Physical Design of 0.35-µm Gate Arrays for Symmetric Multiprocessing Servers

To meet gate density and system performance requirements for the HP Exemplar S-class and X-class technical servers, a physical design methodology was developed for 1.1-million-raw-basic-cell, 0.35-µm CMOS gate arrays. Commercial and ASIC vendor-supplied tools were augmented with internally developed tools to put together a highly optimized physical chip design process.

by Lionel C. Bening, Tony M. Brewer, Harry D. Foster, Jeffrey S. Quigley, Robert A. Sussman, Paul F. Vogel, and Aaron W. Wells

This article provides a broad overview of the gate design and layout of 0.35-µm gate arrays for the new S-class and X-class members of the Hewlett-Packard Exemplar technical server family. The design process was built around third-party tools, but several internally developed tools were needed because commercial offerings were insufficient or unavailable. Generally, these tools automated the generation of the design or dramatically improved the logistics of the flow. Without these internally developed tools, meeting density and speed objectives (the optimization of the design) would not have been possible in acceptable calendar time with the design staff available.

In-house place-and-route tools played a key role in this design flow. As in previous projects, a close working relationship was developed with the gate array vendor. Among other benefits, this allowed the design staff to use the GARDS placement and routing software from SVR (formerly Silvar Lisco). Most of the internally developed tools written for the project operate directly on place-and-route information. Examples include a floor planner that understands design-specific routing obstructions, custom clock tree generation, and placement-based resynthesis of the gates.

Server Architecture

The target servers are symmetric multiprocessing systems built around the HP PA 8000 processor. S-class machines (also called nodes) connect 16 processors together to form a single system. X-class machines connect multiple nodes together to form a single system. X-class hardware can create a system containing up to 120 nodes (1920 processors). The PA 8000s will initially run at 180 MHz, with the rest of the system running at 120 MHz. Except for the PA 8000 and associated SRAMs and DRAMs, the bulk of the system logic is implemented in Fujitsu CG61 0.35-µm gate arrays, as shown in Table I. One additional gate array is implemented in the much less expensive CG51 0.5-µm process. This chip shared all the same tools and design flow as the 0.35-µm gate arrays.

Table I
Gate Arrays for S-Class and X-Class Servers

Chip Name	All Logic (Basic Cells)	Random Logic (Basic Cells)	Latch Array Bits	Base Type
Processor Interface	550 k	317 k	28 k	0.35 µm
Crossbar	500 k	206 k	29 k	0.35 µm
Memory Interface	570 k	358 k	25 k	0.35 µm
Node-to-Node Interface	300 k	175 k	43 k	0.35 µm
I/O Interface	150 k	106 k	6.4 k	0.5 µm

The characteristics of the Fujitsu CG61 0.35-µm CMOS gate array are as follows:

- 1.1 million raw basic cells. One basic cell can implement one low-power, two-input NOR gate or one medium-power inverter.
- Overall die size: 13.5 by 13.5 mm. Core size: 11.7 by 11.7 mm.
- 4 routing layers.
- 1.75-µm routing pitch on metal layers 1, 2, and 3; 14-µm on metal layer 4.

- 560 I/O signals.
- Flip-chip connection to package. Signals attach at periphery. Power and ground attach on a uniform grid across the die.
- Macros* generally had several functionally equivalent versions with different drive strengths.
- Target 45% utilization of raw basic cells in random logic regions.
- Many paths with 14 macros between registers at 120 MHz.

Custom I/O macros were developed to meet the stringent system electrical and timing requirements. The board designers analyzed all system paths with SPICE. SPICE models of the board nets included gate array driver and receiver macros.

The static flow is depicted in Fig. 1. The static flow does not reflect the actual day-to-day activity of the design process. Many iterations and subiterations were made until a chip could flow cleanly without timing or routing problems from start to finish. In addition to feeding back problems into the flow—for example, to update Synopsys constraints or make behavioral model changes—the design staff refined tools and techniques along the way. Rapid iterations through the place-and-route flow were vital to this process.

Typically, four designers worked on the larger arrays. One person worked almost exclusively on the floor plan, layout, and routing. The other designers, in addition to all other design duties, iterated through Synopsys and floor plan timing. Four full-time CAD engineers built and supported the flow.

RTL Behavioral Model Styles

The S-class and X-class server gate arrays were described in RTL-level Verilog (RTL stands for Register Transfer Language). These models were considered to be the "golden" models and were maintained throughout the life of the project. Design verification used the RTL models exclusively. The Boolean equivalence tool (lover) guaranteed that the RTL Verilog and gate Verilog were equivalent. The RTL Verilog style varied depending on the tools and the tasks for which it was written. Thus, the style was driven by the needs of Synopsys, the Boolean equivalence tool, and high-speed RTL Verilog simulation.

Style Driven by Synopsys. Synopsys was the primary tool to map RTL behavioral models to gates. Anyone familiar with this tool knows the requirements it imposes on the coding style: many small modules with lots of hierarchy, instantiated gates in the RTL, and code almost at the equation level. The project had two methods of instantiating gates in the RTL:

- Macro Group Specification. Based on previous projects' difficulties in synthesizing multiplexer macros, all scan registers and multiplexers were explicitly specified in the RTL. To make the RTL more readable and to improve RTL simulation speed, groups of macros were bundled under one level of hierarchy. For instance, 32 high-drive 2-to-1 multiplexer macros would appear in the RTL as:

```
MUX21_HI_32 muxes (
  .S  (<32 bit port connection>),
  .I0 (<32 bit port connection>),
  .I1 (<32 bit port connection>),
  .O  (<32 bit port connection>)
);
```

An internally developed tool read the RTL Verilog and generated the corresponding modules' definitions based on a vendor-specific mapping of the generic type to the actual type.

- 'ifdef GATES. Frequently, designers wanted to specify explicit macro types for some lines of code in a synthesized RTL module. The module would be written with:

```
'ifdef GATES
  <macro instance specifications>
'else
  <behavioral of the above gates>
'endif
```

For example:

```
'ifdef GATES
  wire t1, t2, t3;
  XOR3 xortree1 (t1, in<0>, in<1>, in<2>);
  XOR3 xortree2 (t2, in<3>, in<4>, in<5>);
  XOR3 xortree3 (t3, in<6>, in<7>, in<8>);
  XOR3 xortree4 (perr, t1, t2, t3);
'else
  perr = ^in;
'end
```

* In this context, a macro is a collection of basic cells that implement a primitive function like a multiplexer or register.

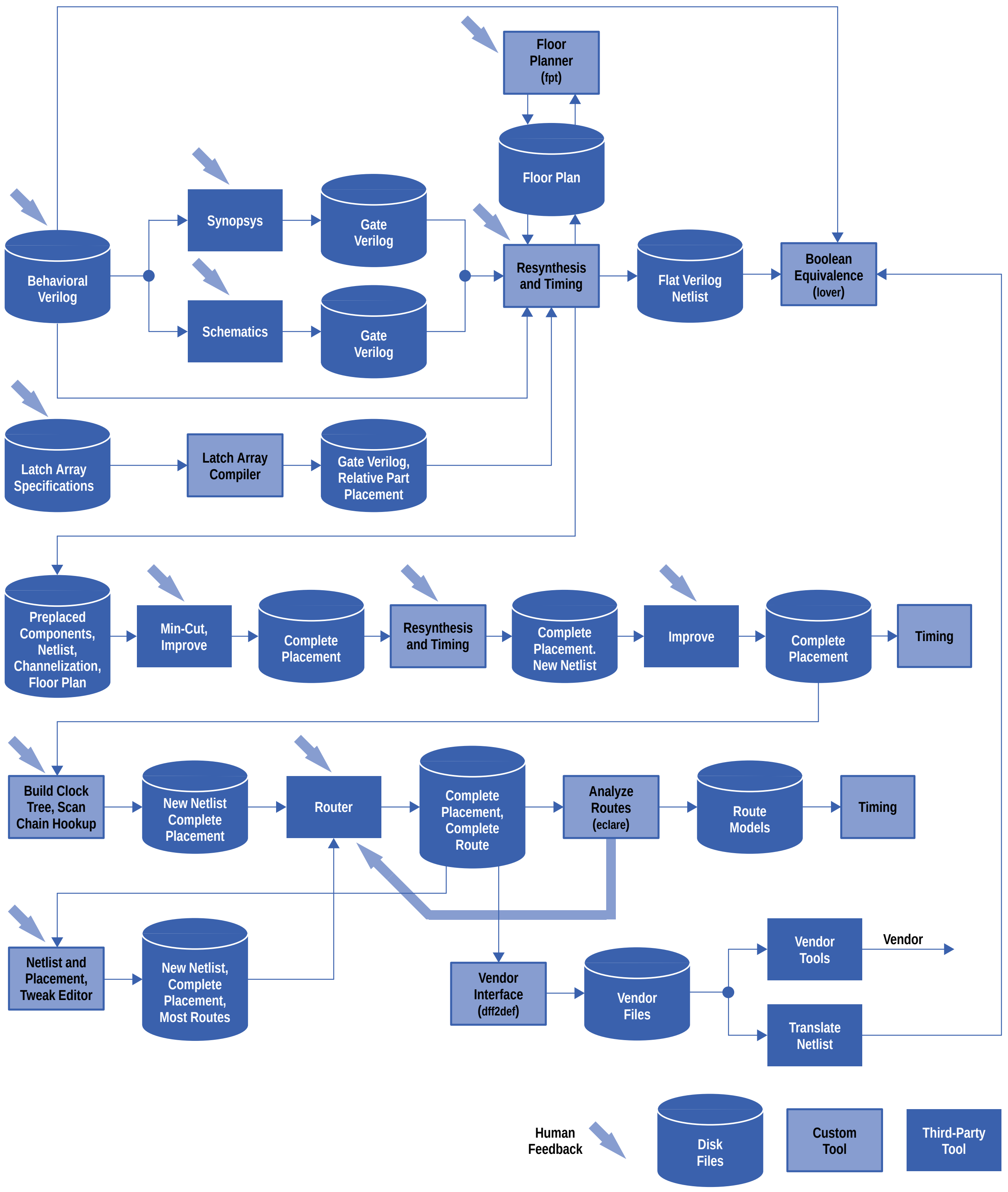


Fig. 1. *Simplified physical design flow.*

The GATES text macro was defined when synthesizing. This preserved clarity and speed of simulation in the RTL while at the same time generating the correct gate design in Synopsys. The Boolean equivalence tool ensured that the gate descriptions matched the behavioral descriptions.

Style Driven by Boolean Equivalence Verification. The Boolean equivalence tool (see “Formal Verification—Boolean Equivalence” below) logically compared the RTL behavioral model and the gate level netlist. This tool partitioned the design into cones of logic based on *equivalence points*, which are essentially net names that exist identically in both models. At a minimum, all register outputs are equivalence points. To keep the comparison tool simple, each equivalence point had to match 1-to-1 between the models. This simplification required that registers replicated for fanout purposes in the gate netlist be replicated identically in the RTL Verilog.

Style Driven by Simulation Performance. Simulation performance was an important factor in each gate array's design quality and its development schedule. Faster simulation meant that we could get to a functionally correct gate array physical design more quickly. Thus, some of the RTL style methodology was driven by simulation performance.

For all of the hand-instantiated miscellaneous logic prefixed by 'ifdef GATES, the designers included the corresponding behavioral statement(s) bracketed with 'else ... 'endif. Also, guidelines were developed that pointed designers towards behavioral Verilog code constructs that simulate faster, and a linting tool was developed that pointed out where they could use these faster constructs in their Verilog. Examples included:

- Use of a mask plus a unary operator in place of the corresponding binary logic operation on selected bits of a bus, as in error correction bit calculations
- Use of a temporary variable in place of frequently referenced bit positions
- Concatenation in place of subrange assignments and shifts by constants.

In addition to simulating with vendor Verilog simulators, an in-house Verilog-to-C translation tool was developed to generate cycle-based models for each gate array type. The resulting simulations ran five to twelve times faster than the same simulations on vendor simulators.[1]

Because the Verilog language is extensive and expensive to support in its entirety, and because the internally developed tools were never intended for wide use beyond the boundaries of the project, the cycle-based Verilog-to-C translation tools supported only a subset of behavioral Verilog that is sufficient for cycle-based simulation. Where there were several different ways of describing the same design in Verilog, the tools supported the style that was prevalent among designers and less expensive to parse. Time constraints forced tool development to leave out constant expressions, loops, parameters, and variables as bit subscripts from the supported behavioral Verilog language style.

Latch Array Compiler

The latch array compiler allowed the designers to create new configurations of memory blocks easily. The memory blocks were implemented as a row of latches per data bit, with a column of latches selected by a read or write address. All latch arrays had a dedicated write port and either one or two dedicated read ports. All address, data in, and data out ports of the latch array had registers to isolate the internal timing of the latch array from the surrounding logic. Optionally, parity going into or coming out of the latch array could be checked. The placement locations of the write and read address ports were fixed, but the data input and output ports could be independently placed to the left or right of the core area.

The compiler's output consisted of the RTL behavioral and gate Verilog models for the configurations, as well as the relative placement information. This placement information was then given to the floor planner so that the latch arrays could be relocated to any desired site.

There were 16 to 41 latch arrays per design (118 total for four designs), with each latch array storing between 48 and 2112 bits, for an approximate total of 25K to 43K bits per chip, representing 35% to 60% of the basic cells used for each design. Through the use of custom macros designed specifically for the project, the core area of the latch arrays achieved over 87% utilization. For a single-read-port array, the core area consumed about 5.6 basic cells per bit, while the core area for a dual-read-port array used about 9.6 basic cells per bit. The core of the latch array was routed completely in metal 1 and 2, allowing the metal 3 over the core area to be used for normal routing.

Synopsys

Synopsys was used as the primary means to map nonregister and nonmultiplexer RTL to gates. The designers did not rely on Synopsys to generate the final gates. The designer's goal was just to get close to the final timing and logic area and then use the resynthesis step described later to improve the gates. There are two main reasons for this. First, Synopsys cannot read and optimize an entire design in a practical amount of time, and second, Synopsys cannot synthesize fanout trees based on the actual placement of macros.

Constraints for Synopsys were generated by hand.

Formal Verification—Boolean Equivalence

Formal verification using an internally developed Boolean equivalence verification tool (lover) eliminated a tremendous amount of gate level simulation. In fact, the system simulation remained at the RTL level throughout the entire project. Boolean equivalence verification, unlike simulation, guarantees complete results, since it uses mathematical formal proof techniques to verify logical equivalence. lover ran in three fundamental modes: RTL-to-RTL, RTL-to-gate, and gate-to-gate.

RTL-to-RTL comparison was critical when it became necessary to modify modules in the RTL to remove redundancies across module boundaries. Also, RTL-to-RTL comparison allowed the designers to explore different implementations of an equivalent design.

RTL-to-gate comparison took a lot of pain away from doing hand-captured gates. Miscompares could be isolated down to a few gates and fixed quickly. Also, RTL-to-gate comparison was useful as a revision control check by ensuring that any last-minute changes in the RTL source were synthesized into gates, or in other words, that design verification's RTL matched the actual netlist.

Gate-to-gate comparisons ensured that the CAD system (resynthesis in particular) maintained logical correctness throughout the flow. In addition, gate-to-gate comparison was used to ensure that all hand edits performed during the layout process were error-free. This allowed last minute changes to be made in the design with confidence.

Maintaining a consistent Verilog naming convention throughout the CAD flow facilitated the mapping of flat Verilog wire and register names to their hierarchical equivalents. This convention dramatically improved the performance of lover by providing additional *subequivalence points*. Using subequivalence points, large cones of logic whose boundaries consist of ports and registers can be broken up into smaller cones of logic automatically.

A complete gate-to-gate verification of a 550,000-gate design ran on an SPP-1000 (an 8-way symmetric multiprocessing machine with 2G bytes of main memory) in under 20 minutes and required 1.2G bytes of physical memory (see Table II). A complete RTL-to-gate verification of a 550,000-gate design was completed in 1 hour and required 1.2G bytes of memory. Most RTL-to-RTL subhierarchical comparisons completed in under 20 seconds. lover reads and compiles the RTL source files, netlists, and libraries directly. These times represent a combination of the compilation and comparison processes.

Table II
Gate-to-Gate Boolean Equivalence Run-Time Performance and Memory Requirements

Chip Name	Logic Size (Basic Cells)	Minutes	Virtual Memory
Processor Interface	550 k	20	1.2G bytes
Crossbar	500 k	9	0.9G bytes
Memory Interface	570 k	20	1.2G bytes
Node-to-Node Interface	300 k	10	1.0G bytes
I/O Interface	150 k	4	0.3G bytes

Floor Plan

The floor plan tool (fpt) helped the designer generate the placement for the major blocks of logic. The physical hierarchy of the design initially matched the logical hierarchy (a module in the source Verilog became a block for placement), but by editing the placement definition file, the physical hierarchy could be changed to anything the designer desired.

fpt read the same placement obstructions as the gate placement tool, so an accurate determination of the utilization of each block could be made as it was placed. The tool also understood the concept of fixed-size blocks (such as the latch arrays) as opposed to the malleable blocks which were primarily composed of synthesized logic, and allowed the latter block types to be reshaped.

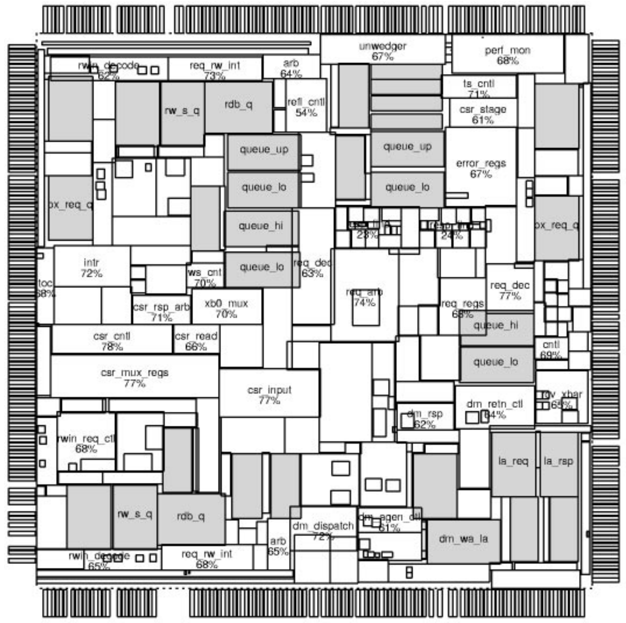
Interconnect lines could be drawn between the block drivers and receivers, with the width of the drawn line representing the number of signals crossing the boundaries of the blocks. Inputs and outputs from a block could be drawn in separate colors to help illustrate the flow through the blocks.

fpt could display a *density map* that quickly identified local hot spots of high placement density. There was also a method to show information from a prior postroute analysis so that the floor plan could be adjusted to provide more room for congested routing areas.

Fig. 2a shows an example of a floor plan for one of the chips and Fig. 2b shows the placement view.

Timing

Timing analysis was performed by an internally developed tool because a third-party tool with the necessary features and performance was not available and resynthesis (see below) required tight coupling of the timing analysis to the automatic netlist editor. Each step in the flow (floor plan, placement, routing) used the same timing algorithm and timing tool. Differences in delays at each step resulted solely from the accuracy of the π-models and each macro input pin's T_{line} (RC delay). (A π-model represents the net + gate load as seen by the macro output.) For different steps in the design flow, the net π-model and RC delays were calculated as follows:



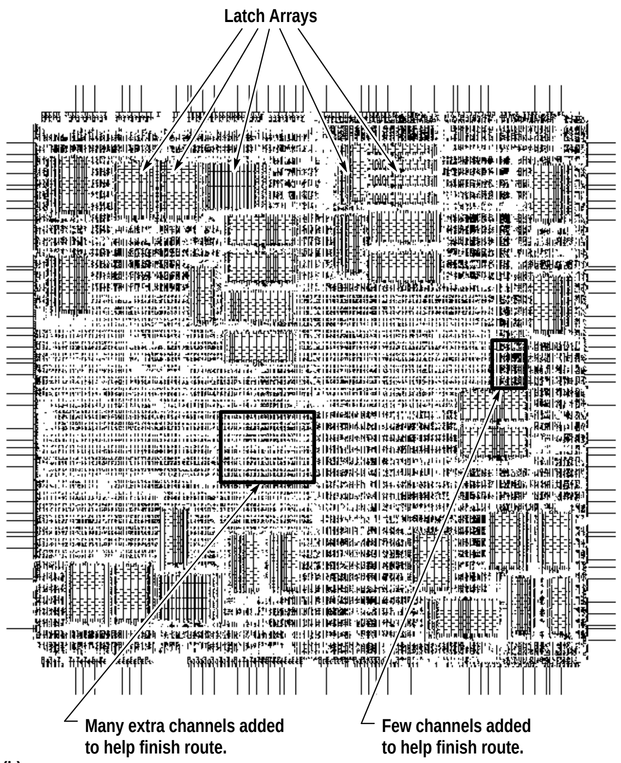

Latch Arrays
Many extra channels added to help finish route.
Few channels added to help finish route.
(b)

Fig. 2. *Processor interface gate array floor plan and placement views. (a) Floor plan tool (fpt) view. The block name and effective percentage utilization are shown. The effective utilization is based on the underlying obstructions used by the macro placement tools. Obstructions are usually channelization, latch array, or I/O regions. The shaded boxes are latch arrays. Blocks can be overlapped. A density map can be displayed, which shows where overlapped areas result in density that may be too high to place efficiently. Floor plan blocks that show similar effective utilization can actually be placed at different real utilization. For instance, csr_input at 77% (middle of die) and req_dec, also at 77% (right side middle above queue_hi), show similar effective utilization. Compared to the actual placement, csr_input is placed at lower real utilization because of the additional routing channels. (b) Placement view. In the placement view, the regions of custom channelization can be clearly seen. The latch array regions are also clearly visible. The actual size and number of the channels were determined by many iterations through the place-and-route flow, attempting to get a complete route. Register and multiplexer macros clustered around the I/O sites were placed identically to control chip-to-chip skew.*

- Floor plan. Wire table lookup was used for sections of the net that were completely contained within a block. Sections that spanned blocks were pseudorouted. The composite was then reduced to the π-model and T_{line} values.
- Placement. A pseudoroute was constructed for each net and then reduced to the π-model and T_{line} values.
- Routing. A special route analysis program was developed to analyze all metal layers and generate the π-model and T_{line} values.

Delay calculations were nonlinear and edge rate dependent. Presentation of edge rates in the critical path reports was very helpful in fixing slow paths. Slow edge rates were the first thing to look for (and fix) in a broken timing path.

Two delay values and two edge rate values were calculated per macro level:

- T_{gate} is the input-pin-to-output-pin intrinsic macro delay. T_{gate} is a function of the timing arc through the macro, the input pin edge rate (T_{sin}) and the π-model on the macro output. There are several T_{gate} calculations per macro, each corresponding to a different timing arc through the macro.
- T_{sout} is the edge rate seen at the macro output pins. For each T_{gate} calculation there was a matching T_{sout} calculation.
- T_{line} is the RC delay. T_{line} is a function of the net topology only.
- T_{sin} is the edge rate as seen at the macro input pins. T_{sin} is always greater than T_{sout} of the driving pin and represents the degradation of the signal edge as it propagates through the RC tree of the net. The edge rate degradation was a simple function of T_{line} and was not very accurate for complicated net topologies.

Each gate array used the same simple clocking scheme. At most there were three different clock phases, which were generated from the same master clock pin of the chip. The clock phases are 1 × period, 2 × period (2a), and 2 × period with the opposite phase (2b). The simple clock scheme allows a simple forward traversal algorithm to sum the delays T_{gate} and T_{line} to get path timing. Macro output pin edge rates (T_{sout}) were propagated and derated to the input pins (T_{sin}) during the same forward traversal. Typically, 1→1, 2a→1, and 1→2b paths were calculated in one pass, 1→1, 2b→1, 1→2a paths were calculated in another pass, and 2a→2a and 2b→2b paths were calculated in a third pass.

Resynthesis

Resynthesis can be thought of as an elaborate in-place optimization of the entire design. Resynthesis was used to reduce or remove logic, build and place optimal fanout trees, and change macro power levels to the minimum necessary to meet timing requirements. There were six steps in the resynthesis process:

- Flatten. The gate level Verilog modules were read in, linked and flattened so that the entire design could be easily optimized across hierarchical boundaries.
- Rip/join. Buffers and inverter pairs were deleted, simple macros were joined together, logic was reduced through constant propagation, and unused logic was deleted.
- Coarse fix. Each macro output pin in the design was either powered up (i.e., its drive strength was increased) or fanned out such that the output pin edge rate was below an arbitrary value and the input pin edge rates on the net were all below a (different) arbitrary value. This step also ensured that each macro output pin did not drive more than its maximum capacitive load.
- Complete timing analysis.
- Fine pass. Once the coarse fix was made and macros were powered up to meet the edge rate specification, a topological traversal starting from the register inputs and working backwards was made to further power up macros that were in broken critical paths. This step typically did not add more than 2000 basic cells to the design. Timing and pseudoroutes were updated after each macro power-up. The timing update algorithm retimed only those macros affected by the power-up.
- Power down. The coarse fix step and Synopsys can overpower components. A topological traversal was made starting from register inputs and working backwards to power down those components in noncritical paths (i.e., that had positive global slack) as long as the net loading rules were met. Timing and pseudoroutes were updated as the algorithm proceeded.

Resynthesis was run at two different steps in the flow. A first-pass resynthesis was done using floor plan net lengths and floor-plan-based approximate macro locations. This design was then placed and improved. The improved placement and netlist from the first resynthesis were then fed back for a second complete resynthesis. (Fanout trees built during the first resynthesis were ripped out by the rip/join step.) The second resynthesis resulted in about 1000 fewer fanout buffers and 10,000 fewer basic cells used than the initial floor-plan-based resynthesis.

Resynthesis and timing of the processor interface chip (317,000 random logic cells, ~91,000 macros) typically took 16 CPU minutes on an HP 9000 Model 735 workstation and consumed 320M bytes of virtual memory.

Figs. 3 and 4 show the differences in the delays seen at register inputs between timing runs made at different steps in the flow. Fig. 3 shows the differences between floor plan timing after resynthesis and final placement timing. Values less than zero show floor plan timing that was conservative compared to the final placement timing. Values greater than zero show floor plan timing that was optimistic compared to final placement timing.

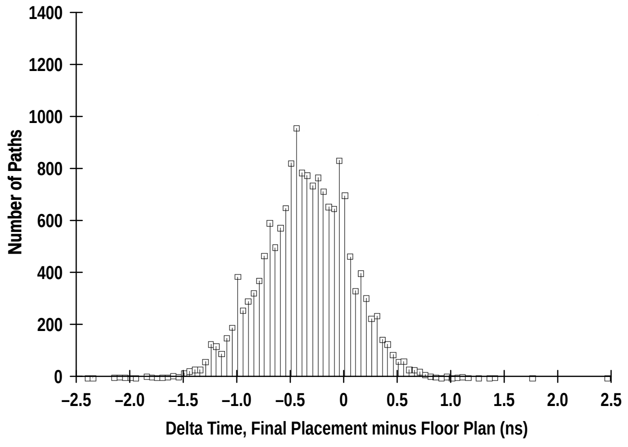


Fig. 3. *Timing comparison: floor plan versus placement.*

Fig. 4 shows good, albeit conservative, correlation between the placement and route timing. Relatively few changes had to be made in the design once the designers had timing working at placement. The histograms clearly show that an attempt to "make timing" at the Synopsys/floor plan step is unnecessarily conservative.

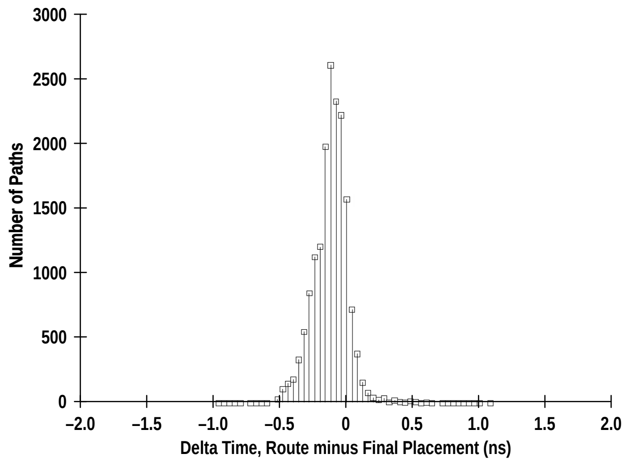


Fig. 4. *Timing comparison: placement versus route.*

Placement

The GARDS gate array suite of tools from Silicon Valley Research (SVR, formerly known as Silvar-Lisco), was chosen as the central set of tools for placement of the server gate arrays. The choice of GARDS was based on a history of successful experience with these tools. A long-held philosophy in our laboratory has been to put the place-and-route tools in the hands of the designers, and to choose the tool that best allows the designer to parlay design knowledge into an intelligent placement. With only minor exceptions, the GARDS tools functioned capably and fulfilled the place-and-route needs for which they were chosen.

Unlike many other design flows for 0.35-μm gate arrays, hierarchy was flattened by the resynthesis step, and the placement tool had free rein to optimize placement across boundaries. A drawback of this approach is that major updates to a current placement (e.g., synthesize one block and feed the changed block into the existing placement) were virtually impossible. In such cases, the placement process was restarted from the beginning.

Complementary internal tools constructed all necessary files in GARDS format, such as placement obstruction files for opening extra routing channels and for inhibiting automatic placement in critical regions such as latch arrays.

A valuable feature of the GARDS tool set is the C programming language database access routines known as GEARS. These made it easy for internal tools to extract design information to customize the integration of GARDS into the tool flow.

The GARDS libraries for placement and routing were created internally by translating Cadence LEF (Library Exchange Format) files provided by the chip vendor. The general placement scheme consisted of the following series of steps:

1. Import of preplaced gates (I/Os, internal macros in the I/O-to-register or register-to-I/O paths, and latch arrays)
2. Import of placement obstructions (for channelization, latch array protection, and clock tree allocation)
3. Min-cut placement of the imported floor plan. Min-cut placement is a recursive subdividing algorithm that seeks to minimize connections across the dividing line. The resulting placement has minimum wiring between macros.
4. Run checkfif (internally developed tool).
5. Unplace errantly placed components.
6. Min-cut placement of floor plan file containing only the errantly placed components.
7. Run a series of walking-window improvement passes.
8. Repeat steps 1 through 7 after a layout-based resynthesis.

Steps 1 through 3 generally took about two hours for the larger designs, and could be automated through scripting. After the initial read of the floor plan, it was necessary to check the integrity of placement adherence to the floor plan. An unfortunate feature of the placement tool was that if components could not be placed within their assigned floor plan bin, they would be placed according to the center of gravity of the connectivity, which usually was not acceptable.

Step 4 involved an internal tool, checkfif, which was developed to unplace errantly placed components and to create a new reduced version of the floor plan bin file that could increase the original dimensions of the bins. Steps 5 through 7 could also be scripted, with step 7 consisting of a mix of net-length-based improvements and congestion-based improvements. Step 7 represents the largest investment in CPU cycles, with a typical set of improvement passes taking about 24 hours.

Clock Tree Builder

Once placement was complete, the clock tree builder was used to generate different gated types of the master clock. The structure of the clock tree, which was identical for all CG61 arrays, was a uniform grid of taps driven from a central point (see Fig. 5). A tap was either an OR gate, which generated some form of gated clock, or a buffer, which generated a free-running clock. Macro positions and most connections were predefined, with the only variables being the tap type (OR gate or buffer), the tap flavor (which gating term drove the OR gate, if any), and the last-level connection to the registers.

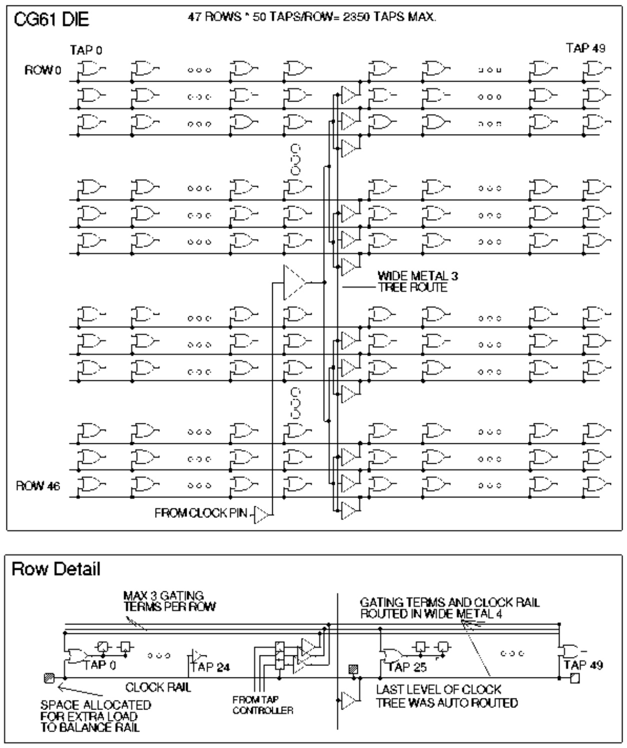

CG61 DIE
47 ROWS * 50 TAPS/ROW= 2350 TAPS MAX.
TAP 0
TAP 49
ROW 0
WIDE METAL 3
TREE ROUTE
ROW 46
FROM CLOCK PIN
Row Detail
MAX 3 GATING TERMS PER ROW
GATING TERMS AND CLOCK RAIL ROUTED IN WIDE METAL 4
TAP 0
TAP 24
TAP 25
TAP 49
CLOCK RAIL
SPACE ALLOCATED FOR EXTRA LOAD TO BALANCE RAIL
FROM TAP CONTROLLER
LAST LEVEL OF CLOCK TREE WAS AUTO ROUTED

Fig. 5. *Clock distribution tree.*

The clock tree builder removed taps obstructed by latch arrays, figured out a best initial assignment of taps associated with a gating term register, and then minimized the last level of clock tree routing by optimizing the clock-load-to-tap assignments. The last level of clock tree routing was performed by the autorouter. The clock tree builder provided the user with the flexibility of manually selecting tap gating term assignments and tap load assignments. The complete clock tree consumed 1.5% to 2% of the die area.

Skew

Intrachip skew was calculated as the difference in delay between the fastest tap and the slowest tap. The maximum intrachip skew for these gate arrays totaled 400 ps, of which 200 ps was actual metal and gate load variations and 200 ps was intrachip variance.

Metal and gate load variations were mostly due to different numbers of loads and different route lengths in the last level of the tree. Dummy loads were added to the tips and center of the main clock rails after routing to balance the rails precisely. Special control of the last-level loading and routing of I/O registers reduced the metal and gate load skew on those registers to 100 ps.

The variance portion of skew included the voltage, temperature, and process variations possible across the die. Variance also accounted for delay modeling uncertainty between OR gates and buffers in the last level of the tree.

Scan Chain Hookup

Once all the scan elements were placed, the scan chain was automatically connected. Each element had a property that identified the element's scan ring. The beginning and end of each ring had a property that marked these points as special. Buffers and pairs of inverters that were connected to the scan points were marked as equivalent to the scan point and could also be used. Each ring was stitched together using a "greedy" algorithm that chose the best element to add at each step. The best element was determined by assigning a penalty value to each available element and then choosing the one with the minimum penalty. The penalty value was a summation of four individual penalties:

- Distance. The distance between the element and the connection point. Reducing this value decreased the amount of routing needed to connect the ring.
- Clock. A penalty was assessed if there was a difference between the clock for the element and the clock for the connection point, that is, if the clocks came from different taps or rails. The more the clocks were similar, the less the chance for hold errors while scanning.
- Buffering. A penalty was assessed if the connection point was not buffered from the true scan point. Preferring the scan points that were buffered reduced the possibility of hold problems.
- Center crossing. The connection crossed the vertical centerline. The main clock distribution was in this area and horizontal routing was limited. Reducing the horizontal scan connections through this area simplified the routing for the design.

Each of the above individual penalties had an associated weight so the relative importance of each could be adjusted.

For the designs that had routing completion problems, a post-hookup optimization could be applied. This optimization only considered the distance between scan points as the objective to minimize and often resulted in over a meter of wire removed from the scan chain.

After all of the scan rings were connected, the wires added at the gate level could also be inserted into the behavioral model.

Routing

Routing of the chips was accomplished with the GARDS route tools (garout/linesearch, garout/maze, redit/ripup-reroute). Normal signal routing occurred on metal layers 1, 2, and 3, with only clock net routing on metal layer 4. The version of GARDS used by the server project did not include support for fourth-layer routing, so fixed metal 4 segments were specified and connected with internally developed postprocessing tools dff2def and eclare (see below).

The routing process began with certain special nets. These were mostly latch array nets and certain clock nets, which needed to be routed with the maze router (garout/maze). Then the line search router (garout/linesearch) was used to route nearly all of the remaining connections, leaving only about 1% to 1.5% failing to route. These were routed by the maze router, usually to within about 200 disconnects. At this point the design was taken into the redit tool, using the ripup-reroute feature of that tool. This normally completed the remaining connections. The whole routing process generally took one to two days to complete.

Some special techniques were exploited to push the envelope of routing density. It was observed that metal layer 1 was underutilized, and that some of the saturated metal 2 routing could be forced down to metal 1, thus allowing extended use of metal 2. This was done with an internally developed tool that read the routing that had occurred up to a certain stage, determined candidates for moving metal 2 horizontal segments to metal 1 horizontal segments (nonpreferred direction), and finally moving those segments. This technique helped complete the routing in critically dense areas.

Another technique for achieving better routing completion was channelization (see Fig. 2). When a routing attempt failed to make all connections, additional routing channels were created in regions where concentrations of disconnects were observed. The channel obstructions were fed back to the floor planner, where the designer could make sure that the effective utilization of each block was still below the safe limit. If necessary, blocks were enlarged.

Postroute Analysis

After placement and routing were complete, the capacitance and RC delays were extracted by an internally developed tool, eclare. eclare read the GARDS database to obtain the macro and routing information for the design. The Verilog gate level netlist was also read so that the connectivity present in the GARDS database could be verified against the netlist connectivity. eclare then used the GARDS data to construct four arrays of grid points (one per metal level) that described the metal present at every grid location. The information stored at each grid point included metal width, type of metal, directions of connections, an index into a table of pin and net names, and a visited flag.

Storing this much information per grid point is expensive in terms of memory (720M bytes of storage is required to hold the grid data for the current chips), but it greatly simplified the determination of neighbors during the extraction process. The project had access to several machines that had 1G bytes (or more) of physical memory, so the storage requirement was not a serious impediment. This data structure also allowed the run time of the tool to be fairly short. eclare ran in less than 30 minutes, with about half of the time being file input/output.

The extraction process began after all of the grid information was loaded. For every driver on each net, eclare traced all of the metal attached to the driving pin. As the net was traced, a list of nodes and segments was built for use in solving for the equivalent π-model and RC delays.

For each grid point traced, nearby grid locations were examined to determine their contents. When all of the neighbor information was determined for the current grid point, a lookup table was consulted to obtain the capacitance value for this neighborhood configuration. This capacitance and the resistance for the current metal level were added to the appropriate entry in the segment list. The capacitance value was also added to a counter for each of the neighboring nets for use later when determining the coupling factor.

When all of the net had been traced, the list of nodes and segments was used to solve for the RC delays to the input pins and the equivalent π-model at the driver. All neighboring nets were examined to determine the maximum amount of capacitance that was coupled to the target net. If the maximum coupled capacitance to any single neighboring net exceeded 20% of the total capacitance for the current net, the routing had to be adjusted to decrease the coupling. The coupling percentage was used to create a minimum/maximum range of capacitance values in the equivalent π-model that was later used by the timing verifier.

eclare performed a few other tasks in addition to the extraction. The Fujitsu toolset checked for nets that failed their antenna rule (too much unterminated metal connected to the gate of a transistor during fabrication). eclare also made this check, but earlier in the overall design flow. Optionally, eclare could output a distributed RC SPICE model for any of the nets in the design. Finally, eclare produced a file of clock rail loading that was used to determine the number of rail balance loads needed to equalize the total capacitance on each rail.

Last Minute Changes

Given the large latency from RTL behavioral change through Synopsys and resynthesis to placed design (usually three to four days), an interactive *tweak editor* was constructed to allow designers to edit the netlist and placement interactively. This X Windows tool displayed the placement, netlist, and timing information graphically and then allowed the designers to add, change, delete, and move parts and make other changes. Global chip timing information was updated in real time so that the effect of these netlist edits was instantaneously viewable.

The tweak editor gave the designers more than enough latitude to change the logical behavior of the gate design (and sometimes that was the exact intent). The new netlist was logically verified against the RTL behavioral model with lover, the Boolean equivalence verification tool.

Vendor Interface

The vendor interface consisted of a front end, where libraries of various formats were received from the vendor, and a back end, where ultimate tape-out-ready design files were sent to the vendor for go-to-make.

On the front end, the vendor supplied all necessary libraries. Some of these were characterization data files in the vendor's internal format, and the rest were in various standard formats, such as Verilog and LEF. Also provided was an integrated set of programs that were pieces of the vendor's own proprietary ASIC development package. These programs, which included the vendor's LCADFE and GLOSCAD packages, also included library data.

The physical libraries provided by the vendor were in the standard Cadence LEF format. An internally developed tool, lef2gards, was used to translate the LEF description of the base array and the macros into GARDS library formats. The lef2gards translator had been used in previous projects, and only required some minor modifications to work with the vendor's CMOS LEF files.

On the back end, another tool, dff2def, was developed to read the GARDS design file (using GEARS access routines) and produce a logical netlist in the vendor's proprietary format, FLD, as well as a Cadence DEF (Design Exchange Format) file, which contained all of the physical information such as placement and routing. The FLD file was imported into the LCADFE vendor package, where certain netlist expansions and design rule checks were done. Then the GLOSCAD package was used to import the LCADFE logical data and the DEF file physical data. This was followed by a run of the GLOSCAD laychk program, which did a thorough physical design rule check. Finally, the GLOSCAD pdilmake program produced the π-model and RC data, which was folded back into the LCADFE package for postlayout timing simulation. A complete set of vector simulations was then run to verify both timing and logical aspects of the design.

After determination of successful timing and clean LCADFE and GLOSCAD runs, and after passing a stringent tape-out review, the appropriate design files (essentially tar snapshots of the vendor's LCADFE and GLOSCAD user design tree) were transferred via ftp to the vendor's site. A go-to-make release usually followed within a few days of the original transfer.

Conclusions

Commercial synthesis and layout tools alone were not sufficient to achieve the ASIC performance, density, and time-to-market goals of the S- and X-class machines. A significant investment was made in both internally developed and commercial CAD tools to put together a highly optimized physical chip design process. The resynthesis process coupled with in-house place-and-route tools minimized direct human interaction in optimizing the synthesized designs and made it possible to achieve the project design goals, which would otherwise not have been possible.

Acknowledgments

The authors cannot overstate the importance of the work of Harold Dozier in providing the fundamental technical interface with Fujitsu and implementing many of the features of this flow.

Reference

1. L. Bening, *Putting multithread behavioral simulation to work*, 1996, accessible on the World-Wide Web at:
http://www.convex.com/tech_cache/supercon96/index.html

▶ Go to Next Article
▶ Go to Journal Home Page