ARM

White Paper

Prototyping ARM® Cortex®-A Processors using FPGA platforms

Brian Sibilsky and Fredrik Brosser

April 2016

Contents

Introduction

With the increasing cost and complexity involved in new SoC (System-on-Chip) designs, FPGA (Field Programmable Gate Array) prototyping is becoming an increasingly important, or even crucial, part of new SoC projects. By offering a way to get to hardware sooner, FPGA prototyping allows hardware verification and software work to begin earlier, before first silicon, effectively pipelining the design process. Modern reprogrammable FPGAs are flexible and versatile computing and prototyping platforms - the ease of reconfiguring the development system for testing successive passes at the overall design offers a major advantage to the developer and gives confidence in the design before committing to producing a costly ASIC. Prototyping in FPGA also allows for debug and observability techniques that would otherwise not be available, such as inserting signal probes directly in the FPGA fabric. However, prototyping an SoC by implementing it into an FPGA does present some unique challenges that need to be considered. The underlying FPGA architecture and resources offer both limitations and possibilities when mapping an SoC design onto FPGA.

In FPGA, it is rarely possible to achieve the speeds that the IPs being implemented are intended to achieve in silicon. Due to various factors (such as pin multiplexing), the maximum frequency in a multi-FPGA design has traditionally been constrained to speeds well below the fabric limit of the FPGA.

In the past, with smaller ARM cores and less complex systems, it was normally possible to fit an entire system onto a single FPGA. Currently, even given the greatly increased capacity and diverse set of resources available on modern FPGA platforms, with the current demand for more powerful application processors and larger ASSPs, all of the building blocks constituting a system might not always fit into a single FPGA – even if using the largest FPGAs commercially available at the time of writing. It is therefore sometimes necessary to break up the design into smaller blocks and fit them into several connected FPGAs. This presents the additional problem of how to best partition a system or design across multiple FPGAs.

In this whitepaper, we discuss commonly encountered issues when prototyping ARM Cortex-A class processors using FPGA platforms. We show how to adapt ARM processor IP for implementation in FPGA, and give guidelines on how to approach partitioning a system across multiple FPGAs. FPGA platforms, boards and tools vary between vendors and versions. For documentation and support on these, it is advisable to contact the relevant vendor directly.

Clock Gating

ARM processor designs use gated clock structures, which are used for both functional and power management purposes. For example, when a processor core is quiescent, its clock can be gated off and the core placed in a power-saving mode. In ARM processors, these clock gating structures typically make use of latches to derive a gated version of a clock signal, as shown in Figure 1.

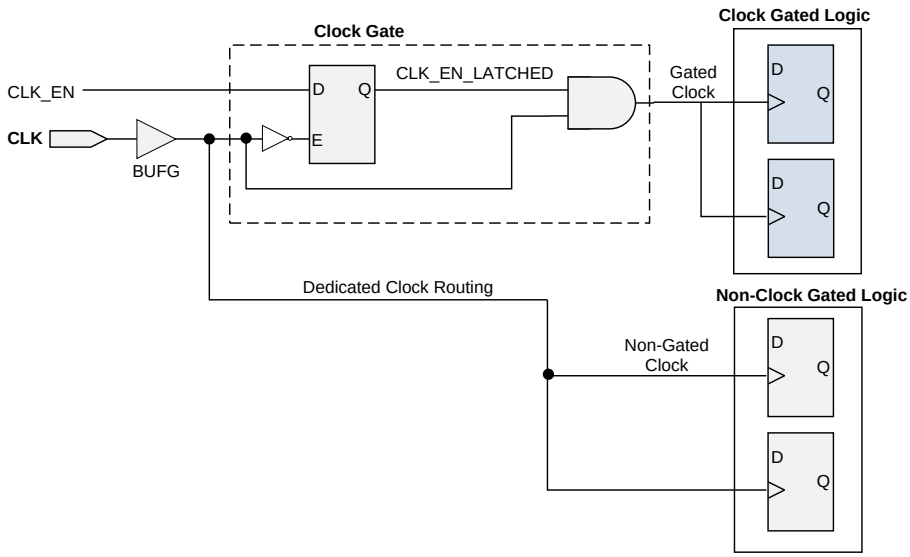

Figure 1. ARM clock gating

This presents a problem when porting the processor to FPGA: latches are not available as resources in FPGA, and FPGA architectures feature a fixed global backbone clock tree distributing the clock to the logic fabric and other resources. In FPGA, the clock gating scheme shown in Figure 1 would result in separate clock signals, which would normally be routed on dedicated, low-skew clock nets, being re-routed onto the general routing pool. This adds delays to the clock(s) and can result in a large number of hold time errors when the FPGA design is implemented.

Modern FPGA synthesis tools have features capable of recognizing gated clock structures within a design and converting them into functionally equivalent FPGA structures. This conversion prevents the synthesis tools from creating a new clock signal; instead, the enable signal is driven to the enable input of the relevant DFFs, as shown in Figure 2.

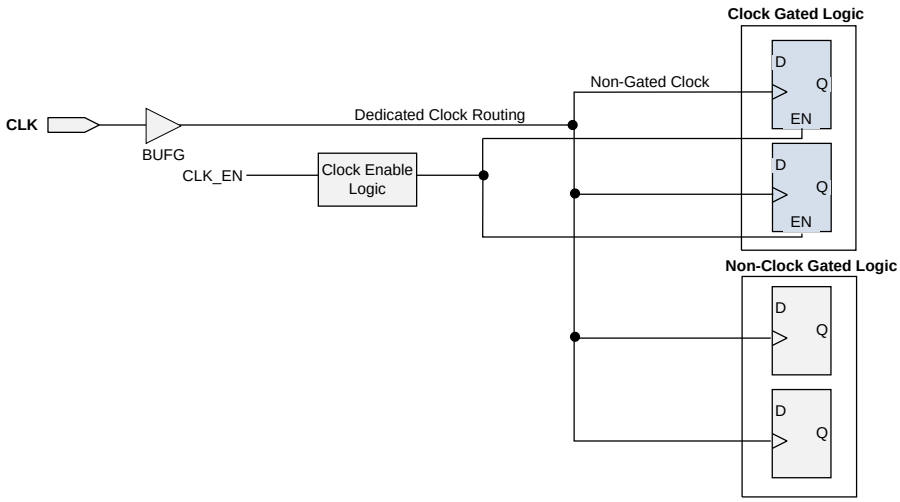

Figure 2. Converted clock gating structure

In order for this conversion to happen, it is sometimes necessary to modify a few key parts of the design (specifically the clock gating RTL modules), such that the FPGA synthesis tools recognize the clock gate. Clock gating constructs in ARM processor designs are typically implemented similarly to the behavioral RTL shown in Example 1, which also corresponds to Figure 1.

```
always @(clk or clk_en)
   begin
      if (clk == 1'b0)
         clk_en_reg <= clk_en;
   end

assign clk_gated = clk & clk_en_reg;
```

Example 1. ARM clock gating construct RTL

It might be necessary to modify this RTL to remove the latch, such that the FPGA synthesis tools will be able to recognize the block as a clock gate and convert the clock gate to the adapted design shown in Figure 2. Typical RTL clock gate modifications may include simply using an AND gate as the clock gate logic.

```
assign clk_gated = clk & clk_en;
```

Example 2. Modified clock gating construct RTL

The modifications necessary will depend on the requirements of the FPGA synthesis tool used, and some tools also require the clocks to be specified in a constraints file. Because the options and requirements vary between tools, it is always advisable to refer to the relevant tool documentation for more information, or to contact the tool vendor directly. Some tools use special flags or options to enable automatic conversion of clock gating cells. The relevant options (at the time of writing) for some of the more commonly used tools are listed below.

Altera Quartus II	See 'Auto Gated Clock Conversion'	(15.1)
Xilinx Vivado	See '-gated_clock_conversion'	(2015.4)
Synopsys Synplify Pro	See '-fix_gated_and_generated_clocks'	(2016.03)

It is always important to verify that the clock gate synthesis has been applied correctly by consulting the post-synthesis logs or by checking the actual synthesized design.

It is also worth noting that the suggested clock gate modification might cause failures in simulation. Therefore, if both simulation and FPGA implementation are required for the same RTL, it might be preferable to use a Verilog parameter in a define-ifdef structure in the clock gate RTL to allow switching between the original clock gate functionality (for simulation) and the modified clock gate (for FPGA implementation). This approach is shown in Example 3.

```
`ifdef SIMULATION
  always @(clk or clk_en)
    begin
      if (clk == 1'b0)
          clk_en_reg <= clk_en;
    end

  assign clk_gated = clk & clk_en_reg;
`else
  assign clk_gated = clk & clk_en;
`endif
```

Example 3. Modified clock gating construct supporting simulation and FPGA synthesis

RAM Implementation

The L1 and L2 cache memory structures in ARM processors are built up hierarchically from behavioral RAM models. ARM processor deliverables are supplied with a set of generic RAM models. Allowing the FPGA tools to consume these and automatically map them onto the available FPGA resources typically results in very inefficient use of the FPGA logic fabric and routing nets. This can happen when the toolchain selects distributed RAM instead of the embedded RAM resources for modelling large SRAM areas. This wastes FPGA fabric that could otherwise be used to implement the processor logic. The generic RAM models should be replaced with implementation specific RAM models. Typically the most efficient way to do this for implementation in FPGA is by manually instantiating the appropriate FPGA primitives into the cache memory models.

ARM processor cache models make extensive use of small RAM blocks with multiple bit-wise write enable signals. This presents a problem if the FPGA RAM primitives only support byte-wise write enable signals, for example as in the Block RAMs (BRAM) in Xilinx Virtex FPGAs. Allowing the FPGA toolchain to automatically map the processor RAMs onto BRAM resources would result in approximately eight times more RAM blocks being used than are actually needed. Because RAM blocks are in limited supply, we need a method of using them in a more efficient way. Figure 3 shows how bit-wise write enable functionality can be achieved using a dual-ported BRAM block.

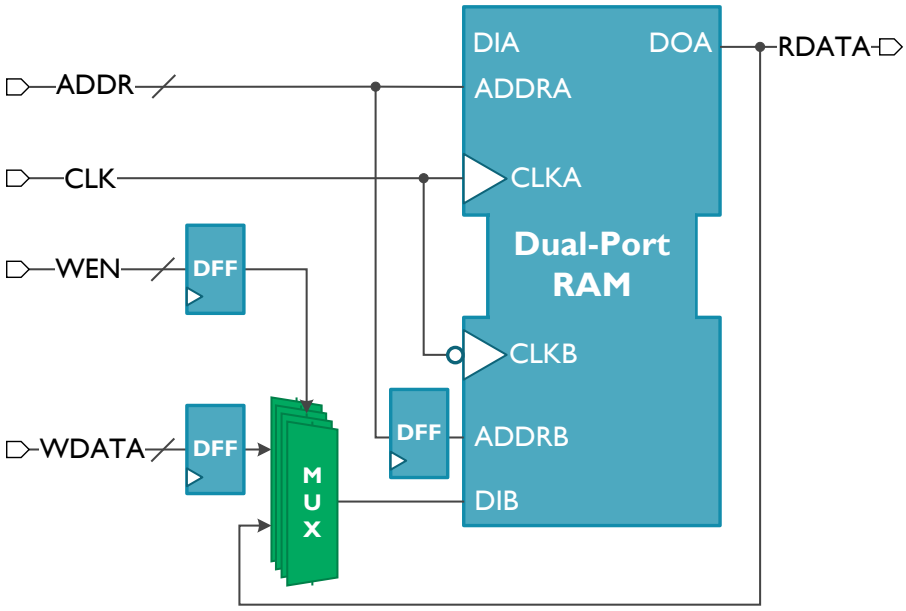

Figure 3. BlockRAM write enable scheme

Bit-wise write enable functionality is achieved by combining the new write data with data that has been read from the same location. The bits to be written from the externally attached module are substituted into the data read from the output port by means of a MUX block. A dual-port RAM block is required to achieve this functionality because the output data must be both read from and written back into the RAM array within the same clock cycle.

When a write occurs, data is read from port A on the positive edge of the clock. On the following negative edge, the combined read and write data is written into the B port of the RAM block, due to the inversion of that clock input. This means that the correct updated data will be available for a subsequent read access to the same address on the next clock cycle.

Some ARM processors use latches on the RAM read outputs. In an FPGA implementation, these latches can be replaced with DFFs. For further information, refer to the relevant Configuration and Sign-off Guide.

Design Partitioning

It is sometimes necessary to partition a design across multiple FPGAs, especially when prototyping larger designs such as a complete processor system. Figure 4 shows an example of a typical ARM-based processor subsystem and some of the internal building blocks that might make up such a design. Such a system might contain the following elements: one or more clusters of ARM Cortex-A core processors with integrated level 2 cache memory, debug and trace logic, a quantity of peripherals and an interconnect subsystem.

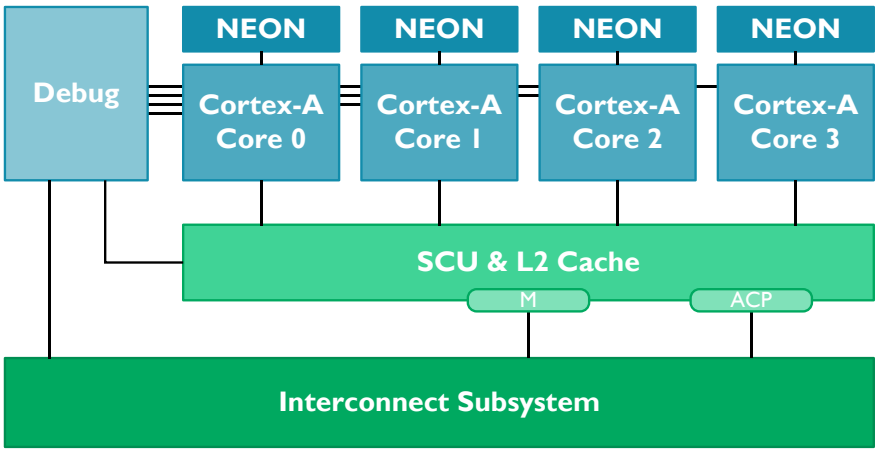

Figure 4. Example ARM processor subsystem

Each of these building blocks is typically made up of several RTL source code modules. In theory, these modules could be distributed between different FPGAs, but this is strongly discouraged. Doing so would risk adding unnecessary complexity to an already difficult task. It could also break the design by adding clock skew or signal timing delays inside a functional block unable to handle that additional delay.

If partitioning is necessary, it is typically preferable to partition a design at natural device boundaries. For example, a single processor core and its private peripherals could be placed in a single FPGA. The design of ARM processors and associated peripheral and debug logic is appropriate for this approach, because the interfaces between the blocks are usually registered. This makes it easier to meet timing requirements and to achieve reliable operation in the resultant FPGA design(s). Another potential benefit of partitioning a design is to avoid over-utilization of the resources on any single FPGA, and the associated potential negative performance impact. However, this needs to be balanced with the cost of routing signals between FPGAs, because it is considerably slower than internal routing.

The dotted lines in Figure 5 show potential places to divide the design. Each dotted area represents the contents of one FPGA in the multi-FPGA system that would be needed to house the complete SoC. However, it is generally preferable to avoid over-partitioning the design, because of the issues related to routing signals between FPGAs. It should be noted that with contemporary large FPGAs, it is unlikely that the amount of partitioning shown in Figure 5 would be necessary, and as such it serves as an example only.

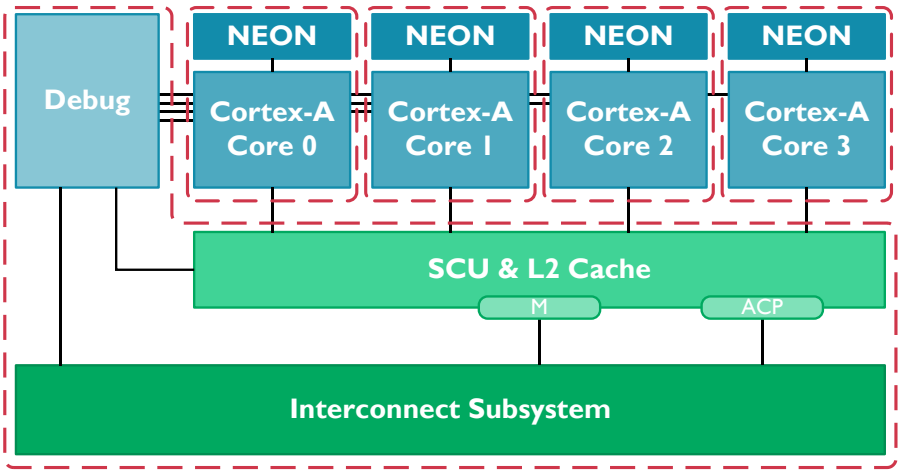

Figure 5. Partitioned system

In Figure 5, the debug logic, SCU & L2 memory system and the interconnect subsystem are all shown as residing in one FPGA. We also show each Cortex-A processor core occupying one FPGA. The method for deciding which parts of the partitioned design will fit into which FPGA will depend on the architecture of the development system. It can be seen that the split system in Figure 5 would best fit in an FPGA platform which has five smaller devices: one central FPGA for the debug and memory system components, and four satellite FPGAs for each of the processor cores. Such a partitioning is illustrated in Figure 6.

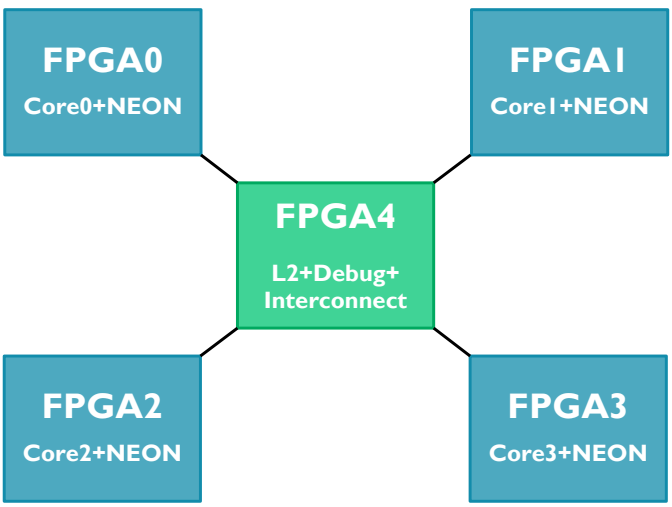

Figure 6. FPGA partitioning

Signal Multiplexing

With the partitioning shown in Figure 5, there are many signals going between the Cortex-A cores and the L2/SCU. Depending on the processor and the target FPGA, the number of signals might exceed the number of available I/O pins available on the FPGA. The solution to this is to multiplex inter-FPGA signals as efficiently as possible, while taking care not to cause timing problems.

The number of multiplexed signals that can share one I/O pin depends on a number of factors, including the MUX ratio available on the FPGA, the system clock speed, and the quality of the transmission lines between the FPGAs. Figure 7 shows an example of such a scenario, where a large number of signals need to be carried over a smaller number of physical wires between FPGAs or between boards.

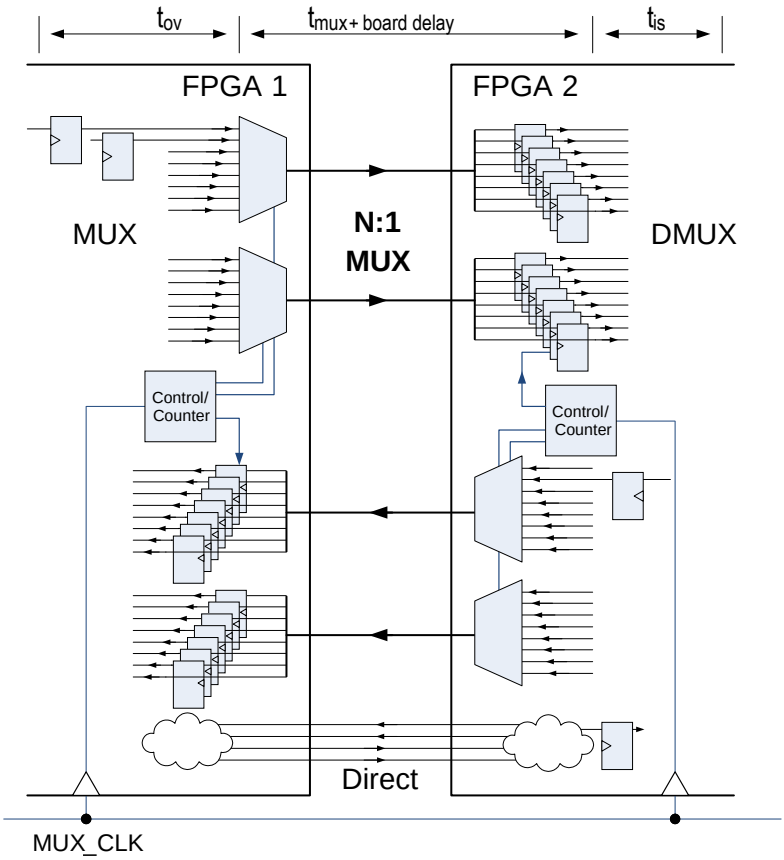

Figure 7. Signal multiplexing

The basic principle illustrated in Figure 7 is that a large number of signals can be multiplexed onto a single wire that passes between two FPGAs. Those FPGAs may reside on the same physical PCB, or they may be on different PCBs with header connectors making the connection between the boards.

In this simplified example, the upper left part of the diagram shows a group of signals being concentrated onto two physical wires. On each successive MUX_CLK clock edge, a different input to each MUX is selected by the block marked 'Control / Counter' and placed onto the wire that runs between the FPGAs. These blocks keep track of which cycle in the MUX process is currently active, and drive the MUX select inputs accordingly.

FPGA partitioning tools can typically insert these control blocks automatically into the design. In these cases, the developer often only needs to specify which internal FPGA signals are to be used for the MUX_CLK and MUX process synchronization.

On the upper right side of the diagram in Figure 7, synchronous data elements are used to store the intermediate values that appear at the inputs on each MUX clock cycle. Eventually, all of these data elements will hold a value that corresponds to the original data. The data must be passed across the link and reassembled by the data elements before the next master clock edge (not shown). The master clock in this case is the main clock signal for the ARM processor that is being split between several FPGAs. The lower half of Figure 7 shows an equal number of multiplexed signals running in the opposite direction, and a group of combinatorial logic signals passed across non-multiplexed.

Assuming an example MUX clock speed of 100MHz and a MUX ratio of 8:1, the multiplexing scheme in Figure 7 can be implemented with the timing shown in Figure 8.

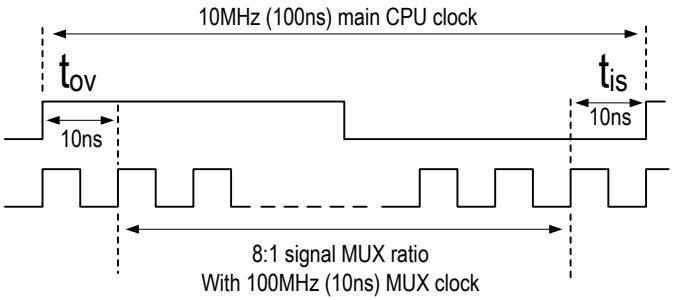

Figure 8. MUX timing

Allowing one MUX clock cycle each for the system output data to become valid at the source, and the de-multiplexed data to be ready at the destination, the 10:1 clock ratio shown would allow 8 successive sets of data to be sent across and registered. This simple method is known as Synchronous multiplexing. Depending on the FPGA model used, this multiplexing scheme will be limited by the MUX ratio and the clock frequency limitations of the FPGA MUX logic. There may also be upper and lower limits to the PLL clock generation modules in the FPGAs that are required to generate MUX_CLK.

An alternative option is to employ a 'Source Synchronous' solution with DDR clocking on the multiplexed signals. Modern FPGAs feature DDR blocks, which are specialized elements present in the FPGA I/O pad blocks that combine the DDR and multiplexing structures needed. These blocks are normally also instantiable as primitives, and offer automatic multiplexing without the need for manual control of the MUX select. Many FPGA synthesis tools are also able to infer these. DDR logic blocks typically give very predictable and stable timing characteristics and allow the data to be sent at double the normal rate. Figure 9 illustrates this method.

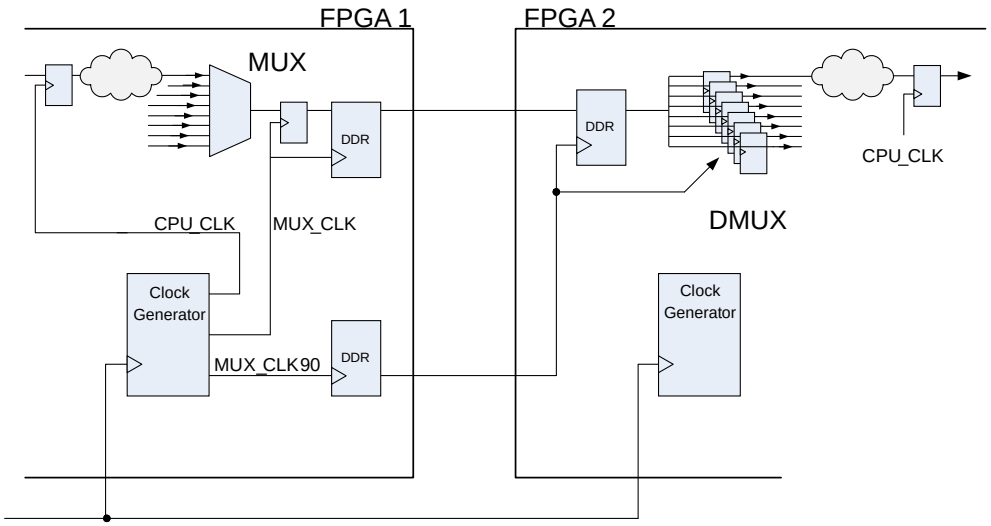

Figure 9. DDR blocks

To account for the board delay between the two FPGAs, the DMUX clock is shifted by 90°. This means that the DMUX clock edges coincide approximately with the center of the received data values, allowing reliable capture of the data. Figure 10 illustrates this principle.

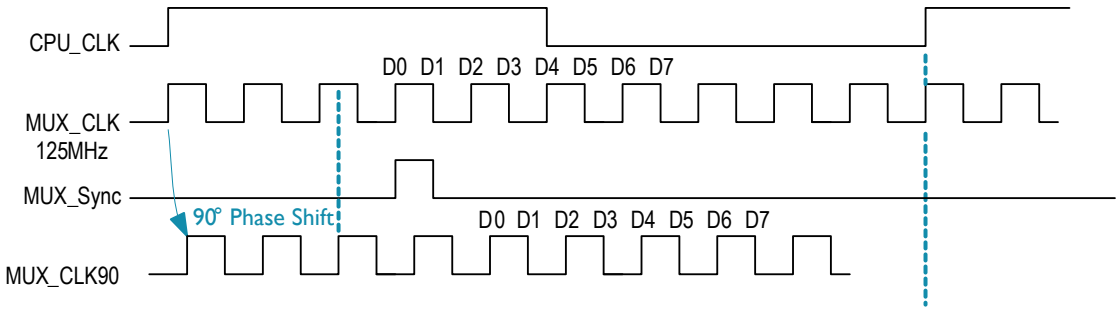

Figure 10. Clock edge shifting

Some recent FPGA families also offer the possibility of implementing multiplexing based on I/O source-synchronous serialization/de-serialization (I/OSERDES) primitives. These blocks can be used with I/O delay primitives and calibration logic implemented in the normal FPGA logic fabric to build structures with high multiplexing ratios, making it possible to achieve significantly higher speeds per wire.

Route-through Signals

In some instances it may be necessary to pass signals that originate in one FPGA straight through one or more intermediate FPGAs in order to get to a destination in another FPGA. This could for example be required where boards are stacked on top of each other with no direct PCB trace connections from the first FPGA in the chain to the last – such as is the case when stacking ARM *LogicTile Express* 20MG boards.

Passing signals straight through an FPGA with a simple assignment statement in RTL may cause timing problems for the system design as a whole. The pad and routing delays from one side of an FPGA to the other can be enough to break the timing model for an inter-FPGA signal multiplexing scheme.

One solution to this problem is to register the route-through signals in each FPGA and add them to the MUX. The signals can then be de-multiplexed and re-assembled at the target destination, in time for the next master clock edge. Figure 11 shows an example of this solution type.

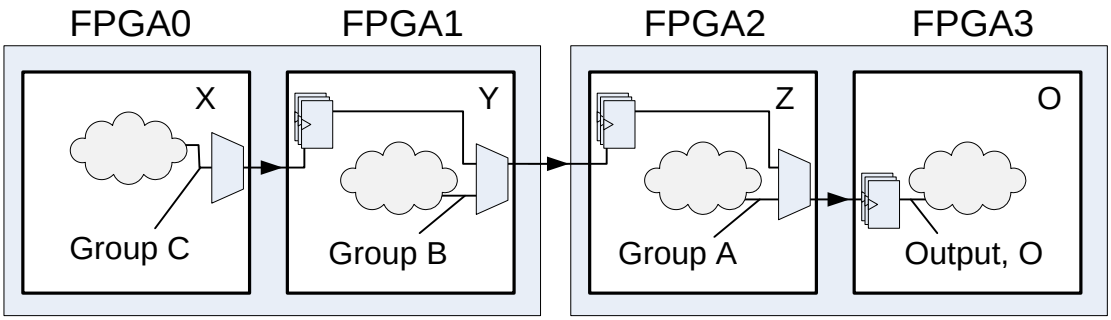

Figure 11. Route-through scheme

Figure 12 shows how the multiplexed data from points X, Y and Z in Figure 11 would appear at the destination, O.

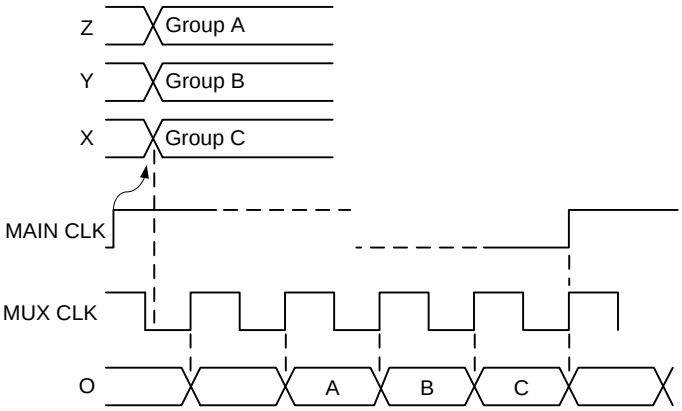

Figure 12. Route-through timing diagram example

It should be noted that while such a scheme is entirely possible to construct, the task of arranging the routing so that all relevant signals are multiplexed, de-multiplexed and re-assembled at the correct points in time can be a complex one. It should also be noted that depending on the FPGA partitioning tool used, it may not be possible for the tool to automatically recognize or handle the routing of route-through signals. Therefore, if this type of route-through scheme is used, additional care should be taken to ensure that timing is met.

Handling Asynchronous Resets

It is quite common to have an asynchronously generated reset input to a development system. This reset input would normally be synchronized to the master clock by means of cascaded DFFs. The reset synchronizer construct should not be replicated into the other FPGAs that house part of a partitioned design. This is because each replicated synchronizer would get its asynchronous reset input at a slightly different point in time, due to board delays.

With such a setup, we might get the scenario where different synchronizers align their respective copies of the asynchronous reset input to different master clock edges. This can happen where one synchronizer sees the asynchronous input change very close before a master clock edge, but due to board or FPGA routing delays, the other synchronizers see the asynchronous input change just after that clock edge.

To avoid this scenario, there should be only one reset synchronizer, as shown in Figure 13. It is typically safe to export the synchronized reset signal from one FPGA to the others in the system, because the routing delay on that exported signal should be small in comparison to the master clock period.

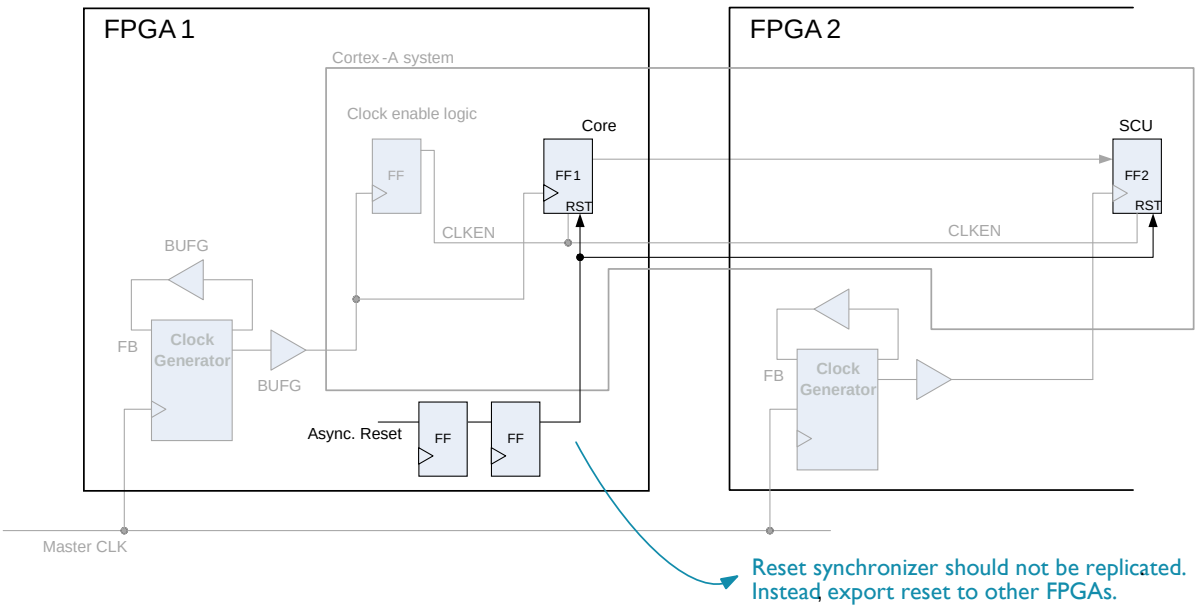

Figure 13. Reset synchronizers in partitioned system

Additional Clock Gating Considerations in Partitioned Systems

If an SoC design is to be split between multiple FPGAs, modifications to the clock gating logic are necessary. This is in addition to the process detailed in the Clock Gating section of this document.

One master clock should be fed to all the FPGAs that contain the partitioned parts of a larger design. It is not advisable to export a clock signal that is being used in one FPGA and feed it into another, as this will cause clock skew between the FPGAs. To ensure that each FPGA uses a synchronized clock internally, it is necessary for the clock gating logic to be copied to each FPGA so that each internal clock signal has the same timing. Some partitioning tools are able to recognize the clock gating logic and replicate it into the other FPGAs. The exact tool flow will vary between FPGA tools, but typically involves the steps listed in Example 4.

1. Perform a preliminary synthesis run on the entire design.
2. Feed the preliminary netlist into the chosen partitioning tool.
3. Partition the design using the partitioning tool.
4. Replicate clock generation/gating blocks into the other FPGA(s).
5. Re-synthesize each partitioned FPGA.

Example 4. Clock gating replication flow

The tool flow steps in Example 4 are illustrated in Figure 14 (step 1), Figure 15 (steps 2-4) and Figure 16 (step 5).

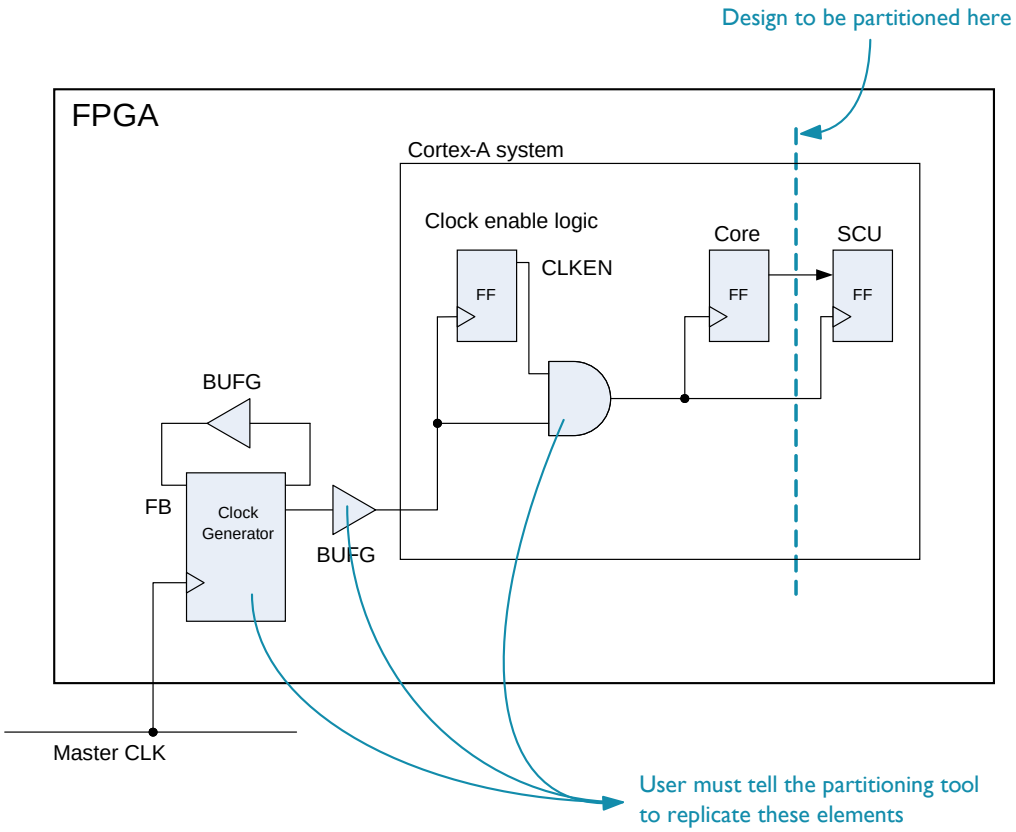

Figure 14. Before partitioning (step 1)

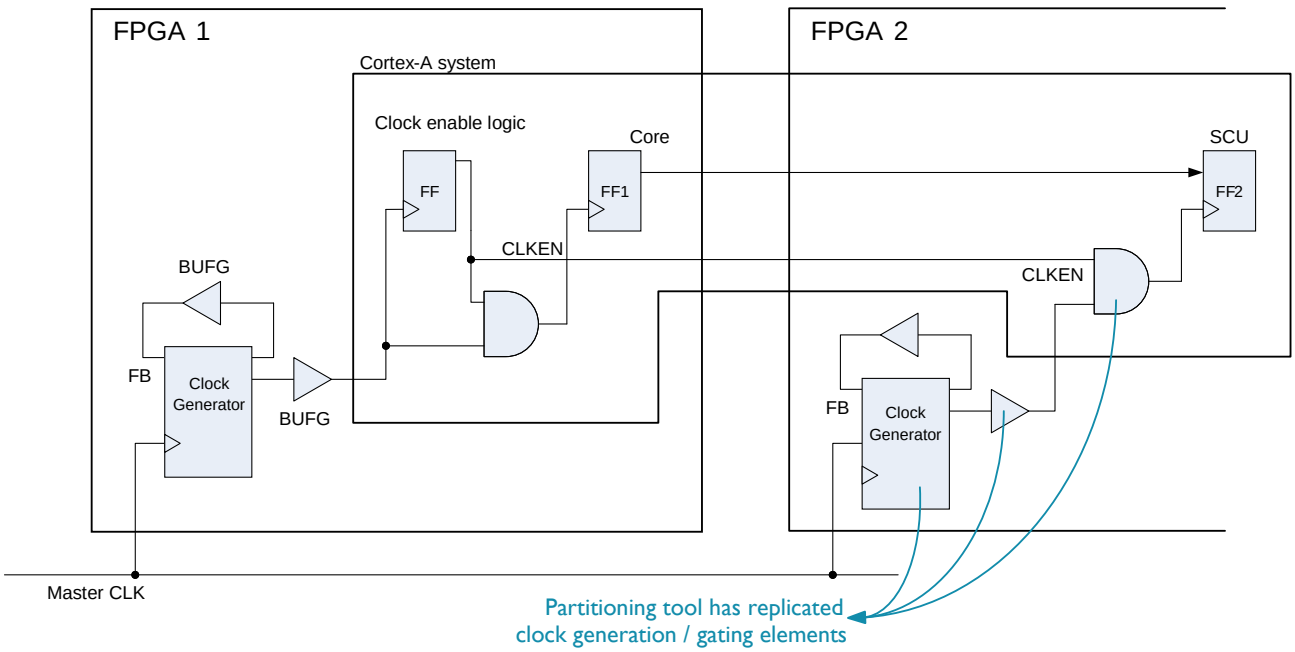

Figure 15. Partitioned design (steps 2-4)

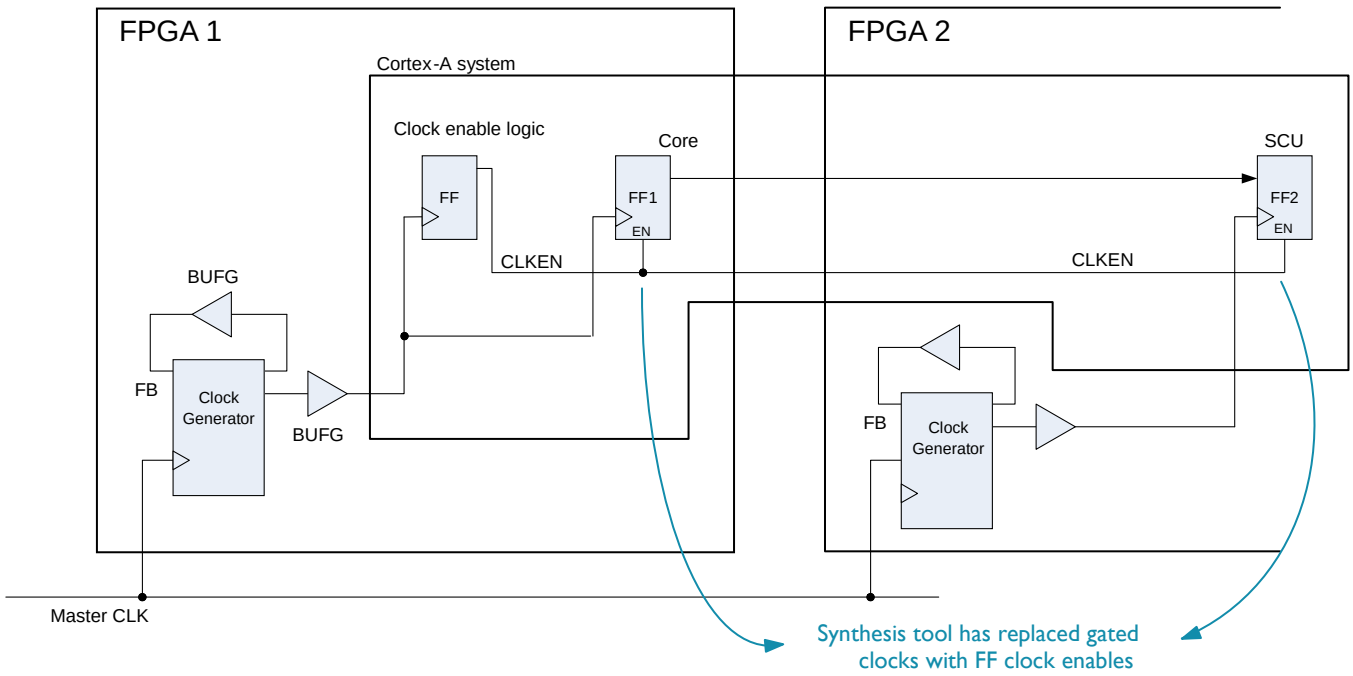

Figure 16. Partitioned design with converted clock gates (step 5)

Trademarks