# Peephole Optimization of Asynchronous Macromodule Networks

Ganesh C. Gopalakrishnan*, Prabhakar N. Kudva, Erik L. Brunvand*

Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112

## Abstract

*Most high level synthesis tools for asynchronous circuits take descriptions in concurrent hardware description languages and generate networks of* macromodules *or* handshake components. *In this paper we describe a peephole optimizer for such macromodule networks that often effects area and/or time improvements. Our optimizer first deduces an equivalent black-box behavior for the given network of macromodules using Dill's trace-theoretic* parallel composition *operator. It then applies a new procedure called* Burst-mode reduction *to obtain* burst-mode machines, *which can be synthesized into gate networks using available tools. Since burst-mode reduction can be applied to any macromodule network that is delay-insensitive as well as deterministic, our optimizer covers a significant number of asynchronous circuits, especially those generated by asynchronous high level synthesis tools.*

## 1 Introduction

Asynchronous systems have been shown to exhibit a number of inherent advantages [1]. In order to facilitate the design of asynchronous circuits, several groups [2, 3, 4] have developed *high level synthesis tools* that translate concurrent program-like descriptions (semi-) automatically into a network of *macromodules* [5, 1, 6] or *handshake components* [7]. The macromodule networks generated by these tools often contain repeated occurrences of the same macromodule subnetwork (also observed in hand-designed macromodule networks). Our optimizer synthesizes custom replacements for these subnetworks.

The peephole optimization problem for macromodule networks has been studied by Brunvand [8] as well as van Berkel [9]. Both Brunvand and van Berkel propose optimization rules to translate macromodule networks into more area- and time-efficient macromodule networks that are *refinements* of the original networks in a formal sense [10, 9]. In this paper, we take the different approach of translating macromodule networks into *burst-mode machines*—a sub-class of multiple-input change machines. In recent years, numerous automatic synthesis tools to translate burst-mode machine descriptions into hazard-free gate-level networks have been developed by a number of researchers including Davis et. al. [11], Nowick et. al. [12, 13], and Yun et. al. [14]. We pick burst-mode machines as the target primarily based on their demonstrated high efficiency in large applications.

*Supported in part by NSF Award MIP-9215878

## 2 Illustration of Our Approach

In order for our peephole-optimizer to be applicable on a network, the joint behavior of the *unoptimized* network and its environment must obey the following restrictions. After power-up, the network must be *quiescent—i.e.*, it must not produce any output signal transitions without first consuming input signal transitions. Whenever quiescent, the network must first await a collection of one or more high-going or low-going input signal transitions (*i.e.*, an *input burst* [11, 12]) to arrive in some order. After absorbing the input-burst, the network should, in a finite amount of time, produce a collection of high-going or low-going output signal transitions (*i.e.*, an *output burst* [11, 12]) in some order. The production of all the transitions of an output-burst should be an indication that the network has attained quiescence. This mode of interaction between the network and its environment is called the *burst-mode* behavior, which is a special case of fundamental-mode [15] operation.

The optimizer first obtains the overall behavior of the macromodule subnetwork being optimized using the composition operator on trace structures [16]. *The behavior inferred in this fashion leaves out combinations of behaviors of the submodules that can never arise, or can lead to internal hazards.* The inferred behavior is converted into an Encoded Interface State Graph (EISG) [17]. EISGs are automata that label their state transitions with polarized signal transitions (*e.g.*, "a rising" (a+), "b falling" (b-), *etc.*). Finally, the optimizer converts the EISG into a burst-mode machine using our algorithm 'Burst-mode reduction' (detailed later), and synthesizes the resulting burst-mode machine using an already available tool (*e.g.*, see [13, 14, 11]). Any delay-insensitive and deterministic module $M_{DI}$ can be reduced (via burst-mode reduction) to a corresponding burst-mode machine $M_{BM}$ in such a way that the operation of $M_{DI}$ would be exactly the same as that of $M_{BM}$, provided the environment obeys the fundamental-mode timing

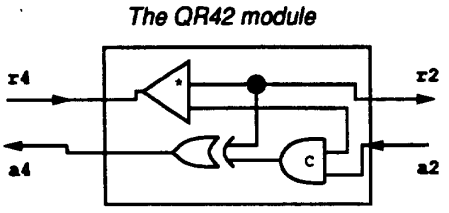


Figure 1: A four-to-two quick-return converter

constraint associated with the burst-mode behavior.

Consider the example shown in Figure 1. This subnetwork accepts a four-cycle handshake sequence [1] on r4 and a4 and generates a two-cycle handshake sequence [1] on r2 and a2, with the property that some of the events in these handshake sequences can overlap in order to provide a high degree of concurrency, as shown by the Petri-net in Figure 2. Assume that all interface signals are low to begin with. When r4+ occurs, the toggle element generates an r2+ as well as an a4+ (through the XOR gate). Transition a4+ is treated as the 'ack' by the four-cycle side which generates an a4- transition which, in turn, is forwarded by the toggle element to the upper input of the C-element as a rising signal transition. Meanwhile, r2+ is treated as a request by the two-cycle side which generates an a2+. This causes the C-element to receive rising transitions on both its inputs. Therefore, it generates an a4- through the XOR-gate. A similar sequence of steps now ensues during which the two-cycle interface returns to its initial state, and then the whole cycle repeats.

In optimizing QR42, we must first compose the behaviors of the three macromodules shown in Figure 1 along with the behaviors of "fictitious modules" that express the following constraints on its interfaces: the four-phase interface must witness a progression of events `r4; a4; ...`, and the two-phase interface must witnesses a progression of events `r2; a2; ...`. In addition, we must *hide* the internal signal that connects the lower output of the toggle element to the upper input of the C-element. The resulting black-box behavior for QR42 is shown by the Petri-net in Figure 2. Next, we must obtain the state graph (EISG) corresponding to this Petri-net. We can see that QR42 is initially quiescent, waiting for the singleton input-burst `r4`. After receiving `r4`, it has the option of producing `a4` or `r2`. If `a4` is produced first, QR42 is in a state where output `r2` as well as input `r4` are possible. If the environment were to follow the burst-mode behavior, however, it would first allow `r2` to be produced before supplying the inputs `r4` and `a2` concurrently as an input-burst. The important point to note is that even though a deterministic delay-insensitive module may possess a large number of behaviors, an environment that follows the burst-mode operating conditions invokes only a proper subset of these behaviors. Now we can perform *Burst-mode reduction*, which retains only the heavy arrows in Figure 2, and constructs the Burst-mode machine shown. We can ignore the dashed arrows because of the assumption of delay insensitivity (for reasons given later). Finally, we can synthesize the burst-mode machine (using Yun's tool [14], in our case) to obtain logic equations shown in the figure. Burst-mode machines are Mealy-style machines in which every transition is labeled with pairs "$I/O$" where $I$ is a non-empty set of polarized signal transitions called the *input burst*, and $O$ is the output burst. Contrary to the original definition [11], we require that $O$ be non-empty, which is consistent with our assumption of delay insensitivity of macromodules. The environment can also be given a burst-mode specification by *mirroring* [16]. The network should also be initially quiescent, and should attain quiescence after processing every successive input burst. Finally, the network must be DI.

Udding [18] has provided four necessary and sufficient conditions that characterize delay insensitivity. The ones relevant for this paper are now briefly outlined.

- If a module accepts (generates) two inputs (*i.e.*, input signal transitions) $a$ and $b$ in the order $ab$, it must also accept (generate) them in the order $ba$ (Condition a).

- For input symbol $a$ and output symbol $b$, and for arbitrary trace $t$, if the behaviors $ta$ and $tb$ are legal for the module, then the behaviors $tab$ as well as $tba$ must also be legal (Condition b).

## 3 Details of the Optimizer

Currently we identify the sub-network to optimize manually. In order to guarantee that the environment of the sub-network will obey the burst-mode assumption, delay analysis (currently done through simulation) is necessary. The environment of the sub-network (*e.g.* channels and datapath connections) must be suitably specified, to avoid obtaining too general a result (currently done by introducing fictitious modules that possess the required I/O traces). After the network being optimized has been composed into a single trace structure, its description is converted into an EISG by exhaustively "simulating" all their possible moves until all their reachable configurations are covered. Once EISGs are obtained, they are converted into equivalent burst-mode machines by means of *burst-mode reduction*. Basically, this algorithm traverses a path of the state graph starting from the starting state, collecting input transitions occurring along the way into the set *input-burst*, till it encounters a state that has only arcs labeled by output-signal transitions exiting it. The traversal is continued, now forming the set *output-burst*, till a state that has only arcs labeled by input-signal transitions exiting it. A burst-mode machine transition is now formed, and the algorithm continues processing the rest of the state graph. *The basic intuition behind* burst-mode reduction *is that whenever "lattice shapes" representing concurrency are encountered*

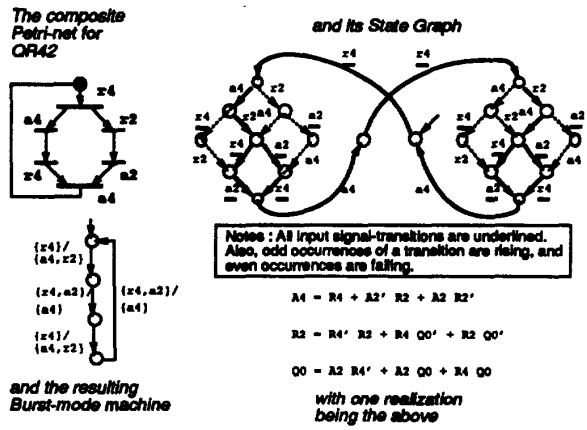


Figure 2: Optimization of the four-to-two converter

*in the state graph, such lattices are collapsed into input- and output-bursts.*

The input to burst-mode reduction is an EISG, which is a state graph with circles denoting states, and arcs between states labeled by a single polarized transition of an input signal or an output signal. Only those EISGs obtained by composing macromodules obeying restrictions stated earlier are considered. The output is a burst-mode machine. The method is as follows.

1. Mark all states as "not visited", and call the starting state *current*.

2. This step addresses the collapsing of "lattice shapes" in the state graph. Specifically, if *current* has not been visited, mark it as visited. If *current* has an exit through at least one output transition, retain any arbitrary output transition, while eliminating all others. Call the destination of the retained transition as *current*, and continue with Step 2. Else (all exits are through input transitions) retain all the transitions out of *current*, and consider all their destination states to be *current*, in turn, and continue with Step 2 for these states.

3. (We reach here after the initial "transition elimination" portion of the algorithm is over.) Remove unreachable portions of the state graph.

4. Set the starting state of the state graph as *current*.

5. Go to the *current* state. It will have exits only through input transitions. (This invariant is initially true due to the quiescence of the starting state, and is preserved by the way the following loop will work.)

   Take any path out of *current* and traverse it, collecting input transitions encountered along the way into a set *input-burst*. (We will never encounter a state in the interim that has both an input exit as well as an output exit.) Continue collecting input transitions, till we encounter a state with exactly one output exit. Call this state *intermediate*.

6. Continue traversing along an arbitrary path from state *intermediate* collecting output transitions into a set *output-burst* till a state which has no exits through an output transition is encountered. Call this state *next*.

7. Construct a burst-mode machine transition from *current* going to *next* labeled by *input-burst/output-burst*.

8. Repeat the procedure from Step 5 for all paths emanating from *current*.

9. Repeat Step 4, now treating all the states marked *next* as *current*, and till all states have been visited.

10. Eliminate all duplicate transitions in the burst mode machine.

The steps in the algorithm can be justified as follows.

In Step 2, the algorithm chooses an arbitrary output transition among competing output transitions. This is justified because by Udding's Condition (a), (as evidenced by the "lattice shape" of the state graph), the ignored outputs are guaranteed to appear later in sequence. In Step 2, the algorithm "prefers" output transitions over input transitions. This is justified on two counts:

- Because of Udding's Condition (b), competing inputs and outputs are also guaranteed to appear in all possible orders. Therefore, even if an input transition is ignored when it competes with an output transition, that input transition will be offered later in sequence.
- Due to the burst-mode assumption, the environment must

| Circuits | BM m/c size (gates) | MM m/c size (gates) | BM speed (nS) | MM speed (nS) |
|---|---|---|---|---|
| QR42 (hand) | 13 | 13 | 10 | 15 |
| QR42 (v1) | 13 | 74 | 10 | 20 |
| QR42 (v2) | 13 | 60 | 10 | 15 |
| Call2 | 17 | 21 | 4 | 8 |
| Call-C Idiom | 27 | 27 | 10 | 11 |
| Decision Wait (2x1) | 26 | 18 | 8 | 15 |
| Simple GVT (p.1) | 15 | 18 | 4 | 4 |
| Simple GVT (p.2) | 8 | 12 | 10 | 14 |
| Call3-Merge Optimization | 68 | 45 | 11 | 16 |
| Completion-tree size 3 | 8 | 10 | 6 | 10 |
| Completion-tree size 4 | 11 | 15 | 8 | 13 |
| Completion-tree size 5 | 14 | 20 | 8 | 17 |

Figure 3: Performance of our Optimizer

allow the output transitions to happen before it applies inputs to the system. This is why output transitions are "preferred over" input transitions. After the state graph has been pruned on the basis of the above statements, we enter the phase of forming input and output bursts for burst-mode transitions. In this process, it is not necessary to consider the particular order in which inputs or outputs appear in sequence. This is because a delay-insensitive system cannot count on inputs/outputs appearing in any particular order (Udding's Condition (a)). In traversing from state *current* to state *intermediate*, if different sequences of input transitions coalesce into the same input burst set (*e.g.*, `r4` and `a2` of QR42), then all these input transition sequences must lead to the same behavior from state *intermediate* onwards (in other words, all these input transition sequences must cause the same set of output bursts and enter equivalent next states). This will result in duplicate burst-mode transitions which can be eliminated, as in Step 10.

The above reasoning shows that Burst-mode reduction results in a burst-mode machine that has the same behavior as the original macromodule network when that network is operated under the burst-mode assumption. The following well-formedness conditions of burst-mode machines are also guaranteed. The fact that all input bursts are non-empty is guaranteed by the *quiescence* requirement that is an invariant of the loop beginning at state *current* in Step 4. The subset property requires that no input burst can be a subset of another. This is true for the following reasons. In traversing from state *current* to state *intermediate* in Step 4, a sequence $s$ of inputs is collected to form the set *input-burst*. Due to Udding's Condition (a), these inputs will appear in all permutations. Therefore there can be no proper subsequence of $s$ that also goes between states *current* and *intermediate*. The other possibility is that a proper subsequence $s'$ of $s$ leads to a *different* state *intermediate*. Then, the state-graph exhibits non-deterministic choice, and by definition we cannot handle non-deterministic machines. The final possibility is that $s$ and $s'$ are identical and lead to the same state *intermediate*. This will result in duplicate transitions that get eliminated in Step 10. Finally, the unique entry condition is guaranteed by the way an EISG is generated. Essentially a *state* of an EISG includes the state of the interface signals; hence, there cannot be a state-conflict in the burst-mode machine because the EISG will allocate two separate states for non-compatible interface-signal assignments.

## 4 Results and Concluding Remarks

We have a prototype implementation for all the phases of our optimizer described here. Though the execution times of the parallel composition tool and the burst-mode reduction program are worst-case exponential, our examples were processed quickly. In general, the burst-mode circuits generated by us are smaller and faster, as shown in Figure 3. To obtain the gate count of an un-optimized network, the gate counts of the macromodules used in that network were added up. The gate count of the optimized network was obtained from the AND/OR realization that Yun's tool [14] produces. In this table, the circuits *Call-C Idiom*, *Simple GVT* (part 1 and 2), *Control-Block Sharing*, and *Call3-Merge* are various networks produced by the Occam or SHILPA compilers, and *decision-wait* is a primitive similar to a generalized C-element. Notice that our optimizer achieves significant optimization for completion trees. In obtaining speed estimates, we focussed on *cycle time* [19], measured by "closing off" the I/O ports of the asynchronous circuit (modulo the burst-mode behavior) thus turning it into an oscillator, and measuring the speed using a unit-delay simulator. More realistic examples as well as measurement techniques are under exploration. In practice, one may carry out macromodule subnetwork replacement until the required degree of performance is achieved. At this point, one may leave some macromodules (at the "top level") unaltered as they make the control

organization of a large system quite clear.

The authors would like to thank Nick Michell, Ken Yun, and Steve Nowick for their help.

## References

[1] Ivan Sutherland. Micropipelines. *Communications of the ACM*, June 1989. *The 1988 ACM Turing Award Lecture.*

[2] Jaco Haans, Kees van Berkel, Ad Peeters, and Frits Schalij. Asynchronous multipliers as combinational handshake circuits. In *Proceedings of the IFIP Working Conference on Asynchronous Design Methods, Manchester, England, 31 March - 2 April, 1993*, 1993. *Participant's edition.*

[3] Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *International Conference on Computer Design (ICCAD), IEEE*, pages 262–265, nov 1989.

[4] Venkatesh Akella and Ganesh Gopalakrishnan. SHILPA: A High-Level Synthesis System for Self-Timed Circuits. In *International Conference on Computer-aided Design, ICCAD 92*, pages 587–591, November 1992.

[5] S. M. Ornstein, M. J. Stucki, and W. A. Clark. A functional description of macromodules. In *Spring Joint Computer Conference*. AFIPS, 1967.

[6] Erik Brunvand. A cell set for self-timed design using actel FPGAs. Technical Report 91-013, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1991.

[7] Kees van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1993.

[8] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[9] Kees van Berkel. *Handshake circuits: an intermediary between communicating processes and VLSI*. PhD thesis, Philips Research Laboratories, Eindhoven, The Netherlands, 1992.

[10] Ganesh Gopalakrishnan, Nick Michell, Erik Brunvand, and Steven M. Nowick. A correctness criterion for asynchronous circuit verification and optimization. *IEEE Transactions on Computer-Aided Design*, 1992. *Accepted for Publication.*

[11] Al Davis, Bill Coates, and Ken Stevens. The Post Office Experience: Designing a Large Asynchronous Chip. In T.N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proceedings of the 26th Annual Hawaiian International Conference on System Sciences, Volume 1*, pages 409–418, January 1993.

[12] Stephen Nowick. *Automatic Synthesis of Burst-mode Asynchronous Controllers*. PhD thesis, Stanford University, 1993. Ph.D. Thesis.

[13] Steven M. Nowick, Kenneth Y. Yun, and David L. Dill. Practical Asynchronous Controller Design. In *Proceedings of the International Conference on Computer Design*, pages 341–345, October 1992.

[14] Kenneth Y. Yun, David L. Dill, and Steven M. Nowick. Synthesis of 3d asynchronous state machines. In *Proceedings of the International Conference on Computer Design*, pages 346–350, October 1992.

[15] Stephen H. Unger. *Asynchronous Sequential Switching Circuits*. John-Wiley, 1969.

[16] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. MIT Press, 1989. *An ACM Distinguished Dissertation.*

[17] Ivan Sutherland and Bob Sproull. Chapter 6 and chapter 7 of ssa notes # 4702 and # 4703, volume 1, on interface state graphs. Technical memo, Sutherland, Sproull, and Associates, 1986.

[18] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, (1):197–204, 1986.

[19] Steven Burns and Alain Martin. Performance analysis and optimization of asynchronous circuits. In Carlo Sequin, editor, *Advanced Research in VLSI : Proceedings of the 1991 University of California Santa Cruz Conference*, pages 71–86. The MIT Press, 1991. *ISBN 0-262-19308-6.*