Classification Of Static Asynchronous Pipelines

Albin Frischenschlager 0926427, Thomas Frühwirth 0927088

INTRODUCTION

In this chapter, static pipelines will be examined and discussed in detail. As already written in the previous chapter static vs. dynamic pipelines, the former ones need storage elements between the pipeline stages [8]. This chapter presents different pipeline-styles which use different storage elements such as latches, flip-flops, pseudo-static Swensson-style double edge-triggered D-flip-flops (DETDFF) or just simple storage loops.

These pipeline-styles don't only differ in the choice of the storage elements, but also in the communication protocols, which are used by the pipeline stages to communicate among each other. Actually the choice of the communication protocol strongly influences the usable storage elements. The presented pipeline-styles in this chapter either use the common 2-phase, 4-phase protocol or pulses.

The chapter static pipelines is structured as follow. First the basics of important concepts like 2-phase-, 4-phase protocol and signal transition graphs will be presented for the reader, who isn't familiar with static pipelines. Then based on the famous micropipeline, invented by Ivan Sutherland, operation of a static pipeline will be introduced. Following, different pipeline-styles will be presented in the following order: 2-phase pipeline-styles, 4-phase pipeline-styles and pipeline-styles which use pulses. At the end, simulation results for all presented pipeline-styles will be showen and interpreted.

2-PHASE VS. 4-PHASE

Communication in asynchronous circuits is done via so called handshakes. The sender informs the receiver, when new valid data is available. When the receiver safely received the data, it informs the sender, that it is ready for new data. There are various ways, how this information process can be done. The details of the information process are defined by a handshake protocol. This chapter will introduce the two most common handshake protocols, where there is a dedicated line for the sender to inform the receiver, the request line, and a dedicated line in the other direction, the acknowledge line.

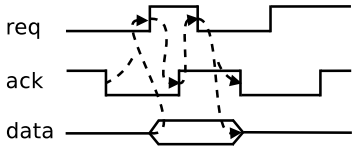

Fig. 1. 4-phase protocol [10]

The first presented handshake protocol is the so called 4-phase protocol, also known as return-to-zero protocol, which is shown in figure 1. A new phase is initiated by the sender via a rising edge on the request line, indicating that new valid data is available. With a rising edge on the acknowledge line, the receiver signals the completion of the data capturing. Then the sender responds with a falling transition on the request line, which is again acknowledged by the receiver with a falling transition on the acknowledge line ending the phase. The 4-phase protocol got its name from these four activities/transitions.

There are two additional attributes, the data attribute and the channel attribute, which influence when and how long the data is valid. For example, the 4-phase protocol shown in figure 1 is a broad-data, push-channel 4-phase protocol. Nevertheless, the sequence of the transitions stays the same in all 4-phase protocols, regardless of the attribute. For completeness, [10] mentions the attributes early-, broad-, late- and extended early-data and push- and pull-channel.

As one can see, the last two steps (the return-to-zero transitions) are not necessary for the previously described handshake. This conclusion leads directly to the second presented handshake protocol.

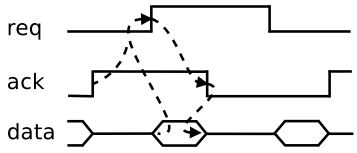

Fig. 2. 2-phase protocol [10]

The so called 2-phase protocol, also known as non-return-to-zero protocol is shown in figure 2. The behaviour is exactly as previously required. Via any transition on the request line the sender informs, that new valid data is available. Thus, a new phase is started. When the receiver safely captured the data, it will do the same transition on the acknowledge line, finishing the phase.

Similar to the 4-phase protocol, there is the channel attribute, to influence, when and how long the data is valid. Figure 2 shows the push-channel, the alternative mentioned in [10] is the pull-channel.

SUTHERLAND'S MICROPIPELINE

The Micropipeline presented in the famous Turing-Award lecture from Ivan Sutherland [12] is the fundamental design for static bundled-data pipelines. The beauty of this approach lies in the simple structure and therefore, in the easiness to understand the concept of static bundled-data pipelines. As Sparso in [10] writes, nearly every static bundled-data pipeline uses the ideas of the micropipeline and extends it in various ways. Therefore, it's beneficial to understand the Micropipeline to find emergent patterns in other static bundled-data pipelines.

For better understanding figure 3 only shows the control circuit of the Micropipline. One can see immediately that the

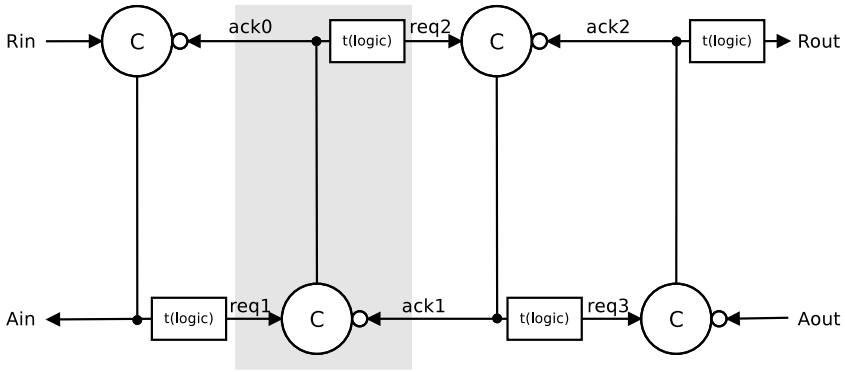

Fig. 3. Control circuit of the Micropipeline [12]

pipeline structure is very simple, because a stage only consists of one Muller C-element, one inverter and one delay element to fulfil the bundling constraint. Originally, the Micropipeline was designed to use 2-phase communciation, but one neat feature is, that also 4-phase communciation can be used, without requiring changes on the control circuit. The only difference is how to interpret the signals [10].

The general rule for a stage can be described as follows: if the predecessor and the successor differ in state, then copy predecessor's state, else hold the current state [12]. The circuit is built in a way, that the predecessor state equals the incoming request line and that the successors state equals the acknowledge line. Thus, the rule for a stage can be reformulated into: if the incoming request line and the incoming acknowledge line are different, then copy the state of the incoming request line, else hold the current state.

Initially all Muller C-elements are initialized to one state, let's say low. Therefore, each incoming request and acknowledge line have the same state. Since a 2-phase communication protocol is used, it doesn't matter if the initialized state is high or low, as long as the whole system has the same initial state. If the request line Rin transitions to high the Muller C-element of the first stage drives high too. This matches exactly with the reformulated rule, because request and acknowledge differ. Consequently, Ain and $req1$ get driven high too. After this, two parallel events happen. The second stage will notice, that $req1$ is not equal to $ack1$ and will do the same as the first stage, i.e drive the stage to high. Thus, stage one and stage two will have the same local state, i.e. high. The same will happen with the third stage and so on, until every stage has the local stage high. Concurrently, the component feeding the pipeline from the left, will notice that Rin and Ain have the same state, and will maybe toggle Rin again. If this happens after stage 2 has copied the previous state of stage 1, then by the above rule, stage 1 can copy the new state. Contrary, if this happens when stage 2 hasn't copied the previous stage of stage 1 yet, stage 1 won't copy the new state, since the Muller C-element gets different inputs. This matches with the rule, to hold the state when the request line and the acknowledge line are the same.

As already stated, the component on the left side of the Micropipeline can feed new data into the pipeline, when Rin and Ain are equal. On the right side, it's directly the other way around. When $Rout$ and $Aout$ differ, the component on the right side of the Micropipeline can fetch new data from the pipeline, and thus, enabling the rightmost pipeline stage to capture new data from the predecessor stage.

Since the Micropipeline uses a 2-phase communication protocol, storage elements, which behave the same on rising and on falling transitions are needed. Hence, in [12] Sutherland presented the capture/pass register, which supports this behaviour. This storage element has a capture input and a pass input and corresponding outputs. On a transition on the capture input, the storage element closes, thus, capturing the data. Then, the transition is forwarded to the capture output. On a transition on the pass input, the storage element gets transparent again and then, the transition is forwarded to the pass output.

Figure 4 shows how capture/pass registers are used to add a datapath to the Micropipeline. The behaviour stays the same, only capture/pass registers and logic get added.

This chapter concludes the introduction of important concepts in static pipelines. In the next chapters, 2-phase pipeline-styles are presented, which evolved from Sutherland's Micropipeline.

2-PHASE COMMUNICATION AND PHASE CONVERSION IN COMBINATION WITH CONVENTIONAL LATCHES

A. Introduction

Since capture-pass registers as used in Sutherland's micropipeline are costly in area the idea to use ordinary storage elements came up. But conventional latches are level sensitive, i.e. they are transparent when the enable (En) signal is high and blocking when En is low or vice versa. Obviously the single C-element of Sutherland's micropipeline is not suitable for controlling these latches, at least as long two-phase handshaking is used. More advanced latch control circuits need to be designed. Starting with the simple idea of implementing a two-phase to four-phase conversion, Day and Woods propose several enhancements of this technique in [3].

B. Structure

The basic structure of the pipeline is based on Sutherland's micropipeline with some obvious modifications. Firstly Suther-

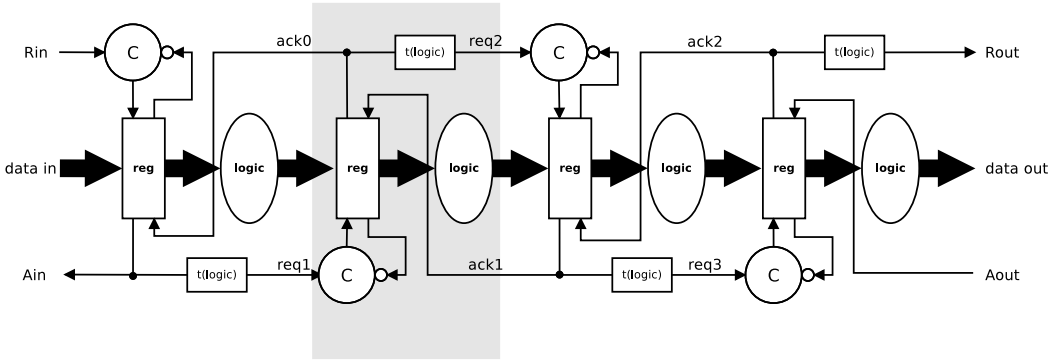

Fig. 4. Micropipeline with processing logic [12]

land's capture-pass registers are replaced by so called pass-transistor transparent latches as depicted in figure 5. Each latch consists of a transmission gate and a storage loop, which holds its value using a weak driver. Opening the transmission gate by setting nEn low and En high overrides the present value of the storage loop. Note that the output of the latch is inverted resulting in the necessity to either add an extra inverter or adopt the processing logic accordingly.

Secondly a more complex latch control circuit as shown in figure 6 is needed. As usual, signals Rin and Ain are used to perform the input handshake with the previous stage, Rout and Aout perform the output handshake with the following stage. nEn and En are the signals used to control the latch. The two-phase to four-phase conversion is done by the exclusive or, which changes its output whenever any input changes. On the other hand some four-phase to two-phase conversion is needed as well. This conversion is done by the Toggle-Box as follows: The first input event is propagated to the output marked with a black dot, the second event is then propagated to the blank output and so on. An event is any kind of input transition. Thus the corresponding output is toggled regardless whether there was a rising or a falling edge on the input.

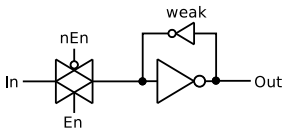

Fig. 5. Pass transistor latch consisting of a transmission gate and a storage loop [3]

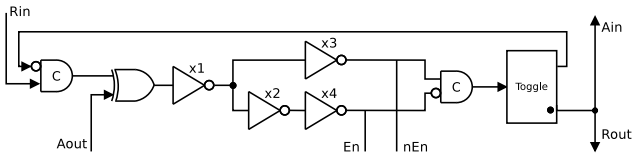

Fig. 6. Control circuit implementing phase-conversion [3]

C. Protocol

Two neighbouring stages communicate using exactly the same two-phase protocol already described in section "2-PHASE vs. 4-PHASE".

D. Performance

In the initial state all latches are transparent and all event lines are low. The left most C-element is armed to change its output when Rin rises. Assume the previous stage puts some data on the data path an then toggles Rin. The output of the XOR rises and closes the latch. The right C-element waits until both latch control signals have changed their state and then rises its output. This transition causes the Toogle-Box to rise its output marked with the black dot signalling the previous stage that the data has been consumed as well as the next stage that new data is available. Eventually the next stage will acknowledge this transition by pulling up Aout. This event opens the latches, propagates through the blank output of the Toggle-Box and armes the left C-element for a new event on Rin. This completes the cycle.

Thus for the cycle time of the circuit the delay of an arbitrary stage i, the successive stage i+1 and the predecessor stage i-1 have to be analysed. $t_{Rin\rightarrow Rout} = t_{Rin\rightarrow Ain}$ is the time needed to toggle Rout and Ain after Rin has toggled. This time has to be added twice because stage i first toggles Rout and the waits for acknowledgement of stage i+1. $t_{Aout\rightarrow ready}$ is the time the circuit needs to close the latches and arm the left C-element for the next input transition after Aout has been toggled by the successive stage. The delay line matching the logic delay is as usual located between Rout of stage i and Rin of stage i+1 and its delay is termed t_{logic}.

The cycle time is thus given by:

$$T = t^i_{Rin\rightarrow Rout} + max((t^i_{logic} + t^{i+1}_{Rin\rightarrow Ain} + t^i_{Aout\rightarrow ready}), (t^{i-1}_{Aout\rightarrow ready} + t^{i-1}_{logic})) \quad (1)$$

where

$$t_{Rin\rightarrow Rout} = 2 \cdot t_{celem} + t_{xor} + t_{x1} + max(t_{x2} + t_{x4}, t_{x3}) + t_{celem} + t_{toggle} \quad (2)$$

and

$$t_{Aout \rightarrow ready} = t_{xor} + t_{x1} + max(t_{x2} + t_{x4}, t_{x3}) + t_{celem} + t_{toggle} \quad (3)$$

Combining these results finally gives:

$$T = 5 \cdot t_{celem} + t_{logic} + 3 \cdot (t_{xor} + t_{x1} + max(t_{x2} + t_{x4}, t_{x3}) + t_{toggle}) \quad (4)$$

Of course it's not quite accurate to assume equal delays for all C-gates regardless of the number of inputs, rising and falling transitions, output loads, etc. Taking all these factors into account would result in enormously complex formulas, especially when more complex pipelines are analized. Nevertheless, the formulas give an idea of the number of basic circuit elements and the logic delay-lines involved.

E. Timing Constraints

Day and Woods claim that assuming that the bundling constraints are met, the circuit can be considered delay-insensitive. This statement however relies on the assumption that the latch delay is shorter than $t_{Rin \rightarrow Rout}$, which is true in most cases anyway and can always be fulfilled by taking the latch delay into account when the logic delay is determined. Furthermore using the term "delay-insensitive" in this context might cause confusion because a bundling data approach can never be truly delay-insensitive.

In subsection "Optimizations" faster versions of the circuit are presented. However some more restrictive timing constraints have to be assumed for these circuits to function correctly.

F. Initialization

In the initial state the output of the C-element as well as both outputs of the Toggle-Box are low.

G. Optimizations

One possible improvement of the circuit of figure 6 is illustrated in figure 7. Here, Rout is fast forwarded, i.e. it is wired from the output of the left C-element directly. This reduces the output request propagation delay and the cycle time. The input acknowledgement event Ain still waits until the latch has closed to ensure that the data is not violated during latching. For this circuit to meet all timing constraints the delay introduced by the C-element must be larger than the latch delay.

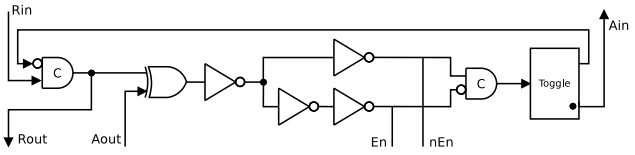

Fig. 7. Control circuit fast forwarding Rout [3]

Another optimization option is termed single-phase transparent latch structures in the literature. The control circuits shown in figure 6 can be improved by removing the right C-element. The inverter x1 thus can be hidden in the XOR, resulting in an XNOR without any additional hardware effort or delay. Again two inverters are used as buffers for driving the latch enable signals. Since En is initially high the first event arriving at the toggle box would be a falling edge, compared to a rising edge as before. Thus an inverter is needed at the input, which can again be hidden in the Toggle-Box. The resulting circuit is depicted in figure 8. Again the cycle time of the circuit is reduced. However the pass transistor latches used so far need En as well as nEn signals to perform their operation. Figure 9 shows a possible implementation of a static single phase transparent latch originally designed by Yuan and Svensson [13]. The weak pull up transistor "wk" prevents the second transistor stack from drifting low and spuriously changing the value stored in the inverter loop when In was low and changed to high after En has gone low. Note that the output of the latch is again inverted. Furthermore it's again possible to implement a fast forward version of the circuit.

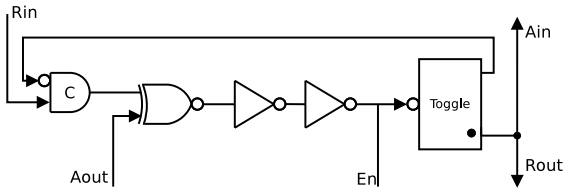

Fig. 8. Single-phase latch micropipeline control circuit [3]

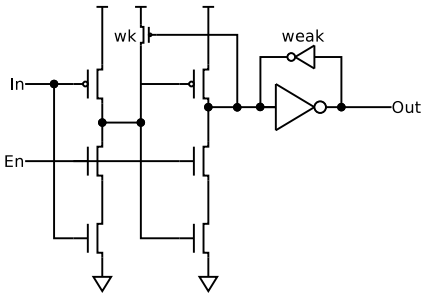

Fig. 9. Single-phase static transparent latch [3]

H. Issues

The drawback of all circuits presented is the complexity of the whole circuit in general and especially of the logic hidden so far by just calling it Toggle-Box. Yun, Beerel and Arceo give a possible implementation of the toggle-circuit in [14]. The circuit is depicted in figure 10. It is built of 18 transistors. Depending on the implementation of the C-element this makes up to 37,5% of the overall circuit's transistor count and, additionally, is responsible for a considerable part of the cycle time.

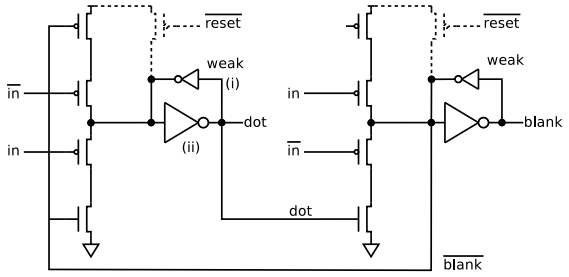

Fig. 10. Implementation of the toggle circuit [14]

2-PHASE HANDSHAKE PROTOCOL AND DETDFF STORAGE ELEMENTS

A. Introduction

Yun, Beerel and Arceo introduce in [14] a new storage element for 2-phase micropipelines called pseudo-static Swensson-style double edge-triggered D-flip-flop (DETDFF). Two neighboring stages can still use energy efficient 2-phase communication. However the use of DETDFF overcomes the necessity of implementing Sutherland's capture-pass latches or 2-phase to 4-phase converters, which are necessary when conventional latches are used, but are costly in area.

B. Structure

Figure 11 shows a four stage micropipeline. The structure depicted is quite similar to Sutherland's micropipeline, except for the capture-pass registers being replaced by DETDFF. The DETDFF employ the blocking latch scheme. [1] The bundling delay t_d is again necessary to ensure that the data is valid at stage i+1 when the request signal toggles. Furthermore for driving a bunch of DETDFFs a buffer is necessary. This buffer introduces the delay t_{buf}.

C. Protocol

Two neighboring stages communicate using exactly the same 2-phase protocol already described in section "2-PHASE vs. 4-PHASE".

D. Performance

A cycle starts when Rin toggles and is finished when another request toggles Rin again and additionally the C-gate is armed for this new request. Assuming that stage i+1 is empty and with respect to timing assumptions discussed in subsection -E the cycle time is given by

$$T = t^i_{R\to A} + max((t^{i-1}_{A\to A} + t^{i-1}_d), (t^i_d + t^{i+1}_{R\to A})). \quad (5)$$

[1] In the blocking latch scheme energy consuming glitches caused by the processing logic don't propagate through the whole pipeline but only between two successive stages.

E. Timing Constraints

Yun, Beerel and Arceo mention two timing constrains that have to be met in order for the pipeline to function correctly:

1) Data setup time
The bundling delay(t_d) must be long enough to meet the data setup time. According to Figure 11 this requires the following inequality to hold:

$$t^i_d + t^{i+1}_{R\to A} + t^{i+1}_{buf} > t^i_{buf} + t^i_{ck\to Q} + t^i_{logic} + t_{su} \quad (6)$$

Assuming $t^i_{buf} \approx t^{i+1}_{buf}$, which can be met by careful routing, the above formula simplifies to

$$t^i_d > t^i_{ck\to Q} + t^i_{logic} + t_{su} - t^{i+1}_{R\to A} \quad (7)$$

2) Data hold time
After the acknowledge line has been toggled by stage i+1 the data still has to be valid for at least the data hold time(t_h):

$$t^i_{A\to A} + t^i_{buf} + t^i_{ck\to Q} + t^i_{logic} > t^{i+1}_{buf} + t^{i+1}_h \quad (8)$$

This inequality is trivially satisfied because of the wire delay of the data lines. The exact same assumption is inherent in the synchronous design principle and has proven to hold.

In fact the timing constraints discussed above have to hold for all bundled data pipelines in a similar manner. Throughout the rest of the paper $t_d = t_{logic}$ is assumed for the sake of easy comparison of the different pipeline styles. This means that the delay line exactly matches the logic delay. Assuming that $t^i_{buf} = t^{i-1}_{buf}$ simplifies the data setup time constraint to

$$t^i_{R\to A} > t^{i-1}_{ck\to Q} + t_{su}. \quad (9)$$

Therefore additional delay has to be to added somewhere if this inequality is not fulfilled by the gates and flip-flops used. However, strictly spoken this additional delay is not part of the logic delay t_{logic} even if it might be implemented that way in a practical realization.

F. DETDFF CMOS implementation

A possible implementation of the DETDFF is given in figure 12. The DETDFF is build up of a rising edge-triggered DFF (RETDFF) and a falling edge-triggered DFF (FETDFF). The RETDFF is formed by the top half and the FETDFF is formed by the bottom half of the circuit. The functionality of the circuit is most easily described using an example. Assume the data input D as well as the control input ϕ are low, i.e. the next rising edge on ϕ will cause the circuit to store D = 0. Then transistors $P1$ and $P2$ are turned on and node x is pulled up. Since ϕ is 0, $P3$ is also turned on and pulls node y up, which has no effect on the output because $P4$ as well as $N5$ are turned off. Obviously toggling D has no effect on the output as long as ϕ stays 0. Toggling ϕ causes y to be pulled down, Q' to be pulled up and Q to be pulled down, thus Q = D = 0 as desired. y being low turns on $P5$, which, as a consequence, pulls y low. In this case $P5$ and N3 plus $N2$ work as a storage loop. Again toggling D has no effect on Q as long as ϕ does not toggle.

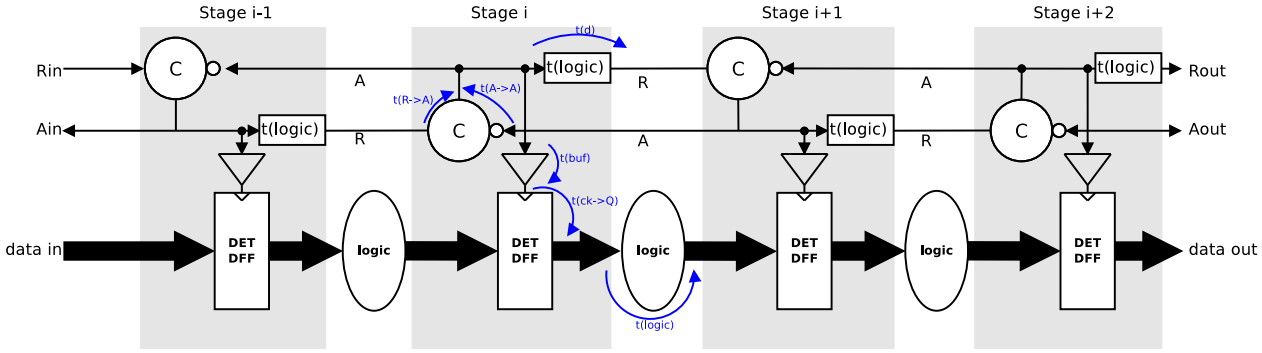

Fig. 11. four stage pipeline using DETDFFs [14]

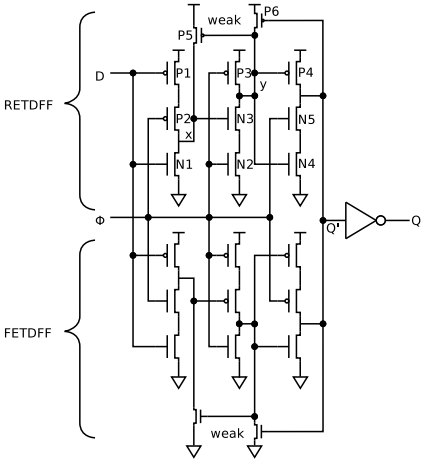

Fig. 12. CMOS implementation of a DETDFF [14]

G. Initialization

On initialization the pipeline is empty and all C-element outputs are low.

H. Issues

The main disadvantages of the circuit presented concern the DETDFFs. Each DETDFF is about twice the size of a standard DFF and the capacitance on the DETDFF control lines is approximately four times higher. Compared to standard latches and 4-phase control circuits however, DETDFFs require only half as many input transitions. Furthermore they are naturally blocking, so the total energy consumption heavily depends on the processing logic used.

MOUSETRAP

A. Introduction

MOUSETRAP stands for minimal-overhead ultra-high-speed transition-signaling asynchronous pipeline [9]. As the name already implies, MOUSETRAP uses transition-signaling (i.e 2-phase) to communicate with neighboring pipeline stages. Level-sensitive D-latches, which are small and fast are used as storage-elements. When a stage is waiting for data, the latch is open. As soon as new data arrives, the stage closes the latch to save the arrived data. This allows fast operation at the cost of power consumption (see section Two-phase handshake protocol and detdff storage elements). The huge advantage of MOUSETRAP is, that it can be built with standard components, but has nearly the performance of Pipelines with very specialised components as Gasp and IPCMOS.

B. Structure

Figure 13 shows a MOUSETRAP stage. As one can see, a latch controller only consists of a XNOR-Gate. In combination with the standard level-sensitive D-type transparent latch, this gives MOUESTRAP its high performance. The latch controller communicates via Request (Rin and $Rout$) and Acknowledge (Ain and $Aout$) with neighboring stages, where Request and Acknowledge have the same semantics as in the traditional micropipeline (see section Sutherlands Micropipeline). As already hinted by the name, the latch controller controlls the latch and issues when to open or close it via the en signal. Although in figure 13 there is just one latch drawn with two inputs (Rin und $data\ in$), actually two latches are used. One for Rin and one for $data\ in$, but both are controlled in parallel via the en signal coming from the XNOR-Gate.

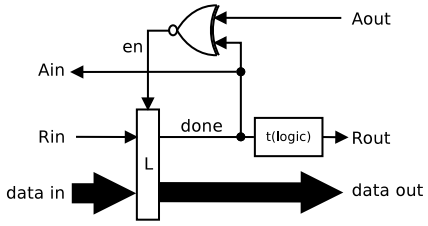

Fig. 13. MOUSETRAP pipeline-stage [9]

C. Protocol

Since a latch controller only consists of a XNOR gate, the protocol is rather easy. With the help of figure 14, the mechanics of the MOUSETRAP pipeline shall be discussed.

When $ack(i)$ and $done(i)$ have the same logic value, the XNOR gate drives en high, which issues the latch to be

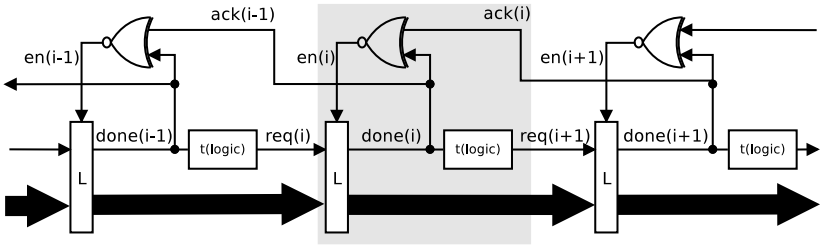

Fig. 14. MOUSETRAP pipeline with processing logic [9]

transparent. When they have different logic values, the XNOR drives en low, which closes the latch, and thus captures the last arrived data. In the initial state, when the pipeline is empty, all $done$, req and ack signals are low. Hence, all latches are transparent, since all req and $done$ have the same logic value. When new data arrives ($req(i)$ toggled) at pipeline-stage i, the following three actions take place in parallel: 1) the incoming data is passed to the next pipeline-stage $i+1$ and $req(i+1)$ gets toggled. 2) $done(i)$ gets toggled. Hence, $done(i)$ and $ack(i)$ have different logic values, which closes the latch. 3) $ack(i-1)$ gets toggled. Hence, $ack(i-1)$ has the same logic value as $done(i-1)$, which frees the predecessor pipeline stage $i-1$.

After the successor stage $i+1$ acknowledged the new data (i.e. toggles $ack(i)$) $done(i)$ and $ack(i)$ have the same logic value. Therefore the latch controller issues the latch to be transparent again.

It's worthwhile to note that the protocol is a hybrid protocol. It uses transition signaling for the communication between the pipline stages and level signaling for the controlling of the latch. With this, the useless empty phases of 4-phase protocols are avoided, but still, fast standard level-sensitive D-type transparent latches can be used.

D. Performance

The cycle for stage i in MOUSETRAP consists of three steps: 1) new data arrives at stage i, passes through latch and logic of stage i, 2) the arrived data at stage $i+1$ passes through the latch of stage $i+1$, and thus produces Ain, 3) $Aout$ issues the XNOR gate to drive en high, hence, enabling the stage for capturing new data. The equation looks as follows:

$$T = t^i_{latch} + max((t^{i-1}_{XNOR\uparrow} + t^{i-1}_{latch} + t^{i-1}_{logic}), (t^i_{logic} + t^{i+1}_{latch} + t^i_{XNOR\downarrow})) \quad (10)$$

E. Timing Constraints

The correct behaviour of MOUSETRAP is only guaranteed, when two simple time constraints, besides the bundling constraints, are fulfilled.

Setup Time: When new data arrives at a transparent latch, it has to be open long enough, so the data can pass through it. This "long enough" is quantified via the setup time t_{setup} of the latch. For MOUSETRAP this means, that the time from arrival of Rin to closing the latch via a falling en (XNOR switching low) has to be greater than t_{setup}:

$$t_{Rin\rightarrow done} + t_{XNOR\downarrow} > t_{setup} \quad (11)$$

Singh and Nowick state in [9], that normally, this timing constraint is easily fulfilled.

Data Overrun (Hold Time): Data in stage i has to be safely captured, before new data from the preceding stage is arriving. If this is not guaranteed, the data that is not captured yet will be overwritten and hence data is lost. This can happen in MOUSETRAP, because Ain and $done$ are generated in parallel. Hence, the time to close the latch (XNOR switching low) must be shorter than the time that is needed to generate and send new data. Of course, the hold time of the latch must be observed:

$$t^{i-1}_{XNOR\uparrow} + t^{i-1}_{latch} + t^{i-1}_{logic} > t^i_{XNOR\downarrow} + t^i_{hold} \quad (12)$$

The left side represents the shortest path from Ain to arrival of new data at stage i. The right side is the path to close the latch in stage i. The equation can be simplified:

$$t^{i-1}_{latch} + t^{i-1}_{logic} > (t^i_{XNOR\downarrow} - t^{i-1}_{XNOR\uparrow}) + t^i_{hold} \quad (13)$$

When $t^i_{XNOR\downarrow} \cong t^{i-1}_{XNOR\uparrow}$ is assumed, the term in the paranthesis cancels out. The resulting inequality is easily met, because normally the latch and logic delay is greater than the hold time of a latch [9].

F. Initialization

The pipeline is initialized via a global reset line, connected to the latch in every stage. This reset line controls a pull-up transistor, which pulls the latch's internal node high, when the line is issued. Thus, the latch outputs low and all req, ack and $done$ signals are low, forcing the XNOR gates to drive all en to high. Hence, all latches are empty and are waiting for new data to arrive.

G. Optimizations

Via waveform shaping of the XNOR gate the cycle time and therefore the performance of MOUSETRAP can be further improved. The idea is to make the rising and falling transition of the XNOR gate asymmetric, which is done via transistor sizing. As one can see in equation 10, decreasing the time needed for a rising transition ($t_{XNOR\uparrow}$) decreases the cycle time. To maintain the same amount of loading in the controller circuit, the falling transition ($t_{XNOR\downarrow}$) has to be made longer. Hence, some loss of timing margins has to be accepted (see equation 11 and 13).

Figure 15 shows four possible levels of waveform shaping of the XNOR gate.

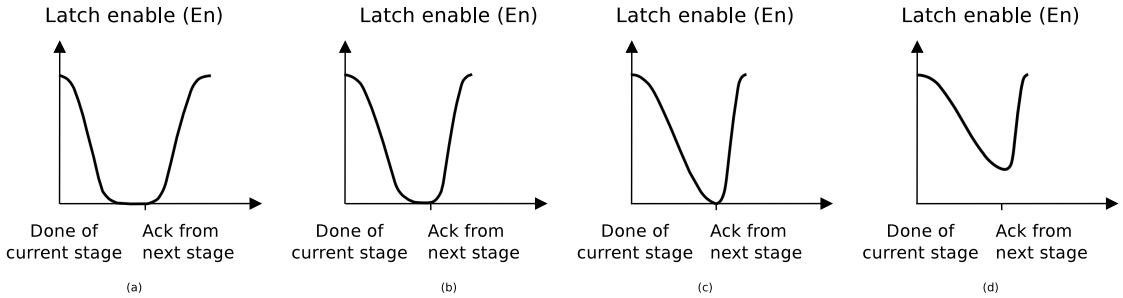

Fig. 15. Waveform shaping optimization. Increasing aggressivity from (a) no waveform shaping to (d) extreme case with reduced voltage swing [9]

SIMPLE 4-PHASE LATCH CONTROL CIRCUIT

A. Introduction

Circuits presented so far use 2-phase communication between 2 neighbouring stages. Due to the fact that conventional latches are level sensitive, 2- to 4-phase conversion was necessary in each stage. These phase-conversion circuits are costly in area and investigations have been made how to avoid this overhead. One example was using DETDFF (double edge triggered DFF). Another possibility is using 4-phase communication between the latch controllers. This introduces the necessity of a reset-phase in the protocol, also called "return to zero" phase. Since the reset-phase is part of the communication protocol no logic processing can be performed during this time. Still in many cases the savings in control logic result in shorter cycle-times and higher performance, especially when asymmetric delay-lines are used.

Furber and Day present three different 4-phase latch control circuits in [4]. All of them have different properties regarding parallelism of operation as will be discussed in the next sections. In this section a very simple 4-phase latch control circuit is introduced.

B. Structure

Figure 16 shows the simple 4-phase latch control circuit. As usual, Rin and Ain perform the input handshake, Aout and Rout perform the output handshake and Lt is used to control the latch. Figure 17 shows the STG used to derive the circuit. Initially all control lines are low. Note that control for the latches is negated in this case, i.e. the latches are transparent when Lt is low and closed when Lt is high. Thus they are initially transparent.

A new cycle starts when the predecessor of the stage feeds data to the latches and then pulls up Rin. Assuming Aout is low, the rising edge of Rin propagates through the C-element and Rout is risen, indicating that new data is available at the output. After some delay introduced by the buffer needed to drive the latches, Lt goes high and closes the latches. Simultaneously Ain goes high, informing the predecessor stage that the data has been consumed and can be removed. As a consequence of Rout+, Aout will eventually be pulled high by the successor stage. Simultaneously, as a consequence of Ain+, Rin will be pulled low by the predecessor stage. This causes the output of the C-gate to fall, having three effects. Firstly the output handshake cycle can now be completed by the successor stage by pulling Aout low. Secondly Lt- again opens the latches. And thirdly Ain is pulled low and thus the input handshake cycle is completed.

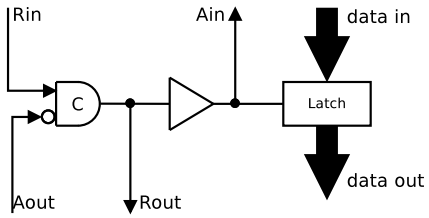

Fig. 16. Simple 4-phase latch control circuit [4]

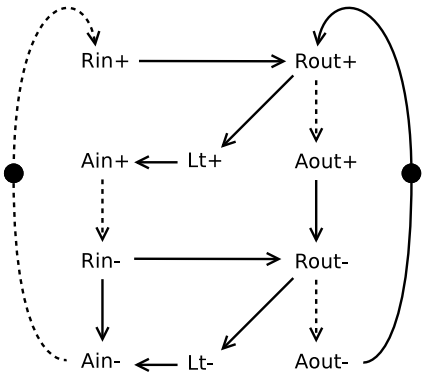

Fig. 17. STG of the simple 4-phase latch control circuit [4]

C. Protocol

The communication protocol used by two neighbouring stages is the standard 4-phase protocol introduced in section "2-PHASE vs. 4-PHASE".

D. Performance

The logic delay t_{logic} delays the *Rout* signal as usual in bundled data pipelines. Since the delay of an asynchronous delay line differs for the the rising and the falling edge, $t_{logic\uparrow}$ and $t_{logic\downarrow}$ are used to reflect this property. A cycle starts when

Rin goes high and is finished when another request pulls up Rin again and, additionally, the C-element is armed for this new request. The three stages involved in a cycle are termed i, i-1(predecessor) and i+1(successor). The logic on the input side of the stage considered has the matched delay t^{i-1}_{logic} and on the output side t^{i}_{logic}.

Using this notation the cycle time is given by

$$\begin{aligned} T = max((t^i_{celem} + t^i_{buf} + t^{i-1}_{celem} + t^{i-1}_{logic\downarrow}), \\ (t^i_{celem} + t^i_{logic\uparrow} + t^{i+1}_{celem} + t^{i+1}_{buf})) \\ + max((t^i_{celem} + t^i_{buf} + t^{i-1}_{celem} + t^{i-1}_{logic\uparrow}), \\ (t^i_{celem} + t^i_{logic\downarrow} + t^{i+1}_{celem} + t^{i+1}_{buf})). \end{aligned} \tag{14}$$

Assuming that all C-elements and all buffers have equal delay this simplifies to

$$\begin{aligned} T = 4 * t_{celem} + 2 * t_{buf} \\ + max(t^{i-1}_{logic\downarrow}, t^i_{logic\uparrow}) + max(t^{i-1}_{logic\uparrow}, t^i_{logic\downarrow}). \end{aligned} \tag{15}$$

Only if the pipeline stages are unequally balanced, i.e. if the logic delays of the stages differ, the use of asymmetric delay lines will result in increased performance.

E. Timing Constraints

Assuming the handshake with the successor stage has already been completed, $Aout$ is 0 and the C-element is armed to toggle as soon as Rin rises. Thus $Rout+$ will follow $Rin+$ after the delay introduced by the C-element. This delay has to be longer than the time needed by the data to propagate through the latch. Otherwise $Rout$ would be risen before the data is actually valid. Note that the buffer delay t_{buf} is built into the path from Rin to Ain because the data must not be removed until Lt has gone high and the latches have closed. It is however not built into the path from Rin to $Rout$ because there is no need for the latch to close before $Rout$ is risen.

F. Initialization

As stated above all control lines are initially low. Thus all C-elements have to be initialized to 0.

G. Issues

The drawback of this circuit is the interdependency of the input and the output handshake cycle. Lt of stage i can only go high when stage i+1 is empty, i.e. Aout is low. As a consequence only every other pipeline stage can be occupied at any time which is undesired for certain applications. Semi-decoupled and fully-decoupled control circuits were developed to overcome this problem and will be discussed in the next sections.

SEMI-DECOUPLED 4-PHASE LATCH CONTROL CIRCUIT

A. Introduction

As argued in the previous section, only every other pipeline stage can hold data when the simple 4-phase latch control circuit is used. This is of course undesired for many applications, in particular FIFOs. Another more enhanced 4-phase controller, presented by Fuber and Day in [4], is called semi-decoupled 4-phase latch controller. This circuit overcomes the drawback of the simple controller.

B. Structure

Figure 18 shows the STG of the semi-decoupled 4-phase latch control circuit. Parallelism of the circuit's operation is increased by introducing the new internal variable A, used to record when the input side is ready to proceed. According to the STG it's now possible to close the latches ($Lt+$) before the successive stage is empty (i.e. $Aout$ need not be low), since $Lt+$ is concurrent with $Aout-$. Thus the input handshake is partly decoupled of the output handshake, i.e. they are semi-decoupled. Because Lt and Ain follow directly from A, with internal dependencies only, these two states were omitted when constructing the not dashed part of the state graph shown in Figure 19. The dashed transitions will be referred to in subsection "Optimizations". A possible implementation using asymmetric C-elements is illustrated in Figure 20.

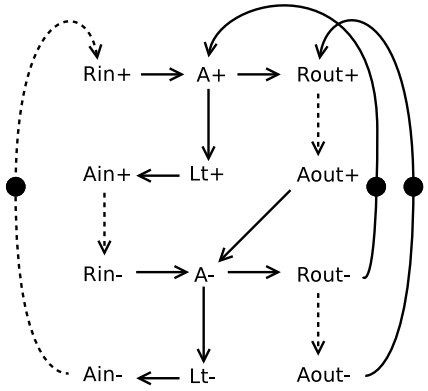

Fig. 18. STG of the semi-decoupled 4-phase latch control circuit [4]

C. Protocol

The communication protocol used by two neighbouring stages is the standard 4-phase protocol introduced in section "2-PHASE vs. 4-PHASE".

D. Performance

As usual, the cycle starts with a rising edge on Rin. This event propagates trough the left C-element, through the buffer for driving the latches and trough the bottom C-element. Ain however cannot go low until $Aout$ has gone high, a property reflected by the max term in the following formula. The handshake is completed when Rin is again risen by the

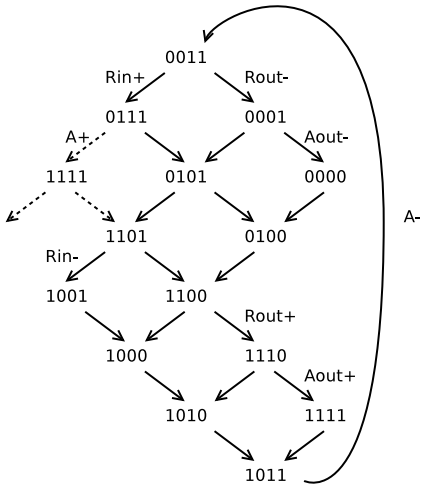

Fig. 19. State graph generated from figure 18 [4]

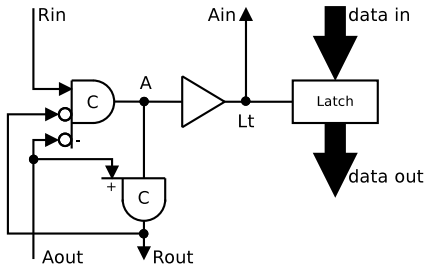

Fig. 20. Semi-decoupled control circuit [4]

predecessor stage, as a consequence of the falling edge on Ain. To simplify the following formulas, the path leading from the bottom C-element ($celem2$) to the the left C-element ($celem1$) is assumed to be faster than any path involving a neighbouring stage.

The cycle time is thus given by

$$T = max((t^i_{celem1} + t^i_{buf} + t^{i-1}_{celem1} + t^{i-1}_{celem2} + t^i_{logic\downarrow}), \\ (t^i_{celem1} + t^i_{celem2} + t^i_{logic\uparrow} + t^{i+1}_{celem1} + t^{i+1}_{buf})) \\ + t^i_{celem1} + t^i_{buf} + t^{i-1}_{celem1} + t^{i-1}_{celem2} + t^{i-1}_{logic\uparrow}. \tag{16}$$

.

Again assuming equal delays for all C-elements and buffers this gives

$$T = 6 * t_{celem} + t_{buf} + max(t^{i-1}_{logic\downarrow}, t^i_{logic\uparrow}) + t^{i-1}_{logic\uparrow}. \tag{17}$$

E. Timing Constraints

As already mentioned in the previous section, again, the path leading from Rin to Rout has to be slower than the latching delay. This constraint is easily met in general, because there are two C-elements involved in the control path.

F. Initialization

All C-elements have to be initialized to 0.

G. Optimizations

The STG depicted in figure 18 ensures that $A+$ waits for $Rout-$, a condition necessary for speed independent operation. However it's legitimate to assume that the path from A via the bottom C-element ($celem2$) to the left C-element ($celem1$) takes less time than it takes the predecessor stage to toggle Rin. Removing the condition $Rout- \rightarrow A+$ from the STG, adds the state "1111" to the state graph of figure 19, which will never be reached anyway, as already argued. Furthermore the path from $Rout$ to $celem1$ in figure 20 is removed. This simplifies the left C-gate and increases performance.

H. Issues

The drawback of the semi-decoupled controller is the dependence of $Ain-$ and $Aout+$ i.e. it's not possible to complete the input handshake cycle, by acknowledging the return to zero, as long as $Aout$ has not gone high. Thus the input and the output side of the circuit are still linked, resulting in possible performance loss when used with processing logic. A solution for this problem is called fully-decoupled latch control circuit and will be presented in the next section.

FULLY-DECOUPLED 4-PHASE LATCH CONTROL CIRCUIT

A. Introduction

To overcome the problem of the semi-decoupled controller, where the input handshake cannot be completed until $Aout$ is risen by the following stage, Furber and Day propose another 4-phase pipeline controller, termed "fully-decoupled" [4]. As the name suggests, the input and the output handshake cycle do not depend on each other any more, when this more advanced but also more complex controller is used.

B. Structure

Again the STG, state graph and a possible implementation are illustrated in figures 21, 22 and 23. To finally achieve full decoupling of the input and the output handshake, the new state variable B is added. It's now possible to perform $Ain-$, i.e. to pull Ain low, independent of the occurrence of $Aout+$. Thus the reset phase of the input handshake can be completed, regardless whether $Aout$ has already been pulled up by the successive stage, or not. The STG already suggests, that both handshakes can be interleaved in many different ways and this is also reflected by the generated state graph. Again Lt is omitted in the state graph since it follows directly from A, as illustrated in the resulting circuit.

In the initial state all control lines, as well as the internal nodes A and B, are low and the latches are open. On $Rin+$, A goes high. This closes the latches ($Lt+$), pulls up Ain and B via the top and the left C-gate, respectively, and causes $Rout+$. With B being high, the top C-gate is armed to immediately change its output to low on $Rin-$. Hence, finally the input handshake cycle is truly independent of the output handshake cycle. Of course the data has to be acknowledged by the successive stage before new data can be latched. Assume Rin and, consequently, Ain have gone low. Then a rising

edge on $Aout$ will propagate trough the circuit as follows: A is pulled low by the middle C-gate, thus $Lt-$ and $Rout-$ follow. The left C-gate then pulls B low. Now two transitions are possible. Firstly $Aout-$, then the circuit is again in its initial state. Secondly Rin could rise. Thereupon, $A+$, $Lt+$, $Ain+$ and $B+$ follow, but $Rout$ still stays low. $Rin+$ now has no immediate effect because B will stay high as long as $Aout-$ has not taken place. This behaviour ensures that data propagates correctly through the pipeline.

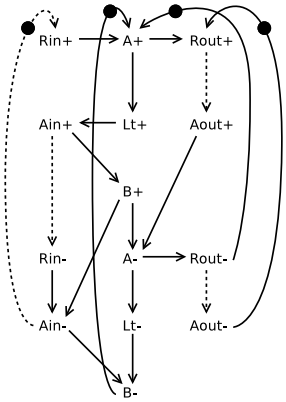

Fig. 21. STG of the fully-decoupled 4-phase latch control circuit [4]

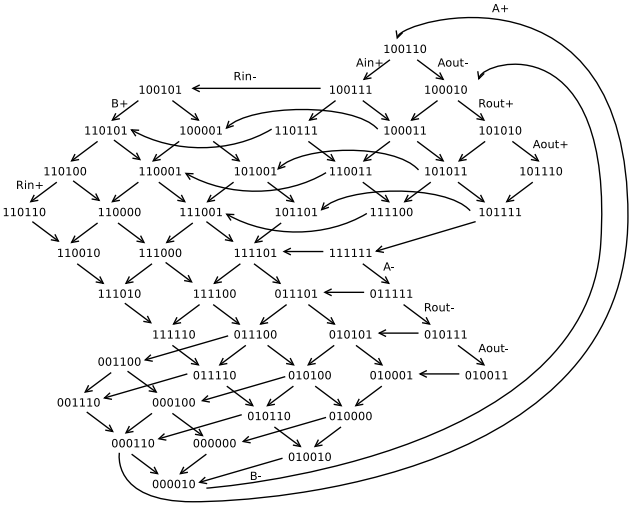

Fig. 22. State graph corresponding to 21 [4]

C. Protocol

The communication protocol used by two neighbouring stages is the standard 4-phase protocol introduced in section "2-PHASE vs. 4-PHASE".

D. Performance

There are several transitions depending on an internal event as well as on an external event. Internal, in this context, means

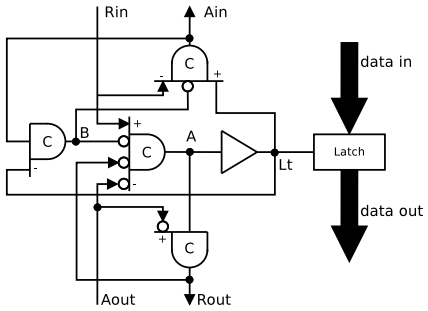

Fig. 23. Fully-decoupled control circuit [4]

just A or B. A typical example can easily be seen in the STG where $Ain-$ is dependent on $B+$ (internal event) and $Rin-$ (external event). Its reasonable to omit these details, when determining the cycle time, for several reasons. Firstly, in any implementation an external event will take longer than an internal event because there are more components involved and the external event still has to propagate trough the delay line. Secondly, this assumption is not a timing constraint, i.e. the consequence of the external event would just be delayed until the internal event has finished. Thirdly, if all parallel paths would be considered in the cycle time, this would result in a very complex formula, not giving any additional information for the reasons mentioned.

Using $t_{ctop}, t_{cmiddle}$ and $t_{cbottom}$ for the delays of the C-elements, the cycle time is given by

$$T = t^i_{cmiddle} + t^i_{buf} + t^i_{ctop} + t^{i-1}_{cbottom} + t^{i-1}_{logic\downarrow} + t^i_{ctop} + t^{i-1}_{cbottom} + t^{i-1}_{logic\uparrow} \tag{18}$$

.

Again assuming equal delays for all C-elements this gives

$$T = 5 * t_{celem} + t_{buf} + t^{i-1}_{logic\downarrow} + t^{i-1}_{logic\uparrow}. \tag{19}$$

E. Timing Constraints

As in the semi-decoupled version of the circuit, the latch delay has to be shorter than the delay from Rin to $Rout$.

F. Initialization

On initialization the outputs of all C-elements, and therefore all internal nodes as well as all control lines, are low, and the latches are open.

G. Optimizations

Again the condition $Rout- \rightarrow A+$ may be removed from the STG depicted in figure 21. The same conditions, as well as the same consequences, already described for the semi-decoupled controller also hold, slightly adopted though, for the fully-decoupled version.

H. Issues

For increasing the decoupling between neighbouring stages, additional variables had to be added to the STG, resulting in a more complex control circuit. The consequences are higher power and area consumption.

DOUBLY-LATCHED ASYNCHRONOUS PIPELINE

A. Introduction

Kol and Ginosar describe in [5] the doubly-latched asynchronous pipeline (DLAP). DLAP employs a 4-phase communication protocol between the stages and each stage consists of two storage elements. Hence, the operation of a synchronous master-slave pipeline is imitated. This enables truly decoupled operations, which is an outstanding characteristic of the proposed design. No other static pipeline in this paper has this unique capability.

Either edge-triggered registers or transparent latches can be used as storage elements. For both storage elements a stage controller for DLAP is proposed in [5].

B. Structure

Figure 24 shows a single DLAP stage. As already mentioned, such a stage consists of two storage elements, a master and a slave element. The storage elements are controlled via a controller, which communicates via Request (Rin, $Rout$) and Acknowledge (Ain, $Aout$) with neighboring stages. The semantics of Request and Acknowledge follows the traditional 4-phase protocol. The storage elements are controlled via Lm and Ls and the Done signals (Dm, Ds) indicate when the operation finished.

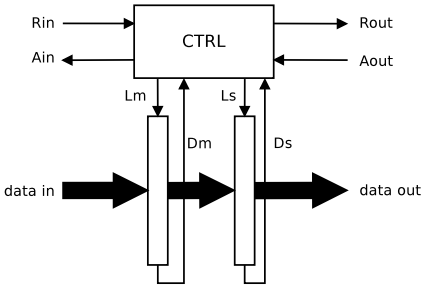

Fig. 24. A DLAP stage [5]

C. Protocol

The choice, which storage elements are used, influences the stage-controller. Edge-triggered registers lead to simpler stage-controllers, where transparent latches allow the usage of simpler storage elements. In the following two sections both alternatives shall be eloberated.

Edge-Triggered DLAP

Figure 25 shows the STG [1] of an edge-triggered DLAP stage-controller. Kol and Ginosar used Petrify [2] to ensure that the STG can be used to implement the controllers as speed

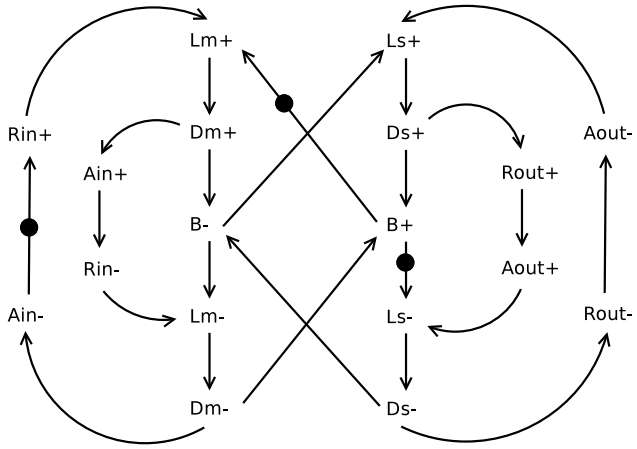

Fig. 25. STG of edge-triggered DLAP [5]

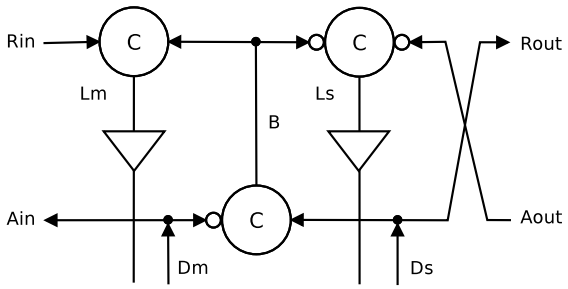

Fig. 26. Implementation of edge-triggered DLAP [5]

independent circuits. Moreover Petrify synthesized the stage-controller to the circuit shown in figure 26.

The general behavior is as follows: Lm is rised, and thus the master register gets transparent to capture data, when new data arrives at the stage (Rin is high) and the slave register is empty (internal signal B, for bubble, is high). Ls goes up, and thus the slave register gets transparent, when new data is in the master register (B is low) and the master register of the next stage is empty ($Aout$ is low).

In the initial state Lm is low (master register is empty) and Ls is high (slave register is full). Thus, Dm is low and Ds is high, therefore the middle Muller-C gate drives B to high. Two parallel and independent events can now happen. 1) Rin goes high, indicating the arrival of new data. Hereon Lm goes high, issuing the master register to capture the new data. After this capturing is done, Dm rises, which acknowledges the recieved data. 2) $Aout$ rises, indicating that the data of the slave register are captured in the master register of the next stage. Hence, Ls goes low, after which Ds and thus $Rout$ go low.

After both events happened, the middle Muller-C gate drives B to low. Now again, two parallel events happen. 3) If Rin isn't low yet, it falls, issuing Lm to fall too after which Dm and thus Ain fall too. Hence, the handshake with the predecessor stage is completed. 4) If $Aout$ isn't low, it falls, completing the handshake with the successor stage. Hereupon, Ls rises, issuing the slave register to capture the data from the master register. After this capturing is done, Ds rises and thus

generates a new request for the sucessor stage.

After both events occured, the middle Muller-C gate drives B to high, and thus the controller enters the initial state again.

Latched DLAP

As already mentioned, latches are simpler storage elements than edge-triggered registers. DLAP uses a blocking latch scheme, thus latches are just opened, when new data is available. This saves power, since hazards can't propagate, but as a tradeoff the stage-controllers are more complex than the edge-triggered variants. It's important to notice, that master and slave latches can't be both open at the sime time. Otherwise data would pass through the stage, ignoring the protocol and thus possible overwritting not yet safed data at the next stage. Figure 27 shows the STG of the stage-controller and figure 28 the corresponding circuit, synthesized by Petrifiy. To ensure that the STG has a complete state coding signal, G needs to save, which latch was opened last. Beside this, the protocol is the same as for edge-triggered DLAP.

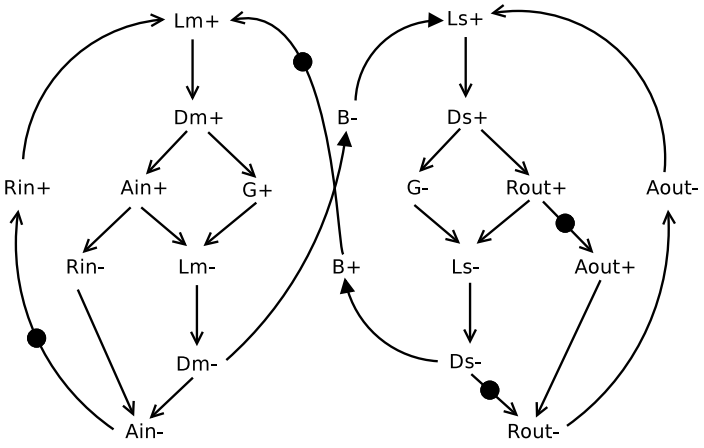

Fig. 27. STG of latched DLAP [5]

D. Performance

DLAP shines, when processing logic between stages is not balanced and conditional matched delays are used. In the case that the processing logic of the predecessor stage is faster than the processor logic of the current stage, the two storage elements in one stage can hold two data items from the predecessor stage. Thus, the predecessor stage will not be paused. Kol and Ginosar give the following example in [5]: In a pipeline consisting of two stages A and B, three tasks (i, j, k) shall be computed. Table I shows how much time each task needs in each pipeline stage. In a synchronous pipeline, the computation needs 8 time units, since the clock has to be adjusted to the worst-case of all calculations over all stages (figure 29 (a)). Since in a semi-decoupled pipeline only every second stage can operate at a time, in this example pipeline, stage A and B have to operate alternating. Hence, 8 time units are needed in a semi-decoupled pipeline too (figure 29 (b)). In a fully-decoupled pipeline, stage A stalls after it executed task j, since stage B is occupied with task i. Hence, the computation needs 6 time units (figure 29 (c)). Since DLAP is truly decoupled, stage A doesn't stall, even when stage B is occupied. Thus, only 5 time units are needed for the computation (figure 29 (d)).

	Task i	Task j	Task k
Stage A	1	1	2
Stage B	2	1	1

TABLE I. PROCESSING TIMES [5]

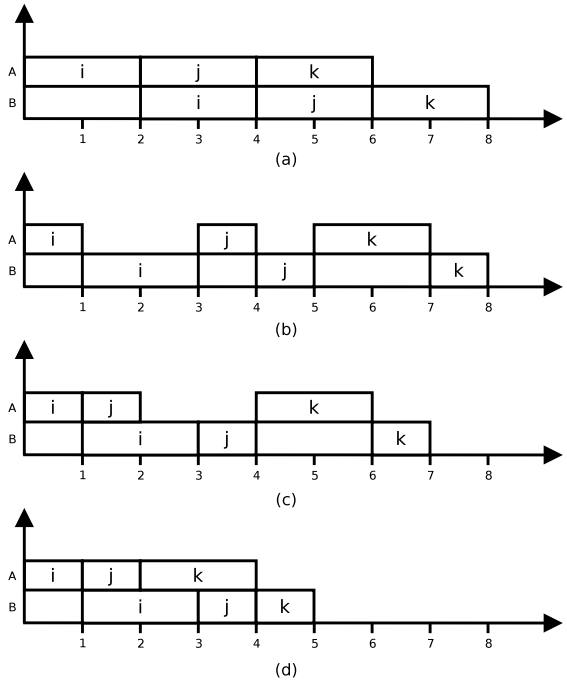

Fig. 29. Time needed in (a) a synchronous pipeline, (b) a semi-decoupled pipeline, (c) a decoupled pipline and (d) DLAP [5]

It's important to notice, that DLAP doesn't solve the problem of unbalanced logic completely. If the processing logic of the predecessor stage is a lot faster, it will stall nevertheless, since the second storage element is eventually full too.

As in the fully-decoupled pipeline, in DLAP all pipelines stage can operate simultaneously. Thus, when much logic is placed between the stages, both designs will operate faster than the semi-decoupled pipeline. DLAP offers one more advantage compared to semi- and fully-decoupled pipelines. It's more tolerable to changes in the output rate from the pipeline than the two mentioned pipelines before [5]. This is due to the fact, that not all storage elements get occupied in the case that the pipeline runs full.

Figure 30 shows, what happens when data is added to DLAP (a) and fully-decoupled (b) but no data gets removed. Note that both storage elements in DLAP stage 1 and 3 store the same data. Thus, both master storage elements can store in principle new data, since the slave elements have already captured the data. Hence, both master storage elements store a so called 'bubble'. In the fully-decoupled pipeline, currently there is no bubble. When data gets removed on the right hand side of the

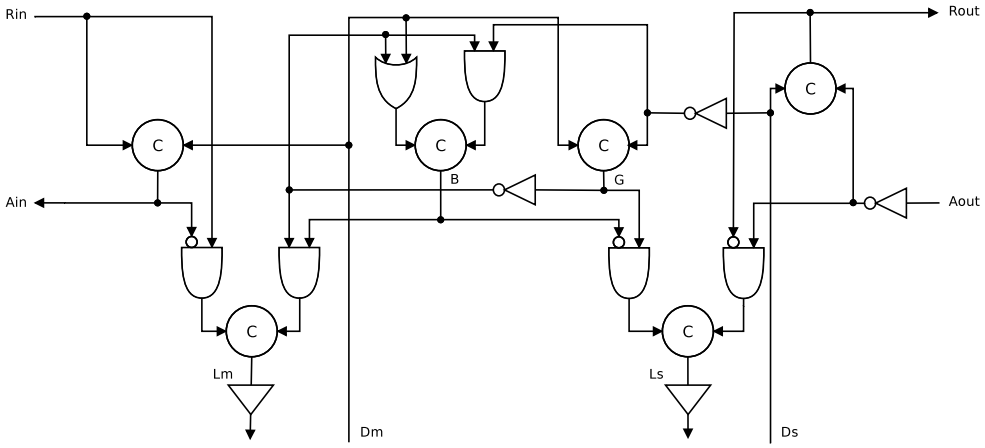

Fig. 28. Implementation of latched DLAP [5]

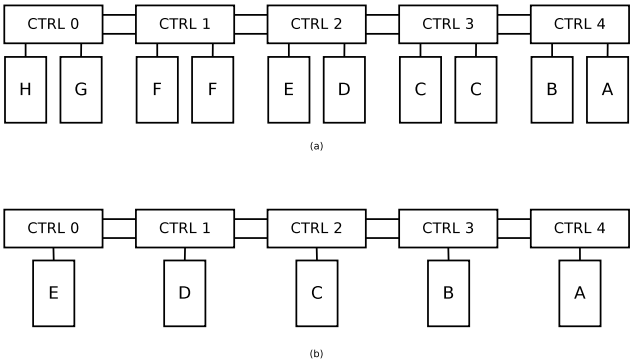

Fig. 30. Full DLAP (a) and full fully-decoupled pipeline (b)

pipeline, a bubble is introduced. In the fully-decoupled pipeline this bubble has to flow backward to enable the pipeline to store new data, which results in a latency overhead relative to the length of the pipe [5]. Since in DLAP more bubbles are available, the coupling between the stages is more relaxed, which automatically leads to the mentioned advantage.

A cycle of a DLAP stage i with registers is as follows: 1) When Rin rises, the master flip-flop in stage i is informed, that it has to capture the new data, 2) the flip-flop captures the data, 3) the slave flip-flop of stage $i-1$ is informed, that it can release the data, 4) the flip-flop releases the data 5) the master flip-flop in stage i is informed, that it can release the new data, 6) the flip-flop releases the data, 7) the slave flip-flop of stage $i-1$ is informed, that it can capture new data, 8) the flip-flop captures the data.

Therefore, if internal paths are assumed to be faster than external paths, the cycle time of DLAP is as follows:

$$T = C^i_{\uparrow} + t^i_{buf} + t^i_{flip-flop} + C^{i-1}_{\downarrow} + t^{i-1}_{buf} + t^{i-1}_{flip_flop} + t^{i-1}_{logic\downarrow} + C^i_{\downarrow} + t^i_{buf} + t^i_{flip-flop} + C^{i-1}_{\uparrow} + t^{i-1}_{buf} + t^{i-1}_{flip-flop} + t^{i-1}_{logic\uparrow} \quad (20)$$

$$T = 2 \cdot (C_{\uparrow} + C_{\downarrow}) + 4 \cdot (t_{buf} + t_{flip-flop}) + t_{logic\downarrow} + t_{logic\uparrow} \quad (21)$$

E. Timing Constraints

The DLAP stage-controllers are delay-insensitive and thus, no additional time constraints, besides the bundling constraints, are required.

DLAP finishes the presenation of 4-phase pipeline-styles. The next two introduced pipeline-styles use pulses for communciation and thus, differ greatly from the previous shown pipeline-styles.

GasP: A Minimal FIFO Control

A. Introduction

A radically new idea, of how static asynchronous pipelines can be designed, is called GasP and is proposed by Sutherland and Fairbanks in [11]. The asP in GasP is the abbreviation for Asynchronous Symmetric Persistent Pulse Protocol - a pipeline style which uses a pulse like protocol to advance data along a pipeline of conventional latches, originally presented in [6].

B. Structure

Every GasP pipeline is built of two kinds of circuits: a PLACE holds data and a PATH controls the flow of data between two PLACEs. A PLACE is simply a storage element that holds data until it can be copied to the the next PLACE as well as some mechanism to signal whether the PLACE holds data or it doesn't. A PATH is basically a single transistor and an associated control circuit to determine whether data may be copied from one PLACE to its successor PLACE in the pipeline. For this purpose it's useful to introduce the terms

FULL for a PLACE holding data and EMPTY for a PLACE not holding data, respectively. Data has to be copied from one PLACE to the next exactly if they are in a FULL-EMPTY state. Afterwards the state is changed to EMPTY-FULL. PLACEs and PATHs alternate along the pipeline and a pipeline stage is one alternation of a PATH and a PLACE.

Figure 31 shows a GasP pipeline consisting of two stages and parts of an additional PLACE to illustrate a connection to its predecessor stage. The state of a PLACE is stored on a single wire - a so called state conductor. To enable the state conductor to hold its value, a keeper is needed. Any PATH will drive the state conductor to its right side low when data has been forwarded and it will drive the state conductor to its left side high when data has been consumed, i.e. LO = FULL and HI = EMPTY.

Initially all state conductors are HI. As data flows through the pipeline the leftmost state conductor will eventually go LO. This drives the output of the inverter $not1$ high and causes the NAND gate consisting of transistors $N4$ and $N5$ in figure 31 to fire, to drive node (1) low and therefore to make the associated PATH transparent. Furthermore the output of $not2$ is high, thus pulling the successive state conductor LO = FULL because transistor $N4$ is enabled. Similarly, the predecessor's state conductor is driven HI = EMPTY via transistor $P1$. Since $P1$ is enabled, the NAND gate will be disabled again three gate delays from the time instant when the NAND gate had fired. In parallel, the same pulse propagates through the self-reset inverter loop, which also has a delay of three transistors. This generates a short positive pulse that closes the paths and positively charges nodes (1), (2), (3) and (4).

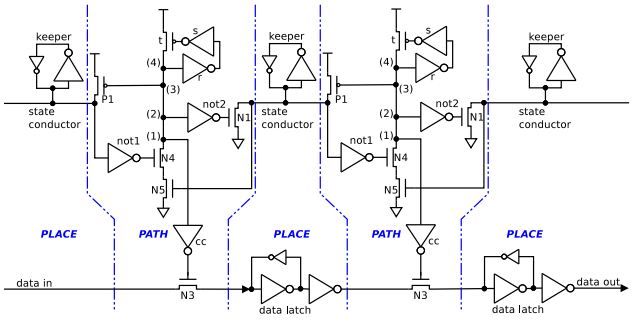

Fig. 31. GasP with self-resetting NAND [11]

Sutherland and Fairbanks claim that the use of the self-reset has very low logical effort. However, the resulting circuit, as it is presented, is not static because e.g. node (2) might drift low when it's not driven for some time and thus erroneously close transistor $N1$. The use of a conventional NAND gate overcomes this drawback. Furthermore it's not easily possible to add processing logic because of the delay line needed. Adding a second control signal between two neighbouring stages is one possibility to solve this problem. A very similar (but slightly adopted to make its operation static) approach is depicted in figure 32. Although $s1$ and $s2$ are driven from both sides, only the right path monitors the state of $s2$ and only the left path monitors the state of $s1$. This is because s1 is used by the right path to indicate that data has been consumed and s2 is used by left path to indicate the availability of new data. Therefore it's now possible to add a delay line between the inverter $not2$ and the transistor $N2$.

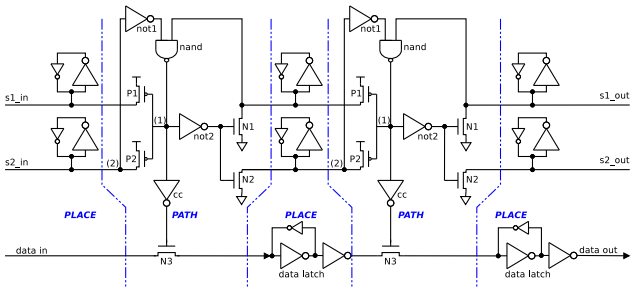

Fig. 32. GasP with twin state conductors and conventional NAND [11]

C. Protocol

Contrary to many other pipelines presented so far, GasP uses pulses for the communication of two neighbouring stages. A negative pulse indicates the availability of new data whereas a positive pulse acknowledges the data and thus indicates that the data has been consumed. This is true for the implementation presented, however the meaning of positive and negative pulses might differ if another implementation is used.

D. Performance

To determine the cycle time, a delay line has to be added at the input of transistor $N2$ of the circuit shown in figure 32.

The cycle starts when a positive pulse arrives at $s2in$ and is finished when $s2in$ may become high again. Assuming the pipeline was empty and thus s1out was high. When node (2) is pulled low both inputs of the NAND gate are high and node (1) is pulled low. This has several consequences:

1) The associated PATH is opened.
2) The state conductor for $s1in$ is pulled high, i.e. the data is acknowledged.
3) The state conductor for node (2) is pulled high.
4) $s1out$ is pulled low, i.e. the state of the following PLACE is set to full.
5) A request is sent to the next stage by driving $s2out$ high.
6) As a consequence of node 2 becoming high and $s1out$ becoming low, the output of the NAND gate becomes high. The gate is resetted and the corresponding PATH is closed.

The cycle time is given by:

$$T = t^i_{not1} + t^i_{nand} + t^i_{P1} + t^{i-1}_{nand} + t^{i-1}_{not2} + t^{i-1}_{logic} + t^i_{N2} \quad (22)$$

resulting in

$$T = 2 * t_{not} + 2 * t_{nand} + t_{P1} + t^{i-1}_{logic} + t_{N2} \quad (23)$$

E. Timing Constraints

The pulses generated by the self-resetting mechanism have to be long enough to override the value stored in the corresponding state conductor. This can be ensured by using proper sizing.

F. Initialization

On initialization all state conductors are high, i.e. empty. Furthermore the output of the self resetting NAND of figure 31 has to be high.

The same holds for the circuit depicted in figure 32. However, for this circuit the output of NAND gate is automatically forced high by properly initializing all state conductors.

G. Issues

The simple n-type PATH transistor $N3$ used is not well suited for driving a logical high on the data path. Transmission gates might be preferable.

INTERLOCKED PIPELINED CMOS

A. Introduction

Interlocked pipeline CMOS (IPCMOS) [7] is a unique approach to implement a bundled data pipeline. Instead of traditional 2- or 4-phase protocols, IPCMOS uses positive pulses for communication between pipeline stages. Furthermore, it implements blocking latch scheme to reduce power consumption. In combination with specialised stage controllers, IPCMOS is able to perform high speed operations. IPCMOS is especially useful in nonlinear pipelines. A nonlinear pipeline is a pipeline, in which the stages have multiple predecessors and/or successors. For example in figure 33, stage D has three predecessor- and two successor stages.

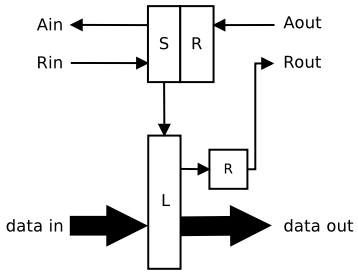

Fig. 33. IPCMOS pipeline [7]

B. Structure

Figure 33 shows an IPCMOS pipeline stage. Each stage consists of a stage controller and a latch. As other asynchronous pipelines, each stage communicates with neighboring stages via request and acknowledge signals. Moreover, each stage controller consists of three parts: 1) a strobe circuit, waiting for the requests from predecessor stages, controlling the latch via $clken$[2] and generating acknowledge pulses for the predecessor stages, 2) a reset circuit, waiting for the acknowledgements from successor stages, 3) a request circuit, generating the request pulses for the successor stages. Each circuit will be discussed in detail in the next section.

As already mentioned, IPCMOS is advantageous for nonlinear pipelines. The reason is the way, how IPCMOS joins the different request (acknowledge) signals of the predecessor (successor) stages. To join different predecessor stages, conventional pipelines use Muller C-elements, in which the transistor stacks grow with every predecessor stage. The IPCMOS strobe circuit avoids this problem via a static NOR and thus reduces the transistor stack depth to 1. Hence, performance doesn't get as much degraded as in other designs when the number of predecessor and successor stages increases.

C. Protocol

Since IPCMOS is primary designed as a nonlinear pipeline, the protocol and the circuits shall be discussed for block D in figure 34. Figure 35 shows a timing diagram with the most important signals.

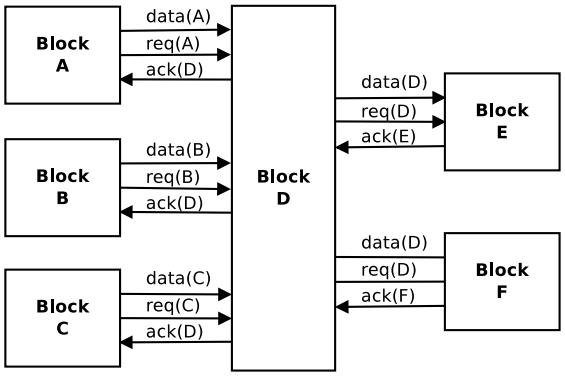

Fig. 34. An abstract IPCMOS stage with multiple predecessors and successors [7]

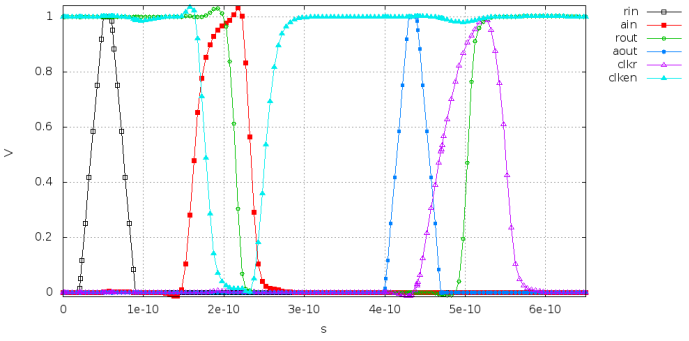

Fig. 35. Timing diagram of an IPCMOS stage

Strobe

Figure 36(a) and (b) show the strobe circuit, consisting of the

[2]Although we are talking about latches and asynchronous circuits, this is the notation used in [7] which we follow.

previous mentioned static NOR, formed by transistors a-e and a switch for every predecessor stage. Due to the static NOR, the pipeline stage only captures new data, when all internal signal $Vint(A)$ - $Vint(C)$ and $Rint(D)$ are low. This happens, when all predecessor stages announced that new data is available via a positive pulse on the req lines, and all successor stages signalled that the previous data is captured via a positive pulse on the ack lines. The Reset-Block handles the incoming pulses on the ack line and the detailed function will be discussed in the next section. Note, that in the original design from Schuster and Cook in [7], the dashed storage loop at $Rint(D)$ is missing. Probably it was forgotten to draw it, since in our opinion the loop is needed to ensure the intended functionality.

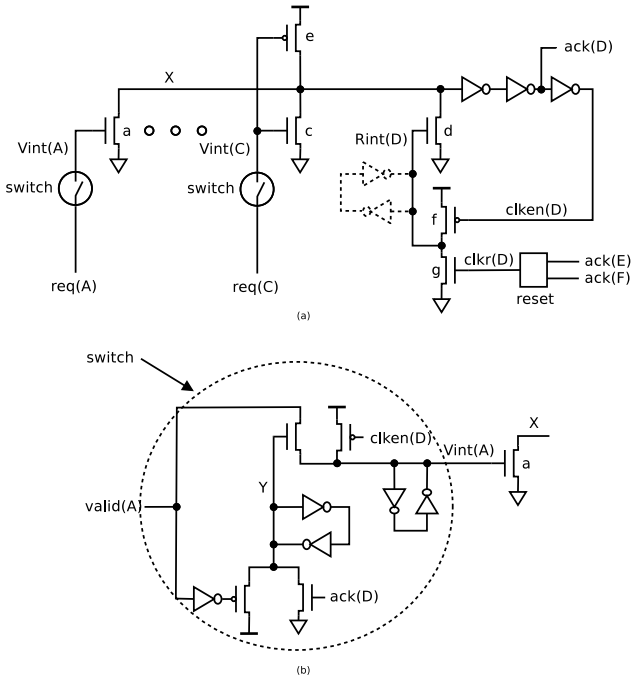

Fig. 36. (a) Strobe circuit (b) Strobe circuit switch [7]

In the initial state, all external req and ack signals are low, the internal signals $Vint(A)$ - $Vint(C)$ are high and $clkr(D)$ and $Rint(D)$ are low. Since $Vint(A)$ - $Vint(C)$ are high, internal signal X is low, thus $ack(D)$ is low and $clken(D)$ (clock enable negative) is high. On a rising edge of an external req signal, the corresponding switch gets closed (Y is driven high), but the corresponding internal signal $Vint$ stays high. On the falling edge of the external req signal, $Vint$ goes low. Thus a positive pulse occurred on this req line, the corresponding predecessor signalled that new data is available. When this pulse occurs on all req signals, internal signal X goes high. Thus, $ack(D)$ goes high and $clken(D)$ goes low (opens the latch). Furthermore, all switches get opened, since Y is reset to low and all $Vint(A)$ - $Vint(C)$ are set to high. Simultaneously $Rint(D)$ gets driven to high by $clken(D)$. After this, signal X is high, and thus $ack(D)$ and $clken(D)$ are in the initial value again, closing the latch again and thus capturing the new data. It's important to notice, that a high $Rint(D)$ prevents the stage to capture new data, since internal signal X is fixed to low. When all successor stages acknowledged the actual data, $clkr(D)$ gets driven high by the reset circuit and thus, resetting $Rint(D)$ to low, enabling the strobe circuit to capture new data.

Another important aspect about the strobe circuit is, that only one internal $Vint$ signal controls the pull-up transistor of the internal X signal. Hence, the last request pulse should occur on the corresponding req line. With appropriate transistor sizing, the circuit works correct, even if this critera isn't met. If the pull up transistor is weak, the pull down transistors of the internal signals, which haven't detected a request pulse yet, will keep X low.

Reset

In the reset circuit (see figure 37(a) and (b)), like in the strobe circuit, a static NOR is used to wait that on every ack line a positive pulse occurs, signalling that the corresponding successor stage captured the previous data. Hence, only when a positive pulse occurred on all ack lines, $clkr$ goes high, driving $Rint(D)$ in the strobe circuit low, enabling the stage to capture new data. How the reset circuit is connected with the strobe circuit is shown in Figure 36(a).

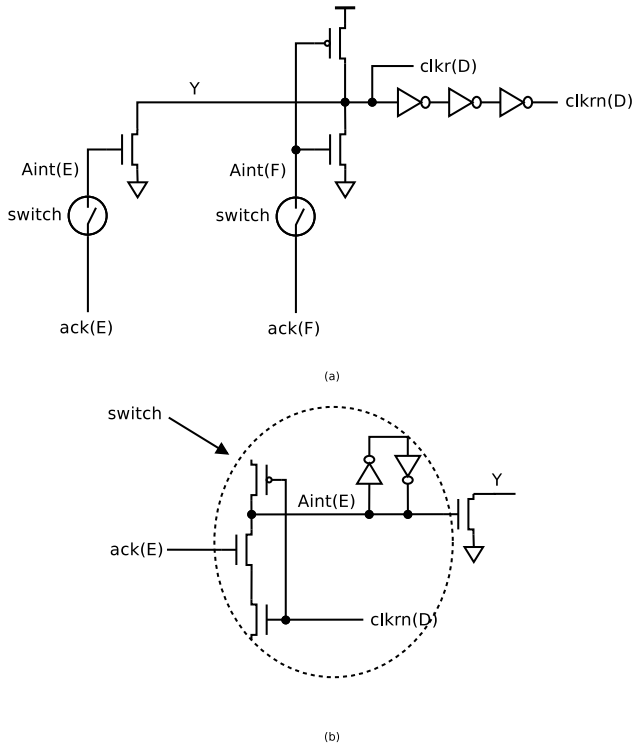

Fig. 37. (a) Reset circuit (b) Reset circuit switch [7]

In the initial state, the external signals $ack(E)$ and $ack(F)$ are low and the internal signals $Aint(E)$ and $Aint(F)$ are high. Thus, Y and $clkr(D)$ are low and $clkrn(D)$ is high. On rising edges on the ack line the corresponding $Aint$ gets high. When both internal signals $Aint(E)$ and $Aint(F)$ go high, Y and thus $clkr(D)$ will become high and $clkrn(D)$ low. A falling $clkrn(D)$ resets $Aint(E)$ and $Aint(F)$ to high

and thus, enables the reset circuit for a new cycle.

Latch

One possibility for a latch used in IPCMOS is shown in figure 38. It consists of a transmission gate, which lets data pass when $clke$ is high, and a storage loop, which is parallel to the data flow. With this design, only very little delay is added to the data path [7].

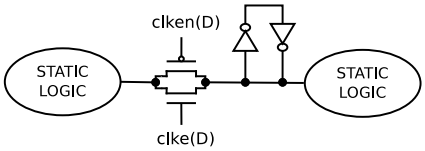

Fig. 38. Possible latch [7]

Request

A positive request pulse shows the successor stages, that new data is valid and ready to capture. Figure 39 shows the request circuit, consisting of two transistors, an inverter and a storage loop. Again the dashed inverter isn't present in the original design in [7], but in our opinion needed for correct functionality. At the beginning of a cycle, the storage loop drives the req signal high. When the strobe circuit detects that new data is available (every predecessor stage send a positive request pulse), it sets $clken$ to low for a short period of time. Hence, req is driven to low too, thus, completing the sending of a positive request pulse to the successor stages, indicating new data. After every successor stage acknowledged this new data with a positive acknowledge pulse, the reset circuit sets $clkr$ to high for a short period of time. Thus completing a cycle and resetting the request pulse to high again.

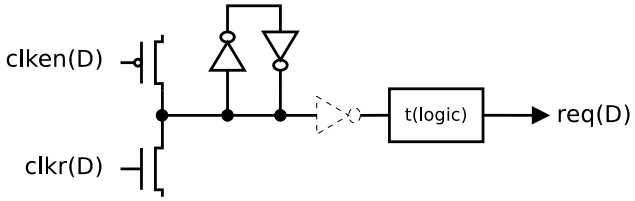

Fig. 39. Request circuit [7]

D. Performance

A cycle of stage i in a linear IPCMOS pipeline is as follows: 1) on the req line of stage i a positive pulse occurs, 2) stage i captures the new data 3) stage $i+1$ is informed, that new data is available with a positive pulse, 5) stage $i+1$ captures the new data, 6) with a positive pulse, stage i is informed, that the next data can be stored. Therefore, the cycle time of IPCMOS is as follows:

$$\begin{aligned} T = {} & t^i_{switch\_strobe} + t^i_{static\_nor} + 2 \cdot t^i_{inv} \\ & + max((t^{i-1}_{switch\_reset} + t^{i-1}_{static\_nor} + t^{i-1}_{Rint\_clear} + t^{i-1}_{logic}), \\ & (t^i_{inv} + t^i_{req} + t^i_{logic} + t^{i+1}_{switch\_strobe} + t^{i+1}_{static\_nor} \\ & + 2 \cdot t^{i+1}_{inv} + t^i_{switch\_reset} + t^i_{static\_nor} + t^i_{Rint\_clear})) \end{aligned} \tag{24}$$

E. Timing Constraints

One important timing constraint is the pulswidth of the $clken$ signal, which is determined by:

$$T_{CLKEN\_high} = t_{pmos} + t_{nmos} + 3 \cdot t_{inverter} \tag{25}$$

On one side, $clken$ has to stay high long enough, that the newly arriving data can be safely captured, i.e. the setup-time of the storage element.

$$T_{CLKEN\_high} > t_{setup} \tag{26}$$

But on the other side, $clken$ mustn't stay high so long, that new data arriving from the predecessor stage gets captured in the same cycle too.

$$\begin{aligned} T_{CLKEN\_high} < {} & t_{switch\_reset} + t_{static\_nor} + t_{Rint\_clear} + \\ & t_{logic} \end{aligned} \tag{27}$$

IPCMOS is the last presented pipeline-style. In the next chapter, we show the result of the simulation of all presented pipeline-styles, and compare them regarding different performance parameters.

SIMULATION RESULTS

A. General Information

All pipeline styles described in the previous sections, except for the Sutherland micropipeline, were implemented and simulated using Spice, HSpice in particular. For this purpose five stages of each pipeline style as well as four delay lines in between and a tester circuit were connected as indicated in figure 40. Thus, the test circuit consists of:

- **5 stages:** For each pipeline style five identical stages were used.
- **4 delay-lines:** Between two neighbouring stages the logic's effect on performance is subsumed in a delay element. However, the logic itself was neither implemented nor simulated because it's standard CMOS logic well known from synchronous design. The complexity of the pipelines operation can be varied by adjusting the delay lines. E.g. by setting the delay of all delay lines to 0, the pipeline behaves like a FIFO storage. For four phase pipelines it's beneficial to use asymmetric delay lines. For two phase pipelines as well as GasP and IPCmos, symmetric delay lines have been used. To ease implementation and also automatic simulation, the delay lines were not implemented in hardware but abstracted using Voltage Controlled Voltage Sources (VCVS) provided by HSpice. A symmetric as well as an asymmetric version of the delay line are depicted in figure 41.
- **tester circuit:** A special circuit generates appropriate input signals depending on the pipeline style (e.g. two-pase, four-phase, pulses) and measures results.

The values of interest are the forward latency, backward latency and the throughput.

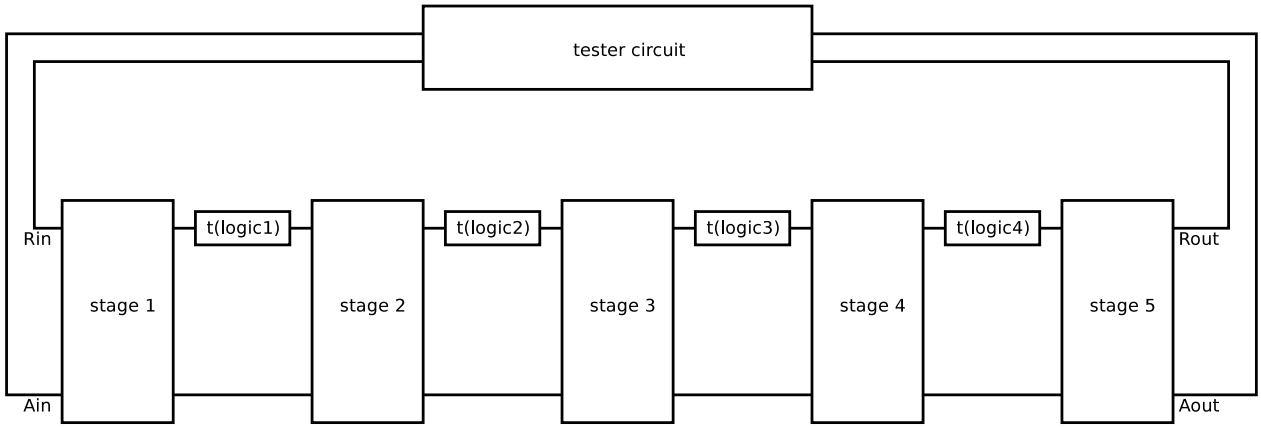

Fig. 40. Set-Up used for simulation

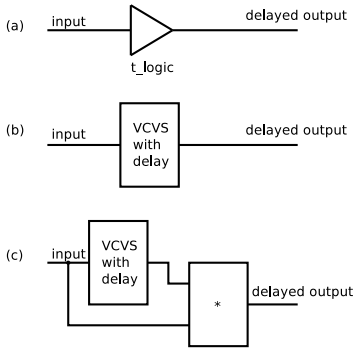

Fig. 41. (a) abstracted delay line (b) symmetric delay line (c) asymmetric delay

B. Forward latency

Using figure 40 the forward latency is easily obtained:

$$T = t_{stage1} + t_{stage2} + t_{stage3} + t_{stage4} + t_{stage5} + \\ + t_{logic1} + t_{logic2} + t_{logic3} + t_{logic4}. \quad (28)$$

For equal delays of all stages and all delay lines this gives:

$$T = 5 * t_{stage} + 4 * t_{logic} \quad (29)$$

Thus increasing the delay introduced by the the delay lines has the same effect on all pipelines and the forward latency is determined only by the complexity of the stage itself. This is also reflected by the simulation results given in table II and figure 42

C. Backward latency

To measure backward latency the pipelines were fully filled with data items by not acknowledging a certain number of requests. Then a single data item was acknowledged and the time span between an event on Aout and the corresponding event on Ain in figure 40 was measured. For all pipelines, except for the one using semidecoupled pipeline controllers, no logic delay-lines (t_{logic}) are involved in the path leading from Aout to Ain and thus the backward latency is constant. The exact results are given in table III and figure 43

Pipeline \ t_{logic}[ps]	0	50	100	200
phaseconversion	1.46	1.60	1.80	2.21
sutherlandwithDETDFF	0.70	0.90	1.10	1.50
mousetrap	0.49	0.69	0.89	1.29
simple	0.17	0.43	0.60	0.98
semidecoupled	0.45	0.72	0.88	1.26
fullydecoupled	0.53	0.80	0.96	1.34
dlap	1.33	1.57	1.73	2.13
GasP	0.34	0.54	0.74	1.14
ipcmos	0.67	0.87	1.07	1.47

TABLE II. SIMULATION RESULTS FOR FORWARD LATENCY. VALUES GIVEN IN NS.

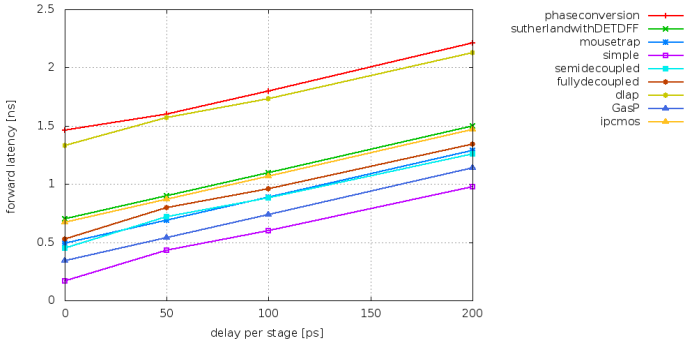

Fig. 42. Forward latency.

D. Throughput

The throughput was measured while constantly feeding the pipeline with new data items by generating events on Rin as soon as the latest item had been acknowledged on Ain. For each pipeline style the time needed to process exactly 20 data items is given in table IV and figure 44. Thus low values in the table and the graph indicate high performance for a specific pipeline and a specific logic delay per delay-line.

CONCLUSION

According to the simulation results, GasP is best pipeline regarding throughput. However, working with pulses results in high effort regarding sizing an positioning. Using two-phase communication and DETDFF also results in very high

Pipeline $\backslash t_{logic}$[ps]	0	50	100	200
phaseconversion	2.58	2.58	2.58	2.58
sutherlandwithDETDFF	0.33	0.33	0.33	0.33
mousetrap	0.71	0.71	0.71	0.71
simple	0.40	0.40	0.40	0.40
semidecoupled	1.07	1.37	1.54	1.92
fullydecoupled	1.73	1.73	1.73	1.73
dlap	1.28	1.28	1.28	1.28
GasP	0.21	0.21	0.21	0.21
ipcmos	0.59	0.59	0.59	0.59

TABLE III. SIMULATION RESULTS FOR BACKWARD LATENCY. VALUES GIVEN IN NS.

Pipeline $\backslash t_{logic}$[ps]	0	50	100	150	200	300	400	500
phaseconversion	16.5	17.5	18.5	19.5	20.5	22.5	24.5	26.5
sutherlandwithDETDFF	4.4	5.3	6.3	7.3	8.3	10.3	12.3	14.3
mousetrap	5.2	6.3	7.3	8.3	9.3	11.3	13.3	15.3
simple	5.1	7.4	9.0	10.9	12.8	16.8	20.8	24.8
semidecoupled	6.9	9.6	11.3	13.1	15.1	19.1	23.1	27.1
fullydecoupled	9.1	10.4	11.3	12.2	13.2	15.2	17.2	19.2
dlap	8.1	9.7	10.5	11.3	12.3	14.2	16.2	18.1
GasP	2.4	3.4	4.4	5.4	6.4	8.4	10.4	12.4
ipcmos	8.1	10.2	11.8	13.1	14.1	17.6	20.1	23.1

TABLE IV. SIMULATION RESULTS FOR THROUGHPUT. VALUES GIVEN IN NS.

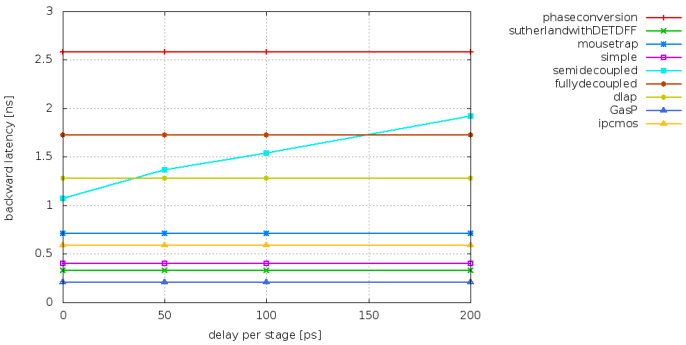

Fig. 43. Backward latency.

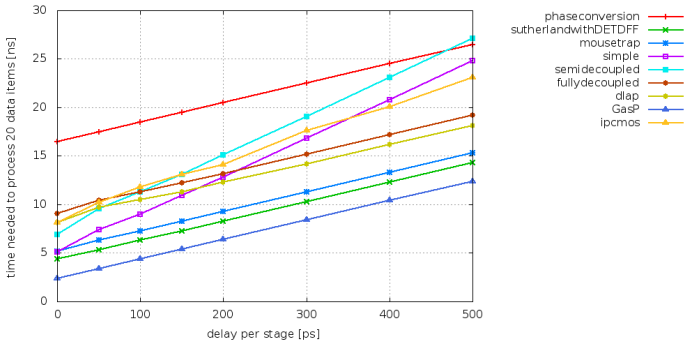

Fig. 44. Throughput

performance but at the price of complex storage elements, a drawback that is overcome by mousetrap. The data also shows an interesting fact about decoupling: The fully decoupled pipeline can gain full advantage of the asymmetric delay line while the simple and the semidecoupled pipeline suffer from the fact that only every other pipeline stage can perform calculations. This performance drawback is also reflected by the cycle times for the simple and the semidecoupled controllers where the logic delay is included twice. Thus even the naive approach of using two-phase communication and a phase conversion circuit outperforms the simple as well as the semi decoupled pipeline when the complexity of the logic increases. On the other hand the fully decoupled approach is a bad choice for FIFO pipelines or pipelines with low logic complexity because of the overhead introduced by the rather complex controller. The controller for the dlap pipeline is simpler and thus slightly faster than the one of the fully decoupled pipeline but still implements the same functionality. However, a tradeoff between the minor increase in performance and doubling the latch count per stage has to be made. Last but not least, at first sight ipcmos performs really bad, especially when the logic effort for the controller and the implementation effort necessary because of the use of pulses is considered. However, ipcmos is better suited for more complex pipelines including branches than for the pipeline used here, which is a major reason for its bad performance.

Any of the presented pipeline styles has advantages and disadvantages compared to other styles and it's impossible to state the "best" synchronous pipeline style. Nevertheless, given certain characteristics (e.g. complex logic vs. FIFO, every or just every other pipeline stage filled, etc.) and preferences (high speed, small area, low power, etc.) this document provides the knowledge to choose an appropriate solution.

REFERENCES

[1] Tam-Anh Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 1987.

[2] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alex Yakovlev. Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers, 1996.

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

[4] S.B. Furber and P. Day. Four-phase micropipeline latch control circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 4(2):247–253, 1996.

[5] R. Kol and R. Ginosar. A doubly-latched asynchronous pipeline. In *Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings., 1997 IEEE International Conference on*, pages 706–711, 1997.

[6] C.E. Molnar, I.W. Jones, W.S. Coates, J.K. Lexau, S.M. Fairbanks, and I.E. Sutherland. Two fifo ring performance experiments. *Proceedings of the IEEE*, 87(2):297–307, 1999.

[7] S.E. Schuster and P.W. Cook. Low-power synchronous-to-asynchronous- to-synchronous interlocked pipelined cmos circuits operating at 3.3-4.5 ghz. *Solid-State Circuits, IEEE Journal of*, 38(4):622–630, 2003.

[8] M. Singh and S.M. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000) Proceedings. Sixth International Symposium on*, pages 198–209, 2000.

[9] M. Singh and S.M. Nowick. Mousetrap: High-speed transition-signaling asynchronous pipelines. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(6):684–698, 2007.

[10] Jens Sparso and Steve Furber. *Principles of Asynchronous Circuit Design: A Systems Perspective*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[11] I. Sutherland and S. Fairbanks. Gasp: a minimal fifo control. In *Asynchronus Circuits and Systems, 2001. ASYNC 2001. Seventh International Symposium on*, pages 46–53, 2001.

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

[13] J. Yuan and C. Svensson. High-speed cmos circuit technique. *Solid-State Circuits, IEEE Journal of*, 24(1):62–70, 1989.

[14] K.Y. Yun, P.A. Beerel, and Julio Arceo. High-performance asynchronous pipeline circuits. In *Advanced Research in Asynchronous Circuits and Systems, 1996. Proceedings., Second International Symposium on*, pages 17–28, 1996.