Static Scheduling of Multiple Asynchronous Domains For Functional Verification

Murali Kudlugi **Charles Selvidge**
Emulation Systems Group
IKOS Systems Inc.
Waltham, MA
{murali,selvidge}@ma.ikos.com

Russell Tessier
Dept. of Electrical and Computer Engg.
University of Massachusetts
Amherst, MA
tessier@ecs.umass.edu

Abstract

While ASIC devices of a decade ago primarily contained synchronous circuitry triggered with a single clock, many contemporary architectures require multiple clocks that operate asynchronously to each other. This multi-clock domain behavior presents significant functional verification challenges for large parallel verification systems such as distributed parallel simulators and logic emulators. In particular, multiple asynchronous design clocks make it difficult to verify that design hold times are met during logic evaluation and causality along reconvergent fanout paths is preserved during signal communication. In this paper, we describe scheduling and synchronization techniques to maintain modeling fidelity for designs with multiple asynchronous clock domains that are mapped to parallel verification systems. It is shown that when our approach is applied to an FPGA-based logic emulator, evaluation fidelity is maintained and increased design evaluation performance can be achieved for large benchmark designs with multiple asynchronous clock domains.

1. Introduction

To meet the need for more complete functional coverage, many ASIC designers are turning to parallel verification platforms which perform many logical evaluations concurrently. These systems, which include logic emulators [6,7,8] and parallel cycle-based simulators, often contain special-purpose logic processors or FPGAs which evaluate logic and communicate results using a high-speed system clock. Since numerous logic evaluations are required per user design clock cycle, a fixed relationship must be created between the behavior of the system and design clocks. This relationship can then be used to determine when specific logic functions should be evaluated inside logic processors and when data should be communicated between processors. Multiple design clocks with known phase relationships can also easily be handled by deriving a base frequency which can be used for logic and communication scheduling.

A significant problem arises when ASIC designs with multiple asynchronous timing domains are mapped to parallel logic verification hardware [6]. The unknown phase relationship between domains can make it difficult to determine when individual logic functions should be evaluated (the hold-time problem), and when inter-processor communication should be performed (the multi domain transport problem). As an example, for logic emulators [1,3], special compilation and/or manual steps have been required to isolate individual asynchronous domains in hardware by directly mapping communication paths to special system hardware at the expense of performance and mapping flexibility. Not only has this approach been difficult and time-consuming, but often results are unpredictable, leading to verification flaws.

In this paper, we present a general approach to address the parallel verification of designs with multiple asynchronous clock domains. The basis of this approach is the formulation of a set of constraints which can be integrated into logic evaluation and processor communication scheduling. It will be shown that this approach can be scaled to handle an unlimited number of asynchronous domains and can achieve provable modeling fidelity. After formulating the scheduling problem and describing our general approach, a discussion of the integration of our algorithms with a commercial FPGA-based logic emulation system from Ikos Systems is provided [7]. It is shown that the new constraints and algorithms achieve modeling fidelity and overall system performance improvement versus a previous "hard-wired" approach for two large commercial ASIC designs that contain multiple asynchronous clock domains.

2. Background

The approach described in this paper can be applied to numerous logic emulation and parallel cycle-based systems that statically-schedule logic computation and signal communication at compile-time. Example verification architectures that fit this model include Quickturn CoBalt [8] and Arkos emulators [6], and Ikos VirtuaLogic emulators. The target system for this paper is an Ikos VirtuaLogic emulation system that contains 384 Xilinx XC4062XL FPGAs.

Inter-FPGA communication in VirtuaLogic systems is based on Virtual Wires technology, an inter-FPGA communication scheduling technique. This approach pipelines multiple logical signals called *Virtual wires* across single inter-FPGA wires to overcome FPGA pin limitations[3, 4]. Logic designs are mapped to multi-FPGA VirtuaLogic systems through a series of compilation steps. These steps include design partitioning into logic blocks small enough to fit within FPGAs, placement of logic blocks onto specific FPGAs, and scheduling of both intra-FPGA logic evaluation and inter-FPGA communication. Both logic evaluation and signal communication are controlled by a high-speed clock called a *Virtual Clock* which serves as a discrete timebase, providing a reliable mechanism for controlling the order of events at a fine granularity. Since many combinational evaluations and signal transfers may occur in a single design clock cycle, the virtual clock by necessity runs at a much higher frequency than the design clock. For logic emulation systems, inter-FPGA (processor) communication scheduling is based on the virtual clock.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.
Copyright 2001 ACM 1-58113-297-2/01/0006…$5.00.

3. Multi Domain Problems

There are a number of problems that make multi domain circuits interesting and challenging from a functional modeling point of view.

Modeling Logic In Multiple Domains

Functional Axiom 1: ***Timing Closure***

Combinational logic plus transmission delay plus setup time between two sequential elements in the same domain takes less than one clock period of the fastest clock attached to either of the sequential elements.

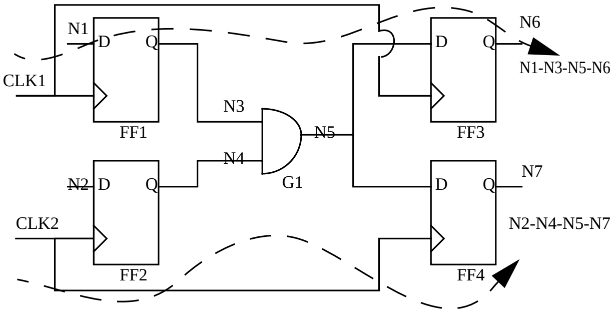


Figure 1: A Multi Transition and Sample Domain (MTSD) Example.

Consider the circuitry shown in Figure 1 where two asynchronous clocks CLK1 and CLK2 drive state elements (FF1,FF3) and (FF2, FF4) respectively. This circuit contains two same domain paths, FF1.Q-N3-N5-FF3.D in the domain of CLK1 and FF2.Q-N4-N5-FF4.D in the domain of CLK2. Note that the net N5 transitions and is sampled in both clock domains. It is called a ***MTSD*** *(Multi Transition and Sample Domain)* net. The correct functional model of this circuit must simultaneously satisfy the timing closure axiom in each constituent domain. This means the data at FF1.Q must reach FF3.D in exactly one cycle of CLK1 and data at FF2.Q must reach FF4.D in exactly one cycle of CLK2 irrespective of combinational delays or multi domain segments in the paths.

Transporting Multi Domain Values

Functional Axiom 2: **Causality**

The occurrence times of combinational logic form a partial order based on causality. If part A feeds part B, events on A must have occurred before events on B.

Another verification issue involves the transport of multi value signals in a system where inter-FPGA communication needs to be synchronous to a system clock (e.g. virtual clock). Previous work suggests that we either avoid such a situation by limiting the size of asynchronous-domain logic to one FPGA or dedicate special inter-FPGA wires to transport the values (hard-wiring) [3]. Since hard-wired signals cannot be multiplexed to carry non-MTSD nets, pin limitation problems [2] can result leading to reduced system performance. To avoid this problem, it is desirable to split multi domain values into constituent domain values and to route (schedule) them in respective domains and recover the multi-domain value at the destination FPGA or processor. This solution poses another problem because of unpredictable route timing that is inherent in statically routed systems. Consider a situation where the circuit in Figure 1 is partitioned such that the multi domain value N5 needs to cross over an FPGA boundary to another FPGA that may be located some routing distance away. Due to unpredictable routing delays such as routing congestion, it is possible for the domain1 (D1) value of N5 to start from the source FPGA sooner than the domain2 (D2) value but still arrive after the D2 value reaches its destination. This can break the causality principle and cause the clobbering of the D2 value. Figure 2 illustrates such a case. One key requirement in transporting multi-domain signals is to ensure that causality of events is guaranteed within each of the constituent domains irrespective of routing delays.

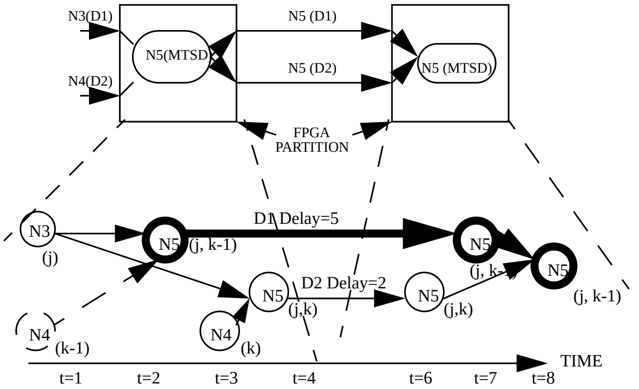


Figure 2: Transporting Multi Domain Values.

Hold Time Problem in Multi Domains

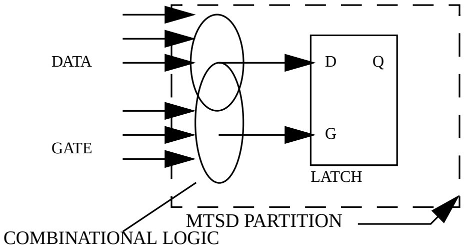
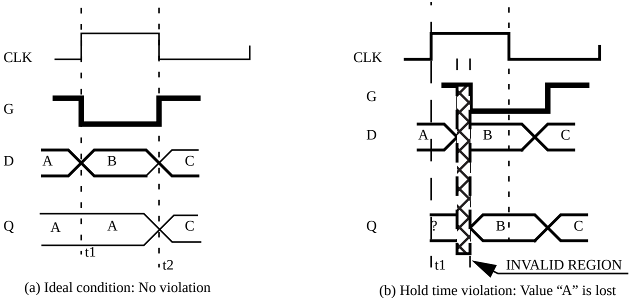


Figure 3: MTSD Latch Example.

The correct functioning of state elements requires that data signals arrive at an element a certain period of time (setup time) before the triggering signal and are held steady for certain period of time (hold time) after the triggering signal arrives. If the triggering signal arrives at a time when the data signal is invalid, a violation occurs and causes incorrect operation of the circuit. This is a very common problem in delay sensitive circuits. Consider a simple latch shown in Figure 3, which has combinational logic sourcing it’s Gate and Data inputs, and the waveforms shown in Figure 4. Here D, G and Q represent Data, Gate and Output waveforms of the latch. Figure 4(a) shows an ideal condition where the edge on user clock CLK at t=t1 causes a change in Gate and Data values at the same instant of time and the old value “A” gets stored in the latch as a result. Figure 4(b) shows more realistic waveforms where routing delays cause the Gate and Data to arrive at the latch inputs at different points in time in response to CLK. A problem arises if the new data “B” reaches the latch sooner than the new gate and clobbers the old value “A”. This can happen if the routing delay on the Clock/Gate path is greater than the routing delay on the Data path due to combinational logic in those paths. In a case where both Gate and Data paths are in the same domain, it is easy for a scheduler to compute regions of time when Gate is invalid and mask those regions so that latches are not evaluated. This solution fails if Data and Gate nets are multi domain nets because regions of validity for latch evaluation in one domain may conflict with regions of invalidity in other domains. The key challenge here is to satisfy hold time requirements for every (D,G) pair in each of the constituent domains.



(a) Ideal condition: No violation

(b) Hold time violation: Value "A" is lost

Figure 4: Hold Time Violation in MTSD States

4. Definitions

MTSD Net. A net which transitions (changes value) and is sampled (read) by more than one clock domain. In Figure 1, net N5 is a MTSD net.

MTSD Gate. Any combinational gate whose output is connected to an MTSD net. In Figure 1, gate G1 is a MTSD gate.

MTSD State. A latch/flip-flop whose gate/clock input is sourced by a multi transition net.

MTSD Block. The MTSD logic is partitioned into chunks of size that are small enough to fit into a FPGA. It is at the block boundary all the inter-FPGA communication (routing) takes place.

5. The Approach

Observation 1:

For any relationship Ri(A, B) in a multi domain circuit containing domains A and B, it is sufficient to satisfy Ri(A) and Ri(B) for correct functional verification.

For example, in the circuit showing Figure 1, we only need to satisfy the timing closure property for the same domain paths FF1.Q-N3-N5-FF3.D and FF2.Q-N4-N5-FF4.D but not for the cross domain paths FF1.Q-N3-N5-FF4.D or FF2.Q-N4-N5-FF3.D. Similarly hold time must be satisfied for each same domain (D,G) pairs. Essentially, the multi domain problem reduces to satisfying functional requirements within each of the constituent domains simultaneously.

Multi-Domain Data Transport

Inter-FPGA data transport of an MTSD net can be decomposed into the independent transport of a set of signal components from each domain which are causally merged at the destination. We represent these flows by adding FORK/MERGE operator pairs at FPGA boundaries, resulting in a set of same domain signals on FPGA boundaries. From *Observation 1*, notice that flow and dependence relationships on intra-FPGA paths only need to consider combinationally connected signals from the same domain. Causal merging can be accomplished by dynamically selecting an appropriate single domain signal at a MERGE point. Our scheduler ensures that the transport delays of paths from independent domains are equal so that the value arriving at the merge point is guaranteed to be causally correct.

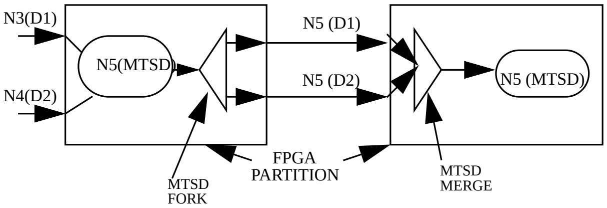


Figure 5: Multi-Domain Data Transport.

Hold Time Constraints on a MTSD Latch

Observation 2:

For a multi domain latch, instantaneous Setup time violations are correctable whereas instantaneous Hold time violations are not.

The basis of *Observation 2* stems from the fact that when a latch is evaluated with OLD data against a new gate, (a Setup time violation), new data arrival results in re-evaluation of the latch output if the latch gate is open. If the latch gate is closed, OLD data does not corrupt the output. Alternatively, if a latch is evaluated with NEW data against an OLD gate that is open (a Hold time violation), the correct latch value may be irretrievably lost since the OLD data is no longer available.

We will use notation V(Ai,Bk) to indicate the value of signal V which occurs in response to the ith clock edge of domain A and kth clock edge of domain B.

For any latch with Data D(Ai, Bk) and Gate G(Aj, Bk) on some clock edge k in Domain B, we have three possible conditions:

- (i < j) => instantaneous Setup time violation.
- (i == j) => both Setup and Hold Time satisfied
- (i > j) => instantaneous Hold time violation

Observation 1 implies that every edge k in domain B requires an evaluation of the latch with D(Ai, Bk) against G(Aj, Bk) which satisfies both setup and hold time with respect to B. *Observation 2* implies that when performing such an evaluation, it is legitimate to have i < j or i == j but not legitimate to have i > j. The symmetrical relationship holds with respect to evaluations against A. This relationship can be extended to an arbitrary number of domains.The implication of this is that every edge for any domain results in an evaluation which satisfies both setup and hold time with respect to that domain and any evaluation not satisfying setup against some domain is subsequently followed by a correcting evaluation against that domain.

Our new static scheduler algorithm ensures that the Data never arrives before Gate for any edge on any domain and that both arrive prior to subsequent edges from that domain.

Transforming MTSD flip-flops

MTSD flip-flops are not covered by O*bservation 2* and are more difficult to address than MTSD latches. Our approach to handle MTSD flip-flops is to transform them into master slave latch pairs and then subject them to the same processing as other MTSD latches.

6. Static Scheduling

We have used a modified TIERS scheduling algorithm to route communication paths between blocks[4]. This is a reverse scheduling algorithm in that it routes paths starting from primary outputs to primary inputs. Note that the techniques explained are also applicable to forward routing. In this section we describe the basic steps involved in static routing. In the next section we describe in detail the specific steps involved in MTSD latch scheduling.

A *route-link* (Pi, Pj) represents a logical connection from block output terminal Pi to block input terminal Pj located on a different FPGA. A route-link often has to cross multiple FPGAs before reaching its destination. We calculate link depths that represent the longest time required to propagate through the network from the source FPGA to the destination FPGA. We create a partial order by sorting route-links by depth to ensure that all the route-links upon which a given route-link depends are scheduled before the route-link itself. The core scheduling algorithm involves the following steps:

For each route-link(Pi,Pj),

1. Find the latest time, called *ReadyTime* at which a value must arrive at its destination for further evaluation. For Pj terminating at design primary output k, ReadyTime is Delay(Pj to k).
2. Find the shortest path ‘sp’ from Pi to Pj such that data arrives by ReadyTime(Pj). We use a modified Dijkstra’s algorithm[6].
3. Reserve wiring resources along the path sp.
4. Compute *DepartureTime*(Pi) at the source Pi:
   DepartureTime(Pi) = ReadyTime(Pj) - PathLength(sp)
5. Update input ReadyTimes at the block,
   for each terminal Pk in Parent(Pi)
      ReadyTime(Pk) = DepartureTime(Pi) - Delay(Pk to Pi)

7. MTSD Scheduling

In this section we focus on how MTSD paths are scheduled such that we satisfy hold time requirements on every MTSD latch in each of the constituent domains.

MTSD Dependency and Depth

The key issue here is to ensure that we create a causally correct order of route-links such that when scheduled, the order satisfies the dependency between route-links in a given combinational path. The MTSD paths between fork and merge are split into a group of route links that belong to different domains which collectively transport the MTSD value across FPGAs. If the scheduler can ensure that these route-links are scheduled such that they all take an equal number of virtual clocks to propagate the value, the causally correct value can be easily regenerated at the destination.

To aid scheduling, we compute two types of dependency classes, *SameDomainDependency* which tracks link dependencies within a single domain and *AllDomainDependency* which tracks link dependencies within all domains including cross domain paths. We use AllDomainDependency to sort all route links in all domains and to produce a partial order that is consistent in each of the domains. During scheduling, we are only interested in following same domain paths to compute an optimal schedule[4]. In addition we compute MinDelay(i, L) and MaxDelay(i, L) for each block input terminal i to latch L as there can be multiple combinational paths from a block input terminal to a latch.

MTSD Latch Ordering

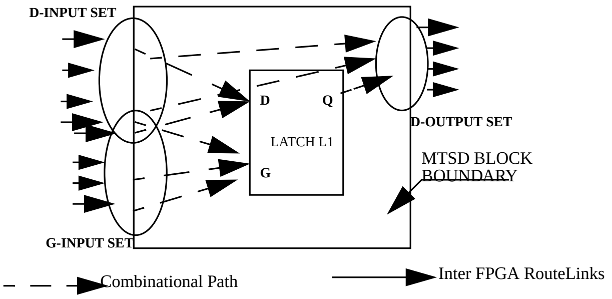


Figure 6: MTSD Latch Ordering.

As noted earlier, to satisfy the hold time of a MTSD latch, we need to schedule such that Gate signal information from output to input arrives at the latch at or before the time the Data arrives. This imposes an additional ordering requirement on route-links. Here we describe the scheme that helps to compute the evaluation order of route-links and latches. To aid in latch ordering, we analyze each MTSD partition and create the following block terminal sets for each latch as shown in Figure 6:

- D-INPUT Set: Group of all MTSD Block terminals that combinationally reach the Data terminal of the latch. This includes any input that reaches both Data and Gate (called DG input).
- G-INPUT Set: Group of all Gate only inputs to a latch.
- D-OUTPUT Set: Group of all block terminals which are dependent children of terminals in D-INPUT set.

Latch Groups

Since the same block terminal can combinationally reach more than one latch, it is necessary to analyze evaluation order between latches. This imposes additional constraints on the order in which block terminals need to be scheduled. We analyze each MTSD partition and compute the following latch relationships:

- DD-type: When a block terminal is connected to Data inputs of two or more latches, it represents a sibling relationship between latches. We combine such latches by merging their D-INPUT and D-OUTPUT sets so that these latches are evaluated together.
- DG-type: If a block input reaches Data of latch L1 and Gate of latch L2 then the latch L2 needs to be scheduled before the latch L1, since Gate terminals are evaluated first. This essentially forms a Parent-Child dependency relationship between L2 as parent and L1 as child. DGChild(L) contains a set of

latches which must be evaluated after latch L.

- DG-Cycle: When we have a cyclical DG relation involving two or more latches, the only way to satisfy DG constraints on all latches is to evaluate all of them simultaneously. We treat this case the same as DD-types by combining latch D-INPUT and D-OUTPUT sets.

Computing MTSD Latch Dependency

The DG constraint implies that: D-INPUT terminals must be evaluated after all of the dependent G-INPUT terminals are evaluated but before the latch itself is evaluated. This constraint must hold valid in each of the same domain (Di,Gj) pairs for Di in the D-INPUT Set and Gj in the G-INPUT set. Essentially we are introducing two types of dependencies into the system:

- Dependency introduced between terminals in the D-INPUT set and terminals in the G-INPUT set.
- Dependency introduced between latches/latch groups due to DG relationships.

We use these dependencies to order latch route-links with other route-links.

Latch Evaluation

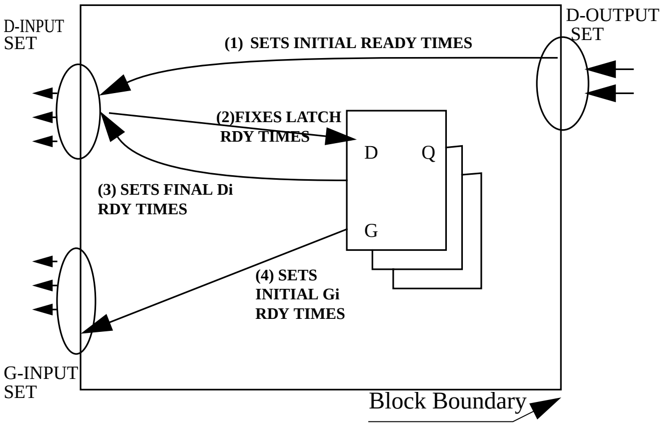

D-INPUT SET
(1) SETS INITIAL READY TIMES
D-OUTPUT SET
(2)FIXES LATCH RDY TIMES
D
Q
(3) SETS FINAL Di RDY TIMES
G
(4) SETS INITIAL Gi RDY TIMES
G-INPUT SET
Block Boundary

Figure 7: MTSD Latch Evaluation.

Figure 7 illustrates the basic steps involved in Latch scheduling. Due to the latch order described earlier, by the time a latch gets evaluated, we know the *DepartureTimes* of all the terminals in it's D-OUTPUT set. Note that in the above diagram arrows indicates the flow of ReadyTime, the time at which the value must be ready for consumption by dependent logic. The ready-time evaluation sequence is indicated by the numbers in the parenthesis. This algorithm computes the final ready time on D-INPUTs and the lower bound for the ready times on G-INPUTs.

For each latch L,

1. Compute the initial ReadyTimes for each terminal Di in D-INPUT(L) set based on the DepartureTimes of their fanouts(D-OUTPUTs). Note that these are not final ReadyTimes because they do not take into account the latch's ReadyTime.

For each output terminal Oj in DependChild(Di),

ReadyTime(Di) = MAX (DepatureTime(Oj) - MaxDelay(Di to Oj))

2. Evaluate the difference between each ReadyTime(Di) with the ReadyTime(L) and if the difference is less than the minimum delay from Di to the latch L, then update the ReadyTime(L).

For each Di in D-INPUT(L),

ReadyTime(L) = MAX (ReadyTime(Di) + MinDelay(Di to L))

3. If latch L has DG relationship on other latches, take the maximum ReadyTime of child latches:

For each child latch Lc in DGChild(L),

ReadyTime(L) = MAX (ReadyTime(L), ReadyTime(Lc))

4. For each Di in D-INPUT(L),

4.1. Compute the *RequiredReadyTime*. The value is called *required* ReadyTime because, if data arrives any sooner than that time, there is a risk of violating DG Constraint.
RequiredReadyTime(Di) = ReadyTime(L) - MinDelay(Di to L)

4.2. Compute the final ReadyTime. This is the ReadyTime that is used by parent links of Di for further computation
ReadyTime(Di) =
MAX (ReadyTime(Di), RequiredReadyTime(Di))

4.3. If *ReadyTime(Di)* is greater than *RequiredReadyTime(Di)*, add delay compensation in the Di to L path to ensure that Data does not arrive at the latch sooner than required.
DelayCompensation(Di, L) =

ReadyTime(Di) - RequiredReadyTime(Di)

A delay equal to *DelayCompensation(Di,L)* is injected in the path from Di to latch L by adding a chain of Virtual Clock triggered flip-flops.

5. Propagate *ReadyTime(L)* to each of the terminals in G-INPUT(L) as initial *ReadyTime(Gi)*. This is initial ReadyTime because there could be other dependent children on Gi which can further alter its ReadyTimes.

For each Gi in G-INPUT(L),

ReadyTime(Gi) =

MAX (ReadyTime(Gi, ReadyTime(L)) - MaxDelay(Gi to L))

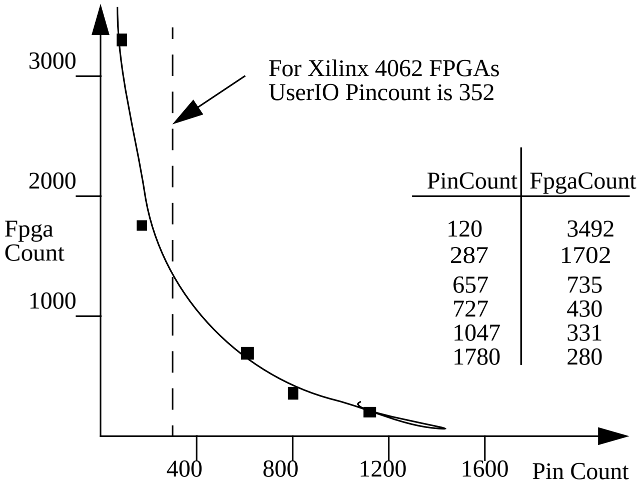
With the above algorithm it is guaranteed that the *ReadyTime(Gi)* is always less than or equal to *ReadyTime(Dj)* for any (Gi, Dj) pair on a given latch which ensures that the Gate value always arrives before the Data value on any MTSD latch. Notice that in the above equations, we have used MinDelay from Data terminals to Latches but instead have used MaxDelay from Gate terminals to Latches. This is to ensure that the delay from any Gi to a Latch does not exceed the delay from Di to the Latch after compensation (performed in step 4.3). Without this compensation it is still possible to violate hold time requirements at the latch even if DG constraints at the block boundary are met.

8. Experimental Results

We have implemented the algorithms described in this paper and integrated them into the Ikos VirtuaLogic Compiler[8] for the VStation-5M Emulator. We have taken two industrial designs containing asynchronous domains and compiled them using the VirtuaLogic compiler. Design1 has a smaller percentage of MTSD logic when compared to Design2 and has fewer memory modules. Table 1 compares the results of scheduled MTSD Virtual routing to hardwire routing. Note that Virtual routed wires and pins are multiplexed to achieve better FPGA pin utilization while hard routed wires require dedicated physical wires and pins. To determine the results for hard routing experiments we ran a pre-routing step which reserved physical pins between source and destination FPGAs for each MTSD wire and removed those pins from consideration during virtual routing of non-MTSD wires. Note that the number of Virtual clocks in the critical path for Design2 is much higher than Design1. This is because experiments for Design2 were dominated by memory transactions. It can be seen that the MTSD routing results in a slightly smaller number of Virtual clocks (hence faster execution) as compared to the hard wired approach. This is because if some physical wires are removed, the remaining wires have to carry a greater load of non-MTSD communication. Maximum emulation clock speeds in rows 10 and 11 are estimated based on a 34 MHz Virtual clock on a VStation-5M Emulator. To further illustrate the usefulness of virtual routing, we varied the partition sizes for Design1 and compared the resulting IO pincounts. Figure 8 shows the number of FPGAs needed vs. per-FPGA Pincounts. Since there is a hard limit on the number of pins on a FPGA, unless time scheduled Virtual routing is used, it is necessary to reduce the partition size in order to keep the pincount below this hard limit. This results in a need for substantially more FPGAs to fit the same design for hard routing versus Virtual routing.

Testcase	Design1	Design2
1. Num. Total Modules	543000	57000
2. Num. MTSD Modules	3100	7400
3. Num. Clock Domains	3	2
4. Num. MTSD Paths	173	213
5. Num. MTSD FPGAs	23	24
6. Clock Domains	d1 d2 d3	d1 d2
7. Num. Non MTSD FPGAs	11 43 180	4 7
8. Critical Path (Virtual-Clocks) MTSD Hard Routed	42 47 49	85 131
9. Critical Path (Virtual-Clocks)MTSD VirtualRouted	37 38 46	68 108
10. Est. Max Speed MTSD HardRouted	346 KHz	129 KHz
11. Est. Max Speed MTSD VirtualRouted	369 KHz	157 KHz

Table 1: MTSD Virtual Routing vs. Hard Routing



PinCount	FpgaCount
120	3492
287	1702
657	735
727	430
1047	331
1780	280

Figure 8: Hard Routing Vs. Virtual Routing.

9. Conclusions

In this paper we have described a new, general approach for dealing with multiple asynchronous clock domains in parallel functional verification systems. The goal of this work was to automatically handle multi domain asynchronous designs with higher modeling accuracy while maintaining high performance. The approach has been demonstrated on a VirtuaLogic emulation system for two large commercial benchmark designs. Experimental results show that the approach is scalable and provides good modeling fidelity. As a result of this scalability, an improvement in overall system performance was also obtained.

We plan to extend this approach to deal with memories under test and hard-wired cores. The heterogeneous nature of these blocks presents special considerations for scheduling and interfacing.

10. Acknowledgments

We thank Ken Crouch and Matt Dahl for many insightful discussions on scheduling and routing.

11. References

[1] Gopi Ganapathy, et al., "Hardware Emulation for Functional Verification of K5", Proceedings, 33rd Design Automation Conference, June 1996.

[2] J. Babb, R. Tessier et al. “Virtual Wires: Overcoming Pinlimitations in FPGA based logic emulators”. In Proceedings of IEEE Workshop on FPGA based Custom Computing Machines, pages 142-151, Napa, CA, April 1993.

[3] J. Babb, R. Tessier, et al. “Logic Emulation and Virtual Wires”. In IEEE Transactions on CAD, June 1997, Vol 16, No.6, Pages 609-626.

[4] C. Selvidge, et al. “TIERS: Topology Independent Pipelined Routing and Scheduling for VirtualWire Compilation”. In Proceedings of FPGA’95, pages 25-31, Berkeley, CA, Feb 1995.

[5] Corman et al. Introduction to Algorithms, MIT Press, 1992.

[6] Shekhar Patkar and Pran Kurup, "ASIC Design Flow Scores on First Pass", Integrated Systems Design Magazine, Aug 1997.

[7] IKOS Systems Inc, VirtuaLogic Datasheet, http://www.ikos.com/products/vsli/index.html

[8] Quickturn Design Systems, Cobalt Data Sheet, http://www.quickturn.com/products/cobalt.htm