A Quasi-Delay-Insensitive Method to Overcome Transistor Variation

C. Brej, J.D. Garside
Dept. of Computer Science, The University of Manchester, Oxford Road, Manchester, M13 9PL, UK.
{cbrej,jdg}@cs.man.ac.uk

Abstract

Synchronous design methods have intrinsic performance overheads due to their use of the global clock and timing assumptions. In future manufacturing processes not only may it become impractical to distribute the clock globally but any timing assumptions will require increasingly large timing margins. This paper presents a method of overcoming these overheads to take full advantage of the improved manufacturing processes. By removing the clock and using self-timed techniques clock related constraints can be discarded. Removing its timing assumptions allows a circuit to perform at a higher speed. An asynchronous logic method allowing the generation of results before the presentation of all input and techniques to allow speculatively fetched data to be removed with a reduced impact on the performance are presented.

1. Introduction

Synchronous circuits use worst case timing assumptions to determine the maximum system clock speed. In addition to the worst case delay the clock period must also take into account an extra margin to compensate other effects such as clock skew/jitter, environmental effects and variability and add them to the slowest stage delay in the system. The effect being "Some designs work twice faster than needed by spec"[1] (see figure 1). Asynchronous computing tries to exploit these margins by removing the clock.

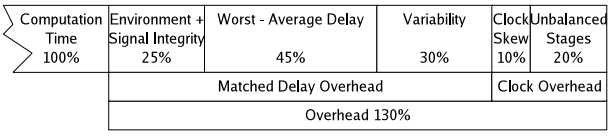

Figure 1: Synchronous timing overheads

1.1. Asynchronous logic

"Asynchronous logic" is a vague term describing any system which does not rely on an external clock as a timing reference. Over time, many categories of asynchronous design styles have evolved. These categories can be split into three groups depending on the method of their timing. Some designers recreate a clock locally using distributed clock generators[2]. The second group ("Bundled data"[3]) uses local delays rather than a global clock to allow pipeline stages of varying depth (and delay) to operate independently. As environmental effects such as voltage and temperature swings affect the delay as well as the logic the circuit can operate correctly in conditions where the synchronous design would require a downrated clock to avoid failure. Although this asynchronous design style makes it easier to meet timing assumptions than the globally clocked version, it still relies on an accurate delay model. With shrinking process geometries the behaviour of individual transistors will become more varied [4][5]. As the gate area moves down towards atomic scales and the number of dopant atoms drops, variation can degrade the performance of some transistors. Using a global clock forces all stages to run at the speed of the longest path in the slowest stage with the worst possible transistors. This is in addition to the clock jitter and skew effects[6] which require even greater timing margins.

There is a third approach which places no assumptions on the delay of any elements.

1.2. Delay insensitivity

Asynchronous designs can be split into classes depending upon the type of timing assumptions they make. Of these classes the most robust is called "Delay Insensitive"[7] (DI). DI circuits make no assumptions on the amount of time a gate or wire takes to propagate a signal. In this model all transitions have to be acknowledged with another transition before the signal can return to its original value. This ensures all messages passed between units have been received correctly, irrespective of the delay of the communication medium or time taken by the receiving unit. Unfortunately due to the heavy restrictions imposed in the DI class no computing circuits are possible, but the ideal can be approached with a small compromise.

1.3. Quasi-Delay-Insensitivity

The Quasi Delay Insensitive (QDI) circuit class is almost as robust as DI. It admits isochronic forks [8] which allows the generation of computing circuits. This is the smallest possible compromise in robustness to the DI model. An isochronic fork is a wire fork where a transition is sensed and acted upon on one branch of the fork and assumed to have been noticed on the other branch. This effectively allows a unit to send a single message to two destinations but require only one to acknowledge.

1.4. Four Phase Protocol

The four phase protocol [9] is one of the methods of communicating between computing units in asynchronous systems. The protocol uses the robust DI class of restrictions enforcing an acknowledge transition for every transition of the messaging signal. Transitions of these signals, named Request (Req) and Acknowledge (Ack), are interleaved ensuring each signal transition is acknowledged by its counterpart.

A message-pass between two units is often called a handshake. The sequencing of a handshake is demonstrated in

Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID'05)
1063-9667/05 $20.00 © 2005 **IEEE**

figure 2. The initiating signal, Req, rises to begin the sequence. This is then acknowledged by the Ack upwards transition. Although a message has been sent between the two units, both wires are now in the 'active' position. The four phase protocol only passes messages by the upwards transition and returns wires to zero between messages. By dropping the request signal which is acknowledged by the drop of the acknowledge signal the system can return to its original state and another handshake can begin.

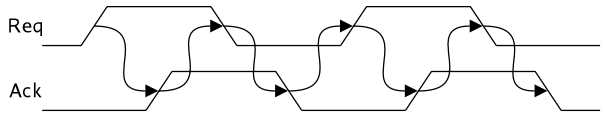

Figure 2: Four-Phase Protocol

Only using one (the upward) edge for message passing might seem inefficient, for much the same reasons as the use of only the rising edge of the clock in flip-flops to latch data, the design becomes simpler. The return to zero also allows the protocol to be used to perform computation as well as communication.

1.5. Dual Rail

In order to create a computing system the units must be able pass data, rather than just empty messages, and perform logical operations on it. Bundled data designs pass data alongside the handshaking channel and rely on timing assumptions between the data computation and the matched delay. Rather than bundling the data with the request and assuming the delay the request signal must pass through is longer than the data propagation, the data can be encoded onto the request signal itself. Separating the request into two or more wires allows a transition of each request to signify different data being propagated. The same acknowledge signal is used to acknowledge any of the requests.

Figure 3 shows the request being split into two signals representing a 'zero' or a 'one'. This "dual-rail" [7] system, a subset of class of "1-of-n" data encoding, transmits one bit of information per cycle.

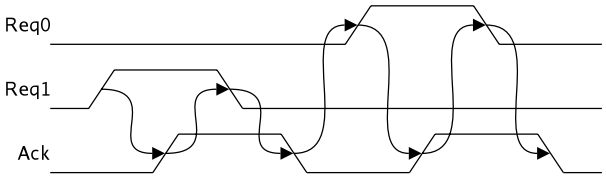

Figure 3: Dual-Rail Signalling

1.6. Dual Rail Latches

Dual-rail latches retain and propagate data. Once the data has arrived the latch will hold the value and acknowledge the input. The latch will assert the output data, even if its input has returned to zero, until it is acknowledged at its output. The latch preserves the correct sequencing required by the four phase protocol by only changing its outputs (request out or acknowledge in) once the output's counterpart has changed.

Figure 4 shows the design of a dual-rail latch. The component marked with the letter 'C' is a C-element [7]. A C-element is a commonly used asynchronous component whose output only changes when all inputs are in the same state. When all inputs are low the output becomes (or remains) low; only when all inputs have become high will the output become high; it will then remain high until all inputs have dropped again. Because of the C-elements hysteretic property, it latches the data. An OR gate then takes the outputs of the latching C-elements and feeds back an acknowledge signal whenever the latch is holding data. The second input of both C-elements is the acknowledge from the output side. This combination enables the latch to output data and acknowledge the input until the acknowledge has arrived on the output and the data has been released on the input. It will then wait until the output acknowledge has been released and new data has arrived before starting another cycle.

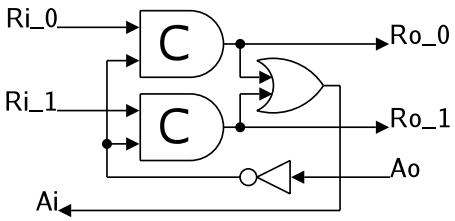

Figure 4: Dual-Rail Latch

1.7. DIMS

Delay Insensitive Minterm Synthesis (DIMS) [7] is one generally accepted method of creating QDI four-phase, dual-rail logic. Figure 5 shows a typical gate constructed using the DIMS method. Each of the four C-elements represents a possible input combination (minterm). Minterms belonging to each output are collected using OR gates. In situations where there is only one minterm associated with an output (like the one pictured) the single input OR gate can be omitted. The two possible outputs (zero and one) can only be generated when one of the minterm C-elements fires and is picked up by an OR gate. On each cycle only one minterm can fire and this will only happen once all inputs have arrived.

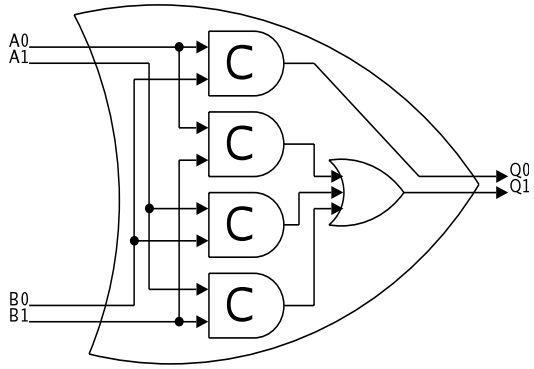

Figure 5: DIMS Gate

1.8. Composition

The three necessary components to create a computing system (communication, computation and hysteresis) have now been shown. Complex logic stages can be created by

connecting gates and latches using dual-rail channels.

Figure 6 shows one such composition. The only part not yet discussed is the fork of the S channel. The request wires are simply forked but the acknowledge wires need to be synchronised with a C-element before being propagated back to the input. This guarantees an input will only be acknowledged once all the latches to which its data propagates have acknowledged.

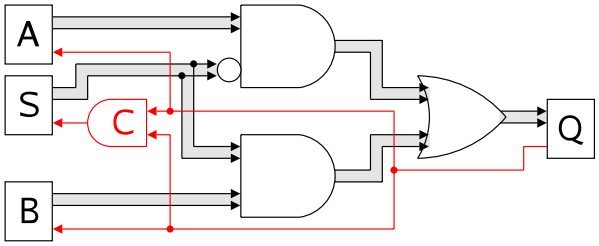

Figure 6: DIMS Multiplexer Stage

2. Early Output

DIMS circuits are very robust but this comes at a high cost in both speed and area. In a comparison with their synchronous equivalents, every two input DIMS gate is over ten times larger and has two to three times longer delay. This means the technology upon which the DIMS approach is applied must have wildly varying transistor delays for the method to have any beneficial effect.

In current high performance microprocessors the trade-off between area and speed has often gone as far as doubling the area for as little as 10% increase in performance[10] (see table 1). Increased caches, speculation units and overall complexity have only given marginal performance boosts.

Table 1: Speed-Area trade-off

	Pentium4 3.4GHz	**Pentium4 3.4GHz EE**	**% Increase**
Transistors (million)	55	169	**207.27%**
Area (mm^2)	127	237	**86.61%**
Benchmarks:			
XVid	6:15	6:06	**2.46%**
KirbiBench	9.01	9.85	**9.32%**
3D Mark	20173	21521	**6.68%**
Comanche 4	69.16	76.58	**10.58%**
X^2	132.11	142.10	**7.56%**
Call of Duty	160.2	177.9	**11.05%**
Benchmark Average			**7.95%**

Even the industry's current willingness to employ large areas of silicon to increase performance would not warrant the use of DIMS with its ten times larger area. In order to justify a delay insensitive approach both the area and the speed of the implemented circuits must be comparable with the synchronous equivalents. Only then can the performance benefits of removing the clock and the worst case delay aspects be realised in the implementation.

The DIMS approach suffers on three fronts: speed, area and the inability to generate results before all inputs have arrived. "Early Output" logic [11] tackles all three areas in which DIMS is lacking, yet still with the possibility of implementing QDI circuits. The main reason DIMS circuits are so large is the enforcement that all timing information is carried on the two data wires. The output is only allowed to become valid once all inputs are valid and released once all inputs are released. Only with this restriction is it possible to acknowledge all inputs safely in the knowledge that they have arrived.

Early output removes the restriction that the output of a gate must signify both the data and the presence of all inputs. Instead, only the data is signalled - the presence of inputs is determined using separate 'guarding' logic. This separation of tasks allows the generation of much smaller implementations dealing only with the logical operation. The guarding ensuring the presence of inputs prior to their acknowledgement is described in subsection 2.2.

Figure 7 presents an early output gate implementation. The early output gate has a delay equal to that of the synchronous/ bundled data delay, and an area only twice the size. Compared to DIMS it is 2 to 3 times faster and over 5 times smaller. The third advantage over the DIMS implementation is the ability to output data once sufficient inputs have arrived.

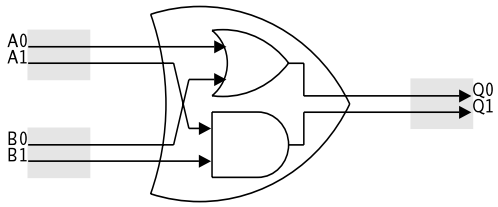

Figure 7: Early Output Gate

2.1. Early Output Cases

Unlike DIMS logic, early output logic generates results as soon as sufficient inputs have arrived to determine the output. Early output cases are common in nearly all circuits. Logic gates such as AND and OR have a 50% probability of generating a result with only one of their inputs present. Other structures such as multiplexers only require the select input and the selected data to generate an output; other data inputs are not necessary.

Most early output cases are intrinsic to circuits composed from early output gates. No design effort needs to be spent on catching most cases. The logic generated from these gates even catches many early output cases the designer could easily overlook, such as the multiplexer being able to generate an output once all data inputs have arrived and are equal. There are often situations where the composed logic does not catch all cases, but a simple tool[12] can easily generate the 'perfect' implementation which can be fed back into the design. These so called "perfect" implementations take advantage of all early output cases.

Early output cases are beneficial to the system performance but they also introduce hazards which must be protected against. As mentioned before, the DIMS gate performs the action of synchronization as well as the logical operation. The output of the early output gate does not reflect the presence of

inputs. This means an input which was late in supplying data unnecessary to complete the operation could receive an acknowledge before presenting its data to the stage. This would break the sequencing required by the four phase protocol and can be considered a hazard. To ensure this does not happen and the acknowledge is kept from the input latch until it has presented its data, a form of guarding must be implemented in the system.

2.2. Guarding

Guarding of early output circuits is essential to create hazard free designs. Guarding relies on the use of an extra signal (Valid) which indicates the presence of data. Validity is indicated by latches and propagated with the data signals. Once it enters the latch with the result of the computation it is used to validate the acknowledgement. Only once the validity has gone high can the acknowledge be propagated back down the pipeline. Figure 8 shows the early output latch design with its validity in and out ports (Vi and Vo). There are several methods of guarding and the resultant designs have different behaviours.

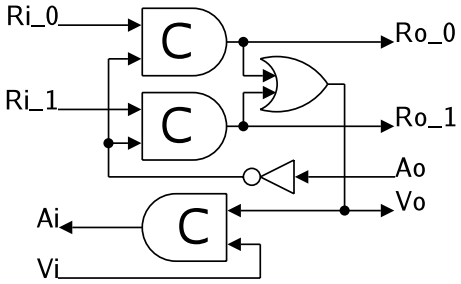

Figure 8: Early Output Latch

2.2.1. Loose Guarding

Loose guarding is the simplest and cheapest guarding method. In a pipeline stage, each output latch gathers the validity signals of all inputs using a C-element. The output of this "gathering C-element" is accepted as the "valid in" input. Only once all inputs are present will the gathering C-element fire and allow the propagation of the acknowledge.

This guarding style treats the whole pipeline stage as a single logic block. It is normal to assume that once a gate's inputs and its output have returned to zero the gate is reset. Here, however, is a complex logic block with many wires inside which are undetected on the outside; even if the inputs and the output have returned to zero this does not signify the circuit is fully reset. These long paths within the logic block are often referred to as 'orphans'[14]. At this point the data from the next set of inputs could interact with the orphans and possibly generating an incorrect result. For this reason the resultant circuit is not QDI. Even though the timing assumptions may be reasonable, in a future design process this might not be the case.

2.2.2. Forward Guarding

In order to ensure the logic is fully reset before allowing another phase of data to enter it, a test of all data wire pairs must be made. OR gates are placed across the data wire pairs. The outputs of these or gates signify the presence of data which must be removed before the acknowledge is released. Figure 9 presents the arrangement of each gate with forward guarding. The validity signals from both inputs are combined along with the output of the OR gate to create a validity of both the inputs and the logic tree.

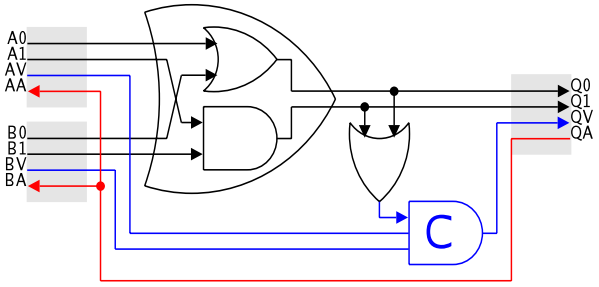

Figure 9: Forward Guarding

2.2.3. Backwards Guarding

Created with exactly the same components as forward guarding, backward guarding simply reverses the direction in which the circuit validity is tested. Instead of testing data validity on the valid propagation it is tested on the acknowledge propagation instead. The acknowledge can reach valid gates but gates with only one valid input will not propagate the acknowledge. Subsection 3.3 explains the differences in the behaviour of the two guarding methods.

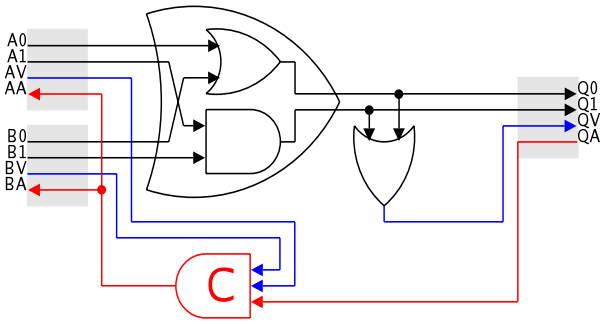

Figure 10: Backward Guarding

3. Results

Early Output is the first delay insensitive approach to match the forward propagation performance of synchronous logic. Comparisons between the logical parts of early output and synchronous designs show exactly the same delays. As there is no computational overhead, the design style should be judged on its other performance enhancing properties.

3.1. Average Case Performance

One of the advantages of early output designs is its use of average case performance. Systems which have to make timing assumptions on the delay of a logic stage (be they synchronous or asynchronous bundled data) must ensure the time allowed for the stage to complete is greater than the worst case delay of the stage. In circuits such as 32 bit ripple carry adders the worst case delays are up to seven times larger than the average case [13]. Most circuits do not have such a large

worst to average ratio due to their designers optimising the worst case paths, often by sacrificing the average case performance (e.g. carry lookahead adders). Instead by optimising the modal (most often occurring) case paths it is possible to increase the performance more substantially. Other timing assumption based overheads totalling 130%, according to [1], can also be removed.

3.2. Early output cases

The main benefit of using early output logic over other DI methods is its ability to take advantage of the lossy nature of logic. This performance can be judged by its ability to generate outputs while ignoring the speculatively fetched data inputs and its behaviour when awaiting the late unnecessary inputs.

The generation of early outputs is dependent on the number and the data of inputs presented to a stage. Taking as an example a commonly used design such as a 2:1 multiplexer, with one of its three inputs missing still has a high probability of generating an output. With one input missing there are 12 possible input combinations. A circuit composed from early output gates will be able to generate a result in 5 of the 12 situations, giving a probability of 41.67% of generating an output with one input missing. Due to its coposition, it does not generate a result whilst having both data inputs valid with data values 'one' and the select input not valid. In this situation neither of the AND gates (fig. 6) can forward a result to the OR gate and so the stage cannot generate a result. This can be improved by passing the circuit through the "Early" tool to generate a perfect implementation which catches all early outputs. In this case a perfect circuit would catch 6 combinations which is 50% of the total.

Figure 11 shows the composed circuit, the perfect circuit generated by the "Early" tool before and after resynthesis of the positive half of the 2:1 multiplexer (zero result generating half in not shown). The "Early" tool generates Espresso [15] style two level AND-OR circuits. This is not always optimal and better results can often be reached by resynthesizing.

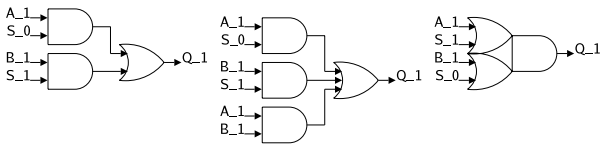

Figure 11: Composed, Perfect and Resynhesized Perfect positive half MUX implementations

Figure 12 shows the probability of an early output in the composed and the perfect circuits with varying number of inputs present in a larger 8:1 multiplexer circuit with 11 inputs. Composed circuit requires on average 9.29 (84.42%) valid inputs to generate an output while the perfect circuit only requires 8.97 (81.57%).

It is easy to see how multiplexers are able to generate early outputs but they do not perform as well as other common circuits. By running the early case test across many benchmark circuits it is possible to see the general pattern of early output generation with the presence of inputs. The benchmark circuits were taken from synchronous designs. These include:

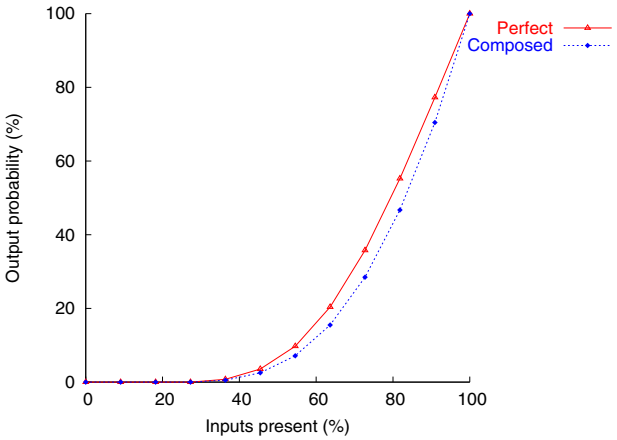

Figure 12: Output probabilities of a composed and a perfect MUX

7 segment display encoder (segment A), a MIPS ALU slice, an 8 input AND gate, bit 8 from an adder, a MIPS processor branch unit, an 8 bit compare equal unit, an 8:1 Multiplexer, MIPS processor memory shift unit, bit 4 of an 8 bit shifter and an 8 input parity generator (XOR gate). The results are shown in figure 13. The best performing circuits are the AND gate and the Compare EQ. In the case of the AND gate just one low input can determine the result and thus with the arrival of the first input the probability of an output moves to 50%. The compare equal unit is similar as it needs only a pair of inputs to be different to generate a result. The worst performing unit was the parity generator which always requires all inputs to be present and no early output cases are possible.

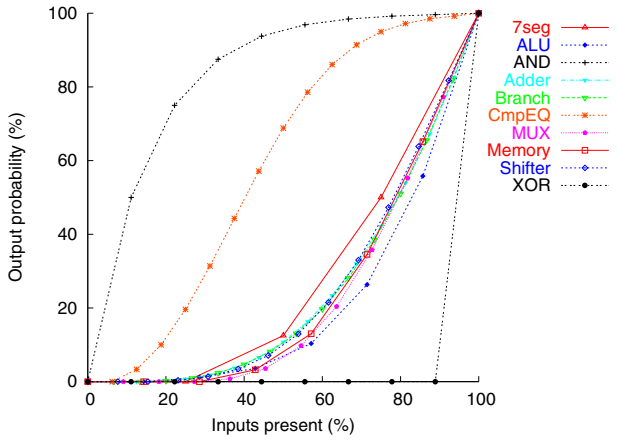

Figure 13: Output probabilities of benchmarks

3.3. Backward Guarding

Forward guarding and backward guarding might seem very similar but they behave very differently. In the computation phase neither of the two methods impede the data propagation. During the reset phase the two techniques have different advantages. The forward propagation parallelises the detection of the circuit being valid with the data propagation while the backwards guarding will only start to test the logic validity once the result has been generated and the

acknowledge starts propagating back through the logic. This allows the forward guarding, which executes the circuit validity checking in parallel with the data operation, to reset faster and start computing the next set of data earlier. In backwards guarding, circuit validity checking can start even before all inputs have arrived, as long as the result has been generated, and the acknowledge can reach the inputs which have arrived and contributed to the generation of the result. Any inputs which have yet to contribute data to the stage are protected from being acknowledged as the acknowledge will only propagate through gates with both inputs valid. The acknowledged inputs can reset and even start another computation cycle while the acknowledge signal surrounds the remaining inputs until they present data to be acknowledged. This allows inputs to desynchronise.

One of the often-stated benefits of asynchronous logic is its composability. With standard interfaces it is possible to connect stages of forward/backward/loose guarding early output and DIMS or with protocol convertors to bundled data and clocked circuits. Forward and backward guarding circuits will perform best in different situations and mixing the techniques across stages with different behaviours gives the optimal result. Forward guarding works best in stages where all inputs arrive at approximately the same time. Backward guarding logic allows the computing stage to be desynchronised from some speculatively fetched inputs. The desynchronisation goes as far as allowing a stage to move onto the next set of inputs while it is still acknowledging some late and unnecessary speculative operations.

4. Conclusions

It is not possible to conduct a fair comparison using only simulation data. Manufacturing process data sheets usually only state the worst case transistor behaviour. The variation in transistor delays is known to be increasing but little data is available. Ultimately the only effects visible in a simulation comparison are the “Worst - Average case delay” and the “Unbalanced Stages”. Both of these are highly dependant on the skill of the designer.

Early output logic does have overheads which may impact performance. Early output gates may be smaller than DIMS implementations but they are also much larger (aprox. 4 times) than the synchronous equivalents. This can be justified on the grounds of performance to area ratio if the resultant circuit executes 20% faster. As tokens flowing through a pipeline are separated by resetting stages, the pipeline occupancy is below 50%. Again this can be justified as asynchronous logic only propagates data through a pipeline when necessary and there is no need to pass spacers (nop instructions).

Often the critical path is much longer than necessary as the last input to arrive at each stage is not always needed. Early output moves the critical path to only the necessary data operations but still keeping the coherency of the system by waiting for the unnecessary data to arrive before removing it.

A common method of increasing performance is increasing speculation. Backward guarding allows the slow, speculatively executed but unnecessary operations to have a minimal effect on the forward propagation in the system. This is beneficial in modern designs where speculation is an increasingly common method of improving performance.

In summary what has been presented is a method of generating circuits which have the same element delay as the synchronous implementations, yet they remove the overheads of both the clock and timing assumptions. Additional performance is gained by early output cases and highly speculative operations with little to no penalty in performance. With only the minimal timing assumptions the design can be guaranteed to work when implemented with poor quality materials and processes yet still operate at optimal speed in a large range of environments with dynamic variations in temperature and voltage.

5. References

[1] Peter A. Beerel, Jordi Cortadella and Alex Kondratyev, "Bridging the gap between asynchronous design and designers", VLSI Design Conference, Mumbai, January, 2004.

[2] Scott Fairbanks and Simon Moore, "The Distributed Clock Generator", Second ACiD-WG Workshop, Munich, January, 2002.

[3] Ivan E. Sutherland, "Micropipelines", Communications of the ACM , Vol. 32, No. 6, June 1989, pp. 720-738.

[4] Mariusz Niewczas, "Characterisation of the Threshold Voltage Variation: a Test Chip and the Results", Proc. of Intl. Conf. on Microelectronic Test Structures, Monterey, pp. 169-172, March 1997

[5] T. M. Mak, “Is CMOS more reliable with scaling?”, IEEE Int. On-Line Testing Workshop, July 2002.

[6] H. Bakoglu. “Circuits, Interconnections, and Packaging for VLSI”, Addison-Wesley Publishing Company, 1987.

[7] D. E. Muller and W. S. Bartky, "A theory of asynchronous circuits," Proceedings of an International Symposium on the Theory of Switching, Cambridge, MA: Harvard Univ. Press, pp. 204-243, 1959.

[8] A. J. Martin, “The limitations to delay-insensitivity in asynchronous circuits”, Advanced Research in VLSI, MIT Press, 1990, pp. 263-278.

[9] S. B. Furber and P. Day, Four-phase micropipeline latch control circuits, IEEE Transactions on VLSI Systems, vol. 4, pp. 247253, June 1996.

[10] Tarinder, “Intel Pentium 4 3.2GHz Prescott, 3.4GHz Northwood, and 3.4GHz Northwood Extreme Edition”, HeXus.net, 1 February 2004, “http://www.hexus.net/content/reviews/review.php?dXJsX3Jldmlld19JRD02OTYmdXJsX3BhZ2U9MQ==”.

[11] C.F. Brej, “An automatic synchronous to asynchronous circuit convertor”, 11th UK Asynchronous Forum, 2001.

[12] “Early resynthesis tool”, http://www.cs.man.ac.uk/~brejc8/early/

[13] J. D. Garside, "A CMOS VLSI implementation of an asynchronous ALU”, IFIP Transactions on Asynchronous Design Methodologies, Mancester, March 1993.

[14] K.M. Fant and S.A. Brandt. NULL conventional logic: Acomplete and consistent logic for asynchronous digital circuit synthesis, International Conference on Application specific Systems, Architectures, and Processors, Chicago, August 1996.

[15] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers, 1984.