Maelstrom: A Logic Synthesis Technique for Asynchronous Circuits

Karthi Srinivasan and Rajit Manohar
Computer Systems Lab, Yale University, New Haven, CT 06520
{karthi.srinivasan, rajit.manohar} at yale.edu

***Abstract*—A new synthesis method and corresponding open-source tool, called Maelstrom, is introduced, that synthesizes CHP programs into asynchronous circuits. The method is agnostic to circuit family, and produces circuits that show significant improvements over the state-of-the-art synthesis techniques for asynchronous circuits in terms of energy, delay and area. The method also supports different datapath implementations and communication protocols. Pre-layout SPICE simulations of generated netlists of several CHP programs in a 65nm node indicate significant performance benefits over the current state of the art. Maelstrom has also been used to successfully synthesize and fabricate a chip from an abstract high-level functional description. This represents a qualitative improvement in logic synthesis of asynchronous logic from behavioral descriptions.**

***Index Terms*—Asynchronous circuits, Logic Synthesis, High-level Synthesis, Micropipelines, Bundled Data Pipelines**

I. Introduction

LOGIC synthesis is the process of converting an abstract behavioral description of a circuit into a gate-level description that implements the specified behavior. For simple descriptions, there are several "textbook" methods such as Karnaugh maps and Quine-McCluskey minimization that start with a sum-of-products expression for Boolean functions, and then systematically optimize the Boolean formula. However, as systems become larger, there is a need for the maintenance of internal state of the circuit to support more complex computations. This leads to a separation of the problem into two components: the design of the computation or datapath logic, which is typically combinational logic for arithmetic/logical operations; and the design of the state machine or control logic, which is typically sequential logic that controls the computation performed.

In synchronous design, register-transfer level (RTL) hardware description languages such as Verilog are used to specify the underlying computation in a more user-friendly format. The synthesis methodology translates the RTL into combinational logic and clocked elements using a standard clocking discipline. The problem of synthesizing the state machine (the control path) is quite straightforward in synchronous design since it is entirely controlled by a global timing reference—the clock signal—that is used to perform a state update for the controller. There are several ways to implement state machines, such as using edge-triggered flip-flops, or level-sensitive latches. Further, the communication between several state machines is realized either through implicit communication at each clock edge, or through data transfer protocols such the classic ready-valid protocol, which uses two additional signals (ready and valid) to perform a data transfer with a shared clock between the sender and receiver.

In asynchronous circuits, there is no global timing reference, and hence the control path must be self-timed in some way. There have been several proposed solutions to this over the decades. One of the first methods to synthesize asynchronous state-machines used Huffman flow-tables [1]. Huffman flow-tables can be compiled into circuits by extending Karnaugh-map style logic synthesis, and by introducing delay lines to form a feedback loop to hold state. More recent methods include burst-mode design [2], which is closely related to synchronous design by virtue of its use of a local timing signal that served the role of the clock. Both methods assume a bound on the delay of local logic, and hence require timing assumptions to ensure correct operation.

At the other extreme are delay-insensitive (DI) circuits, which assume unbounded delays on gates and wires. However, this restricts the class of implementable functions severely [3]; instead, a slightly relaxed version known as quasi-delay-insensitive (QDI) circuits are often used. In the QDI model, wire delays are assumed to be small relative to a sequence of gate delays from an adversarial firing chain of gates, but gate delays can be unbounded. Synthesis approaches for QDI circuits include Martin's synthesis method [4] of handshaking expressions (HSE) to circuits, and Petrify [5], which translates signal transition graphs to circuits. However, these methods require an input description of the system that is at the level of individual signal transitions—much lower level than the RTL-level description used by synchronous circuits. Also, the synthesis algorithms involve complete state-space exploration—a slow step. These two issues renders these methods both time-consuming and error-prone, since hand-translation and optimization of the signal transitions is required to obtain high quality circuits [6], [7].

In order to specify a complex asynchronous system at a higher level, we use a popular behavioral abstraction of asynchronous circuits known as Communicating Hardware Processes (CHP), which is explained in detail in the following section. CHP describes asynchronous circuits as a collection of processes (similar to a state machine in the synchronous context), each of which have some internal state and a behavior defined by a program. Communication between processes is performed via channels, governed by a handshaking protocol for the transfer of data.

A key requirement for general-purpose logic synthesis of

CHP to circuits is preserving the synchronization behavior of the CHP input. This is a more complex problem in the context of asynchronous circuits, since there is no single timing reference that all processes agree upon. Furthermore, changes in the synchronization behavior can in turn introduce new behaviors in the program that did not exist earlier—leading to functional errors [8]. Section II-A includes an example to illustrate this subtle issue.

One such synchronization-preserving synthesis tool is the open-source `chp2prs` [9], which is a direct (and unoptimized) implementation of the syntax-directed translation (SDT) approach to logic synthesis for asynchronous circuits [10], [11]. Balsa [12] is an optimized implementation of SDT, providing an implementation that is supposed to be as efficient as a (no longer available) commercial SDT tool called Haste/TiDE [13]. Other approaches to the translation of CHP programs into asynchronous circuits include high-level synthesis techniques [14]–[17], but these approaches modify the synchronization behavior of the input CHP. Hence, they can only be used in limited scenarios where the original program is slack elastic [8].

In this work, we present a new general-purpose approach for translating CHP programs into gate-level asynchronous logic, and an open-source implementation of our technique (as part of [9]). Our goal is to perform sequential synthesis, i.e. to obtain a circuit that implements the program *exactly as written*, thereby respecting CHP synchronization semantics. This provides an alternative to syntax-directed translation for general-purpose logic synthesis of asynchronous circuits.

Our work makes several contributions to asynchronous logic synthesis. The key novel features of our approach include:

- a set of rewrite rules that convert an arbitrary CHP program into the concurrent composition of a fixed/templated set of CHP building blocks (Section III-A,B,D–H) that is equivalent to the original program;
- a three-way handshaking element with an efficient quasi delay-insensitive circuit implementation that can implement the key CHP building block (Section III);
- an approach to synthesizing registers/storage elements that reduces sharing between different ports in comparison to standard mux-based approaches in existing synthesis engines for asynchronous logic (Section III-C);
- a method that automatically handles multiple channel actions in an asynchronous design that is compatible with the optimized register mapping strategy (Section III-I);
- an efficient translation of the building blocks into other asynchronous circuit families, including GasP, and MOUSETRAP (Section IV).

We compare our results to two existing tools for SDT, showing significant improvements in area (39%), delay (50%), and energy (58%) simultaneously, over the previous state-of-the-art tool (Section V). For small designs, i.e. ones where it is practical to hand-design and optimize circuits in order to achieve the best possible implementation such as the primitive dataflow components used by dataflow synthesis tools [14], the circuits produced by our approach are on par with those obtained manually. This means that our approach can be used as the logic synthesis back-end for dataflow synthesis tools, eliminating the need for a hand-designed dataflow library. We believe our improvements represent a qualitative advance in logic synthesis for asynchronous circuits.

II. Primitives and Background

A. CHP

CHP is a hardware description language used to describe clockless circuits derived from C.A.R. Hoare's Communicating Sequential Processes (CSP) [18]. In CHP, circuits are described at an abstract level as processes, each of which have some internal state, represented by variables, and a program that determines the sequence of actions that this process performs. Processes do not directly have access to internal variables of other processes, but rather communicate values across channels. Channels are unidirectional and ownership of the sending and receiving sides of a channel are fixed. If two processes need bidirectional communication then two channels, with their relative orientations flipped, are needed.

A CHP program, which specifies the behavior of a process can consist of the following constructs:

- Skip: *skip* is an elementary action that does nothing and continues to the next action.
- Assignment: $x := e$ is an elementary action that sets the variable x to the value e, where e is an expression. Since we often set Boolean variables to $true$ or $false$, we introduce the short-hand notation $x\uparrow$ and $x\downarrow$ to correspond to $x := true$ and $x := false$ respectively.
- Send: $C!e$ is an elementary action that sends the value of the expression e over the channel C. $C!$ is a dataless send, where no value is sent over the channel, but a communication is performed nevertheless for synchronization purposes. Channels do not buffer values; hence, a send will block until the corresponding receive is attempted, and vice versa.
- Receive: $C?x$ is an elementary action that receives a value over the channel C and stores it in the variable x. $C?$ is a dataless receive, where the received value is ignored, and only the synchronization is performed.
- Channel Access: C is a shorthand used some places in this work, for either $C!$ or $C?$ when it is not relevant to differentiate between the sending and receiving ends of the channel.
- Channel Probe: $\overline{C}$ is a Boolean used determine if the channel has a pending action to be completed. Both the sending or the receiving end of a channel can be probed. If the sender (receiver) probe is true, that means that the receiving (sending) end is blocked waiting for the sender (receiver).
- Sequential Composition: $S; T$ executes S followed by T, where S and T are any CHP programs.
- Parallel Composition: $S \parallel T$ executes the programs S and T in parallel, i.e. they may be executed in any order.
- Deterministic Selection: $[G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n]$ where G_i, known as guards, are Boolean-valued expressions and S_i are CHP programs, known as branches. The selection waits until one of the guards, G_i, evaluates to $true$, then executes the corresponding program, S_i. This construct is similar to a switch-case statement in several

programming languages. The guards must be pairwise mutually exclusive, i.e. at most one of them can be true. In any selection with two or more branches, the last guard can also be an $else$, which is simply a shorthand for writing $\neg(\vee_i G_i)$, the negation of the disjunction of all the previous guards. The notation $[G]$ is shorthand for $[G \rightarrow skip]$, which corresponds to waiting for G to become true, then proceeding.

- Non-Deterministic Selection: $[|\, G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n \,|]$ is identical to the deterministic selection except that the guards do not have to be mutually exclusive. If two or more evaluate to $true$ simultaneously, then one is picked arbitrarily (not necessarily at random). In a circuit, this choice is implemented by a collection of arbiters. When two or more guards evaluate to $true$ simultaneously, it can cause a metastable state in the arbiter. This metastable state then resolves non-deterministically, giving the grant to one of the branches of the selection statement. Therefore, the digital model of this selection statement is also non-deterministic in such a condition.
- Loop/Repetition: $*[G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n]$ is similar to the deterministic selection statement. However, once a branch has completed execution, the guards are re-evaluated. If one is true (guards must be pairwise mutually exclusive), then the corresponding branch is executed, and the process repeats until none are true. When this occurs, the loop terminates. Note that a loop may terminate without executing any branch even once, if all guards are false the first time it is reached. This construct is similar to a while loop, with the difference of having several possible branches instead of one. $*[S]$ is shorthand for $*[true \rightarrow S]$, which corresponds to an infinite loop – one that perpetually executes S.
- Do-Loop: $*[S \longleftarrow G]$ is similar to a loop, but first executes S then evaluates the guard G. If it evaluates to $true$, then the process repeats, until G evaluates to $false$. Do-loops may only have a single branch. This construct is similar to a do-while loop in programming languages.

The synchronization behavior of a CHP program is a result of the organization of communication actions in the program. Changing the synchronization behavior can modify the underlying computation in a fundamental way. For example, consider the following program that corresponds to a counter.

$$\begin{aligned} x := 0; *[[|\, &\overline{INC} \longrightarrow x := x + 1; INC? \\ [] \;&\overline{INC2} \longrightarrow x := x + 2; INC2? \\ [] \;&\overline{READ} \longrightarrow READ!x \\ [] \;&\overline{ZERO} \longrightarrow x := 0; ZERO? \;\; |]] \end{aligned}$$

Now if we have an environment that executes the following sequence of actions: $ZERO!; INC!; INC2!; READ?v$, then we know that the value of v will be set to 3 in all possible executions. On the other hand, if we change the synchronization behavior of the environment to $ZERO!; INC!; (INC2! \parallel READ?v)$, then the value of v is *non-deterministic*—it could be either 1 or 3. Mapping a CHP program into dataflow components often changes its synchronization behavior. In the example above, since $INC2$ and $READ$ are different channels without any *local* data-dependency, dataflow decomposition will by default permit their parallel execution [15], [19]—resulting in erroneous executions.

It is known that if a CHP program is *slack elastic*, then changing its synchronization behavior does not affect the underlying computation [8]. In general (as seen here), a CHP program need not be slack elastic. Hence, dataflow synthesis tools like Fluid [14] only accept sequential programs that are probe-free as input programs, which have been shown to be slack elastic [8]. The approach to logic synthesis we present here is general, and does not impose any such restriction.

B. Handshakes and C-Elements

In order to enable ease of description of circuit behavior at a lower level, we use a language closely related to CHP, known as handshaking expansions (HSE). HSE is identical to CHP, with the additional constraint that all variables are Boolean-valued, which makes it a natural choice for circuit-level descriptions.

In order to provide some context for the synthesis of CHP to circuits, we first describe some primitives. Firstly, the abstract channels between processes need to be translated to operations on wires/signals. Each channel in our synthesis is implemented with two wires, the request and acknowledge. The process at each end of the channel senses one wire and drives the other, according to the four-phase handshake protocol. The process that drives the request of a particular channel is known as the active side and the side that drives the acknowledge is known as the passive side. The protocol is as follows:

1) Initially both wires are low.
2) To initiate a communication, the active side sets the request wire high.
3) When the passive side wishes to acknowledge, it sets the acknowledge wire high.
4) To reset, after receiving the acknowledge, the active side pulls the request wire low.
5) In response to request being lowered, the passive side pulls the acknowledge low.
6) The channel is now back in its original state and another handshake can be performed.

In our notation, this is written succinctly as:

$$\begin{aligned} &Active: \;\; r\uparrow; \;\; [a]; \;\; r\downarrow; \;\; [\neg a] \\ &Passive: [r]; \;\; a\uparrow; \;\; [\neg r]; \;\; a\downarrow \end{aligned}$$

where $[x]$ denotes waiting for x to be true. Note that the four-phase handshake is not the minimal protocol for communication; there also exist two-phase handshakes where the first two actions by themselves (on each side) constitute an entire handshake synchronization. Instead of using the lowering to just reset the channel variables, they can be used to perform another handshake synchronization.

Next, we turn to the C-element half-buffer circuit [20]. C-elements behave as follows:

1) If all inputs are low, then the output is low.
2) If all inputs are high, then the output is high.
3) In all other cases, they hold state (the output for the previous inputs).

Fig. 1 (a) shows the implementation of a 2-input inverting C-element. An additional inverter would be required on the

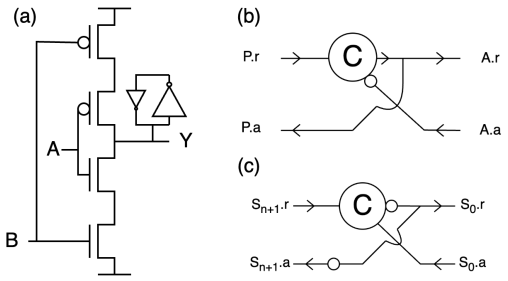


Fig. 1: (a) Transistor-level implementation of a 2-input inverting C-element. (b) C-element buffer circuit. P and A are two channels. The C-element is on the passive side of P (drives the acknowledge) and on the active side of A (drives the request). (c) The initial token buffer, used as an implementation of $*[S_0; S_{n+1}]$, where S_0 is active and S_{n+1} is passive.

output of this gate to create a non-inverting C-element. Fig. 1 (b) shows the half-buffer circuit. The bubble on one input is an inversion, and can be thought of as having an inverter on that input. In effect, it waits for $P.r \wedge \neg A.a$ to be true (false) before raising (lowering) its output. The exact sequence is:

$$[P.r \wedge \neg A.a]; P.a\uparrow, A.r\uparrow; [\neg P.r \wedge A.a]; P.a\downarrow, A.r\downarrow$$

Upon inspection, it is evident that the C-element really *sequences* two handshakes, a passive one on P and an active one on A. Effectively, it is an implementation of the process whose CHP description is: $*[P; A]$, provided the process has the passive side of channel P, and active side of channel A. $P; A$ denotes a channel access on P followed by a channel access on A. The loop $*[..]$ denotes an infinite repetition of the enclosed set of actions. Note that this is a popular circuit, and forms the basis of the control circuits in Sutherland's micropipelines [20] and the first asynchronous microprocessor [21].

III. Sequential Synthesis of CHP

In the previous section, implementation of CHP of the form $*[P; A]$ was described. The obvious next step would be to incrementally extend this to CHP of the form $*[P; C; A]$ where C is another channel, and the process under discussion may have access to either the active or the passive side. In order to perform a three-way handshake sequencing, it might be tempting to use two C-element buffers and connect two of their channels in a way so as to achieve a sequencing on the intervening channel. However, a more efficient way to perform this sequence of actions can be achieved with a single three-input C-element as shown in Fig. 2.

The PAA element (Fig. 2) can be directly written as follows:

$$\begin{aligned} &*[\ [P.r \wedge \neg A.a \wedge \neg C.a]; P.a\uparrow, C.r\uparrow; \\ &\qquad [\neg P.r \wedge A.a \wedge C.a];\ \ P.a\downarrow, C.r\downarrow\ ] \\ \|\ &*[\ [C.a]; A.r\uparrow; [\neg C.a]; A.r\downarrow\ ] \end{aligned}$$

where the second process is simply a wire connecting $C.a$ and $A.r$. For the PPA (Fig. 2) element, the corresponding process is:

$$\begin{aligned} &*[\ [P.r \wedge \neg A.a \wedge C.r]; P.a\uparrow, C.a\uparrow, A.r\uparrow; \\ &\qquad [\neg P.r \wedge A.a \wedge \neg C.r]; P.a\downarrow, C.a\downarrow, A.r\downarrow\ ] \end{aligned}$$

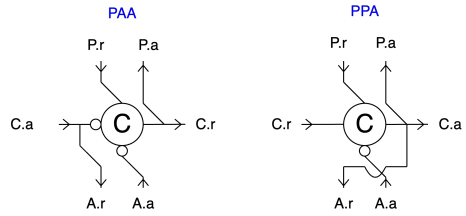


Fig. 2: The two kinds of sequencer elements, depending on whether the middle channel is active or passive. If active, then the sequencer is the PAA element, and vice versa.

These sequencer elements, which sequence three handshakes, form the fundamental building block for program synthesis. As the name suggests, they provide the ability to implement a sequence of actions, which at an abstract level, is precisely the CHP description of a process. These elements, as well as the remaining control circuits described later for each CHP construct constitute a control circuit library, which is used as a basis set for implementing programs. All circuits in this library, which will be described hereafter, are novel contributions of this work.

A. Linear Programs

In order to build up to the complete synthesis methodology for CHP, we start with the simplest kind: linear programs. Linear programs are those where the CHP is of the form $*[C_1; C_2; ..; C_n]$, where each of the terms C_i are elementary actions, i.e. assignments to local variables, sends or receives. In order to synthesize this, note that each semicolon in the program corresponds to a sequencing. Hence, this CHP can be rewritten as:

$$\begin{aligned} &*[S_1; C_1; S_2] \ \ \| *[S_2; C_2; S_3] \ \ \| \ ... \\ &\| \ \ *[S_n; C_n; S_{n+1}] \ \| \ \ *[S_1; S_{n+1}] \end{aligned}$$

where each of the S_i are fresh channels. This rewriting is useful since we already have circuits for the first n programs and synthesis would involve merely connecting them appropriately, under the assumption that the first channel action in any of the new programs is passive, and the third is active. However, under this assumption, the last program, i.e. $*[S_1; S_{n+1}]$ is not of the form implementable by the sequencer elements, since it is an active action followed by a passive action (enforced due to the other sides being passive and active respectively). In order to implement this particular program, we need a special initializer circuit known as an initial token buffer (ITB), shown in Fig. 1 (c). As before, the bubbles represent inversions, and since there is an inversion on the output of the C-element, the initial state for the output is high. From this, it is easy to see that the sequence of transitions for this circuit is:

$$\begin{aligned} &*[\ S_1.r\uparrow, S_{n+1}.a\downarrow; [S_1.a \wedge S_{n+1}.r]; \\ &\qquad S_1.r\downarrow, S_{n+1}.a\uparrow; [\neg S_1.a \wedge \neg S_{n+1}.r]\ ] \end{aligned}$$

which corresponds to performing an active handshake on S_1 and a passive handshake on S_{n+1}, in parallel. In other words, this circuit implements $*[S_1, S_{n+1}]$, which is slightly different from what was required, which was $*[S_1; S_{n+1}]$.

However, this is not an issue since S_1 happening before S_{n+1} is enforced by the rest of the circuit.

In order to understand the circuit topology, notice that the rewritten CHP forms a ring, with each program synchronized with two other programs, which are one step ahead and one step back. Finally, the ITB 'ties up' the first and last ones, closing the ring. Hence, any linear CHP can be implemented using a ring of C-elements, as we have just demonstrated. The channels, implemented by the request and acknowledge wires, form the interface/connections within the ring.

Maelstrom performs precisely this in order to synthesize a linear program. For each action in the program, a sequencer element is instantiated, depending on the type of the action (active/passive). The channels on either side of these elements are then connected in the correct order, forming a chain. Finally, an ITB is instantiated, connecting the passive (incoming) channel of the first sequencer element and the active (outgoing) channel of the last sequencer element, to close the chain up into a ring.

Finally, we observe that the behavior of the ring of sequencer elements is such that the first half-phase (assertion of request and corresponding acknowledge) of handshakes on all the channels S_i are performed first, followed by the second half-phase (deassertion of request and corresponding acknowledge) on all the channels. Essentially, there is a wave of 0-to-1 transitions propagating through the circuit, followed by a wave of 1-to-0 transitions.

B. Parallel Actions

In prior sections, we have seen how to synthesize elementary actions and linear sequences of elementary actions. However, there are several more constructs in CHP which need to be handled in order to obtain a complete synthesis method. The first of these is the parallel construct. The parallel construct allows several elementary actions to happen in parallel, i.e. in any order, as opposed to the sequence, which defines a single order on them. Consider the CHP fragment: $*[..; B, C; ..]$. Here, B and C may complete in any order. Our current synthesis of linear programs is insufficient to handle this, since we can only place a single sequencer element per action and the connectivity defines the order in which they perform the actions. Hence, we need another element, a parallelizer.

The two-way parallelizer is shown in Fig. 3(a). The $prev$ and $next$ channels connect to the rest of the chain of sequencers. When $prev.r$ is raised, the parallel split activates both sequencer elements. The acknowledge on $prev$ is sent back only when both sequencer elements complete (by the C-element in the parallel split). Further, the portion of the circuit following this parallel is only activated when both actions B and C complete (by the C-element in the parallel merge. Finally, the acknowledge is split out into both sequencer elements. The circuits for the N-way parallelizer would involve having N-input C-elements in the parallel split and merge, instead of 2-input C-elements.

C. Datapath Synthesis

In the previous sections, we discussed the synthesis of simple CHP programs, yet this is only the half of the picture.

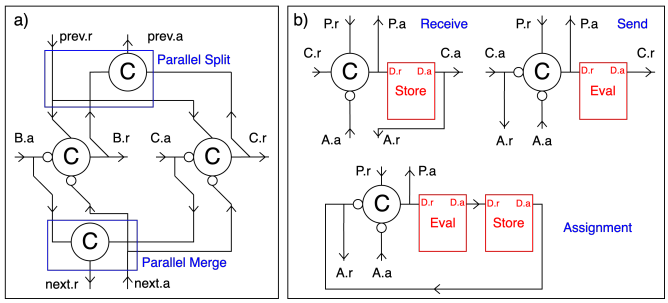


Fig. 3: a) The two-way parallelizer element, composed of two parts: the parallel split and the parallel merge. b) Implementations of the elementary actions of CHP using the sequencer elements. Store elements are data storage elements that capture values. Eval elements are combinational circuits that compute some function of variables. Note that an assignment is equivalent to the combination of a send and a receive, from the point of view of the datapath. Hence a send-receive pair is typically known as a "distributed assignment."

The C-elements that implement each elementary action also need to communicate with the datapath to actually perform the action. In the case of receives, the value that is received needs to be captured in a state-holding element so that it is accessible to the process later. In the case of sends, the expression to be sent needs to be computed and routed to the data wires of the channel that expose the value, in order for the environment to read them. In the case of assignments, both are required, the computation of a value and the placement in a data storage element. Fig. 3(b) shows the exact interaction between the control and data components. The channel between the control and datapath, D, acts as a command channel, which causes the datapath component to perform its action.

The actual implementation of these expression computation (Eval) and data storage (Store) elements depend on the choice of datapath circuit family. The conventional case where bits are encoded on a single-rail, is known as the bundled-data datapath family. Alternatively, instead of representing bits as single wires, encoding them in one of several possible 1-of-N codes results in the quasi-delay insensitive (QDI) datapath. Various types of QDI datapaths have been designed previously [22].

For the bundled data circuit family, we use level-sensitive latches as the storage (Store) elements, and use pulse-generators in order to briefly make the latches transparent and capture values. The acknowledge emanating from the Store element is simply a delayed version of the request, with a delay that corresponds to the capture delay of the latch. For the computation (Eval) blocks, we need combinational logic that implements functions of Boolean variables. We use the ABC logic synthesis and verification system to generate these circuits. For the Eval blocks, similar to the Store blocks, the acknowledge returning to the control is a delayed version of the request, with the delay matched to the delay of the combinational logic that implements the necessary computation.

1) Static Single Assignment Form: The choice of datapath family is crucial to the design process, but it is not the only one. Even within a single circuit family, there are several ways of implementing the same datapath when variables are

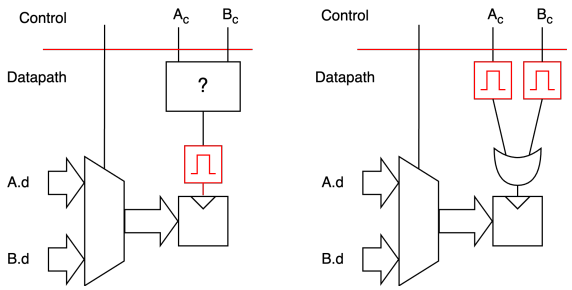


Fig. 4: The problem of sharing pulse generators across control ports for the same latch. The most efficient way to implement this is to have one pulse generator per control port.

assigned and used several times. For example, consider the following CHP: $*[A?x; B!(x+1); C?x; D!(x+2)]$. Here, the variables x is assigned twice. Hence, it is necessary to allow for the value of x to be written/updated from two places. When translating this to hardware, x can be implemented with either a single Store element with two write-ports, or as two Store elements. For the bundled datapath family, this would correspond to a single W-bit latch with a W-wide 2-to-1 mux, or two individual W-bit latches (where W is the bitwidth of the variable x). In general, for a variable that is assigned N times in a program, it can be implemented as a single latch with an N-to-1 mux, or as N separate latches. To determine which strategy is better, it is necessary to count the number of transistors that would be required to implement each.

First, notice that the problem of sharing pulse generators in the single latch implementation is non-trivial. In general, each variable (implemented by a latch) can be written from several different parts of the program, and the pulse generator must trigger when each of these control signals go high, irrespective of the state of the others. Implementing this with a single pulse generator would require a complex state machine that produces a rising edge when each of the control signals are asserted. Due to this, it is evident that the cheapest way to implement this is in fact with several pulse generators - one per control port - with the final latch clock signal being the logical OR of the outputs of these circuits. Hence, we compare our datapath implementation style with this style of using pulsed latches.

Suppose a variable is assigned N times in a program, and a single pulse generator can be implemented with k transistors, where k is a constant that is determined by the required pulse-width for the latch to successfully capture data. We will assume that the variable is a 1-bit variable for simplicity; the results can be easily scaled to get the desired numbers for an arbitrary bitwidth. In the first implementation, where we use a single latch with muxes, we require 12 transistors for the latch, $10(N-1)$ transistors for the N-way mux, kN for the N pulse generators, $3N$ for the N-way OR-gate, which results in the required number of transistors:

$$N_{FET} = \begin{cases} (13+k)N+2, & \text{if } N > 1 \\ (12+k), & \text{if } N = 1 \end{cases}$$

Note that the $3N$ for an N-way OR-gate is actually a lower bound, and the actual count is larger. In the second implementation, we require $12N$ transistors for the latches, and kN transistors for the pulse generators, resulting in $N_{FET} = (12+k)N$, which is uniformly better than the first implementation. Hence, we choose this for our datapath implementation. This technique is effectively equivalent to renaming variables every time they are assigned, and replacing accesses of variables between two assignments with accesses of the correct fresh variable. Thus each fresh variable is assigned exactly once, and only accessed until another of the fresh set is assigned. Effectively, we rewrite the CHP as:

$$*[A?x_0; B!(x_0+1); C?x_1; D!(x_1+2)]$$

This is very similar to a standard representation in software compilers, known as static single assignment form, which makes program analysis easier [23]. Here, it has the added benefit of reducing the transistor count.

D. Initial Conditions and Loop-carried Dependencies

So far, we have assumed that the CHP description of a process is of the form $*[P]$, where P is some sequence of actions. However, not all program can be described by this template. In some cases, there is a need for a defined initial state, followed by a non-terminating program. For example, consider the accumulator:

$$a := 0; *[X?x; a := a + x; A!a]$$

Here, the initial assignment, $a := 0$ is crucial for the correct operation of the process. Note that a is also a loop-carried dependency, i.e. the value of a on one iteration of the loop is needed for the next iteration. In general, we can have a list of initial conditions, resulting in CHP of the form:

$$x_1 := v_1; ...; x_n := v_n; *[\ P\ ]$$

where $\{x_k : k = 1..n\}$ is some subset of all the variables used in the program. Hence, we need additional circuitry in order to implement this. In the context of our current implementation scheme, this is quite straightforward. For each variable that is initialized, we simply instantiate a latch that is initialized to the required value on reset, and the downstream program refers to the value on this latch until the next assignment to the variable. Further, an additional assignment block that stores the 'last' value of the variable in the aforementioned initial-value latch is created, effectively implementing the loop-carry action that is needed for the next iteration of the loop to proceed correctly. This assignment block occurs last in the sequence of actions in P, after all actions in the original program have been completed, guaranteeing that the correct value to carry around the loop is available.

E. Deterministic Selections

In this section, we deal with the first construct in CHP that encodes conditional executions. Consider the following CHP fragment:

$$*[..;\ [G_1 \longrightarrow S_1\ []\ G_2 \longrightarrow S_2\ ..\ []\ G_n \longrightarrow S_n\ ];\ ..]$$

For the deterministic selection, each of the guards are pairwise mutually exclusive, i.e. at most one of the guards may be true. For the program to be free from deadlock, however, it

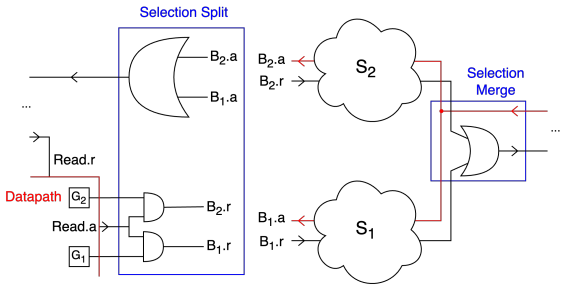


Fig. 5: Implementation of a two-way selection construct, composed of two parts: the selection split and the selection merge. The interface with the datapath is as shown.

may not be the case that none of the guards are true when the selection guards are evaluated; such CHP can be written but is most likely a user error due to the fact that CHP selections are blocking, i.e. if none of the guards are true, then the program execution does not proceed past it, but instead waits for some guard to become true. Once a guard is determined to be true, the corresponding branch, which can contain any CHP, is executed. After the completion of the branch, the execution proceeds to the portion of the program after the selection.

From the description of selections, it is evident that this is the first construct that requires a causal interaction between the datapath and the control circuits. The sequence of actions/executions in prior programs could be statically determined, but that is not the case here. The choice of branch depends on the values of internal variables and may change from one execution of the top-level loop ($*[..]$) to the next. In order to implement this, we notice that we simply need to split the request out forwards and merge the acknowledge backwards, similar to the parallel case, but to the single correct branch, instead of all branches.

The implementation of a two-way selection is shown in Fig. 5. The incoming request is routed to the datapath in order to read out (Eval) the values of the guards. For simplicity of exposition, they are shown to be encoded on a single rail, i.e. the bundled data encoding, but other encodings are also possible, as discussed earlier. This guard evaluation can be thought of as a multiple-output Eval block, with the acknowledge delay being the largest of the delays of the evaluators for each of the guards. The read acknowledge, which implies that the guard values have settled, is used by the selection split to activate the appropriate branch. The selection split also propagates the acknowledge from the activated branch backwards. As before, the interface with the sub-programs are in terms of request-acknowledge channels. Once the activated sub-program completes, the control passes downstream to the rest of the program, via the selection merge that combines the requests and splits the acknowledges correctly. As in the case of the parallel construct, this too can be easily extended from a two-way to an N-way construct.

1) Φ-functions and Merging Logic: Our datapath synthesis was designed with simplicity, and therefore transistor count, in mind. However, selections and conditional executions introduce an inconvenience. To illustrate the problem, we consider the following CHP:

$$*[C?c; [c = 0 \longrightarrow x := 4 [] c = 1 \longrightarrow x := 8]; X!x^2]$$

If each x was renamed every time it was assigned, then it is not possible to determine before runtime the correct latch whose output must be used to calculate the value that is sent out over the channel X. In order to handle this, we introduce logic at the end of every selection, that essentially acts as a value merge, selecting the correct latch based on which branch was taken in that selection. We only perform this for variables that are used after the selection, in the downstream program.

In the static single assignment form mentioned earlier, this is known as a Φ-function, which picks the correct value based on which branch was taken in a program. This analysis of detecting uses of variables is part of a larger live variable analysis pass that is performed as a pre-processing step. The instantiation of merging logic is performed automatically as part of Maelstrom's synthesis flow.

F. Non-Deterministic Selections

The other kind of construct that introduces choice is the non-deterministic selection, which is quite similar to the deterministic selection, except that several guards can be true simultaneously. In this case, the process is allowed to arbitrarily pick a branch with a true guard and proceed. Non-determinism is typically introduced when probes of channels are used as guards, since actions might be initiated on any channel to the process, by the environment of the process. For example, consider the non-deterministic merge:

$$*[[\overline{X}_1 \longrightarrow X_1?x [] \overline{X}_2 \longrightarrow X_2?x]; Y!x]$$

Here, either channel may have an action pending, and the process needs to arbitrate between the two choices. When translating this to hardware, this necessitates the introduction of an arbiter circuit. In general, N-way non-deterministic selections can be implemented using an N-way arbiter. In terms of our synthesis procedure, we rewrite the CHP as:

$$*[[\overline{X}_1 \longrightarrow X_1'!X_1; X_1?[]\overline{X}_2 \longrightarrow X_2'!X_2; X_2?]]$$
$$\| \quad *[[\overline{X}_1' \longrightarrow X_1?x [] \overline{X}_2' \longrightarrow X_2?x]; Y!x]$$

where the first process is precisely implemented by the arbiter circuit. The non-determinism has now been decomposed out into a standard form that can be implemented by standard circuits, leaving the original CHP with purely deterministic selections. This process of decomposing out arbiters is performed automatically as part of our synthesis flow.

G. Loops

The next target CHP construct is the repetition, also known as a loop. So far, we have implicitly used a special case of a loop to represent the top-level infinite repetition that characterizes a circuit. In general, a CHP loop is of the form:

$$*[..; *[G_1 \longrightarrow S_1 [] G_2 \longrightarrow S_2 .. [] G_n \longrightarrow S_n]; ..]$$

The guards are Boolean-valued expressions that are required to be pairwise mutually exclusive, but may not be an `else`. As opposed to selections, loops are non-blocking, i.e. if none of the guards are true, then control passes past the loop to the downstream program. If the loop consists of a single branch

whose guard is the elementary Boolean expression `true`, then the loop reduces to an infinite loop: $*[true \rightarrow S]$. The top-level infinite repetition $*[S]$ that has been implicitly used so far is essentially a shorthand notation for $*[true \rightarrow S]$. Conceptually, loops are the most complex constructs in CHP and the problem of synthesizing a circuit that implements them in the general case is a difficult one. In the simple case of a single branch, with a guard that is identically true, we have shown in prior sections that a ring of C-elements is a valid implementation. However, extending it to several branches directly is quite complex and results in a circuit that is expensive. Firstly, note that top-level loops with multiple branches, i.e. ones of the form:

$$*[G_1 \longrightarrow S_1 [] G_2 \longrightarrow S_2 .. [] G_n \longrightarrow S_n]$$

can be rewritten into a single-branch loop with a multi-branch selection as:

$$*[\ [G_1 \longrightarrow S_1 [] G_2 \longrightarrow S_2 .. [] G_n \longrightarrow S_n]\ ]$$

Note that these are equivalent since loop guards can only consists of local variables. If the first program is non-terminating, then the behavior of the second program is indistinguishable from that of the first. If the first program terminates, i.e. all guards are false, then the second program would be deadlocked since selections in CHP are blocking. In this case as well, the behaviors of the two programs are indistinguishable from the perspective of an external observer. Hence, this rewrite is justified. This addresses the case for when the multi-branch loop is at the top-level. Once this rewrite has been performed, any remaining multi-branch loops must be within another loop. In general, these are of the form described at the start of this section. Instead of synthesizing these loops directly, we excise them into a separate process. This is justified by the fact that the circuit complexity for the two strategies is almost identical.

In order to derive the 'excision' strategy for these loops, recall that we have already computed live variable information at all points in the program. Hence, we can extract this loop out into a separate process, and insert data communications appropriately in order for the loop to perform the correct computation. As a simple example, consider the following CHP, which computes the greatest common divisor of two numbers via Euclid's algorithm:

$$*[\ X?x;\ Y?y;\ *[x > y \longrightarrow x := x - y \ [] \ y > x \longrightarrow y := y - x];\ O!x\ ]$$

Here, the internal loop requires two values/inputs to operate (x and y) and returns another value/output (x). In general, the number of input and output variables can both be greater than one. In this case, we simply concatenate them all and transmit them as one, in order to minimize the umber of channel actions required, which directly minimizes the control overhead.

To perform the excision, we factor out the loop into a separate infinite loop, similar to factoring out code into a function call. The rewritten CHP results in two loops operating in parallel, properly synchronized through the fresh channels L_s and L_f:

$$\begin{array}{l} *[\ X?x;\ \ Y?y;\ \ L_s!\{x,y\}, L_f?x_2;\ \ O!x_2\ ]\ \ \| \\ c := 0; \\ *[\ [c = 0 \longrightarrow\ L_s?\{x_1, y_1\}, c := 1 [] else \longrightarrow\ skip]; \\ \quad [x_1 > y_1\ \ \longrightarrow\ \ x_1\ :=\ x_1 - y_1 \\ \quad [] y_1 > x_1\ \ \longrightarrow\ \ y_1\ :=\ y_1 - x_1 \\ \quad [] else\ \ \longrightarrow\ \ L_f!x_1, c := 0]\ ] \end{array}$$

The reader can verify that the CHP is equivalent to the original and is now composed of two programs operating in parallel, each of which contains only the top-level simple loop, and can hence be synthesized based on techniques discussed in prior sections.

We now state the general method to excise loops. Consider any nested/internal loops of the form below:

$$*[..; *[G_1 \longrightarrow S_1 [] G_2 \longrightarrow S_2 .. [] G_n \longrightarrow S_n]; ..]$$

The appropriate rewritten CHP for this is:

$$\begin{array}{l} *[..; L_s!\{x_1, x_2, .., x_n\}, L_f?\{y_1, y_2, .., y_m\}; ..]\ \ \| \\ c := 0; \\ *[\ [c = 0 \longrightarrow L_s?\{x'_1, x'_2, .., x'_n\}, c := 1 \\ \quad [] c = 1 \longrightarrow skip]; \\ \quad [G'_1 \longrightarrow S'_1 [] G'_2 \longrightarrow S'_2 .. [] G'_n \longrightarrow S'_n \\ \quad [] else \longrightarrow L_f!\{y'_1, y'_2, .., y'_m\}, c := 0]\ ] \end{array}$$

where G'_i and S'_i are the same as G_i and S_i respectively, with the variables replaced by their appropriate primed counterparts. The variable c is a one-bit variable and hence one of the two branches in the first selection will always be true. This transformation is performed automatically as a pre-processing step on the input CHP, prior to circuit generation. Finally, note that there can be several levels of nesting of loops within other loops and in this case, the excision is performed bottom up, recursively.

H. Do-Loops

The method for handling do-loops is quite similar to that of loops. Consider any internal do-loop of the form $*[..; *[S_1 \longleftarrow\ G_1]; ..]$. The appropriate rewritten CHP for this is:

$$\begin{array}{l} *[..; L_s!\{x_1, x_2, .., x_n\}, L_f?\{y_1, y_2, .., y_m\}; ..]\ \ \| \\ c := 0; *[[c = 0 \longrightarrow L_s?\{x'_1, x'_2, .., x'_n\}, c := 1 \\ \qquad\qquad [] c = 1 \longrightarrow skip];\ \ S'_1; \\ \qquad\qquad [G'_1 \longrightarrow skip \\ \qquad\qquad [] else \longrightarrow L_f!\{y'_1, y'_2, .., y'_m\}, c := 0]] \end{array}$$

where G'_1 and S'_1 have the same meaning as in the case of loops. As before, we are left solely with CHP contructs for which the synthesis procedure is already defined. Finally, an interesting observation to be made is that loops can be converted to do-loops and vice versa. The conversion from a do-loop to a loop is straightforward. The conversion of a loop to a do-loop can be performed as follows. Consider a loop of the form: $*[G_1 \rightarrow S_1 [] G_2 \rightarrow S_2 .. [] G_n \rightarrow S_n]$.

This can be rewritten into a do-loop, with a simple guard that essentially checks if the original loop has terminated, and a selection inside it as shown below:

$$*[\ c := 1; [G_1 \longrightarrow S_1 [] G_2 \longrightarrow S_2 .. [] G_n \longrightarrow S_n \ [] else \longrightarrow c := 0]\ \longleftarrow\ \ c = 1\ ]$$

This conversion to do-loops is also a part of the pre-processing that is performed in Maelstrom. Do-loops also have the property of being easier to deal with from the perspective of data-dependency analyses, since the body of the loop is guaranteed to execute at least once. Since loops and do-loops are inter-convertible as shown above, in the next section, we will assume that loops have already been converted to the equivalent do-loops and deal solely with these.

I. Multiple Channel Accesses

The synthesis procedure described so far has overlooked an important issue, which will be addressed in this section. In order to understand the source of the problem, first consider the following simple CHP: $*[X?x; X?x; Y!x]$

This is a simple linear program of the form described in Section III-A, and at first glance, might look like it can be synthesized directly. According to the aforementioned method, this would be rewritten as:

$$*[S_0; X?x; S_1] \quad \| \quad *[S_1; X?x; S_2] \quad \|$$
$$*[S_2; Y!x; S_3] \quad \| \quad *[S_0; S_3]$$

When this is implemented as a circuit, there is now a wiring conflict. Both X actions are on the same channel, i.e. there is a unique pair of request and acknowledge wires between the process and its environment, defining this channel. If two C-elements are instantiated, each driving the request/acknowledge wire, this would result in multiple drivers existing for the same node, which is incorrect. In order to rectify this, the input CHP needs to be rewritten into a form where each channel is accessed at most once in a single iteration. Syntactically, this is equivalent to every channel appearing at most once in the CHP description of the process.

This is achieved with the help of fresh channels that act as aliases for the original channel. For the example above, this can be achieved by introducing aliases X_1 and X_2. In order to correctly manage accesses of these new channels and the single old channel, a corresponding handler process is also created, as follows:

$$*[X_0?x; X_1?x; Y!x] \quad \|$$
$$s := 0; \quad *[X?z;$$
$$\quad [s = 0 \longrightarrow X_0!z, s := 1 [] s = 1 \longrightarrow X_1!z, s := 0]]$$

which results in the original channel (and all newly introduced channels) syntactically accessed only once, which is compatible with our circuit generation methodology as there will be no longer be any multiple driver conflicts when each channel access is realized with a circuit element. Essentially, the handler process correctly sequences accesses of the channel based on their order in the original program. The case of linear programs is straightforward, since the ordering of the accesses of the fresh alias channels can be statically determined. However, once control flow is introduced through the use of selections and loops, there is extra information that needs to be communicated to the handler process in order for correct handling. To see how this works, consider the following program fragment:

$$*[..; \quad A?x; \quad ..;$$
$$\quad [G_1 \longrightarrow [G_{11} \longrightarrow .. [] G_{12} \longrightarrow A?x]; ..$$
$$\quad [] G_2 \longrightarrow ..]; \quad ..;$$
$$\quad *[\quad ..; \quad A?x; \quad .. \quad \longleftarrow \quad G_3]; \quad ..; \quad (p)$$
$$\quad [G_4 \longrightarrow .. [] G_5 \longrightarrow ..]; \quad ..]$$

where all accesses of A are explicitly stated, and a program fragment (..) denotes arbitrary CHP that does not contain accesses of A. Here, after replacing A with freshly generated aliases, there is no sequencing on accesses of these aliases that can be determined without actually executing the program, since the sequence of channel accesses is data-dependent. Hence, the handler process needs information about the execution state of the original program. To be precise, the handler requires information about which branches were taken in the program.

However, notice that complete state information does not have to be transmitted. Intuitively, it is obvious that if there are selections/loops where the channel is not accessed at all, then the choice of branch in that particular selection/loop is inconsequential to the channel access handler. In the example above, once the program is at point p, the handler does not need to know whether G_4 or G_5 was true, since there are no accesses of A within that selection and regardless of which branch is taken, the next access will occur at the one marked in red.

In order to generate the handler process, we first perform a pre-processing step, which involves building a state table from the CHP program. A CHP program, like any other program, is a directed graph, with vertices representing elementary actions and edges representing control flow; and translating this into a state transition table is straightforward to achieve. The annotated CHP, with channel accesses of A replaced by accesses on freshly generated alias channels $\{A_i\}$ is shown below:

$$*[..; \quad (1)A_1?x; \quad ..;$$
$$\quad [G_1 \longrightarrow (8)[G_{11} \longrightarrow (9)..[] G_{12} \longrightarrow (2)A_2?x]; ..$$
$$\quad [] G_2 \longrightarrow (5)..]; (4) \quad ..;$$
$$\quad *[\quad (3)..; \quad A_3?x; \quad .. \quad \longleftarrow \quad G_3]; (6) \quad ..;$$
$$\quad [G_4 \longrightarrow .. [] G_5 \longrightarrow ..]; (7) \quad ..]$$

where the numbers in red encode the state of the program. A state transition table that corresponds to this is shown in Table I (left). For convenience, the state encoding is chosen such that states $\{1, .., n\}$ are the states where an alias access occurs, but this is not necessary for correctness. The condition $True$ is used to denote an unconditional state transition.

The state transition table can be optimized as follows to remove redundant transitions. Consider state 5, where there is no alias access. The only possible state transition from here is an unconditional one to state 4. Hence, the seventh row in Table I (left) can be removed and the second row can be modified such that the resulting state for that row is 4 instead of 5. Precisely the same optimization can be applied to the transition from 6 to 7, and several others. This is in effect, computing the transitive closure of the unconditional state transitions, when the source state is one where an alias access does not occur. The final optimized state table is shown

TABLE I: State Transition Table for CHP with Multiple Channel Accesses for Channel A, before (left) and after (right) optimization.

Current State	Condition	Next State
1	G_1	8
1	G_2	5
8	G_{11}	9
8	G_{12}	2
9	$True$	4
2	$True$	4
5	$True$	4
4	$True$	3
3	G_3	3
3	$\neg G_3$	6
6	$True$	7
7	$True$	1

Current State	Condition	Next State
1	G_1	8
1	G_2	3
8	G_{11}	3
8	G_{12}	2
2	$True$	3
3	G_3	3
3	$\neg G_3$	1

in Table I (right), and the corresponding annotated CHP is shown below:

$$
\begin{array}{l}
*[..;\ \ (1)A_1?x;\ \ ..; \\
\quad [G_1 \longrightarrow (8)[G_{11} \longrightarrow ..[]G_{12} \longrightarrow (2)A_2?x];.. \\
\quad []G_2 \longrightarrow ..];..; \\
\quad *[\ (3)..;\ \ A_3?x;\ \ ..\ \ \longleftarrow\ \ G_3];\ \ ..; \\
\quad [G_4 \longrightarrow ..[]G_5 \longrightarrow ..];\ \ ..]
\end{array}
$$

Once this optimized state encoding and transition table have been generated, the handler process can be generated. We simply introduce communications of branch decisions on fresh channels and construct the handler process as a state machine:

$$
\begin{array}{l}
*[..;\ \ A_1?x;\ \ ..;\ \ C_{g1}!(G_1?\ \ 0:1); \\
\quad [G_1 \longrightarrow C_{g2}!(G_{11}?\ \ 0:1); \\
\qquad\qquad [G_{11} \longrightarrow ..[]G_{12} \longrightarrow A_2?x];.. \\
\quad []G_2 \longrightarrow ..];\ \ ..; \\
\quad *[\ ..;\ \ A_3?x;\ \ ..;\ \ C_{g3}!(G_3?\ \ 0:1) \longleftarrow\ \ G_3];\ \ ..; \\
\quad [G_4 \longrightarrow ..[]G_5 \longrightarrow ..];\ \ ..] \\
\|\ \ s := 1; \\
*[\ [s = 1 \vee 2 \vee 3 \longrightarrow A?x[]else \longrightarrow skip]; \\
\quad\ [s = 1 \longrightarrow A_1!x, (C_{g1}?c; nxt := (c = 0)?\ \ 8:3) \\
\quad\ []s = 2 \longrightarrow A_2!x, nxt := 3 \\
\quad\ []s = 3 \longrightarrow A_3!x, (C_{g3}?c; nxt := (c = 0)?\ \ 3:1) \\
\quad\ []s = 8 \longrightarrow C_{g2}?c; nxt := (c = 0)?\ \ 3:2 \\
\quad\ ];\ \ s := nxt\ \ ]
\end{array}
$$

The reader can verify that these two process in parallel do indeed implement the required behavior. Note that in practice, we actually re-encode the states $\{1, 2, 3, 8\}$ as $\{0, 1, 2, 3\}$ since this would enable us to represent the s and nxt variables with only two bits. In case several channels are accessed more than once, there are more handler process that need to be generated, one per channel that is accessed multiple times. The entire procedure described here is performed automatically by Maelstrom as a pre-processing step, before circuit synthesis.

IV. Additional Synthesis Styles

A. Two-Phase Datapath Circuits

As described earlier, at the end of section III-A, the positive transition wave that propagates through the circuit completes before the negative transition wave starts. This fact allows us to extend our approach to also support data processing on both phases of the handshake easily, i.e. use a two-phase protocol instead of a four-phase.

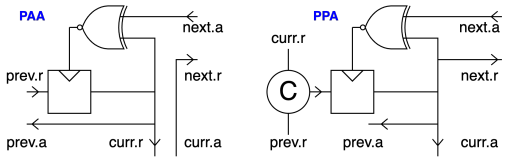


Fig. 6: The PAA and PPA sequencer elements for the Mousetrap pipeline circuit family.

In order to achieve this, we first need to change the datapath circuitry so that it is triggered on both edges, instead of just the positive. For the bundled data circuit family, this simply means changing the pulse-generator for the latch to trigger on both edges. For the QDI circuit family too, there exist straightforward ways to effect a similar change. Next, the implementation of selections assumes that a handshake will only occur on one of the branches, and will reset before another branch is used. To convert this to a two-phase compliant circuit, we need to replace the OR-gates used in the selection with XOR-gates, and replace the 2-input AND-gates in the selection split with 2-input XOR gates.

We note that using a two-phase protocol presents a minor overhead in terms of circuit complexity and thus area. Further, the power consumption of the datapath is also theoretically doubled. However, the throughput of the system doubles as well, as expected, and this is a fair trade. For designs where power constraints are relatively relaxed, having the option to obtain a $2\times$ improvement factor with no added design effort presents a significant advantage.

B. Other Control Circuit Families

So far, we have built our synthesis strategy upon the fundamental C-element micropipeline. But, as mentioned earlier, the methodology is agnostic to the underlying circuits that are actually used to build the control-flow ring elements. As long as sequencers, parallel and selection blocks can be built, then the synthesis strategy can be used. In Figs. 6 and 7, we show the 3-action sequencers for the MOUSETRAP [24] and GasP [25] circuit families. We omit the parallelizers and selection splits and merges, but it is easy to see how these can be constructed, following the same reasoning as for the QDI family. These sequencers have the same timing constraints as the FIFO control circuits that they are derived from. These constraints need to be obeyed in order for correct operation, unlike the QDI control circuits.

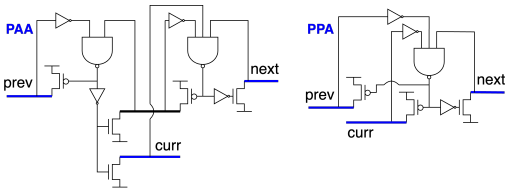


Fig. 7: The PAA and PPA sequencer elements for the GasP pipeline circuit family.

For the MOUSETRAP family, the ITB that initializes the ring, and the parallel split/merge elements are exactly the same as the ones that are used for the C-element family. The selection split and merge elements are slightly different due to the nature of the controller. Reset is incorporated into the ITB and the latches, which reset with their output low.

The GasP family sequencers can also be derived from the GasP FIFO control circuit. The ITB strategy for GasP is significantly different, since it uses a single wire (referred to as the "state conductor") for sequencing. Here, ring elements can be connected in a loop, with all state conductors initially high. The ITB is simply a pulse generator that pulls the first state conductor low when Reset is deasserted. As time progresses, the ring will oscillate without any explicit resetting by an ITB. Again, the selection and parallel blocks can be derived following similar reasoning as before.

We show the complete synthesized circuit for the merge program $(*[C?c; [c = 0 \rightarrow L_0?x [] c = 1 \rightarrow L_1?x]; R!x])$ in Fig. 8. We show the circuits obtained for the QDI and the Mousetrap control families. The datapath circuits that are obtained are exactly the same for both cases.

V. Evaluation

Maelstrom was used to automatically synthesize complex CHP programs into circuits, in a 65nm technology node. As mentioned before, for combinational logic that implements datapath arithmetic, we use the ABC logic synthesis system for our results [26]. We use the standard combinational synthesis script *resyn2* [27], provided with ABC. We also support the open-source Yosys [28] tool (which uses ABC under the hood) and Cadence's Genus synthesis system as alternatives for combinational logic synthesis. Since we use a single-rail bundled datapath and match the delay in the control path using delay lines, the output circuits from ABC are allowed to glitch, as long as they settle within the time constraint imposed by the delay line. For comparison, we synthesize equivalent Balsa programs to Verilog using the Balsa synthesis system [12], and generate a SPICE netlist from the generated Verilog output. Despite Balsa's age, it is the latest complete general-purpose (i.e. supporting slack elastic and inelastic programs) logic synthesis tool for asynchronous circuits in the literature. We also compare against `chp2prs`, an existing naıve SDT method for translating CHP to circuits.

A. Quantitative Results

We report cycle time (a.k.a. cycle period, the inverse of throughput) [29]–[31] and energy-per-cycle metrics from pre-layout SPICE simulations. We also report layout area from a placed, power-routed and global-routed design of 100 instances of each circuit. The reported number is an average, per-instance area, across the 100 instances. The results of our comparison are summarized in Table II, and we use the geometric mean to compare normalized metrics (where 1.0 corresponds to Maelstrom). Across our test cases, on average, our synthesis method shows an average improvement of 39% in area, 50% in cycle time, and 58% in energy-per-cycle over the Balsa synthesis method. We would also like to note that the test case complexity is somewhat limited since Balsa-generated Verilog netlists for more complex programs often displayed incorrect functionality, and these issues could not be resolved due to Balsa no longer being an active project.

The 2-phase results represent the case where the datapath is activated on both phases of the handshakes that propagate through the ring, instead of just on the positive half-phase. The results from Maelstrom 2-phase are in line with what would be expected, with approximately half the cycle-time (0.52×) and energy-per-cycle (0.55×) of Maelstrom. The area for Maelstrom 2-phase is slightly higher due to the added circuit complexity of activating the datapath in both phases (1.04×). Note that the cycle time results for the 4-phase synthesis can be

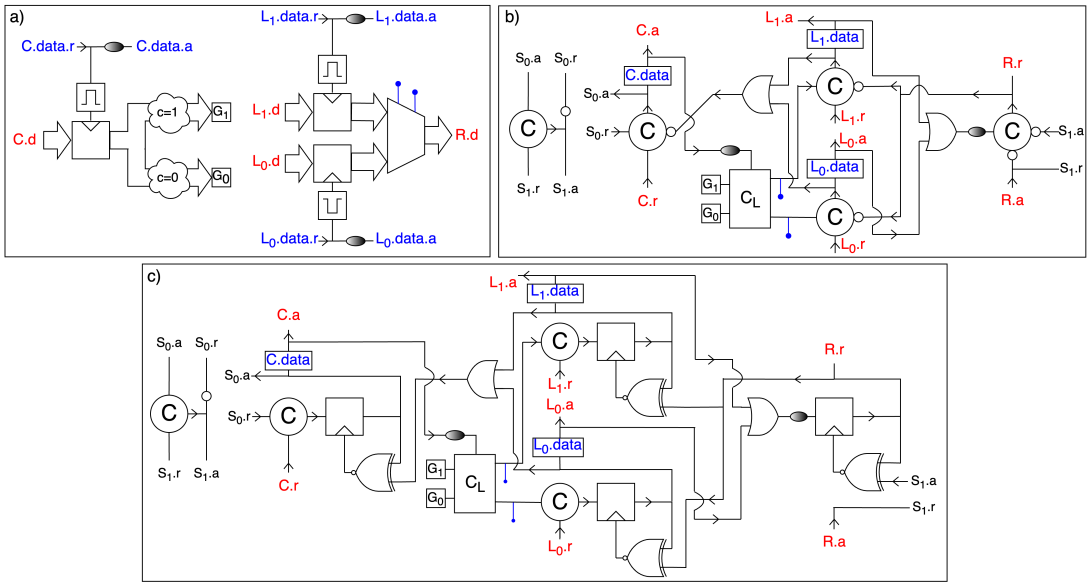


Fig. 8: Synthesized circuit for a merge. Primary inputs and outputs to the process are shown in red. Control-data interface wires are shown in blue. a) Datapath circuit. b) QDI C-element-based control circuit. c) Mousetrap-based control circuit.

TABLE II: Results From SPICE Simulations of Synthesized CHP and Equivalent Balsa programs in a 65nm process. Lower values are better across all metrics. Averages are geometric means, with larger numbers corresponding to greater factors of improvement for Maelstrom. Maelstrom 2-phase produces larger circuits but performs better in terms of delay and energy.

Program	CHP	Method	Area		Cycle Time		Energy	
			μm^2	Ratio	ns	Ratio	pJ	Ratio
Buffer	$*[L?x; R!x]$	Maelstrom (2-phase)	**299** (310)	- (1.04)	0.35 (**0.22**)	- (0.61)	0.10 (**0.06**)	- (0.60)
		chp2prs	1258	4.22	4.25	12.01	0.86	8.48
		Balsa	510	1.71	1.32	3.73	0.36	3.58
Sequence	$*[L_1?x_1; R_1!x_1; L_2?x_2; R_2!x_2; L_3?x_2; R_3!x_3; L_4?x_4; R_4!x_4]$	Maelstrom (2-phase)	**1498** (1557)	- (1.04)	2.73 (**1.44**)	- (0.53)	0.61 (**0.40**)	-
		chp2prs	3595	2.83	25.66	9.40	3.60	5.90
		Balsa	2084	1.64	6.22	2.28	2.67	4.38
Parallel	$*[L_1?x_1, L_2?x_2, L_3?x_3, L_4?x_4; R_1!x_1, R_2!x_2, R_3!x_3, R_4!x_4]$	Maelstrom (2-phase)	**1553** (1620)	- (1.04)	1.06 (**0.56**)	- (0.53)	0.61 (**0.37**)	- (0.61)
		chp2prs	5178	3.32	6.24	5.89	6.97	11.37
		Balsa	2527	1.62	1.94	1.83	1.19	1.94
Adder	$*[L_1?x_1, L_2?x_2; R!(x_2 + x_1)]$	Maelstrom (2-phase)	**1215** (1236)	- (1.02)	2.42 (**1.26**)	- (0.51)	0.35 (**0.23**)	- (0.67)
		chp2prs	2022	1.66	6.38	2.64	2.52	7.28
		Balsa	2760	2.27	3.31	1.37	0.94	2.71
Multiplier	$*[L_1?x_1, L_2?x_2; R!(x_2 * x_1)]$	Maelstrom (2-phase)	**2420** (2470)	- (1.02)	3.66 (**1.91**)	- (0.52)	0.51 (**0.32**)	- (0.63)
		chp2prs	3114	1.29	5.71	1.56	2.68	5.27
		Balsa	-	-	-	-	-	-
Split	$*[C?c; L?x; [c = 0 \rightarrow R_1!x [] c = 1 \rightarrow R_2!x]]$	Maelstrom (2-phase)	**684** (711)	- (1.04)	1.45 (**0.73**)	- (0.51)	0.28 (**0.18**)	- (0.66)
		chp2prs	2879	4.21	10.05	6.93	2.85	10.33
		Balsa	1316	1.93	2.33	1.61	0.39	1.42
Merge	$*[C?c; [c = 0 \rightarrow L_1?x [] c = 1 \rightarrow L_2?x]; R!x]$	Maelstrom (2-phase)	**1124** (1193)	- (1.06)	1.63 (**1.13**)	- (0.69)	0.30 (**0.15**)	- (0.50)
		chp2prs	2981	2.56	15.14	9.29	2.36	7.87
		Balsa	2051	1.82	2.59	1.59	0.44	1.45
GCD	See Appendix A	Maelstrom (2-phase)	**4930** (5072)	- (1.03)	17.22 (**8.93**)	- (0.52)	9.83 (**5.03**)	- (0.51)
		chp2prs	8881	1.81	374.52	21.75	85.60	8.71
		Balsa	6488	1.31	37.64	2.19	28.35	2.89
Fibonacci		Maelstrom (2-phase)	**3921** (4043)	- (1.03)	20.65 (**11.87**)	- (0.57)	8.43 (**4.53**)	- (0.54)
		chp2prs	8150	2.08	212.65	10.30	104.80	12.44
		Balsa	5983	1.53	57.12	2.77	17.33	2.06
Bresenham		Maelstrom (2-phase)	**14275** (14543)	- (1.02)	27.23 (**15.65**)	- (0.57)	18.50 (**9.60**)	- (0.52)
		chp2prs	19276	1.44	312.39	11.47	163.20	8.82
		Balsa	20107	1.51	64.12	2.35	44.80	2.42
Average excluding multiplier	Balsa vs. Maelstrom (1.0)			**1.64**		**1.99**		**2.38**
	Balsa vs. Maelstrom 2-phase (1.0)			**1.58**		**3.84**		**4.30**
	Maelstrom 2-phase vs. Maelstrom (1.0)			**1.04**		**0.52**		**0.55**

improved significantly by the use of asymmetric delay lines which match the logic delay on the positive half-phase but have much lower delay on the other phase where computation is not performed; this improvement applies to circuits where datapath delay dominates, and we plan to incorporate this improvement in future work. Balsa's sequencer optimizations preclude this natural extension to 2-phase datapath activation, as opposed to the (unavailable) Haste/TiDE tools, which could be configured to produce 2-phase circuits. The primary reason that the SDT approach is outperformed by Maelstrom is the difference in the core sequential synthesis algorithm (Section III). SDT works by having a statement request the execution of a sub-statement, wait for its completion, and then report that the statement is complete [10]. Applying this repeatedly leads to a "telescoping" effect—the execution of nested statements are "telescoped" by the outer statements waiting for completion. Maelstrom's approach avoids this overhead, leading to better cycle time.

The cycle time (and therefore the throughput) of an asynchronous circuit that is synthesized by a sequential method depends on the exact input CHP specification. For example, consider the two following programs:

$$*[L_1?x_1; L_2?x_2; [x_1 > x_2 \longrightarrow y := (x_1 - x_2); R!y \;[]\; else \longrightarrow y := (x_2 - x_1); R!y]]$$

$$*[L_1?x_1, L_2?x_2; R!((x_1 > x_2)?(x_1 - x_2) : (x_2 - x_1))]$$

It should be clear that from an I/O perspective, both perform the same operations—that of calculating the absolute difference of the two inputs. But the first version evaluates $(x_1 - x_2)$ and $(x_2 - x_1)$ conditionally, unlike the second one that computes both. When synthesized, the circuit that is produced for the first program is much more expensive than that of the second due to the presence of selections, extra assignments, and multiple channel accesses. The circuit for the first (second) program has an area of 7453 (2899) μm^2, cycle time of 6.80 (3.02) ns, and energy per cycle of 2.96 (1.05) pJ. The process of analyzing and optimizing programs can be performed at the CHP level, and we plan to incorporate CHP re-structuring optimizations of this kind into our design flow in the future. Further, the throughput of an asynchronous circuit is also input-dependent, and varies from cycle to cycle. Hence, the average throughput is the pertinent measure. Note that the delay introduced by the controller is small since it is just a single gate per program statement, and the overwhelming majority of the delay arises from the delay of the combinational logic that implements the computation, except for circuits like FIFOs that have no logic.

Finally, it is important to note that the circuits using this method are practically on-par with hand-designed dataflow circuits when we examine small designs where it is practical to make the comparison. In particular, the building blocks for dataflow circuits (function, split, merge, copy, initial

TABLE III: Results From SPICE Simulations of Maelstrom-synthesized circuits for large CHP programs.

Design	L-1	L-2	L-3	L-4
CHP Statements	20	31	64	28
Total Runtime (s)	1.59	7.98	6.53	7.83
Maelstrom Runtime (%)	10.17	10.10	11.39	22.99
ABC Runtime (%)	76.72	76.70	75.66	56.01
I/O Runtime (%)	13.11	13.20	12.95	21.00
Area (μm^2)	7081	44731	42238	72063
Cycle Time (ns)	1.12	7.10	34.9	47.2
Energy (pJ)	0.99	6.22	32.37	38.37

token, buffer, source, and sink [14], [32]) have the same area/delay/energy compared to those automatically generated by Maelstrom. In particular, this means that existing dataflow synthesis tools like Fluid can be implemented using CHP-to-CHP mapping followed by Maelstrom, rather than requiring their own custom circuit library.

In the past it was significantly easier to generate asynchronous circuits that implement a given program correctly using SDT, but it took significantly more effort to generate circuits with performance similar to hand-optimized designs. Our work demonstrates a path to high quality automated asynchronous logic synthesis, and presents a significant advancement in the logic synthesis methodology for asynchronous circuits.

Table III summarizes results from running Maelstrom on larger programs. The runtime is dominated by ABC and the interface (I/O) between ABC and Maelstrom, which requires printing expressions for ABC to synthesize. Large expression trees in the fourth test case causes Maelstrom's data-dependency analysis, which is required to instantiate registers correctly, to take up a larger fraction of the runtime.

B. Functional Verification

Synthesis tools are a foundational component of any EDA flow, as the entire infrastructure depends on the accuracy and integrity of the synthesized circuits. The correctness of the circuits generated by these tools directly impacts the functionality and performance of the final chip. To ensure this, we have integrated a robust and extensive suite of tests for automated regression testing within Maelstrom. These tests go well beyond the simple programs listed in Table II, providing a more thorough verification process.

The automated regression tests are designed to synthesize and evaluate CHP programs at multiple levels of complexity, ranging from basic examples to larger, more complex designs. By testing at the level of production-rules, i.e. gate-level, these tests ensure that Maelstrom handles complex control flow correctly and that the synthesized circuits perform as expected.

In addition to this, the test suite also covers the different datapath and control styles supported by Maelstrom. This ensures that the tool can generate correct implementations for diverse design styles, making it a reliable part of the ACT EDA flow. This rigorous testing ensures that the output from Maelstrom meets the high standards required in hardware design, contributing to a stable and reliable workflow for designers.

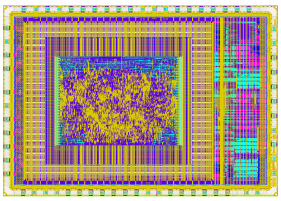


Fig. 9: GDS view of the MD5 hash accelerator chip fabricated in the Skywater 130nm process node.

C. MD5 Hash Accelerator

Maelstrom was also used to automatically synthesize CHP that implements the MD5 hashing algorithm into a transistor-level netlist. The physical design was performed using the ACT design flow [33], in the Skywater 130nm process node. The tape-out was completed using the Caravel test harness from efabless, as part of the chipIgnite solution. The final GDS sent to the foundry is shown in Fig. 9.

VI. Related Work

Due to the high overhead of SDT (Section V), recent work on asynchronous circuit generation has been focused on dataflow-based synthesis [7], [14], [32], [34], [35]. Dataflow synthesis decomposes a CHP program into the concurrent composition of several smaller elementary CHP programs with a fixed structure (dataflow building blocks)—which corresponds to CHP-to-CHP translation. To complete logic synthesis, hand-optimized parameterized circuits for the dataflow building blocks are generated. However, dataflow decomposition *adds* concurrency and relaxes the synchronization behavior of the original CHP program. For example, two data-independent actions that were sequential in the original program may be executed concurrently in the dataflow version [19]. The CHP-to-CHP transformation introduced by dataflow synthesis is only valid for *slack elastic* programs, a condition that usually requires whole program analysis or syntactic restrictions on CHP [8]. High-level synthesis tools for asynchronous circuits such as Fluid [14] are correct because CHP constructs that can lead to slack elasticity violations are absent in C/C++—the source language for Fluid. In other words, SDT is the only previously known *general-purpose* logic synthesis approach that translates general CHP programs into asynchronous circuits.

VII. Conclusion

In this work, we presented a new technique to synthesize behavioral descriptions of asynchronous circuits into gates, and a tool that automatically compiles CHP into these circuits. The approach is agnostic to the underlying control circuit family. The proposed method starts from known high-quality pipeline circuits and uses a simple way to construct sequencer and control-flow elements from them that form a basis set for the synthesis. The method also presents a clear separation between control and datapath circuits, which enables the use of different datapath families. The resulting open-source (available via [9]) synthesis tool improves on the existing state-of-the-art synthesis techniques by a significant factor, in terms of performance metrics such as energy, delay and area.

APPENDIX A

CHP for the iterative GCD computation algorithm:

$$*[\ X?x;\ Y?y; *[x > y \longrightarrow x := x - y \\ [] y > x \longrightarrow y := y - x]; O!x]$$

CHP for the 2n-th Fibonacci number generation algorithm:

$$*[N?n, x := 0, y := 1; *[n > 0 \longrightarrow x := x + y; \\ y := x + y; n := n - 1]; O!x]$$

CHP for Bresenham's Line Algorithm (assuming $x_0 \leq x_1$):

$$*[X_0?x_0, X_1?x_1,\ Y_0?y_0,\ Y_1?y_1;\ dy := y_1 - y_0, \\ dx := x_1 - x_0; D := 2(dy) - dx; \\ *[x_0 \leq x_1 \longrightarrow P_x!x_0, P_y!y_0; \\ [D > 0 \longrightarrow y_0 := y_0 + 1, D := D - 2(dx) \\ [] else \longrightarrow skip]; D := D + 2(dy), x_0 := x_0 + 1]\ ]$$

REFERENCES

[1] S. H. Unger, "Hazards and delays in asynchronous sequential switching circuits," *IRE Transactions on Circuit Theory*, vol. 6, no. 1, pp. 12–25, 1959.

[2] S. M. Nowick and D. L. Dill, "Automatic synthesis of locally-clocked asynchronous state machines," in *IEEE International Conference on Computer-Aided Design*, pp. 318–319, IEEE Computer Society, 1991.

[3] R. Manohar and Y. Moses, "The eventual c-element theorem for delay-insensitive asynchronous circuits," in *IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 102–109, 2017.

[4] A. J. Martin, "Synthesis of asynchronous vlsi circuits," Tech. Rep. CS-TR-93-28, California Institute of Technology, 1991.

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

[6] R. Manohar, "An analysis of reshuffled handshaking expansions," in *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001*, pp. 96–105, IEEE, 2001.

[7] A. M. Lines, "Pipelined asynchronous circuits," Master's thesis, California institute of Technology, 1995.

[8] R. Manohar and A. J. Martin, "Slack elasticity in concurrent computing," in *Mathematics of Program Construction* (J. Jeuring, ed.), (Berlin, Heidelberg), pp. 272–285, Springer Berlin Heidelberg, 1998.

[9] R. Manohar, "chp2prs: Syntax-directed translation of chp programs into production rules." https://github.com/asyncvlsi/chp2prs, 2023.

[10] S. M. Burns and A. J. Martin, "Syntax-directed translation of concurrent programs into self-timed circuits," in *Conference on Advanced Research in VLSI*, 1988.

[11] K. van Berkel and M. Rem, "Vlsi programming of asynchronous circuits for low power," in *Asynchronous Digital Circuit Design*, pp. 151–210, Springer, 1995.

[12] D. Edwards and A. Bardsley, "Balsa: An asynchronous hardware synthesis langugae," *The Computer Journal*, vol. 45, no. 1, pp. 12–18, 2000.

[13] H. S. Inc. https://web.archive.org/web/20090323054030/http://www.handshakesolutions.com/, 2009.

[14] R. Li, L. Berkley, Y. Yang, and R. Manohar, "Fluid: An asynchronous high-level synthesis tool for complex program structures," in *2021 27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 1–8, IEEE, 2021.

[15] C. G. Wong and A. J. Martin, "High-level synthesis of asynchronous systems by data-driven decomposition," in *Proceedings of the 40th annual Design Automation Conference*, pp. 508–513, 2003.

[16] S. Taylor, D. Edwards, and L. Plana, "Automatic compilation of data-driven circuits," in *2008 14th IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 3–14, IEEE, 2008.

[17] J. Hansen and M. Singh, "A fast branch-and-bound approach to high-level synthesis of asynchronous systems," in *2010 IEEE Symposium on Asynchronous Circuits and Systems*, pp. 107–116, IEEE, 2010.

[18] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[19] R. Manohar, T.-K. Lee, and A. J. Martin, "Projection: A synthesis technique for concurrent systems," in *Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 125–134, IEEE, 1999.

[20] I. E. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, p. 720–738, jun 1989.

[21] A. J. Martin, S. M. Burns, T.-K. Lee, D. Borkovic, and P. J. Hazewindus, "The design of an asynchronous microprocessor," *SIGARCH Comput. Archit. News*, vol. 17, no. 4, pp. 99–110, 1989.

[22] A. J. Martin, "Asynchronous datapaths and the design of an asynchronous adder," *Formal Methods in System Design*, vol. 1, pp. 117–137, 1992.

[23] A. V. Aho, M. S. Lam, and J. D. Ullman, *Compilers: Principles, Techniques & Tools*. London, England: Pearson Education, 2007.

[24] M. Singh and S. M. Nowick, "Mousetrap: High-speed transition-signaling asynchronous pipelines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 684–698, 2007.

[25] I. Sutherland and S. Fairbanks, "Gasp: A minimal fifo control," in *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems*, pp. 46–53, IEEE, 2001.

[26] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference*, (Germany), pp. 24–40, Springer, Berlin, 2010.

[27] A. Mishchenko, S. Chatterjee, and R. Brayton, "Dag-aware aig rewriting a fresh look at combinational logic synthesis," in *Proceedings of the 43rd annual Design Automation Conference*, pp. 532–535, 2006.

[28] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, vol. 97, 2013.

[29] W. Hua, Y.-S. Lu, K. Pingali, and R. Manohar, "Cyclone: A static timing and power engine for asynchronous circuits," in *2020 26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 11–19, IEEE, 2020.

[30] S. M. Burns and A. J. Martin, "Performance analysis and optimization of asynchronous circuits," in *Conference on Advanced Research in VLSI*, 1990.

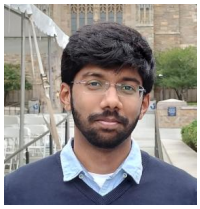
[31] W. Hua and R. Manohar, "Exact timing analysis for asynchronous systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 203–216, 2017.

[32] J. Teifel and R. Manohar, "Static tokens: Using dataflow to automate concurrent pipeline synthesis," in *10th International Symposium on Asynchronous Circuits and Systems*, pp. 17–27, IEEE, 2004.

[33] S. Ataei, W. Hua, Y. Yang, R. Manohar, Y.-S. Lu, J. He, S. Maleki, and K. Pingali, "An open-source eda flow for asynchronous logic," *IEEE Design & Test*, vol. 38, no. 2, pp. 27–37, 2021.

[34] C. G. Wong and A. J. Martin, "High-level synthesis of asynchronous systems by data-driven decomposition," in *Proceedings of the 40th annual Design Automation Conference*, pp. 508–513, 2003.

[35] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An asic flow for ghz asynchronous designs," *IEEE Design & Test of Computers*, vol. 28, no. 5, pp. 36–51, 2011.



Karthi Srinivasan is a PhD candidate in Electrical and Computer Engineering at Yale University. He received a B.Tech+M.Tech in Electrical Engineering from the Indian Institute of Technology Madras. His research interests lie broadly in the field of logic synthesis and optimization. He currently works on developing EDA tools for asynchronous circuits.

Rajit Manohar is the John C. Malone Professor of Electrical and Computer Engineering and Computer Science at Yale University. He received a PhD in computer science from the California Institute of Technology. His group conducts research on the design, analysis, and implementation of self-timed systems. He is the recipient of twelve best paper awards, nine teaching awards, and was named to MIT technology review's top 35 young innovators under 35 for contributions to low power microprocessor design.