Chapter 18
Emulation Using FPGAs

Paresh K. Joshi

18.1 Introduction to Emulation

For the purpose of this chapter, we will use emulation to include prototyping also—since underlying challenges and methodologies are common. We read about simulators in Chap. 11. An emulator is a *simulation-specific* hardware, which is capable of retaining the parallelism of the blocks of the design, thereby significantly improving the speed of execution.

Depending on the capabilities of the emulator, you can get very close to your design environment. Since emulators are dedicated hardware, the speed advantage is obtained at the cost of observability and controllability. Emulation also needs additional setup, which is what this chapter is mostly about. In an ideal scenario, the emulator must support all the features of simulation at a speed and cost advantage.

18.1.1 Types of Emulators

1. *Array of simulation-specific processors* (*Cadence Palladium series*): Array of processors whose instruction set and software is tailored to simulation tasks. One set of such arrays is called a *board*. Each processor on the board can simulate millions of gates in parallel. Furthermore, each processor on the board talks to other processors via a fixed (specific) protocol.
2. *Array of FPGAs (Synopsys ZeBu series):* Array of FPGAs. Each FPGA can have mapped gates programmed into it. Each FPGA in the Array usually has dedicated wiring with other FPGAs.

P.K. Joshi (✉)
Intel Mobile Communications, Bangalore, Karnataka, India
e-mail: paresh.k.joshi@intel.com

S. Churiwala (ed.), *Designing with Xilinx® FPGAs*,
DOI 10.1007/978-3-319-42438-5_18

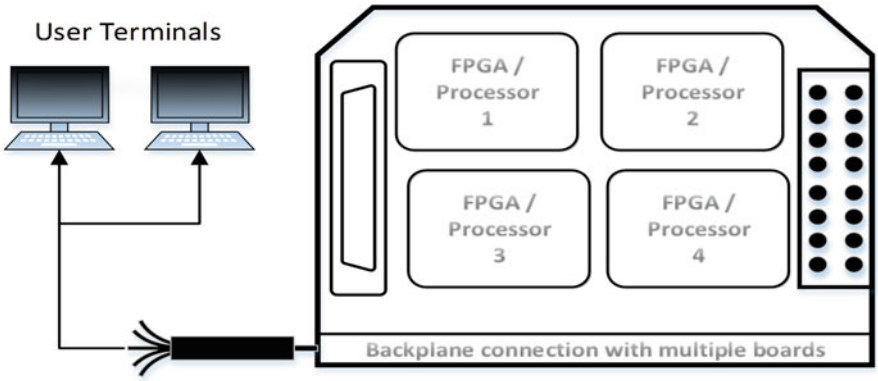

Fig. 18.1 Cascading 4 processors/FPGAs to build a larger emulation system

3. *A hybrid array of both simulation-specific processors and FPGAs (Mentor Veloce series).*

For large designs boards in an emulator can be cascaded. To better utilize the components in the emulator, there are partitions possible which enable multiple users to simultaneously access the resources of the emulator.

Since emulators comprise of hardware components, it is possible to connect the emulator to real external targets like JTAG, UART, QSPI, I2C, etc. The JTAG and UART are used by the software team to do hardware-software co-design and debug at the *programmers view* level.

Figure 18.1 illustrates an FPGA or processor array-based emulator system with multiple user terminals, standard connectors, IOs, and a backplane to cascade multiple such boards. Multiple users can then use the emulator boards for improving resource utilization.

18.1.2 Uses of Emulation/Prototyping

Substitute for simulation: This is the most obvious usage. In practice, however, emulation is resorted to only after the RTL design reaches a certain level of maturity. A not-so-mature RTL design will find iterative debug to be difficult, due to limited observability and controllability of emulation.

Enabling pre-silicon software development: Once the RTL is reasonably mature, the software teams can use the emulator for developing BOOTROM, Software (UBOOT, Linux, Android, RTOS, UEFI), Device Drivers (BSP), etc. Doing so provides several months of lead time to the software teams. This enables the software components to be available and ready for use, immediately after the device silicon is available.

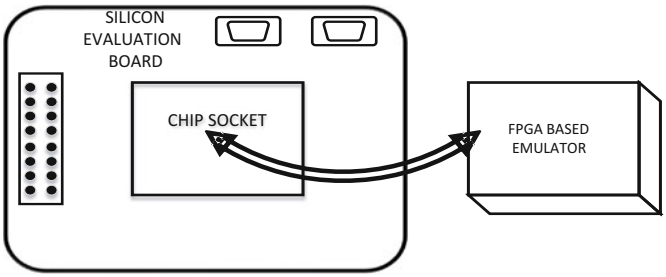

Fig. 18.2 Silicon Evaluation Board with a socket being interposed with FPGA-based emulator

Place-holder for actual silicon: The first silicon bring-up team designs an evaluation board with sockets for the device. Before the actual silicon is available, the emulator can behave as a prototype and fit into the socket using a plug-in board. The evaluation board along with silicon bring-up test cases can be run on the system as shown in Fig. 18.2.

18.2 Emulation Using FPGAs

System designers and prototyping teams have been using FPGAs to their benefit. FPGA tools are available to provide RTL to FPGA mapping. If you have a prototyping environment, the additional activities for going to emulation include:

1. Creation of a synthesizable and reconfigurable testbench.
2. Addition of instrumentation into design for advance debug purposes.
3. Mapping of complex design blocks like IOs, SERDES, DSP blocks, and block RAMs to the FPGA.
4. Remapping of complex clocking structure of the device to the FPGA-based PLLs and clock controllers.
5. Mapping of design IOs to the FPGA IOs to obtain connectivity to the external targets (JTAG, UART, etc.).
6. For designs which require multiple FPGAs:
   (a) *Logic Partitioning*: Partitioning of the design into chunks of logic to fit into individual FPGAs. This depends on the size of the design and the size of placeable gates on the FPGA. The logic and memory closely associated with the said logic are grouped together into pieces which fit on the same FPGA.
   (b) *Pin Partitioning*: Partitioning of the design with appropriate pin count across FPGAs. This depends on the hardware board design and is usually fixed for a particular board.

The additional activities for going to emulation from a simulation setup include:

1. *Observability:* Simulation allows to see the waveforms for all signals at all times. The waveforms are directly dumped into a hard disk during runtime. In an FPGA, there are limited logic and memory resources. So complete runtime waveform dumping is not possible. Thus, you have to add instrumentation to *trigger* the start of waveform dumping for a known limited number of signals and for a known limited amount of time. Furthermore, you need to build in a mechanism to retrieve the waveform data from the FPGA block RAMs. Xilinx Vivado provides ILA core for doing this—as explained in Chap. 17.
2. *Controllability*: For some tests, a specific pin (say: *reset*) may need to be kept at a desired value for a specific duration. In simulation you can force the signal then release it. A similar ability needs to be provided when doing emulation using FPGAs. Xilinx Vivado provides VIO.
3. *Memory initialization*: The DUV usually contains BOOTROM which needs to be programmed (preloaded) with the appropriate bitmapped code. The testbench could have other memory models of flash, DDR, etc. In the simulation environment, the memory load (*$readmemb*) and dump can be used. A similar ability is required for emulation using FPGAs.

Xilinx FPGAs and the Vivado tool set provide the methods and means to make all of the above possible.

18.3 Challenges in Emulation Using FPGAs

The basic challenge is to stitch the hardware, the tool software, and the RTL-mapping flow with the evaluation board and components. This section breaks up the challenge into multiple parts and sections. Section 18.4 then explains on how to deal with these challenges.

18.3.1 Design Logic and Memory Size

The engineering choice is to use one FPGA which fits the design. However, sometimes the DUV may be bigger than the largest FPGA available. Even otherwise, sometimes fitting the DUV into two smaller FPGAs is cheaper than using the largest FPGA available. If the design is skewed toward huge memory blocks, the FPGA tools can map parts of unmapped logic on the FPGA tile for memory blocks. For an emulator using FPGAs, (since the testbench is embedded into the FPGA) large memories like flash, DDR pose mapping problems. In such scenarios the emulator is fitted with large external memories which are then remodeled to behave like flash and DDR. Note that this remodeling is done through custom instrumentation insertion prior to using Vivado P&R tools.

18.3.2 Design Pin Count

The FPGA (or an array of FPGAs) must be able to support the relevant pin count of the device being emulated. In general, for emulation purposes a synthesizable testbench is used, indicating that there are fewer external connections. In certain cases, flash memory can be real components on the board which are then pinned-out to the board.

18.3.3 Clocking

Clocking between FPGAs and ASIC/ASSP is different. In an ASIC/ASSP there could be many hundreds of clock domains with multiple PLLs embedded. Each root clock derived from a PLL can have multiple secondary clock generation logic (say for dividing clocks, test clocking). Furthermore, sets of flip-flops or registers in the design can have clock-gating circuit implemented as part of power-reduction techniques.

FPGAs usually have a limited number of PLLs and a limited number of balanced clock channels incident upon a larger cluster of flip-flops. The challenge is to straighten up the ASIC clocks to map it easily onto the FPGA clocking.

18.3.4 RTL Constructs and Remodeling

Several RTL constructs are not FPGA friendly. These need to be modeled appropriately for FPGA. The remodeling has to be done without modifying the functionality. A module RTL makes it easier and scalable since there is a great usage of common cells in the design.

18.3.4.1 IO Pads Modeling

IO pads typically have tristate functionality. Usually, these IOs of the DUV are connected to the BFMs in the testbench. Recent FPGAs do not have built-in tristate gates. For FPGA usage, you need to remodel the tristates as shown by the example in Fig. 18.3. The Xilinx ISE/Vivado toolset automatically transforms internal tristates into logic elements.

18.3.4.2 ADC Module Modeling

For a module with analog behavior (e.g., ADC/DAC), you need to appropriately model to ensure that its boundary talking to the digital side of the design is clean. For example, an ADC module can easily be modeled with a memory and digital

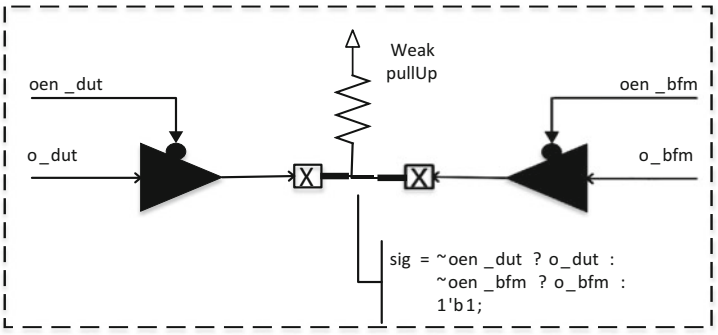

Fig. 18.3 Remodeling of typical IO connectivity within testbench between DUV and BFM

output. The memory can be preloaded with the kind of analog behavior we expect out of the design. Alternatively, an ADC can be placed on the FPGA board and the digital output can be used as an input to the design. If the ADC module is deeply embedded into the DUV, you need to bring out the wires from the embedded hierarchies onto the top level of the testbench.

For Xilinx FPGAs you can use the SYSMON module (explained in Chap. 16). However, you still need to take care of:

- Performance of the SYSMON for emulator clocking
- The analog stimulus to be fed to the SYSMON
- The appropriate remodeling of the ADC to instantiate the SYSMON into it

18.3.4.3 Memory Modeling

Typically the RTL has memories which are either ASIC technology memories or modeled as a memory array. Also, the RTL memory model could have test logic embedded into it. Remodeling memories for FPGA is typically a four-step process.

1. Identify the memories in the design. If the memories belong to the same technology node, then the entity is usually identical except for the *address* and *data* width. Sometimes, there might be variants (e.g., byte-wise write).
2. Remodel the memory component with an equivalent FPGA friendly construct. If you are not interested in test logic, they could be tied to their disabled state. This remodeled memory component is then verified to be true using simulation. If the memory needs to have user-defined preloading or dynamic preloading, then explicit instrumentation needs to be added.
3. One level of FPGA synthesis and run is carried out to flush out the flow.
4. Create a scriptware to convert all the flavors of *data* and *address* widths.

Steps (2), (3), and (4) are true for all types of remodeling done at RTL level, but it deserves a special mention for memories since there are many types.

18.3.4.4 Standard Cells Modeling

It is best to have synthesizable view of the technology standard cells in the design. Most technology libraries provide the synthesizable view of standard cells.

18.3.4.5 Inferred Components Modeling

Some RTL descriptions infer multipliers, dividers, special Register Files, FIFOs, etc., during the ASIC synthesis flows. These components use compiled models/descriptions for simulation. Such components will end up as being unresolved. A way to resolve this problem is to actually do an ASIC synthesis and use the verilog equivalent for the said component. Thus:

FPGA RTL view = synthesized netlist from ASIC tool + the synthesizable RTL view of technology std-cell

18.3.5 FPGA Board Design

The FPGA-based emulation system is very much dependent on the FPGA board design. In particular, the number of FPGAs in the array, the capacity of each FPGA in the array, the external memory connected (for modeling large memories, for dynamic waveform dumping, and for using memory as Look Up Table for large pieces of logic with huge fan-in cones), and the external connectors, switches, GPIOs, and LEDs are provided. Its levels of complexity are higher to move from one FPGA-based emulator to another than it is to move across simulators from different vendors. The basic complexity is due to the use of hardware for emulation and so it is fixed. This complexity makes it difficult to make sound design and financial decisions for the right choice of FPGA-based emulators. FPGA vendors provide a chart with logic gate count estimates, IOs, memory blocks, SERDES blocks, and DSP blocks within the FPGA.

18.4 General Methodology

In this section we provide some known recipes to the challenges explained in Sect. 18.3. The recipes below would help design teams to realize their own FPGA-based emulator. We have assumed (by this chapter, toward the end of the book) a basic understanding of FPGA-based design.

Note that you should perform RTL to RTL Logic Equivalence Check after any RTL transformation.

18.4.1 RTL-Related Transformations

PLLs: All technology ASIC libraries contain PLLs. Each PLL consists of basic *reference clock in*, *clock out*, with pins indicating the multiplier factor in terms of *Numerator* and *Denominator* values. These have to be mapped to the equivalent PLLs in the selected FPGA. The methodology used is to keep the ASIC PLL entity identical but to instantiate the FPGA clocking resource in place. If the PLL has multiple clock outputs, the same are also remapped to the FPGA.

Clock Dividers: If there are dividers in the design, then it is appropriate to remove the divider circuits and replace them with the FPGA clock resource outputs as defined in the MMCM clock tile.

It would be useful to maintain a table similar to Table 18.1.

In the Table 18.1, for (#2) and (#3), the clock frequencies are the same, i.e., 20 MHz. It would be worthwhile to investigate from an ASIC clocking point of view, if it is possible to use the same PLL output of 20 MHz driving the clock end points of both (#2) and (#3). If the clocks are of the same frequency, but asynchronous to each other, it would be OK to reduce the use of a PLL and free up routing resources and reduce complexity of mapping to the FPGA.

Programmable Clock Dividers: Usually there is a use of Programmable Clock Dividers to select a baud rate as it is in the case of UART. In such cases, reconfigurable registers of the ASIC need to be remapped to the Dynamic Reconfiguration Data Input of the Clocking tile. Most emulation designers would put the dynamic reconfiguration data input as part of the instrumentation in the testbench, so that they have better control over the clock.

Clock Gating Cells: Integrated clock gating cells are instantiated by the RTL designer to enable dynamic power reduction. This can be a problem with FPGAs which can get resource limited if there are too many clock gating cells in the design. A solution is to do a tool-based or hand-scripted transformation to the clock gating cells. A typical example is provided in Fig. 18.4.

Table 18.1 Mapping of ASIC clock frequencies to FPGA clocks

#	ASIC clock	ASIC freq	FPGA clock resource	FPGA freq	Comments
1	Clock.A	400 MHz	PLL1.CLKOUT0	40 MHz	All clock scaled as div by 10
2	Clock.B	200 MHz	PLL2.CLKOUT1	20 MHz	
3	Clock.A.div2	200 MHz	PLL1.CLKOUT1	20 MHz	A divider in the path of ClockA is remapped to a clock output synchronous to div2 of the PLL1.CLKOUT0
4	Clock.A.div8	50 MHz	PLL1.CLKOUT2	5 MHz	Div8 of the PLL1. CLKOUT0

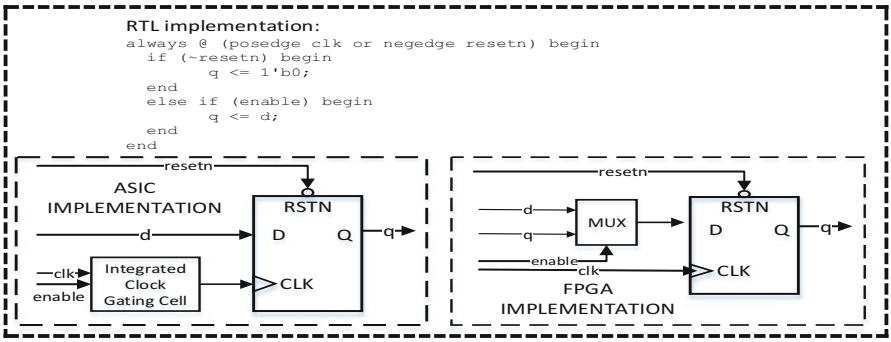

Fig. 18.4 Typical ASIC and FPGA implementation for a clock gating cell

18.4.2 Multiple FPGA Specific (The Partitioning Problem)

Now that the individual pieces of your RTL have been readied for FPGA-based emulation, the next level of complexity comes if the design cannot be mapped on one FPGA. For a particular design, it might not fit into a single FPGA, due to either of the following:

- Design logic size exceeding the logic that can be mapped onto the FPGA.
- Design logic could be mapped, but it could not be routed.
- Design logic was mapped and routed, but design has more memory than the block RAMs on the FPGA.
- Design ran out of IO that could be appropriately mapped on the FPGA.

Irrespective of the situation leading to the use of multiple FPGAs, all of the above need to be resolved on a per FPGA basis on a MultiFPGA emulation system. To start with, get a gate, memory, and pin count estimate for the big blocks in the design. Also, assume that each FPGA may be about 60 % utilized to begin with. Typically, most big IPs would fall within 5 ~ 6 sub-hierarchical levels of logic. This exercise would give a rough estimate of the number of FPGAs required to fit the design and testbench.

The exercise is iterative. Start with partitioning through the most constrained of the three resources (gate count, pin count, memory) and then affect the grouping changes to see if the other constraints can also fit. Figure 18.5 depicts the hierarchical view of the DUV and the testbench BFM components and the Table 18.2 the tabular view of the same. Both these views (hierarchical and tabular) help in converging to the right partitioning between multiple FPGAs.

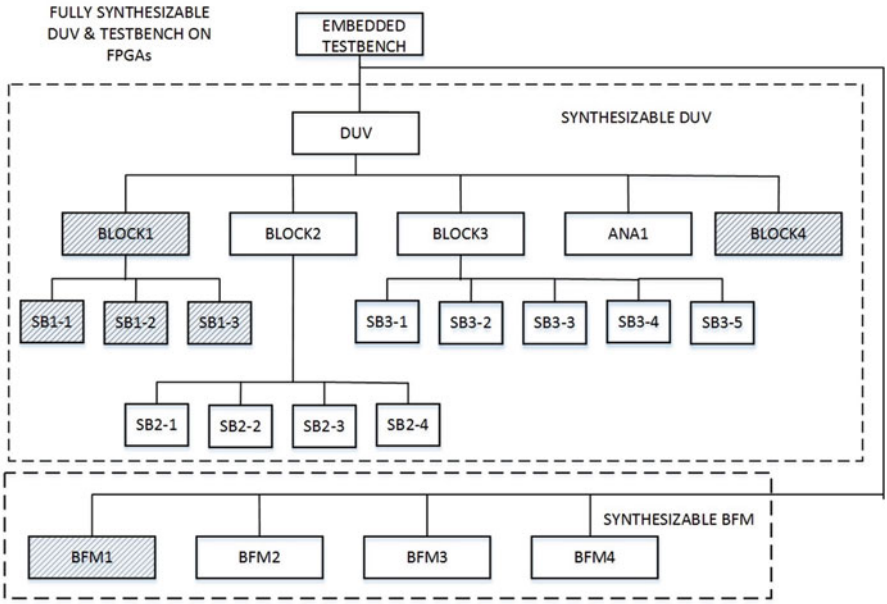

Fig. 18.5 Hierarchical view for embedded synthesizable testbench with DUV and BFM

Table 18.2 FPGA view for the embedded synthesizable testbench with DUV and BFM

subHier Level	ModName	GateCount	PinCount	TotalMemory	Estimate FPGA
1	tb_top	250	200	4 Mbits	
2	tb_top.BFM1	12M	100	200 Kbits	FPGA1
2	tb_top.BFM2	24M	50	100 Kbits	FPGA2
2	tb_top.BFM3	14M	125	250 Kbits	FPGA3
2	tb_top.DUV	200M	350	3.5 Mbits	
3	tb_top.DUV.BLOCK1	75M	450		FPGA1
3	tb_top.DUV.BLOCK2	80M			FPGA2
3	tb_top.DUV.BLOCK3	35M			FPGA3
3	tb_top.DUV.BLOCK4	5M			FPGA1
3	tb_top.DUV.ANA1	5M			FPGA3

18.4.2.1 Partitioning Gate Count Challenge

Once the gross level partitioning is known through analytical method as per Table 18.2, we need to get the same implemented. There are tools which can read in the RTL files and then dump out a regrouped file. Such grouping would result in new hierarchical tables being generated, as shown in Table 18.3.

For this example, considering per FPGA gate count of ~100M gates, Table 18.3 shows that FPGA3 is OK, but FPGA1 and FPGA2 are likely challenges to the P&R

Table 18.3 Sorted list of hierarchies on per FPGA basis

subHier Level	ModName	GateCount	PinCount	TotalMemory	Estimate FPGA
1	tb_top	250M	200	4 Mbits	
2	FPGA1.BFM1	12M	100	200 Kbits	**FPGA1**
2	FPGA1.BLOCK1	75M			**FPGA1**
2	FPGA1.BLOCK4	5M			**FPGA1**
2	FPGA2.BFM2	24M	50	100 Kbits	FPGA2
2	FPGA2.BLOCK2	80M			FPGA2
2	FPGA3.BFM3	14M	125	250 Kbits	**FPGA3**
2	FPGA3.BLOCK3	35M			**FPGA3**
2	FPGA3.ANA1	5M			**FPGA3**

Table 18.4 Actual partitioned pin count vs. available connections between FPGAs

F1<-->F2	F1<-->F3	F1<-->F4	F2<-->F3	F2<-->F4	F3<-->F4
PF12	PF13	PF14	PF23	PF24	PF34
IPF12	IPF13	NA	IPF23	NA	NA

stage. These considerations and iterations go on until there is sufficient convergence. Table 18.3 is deficient in terms of pin count and memory as it is for illustration purpose only.

However, since the module BLOCK2 and BFM2 are closely knit with each other, there could be pin count challenge if some readjustments of modules of BLOCK2 are done onto FPGA3 which seems to be least constrained.

18.4.2.2 Partitioning Pin Count

The MultiFPGA board usually has fixed pin count which can be summarized in a template table as in Table 18.4.

In Table 18.4 PF12 are the physical IO pins that are available between FPGA1 and FPGA2 (F1<-->F2) on the FPGA board.

In Table 18.4 we have a Not Applicable (NA) if the particular FPGA is not used in the implementation. The implemented pin count across the FPGAs (IPF) should be less than the provisioned pin count across the FPGAs (PF). Thus, the pin count criteria can be converged when IPF12<PF12 and so on.

If the pin count criteria are not satisfied, you could resort to pin muxing for the IO. This means that another utility RTL needs to be added to send multiple bits of data over a single IO from one FPGA to another. This utility RTL is inserted prior to the pin-multiplexed IO. Figure 18.6 shows the circuit for the utility RTL on the FPGAs for pin multiplexing. There are three main operations done:

- Load: convert from parallel to serial.
- Shift: shift the serial data from FPGA2FPGA.

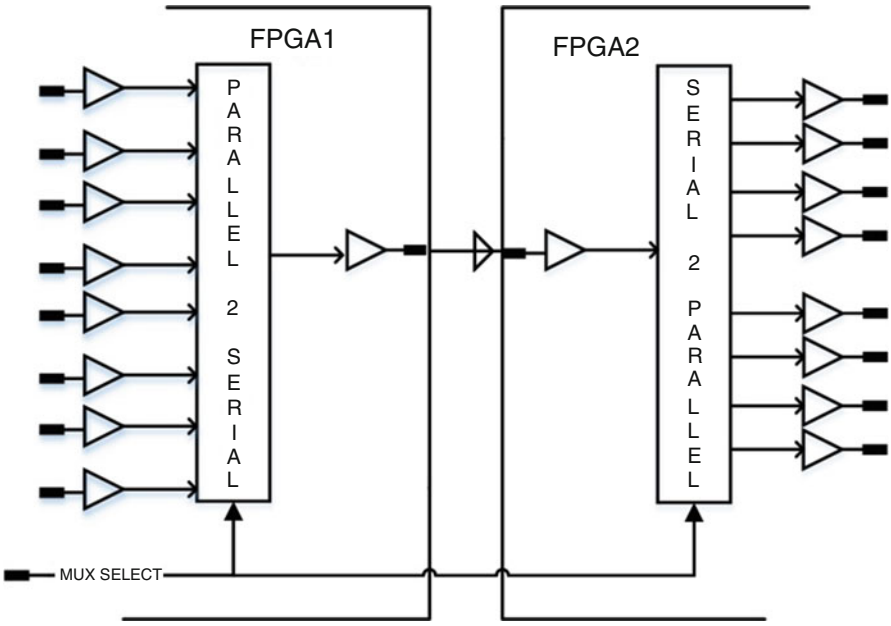

Fig. 18.6 Pin muxing for IOs over two FPGAs

- Restore: convert serial data back to parallel.

EDA Tools like Certify™ from Synopsys® form a major backbone to enablement of this convergence.

18.4.2.3 Using SERDES Lanes

It is also possible to use the FPGA SERDES Lanes as an extension to the pin multiplexing. SERDES provides a convenient *serializer* and *deserializer* over a two-wire network, which can transmit and receive data Gbps (Giga bits per second) range. The SERDES lanes are useful in converting FPGA2FPGA IOs into serial, sending it across at high speed and reconstructing the same at the other end.

18.4.2.4 Handling Clocks Over Multiple FPGAs

As soon as we move into using multiple FPGAs, the clocking complexity increases. One way is to see each hop or evaluation as a phase (a dedicated time slot) and increase the emulation clock period accordingly. This means that the performance of the emulator drops every time there is a signal hop.

18.5 Instrumenting

There are ways of achieving some degree of controllability and observability on an FPGA-based emulator, albeit at the cost of performance, logic area, and memory requirements. A general observation is that about 10 ~ 40 % (depending on design specifics) of the design overhead on an emulator is attributed to addition of instrumentation for controllability and observability. At each step of the instrumentation addition, exercise care to maintain the equivalence of the design.

Let us assume that the emulator adds an instrumentation port (say Instrumentation JTAG or iJTAG) through which it can carry out the functions of observability and controllability to the design. This instrumentation port provides an interface to the user using a host computer. Figure 18.7 logically explains the two ports needed for an emulator. Modern emulators like Synopsys ZeBu use the PCIe as an instrumentation port.

18.5.1 Ability to Stop and Start the Emulation

The emulator start-stop is affected by the clocking. If the clock to the logic blocks does not tick, the emulator is in *stop* state. The instrumentation needed to achieve the purpose are:

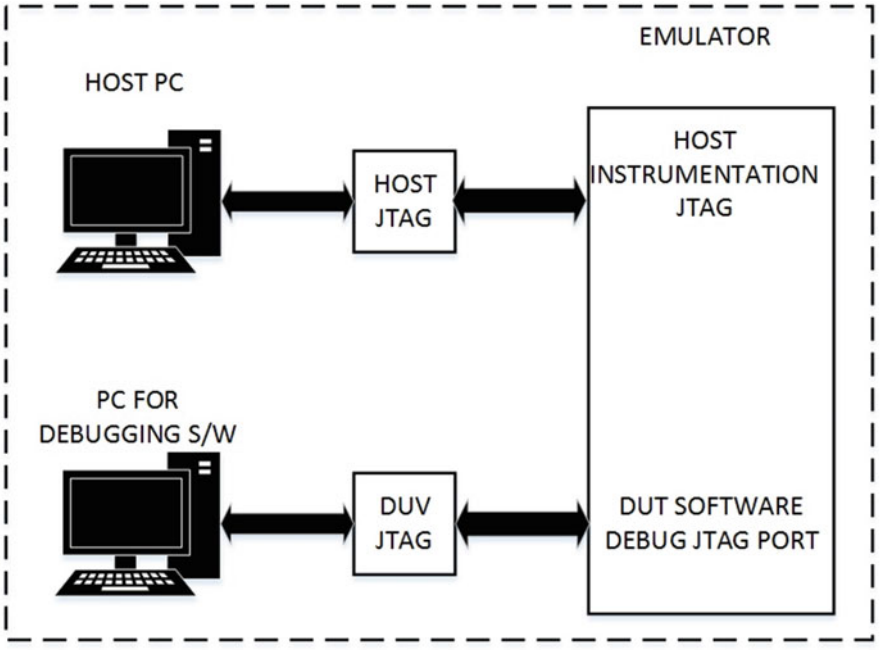

Fig. 18.7 Instrumentation (iJTAG) port connecting host computer and the emulator

1. Create a set of clock gates in instrumentation through the use of the *BUFGCE*, *BUFGMUX*, etc. The *BUFGCE* is used for *Enable*. The *BUFGMUX* is a mux between *instrumentation* mode and *functional* mode.
2. Create a set of counters, preferably one per primary clock. It should be possible to start, stop, and free run the counter. A set of count comparators, then could gate the clock to the functional logic blocks. Through the iJTAG one can write into these instrumentation registers which control the counters and clocks.
3. Using similar control instrumentation, you can also have some DUV internal signals *trigger* or *stop* the emulator clocks.

18.5.2 General Observability of Signals and Registers in the Design

The RTL synthesis process for FPGA optimizes out intermediate combinatorial logic signals. This scenario is in contrast with "array of processor"-based emulators, where each node can be maintained within the processor database.

- For the registers, using the iJTAG port, and decoding logic-related instrumentation, it is possible to have full controllability and observability. Figure 18.8 gives a feel of the instrumentation to be added for a register (flip-flop).
- For intermediate signals (part of combinatorial logic), a monitor flop and control mux can be added to gain controllability and observability.

There are multiple methods to enable these instrumentations:

- Modify the RTL to add pragmas known to Xilinx Vivado tool suite.
- Use a netlist editor tool post functional synthesis.
- Use a dedicated vendor tool for instrumentation insertion. Example Synopsys ZeBu tool suite does a seamless instrumentation insertion tailored to the ZeBu FPGA-based emulator.

18.5.3 Instrumentation for DUV Internal Memory

Often, it is needed to preload internal ROM and SRAMs with the executable code. The *C* program for the application is compiled, linked, and loaded into internal memories. The intent is to release the CPU reset and expect the CPU to execute the code and data loaded into the respective memories. Instrumentation can be added and accessed using the iJTAG as per the Fig. 18.8 even for memories. Note that the functional ROMs can also be preloaded using the iJTAG after instrumentation insertion.

For memories like dual-port memories, the port which has both write and read ports is chosen for instrumentation. Table 18.5 indicates the typical instrumentation that needs to be inserted for commonly used memories within the DUV.

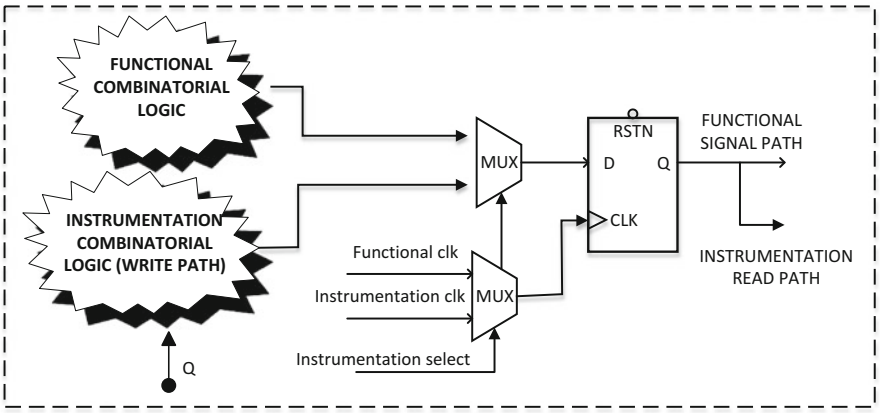

Fig. 18.8 Control and observability for registers using instrumented logic

Table 18.5 Typical instrumentation needs for memories

Memory	Functional	Instrumentation
ROM	Read only	(a) Clock muxing
		(b) Write port addition
		(c) Address and data line muxing
Single-port (SP) RAM	Read and write	(a) Clock muxing
		(b) Address and data line muxing
		(c) Write/read control signal muxing
Dual-port (DP) RAM	Different types	(a) Clock muxing on any one Write Port
	(a) 1 W, 1R	(b) Insertion of read port instrumentation for the write port (if it does not exist)
	(b) 1 W&R, 1R	(c) Address and data line muxing (for instrumented port)
	(c) 1 W&R, 1W&R	(d) Write/read control signal muxing

If the SP/DP RAM has bit- or byte-wise write and read control (functionally strobed lanes), then the instrumentation is suitably adjusted so that all the byte lanes are affected during memory load and dump through iJTAG.

The typical sequence for the usage would be:

1. Stop all the clocks to the emulator. This is through iJTAG-based instrumentation register configuration.
2. Preload the memories using external iJTAG:
   (a) Glitch-free selection of the clock to point to iJTAG_TCK.
   (b) Select the memory to be preloaded.
   (c) Preload the memory with the (address, value) pairs.
3. Apply reset to the DUV.
4. Start the clocks to the emulator.

5. Release reset to the DUV.
6. Expect the design to run the test (application).
7. Stop all the clocks to the emulator.
8. Read the memory (address, value) pairs, and store it to a file on host machine.

18.5.4 Adding Signal Observability (Waveforms)

Observing waveforms is an important part of the debug process and this feature is integral to any emulator. With regard to waveform, there are a few key concepts that need to be put in place as below:

1. *Signal List*: List of signals and buses (full hierarchical names) that you want to be added into the debug waveform.
2. *Trigger Signals and Trigger Expression*: A set of *Trigger* signals and the Boolean expression which would control the start and stop of the waveform capture.
3. *Trace Depth*: The maximum number of *waveform samples* that can be taken using the appropriate sampling clock.
4. *Trace Window*: The period of time when the waveform samples are captured. You can also have a circular trace buffer, allowing for a % trigger start, i.e., the trace starts x% prior to the actual trigger event and lasts up to $(100-x)$% after the trigger event. One can also define a pre-trigger percent or a post trigger percent based on this as is indicated by Fig. 18.9.

Chapter 17 explains various debug cores provided by Xilinx that can be used to capture waveforms. However, often, for deeper level of debug, the ILA is not sufficient, and at times the *Signal List* can span multiple FPGAs. To address this problem, emulators usually have their own external SRAM/DDR memory which can go up to 128 GB to enable deep trace. Intuitively, one can see that the instrumentation needed for this feature is huge. Some basic components are listed in Table 18.6.

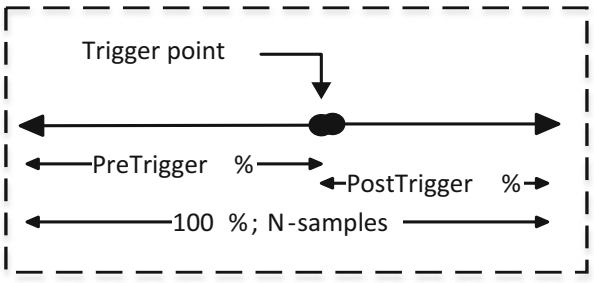

Fig. 18.9 Illustration of Trigger Point and "pre- and post trigger percent"

Table 18.6 Instrumentation components for waveforms using external memory

Instrumentation component	Usage
DDR Memory	The waveform samples would be written in the DDR memory. The samples are then read back and stored onto a host file
DDR Controller	To adhere to the DDR protocol for writing and reading the DDR memory
Signal Funnel	An instrumentation logic which converts (packs) the Signal List into chunks of data for writing and reading to the DDR memory
Instrumentation clock	Addition of an instrumentation clock, which is typically 1× or 2× the frequency of the sampled signals
Optional instrumentation CPU subsystem (iCPU)	The triggering, capturing of set of signals would need an instrumentation CPU to control the flow. The CPU would control the traces written to the DDR, and can also help in reading the traces and formatting for waveform generation by appropriate usage of iJTAG (host port connection)
	If an iCPU is being added, it can also be configured to enable other instrumentation tasks including complex clock management for starting and stopping the emulator