Synchronous Interlocked Pipelines

Hans M. Jacobson Prabhakar N. Kudva
Design Automation Department
IBM T.J. Watson Research Center

Pradip Bose Peter W. Cook Stanley E. Schuster
High Performance Systems Department
IBM T.J. Watson Research Center

Eric G. Mercer Chris J. Myers*
Department of Electrical and Computer Engineering
University of Utah

Abstract

In a circuit environment that is becoming increasingly sensitive to dynamic power dissipation and noise, and where cycle time available for control decisions continues to decrease, locality principles are becoming paramount in controlling advancement of data through pipelined systems. Achieving fine grained power down and progressive pipeline stalls at the local stage level is therefore becoming increasingly important to enable lower dynamic power consumption while keeping introduced switching noise under control as well as avoiding global distribution of timing critical stall signals.

It has long been known that the interlocking properties of asynchronous pipelined systems have a potential to provide such benefits. However, it has not been understood how such interlocking can be achieved in synchronous pipelines. This paper presents a novel technique based on local clock gating and synchronous handshake protocols that achieves stage level interlocking characteristics in synchronous pipelines similar to that of asynchronous pipelines. The presented technique is directly applicable to traditional synchronous pipelines and works equally well for two-phase clocked pipelines based on transparent latches, as well as one-phase clocked pipelines based on master-slave latches.

Background

In our search for new techniques to achieve lower power in synchronous circuits and systems at IBM we have carefully studied, as well as developed [4], asynchronous pipeline techniques. Asynchronous pipelines [5] have several properties that have the potential to benefit todays and tomorrows circuit design. The most promising of these properties is the ability to only activate a pipeline stage in the presence of valid data, and the ability of a stage to make local control decisions through pipeline interlocking. At the same time, there are considerable resources available in design tools and expertise for synchronous techniques that cannot be discounted when designing commercial chips. For this reason, it is desirable to find a middle ground in techniques that can provide the benefits of asynchronous properties in a synchronous context. The interlocked synchronous pipeline techniques presented in this paper are a first step in our efforts to achieve this goal.

*Eric Mercer was with IBM Austin Research Laboratory during part of this research. Mercer and Myers are supported by NSF CAREER award MIP-9625014, SRC contract 97-DJ-487 and 99-TJ-694, and a grant from Intel Corporation.

1. Introduction

Power dissipation is becoming a major design constraint, not only in portable, but also in high-performance VLSI systems. As clock and latch power is nearing 70% of the total power consumption in synchronous integrated circuits that do not employ clock gating, power aware techniques that perform computations only on demand are becoming necessary to meet power budgets. At the same time, growing transistor density and lower transistor thresholds are causing increased switching currents and reduced noise margins. As a consequence, simultaneous switching noise and power supply ringing [1] due to large variances in switching currents is becoming a concern. To implement computing on demand while guarding against large variances in switching current, it is becoming increasingly important to investigate techniques based on fine grained clock gating at the stage level.

In this context, performing pipeline stalls in the backward direction of a pipeline is also an important concern as it affects not only power and switching currents but also signal locality. Compared to the data recirculation approach often used today, clock gating is a low power alternative to implementing pipeline stalls. Stalls have traditionally been performed at a coarse grained global or unit level, rather than at the more fine grained stage level. However, such coarse grained stalls can cause large cycle to cycle variance in switching currents and also require global propagation of stall signals. With stall signals already on the critical path in some of todays designs, cycle time may come to be affected as wire delays do not scale well with technology. Locality principles are therefore becoming increasingly important in pipelined design. Due to the concerns with switching currents and wire delays, it is becoming increasingly difficult to design for, and cost-effectively implement, stalling of synchronous pipelines.

Contrary to synchronous pipelines, asynchronous pipelines are not affected by the problems mentioned above to the same extent. The stages in an asynchronous pipeline are interlocked through request-acknowledge protocol handshakes that ensure correct progression of data through the pipeline. This interlocking provides several benefits. One benefit of asynchronous interlocking is fine grained power down at the pipeline stage level. A stage is only requested to compute when a computation is required. In pipelines with low utilization, significant power can be saved when computation is performed only on demand. Another benefit of interlocking is the inherent locality of control decisions when controlling the progression of data through the pipeline. This allows stalls to be performed on a local basis, one pipeline stage at a time. This way of progressively stalling a pipeline avoids

Proceedings of the Eighth International Symposium on Asynchronous Circuits and Systems (ASYNC'02)
1522-8681/02 $17.00 © 2002 **IEEE**

global distribution of stall signals, keeps the cycle to cycle variance in switching currents low, and allows continued propagation of upstream data in the presence of holes in the pipeline. To our knowledge, no method to similarly interlock stages in synchronous pipelines has been reported.

This paper presents an *elastic synchronous pipeline* (ESP) technique as a cost-effective solution to the local stalling problem in synchronous pipelines. Progressive stage by stage stalling is achieved through local clock gating in the backward direction of the pipeline that is similar to the acknowledge interlocking found in asynchronous pipelines. The introduced technique has no datapath delay overhead and a minimal increase in area. These elastic pipelines are in turn extended to a fully *interlocked synchronous pipeline* (ISP) structure where each stage is interlocked with its neighboring stages in the forward as well as backward direction. An ISP implements the interlocking properties of an asynchronous pipeline in a purely synchronous design: thus, it has the potential to improve the power, slack, and noise characteristics of the pipeline without compromising tool support and design expertise. These interlocked pipeline techniques are further extended to *generalized interlocked pipeline structures* that implement pipeline forks, joins, branches, and selects.

This paper first gives a brief overview of asynchronous pipelines in Section 2. This is followed by a presentation of traditional synchronous pipelines in Section 3 and the methods typically used to achieve fine grained power down and stalling in such pipelines. The elastic synchronous pipelines presented in Section 4 introduce the concept of backward interlocking in ordinary synchronous pipelines such that localized progressive stalls can be implemented. Section 5 extends these elastic pipelines to fully interlocked synchronous pipelines that are interlocked in both the forward and the backward direction. Section 6 further extends these interlocked synchronous pipeline techniques so that they can be applied to general pipeline structures such as forks, joins, branches, and selects. Section 7 discusses how the presented pipelines were formally verified and how such pipelines can be made testable through scan chains. Section 8 provides conclusions.

2. Asynchronous pipelines

Asynchronous pipelines have several interesting properties that are hard to achieve in synchronous pipelines. These properties include fine grained power down, localized stalls, low cycle to cycle variance in switching currents, and the ability to use transparent latches without risking data races. These properties are a result of the way the stages of an asynchronous pipeline are interlocked to their neighboring stages in both the forward and backward direction.

An interlocked asynchronous pipeline is illustrated in Figure 1. In an asynchronous pipeline a request signal is used to let a downstream stage know when valid data is available on its inputs and a new computation is required. This provides for a fine grained power down when there is no computation to be performed as no request is generated in such cases. This request signal provides an interlocking in the forward direction of the pipeline. To guard against data races between latches, an asynchronous pipeline is also interlocked in the backward direction. This is achieved by an acknowledge signal that is used to let an upstream stage know that data has been safely latched and that

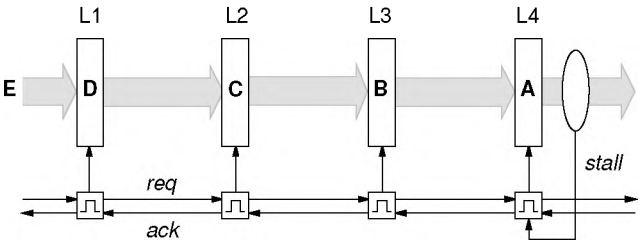

Figure 1. Asynchronous pipeline with abstracted data dependent stall condition.

new data can now be generated. This way, a bubble is created before data can move between stages, and there is no risk of data races between latches. Since no new data may be generated by the upstream stage until an acknowledge has been received, the upstream stage has to stall until such an acknowledge arrives. This backward interlocking is what provides the beneficial local stalling properties of asynchronous pipelines.

Together, the forward and backward interlocking available in asynchronous pipelines provide a powerful technique that not only can achieve fine grained power down and race free latching but also may provide better signal locality and reduced variance in switching currents in pipelines implementing stalls.

3. Synchronous pipelines

Synchronous pipelines traditionally prevent data races between latches, not by interlocking, but by alternating the transparency and opaqueness of latches in adjacent stages. There are two main approaches of this technique. One approach is based on transparent latches where a two-phase clock is used such that only every other pipeline stage is active at a time, the latches in inactive stages are opaque and act as barriers preventing data races between the transparent latches of active stages. The other approach is to use a one-phase clock with master-slave latches where the master and slave latches are alternating between transparent and opaque modes such that there is never a combinational path between two master latches or two slave latches. It is worth noting the similarity between these two methods. The only fundamental difference being that the two-phase pipeline has combinational logic between each array of latches while the one-phase pipeline only has combinational logic between slave and master latches. Although the techniques presented here are mainly illustrated in the context of two-phase pipelines, they work equally well with both types of pipelines.

3.1. Forward interlocking

Asynchronous pipelines have the potential to reduce power consumption since computations are only performed on demand. A pipeline stage does not perform a computation unless requested to. This is achieved by employing request signals in the forward direction of the pipeline implementing a forward interlocking such that data is only latched when there is a computation to perform.

Similar techniques can be used to power down stages in synchronous pipelines. A valid signal that propagates along with the data can be used to indicate when data is valid and a computation should be performed. If the valid signal indicates that data is not valid, the clock to the corresponding pipeline stage is gated for the duration of that clock cycle. Subsequently, a computation

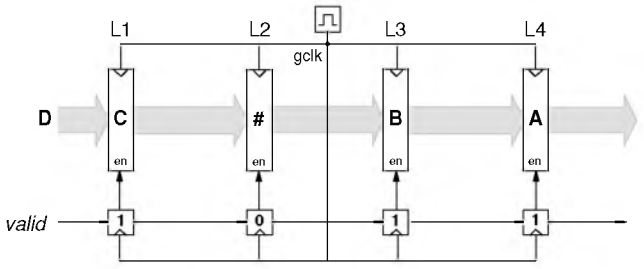

Figure 2. Synchronous pipeline with valid based clock gating.

is performed only when needed. This stage by stage clock gating thus performs a function similar to the request signal in an asynchronous pipeline and achieves a fine grained power down of the pipeline. Figure 2 illustrates a synchronous pipeline with a valid bit that propagates alongside the data in synchronous lockstep and gates the clock when there is no valid data present in a pipeline stage. In the figure, A, B, and C indicate valid data and is accompanied by a valid signal with value 1. The hashmark "#" indicates invalid data and is accompanied by a valid signal with value 0.

The valid signal technique has been explored in the context of saving power in synchronous pipelines [1]. However, to our knowledge, it has not been considered in the context of achieving interlocking between pipeline stages in synchronous pipelines.

3.2. Backward interlocking

Asynchronous pipelines have the ability to make control decisions regarding the advancement of data on a local basis. Decisions whether to halt or restart a pipeline stage can therefore be made independent from other pipeline stages. This may improve the slack on control signals as these are no longer distributed on global wires. This is important for stall signals which are often on the critical path. Another benefit of local control is that a pipeline can be brought to a halt and then restarted one stage at a time rather than all stages at once. This may reduce the cycle to cycle variance in switching currents.

In an asynchronous pipeline, the locality of control signals is achieved through stage level interlocking in the backward direction of the pipeline. An acknowledge signal is used to indicate to upstream stages whether the current stage is ready to receive new data or not. By not generating an acknowledge signal in a pipeline stage, the pipeline is brought to a halt, one stage at a time, as no new acknowledges are propagated backward in the pipeline.

Traditionally, synchronous pipelines have been stalled at the global level where all stages of either the entire pipeline, or a multi-stage unit, are stalled at the same time. Lately, however, cycle time and switching current constraints have started placing restrictions on how many stages can be stalled during the same cycle. The difficulty with stalling synchronous pipelines progressively is that data is lost at stall boundaries. Figure 3(a) illustrates this problem in a synchronous pipeline with master-slave latches. The striped vertical bar in the figure illustrates a stall boundary. The stall boundary indicates the place in the pipeline where upstream stages do not receive the stall signal in time before the next clock edge arrives due to cycle time constraints. While the stages downstream of the stall boundary receive the stall signal and correctly come to a halt, the stages upstream of the boundary

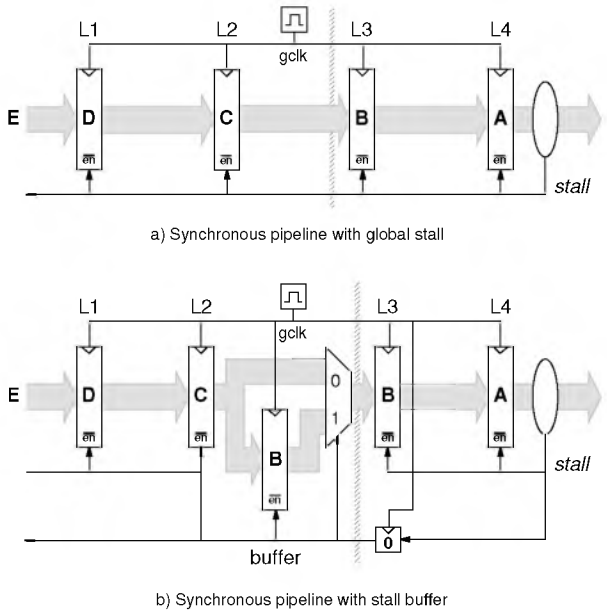

Figure 3. Synchronous pipeline with and without stall buffer.

do not see the stall signal in time and therefore, incorrectly latch new data. In the figure, as stage 3 is stalled, it keeps data item B. Stage 2, however, does not see the stall signal in time and therefore, latches data item D the next clock cycle. The result is that data item C that is stored in stage 2 is overwritten and lost.

Traditional approaches to handle progressive stalls have been to insert buffer stages in parallel to the pipeline at stall boundaries [2]. The buffer stages are used to temporarily store data that would otherwise be overwritten. This situation is illustrated in Figure 3(b). Due to the area, power, and delay overhead associated with buffer stages, stalls have traditionally been performed at a coarse grained level. However, as technology scales, the relative increase in wire delays and demand for shorter cycle time restricts how far a stall signal can propagate without impacting cycle time. As a result, buffer stages may have to be introduced at a finer granularity. At the finest level of granularity, where stalling is performed on a stage by stage basis, introducing extra buffer stages doubles the number of latches in a pipeline. Clearly, this approach is not cost-effective in terms of area and power at more fine grained levels.

As illustrated by the above examples, implementing cost-effective stalls in synchronous pipelines, not to mention finding a solution to backward interlocking, is a difficult problem. To our knowledge, no one has investigated how synchronous pipelines can be made to work with fine grained stalls without inserting extra buffer stages. The following section looks at how ordinary synchronous pipelines can be made to implement backward interlocking and thereby achieve cost-effective progressive stalling at the stage level, without the need for extra buffer stages.

4. Elastic synchronous pipelines (ESP)

Just as forward interlocking, that is, the decision to clock gate based on valid bits propagating in the forward direction of the pipeline can be performed locally, stage by stage, we would like to achieve a similar interlock in the backward direction. This allows stalls to take place locally on a stage by stage basis.

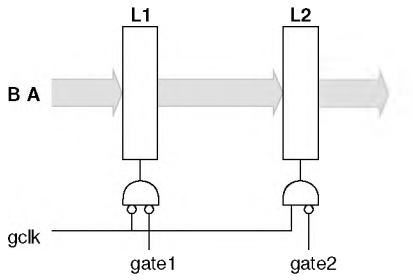

Figure 4. Latch pair.

The approach to achieve backward interlocking in synchronous pipelines presented in this paper reuses latches already present in the pipeline to act as buffer stages during stalls. This section first illustrates how this can be achieved in a synchronous system through sequential storage in a pair of latches. Application to pipelines and queues is considered later in this section.

4.1. Sequential read and store in adjacent latches

Consider how two data items can be stored in a pair of latches, L1 and L2, connected in series as illustrated in Figure 4. The latching of data is governed by a synchronous clock. Assume that the latches become opaque and transparent on opposite edges of the clock as in, for example, adjacent stages of a two-phase clocked pipeline. The sequential storage of data in the latches is achieved through clock gating. Consider how two data items A and B can be sequentially stored in, and then read from, the latches.

Storing A and B. Assume the global clock, *gclk* is low, latch L1 is transparent, and latch L2 is opaque. Data item A is now applied to the input of L1. When the next rising edge of *gclk* arrives, latch L1 becomes opaque and stores data item A and latch L2 becomes transparent. When the next falling edge of *gclk* arrives, latch L2 becomes opaque and stores data item A and latch L1 becomes transparent. Data item B is now applied to the input of latch L1. The clock to latch L2 is now gated by asserting signal *gate2*. When the next rising edge of *gclk* arrives, latch L2 remains opaque and continues to store data item A. At the same time latch L1 becomes opaque and stores data item B. The clock to latch L1 is now gated by asserting signal *gate1*. The data values A and B are now held in latches L2 and L1, respectively, until the clocks to the latches are enabled again.

Reading A and B. Assume that *gclk* is low and that latches L1 and L2 are both gated. The clock to latch L2 is now enabled by deasserting signal *gate2*. When the next rising edge of *gclk* arrives, the environment stores data item A from the output of latch L2. Latch L2, in turn, becomes transparent. The clock to latch L1 is now enabled by deasserting signal *gate1*. When the next falling edge of *gclk* arrives, latch L2 stores data item B. Latch L1, in turn, becomes transparent. When the next rising edge of *gclk* arrives, the environment stores data item B from the output of latch L2.

In the described fashion, any two data items can be sequentially stored and read in a pair of latches, even when the latches are adjacent and clocked by the same synchronous clock. This way of storing data items plays a fundamental part of the elastic nature of the pipelines presented in this paper.

4.2. Application to pipelines and queues

Sequential storing of data through clock gating is applicable also to pipelines and forms the fundamental basis of how to achieve backward interlocking in a synchronous pipeline. The backward interlocking is based on letting each stage generate a stall signal to its upstream neighbor, indicating when the stage is not ready to receive new data.

Consider a two-phase clocked synchronous pipeline based on transparent latches. In such a pipeline, adjacent stages are not active simultaneously. When a given stage is computing, the adjacent upstream and downstream stages are idle. The latches of active stages are transparent and the latches of idle stages are opaque. As a result, only every other stage stores data at any given time. It is important to note that while idle stages store data in their latches, that data is no longer useful as the data has already moved on to the next active stage in the pipeline. Subsequently, active stages contain data and idle stages contain bubbles. This is a fundamental property of our elastic pipelines since this means that half of the stages in a synchronous pipeline are "empty" and can potentially be used as buffer stages to stall the pipeline progressively. Let us take a closer look at how an ordinary synchronous two-phase pipeline can be made elastic.

Under normal operating conditions, the data latches for an active stage are transparent. When an active stage generates a stall signal, the data latches go opaque on the next clock edge and remain opaque until the stall condition goes away. The data latches are kept opaque by gating the clock with the stall signal. The stall signal in turn is propagated backward in the pipeline and is kept in synchronous lock-step to the pipeline by latching it at each pipeline stage. The stall signal thus propagates only one stage per clock edge, and it is thereby kept local to each stage. Note that the stall signal latches are not clock gated. As outlined above, it is sufficient to add a latch and a gating function to each pipeline stage in order to transform an ordinary two-phase clocked synchronous pipeline into an elastic pipeline.

Stalling an elastic pipeline

Consider the two-phase pipeline example in Figure 5. Note that the latches used to propagate the stall signal backward in the pipeline are clocked on the opposite edge of the data latches of their associated stage. Also note that the delay gates normally inserted to remove the skew between the data and stall latch clocks are not shown. The data latches of each stage are clock gated by the output of their associated stall latch. Consider the waveform trace illustrated in Figure 6. The figure illustrates the global and local data latch clocks along with the stall and data signals for each stage. The characters in the data waveforms illustrate distinct data items as they move through the pipeline. A box containing a character indicates that the data latches of the stage are opaque and that the corresponding data item is currently stored in that stage. A line indicates that the data latches are transparent.

The sub-trace between the dotted lines of Figure 6 is illustrated in more detail in Figure 7. In the sub-trace the data stream A,B,C,D,E is applied to the pipeline. The bold text in the trace indicates when data (or stall signal) is stored in the corresponding latch (i.e., the latch is opaque). This includes clock gated conditions. Grey non-bold text indicates that data is propagated through the latch (i.e., the latch is transparent). The shaded polygon in the trace illustrates how the stall condition propagates backward through the pipeline. The shaded polygon corresponds

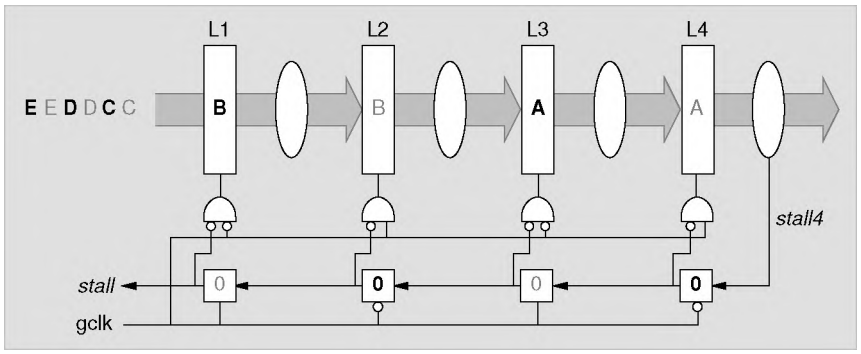

Figure 5. Two-phase clocked elastic synchronous pipeline implementing progressive stall through backward interlock.

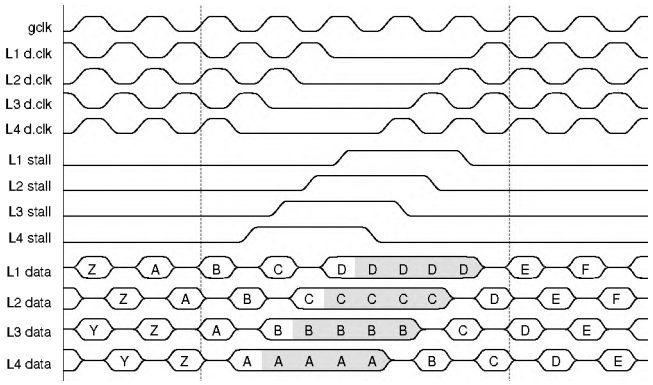

Figure 6. Waveform trace of two-phase clocked ESP.

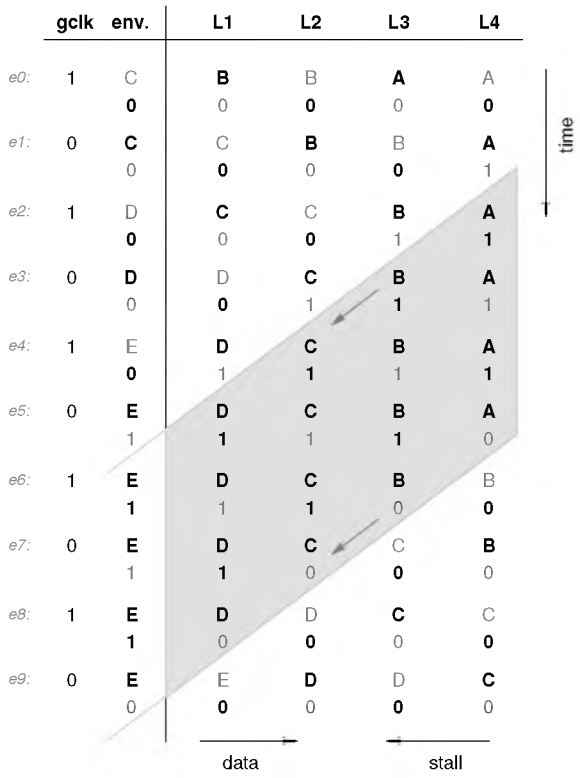

Figure 7. Detailed sub-trace of two-phase clocked ESP.

to the similarly shaded regions in Figure 6. The stall condition can be thought of as a sliding window moving in the backward direction of the pipeline. Outside the window, data is stored in every other pipeline stage as normal for a two-phase pipeline. Within the window, data is *compacted* such that data is stored in every pipeline stage. The adaptive nature of the pipeline storage capacity is what prompted the name elastic pipelines.

Returning to the trace in Figure 7, assume the clock is high and the pipeline is in steady state operation with two data items continuously present in the pipeline. The data latches in stages 1 and 3 at this time are opaque and store data items B and A respectively. The data latches for stage 2 and stage 4 are at this point transparent and do not store any data.

Once the next falling clock edge e_1 arrives, the data latches of stages 2 and 4 become opaque and store data items B and A respectively. At the same time, the stall latches of stages 2 and 4 become transparent. Assume that signal $stall_4$ is now asserted. The stall signal propagates through the transparent stall latch of stage 4 to the clock gating function of stage 4. Since stage 4 is now stalled, it continues to store data item A when the next rising clock edge e_2 arrives. At the same time, stages 1 and 3 store data items C and B respectively, and the asserted stall signal propagates to the clock gating function of stage 3.

When the next falling clock edge e_3 arrives, stages 3 and 4 are both stalled and continue to store data items B and A respectively. Stage 2 in turn stores data item C, and the asserted stall signal propagates to the gating function of stage 2. When the next rising clock edge e_4 arrives, stages 2, 3, and 4 are stalled and continue to store data items C, B, and A. Stage 1 in turn stores data item D, and the asserted stall signal propagates to the gating function of stage 1. The whole pipeline has now been safely stalled without losing any data items, and all stages in the pipeline are filled with valid data items.

Unstalling an elastic pipeline

Similar to stalling the pipeline, unstalling the pipeline enables the latches one stage at a time, recreating the bubbles in the pipeline such that no data is lost when data starts moving through the pipeline again. When the next falling clock edge e_5 arrives, all stages remain stalled and continue to store their respective data items. The stall latches of stages 2 and 4 become transparent. Assume that signal $stall_4$ is now deasserted. This implies that the condition causing the stall has gone away and that the environment is ready to latch the data output of stage 4 the next clock edge. The deasserted stall signal propagates through the transparent stall latch of stage 4 to the clock gating function of stage 4

and enables the data latch clock again. Since stage 4 is no longer stalled, the data latches become transparent once the next rising clock edge e_6 arrives. At the same time, stages 1, 2, and 3 remain stalled and store data items D, C, and B, and the deasserted stall signal propagates to the clock gating function of stage 3.

When the next falling clock edge e_7 arrives, stage 4 stores data item B. Stage 3 is no longer stalled and its data latches become transparent. Stages 1 and 2 remain stalled and store data items D and C. The deasserted stall signal propagates to the clock gating function of stage 2. When the next rising clock edge e_8 arrives, stage 3 stores data item C. Stage 2 is no longer stalled and its data latches become transparent. The deasserted stall signal propagates to the clock gating function of stage 1. When the next falling clock edge e_9 arrives, stages 2 and 4 store data items D and C, respectively. Stage 1 is no longer stalled, and the pipeline has again reached normal steady state operation with an occupancy of N/2 data items.

The fundamental reason that a pipeline can be progressively stalled this way is that, by filling in the bubbles that always exist in two-phase pipelines with data items, we hide the "delay" of propagating the stall signal backward in the pipeline one stage at a time. There are N number of stages in the pipeline, and N/2 data items present in the pipeline to start with. The N/2 stages that are empty can be used as buffers. It takes the stall signal N clock edges to propagate to the start of the pipeline. During this time, N/2 new data items enter the pipeline (in a two-phase pipeline new data enters the pipeline only every other clock edge). Subsequently, there is enough buffer storage such that all data can be safely stored. When all stages have stalled, the pipeline has an occupancy of N data items. When unstalling the pipeline, the delay introduced by propagating the stall signal backward one stage at a time recreates the bubbles in the pipeline such that data can start safely moving through the pipeline again. Once the whole pipeline is unstalled, the occupancy of the pipeline is again down to N/2 data items.

5. Interlocked synchronous pipelines (ISP)

Given an approach for forward interlock through valid based clock gating (Section 3.1) and a backward interlock based on elastic pipeline properties (Section 4), the next step is to merge these approaches into fully interlocked synchronous pipelines. Such pipelines have the computing on demand and localized progressive stall properties of asynchronous pipelines while still being driven by a synchronous clock.

Figure 8 illustrates an interlocked synchronous pipeline. Valid bits are propagated in the forward direction of the pipeline. A valid bit indicates when the associated data is valid and a computation should be performed. In this respect, the valid bit is the equivalent of the request signal used in asynchronous pipelines. Stall bits are propagated in the backward direction of the pipeline. These stall bits indicate when the pipeline must halt, for example, due to access conflicts to a shared resource. The stall bit is the equivalent of the (inverse) acknowledge signal used in asynchronous pipelines. Note that the interlocking referred to in a synchronous pipeline is the interlock of the valid and stall bit with respect to the clock, and not between the valid and stall bits themselves. That said, the effect of this interlocking is very similar to that of interlocking the request and acknowledge signals in asynchronous pipelines.

The main contribution of the valid bits, besides providing fine grained power down, is to indicate holes (absence of valid data) in the pipeline. A stall condition does not need to propagate backward when there is no valid data to stall. The valid bit, when not asserted, can therefore be used to override the propagation of the stall bit. The environment may therefore not have to stall unless the pipeline completely fills up. The valid bit may also serve to improve pipeline latency and throughput in the presence of stalls as upstream stages can continue to compute until all holes have been filled. During a stall condition, the valid bit latch must be clock gated together with the data latches in order to correctly propagate the valid bit along with its associated data.

From a delay perspective, consider a globally stalled pipeline with stall control logic for filling in holes and handling stall signals generated by multiple stages. The delay caused by long wires and the additional stall control logic grows linearly with the number of stages and eventually starts impacting the cycle time. In an interlocked pipeline, however, the stall control logic is local to each stage, adding only a small constant delay. Locally stalled pipelines can therefore give an advantage by increasing the slack on stall signals.

5.1. Interlocked pipeline operation

Consider the interlocked pipeline in Figure 8. An AND function has been added between the stall latches of the pipeline. The AND function ensures that holes in the pipeline are filled in by disabling the stall signal when there is no valid data present. An OR function has been added in stage 4. This OR function detects if a stall condition has been generated by either the local or downstream stage.[1] Now consider the waveform trace illustrated in Figure 9. The waveform illustrates the global and local data latch clocks along with the valid, stall, and data signals for each stage. The characters in the data waveforms illustrate distinct data items as they move through the pipeline. A box containing a character indicates that the data latches of the stage are opaque and that the corresponding data item is currently stored in that stage. A line indicates that the data latches are transparent.

The sub-trace between the dotted lines of Figure 9 is illustrated in more detail in Figure 10. In the sub-trace, the data stream A,#,B,#,C,D,E, where "#" represents invalid data (a hole), is applied to the pipeline. Note that invalid data is not propagated through the pipeline due to the valid bit based clock gating but rather is created in place when the corresponding valid bit turns zero. The bold text in the trace indicates when data (or valid/stall) is stored in the corresponding latch (i.e., the latch is opaque). This includes clock gated conditions. Grey non-bold text indicates that data is propagated through the latch (i.e., the latch is transparent). The two light-grey polygons to the left in the trace illustrate how the clock gated condition due to data being invalid propagates forward in the pipeline. The rightmost dark-grey polygon in the trace illustrates how the clock gated condition due to the stall propagates backward in the pipeline. These shaded polygons correspond to the similarly shaded regions in Figure 9.

Assume the clock is high and the pipeline is in steady state operation with two data items (or holes) continuously present in the pipeline. When data item A reaches stage 4 a stall is generated for two consecutive clock cycles. The stall condition is illustrated by the dark-grey polygon of the trace in Figure 10. In an elas-

[1] Note that deasserting a locally generated stall requires an event from an external non-stalled stage (not illustrated in Figure 8).

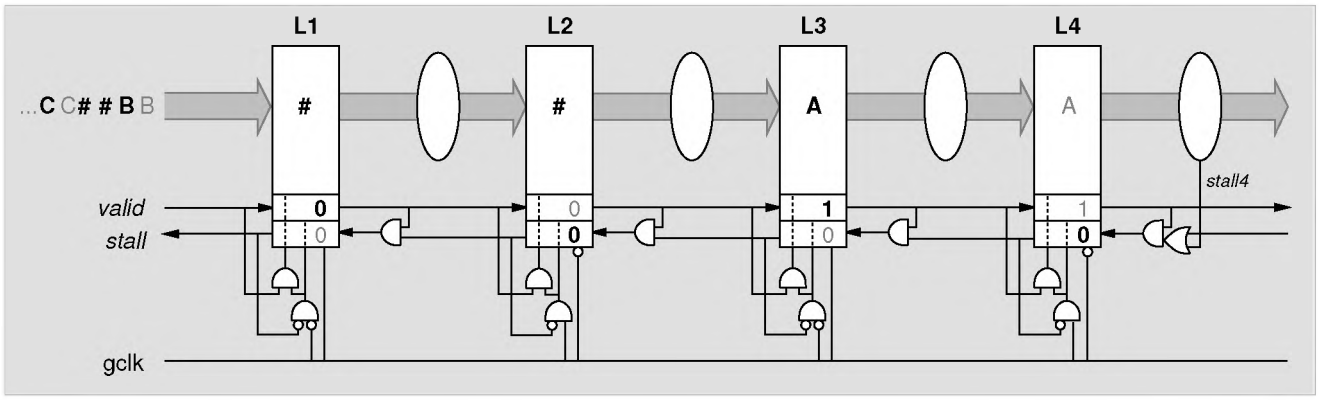

Figure 8. Two-phase clocked interlocked synchronous pipeline implementing forward and backward interlock.

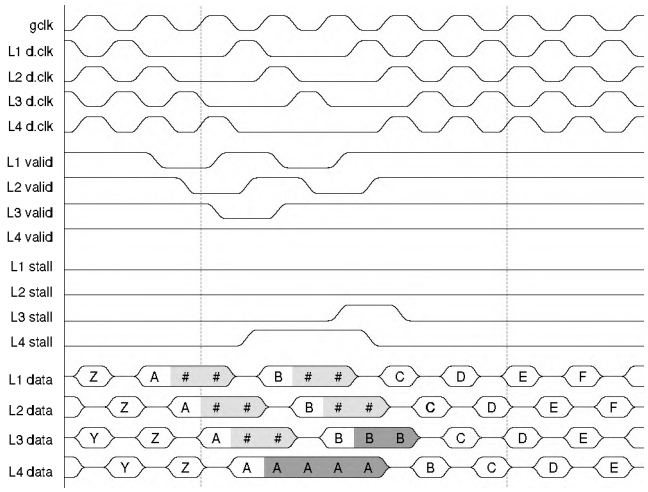

Figure 9. Waveform trace of two-phase clocked ISP.

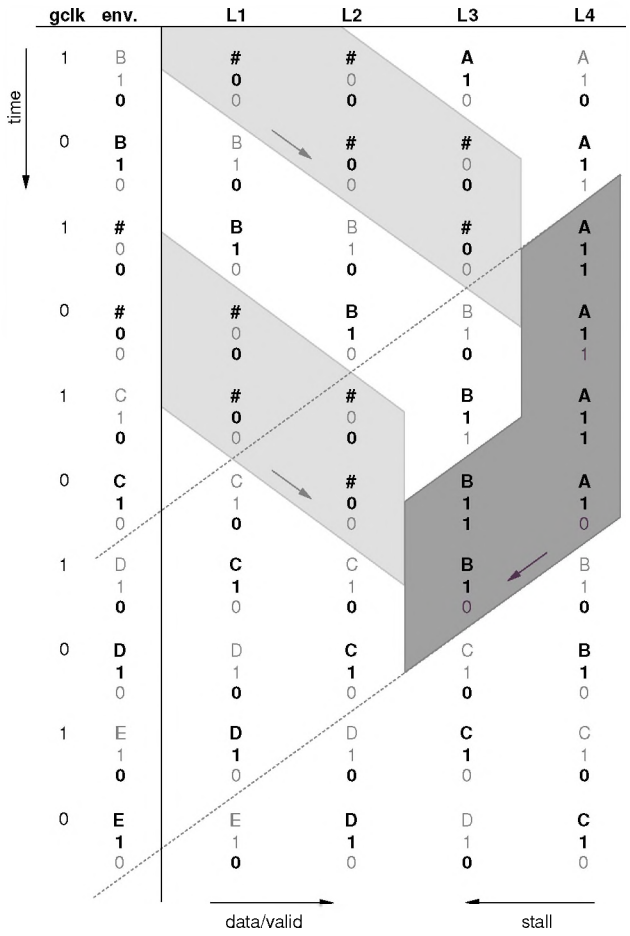

Figure 10. Detailed sub-trace of two-phase clocked ISP.

tic pipeline, the stall condition would have propagated backward in the pipeline unchanged, stalling each stage for two cycles as illustrated by the dotted lines. When a hole is encountered in an interlocked pipeline, however, the valid bit overrides the stall condition by zeroing out the stall signal. This is illustrated by the stall window (dark-grey polygon) in the trace getting truncated when it encounters an invalid window (light-grey polygons). The override in turn cancels out the invalid condition when the hole gets filled with valid data, resulting in the invalid windows getting truncated when encountering the stall window.

The data stream in Figure 10 contains two holes, one after data item A and one after data item B. In an interlocked pipeline, rather than stalling all stages for two cycles, stage 4 stalls for two cycles, while stage 3 stalls for only one cycle, and stages 2 and 1 do not stall at all. The reason for this is that the stall condition is shortened by one cycle at stage 3 where the invalid data following A overrides the asserted stall signal in order to fill in the hole in the pipeline. The first cycle of the two cycle long stall window is therefore zeroed out and is not propagated backward in the pipeline. Rather than being stalled in stage 2 for two cycles, data item B is instead propagated to stage 3 and stalled for only one cycle, thereby filling up the hole in the pipeline. Similarly, as the remaining second cycle of the stall window reaches stage 2, the invalid data following data item B zeroes out the stall window completely. Thanks to the holes in the pipeline, the stall condition never reaches the start of the pipeline, and the environment does not need to stall. Data items C and D in the data stream are therefore not stalled but rather propagate through the pipeline in a normal fashion.

5.2. Application to one-phase clocked pipelines

Although the elastic and interlocked pipelines have been presented in the context of two-phase clocked pipelines, the techniques work equally well for one-phase clocked pipelines based on master-slave latches. Consider the two-phase pipeline in Fig-

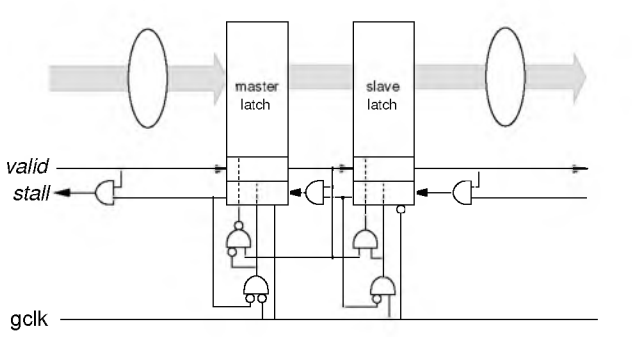

Figure 11. Master-slave based ISP segment.

ure 8. If the combinational logic between latches L1 and L2 and between L3 and L4 is removed, the pipeline would implement the behavior of a one-phase clocked master-slave pipeline. Latches L1 and L2 would then form one master-slave pair and latches L3 and L4 the other. Figure 11 illustrates the interlock logic for a one-phase clocked master-slave based pipeline segment.

The decision to clock gate in a one-phase pipeline is made at the end of each clock cycle, rather than at the first clock edge after new data has been received as in a two-phase pipeline. To guard against glitches in the local clock gating function, the valid signal must be in phase with the clock; therefore, the valid signal must be taken after the master latch, and the output inverter of the AND function gating the master data latch clock must be moved to the clock input instead. The gating functions of the master and slave clock signals subsequently both use the same phase of the valid signal.

5.3. Hazard and timing constraints

In order to ensure glitch-free (hazard-free) operation of the local clock gating functions, the valid and stall signals have to meet standard synchronous timing constraints for clock gating. Assume a new data item is propagated into a pipeline stage at clock edge e_0. The logic functions for the valid and stall signals may produce spurious glitches while evaluating in response to the data changes.

In a two-phase pipeline, glitches on the valid signal are filtered out by the clock gating function due to the polarity of the clock signal during the time the valid signal is computed. The valid signal has to stabilize in time before the next clock edge e_1 arrives at the gating function. In a one-phase pipeline, the valid signal is taken after, rather than before, the master latch. The gating function is protected from glitches during the first half of the clock cycle when the master latch is opaque. During the second half of the clock cycle, glitches on the valid signal are filtered out by the clock gating function due to the polarity of the clock signal. In a one-phase pipeline the valid signal has to stabilize before clock edge e_2 arrives at the gating function.

The timing constraints on the stall signal are the same for two-phase and one-phase pipelines. The gating function is protected from glitches on the stall signal during the first half of the clock cycle when the stall latch is opaque. During the second half of the clock cycle, glitches on the stall signal are filtered out by the gating function due to the polarity of the clock signal. The stall signal has to stabilize before clock edge e_2 arrives at the gating function. Note that delay gates are inserted on non-gated local clocks to zero out the skew introduced by the gating functions on gated clocks.

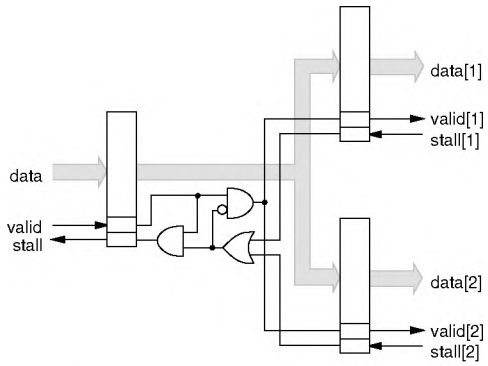

Figure 12. An ISP 1 to 2 fork stage.

6. Generalized interlocked pipeline structures

A typical pipelined asynchronous system is built on a set of data flow primitives that steer data to desired locations of the system. These primitives include pipeline forks, joins, branches, and select structures that can be used to build complex pipelined systems. This section illustrates how the valid and stall signals can be generalized to support such primitives in synchronous interlocked pipeline structures.

6.1. Fork structures

A pipeline fork stage is a 1 to N splitter that copies a data item from an upstream stage into N parallel downstream pipeline stages. A fork stage must stall if any of its downstream stages stalls. When a fork stage is stalled, non-stalled downstream stages must be prevented from receiving duplicate copies of the data from the stalled fork stage. The simplest way of implementing this functionality is through a synchronized, or aligned, fork stage where the valid signals to all downstream stages are zeroed out until all downstream stall conditions have gone away. The data is thus received by all downstream stages simultaneously. The logic required for such a fork stage can be expressed in terms of generally applicable logic template functions. The template functions for the valid and stall signals of a 1 to N synchronized fork stage are as follows:

$$stall = valid \wedge (stall[1] \vee ... \vee stall[N])$$
$$valid[i] = valid \wedge \neg(stall[1] \vee ... \vee stall[N])$$

Figure 12 illustrates the valid and stall logic needed for a 1 to 2 fork stage. Note that the local clock gating functions are not shown in the figure, only the logic needed to derive the valid and stall signals is shown. Alternatively, to model the behavior of an asynchronous fork stage, the fork stage can instead be implemented as a non-synchronized, or non-aligned, fork. Data is then copied to downstream stages on an individual basis as they become non-stalled, giving the computation in non-stalled downstream pipelines an early start. The valid and stall logic of non-synchronized forks has to be implemented as a state machine since state information is needed to keep track of whether data has already been copied to a downstream stage or not.

6.2. Branch structures

A pipeline branch stage is a 1 to 1-of-N selector that propagates data from an upstream stage to one of N parallel downstream stages. The decision to which of the downstream stages data is to be propagated is determined by the datapath logic that

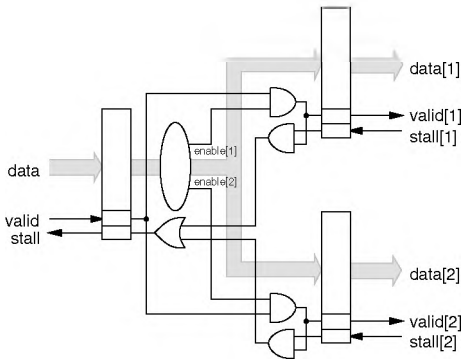

Figure 13. An ISP 1 to 1-of-2 branch stage.

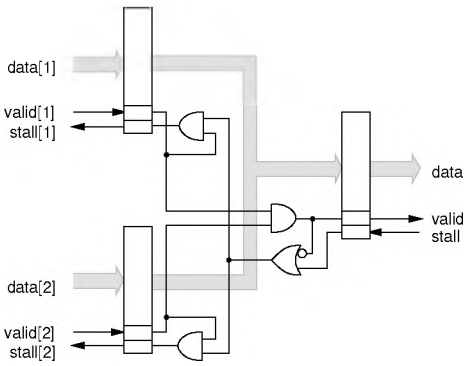

Figure 14. An ISP 2 to 1 join stage.

generates a set of N one-hot encoded enabling signals. These enable signals mask the branch stage valid signal through a set of AND functions such that a valid is propagated only to the selected downstream stage. The branch stage must be stalled only if an already stalled downstream stage is selected as the destination of the data. The logic template functions for a 1 to 1-of-N branch stage then become:

$$stall = valid[1] \wedge stall[1] \vee ... \vee valid[N] \wedge stall[N]$$
$$valid[i] = valid \wedge enable[i]$$

Figure 13 illustrates the valid and stall logic needed for a 1 to 1-of-2 branch stage.

6.3. Join structures

A pipeline join stage is an N to 1 merger that concatenates data from N upstream stages to one downstream stage. The join stage waits until data is valid in all upstream stages before concatenating and propagating the data to the downstream stage. A join stage is used to synchronize and align the data streams of multiple pipelines. Since data in different upstream stages can become valid at different times, any stage that becomes valid must be stalled until all stages have become valid, and, the data can be propagated to the downstream stage. If the join stage stalls all upstream stages must stall. The logic template functions for an N to 1 join stage are given below.

$$valid = valid[1] \wedge ... \wedge valid[N]$$
$$stall[i] = valid[i] \wedge (\neg valid \vee stall)$$

Figure 14 illustrates the valid and stall logic needed for a 2 to 1 join stage.

6.4. Select structures

A pipeline select stage is a 1-of-N to 1 selector that propagates data from one of N upstream stages to one downstream stage. A select stage implements a basic if-then-else multiplexer function. A select stage waits until data is valid in at least one of the upstream stages. One stage is then chosen through priority based selection and its data is propagated to the downstream stage. An upstream stage that contains valid data must stall until it is selected. The logic template functions for a 1-of-N to 1 select stage, where a higher index i indicates a higher priority, are given below.

$$valid = valid[1] \vee ... \vee valid[N]$$
$$stall[i] = valid[i] \wedge (stall \vee ((i < N) \wedge (valid[i+1] \vee ... \vee valid[N])))$$
$$data = if\ (valid[N])\ data[N]\ elsif\ ...\ elsif\ (valid[1])\ data[1]$$

Figure 15 illustrates a 1-of-2 to 1 select stage where stage 2 has priority over stage 1. Note that a select stage also implements the functionality of an arbiter. The priority scheme decides which upstream stage wins the arbitration and which competing stages, if any, must stall. State based selection, rather than priority selection, can be implemented through state machines.

6.5. Multicycle structures

A multicycle pipeline is an N-cycle circular pipeline structure (a ring) with an input and an output stage for reading in data from, and writing out data to, an environment. An out of order completion, fully interleaved, multicycle pipeline is illustrated in Figure 16. The input stage of the ring is implemented as a select stage and the output stage is implemented as a branch stage. The illustrated pipeline allows multiple multicycle computations to be interleaved in the ring hence allowing maximal throughput. Every cycle the feedback input to the select stage is checked for valid data. If the feedback input is not valid new data is instead read into the ring from the input environment, if available. In the branch stage, the datapath logic determines if the current data needs to continue iterating through the ring, or if it should be written to the output environment and generates an enabling signal accordingly.

7. Verification and testing

The presented pipelines have been formally verified to ensure functional and timing correctness of the implementation. Strategies for supporting testability have also been developed.

7.1. Verification

Prototypes of the proposed pipeline structures were implemented at the RTL and transistor level. The RTL models were used to check functional correctness of the pipeline interlocking. Transistor level SPICE models were used to check timing correctness. The pipelines were shown to operate correctly in simulation of these models.

To demonstrate correctness in a broader range of environment and timing scenarios, the pipeline structures were also verified using formal methods. A formal verification was realized using ATACS—a tool for the synthesis and verification of timed circuits [3]. The goal of the formal verification, unlike the RTL and SPICE analysis, is to demonstrate hazard freedom in the clock gating circuits under all possible timing configurations, not just a selected few. Verification begins with a behavioral specification of the pipeline structures using assumed timing bounds for each signal. Safety properties necessary for correct operation, such as hazard freedom in the clock generating circuits, are added to

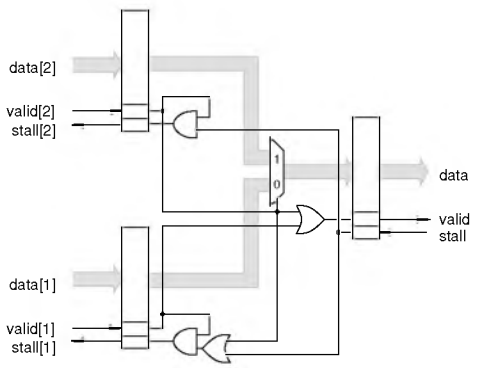

Figure 15. An ISP 1-of-2 to 1 priority based select stage.

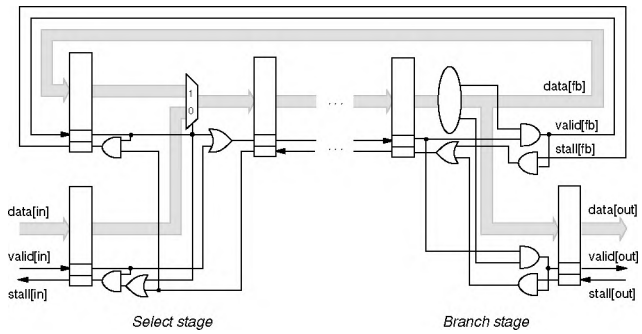

Figure 16. A fully interleaved multicycle ISP pipeline.

the specification. The specification is then verified to conform to the safety properties in all timing configurations through timed state space exploration. The enumerated state space is then used to synthesize a circuit that implements the specification. Actual timing bounds of the resulting gate netlist are then obtained from gate libraries and compared against the timing bounds in the specification. If the gate bounds fall within the specification bounds, then the circuit is correct by construction. Otherwise the timing bounds are updated and the verification process is repeated. Each of the pipeline structures presented in this work were verified in this manner. The synthesized clock gating circuits directly correspond to the interlocking logic illustrated in Figures 5, 8, and 11. These circuits are verified to never produce hazards on clock output signals and to operate correctly under synchronous timing assumptions in all of the presented pipeline structures.

7.2. Testability

Support for testability is a requirement when designing commercial chips. Consider a master-slave latch supporting LSSD based scan chains. Such master-slave latches usually contain three latches, a master latch, a slave latch, and a scan latch. During scan-in and scan-out of test patterns, the master latch is opaque and the scan latch instead acts as a master latch. Together, the scan and slave latches of a latch array form a scan chain through which test vectors can be scanned in and out. Note that the mentioned LSSD latch supports scan-in and scan-out of only one data item. When an elastic pipeline is stalled, however, both master and slave latches contain data.

There are several solutions for allowing scan-in and scan-out of two data items per master-slave latch for testing purposes. Scan-out can be supported without additional hardware by running identical test vectors twice, the first time scanning out data from the master latches, and the second time from the slave latches. Testing of initially stalled pipelines can also be supported without additional hardware since scan-in of distinguishable data items to master and slave latches can be achieved through bit offset techniques traditionally used in AC testing. Using these, or similar, techniques, the presented pipelines can be made fully testable using LSSD techniques without requiring extra scan latches.

8. Conclusions

This paper has presented interlocked synchronous pipelines that achieve interlocking between stages in a synchronous pipeline. Interlocked synchronous pipelines use valid bits propagating in synchronous lock-step with the data in the forward direction of the pipeline, and stall bits propagating in the backward direction of the pipeline together with local clock gating to achieve interlocking between pipeline stages. Generalized interlocked pipeline structures in the form of forks, joins, branches, and selects from which complex pipelined systems can be built have been demonstrated. The stages of an interlocked synchronous pipeline compute only on demand and perform stalls locally on a stage by stage basis. Clock gating based on invalid data and stall conditions has the potential to save significant power. Progressive local stalling, in turn, has the potential to reduce variance in switching currents and to increase slack on stall signals facilitating design of very high frequency stallable pipelines. The presented techniques are directly applicable to traditional synchronous pipelines and work equally well for two-phase clocked pipelines based on transparent latches, as well as one-phase clocked pipelines based on master-slave latches.

The interlocked synchronous pipelines presented in this paper demonstrate a first step towards a middle ground between asynchronous and synchronous pipelines. The interlocked synchronous pipelines can, in a synchronous context, achieve several beneficial properties previously only found in asynchronous pipelines.

Acknowledgements. The authors would like to thank Wendy Belluomini for help with verification, Phil Emma, Bruce Fleischer, David Heidel, and Phil Restle for helpful discussions, and David Kung and Leon Stok for helpful discussions and management support.

References

[1] GOWAN, M., BIRO, L., AND JACKSON, D. Power considerations in the design of the Alpha 21264 microprocessor. In *Proc. ACM/IEEE Design Automation Conference* (1998), pp. 726–731.

[2] MCLELLAN, E. J. Reducing stall delay in pipelined computer system using queue between pipeline stages. *Digital Equipment Corporation, U.S. patent 5325495* (1994).

[3] MERCER, E. G., MYERS, C. J., AND YONEDA, T. Modular synthesis of timed circuits using partial order reduction. In *Proc. of The Theory and Practice of Timed Systems* (Apr. 2002).

[4] SCHUSTER, S., REOHR, W., COOK, P., HEIDEL, D., IMMEDIATO, M., AND JENKINS, K. Asynchronous interlocked pipelined CMOS circuits operating at 3.3-4.5 GHz. In *International Solid State Circuits Conference* (2000), pp. 292–293.

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