

# White paper: Integration of a data path generation in an ASIC flow

## ABSTRACT

We will show a flow that integrates an efficient datapath generation with an ASIC. In any modern modern SoC which integrates CPU/DSP, layout density, speed of the circuit and power consumption is very important. Traditionally custom design techniques were used to achieve these goals, but the engineering cost using custom design are prohibitive to most companies. This paper will show techniques and flows that allow the design team to achieve very high layout densities while maintaining their ASIC flow. Introduction

Most modern SoC integrate one or more type of embedded controller. In most cases the design team.

The focal point of the chip is the media processor. The DSP and the CPU are dominated by huge data path logic, requiring us to pay special attention to these blocks. In the system blocks, we have over 50 small memories and FIFOs. We wanted to achieve a very high density and high performance for our data path design but we could not afford spending our engineering resources on doing custom design and wanted to keep a uniform ASIC design flow to meet the need of our SoC. We collaborated with Nova to provide design services and EDA flow customization, they designed some of the data path blocks on our chip. Data Path Generation

We used &ldquo;Chipmason&rdquo; from Nova to design and generate the data path (dpGen) and small memory arrays (rfGen). &ldquo;dpGen&rdquo; is used by logic designers to design their datapath elements, using several Build-in functions. DP elements can be as simple as 32bit 2to1 Mux or as complex as a Multiplier array that supports 8, 16 and 32 bit multiplications. &ldquo;dpGen&rdquo; generates a verilog gate level netlist and placement file. It uses any generic standard cell library to generate the desired circuit.

Any datapath element will be part of a bigger circuit that may include other datapath elements or other control logic. Control logic is described using verilog HDL and synthesized. During synthesis flow, the generated DP verilog gate level netlist is read into dc\_shell with &ldquo;dont\_touch&rdquo; special case for Datapath element is memory arrays, like small register files, FIFOs, TAG RAMs etc. The memory arrays can be generated using the standard cell library as basic build blocks.

The approach described in this paper can be compared to custom design approach, where the design team uses custom cells, places and routes the complete block, which in turn is used as a hard macro in the rest of their ASIC flow. On the other hand a comparison can be made to a generic ASIC approach where the designers describe the logic using HDL, synthesize their logic, then they can place and route their logic. The custom design approach may give the best density and speed, but the cost can be prohibitive. The ASIC approach, based on synthesis and PnR tools will work in most cases except in specialized circuits. Our experience, shows us that when the circuit is two or more dimensional (like small multi-ported memory arrays, data cross bars and multipliers especially those that need to support many different formats) the best results can be achieved with structured placement.

Our approach, of using dpGen, allowed us to benefit from the advantages of both approaches with minimal cost. Simplified CPU

The following figure shows a simple block diagram of a generic CPU. The main purpose of the figure is to show the different types of logic in the data path and other control logic surrounding it. The register file &ldquo;RF&rdquo; is very structured, 3 read/2 write port memory array, followed by bypass and forwarding logic, which is a structured data path. In this simple example we have two pipeline registers. We have two execution units &ldquo;A&rdquo; and &ldquo;M&rdquo;. The M unit is very structured logic, while the &ldquo;A&rdquo; has some logic that is easier to describe using HDL and then synthesized. In addition, there are several small control blocks for each of the data path elements and finally some more control logic that can be synthesized.

- Simple data path element When the data path is planned the user needs to decide data flow direction. The following picture shows the placement of several elements of 16 bit wide data path. it lines go east-west and word lines/control lines go north-south

The users start by generating the required logic using Chipmaon's data path generator - dpGen which generates verilog gate level netlist. In addition it generates a placement file which can be a text file that describes the instance name and its placement. Another alternative is to generate a DEF file that describes the placement only without net information. The preferred method is DEF file.

- Register file

The following is an example of a multi port register file build using standard cells, the basic memory cell uses from a standard cell library.

- Pre-placement Layout view

As we can be seen in following picture, the core of this register file is already very dense, but gaps were kept intentionally every 16 bits of the RF. The last stage of the address decoders was pre-placed. The decoder and other control logic were kept as RTL blocks and were synthesized.

- Post Placement Layout view

After running a complete P&R flow which included the RF, the PnR tool preserves the placement of all the original cells (blue) adding several thousand control logic gates (red) and CTS (yellow) and In-Place-Optimization "IPO" (green) buffers.

As with any P&R flow, after initial floorplanning the complete verilog gate netlist including the pre-placed cells, are imported into the P&R tool, then the DEF file is read. Every cell in DEF file had "FIXED" property. The rest of the P&R flow is the same for any other block, "placement" followed by "cts", then "routing"; and two stages of post optimizations.

- Data Path combined with Control logic - case #1 In the following example, part of the logic in this circuit was easier to describe as RTL and herefore was synthesized even though this logic is directly attached to the data flow. The preplaced logic acted as soft region for the random logic.

- Pre-placement Layout viewThe above picture shows several pre placed data path element, empty space is left of some control logic

- Post P&R Layout viewThe picture below shows the same data path element (blue) with the control logic filling in the empty space. In addition the P&R tool added extra buffers for long wires which are placed in the empty spaces.

- Data Path combined with Control logic - case #2 This is a similar example to the previous one except that in addition to the control logic the pre-place logic is located in a central location and therefore the P&R tool needs to buffer a significant number of wires. As can be seen, the random logic and the IPO/PPO buffers were added filling in the gaps. The space between the pre-placed logic can be adjusted slightly - increased or decreased - to achieve the most optimal space utilization. Area utilization achieved here exceeds 98%.

- Pre-placement Layout viewThe above picture shows several pre placed data path element, empty space is left of synthesized control logic.

- Post P&R Layout viewThe picture below shows the same data path element (blue) with the synthesized control logic filling in the empty space.

- Pipeline register - clocks

Pipeline registers form an integral part of every data path. They must be generated and pre-placed, but they pose a different problem. What should be? In our case, some of these pipeline registers reach 256 bits wide, the clock buffers including a clock tree gate ("CTG") were generated as part of the pipeline register and were pre-placed. The "CTG" was pre-placed at the edge of the data path, and a micro clock buffer tree was placed to distribute the load. as can be seen in the picture (left side) These are the pre-placed cells of the pipeline register with the flight lines from the different stages on the local clock buffer tree. The picture on the right shows exactly the same pipeline register after routing with the clock nets highlighted. As can be seen the detail router, routed these clock nets - allowing double spacing - almost in a perfect line.

The top level block may include different data path elements, pre-placed pipeline registers, random control logic and variety of DFF scattered in the design. The flow in this case is the same as before, the full gate level netlist is imported into the place and route tool, then the placement info is imported as DEF with "FIXED" property. The imported netlist includes balanced local clock tree. When the Clock Tree Synthesis ("CTS") is run, it recognizes the pre-placed clock tree cells, it does not touch them while balancing all clock tree leaf points.

Within the same pipeline register, even when we have over 256 bit wide registers and when the width of the data path was wider than 1000u, the clock skew between any of the SDFE was less than 5ps after full extraction.

- Pipeline register -scan chainWhen generating very compact pipeline registers, special care has to be given to their scan titching. In order to start running tetraMax as soon as we have the gate netlist, the scan chains or all the SDFE in the design were pre-stitched, which included a pre-stitched pipeline registers. Before starting the top level routing, the scan chains provided to the PnR tool excluded the pre-stitched pipeline registers and other datapath SDFE allowing us to preserve their scan chain, giving the PnRthe freedom to un-stitch and re-stitch all other SDFE in the design.

Conclusions and recommendations

In summary, a data path generation flow was integrated flawlessly into an ASIC flow (This flow was proven with ASIC flows from Cadence, Synopsys and Magma) achieving very high area utilization (over 95%) and ensuring better timing closure by eliminating the need for timing ECOs.

Another advantage for such approach is the repeatability of results. The generated logic with its placement, remains constant as the design changes.

Finally, we achieved a reduction in the power consumption by allowing the use of CTG and decreasing wire length.