Counterflow Pipelining: Architectural Support for Preemption in Asynchronous Systems using Anti-Tokens

Manoj Ampalam and Montek Singh
Dept. of Computer Science
Univ. of North Carolina, Chapel Hill, NC 27599, USA
Chapel Hill, NC 27599, USA
{manoj,montek}@cs.unc.edu

ABSTRACT

This paper introduces a novel approach to efficiently implement several useful architectural features in asynchronous application-specific ICs (ASICs). These features include *speculation, preemption,* and *eager evaluation,* which have so far only been available on CPUs, and have not been adequately investigated for custom ASICs.

For the efficient implementation of the new architectural features, a radically new approach inspired by Sproull's *counterflow pipelines* [7] is proposed. The key idea is to allow special commands, called *anti-tokens*, to be propagated in a direction opposite to that of data, allowing certain computations to be killed before they are completed, if their results are no longer required.

The net impact is a significant improvement in the throughput of a certain class of systems—*e.g.,* those involving conditional computation—where a bottleneck pipeline stage can often be preempted if its result is determined to be no longer needed. Experimental results indicate that our approach can improve the system throughput by a factor of up to 2.2x, along with an energy savings of up to 27%.

1. INTRODUCTION

This paper presents a novel approach to implementing the useful architectural features of *preemption, speculation* and *eager evaluation* in asynchronous pipelined ASICs. These features have so far been only available on CPUs, but are critical for the efficient implementation of many custom dataflow systems. In particular, preemption allows a data item in a computation pipeline to be destroyed before the computation is complete, if it has been determined to be no longer required, thereby saving energy consumption. Speculation builds upon this idea to allow both branches of a computation to be launched concurrently, while the outcome of the condition is awaited; once the outcome is known, the incorrect branch is preempted. Finally, eager evaluation allows a function block to generate its output using a subset of its inputs when possible, followed by preemption of the inputs that were determined to be not required. All of these architectural features have significant performance and energy benefits.

Our approach is based on a novel form of *counterflow pipelining*, which allows special commands, called *anti-tokens*, to be propagated in a direction opposite to that of data. Our counterflow approach is a radical reworking of the earlier counterflow work by Sproull et al. [7], which introduced a processor architecture where instructions and data flow in opposite directions. While the benefit of their approach is the ease of satisfying data dependencies, a significant limitation is its implementation complexity. In particular, their approach uses two distinct pipelines conveying data and instructions in opposite directions, which require arbitration at every stage to ensure synchronization between the data and the instruction streams. In addition, their approach is specifically targeted to processor design, and not directly applicable to other application-specific architectures.

Another related work is by Brej et al. [2, 1], who first proposed an approach to preemption using anti-tokens. This approach can also be classified as "counterflow" because anti-tokens flow in a direction opposite to the flow of data tokens. When an anti-token meets a data token, it cancels the data computation in that stage, and no output is produced. This work also has significant limitations. In particular, it does not address metastability issues that can arise when a data item and an anti-token reach a pipeline stage near simultaneously. Moreover, their circuit implementation can glitch under certain timing scenarios, and it must therefore rely on certain timing assumptions for correct behavior.

This paper makes two key contributions. First, it introduces a novel counterflow pipeline style that is significantly more efficient than prior work. The pipeline style is based on a novel two-phase counterflow protocol that avoids the need for the complex arbitration of Sproull et al. [7], correctly addresses metastability issues, and unlike Brej et al. [2, 1], guarantees correct operation regardless of the arrival times of data tokens and anti-tokens. Our second contribution is a set of architectural templates that are the building blocks for implementing preemption, speculation and eager evaluation in asynchronous pipelined systems. Detailed experimental evaluation is provided for each of these constructs. Results indicate promising performance benefits: a throughput increase by up to a factor of 2.2x. In addition, the saved computation also translates into reduced energy consumption, with up to 27% energy savings.

The rest of the paper is organized as follows. First, Section 2 provides motivation behind our approach, and briefly reviews prior work. Section 3 then introduces our new counterflow pipeline style, including its formal protocol, implementation, and timing analysis. Then, Section 4 introduces several architectural constructs that are useful for implementing systems with the capabilities of preemption, speculation and eager evaluation. Experimental results are given in Section 5, and finally Section 6 gives conclusions.

ICCAD '06, November 5-9, 2006, San Jose, CA

2. MOTIVATION AND PREVIOUS WORK

2.1 Motivation

Consider the following example: a simple specification consisting of an if-then-else statement.

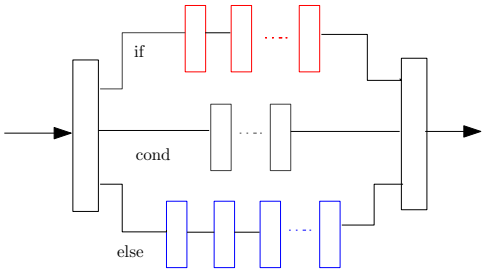

Figure 1: A pipelined dataflow implementation of an "if-then-else" construct

if cond **then**
  easy operation
else
  complex operation
end if

There are two conventional approaches to handling such conditional constructs in asynchronous hardware, and each has its limitations. One approach, which does not use speculation, is simply to first evaluate the Boolean condition, and then execute the correct branch of computation. This approach has the drawback of reduced concurrency, and thereby slower performance.

The second approach, conventional speculation, is to compute both of the branches of the computation concurrently, in parallel with the evaluation of the Boolean condition itself. Once the outcome is known, the result from the correct branch is used; the other result is discarded. While this conventional speculation approach has higher concurrency, and therefore higher performance, is still has two significant drawbacks. First, computation of both branches must finish before the final result is selected. Thus, if the incorrect branch involves a complex long-latency operation, a significant performance bottleneck will be introduced. Second, since both branches of the computation are always executed, much additional energy is consumed.

A variation on the conventional speculation approach above uses pipelining to somewhat increase concurrency, but the overall performance remains quite limited. In particular, often all the three operations—*i.e.,* evaluation of the Boolean condition, the if-branch computation, and the else-branch computation—can be pipelined into multiple stages, as shown in Figure 1. However, this pipelined implementation of speculation still suffers from a performance bottleneck: the final (*i.e.* rightmost) stage typically must wait for all three of its inputs to be available, even though in some situations waiting for either the if-branch or the else-branch would be unnecessary.

Figure 2(a) illustrates this performance bottleneck in conventional speculation. In this example, the if-computation is pipelined into two stages, the Boolean condition evaluation is performed in a single stage, and the else-computation is pipelined into four stages. The six rows represent six successive snapshots of the system's state. The solid blue circles indicate the arrival of an input request (token) at that point in the pipeline. Thus, the top picture represents the instant a new request is fed into the system (*i.e.,* a new execution of the if-then-else construct), and the bottom picture represents the completion of this round of computation. Since the condition requires only one stage for completion, its output is available in the second time step. Similarly, the results of the *if* and *else* branches of computation are available in the third and fifth time steps, respectively. If the condition is evaluated to be true, only the results of the if branch are actually needed in the algorithm, and all information required for generating the output is available at the third step. However,

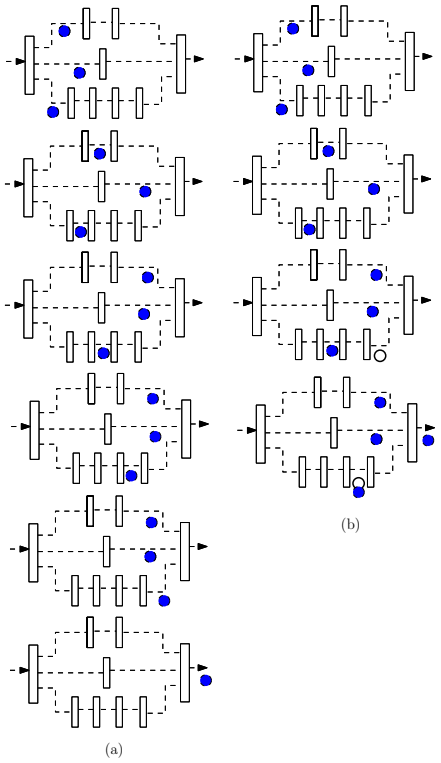

Figure 2: Implementing speculation: (a) traditional approach (without antitokens), and (b) our approach (with antitokens)

in order to preserve the consistency of the asynchronous request-acknowledge handshake protocol, the rightmost stage cannot truly complete this cycle of computation until the result from the else-computation branch is also available. Thus, a slow else-computation branch can slow the entire system down, even in situations when the result of the else branch would be rarely utilized.

Figure 2(b) illustrates how our counterflow approach is applied to implement speculation much more efficiently. Once again, suppose that the Boolean condition evaluates to true in the second time step. Thus, it is determined by the end of the second time step that the result of the else-computation branch is to be discarded. Our counterflow approach introduces the capability to inform the else-computation branch that its result is not required anymore. As a result, the rightmost stage is now freed up to immediately relay the result of the if-computation branch to the output environment, and a complete new cycle of computation can begin without waiting for the else-computation to be completed. In the figure, the hollow circle indicates an antitoken flowing opposite to the flow of data in the $else$ branch. This new approach to speculation has a significant performance benefit. In particular, since the output from the if branch is available in the third step, the final output is produced in the fourth step itself. Thus, this strategy has reduced the latency of a single computation from 6 time units to 4 units.

In general, our counterflow approach is useful for three key applications:

- **Speculation:** We saw above how the counterflow approach makes the implementation of speculation much more efficient. Similarly, switch/case statements and multiplexers can also be efficiently handled.
- **Preemption:** Preemption is useful for providing architectural support for handling exceptions and interrupts: pending operations must be killed before handling the exception.

- **Early Output:** This feature, also called "eager evaluation" or "short-circuit evaluation," allows a pipeline stage to generate a result before all of the inputs to that stage are available, provided the result can be computed from only the available inputs. For instance, if a multiplier function block receives an operand with value zero, a zero output can be immediately generated, without waiting for the other operand to arrive.

The idea of issuing an antitoken along the unwanted branch is useful in two ways. First, it aids in increasing the throughput by reducing the cycle time. Second, it aids in power saving by preventing the unwanted requests flowing through the pipeline and hence unwanted computations.

2.2 Previous Work

Counterflow Pipeline Processor (CFPP). The notion of counterflow was first introduced by Sproull et al. for the design of the Sun Counterflow Pipeline Processor [7]. Unlike our proposed approach, which focuses on application-specific architectures, the Sun approach was applicable only to CPU architecture. In particular, their counterflow processor uses two *distinct* pipelines to carry two different information streams in opposite directions, and introduces interlocks to allow the two streams to interact. One stream carries CPU instructions, and the other carries data fetched from registers and memory. The key benefit is that inter-instruction data dependencies are efficiently handled by this counterflow arrangement: if an instruction modifies a data value flowing counter to it, the modified value is immediately available to the subsequent instruction.

There are two key limitations of the Sproull approach. First, the Sproull approach belongs to a completely different problem domain—CPU architecture—and is not applicable to the design of custom architectures. Second, their approach has a critical drawback: arbitration is required between the two pipelines—one arbiter per pipeline stage—to ensure that corresponding tokens in the two streams do not "skip past" each other, leading to significant implementation complexity and loss of performance, and also to non-determinism in the system's operation. In contrast, our proposed approach achieves counterflow within a single pipeline, by "piggybacking" control tokens on top of the acknowledge signals already required for asynchronous handshaking, and thereby does not suffer from these drawbacks.

Early Output Logic (Brej and Garside). Brej and Garside attempted to solve the problem of preemption [1, 2], but their approach had significant drawbacks. In particular, their approach did not appropriately address a metastability issue: the handshake control in a pipeline stage can become metastable if the stage receives a data token and an anti-token nearly simultaneously. As a result, their circuits have hazards (*i.e.,* glitches), and therefore, their approach cannot be guaranteed safe without making timing assumptions. In addition, their approach incurs significant area and power consumption overheads.

Counterflow Pipelined Multiplier (Hensley et al.) Hensley et al. [4, 5] introduced a counterflow pipelined asynchronous radix-4 booth multiplier, with a novel counterflow organization: the data bits flow in one direction and the Booth commands piggyback on the acknowledgments flowing in the opposite direction. This design shares with CFPP the idea of data and instructions flowing in opposite directions. However, this counterflow pipeline uses only one pipeline as both requests and acknowledgments carry information. The design allowed overlapped execution of multiple iterations of the Booth algorithm. Also the modularity and bit-level pipelining enable the multiplier to be scaled to arbitrary operand widths without requiring gate resizing or cycle time overheads.

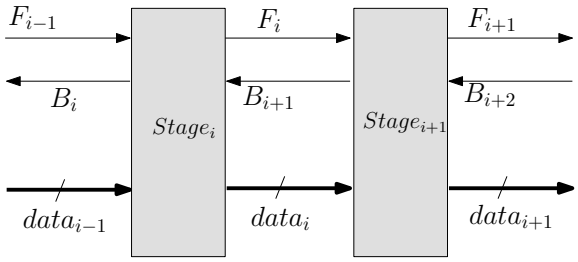

Figure 3: Signal notation in counterflow pipelines

3. THE NEW COUNTERFLOW PIPELINE APPROACH

This section introduces our new counterflow pipeline approach. The new pipeline uses two-phase communication, and a bundled data representation [3]. Events in the new pipeline are coordinately according to a protocl that shares some ideas with MOUSETRAP pipelines [6]; hence we call the new pipeline style "Counterflow Mousetrap."

After introducing some notation, we will describe the protocol used in the new pipeline style, followed by its circuit-level implementation and details of its operation. Finally, we will discuss how metastability is avoided by our counterflow pipeline, and then provide an analysis of its performance.

3.1 Notation

In the MOUSETRAP pipeline and in other asynchronous pipelines in general, the signal wires are named based on the type of signal they carry. In particular, all of the forward flowing signals carry requests, and are hence named *req*. Similarly, all of the reverse flowing signals carry acknowledgments, and are hence named *ack*. However, in the counterflow pipeline we are about to present, these wires can assume different meanings (*i.e.,* reverse their roles) depending on whether a data token is being passed forward, or if an anti-token is being relayed backward. Therefore, to avoid any confusion, instead of referring to a wire as a *req* or *ack*, we will simply refer to it by the direction in which it carries information—forward or backward—and by the index of the stage associated with it. That is, as shown in Figure 3, all the forward flowing signals are named F_i and backward flowing signals B_i. Since data flows in the forward direction only (left to right), the data input to a stage is $data_{i-1}$ and its data output is $data_i$. In addition, each stage generates two control signals, one in the forward direction (F_i), and the other in the backward direction (B_i).

We will use the term *token* to denote a data item and its associated request flowing in the forward direction. On the other hand, an *antitoken* refers to control information flowing in the reverse direction. For the approach presented in this paper, there is no need for any data values to be transmitted in the reverse direction; anti-tokens simply represent control events.

A pipeline stage is said to be in an "idle" state if it is not processing any tokens or antitokens at that moment, and all previous handshake cycles have been completed. Since we are using transition signaling (2-phase), the idle state corresponds to all of the control signals (*i.e.,* inputs F_{i-1} and B_{i+1} and outputs F_i and B_i) being in the same state (either all zero or all one).

We now relate the processing of tokens and antitokens with the states of the stage's control signals:

- A stage i in an idle state is said to have received a token (request from previous stage) if the input F_{i-1} signal toggles. When this happens, the state of the input F_{i-1} signal will be

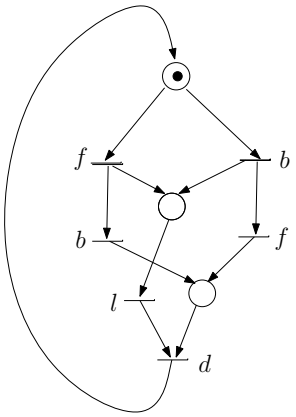

Figure 4: Petri net for a simplified version of a counterflow stage's protocol

different from all the other three signals. The stage can then acknowledge the token by toggling B_i signal and forward the same to the next stage by toggling the F_i signal.

- A stage in an idle stage is said to have received an antitoken (request from the next stage) if the B_{i+1} signal toggles. When this happens, the stage of the input B_{i+1} signal will be different from all the other three signals. The stage can then acknowledge the antitoken by toggling the F_i signal and forward the antitoken to the previous stage by toggling the B_i signal.

It is important to note that toggling a signal wire may imply different actions. Specifically,

- When a stage toggles its F_i signal, it either means forwarding a token or acknowledging an antitoken, depending upon whether the stage is responding to a token or an antitoken.
- When a stage toggles its B_i signal, it either means forwarding an antitoken or acknowledging a token, depending upon whether the stage is responding to an antitoken or a token.

With this notation and setup, we now describe the counterflow pipeline's protocol in detail, followed by its circuit-level implementation.

3.2 Pipeline Protocol

The protocol works as follows. In an idle state, a stage can receive either a token or an antitoken. If the first case, it acknowledges the token and simultaneously forwards it to the next stage. Similarly, if it receives an antitoken, it acknowledges it and forwards it to the previous stage. After forwarding a token or an antitoken, a stage cannot accept a new token or an antitoken until an acknowledgment corresponding to the forwarded token or the antitoken is received. When a stage receives a token and an antitoken simultaneously, it can consider one of the following:

- The received antitoken is the acknowledgment to the token to be forwarded
- The received token in the acknowledgment to the antitoken to be forwarded

We constructed a Petri net conforming to the protocol rules. However, the specification turned out to be too concurrent to be easily drawn or understood. To address this problem, we decided to constrain the protocol by assuming the following restriction on the environment of the stage: a stage in a non-idle state will not receive a new

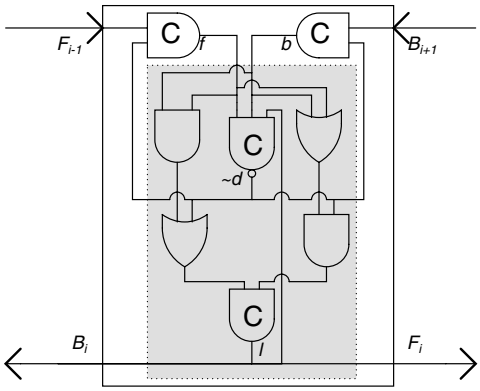

Figure 5: Implementation of a counterflow stage's controller

token or a new antitoken until it has completely processed the existing token or antitoken, *i.e.* until it has received the acknowledgment associated with the previous token or antitoken that it transmitted. This restriction on the environment greatly simplifies the specification of the stage's control, although the controller implementation will now need to embedded in a "wrapper" before it can be safely embedded in the actual environment (as discussed in the next subsection).

Figure 4 shows the simplified specification of stage's controller. f and b are the inputs to the Petri net and l the output. d is an extra variable required to keep track of the state of the Petri net. The transitions labeled f indicate receiving a token (or an acknowledgment to an antitoken) and the transitions labeled b indicate receiving a antitoken (or an acknowledgment to a token). From the given stage in the Petri net, either the left transition or the right transition may fire. If the left transition fires first, it means f has toggled first - a token has been received. This will enable both transitions b and l to fire. In this context, firing l would mean acknowledging the previous stage and forwarding the token the next stage. After an acknowledgment has been received from the next stage (firing b), the Petri net will come back to its start stage. The same explanation can be extended to the case when left transition fires first from the start stage.

3.3 Pipeline Implementation

Each stage of the pipeline consists of a *controller* and a *data latch*. The controller controls the outgoing requests and acknowledgments through which the stages communicated. It also controls the enable signal to the data latch controlling the data flow through the pipeline.

3.3.1 *Controller*

We implemented the controller based on the Petri net description of the protocol. The circuit-level implemented is shown in Figure 5. The subcircuit shown in the shaded area represents exactly the Petri net we discussed above. Since the Petri net above assumed a restriction on the environment's behavior, we need to add a "wrapper" around the stage controller to allow it to be safely embedded in the actual environment. This wrapper consists of what we call $guarding$ C-elements, as shown in the figure. The shaded subcircuit has two inputs f and b and an output l, implemented by the logic equations:

- *d:*
  set : $f\ x\ b\ x\ l$
  reset : $\tilde{f}\ x\ \tilde{b}\ x\ \tilde{l}$
- *l:*
  set : $\tilde{d}\ x\ (f + b)$
  reset : $d\ x\ (\tilde{f} + \tilde{b})$

Table 1: States of a counterflow pipeline stage

One State	Other State	Stage of Controller
r,a,l,d	-	idle
a,l,d	r	accepted a token
a,d	l,r	waiting for token ack
r,l,d	a	accepted an antitoken
r,d	l,a	waiting for antitoken ack
r,a	l,d	token antitoken clashed

The guarding C-elements are provided on all the inputs to the controller. They ensure that a new incoming token or antitoken is not accepted until acknowledgment to the previously forwarded token or antitoken has been received. In specific, a change in d indicates completion of a cycle. Until then any new incoming F or B are blocked. As evident from the Petri net, the states of these four signals indicate a specific state of the controller. This mapping is tabulated in Table 1. The first column represents the subset of the signals which are in a particular state (0 or 1). The second column has the other set.

3.3.2 *Pipeline Operation*

Let us first consider the flow of tokens through the pipeline. Initially, when the pipeline is empty, all the stages are in idle state and all the signals are low. When the first token (data item) flows through, it flips the signal wires to high as it flows through the pipeline from right to left. When the token is at some intermediate stage, all the signals associated with pipeline stages prior to this stage are high and all the signals associated with later stages are low. When the token arrives at the other end of the pipeline, all the stages are high. When a second token flows through, all the signals are toggled back to low again. When tokens are fed in the start of the pipeline one continuously, the signal values of the pipeline stages alternate along the pipeline.

The flow of antitokens will cause exactly the same kind of behavior as in the case of tokens but from the other end of the pipeline. If an antitoken is injected in an empty pipeline, it toggles all the signal lines as it flows through the pipeline from right to left. Once it reaches the other end of the pipeline, all signals are toggled to one. A second flowing antitoken will toggle all the signals back to zero.

Now consider injecting a token and an antitoken simultaneously into the left and the right end of the pipeline, respectively. As the token and antitoken travel towards each other, they leave all the signals toggled in their trail. Finally as they *crash* at some intermediate stage, they *cancel* each other and all the signals will be at level one. Injecting a second token-antitoken pair will bring back all the signals to zero after they *crash* at some intermediate stage. A stage is said to have encountered a token antitoken *crash* if the following happens: An antitoken is received in response to a forwarded token. In this case the antitoken is considered an acknowledgment to the forwarded token.

In a more general implementation than considered in this paper, tokens and antitokens may both carry data packets. In our counterflow protocol, however, we consider tokens as data carrying requests and antitokens merely as *request killers*. Hence, antitokens are not associated with any data; they merely kill the first token they encounter along their path, killing themselves in the process. Therefore, the latch needs to be enabled only when a token is passing through. In general, however, counterflow pipeline can be designed to support data flow in either direction, as was done for the counterflow Booth multiplier in [4, 5].

3.3.3 *Data Latch*

The flow of data through a stage's latch is coordinated by the latch's "enable" signal. We now discuss possible implementation of this enable signal.

In the traditional MOUSETRAP pipeline, the latches are normally transparent, and become opaque as soon as data passes through [6]. Once opaque, the latch is reenabled only when an acknowledgment from the next stage is received. This behavior is simply implemented by the following logic equation:

$$enable = (l \; XNOR \; b)$$

While keeping latches normally open has the advantage of reduced latencies and shorter cycle times, it also has a disadvantage: garbage data (*i.e.,* transients on the data wires, with no associated request) can flow through the pipeline, thereby wasting energy. Ideally, the latches should be made transparent only when there is an impending request (token). This behavior can be obtained by modifying the logic equation above, to enable the latch when not only a previously transmitted token by the stage has been acknowledged by the next stage, but also a new token has been produced by the previous stage:

$$enable = (l \; XNOR \; b) \; \& \; (f \; XOR \; l)$$

The above logic equation is still not complete, however. In particular, it does not consider antitokens. When a stage is processing an antitoken, there is no need to open the latch at all. So, we need to make sure the latch is not opened when F toggles in response to a sent antitoken. The final logic equation for the enable signal is thus as follows:

$$enable = (l \; XNOR \; b) \; \& \; (f \; XOR \; l) \; \& \; (d \; XNOR \; b) \qquad (1)$$

Interestingly, the above logic implementation of the latch enable may glitch in a particular scenario; however, those glitches are completely harmless. In particular, when an idle pipeline stage (with its latch disabled) receives an antitoken from its right neighbor, the current stage's latch may temporarily become enabled (for a certain combination of gate and wire delays), even though it is ideally supposed to stay disabled. This glitch, however, is harmless since it merely affects the data path, and has no impact on the handshaking signals. In particular, the only impact of this glitch may be a slight unnecessary consumption of energy if the contents of the latch change as a result of this glitch. Since the implementation of the handshake control (see Section 3.3.1, Figure 5) is guaranteed to be glitch-free, the operation of the pipeline is guaranteed to be correct.

3.4 Metastability Avoidance

A fundamental challenge in counterflow pipelining is to handle metastability, which can result when a pipeline stage receives a data token and an anti-token nearly simultaneously. In particular, if the data token arrives first, the stage is supposed to process it; if the antitoken arrives first, the data token must be destroyed; however, if both arrive close together, the stage's control may become metastable.

Prior counterflow approaches handle metastability quite suboptimally. In particular, the approach of Sproull et al. [7] requires the addition of arbiters in every pipeline stage to coordinate the movement of forward and backward tokens through stage boundaries. This solution introduces significant complexity into the design, causing substantial area as well as performance overheads. In contrast, Brej et al. [1, 2], do not appropriately address the metastability issue. As a result, their circuits have potential hazards due to metastability, and therefore, their approach cannot be used reliably without making timing assumptions.

Our Approach: Protocol Symmetrization. Our implementation avoids the problems related to metastability by making the handshake controller react in an identical manner irrespective of whether the stage received a token first or an antitoken first. That is, we remove

the burden of deciding whether the token or the antitoken arrived first, by specifying the reactions to both events as being identical. We refer to this idea as *protocol symmetrization.*

The symmetry in the protocol of our counterflow pipeline is evident from the Petri net description in Figure 4. Starting from the initial state, either the left or the right transition may fire, depending upon whether a token or an antitoken is received. If both the token and the antitoken are received concurrently, the Petri net could exhibit two different firing sequences—fire f then fire b, or fire b then fire f—but in either case, the protocol immediately reaches the same state. As viewed from the left and right neighbors of a stage, the stage's external behavior is identical in both scenarios. As a result, the circuit-level implementation of this protocol can avoid metastability altogether.

Therefore, the handshake protocol of our counterflow pipeline works correctly in all timing scenarios, including when a data token and an anti-token arrive at a stage nearly simultaneously.

3.5 Timing Analysis

Cycle time in a pipeline is the sum of forward latency and reverse latency. Forward latency in our case is defined as the time taken by a token to reach from the start of a particular stage to the start of the next stage. In our counterflow pipeline, the forward latency equals the delay of two C-elements plus the delay through the stage's function logic. Reverse latency is the time taken by an acknowledgment to move from one stage to its previous stage. In our pipe this latency is four C-elements. The total cycle time is hence the delay of six C-elements plus the matched delay of logic.

In traditional MOUSETRAP [6], the total cycle time equals the latency of two latches and one XNOR gate, plus the delay through the stage's function logic. If we assume similar latencies for a C-element and a latch, then our counterflow pipeline has a cycle time that is four C-element latencies greater than that of traditional MOUSETRAP. We will see in the results section how this overhead somewhat reduces the throughput improvements obtained when our counterflow approach is applied to the implementation of preemption, speculation and eager evaluation.

4. ARCHITECTURAL TEMPLATES FOR PREEMPTION

In the previous section we introduced a novel counterflow pipeline approach that supports data requests flowing along one direction and antitokens (*i.e.* ignore/kill requests) along the other. For simplicity of presentation, the discussion focused on *linear* pipeline stages, *i.e.* stages with a single left neighbor and a single right neighbor. However, in order to apply our approach to implement preemption, speculation and eager evaluation, more sophisticated stages must be designed, *i.e.* those that can handle forks and joins.

This section presents the design of three special types of counterflow pipeline stages: *fork stage, join stage,* and *if-then-else stage.*

4.1 Fork Controller

A fork stage in a MOUSETRAP pipeline simply forks off its output and the associated request to two or more destinations. It waits till all the acknowledgments are returned before accepting any new request.

In the counterflow pipeline, fork is slightly more complex: it should also be able to deal with antitokens. For simplicity, let us call the stage from which tokens arrive to the fork stage as the *input* stage; assume there is only one input stage. Let us call the stages to which the fork stage transmits tokens as the *output* stages. There can be two or more output stages. When transmitting a token, its behavior is similar to that of MOUSETRAP fork. On the other hand, when it receives antitokens, it must wait until it has received an antitoken from each of its output stages before sending an antitoken to its inputs stage.

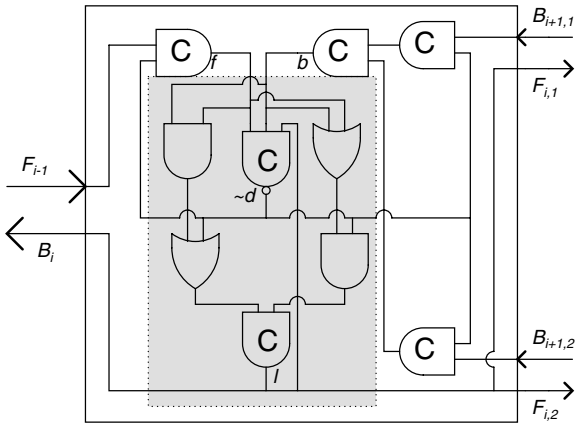

Figure 6: Counterflow Fork Controller

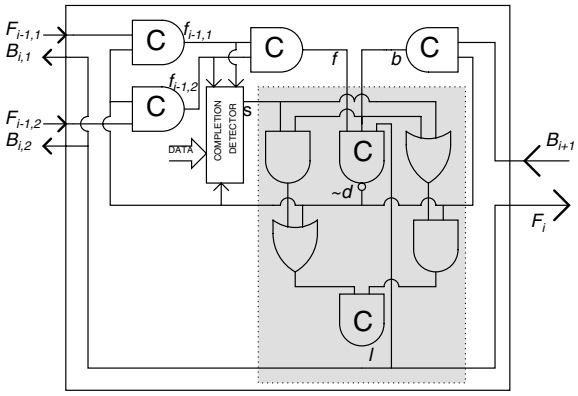

Figure 7: Counterflow Join Controller

Figure 6 shows our implementation of the counterflow pipeline fork stage. The implementation is quite similar to the basic stage controller (Figure 5), but with the following differences: (i) each incoming B signal requires a separate C-element to condition it, and (ii) one extra C-element is required to combine all of conditioned B signals together.

4.2 Join Controller

The join controller is a key component in a counterflow system, since it is this stage that *generates the antitokens.* In particular, the join stage has multiple inputs, and has the ability to decide when some of those inputs are not required; those inputs are then preempted.

In order to generate the antitokens, the join stage needs logic to determine if/when the inputs received are sufficient to compute the output. Thus, a special type of completion detector, called a *"sufficiency detector,"* is used on the *input* side of the join stage.

The implementation of the join controller is best understood by relating it to the implementation of the basic stage controller of Figure 5. In particular, the basic stage controller can be regarded as a degenerate case of a join, *i.e.* a basic stage is a one-way join. Thus, in the basic controller, the f signal (which is a conditioned version of the F input) is effectively a sufficiency signal: it indicates when the sole input to the 1-way join is ready and, therefore, sufficient.

Therefore, in order to implement the join controller, the logic

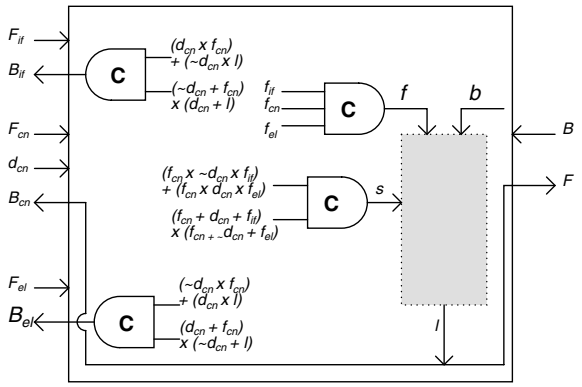

Figure 8: An "If-Then-Else" Join Controller

Table 2: Determining the completion detector logic

f_{cn}	d_{cn}	f_{if}	f_{el}	s
0	0	0	-	0
1	0	1	-	1
0	1	-	0	0
1	1	-	1	1

equations of the basic controller are simply modified to replace the f signal by a new sufficiency signal s:

- *d:*
  set : f x b x l
  reset : f x b x l

- *l:*
  set : d x (s + b)
  reset : d x (s + b)

The sufficiency signal s itself is generated using a sufficiency detector, the implementation of which would be specific to any given application (the next subsection provides an example).

Figure 7 shows our implementation of the join controller. The main addition is the logic to determine *sufficiency* - shown as completion detector. This logic takes as input $f_{i,1}$, $f_{i,2}$, d, and the data input from both the input streams. A toggle along the output indicates sufficiency. Once sufficiency is determined, the controller generates the outgoing token, acknowledgments along the input lines received and antitokens along the other lines. The completion detector logic depends on the specific logic implemented by the join stage. In the next subsection, we present one such example.

4.3 If-Then-Else Controller

The join controller for if-then-else cases is a special kind of join where an extra optimization can be applied. The antitoken along the unwanted branch can be generated even before token to the next stage is generated. This is because, once the condition bit is evaluated, the join can immediately send an antitoken along the unwanted branch. It however has to wait till the input arrives on the right branch before it can produce an outgoing token. This optimization does not necessarily save time but it helps in killing computation along the unwanted Branch by producing an early antitoken.

Figure 8 shows our implementation of the if-then-else join. Some circuit details evident in the previous described general join stage are deleted for clarity. The if-then-else join has 3 input channels - if branch (with F_{if} and B_{if} control signals); $else$ branch (with F_{el} and B_{el} control signals); and the condition evaluation branch (with F_{cn} and B_{cn} control signals). The join also has the condition bit $d + cn$

Table 3: Effectiveness with varying "if branch taken" probability

Prob$_{if}$ (%)	Time (μsec)	Energy (10^4) units	Normalized Throughput	Energy Savings (%)
5	3.36	35.94	0.95	-8.90
10	3.29	35.30	0.98	-6.96
20	3.17	34.62	1.01	-4.92
30	3.05	33.91	1.05	-2.77
40	2.91	33.06	1.10	-0.17
50	2.72	31.88	1.18	3.40
60	2.49	30.52	1.29	7.51
70	2.26	29.15	1.42	11.66
80	1.97	27.32	1.63	17.21
90	1.67	25.45	1.93	22.87
95	1.45	24.10	2.22	26.98

as input, used in the completion detector logic. For this specific join, the completion detector can be implemented as follows. We first draw a truth table specifying the output s for a given set of inputs . The inputs in this case are f_{cn}, d_{cn}, f_{if} and f_{el}. Table 2 summarizes the different cases. The cases in the truth table with $s = 1$, constitute the minterms that set s. Other entries with $s = 0$reset s. The logic can be implemented using a C-Element with input set and $reset$.

To generate logic that produce early antitokens, the same of approach described above can be used. The two C-Elements used to compute B_{if} and B_{else} constitute the logic that resulted in the process.

5. RESULTS

This section presents experimental results to demonstrate the effectiveness of our counterflow approach for implementing preemptive architectures.

Experimental Setup. Several circuit examples were designed and simulated to quantify the benefit of our approach. Each example was implemented in two different ways: (i) using traditional MOUSETRAP pipelines, to serve as the base case for comparison; and (ii) using the proposed counterflow pipeline approach. All designs were implemented in structural Verilog, and simulated using the Verilog-XL switch-level simulator from the Cadence tool suite. A simple delay and energy model was used: each C-element and each single-bit latch was assumed to have a unit delay latency (= 1ns), and unit energy consumption per transition. Function logic blocks were modeled at a behavioral level.

Application I: Speculation. The first application considered is an example consisting of a single if-then-else statement. The if-computation branch was assumed to be relative low latency, with only two pipeline stages. On the other hand, the else-computation branch had a higher latency, with eight pipeline stages. The Boolean condition was evaluated using a single pipeline stage. Each pipeline stage had function logic with a fixed latency of five delay units.

The first set of experiments was aimed at observing the throughput and energy benefits of our approach as the branch probability was varied. At each branch probability, a total of 1000 distinct data items were evaluated.

Table 3 summarizes the results. The first column lists Prob$_{if}$, the probability that the if-branch is taken, for each simulation. This probability was varied from 5% to 95%. The second column indicates the total time taken by the counterflow pipelined implementation to execute the simulation run containing 1000 distinct data items. The third column lists the total energy consumed. By comparison, the

Table 4: Effect of varying if-/else-computation latencies

N_{if}	N_{else}	Normalized Throughput
2	2	0.86
2	4	1.12
2	6	1.29
2	8	1.42
2	10	1.51

baseline MOUSETRAP implementation obtained an execution time of $3.2\mu sec$ and energy consumption of 33×10^4 units; these numbers do not vary with the branch probability because this baseline implementation does not use the new approach of this paper, and therefore cannot preempt wasteful computation. Finally, the last two columns present the throughput and energy numbers for the counterflow approach relative to the baseline implementation: normalized throughput (*i.e. speedup*), and %age energy savings.

Discussion. As the probability of the if-computation branch being taken increases, the benefit of the counterflow approach increases. This is because the likelihood increases that the relatively expensive else-computation will not be needed (and can therefore be preempted). At very low probabilities (5–10%), however, the overheads of the counterflow approach erode any potential benefit achieved, resulting in a marginal decrease in throughput (up to 5%), and a slight increase in energy consumption (up to 9%). At higher probabilities, especially greater than 50%, the benefits of our approach are significant: an increase in throughput by a factor of up to 2.22x, and up to 27% savings in energy consumption.

Another experiment was performed to study the impact of if-computation and else-computation latencies on the effectiveness of our approach. Table 4 summarizes the results of this experiment. For these simulations, the if-computation latency (N_{if}) was fixed at 2 pipeline stages, while the else-computation latency (N_{else}) was varied from 2 to 10 pipeline stages. The branch probability was fixed at 70% for all cases. Once again the throughput of our counterflow approach is presented normalized to the baseline MOUSETRAP implementation. The results show that, as expected, the effectiveness of the counterflow approach improves as the else-computation blocks increases in its latency.

Application II: Eager Evaluation. The second application considered is an example consisting of a stage that is capable of eager evaluation, *i.e.* it can sometimes produce an output even when one of its two inputs are not available. In particular, there are two pipelines—one low-latency and the other high-latency—feeding into the "eager" stage. The eager stage always requires input from the low-latency pipeline to generate an output, but it may or may not require input from the high-latency pipeline to compute the output. Let Prob_{eager} represent the probability that input from the low-latency pipeline alone is sufficient to compute the result (*i.e.,* the result can be "eagerly" computed). In our experiment, the low-latency pipeline had 2 pipeline stages; the high-latency pipeline had 5 pipeline stages with one of those five stages in turn containing a complex high-latency logic function block. All pipeline stages had 5 units latency, except for the complex function block which had 40 units of latency. Simulations were performed at different Prob_{eager}. At each Prob_{eager}, a total of 1000 distinct data items were evaluated in a simulation run.

Table 5 summarizes the results of this experiment. The first column lists Prob_{eager}, the probability that the result can be "eagerly" evaluated. The next two columns present the total execution times and energy consumption for the counterflow approach. By comparison, the baseline MOUSETRAP implementation obtained an execution time of $4.0\mu sec$ and consumed 30×10^4 units of energy, irrespective of the value of Prob_{eager}. The final two columns present throughput and energy benefits of the counterflow approach relative to the baseline MOUSETRAP approach.

Table 5: Early output logic using counterflow protocol

Prob_{eager}	Time (μsec)	Energy (10^4 units)	Normalized Throughput	Energy Savings (%)
5	4.37	31.74	0.92	-6.17
10	4.31	31.41	0.93	-5.04
20	4.18	31.37	0.96	-4.93
30	4.03	30.10	1.00	-0.67
40	3.83	29.52	1.05	1.28
50	3.60	28.82	1.11	3.61
60	3.36	28.15	1.19	5.84
70	3.10	27.49	1.29	8.05
80	2.83	26.83	1.41	10.27
90	2.57	26.23	1.56	12.28
95	2.40	25.80	1.67	13.71

Discussion. The results indicate a throughput improvement of up to 1.67x and 13.7% improvement in energy usage at very high rates of Prob_{eager}, i.e only the smaller branch input is sufficient most of the time to compute the output. However, when eager evaluation is less likely (especially less than 60%), the counterflow protocol is much less beneficial.

6. CONCLUSION

This paper introduced a new conterflow pipeline style based on a novel protocol that allows anti-tokens to flow backward through the pipeline and preempt or cancel data tokens in the pipeline. Building upon this pipeline style, a whole set of architectural templates were introduced to enable the design of asynchronous pipelined systems with useful features such as speculation, preemption and eager evaluation. Experimental analysis suggests promising throughput improvements and energy benefits.

7. REFERENCES

[1] C. Brej. *Early Output and Anti-Tokens*. PhD thesis, Department of Computer Science, University of Manchester, 2005.

[2] C. Brej and J. Garside. Early output logic using anti-tokens. In *International Workshop on Logic Synthesis*, 2003.

[3] A. Davis and S. M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, Dept. of Computer Science, University of Utah, Sept. 1997.

[4] J. Hensley, A. Lastra, and M. Singh. An area- and energy-efficient asynchronous booth multiplier for mobile devices. In *Proc. Int. Conf. Computer Design (ICCD)*, 2004.

[5] J. Hensley, A. Lastra, and M. Singh. A Scalable Counterflow-Pipelined Asynchronous Radix-4 Booth Multiplier. In *Proc. Int. Symposium on Advanced Research in Asynchronous Circuits and Systems*, Mar. 2005.

[6] M. Singh and S. M. Nowick. MOUSETRAP: Ultra-high-speed transition-signaling asynchronous pipelines. In *Proc. Int. Conf. Computer Design (ICCD)*, pages 9–17, 2001.

[7] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, Fall 1994.