Static Scheduling of Multi-Domain Memories 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

Over the past decade both the quantity and complexity of available on-chip memory resources have increased dramatically. In order to ensure accurate ASIC behavior, both logic functions and memory resources must be successfully verified before fabrication. Often, the functional verification of contemporary ASIC memory is complicated by the presence of multiple design clocks that operate asynchronously to each other. The presence of multiple clock domains presents significant challenges for large parallel verification systems such as parallel simulators and logic emulators that model both design logic and memory. Specifically, multiple asynchronous design clocks make it difficult to verify that design hold times are met during memory model execution and causality along memory data/control paths is preserved during signal communication. In this paper, we describe new scheduling heuristics for memory-based designs with multiple asynchronous clock domains that are mapped to parallel verification systems. Our scheduling approach scales to an unlimited number of clock domains and converges quickly to a feasible solution if one exists. It is shown that when our technique is applied to an FPGA-based emulator containing 48MB of SRAM, evaluation fidelity is maintained and increased verification performance is achieved for large, memory-intensive circuits with multiple asynchronous clock domains.

1. Introduction

As ASIC design sizes have expanded over the past decade, parallel verification platforms, which have the capability to perform many logic evaluations and memory operations concurrently, have increased in popularity. These systems, which include logic emulators [7, 10] and parallel cycle-based simulators [13], often use special purpose logic processors or FPGAs for logic evaluation and companion memory devices for memory emulation. Memory accesses in these systems are usually synchronized to a high-speed system clock to achieve maximum verification system performance. In general, these memory accesses require multiple discrete verification steps per user design cycle to accommodate address and control signal setup and data sampling. As a result, a fixed relationship must be created between the behavior of the verification system clock and the design clocks that control access to memory structures in the verified design. This relationship can then be used to determine exactly when memory control signals should be updated and when data may be safely sampled.

A significant problem arises when memory-intensive ASIC designs with multiple asynchronous timing domains are mapped to parallel verification systems [5]. The lack of a fixed phase relationship between design clock domains complicates data evaluation for both computation and communication. Specifically, it is necessary to determine when individual memory operations should be evaluated (the hold-time problem) and when inter-processor communication of memory access results should be performed (the multi-domain transport problem). Previously, the translation of design memory to verification system data storage has required manual steps to isolate asynchronous domains and achieve accurate memory modeling behavior [4, 5, 6]. This manual approach has often required significant effort and can lead to verification flaws.

In this paper, a heuristic approach to performing parallel verification of memory-intensive designs with multiple asynchronous clock domains is described. The approach analyzes memory port dependencies across multiple clock domains and subsequently schedules memory evaluation. Our technique is applicable to a variety of parallel verification systems and is scalable to an unlimited number of asynchronous clock domains. To validate the developed approach, our algorithms were integrated with a commercial FPGA-based emulation system from Ikos Systems [7]. For two memory-intensive designs it is shown that modeling accuracy for memory-based designs with asynchronous clock domains can be achieved without the need for time-consuming manual mapping steps. For both designs, an improvement in overall verification system performance is shown.

2. Background

The approach described in this paper can be applied to numerous logic emulation and parallel cycle-based simulators that statically schedule memory accesses at compile time. Example verification architectures that fit this model include the Tharas Hammer [13] parallel simulator, Quickturn CoBalt emulator [10] and Ikos VirtuaLogic emulator [7]. For these systems, the derived relationship between the high-speed verification system clock and the emulated design memory is determined from detailed memory mapping. To support memory verification, these systems [7, 13] contain collections of single-ported SRAM devices that can be used to emulate a range of design memory structures. During the design compilation process, control logic, which is sequenced by the system clock, is inserted into the emulated design to allow for the evaluation of complex design memory using simpler physical memory devices. Memory accesses are scheduled relative to the system clock to avoid data, address, and control signal conflict. In a manner similar to high-level synthesis [11], multiple small design memories can be packed into a single physical memory located in the parallel verification hardware. Unlike memory scheduling in high-level synthesis [8], however, verification-based memory scheduling must be per-

formed at a fine-grained level and as a result can be adversely affected by multiple asynchronous design clock domains.

The target system for this paper is an Ikos VirtuaLogic emulation system that contains 384 Xilinx XC4062XL FPGAs for logic emulation and 192 64Kx32 single-ported SRAM devices for memory emulation. Inter-FPGA communication and memory accesses in VirtuaLogic systems are based on Virtual Wires technology, an inter-FPGA communication scheduling technique [12]. This approach pipelines multiple logical signals called virtual wires across single inter-FPGA wires to overcome FPGA pin limitations [1, 2]. Logic designs are mapped to multi-FPGA systems through a series of compilation steps which include partitioning of logic into blocks small enough to fit within FPGAs, placement of logic blocks onto specific FPGAs, assignment of design memory to available emulation memory, and scheduling of logic evaluation, inter-FPGA communication and memory accesses. All logic, memory and communication activities 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 memory access steps 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, emulation memory accesses are based on the emulation clock.

The basic approach of scheduling asynchronous domain data transfer for logic emulation systems has previously been applied to latches and combinational logic in [9]. Due to the complexity of memories, this previous approach targeted to latches cannot be easily extended to cover multi-ported memories. In this work we consider scheduling for asynchronously controlled memory address, data, and control signals.

3. Multi Domain Read and Write Ports

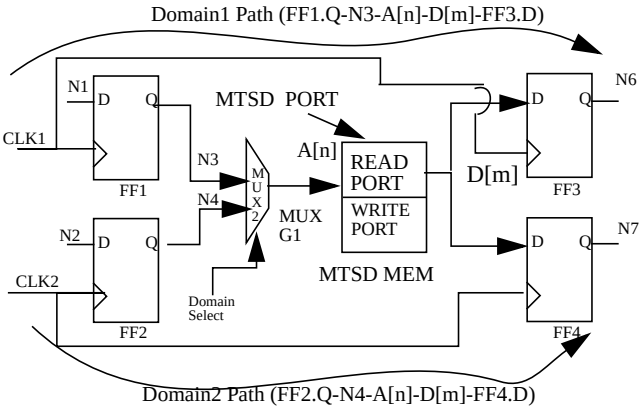


Figure 1: Multi Domain Read Port

A multi-domain memory port is a port that is accessed using signals that are sourced from more than one clock domain. In industrial designs such as graphics processors and network switches, memory ports are often accessed by several clock domains through the use of multiplexers which steer signals from one domain or another to the inputs of the memory port. At any particular instant in time, a port is accessed by only a single clock domain path, but over the course of time, it may be accessed by several different domains. As a result, a port must be considered to be potentially active for all domains at any point in time. More generally, any circuit in which signals are sourced by independently-clocked elements and fan out to memory structures results in multi-domain memory ports.

Consider the circuitry shown in Figure 1 where two asynchronous clocks CLK1 and CLK2 drive state elements (FF1,FF3) and (FF2, FF4) respectively. The portion of the circuit that reacts to clock CLK1 is logically grouped as Domain1 and similarly the portion of the circuit that reacts to CLK2 is grouped as Domain2. The Read port receives signals from both Domain1 registers and from Domain2 registers. The Read Port output is sampled by both Domain1 and Domain2 registers. At any point during design evaluation the value of the design signal Domain-Select determines which domain path accesses the memory. A memory with a Read port input that transitions (changes value) on multiple domains and a Read port output that is sampled (looked at) in multiple domains is called an MTSD (Multi Transition and multi Sample Domain) memory. Figure 2 shows a slightly more general example of multi-transition nets reaching both memory read and write ports. In the figure a multi-transition WriteEnable signal applied to a Write Port triggers memory to be written in one of two domains. As a result of this behavior the port can be classified as an MTSD Write port.A memory module which has at least one MTSD Read or Write port can be classified as an MTSD memory.

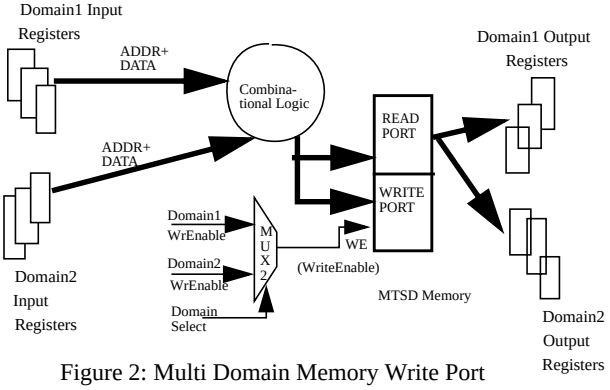


Figure 2: Multi Domain Memory Write Port

The memory access scheduling approach described in this paper can be applied to both single and multi-ported user design memories. To facilitate illustration of the algorithm, dual port memories are used as examples throughout the remainder of the paper. Note that simultaneous multi-domain read/write access to a single-port memory can be arbitrated by external logic in user designs or by synthesized memory access control logic. In either case, the logic sequences memory access to the single port to obtain correct logical memory behavior. Verification of simultaneous memory access on a single-ported memory is described in [7] and a detailed description is beyond the scope of this paper. By assuming a dual port memory implementation for illustrative purposes it is possible to explain the central ideas of the paper which are independent of the actual implementation.

4. Multi Domain Problems

There are a number of problems that make multi-domain memory access 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 period of the clock attached to the sequential elements.

Consider the circuit in Figure 1 where design clocks CLK1 and CLK2 are asynchronous, and the memory accesses in each domain can overlap with each other. The correct functional model of this circuit must simultaneously satisfy the timing closure axiom in each constituent domain. This indicates that all of the Read Address bits must travel from source registers to the Address terminals of the Read port, a Read Access must be performed, and the resulting Data output must reach the input of the output registers of the same domain in exactly one clock cycle, irrespective of combinational or routing delays. As an example, the contents of flip-flop FF1 must reach the input of the Read port, the read access must be performed, and the result of read access must reach the input of flip-flop FF3 in exactly one cycle of user clock CLK1. The same requirement holds on the corresponding Domain2 path.

Multi Domain Multi Port Access

Functional Axiom 2: ***Transparent Memory***

Should there be both Read and Write accesses to the same memory location in the same user design cycle, the Read output must reflect the most recent write. As a result, the Write access should be processed before the Read access.

In order for the Read port Data output to consistently reflect the state of a memory location, transparency between Read and Write accesses should be assured in the case that both accesses are made to the same memory location in the same user design cycle. This constraint requires that the Write access must occur before the Read access so that the Read port output can always reflect the result of the most recent Write access. This access sequence is easy to achieve when both Read and Write ports are in the same domain. However, the determination of the access sequence is more complicated in the presence of MTSD Read and Write ports since accesses in each domain are completely asynchronous to each other. Assuming that Read and Write addresses are the same in a user design cycle, a Transparent Write (e.g. write followed by read) in Domain1 of Figure 2 must be completed in exactly one cycle of CLK1 independently of CLK2. This transaction includes Write Port access, Read Port access, and address and data communication. The same holds true for a Transparent Write in Domain2 with clock CLK2 independent of CLK1.

Transporting Multi Domain Values

*Functional Axiom 3***: Causality**

The occurrence times of events in 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-domain 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) [2]. Since hard-wired signals cannot be multiplexed to carry non-MTSD nets, pin limitation problems [1] 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 2 is partitioned such that the multi-domain value WE (WriteEnable) 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 WE 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 3 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.

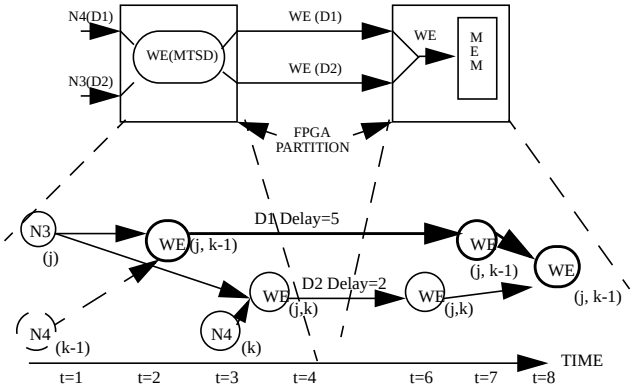


Figure 3: Transporting Multi Domain Values.

Hold Time Problem in Multiple Domains

The correct functioning of state elements and memories requires that data signals arrive at an element a certain period of time (setup time) before the triggering (enable/clock) signal and are held steady for a 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 *ActiveHigh* MTSD memory Write Port shown in Figure 2 where a combinational logic is sourcing its WriteEnable,

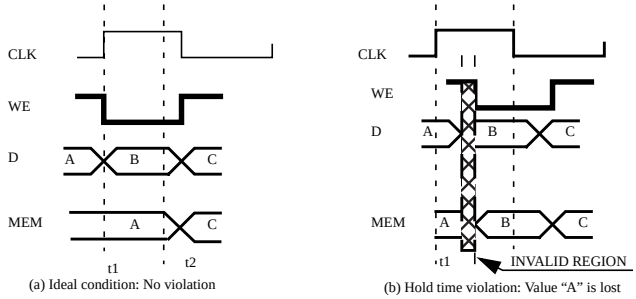


Figure 4: Hold Time Violation in MTSD Memories

Address and Data inputs. The waveforms for the same circuit are shown in Figure 4, where D, WE and MEM represent Data, WriteEnable and resulting memory contents for some Address. Figure 4(a) shows the ideal functional model execution where an edge on the user clock CLK at t=t1 causes WE and D to transition in 0-delay. The old value "A" is stored in the memory as a result. Figure 4(b) shows more realistic waveforms where routing and combinational delays cause the WE and D to arrive at the memory inputs at different points in time in response to CLK. A problem arises if the new D value ("B") reaches the memory sooner than the new WE. This causes memory to be evaluated with new D against old WE resulting the destruction of the old value "A". This can happen if the routing delay on the Clock/WE path is greater than the routing delay on the D due to combinational logic in those paths. In a case where both Address/WE and Data paths are in the same domain, it is easy for a scheduler to compute regions of time when WE is invalid and mask those regions so that the affected memory port is not evaluated. This solution fails if D and WE nets are multi-domain nets because regions of validity for memory port evaluation in one domain may conflict with regions of invalidity in other domains. The key challenge here is to satisfy hold time for every *(D, WE)* pair in each of the constituent domains.

5. Definitions

MTSD Net: A net which transitions and is sampled in more than one domain. In other words, a net is an MTSD net n if:

$$|Td(n) \cap Sd(n)| > 1$$

where Td(n) is the set of domains in which net n transitions and Sd(n) is the set of domains in which net n is sampled.

In Figure 1, net A[n] is a Multi Transition and Sample Domain (MTSD) net since there is a combinational path from A[n] to D[m] of the read port.

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

MTSD Read Port: A memory Read port whose address lines are driven by MTSD nets and whose data outputs are sampled in multiple domains. In Figure 1 MTSD address net A[n] for address bit n and MTSD data output net D[m] for data bit m classify the read port as an MTSD Read Port.

A Read Port is an MTSD Read Port if:

$$\left| \left\{ \bigcup_{i \in AddrBus} (Td[i]) \right\} \cap \left\{ \bigcup_{j \in DataBus} (Sd[j]) \right\} \right| > 1$$

where Td[i] is the set of transition domains of Address bit i and Sd[j] is the set of sample domains of Data bit j

The circuit in Figure 1 contains two same-domain paths, FF1.Q-N3-A[n]-D[m]-FF3.D in the domain of CLK1 and FF2.Q-N4-A[n]-D[m]-FF4.D in the domain of CLK2 and two multi (or cross) domain paths FF1.Q-N3-A[n]-D[m]-FF4.D and FF2.Q-N4-A[n]-D[m]-FF3.D.

MTSD Write Port: A memory Write port whose Write Enable or Address is driven by an MTSD net.

A Write Port is an MTSD write port if:

$$\left(\left| \left\{ \bigcup_{i \in AddrBus} (Td[i]) \right\} \right| > 1 \right) or (|Td(WE)| > 1)$$

Where Td(WE) is the set of transition domains of WriteEnable and Td[i] is the set of transition domains of Write Address bit i

MTSD Domain: A collection of connected gates, states, memories and nets that are MTSD.

MTSD Block: A partition of MTSD Domain that is small enough to fit into an FPGA. It is at this block boundary that all inter-FPGA scheduling takes place**.**

6. Approach

Observation 1:

For any constraint 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 shown in Figure 1, it is only necessary to satisfy the timing closure property for same-domain paths, such as FF1 to FF3, but not for cross domain paths, such as FF1 to FF4. Similarly, hold times must be satisfied for each same-domain *(D,WE)* pairs and transparent writes must be guaranteed between same-domain read-write ports. Essentially, the multi-domain problem reduces to simultaneously satisfying functional requirements within each of the constituent domains.

Multi-Domain Data Transport

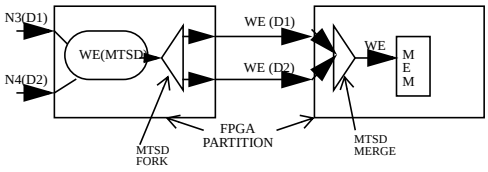


Figure 5: Multi-Domain Data Transport.

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 non-MTSD signals on FPGA boundaries as shown in Figure 5. 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 derived at the merge point is guaranteed to be causally correct.

Hold Time Constraints on MTSD Memories

Observation 2:

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

Observation 2 stems from the fact that when a level sensitive Write Port is evaluated with OLD Data against a NEW Address/WE, (a Setup time violation), new Data arrival results in re-evaluation of the memory port if the WE is open. If the port WE is closed, OLD Data does not corrupt the memory. Alternatively, if a memory port is evaluated with NEW Data against an OLD WE that is open (a Hold time violation) the correct memory value may be irretrievably lost since the OLD Data is no longer available. Notice that a similar observation can be made about the relationship between Data and Address buses i.e., evaluating an OLD address against a NEW Data will result in irrecoverable loss of contents at the OLD address.

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 memory with Data D(Ai, Bk) and Address/WriteEnable AW(Aj, Bk) on some clock edge k in Domain B, there are 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 memory write port with D(Ai, Bk) against AW(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 domain A. This relationship can be extended to an arbitrary number of domains. The implication of this is that every domain edge results in an evaluation which satisfies both setup and hold time with respect to that domain and each evaluation not satisfying setup against a domain is subsequently followed by a correcting evaluation against that domain.

Our new static scheduling algorithm also ensures that the Write Data never arrives before Address/WriteEnable for any edge in any domain. Additionally, the algorithm ensures that both Data and Address/WriteEnable arrive prior to subsequent domain edges so that all instantaneous setup violations are corrected before any sampling occurs.

Splitting of MTSD Memory Ports

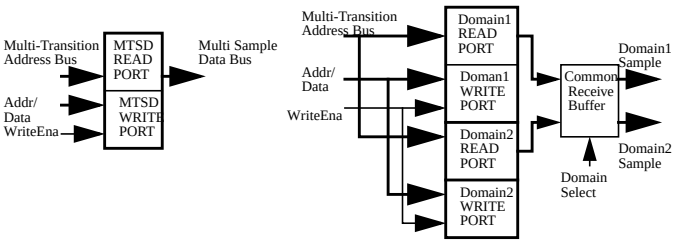


Figure 6: Splitting of MTSD Ports in Single Domain Ports

The goal of providing transparent memory can be addressed by conceptually splitting Read/Write MTSD ports into constituent domain ports. This approach replaces MTSD ports with a set of constituent single-domain ports as shown in Figure 6. By splitting the ports into multiple single domain ports, it is possible to track same-domain dependencies between Write and Read ports and schedule port usage for transparent memory accesses in each domain. As shown in Figure 7, the individual ports are placed along with an MTSD block inside an FPGA. This FPGA interacts with surrounding single-domain FPGAs which drive and sample data. As a result of port splitting, it is possible to ensure that all same-domain requirements are met and all multi-domain hold time violations are avoided. A common receive buffer is used to ensure a consistent image of Read data that can be sampled in both domains. It is important to note that port-splitting only increases the number of accesses to memory and does not increase either the capacity requirements or the total number of ports of the underlying physical implementation in the emulation system. As a result, cost is only measured in terms of performance and not in terms of capacity. Since emulation system memory chips which model user design memories are typically much faster than the FPGAs which model design logic, additional memory accesses generally do not negatively impact the performance of the verification system.

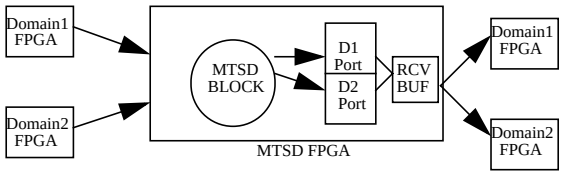


Figure 7: Dependency flow through the split ports

Transforming MTSD Edge Sensitive Memories

MTSD Edge sensitive write ports are not covered by Observation 2 and are more difficult to address than MTSD level sensitive ports. Our approach to handle edge sensitive memories is to transform them into master-slave level sensitive memories, just as an edge-triggered flip-flop can be converted into a master-slave latch pair. The derived level-sensitive memories are then subject to the same processing as other MTSD level sensitive memories.

7. Static Scheduling

We have used a modified TIERS scheduling algorithm to route communication paths between blocks and memory ports [12]. 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 the basic steps involved in static routing are described.

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 [3].
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)

8. MTSD Memory Scheduling

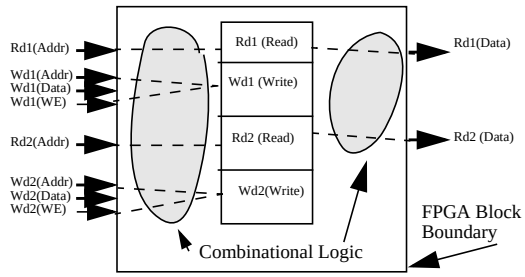
In this section MTSD path scheduling is described. Scheduling is performed so that hold time requirements are satisfied on every MTSD memory in each of the constituent domains.

Dependency and Depth of MTSD Logic

A key issue in static scheduling is to create a causally correct ordering of route links. When scheduled, this 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 an 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.

Figure 8 shows how the dependencies flow across memory ports and combinational blocks. The MTSD paths are split at the block terminals so that only single domain route-links are needed for inter-FPGA communication. To aid scheduling, two types of dependency classes are computed, *Same-Domain Dependency* which tracks link dependencies within a single domain and *MtsdDependency* which tracks link dependencies within all domains including cross domain paths. Using this dependency information two types of depths are computed for each route-link: normal *Depth* which takes into account only same-domain dependencies and *MtsdDepth* which is normalized Depth across all domain dependencies.

MtsdDepth is used to sort all route links in all domains, including those links targeted to memory ports, and to produce a partial order that is consistent in each of the domains. In addition, *MinDelay(i, p)* and *MaxDelay(i, p)* are computed for each block input terminal i to memory port p as there can be multiple combinational paths from a block input to a port.



Pd(T) is used to indicate a set of route-links combinationally reaching from Block terminals to the port terminals, where
P is the type of port: “R” for read “W” for write
d is the domain of the port
T is the functional terminal (address/data/WE) of the port

Figure 8: MTSD Memory Dependency Computation

Dependency Computation of Memory Ports

As noted earlier, to satisfy the hold time of an MTSD Write port, memory access must be scheduled such that the WriteEnable and Address signals arrive at the memory at or before the time the Data arrives. This imposes an additional ordering requirement on route-links. The following describes an approach to compute the evaluation order of route-links and MTSD memory ports. To aid in memory ordering, each MTSD partition is analyzed and block terminal sets are created for each MTSD write Port. These sets are shown in Figure 9 are:

- **D-INPUT** Set: Group of all MTSD Block terminals that combinationally reach the Data terminals of the MTSD write port. This also includes any input that reaches both Data and Address/WriteEnable.
- **AW-INPUT** Set: Group of all inputs which reach Address or WriteEnable of the Write port from all domains.
- **RD-OUTPUT** Set: Group of all block terminals which are output Data terminals of dependent Read Ports.

These dependencies are used to order memory route-links with other route-links. Note that input nets can fan out to more than one domain port. The diagram shows that terminals are split at the Block boundary for the purpose of scheduling. As a result, MtsdDependency can be determined for both same domain and cross domain terminals.

Depth Computation of MTSD Ports

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

- Dependency introduced between terminals in the D-INPUT set and terminals in the AW-INPUT set.
- Dependency introduced between Write Ports and Read Ports (including cross domain Read Ports).

As described in Section 6, each MTSD port can be logically split into single domain ports. To allow for transparent write, dependency must be created between Read and Write ports. MtsdDepths are used to sort all the route links to arrive at a partial order that satisfies same domain dependencies while maintaining cross domain port relationships.

Write Port Evaluation Algorithm

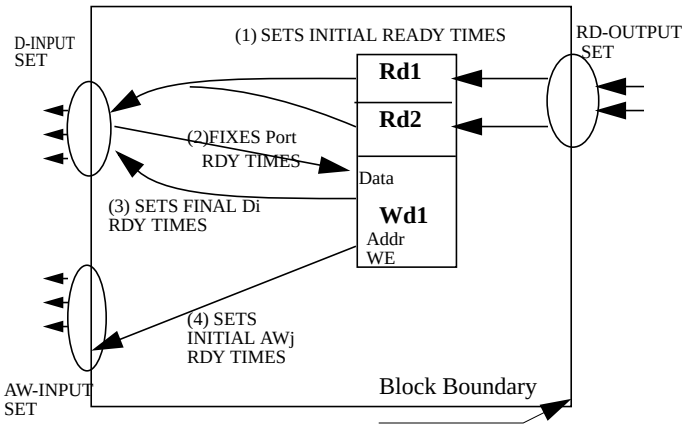


Figure 9: MTSD Write Port Evaluation.

Figure 9 illustrates the basic steps involved in MTSD Write port scheduling. Due to the port order described earlier, by the time Write port is evaluated, the *DepartureTimes* of all the terminals in its *R-OUTPUT* set are known. As a result Read Ports Rd1 and Rd2 are ready to be scheduled in their respective domains.

For each Read Port Ri,

$$ReadyTime(Ri) = \left(\begin{matrix} MAX(DepartureTime[j]) \\ j \in Data(Ri) \end{matrix}\right) - ReadAccess$$

Note that in Figure 9 arrows indicate 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.

The following algorithm computes the final ready time on *D-INPUT* terminals and the lower bound for the ready times on *AW-INPUT* terminals.

For each Write Port W,

1. Compute the initial ReadyTimes for each Data input based on the ReadyTimes of their dependent Read ports. Note that these are not final ReadyTimes because they do not take into account the Write port's ReadyTime.

For each Di in D-INPUT(W)

$$ReadyTime(Di) = \begin{matrix} MAX(ReadyTime(j)) \\ j \in dependReadPort(W) \end{matrix} + WriteAccess$$

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

For each Di in D-INPUT(W),
ReadyTime(W) =
MAX(ReadyTime(Di) + MinDelay(Di to W))

3. For each Di in D-INPUT(W),

3.1. Compute the *RequiredReadyTime*. The value is called *required* ReadyTime because, if data arrives sooner than this time, there is a risk of violating the (D,AW) constraint.

RequiredReadyTime(Di) =

$$MAX(ReadyTime(Di), (ReadyTime(W) - MinDelay(Di\ to\ W)))$$

3.2. Compute the final ReadyTime. This is determined by the Routing algorithm described in [12] based on the available routing resources. This algorithm uses a modified Dijkstra's algorithm [3] to find the shortest path from the FPGA sourcing the route-link to the FPGA where this memory is located. The algorithm then reserves resources along this shortest path such that the communication is completed and value is ready at Di by the *RequiredReadyTime*.

3.3. If the final *ReadyTime(Di)* is greater than *RequiredReadyTime*, add delay compensation in the Di to W path to ensure that Data does not arrive at the Write Port sooner than required.

DelayCompensation(Di, W) =

ReadyTime(Di) - RequiredReadyTime(Di)

A delay equal to *DelayCompensation(Di,W)* is injected into the path from Di to Write Port W by adding a chain of Virtual Clock triggered flip-flops.

4. Propagate *ReadyTime(W)* to each of the terminals in AW-INPUT(W) as initial *ReadyTime(AWi)*. This is initial ReadyTime because there could be other dependent children on AWi which can further alter the ReadyTime.

For each AWi in AW-INPUT(W),

ReadyTime(AWi) =

$$MAX(ReadyTime(AWi), (ReadyTime(W) - MaxDelay(AWi\ to\ W)))$$

The above algorithm guarantees that the ReadyTime(AWi) is always less than or equal to ReadyTime(Dj) for any (AWi, Dj) pair. This ensures that the Address/WriteEnable value always arrives before the Data value on any MTSD Write port. Notice that in the above equations, MinDelay has been used for Data terminal to port delay calculations but MaxDelay has been used for Address/WriteEnable terminal to port delay calculations. This is to ensure that the delay from any AWi to a port does not exceed the delay from Di to the port after compensation (performed in step 3.3). Without this compensation it is still possible to violate hold time requirements at the MTSD port even if (D,AW) constraints at the block boundary are met.

9. Experimental Results

We have implemented the algorithms described in this paper and integrated them into the Ikos VirtuaLogic compiler [7] for the VStation-5M Emulator. Two industrial designs (a telecom design and a graphics processor) containing asynchronous domains have been compiled. Table 1 compares the results of scheduled MTSD Virtual routing to dedicated hard-wire routing in which MTSD memory control signals are transported between FPGAs using dedicated rather than time-scheduled wires. Design1 has a smaller percentage of MTSD logic when compared to Design2 and also has a smaller percentage of memory modules as shown in Table 2. 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. Maxi-

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. Estimated Max Speed MTSD HardRouted	346 KHz	129 KHz
11. Estimated Max Speed MTSD VirtualRouted	369 KHz	157 KHz

Table 1: MTSD Virtual Routing vs. Hard Routing

Testcase	Design1	Design2
1. Num. Memories	100	37
2. Mtsd Memories	23	24
3. Num. Read Ports	139	37
4. Num. Write Ports	119	37
5. Memory addresses	6364	5952
6. Total Data bytes	28292	7808

Table 2: Memory Statistics

mum emulation clock speeds in rows 10 and 11 are estimated based on a 34 MHz Virtual Clock on a VStation-5M Emulator.

10. Analysis of Results

Notice that in Table 1, row 9, the number of Virtual Clocks in the critical path for Design2 is much higher than the number for Design1. This is because experiments for Design2 were dominated by memory transactions. It can be seen from rows 8 and 9 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.

The number of scheduled memory accesses for any MTSD memory is given by:

$$TotalMemoryAccesses = \frac{nE \times nD \times (nR + nW)}{nP}$$

where
nE = number of phases of the user design clock
nD = number of asynchronous domains
nR = number of Read Ports
nW = number of Write Ports
nP = number of ports in the physical SRAM modeling Memory

One may observe that we are making a conservative estimate of the memory accesses: one access per port per every edge of the user design clock. This is necessary for multi-domain designs since one cannot predict how accesses across domains interleave. If special knowledge about the user design is known at compile-time the scheduling algorithm could be tuned to improve scheduling results. For example, if it is known that one domain always performs memory writes and another domain always performs memory reads the algorithm could be tuned to only schedule accesses in distinct domains and to ignore cross domain dependencies.

There is one known limitation in the algorithm used to compute port routing order: it cannot handle cyclic dependencies between Read and Write Port nets that cross FPGA partition boundaries. An example is a Read port output net feeding back to serve as a Write port data net. Currently, we avoid this scenario by using a pre-partitioning step which identifies and bundles combinational paths from a Read port to a Write port together in the same MTSD block. This ensures that the ports are placed in the same partition that includes the physical interface to memory.

11. Conclusions

In this paper we have described a new parallel verification approach for dealing with memory accesses in designs that contain multiple asynchronous clock domains. This scalable approach allows for the correct evaluation of an unlimited number of access paths for a single memory, even if they occur in disjoint clock domains. The developed scheduling algorithm statically determines multi-domain memory port accesses for parallel verification equipment so that setup and hold time violations are avoided. The approach has been integrated into a commercial verification software package and demonstrated on a VirtuaLogic emulation system for two large commercial benchmark designs. Experiment results show that the approach is scalable and provides good modeling fidelity. As a result of this scalability, an improvement in overall system performance has also been obtained.

We plan to extend this approach to deal with visibility and debug aspects of the design under test and hard-wired cores. The heterogeneous nature of the MTSD signals presents special challenges for scheduling and interfacing with the signal capture tools.

12. References

[1] J. Babb, R. Tessier, and A. Agarwal, Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators, in the Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, Napa, California, April 1993.

[2] J. Babb, R. Tessier, M. Dahl, S. Hanano, D. Hoki, and A. Agarwal, Logic Emulation with Virtual Wires, in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, June 1997.

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

[4] M. Dahl, J. Babb, R. Tessier, S. Hanono, D. Hoki, and A. Agarwal, "Emulation of a Sparc Microprocessor with the MIT Virtual Wires Emulation System", in the Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, Napa, California, April 1994.

[5] J. Gallagher, "Prototypes Ensure Pre-Verification", EE Times, June 13, 2000

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

[7] Ikos Systems, Inc., VirtuaLogic Datasheet, http://www.ikos.com/products/vsli/index.html, 2001.

[8] D. Kolson, A. Nicolau, N. Dutt, "Elimination of redundant memory traffic in high-level synthesis", IEEE Trans. on Comp.-aided Design, Vol.15, No.11, pp.1354-1363, Nov. 1996.

[9] M. Kudlugi, C. Selvidge, R. Tessier, "Static Scheduling of Multiple Asynchronous Domains For Functional Verification", In the Proceedings of 38th DAC, pp 647-652, Las Vegas, June 18-22, 2001.

[10] Quickturn Design Systems, Cobalt Datasheet, http://www.quickturn.com/products/cobalt.htm, 2001

[11] H. Schmit and D. Thomas, "Address Generation for Memories Containing Multiple Arrays," IEEE Transactions on CAD, vol.17, no.5, p. 377-385, May, 1998.

[12] C. Selvidge, A. Agarwal, M. Dahl, J. Babb,. "TIERS: Topology Independent Pipelined Routing and Scheduling for VirtualWire Compilation". In Proceedings of FPGA'95, pages 25-31, Berkeley, CA, Feb 1995.

[13] Tharas Systems, Hammer Datasheet http://www.tharas.com/images/Datasheet.pdf, 2001