Reduction in synchronisation in bundled data systems

C.F. Brej, J.D. Garside
Dept. of Computer Science, The University of Manchester, Oxford Road, Manchester, M13 9PL, UK.

{cb,jdg}@cs.man.ac.uk

Abstract

With the ever increasing complexity of asynchronous systems the performance cost of synchronisation during the transfer of data becomes greater. This paper describes methods of reducing synchronisations in four phase bundled data systems. The generation of early outputs when sufficient data has arrived to ensure a correct result is used to desynchronise the output of a stage from the late arriving and unnecessary inputs. Using early output logic as a basis, further extensions can be implemented.

Anti-tokens allow not only the removal of a synchronisation between inputs to a stage but also actively progress backwards through the pipeline to remove the unwanted data. This method halts speculative operations when the result is found to be not needed in an attempt to lower power consumption.

1. Introduction

As both the speed of operation and the number of devices on a system-on chip rise there is increasing difficulty in maintaining a synchronous model of system operation. One solution is GALS (Globally Asynchronous, Locally Synchronous) design which exploits asynchronous interconnection of existing synchronous, locally clocked macrocells.

Another alternative is a totally asynchronous design approach. Large scale designs such as the Amulet[2] or the MiniMIPS[3] have shown asynchronous benefits in many areas such as power consumption and reduced EMI.

Asynchronous designs may be broadly divided into two categories: those which minimise delay assumptions (i.e. speed-independent (SI) or quasi-delay insensitive (QDI)) and those which make local delay assumptions (e.g. bundled data) [1]. The former category are more robust against layout variations and changes in operating conditions (e.g. voltage, temperature) but impose a significant overhead in area and, often, performance.

However there is another degree of freedom allowed by SI/QDI systems in that, because each bit symbol carries its own timing information, it is possible to pipeline systems very finely – down to bit level – with almost no additional cost. This freedom can allow the exploitation of more of the oft-quoted advantages of asynchronous circuits [1].

Fine grain pipelining makes less sense for bundled data systems; as in a clocked system their advantages come from exploiting common timing assumptions. There is therefore a temptation to contain a large number of bits in a bundle.

However as systems become more complex the designer is faced with an increasing number of bits flowing around the chip. Large scale synchronisation (if achievable) would lead to a largely synchronous device. Instead the system can be broken down to moderate sized pipelines and benefit from higher concurrency and throughput without the large cost of bit-level synchronisation elements.

As an example, consider a microprocessor. A very simple system is unpipelined and is a simple finite state machine. A higher performance system will exploit pipelining to increase throughput. A simple pipelining will be linear and analogous to a FIFO; a relatively simple system to implement asynchronously.

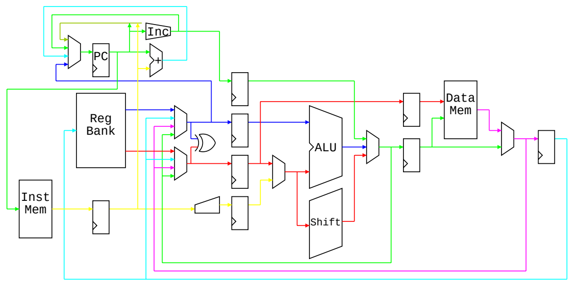

Figure 1: Example complex system

As complexity increases the ‘pipeline’ becomes more complex. If register forwarding paths are added to alleviate dependency problems, the architecture becomes more like a network than a pipeline. This added complexity means that a ‘stage’ tends to have an increasing number of neighbours with which it must synchronise to communicate. In this environment any particular communication may be infrequent, but unpredictable. In the absence of a synchronising clock messages may be sent in case they will be needed.

This paper describes ‘early output logic’ – a variation of weak conditioned logic [4] – applied to a bundled-data system where a method of reducing the speculative traffic by allowing unwanted inputs to be ‘cancelled’ before they have sent data. This avoids unnecessary synchronisations between units where the data communication is not needed.

1.1. Asynchronous circuits

Unlike synchronous implementations, asynchronous systems need to pass timing information with the data. These are handshakes to inform the receiver that data is ready and the transmitter that data is accepted. The computation stages can take an arbitrary, possibly data dependent, time between communications. This allows the system to take advantage of improved performance of simple computations and move towards ‘average case’ performance.

Data packets flowing through a system can be thought of as tokens which reside briefly in a unit before being passed on. The passing of a token synchronises the two communicating units momentarily.

The handshake – ‘data is ready’ (request) and ‘ready for another piece of data’ (acknowledge) – protocol can be constructed in many ways.

All circuits described in this paper use the early ‘four phase’ ‘bundled data’ protocol. To pass a token the request is asserted by the producer once the data on the bus is stable; this is acknowledged by the consumer asserting the acknowledge line. At this point the producer can change the data on the bus and deassert the request. Both the producer and consumer must wait for the signal from the other to signify it may generate another transition. This ensures that both the consumer must hear the producer and vice-versa. This also allows either of the connected units to pause the transaction arbitrarily at any time.

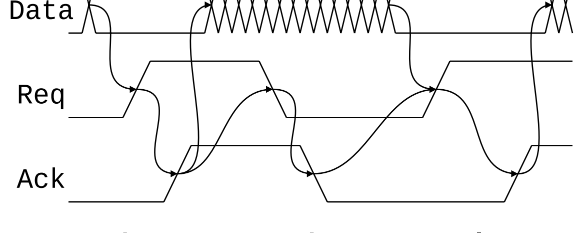

Figure 2: Four phase protocol

1.2. Asynchronous latches

The Muller C-element [5] is one of the basic building blocks of asynchronous circuits. It may have two or more inputs and one output. When all inputs are in the same state the C-element switches state to the same state as the inputs. It will then keep this state until all inputs have switched to the other state.

Typical asynchronous latches use C-elements to enforce synchronisation. There are many such designs [6]. Figure 3 shows an example latch design. The operation of this asynchronous latch is very simple. The request out (Ro) signal is asserted when the request signal from the previous stage (Ri) becomes asserted while acknowledge out (Ao) signal is not and remains so until the request in is released and the next stage has acknowledged the request. Here the latching of the data is done using a ‘clocked’ latch to simplify the illustration. This ‘clock’ signal is taken from the Ro line but it is possible to take the signal from the Ri line. A delay is placed on the output of the Ro line to ensure the data has been updated to the new values before the request signal is emitted. The sequencing of the request to be after the data is called the bundling constraint.

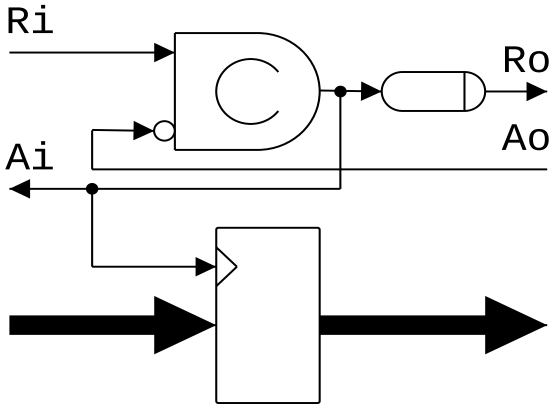

Figure 3: Example four phase latch

1.3. Asynchronous logic

As asynchronous circuits have no clock to provide a reference delay, they have to rely of matched delays to ensure the result of an operation is ready before it is latched.

As shown in figure 4 the request from the input latches has to pass through a matched delay to allow time for the data to pass through logic and be ready to be latched by the time the request signal reaches the output latch. The delay may be asymmetric as it is only necessary to delay the request transition on the rising edge.

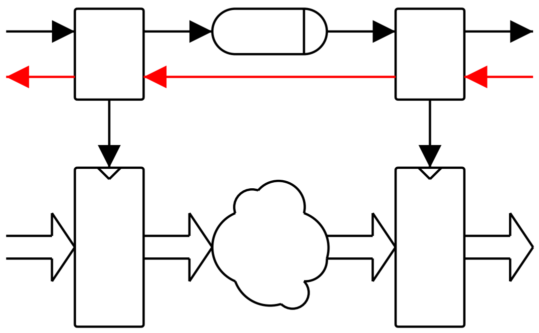

Figure 4: Asynchronous pipeline

1.4. Complex constructions

Figure 4 deals with pipelines with only one input and one output. In order to allow the construction of circuits other than simple FIFOs the system must allow stages with multiple input and output pipelines. Figure 5 shows the datapath of an example circuit with three inputs and two outputs. It is a multiplexer taking two inputs (A and B) and a control input (S) and passes the multiplexer result to output X, while forwarding data S to output Z. Each of the inputs has a set of outputs into which their data feeds and each output has a set of inputs it relies on to generate the result to pass to the next stage.

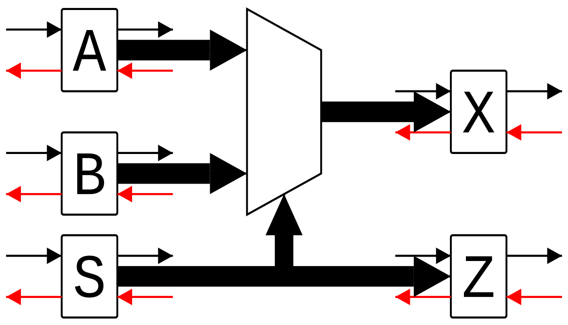

Figure 5: Multiple input and output stage

Figure 6 shows a simple method of generating the completion and acknowledge signals for a pipeline stage described in figure 5. The stage has completed when all the inputs have arrived and their request signals have fired the request gathering C-element, the output of which passes through the matched delay to the output latches. The acknowledge signals are also gathered using a C-element and passed to all input latches.

1.5. Separating complex constructions

Although the method described above is sufficient to produce the correct behaviour, there are unnecessary restrictions on the generation of the request and acknowledge signals. Output Z does not rely on inputs A or B but still has to wait for these inputs before its request signal is generated. Also inputs A and B wait for output Z to accept its data before they receive an acknowledge even though Z is not in their output sets. By separating the request and acknowledge generation for each input and output it is possible to allow the system more freedom and concurrency. The generation of individual request signals also allows better matching delays (e.g. the delay from S to Z may even be removed as no computation is conducted on the data before being moved to the next stage).

When pipelines fork and join some synchronisation must be imposed. Figure 7 shows a simple way of achieving this; the upper C-element ensures that all three inputs necessary to derive X have arrived, with the corresponding acknowledgement broadcast to these sources. The other C-element provides the converse function for the two destinations supplied by input S.

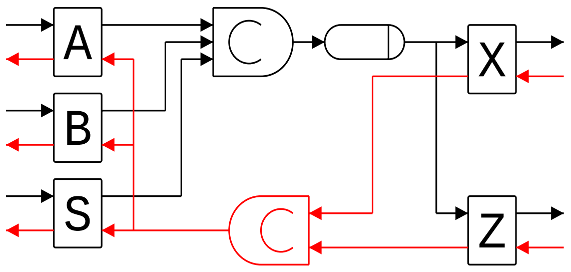

Figure 6: Naive circuit control

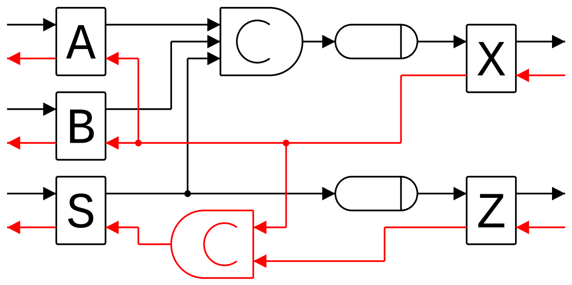

Figure 7: Standard request and acknowledge generation

2. Counterflow systems

Conventional logic systems have a unidirectional computational flow. Methods to allow bidirectional communication along a single channel with the tokens flowing in opposite directions has some useful properties.

2.1. Counterflow pipelines

Sproull's "Counterflow Pipe-line Processor Architecture" [7] is an example system which allows results to flow up the pipeline in the opposite direction to the instructions. In a system such as this, when a result passes an instruction it may be taken into the instruction and replace one of its operands. The result can also be destroyed as it is no longer valid due to the instruction's destination matching the result's.

The counterflow system is generated using two pipelines pointing in opposite directions interlinked with arbiters. As tokens move down the pipeline they place a request to an arbiter to allow them to move to the next stage of the pipeline. There is a mutual exclusion between tokens moving past each other. When moving past a token, flowing in the opposite direction, the arbiter ensures the tokens will meet and not simply move past one another.

The need to perform a complex operation upon the collision of tokens and the need to preserve them after the collision requires the use of expensive arbiters. But often it is simply required that on a collision the two tokens should annihilate each other. This allows a much simplified pipeline structure as tokens never have to move past one another. Such a system could be implemented using just one pipeline.

2.2. Counterflow network

The counterflow network [8] is a system which allows bidirectional communication of control signals. These signals upon collision combine and then remove themselves.

Tokens flowing through the system have a direction of travel and can stretch over several stages. This means the token's leading edge can move several stages ahead of the trailing edge.

The tokens when flowing in the same direction will queue and not combine but when flowing in opposite directions the leading edges of the tokens will collide and combine the two tokens together. The trailing edges then remove the token from all stages until they too collide. This has effectively combined and destroyed the two tokens.

The two building blocks of counterflow networks are nodes and links. Nodes are meeting places for several (at least two) pipelines. Links are used to connect two nodes together.

2.3. Counterflow network circuit

The construction of counterflow networks is very simple. Figure 8 shows the circuit used to generate a counterflow network composed of two nodes connected by a link. The most important thing to notice about the circuit is that it is symmetrical. This means that there is no distinction between tokens flowing in either direction.

Nodes and links communicate using three signals: N (Node request), L (Link request) and R (Ready). A node will 'fire' once all its link neighbours are ready and the firing condition has been met. The firing condition is generated by applying a logic function (cloud in figure 8) on the latch request signals. The function decides which neighbouring nodes must have fired in order for this node to fire. In order to make a single direction FIFO the firing condition is an identity of one of the inputs. A bidirectional pipeline can be created by using an OR gate as the firing condition function.

A link simply forwards the request of one node to its neighbour but once both the neighbours have fired this request is dropped and the link becomes 'not ready' until both nodes have released their requests.

During a token collision both the node and the link elements are unaware the collision took place. The link simply forwards the 'node request' signals from each side to the other node and when both of its neighbours are requesting it drops its ready signal waiting for both to release their request. The node during a collision will 'fire' when any (one or more) of the links pass it a request. The effect of this is that the two leading edges of the token will combine the two tokens. Both the latches and the nodes which have fired will wait for the trailing ends of the tokens to arrive and release them. This has effectively combined and destroyed the two tokens.

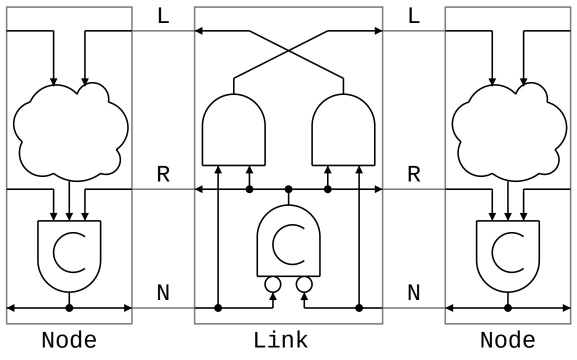

Figure 8: Counterflow network circuit

3. Early output

Early output logic is an extension to the standard four phase bundled data system described in section 1. Separating the completion circuits, shown in section 1.5, allows some outputs to be generated before others. Output latches only have to wait for the inputs in their input set rather than all inputs.

In practice it is often possible to determine an output state before all the inputs have arrived. Methods such as 'weak condition' logic [4] try to generate data as early as possible as this is beneficial to the system performance.

In the datapath shown in figure 5 the separation of the X and Z output bundles is not the only optimisation possible. Although creation of output Z is impossible before input S arrives, the generation of the result X maybe possible before all inputs have arrived.

This can be achieved by generating the request signal once sufficient data to carry out the operation has arrived and a matched delay has passed. The completion signal can be generated by observing the arrival of the input requests along with their data. In the case of the multiplexer the result is ready when either the select line of the multiplexer is '0' and inputs A and S have arrived or the select line is '1' and inputs B and S have arrived.

There is one more case in this example where the output can be generated with an incomplete input set. When inputs A and B are equal the select signal and input S is irrelevant to the result to the operation. To detect this early completion state a comparator is needed and as the circuit data width may be large this can be expensive. It is not necessary to detect all the early output states. In cases such as this the detection of this case is not beneficial as: the area cost maybe too high, the probability of the case happening is too low and the timing of the design ensures that the unnecessary input is never very late. To get the optimal performance the design should only detect early output cases where doing so is beneficial to the performance of the design.

A circuit for early output request generation for the multiplexer stage example is shown in figure 9. As shown this circuit is 'broken', but serves to illustrate how the request to X could be derived. Generating the result before all inputs have arrived breaks the sequencing in the four phase protocol; the last input stage could receive an acknowledge before it generated a request. If the outgoing request is early it is therefore important to ensure all the inputs are collected before completing the cycle.

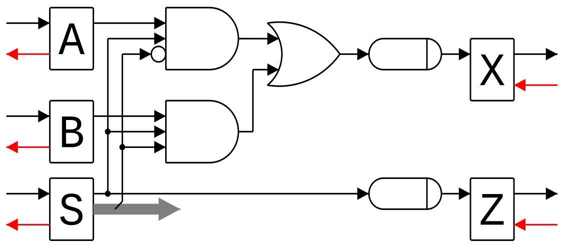

Figure 9: Early output request generation

3.1. Guarding

To protect the input latches from receiving acknowledge signals before being ready to accept them, guarding C-elements are introduced (figure 10). These ensure that inputs will only receive an acknowledgement once they are ready. A guarding C-element takes the acknowledge signal from the output latch and combines it with the request signals of all inputs. Only after all inputs have arrived can the acknowledge signal pass through the guarding C-element.

The C-elements will not release or assert the acknowledge signal until all inputs have released or asserted their request. This ensures that all inputs have accepted the acknowledge transition before continuing to the next transition. This was done by the old request generation C-element but is no longer present in the new version.

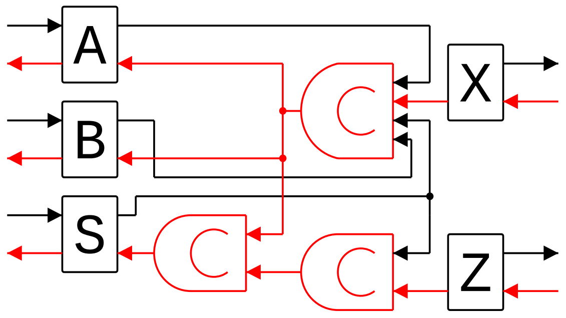

Figure 10: Guarding logic

3.2. Validity

The input request is now used for two purposes: the request signal generation logic and the guarding logic. These have different requirements (e.g. the request passed to the guarding logic does not need to pass through a bundling delay).

Creating a separate signal specifically to drive the guarding logic permits improvements in designs (such as the semi-decoupled latches introduced below) as well as boosting the performance of the system slightly. The performance increase is due to the late arriving tokens not needing to meet latch delay constraints before signalling the guarding C-elements and allowing an acknowledgement. The new signal, separated from request, is called 'Validity out' (Vo).

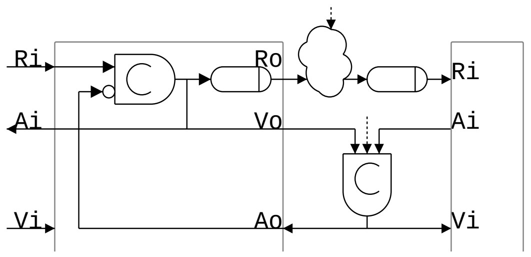

Figure 11: Validity signal connections

In standard latches the Vo signal is generated the same way as the Ro signal except it does not need to pass through the latch modelling delay. Its function is to indicate that the latch is ready to receive an acknowledge.

Another signal is taken into the latch but is not necessary in most latch designs. The Vi signal is generated by the guarding C-element in the previous stage and can be used to tell the latch if the previous stage has completed. This is not necessary in most latch designs as it is possible to observe this by monitoring the Ri signal instead, but it becomes essential when constructing latches which need not receive an Ri signal before acknowledging such as the anti-token latch described in section 4.

3.3. Semi-decoupled latches

A semi-decoupled latch [6] controller releases its output request as soon as it is acknowledged; this allows the output latches of a stage to reset even before all inputs have been released. If this is not done the request can still propagate but will 'stretch' across several pipeline stages.

By separating the request functions into request and validity the construction of a semi-decoupled latch controller becomes simpler. The designer can now make use of the relationship of the inputs and outputs of the design, or more specifically the fact that the Vo (Validity out) line passes through guarding C-elements, to return as Ao (Acknowledge out). This ensures that an acknowledge signal does not arrive until the latch is ready to accept it and has announced this by raising the Vo line. Ai and Vi have a similar relationship which can be exploited (demonstrated in section 4).

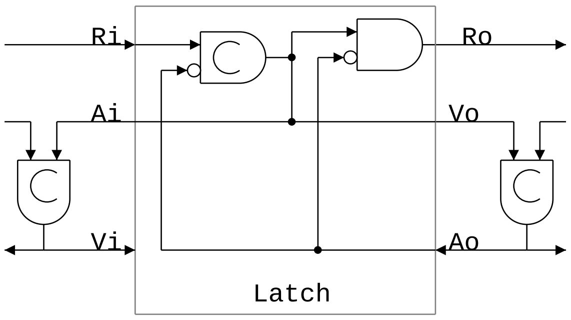

Figure 12: Early output semi-decoupled latch design

Normally semi-decoupled latches require two C-elements but, because we have unbundled the request from the validity lines, there are C-elements outside the latch which can be exploited. Figure 12 shows a design of a semi-decoupled latch making use of the external C-elements and requiring only one internal C-element. The only alteration to the design from the original design in figure 3 is the addition of the AND gate to remove the request signal as soon as the acknowledge signal arrives. The arrival of the acknowledge does not necessarily release the Vo signal and this will stay active until the request input to the latch has been released. This prevents the following stage from starting to compute but does release the request signals to allow stages further down the pipeline to do so.

4. Anti-tokens

Semi-decoupled latches allow early output tokens to flow ahead freely. This removes one of the synchronisation constraints in an asynchronous network. However tokens are still generated and propagated unnecessarily; for example the unwanted input to the multiplexer in figure 5

The remaining step is for a redundant logic input to signal that it is no longer required. This involves propagating information *backwards* along the pipeline. Such signals are designated anti-tokens.

The counterflow network, described in section 2.2, has many properties similar to the early output system. The nodes and pipeline stages are similar in the respect that they collect the ready/valid signal from all inputs using a C-element and generate a fire/request signal by performing some logical function on the request lines of the inputs. The links and latches connect pipeline stages/nodes and propagate the request signals between stages.

By implementing a cross between counterflow circuits and early output logic the resultant system will allow the creation of anti-tokens.

Anti-tokens [9] are like tokens but flow in the opposite direction to the general computation flow. When an anti-token collides with a forward moving token they combine and destroy each other. An anti-token can be dispatched to eliminate a late, approaching input, allowing the stage to move on to its next computation.

By flowing backwards through the pipeline the anti-token moves towards the data source and can do so faster than the normal forward propagation of tokens because there is no need for computation.

4.1. Anti-token latch design

The counterflow network circuit can be recreated in the early output system. The design in figure 13 is visibly similar to that in figure 8.

The counterflow network design has been adapted and used to generate an anti-token latch design. The Ri signal becomes active when sufficient inputs have arrived to a stage. This signal is combined with what would have been the link request in the counterflow network design. This allows the output latch to cause the stage to reset. The output latch does not need any inputs to be present to cause a reset of the stage, so an OR gate is used to allow a stage reset when either the output latch or a sufficient set of input latches have fired.

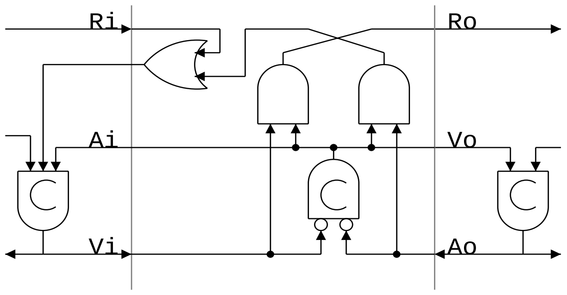

Figure 13: Adapted anti-token latch design

The latch design described above is not optimal and does not fit well with the early output logic system. The latch outputs two signals which need to be attached to the guarding C-element in the input stage. It would be advantageous to implement a latch design which has the same input and output set as the other latch designs. Additionally this design does not have an early-output property. The latch will only pass the request forwards when the input stage has both received the necessary set of inputs and all inputs have presented their validity. In order to generate early output behaviour the Ri line must cause a rise in Ro and Ai.

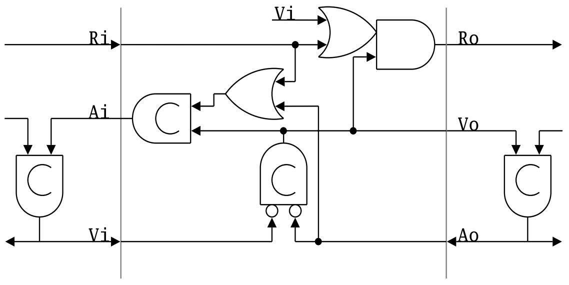

Figure 14: Correct anti-token latch design

Figure 14 shows a optimised design which instead of waiting for the input stage to complete the request out is generated once the request in has arrived and before all inputs to the previous stage have become valid.

4.2. Anti-token operation

The anti-token latches are able to accept anti-tokens by asserting their validity before they have any data. By asserting their validity early they enable the generation of the acknowledge signal. There are two reasons why a latch may receive an early acknowledge, the stage may have generated a result without needing to rely on the presence of the data supplied by the latch or the output latch of the stage propagating an anti-token. This acknowledge signal is then propagated backwards through the latch to the stage feeding it. In this stage the acknowledge signal (Ao) overrides the request in (Ri) and causes an acknowledge in (Ai). If all latches inputting to this stage have raised their valid signals they (either due to the fact that they have data or they are anti-token latches and raise the validity early) the stage will complete and an acknowledgement signal is sent to all inputs. Inputs which have tokens will receive an acknowledge and remove their tokens while the anti-token latches which receive an acknowledgement before data will propagate it backwards.

In figure 15 is the example circuit presented earlier. The stage was unable to complete as only one of the inputs had arrived and this was enough data to generate only one of the outputs. The second output has received an anti-token and can now propagate this to the inputs of the stage. If the input latches were not anti-token latches then they would be protected against receiving an early acknowledge and the stage would not be able to complete. If they are anti-token latches then they will allow the stage to complete and receive an early acknowledge. The anti-token is thus propagated and removes the tokens presented to the stage. Latches which have not presented tokens will accept and propagate an anti-token. This example shows how an anti-token can both be split into many anti-tokens but if the stage had already received all inputs then the anti-token would be removed along with all inputs.

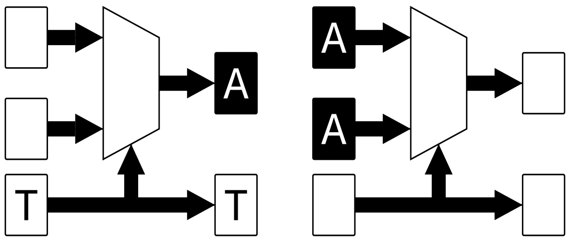

Figure 15: Anti-Token propagation

4.3. Anti-token timing assumptions

The anti-token latches rely on a timing assumption in the system to ensure correct operation. As the Ri line is not sensed in the anti-token passing action the release of this line by all inputs is not noted. The assumption that the delay of the request generation logic (figure 9) is shorter in the reset phase than the delay of the guarding C-element, and the anti-token latch C-element. Although this is easily met with simple circuits it becomes more difficult when more complex request generation methods are used.

A solution to this problem is the use of precharge logic to construct the request generation. This would be reset by the guarding C-element and remove any slow progressing signals in the logic.

5. Conclusions

The generation of early results is a definite advantage to circuit performance with a small overhead cost. This also facilitates the employment of specialist latches which allow more flexibility in system design, boosting performance further. The ability to use a mix of latch designs safely allows the use of the latches that best suit the application.

As pipeline networks become more common, there are an increasing number of external C-elements needed to coordinate responses from different paths. Early output latches can exploit these and thus reduce their internal complexity hence reducing area and power cost.

The removal of input dependencies further desynchronises the system thus may reduce harmonic electromagnetic emissions.

The effects on circuits specifically built to take advantage of anti-tokens would be even greater. Due to the average case performance not being a factor in synchronous designs the average case performance of generic circuits is poor. Designing circuits where the worst case performance is poor but the average case performance is good can be advantageous. Speculatively starting long operations, rather than waiting for the result in order to throw it away, reduces system stalls. This allows the generation of results as soon as possible yet does not take the penalty of an increased average stage timing.

The anti-token system is an effective method of reducing the cost of speculative operations by safely terminating them during their execution. This method having been demonstrated on simple circuits such as small multiplexers becomes more effective on larger and more complex designs. For example, the probability that the required value to the multiplexer will be the last to arrive reduces as the number of inputs increases.

The anti-token systems as presented appear to offer a benefit in asynchronous pipelines. It is possible that large systems can be built on this principle, expanding the scope of asynchronous circuits further.

6. References

[1] J. Sparsø and S. Furber, “Principles of Asynchronous Circuit Design”, Kluwer Academic Publishers, 2001, (ISBN 0-7923-7613-7)

[2] S. B. Furber, P. Day, J. D. Garside, N. C. Paver and J. V. Woods, "AMULET1: A Micropipelined ARM", Proc. IEEE Computer Conference (CompCon’94), March 1994.

[3] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nystrom, Paul Penzes, Robert Southworth, Uri Cummings and Tak Kwan Lee, “The Design of an Asynchronous MIPS R3000 Microprocessor” Proc. 17th Conference on Advanced Research in VLSI, IEEE Computer Society Press, 1997.

[4] C. L. Seitz, "System Timing," in Introduction to VLSI Systems, Addison-Wesley Publishing Company, (ISBN 0-201-04358-0), 1980.

[5] D.E. Muller, “Asynchronous logics and application to information processing”, Switching Theory in Space Technology, Stanford, University Press, Stanford, CA, 1963.

[6] J. Liu, “Arithmetic and Control Components for an Asynchronous System”, PhD Thesis, Dept. of Computer Science, University of Manchester, 1998.

[7] R.F. Sproull, “I.E. Sutherland and C.E. Molnar, Counterflow Pipe-line Processor Architecture”, Sun Microsystems Laboratories Technical Report, April 1994.

[8] C.F. Brej,”Counterflow Networks”, 13th UK Asynchronous Forum, 2001.

[9] C.F. Brej,”Early Output Logic using Anti-Tokens”, Twelfth International Workshop on Logic and Synthesis (IWLS 2003), May 2003