Asynchronous Design Compiler Optimization

Zeb Mehring, *EENG 471—Advanced Special Projects*

***Abstract*—Asynchronous designs are usually described at the high-level by Communicating Hardware Processes (CHP) programs. The first step in translating a piece of CHP into a physical layout involves manually compiling it into a set of stable, non-interfering production rules. This project presents an initial design for a CHP compiler, which generates a file in the ACT format that can be translated into production rules using `aflat`.**

I. Overview

Currently, asynchronous design relies on the expertise of CHP programmers to describe chips that minimize power usage and maximize computational speed, among other optimizations. While this strategy has produced many successful designs, human effort cannot hope to scale perfectly to systems with billions of components. Furthermore, manual CHP optimization requires an unnecessary level of specialization and attentiveness on behalf of the designer. Just as a modern programmer does not need to master the manipulation of a computer's instruction set thanks to advances in compiler technology, it seems unreasonable to demand that an asynchronous chip designer worry about making manual optimizations for each piece of circuitry when designing a processor.

The purpose of this project was to implement a basic but efficient CHP compiler which optimizes for common performance markers: speed, power use, etc. The result is a program capable of analyzing a piece of CHP written by a designer and compiling it into an equivalent description in production rules, making a few modifications along the way to improve on speed and power use in particular.

Fully realized, a robust and efficient CHP compiler will steepen the shallow learning curve associated with current asynchronous design, dramatically lowering the barrier to entry established by the inaccessibility of tedious design principles. Eliminating the relevance of arcane details about the various steps and frameworks involved in the design process will open up the field to curious individuals who do not wish to spend years mastering the minutiae of layout milieu. Rather than attempting to debug interference or stability issues in `prsim`, a future designer should be able to obtain a layout that "just works" from a correct CHP program.

Furthermore, an efficient CHP compiler will greatly reduce the time spent by extant designers in optimizing the implementation of their processors, as an automated algorithm tends to be much more efficient (not to mention much more successful) at repetitive tasks (like signal rewiring) than a human. The consequence of this benefit is that more time can be devoted to improving the functionality of the design, which will in turn facilitate more complex asynchronous processors. Ultimately, this type of software hopes to help bridge the popularity gap between synchronous and asynchronous design, allowing the latter more opportunity to realize success in areas where the former has failed.

This project demonstrates that automated production-rules generation and optimization is a feasible and fruitful realm of exploration for asynchronous designers looking for tools to improve their work. There still remain an incredible number of improvements beyond those that have been accomplished here, and the construction of a robust, reliable CHP compiler is only in its infancy. Aside from the clear steps forward outlined in later sections, more rigorous simulation and analysis will be needed to determine what further optimizations to implement and which choices to make when expanding on this tool in the future.

II. Previous Work

This project began with the creation of a proof-of-concept: an extremely bare syntax-directed translation (SDT) compiler to generate ACT "translations" of simple CHP programs. It only supports boolean variables, a few boolean expressions ($\neg$, $\vee$, and $\wedge$), two-way (if-else) selections, and a single guard in loops. The structure of this mini-project served as a starting point for the backbone for the subsequent, fully-fledged compiler.

The general idea is as follows: the SDT-inspired control flow crawls down a program's parse tree, recursively instantiating ACT modules and connecting the go and output signals appropriately. Because of the simplicity of the base cases for expressions (1-bit variables and integer primitives), this process can be done quite elegantly (with the correct labeling scheme, it is possible to reduce the requirements for stored parameters to just two integers). Furthermore, the base cases for CHP constructs like assignments, sends, and receives also

require only a few lines of ACT to instantiate, since no multi-bit expressions are involved. This means that complex structures like completion trees are not required.

Note that while data processing for most applications cannot be done with the limitations described above, control flow in a CHP program is often abstracted into a sequence of 1-bit channel operations. Thus, the control flow part of a fully-decomposed CHP program could be reasonably compiled with a tuned-up version of this program, though modifications would be required to provide basic functionality (e.g. $>$2-guard selections), fix bugs, and optimize the result.

III. Basic Features

In order to allow for efficient data processing, several "basic functionality" features were added to the bare-bones compiler developed in the previous stage. The features include: a type-checker, multi-bit variable support, boolean expression support (this includes both an expansion on the variety of single-bit operations available and the addition of their multi-bit counterparts), arithmetic expression support for all operations but division, comparison operation support, probe support, and support for n-way selections and n loop guards.

A. Correctness-Checker

The CHP parser for the ACT framework, along with files defining custom data structures for storing CHP programs, were provided at the start of this project. Unfortunately, parsability alone is not enough to guarantee that a CHP program is valid—it does nothing beyond ensuring the program's syntactical correctness.

To help identify programmer errors clearly, a simple type-checker was implemented to warn the user about mistakes that would lead to a failed compilation. Specifically, the type-checker ensures that:

1) All used variables and channels are declared.
2) Variables and channels are used correctly. Assignment can only involve variable expressions, and sends and receives may only occur over channels. Expressions that are sent, received, or assigned may only be variables.
3) Probes are used correctly. Probes of non-channel expressions are invalid.
4) Guards are boolean-valued.
5) Function blocks are declared correctly. They must be defined by the user and be of the form `<name>_<bitwidth>`.
6) All expressions have operands of compatible (equal) bit widths.

The final item is a design choice made both for simplicity and in anticipation of eventually abstracting away explicit bitwidth declaration from the programmer. In short, all expressions which "interact" with each other are expected to be of equal bitwidth. To put it more concretely, the operands of all arithmetic and boolean expressions must have equal bitwidth, and expressions sent/received over channels must have the same bitwidth as the channel.

Variables and channels are thus "typed" by their bitwidth, as in the C programming language (though this compiler does not yet support type casting, which would potentially make this design choice much more useful). Integer primitives have no inherent "type" (bitwidth), rather, they match that of their first typed parent expression. It is the programmer's responsibility to ensure that variables are large enough to store and do arithmetic with the integers used, or otherwise check for overflow.

Furthermore, the results of arithmetic operations are truncated to the bitwidth of their operands. An overflow bit is output by all relevant ACT modules included with this compiler, but it is unused in the actual compilation. It is the responsibility of the programmer to add explicit overflow/underflow checking for these operations if it is desired.

B. Bitwise Operations

The following bitwise operators have been created as ACT modules: AND (`&`), OR (`|`), XOR (`^`), NOT (not intantiatable in CHP), and NAND (not instantiatable in CHP). The latter two are used to implement other modules internally, while the first three can be explicitly generated by the compiler for a CHP program which contains their corresponding operator tokens. The delay-insensitive versions (which accept dualrail inputs and produce dualrail outputs) are located in the file `syn.act`, while the bundle-data versions (boolean inputs and outputs) are in `bundled.act`.

Each multi-bit operation is implemented as an array of its corresponding 1-bit gates, which have been given production-rule implementations in the aforementioned files (and can be instantiated directly with: `<syn/bundled>_expr_<op>`).

All modules are bit-parallel, so the time complexity of each is constant:

$$O_T(B) = O(1) \tag{1}$$

The space complexities are all equal and obviously scale in the number of bits (denoted here as n):

$$O_S(B) = O(n) \tag{2}$$

C. Arithmetic Operations

All arithmetic operations aside from division have been implemented as ACT modules. The delay-insensitive modules are included in the file `syn.act` while the bundle-data modules are found in `bundled.act`. Note that the latter depends on the former. The required modules are imported according to the flags specified to the compiler, though the paths generated by the compiler will need to be modified to match the file system structure of the host machine (see the README file). All modules assume a two's complement representation of values. Additionally, all arithmetic operations are templated by a single parameter N which dictates the bitwidth of the inputs and outputs.

For delay-insensitive implementations, the validity of the output can be assessed by a completion tree (a tree of C-elements), which accepts the boolean OR of each pair of dualrail outputs. Bundled implementations require timing assumptions to generate an acknowledge (see Section IV-A on this protocol).

1) Addition

Two different modules were created for addition: a simple ripple-carry adder (RCA) and a quicker (and more expensive) carry-lookahead adder (CLA). Both are based on full-adder blocks included in the ACT libraries, which are implemented in production rules. The operator token for addition is $+$.

The compiler-generated addition module is a simple wrapper for one of these template implementations, for programmer convenience. The wrapper automatically sets the carry in bit to zero, as is desired when computing something like $a + b$ in a CHP program. A programmer who wishes to explicitly manage the carry bits can instead instantiate one of the adder modules described above. Due to unresolved instability issues with modules utilizing the CLA (other than pure addition), the RCA is used as the basic addition module for this compiler.

Both the time and space complexities of the RCA are:

$$O_T(A_{RCA}) = O_S(A_{RCA}) = O(n) \tag{3}$$

where n is the bitwidth of the operands (and thus the result). Comparatively, the generate-propagate tree has time complexity only $\log n$, and because the carry chain is the bottleneck for the computation, the complexity of the CLA module is:

$$O_T(A_{CLA}) = O(\log n) \tag{4}$$

The required number of gates for the chain increases by a logarithmic factor for this speedup, though, giving a space complexity of:

$$O_S(A_{CLA}) = O(n \log n) \tag{5}$$

For the purposes of further discussion, we will use $O(A) \equiv O(A_{RCA})$, since this was the module used in practice.

2) Subtraction

Subtraction has been implemented internally with an addition module. The result is computed by adding the result of a bitwise NOT ($\neg$) on the second input to the first input, with a carry-in bit. This is equivalent to adding the first operand to the result of a unary minus (see below) applied to the second operand. The operator token for subtraction is $-$.

As mentioned above, this module especially relies on a two's complement representation of values. Applications for other schemes will need to revise the logic in the ACT module to compute the negative of the second input correctly, or to implement some other strategy entirely.

The time complexity is:

$$O_T(S) = O(O_T(A)) \tag{6}$$

where n is the bitwidth of the operands and $O(A)$ is the time complexity of the addition module, since the bitwise NOT adds only a constant amount of work to the process. The space complexity is:

$$O_S(S) = O(n + O_S(A)) \tag{7}$$

due to the additional n-bit NOT operation.

3) Multiplication

The traditional "schoolbook multiplicaiton" algorithm was used to implement the multiplication module. An array of partial products is produced by computing the logical AND of each bit of the multiplicand with the multiplier and shifting the result appropriately. The partial products are then wired into a reduction module which adds all its inputs together and produces the output. The operator token for multiplication is $\star$.

The first implementation of the reduction module was a linear cascade of carry-save adders (CSAs). A CSA (as it is defined in this paper) operates by altering the usual convention of the full adder, viewing it as a black box which takes three inputs and produces two outputs. A CSA is simply a collection of n full adders which accept three n-bit numbers bitwise into their input ports and produce two n-bit numbers from their output ports

which, when added, are the sum of the original inputs. CSAs are also known as 3:2 compressors. To add m partial products, then, one can add the first three with a CSA, add the outputs of the first CSA and the fourth product with another CSA, and so on. Once the outputs of the $m - 3^{\text{rd}}$ CSA and the m^{th} partial product have been compressed, the results from the (final) $m - 2^{\text{nd}}$ CSA can be added with a true addition module to produce the result of the multiplication. This is the linear CSA cascade described above.

This initial implementation was revised into a Wallace tree of 4:2 compressors (implemented as linear cascade of two CSAs), which dramatically reduces the time required for the results to propagate. From a graph theory perspective, a linear cascade is simply a horribly imbalanced binary tree which has height linear in the number of inputs. The Wallace tree is a balanced structure which recursively instantiates compressors so that the height is logarithmic in the number of inputs.

The time complexity for multiplication with the Wallace tree is:

$$O_T(M) = O(\log n + O_T(A)) \tag{8}$$

where n is the bitwidth of the operands and $O_T(A)$ is the time complexity of the addition module. The partial products array can be computed in constant time (each involves a parallel sequence of independent operations), and each of the compressors in the tree can compute a result in constant time (each full adder is independent in a CSA), which means that the only factors which play a role in the multiplier's asymptotic runtime are: the height of the tree and the time for the final addition. The space complexity goes like:

$$O_S(M) = O(n^2 + 2n \log n + O_S(A)) \tag{9}$$

where the non-dominant $2n \log n$ term has been included because, in practice, operand bitwidths are small enough for it to be significant. The quadratic term comes from the $n \times n$ bits required to store the partial products, the linearithmic term from the $2 \log n$ CSAs in the Wallace tree, and the adder module complexity from the obvious place.

4) Unary Minus

As described in the subtraction section, the unary minus module is implemented by taking the bitwise complement of the input and adding the result to zero with a high carry in bit.

The time complexity is:

$$O_T(U) = O(O_T(A)) \tag{10}$$

since the bitwise complement can be taken in parallel across bits. The space complexity is:

$$O_S(U) = O(n + O_S(A)) \tag{11}$$

to account for the additional n NOT gates required.

D. Comparison Operations

All comparison operators ($=$, $\neq$, $<$, $>$, $\leq$, $\geq$) have been implemented as ACT modules. For the most part, they leverage the already-defined arithmetic modules (namely, subtraction) and the assumption of two's complement notation that the MSB of any value gives its sign.

The equality module simply verifies that both inputs are bitwise equal (and valid, for delay-insensitive inputs). The inequality operation is just the negation of an equality module. The time complexity of both of these modules is constant:

$$O_T(E) = O_T(\overline{E}) = O(1) \tag{12}$$

while the space complexity is a function of the inputs' bitwidth:

$$O_S(E) = O_S(\overline{E}) = O(n) \tag{13}$$

The less-than module takes the difference of the two inputs using the subtraction module and outputs the MSB of the result. The greater-than-or-equals module is just the negation of a less-than module. The time complexity of both is the same as the subtraction module:

$$O_T(LT) = O_T(GE) = O(O_T(S)) \tag{14}$$

and the space complexity is likewise:

$$O_S(LT) = O_S(GE) = O(O_S(S)) \tag{15}$$

The less-than-or-equals module is, shockingly, a logical AND of the outputs of a less-than and an equals module. The greater-than module is just the negation of a less-than-or-equals module. Their time complexities are thus:

$$O_T(GT) = O_T(LE) = O(O_T(LT) + O_T(E)) \tag{16}$$

and in space:

$$O_S(GT) = O_S(LE) = O(O_S(LT) + O_S(E)) \tag{17}$$

E. Probes

Probes for both single-bit and multi-bit data channels have been implemented *for delay-insensitive compilations only*. Each accepts as input a dualrail data channel with a boolean acknowledge signal. For the single-bit channel case, the probe module produces a high output

if either of the input rails are set but the acknowledge bit is not set. The multi-bit channel probe simply checks that *all* the channel's data bits are set and its acknowledge signal is low. It is implemented as an array of single-bit channel probes on each of the channel's bits, where the acknowledge signal for each single-bit probe is the common acknowledge for the channel.

The time complexity of an n bit probe is constant, since the constant boolean logic on each bit can be done in parallel across all bits:

$$O_T(P) = O(1) \tag{18}$$

The space complexity scales with the bitwidth of the channel:

$$O_S(P) = O(n) \tag{19}$$

F. n-way Guards

The support for n-way guards was provided courtesy of Rajit Manohar and was edited to allow for the continous evaluation of selection statements in the case that all the guards are false. It is accomplished via a simple sequential chaining of guard statements during compilation. Note that this strategy can implement both deterministic and non-deterministic selection statements. As guards are evaluated, the false rail of the $i-1^{\text{st}}$ guard is connected to the "go" signal of the i^{th} guard. The 0^{th} "false rail" is simply the "go" signal for the loop/selection statement (technically, it is a bit more complicated to allow for guard re-evaluation). The acknowledge bit for the entire statement, in the case of selection statements, is just a composite OR of all the component guards' true rails. For loops, the acknowledge is high if the request signal is high *and* all the guards' false rails are high (the latter is a negation of the composite OR produced for selection statements, by one of DeMorgan's Laws).

G. `chp2prs`

In order to avoid the double-step process of first compiling CHP to ACT, and then ACT to production rules, a shell script has been provided to accomplish both with one command. It accepts the same command line arguments as the compiler (with exception of the `-o` flag, which has been hardcoded into the script for development purposes, see the final paragraph of this section), but also will take an arbitrary number of CHP files to compile. It first invokes the CHP compiler with the specified options, then calls `aflat` to compile the result into production rules. The results are not printed to the standard output, but rather, redirected to output files.

Note that for this script to work correctly on another filesystem, several parameters will have to be modified. First are the constant paths defined at the top of the script itself, which define the locations of the required ACT libraries, the CHP compiler, and the `aflat` binary. Secondly, the arguments following the `-o` flag given to the compiler (also in `chp2prs.sh`) will need to be changed to the desired output directory. Thirdly, the redirected output of `aflat` might need to be altered to reflect the directory structure of the output destination. Finally, the path defined at the top of `~/chp/src/cartographer.c` will need to be modified to correspond to the path to the ACT libraries. See the README file for full installation instructions.

IV. Optimization Features

A. Bundled Data

Bundled data is a data-handling protocol which attempts to reduce power consumption by replacing dualrail logic with pure combinatorial (boolean) logic. It comes at the cost of introducing timing assumptions to portions of the circuit and likely increasing the speed of the computation (though well-crafted bundled blocks will minimize this second parameter). Without a dualrail encoding of values, the status of an operation can no longer be determined in the standard manner (a completion tree on the outputs). Because of this, a bundled logic block is not delay-insensitive—outputs may change at any time because of fluctuating inputs and propagation delays, and there is no clear way to determine whether or not the output is stable without introducing further assumptions about the block's operation.

Instead, a bundled logic block accepts a "go" signal, indicating that the input(s) are stable, and produces a stable result after an established delay. Fundamental arithmetic and boolean operations must have known, reliable delays in order for this to work. The delays of more complex circuits can be constructed from these basic assumptions.

By "bundling" the logic in this manner, the entire block behaves very much like a delay-insensitive module with a request-acknowledge control channel, and can be integrated into an asynchronous circuit effectively despite the internal timing assumptions. The only further modifications that must be considered are that: the request for each logic block must verify that the inputs are stable and the output for a logic block must not be used until the acknowledge bit is generated by the delay. The rest is just a wiring problem.

Analogs for many of the modules in `syn.act` which

operate according to this protocol have been implemented in `bundled.act`. One notable omission is probes, which currently only have a delay-insensitive implementation. Delay lines are implemented as a serial chain of inverters, which worked well for simulations in `prsim`. More complex delay schemes and revised delay lengths will need to be computed for production. Furthermore, delays are not inherent to the bundled modules, but rather, are generated by the compiler. The delay for an arbitrary tree of bundled logic is computed by traversing upward from the output and accumulating the maximum delay of each level in the tree. The result is the delay of the critical path, which is used as the delay for the entire block. Generous values were used for the delay of each module in order to avoid simulation errors (additionally, since the use of this delay model is unlikely to carry over to production, further investigation was not warranted). The request channel for the logic block is connected to the composite delay line's input, while the acknowledge channel triggers a receive into a latched dualrail value.

B. Carry-Lookahead Adder

The critical path of a ripple-carry adder is the carry-chain, which creates a linear time dependence in the bitwidth of the operands. A carry-lookahead adder reduces this to a logarithmic dependence by adding gates to compute the carry bits separately (and more quickly).

Consider an addition module which computes the sum of its n-bit inputs A and B. The i^{th} full adder in this module is said to "generate" a carry bit if both of its operand bits are high ($g_i = a_i \wedge b_i$). It is said to "propagate" a carry bit if either of its operand bits are high ($p_i = a_i \vee v_i$). Utilizing this terminology, the i^{th} full adder in the module will have a carry in bit if: the $i - 1^{\text{st}}$ full adder either generates, *or* the j^{th} (where $j < i$) full adder generates and *all* the full adders k such that $j < k < i$ propagate. It is clear that $G = A \wedge B$ and $P = A \vee B$. The i^{th} carry bit is thus:

$$c_i = \left[\bigvee_{j=0}^{i-1} \left(g_j \wedge \bigwedge_{k=j+1}^{i-1} p_k \right) \right] \tag{20}$$

where the bounds are inclusive, a 0-based indexing scheme is used, and the empty "summation" yields its respective identity element (0 for $\vee$, 1 for $\wedge$).

In a CLA, the carry bits are computed according to Equation 20. Note that the n-bit values G and P can be computed in constant time, so that the carry in to the highest order bit can be computed in time logarithmic in the bitwidth of the operands. This ensures that the entire operation can be done in $O(\log n)$ time.

A CLA has been implemented in `syn.act` and `bundled.act`. It accepts two n-bit numbers where $n > 2$, in order to avoid errors in compiling the base cases (`aflat` exhibits intriguing behavior when production rules loops have non-positive bounds, so the carry in to the first and second bit positions must be instantiated manually). As mentioned, the module functions as intended for simple addition, but seems to introduce complications when used as a piece of other modules (like subtraction and multiplication). Further investigations into this behavior are warranted.

C. Common Sub-Expression Elimination

Consider the following CHP program:

```
...
(x := y + z , ... , s := z + y) ;
...
t := y + z ;
...
```

Listing 1. A redundant CHP program.

A naive syntax-directed translation based-program (like the compiler provided) would simply generate the required addition modules, connect the inputs and outputs, and move on. A clever designer, however, would realize that the repeated evaluation of $y+z$ is potentially wasteful. So long as the code in the intervening ellipses does not contain any assignment to y or z, only one addition module is needed. Ideally, this would reduce power consumption and area (fewer circuits needed) and also potentially deliver a performance boost (consider the assignment to t, which would no longer be dependent on the delay of an addition module). This optimization is known as common sub-expression elimination (CSE) since, in general, the redundant computations may be sub-expressions of more complicated code.

A basic form of CSE has been implemented for this project. Note that due to the way the compiler latches values and generates "go" signals for sequential statements, in the program above, only the assignment to s would be a candidate for this kind of optimization (if this weren't the case, the result would be deadlock). The compiler's implementation of this is as follows.

If the optimization is enabled, the compiler creates a hash table which it populates with computed sub-expressions. Before any expression is generated, the compiler consults the table, which stores keys of the form `<op>(<op1>[,<op2>])` associated with the values `<res>`, where `op` is the operator associated with the expression, `op1` (and `op2`) are its operand(s), and `res` is the node containing the result at the port `<res>.out`. If the operation has already been com-

puted (i.e. it is in the table), the compiler simply uses <res>.out as the output of the sub-expression. If not, it stores the expression (and its commutative counterpart, for boolean operators, equality/inequality, addition, and multiplication) in the form above. When an assignment to or a receive into a variable is made, the entries to which it is a part (if any) and all recursive dependences are cleared from the hash table. Additionally, the hash table is cleared between sequential expressions, though this should be eliminated when a method for forcing expression re-evaluation is implemented.

Note that CSE has not undergone the same level of testing as the other features of this compiler, and may still have glaring bugs in its implementation. Furthermore, it is not even necessarily clear that such an optimization will be of benefit. The increased power draw on the common expressions' output ports may more than make up for the savings of utilizing only one circuit. The increased demands might even make the operation slower. This feature was partly introduced to investigate these questions, and it should be fleshed out to do so more rigorously.

V. Testing

Beyond those used for correctness-verification (see the directory ∼/chp/tst/), several CHP programs were created for benchmarking the compiler. The results from running the generated production rules in prsim allow for a comparison of the speed and power use between the various compilations, based on the options provided to the compiler.

Specifically, speed (as measured by simulation cycles) and energy use (as proxied by transition count) for delay-insensitive and bundle-data circuits were measured using deterministic timings in prsim. Programs comparing these features among both arithmetic operations (addition, subtraction, and multiplication) and CHP constructs (communication actions along with selection statements/loops) were written. The same variables were used to analyze the carry-lookahead adder and basic CSE, in delay-insensitive compilations.

All of the arithmetic benchmark programs are of the same form. They begin with the sequential assignment of two pseudorandomly generated integers to two variables, followed by the assignment of the result of the operation performed on the two variables to a third variable. All operations are sequential, and the specific integer values are the same for addition and subtraction (the integers generated for multiplication were chosen from a much smaller range, to prevent overflow). Programs were generated for both 8-bit and 32-bit integers.

Three programs were used for benchmarking CHP constructs. The first is a simple for loop which computes 2^{10} by repeatedly multiplying a stored variable by two, whilst incrementing an iteration variable by 1. At loop termination, the result is assigned to another variable. The second program serves to benchmark simple communication over a channel. A pseudorandomly generated value is first assigned to a local variable, sequentially followed by a concurrent send/receive over a shared channel. The received value is stored into another local variable. The final program tests selection statements: it assigns two local variables to two pseudorandomly generated values, then manipulates a third variable based on comparisons between the two. All variables in the CHP-construct benchmark programs are 32 bits.

All data were gathered using basic commands in prsim. Simulations were run identically, using the commands specified in Listing 2. In words, the simulation was first initialized and the circuit was reset. The reset signal was then turned off, and finally, the top-level "go" signal was activated.

```
$ initialize
$ set Reset 1
$ set _Reset 0
$ set t.go.r 0
$ cycle
...
$ set Reset 0
$ set _Reset 1
$ cycle
...
$ set t.go.r 1
$ cycle
...
```

Listing 2. prsim simulation control commands.

Cycle count was measured by the simulation cycle of the last transition in the program. Since initialization was done identically across all benchmarking programs, the relative cycle times can be used as a meaningful measure of circuit speed. Energy use was approximated by the number of production rules fired, as given by the dumptc command. Though this does not account for heat or leakage currents, it can be used to extrapolate a rough lower bound on the power requirements of the generated production rules.

VI. Results

A. Arithmetic Operations

As expected, arithmetic operations were much faster for delay-insensitive than for bundled operations (no thanks to the generous delays implemented by the compiler). And though addition and subtraction did not

favor the bundled data protocol in energy use either, more complex operations like multiplication (especially for larger bitwidths) revealed a massive reduction in transition count over their delay-insensitive counterparts. The full results for the basic arithmetic benchmarks are in Table I.

TABLE I
PERFORMANCE COMPARISON FOR ARITHMETIC BENCHMARK PROGRAMS

Program	Protocol	Cycles	Transitions
add_8.chp	delay-insensitive	1,360	4,792
add_8.chp	bundled	3,880	5,789
add_32.chp	delay-insensitive	2,020	19,836
add_32.chp	bundled	13,960	22,704
sub_8.chp	delay-insensitive	1,400	6,152
sub_8.chp	bundled	4,520	6,621
sub_32.chp	delay-insensitive	2,060	25,228
sub_32.chp	bundled	16,520	26,385
mul_8.chp	delay-insensitive	1,820	20,324
mul_8.chp	bundled	8,360	12,198
mul_32.chp	delay-insensitive	2,710	226,148
mul_32.chp	bundled	93,320	115,880

Of note among these is the case of 32-bit multiplication. While the bundled data protocol reduced transition count by about half (226,148 to 115,880), the cycle time was over 34 times greater. Compare this to 8-bit multiplication, which only saw an approximate four-fold increase in time delay. The massive discrepancy is a result of using an n^2 delay wire for the multiplication module, which can likely be refined with more tuning and then even further with a more efficient multiplication algorithm.

The addition and subtraction modules saw two to four-fold increases in delay for the bundled data protocol, and slight increases in transition count as well. This is likely due to the brevity of the implemented CHP programs, and the fact that operations like assignment are more expensive (in terms of power) using bundled data (extra inverters are used to convert the data between boolean and dualrail encodings). More complex programs which involve these operations should see the transition count between the two protocols diverge to favor bundled data.

B. CHP Constructs

The CHP construct-based benchmarking programs showed similar results. While computation times were vastly increased when using bundled data, transition counts were on par or fewer. The full results can be found in Table II.

In a complex program like the loop benchmark, transition count was reduced by almost four times using bundled data (the 44-fold increase in computation time is, again, due to the generous delay of the multiplication module). The simple communication actions in comm.chp did not display any benefit for bundled data, with a likely explanation being the same as that postulated for the addition/subtraction benchmarks. Likewise, the lack of significant change in transition count between the two protocols in the selection statement benchmark is probably due to the plethora of expensive assignment operations and only very simple data operations (comparisons, which are essentially subtractions). It is hypothesized that more complex CHP programs which leverage these constructs will demonstrate an improvement in power use with bundled data.

TABLE II
PERFORMANCE COMPARISON FOR CHP CONSTRUCT BENCHMARK PROGRAMS

Program	Protocol	Cycles	Transitions
for_loop.chp	delay-insensitive	22,100	1,103,663
for_loop.chp	bundled	976,620	320,478
comm.chp	delay-insensitive	940	10,446
comm.chp	bundled	8,040	13,379
comp.chp	delay-insensitive	8,190	126,270
comp.chp	bundled	52,790	138,430

C. Addition Modules

Comparisons between the delay-insensitive ripple-carry adder and carry-lookahead adder were also made. As expected, the CLA was faster, and more so with larger operands. While the speed improvement for 8-bit integers was meager, a 25% decrease was observed for 32-bit integers using the CLA. Transition count only scaled slightly with the CLA implementation, with only a 15% increase in the 32-bit case. The specific figures are found in Table III.

TABLE III
PERFORMANCE COMPARISON FOR ADDITION MODULES

Program	Addition Module	Cycles	Transitions
add_8.chp	RCA	1,360	4,792
add_8.chp	CLA	1,290	5,544
add_32.chp	RCA	2,020	19,836
add_32.chp	CLA	1,470	22,892

D. CSE

Thanks to the requirement of parallel expression evaluation, the optimized (CSE-enabled) implementation revealed no difference in speed compared to the ordinary parallel version. Transition count was observed to decrease, however, by about 10%. As mentioned in Section IV-C, this would have to be more rigorously analyzed with a better metric to verify the actual reduction. Regardless, the optimization seemed to do what was intended. The exact results are in Table IV.

TABLE IV
PERFORMANCE COMPARISON FOR REDUNDANT COMPUTATIONS

Program	Optimization	Cycles	Transitions
`cse.chp`	none	2,020	29,612
`cse.chp`	`-O1`	2,020	26,264

VII. KNOWN ISSUES

A. Bundled Multiplication

The multiplication of certain 32-bit numbers with the bundled multiplication module still results in floating nodes after evaluation (see `mul_32.chp` for an example). Though glitches in combinatorial logic are expected to occur during evaluation (unless the designer explicitly designs glitch-free logic), these discrepancies should resolve after the computation is complete and all signals have settled. The cause of this is thought to be in the implementation of the bundled full-adder, though various sets of production rules have been tried unsuccessfully (for a summary, see the commented-out portions of this module in `bundled.act`). 32-bit multiplication does not generate floating nodes in `for_loop.chp`, so multiplication by two may be a special case. Furthermore, correct answers may still be produced despite the errors. Regardless, they should be investigated and rectified as a next step.

B. Carry-Lookahead Adder

The delay-insensitive CLA implementation currently causes instabilities when it is used in any module other than basic addition, resulting in instability errors and floating nodes. The source of this issue is currently unknown, but it is thought to be caused by varying signals in the propagate-generate tree exposing unstable production rules in the full adder module. Various implementations of the full adder module have been attempted, to no avail. The CLA does not seem to generate errors for programs which only do pure addition (though this should be verified). As with the bundled multiplication errors, fixing these should be given priority in future work.

VIII. FUTURE WORK

A. Channels

There is no current mechanism to allow for repeated use of a channel. When such a program is compiled and simulated, interference errors result because multiple latches become connected to the channel and attempt to drive it simultaneously. A clean way to latch values and reset channels should be implemented to make this compiler more robust (and to support loops which involve sends and receives).

B. Other Operators

The lack of division support is a notable omission from this compiler. Many algorithms exist to implement it on the hardware level, and all that remains is to implement one in the ACT libraries provided (though this is admittedly much easier to claim than to do—the algorithms are a bit hairy). Modular arithmetic would supposedly be an added bonus with this feature, since an integer division module will almost always compute a remainder.

A few other operator tokens remain as well. Of note is the logical shift left operator, which has been tentatively implemented in `syn.act` as a 2D array of 2:1 1-bit multiplexors. The left-hand input (the number to be shifted) is fed into the leftmost column of the array, while the right-hand input (the shift amount) is given bitwise as the select bit to each column. Thus each column is responsible for shifting its input by a power of two according to its position and select bit (e.g. column 3 shifts the input 2^3 bits left if the select bit is high, or not at all if the select bit is low). The current version causes instabilities and glitches, though, and needs to be debugged before use. Logical and arithmetic right shift operators can be implemented in a similar fashion once the left shift bugs are fixed.

Also in the "todo" section is the bullet operator, which ensures concurrent completion of both of its operand statements. This CHP feature is not widely used, but may be desired for some applications, and would be a worthwhile addition in order to make the compiler comprehensive.

Finally, probes could be expanded on and improved, especially since their current implementation is quite tentative. Probes for different types of channels could be investigated, along with ensuring the correctness of the current implementation for `a[N]1of2` channels.

C. Optimizations

Endless optimizations remain to be added to the compiler, and they are arguably where this tool will demonstrate its true advantage over manual compilation. Though an automated compiler is certainly nice to have, its true potential will be realized in laying out circuits which are much faster and more efficient than those created by human designers. Next steps in this regard come in two different flavors: ACT optimization and

compiler optimization.

The underlying ACT libraries provided are not perfect and should be improved upon. Examples of work to do here would be: fully replacing the ripple-carry adder with a carry-lookahead adder, replacing the schoolbook multiplier with a Karatsuba multiplier, eliminating perceived inefficiencies in basic modules (full adders, multiplexors, etc.), and more. Improvements at this level, while largely invisible to the compiler written for this project, could realize massive gains in performance and power efficiency.

There are also optimizations to be run at compile time. Deadcode elimination, more refined CSE (especially expanding it to work between sequential statements), parallel analysis of guards in selection statements, and avoidance of unnecessary latching/receiving are all possible avenues for improvement. It goes without saying that the literature on traditional compiler optimization should be consulted here, to benefit from the decades of knowledge already codified about similar challenges. Not only will this allow for the adoption of already-proven solutions, but it will preemptively facilitate the finding of areas ripe for optimization in the first place. These kinds changes will perhaps require more thought and analysis than simple "module upgrades," and at the very least will involve some mucking about in C, but in the end may prove to be just as fruitful (if not more so).

Finally, a more robust system of analysis should be established for introducing new optimizations. While relative cycle times and transition counts were an easy metric for the purposes of this project, more rigorous investigations into the effects of various design choices should be conducted. Sets of production rules generated by a compiler should be analyzed by physical metrics which actually impact applications. A robust and effective method of analyzing the compiler's results will save not only silicon, but perhaps more importantly, time.

IX. Conclusion

The initial design for a CHP compiler has been presented and analyzed. While many features remain to be implemented (or debugged), this project proves that such a compiler is feasible and capable of vastly improving the asynchronous design process. Furthermore, it will allow for the generation and assessment of multiple implementations of the same CHP program, tweakable with the press of a button depending on the desired constraints (in this case, speed and power efficiency were examined, but other variables are theoretically possible to account for). With future development inspired by existing research into compiler optimization for conventional programming languages, a fully vetted and robust CHP compiler is a realistic and tempting goal. It is hoped that its eventual creation will herald as an important milestone in asynchronous design and allow for advances thus far impossible thanks to human limitations.