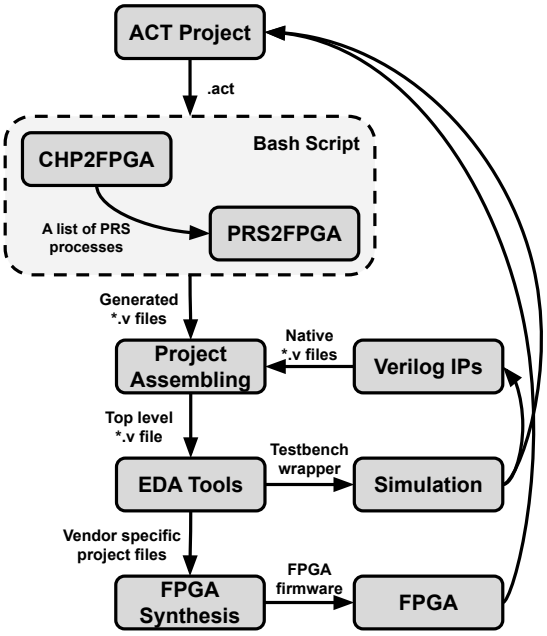

Fig. 10: A flow diagram demonstrating all stages of the emulation framework presented in this paper, along with the data formats or types of the input and output files.

The presented approach to logic generation makes it possible to reuse the code we developed for the state machine generation in the previous section.

V. Framework Summary

The presented ACT2FPGA framework is shown in Figure 10. An input to the framework is a project described in ACT language. It can be CHP, PRS, or a mixed design. In the first stage, a Bash script calls CHP2FPGA – the tool performing translation proposed in this paper. CHP2FPGA runs an analysis of the ACT project and does the following: (i) translates CHP processes to Verilog and instantiates PRS processes; (ii) generates a glue logic; (iii) outputs a list of PRS processes. Next, the Bash script reads the output list and calls PRS2FPGA [1] for each of the processes in the list, producing missing Verilog models. Both CHP/PRS2FPGA tools guarantee that the original interfaces of the processes are preserved so that generated Verilog modules are connected naturally. The output of this step is a set of files in the *.v extension, one per each process. The next step is a project assembly, where a user has the option to connect the generated model to the modules originally described in Verilog. The assembled project can be loaded into any FPGA EDA tool due to the absence of vendor-specific code. From here, the user decides whether to proceed with the simulation or to synthesize the project and run it on the FPGA.

The generated model is fully synchronous, uses only one clock signal, and is described exclusively using FPGA-friendly and vendor-independent Verilog structures. These properties make it easy to use any available commercial and open-source EDA flow for either simulation or synthesis. For simulation, users have to implement their own environment, generating

a clock signal, reset, and other I/O stimuli. For synthesis, EDA tools usually require a set of design constraints in a vendor-specific format. In general, only one clock constraint and pin assignments are necessary to implement the model on the FPGA. To improve the performance, one can implement multiple independent clock domains, constrained placement, etc. It is worth noting that our flow does not produce either a simulation environment or design constraints.

We admit that the original model with one clock may not result in the most efficient implementation. However, as mentioned earlier, our goal is to provide a functionally equivalent model that works as is. Any further performance improvements are left to the users.

VI. Case Study

A. Greatest Common Divisor

In this section, we present an example program to clarify the architecture of the generated model. We use a GCD algorithm described in CHP, which is shown in the listing below.

$$
\begin{aligned}
&\boldsymbol{GCD} \equiv \\
&\quad *[\\
&\qquad IN_1?v_1;\ IN_2?v_2\,; \\
&\qquad *[\ v_2 < v_1 \rightarrow v_1 := v_1 - v_2 \\
&\qquad\ \ []v_1 < v_2 \rightarrow v_2 := v_2 - v_1 \\
&\qquad\ \ ]; \\
&\qquad OUT!v_1 \\
&\quad]
\end{aligned}
$$

Figure 11a shows a parse tree of the CHP above. An outer loop statement without explicit guards is interpreted as an infinite loop with a single guard always set to $true$. Figure 11b demonstrates what a generated model looks like. It consists of 7 smaller state machines, one for each statement in the original CHP. A *start* condition for the infinite loop state machine is implicitly set to *high* as it is the root statement in the design. Blue dashed arrows show the directions and destinations of *start/complete* signals. Purple dashed arrow shows the *complete* condition propagating from the send statement back to the infinite loop in the branch state.

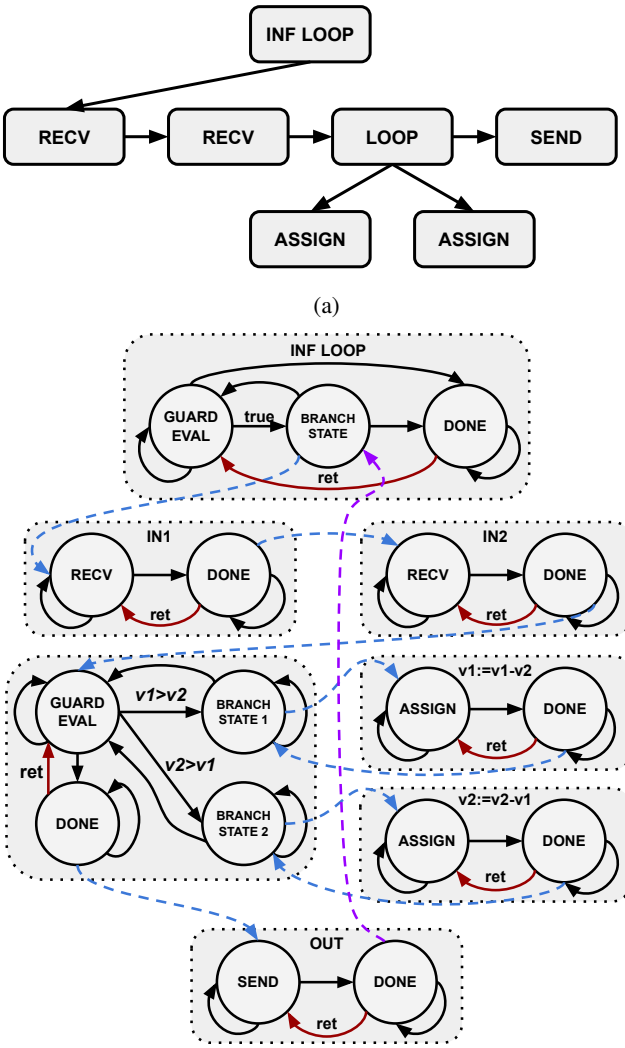

Fig. 11: Greatest Common Divisor case study. (a) A parse tree of the CHP implementation of the algorithm. (b) An architecture of a synchronous emulation model generated with the framework presented in this paper.

B. RISC-V CPU

This section presents the results of an asynchronous RISC-V CPU emulation on a synchronous FPGA. The CPU was originally designed in CHP and supported 32-bit implementations of the I, M, and C standard RISC-V extensions. The architecture included:

- PLIC - Platform Level Interrupt Controller (RISC-V Compliant)
- CLINT - Core Level Interrupt Controller (RISC-V Compliant)
- Separate Instruction and Data Caches
- Memory-mapped GPIO and UART devices

The CPU was extensively verified using an asynchronous circuit simulator and an open-source random instruction generator *riscv-dv* [27]. To port software, develop device drivers, and extend the verification, we implemented an emulation model of the CPU on the FPGA board Digilent Genesys 2. The goal of this step was to boot an operating system, Zephyr RTOS (Real-time Operating System), and run an application that would demonstrate the functionality of all major units.

A block diagram of the project is shown in Figure 12. A dashed dark grey box represents the FPGA board, where pink blocks are the physical components and the light grey box is the FPGA itself. The contents of the light grey box show the FPGA firmware. Processes originally described in CHP and converted into Verilog are blue, and processes originally described in Verilog are orange. The yellow box is the computer with a serial terminal.

To prepare the model for FPGA implementation, we leveraged the synchronous nature of the generated model and co-emulated it with the originally synchronous processes. The only requirement for such processes was to have an interface with valid/ready communication protocol described in Section III-H. Figures 13a, 13b, and 13c show three possible

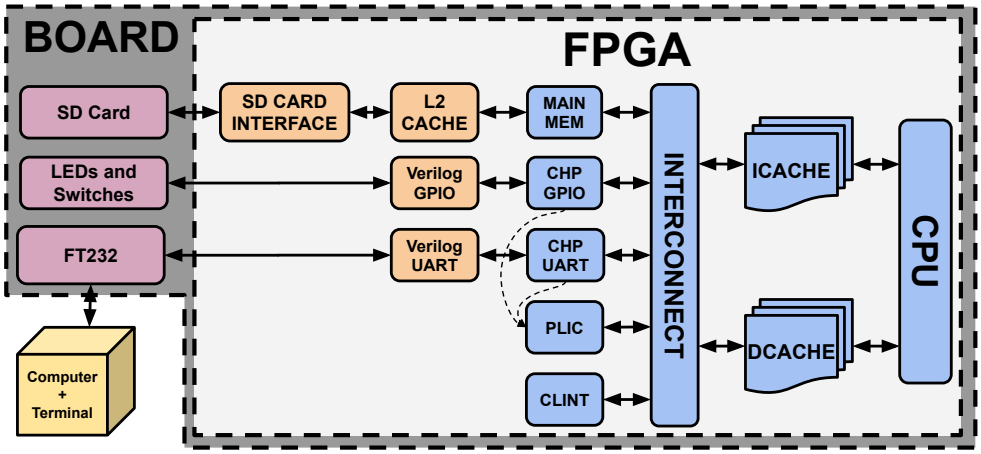

Fig. 12: A complete architecture of an asynchronous RISC-V CPU emulation platform, including translated synchronous models (blue), originally synchronous modules (orange), onboard components (pink), and PC (yellow).

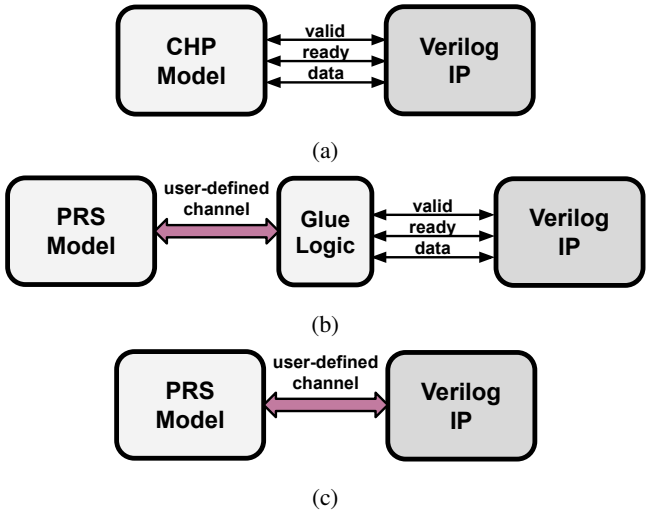

Fig. 13: An environment for the behavioral and gate-level synchronous models co-emulation with native Verilog processes connected through the abstract and user-defined communication channels. (a) CHP to Verilog abstract channel connection; (b) PRS to Veriog abstract channel connection through the glue logic; (c) PRS to Verilog user-defined channel connection.

types of connection to the generated model, i.e., via an abstract channel to CHP or PRS through the glue logic or via a user-defined channel directly to PRS. Such functionality can be useful when an FPGA vendor provides a soft IP for the on-chip components, such as transceivers, and users with asynchronous designs only need to write interface wrappers instead of designing new IPs. More details about our use cases are presented in the following paragraphs.

In Figure 12, the CPU block is the main execution pipeline. It is connected to the Instruction and Data Caches and through the interconnect to the rest of the system.

The purpose of the main memory (MAIN MEM) unit was to communicate with an external memory. In simulation, this block emulated this behavior with a C function. In the emulation, the main memory unit acted as an interface to the physical memory, which we emulated with an SD Card. We used the SD Card without a file system and uploaded OS memory images directly to the memory. A simple L2 Cache was added to slightly reduce off-chip communication traffic by storing eight 512-bit lines. The cache communication was 64-bit, and SD Card minimal block size was 512-bit, which justified the use of an extra layer in the memory subsystem. The L2 Cache was connected to the main memory unit with a valid/ready protocol, as mentioned earlier.

In the diagram, both IO interfaces UART and GPIO have two parts – one described in CHP, responsible for data processing, storage, and configuration, and another described in Verilog for data sampling and, in the case of UART, serialization/deserialization. Corresponding parts were connected with the valid/ready protocol.

The assembled project was successfully synthesized and implemented on the synchronous FPGA using Xilinx Vivado EDA. Zephyr RTOS was uploaded to the SD Card. The OS was configured for our chip and board and included necessary device drivers for PLIC, CLINT, GPIO, and UART. Figure 14 shows the terminal window on the computer and the FPGA board. The yellow box shows where an SD Card is plugged on the back side of the board. The blue box shows the UART port for communication with the computer. The CPU prints messages to the terminal via UART. The first message is sent during the OS boot process (line 1). Execution starts with the configuration of UART (line 2), Timer (line 3), and GPIOs connected to the LEDs (line 4) and Switches (line 5). After the configuration, the main application starts (line 6). For the sake of the experiment, the main loop in the application computed the Fibonacci sequence. Every second, the Timer triggered an interrupt (lines 11, 17). Manually toggling switches (Figure 14 red box) triggered GPIO input interrupts of different kinds, e.g., negative (line 14) and positive edges (line 16). Finally, sending hexadecimal symbols via serial interface (line 19) triggered UART receiver interrupts and set the corresponding output values to the LEDs (Figure 14 green box).

The emulation model was very useful in the software debug-

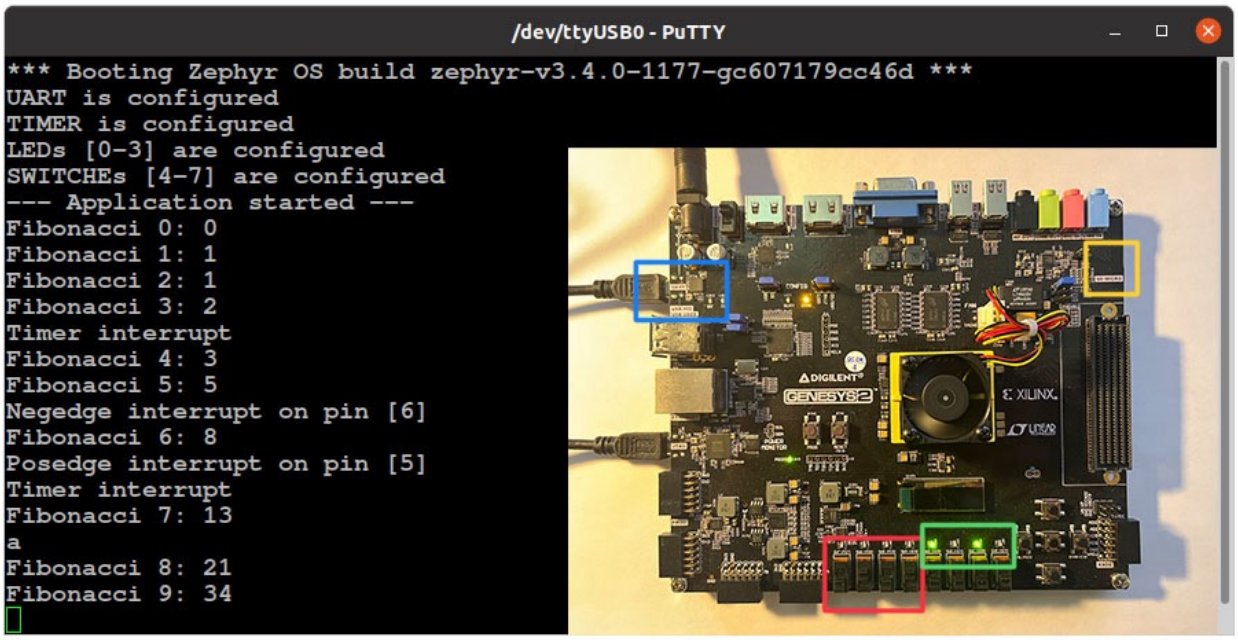

Fig. 14: A demonstration stand with a serial terminal and an FPGA board containing RISC-V CPU emulation platform (Figure 12) running Zephyr RTOS.

ging process. Although booting an OS itself cannot guarantee the correctness of the hardware design, as an addition to our existing test suite, it extended functional coverage and helped us to catch several bugs. Additionally, testing interrupts in the asynchronous simulator was problematic. By using the emulation model, we covered execution scenarios for the External and Timer interrupts and tested the exception handling mechanism implemented in the pipeline.

VII. Evaluation

In this section, we present the results of FPGA models synthesis. All results were obtained using a Xilinx design flow.

A. Glue Logic Synthesis

In this section, we present the results of the glue logic synthesis. As an example, we use a QDI channel shown below.

```
defchan a1ofN (int⟨N⟩ d; bool a)
{
    methods {
        send_init { (,i : N : d[i]−) }
        set { [([]i : N : self = i → d[i]+)] }
        send_up { [a] }
        send_rest { (,i : N : d[i]−); [∼ a] }

        recv_init { a− }
        get { [([]i : N : d[i] → self := i)] }
        recv_up { a+ }
        recv_rest { [(&i : N :∼ d[i])]; a− }
    }
}
```

This code snipped describes a full 4-phase handshake for a QDI channel where data is represented in a one-hot encoding and an acknowledge signal is active high. Table II shows the results of the synthesis. Columns 2 and 3 demonstrate resource utilization for various encoding configurations, starting from a simple single rail format up to 1 of 8 encoding. Columns 4 and 5 show the number of cycles it takes to complete one transaction on the channel, i.e., from the beginning of the communication till the end, when a new communication can be initiated. In this particular case, these numbers are constant across different encodings; however, it is not guaranteed for all data types and depends on the user's specification.

TABLE II: Synthesis and simulation results for the QDI channel *a1ofN*, where N is a data width.

Channel type	Send (LUT/FF)	Receive (LUT/FF)	Send Delay	Recv Delay
a1of1	15/13	11/10	30	15
a1of2	21/16	17/13	30	15
a1of4	32/22	29/18	30	15
a1of8	43/32	49/25	30	15

B. Behavioral and Gate level Verilog models comaprison

This section presents the results of a comparison between our previous solution [1] and the one presented in this paper. In [1] we showed how the gate-level description of an asynchronous circuit can be converted into a functionally equivalent synchronous Verilog model and mapped onto synchronous FPGA. We proposed two approaches to model generation where Option I was the model generated with the original algorithm, and Option II was a resource-optimized version of the same model. In this evaluation, we reuse the synthesis and runtime results for Option II from paper [1] (Section IV.C

Table II). As benchmarks, we use the same QDI circuits that were randomly generated and consist of standard control cells such as split, merge, source, drain, etc. This comparison aims to show the difference in resource utilization and performance rather than prove that one is better. Both solutions serve their purpose well and contribute to the framework's functionality.

Table III demonstrates synthesis results of the gate and behavioral level models and CPU and FPGA total run times. For the readers' convenience, the bottom row in the table shows the columns' indices. Column 2 shows the number of process nodes in the project hierarchy. Columns 3-4 demonstrate synthesis results for the gate and behavioral level models, respectively. The upper numbers show LUT utilization, and the lower numbers show Flip-Flop utilization. Column 5 shows the CHP over PRS ratio of LUT(upper) and Flip-Flop(lower) counts. Columns 6-11 contain performance results. In columns 9-11, the upper number corresponds to the CHP version of the circuit and lower to the PRS version. Column 6 shows the CHP over PRS synthesized model frequency ratio. Column 7 shows the average number of cycles each model takes to complete an internal cycle and return to the initial state. This metric defines the number of cycles it takes from a request on the primary input to the point where another request can be accepted. Column 8 shows the ratio of the numbers from column 7. Column 9 shows the total compilation and simulation runtime in a popular commercial synchronous circuit simulator. Column 10 shows the total runtime of a CPU-based asynchronous circuit simulator. In this paper, we use an updated version that supports both CHP and PRS simulation. Column 11 shows the FPGA runtime to bring the circuit to the same state as in the CPU-based simulation. Finally, column 12 shows PRS over the CHP FPGA model runtime ratio.

The results show that FPGA models for both behavioral and gate levels are similar in resource utilization and target frequency. Slightly bigger resource utilization can be explained by a higher overhead for an abstract model description and a good resource optimization approach for the gate level model described in [1]. Similar frequencies, with a slight CHP slowdown for larger benchmarks, are explained by greater resource utilization. It is worth noting that the PRS model outperforms the CHP model in the commercial simulator (Table III Column 9), while original circuits in the asynchronous simulator (Table III Column 10) behave in the opposite way. A better performance of CHP in the former case is explained by the complexity of the gate-level simulation, where various physical parameters of the transistors and the circuits are validated at runtime, e.g., relative transistor strength or interfering signals. Simulating CHP, on the other hand, is much easier because it does not have complex physical and timing properties. However, the emulation model of PRS only replicated the functionality of the original circuit and does not have all its physical properties making the model computationally lightweight. At the same time, the CHP version has a much higher overhead on the control logic, which triggers longer simulation runtime. The FPGA runtime of a CHP model is better than the runtime of a PRS model due to a shorter cycle time, and at the same frequency, it takes from $0.52\times$ to $0.64\times$ PRS time to compute the same result.

C. Experimental Results

In this part, we demonstrate the experimental results of our solution being used for a variety of real cases. Table IV shows synthesis results for:

- **Greatest Common Divisor** algorithm implementation with 32-bit operands;
- **Fibonacci Sequence** N-th element generator in a 32-bit numbers range;
- **RISC-V CPU** – a 32-bit implementation of IMC extensions and Physical Memory Protection.
- **Neuromorphic Compute Core** – a single unit implementing a brain-inspired computation model with spiking neural networks.
- **Network Interface Card** implementation with asynchronous elements.

These models demonstrate that the resulting target frequency depends not just on the size of the source CHP but on its complexity and design style.

We note that the RISC-V CPU, neuromorphic computing core, and network interface card were both co-emulated with native Verilog modules.

VIII. Conclusion

In this paper, we presented a complete toolchain for asynchronous circuits emulation on the synchronous FPGAs. We demonstrate an automated methodology for CHP to Verilog translation and a generation of glue logic to connect behavioral and gate-level models. We showed a case study of asynchronous and synchronous circuit co-emulation. Finally, we demonstrated how our previous work [1] is combined with the presented solution. Generated models are synthesizable and demonstrate good performance results, providing developers with an environment for hardware and software development and debugging. The resulting CHP and PRS models have similar performance, making the integration process easier with minimal performance loss.

An improvement to the current solution could be the option of using Block RAM on FPGA chips to reduce LUT utilization. Bypassing some of the conditions in the control state machines could be a good way to improve model performance. However, it must be done carefully to avoid direct combinational paths through the buffers and, as a result, combination cycles.

IX. Acknowledgment

We would like to thank users of our tools for providing their valuable feedback and synthesis results:

- Congyang Li from AVLSI Lab, Yale University
- Amirmohammad Nazari from Prof. Robert Soulé's Lab, Yale University
- Leo Liu from Brains In Silicon Lab, Stanford University

References

[1] R. Dashkin and R. Manohar, "General Approach to Asynchronous Circuits Simulation Using Synchronous FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 10, pp. 3452–3465, 2022.

TABLE III: Benchmark results for the set of circuits with a random topology implemented in CHP and PRS (Section VII-B).

Name	Total number of nodes	Number of LUT / Flip-Flop (PRS)	Number of LUT / Flip-Flop (CHP)	CHP/ PRS LUT ratio/ FF ratio	CHP / PRS Model Freq. ratio	Cycle time CHP / PRS, clock cycles	CHP / PRS Cycle time ratio	Commer. Simulator runtime CHP / PRS, sec	Async. Simulator runtime CHP / PRS, sec	FPGA runtime CHP / PRS, sec	FPGA runtime ratio, sec
gb1	100	314 / 281	431 / 404	1.37 / 1.43	1	7 / 11	0.64	9.88 / 4.61	6.39 / 19.05	0.00056 / 0.00088	0.64
gb2	200	765 / 666	974 / 914	1.27 / 1.37	1	12 / 19	0.62	14.02 / 8.06	11.11 / 31.31	0.00067 / 0.00106	0.63
gb3	400	1467 / 1272	1858 / 1742	1.27 / 1.37	1	21 / 39	0.54	13.71 / 8.29	10.51 / 28.82	0.00055 / 0.00102	0.54
gb4	800	3990 / 3399	4651 / 4312	1.17 / 1.27	1	47 / 91	0.52	16.16 / 10.57	11.58 / 31.52	0.00052 / 0.00102	0.52
gb5	1600	6042 / 5258	7584 / 7150	1.26 / 1.36	1	25 / 47	0.53	48.84 / 32.17	43.81 / 119.46	0.00054 / 0.00102	0.53
gb6	3200	13107 / 11340	16152 / 15078	1.23 / 1.33	1	27 / 51	0.53	100.66 / 68.81	96.86 / 269.52	0.00054 / 0.00102	0.53
gb7	6400	26219 / 22687	32149 / 30212	1.23 / 1.33	0.95	34 / 57	0.60	199.76 / 144.03	222.86 / 596.47	0.00067 / 0.00108	0.63
gb8	12800	52601 / 45535	63256 / 60740	1.20 / 1.33	0.74	58 / 108	0.54	250.57 / 130.96	250.01 / 616.26	0.00070 / 0.00096	0.81
Column index	2	3	4	5	6	7	8	9	10	11	12

TABLE IV: Generated FPGA models synthesis results for a variety of real projects.

Project	LUT	FF	DSP	Frequency,MHz	Board
32-bit GCD	124	71	0	500	Digilent Genesys 2
32-bit Fibonacci	83	171	0	500	Digilent Genesys 2
32-bit RISC-V CPU	13872	8990	3	20	Digilent Genesys 2
Neuromorphic Computing Core	12388	2702	0	100	Digilent Genesys 2
Network Interface Card	35397	91638	0	250	Xilinx Alveo 250

[2] C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, no. 8, p. 666–677, aug 1978.
[3] 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. Washington, DC, USA: IEEE Computer Society Press, 1991, p. 384–389.
[4] I. Inc., *Occam Programming Manual*. Prentice Hall Trade, 1984.
[5] Edwards, Doug and Bardsley, Andrew, "Balsa: An Asynchronous Hardware Synthesis Language," *The Computer Journal*, vol. 45, no. 1, pp. 12–18, 2002.
[6] Nielsen, Sune Fallgaard and Sparsø, Jens and Jensen, Jonas Braband and Nielsen, Johan Sebastian Rosenkilde, "A Behavioral Synthesis Frontend to the Haste/TiDE Design Flow," in *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, 2009, pp. 185–194.
[7] A. J. Martin, "Synthesis of asynchronous VLSI circuits," California Institute of Technology, Tech. Rep. CS-TR-93-28, 1991.
[8] A. Martin J, "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits," *Distributed Computing*, 1986.
[9] S. M. Burns and A. J. Martin, *Syntax-directed Translation of Concurrent Programs into Self-timed Circuits*, 1988, pp. 35–50.
[10] J. Sparsø, *Introduction to Asynchronous Circuit Design*. Independently Published, 2020.
[11] M. Tranchero, L. M. Reyneri, A. Bink, and M. de Wit, "An automatic approach to generate haste code from simulink specifications," in *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, 2009, pp. 175–184.
[12] Nielsen, S.F. and Sparso, J. and Madsen, J., "Towards behavioral synthesis of asynchronous circuits - an implementation template targeting syntax directed compilation," in *Euromicro Symposium on Digital System Design, 2004. DSD 2004.*, 2004, pp. 298–305.
[13] A. Peeters and K. Van Berkel, "Synchronous handshake circuits," in *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001*, 2001, pp. 86–95.
[14] J. O'Leary and G. Brown, "Synchronous emulation of asynchronous circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 2, pp. 205–209, 1997.
[15] W. Luk, D. Ferguson, and I. Page, "Structured Hardware Compilation of Parallel Programs," in *Selected Papers from the Oxford 1993 International Workshop on Field Programmable Logic and Applications on More FPGAs*. Abingdon EE&CS Books, 1994, p. 213–224.
[16] I. Page and W. Luk, "Compiling Occam into Field-Programmable Gate Arrays," *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, vol. 15, pp. 271–283, 1991.
[17] R. M. Peel and B. M. Cook, "Occam on Field-Programmable Gate Arrays-Optimising for Performance," in *Communicating Process Architectures, Proceedings of WoTUG*, vol. 23, 2000, pp. 227–238.
[18] I. Inc., "Compiling Occam into silicon," INMOS, Technical Note 72-TCH-023.
[19] J. O'Leary, G. Brown, and W. Luk, "Verified compilation of communicating processes into clocked circuits," *Formal Aspects of Computing*, vol. 9, pp. 537–559, 1997.
[20] M. Renaudin, P. Vivet, and F. Robin, "A design framework for asynchronous/synchronous circuits based on CHP to HDL translation," in *Proceedings. Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1999, pp. 135–144.

[21] Frits P. KUIPERS , Rinse WESTER , Jan KUPER and Jan F. BROENINK, "Mapping CSP Models to Hardware Using CλaSH," in *Communicating Process Architectures*, 2016.
[22] A. Saifhashemi and P. A. Beerel, "High level modeling of channel-based asynchronous circuits using Verilog," in *Communicating Process Architectures 2005*. IOS Press, 2005, pp. 275–288.
[23] P. A. B. Arash Saifhashemi, "SystemVerilogCSP: Modeling Digital Asynchronous Circuits Using SystemVerilog Interfaces," in *Communicating Process Architectures, Proceedings of WoTUG*, 2011, pp. 287–302.
[24] Rajit Manohar, "An Open-Source Design Flow for Asynchronous Circuits," Government Microcircuit Applications and Critical Technology Conference, March 2019.
[25] ACT Framework. [Online]. Available: https://github.com/asyncvlsi/act/
[26] Manohar, Rajit and Martin, Alain J., "Slack elasticity in concurrent computing," in *Mathematics of Program Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 272–285.
[27] riscv-dv. [Online]. Available: https://github.com/chipsalliance/riscv-dv