Implementation of the ATI Flipper Chip

Anand Mandapati
ATI Research Inc.
Santa Clara, CA, USA

Abstract

The Nintendo GameCube(tm) video game console system is designed to outpace all other such systems when released. Formerly known by the codename Dolphin, this system includes an IBM PowerPC(tm) processor and specialized hardware from ATI. This specialized hardware is embodied in ATI's Flipper chip, the centerpiece in the Dolphin design. Flipper functions as the graphics processor, audio processor, host controller, memory controller, and I/O processor of the Dolphin system. Such a complex chip requires a very robust design flow to get to functioning silicon in as little time as possible. Here we will describe that design flow, developed by ATI engineers to implement the Flipper design. The goal was to develop a flow to implement the best gaming hardware on a chip that needed to be as cost-effective as possible. There were many challenges the design offered, requiring optimal use of a small design team with a minimal budget to achieve aggressive schedules. The biggest challenge the team was presented was that of area. With high volumes, chips for consumer devices can benefit greatly from smaller die sizes, due in part to higher yields and also in part to lower power and cheaper packages. Another daunting challenge the design offered was that of the use of embedded DRAM. The Dolphin architecture called for the use of an embedded frame buffer and texture memory buffer for fast access. These embedded DRAM were naturally very large in size and presented interesting physical problems. During the architecture definition and RTL coding stages, the design team made sure to consider the implications on the physical implementation of the chip. This was evident in the lack of false and multi-cycle timing paths in the design. It was also evident in the lack of critical timing paths between blocks due to the use of registers at most interfaces. The ATI design team realized from experience that it is easier to solve these problems early in the design cycle. Once the first synthesized chip netlist was ready, we proceeded to floorplan the chip. Due to our extreme necessity to reduce area, we decided that we would not use a traditional hierarchical floorplanning methodology but instead one without any top-level cells or routing, with each block abutting to another. This tile-based approach offered many benefits in addition to reduced area. By not having any cells or routing at the top-level, we were able to reduce, if not remove, the need for running full-chip RC extraction, LVS, DRC, and antenna checks. The tile-based approach also presented a number of problems that existing floorplanning tools could not handle at the time and so we decided to write our own tool. Among the problems was how pins, feedthroughs and fanout among the various blocks and tiles were handled. We decided that our floorplanner would also be an integrated netlisting tool that could group portions of the netlist both logically and physically. We also decided, for simplicity's sake, that we would integrate the pad ring into the tiles. This eliminated the need for cutouts or wasted space for IO pads of varying size. The pad ring itself was created using internally written tools that created from an ordered list not only the physical location of each IO cell and bond pad but also the RTL for each IO cell as well as the RTL associated with each IO cell (e.g., input receiver and output driver flops). We decided not to integrate this associated RTL, what we called near-pad logic, into the actual IO cell so that we would have the flexibility to change the design at a late stage in the schedule. This near-pad logic was placed by the floorplanning toolset in the nearest standard cell row to the IO cell. Other macros, such as SRAM, ROM, and eDRAM macros were placed using a Logo/Turtle style programming language that allowed great flexibility when the overall floorplan changed slightly. Ultimately, the large eDRAM macros, approximately 35% of the core area of the chip, and the pad ring design, defined by board restrictions, forced a single floorplan to be used on the design. Once we had the initial floorplan, we decided to develop a place-and-route flow that had as its main goal a 3-day turnaround time from netlist generation to GDSII. This required a high-level of automation so as to minimize the need for human intervention and maximize the use of computer time. We decided to use point tools, mostly from Cadence, driven by dependency-driven Makefiles to automate the process as much as possible. We developed our own scripting interface to the Cadence tools using Perl. Though the basic flow was mostly the standard Cadence timing-driven flow using QPlace, CTGen, PBOpt, and Warp Route, we developed our own tools for such tasks as pin optimization and scan stitching. For these and other internal tools, we developed a

physical database format, based on the DEF format, which we could use to access a design as quickly as possible. This database had the underlying code written in C that we accessed through Perl using SWIG as our interface generator. We also created a complementary database to represent the various physical and logical views of the library elements. The two databases allowed us to quickly write tools that could process our design data in a fast and efficient manner. During the RTL coding, floorplanning and place-and-route process, we came upon methodologies to handle power grid design, clock tree design (including use of gated clocks for power savings), and various signal integrity problems. Among the signal integrity problems we tackled were crosstalk. Not having access to the still immature crosstalk analysis tools, we decided we would prevent problems by design. We ensured that all our wires had low slew rates, especially on long wires, to reduce the likelihood of coupling. Once done with the place-and-route process, we verified the design hierarchically by extracting at the tile level using Simplex Fire & Ice QX and combining the parasitic information to be used by the timing tools at a full-chip level. We ran timing analysis in more than just the normal best- and worst- case corners by scaling our RC parasitic information to reflect metal etching issues. This provided an extra margining that would not have been taken into account otherwise. For physical verification including LVS, DRC, and antenna, we were able to run most of the checks using Mentor Graphics Calibre only at the tile level and do rudimentary checks at the chip level, thereby saving precious system resources and shortening the schedule. To fix timing problems remaining after the place-and-route process or to fix logic bugs that were found after the place-and-route process, we developed an ECO language that the designers could use. This language, similar in nature to the Synopsys netlist editing capability, allowed the designers to edit the physical and logical netlists at the same time. We also developed tools to analyze the timing reports and fix most common problems including setup, slow node, hold, and gated clock violations automatically. In addition, we developed tools to analyze and fix certain post-route DRC and antenna problems automatically. All these tools were based on our internally developed databases and thus were able to operate on the DEF very efficiently and quickly. For netlist verification, we generated a hierarchical Verilog netlist from the DEF using an internally developed tool, the result of many frustrations with the equivalent Cadence tool. We ran LVS using this netlist and the GDSII. Since the netlist was generated from DEF, this only verified the quality of the Cadence place-and-route tools but did not verify the logical functionality of the netlist. To verify this, we used the Tuxedo-LEC tool from Verplex to guarantee equivalency of the pre- and post-route netlists. We also used the Tuxedo-LEC tool to verify logical ECOs by comparing RTL to synthesized netlist to post-ECO netlist. Finally, we rounded out our netlist verification with full-chip gate-level simulations, including critical scan and reset simulations. For final integration, we wrote a set of our own GDSII processing utilities. This saved us the time, resources, and cost required to use tools like Design Planner or Virtuoso to do the same tasks. The end result of all this work was a chip that taped out on schedule while still meeting the area goals we set for ourselves. We learned through the experience that a well-defined, highly automated design flow is crucial for success. We also learned that it is possible to develop a design flow that successfully unites tools from various EDA vendors with our own internal ones. We also learned that though developing internal tools may be required for certain tasks, we as a design team are not capable of supporting too many of these tools. In the future, we expect to continue developing our own internal tools, but also expect to depend heavily on EDA vendors for our success.