FPGA Prototyping: HDL migration and FPGA Debug

The main difficulty in FPGA prototyping is the short development cycle for complete SoC testing. This cycle includes RTL migration to FPGA for complete design on board level and its debugging strategies. Here, we focus on HDL migration and FPGA debugging methodologies to meet the said criteria in an efficient way.

By Sachin Gorkhe, Einfochips Bangalore Ltd., Ahmedabad, India

(December 20, 2011)

Abstract:

The complex manufacturing requirements of next generation devices have increased the cost of ASIC and ASSP development. The complex, low-power designs requires early integration of various hardware features with corresponding firmware onto one silicon device. With increased complexity, another cost becomes a limiting factor to the effectiveness of verification, simulation runtime and inaccuracy of stimulus models e.g., Audio-video based SoC with content protection algorithm in real time.

FPGA prototyping is an efficient approach to validate the pre-silicon system-on-chip(SoC), in parallel development with software development and Time-to-Market (TTM) requirement of an ASIC chip. The main difficult criteria for the FPGA prototyping is the short development cycle for complete SoC testing. This cycle includes RTL migration to FPGA for complete design on board level and its debugging strategies.

In this paper, we will focus basically on HDL migration and FPGA debugging methodologies to meet the said criteria in an efficient way.

Introduction

FPGA prototyping is rapidly becoming key to ASIC functional verification. It offers a low cost way of verifying the RTL under real stimulus and developing the complex software that now comprises 50% of development efforts as well as accelerated software and firmware development.

With advantages like faster validation, real-time validation, at-speed interfaces, real time data handling, early software development, post tape-out ECO validation etc. prototyping on FPGAs is fast becoming the preferred methodology for ASIC verification.

Following are the objectives for FPGA prototype to be effective:

1) Conversion of ASIC RTLs to FPGA compatibility (migration of ASIC RTL) without changing or affecting its functionality.
2) Minimal changes for the RTL and maintainability of the FPGA database
3) Easy readability within the RTL after migration

4) Short development cycle from RTL fixing to board level bring-up of FPGA
5) High system performance and zero bugs

The above objective can be targeted by grouping them.

i) RTL migration from ASIC to FPGA
ii) FPGA type or board selection for the functionality
iii) Synthesis options/tools
iv) Debugging strategy
v) Bug/timing closure

Following are major challenges in achieving said objectives:

a) Target technology Library
b) FPGA Database maintainability
c) Synthesis and debugging strategy
d) Firmware changes

Prerequisites of FPGA prototyping

FPGA validation is depends on the complete understanding of SoC level architecture for a FPGA validation engineer. FPGA engineer needs to work closely with SoC design team and micro-architecture team of the complete SoC. As SoC design is accomplished by other team FPGA team need to capture some vital information from those teams.

Below are some ASIC attributes which need to be collected by FPGA engineer prior to develop the FPGA RTL changes.

S.No.	ASIC Design Attributes
1.	Number of clocks used and their targeted frequencies
2.	Gated clock implementation
3.	Clock dividers, PLL/DLL, derived clock information
4.	Reset logic
5.	IO standard requirements
6.	Memory instantiated, type and size
7.	Macros used, specific SoC library, IPs used
8.	Any embedded Processor, specific bus architecture
9.	Cross domain clock
10.	Debugging strategy for each module
11.	Synthesis attributes for signals to keep/preserve
12.	No. of signals requirement for FPGA in-built signal analyzer

Table 1: ASIC SoC Design Attributes

FPGA selection

The FPGA selection can be chosen from the information given in the above ASIC attributes. The base and the approximate estimation can be calculated using these attributes. The main things for such FPGA would be total resources like gate count, BRAM, hard core processor if require, DCMs/PLLs, global IOs for clocks and resets, no. of IOs available and available standards for IOs.

Now the FPGA resources should be atleast 10-20% higher to accommodate the complete RTL and in-built logic analyzer in the FPGA for the debugging purposes.
Often multiple FPGA devices must be used to prototype a single ASIC. The obstacle of using multiple devices is the task of connecting all of the logical blocks of the ASIC design across multiple FPGA devices.

FPGA Prototype setup

Now a day, many FPGA vendors are having FPGA development boards to test almost every standard protocol or for high end protocols (like PCIe, USB3.0, DisplayPort etc.) or with the

third party vendors with free portability of vendor IPs (like analog PHY) for said protocol. One must can check his requirement with those availability or can get information from FPGA board vendors for the ready to use boards. These boards normally come with high end FPGA mounted with high logic gate count and other resources (DLL/PLL, IOs, BRAM etc.). This will save the time for designing a custom board for FPGA and less time to market.

RTL code migration

Migration from ASIC to FPGA will require RTL changes. The RTL stable release (stable can be define by basic functionalities passes with RTL simulation) can be used for RTL changes. With the first changes, compiler directives should be used in the RTL for FPGA changes. These directives should be conditionally compile code. Basically “ifdef” in verilog should be used. This is because the ASIC RTLs can be release many times during FPGA validation phase. ASIC RTL and FPGA RTL cannot be maintained with different database as these are very difficult to maintain with every RTL release.

For example -

```
`ifdef FPGA_EN
assign cg_intclk = ~cg_intclk;
`else
assign cg_intclk = scanmode ? occ_135clk : ~cg_intclk;
`endif //FPGA_EN
```

The above example is the gated clock logic which has been removed in FPGA approach. The above piece of RTL code tells compiler which part to be compiled and which should be neglected for both either way ASIC and FPGA. Due to this approach the functionality mismatch will not be happened.

RTL Code Migration steps

Given below are the10 steps to be followed for RTL design conversion

Step 1: Replace ASIC RAMs to FPGA RAMs (using CORE Gen. tool)

Step 2: ASIC PLLs to FPGA DCM & PLLs (using architecture wizard), also use BUFG/IBUFG for global routing.

Step 3: Convert SERDES (Using Chipsync wizard)

Step 4: Convert DSP resources to FPGA DSP resources (using FPGA Core gen.)

Step 5: Replace IPs (using CORE Gen. with available IPs)

Step 6: Use specific SRL (HDL – for design pipelining if any)

Step 7: Use Clock enables (HDL – for gated clocks)

Step 8: Employ other good HDL coding styles (HDL – for better performance)

Step 9: Synchronous design practices (Pipeline comb. Logic)

Step 10: Examine the coding style especially for FSM (Should not be too large, does not create timing problems etc.)

Step 1: Replace ASIC RAMs to FPGA RAMs

Today technology is getting advanced and continuously shrinking the die sizes, there are challenges to accommodate large memory sizes and logic on the same area within the silicon. ASIC vendors are using hard macros and instantiated those memory components with various sizes available into the design. Due to those hard macros implementation, only simulation models are available for simulation purposes but for FPGA implementation, vendor specific FPGA BRAM should be generated and instantiated to meet functionality and sizes available with FPGA vendors.

Approach:

The memory can be generated using FPGA vendor specific tool (e.g. coregen for Xilinx). As this is the vendor specific approach, it would not be the same port names as ASIC RTL have. So need to write a wrapper which will handle all the memory transactions.

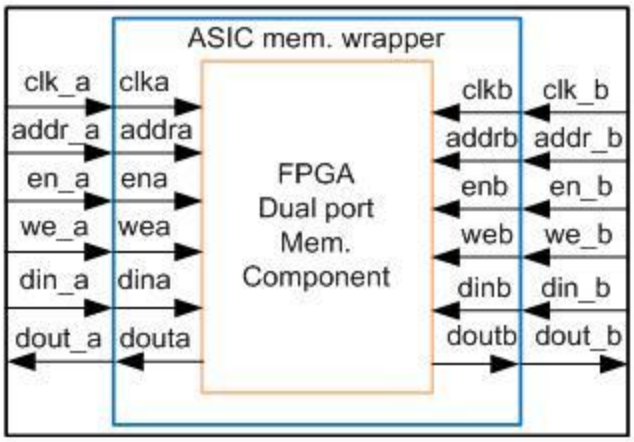

Figure 1: ASIC memory wrapper

Step 2: ASIC PLLs to FPGA DCM and PLLs

In an FPGA, clock resources are very limited but in ASIC RTL has a clock gen. module which is having multiple clocks to control the entire SoC. The frequency control in the FPGA is fixed, we cannot vary the particular clock in FPGA, the multipliers in the DCM/PLL are fixed valued. Clock multiplexers are also limited into the FPGA. In ASIC RTL, many macros, buffers which are generating desired clock frequencies under register programmable control. This we cannot implement into the FPGA.

Approach:

With this limitation in the FPGA, there are some approaches with which it still can implement clock dividers and multiplexers in FPGA by generating derived clock and re-entering it through global clock pin using IBUF/IBUFG component for quick compile time, good timing analysis and for proper implementation structure. Also many FPGAs supported clock buffers though they are limited and have location based routing limitations. These buffers can be connected to the clock to avoid driving the clock out and re-entering again.

All the buffers implemented in the ASIC RTL need to be removed and equivalent assignment statements have to be implemented in FPGA RTL.

Step 3: Convert SERDES (Using Chipsync wizard)

Increasing data transmission rates are making it more difficult to verify and prototype ASIC based systems. For example, prototyping SerDes in an FPGA has significant limitations. Not only are the electrical and logical behaviors difference between ASIC and FPGA SerDes, but the RTL verified in the prototype also requires special attention to adapt to the functionality differences.

Current FPGA technology does not support 10G links(SerDes), so if any serdes is being used in the ASIC RTL then it needs to be converted for FPGA for supported links and speeds.

Approach:

Though FPGAs not supported 10G speed for Serdes, FPGA prototype can be validated on degraded supported speed. Some FPGA vendors like Xilinx describes how a fast data link was implemented by using a Xilinx RocketIO™ SERDES running at 3.125 Gb/s with 8b10b encoding to achieve a 2.5 Gb/s payload. It also describes how a much slower data link was implemented

within the same system by using a double data rate (DDR) LVDS with clock forwarding and at a clock rate of 156 MHz to achieve a 400 Mb/s payload. The configurable chipsync wizard can be used for such serdes generation for xilinx FPGAs.

With this unique built-in Chipsync technology realize over 2+ Gbps performance. Chipsync source synchronous technology embedded with every I/O. Also dynamic programmable delay/data centering with per bit de-skew on every I/O supported.
Step 4: Convert DSP resources to FPGA DSP resources (using FPGA Core gen.)
In ASIC RTL, multipliers/dividers are either left to synthesis tool to infer from behavioral description or instantiated from DesignWare library from Synopsys. Hence these multipliers/dividers are not synthesizable for FPGA unless the RTL is modified. Resources point of view FPGA having limitations on DSP algorithm implementation.

Approach:

All current FPGA architectures contain dedicated arithmetic resources. Such resources can be used to perform multiplication, as in many DSP algorithms.
Nowadays most of the FPGA families have multiplier hard-macro elements optimized for performance. The ASIC multipliers can be replaced by creating a wrapper for the multiplier core of desired configuration generated by the FPGA core generator. These macros are good in performance and optimized in area for better timing.

Step 5: Replace IPs

For prototyping it is very important to check if any specific IP instantiated into the SoC, if it is in RTL format then it is easy for any synthesis tool to synthesize it directly. Also if any technology independent netlist is available then for FPGA it is required to have such component from specific vendor for synthesis. If it is in .db netlist then FPGA compiler from Synopsys is required or any similar netlist from any other vendor.

Approach:

Some synthesis tools like Synplicity, Certify from Synopsys supports many netlist formats to synthesize. Certify tool supports synthesis of DesignWare components, but it may not supported all components. In such cases, it needs to be created in verilog or should be converted to any other supported netlist (gtech netlist) form. Also it is noted that check if the SoC is having mixed languages (VHDL, Verilog, SV etc.) because all synthesis tools not supported mixed languages for synthesis. So it is required to check which tool is best for such support and suits the requirement.

Step 6: Use specific SRL without reset (HDL)

ASIC RTL may have implemented shift registers for many serial communication protocols (like Serdes), which is suitable as per the DesignWare library it is targeted for. These shift registers in

ASIC RTL having reset logic (sync. Or async.) depending upon the logic implemented. But in FPGAs, avoid resets on shift registers as it prevents inference of area and performance.

Approach:

A reset cannot be described in the shift register RTL code because the SRL library component (Xilinx) does not have a reset. Using resets in the code that infers shift registers requires either several flip-flops or additional logic around the SRL to allow the reset function.

Code without reset on shift registers generally produces a single register on the output, which is optimal for area and performance. The additional resources (flip-flops and routing) can have a negative influence on the placement and routing choices for other part of the design, possibly resulting in longer routing delay for other part of the design.

Step 7: Use Clock enables (HDL – for gated clocks)

FPGA generally recommends using the dedicated clock-enable port instead of gating the clock port, but in ASIC implementations, clock gating is a common practice for power saving. Gated clocks can cause glitches, increased clock delay, clock skew, and other undesirable effects. Using clock enable saves clock resources and can improve timing characteristics and analysis of the design.

Also occasionally derived signals are also used as clocks in ASIC implementations. But in FPGA designs, it is not possible to route such derived signals as clocks with low delay and skew.

Approaches:

There are several ways to use clock-enable resources available on devices. To gate entire clock domains for power reduction, it is preferable to use the clock-enabled global buffer resource called BUFGCE. For applications that only attempt to pause the clock for a few cycles on small areas of the design, the preferred method is to use the clock-enable pin of the FPGA register.

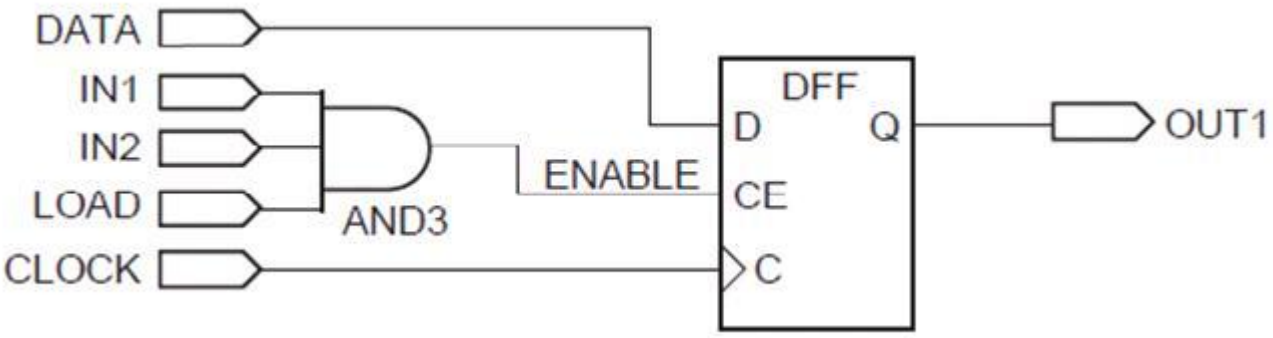

Figure 2: Clock Enable - efficient way of gating a clock signal

Step 8: Employ other good HDL coding styles (HDL – for better performance)

To efficiently usage of memory elements, following factors must be considered:

• deciding to use dedicated blocks or distributed RAMs

• using the output pipeline register

• Avoid asynchronous reset because it prevents packing of registers into dedicated resources and affects performance, utilization, and tool optimizations

Other factors, namely HDL coding style and synthesis tool settings, can substantially impact memory performance.

Synthesis tools are able to infer following modes depending on the coding style.

• 'write first' or transparent mode
• 'read first' or read before write(slower)
• 'no change' mode

Avoid "read before write" mode to achieve maximum block RAM performance. Check synthesis tools settings, inference templates, and limitations to maximize memory block's performance.

Approaches:

1. Use of asynchronous reset:

The use of asynchronous resets can inhibit optimization. If an asynchronous reset is not used, then the resources that such a signal would use are available and can be used to optimize other synchronous paths driving this register.

Example One: To implement the asynchronous reset code, the synthesis tool must infer two LUTs for the datapath because five signals are used to create this logic.

```
always @(posedge CLK, posedge RST)
if (RESET)
Q <= 1'b0;
```

```
else
Q <= A | (B & C & D & E);
```

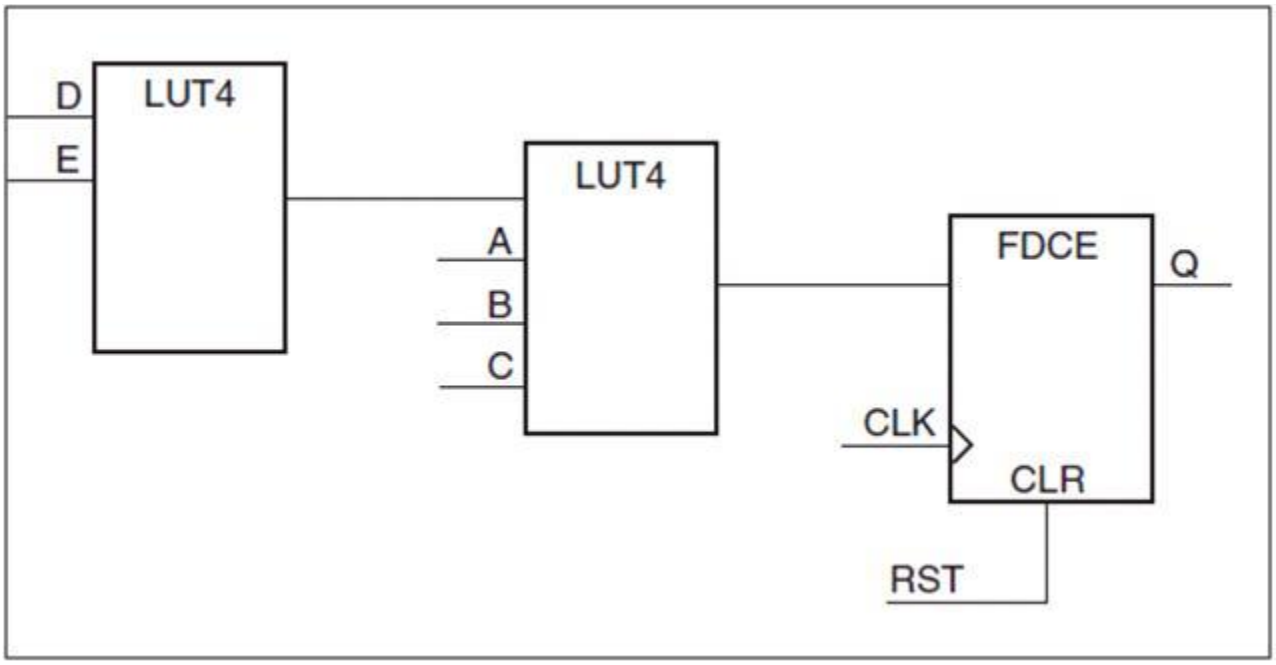

Figure 3: Asynchronous reset implementation

Example Two: Rewriting this same code with a synchronous reset gives the synthesis tool more flexibility in implementing this function.

```
always @(posedge CLK)
if (RESET)
Q <= 1'b0;
else
Q <= A | (B & C & D & E);
```

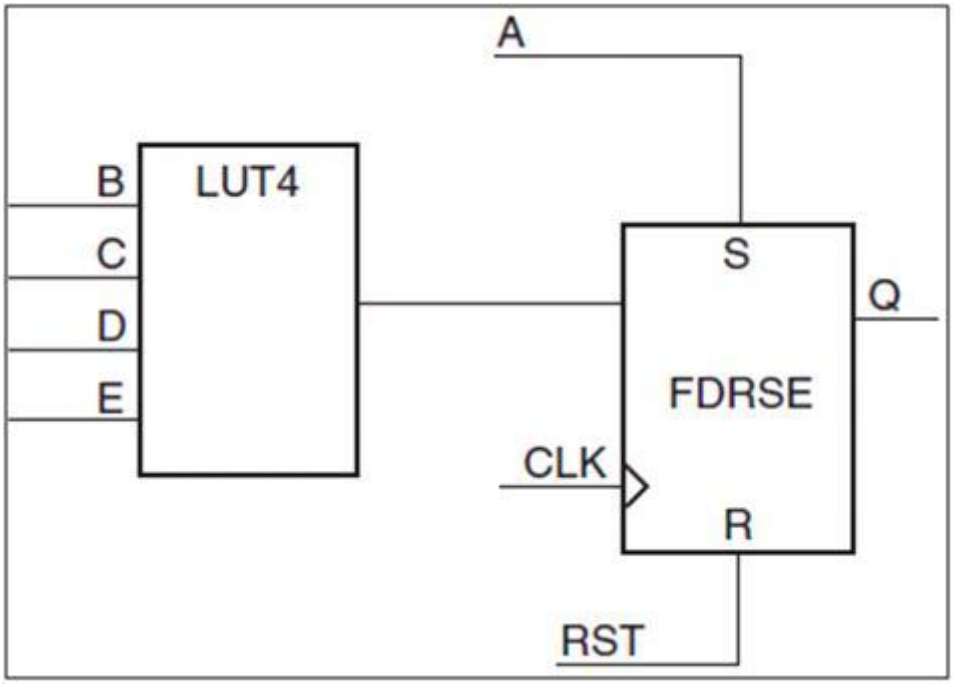

Figure 4: Synchronous reset implementation

With the implementation in above figure, the synthesis tool can identify that any time A is active High, Q is always a logic 1 (the OR function). With the register (FDRSE) now configured as a synchronous set/reset operation, the set is now free to be used as part of the synchronous datapath. Result in a faster design due to the reduction of logic levels for practically every signal creating this function. Because most of the logic in a design is synchronous, using synchronous or no reset at all allows for further design optimizations, reduced area, and optimal performance.

1. General Use of Registers

FPGA architectures have one register for every LUT, with additional registers available in I/Os and dedicated blocks, such as memory and DSP elements. Using these resources is important to achieve maximum performance. Registers can serve multiple purposes for accelerating design performance. They can be used to reduce the number of logic levels in critical paths, critical nets fanout, set-up, and clock-to-out of I/Os or dedicated blocks.

Approaches:

It is, therefore, important to consider:

• the timing of logic driving or being driven by dedicated blocks
• the positioning of registers in the HDL code

Tips

• When inferring dedicated blocks, synthesis constraints should be applied if the synthesis tool does not enable the best set of registers for performance. (Instantiation can also be considered).

• When selecting which register to enable for instantiated components, ensure the maximum number of registers are used and take into account delays to, from, and between registers within the block.

2. Use of I/O Registers

All the FPGAs contain dedicated registers on the FPGA input and output paths. By utilizing these registers, set-up times for the input paths and clock-to-out times for the output paths can be minimized.

Approaches:

If the synthesis tool does not support automatic placement or if manual control of register placement is desired, the following steps must be performed:

1. Disable any global I/O register placement options for the synthesis tool (refer to synthesis tool documentation).

2. Specify whether the register should be placed into the I/O by adding an IOB=TRUE in the UCF file or source HDL code (refer to the Constraints Guide for details on the IOB constraint).

3. Disable the Map option "Pack I/O Registers/Latches into IOBs" in ISE Project Navigator (or do not use the -pr switch if running from command-line). This disables automatic pushing of registers into the I/O.

Another notable guideline is to describe registers on all input and output ports to the FPGA on the top-level of the HDL code. Specifying the registers on the top-level hierarchy of the code avoids placement conflicts when using hierarchical design methods for implementing the FPGA.

4. Nested If-Then-Else, Case Statements, and Combinatorial For-Loops

Too many if statements inside of other if statements make the line length too long and can inhibit synthesis optimizations. When describing for-loops in HDL, it is preferable to place at least a register in the datapath. During compilation, the synthesis tool unrolls the loop. Without these synchronous elements, the synthesis tool concatenates the logic created at each iteration

of the loop, potentially resulting in very long combinatorial paths and limiting design performance.

Step 9: Add levels of Pipeline

To increase performance is to restructure long datapaths with several levels of logic and distribute them over multiple clock cycles. This method allows for a faster clock cycle and increased data throughput at the expense of latency and pipeline overhead logic management. With this technique, the datapath spans multiple cycles; therefore, special considerations must be used for the rest of the design to account for the added path latency.

Approach:

Improve design performance by balancing the number of logic levels between registers. Add levels of pipeline in the RTL code, apply the synthesis tool's retiming option whenever relevant, or both.

Step 10: Examine the coding style especially for FSM

Following are some tips and guidelines for better coding style of HDL FSM

1. Each FSM design should be coded separately as a verilog module

• It is easier to maintain the FSM code if each FSM is a separate module, plus third-party FSM optimization tools work best on isolated and self-contained FSM designs.

2. For verilog constant, use parameters instead of global `define macro

• After parameter definitions are created, the symbolic parameter names are used throughout the rest of the design, not the state encodings.

3. To define state encoding, use parameters instead of global `define macro

• Parameters are constants that are local to a module and whenever a constant is added to a design, one should think "use parameters" first, and only use a global macro definition if the macro is truly needed by multiple modules or the entire design.

4. Make state and next (state) declarations right after the parameter assignments.

• Some FSM optimization synthesis tool requires state parameter assignments to be declared before making the state and next state declaration.

5. All sequential logic should be define using non-blocking assignment

6. All combinational logic should be define using blocking assignment

• These two[5],[6] guidelines help to code a design that will not be vulnerable to Verilog simulation race conditions.

FPGA Debug Methodologies

During the FPGA Debug Phase, we need to find the hard problems that were not caught by simulation. Doing this in a time-effective way is the challenge.

There are two basic in-circuit FPGA debug methodologies:

The first approach involves inserting a core into the FPGA design and using internal FPGA memory for trace storage and trace capture via JTAG. The second requires routing of internal signals to debug pins and using a logic analyzer or mixed signal oscilloscope to capture traces.

Approach 1:

FPGA vendors offer embedded logic analyzer cores (e.g. Chipscope from Xilinx or Signal Tap from Altera) which can be easily ported into the FPGA along with the SoC. These cores provide triggering and storage capabilities. FPGA resources are used to implement the trigger circuit and FPGA internal block RAMs are used to implement the storage capability.

Once the core is inserted with all information like trigger setup and data capture setup with the help of inserter, JTAG is used to configure the operation of the core. Also JTAG is used to pass the captured data to attached PC for analysis and debug purposes. On PC, the captured data and signals can be viewed using analyzer software. This analyzer actually reads the information stored into the FPGA memory through JTAG chain attached to the FPGA.

Some trade-off with embedded logic analyzer -

1. It uses internal FPGA logic resources and memory blocks that could be used to implement the design.

2. The embedded logic analyzer gives visibility into the operation inside the FPGA, it do not have a way of correlating that information to board-level or system level information.

3. Embedded logic analyzer cores offer less functionality than a full-function logic analyzer – functionality that is often needed to capture and analyzer your tough debug challenges.

Approach 2:

Because of some of the limitations of the embedded logic analyzer methodology, many FPGA designers have adopted a methodology that uses the flexibility of the FPGA and the power of an external mixed signal oscilloscope. Internal signals of interest are routed to unused pins of the FPGA, which are then connected to the external test equipment. This approach leverages the very deep acquisition memory in the external test equipment, which is useful when debugging problems where the symptom and the actual cause are separated by a large amount of time. It also offers the ability to correlate the internal FPGA signals with other activity in the system.

Conclusion

This paper covered all aspects of RTL migration for SoC on FPGA and their efficient approaches. These aspects describes all RTL database including replacement of hard macros, standard cell components, various memory types and sizes, DSP blocks, reset logics, clocks including gated or derived clock implementation, various HDL design practices. This paper also describes some FPGA debug strategies and techniques which normally followed by design engineer at the time of FPGA debugging stage. This paper can help in targeting SoC prototype on FPGA platform from shortening complete development cycle, fast debug time and time to market.

References

1. Xilinx White Paper: HDL Coding Practices to Accelerate Design Performance

2. (i) Verilog coding style and papers from Sunburst Design (ii) FSM design practices from Sunburst Design, http://www.sunburst-design.com/papers/

3. Vijay Kumar Kodavalla and Nitin Raverkar, “FPGA prototyping of complex SoCs: Partitioning and Timing Closure Challenges with Solutions”, IPSOC 2005.

4. King Ou, “Using ASIC Prototyping to Reduce Risks”, SNUG, San Jose 2005.

About the author:

Sachin Gorkhe is Member, Technical Staff at eInfochips Bangalore Ltd. and can be reached at sachin.gorkhe@einfochips.com

Related URL:

Combining prototyping solutions to solve hardware software integration challenges