Advanced FPGA Design Methodologies with Xilinx Vivado

Alexander Jäger
Computer Architecture Group
Heidelberg University, Germany

***Abstract*—With shrinking feature sizes in the ASIC manufacturing technology, modern FPGAs get bigger and bigger, containing millions of logic elements. This leads to the ability to implement designs with synthesis and implementation runtimes of several hours. Even a small RTL change like the change of a reset value of a register results in the requirement makes it necessary to resynthesize and to reimplement the entire design.**

The Incremental Compile feature of the Xilinx Vivado Design Suite promises to halve the implementation time, if 95% of original design can be reused. This paper examines the Incremental Compile feature and its impact on the overall FPGA turnaround time. A second field of interest is the impact on timing and device resource utilization.

The paper starts with an introduction to reasons why and where FPGAs are use. Then the Basic FPGA Design Flow will be explained. Afterwards the Design Flow used by Xilinx Vivado will be shown. Then comes the main part explaining Incremental Compile and the test methodology and then presenting the test results. The paper closes with a short conclusion.

Keywords*—FPGA; Design Flow; Incremental Compile; Vivado; Xilinx

I. Introduction – Why and where to use FPGAs

In the year 2005 the FPGA market had a value of more than two billion dollars with more than 100 million units shipped. To 2012 the market rose to more than four billion dollars with more than 200 million units shipped. [1] So the market has roughly doubled in only seven years.

The question is where and why FPGAs are used. Two fields can be found:

1. Digital systems with small to medium quantities.
2. Prototyping of digital systems for evaluation and verification.

Several reasons can be determined why FPGAs suit best for the mentioned purposes. First of all, all possible digital functions can be implemented. The result is, that in principle every digital ASIC can be emulated in an FPGA. Next important thing is, that they are user programmable. Therefore the user himself can program them and the need for very expensive mask sets, which must be used in ASIC manufacturing, is no longer required. So without the need of expensive mask sets the developers are not bound on timelimes from ASIC manufacturers. The design has not to be ready at a specific date and the developers don't have to wait for months until their ASIC is coming from production. This results in a shorter time to market. Next thing is, that an ASIC can't be changed after production. The developers have to do a respin, which is money and time consuming. In opposite of this an FPGA implementation can easily changed when a bug is found or if the requirements have changed. But an FPGA also has disadvantages. In comparison with an ASIC , the FPGA is slower and needs more power. In high volume production the price per chip is relatively high, causing that an ASIC is cheaper at all. Hence for high volume production or when high frequencies are needed, then an ASIC must be used. Because of the fact, that FPGAs in general do not have flexible analogue elements, a mixed-signal ASIC can only be substituted with an FPGA and a separate analogue part. FPGAs are used for prototyping for following reasons. As said before a mask set is very expensive, an FPGA in comparison to a mask set costs is much cheaper. Furthermore simulation and verification runs are relatively slow. FPGAs can speedup simulation and verification with a functional identical implementation with regard to the later ASIC.

II. Basic FPGA Design Flow

Figure 1 shows the basic FPGA design flow. During the Design flow timing and implementation constraints are used. Constraints are properties of the design, which can't be derived from the tools themselves. They are needed to give the tools information of specific design requirements. Timing constraints tell the tool at which frequency the circuit should run, the frequency of used

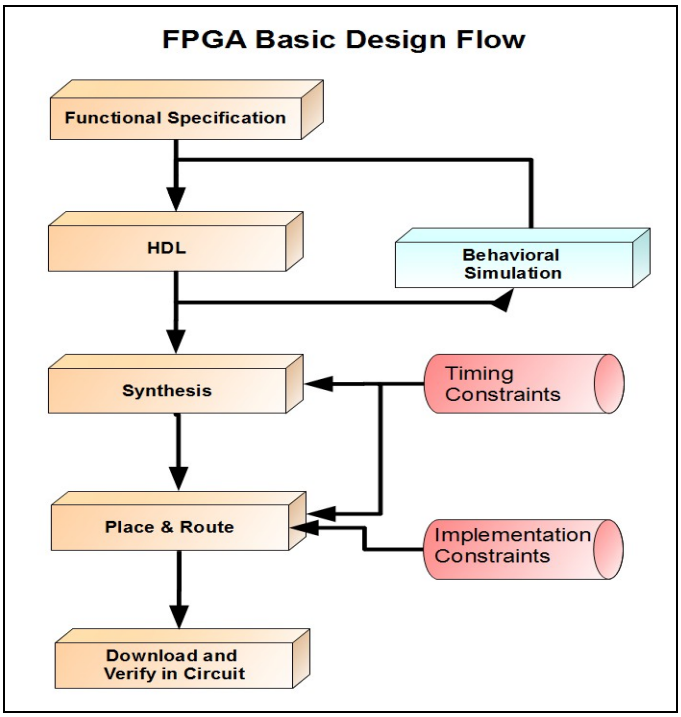

Figure 1 – Basic FPGA Design Flow

clocks, information about clock domains, signal runtimes on the pcb and so on. Implementation constraints are also called physical constraints. They tell the tool for example, which package pins it has to use and which not, which IO-standard is used and some more. Examples for timing constraints are the used clocks with their characteristics like frequencies, rise and fall times or the input delay of the input pins. An example for design constraints is the specification of an input or output pin to a package pin.

As shown in figure 1 the basic design flow consists of the following steps:

1. Functional Specification
2. HDL
3. Synthesis
4. Place & Route
5. Download and Verify in Circuit

The first step is the Functional Specification. In this step the target functions of the hardware are specified. The system hierarchy and the place of the FPGA design in this hierarchy are specified. Other specifications are inputs and outputs and interface descriptions or the desired frequency. For the FPGA design itself a design hierarchy can be decomposed. The result of this step is a complete specification of the desired system. It's a description without implementation details and acts like a roadmap for following steps in the design flow. Mistakes or imprecise specifications in this step can cause very big problems in the later steps. The sooner a mistake in the design flow is found the easier is its correction.

Second step is HDL. HDL stands for Hardware description language. In this step a description of the hardware on register-transfer-level is created. This description of the hardware can be simulated through behavioral simulation. For doing this simulation a testbench with some testcases is created. In the created testbench the hardware description is instantiated and fed with the stimulus from the testcases. The designer can look if the hardware description works as desired.

Third step is the Synthesis. In the Synthesis the hardware description is translated from register-transfer-level to a gate-level-netlist. This means the abstract hardware description is mapped into digital primitives, like and-gates, or-gates and multiplexer, and their connections between them. The Synthesis uses the given timing constraints for optimizing the gate-level-netlist in order to fulfill the timing requirements.

Fourth step is Place and Route. Place and Route can be divided into two steps. First placing and second routing will be done. But this is normally done in one big step. During Placement the primitives from the gate-level-netlist are placed onto the available digital resources of the FPGA. The placement algorithm tries to put the elements, which belong together, close to each other in order to fulfill the timing requirements. After the Placement Routing is done, which connects the elements according to the gate-level-netlist. Place & Route additionally uses the implementation constraints to fulfill the physical requirements. Additional minor steps during Place & Route are DRC checks , timing checks and some more optimizations.

The last step is Download and Verify in Circuit. At first a bitfile, which contains the FPGA configuration, is generated. This bitfile is programmed into the FPGA. Now it can be verified if the FPGA interacts with the other hardware on the PCB as specified.

III. Xilinx Vivado Design Flow

Figure 2 shows the Vivado design flow with additional options. High-Level-Synthesis, DSP Design and the use of Intellectual Property (IP) are optional in the design flow. As this options are not relevant for the further understanding, they won't be explained here.

Starting from RTL (Register-transfer-level) System-Level-Integration the Vivado Design Flow is very similar to the basic design flow explained before. Missing in the Vivado Design Flow is Step 1, the Functional Specification. This isn't covered in Vivado and has to be done before starting with RTL System-Level-Integration. Completed with the missing first step, the Vivado Design Flow consists of the following steps:

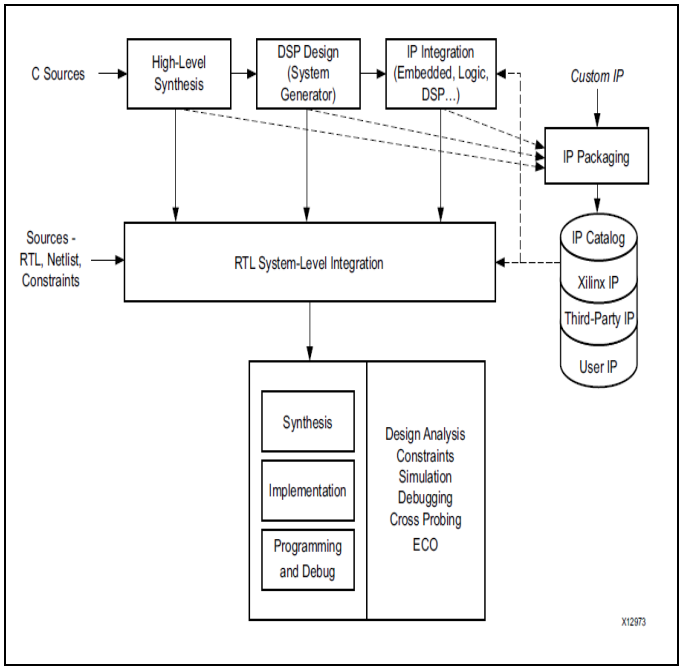

Figure 2 – Vivado Design Flow [2]

1. Functional Specification
2. RTL System-Level Integration
3. Synthesis
4. Implementation
5. Programming and Debug

RTL System-Level Integration is equivalent to HDL, Implementation is equivalent to Place & Route and Programming and Debug equivalent to Download and Verify in Circuit.

In Vivado the entire design flow, except the Functional Specification, can be done in one Application without the need of using different tools for different steps. Additional the Flow Navigator in Vivado, shown in figure 3, makes it possible to start most of the steps only by a simple click.

IV. Incremental Compile

Incremental Compile is an advanced option of the Xilinx Vivado Design Suite. In theory it can be used to reduce the runtime of the implementation step in the design flow. The requirement is an only small RTL or netlist change from an already implemented reference design. When Incremental Compile is used, Vivado Design Suite determines which already placed cells and nets from the reference design can be reused, because they are identical in the reference design and the changed design. Then only the cells and nets, which can't be reused and the new one's will be placed and routed and

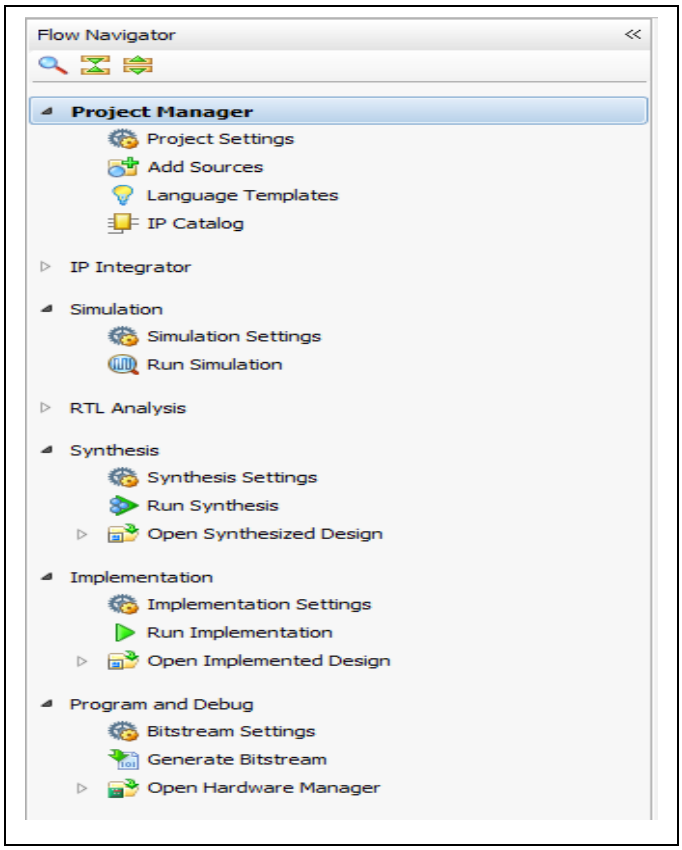

Figure 3 – Flow Navigator

the others will be taken from the reference design. The more matching cells and routes can be found and reused, the better is the speedup through Incremental Compile. Xilinx promises a runtime reduction of 50% if 95% of the cells and nets can be reused. If less then 85% of the already placed cells and nets can be reused, Incremental Compile won't be run, because no more runtime reduction can be achieved in this case. Figure 4 shows the Incremental Compile Flow.

Incremental Compile can be used as follows. The first step is to implement a reference design with the basic design flow. In this step Incremental Compile can't be used and no speedup is available. The second step is to create a new synthesis and implementation run. For using Incremental Compile the Incremental Compile Property must be set by choosing a design checkpoint of the reference design. Checkpoints are automatically saved in Vivado and no manual user action is required. Checkpoints with complete placed and routed designs, only placed design or partially placed and routed designs can be chosen. Next step is to make the required changes to the design. Last step is starting the implementation as normal. Because of the Incremental Compile Property Vivado will run the Incremental Compile Flow. The netlist will be synthesized as normal and no runtime reduction is provided. During Implementation Incremental Compile is used.

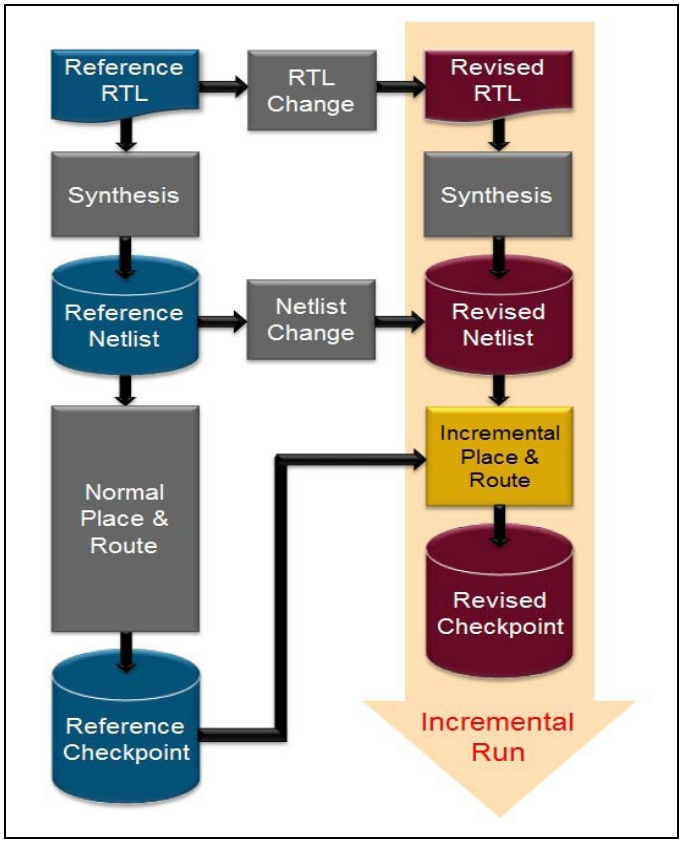

Figure 4 – Incremental Compile Flow [3]

In theory good runtime reductions can be achieved by using Incremental Compile. So the aim is to determine the real impact on the runtimes of the implementation. Secondary aim is to look if there is an impact on timing and used resources. To achieve this a real project was used as test design. This design was originally implemented on a Virtex-7 XC7VX690T device. The design is used in two principal configurations. First an implementation with a logic analyser IP core from Xilinx and second the same without the logic analyser core. Basic operating numbers of the designs are:

1. Design 1:
   - Worst Negative Slack: 0.102 ns
   - Worst Hold Slack: 0.057 ns
   - Slice LUTs: 22106
   - Slice Registers: 23592
   - LUT Flip Flop Pairs: 26642
   - Block Ram Tile: 252
2. Design 2:
   - Worst Negative Slack: 0.102 ns
   - Worst Hold Slack: 0.052 ns
   - Slice LUTs: 18392
   - Slice Registers: 16816
   - LUT Flip Flop Pairs: 21392
   - Block Ram Tile: 16

The following test methodology was used. First a list of test cases for the Incremental Compile was created. First small design changes were added. Examples for small design changes are changing of the reset value of a register or changing of a pin location constraint. As the design contains two fifos with the initial count of 8 entries, a variation of the entry count was used to create medium design changes. To complete the list with a big design change, an incremental compile run with the logic analyzer based on the second design without the logic analyzer core was added. The completed list looks as follows:

- change a register reset value
- alternate a package pin constraint
- replace an AND-gate with an OR-gate
- change fifo entry count to 32
- change fifo entry count to 64
- change fifo entry count to 96
- change fifo entry count to 128
- change fifo entry count to 256
- change fifo entry count to 512
- add logic analyzer core

At next each entry on the list has been implemented twice, first without Incremental Compile and second with Incremental Compile using a design checkpoint of the first design where Place and Route had been completed, except the logic analyzer case, where the second design was used.

At last for every test case the two implementation runs were compared for runtime , timing requirements and resource utilization.

As hardware for running the tests a notebook with a AMD A8-3500M CPU, 8 GB DDR3 RAM and Windows 7 was used. Design 1 had a implementation runtime of about 31 minutes, Deesign 2 of about 24 minutes.

V. Test Results

Figure 5 and Table 1 show the runtime differences between the Standard Implementation Flow and the Incremental Compile Flow. On the X-axis the percentage of the reused cells is plotted and on the Y-axis the belonging runtimes as percentage of the runtime of the standard implementation. For existing runs with a nearly

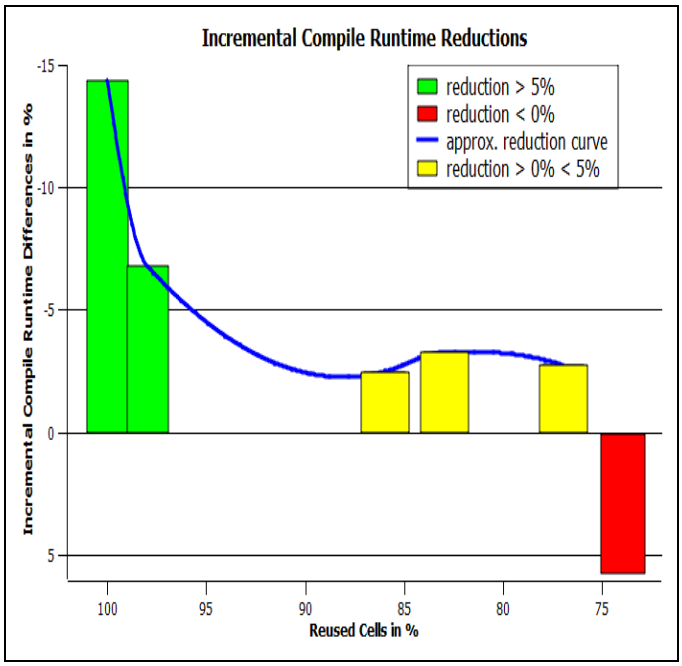

Figure 5 – Runtime Differences

Table 1 – Runtimes

Reused Cells in %	Runtime in %	Number of Runs
99,99	85,64	3
98	93,2	2
86	97,53	2
83	96,74	1
77	97,25	1
74	105,74	1

identical reuse percentage the arithmetic average is plotted.

As expected the runtime improvement is best when almost the entire design can be reused. The runtime improvement is dropping very fast. By reusing 99.9% of the design a runtime improvement of about 15 % can be achieved, by reusing 98% of the design the improvement falls under 7%. It's obvious that values between 98% and 86% are missing. The cause are not missing test cases as it seems. Instead a testcase with a expecting reuse percentage of about 94 – 96% also had a reuse percentage of about 98% and the next testcase with expected reuse of about 89 – 92 % had a reuse percentage of 86%. In the reuse region between 77% and 86% the runtime improvement amounts to about 3%. The runtime improvement is better with 83% than with 86% reuse. But the difference is only a half percent and in absolute values only some seconds. Statistically the difference can be ignored. So what's left is a linear region, where the runtime improvement drifts around 3%. If the reuse percentage drops further, Incremental Compile can't be performed any more and the runtimes get higher to about 6% longer than without Incremental Compile. This 6% is the time the algorithm needs to determine similarities and differences between the reused netlist and the new one and to decide not to run Incremental Compile. The maximal achieved runtime improvement was 21.22 %. In that run only a port constraint was changed.

Figure 6 shows the runtime improvements with subtracted Incremental Compile overhead. The Incremental Compile Overhead is mainly the time Vivado needs to compare the reference netlist with new one and to determine which cells and net and so on can be reused. This makes sense, because the used design had a implementation runtime of about 30 minutes and the impact of the overhead distorts the runtime improvements for big designs with runtimes of several hours. In this curve can be seen that the maximum runtime improvement is about 20% and with 98% reused cells the improvement after all is about 12.5%. This means a runtime reduction of about 45 minutes with a design, which has a runtime of four hours.

Timing and resource utilization showed some surprises. Every timing constraint in every run was met. This means no negative impact on the timing was measured. Looking at the utilization shows no difference with used Block RAM and used Slice Registers. The first differences can be seen in used Slice LUTs in every second run, but the differences are very small, always smaller than 0.2%. Surprisingly the used LUT count drops in most cases. The really surprise comes looking at used LUT-FlipFlop Pairs and used Slices. In more than the half test runs the utilization drops and not so

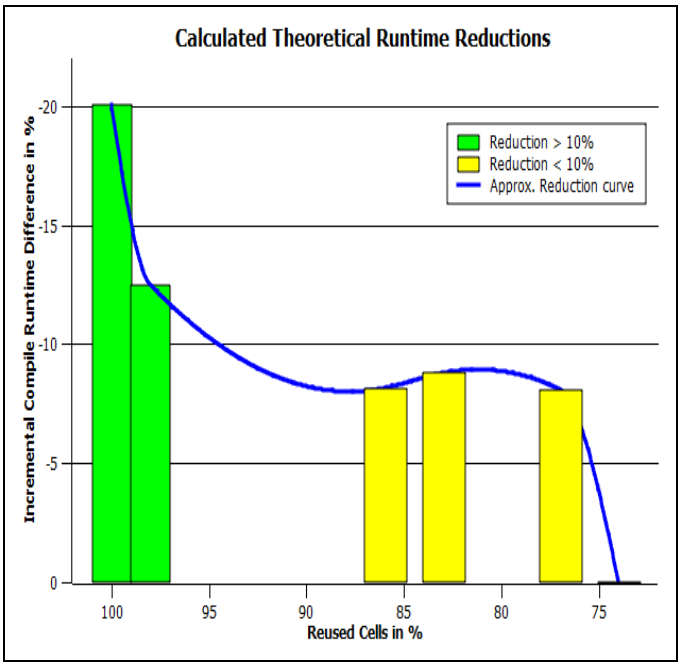

Figure 6 – Theoretical Runtime Reductions

minimalistic as the Slice LUT usage. The utilization drops in most cases more than 2% and in the best case more than 3%. This is very surprising as a rising utilization would be expected. The cause is not clear and the results may vary for different designs.

VI. Conclusion

The Vivado Design Suite automatically creates design checkpoints. The usage of Incremental Compile is very easy, because this checkpoints can be used. The only thing to do is to create a new implementation run and set the Incremental Compile flag with one of the checkpoints. No negative impact on timing and resource utilization could be measured. Exactly the opposite could be observed. On timing no impact at all could be observed and the resource utilization got even better in many cases. If this phenomena is design dependent is not clear. The runtime improvements are far away of the Xilinx promises, but can be measured steadily. The runtime reductions start from about 3% and have their maximum at about 21%. A better reuse quota leads in general to a shorter runtime. The measured resource utilization and runtime reductions and also the ease of use indicate, that Incremental Compile should be used, where applicable.

Not answered is how different and larger designs have an impact on the measured key figures.

This indicates the starting points for future work. Implementation runs with different designs are the nearest option. This would show the impact of different designs on runtimes, timing requirements and resource utilization. Implementation runs with very large design can be made, too see if Incremental Compile really scales like calculated.

Acknowledgment

Thanks to my mentor Juri Schmidt who answered all of my questions, even in his holidays.

References

[1] Joanne Itow – Programming the Future http://semiengineering.com/programming-future/ , March 21st, 2013 (9.1.2015)

[2] Xilinx Inc. - Vivado Design SuiteUser Guide - Design Flows Overview, 1.10.14, S.6

[3] Xilinx Inc. - Vivado Design Suite User Guide - Implementation, 15.10.14, S.83