# MINIMALIST: An Environment for the Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines

**Columbia University Computer Science Dept.**
**Tech Report #CUCS-020-99**

Robert M. Fuhrer* Steven M. Nowick* Michael Theobald*
Niraj K. Jha† Bill Lin‡ Luis Plana*

*Dept. of Computer Sci.
Columbia University
New York, NY 10027

†Dept. of Electrical Eng.
Princeton University
Princeton, NJ 08540

‡Dept. of Elec. & Comp. Eng.
Univ. of Calif. at San Diego
La Jolla, CA 92093

July 26, 1999

**Abstract**

MINIMALIST is a new extensible environment for the synthesis and verification of burst-mode asynchronous finite-state machines. MINIMALIST embodies a complete technology-independent synthesis path, with state-of-the-art exact and heuristic asynchronous synthesis algorithms, e.g. optimal state assignment (CHASM), two-level hazard-free logic minimization (HFMIN, ESPRESSO-HF, and IMPYMIN), and synthesis-for-testability. Unlike other asynchronous synthesis packages, MINIMALIST also offers many options: literal vs. product optimization, single- vs. multi-output logic minimization, using vs. not using fed-back outputs as state variables, and exploring varied code lengths during state assignment, thus allowing the designer to explore trade-offs and select the implementation style which best suits the application. MINIMALIST benchmark results demonstrate its ability to produce implementations with an average of 34% and up to 48% less area, and an average of 11% and up to 37% better performance, than the best existing package [38]. Our synthesis-for-testability method guarantees 100% testability under both stuck-at and robust path delay fault models, requiring little or no overhead. MINIMALIST also features both command-line and graphic user interfaces, and supports extension via well-defined interfaces for adding new tools. As such, it is easily augmented to form a complete path to technology-dependent logic.

## 1 Introduction

While asynchronous circuits have undergone a renaissance driven by significant renewed interest in the last decade, their promises — reduced power, increased performance, and robustness — have only begun to be fully realized [38][6][21][16][29][15][18][31][19]. Although several of these methods have been effective, several synthesis steps still lack optimal solutions or practical tools. Likewise, a lack of well-integrated and extensible environments within which to embed these tools leaves designers without a smooth synthesis

*This research was funded in part by NSF CAREER Award MIP-9501880, NSF RIA grant MIP-9308810, NSF grant CCR-97-34803, by a supplement to NSF MIP-9501880 for CAD software tool development and by an Alfred P. Sloan Research Fellowship.

path. By contrast, the synchronous community possesses a wealth of such tools and environments, both commercial and academic [10], which benefit both researchers and end-users.

Thus, MINIMALIST makes contributions on several fronts:

- An integrated synthesis path consisting of state-of-the-art asynchronous synthesis algorithms:
  - CHASM, the first general optimal state encoding tool for asynchronous machines, providing both exact and fixed-length modes, and which can produce exactly-minimum output logic, a key parameter in asynchronous system performance
  - HFMIN, the only exact hazard-free *symbolic* two-level logic minimizer, supporting both single- and multi-output implementations
  - IMPYMIN, a new *implicit* exact hazard-free two-level logic minimizer, capable of solving all available benchmark problems in under 15 minutes, including several previously unsolvable problems
  - ESPRESSO-HF, a very fast new heuristic hazard-free two-level logic minimizer, which typically produces optimal or near-optimal results in under 3 seconds
  - Synthesis for testability, yielding 100%-testable multi-level implementations under either stuck-at or robust path delay fault models, with little or no area overhead
- In contrast to existing synthesis paths, MINIMALIST provides a single synthesis path able to produce implementations in a variety of styles (e.g., single-output vs. multi-output, using vs. not using feedback outputs as state variables, exploring various state code lengths) under various cost functions, allowing the exploration of design trade-offs
- The first complete and practical technology-independent synthesis path for burst-mode circuits using fast optimal algorithms
- An easily-usable environment with a software framework which can readily incorporate new tools

MINIMALIST currently supports widely-used plain *burst-mode* [22][32] specifications. Extended burst-mode specifications [42] will be supported in a forthcoming release.

## 2 Background and Overview

### 2.1 Asynchronous Synthesis

Asynchronous controller synthesis follows a flow similar to that of synchronous synthesis; however, it presents unique problems requiring significantly different solution methods. Like synchronous synthesis, the synthesis trajectory is divided for tractability's sake into several steps: state minimization, state encoding, two-level logic minimization, multi-level transformation, synthesis for testability, and so on. Each of these steps can be modeled roughly after its synchronous counterpart, but poses additional complications. We now review each step, outlining the basic problems unique to asynchronous synthesis.

The task of *state minimization* is to find a minimum-cardinality closed state cover for the original burst-mode specification. The result is a reduced machine realizing the original specification [13]. As with synchronous machines, this problem can be solved by first forming a set of compatibles and then forming a binate covering problem expressing the two basic sets of constraints (coverage and closure) [12]. Asynchronous machines, however, require different forms of compatible in order to be assured of the existence of a hazard-free logic implementation [23].

*State encoding* produces a set of binary codes for the symbolic states of the reduced machine. For synchronous machines, all encodings which distinguish the states are valid; however, typically this is performed judiciously, so as to minimize logic area [9], improve performance, or reduce power consumption. By contrast, asynchronous machines must be encoded so as to avoid *critical races* [35]. Further, if optimal logic is to be obtained, *logic hazards* [36][24] must be taken into account [11].

Finally, to ensure correct operation, *two-level logic minimization* for burst-mode asynchronous machines must also take care to avoid logic hazards. Recent developments in this area include exact multi-valued-input (*i.e.,* mvi)/multi-output minimization [11], fast heuristic minimization [33], and exact implicit minimization [34].

An additional issue facing asynchronous synthesis is the potential for using fed-back outputs to reduce the number of state variables and the overall implementation complexity. In this *machine implementation style*, primary outputs are fed back as additional input variables, which help to identify the machine's present state, thereby reducing the need for distinct state variables. The loading on the outputs may be minimal (only a short path to a feedback buffer is added to its fan-out), but the savings in overall logic complexity can be dramatic. Care must be taken, however, in various synthesis steps, in order to ensure that the use of fed-back outputs does not introduce hazards or critical races.

## 2.2 Burst-Mode Specifications

MINIMALIST addresses the class of asynchronous controller specifications known as *burst-mode*, a generalization of the traditional multiple-input change (MIC) mode [36]. Burst-mode was first formalized by Nowick [22], who also developed UCLOCK, a systematic synthesis method for hazard-free implementations. This specification style is based on more ad-hoc methods used earlier by Davis et al.. [8].

Burst-mode machines allow multiple inputs to change concurrently, but, unlike MIC machines, in any order and at any time. This relaxation considerably reduces the timing constraints placed on the environment, but nonetheless allows economical and high-performance implementations. In particular, applying Nowick's method for exact two-level hazard-free logic minimization [25] yields low-area, high-performance circuits.

Burst-mode has been successfully used by both academia and commercial interests to design and implement a number of significant circuits, for example, at Stanford, UCSD, HP, AMD and Intel.

The specifications are most easily illustrated by example. A burst-mode specification for a distributed mutual-exclusion controller with 3 inputs and 3 outputs is shown in Figure 1. The unique starting state (`S0`) is indicated by a `'v'`, and initial input and output values are either explicitly specified or (as in the figure) default to `0`. Each arc is labelled with a set of input and output transitions, known as *bursts,* separated by a `'/'`. Rising transitions are denoted by a `'+'`; falling transitions, by a `'-'`.

The operation of a burst-mode machine is as follows. Starting in a given state, the machine remains stable in that state until a complete input burst arrives. Individual inputs within that burst may arrive in any order and at any time. Once the last input arrives, the burst is complete. The machine then generates the corresponding output burst, if any, and moves to the specified next state. The environment allows the machine to settle, and the next cycle begins.

Figure 1 illustrates burst-mode operation. For example, consider the transition from `S2` to `S0`, with corresponding input and output bursts `LIN-,RIN-` and `LOUT-`, respectively. As a result, when in `S2`, if the pair of input changes `LIN-` and `RIN-` arrive at any order, and within any time window, the machine responds with a falling edge on `LOUT` and a transition to `S0`.

Burst-mode specifications must obey two important restrictions. First, input bursts must not be empty; in the absence of input changes, the machine remains stable in its current state. Second, the so-called *maximal set property* stipulates that no arc leaving a given state may possess an input burst that is a subset of any other arc leaving that state. This property guarantees that, at all times, the machine can

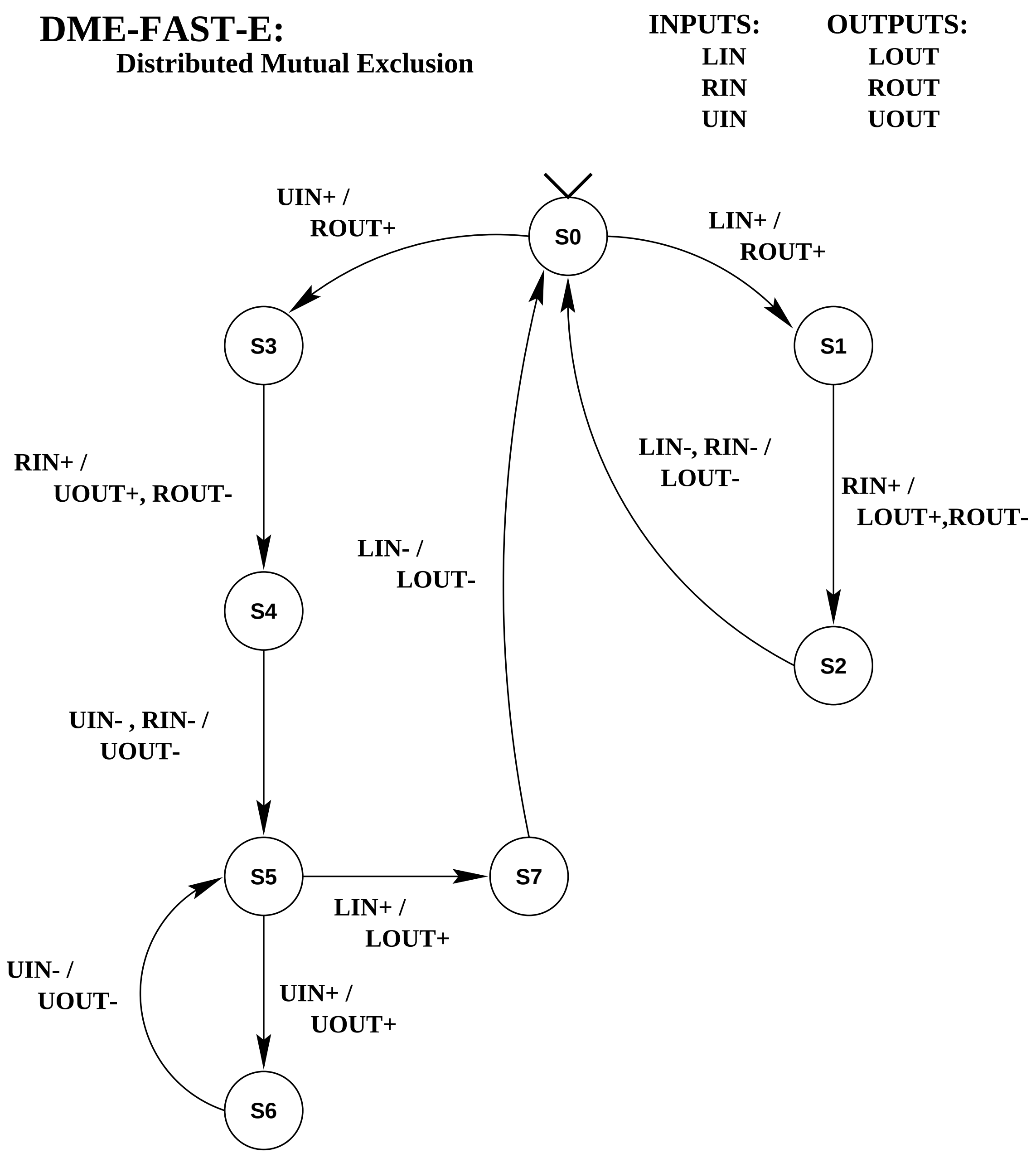

Figure 1: Burst-mode specification for a distributed mutual-exclusion controller

```
name DME_FAST_E

input LIN 0
input RIN 0
input UIN 0

output LOUT 0
output ROUT 0
output UOUT 0

0 1  LIN+ |  ROUT+
0 3  UIN+ |  ROUT+
1 2  RIN+ |  LOUT+ ROUT-
2 0  LIN- RIN- |  LOUT-
3 4  RIN+ |  UOUT+ ROUT-
4 5  UIN- RIN- |  UOUT-
5 6  UIN+ |  UOUT+
5 7  LIN+ |  LOUT+
6 5  UIN- |  UOUT-
7 0  LIN- |  LOUT-
```

Figure 2: Textual (**.bms**) burst-mode specification for **DME-FAST-E**

unambiguously decide whether to follow a transition or remain stable.

It is important to understand that, in a given burst-mode specification, *any unspecified input combinations are forbidden.* For example, the input burst `RIN+` in state `S0` in the specification of Figure 1 is prohibited. In other words, the surrounding circuitry must never generate that input combination. Any such combinations can thus be treated as don't-cares, and used to optimize the machine's implementation.

It is also conventional to adhere to a simple constraint, in order to ensure that the burst-mode specification can be properly synthesized[1]. Each state must have a *unique entry point*, i.e., a unique set of input and output values upon entry. In other words, each state has a *single* total state that is the destination of one or more transitions. Note that this property is not necessary for proper burst-mode operation. However, it tends to produce machines which are more easily minimized, and which are more easily proven to have hazard-free implementations.

An equivalent textual burst-mode specification (as used by MINIMALIST and MEAT [7]) appears in Figure 2, and an equivalent flow table in Figure 3.

It is easy to see from Figure 3 that burst-mode specifications frequently offer significant opportunity for state minimization. This due to the unique entry point criterion, which generally results in states which bind the output and next-state functions in only a few input columns.

## 2.3 Previous Work

We now briefly review two previous burst-mode asynchronous synthesis systems, and compare them to MINIMALIST.

The UCLOCK [23] system is a nearly complete path from plain burst-mode specifications to two-level logic. It incorporates a safe, exact state minimization algorithm, and the first exact hazard-free single-output logic minimization algorithm [24]. Unlike MINIMALIST, however, it offers no automated method for

[1]This is in fact a sufficient, but not necessary, constraint.

```
Inputs:  LIN, RIN, UIN;
Outputs: LOUT, ROUT, UOUT;
#Sn:    000    001    011    010    110    111    101    100
S0:  S0,000 S3,010  -,---  -,---  -,---  -,---  -,--- S1,010;
S1:   -,---  -,---  -,---  -,--- S2,100  -,---  -,--- S1,010;
S2:  S0,000  -,---  -,--- S2,100 S2,100  -,---  -,--- S2,100;
S3:   -,--- S3,010 S4,001  -,---  -,---  -,---  -,---  -,---;
S4:  S5,000 S4,001 S4,001 S4,001  -,---  -,---  -,---  -,---;
S5:  S5,000 S6,001  -,---  -,---  -,---  -,---  -,--- S7,100;
S6:  S5,000 S6,001  -,---  -,---  -,---  -,---  -,---  -,---;
S7:  S0,000  -,---  -,---  -,---  -,---  -,---  -,--- S7,100;
```

Figure 3: Asynchronous flow table for **DME-FAST-E**

state encoding or multi-output logic minimization. [2] Further, its Lisp implementation and slow algorithms for state minimization and logic minimization severely limit its usefulness. Finally, it does not allow fed-back outputs, missing an opportunity to significantly reduce implementation complexity.

The 3D system, presented in [38][40][39], also synthesizes two-level implementations, but accepts extended burst-mode specifications — a larger class of specifications than either UCLOCK or MINIMALIST (at present) handle. Unlike UCLOCK, 3D uses fed-back outputs; unlike MINIMALIST, their use is not an option: it is required. In contrast to MINIMALIST, it uses heuristic greedy state minimization and encoding algorithms. It also always performs exact single-output logic minimization (using HFMIN [11]), to produce reasonably high-performance implementations. Even so, none of its methods (save HFMIN) offers any guarantee of optimality; benchmarks show that MINIMALIST's algorithms give better results.

Finally, whereas both UCLOCK and 3D support only a single implementation style and one cost function, MINIMALIST supports *multiple* implementation styles and cost functions. MINIMALIST thus allows designers to explore various trade-offs and choose the implementation which best suits their application.

## 2.4 Comparative Overview: MINIMALIST vs. Previous Tools

The following table provides an overview of the choices available in the most important dimensions of the solution space for MINIMALIST and the two competing burst-mode synthesis toolkits, 3D and UCLOCK. Each dimension is correlated with the relevant operating mode or tool option, which will be defined later in this paper.

| synthesis pkg | fed-back outputs (`machine impl style`) | state-min | code length (`constr. sat. mode`) | type of logic (`logic impl. style`) | cost func |
|---|---|---|---|---|---|
| UCLOCK | non-fed-back only | exact | one solution | single-output only | products |
| 3D | fed-back only | heuristic | one solution | single-output only | literals† |
| MINIMALIST | both | both | many solutions (varied code length) | single-, multi-output, or output-disjoint | both |

† In the original 3D implementation, the sole cost function was product count.

[2] In practice, critical race-free codes for UCLOCK were produced either manually, or using auxiliary programs.

## 3 MINIMALIST Framework

The MINIMALIST framework consists of several key pieces: core data structures, a class and algorithm library, and an extensible interpreter, implemented in roughly 45,000 lines of C++ (including several of the core tools, e.g. HFMIN and CHASM).

The MINIMALIST framework incorporates a simple set of C++ classes to represent the original burst-mode specification. Early synthesis steps such as state minimization and state encoding simply transform or place annotations on these structures. As a result, additional steps or transformations are easily accommodated.

To assist in implementing new synthesis algorithms, MINIMALIST offers class libraries for manipulating both asynchronous burst-mode specifications, two-level logic (hazard- and non-hazard-free), dichotomies, unate and binate covering problem instances, arbitrary-length bitstrings, and the like. To facilitate interfacing to external programs, a small number of basic translators to common formats (e.g. Berkeley PLA or BLIF) is incorporated.

Finally, MINIMALIST provides a shell-like interpreter, extensible with commands written in C or C++. The interpreter supports user-defined shell functions, on-line help, command- and filename-completion, variables, control constructs (loops, conditionals), arithmetic, external process invocation, input/output redirection, and the like. Also, functionality can be augmented by dynamically-loaded code, without having to re-link the executable.

The result is a uniquely flexible, potent context for integrating synthesis tools, that we find lacking in existing packages [23][38].

## 4 MINIMALIST Tools

The following sections describe the set of synthesis, verification and testability tools currently incorporated into the MINIMALIST toolkit. Work is currently under way to integrate other existing high-quality tools such as technology mapping [14, 1, 2] and timing analysis [4], taking advantage of MINIMALIST's plug-in framework. With these additions, MINIMALIST will offer an even more capable path for asynchronous synthesis and verification.

### 4.1 State Minimization

MINIMALIST includes two new and very efficient algorithms for exact state minimization. In contrast, the 3D method uses a heuristic greedy minimization algorithm. Therefore, in this section, we will focus on a more direct comparison: the MINIMALIST exact algorithms and an earlier exact state minimization algorithm found in UCLOCK.

MINIMALIST improves significantly on UCLOCK's state minimization approach in two ways — (i) by allowing outputs to be fed back as inputs, and (ii) by dramatically reducing run-time complexity.

MINIMALIST offers two new exact state minimization algorithms: 1) for implementations *without* fed-back outputs (loosely based on UCLOCK's method), and 2) for implementations *with* fed-back outputs. The latter is the first exact algorithm for state minimization that handles fed-back outputs. Thus, it supports two **machine implementation styles**. As mentioned earlier, fed-back outputs can dramatically reduce the implementation's logic complexity.[3] In particular, their use allows merging certain states which would otherwise be incompatible. MINIMALIST makes use of this fact by relaxing UCLOCK's compatibility relation. In fact, several benchmark specifications collapse into a single state using the new relation, whereas UCLOCK's relation results in 2 or more states.

[3]See the benchmark results.

Second, MINIMALIST improves run-time by several orders of magnitude over the UCLOCK method. UCLOCK uses an expensive algorithm to generate maximal compatibles. In contrast, MINIMALIST uses a simple transformation which allows it to generate maximal compatibles using a fast unate prime generation algorithm instead. In addition, both UCLOCK and MINIMALIST (currently) approximate the binate covering step by a unate one followed by a closure check[4]. UCLOCK, however, employs Petrick's method to solve the unate problem, while MINIMALIST employs a state-of-the-art tool, MINCOV [27]. The combination of these two algorithmic enhancements reduce the run-time of state minimization by two or more orders of magnitude. To date we have encountered only one specification for which state minimization requires more than a few seconds, whereas run-times of many minutes were common for UCLOCK. For such large specifications, both of MINIMALIST's algorithms also feature an *approximate mode* which further reduces run-time.

## 4.2 CHASM

For state encoding, MINIMALIST uses CHASM (**C**oding for **H**azard-free **A**synchronous **S**tate **M**achines), the first exact method for input encoding of multiple-input change asynchronous machines. CHASM has many operating modes. One highlight is that its "exact mode" can be used to produce *exactly optimum two-level output logic*, over *all* critical race-free encodings, thus optimizing the key performance parameter for asynchronous networks (output latency). Its approximate mode also gives significant reductions in overall implementation cost.

CHASM, as reported in [11], loosely follows the flow of the KISS [9] method for *input encoding* of synchronous machines. Several significant modifications are required to handle asynchronous machines. There are three steps. First, *symbolic hazard-free logic minimization* is performed. Second, a set of *encoding constraints* is generated, which properly subsumes both the classic synchronous KISS ("face embedding") optimality constraints as well as asynchronous critical race-free [35] constraints. The constraints are in the form of generalized dichotomies [35] (not face embedding constraints). Finally, the *constraints are solved.* For constraint solution, CHASM has two modes: (i) an **exact mode**, which uses DICHOT [30], and (ii) an **approximate mode**, using NOVA's [37] simulated annealing engine. The approximate mode, as in NOVA, attempts to solve as many constraints as possible, under the restriction of a *fixed code length*; it has the advantage that it may reduce next-state logic complexity.

For MINIMALIST, CHASM has been extended in several new ways.

First, CHASM is now being applied to implementations with fed-back outputs. Specifically, we have proven that CHASM requires no modification to properly encode implementations with fed-back outputs. A modified functional specification is provided as input, which simply identifies the primary outputs as fed-back inputs. The symbolic two-level logic minimizer then forms a symbolic cover on this new function.

Second, CHASM can now target three **logic implementation styles**: (i) *multi-output* (where outputs and next-state may share products), (ii) *output-disjoint* (where products are shared among outputs, but not between outputs and next-state), and (iii) *single-output* (where *no* product terms are shared between any output functions). The motivation is that the "single-output style" is most suitable for performance-optimal designs: each output is individually optimized. Note that, in asynchronous machines (unlike synchronous), output latency is often the key parameter to overall system performance in a network of interacting machines. The "multi-output style" is most suitable for area-optimal designs, since it uses maximal sharing of logic. Finally, the "output-disjoint style" is a balanced compromise.

For modes (ii) and (iii), a novel feature of CHASM is that it produces *exactly optimum two-level output logic*, over all critical race-free encodings. This result holds, because the optimal output-only state encoding problem is a true "input encoding problem", unlike the general optimal state encoding problem (which is

[4]We have yet to see the closure check fail, due in part to the manner in which the set of state compatibles is refined to a partition.

an approximation).

Finally, CHASM targets two distinct **cost functions:** (i) *product cardinality*, and (ii) *literal count*, at the symbolic level. It does the latter by performing weighted unate covering during symbolic logic minimization. This technique is only a heuristic, however, because the literal count of the final binary cover can only be estimated. In practice, the heuristic nonetheless yields significant reduction in literal count.

## 4.3 HFMIN

After state minimization and encoding, MINIMALIST performs two-level hazard-free logic minimization. This step is normally performed using HFMIN, the first exact multi-output symbolic hazard-free two-level logic minimization tool.

HFMIN, as reported in [11], uses ESPRESSO to generate ordinary prime implicants, then refines them as needed to dynamic hazard-free (DHF) primes [24], and finally, performs a unate covering step using MINCOV [28], covering *required cubes*[24] in lieu of minterms. HFMIN's use of such highly-optimized algorithms for sub-steps allows it to readily handle most minimization problems we have encountered.

HFMIN has also been enhanced for MINIMALIST with the ability to produce output-disjoint[5] and single-output covers, and the ability to minimize literal count. Output-disjoint and single-output covers are formed by generating a suitable set of prime implicants before DHF refinement takes place. The rest of the algorithm proceeds unchanged.

To minimize literal count, HFMIN performs a weighted unate covering step where prime implicants are assigned weights according to their literal count. In addition, HFMIN now supports a limited post-processing step that further reduces literal count. This step is similar in spirit to the *make-sparse* operation of ESPRESSO [28]. A single pass is made over each selected prime implicant, removing output literals as long as the result remains a valid (hazard-free) cover. The input portion is then expanded, if possible. Unlike *make-sparse*, our current method makes no guarantee that the resulting product is maximally expanded. Nonetheless, despite its simplicity, the operation often yields significant reductions in literal count.

HFMIN is now widely used in several other burst-mode CAD packages, including the 3D method [38] and ACK [16]. It has also been used as part of the asynchronous tool suite at Intel Corporation, where it has been applied in the design of a high-speed experimental asynchronous Instruction-Length Decoder (see [5]).

## 4.4 ESPRESSO-HF

For very large problems which HFMIN is unable to solve in reasonable time, MINIMALIST offers ESPRESSO-HF [33], a new fast heuristic two-level logic minimizer. ESPRESSO-HF uses an algorithm loosely based on ESPRESSO (but substantially different from it), to solve problems with up to 32 inputs and 33 outputs. On benchmark examples, ESPRESSO-HF can find very good covers — at most 3% larger than a minimum-size cover — in less than 105 seconds. For typical examples, ESPRESSO-HF obtains an exact or near-exact result in under 3 seconds.

Currently, ESPRESSO-HF only implements multi-output minimization targetting product cardinality, and so it is normally used only for designs which exceed HFMIN's capacity. However, output-disjoint or single-output minimization can easily be implemented by a trivial modification of the code, and will be available in future releases of MINIMALIST.

## 4.5 IMPYMIN

IMPYMIN [34] is a new state-of-the-art fully-implicit exact two-level hazard-free logic minimizer. It greatly exceeds the capacity of previous exact tools (e.g. HFMIN), minimizing very large multi-output functions,

[5]products are shared among outputs, but not between outputs and next-state

including some for which no exact result had previously been obtained. Run-times are typically under 16 seconds. The most difficult problem available, with 32 inputs and 33 outputs, had never before been solved exactly, but required only 813 seconds using IMPYMIN.

Both ESPRESSO-HF and IMPYMIN can solve all currently-available benchmark examples, including several which have not been previously solved. For larger examples that can be solved by HFMIN, these two minimizers are typically several orders of magnitude faster.

### 4.6 Synthesis-for-Testability

MINIMALIST incorporates a recent method [26] for synthesis-for-testability targetting multi-level logic. The method produces circuits that are both hazard-free and 100% testable under either stuck-at or robust path delay fault models, with little or no overhead. First, it uses a novel two-level hazard-free logic minimization algorithm which minimizes the number of redundant cubes, as well as the number of non-prime cubes. (The tool currently operates only in single-output mode.) This helps maximize testability without using additional inputs. If not yet completely testable, the circuit is converted to a multi-level form which is completely testable (if possible). If still not completely testable (rarely the case), controllable inputs are added, yielding 100% testable logic. Finally, hazard- and testability-preserving multi-level transformations are used to reduce the area of the resulting circuit. The area overhead is typically zero, and in all cases is less than 10% [26].

### 4.7 Verifier

MINIMALIST features a simple tool to verify that a given logic implementation (produced by any method) realizes the specified burst-mode behaviour, independent of the particular state merges or encoding which have been performed. The verifier simulates the implementation's response to each specified input burst, and compares it to the specification; any mismatches of output or state at the end of the burst and any logic hazards are reported. Although each burst is considered only once, the analysis that is performed accounts for all possible interleavings of individual input changes. Since the verifier needs to traverse each edge in the specification's state graph only once, this tool is eminently practical even for very large specifications — the time required is never more than a few seconds. Currently, verification of only two-level AND/OR implementations is provided. However, the framework allows for verification of multi-level implementations as well (using the 9-valued algebra developed by Kung [17]), which will be completely supported in the near future. Also, the current version of the verifier does not detect output changes made after *partial* input bursts. Finally, being a purely combinational verifier, it does not verify the one-sided timing constraints also needed to ensure correct operation [3]. Such a capability may be added in a future release.

## 5 A Synthesis Session

```
$ MinShell                                                  # Start the MINIMALIST shell
minimalist> help                                                # Show list of commands
    assign-states          break           call            cd
         continue         define           echo          expr
              for           help             if  impymin-logic
    make-testable      min-logic     min-states           pwd
             quit      read-spec            set  set-encoding
  set-state-cover  show-encoding     show-logic        source
     verify-logic          while     write-flow write-instant
       write-spec write-symbolic    write-trans

minimalist> pwd                                                # Show current directory
/u/minimalist/demo
```

```
minimalist> ls                                                        # Run 'ls'
bin  ex   lib

minimalist> ls ex
dme-e        dram-ctrl    pe-send-ifc  scripts      stetson
dme-fast-e   hp-ir        pscsi        scsi-iccd92

minimalist> cd ex/dme-e                              # Move to another directory

minimalist> ls
dme-e.bms

minimalist> help read-spec                           # Show syntax of 'read-spec'
        read-spec [-v] <file> [<spec-var>]
Read the Burst-Mode specification in <file> and store it in <spec-var>,
or, if <spec-var> is not specified, 'theSpec'.

minimalist> read-spec dme-e.bms                # Read the Burst-Mode specification
Specification passed validity check.
Specification has 3 inputs, 3 outputs, and 8 states.

minimalist> plot_graph dme-e.bms
[... a window displays the burst-mode graph; press Ctrl-Q or the Quit button to dismiss it ...]

minimalist> min-states                              # Perform state minimization
*** Performing state minimization... ***
State cover: { { S0 S1 S4 }, { S2 S3 }, { S5 S6 S7 } }
Machine has 3 states after minimization.

minimalist> assign-states -F                   # Assign states with a CRF encoding
No encoding style specified; defaulting to critical race-free
*** Performing state assignment... ***
Invoking 'chasm' as 'chasm -C DME_E-F.func DME_E-F.trans'
*** Machine encoded by CHASM ***
State S0': 11
State S1': 10
State S2': 00

minimalist> min-logic -F -L                    # Produce the logic implementation
*** Performing logic minimization... ***
Invoking 'hfmin' as 'hfmin -P -C -l -S -o DME_E-FL.sol DME_E-FL.pla DME_E-FL.btrans'
*** Final PLA ***
# PLA file for machine DME_E-FL
.i 8
.o 5
.ilb LIN RIN UIN LOUT_i ROUT_i UOUT_i Y0 Y1
.ob y0 y1 LOUT ROUT UOUT
.type fr
.p 7
10-----1 00010
-01----1 00010
-0-----1 01000
10-----0 10100
00----1- 11000
--0---1- 10000
-01----0 00001
.e
Number of products:       7                         Total number of literals: 28
Number of products implementing outputs:      4  Literals in products implementing outputs: 17
Literals in products implementing next-state: 16  Average literals per output:         5.66667
Result stored in 'DME_E-FL.sol'.

minimalist> verify-logic -F -L                                    # Verify the implementation
Using logic implementation stored in 'DME_E-FL.sol'.
*** Starting Burst-Mode machine simulation. ***
*** Implementation verified successfully! ***

minimalist> plot_graph DME_E-FL.sol                              # Display the implementation
[... a window displays the 2-level logic; press Ctrl-Q or the Quit button to dismiss it ...]
                                                   # Run the above synthesis steps using a script

minimalist> source ../scripts/script-FBO.MOL-CRF dme-e.bms
[... same results as above, without interaction ...]

minimalist> quit
```

## 6 Results

This section compares synthesis results using MINIMALIST to those using the preeminent burst-mode asynchronous synthesis paths, namely 3D [38] and UCLOCK [23]. We highlight MINIMALIST's unique ability to support various cost metrics and implementation styles by showing several different experiments. For each, we indicate the cost function which we target, and the corresponding settings of MINIMALIST's modes.

### 6.1 Experimental Set-up

The benchmark suite consists of 23 burst-mode circuits, including several industrial designs, as well as a number of large asynchronous machines (e.g., see `sc-control` and `oscsi`). The circuits `it-control`, `rf-control`, `sc-control`, and `sd-control` are part of a low-power infrared controller designed at HP Labs as part of the Stetson project [20]. `pe-send-ifc` and `sbuf-xxx-ctl`, also from HP Labs, are part of a high-performance adaptive routing chip, used in the Mayfly parallel processing system [32]. Several others (e.g. the `scsi-xxx` and `pscsi-xxx` suites) come from a high-performance asynchronous SCSI controller designed by Yun while at AMD [41]. A DRAM controller circuit for Motorola 68K processors [38], `dram-ctrl`, completes the suite.

All MINIMALIST results are the best of a very small number of trials using fixed-length encodings. Generally, near-minimum code lengths are used. Here, minimum length refers to the smallest length sufficient for a critical-race free encoding, which is necessary to ensure correct operation. However, as demonstrated below, a trade-off exists, whereby significantly wider encodings sometimes offer better output logic at the expense of added next-state logic complexity. Hence, the results below occasionally make use of longer codes.

Run-times for the complete synthesis path are comparable for all tools (MINIMALIST, 3D, and UCLOCK), ranging from under 1 second to several minutes for the largest designs.

### 6.2 Performance-Oriented Comparison with 3D

The first experiment (shown in Table 1) shows synthesis results using both MINIMALIST and 3D for a **performance-oriented implementation**.

For asynchronous burst-mode machines, the metric that best approximates performance is *output latency*. In an asynchronous system, unlike synchronous, the input-to-output latency typically determines a machine's performance, as well as overall system performance. State changes are not bound to a clock period, and in practice are usually non-critical (see [20]).

```
define syn_FBO_so_lit { specFile codeLen } { # Single-output, optimize lit count
   read-spec $specFile
   min-states -F
   assign-states -F -s -L -O -l $codeLen
   min-logic -F -s -L
   verify-logic -F -s -L
}
call syn_FBO_so_lit dram-ctrl.bms $codeLen
```

Figure 4: Performance-oriented synthesis script for MINIMALIST

Based on the above observation, we now indicate the settings of the various modes of MINIMALIST for this experiment. In the context of technology-independent two-level logic, the *cost function* that most reasonably approximates output latency is the average number of literals per output. When comparing two results for the same machine (so that the number of outputs is fixed), this cost is equivalent to **total output literal count**, so this is used instead.

Roughly half of the MINIMALIST results in this set of runs make use of the **fed-back output machine implementation style**. Table 1 identifies the particular style chosen for each design in the column labelled 'FBO'. MINIMALIST is directed to use the **single-output** logic implementation style, and the **literal count** cost function. This combination of modes best minimizes average output literal count. This cost function also allows for a fair comparison to 3D, which produces single-output logic with minimal literal count. Finally, the encoding step uses **fixed-length constraint satisfaction** mode, attempted under several code lengths.

The MINIMALIST script in Figure 4 summarizes the selected modes. The script is parameterized by code length, using the variable `$codeLen`, and proceeds as follows. First, the specification is read from a file and checked for validity. Next, the machine is subjected to exact state minimization using fed-back outputs.[6] The states are encoded using CHASM, with the fed-back output ('`-F`'), single-output ('`-s`'), literal-count ('`-L`'), and fixed-length ('`-l`') flags. The final two-level logic is then synthesized using HFMIN, again passing the single-output and literal-count flags. Finally, the resulting logic is verified using the algorithm sketched in Section 4. The script was run in batch mode several times. The run having the lowest output literal count over 1-3 code lengths near the minimum is reported.

The 3D results were obtained using the 3D tool on the Unix platform. Unlike MINIMALIST, 3D embodies a hard-wired synthesis path, and produces a single deterministic result. Specifically, 3D first performs heuristic state minimization, followed by heuristic state encoding, and finally, exact two-level single-output logic synthesis using HFMIN, targetting total literal count. Thus, a single run for each design gives the reported (and the only possible) result.

Table 1 summarizes the comparison. MINIMALIST synthesis results demonstrate an average reduction of 11% in *output literals*, with the best being a 37% reduction for `sd-control`. In exchange for MINIMALIST's simplification in output logic, an increase in total literal count is occasionally observed (see for example `pe-send-ifc` and `pscsi-tsend`). However, MINIMALIST also frequently achieves a reduction in *total literal count* as well, with larger designs such as `stetson-p1` and `oscsi` among the most impressive gains (38% and 25%, respectively).

Clearly, MINIMALIST's gains come in part from its ability to explore wider encodings. In fact, in 12 of the 23 designs, the best result is seen at a longer code length than 3D uses. The greatest improvement overall, in `sd-control`, is seen at a *significantly* longer code length — 7 bits for MINIMALIST vs. 3 bits for 3D. For two designs, MINIMALIST chooses a shorter encoding than 3D: `dram-ctrl` (whose 0-bit encoding is enabled by the use of fed-back outputs), and `oscsi` (where 3D curiously uses 7 bits, despite no gain

[6]A nearly identical script exists in which all steps do not make use of fed-back outputs.

in output literals and a considerable increase in overall logic complexity). For the remaining examples, Minimalist achieved its best result at the same code length as 3D.

### 6.3 Area-Oriented Comparison with 3D

Our second experiment (also shown in Table 1) shows the results of an **area-oriented comparison** of Minimalist and 3D.

The cost metric that best approximates area for technology-independent two-level logic is total literal count; hence, *total literal count* is used in this comparison.

Based on the above observation, we now indicate the settings of the various modes of Minimalist for this experiment. The vast majority of the Minimalist results in this set of runs use the **fed-back output** machine implementation style. Again, the table identifies the particular style chosen for each design. Throughout, Minimalist is directed to use the **multi-output** logic implementation style, and the **literal count** cost function, which best minimizes total literal count. Finally, the encoding step uses **fixed-length constraint satisfaction** mode.

These runs were obtained using a script identical to that of Figure 4, but using the multi-output logic implementation style. In particular, the single-output (`'-s'`) flag was removed from the state encoding and logic minimization steps. Again, the cost function used was total literal count.

As shown in Table 1, Minimalist's term-sharing across outputs and next-state provides for significant reductions in total area. Minimalist's results for the area-targetted run show an average reduction of 33% in total literal count over 3D, the best being 48% for `sc-control`. For all designs, Minimalist achieved strictly better results than 3D. Although these runs did not target product count directly, they offer similarly dramatic reductions by that metric as well. An average of 42% improvement is observed, the best being 57% for `sc-control`. Again, Minimalist's results are strictly better than 3D in every case.

Unlike the performance-targetted runs, the code length used by Minimalist rarely exceeded that of 3D (only 3 times out of 23 designs), and never by more than 1 bit. In fact, Minimalist uses slightly *fewer* total state bits over the entire benchmark suite than does 3D, by roughly 5%.

### 6.4 Area-Oriented Comparison with UCLOCK

The final comparison, in Table 2, shows synthesis results for Uclock (as reported in [11]) and Minimalist.

For a fair and interesting comparison, we plugged some of the Minimalist tools into the Uclock path, to isolate and highlight two differences: (i) *machine implementation style* (choice of fed-back vs. no fed-back outputs), and (ii) *state minimization algorithms.* Even though Uclock does not use any optimal state assignment algorithms, we attached Chasm and Hfmin as a back end, to isolate these front-end differences. We also limited Minimalist to the only logic minimization modes that are available in Uclock: the cost function is **product cardinality**, and the logic implementation style is **multi-output**.

Table 2 shows the experimental results. In both Minimalist and the "improved" Uclock, reported results are the best of several fixed-length trials at or near the minimum code length. The majority of the Minimalist results use the **fed-back output** machine implementation style.

Not surprisingly, many Minimalist and Uclock results are nearly identical, since the operating modes are very similar. However, Minimalist's use of fed-back outputs results in significant gains in several cases (e.g., `dram-ctrl` and `scsi-isend-bm`). In addition, Minimalist obtains synthesis results in several cases where Uclock failed to complete, again due in part to Minimalist's more capable state minimization method.

A performance-oriented comparison to Uclock (like the above comparison to 3D) is possible, but is omitted, due to space considerations.

| | | 3D | | | | MINIMALIST | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | performance (single-output) | | | | | area (multi-output) | | | |
| **design** | i/s/o | codelen | prods | **lits** | **outlits** | FBO | codelen | prods | lits | **outlits** | FBO | codelen | prods | **lits** |
| dram-ctrl | 7/12/6 | 1 | 21 | 71 | 57 | ✓ | 0 | 17 | 57 | 57 | ✓ | 0 | 14 | 51 |
| pscsi-ircv | 4/6/3 | 2 | 13 | 44 | 30 | | 3 | 14 | 46 | 26* | | 3 | 10 | 38* |
| pscsi-trcv | 4/6/3 | 1 | 10 | 35 | 30 | | 2 | 13 | 43 | 27 | ✓ | 1 | 7 | 30 |
| pscsi-isend | 4/9/3 | 4 | 31 | 105 | 44 | | 6 | 31 | 111 | 38 | ✓ | 3 | 18 | 67 |
| pscsi-tsend | 4/10/3 | 4 | 22 | 77 | 41 | | 7 | 31 | 116 | 35 | ✓ | 3 | 18 | 66 |
| pscsi-trcv-bm | 4/7/4 | 2 | 18 | 58 | 45 | ✓ | 2 | 17 | 58 | 43 | ✓ | 2 | 11 | 45 |
| pscsi-tsend-bm | 4/10/4 | 3 | 24 | 86 | 43 | | 7 | 31 | 118 | 43 | ✓ | 3 | 16 | 63 |
| sbuf-read-ctl | 3/7/3 | 1 | 7 | 22 | 17 | | 2 | 9 | 26 | 15 | ✓ | 1 | 6 | 21 |
| sbuf-send-ctl | 3/8/3 | 2 | 17 | 51 | 32 | | 3 | 14 | 41 | 24 | | 2 | 11 | 36 |
| pe-send-ifc | 5/11/3 | 2 | 24 | 89 | 56 | | 5 | 32 | 136 | 55 | ✓ | 3 | 18 | 88 |
| scsi-isend-bm | 5/10/4 | 2 | 26 | 87 | 59 | ✓ | 2 | 23 | 76 | 56 | ✓ | 2 | 17 | 63 |
| scsi-isend-csm | 5/8/4 | 2 | 22 | 66 | 44 | | 2 | 18 | 53 | 41 | | 2 | 12 | 49 |
| scsi-trcv-bm | 5/10/4 | 2 | 24 | 78 | 50 | ✓ | 2 | 22 | 71 | 50 | ✓ | 2 | 17 | 64 |
| scsi-trcv-csm | 5/8/4 | 2 | 21 | 64 | 42 | | 2 | 18 | 52 | 40 | | 2 | 12 | 46 |
| scsi-tsend-bm | 5/11/4 | 2 | 28 | 92 | 67 | | 4 | 24 | 86 | 48 | | 3 | 19 | 75 |
| scsi-tsend-csm | 5/10/4 | 2 | 20 | 57 | 41 | | 6 | 24 | 75 | 34 | ✓ | 2 | 14 | 52 |
| it-control | 5/10/7 | 1 | 21 | 73 | 56 | ✓ | 1 | 19 | 68 | 56 | ✓ | 1 | 13 | 54 |
| rf-control | 6/12/5 | 2 | 12 | 44 | 34 | | 5 | 15 | 67 | 30 | ✓ | 2 | 11 | 43 |
| sc-control | 13/33/14 | 3 | 122 | 512 | 301 | ✓ | 3 | 93 | 359 | 275 | ✓ | 3 | 53 | 267 |
| sd-control | 8/27/12 | 4 | 50 | 217 | 155 | ✓ | 7 | 43 | 160 | 98 | ✓ | 4 | 28 | 125 |
| stetson-p1 | 13/33/14 | 3 | 87 | 371 | 209 | ✓ | 3 | 61 | 232 | 179 | ✓ | 3 | 45 | 199 |
| stetson-p2 | 8/25/12 | 4 | 46 | 194 | 148 | ✓ | 4 | 45 | 177 | 130 | ✓ | 4 | 30 | 140 |
| oscsi | 10/45/5 | 7 | 129 | 529 | 187 | ✓ | 4 | 89 | 395 | 185 | ✓ | 4 | 65 | 334 |
| **Total** | | 58 | 795 | 3022 | 1788 | | 82 | 703 | 2623 | 1585 | | 55 | 465 | 2016 |
| **Diff wrt 3D** | | - | - | - | - | | +41.4% | -11.6% | -13.2% | -11.4% | | -5.2% | -41.5% | -33.3% |

Table 1: A comparison of MINIMALIST and 3D synthesis results

| | | UCLOCK | | | MINIMALIST | | | |
|---|---|---|---|---|---|---|---|---|
| **design** | in/state/out | codelen | **prods** | lits | FBO | codelen | **prods** | lits |
| dram-ctrl | 7/12/6 | 2 | 22 | - | ✓ | 0 | 14 | 71 |
| pscsi-ircv | 4/6/3 | 2 | 9 | - | | 2 | 9 | 41 |
| pscsi-trcv | 4/6/3 | 1 | 10 | - | ✓ | 1 | 7 | 32 |
| pscsi-isend | 4/9/3 | 3 | 17 | - | ✓ | 3 | 17 | 80 |
| pscsi-tsend | 4/10/3 | 3 | 18 | - | ✓ | 3 | 16 | 86 |
| pscsi-trcv-bm | 4/7/4 | 2 | 12 | - | ✓ | 2 | 12 | 53 |
| pscsi-tsend-bm | 4/10/4 | † | † | - | ✓ | 3 | 16 | 84 |
| sbuf-read-ctl | 3/7/3 | 2 | 7 | - | ✓ | 1 | 6 | 23 |
| sbuf-send-ctl | 3/8/3 | 2 | 11 | - | | 2 | 11 | 47 |
| pe-send-ifc | 5/11/3 | 3 | 18 | - | ✓ | 3 | 18 | 118 |
| scsi-isend-bm | 5/10/4 | 2 | 21 | - | ✓ | 2 | 15 | 92 |
| scsi-isend-csm | 5/8/4 | 2 | 12 | - | ✓ | 2 | 12 | 62 |
| scsi-trcv-bm | 5/10/4 | 2 | 18 | - | | 2 | 18 | 99 |
| scsi-trcv-csm | 5/8/4 | 2 | 12 | - | | 2 | 12 | 61 |
| scsi-tsend-bm | 5/11/4 | 3 | 17 | - | | 3 | 17 | 101 |
| scsi-tsend-csm | 5/10/4 | 2 | 14 | - | ✓ | 2 | 13 | 77 |
| it-control | 5/10/7 | 3 | 15 | - | ✓ | 1 | 13 | 61 |
| rf-control | 6/12/5 | 3 | 13 | - | ✓ | 2 | 11 | 45 |
| sc-control | 13/33/14 | † | † | - | ✓ | 4 | 56 | 458 |
| sd-control | 8/27/12 | 5 | 29 | - | ✓ | 4 | 28 | 182 |
| stetson-p1 | 13/33/14 | 4 | 53 | - | ✓ | 3 | 42 | 317 |
| stetson-p2 | 8/25/12 | 4 | 31 | - | | 4 | 28 | 173 |
| oscsi | 10/45/5 | † | † | - | ✓ | 4 | 64 | 487 |
| **Total** | | 52 | 359=359+??? | - | | 55 | 461=325+136 | 200 |
| **Diff wrt UCLOCK** | | - | - | - | | +5.8% | -9.5% | - |

† Failed to complete within a reasonable time

Table 2: An area-oriented comparison of Minimalist and Uclock

| **design** | in/state/out | **codelen** | outprods | **outlits** | nsprods | **nslits** | totprods | **totlits** |
|---|---|---|---|---|---|---|---|---|
| scsi-tsend-csm | 5/10/4 | **3** | 14 | 37 | 9 | 24 | 23 | 61 |
| " | " | **4** | 14 | 36 | 7 | 26 | 21 | 62 |
| " | " | **5** | 13 | 35 | 10 | 34 | 23 | 69 |
| " | " | **6** | 13 | 34 | 11 | 41 | 24 | 75 |

Table 3: Effect of varying code length on synthesis results for a single design

### 6.5 Exploring Varying Code Lengths

This section briefly shows the effect on the synthesis results for a single design when varying code length in MINIMALIST.

Because a simple cost metric often fails to capture an application's cost completely, MINIMALIST better assists the designer in finding the point which best fits the application, by providing the opportunity to explore trade-offs. For example, Table 3 shows an interesting trade-off involving code length, arising from two competing tendencies. Output logic tends to improve, while next-state logic tends to grow, with increasing code length.

For these runs, a single design (`scsi-tsend-csm`) is synthesized using the performance-oriented script, but targetting the *fed-back output* machine implementation style. Code length is varied from 3 (the minimum needed to ensure a critical race-free encoding) to 6 (one less than the code length resulting when CHASM uses its exact, rather than its fixed-length, mode).

As the results show, the output logic complexity decreases as the code length increases, in exchange for a more expensive next-state implementation. This reflects the fact that CHASM'S input encoding model is exact for outputs, but is only approximate for next-state. Specifically, the fixed-length constraint satisfaction method favors neither output nor next-state constraints [11]. Thus, longer codes tend to satisfy a greater number of *both* kinds of constraints. So, output logic complexity decreases, because the corresponding constraints precisely model logic optimality. However, the next-state, whose constraints are less accurate, experiences an increase in logic complexity.

Without the ability to explore such trade-offs, a designer is forced to choose whatever single point in the solution space the synthesis path chooses. For example, suppose a synthesis path always chose to minimize output literal count. Given the results in Table 3, this would force the designer to accept an 8% decrease in output logic complexity, in exchange for a 71% increase in next-state logic complexity, which might be intolerable. MINIMALIST's ability to explore such trade-offs is unique among burst-mode synthesis toolkits.

## 7 Conclusion

MINIMALIST distinguishes itself in several respects. First, it integrates a suite of state-of-the-art algorithms for asynchronous burst-mode synthesis. Second, benchmark results demonstrate the effectiveness of the synthesis path on a large number of examples. Third, its support for multiple implementation styles and cost functions allows it to accommodate a variety of applications. In particular, the MINIMALIST tool chain provides well-optimized implementations using fed-back outputs which are guaranteed correct. Finally, its software framework provides both a potent end-user environment and the extensibility to allow it to encompass technology-dependent synthesis and other down-stream tasks. In short, MINIMALIST represents a first-of-a-kind environment for asynchronous synthesis, with significant contributions in algorithms, quality of results, and extensibility.

## References

[1] P. A. Beerel, K. Y. Yun, and W. C. Chou. Optimizing average-case delay in technology mapping of burst-mode circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[2] Peter A. Beerel, Wei chun Chou, and Kenneth Y. Yun. A heuristic covering technique for optimizing average-case delay in the technology mapping of asynchronous burst-mode circuits. In *Proc. European Design Automation Conference (EURO-DAC)*, September 1996.

[3] Supratik Chakraborty, David L. Dill, Kenneth Y. Yun, and Kun-Yung Chang. Timing analysis for extended burst-mode circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.

[4] Supratik Chakraborty, Kenneth Y. Yun, and David L. Dill. Practical timing analysis of asynchronous systems using time separation of events. In *Proc. IEEE Custom Integrated Circuits Conference*, May 1998.

[5] W.-C. Chou, P.A. Beerel, R. Ginosar, R. Kol, C.J. Myers, S. Rotem, K. Stevens, and K.Y. Yun. Average-case optimized technology mapping of one-hot domino circuits. In *Proc. Int. Symp. Adv. Research in Async. Ckts. and Sys.*, pages 80–91, March 1998.

[6] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Synthesizing petri nets from state-based models. In *ICCAD*, pages 164–171, 1995.

[7] A. Davis, B. Coates, and K. Stevens. Automatic synthesis of fast compact self-timed control circuits. In *1993 IFIP Working Conference on Asynchronous Design Methodologies, Manchester, England*, March 1993.

[8] A. L. Davis. A data-driven machine architecture suitable for VLSI implementation. In *Proceedings of the Caltech Conference on Very Large Scale Integration*, pages 479–494, January 1979.

[9] G. De Micheli, R.K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Trans. on CAD*, CAD-4(3):269–285, July 1985.

[10] E.M. Sentovich et al. Sequential circuit design using synthesis and optimization. In *Proc. Int. Conf. Computer Design*, October 1992.

[11] R.M. Fuhrer, B. Lin, and S.M. Nowick. Symbolic hazard-free minimization and encoding of asynchronous finite state machines. In *ICCAD*, 1995.

[12] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IEEE TEC*, EC-14:350–359, June 1965.

[13] J. Hartmanis and R.E. Stearns. *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, 1966.

[14] K. W. James and K. Y. Yun. Average-case optimized transistor-level technology mapping of extended burst-mode circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 70–79, 1998.

[15] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* Series in Parallel Computing. John Wiley & Sons, 1994.

[16] Prabhakar Kudva, Ganesh Gopalakrishnan, and Hans Jacobson. A technique for synthesizing distributed burst-mode circuits. In *DAC*, 1996.

[17] D.S. Kung. Hazard-non-increasing gate-level optimization algorithms. In *ICCAD*, 1992.

[18] Luciano Lavagno and Alberto Sangiovanni-Vincentelli. *Algorithms for Synthesis and Testing of Asynchronous Circuits.* Kluwer Academic Publishers, 1993.

[19] B. Lin and S. Devadas. Synthesis of hazard-free multi-level logic under multiple-input changes from binary decision diagrams. *IEEE Transactions on CAD*, 14(8):974–985, August 1995.

[20] A. Marshall, B. Coates, and P. Siegel. The design of an asynchronous communications chip. *Design and Test*, June 1994.

[21] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng. Automatic synthesis and verification of gate-level timed circuits. Technical Report CSL-TR-94-652, Stanford University, January 1995.

[22] S.M. Nowick. Automatic synthesis of burst-mode asynchronous controllers. Technical report, Stanford University, 1993. Ph.D. Thesis.

[23] S.M. Nowick and B. Coates. Uclock: Automated design of high-performance unclocked state machines. In *ICCD*, 1994.

[24] S.M. Nowick and D.L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. In *ICCAD*, 1992.

[25] S.M. Nowick and D.L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. *IEEE Transactions on CAD*, 14(8):986–997, August 1995.

[26] S.M. Nowick, N.K. Jha, and F. Cheng. Synthesis of asynchronous circuits for stuck-at and robust path delay fault testability. In *VLSI-Design-1995*, January 1995.

[27] R. Rudell. Logic synthesis for VLSI design. Technical Report UCB/ERL M89/49, Berkeley, 1989.

[28] R. Rudell and A. Sangiovanni-Vincentelli. Multiple valued minimization for PLA optimization. *IEEE Trans. on CAD*, CAD-6(5):727–750, Sept. 1987.

[29] J. Rutten and M. Berkelaar. Improved state assignments for burst mode finite state machines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, April 1997.

[30] A. Saldanha, T. Villa, R.K. Brayton, and A. Sangiovanni-Vincentelli. A framework for satisfying input and output encoding constraints. In *DAC*, 1991.

[31] M. Sawasaki, C. Ykman, and B. Lin. Externally hazard-free implementations of asynchronous control circuits. *IEEE Trans. on CAD*, CAD-16(6), August 1997.

[32] K.S. Stevens, S.V. Robison, and A.L. Davis. The post office - communication support for distributed ensemble architectures. In *Sixth International Conference on Distributed Computing Systems*, 1986.

[33] M. Theobald and S.M. Nowick. Espresso-hf: A heuristic hazard-free minimizer for two-level logic. In *DAC*, June 1996.

[34] M. Theobald and S.M. Nowick. An implicit method for hazard-free two-level logic minimization. In *Proc. Int. Symp. Adv. Research in Async. Ckts. and Sys.*, March 1998.

[35] J.H. Tracey. Internal state assignments for asynchronous sequential machines. *IEEE Trans. on Elec. Comp.*, EC-15:551–560, August 1966.

[36] S.H. Unger. *Asynchronous Sequential Switching Circuits.* New York: Wiley-Interscience, 1969.

[37] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State assignment of finite state machines for optimal two-level logic implementations. In *DAC*, pages 327–332, 1989.

[38] K. Yun, D. Dill, and S.M. Nowick. Synthesis of 3D asynchronous state machines. In *ICCD*, 1992.

[39] Kenneth Y. Yun. *Synthesis of Asynchronous Controllers for Heterogeneous Systems.* PhD thesis, Stanford University, August 1994.

[40] K.Y. Yun and D.L. Dill. Automatic synthesis of 3D asynchronous finite-state machines. In *ICCAD*, 1992.

[41] K.Y. Yun and D.L. Dill. A high-performance asynchronous scsi controller. In *ICCD*, 1995.

[42] K.Y. Yun, D.L. Dill, and S.M. Nowick. Practical generalizations of asynchronous state machines. In *EDAC*, 1993.