A 2-Phase Asynchronous Event Driven Buffer with Completition Detection Signalling

L. Lloyd, A. V. Yakovlev, A. M. Koelmans
Department of Computing Science,
University of Newcastle upon Tyne, NE1 7RU, England.

January 15, 1997

Abstract

A design is presented for the implementation of an event driven asynchronous buffering mechanism which functions within a delay-insensitive two-phase handshake environment and performs in a manner similar to that of a synchronous 'tri-state' driver element.

The design is derived from a series of currently available asynchronous logic elements that perform latching functions from within either a two-phase or a four-phase operational medium. A solution to the problem of the correct generation of completion signals from within such a two-phase functionality is presented.

The final buffer design is verified using both a Petri Net and Signal Transition Graph analysis with the resultant circuit realisation being simulated using the WARP VHDL Development System.

Keywords: Asynchronous Logic, Petri Nets, Signal Transition Graphs, VHDL Simulation.

1 Introduction

The past decade has seen an almost exponential growth of interest into the asynchronous logic design methodology [22]. Almost all of the asynchronous devices that have actually been realised in silicon and that have utilised the *delay-insensitive model* have been implemented using a combination of both *two-phase* and *four-phase* transition signalling [12] [1] [3].

This paper presents a design for an asynchronous buffering element that is to function within a purely two-phase signalling environment. The requirement for a logic element of this nature has grown out of the research at the University of Newcastle concerning the development of structures whose functionality lies within the two-phase transition signalling domain. Most recently the ADLX [15] was the production of a high-level asynchronous specification of the synchronous DLX microprocessor [13] with the resultant implementation being verified via the use of various Petri Net modeling tools. Continuing work at Newcastle is now involved with the production of a complete asynchronous circuit description of the ADLX with the aim of this implementation task being two fold; in the first case a model of a complete two-phase asynchronous microprocessor will be gained that will allow the simulation and analysis of such a design to be carried out. In the second case the justification, performance and applicability of tools being developed at Newcastle for just such an implementation task can be evaluated and assessed.

The investigation into wholly two-phase asynchronous logic has led to the development of the buffering mechanism described in this paper. The development of this buffer has led to the subject of the completion signalling of such logic (see sections 2 and 4) and some answers to some general questions as raised by such completion signalling will be discussed there.

The remainder of this paper will be set out as follows; Section 2 is concerned with the various design models that are present within the main asynchronous design methodology. Particular emphasis will be placed on the uses and limitations of employing a two-phase transition signalling protocol within the Delay-Insensitive domain. In Section 3 the nature of the problem that led to the requirement of the buffer is described to be followed by the complete development process that led to the final buffer design. Section 4 addresses some of the problems of the generation of completion signals from within two-phase asynchronous structures. Section 5 presents, via the use of a Petri Net and Signal Transition Graph analysis, the modeling of the internal configuration of the buffer and the complete signalling state changes that may occur within the resultant buffer design. Section 6 will simulate the circuit realisation of the buffer through the use of the WARP VLSI Development System[19]. Section 7 concludes the paper.

2 Delay Insensitive Circuits

The way to approach a discussion of the various asynchronous design methodologies is by examining the different asynchronous design models that can be implemented. In Figure 1 the main design hierarchy can be seen and the five main timing models can be described briefly as follows;

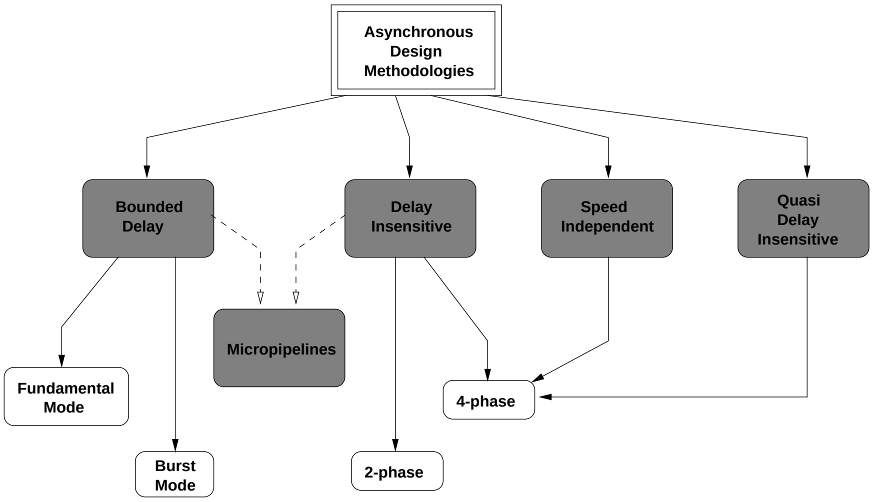

Figure 1: Asynchronous Design Hierarchy.

- **Bounded-Delay :** these are models which are similar in nature to synchronous designs. In any bounded-delay circuit the delays associated with all logic elements and wires are known or are assumed to be within some finite limit.

- **Delay-Insensitive :** these models assume that there is no bounded delay associated with any logic element or wire.

- **Micropipelines :** these are a hybrid of both the bounded-delay and delay-insensitive models. Specifically they are composed of a bounded-delay datapath that is regulated by a delay-insensitive control circuit.

- **Speed-Independent :** these models assume that all gate delays are unbounded whilst all wire delays are insignificant (all wire delays are less than the minimum gate delay).

- **Quasi-Delay-Insensitive :** these models use delay-insensitive arguments suplemented by *isochronic forks (forking wires have equal delays to specific destinations).*

As the design of the event driven buffer is based on the delay-insensitive methodology a fuller description of that timing model will be given here. For a complete description of the asynchronous design heirarchy shown in figure 1 see [18].

The fact that a delay-insensitive circuit knows nothing about the delays associated with any logic element or wire raises the issue of how any such circuit can determine when a new input has been correctly received. The solution to this problem ensures that any recipient of input informs the sender when such data has been accepted through the use of a specific *completion* signal. The senders task during this interchange is to ensure that any output is held stable until that completion signal is detected. This communication of control information, the act of one logic element informing another logic element that they may commence computation, is performed using the following protocol;

- a request transition is forwarded from a sender to a receiver.

- an acknowledge transition, generated by completion logic, is forwarded to the sender.

This specific protocol is termed *two-phase handshaking* and is illustrated in figure 2.

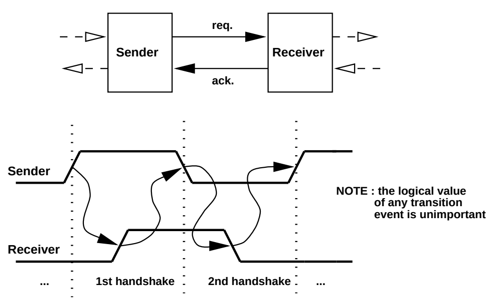

Figure 2: Two-phase handshake protocol.

A drawback to the use of the two-phase handshake is how to ensure the correct transmission of data. When using transition signalling the opposite to a transition (i.e. no transition) cannot be distinguished from a signal that is simply delayed. In a pure two-phase protocol two wires are therefore required between a sender and receiver when an exchange of a single data bit is reqiured.

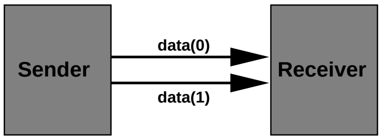

Figure 3: Two-phase handshake (transition signalling).

Figure 3 illustrates how a transition on the wire **data(0)** could inform the receiver that a data bit equal to zero had arrived whilst a transition on **data(1)** could inform that a data value of one had arrived. Using this method of data transmission (known as *dual rail encoding*) is costly in the number of wires that must be employed within a design. A modification therefore to the dual-rail model is to use the *bundled-data* protocol. In this case a single wire is employed for the transmission of each data bit (violating the delay-insensitive model) whilst a single control wire regulates each data bit or word, see figure 4.

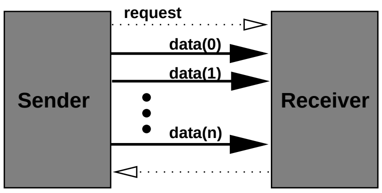

Figure 4: Bundled-data protocol.

The assumption in the bundled-data case is to ensure that the delay of the control wire (the request signal) will be longer than the delays of any of the data wires. This means that when the receiver detects the control signal that receiver will know the data has already arrived.

An extension to the two-phase protocol is *four-phase handshaking* where this is simply the employment of an extra set of phases that are used to reset the control wires to some initial request state. Four-phase handshaking may mean slower data transmission phases are employed but this can lead to simpler circuit designs than their equivalent two-phase counterparts.

The use of completion and transition signalling within the delay-insensitive model enforces good asynchronous circuit design. Completion signalling will allow the average-case[1] behaviour of circuits whilst removing hazards. Transition signalling on the other hand is used to indicate when processing of data is to begin thus allowing for the faster transmission of data[2].

[1]the logic can latch a result immediately after processing instead of waiting for the worst case delay.

[2]since there are no idle or reset phases

3 Design of the Buffer Element

3.1 Problem Statement

The design of the ADLX microprocessor embraces a micropipelined organisation that employs two-phase handshaking, see figure 5;

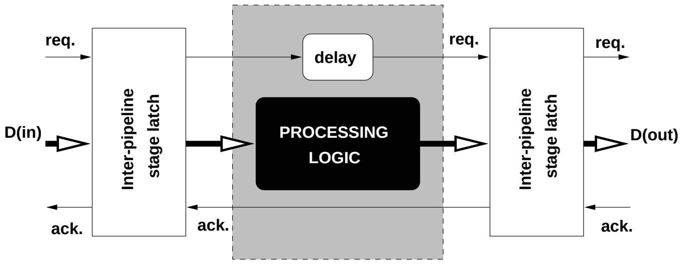

Figure 5: Two-phase micropipeline example.

Data latching within the ADLX is explicitly carried out at the inter-pipeline stage latches which are effectively the boundary interfaces between the various pipeline stages. Data latching within the processing logic was found to be unnecessary (except in specific cases) when often all that was required was a simple buffering mechanism that would either delay a signal or direct a signal to a specific output path, figure 6;

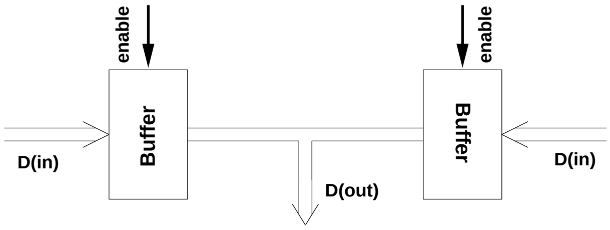

Figure 6: ADLX datapath conflicts.

In a synchronous circuit design (or an equivalent four-phase asynchronous design) the solution to any datapath conflict problem would be to employ the use of tri-state drivers, figure 7;

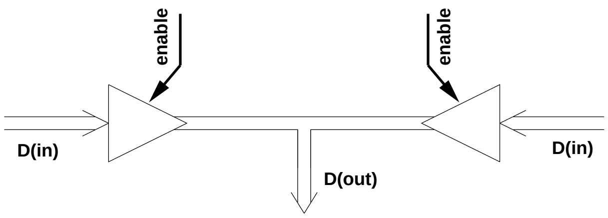

Figure 7: Synchronous tri-state drivers.

Tri-state drivers though could not be employed within the ADLX as they are *level-based*[3]

[3]the activation of a logic element on receipt of a specific logical input value

elements. To comply with the two-phase protocol an *event-driven*[4] mechanism was required.

3.2 A simple solution to the event driven buffer problem

An effective method of solving the event driven buffering problem is to employ a Sutherland transparent latch [8] that has been initialised to be in the *capture* state. In figure 8(a) a transparent latch is seen in the transparent or *pass* mode. To set such a latch into the capture mode initialisation may simply involve providing the complement of one of the initialisation signals to that latch (depending upon the specific logic implementation) in order to realise the switch configuration as seen in figure 8(b).

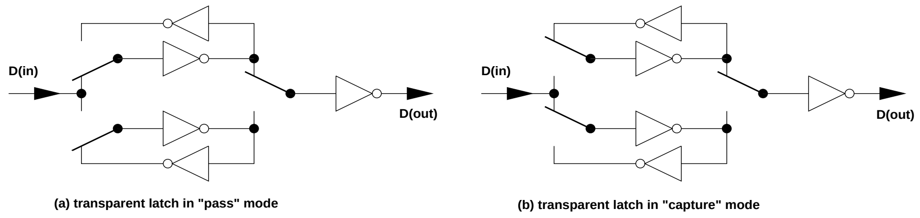

Figure 8: Implementations of a Sutherland transparent latch.

Any subsequent transition that then enabled the latch would cause a sequence of operation as follows;

$$\textbf{pass} \Rightarrow \textbf{capture} \Rightarrow \textbf{pass} \Rightarrow \textbf{capture} \Rightarrow ...$$

which would perform the exact operation as required by an event driven buffer.

There are two arguments though against the using of this type of buffering mechanism;

- the buffer will be a latch and latches are not generally required within the processing logic of the ADLX.
- there are significant *timing hazards* that can arise from the use of these latches due to the problem of how to generate correct process completion signals (this problem is explicitly detailed in section 4).

3.3 Solutions to the buffering problem that currently exist

By staying with a latching element approach it is possible to examine other solutions to this buffering problem. The AMULET1 microprocessor [7] used a variety of latches in different configurations to provide various buffering constraints. In figure 9 a typical micropipeline transparent latch can be seen;

[4]transition signalling.

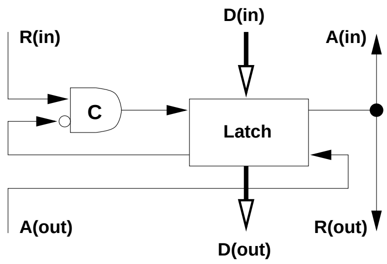

Figure 9: Typical transparent latch implementation.

and the AMULET1 implementation of this latch can be seen in figure 10. This particular latch used four-phase control, specifically to eliminate timing hazards but also to cut down on the number of transistors employed in the design[5]

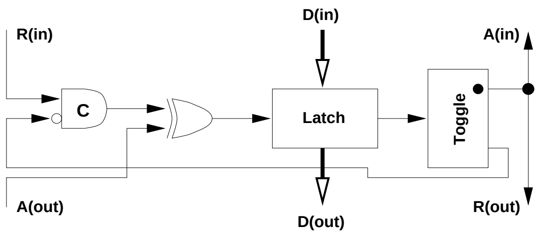

Figure 10: AMULET1 transparent latch.

A modification to the above latch was a configuration in which the latch is initialised to be in a *blocked* or the capture state, see figure 11;

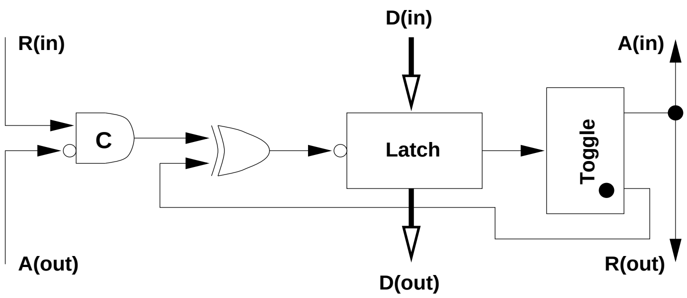

Figure 11: AMULET1 blocked latch.

and was specifically used to prevent signals from being forwarded down through a pipeline.

The latches illustrated in figures 10 and 11 used four-phase control signalling. These latches were found to be expensive in the number of control wires that were required, especially when the data path was particularly wide, and so alternative latch designs were considered. One such idea was the development of an *event latch* which was enabled using transition signalling [5], figure 12;

[5]six transistors as opposed to Sutherland's twenty-four.

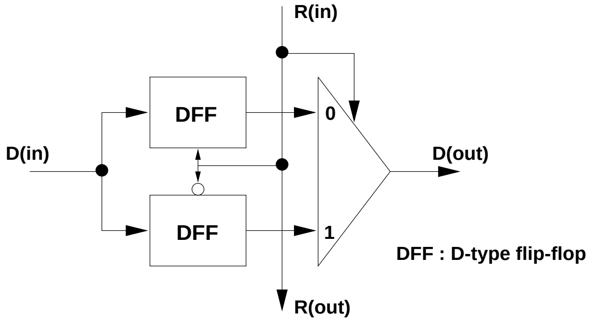

Figure 12: AMULET1 event latch.

In the stable state this latch has one DFF opaque and one DFF transparent with the multiplexor providing the output from the opaque latch. This device did provide an event driven latching mechanism but was costly in the number of transistors used in implementation[6] and so was only employed in certain areas within the AMULET1 design.

3.4 Derivation of an event driven buffering mechanism

As a starting point for the derivation of an event driven buffer we can again consider the transparent latch mechanism shown in figure 8(a). As there is no requirement to store a data value within such a buffer the configuration of such a buffer can be illustrated in figure 13;

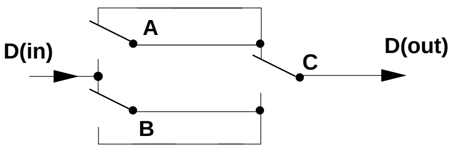

Figure 13: Initial event buffer design.

To construct this mechanism the switch circuitry, labelled **C** above, must be modified in order to realise the switch configurations labelled **A** and **B** . If the Sutherland transparent latch circuit is considered, figure 14;

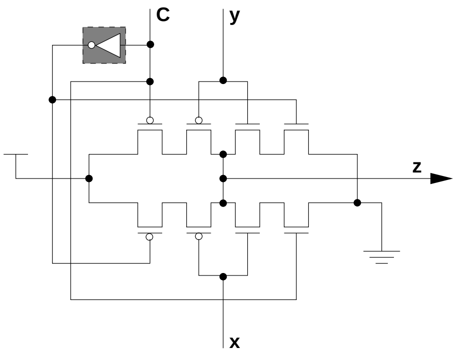

Figure 14: Sutherland transparent latch switch mechanism.

[6] approximately 60 transistors, see [6]

that has a diagramatic notation;

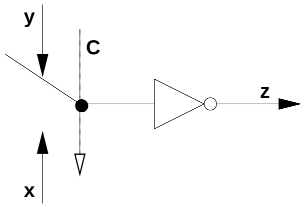

Figure 15: Initial switch configuration.

and as the inverted output is not required for the switches **A** and **B** , e.g.;

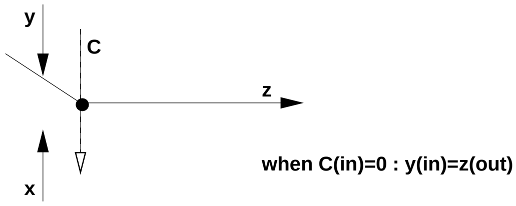

Figure 16: Revised switch configuration.

the following circuit structure can be derived, see figure 17;

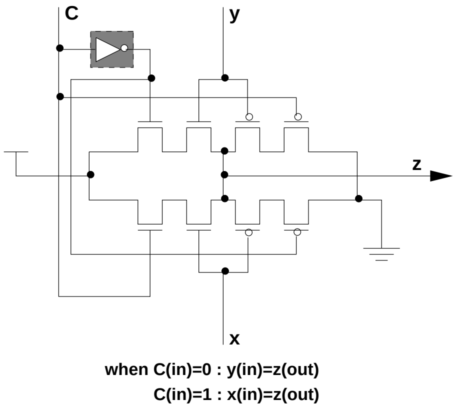

Figure 17: Revised switch implementation.

Using three modified switches allows the event buffer shown in figure 13 to be realised. One point to notice about this buffer is that the complement enable signals necessary for the correct operation of this device will be generated from within the buffer itself (see shaded sections, figures 14 and 17)[7].

[7]In all cases in this paper when a count is given of the number of transistors in the design of the buffer mechanism the number of transistors required for the generation of the enable complement signals will also be included. This exclusion of a complement enable transistor count is a feature that is often prevalent amongst many contemporary latch designs, partly because there may actually be a seperate mechanism to generate such signals but also because it is relatively trivial. Excluding such logic allows for the reporting of low transistor counts in latch designs.

Figure 13 is one possible implementation of the event buffer but the number of transistors requires, 30 in total, is excessive for such a simple logic device. The initial design can be simplified to use only two switches, see figure 18 below;

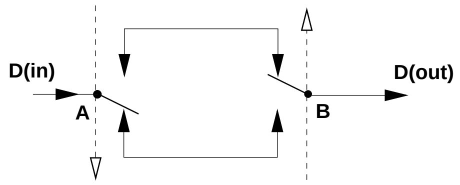

Figure 18: Refined event buffer design.

By modifying the switch design in figure 17 the switch labelled **A** above can be implemented as follows;

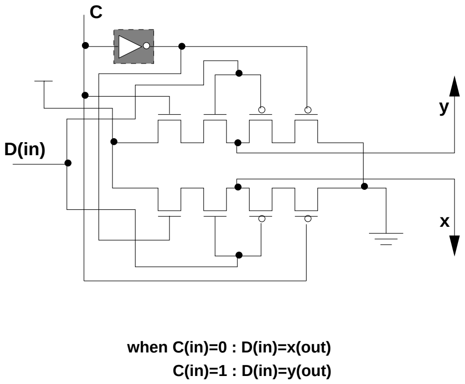

Figure 19: Switch **A** for the revised event buffer design.

and so using one Sutherland transparent latch switch and one switch from figure 19 the refined buffer above can be realised using a total of 20 transistors.

There is a further refinement that can be made to the design of the event buffer. The internal configuration of an AMULET1 two-phase transparent latch using transmission gates and having a count of 18/22[8] transistors, can be seen in figure 20;

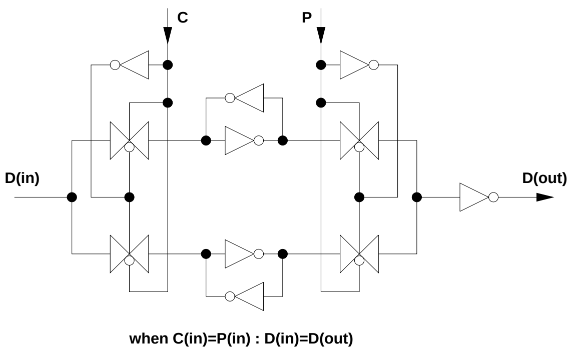

Figure 20: AMULET1 transparent latch implementation.

The design above can be directly translated into the event buffer design shown in figure 18

[8] with/without complement enable signal generation

by the following actions;

- removal of the internal latches.
- inversion of the **C** enable logic (to achieve the switch configuration labeled **A** in figure 18).

and can be illustrated in figure 21 below;

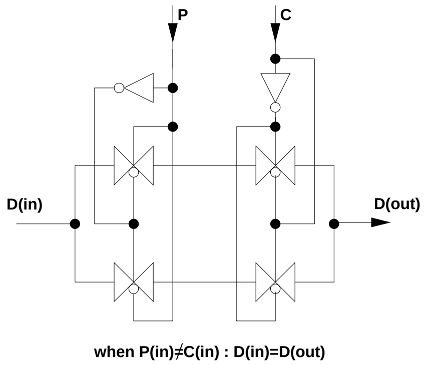

Figure 21: Final event driven buffer design.

The final design of the event driven buffer uses a total of 8/12 transistors and is given the diagramatic notation;

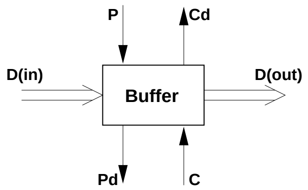

Figure 22: Event driven buffer notation.

4 Completion Detection

4.1 The problem of hazards

In order to describe completion detection in a delay-insensitive asynchronous circuit it is necessary to examine the problem that requires such a mechanism to need to be employed. Consider the asynchronous circuit below;

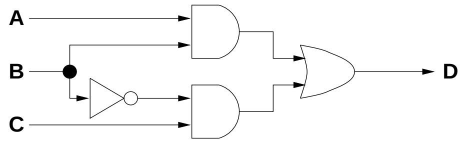

Figure 23: Example asynchronous circuit.

where each gate element is considered to have a delay time of 1. If at time **t=0** the input to the circuit, **I(A, B, C)**, is **(1, 1, 1)**, then at **t=3** the circuit output, **O(D)**, will be **(1)**.

If the input to this circuit is now changed, e.g. **I(A, B, C)** = **(1, 0, 1)**, there is the possibility that an intermediate hazard stage can be reached, for example;

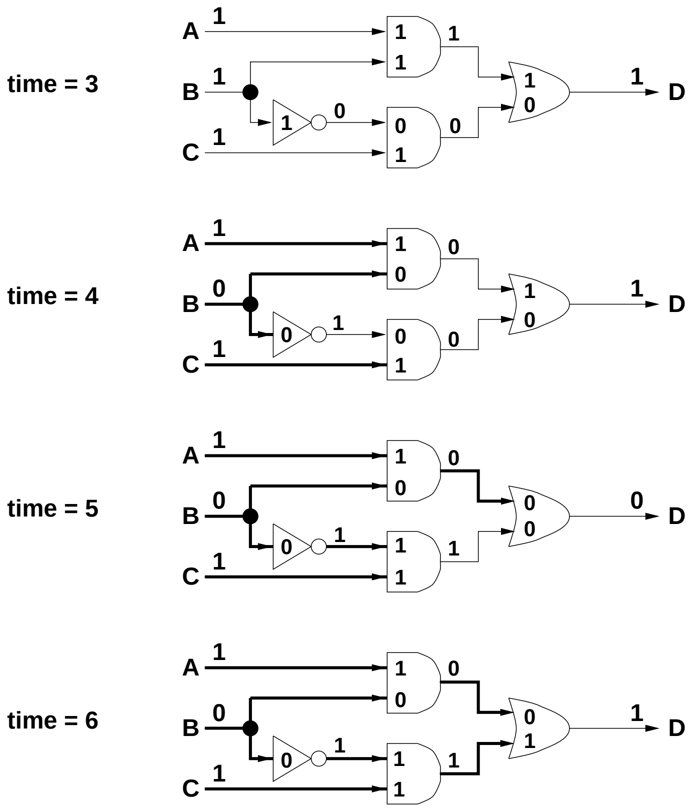

Figure 24: Hazard example.

At **t=5** it can be seen that there is a brief change of output of the circuit when **O(D)** = **(0)**. In a synchronous clocked circuit this brief change may be considered insignificant. If that synchronous circuit had a clock cycle that was active at **t=3**, **t=6** etc. then the hazard at time **t=5** would never be seen. This hazard though could not be ignored in either a two-phase or a four-phase asynchronous framework as the output change could actually be interpreted as a transition event;

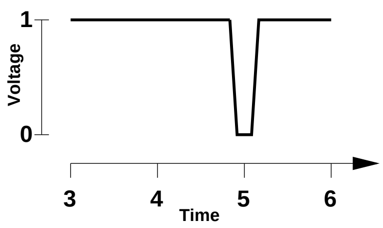

Figure 25: Circuit hazard output.

Asynchronous circuits must be able to deal with these hazard problems.

4.2 Hazard solution 1: A bounded-delay approach

In a bounded-delay circuit the delays due to all wire and logic elements will be known to be within some finite limit. In order to produce a *completion signal,* t_{CD}, from a particular logic block the generation of that signal can be carried out after the *maximum time,* T_d, that that logic is known to function for, see figure 26;

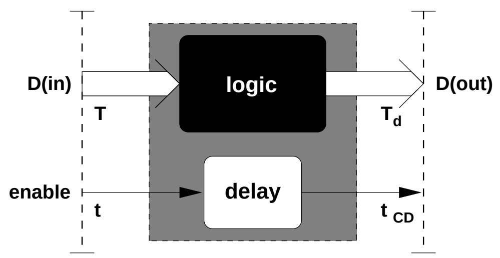

Figure 26: The bounded-delay hazard solution.

where;

$$T_d = \{x : x \in \Re^+ \, and \, l_{min} \leq x \leq l_{max}\}$$
$$t_{CD} = \{x : x \in \Re^+ \, and \, x \geq l_{max}\}$$

when;

$$l_{min} : minimum \, time \, for \, processing \, within \, the \, logic \, block$$
$$l_{max} : maximum \, time \, for \, processing \, within \, the \, logic \, block$$

A specific example of this delay mechanism occurs within the event latch as illustrated in figure 12. There is an assumption made that after a finite period of time the output from the opaque DFF latch will become stable. This assumption is supported by employing specific delay logic to control the output that is produced from the multiplexor, see below;

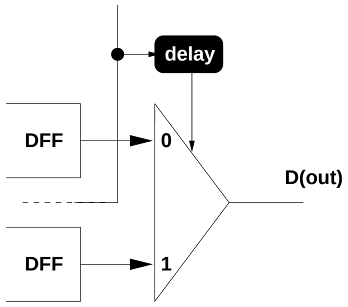

Figure 27: The event buffer delay solution.

4.3 Hazard solution 2: A delay-insensitive approach

In a delay-insensitive circuit the correctness of function of any operation will be insensitive to delays associated with any logic element or wire [9] [21] [17] [4]. For such a circuit to function correctly there must be employed some *completion detection logic* , (CDL), that is used to determine when that circuit has completed all processing actions, e.g;

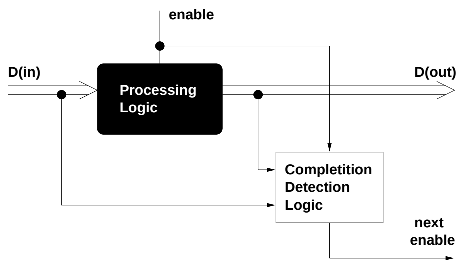

Figure 28: Example of completition detection.

The CDL will act as a *synchronisation mechanism* which when using the above example will perform the function;

A subsequent enable signal will be generated provided that;

- the current enable signal (a transition event) and
- the data output from the processing logic

are both stable and have been detected by the CDL.

With regards to the event driven buffer the difficulty arises as to how any CDL would determine that fact that there had been no change of input of data value even though a change of enable signal had been detected. A data value that is delayed for an indefinite period of time is not distinguishable from no change of data at all. The solution to this problem is straightforward providing that the circuit behaviour of the event buffer is such that;

- the **D(in)** value will **always** arrive at the buffer before the **P** enable signal (for a single sequence of operation of the buffer).
- any CDL must be allowed to detect, enter some pre-defined state (depending on whether or not there has been a change of logical value of **D(in)**) and stabilise before the arrival of the **P** enable signal.

The completition detection circuitry that would be necessary to ensure a delay-insensitive operation of the event buffer can be seen in figure 29 below. In figure 29 there can be seen two main additions to the logic of the event buffer, a transmission gate and an XNOR element.

- the transmission gate is designed to be transparent when the enable signal for that gate has a logical value of 1. This transmission gate is specifically employed to prevent the propagation of any **Pd** signal until any change of input of data has been made stable.
- the XNOR element is designed to act as a synchroniser which detects that the input of data has the same logical value as the output value as provided by the event buffer. When these two values are logically equal the XNOR provides a signal that is used to enable the transmission gate.

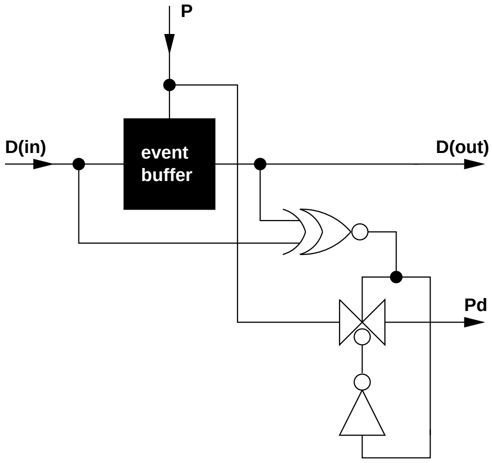

Figure 29: A delay-insensitive event driven buffer.

Using the above logic allows for three cases to be detailed which define the operation of the event buffer CDL mechanism.

case 1: initial buffer state

The initial state of any event buffer will have a configuration as illustrated below;

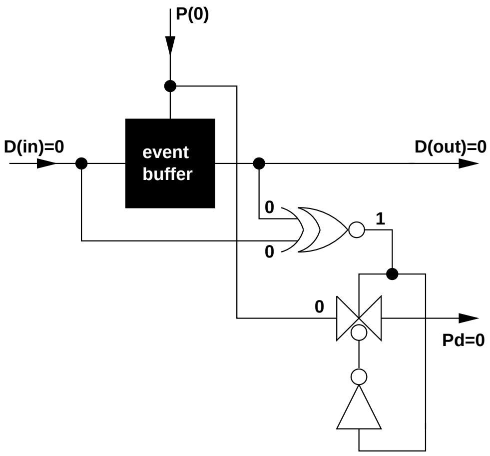

Figure 30: Event buffer initial state.

The signals **D(in)** , **P** and **C** (see figure 21) will be initialised to 0. As the event buffer is in the *capture* mode in the initial state (**D(in)** will not be propagated through the buffer)

D(out) must also be specifically set to 0. **D(in)** and **D(out)** will thus cause the XNOR to output a value of 1. This XNOR output is used as a preset value which sets the **P/Pd** gate to be transparent.

case 2: no change of input data

When there is no change of input data to the event buffer the **P/Pd** gate will be set to transparent (due to a previous data change made stable or from the initial state). The **P** enable will then arrive at the event buffer and will be forwarded to the **P/Pd** gate which can then immediately pass that **P** enable as the **Pd** signal. Effectively what is occuring here is that there is no delay in the generation of the **Pd** signal when no data change occurs.

case 3: change of input data

When there is a change of logical value of the input to the event buffer this new data value will be forwarded to the event buffer and to the XNOR element. The XNOR element will use both the current and the new values of the buffer data to produce an output value of 0. This new output of the XNOR will then be used to set the **P/Pd** gate to the *blocked* state, e.g;

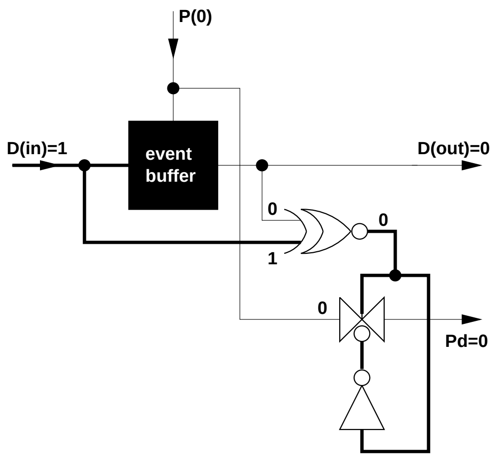

Figure 31: Blocking the **P/Pd** gate.

When the above circuit is stable, e.g. the **P/Pd** gate is closed, only then will the **P** enable signal arrive[9]. The **P** enable will be forwarded to the event buffer and to the blocked **P/Pd** gate;

[9]the mechanism for ensuring the delay of the **P** signal is not discussed here - it may simply be the case that the natural topology of the surrounding environment of the event buffer will provide a significant enough delay of the **P** signal

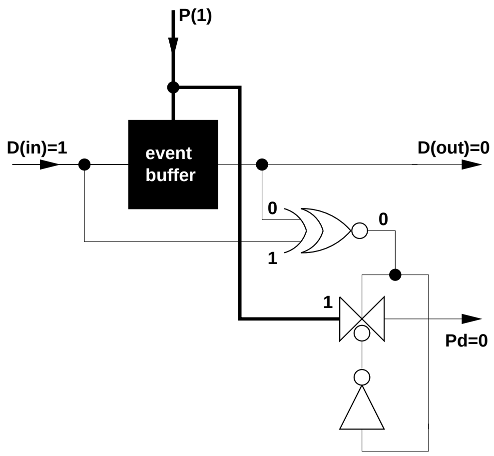

Figure 32: Arrival of the **P** enable signal.

The enabling of the event buffer will now allow the new data value to propagate through the buffer logic. This new data value will be presented at **D(out)** and at the XNOR element. Both inputs to the XNOR element will now be the new data value and so the XNOR will produce an output of 1. The new output from the XNOR is now used to set the **P/Pd** gate to transparent thus allowing the forwarding of the **Pd** signal.

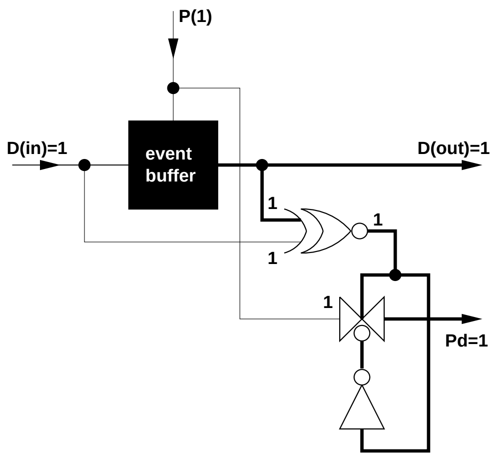

Figure 33: Forwarding the **Pd** signal.

A final note describing the operation of the event buffer is concerned with the **C/Cd** signals that are used to reset the event buffer to a blocked state. No CDL is necessary for the controlling of this action as the sole purpose here is to *block* the event buffer as soon as possible after the arrival of the **C** enable signal.

5 Verification of the Buffer Element

The event driven buffer can be modeled and verified by using a combination of both Petri Net and STG analysis. In the case of the Petri Net modeling the internal configuration of the event buffer will be examined (figure 21) to be followed by the modeling of the CDL mechanism (figure 29). The STG analysis will involve the modeling of the transitions that represent the change of values of circuit signals belonging to the buffer CDL mechanism.

5.1 Petri Net analysis

A Petri Net is a graphical tool that can be used for the simulation of both dynamic and concurrent systems. Any Petri Net is a tuple (representing a graph) such that P = (S, T, F, M_0) where; S = the set of vertices that represents the state components of the graph, T = the set of transitions (or actions) that can be preformed, F = the flow relation, defined as $F \subseteq (S \times T) \cup (T \times S)$ (both S and T are finite disjoint sets) and M_0 = the initial state marking of the graph, defined as $M_0 \subseteq S$. For a fuller description of the semantics of Petri Nets see [20].

The Petri Net model of the internal configuration of the event buffer can be seen in figure 34 and the corresponding Petri Net model of the event buffer CDL mechanism can be seen in figure 35;

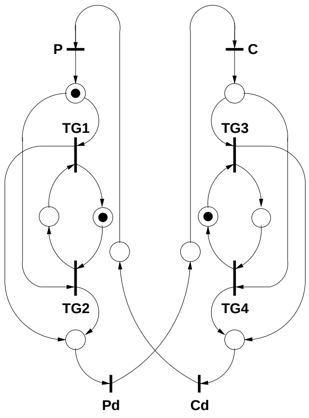

Figure 34: Petri Net model of the event buffer in the initial state.

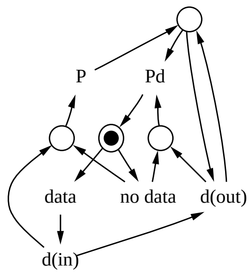

Figure 35: Petri Net model of the event buffer CDL mechanism.

5.2 Signal Transition Graph analysis

A Signal Transition Graph, STG, can be described as a *Labelled Petri Net* . Any STG is a tuple such that G = (P, X, Z, Δ) where; P = a marked Petri Net, X and Z are disjoint sets of input and output signals, Δ = the labels of each transition of P, defined as $\Delta : T \to (X \cup Z) \times \{+, -\}$. For a fuller description of the semantics of Signal Transition Graphs see [2] [14].

The complete set of input transitions that will result in a change of output state of the event buffer can be seen in table 1 below;

transition	data	enable	data	enable	data	enable	data	enable
current state	0	0	0	1	1	0	1	1
next state	0	1	0	0	0	1	0	0
current state	0	0	0	1	1	0	1	1
next state	1	1	1	0	1	1	1	0

Table 1: Complete set of event buffer output states as derived from all input states.

which translates into the state diagram shown in figure 36;

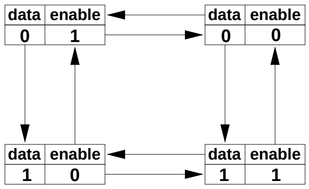

Figure 36: Event buffer state diagram.

The resultant Signal Transition Graph that describes the behaviour of the event buffer has been resynthesized using a PETRIFY [11] [10] analysis and can be seen in figure 37.

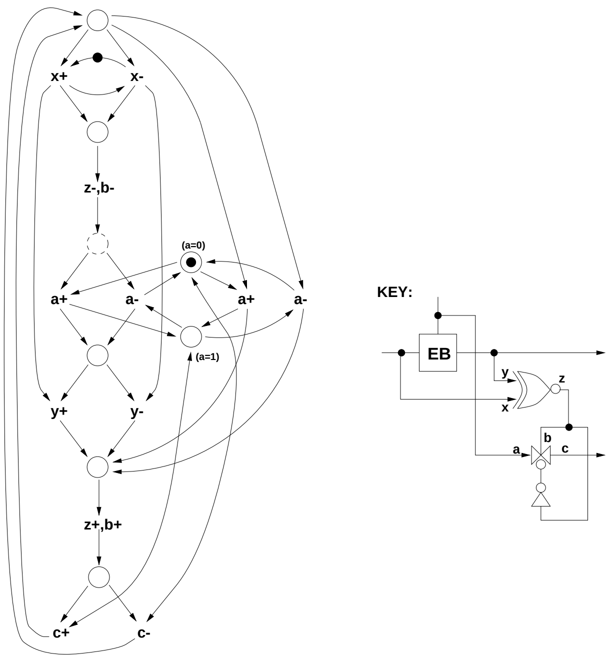

Figure 37: STG for the event buffer.

NOTE : Figure 37 above shows one place drawn with a dashed line. This place is an artificial construct with the purpose here being to represent some timing constaint that is required for the correct operation of the event buffer (see case 3, section 4.2).

The complete state graph for the event buffer can be seen in figure 38 below;

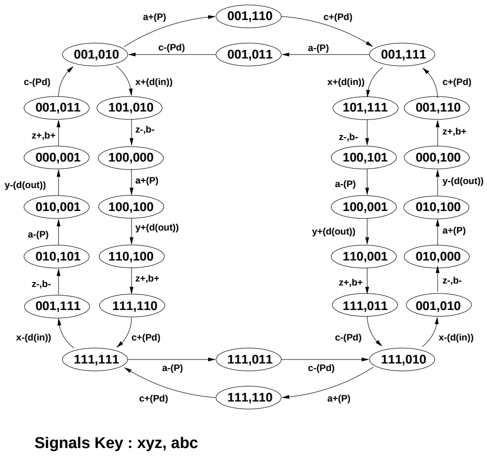

Figure 38: State graph for the event buffer.

6 Simulation of the Buffer Element

The development of the 2-phase ADLX microprocessor has involved the use of the event driven buffer in several areas of the main design. One such area is at the program counter incrementer unit, see figure 39 below;

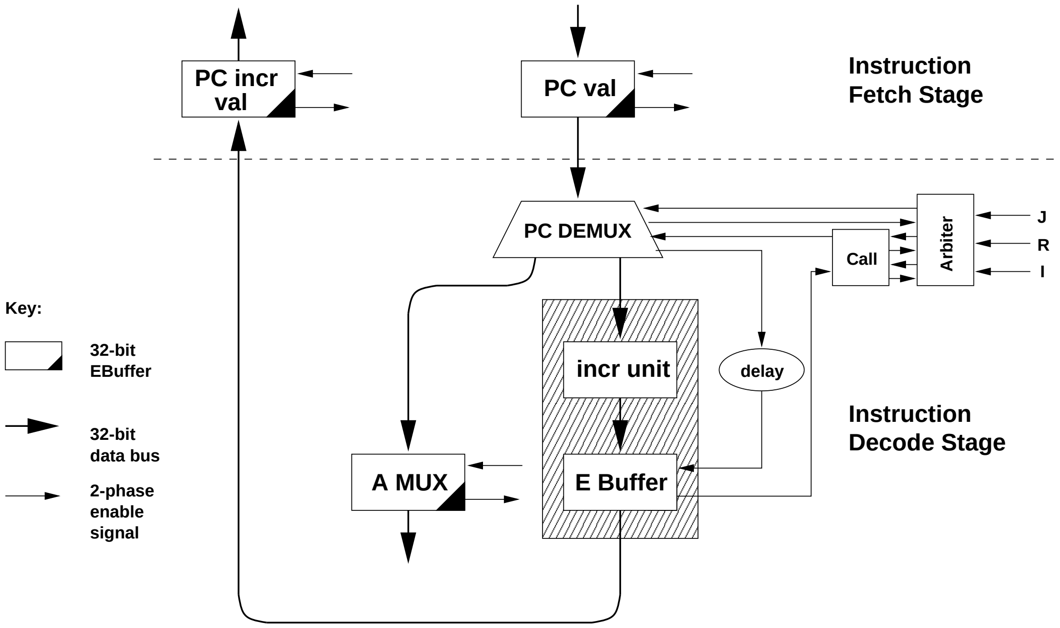

Figure 39: ADLX program counter incrementer unit.

The PC incermenter unit is designed to take a 32-bit program counter value and add to this a fixed value of four. The new value of the program counter that is calculated is then forwarded back to the instruction fetch stage to be used as an address for the retrieving of the next instruction from memory.

In order to simulate the operation of the event buffer, the incrementer unit, as per the shaded region in the figure above has been taken and modeled in VHDL. This particular model was reduced from a 32-bit data path to an 8-bit data path for ease of modeling and viewing of simulation results. The tool used to perform this simulation has been the Cypress Semiconductor WARP VHDL Development System which is primarily designed to be used for the implementation of FPGA ASIC circuits. WARP is currently under investigation at Newcastle with regards to providing a facility for the fast development of FPGA ASIC designs and also as an undergraduate teaching tool.

In figure 40 three test cases showing the operation of the incrementer unit can be seen that have been executed using the Cypress Semiconductor NOVA Simulation package:

test case 1: initialisation

The first transition of the **en_in** signal outputs the sum of the input, 0, with the value of the hardwired incrementer giving a total of 4. In this first instance the CDL mechanism can be seen to be performing correctly as no **en_out** signal is propagated before any change in output[10].

The second transition on **en_in** shows again that when there is no change of input there is no change of output. Both of these transitions from low ⇒ high and high ⇒ low describe the operation of the incrementer unit when there is no change of data.

test case 2: change of data input (increasing and decreasing)

The third transition of **en_in** allows the value of 174, (10101110), to be summed with the increment value and then propagated through the event buffer giving a total of 178, (10110010), being seen at the output. Again the CDL functions correctly as no **en_out** is propagated until all changes of data have been made stable.

The fourth transition of **en_in** sees a reset of the input data to 0 and so again the output from the event buffer is a total of 4. A general comment about the actual incrementer unit can be made here. In usage the incrementer will essentially count "up", barring branch address calculations which are carried out elsewhere. Our example here shows that the incrementer and CDL can cope with both increasing and decreasing input data values.

[10]remember that when there is no change on d(in), and thus no change on d(out), the propagation of any **en_out** cannot occur until the output from all the event driven buffers is stable. In this case the only change in output occurs on the **k** line.

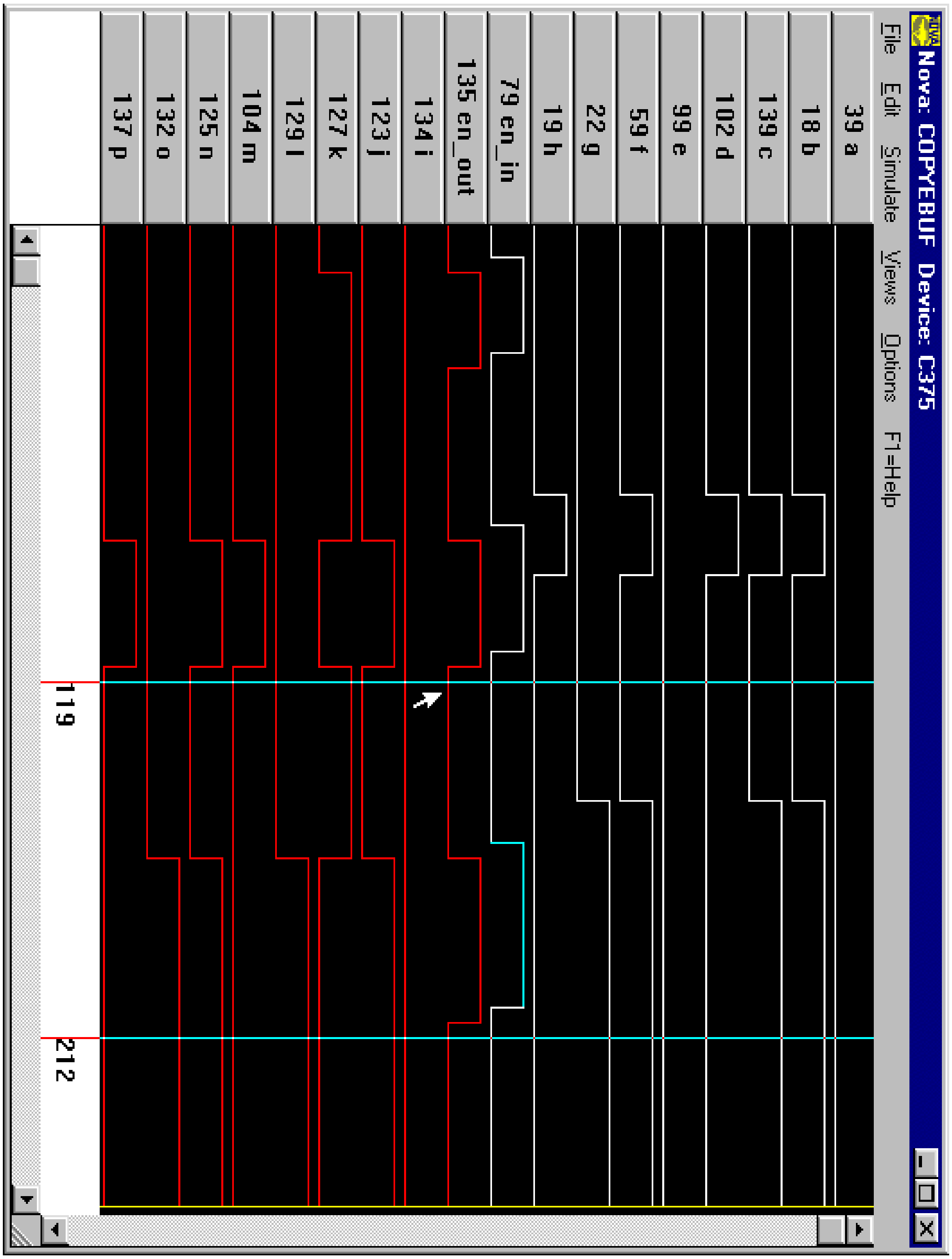

Figure 40: NOVA simulation of the incrementer unit.

Key to figure 40
signals, **a** to **h** are data input (old PC value)
signals, **i** to **p** are data output (new PC value)
signals **en_in** and **en_out** respectively the enabling and propagated signals

test case 3: change of data input (increasing and no change)

The fifth transition of **en_in** allows the input value of 102, (01100110) to be summed with the increment value giving an output total of 106, (01101010).

The sixth transition of **en_in** sees no change of the data input value of 102 and so again the value at the output is 106. Again in both cases there is no **en_out** propagated before any data change.

The simulation of the event driven buffer has shown that the design functions as specified over a range of operational conditions. The CDL mechanism has been shown to be correct and to perform exactly as would be required in a 2-phase transition signalling environment.

The VHDL source code listing for the event driven buffer design can be found in Appendix A.

7 Conclusions

The design, verification and simulation of the event driven buffer has shown that useful 2-phase asynchronous logic elements can be developed and successfully analysed using a wide range of both synchronous and asynchronous design tools. The event buffer itself is used in almost all of the major logic structures in the ADLX microprocessor and has been shown to perform, through the WARP simulation, as according to the design specifications.

Concerning the tools used, Petrify was employed in the construction of the STG model of the completition detection mechanism of the event buffer whilst the WARP VHDL tool provided the simulation of the complete event buffer design. WARP is designed to be used primarily for the development of synchronous FPGA ASIC circuits but in this case performed well in the execution of the event buffer simulation.

8 Acknowledgements

Thanks must go to Keith Heron for discussions about the general nature and types of environment in which such an event buffer may be expected to operate.

References

[1] Davis A. **Synthesising Asynchronous Circuits: Practice and Experience** . In Birtwistle G. and Davis A., editors, *Asynchronous Digital Circuit Design*, pages 104–150. Springer-Verlag, 1995.

[2] Yakovlev A., Lavagno L., and Sangiovanni-Vincentelli A. **A Unified Signal Transition Graph Model for Asynchronous Control Circuit Synthesis** . In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 104–111. IEEE Computer Society Press, November 1992.

[3] Furber S. B., Day P., Garside J. D., Paver N. C., Temple S., and Woods J. V. **AMULET 1: A Micropipelined Arm** . In *Proc. IEEE Computer Conference (COMPCON)*, pages 476–485, March 1994.

[4] Ebergen J. C. **A Formal Approach to Designing Delay-Insensitive Circuits** . *Distributed Computing*, 5(3):107–119, 1991.

[5] Farnsworth C. **Micropipeline Approach to Design Examples** . Technical report, Department of Computer Science, University of Manchester, England, 1993. Funded by: ESPRIT project EXACT (6143) and the ESPRIT Basic Research Working Group 7225 - ACiD-WG.

[6] Farnsworth C. **Amulet Group Low Power Cell Library Data Sheets** . Technical report, Department of Computer Science, University of Manchester, England, 1995. part of the OMI-EXACT Project.

[7] Paver N. C. **The Design and Implementation of an Asynchronous Microprocessor** . PhD thesis, Department of Computer Science, University of Manchester, England, June 1994.

[8] Sutherland I. E. **Micropipelines** . *Communications of the ACM*, 32(6):720–738, June 1989.

[9] Unger S. H. **The Generation of Completition Signals in Iterative Combinatorial Circuits** . *IEEE Transactions on Computers*, C-26(1):13–18, January 1977.

[10] Cortadella J., Kondratyev A., Kishinevsky M., Lavagno. L., and Yakovlev A. V. **Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers** . In *Proc. 11th Conference on Design of Integrated Circuits and Systems (DCIS'96), Barcelona*, pages 205–210, November 1996.

[11] Cortadella J., Kishinevsky M., Lavagno L., and Yakovlev A. V. **Synthesizing Petri Nets from State-Based Models** . In *Proc. Int. Conf. on CAD (ICCAD'95), San Jose, CA.*, pages 164–171. IEEE Comp. Soc. Press, N.Y., November 1995.

[12] Martin A. J., Burns S. M., Lee T. K., Borkvic D., and Hazewindus P. J. **The Design of an Asynchronous Microprocessor** . In *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373. MIT Press, 1989.

[13] Henessey J. L. and Patterson D. A. **Computer Architecture: A Quantative Approach** . Morgan Kaufman Publishers, Inc., P. O. Box 50490, Palo Alto, California 94303, 1990.

[14] Lavagno L. **Synthesis and Testing of Bounded Wire Delay Asynchronous Circiuts from Signal Transition Graphs** . PhD thesis, Department of Electrical Engineering and Computer Science, University of Califirnia at Berkeley, 1992.

[15] Lloyd L. **The ADLX: An Asynchronous RISC Implementation of the DLX Microprocessor Architecture** . Master's thesis, Department of Computing Science, University of Newcastle upon Tyne, September 1995.

[16] Sentovich E. M., Singh K. J., Lavagno L., Moon C., Murgai R., Saldanha A., Savoj H., Stephan P. R., Brayton R. K., and Sangiovanni-Vincentelli A. **SIS: A System for Sequential Circuit Systhesis** . Technical Report UCB/ERL M92/41, Department of Computer Science, University of California, Berkeley, May 1992. Electronics Research Laboratory Memorandum.

[17] Lam P. N. and Li H. F. **Hierarchical Design of Delay-Insensitive Systems** . *IEEE Proceedings, Part E, Computers and Digital Techniques*, E-137(1):41–56, January 1990.

[18] Hauck S. **Asynchronous Design Methodologies: An Overview** . In *Proc. of the IEEE, Vol. 83, No 1*, January 1995.

[19] Cypress Semiconductor. **WARP^{TM} VHDL Development System: WARP Synthesis Compiler Manual** . Cypress Semiconductor, 3901 North First Street, San Jose, CA 95134, January 1995.

[20] Murata T. **Petri Nets: Properties, Analysis and Applications** . *Proceedings of the IEEE*, 77(4):541–580, 1989.

[21] Udding J. T. **A Formal Model for Defining and Classifying Delay-Insensitive circuits** . *Distributed Computing*, 1(4):197–204, 1986.

[22] van Berkel K. & Rem M. **VLSI Programming of Asynchronous Circuits for Low Power** . In Birtwistle G. and Davis A., editors, *Asynchronous Digital Circuit Design*, pages 151–210. Springer-Verlag, 1995.

Appendix A

Complete VHDL code listing used for the simulation of the event driven buffer mechanism within the WARP VHDL Development System.

```
--$Binding file
use work.cypress.all;
use work.rtlpkg.all;

-- *******************************************
-- design of the 8bit event buffering mechanism
-- *******************************************

------------------------------------------------------------
-- pass gate for the async buffer, declaration and behaviour
------------------------------------------------------------

ENTITY pass_gate IS
        -- external ports
        PORT (d_in, en : IN BIT; out_a, out_b, en_out : OUT BIT);
END pass_gate;

ARCHITECTURE arch_pass_gate OF pass_gate IS
        -- internal signals
        SIGNAL int1 : BIT;
BEGIN
        -- internal behaviour
        -- the enable signal (pass) is used to 'open' one transmission
        -- gate whilst (due to an invertor) closing the other.  This
        -- means that d_in appears on only one of the 'd_out' lines

        assert_pass_gate : PROCESS
        BEGIN
                IF (en = '1') THEN
                        out_a <= d_in;
                        en_out <= en;
                ELSE
                        out_b <= d_in;
                        en_out <= en;
                END IF;
```

```
                WAIT ON en;
        END PROCESS assert_pass_gate;
END arch_pass_gate;

------------------------------------------------------------
-- done gate for the async buffer, declaration and behaviour
------------------------------------------------------------

ENTITY done_gate IS
        -- external ports
        PORT (in_a, in_b, en : IN BIT; d_out, en_out : OUT BIT);
END done_gate;

ARCHITECTURE arch_done_gate OF done_gate IS
        BEGIN
        -- internal behaviour
        -- the enable signal (pass) is used to 'open' one transmission
        -- gate whilst (due to an invertor) closing the other.  The
        -- propagated data signal will appear on the d_out line

        assert_d_out : PROCESS
        BEGIN
                IF (en = '1') THEN
                        d_out <= in_b;
                        en_out <= en;
                ELSE
                        d_out <= in_a;
                        en_out <= en;
                END IF;

                WAIT ON en;
        END PROCESS assert_d_out;
END arch_done_gate;

------------------------------------------------------------------
-- completition detection logic harness, declaration and behaviour
------------------------------------------------------------------

ENTITY completition_detection IS
```

```
        -- external ports
        PORT (d_in, d_out, en : IN BIT; en_out : OUT BIT);
END completition_detection;

ARCHITECTURE arch_completition_detection OF completition_detection IS
BEGIN
        -- internal behaviour
        -- the CDL will forward the enable signal once d_in = d_out

        assert_output : PROCESS
        BEGIN
                IF ((d_in = '1') AND (d_out = '1')) THEN
                        en_out <= en AFTER 3 ns;
                END IF;

                WAIT ON d_out;
        END PROCESS assert_output;
END arch_completition_detection;

---------------------------------------------------
-- complete event buffer, declaration and behaviour
---------------------------------------------------

ENTITY event_buffer IS
        -- external ports
        PORT (d_in, en : IN BIT; d_out, en_out : OUT BIT);
END event_buffer;

ARCHITECTURE arch_event_buffer OF event_buffer IS
        -- internal signals
        SIGNAL int1, int2, int3, int4, int5 : BIT;
BEGIN
        -- internal behaviour
        -- the event buffer will forward the buffer enable signal
        -- provided the data has been seen at the output of the buffer
        -- first

        g1 : pass_gate PORT MAP (d_in, en, int1, int2, int3);
        g2 : done_gate PORT MAP (int1, int2, int3, int4, int5);
        g3 : completition_detection PORT MAP (d_in, int4, int5, en_out);
```

```
        d_out <= int4;
END arch_event_buffer;

--------------------------------------------
-- 8 input MullerC declaration and behaviour
--------------------------------------------

ENTITY mullerC_8 IS
        -- external ports
        PORT (a, b, c, d, e, f, g, h : IN BIT; i : OUT BIT);
END mullerC_8;

ARCHITECTURE arch_mullerC_8 OF mullerC_8 IS
        BEGIN
        -- internal behaviour
        -- the 8 input MullerC is used to determine when all the event
        -- buffers have propagated their enable signals.  These signals
        -- are combined into one single enable signal.

        assert_output : PROCESS
        BEGIN
                IF ((a = '1') AND (b = '1') AND (c = '1') AND
                    (d = '1') AND (e = '1') AND (f = '1') AND
                    (g = '1') AND (h = '1')) THEN
                        i <= '1';
                ELSIF ((a = '0') AND (b = '0') AND (c = '0') AND
                       (d = '0') AND (e = '0') AND (f = '0') AND
                       (g = '0') AND (h = '0')) THEN
                        i <= '0';
                END IF;

                WAIT ON a, b, c, d, e, f, g, h;
        END PROCESS assert_output;
END arch_mullerC_8;

---------------------------------------------------------------
-- 8 input event buffering mechanism, declaration and structure
---------------------------------------------------------------
```

```
ENTITY event_buffer_8 IS
        -- external signals
        PORT (a, b, c, d, e, f, g, h, en_in : IN BIT;
              i, j, k, l, m, n, o, p, en_out : OUT BIT);
END event_buffer_8;

ARCHITECTURE arch_event_buffer_8 OF event_buffer_8 IS
        -- internal signals
        SIGNAL int1, int2, int3, int4, int5, int6, int7, int8 : BIT;
BEGIN
        -- internal behaviour
        -- the 8 input event buffering mechanism is designed to
        -- buffer an 8-bit PC value until there is a change of
        -- level (transition) of the enable signal.

        g1 : event_buffer PORT MAP (a, en_in, i, int1);
        g2 : event_buffer PORT MAP (b, en_in, j, int2);
        g3 : event_buffer PORT MAP (c, en_in, k, int3);
        g4 : event_buffer PORT MAP (d, en_in, l, int4);
        g5 : event_buffer PORT MAP (e, en_in, m, int5);
        g6 : event_buffer PORT MAP (f, en_in, n, int6);
        g7 : event_buffer PORT MAP (g, en_in, o, int7);
        g8 : event_buffer PORT MAP (h, en_in, p, int8);

        g9 : mullerC_8 PORT MAP (int1, int2, int3, int4, int5,
                int6, int7, int8, en_out);
END arch_event_buffer_8;

-- ********************************
-- design of the 8bit PC incrementer
-- ********************************

-------------------------------------
-- invertor declaration and behaviour
-------------------------------------

ENTITY invertor IS
        -- external ports
        PORT (a : IN BIT; b : OUT BIT);
END invertor;
```

```
ARCHITECTURE arch_invertor OF invertor IS
BEGIN
        -- internal behaviour
        b <= NOT a AFTER 1 ns;
END arch_invertor;

-----------------------------------------
-- 2 input NAND declaration and behaviour
-----------------------------------------

ENTITY nand2 IS
        -- external ports
        PORT (a, b : IN BIT; c : OUT BIT);
END nand2;

ARCHITECTURE arch_nand2 OF nand2 IS
BEGIN
        -- internal behaviour
        c <= a NAND b AFTER 1 ns;
END arch_nand2;

---------------------------------------
-- 2 input OR declaration and behaviour
---------------------------------------

ENTITY or2 IS
        -- external ports
        PORT (a, b : IN BIT; c : OUT BIT);
END or2;

ARCHITECTURE arch_or2 OF or2 IS
BEGIN
        -- internal behaviour
        c <= a OR b AFTER 1 ns;
END arch_or2;

---------------------------------------
-- half adder declaration and structure
---------------------------------------
```

```
ENTITY half_adder IS
        -- external ports
        PORT (a, b : IN BIT; sum, carry : OUT BIT);
END half_adder;

ARCHITECTURE arch_half_adder OF half_adder IS
        -- internal signals
        SIGNAL int1, int2, int3 : BIT;
BEGIN
        g1 : nand2 PORT MAP (a, b, int1);
        g2 : nand2 PORT MAP (a, int1, int2);
        g3 : nand2 PORT MAP (b, int1, int3);
        g4 : nand2 PORT MAP (int2, int3, sum);
        g5 : invertor PORT MAP (int1, carry);
END arch_half_adder;

---------------------------------------
-- full adder declaration and structure
---------------------------------------

ENTITY full_adder IS
        -- eternal ports
        PORT (a, b, c : IN BIT; sum, carry : OUT BIT);
END full_adder;

ARCHITECTURE arch_full_adder OF full_adder IS
        -- internal signals
        SIGNAL int1, int2, int3 : BIT;
BEGIN
        g1 : half_adder PORT MAP (a, b, int1, int2);
        g2 : half_adder PORT MAP (c, int1, sum, int3);
        g3 : or2 PORT MAP (int2, int3, carry);
END arch_full_adder;

---------------------------------
-- incrementer unit and behaviour
---------------------------------

ENTITY incrementer IS
```

```
        -- external ports
        PORT (a, b : OUT BIT);
END incrementer;

ARCHITECTURE arch_incrementer OF incrementer IS
        -- internal signals
        SIGNAL set : BIT := '0';
BEGIN
        -- the incrementer is designed to provide constant (0 & 1)
        -- logical values to the 8-bit adder - used as a hard encoded
        -- value that is added to the PC value (in this case "00100000").

        assert_output : PROCESS
        BEGIN
                a <= set;
                b <= NOT set;
        END PROCESS assert_output;
END arch_incrementer;

---------------------------------------------
-- full 8-bit adder declaration and structure
---------------------------------------------

ENTITY adder_8bit IS
        -- eternal ports
        PORT (a, b, c, d, e, f, g, h : IN BIT;
              i, j, k, l, m, n, o, p : OUT BIT);
END adder_8bit;

ARCHITECTURE arch_adder_8bit OF adder_8bit IS
        -- internal signals
        SIGNAL int1, int2, int3, int4, int5, int6, int7, int8 : BIT;
        SIGNAL low, high : BIT;
BEGIN

        g1 : incrementer PORT MAP (low, high);

        g2 : half_adder PORT MAP (a, low, i, int1);
        g3 : full_adder PORT MAP (b, low, int1, j, int2);
        g4 : full_adder PORT MAP (c, high, int2, k, int3);
```

```
        g5 : full_adder PORT MAP (d, low, int3, l, int4);
        g6 : full_adder PORT MAP (e, low, int4, m, int5);
        g7 : full_adder PORT MAP (f, low, int5, n, int6);
        g8 : full_adder PORT MAP (g, low, int6, o, int7);
        g9 : full_adder PORT MAP (h, low, int7, p, int8);
END arch_adder_8bit;

-- ******************************************************************
-- top level design of the complete program counter incrementer unit
-- ******************************************************************

--------------------------------------------
-- PC incrementer, declaration and structure
--------------------------------------------

ENTITY PC_incrementer IS
        -- external signals
        PORT (a, b, c, d, e, f, g, h, en_in : IN BIT;
              i, j, k, l, m, n, o, p, en_out : OUT BIT);
END PC_incrementer;

ARCHITECTURE arch_PC_incrementer OF PC_incrementer IS
        -- internal signals
        SIGNAL a1, b1, c1, d1, e1, f1, g1, h1 : BIT;
BEGIN
        -- internal behaviour

        g1 : adder_8bit PORT MAP (a, b, c, d, e, f, g, h,
                                  a1, b1, c1, d1, e1, f1, g1, h1);
        g2 : event_buffer_8 PORT MAP (a1, b1, c1, d1, e1, f1, g1, h1, en_in,
                                      i, j, k, l, m, n, o, p, en_out);
END arch_PC_incrementer;

-- ******************************************************************
```