A3. High Level Synthesis for FPGAs: From Prototyping to Development

1. HLS tools are the tools which aid in the process of High Level Synthesis, i.e., it provides advancements in interpreting an algorithmic description of a desired behavior and creates digital hardware that implements that behavior.

The increasing System on chip design complexity is pushing the design community to raise the level of abstraction beyond the register transfer level. The latest generation of HLS tools have been introduced to provide a significant progress in wide language coverage and robust compilation technology, in providing platform-based modeling, advancements in core HLS algorithms and a domain-specific approach.

A growing number of FPGA designs are produced using HLS tools because of the following reasons: -

* HLS flow allows to experiment with different hardware/software boundaries and explore area/power/performance tradeoffs from a single common functional specification.
* When the HLS tools makes use of high-level specification in C, C++, or System C it results in reduction of code density and ultimately results in a much reduced design complexity.
* HLS tools make use of behavioral IP reuse that improves the design productivity.
* The SystemC based HLS solutions that can automatically generate RTL code avoid slow and error-prone manual RTL re-coding.
* For FPGAs, the HLS tools provide a wide simulation coverage and design iterations can be done quickly and inexpensively without huge manufacturing costs.
* The modern HLS tools allow the application of platform-based design methodology and achieve higher quality of results.

1. The HLS tool flows are in many ways different from the flows discussed in class. The two examples are control flow graph and data flow graph. The HLS flow allows designers to specify design functionality in high-level programming languages such as c/c++ for both embedded software and customized hardware logic on SoC. This way they can quickly experiment with different hardware/software boundaries and explore various areas or power or performance tradeoffs from a single common functional specification.

The HLS tools are based on AutoPilot which accepts c, c++ or systemc as input. It then performs platform-based code transformations and optimizations to generate the RTL. In class we have not used we have not taken such inputs and not verified the RTL output correctness in that manner.

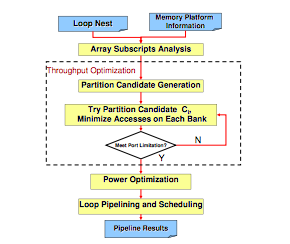
c) In order to evaluate the success or quality of HLS tools there are certain design goals for a HLS tool to fulfil, which are as follows: -

* The HLS tool should be such that the source code could be readable by algorithm specialists. The tool should be able to capture designs at a bit-accurate, algorithmic level. In simple words the HLS tools should be able to synthesize high level programming languages.
* For parallelizable algorithms, the HLS tool should be able to generate efficient parallel architectures with minimal modification to the source code.
* The HLS tools should provide the designer the capability to improve the quality of results by successive code modification, refactoring and refinement on synthesis options.
* Generate implementations that are competitive with synthesizable RTL designs after automatic and manual optimization.

The results which the author highlights in section IX are: -

* Summary of BDTI HLS Certification
* This contains the comparison of the results of an HLS tool and the Xilinx Spartan 3 FPGA that is part of the Video Starter kit, with the result of a conventional DSP processor and with the results of a good manual RTL implementation for an optical flow algorithm and a wireless application.
* Comparing comparably priced consumer-grade FPGA and DSP targets for the optical flow algorithm; the AutoPilot achieved approx. 30 times better throughput per dollar than optimized DSP implementation.
* For the “extent of modifications to source code” necessary to implement optical flow algorithm, the DSP processor implementation was fair while it was rated good for the AutoPilot.
* For the wireless application (DQPSK), for the QoRs the AutoPilot implementation received slightly lower resource usage than RTL.
* BDTI in the end concluded that the DSP tool flow was still significantly easier to use primarily due to difficulties installing the FPGA tools and a lack of sufficient platform infrastructure.
* Sphere Decoder
* Using the AutoPilot version 2010.07.ft, they were able to generate a design that was smaller than the reference implementation in less time than a hard RTL implementation.
* Exactly same architecture was implemented and AutoPilot after only three weeks had met timing and throughput goals but required more logic resources than the RTL design.
* In both RTL and AutoPilot design approaches, the 4x4 case was implemented first, and the 3x3 and 2x2 cases were derived from the 4x4 case.
* In AutoPilot, 2x2 case it uses additional BRAM but for the 4x4 case the RTL designer would have to do a complete redesign. Considering the overall design, however they were able to reduce the BRAM usage.

d) The section VI describes the recent algorithmic advancement in HLS. One of the major HLS algorithmic advancement is Memory Analysis and Optimizations. Hardware acceleration is crucial in modern embedded system design to meet the explosive demands on performance and cost. Selected computation kernels for acceleration are usually captured by nest loops, which are optimized by state-of-the-art techniques. These techniques are like loop tiling and loop pipelining. However, memory bandwidth bottlenecks prevent designs to reach optimal throughput with respect to available parallelism. This idea integrates front-end transformations and operation scheduling in an iterative algorithm. A probable method would be partitioning memory blocks to enable the access which will be simultaneous to different elements of the array. As shown in the Fig4 of the provided paper, an array A is taken that has subscripts i,2xi+1, and 3xi+1 in the ith iteration as shown in the for loop. Partitioning into two banks, the first contains elements with even indices and second contains odd. A new loop iteration begins as shown in the fig4 (b), which may lead to port conflicts. However, the 4(c) could give two simultaneous accesses. There is a memory portioning algorithm which is explained in fig below.



source: J. Cong, W. Jiang, B. Liu, and Y. Zou, “Automatic memory partitioning and scheduling for throughput and power optimization,” in *Proc.* *ICCAD*, Nov. 2009, pp. 697–704.

The first step would contain the problem formulation in which the iteration space I of a depth-l loop nest is defined in the l-dimensional space, where each iteration corresponds to an integer vector identified by its index values I= (i1,i2,….il). This process is flowed by the partitioning algorithm as shown in the figure above. Computational kernels for speedup are identified by profiling and provided as the input to the memory portioning algorithm. The first part is to find good partition candidates based on array references and memory platform information; after candidates are generated a branch and bound algorithm will search the best combination of partitions on different array dimensions to meet the port limitation and minimize the partition cost. This is followed by the handling of irregular array access. This is followed by the power optimization in which the goal is to group frequently visited elements into small memory banks, because average power consumption decreases with memory size.