Maelstrom: A Logic Synthesis Technique for Asynchronous Circuits

Karthi Srinivasan
Department of Electrical Engineering
Yale University
New Haven, USA
karthi.srinivasan@yale.edu

Rajit Manohar
Department of Electrical Engineering
Yale University
New Haven, USA
rajit.manohar@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, in a 65nm node, of generated netlists of several CHP programs indicate a significant performance improvement over the current art.**

***Index Terms*—Asynchronous, Logic Synthesis, Pipelines**

I. Introduction

Logic synthesis is the process of converting an abstract behavioral description of a circuit into actual gates. 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, it becomes impractical to specify the entire circuit at this level of detail. In synchronous design, RTL-level hardware description languages such as Verilog [1] are used to specify the underlying Boolean formula in a more user-friendly format. This RTL description is synthesized into sequential and combinational logic. The synthesis methodology translates the RTL into combinational logic and clocked elements through a built-in supported clocking discipline; RTL that does not conform to this discipline is viewed as "unsynthesizable."

For asynchronous circuits, the equivalent problem of logic synthesis has had several solutions proposed over the decades. One of the first methods to synthesize state-machines used Huffman flow-tables [2], and compiled this into circuits, with delay lines forming a feedback loop, to hold state. Other methods include burst-mode [3], which was closely related to synchronous design, by virtue of its use of a local timing signal that served the role of the clock.

However, such these methods assume a bound on the delay of circuit elements, which leads to designs that assume the worst-case delays, reducing one of the potential benefits of asynchronous logic. At the other extreme are delay-insensitive circuits, which assume unbounded delays on gates and wires. However, this restricts the class of implementable functions severely [4]; instead, a slightly relaxed version, known as quasi-delay-insensitive circuits are used where relative wire delays are assumed to be small relative to gate delays, which can still be unbounded. Synthesis approaches for these include Martin's synthesis method [5] of handshaking expressions (HSE) to circuits, and Petrify [6], 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 clocked circuits. This renders these methods very time-consuming and error-prone.

In order to enable ease of specification of a complex asynchronous system, we use a behavioral abstraction of asynchronous circuits, known as CHP (Communicating Hardware Processes). CHP is a hardware description language where an asynchronous circuit is specified as a collection of message-passing parallel programs (called processes), all running in parallel. Each process has some internal variables and an associated program that governs the actions that the process performs. Each process can also contain communication channels, which are used to communicate values between the process and its environment (i.e. other processes in the system). This level of abstraction is closer to the input to traditional high-level synthesis (HLS) tools.

Existing methods to translate CHP to circuits include syntax-directed translation (SDT) and dataflow synthesis with templated circuits. A direct implementation of syntax-directed translation is the open-source `chp2prs` tool that can synthesize CHP into gates. However, the resulting circuits have high control overhead as `chp2prs` implements the SDT algorithm directly without optimizations. Balsa is an optimized implementation of SDT [7], providing an implementation that was as efficient as a (no longer available) commercial SDT tool called Haste/TiDE [8]. Another approach to logic design uses dataflow components as building blocks, and translates a subset of CHP programs into a dataflow component library (either manually [9], or through automated mapping of a limited CHP subset [10], [11]). Dataflow synthesis introduces concurrency, but this transformation changes the synchronization behavior of the CHP program and can change the behavior of the program in the general case [12], [13]. This approach often

results in circuits that are over-pipelined and hence energy- and area-inefficient [14].

In this work, we present a new logic synthesis approach to translating CHP programs into asynchronous logic. Our goal is to perform sequential synthesis, i.e. to obtain a circuit that implements the program exactly as written and that respects the CHP semantics. We compare our results to two existing tools for SDT, showing that we significantly improve upon their results in area, delay, and energy simultaneously. For small designs, the circuits produced by our approach are on par with those that would be designed manually. The method is also quite general, and can easily be extended to any circuit family.

In section II, we formalize our assumptions and describe the source language. Section III contains a derivation of the fundamental sequencer elements that form the basis of the synthesis method. Section IV describes how selections, i.e. conditional executions are implemented. Section V outlines our datapath synthesis strategy. Section VI details how loop-carried dependencies and initial conditions of a program are handled. Section VII shows how our method extends beyond the QDI control circuit family. In section VIII, we compare the new synthesis method against prior art across various metrics. Sections IX and X make some key observations and conclude the paper.

II. Programs and Rings

The abstract program that is to be synthesized is described in CHP, the syntax of which is detailed in Appendix I. Every program is composed of a set of actions (assignments, sends and receives) that perform data processing and communication, and a control flow that determines the order these actions are performed—either conditionally or unconditionally. Typically, the program has a "top-level" infinite loop since we want the resulting circuit to continue performing these actions as long as it is powered on. For example, a simple buffer that receives a value on an input channel L, stores in to local variable x, and sends the received value out on output channel R is written as $*[L?x; R!x]$. The semi-colon indicates that one action must complete before the next begins, and the $*[...]$ syntax denotes an infinite repetition of the program within the brackets.

To start with, we make some assumptions about candidate programs for synthesis. First, each channel must be accessed at most once in one iteration of the top-level loop of the program. Second, the program does not contain any internal (nested) loops. These two assumptions are made in order to generate efficient circuits. Note that these are not restrictions on the programs that can be synthesized, since a program with multiple channel accesses and internal loops can be rewritten systematically into a set of programs that do not have these properties, each of which can then be synthesized independently. The third constraint is that programs cannot have non-deterministic selections, which necessitate the introduction of arbiters. In this case too, arbiters are factored out automatically before proceeding with synthesis. Fourth and most importantly, we assume that there are no shared variables across programs that are running in parallel. Any values that need to be communicated are sent/received over channels. This is a true constraint and must be obeyed by all candidate programs; if this constraint is violated, then correct synthesis requires exploration of the possible interleavings of all the concurrent processes that are involved either directly or indirectly in controlling access to the shared variable. Further, in the rest of this work, we focus on the quasi-delay insensitive (QDI) C-element style pipeline, coupled with a bundled datapath for expositional purposes. However, our synthesis approach is not limited to this circuit family (see Section VII).

Under these assumptions, a candidate CHP program forms a directed acyclic graph (discounting the top-level loop). First, consider the case of a linear program, which is a semi-colon separated sequence of actions. In general, such a program would look like $*[S_1; S_2; ..S_n]$, where each of the S_i are actions. A naive but incorrect way to implement this program would be to take each action in the program and synthesize it into a circuit. However, this would lead to all the statements running in parallel, which is not what the semantics of the program specifies. In order for these actions to occur in the sequence that is specified, we need a controller that enforces the ordering. In the simplest case when each of the S_i are assignments to local variables, then, from Fig. 1 it is easy to see how a token ring would implement this program. Each element in the ring triggers the datapath to perform the action that it implements via the pulse generator (in our standard datapath, we use pulsed latches as state holding elements), delays its output via a delay line that is matched to the delay of the datapath, and then activates the next element. In this way, each action is performed in sequence. Finally, there is a initial token buffer (ITB) that activates the first action and waits for the last action to complete before restarting the whole process, thereby implementing the infinite loop.

But such a program would be of no use, since there is no communication with the environment outside the ring to send or receive values, and internal values cannot be read/modified since we assumed that there are no shared variables. To introduce data into and extract data out of the ring, we need to implement communication actions that can be inserted in place of the local assignments in the example above.

III. Actioning Ring Elements

To see how communication actions can be inserted, notice that from the output of each C-element, to the output of its corresponding delay line, there is really a communication that already exists. This communication action is implicit since it is between a control stage and its corresponding datapath stage. In the case of a bundled datapath, the delay is known and thus after waiting for a sufficient amount of time, the control path and proceed to its next stage without waiting for a signal from the datapath. In order to insert a communication with the environment, we break open the control path at this point, as in Fig. 2 (a), and expose the channel. Additionally, we add an invalidation strengthening to the C-element, in order to ensure that the request is acknowledged before its deassertion.

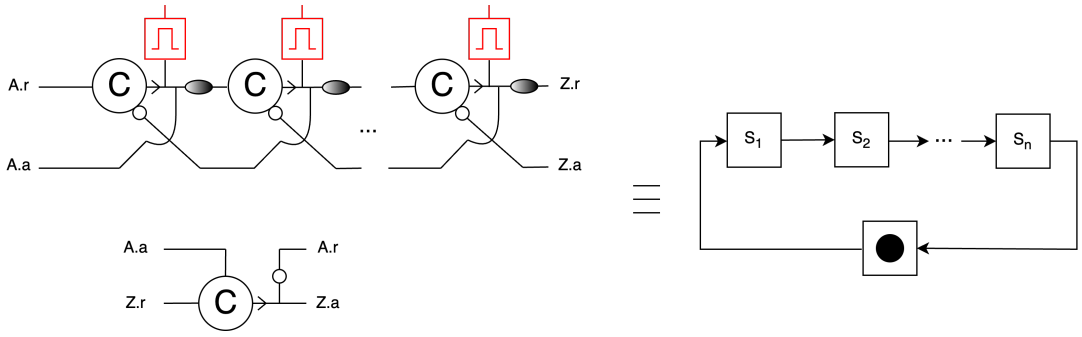

Fig. 1. Implementation of a simple linear program with only local assignments. The arrows denote the output port of the C-element. We show non-CMOS implementable circuits in the interest of clarity. The ellipses represent delay lines that are chosen to match the delay of the datapath. The output of the pulse generator triggers the latches. The figure on the right is an abstract ring used as a shorthand, with the filled circle denoting the token that flows through the ring.

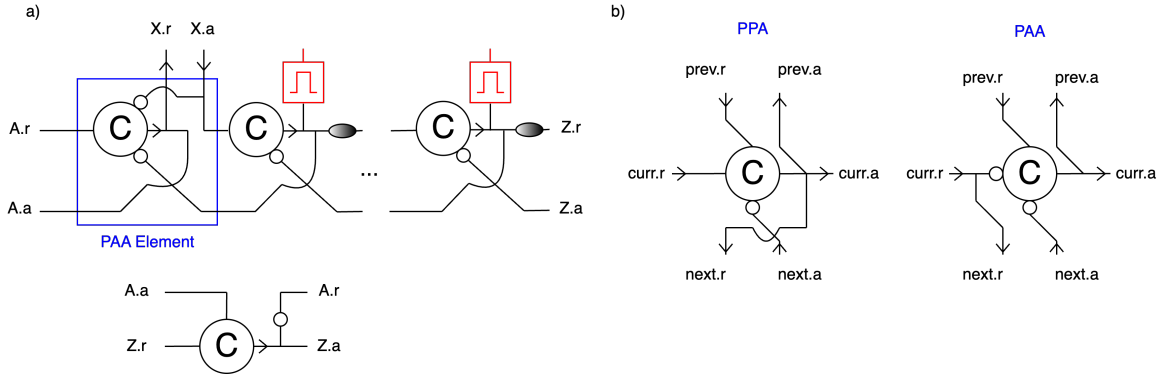

Fig. 2. (a) Derivation of the actioning ring element from the standard C-element pipeline. X is a communication channel with the environment, whose request and acknowledge wires are shown. The C-element on the bottom is the initial token buffer that is used to connect the ring back onto itself to close the loop. (b) The two fundamental actioning ring elements. The naming indicates the type of actions that they sequence, either passive or active.

The resulting pipeline element, with an active communication in place of a matched delay line is the first of the *actioning ring elements*. This PAA (passive-active-active) element sequences three actions of the aforementioned type[1], where the first and third are part of the synchronizations of the ring and the second is the communication with the environment. Fig. 2 (b) shows the two fundamental kinds of ring elements that sequence different action types, which along with the ITB, form a complete basis set for implementing linear programs. It is also possible to use an equivalent but alternate pair of elements where the incoming synchronizations are active and the outgoing are passive (the equivalent notation would be AAP and APP), which would result in a different structure for the ITB. Note that when two passive actions are sequenced, they are more tightly coupled than a semicolon, i.e. they must complete at the same time. In other words, the PPA element waits for both, the incoming synchronization and the action request, and acknowledges both at once.

At this point, we can synthesize any linear program. However, to synthesize programs that have two or more actions in parallel, we need parallelizer elements, shown in Fig. 3(a). The parallel split, as the name implies, splits the request out into the branches that need to proceed in parallel. The parallel merge waits for all the branches to complete, merges the incoming requests and activates the next action in the sequence. Although we have shown a single action in each branch, this need not be true. Each branch can contain any sub-program, which will be synthesized the same way, except instead of being looped back onto itself with an ITB, it will be connected to the ports of the parallel split and merge. The entire synthesis procedure is modular—as the program is traversed, each action has a corresponding ring element stamped down and connected to the existing sequence of rings, and finally the last element is connected back to the first through a dataless ITB that initializes the first action and waits for the last action to complete before enabling the first action again.

An interesting special case occurs when there are three or fewer actions in the program, such as $*[A_1; A_2; A_3]$, which results in a 3-element ring, along with an ITB. Notice that each ring element intrinsically sequences three actions, and hence this program can synthesized with a *single* ring element of the appropriate type, based on the action types. This results

[1]In asynchronous circuits, a passive channel is one that waits for its environment to initiate communication, while an active channel is one that initiates the communication with the environment.

in a circuit that is identical to one that would be designed manually. We apply this optimization automatically whenever possible.

IV. Handling Selections

The final piece for a sequential synthesis is a general way to handle conditional execution of statements, which are written as selections in CHP. In order to route the control in the presence of selections, some extra combinational logic is needed in the control path to choose the right branch, based on the truth value of a set of guards. To see this, consider the following example CHP program fragment (letters in brackets denote the action type—either active or passive):

$$
\begin{array}{l}
*[\ \ldots\ P_1(a); \\
\quad [G_1 \longrightarrow Q_1(a); Q_2(a) \\
\quad [\!] G_2 \longrightarrow \mathit{skip} \\
\quad [\!] G_3 \longrightarrow S_1(p) \\
\quad]; \\
\quad P_2(a);\ \ldots\ ]
\end{array}
$$

where G_i are the guard expressions, only one of which can be true when the selection is reached, and hence only one path can be taken through the selection. In this case, after the action P_1, a decision must be made about which branch to activate, following which the actions in that branch must be executed. Finally, upon reaching the end of a branch, the control must be merged and then proceed with the rest of the program, starting from P_2. We synthesize such a fragment as shown in Fig. 3 (b). We instantiate guard evaluators that output the boolean value on a single rail. The outputs of these blocks, after a certain delay, is guaranteed to be mutually exclusive-high, from the structure of the program. Another combinational logic (C_L) block takes in these guards and the incoming request, and splits the request into the correct branch. This block also delays the incoming request so that the guard evaluators can settle to the correct values, since it is possible that the evaluators can glitch. The C_L block consists of a single delay line, N 1-bit latches and N 2-input AND-gates. The latches latch the guards when the control arrives, and the AND-gates have the latch outputs and a delayed version of the control as their inputs. The final set of outputs is the outputs of the AND-gates. Similarly, since only one branch will be executed, an OR-gate whose inputs will be mutually exclusive-high transmits the acknowledge backwards from the branches. At the end of the selection, another OR-gate whose inputs are also guaranteed to be mutually exclusive-high, merges the requests and activates the post-selection part of the program.

We now have a method to recursively implement any program that satisfies the assumptions that we made in Section II . An empty branch, such as the second one in the above program results in the control directly passing through from the start to the end of the selection. Essentially, skips have no overhead in this synthesis method.

V. Datapath Synthesis Strategy

In the bundled datapath discussed thus far, each variable is synthesized into k sets of N latches, where N is the bitwidth of the variable and k is the number of times the variable is assigned. This strategy is cheaper in terms of transistor count than a single set of N latches per variable, with write and read muxes. This can be understood by observing that two tri-state inverters can be used to implement a latch. However, a 2-to-1 mux is also just two tri-state inverters with their outputs shorted together. Hence, implementing a variable that is written twice as two latches, instead a latch and a mux, is cheaper and also avoids the overhead of combinational logic that is required to share a pulse generator. This strategy is analogous to static single assignment form in the software compiler literature [15] and static token form used for hardware dataflow compilation [11]. For linear programs, the implementation is straightforward, but branched programs present a challenge.

Consider the case of a program that contains a single selection, where a variable is assigned within more than one branch in a selection and is used later, in the linear part of the program. In this case, the correct value of the variable to be used is execution-dependent. To address this, at the end of every selection, we instantiate a mux that correctly combines the values of the variables from the branches in which they were assigned, with the control generated from the selection split block, so that the downstream program has access to the correct value of the variable in that execution of the program. This is analogous to the gated ϕ-functions used in static token form [11].

VI. Initial conditions and loop-carried dependencies

So far, we have synthesized a given program by starting from the first action and walking the program graph, instantiating a ring element of the correct type for every action, a pair of parallel splits and merges for every set of parallel branches, and a pair of selection splits and merges for every selection. This is sufficient if all the variables in a program are discarded at the last action and fresh variables are used when the first action in the ring executes again.

However, some programs do not satisfy this constraint. Several interesting programs are of the form where there are variables that are initialized before the start of the infinite program loop, and used within the program. In order to accommodate these, we now extend the allowed set of programs to include initial assignments as well, so that they are of the form: $x_1 := v_1; ...; x_n := v_n; *[S_1; ...; S_n]$ where $\{x_k : k = 1..n\}$ is some subset of all the variables used in the program. Note that variables that are initialized outside the main loop will typically be loop-carried dependencies as well. It is possible to write inconsequential initial conditions such that the value is discarded and never used, but if the value is used, then every use in some iteration will be the value that was defined in the previous iteration. Further, it is possible that some variables are not defined as initial conditions but are loop-carried dependencies nonetheless. These variables are detected during a pre-synthesis program analysis and are added to the initial conditions so they can be synthesized correctly.

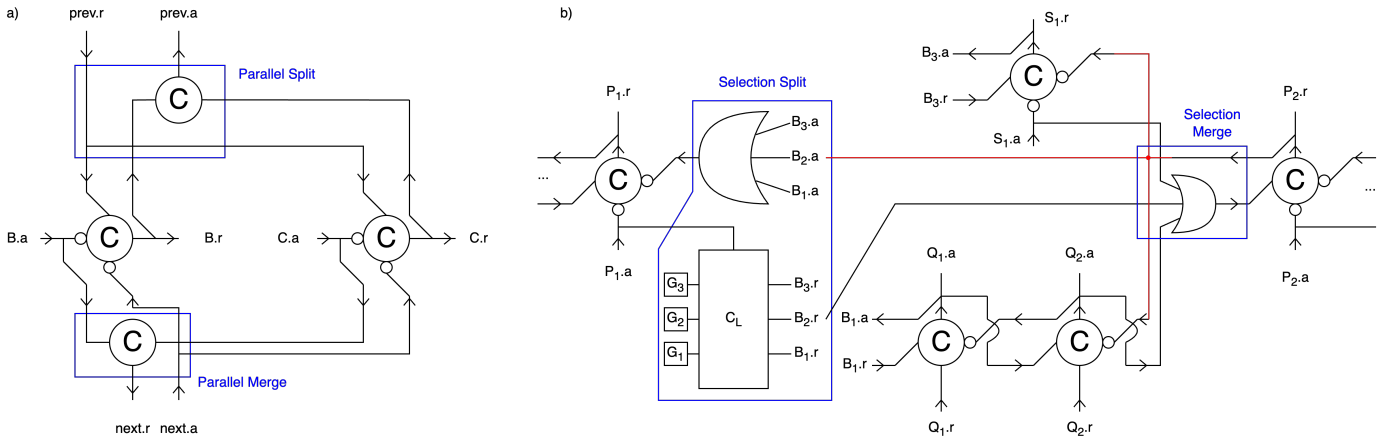

Fig. 3. (a) A 2-way parallelizer element. B and C represent the two actions, both active, that need to occur in parallel. If any were passive, their elements would be changed to the PPA element. (b) Synthesis of the example CHP fragment containing selections. The combinational logic C_L selects one of the branch requests $B_i.r$ based on the guards, when the action P_1 completes. This can be implemented with a set of two input AND-gates, and 1-bit latches. Note that the input request must be delayed sufficiently so that the guards can settle to their correct value.

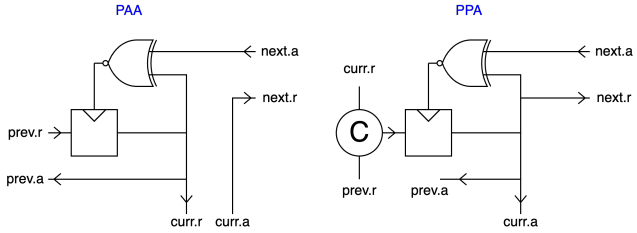

Fig. 4. The PAA and PPA sequencer elements for the Mousetrap pipeline circuit family. The selection and parallel blocks follow the same pattern as the ones for the C-element family.

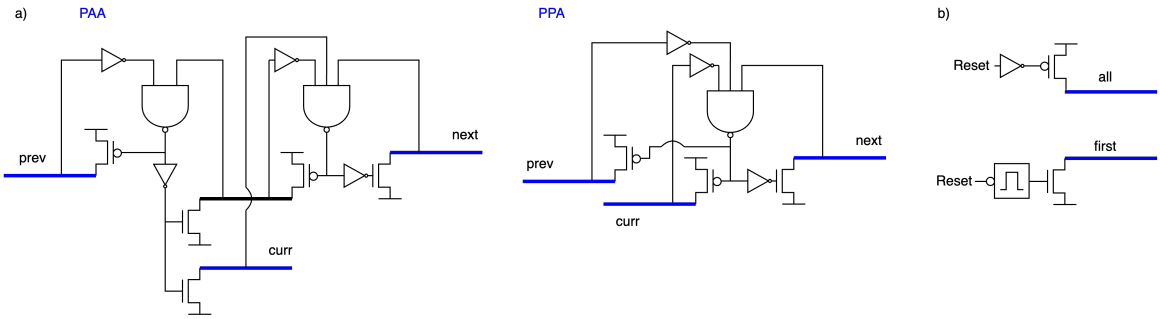

Fig. 5. (a) The PAA and PPA sequencer elements for the GasP pipeline circuit family. (b) The required circuits to correctly initialize all the state conductors to high and to pull down the state conductor that activates the first action, when Reset is deasserted. The selection and parallel blocks follow the same pattern as the ones for the C-element family.

In order to synthesize programs of this kind, we first iterate through all the initial conditions and instantiate sets of latches that reset with the initial value of the corresponding variable on their outputs. Following this, whenever the program uses the value held by the variable (before assigning to it and invalidating the value), the outputs of these latches are used. So far, it is pretty straightforward. However, another step must be done. Once we step through the entire program and complete the synthesis, all of the variables will have some final value, either on a latch output or on the output of a mux that merged values from several latches. We now connect the outputs of these latches/muxes back to the inputs of the latches that were instantiated during the first initial condition handling stage. Finally, a pipeline element that pulses all of these latches is inserted into the ring as the last action, effectively implementing the carrying of the values around the loop, from one iteration to the next.

Program	CHP	Method	Area (μm^2)	Ratio	Cycle Time (ns)	Ratio	Energy (pJ)	Ratio
Buffer	$*[L?x;$ $R!x]$	Maelstrom	299	-	0.35	-	0.101	-
		Maelstrom 2-phase	310	1.04	0.22	0.61	0.061	0.60
		chp2prs	1258	4.22	4.25	12.01	0.858	8.48
		Balsa	510	1.71	1.32	3.73	0.362	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	1498	-	2.73	-	0.610	-
		Maelstrom 2-phase	1557	1.04	1.44	0.53	0.401	0.66
		chp2prs	3595	2.83	25.66	9.40	3.604	5.90
		Balsa	2084	1.64	6.22	2.28	2.672	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	1553	-	1.06	-	0.612	-
		Maelstrom 2-phase	1620	1.04	0.56	0.53	0.373	0.61
		chp2prs	5178	3.32	6.24	5.89	6.967	11.37
		Balsa	2527	1.62	1.94	1.83	1.188	1.94
Adder	$*[L_1?x_1,$ $L_2?x_2;$ $R!(x_2 + x_1)]$	Maelstrom	1215	-	2.42	-	0.346	-
		Maelstrom 2-phase	1236	1.02	1.26	0.51	0.233	0.67
		chp2prs	2022	1.66	6.38	2.64	2.516	7.28
		Balsa	2760	2.27	3.31	1.37	0.936	2.71
Multiplier	$*[L_1?x_1,$ $L_2?x_2;$ $R!(x_2 * x_1)]$	Maelstrom	2420	-	3.66	-	0.510	-
		Maelstrom 2-phase	2470	1.02	1.91	0.52	0.322	0.63
		chp2prs	3114	1.29	5.71	1.56	2.679	5.27
		Balsa	-	-	-	-	-	-
Split	$*[C?c; L?x;$ $[c = 0 \rightarrow R_1!x$ $[]c = 1 \rightarrow R_2!x$ $]]$	Maelstrom	684	-	1.45	-	0.276	-
		Maelstrom 2-phase	711	1.04	0.73	0.51	0.182	0.66
		chp2prs	2879	4.21	10.05	6.93	2.850	10.33
		Balsa	1316	1.93	2.33	1.61	0.392	1.42
Merge	$*[C?c;$ $[c = 0 \rightarrow L_1?x$ $[]c = 1 \rightarrow L_2?x];$ $R!x]$	Maelstrom	1124	-	1.63	-	0.300	-
		Maelstrom 2-phase	1193	1.06	1.13	0.69	0.151	0.50
		chp2prs	2981	2.56	15.14	9.29	2.360	7.87
		Balsa	2051	1.82	2.59	1.59	0.435	1.45
GCD	See Appendix II	Maelstrom	4930	-	17.22	-	9.83	-
		Maelstrom 2-phase	5072	1.03	8.93	0.52	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	3921	-	20.65	-	8.43	-
		Maelstrom 2-phase	4043	1.03	11.87	0.57	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	14275	-	27.23	-	18.50	-
		Maelstrom 2-phase	14543	1.02	15.65	0.57	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		**1.64**		**1.99**		**2.38**
	Balsa	vs. Maelstrom 2-phase		**1.58**		**3.84**		**4.30**
	Maelstrom	vs. Maelstrom 2-phase		**0.96**		**1.93**		**1.81**

Fig. 6. Synthesis results for CHP programs, using Maelstrom and chp2prs, and the Balsa synthesis system for equivalent Balsa programs. All metrics are from SPICE simulations in the TSMC 65nm technology node. Energy is reported on a per-cycle basis. Balsa cannot synthesize complex arithmetic blocks such as multipliers. Averages reported are geometric means, ignoring the multiplier.

VII. Other Circuit Families

So far, we have built our synthesis strategy upon the fundamental C-element micropipeline. But 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 is valid. In Figs. 4 and 5, we show the 3-action sequencers for the MOUSETRAP [16] and GasP [17] circuit families. We omit the parallelizers and selection splits and merges in the interest of space, but it is easy to see how these can be constructed, following the same reasoning as for the QDI family. These control circuits have some timing constraints that need to be obeyed in order for correct operation, unlike the QDI control circuits.

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 be derived from the Gasp FIFO control circuit, via the same method as described in section 2. 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.

VIII. Evaluation

Maelstrom, an implementation of the aforementioned ring-based synthesis was used to automatically synthesize low- and medium-complexity CHP programs, in a 65nm technology node. In order to generate optimized combinational logic for Boolean expressions needed to implement computation in the datapath, we use the ABC logic synthesis system [18]. For comparison, we synthesize equivalent Balsa programs to Verilog using the Balsa synthesis system [7], and generate a SPICE netlist from the generated Verilog output. We also compare against `chp2prs`, an existing naıve syntax-directed translation method for CHP to circuits.

We report cycle time 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; we use 100 copies to avoid artifacts in the case of small benchmarks. The results of our comparison are summarized in Fig. 6, and we use 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 1.64× in area, 1.99× in cycle time, and 2.38× 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 highly complex programs sometimes displayed incorrect functionality, and these issues could not be resolved due to Balsa no longer being an active project.

Finally, it is important to note that the circuits using this method are practically on-par with hand-designed dataflow circuits, for small designs where it is practical to do so. Thus far, it has been easier to generate circuits that implement a given program correctly than to generate ones that also show performance similar to ones that are manually optimized. Our work presents a qualitative advancement in the logic synthesis methodology for asynchronous circuits.

IX. Discussion

The key idea presented here is the fact that the control element in an asynchronous pipeline actually sequences three actions, but it typically appears to only sequence two - the incoming and outgoing synchronization since the third is always an implicit communication with the datapath. However, recognizing that this too is an action enables the insertion of a communication on that port, which results in the PAA element, which is the keystone for constructing a basis set that enables a sequential synthesis of programs. Extending this logic to derive the other control flow elements follows naturally.

Further, using single-token rings to implement programs ties in well with the intuitive way of stepping through a written program one action at a time mentally in order to understand it, with a pointer that indicates the action currently in progress, and following the program graph from start to end.

Another interesting observation is that ring-based synthesis also has the added benefit of enabling data actions on either one phase or both phases of the handshake. Throughout this paper, we have used positive-edge triggered pulse generators in order to latch data. However, since the first half-phase of handshakes propagates through the entire ring, followed by the reset half-phase wave, this phase can also be used to perform another iteration of the loop. The only changes that are needed are making the pulse generator trigger on both the positive and the negative edge, replacing the N-input OR-gates in the selection blocks with N-input XOR-gates, and replacing the N 2-input AND-gates in the C_L block with N 2-input XNOR-gates. The delay line that matches the datapath logic would also have to be symmetric (same propagation for a zero-to-one and one-to-zero transition), as opposed to the case where only one phase is used for data, where the optimal delay line would propagate a one-to-zero transition extremely quickly.

X. Conclusion

In this work, we presented a general method to synthesize behavioral descriptions into asynchronous circuits that is agnostic to the underlying circuit family, and a tool that automatically compiles CHP into these circuits. The method starts from known 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 method 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 I

Communicating Hardware Processes (CHP) is a hardware description language used to describe clockless circuits derived from C.A.R. Hoare's Communicating Sequential Processes (CSP) [19]. A full description of CHP and its semantics can be found in [5]. Below is an informal description of a subset of that notation that we use, listed in descending precedence, replicated from [20]. For a complete discussion of the interaction between the handshake expansions of channel actions like send and receive and the composition operators, see [21].

A **Channel** X consists of a **request** X.r and either an **acknowledge** X.a or **enable** X.e. The acknowledge and enable serve the same purpose, but have inverted sense. With these signals, a channel implements a network protocol to transmit data from one QDI process to another.

- **Skip**: *skip* does nothing and continues to the next command.
- **Assignment**: $x := e$ sets the variable x to the value e, where e is an expression.
- **Send**: $C!e$ sends the value of e over the channel C.
- **Receive**: $C?x$ receives a value over channel C and stores it in the variable x.
- **Sequential Composition**: $S; T$ executes the programs S followed by T.
- **Parallel Composition**: $S \,||\, T$ executes the programs S and T in any order.
- **Deterministic Selection**: $[G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n]$ where G_i, called a guard, is a dataless expression and S_i is a program. The selection waits until one of the guards, G_i, evaluates to *true*, then executes the corresponding program, S_i. The guards must be stable and mutually exclusive. The notation $[G]$ is shorthand for $[G \rightarrow skip]$, which corresponds to waiting for G to become true.
- **Non-Deterministic Selection**: $[|\, G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n \,|]$ is the same as Deterministic Selection except that the guards do not have to be stable or mutually exclusive. If two or more evaluate to *true* simultaneously, then one is picked arbitrarily (not necessarily random). In a circuit, this choice is implemented by a collection of arbiters and synchronizers. When two or more guards evaluate to *true* simultaneously, it can cause a metastable state in the arbiter or synchronizer. 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.
- **Repetition**: $*[G_1 \rightarrow S_1 [] ... [] G_n \rightarrow S_n]$ is similar to the selection statements. However, the action is repeated until no guard evaluates to *true*. $*[S]$ is shorthand for $[true \rightarrow S]$.

APPENDIX II

The CHP for the iterative GCD computation program is as follows:

$$
\begin{array}{l}
*[X?x;\, Y?y; \\
\quad *[x > y \;\longrightarrow\; x := x - y \\
\quad\; [] y > x \;\longrightarrow\; y := y - x \\
\quad\;];\;\; O!x \\
]
\end{array}
$$

However, prior to synthesis, internal loops are systematically flattened up using state variables, and the resulting program that is used as the input to the synthesis engine is:

$$
\begin{array}{l}
c := 0; \\
*[[c = 0 \longrightarrow X?x;\, Y?y;\;\; c := 1 \\
\quad [] c = 1 \longrightarrow [x > y \longrightarrow x := x - y \\
\qquad\qquad\qquad [] y > x \longrightarrow y := y - x \\
\qquad\qquad\qquad [] else \longrightarrow O!x;\;\; c := 0 \\
\qquad\qquad\qquad] \\
\quad] \\
]
\end{array}
$$

The CHP for the 2n-th Fibonacci number generation program is as follows:

$$
\begin{array}{l}
*[N?n, x := 0, y := 1; \\
\quad *[n > 0 \longrightarrow x := x + y; y := x + y; n := n - 1]; \\
\quad O!x \\
]
\end{array}
$$

As before, this is flattened, using a state variable, into the following form prior to synthesis:

$$
\begin{array}{l}
c := 0; \\
*[[c = 0 \longrightarrow N?n, x := 0, y := 1; c := 1 \\
\quad [] c = 1 \longrightarrow [n > 0 \longrightarrow x := x + y; y := x + y; \\
\qquad\qquad\qquad\qquad\qquad n := n - 1 \\
\qquad\qquad [] else \longrightarrow O!x; c := 0 \\
\qquad\qquad] \\
\quad] \\
]
\end{array}
$$

The CHP for the Bresenham's Line Drawing Algorithm is as follows (assumes $x_0 \leq x_1$):

$$
\begin{array}{l}
*[X_0?x_0, X_1?x_1,\, Y_0?y_0,\, Y_1?y_1; \\
\quad dy := y_1 - y_0, dx := x_1 - x_0; \\
\quad D := dy + dy - dx; \\
\quad *[x_0 \leq \;\; x_1 \longrightarrow P_x!x_0, P_y!y_0; \\
\qquad\qquad [D > 0 \longrightarrow y_0 := y_0 + 1, \\
\qquad\qquad\qquad\qquad D := D - dx - dx \\
\qquad\qquad [] else \longrightarrow skip \\
\qquad\qquad]; \\
\qquad\qquad D := D + dy + dy, x_0 := x_0 + 1 \\
\quad] \\
]
\end{array}
$$

Note that multiplications by constants are explicitly written as repeated additions/subtractions for compatibility with Balsa. The flattened version used for synthesis is:

$$
\begin{array}{l}
c := 0; \\
*[\\
\quad [c = 0 \longrightarrow X_0?x_0, X_1?x_1,\, Y_0?y_0,\, Y_1?y_1; \\
\qquad\qquad dy := y_1 - y_0, dx := x_1 - x_0; \\
\qquad\qquad D := dy + dy - dx; \\
\qquad\qquad c := 1 \\
\quad [] (c = 1 \;\wedge \\
\qquad x_0 \leq x_1) \longrightarrow P_x!x_0, P_y!y_0; \\
\qquad\qquad [D > 0 \longrightarrow y_0 := y_0 + 1, \\
\qquad\qquad\qquad\qquad D := D - dx - dx \\
\qquad\qquad [] else \longrightarrow \;\; skip \\
\qquad\qquad]; \\
\qquad\qquad D := D + dy + dy, x_0 := x_0 + 1 \\
\quad [] (c = 1 \wedge x_0 > x_1) \longrightarrow c := 0 \\
\quad] \\
]
\end{array}
$$

REFERENCES

[1] "Ieee standard for verilog hardware description language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–590, 2006.

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

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

[4] R. Manohar and Y. Moses, "The eventual c-element theorem for delay-insensitive asynchronous circuits," in *2017 23rd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 102–109, IEEE, 2017.

[5] A. J. Martin, "Synthesis of asynchronous vlsi circuits," 1991.

[6] 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.

[7] A. Bardsley and D. Edwards, "The balsa asynchronous circuit synthesis system," in *Forum on Design Languages*, vol. 224, 2000.

[8] Handshake Solutions Inc., https://web.archive.org/web/20090323054030/http://www.handshakesolutions.com/.

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

[10] 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.

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

[12] 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.

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

[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] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, *Compilers Principles, Techniques & Tools*. pearson Education, 2007.

[16] 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.

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

[18] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, pp. 24–40, Springer, 2010.

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

[20] N. Bingham and R. Manohar, "A systematic approach for arbitration expressions," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 12, pp. 4960–4969, 2020.

[21] 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.