Timing Verification for Asynchronous Design

Rhodri M. Davies and John V. Woods
Department of Computer Science, University of Manchester,
Manchester, M19 3PL, U.K.
Email: rdavies@cs.man.ac.uk jvwoods@cs.man.ac.uk

Abstract

This paper describes a technique for verifying timing conditions inherent in self-timed VLSI designs that make use of the micropipeline design strategy. By checking bundling constraints during simulations, design faults may be detected, whilst timing information extracted during the processing may be used to identify modules requiring optimisation. These analyses may be built around existing simulators.

1 Introduction

Recently there has been a resurgence of interest in self-timed (or asynchronous) design. In this methodology there is no global clock controlling the synchronisation of a circuit; instead, adjacent modules communicate with each other, negotiating the transfer of data.

The renewed interest in self-timed design arises from its potential to address areas in which there are perceived to be problems with the synchronous approach:

Clock Distribution: As feature sizes have been reduced and clock speeds have increased, the generation and distribution of global clocks has become an increasingly complex task. Recent designs, such as the DEC Alpha [4], have shown that with careful work and sophisticated CAD support these problems may be overcome, but that this requires significant resources, both in terms of design complexity and silicon area.

Complexity: VLSI designs are becoming increasingly complex. Managing this complexity can be a challenging task, particularly where all the components in the design are interlinked by the global clock.

One approach to reducing complexity is to introduce increased abstraction by dividing the design into modules. The top levels of the design are then specified in terms of the interfaces of those modules without needing to be aware of their internal implementation. This also facilitates the re-use of existing designs through the development of libraries of complex components. The self-synchronised communication in a self-timed design makes it easy to apply modular abstraction.

Performance: In a globally synchronous design the clock period is determined by the worst case delay through the slowest signal path between clocked register stages. In a self timed design, adjacent modules may transfer data as soon as the producer has completed its operation and the receiver is ready to accept more data. Modules may even adjust their timing according to the data they are processing. For example, an adder could operate more quickly in cases where the carry propagate chains are short than when the carry must be propagated across the entire word. Consequently the average delay through a self-timed module approaches the 'typical' case rather than the worst-case.

Power Consumption: In basic synchronous modules, components such as latches consume power on every clock transition, irrespective of whether the module is active or not at that moment. This problem may be reduced by adding clock gating, but the equivalent behaviour is inherent in a self-timed circuit, which will become dormant whilst it waits for fresh input.

None of the major CAD systems provide specific support for self-timed design, although many existing tools for tasks such as schematic capture, layout and simulation are equally applicable to both synchronous and asynchronous design. In other areas such as synthesis, optimisation and validation, existing tools are not suitable for use with self-timed circuits.

To compensate for this, specialised packages have been developed. Most of these allow specification in some form of CSP based language [6], signal transition graph [1] or state machine [2] and then synthesise a circuit, normally using standard cells as the basic building block. Many of these packages are limited in the size and complexity of the designs which can be processed. The most complete of these systems is based on a CSP-like language called Tangram [8]. Circuits specified in Tangram may be simulated, synthesised, laid-out and optimised. The circuits produced using this system are *delay insensitive*, operating correctly irrespective of delays in logic elements and wiring.

At Manchester University the design of two generations of a self-timed implementation of the ARM architecture[1] have been undertaken. These AMULET processors [5] are based on Sutherland's *micropipelines* which compromise delay insensitivity, selecting a more cost-effective solution using local delay management.

2 The micropipeline design strategy

The micropipeline framework was described by Ivan Sutherland in his 1988 Turing Award Lecture [7]. Adjacent modules in a micropipeline are connected by two handshake signals and a data bus (see figure 1).

2.1 Bundled data interface

In this strategy, the sender module places data on the bus then sends a *req* signal. This indicates to the receiver that its input data is available so it may begin processing. When the receiver has no further requirement for the input data it sends back an *ack* signal. The sender must hold the data steady until it receives the *ack*. It may then output the next set of data. This arrangement is referred to as the *bundled data interface*.

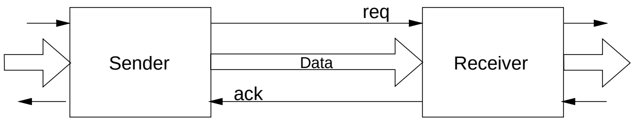

Figure 1. The bundled data interface.

The information in the data bundle is encoded in the conventional manner with the voltage levels of the signals indicating the data value. This is known as a *single-rail encoding* as opposed to the *dual-rail encoding* used in some other asynchronous techniques. The later uses no handshake signals but instead two wires are used for each data bit, with transitions not only indicating the value but also providing the timing information. The single-rail datapaths used in micropipelines are similar to those used in equivalent synchronous circuits. The absence of timing information within the data signals requires the additional handshake signals to provide local control of delays.

[1]ARM is a trademark of Advanced RISC Machines Ltd.

2.2 Bundling constraints

The timing constraints that must be applied to ensure correct operation of a bundled data interface are known as *bundling constraints*. The basic constraints are:

1. The receiver must not see a *req* until its input data is stable.
2. The receiver must not send an *ack* until it has no further need for its input data.
3. The sender must hold its output data steady between sending the *req* and receiving the *ack*.

In practice these simple rules are complicated by the need to allow suitable set-up and hold times and by the edge times of the signals involved.

As a result of these restrictions, circuits must be constrained to ensure that the delay through the control path is greater than that through the data path. Careful timing analysis of the paths may be required together with the inclusion of additional matching delays in the control path. This strategy is known as a *bounded-delay* timing model since the circuit will operate correctly only if delays on the data lines are within the limits set using the control signals.

In this scheme there is an inherent conflict between the need to add extra delay to the control path to allow a suitable safety margin and the need to match the two paths as closely as possible to ensure maximum performance of the circuit.

2.3 Two-phase protocol

In his paper, Sutherland describes a *two-phase* communications protocol for handshake signals (see figure 2). Information is carried by transitions rather than by the voltage level of the signal; rising and falling edges are equivalent in meaning. This protocol, which minimises the number of transitions on each wire, was used in the design of AMULET1.

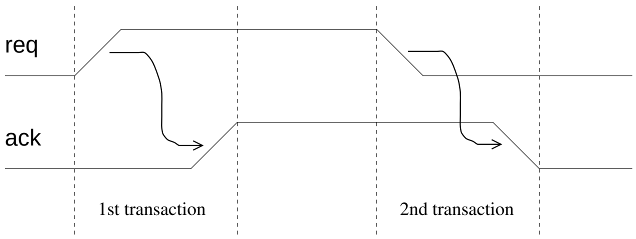

Figure 2. The two-phase handshake protocol.

A selection of event control modules are used to build handshake control circuits. The most fundamental of these are the XOR and Muller C gates which respectively perform

OR and AND functions on transition based signals. Other modules perform more sophisticated functions such as arbitrating between two asynchronous requests.

2.4 Four-phase protocol

It has been found that, in practice, two-phase signals often need to be converted into *four-phase* signals to drive latch circuits. A four-phase signalling protocol is illustrated in figure 3, from which it can be seen that a four-phase protocol may be interpreted in terms of the voltage levels of the signals. This protocol involves more transitions on the handshake lines, but it is better matched to the requirements of efficient latch circuits and so avoids the need for frequent conversions between protocols which has been found to be a bottleneck in AMULET1 [3]. Consequently, four-phase signalling has been used extensively in AMULET2 and the timing verification described in this paper will be discussed in terms of a four-phase protocol. The basic techniques described here are equally applicable to two-phase circuits.

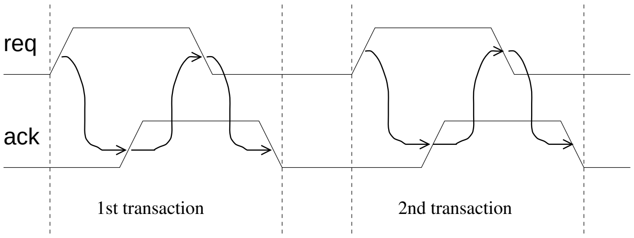

Figure 3. The four-phase handshake protocol.

3 Design flow

The selection of the micropipelined approach with its single-rail datapaths allowed the use of standard cells, tools and test vectors from the synchronous versions of the ARM architecture as a basis for the AMULET development work. This resulted in a reduction in the resources required to develop the self-timed processors and also influenced the selection of the design flow and tools used (see figure 4).

A behavioural model of the processor was first developed. This was then refined through schematics to layout, making use of both custom design and standard cells. These design operations and the associated simulations were performed with the support of the standard synchronous design tools provided by the Compass CAD system[2]. Simulations at the schematic level were performed using the Compass switch level simulator and TimeMill[3]. Spice was used for characterisation of the custom blocks, while TimeMill was used for accurate but fast simulation of the extracted layout.

[2]Produced by Compass Design Automation.
[3]TimeMill is a trademark of EPIC Design Technology Inc.

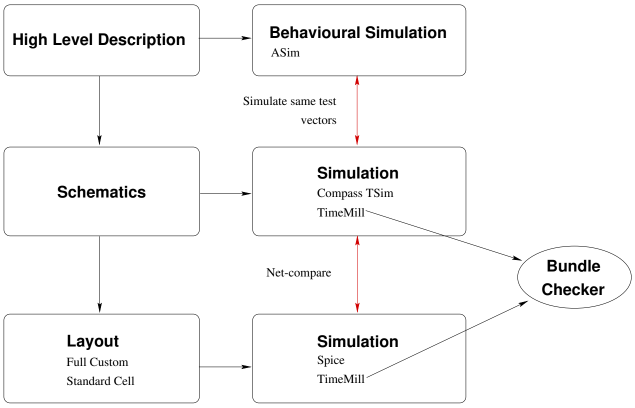

Figure 4. AMULET2 design flow.

3.1 Validation

During the debugging of micropipelined circuits it is advantageous to check the bundling constraints since errors resulting from their violation can be hard to track back to their cause. In addition, such checks add to the confidence in the design obtained from other validations.

In performing bundling constraint checks it is required that:

- The checks may be applied at different levels in the hierarchical design, starting with individual components then moving up to progressively more complex modules until the design may be checked as a whole.
- The bundle definitions from low levels in the hierarchy may be included, without modification, into the validations of higher levels in the design.
- The bundles and associated handshake signals may be specified in some concise, easily read format.
- The active edges of the handshake signals may be specified since both active-high and active-low signals may be used.
- Although the default set-up and hold times may be specified for ease of use, more specific set-up and hold times appropriate to each bundle may be declared, reflecting the different kinds of latches used in the design.

Some existing synchronous design tools do allow the checking of set-up and hold times, but these are based around the assumption that there are only a few clock signals and set-up and hold times are normally measured relative to the same clock signal. It was not considered practical to make use of these existing tools in view of the large numbers of handshake signals and the requirements listed above.

Due to the interactions between modules a static analysis would be complex and would either require extensive modifications to the source of an existing path analysis tool or the development of a completely new tool. As a result a dynamic, empirical approach was adopted, observing the behaviour of the circuit during simulations, as described in section 4.

3.2 Optimisation

Having completed the initial design of a processor and validated its operation, the next stage is to optimise its performance. In synchronous design the basic optimisation task is to find a way of speeding up the critical worst-case path that determined the clock speed. In a synchronous pipelined implementation the critical path through each stage can be analysed in isolation from the other stages and it is easy to identify the stage with the longest critical path, the determination of which may be based on a static analysis of the worst case delay for each component; there are tools to automate this process.

In asynchronous design the optimisation goals are not as clear cut. The main target for optimisation is the "typical" behaviour of the circuit, rather than the worst case. Identification of this "typical" behaviour may not be easy, particularly where the delay through the circuit is data dependent. Although long worst case delays can be tolerated, they can have an adverse effect on other stages in the micropipeline because they cause the pipeline to fill up, and this may need to be taken into account during the optimisation of the circuit. Similarly extremely fast responses under some circumstances can result in *bubbles* in the pipeline, during which the pipeline is not fully utilised. Consequently it is desirable to measure the maximum, minimum and 'typical' delays for each module.

The main measures of the performance of a micropipeline (or an individual module) are the *latency* and the *cycle time*. The latency is the time for information to propagate from the input to the output, whilst the cycle time is a measure of how rapidly the micropipeline can respond; that is the time from accepting one item of data to being able to accept the next. One common optimisation is to trade latency for cycle time or vice versa, according to the environment in which the module is being used. Consequently, determination of these statistics is useful in the rational optimisation of micropipelines.

Static analysis of the maximum and minimum latency is possible using existing critical path analysis tools. Analysis of the "typical" latency is more difficult, particularly where the delays in a module are data dependent. Analysis of the cycle time is more complex since it is affected by the speed at which "downstream" stages reply to the module under examination, so it is dependent on both the structure of the micropipeline and the data passed through it.

Mathematical modelling of micropipelines is a current research topic and although this may produce tools for predicting and optimising the performance of self-timed circuits, for the moment the only practical analyses of cycle times and typical latencies are empirical. Acquisition of these statistics requires the same basic information as bundle checking, so the two operations may be combined into a single tool.

4 Bundle checking

As a result of the consideration of the points raised in the previous section and taking into account the resources available, it was decided to develop a dynamic bundle checker and statistics gathering tool, built around an existing simulator. Although a dynamic analysis will be limited by the coverage of the simulations that it observes, it was not considered feasible to develop any form of static analyser.

Previous experience with building tools around simulators had demonstrated that although "screen-scraping" the normal human-readable output of a simulator is possible, it not efficient. It is more satisfactory for the simulator to produce the same information in a more easily accessible, concise format. In addition, the accuracy of the analysis is clearly dependent on the accuracy of the underlying simulator, both in terms of timings and voltage levels. The TimeMill package was chosen as the underlying simulator since its timing accuracy was suitable and it can produce a trace file that may be read by the bundle checker.

4.1 Error checks

The trace information available to the bundle checker records the time at which a signal crosses either of the logic thresholds, and this information is sufficient to allow the tool to perform some "sanity checks" on the handshake and data signals as well as checking the basic bundling constraints. The checks that it performs are illustrated in figure 5 and described below. As shown on the diagram, the edge times of the signals are taken into account in the checks, which are based on the 'worst-case' relationships between the different signals and their respective edge times.

Bad handshake signal The handshake signals are checked to ensure that all transitions are "clean". For example an error is recorded if a handshake signal goes from 1 to undefined back to 1 instead of making the clean transition 1-undefined-0.

Bad data An error is recorded if a bundled data signal is found to be undefined when *req* becomes active.

Constraint violation This form of error is generated when a bundled data signal changes during the period between *req* becoming active and *ack* becoming active.

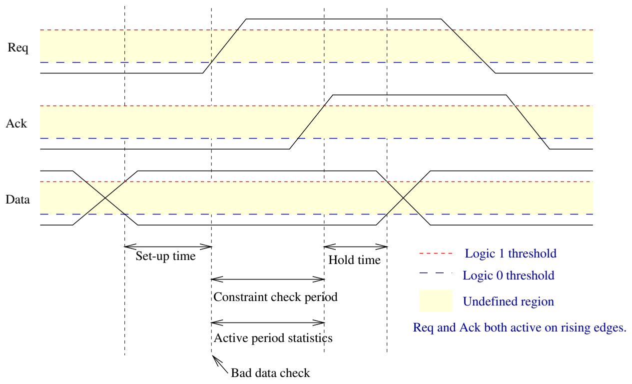

Figure 5. Checks performed by the tool.

Set-up time violation When a data signal change is too close to the active edge of the associated *req* signal, a set-up time violation is recorded. The set-up time is measured from the end of a transition on a data signal to the start of the active edge on the *req* signal.

Hold time violation If a data signal is not held steady long enough after the active edge of the associated *ack* signal a hold time violation is flagged. The hold time is calculated from the end of the active edge on the *ack* signal to the start of an edge on the data signal

In order to support these checks the tool makes use of several interlinked data structures. One of these relates each signal to any bundles in which it is used and indicates whether it is used as a *req*, *ack* or data line, bearing in mind that one signal may be included in several bundles and may perform different functions within those bundles. For example in some latch structures the same signal is used as a *req* to the next stage and an *ack* to the previous one. A further data structure recorded the times of the most recent transitions on the *req*, *ack* and data lines for each bundle. By examining and updating these data structures it is possible to perform all the checks described above.

4.2 Bundle descriptions

A simple plain-text format was selected for the description of bundles since this facilitated easy creation and modification of the definitions and, with provision for the inclusion of comments, allowed the definitions to be easily readable and self-documenting. Table 1 illustrates a typical bundle definition file.

It is easiest for designers to work in terms of symbolic names, so these are used in the definitions file. They are then converted to the corresponding net numbers within the bundle checker.

As can be seen from the table this definition format makes provision for the use of default set-up and hold time and for specification of the active edges of the handshake signals.

```
;example bundle definition file
;default set-up and hold times:
def sut = 2
def ht = 1
;The list of bundles
; req   ack    rqedg   akedg   sut   ht   bundle
req1    ack1   f       f       *     *    data1[0:4]
req2    ack2   r       f       3     5    A_data[0:31]
;Include bundles from the next level in the design
include level1.bundles   level1
```

Table 1. A bundle definitions file.

4.3 Hierarchical designs

When building complex circuits it is normal to make use of a hierarchy of units, and unique signal names are generated by catenating the names of all the units in the hierarchy onto the basic signal name, for example, level0.level1.level2.signal.

The bundle checker was designed to work with such hierarchical structures by allowing the inclusion of bundle definition files for each level in the design. Table 1 includes an illustration of the way in which this is done. The *include* declaration specifies the name of the next bundle file in the hierarchy and the unit name that is to be prepended to any signal names mentioned in that file. Each bundle definition file may include many definition files, and those included files may include further files. In this way it is possible to check individual low level components independently of the rest of the structure, but it is also easy to provide complete checks of the whole structure with each component being checked in the environment in which it is expected to operate.

4.4 Practical considerations

The basic concept and implementation of the bundle checker are simple, but there are a number of practical considerations that complicate its design.

Firstly, traces can be so large that it may not be possible to store them as files. The bundle checker has been designed so that, if the simulator allows it, data can be transferred via a pipe rather than a normal file, making the processing time the limit to the size of problem that can be tackled.

The use of symbolic names rather than net numbers in the bundle definitions is clearly desirable in the interests of readability, but it can cause problems due to aliasing. It is common practice to have several alternative symbolic names for the same signal at different levels in the design hierarchy, but with the underlying tools used by the bundle checker only one of these names will be visible. Further difficulties occur where a signal crosses a hierarchical boundary, but is known by the same name at both levels. For example, a signal may only visible as level0.signal when the hierarchy implies that level0.level1.signal should be an alias for the same signal. These considerations do not affect the checking of a single level design, but can cause complications when the "hierarchical-include" mechanism is used. To avoid these difficulties the bundle definitions file makes provision for the declaration of aliases and for dropping selected unit name components from a signal name when the bundle definition is used as part of a hierarchy.

One further complication may arise during circuit initialisation, during which it is possible that conditions may occur which the checker considers to be errors. To avoid these spurious errors the tool may be set to ignore any errors during a specified initialisation period.

4.5 Statistics

The bundle checker was also instrumented to produce statistics useful during optimisation of a design. The current version of the tool does not record the actual cycle time; instead it records the time between the *req* signal for a bundle going active and the associated *ack* being received, a time marked as the *active period* on diagram 5. This records the time between a sender making a new set of data available and a receiver acknowledging that data. The full cycle time includes this active period plus the time for the sender to prepare the next set of data.

The measurement of the active period rather than the cycle time reflects the bundle checker's view of the design in terms of bundles of signals rather than in terms of modules with input and output signals: the definitions file contains no information about components, only about signals. This also accounts for the absence of any latency statistics. It would be possible to calculate latency and cycle times using the same simulator traces, but this would require additions to the definitions file and extensions to the program or the provision of an additional tool. Consequently the measurement of the active period has been used, since it provides much of the same information as the cycle time. To aid interpretation of the results, the tool records the maximum, minimum and average active periods for each bundle. It also records the minimum set up and hold times, making it possible to identify over-generous matching delays.

5 Conclusions

The Amulet project has demonstrated that complex self-timed designs are feasible, and that existing VLSI CAD packages can support the development of these designs. However, experience with AMULET has shown that the addition of a few simple analyses can greatly improve the ability to optimise and validate the circuits produced.

A tool implementing the bundle checking functions described here was implemented. The source code and documentation for the tool is available from:
http://www.cs.man.ac.uk/amulet/projects/horn/index.html
It was used in the design of AMULET2, during which it was able to identify a number of potential problems which might otherwise have been overlooked or have resulted in errors that were hard to track down. As shown in figure 4, the checker was applied to schematic based simulations to give an early indication of potential problems and was later used with extracted layouts to obtain more accurate analyses.

The bundle checker uses the results produced by simulators capable of accurate time modelling. Should the use of self-timed design become more extensive, CAD support of this type will be required and these analyses could be integrated into existing simulators. Alternatively, as described here, they can be performed by separate programs if the simulator provides suitable "hooks" for attaching additional analyses.

This work was funded by the European Community through the OMI HORN project (project number 7249). The input and assistance of the OMI-DE AMULET2 development team is gratefully acknowledged

References

[1] T. A. Chu and L. A. Glasser. Synthesis of self-timed control circuits from graphs: An example. In *Proceedings of ICCD*, pages 565–571, 1986.

[2] B. Coates, A. Davis, and K. S. Stevens. Automatic synthesis of fast compact self-timed control circuits. In *Proceedings of the IFIP working conference on Asynchronous Design Methodologies*, Manchester, U.K., 1993.

[3] P. Day and J. V. Woods. Investigations into micropipeline latch design styles. *IEEE Transactions on VLSI Systems*, 3(2):264–272, June 1995.

[4] DEC. A 300 MHz quad-issue CMOS RISC microprocessor (21164). In *Proceedings of ISSCC*, pages 182–183, San Francisco, Feb. 1995.

[5] S. Furber, P. Day, J. Garside, N. Paver, and J. Woods. AMULET1: A micropipelined ARM. In *Proceedings of CompCon'94*, San Francisco, Mar. 1994. IEEE Computer Society Press.

[6] A. J. Martin. Synthesis of asynchronous VLSI circuits. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. North Holland, 1990.

[7] I. E. Sutherland. Micropipelines. *Comm. ACM*, 32(6):720–738, June 1989.

[8] C. H. van Berkel, J. Kessels, M. Roncken, R. W. J. J. Saeijs, and F. Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proceedings of EDAC*, pages 384–389, 1991.