Early Output Logic using Anti-Tokens

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

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

Abstract

Delay-insensitive dual-rail and bundled data design methodologies are the two main approaches used for the creation of asynchronous circuits. Bundled data allows the creation of fast, low overhead circuits, whereas dual-rail allows bit-level pipelining and average case performance. This paper describes 'Early output' logic, which combines the positive features of the two methods to create faster asynchronous circuits. This method allows the creation of circuits yielding performance faster than their synchronous counterparts.

Early output implementations allow logic to evaluate results before all inputs are presented. The results move to the next stage, but the current stage stalls while waiting for the late inputs to arrive simply to acknowledge them. This unnecessary wait can be removed by allowing backwards propagating 'Anti-Tokens' to remove the late inputs. The use of anti-tokens and improved semi-decoupled latches allows the removal of many stalls due to unnecessary synchronisations, thus improving the performance of the circuit.

1. Introduction

Self-timed logic [1] is a form of logic implementation where the timing of operations is implicit in the logic itself. Unlike externally timed logic – which need only evaluate a desired function – self-timed logic must also carry encoded timing information.

Self-timed systems have been classified according to the number of assumptions which need to be made about the timing within them [2]. A Delay Insensitive (DI) system will function correctly with arbitrary delays in any logic or interconnection element; Speed-Independent (SI) systems assume isochronic interconnection; bundled data systems make many more assumptions in modelling the limits of performance and are not considered here.

Dual-rail logic [3] is often used in the creation of DI or quasi-delay insensitive (QDI) systems – the latter having arbitrary computation delays but assuming that a signal fans out to all its destinations at the same time. Dual-rail encoding carries both data and timing using different signals to indicate the arrival of each states. This allows systems to adapt to run at their best speed, although there is some overhead implicit in the detection and acknowledgement of data elements.

In order to make the fastest possible computational systems data must be passed to the next stage as soon as they are ready. By encoding the timing on the data wires, a datapath can be split into very small segments, even down to bit-level pipelining. This allows parts of the data to move to the next pipeline stage while the rest is still being processed.

1.1. Two-Phase Dual-Rail Logic

Two signalling protocols are possible: two phase and four phase [1]. In two phase logic, a transition is created on one wire to transmit a '1' and a transition is created on the other wire to transmit a '0' (fig. 1). This 'transition signalling' minimises switching in the system. Unfortunately, logic retains the previous state of all inputs which introduces overheads into logic gates making them large and slow.

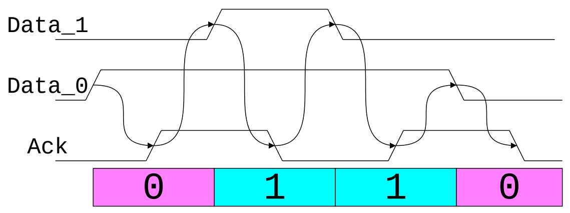


Figure 1: Two-phase dual-rail signalling

Level-Encoded Dual-Rail (LEDR) [4] also uses two-phase, dual-rail transmission, but encodes the data in a different manner. One of the wires always indicates the data value and also the timing when the data changes. If there is no change in the data value a transition on the other 'parity' wire conveys the timing. Each data token has its own parity, either odd or even, which alternates to keep tokens separate (fig. 2). This 'data strobe encoding' is also used in high speed communications such as IEEE 1394 ('Firewire').

Using LEDR it is possible to construct logic much more simply, by using only the 'data' wires to feed the logic part

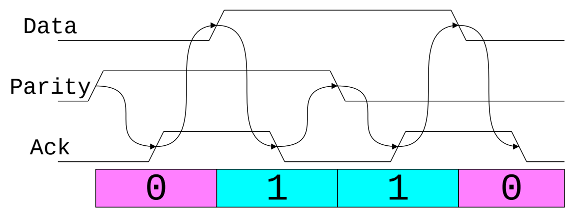


Figure 2: LEDR dual-rail signalling

of the circuit. The timing part will wait for all inputs to arrive and then wait for the matched delay of the logic side before latching the result. This method, due to the use of a matched delay, does not yield a delay insensitive design, but the resultant circuits match and often surpass the speed of their synchronous counterparts. Even better results have been achieved by outputting the result early when waiting for a late input which was not needed to create the result [5]. This can be done, for example, when the first input arriving at an AND gate is a logic '0'; it is not necessary to know the value of later inputs before producing the correct output.

1.2. Four-Phase Dual-Rail Logic

The four-phase dual-rail [3] approach returns both wires to zero after each transaction (fig. 3). This allows fully quasi delay insensitive circuits to be created which do not need to hold state. Delay-Insensitive Minterm Synthesis (DIMS) is the approach often taken to create QDI circuits. It allows logic to be constructed without the need for matched delays. Unfortunately DIMS gates are large, slow and power-hungry (fig. 4). Additionally, the four phase protocol forces each stage to waste as much time returning to zero as it uses to calculate the data. Although DIMS works well in creating bit-level pipelined and average case timed circuit the gates are too slow to compete with other design styles.

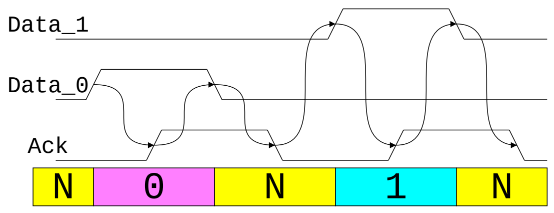


Figure 3: Return to zero signalling

There are four problems with DIMS which disqualify it as a method to create fast circuits.

1) Latency: the use of C-elements (for synchronisation) is very expensive, especially in three or more input gates.
2) Size: compared with a standard gate, the DIMS gate is ten (or more) times the silicon area.
3) Worst case behaviour: each gate waits for all of its inputs to arrive before creating the result. This is enforced even if the inputs that have arrived are sufficient to calculate the result.
4) Throughput: the return to zero (RTZ) phase takes as long as the active phase. This ensures that at most only half the circuit is operating at one time.

In some of the following descriptions the 'spacer' between data elements is referred to as 'NULL', a term borrowed from NCL [6]; this does not imply that this system uses NCL gates however.

2. Early output logic

Early output logic [7] addresses some of the problems in DIMS logic to create fast circuits whilst preserving many of its beneficial properties.

Many of the problems in the DIMS approach are caused by the gates trying to manage the timing as well as the logic of the function. By separating the timing and logic parts of the circuit (and losing the 'QDI-ness'), a faster circuit can be created, as demonstrated with LEDR. The communication protocol remains QDI and timing assumptions are only applied in local logic.

DIMS gates are large and slow due to the C-elements required to ensure the output only rises when all inputs are valid and falls only when they both return to NULL. If these restrictions are moved into separate guarding logic the gates can be much smaller and faster.

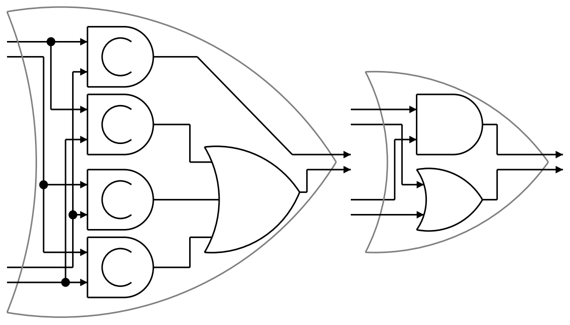


Figure 4: DIMS and early output OR gates

Effectively all the gate is now required to do is to execute the logical operation on the dual-rail inputs. Figure 4 shows an early output OR gate compared with its DIMS counterpart. The delay through the early output gate is equal to that of a single gate stage.

DIMS OR

	0	N	1
0	0	N	1
N	N	N	N
1	1	N	1

Early Output OR

	0	N	1
0	0	N	1
N	N	N	1*
1	1	1*	1

Figure 5: DIMS vs early output transition tables

Figure 5 shows the output of DIMS and early output OR gates during transitions from the NULL state. The early output gate outputs early in two cases (marked with a *).

Although this is beneficial because the result arrives at its destination sooner it does not ensure that all inputs have arrived.

A DIMS gate:

1) outputs NULL when all inputs are NULL.
2) executes the required logical operation.
3) only outputs a valid value when all inputs are valid.
4) only returns to NULL when all inputs are NULL

The early output only has properties 1 and 2, so some other mechanism must be provided to ensure correct operation.

2.1. Guarding

Property 3 (above) ensures that only when all inputs into a stage are valid will the result become valid. In an early output logic system this is undesirable. Instead the requirement is merely that all the inputs must have been asserted before they can be acknowledged.

Figure 6 shows an example of an early output pipeline stage. The thick, grey symbols indicate the logic circuits which carry implicit timing information; the output latches will capture data and acknowledge the input (Ai) as soon as it is available. The logic may produce a result ‘early’; for example if the upper input latch sources a ‘0’, both output latches receive ‘0’ and can acknowledge this.

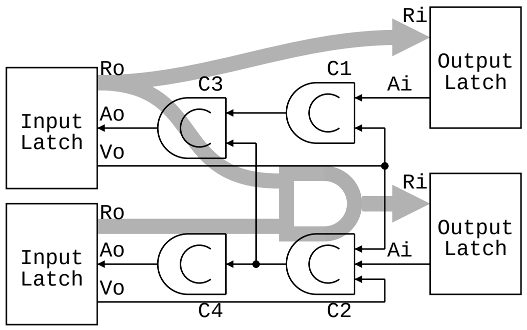


Figure 6: Guarding example

Two banks of C-elements are used to guarantee operation. The first C-elements (C1 and C2) are adjacent to the output latches; these ‘guarding’ C-elements pass on an acknowledgement when all the contributing input stages have output valid data, as indicated by Vo (e.g. see fig. 9). In the example above C1 will fire but C2 will wait for the lower input latch to assert data. C1 and C2 reflect the fan-in of their respective output latches.

The second bank of C-elements ensures that fan-out constraints are met for their respective input latches. C3 therefore waits for the acknowledgement from both outputs, whereas C4 waits only for the lower output.

The C-elements appear cumbersome but, in many cases (such as C4 in fig. 6), they elements may be degenerate and can be removed. In other cases optimisations are possible (e.g. C1 and C3 can be combined making the Vo input of C1 redundant).

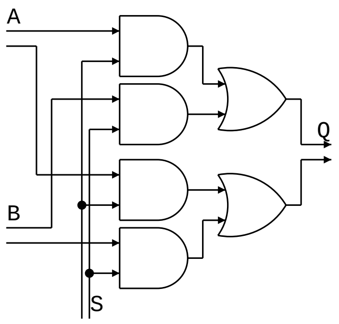
These C-elements also ensure that all relevant latches achieve a NULL state between data values (property 4, above). There is a potential hazard introduced here in that late-arriving (unnecessary) data will begin to traverse the logic in parallel with its ‘valid’ signal triggering its removal. It is therefore necessary to ensure that this ‘runt’ data does not both survive and have a sufficient delay for it to reach the output latch after the other inputs have returned to NULL. In this circumstance a false data packet could be introduced. However the timing constraints on this appear to be fairly easy to meet. Indeed in some logic families – such as dynamic logic – this is met automatically.

2.2. Weak condition logic

Early output logic is a member of the weak conditioned logic family [8]. All examples of this design style need to ensure the state of the inputs before acknowledging their source. This is commonly done by ensuring the validity of all outputs before acknowledging, a process which may entail the adding of extra logic in order to ensure the state of every input is visible at the logic output. This method allows *some* outputs to become valid early and removes the need for guarding C-elements. Unfortunately it also synchronises all signals at the stage’s input. Early output logic is not only more general but later sections will show how to take advantage of the guarding C-elements to improve performance further. Additionally, in early output logic, many stages are able to acknowledge some inputs while other parts of the stage are still processing.

2.3. Early output states

Early output gates may output a value before all inputs are valid. This increases the speed of computation. These early output states allow the result to move forward while



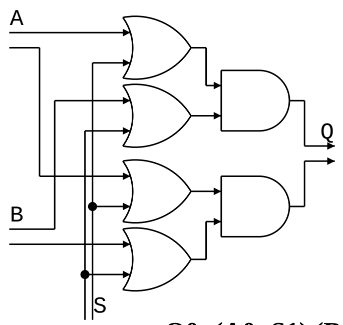
		S		
A	**B**	**N**	**0**	**1**
N	N	N	N	N
0	N	N	0*	N
1	N	N	1*	N
N	0	N	N	0*
0	0	N	0	0
1	0	N	1	0
N	1	N	N	1*
0	1	N	0	1
1	1	N	1	1

Q0=(A0.S0)+(B0.S1)
Q1=(A1.S0)+(B1.S1)

Figure 7: Early output multiplexer design

current stage waits for all inputs to arrive before acknowledging them. A good early output design will use as many of these early output cases as possible. Figure 7 shows a design for a dual-rail 2:1 multiplexer along with its truth table. Although the design is correct it does not capture all early output states.

By rearranging logic it is often possible to create circuits which capture more of the early output cases. Such circuits will stop and wait for late inputs less often. Sometimes an optimal circuit may be uneconomic, but in other cases it can be quite straightforward. Figure 8 shows an improved multiplexer design which includes two extra early output cases. This circuit will be able to create a valid output if both data inputs are equal and thus the select input is irrelevant.



		S		
A	**B**	**N**	**0**	**1**
N	N	N	N	N
0	N	N	0*	N
1	N	N	1*	N
N	0	N	N	0*
0	0	0*	0	0
1	0	N	1	0
N	1	N	N	1*
0	1	N	0	1
1	1	1*	1	1

$$Q0=(A0+S1).(B0+S0)$$
$$Q1=(A1+S1).(B1+S0)$$

Figure 8: Improved early output multiplexer

3. Semi-decoupled latches

Although gates can output results early, guarding logic ensures that the pipeline stage waits for the late inputs before acknowledging. During this time the data on the output is valid and must remain so at least until the last input arrives. This stops a subsequent stage from moving more than half a cycle ahead; it can evaluate with the input just fed to it but, until the stalled stage is freed, the subsequent NULL cannot be generated.

In principle a latch between stages could generate a NULL when its output is acknowledged, allowing the subsequent pipeline to complete and recover. This latch would have to wait until the input has returned to zero before continuing to pass data. These are the properties of a semi-decoupled latch.

Figure 9 shows the standard dual-rail latch design [9] adapted for use in early output logic. The OR gate is used to both provide an acknowledge backwards and a validity forwards. Figure 10 shows a commonly used dual-rail semi-decoupled latch [9]. In this latch the data output will return to NULL once the acknowledge signal reaches it, even if the input data has not entered the NULL state. The

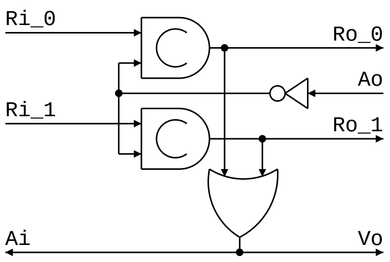


Figure 9: Standard dual-rail latch

component is more complex than the standard dual-rail latch as it requires a C-element to store some control state. The state holding C-element fires when the output signal is acknowledged. It remains active until the data C-elements have returned to zero.

The semi-decoupled latch allows pipelines to operate with fewer internal synchronisations. Unfortunately, the cost in performance and area of the above design is high.

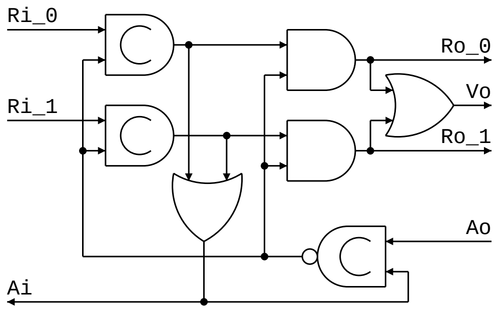


Figure 10: Standard semi-decoupled latch

3.1. Early output semi-decoupled latch

A cheaper semi-decoupled latch is desirable. All latches used by early output circuits described in this paper output a validity signal. The Vo (Valid out) line is connected through C-elements to the Ao (Acknowledge out). This allows the data signals to return to NULL but the Ao line will remain active until Vo has been released. This stops the stage from completing the acknowledge but later stages can complete the cycle and start a new one.

Using early output logic it is safe to force data to NULL and still ensure the stage does not complete the acknowledge phase. Figure 11 shows a much cheaper semi-decoupled latch design. The cost of the two AND gates is very low: due to the sequencing of the transitions.

Data out lines (Ro_0 and Ro_1) are driven high when the C-elements switch high and low when the acknowledge becomes high. The AND gate can therefore be implemented using only two transistors: a P-type pass transistor, to propagate the data when the acknowledge is low, and an N-type transistor to force the signal to ground when the acknowledge is high.

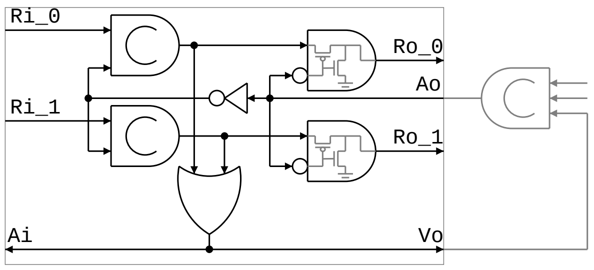


Figure 11: Early output semi-decoupled latch

4. Anti-Tokens

With a block of logic circuit such as an OR gate, if one input to the gate in a given operation is ‘1’, all other inputs can be ignored when they arrive. By extension if a message can be sent backwards, the incoming token can be eliminated *before* it arrives, potentially both speeding up the operation and reducing power consumption. This leads to the idea of counterflow pipelines, where data ‘tokens’ flow in the forward direction and ‘anti-tokens’ flow the opposite way. An anti-token may be generated when a data input becomes redundant and it can then travel back to eliminate the token (and itself) closer to the data’s source.

Perhaps the best known example of an asynchronous counterflow pipeline is the Counterflow Pipeline Processor (CFPP) [10] which allows instructions to move one way along a processing pipeline while results flow freely in the opposite direction. This addresses the issue of inter-instruction dependencies in a processor; a complex problem in that it is unknown how many times (if any) a result will be used in the near future. In the CFPP various complex interactions between packets flowing in opposite directions are possible; counterflowing packets are synchronised if they collide within a stage so that interactions can be managed. This requires arbitration to guide the actions at each stage of the pipeline.

Anti-token propagation is much simpler; tokens and anti-tokens cannot pass each other, they can only collide. The decision making process is therefore much simpler and – it is believed – can be satisfied by meeting a set of timing constraints.

4.1. Protocol

In early output circuits the ‘valid out’ signal (Vo) delays an incoming acknowledgement signal until the latch is ready to accept it. It is asserted when the data is ready and will both delay an early acknowledgement until data is present and prevent the acknowledgement from being released until the latch has returned to NULL.

A latch which is able to process anti-tokens can assert the valid signal *before* outputting data. If an acknowledgement then arrives an anti-token has been received and can be captured. This can then be passed backwards by asserting the input acknowledge before data has been received. As a latch can only hold one anti-token it has to prevent the following stage from giving it another early acknowledgement. This is done by simply not raising the valid line again.

A latch can raise its acknowledge signal even before it receives any data as the acknowledgement will not reach the input latches until they all raise their valid lines. A latch holding an anti-token can acknowledge early but has to keep the acknowledge high until all inputs assert their valid lines and the guarded acknowledge has activated. This is because the latch cannot sense if the stage moved to the reset phase by only observing the data lines. The data lines never went high so the latch can’t wait for them to return to zero.

To allow the latch to snoop on the state of the previous stage the guarded acknowledge line is fed into the latch as Vi. This allows the latch to send and receive early acknowledgments.

This type of anti-token latch can inter-operate with the other latches described earlier. For instance a semi-decoupled latch will never assert its validity before it has data and will thus not accept an anti-token, even if it is offered. The responsibility of eliminating the data token then remains the responsibility of the downstream latch.

4.2. Anti-tokens and logic

Anti-tokens can move backwards through logic as well as FIFOs. Figure 12 shows a situation where an anti-token (A) has arrived at a logic stage where some of the inputs have arrived. In such a case, if the remaining latches which have no output data are anti-token latches, they will receive an early acknowledge and accept an anti-token. The latches with data will receive an acknowledge and reset to NULL.

Early output logic will generate anti-tokens automatically. The example in figure 13 shows a multiplexer with only the data inputs valid. As shown in section 2.3, a 2:1 multiplexer can be designed so that, in some cases, it can create a valid output before the select signal arrives. Once the result arrives at the output latch, it can then acknowledge. If the acknowledgement is allowed

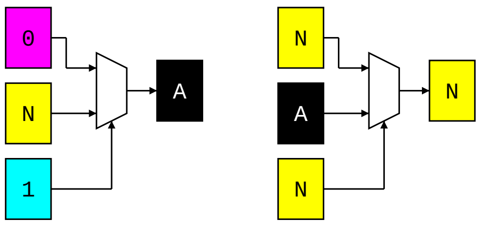


Figure 12: Anti-token propagation through logic

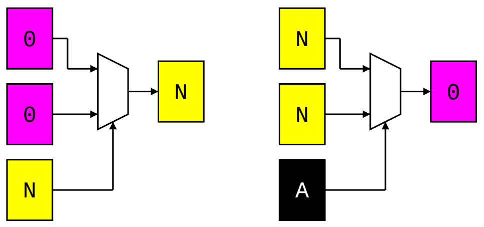


Figure 13: Anti-token generation

through the guarding logic (select input is 'valid', i.e. able to accept an anti-token) then latches which do not hold data will receive anti-tokens while the latches which do will receive a normal acknowledgement.

4.3. Anti-Token pipeline

Figure 14 shows an example of an early output circuit. In this circuit the two data C-elements are abstracted and are outside the latch to simplify the description. This C-element is driven by signal S from the latch control unit. The C-element is asymmetric because, while passing an anti-token, the 'S' signal is withdrawn. This could cause the C-element to become metastable (if data is just arriving) but the data state will be ignored. 'S' forces the C-element reset when it drops. The Ai signal is combined with Vo signal to create the guarded acknowledge. This can then be passed to the input latch as the Ao and to the output latch as Vi. The latch also snoops on Ri and Ro signals. The Ri signal is observed to ensure that the data C-elements are not re-enabled while there is still data from the previous operation.

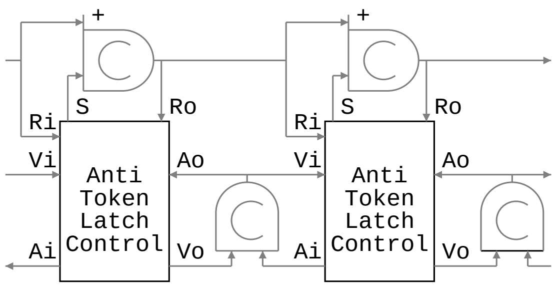


Figure 14: Anti-token FIFO

Figure 15 shows a Burst-mode machine description of the latch controller. State 0 is the initial state, in which outputs S and Vo are high while Vo and Ai are low. In state 0 the latch is waiting for the sign of a token or an anti token. If the Ro+ transition happens first the latch will go through states 1, 2 and 3 while passing a token. If Ao+ arrives first then the latch will pass an anti-token by going through states 4 and 3.

4.4. Anti-token latch behaviour

Figure 16 depicts the operation of the anti-token latch when passing a token. In state 0 both the S and Vo signals

$$\begin{tikzcd}
2 \arrow[r, "Ro-Vi+/Ai-Vo-"] & 3 \arrow[d, "Ao-Vi-/S+Vo+"] & \\
 & & 4 \arrow[ul, "Vi+/Ai-Vo-"'] \\
1 \arrow[uu, "Ao+/S-"] & 0 \arrow[l, "Ro+/Ai+"] \arrow[ur, "Ao+/Ai+S-"'] &
\end{tikzcd}$$

Figure 15: Description of the anti-token latch Burst-mode machine

are high, so both the data and the guarding C-elements are ready to fire. In this example the data C-element will fire first and raise Ro (i.e. the data signals). The latch then passes a token as would a standard dual-rail latch. The only difference is that removing the S signal directly forces Ro to NULL, even if Ri is still valid. Instead of waiting for Ro to drop, Ri is tested directly to check that it has returned to NULL before releasing the acknowledge. This gives the latch a semi-decoupled behaviour.

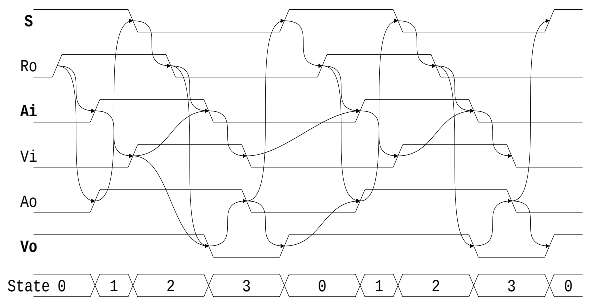


Figure 16: Token passing

Passing anti-tokens is simpler than token passing. In figure 17 the latch receives the Ao+ transition first. Ao+ causes S to be withdrawn. (During the anti-token pass transitions on S are not observed, so the circuit is not QDI.) Once S is low data trying to enter the latch will be ignored. If Ri rises just before S drops (due to accepting an anti-token) a short glitch on the data output (Ro) may be caused; if S and Ri change at the same time the C-element could even become metastable. However, as the C-element is asymmetric its output is forced inactive by S. The latency of the logic through which this glitch could propagate must be shorter than the reset cycle time to ensure the data cannot 'leak' past the anti-token.

Providing this timing constraint is met, arbitration can be avoided as both the two initial transitions (Ro+ or Ao+) will have the same effect (Ai+). The anti-token pass is equivalent to the token pass with the exception that during a token pass the latch waits for the data lines to drop before continuing. As the data lines remain low during an anti-token pass, no stall is required.

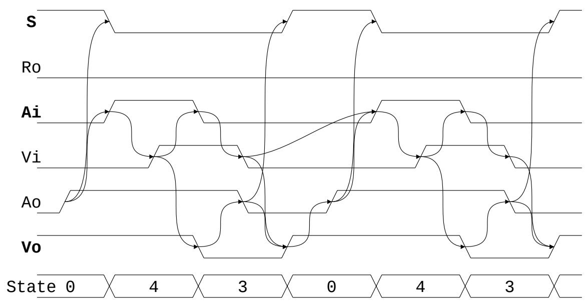


Figure 17: Anti-token passing

4.5. Anti-token latch schematic

Figure 18 shows a schematic of the latch, synthesized by Minimalist [11] and then hand optimised. The circuit is only slightly larger than the standard semi-decoupled latch and retains the semi-decoupled property.

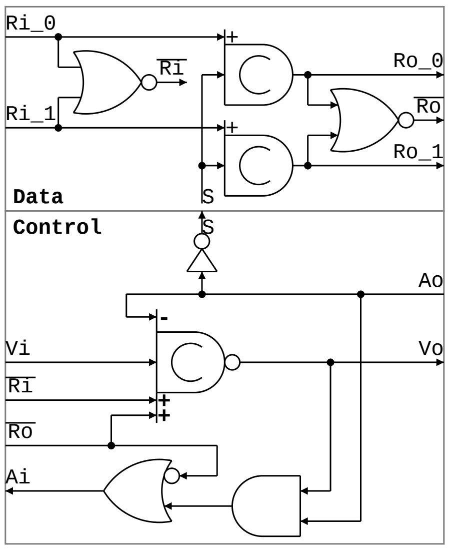


Figure 18: Anti-token latch schematic

4.6. Anti-token/token collisions

In each other's absence tokens and anti-tokens may flow freely. When they collide they eliminate each other, therefore there is no decision to make as to which reached a latch first. An anti-token begins with an acknowledge so the token 'believes' that it is being acknowledged, and vice versa. The latch sourcing the token passes it on, but the subsequent latch 'perceives' the request as an acknowledgement of its anti-token. Timing constraints ensure that the data payload does not survive.

Although the problem of deciding whether to pass a token, pass an anti-token or destroy both would seem to require an arbiter, by creating an interface in which the behaviour is the same in all three situations no arbitration is necessary. Either side of the latch can initiate the cycle but both sides need to resynchronise in order to complete it.

5. Results

Large circuits have been created using early output logic with encouraging results. The largest example was a 5 stage, 32-bit microprocessor core [12]. The pipeline was created using early output logic although the register bank was made using matched delays. Anti-token latches have been tested in this circuit, but not yet used to full effect.

Because each latch must collect the acknowledge signals from all latches it outputs to, in large designs this requires C-elements with hundreds of inputs. Large fan-in and fan-out causes the circuits to be very large and synchronised. To remove the need for large C-elements the design is split up into small parts of only a few gates with pipeline latches. Pipeline latches may be latches of any of the types described above. Effectively the design becomes super-pipelined. If the pipeline latches are placed vertically (e.g. across carry lines in an adder) the design also becomes bit-pipelined. Individual bits will progress to the next stage while others are being processed. In long series of additions the data wave-front will become skewed allowing the bottom bits of the adder to process the next set of inputs while the top of the adder is processing data from the previous instructions.

Additional pipeline latches were added to the microprocessor design but without more information as to where to place them, only intelligent guesses could be made as to where they would give a positive effect. Additionally, only standard latches were added as the effect of different latch designs was not fully explored.

The first attempt to create the design using a synthesized register bank (tried initially) required 1000+ input C-elements, due to dependencies on all of the bits in the 32 entry, 32-bit register bank. Replacing the register bank with a matched delay version and hand placing pipelining latches gave much more positive results. The synchronous design had a worst case delay path of about 80 gates. The asynchronous version on fast instructions (e.g. r:= 0 + 0) had cycle times of about 60 gate delays. Slow instructions (e.g. r:= 0 - 0) could take up to 90 gate delays but as they occur very rarely the average cycle time is much lower than the synchronous version.

The result were measured in gate level simulations against a very naive synchronous design which used a 64 gate delay carry ripple adder.

5.1. Comparisons of latches

Figure 19 shows the gate delay count of all latches shown. The units are measured in gate delays of the optimised versions of the latches. C-elements require 2 gate delays to switch and the optimisations remove the need to invert the Ao line.

The early output, semi-decoupled latch gives much improved results over the standard semi-decoupled latch. Ao↓ to Vo↑ delay is two times smaller and the overhead over the original design is so low that it seems beneficial to use the early-output latches throughout most designs. The anti-token latch is only marginally slower and, in the case of Ao↓ to Vo↑, is actually faster than any other latch. The only problem with using anti-token latches as standard is their physical size.

From	**To**	**Original**	**Standard S-D**	**Early output S-D**	**Anti-token**
Ri_?↑	Ro_?↑	2	3	3	2
Ri_?↑	Ai↑	3	3	3	4
Ri_?↑	Vo↑	3	4	3	N/A
Ao↑	Ro_?↓	2	2	2	2
Ao↑	Vo↓	3	4	3	5
Ao↑	Ai↑	N/A	N/A	N/A	2
Ri_?↓	Ai↓	3	3	3	4
Ri_?↓	Vo↓	3	N/A	3	4
Ao↓	Ro_?↑	2	5	3	2
Ao↓	Vo↑	3	6	3	2
Token Pass		14	20	16	19
Anti-Token Pass		N/A	N/A	N/A	16
Transistor count		24	42	28	41

Figure 19: Transition delays

6. Conclusions

Early output logic looks very promising when compared with synchronous designs for speed. It is important to remember that although the speed improvement might be sought after, the area and power consumption costs are high.

Early output logic is the basis of features such as improved semi-decoupled latches and anti-token latches. Although these latches look beneficial, further work is needed to show where they should be placed in a design to gain a positive effect.

Early output circuits require some timing assumptions in the logic part of the circuits. This can be easily met if the logic functions are fewer than four inversions deep. This restriction requires very fine grain pipelining but this is the intended target for these techniques. A method of finding these timing requirements and reorganising the logic, placing delay lines or adding extra inputs to guarding C-elements is required to allow a more flexible designs.

Anti-token latches offer further improvements. For example these could be used extensively in a processor to signal that a register forwarding path is not required, thus removing unnecessary synchronisation in a self-timed system. Like other early output circuits certain timing criteria must be met, but it is believed that these constraints should not be too limiting in most circumstances. These developments are being incorporated into a larger asynchronous system where it is hoped they will give significant extra flexibility in future asynchronous designs.

7. References

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

[2] K. Van Berkel, F. Huberts and A. Peeters, “Stretching Quasi Delay Insensitivity by Means of Extended Isochronic Forks”, Proceedings of the Second Working Conference on Asynchronous Design Methodologies, London, UK, 1995.

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

[4] M.E. Dean, T.E. Williams and D.L. Dill, “Efficient Self-Timing with Level-Encoded 2-Phase Dual-Rail (LEDR), Advanced Research in VLSI, 1991.

[5] R. B. Reese, M. A. Thornton and C. Traver, “Arithmetic Logic Circuits using Self-timed Bit-Level Dataflow and Early Evaluation”, Proceedings of the 2001 Conference on Computer Design, September 2001.

[6] K.M. Fant and S.A. Brandt. “NULL conventional logic: A complete and consistent logic for asynchronous digital circuit synthesis”, International Conference on Application-specific Systems, Architectures, and Processors, 1996.

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

[8] Christian D. Nielsen. “Evaluation of Function Blocks for Asynchronous Design”, Proceedings of EURODAC, 1994.

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

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

[11] R. Fuhrer and S. Nowick, “MINIMALIST: An Environment for the Synthesis, Verification and Testability of Burst Mode Asynchronous Machines”, Department of Computer Science, Columbia University Technical Report, 1999.

[12] C.F. Brej, “Third year project report”, 2001, http://www.cs.man.ac.uk/~brejc8/yellow_star.html.