Biscotti: a Framework for Token Flow based Asynchronous Systems

Charlie Brej
School of Computer Science, The University of Manchester,
Oxford Road, Manchester, M13 9PL, UK.
cbrej@cs.man.ac.uk

Abstract

Examining novel asynchronous structures, synthesis methods or optimisation techniques usually relies on being connected to a large synthesis system, such as Balsa or Haste, which already has all components in place to construct large circuits. Not being linked to a large tool yields results which can not be easily analysed by the community. This paper presents a framework which can construct and analyse token flow style circuits and enable a rapid implementation of novel algorithms and techniques.

1 Introduction

The Sedate project is concerned constructing circuits with a variety of data encodings and completion conditionings. Although the project is aimed at exploring these variables, as a designer it is beneficial to abstract away the asynchronous nature of the pipelines and the data encoding used, and instead concentrate only on the functional aspects of the design. The Biscotti framework aims to automatically generate the pipelining and completion detection structures as well as choosing the most appropriate encoding style and logic implementation.

All target circuits are push only token-flow [1] style constructions which make it possible to apply a number of optimisations irrespective of the encoding style used. Although many of the optimisations can be applied in a static manor, it is beneficial to extract the behaviour of the design in a test simulation to determine the type of optimisation most beneficial for a circuit in the target environment. To enable this a simulator is included in the system. Because synchronous technology mapping tools have no knowledge of certain timing assumptions, they cannot be used without checking that timing assumptions have been upheld and no illegal transformations (e.g. extending isochronic wire forks) were applied. Thus a timing extraction and techmapping components were also incorporated into the framework.

When developing new algorithms and techniques for asynchronous circuit generation, it is difficult to perform a fair comparison between the new method being researched and a heavily optimised method with an entire tool suite of tools. The aim of the framework is to allow an easy incorporation of advanced asynchronous techniques with the ability to analyse their effect and their benefits when applied in conjunction with the set of already implemented optimisations.

2 Framework Parts

The Biscotti system is composed of a series of libraries. The basis of the whole framework is the netlist library.

2.1 Netlist

The netlist library handles circuits and component libraries. These can be loaded from structural verilog files or constructed by the other tools in the framework. To enable the other tools to perform operations on the circuits, a series of functions are included. There are simple functions allowing the insertion, removal and amendment of elements, wires and buses. There are also powerful functions such as flattening to the level of a set of base gates.

This library is not asynchronous logic specific and can be used to manipulate any style of circuits. It is the job of the next library to perform asynchronous logic specific tasks.

2.2 Async Transformations

Circuits in the netlist library can be thought of as having one of two representations. The first is one of being base level where every gate represents a real gate in the design. The in the second representation every component is an abstract functional description of the desired construction. These abstract components can operate using multi-valued signals (as opposed to just 0 and 1). They may also contain high level primitives which communicate the pipelining structure, and the placement of the initial tokens (initial marking) in the design.

In the abstract level, it is trivial, for example, to divide a stage into two using a series of half-buffers. This is achieved by placing abstract half-buffers across the stage. Alternatively, in the non-abstract circuit level, the pipeline would have to have all its completion detection logic and data encoding reverse engineered and re-implemented. This is a difficult and risky task for a relatively common procedure in slack matching[2, 3] type optimisations.

2.2.1 Pipeline Generate

The abstract description contains two types of elements; the pipelining control elements, and the data processing logic elements. To transform the abstract netlist into a concrete circuit, these two types of elements must be replaced with their concrete implementations.

When implementing the pipelining structure, the first task is to generate the acknowledgement tree. The acknowledgements

feed from the output latches of each stage, forking at every multiple input logic element, merging (using a C-element) at all forks and multiple output elements. This generates a clumsy structure which often contains redundant parts. It be easily improved, but at this stage it is left in its suboptimal state to be processed by later optimisation stages. If early output logic [4] is used, an additional validity tree is constructed, flowing in the opposite direction to the acknowledge tree.

The second job of the pipeline generation is the construction of the latch controllers. The system currently assumes a return to zero code with a pre-defined completion detection scheme aimed at the target data encoding scheme. In the case of a dual rail (1-of-2) encoding, the data signals are transformed into two wires and an OR gate is used to generate the completion detection. The data encoding style is specified by the logic synthesiser which is capable of generating many logic styles form a single description. It is also possible to avoid the logic synthesis stage and instead leave the logical blocks in their abstract state. The completion detector used will use multi-valued logic values to communicate the value and spacer codes along a single signal. This enables the designed system to be tested independent of the logic implementation.

2.2.2 Logic Synthesis

The primary job of the logic synthesiser is generating the gate level implementation of the abstract logical function description elements. It also, as mentioned above, communicated the data encoding style used. Currently there are two logic styles fully implemented: early output and DIMS. These both use dual-rail signalling. A further style which will be implemented in the near future as a part of the Sedate project will use a variety of data encodings and conditionings.

2.2.3 Wagging

One of the most powerful transformations is wagging [5]. Because four-phase circuits have a large cycle time compared to their latency, it is beneficial to perform the next cycle of computation on a different unit than one used last (as the one used last will be resetting for a period of time). The wagging transformations replicate the whole design several times, attaching the signals which would be feeding to latches in one slice, instead to the latches in the next slice. This can be easily done in the abstract level.

2.3 Simulation

The simulation library allows for designs, in their test benches, to be simulated and their performance determined and analysed. The simulator is capable of simulating the abstract level circuits which can communicate multiple codewords across a single signal wire. The library can also be used to extract the slowest trace of a simulation. The data from this can then be used to improve the performance of the design.

The circuit can be simulated in a number of different timing models. The single gate delay model uses a value of 1 as the delay of all elements. The unit gate delay model uses a value estimated by looking at the complexity of the element (input and output counts). The unit delay model is especially useful when dealing with abstract functions which have no timing technology based information. The extracted model takes the timing from a timing extractor to give reasonably accurate delay values. The timing extraction is either performed by a commercial tool and read in using the SDF format, or it can be performed by another library in the framework.

2.4 Timing Extraction

Timing extraction performed by tools such as PrimeTime is often not suitable for asynchronous circuits due to their cyclic timing dependencies. In synchronous circuits all activity in the design is caused (directly or indirectly) by a clock edge. The activity in asynchronous circuits is caused by other activity leading back further and further to form a loops of odd numbers of inversions. Because, in timing extraction, the calculated slew of the output is based on slew of the input, the loops cause cyclic dependencies in the algorithm to arise. The solution to this problem is the non-ideal breaking of all loops. The library aims to solve this problem by reimplementing the algorithms to suit asynchronous circuits.

The cell timing descriptions can be read from TLF (Timing Library Format) files. These contain spline tables describing the timing of an element in a array of possible environmental conditions. This information can then be applied to a circuit to extract all input to output path (rising and falling) transition delays.

2.5 Tech-map

In addition to the standard verilog primitives, the netlist library has additional asynchronous logic specific primitives. The majority of these are the abstract pipelining elements, but there are two concrete elements which are useful when mapping the design into a library of cells. These are the C element and the C element with withdrawable inputs. The distinction between these elements being that the C-element with withdrawable inputs cannot be constructed from a tree of C-elements. Luckily these are rarely used and thus most C-element blocks can be processed without having to abide by that restriction.

2.5.1 Resynthesis

Blocks of C-elements make up a large portion of most designs. Because the availability of large C-elements is technology dependant, the process of forming the the C-element blocks into trees is only performed once the available C-element sizes are established. The flattened design is firstly hierarchysed into separate blocks of C-elements and logic. Any un-understood and locked elements remain in the top level netlist. The C-element blocks, which by this stage are directed flow graphs of a number of inputs and outputs, are then turned into a dependency table. This table is then processed by adding implicants which can be shared by multiple outputs, thus reducing the overall size.

This approach gives a lower area (and power) design, but it does not target performance. In order to improve the performance of C-element blocks, a slowest trace is used to guide the resynthesis procedure. The critical input to output transitions of the block are identified and when the

resynthesis procedure is operating it avoids using intermediate implicants which will increase the gate distance between the input and output transitions. The analysis and resynthesis can be repeated any number of tiles until the performance cannot be improved any more.

The logic blocks can also be resynthsised. This work has not been carried out yet but the process will be slightly different from that used in the C-element blocks. When dealing with logic, rather than fully resynthsising the entire block back from the reverse engineered specification, only the critical portions of the circuit are altered. Using the slowest trace an input to output path can be identified which can be shortened. But, instead of shortening the path the tool will place an auxiliary path which will, in some circumstances, provide an output transition earlier than by passing through the original path. One example of this is an addition of an extra input into an OR gate in a carry chain. This extra input can detect, using a wide gate, a positive carry in the target bit. This way the result is generated as usual but in the observed situation the result will also be generated using the supplemental hinting logic.

2.5.2 Drive Strength

The techmapper by default uses the smallest gates in the library to decrease the area and power. The slowest path can also be used to direct the tech-mapper as to increase the performance of the circuit by directing it to examine the input to output transitions of gates in the slowest trace. The techmapper then examines possible implementations of the gate in question and input pin rearrangements. Using the timing extraction library it then picks the one which gives the greatest performance improvement (if any). There is a reasonable variation of up to 20% in the delay of a gate depending on which input pin caused the transition. Rearranging the input pins alone gives performance improvements of 5% to 10%. Choosing the gate drive strength depending on simulation patterns reather than static methods allows lower area components to be used in non critical paths, and additionally reduces the capacitance of wires. Further optimisations such as buffering off (thus reducing the capacitance) non-critical net destinations should yield additional performance improvements.

3 Processes

Although the the framework is still not easily usable it is currently capable of performing many of the tasks for which it was designed.

3.1 Timing Extraction

One of the first uses of the framework was to perform timing extraction of asynchronous circuits which were problematic in for commercial tools. To examine the accuracy of the method, the timing results of the framework's implementation were compared to those from a commercial tool. The delays were within one percent of the commercial tool's values. The method can be used to generate SDF files which can be exported to external simulators, or be used directly by the internal simulator.

3.2 Slowest Trace Extraction

Slowest trace extraction is the primary reason for implementing a simulation system into the framework. The slowest trace can be extracted from either circuits read directly from verilog or ones generated by the system. The information can be used by the designer to determine bottlenecks in the design. The data can also be used by the internal optimisation system to improve the performance of the design.

3.3 Drive Strength Optimisation

The slowest trace supplies the input pin to output pin delay which needs to be improved. This can be achieved by swapping the input pin connections, as cells have different timings for different input to output transitions. Pin swapping is not a technique which is applied in most techmappers so it can be applied to good effect even in optimised designs. In addition to pin swapping the method also performs element replacements for ones with a drive strength most suitable to a particular placement. Because an optimisation changes the timing of the circuit, the slowest path may change and the slowest trace extraction process should be repeated to optimise a new set of weak links in the design.

3.4 Desynchronisation

The simplest flow to generate asynchronous circuits is using desynchronisation which allows the designer to construct the functional behaviour of the design without having to also construct the self-timed logic, latching and acknowledge networks. The input design circuit is very close in form to the abstract circuit representation so only minor processing needs to be applied (clock net removal and removal of clock pins on flip-flops).

The design can then be simulated in it's abstract form, or be transformed into an asynchronous circuit (at this stage either DIMS or early output).

3.5 Sedate Flow

Implementing the full Sedate flow is the target of the framework. This involves reading in a behavioural description and transforming it into an abstract pipeline representation. In this stage the abstract model is analysed and the pipelining of the design is mapped out. Once the pipelining is fixed in place, the design can be simulated in more detail. Extracting from the simulation the information about the use of each logic unit, the next step is to fix in place the data encodings used for each channel in the design. After this, circuit implementations the logical blocks can be constructed. Replacing the abstract logic blocks with concrete implementations allows a full physical simulation. Again, from this simulation more data can be extracted which is fed to the logic block generator. The logic block generator uses this information to make more suitable implementations. These new implementations are replaced into the design and the simulation cycle can be repeated.

The logic block generator also generates timing assumptions which must be abided by for the circuit to function correctly. These timing assumptions make certain implementations of logic blocks not applicable in some circumstances. The

optimisation system chooses the fastest blocks for a situation, but with timing assumptions which can be abided by. The techmapper is the last stage in the process which aims to improve the performance of the design, yet not break any timing assumptions.

4 Conclusions

Because the tool set is constructed depth first, at this stage it is only usable for a very narrow range of tasks. This is a temporary measure aiming at exploring the thesiability of methods and algorithms. Ultimately it is hoped that the same framework can be the basis for a number of language inputs, logic styles and optimisation schemes. With a suitable framework it would be possible to rapidly research new alogorithms and techniques on an array of design methodologies without having to construct an entire system.

The project is now capable of performing many complex techniques but there are still many more elements to be constructed to make it general enough to be used as a universal framework. Another part of future work will be implementing many of the alogorithms described in asynchronous community publications and examining their productivity in a fair environment.

The project is available under the GNU public licence.

References

[1] J. Sparsø and S. Furber. *Principles of Asynchronous Circuit Design - A Systems Perspective*. Kluwer Academic Publishers, dec 2001.

[2] Peter A. Beerel, Nam-Hoon Kim, Andrew Lines, and Mike Davies. Slack matching asynchronous designs. In *International Symposium on Asynchronous Circuits and Systems*, pages 184–194, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[3] Piyush Prakash and Alain J. Martin. Slack matching quasi delay-insensitive circuits. In *International Symposium on Asynchronous Circuits and Systems*, page 195, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Charles Brej. *Early Output Logic and Anti-Tokens*. PhD thesis, 2005.

[5] Charles Brej. High performance asynchronous circuit design method and application. In *UK Asynchronous Forum*, 2007.