GHAZI: An Open-Source ASIC Implementation of RISC-V based SoC

Zain Rizwan Khan, Wajeh ul Hasan, Zeeshan Rafique, Ali Ahmed Ansari, Syed Roomi Naqvi

***Abstract*—Due to the closed source, expensive nature of digital design tools and licensing cost of System on Chip (SoC) IPs for ASIC, the hardware industry lacks innovation and design reuse. In the last few years, the hardware industry is seeing open-source adaptions, just like the software ecosystem. This paper presents a methodology of adapting complete open-source digital tooling, ISA, IPs and manufacture-able PDKs to tapeout a minimalist RISC-V based SoC named GHAZI. The methodology uses an RV32IMC core and an SoC reference design from OpenTitan (Ibex core and peripherals respectively) at base, adding instruction and data memories, converting the design into verilog for the RTL to GDSII flow with opensource tools, alongside an FPGA implementation for Xilinx Arty-7 FPGA, finally generating the GDSII layout using OpenLane on Skywater 130nm PDK. Verification was done on all stages, i.e. RTL, gate level simulations and LVS, before integrating GHAZI-SoC with Caravel, the template SoC design for the Open multi-process wafer (Open-MPW) shuttle. The GHAZI-SoC design works on 1.8V DC power signal and a switching frequency of 12.5MHz. With over 90,000 physical cells and a die area of 6.9mm2, it utilized a place and route flow for the physical design and implemented it using a high density library of PDK. The fabrication was fully sponsored by Google with the Open-MPW shuttle from Efabless and was done at Skywater Foundries. To the best of our knowledge, this is the first effort of utilizing Open-source technologies to tape-out a design containing OpenTitan IPs (including the Ibex core). The generated GDSII, netlist and RTL files of the SoC, along with the relevant documentation, are present in a GitHub repository for public access.**

***Index Terms*—RISC-V, OpenTitan, FPGA, SoC Design, ASIC, Opensource**

I. Introduction

THE open-source movement and commercial entities have co-existed since more than thirty years ago. Nowadays, the whole software development ecosystem is based on this movement, which was huge, because, from the very start, way all those years back, the public started seeing projects such as Linux, the core element 'kernel' of a truly open-source operating system[15], followed by the development of Apache. Most of the success of the open-source comes from rapid adoption as well as the ability to build on top of the existing stack, which has resulted in a wide range of robust, stable and flexible products. Commercially providing complementary services and products that are not supplied efficiently by the open-source community further becomes a source of learning for the community itself and the ecosystem therefore keeps evolving.

Now the question arises if a similar development ecosystem for open-source hardware works in the same way as its successor does for open-source software[4]. This would counter the current high design time for hardware development but may also take a great toll on the quality of designs, especially on the quality of those designs that would turn up on a high volume System on Chip. Also the vendors of different EDA tools as well as FPGA and ASIC vendors and the silicon foundries must equally participate in the development of the open-source hardware[27]. There is a need for not only the continuous improvement of digital designs, but also to ensure that the software built on top of the evolving hardware can be backwards compatible or that it should at least not require much change with every release of a digital design or hardware specification. This does lead to a transparent verification flow though, where hardware developers using an open-source intellectual property (IP) can reproduce results at their end too.

An important enabler of the path leading to open-source hardware ecosystems is, perhaps, the RISC-V instruction set architecture[35], which is a standardized open-source ISA and gives freedom to a hardware designer to not only use it but to implement it however they want. The implementation can be in the Verilog hardware description format, which is a subset of SystemVerilog[12], and is itself an open-source industry standard format. The semiconductor industry is headed towards an era where it will not be driven by a single product in the next generation of computers. Instead, a wide range of new applications are being developed that are fueled by the advent of foundations such as the free and open-source foundations (FOSSi)[34] and the open-source ISA itself, which demanded a fundamental change in custom silicon design. Such changes often come as tidal waves[21] that periodically increase the growth of semiconductor industry. For example, RISC-V architecture based flash controllers are being developed and used by Western Digital, out of which, the SweRV family of RISC-V cores[18] have also been released for anyone to adopt it in their design as well. Then there is Ibex, a small and efficient, 32-bit, in-order RISC-V core with a two stage pipeline that implements the RV32IMC instruction set[35]. It is written in SystemVerilog (same as the SweRV cores) and is a low-performance, area-optimized core[29] specifically designed for Internet-Of-Things platforms. It started out as the zero-riscy implementation[9] by the PULP platform and is now managed by lowRISC[17].

The core needs some sort of peripherals, to communicate with devices outside the chip area, which are connected with the core via a system interconnect bus. These components, along with memories, an optional debug module, PLL and pads control logic complete the design of a System-on-Chip (SoC). To implement such an SoC on an application-specific integrated circuit (ASIC) the register-transfer level design of

the SoC has to be converted to a graphic design system (GDSII) stream format, which is then fabricated into the ASIC chip. This is called taping out a design. However, If anyone from the open-source community wants to tape out their design, not only would they need an open-source electronic design automation (EDA) tool[13], but would also require the availability of a free and production capable process design kit (PDK) from the same foundry that would fabricate the final chip. This is because until now, every EDA tool used to require access to proprietary information in a PDK[6].

For years, this was a bottleneck; the RTL could be written/reused and open-source EDA tools were also available. Nevertheless, the community never saw an open-source PDK. Last year was a huge breakthrough[19] when Google, in collaboration with Efabless, announced an open-source 130nm PDK from SkyWater. The open-source sky130A PDK includes the following features:

- Support for internal 1.8V with 5V I/Os [operable at 2.5V]
- 6 layers [1 level of interconnect, 5 levels of metal]
- Is inductor-capable
- Supports 10V regulated supply
- Includes SONOS shrunken cell
- Optional MiM capacitors

The sky130A is, in fact, a derived open-source release of the original sky130 process node and PDK which is believed to be usable for doing test chips and initial design verification[31]. To take a design all the way to GDSII using such a PDK for working with various open source tools EDA tools, it is necessary to make sure that the provided files and directories adhere to the known open standard formats[5]. These are library exchange format (LEF) files, SPICE and behavioral (Verilog) models, liberty files, etc. Netgen, for example performs the layout versus schematic (LVS) check which is a comparison of either a SPICE model[20] extracted from a layout, and a verilog gate-level netlist based on the behavioral model of all the cells.

A. *Using OpenLane*

Though one can have a doubt on the reliability of open-source tools and how good they are in meeting the different physical requirements for taping out an ASIC chip, efabless themselves taped out their striVe family of SoCs using OpenLane[24], an RTL to GDS flow based upon the OpenROAD project[25][30]. The tool can now be used to assist any design, for any process technology, in the end-to-end silicon compilation[2]. The flow performs all the steps for an ASIC implementation. The primary input to the flow is the RTL design written in Verilog/SystemVerilog. The flow is the explicit combination of EDA tools to achieve the design of an integrated circuit[1]. The steps can be as simple as the conversion of an abstract specification of desired circuit behavior into a design implementation in terms of logic gates (synthesis), assigning exact locations of those logic gates within the chip's core area (floor-plan and placement) and adding wires needed to properly connect all those logic gates (routing) while obeying all design rules for the IC. These rules are often referred to as constraints and analysis flows are performed amidst to ensure design closure, that is, with every step, the expanding enumeration of design constraints and objectives are met. The aim of this aggregation was to make an automated RTL to GDS flow, as shown in figure 1, where the tool would automatically pass the output of the previous stage to the succeeding step.

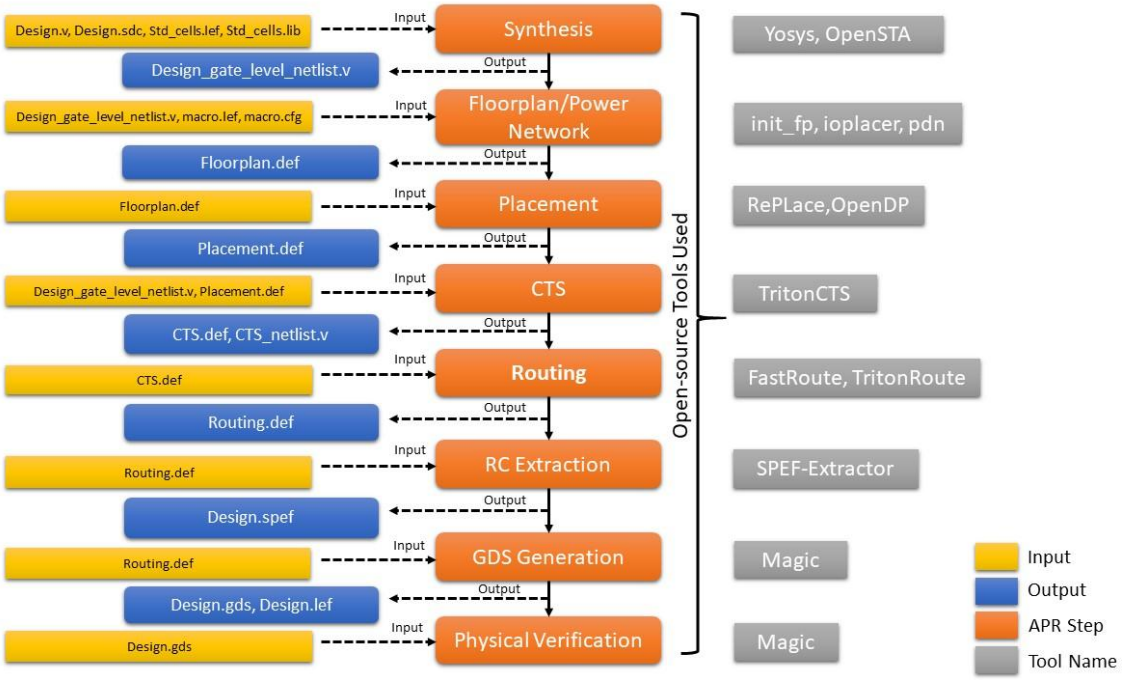


Fig. 1: Automatic place and route flow

Apart from the usual automatic place and route (APR) flow, we use the popular Magic VLSI Layout tool[5][26] within OpenLane for the sub-processes in table I:

Dummy fill insertion	Density checks and fill patterns
GDSII stream out	Mask-level GDS data file, LEF views as hardened macro cells
Design rule checking (DRC)	Physical verification
SPICE netlist export	For LVS checks
Antenna violation handling	To place (and replace) both dummy and real antenna diode cells

TABLE I: Magic tool usage

For GHAZI SoC, we decided to go with OpenTitan[23], using part of its reference design, and the Ibex RISC-V core sub-project. OpenTitan is the first open-source initiative to create a transparent, high-quality reference design for silicon root of trust (RoT) chips, as well as integration guidelines. For the open-source community, this means that they can have improved, pre-verified intellectual properties (IPs) and tapeout readiness[28]. Yosys is a framework for RTL synthesis tools that is free/libre and open-source[37]. It presently supports Verilog-2005 extensively and includes a basic set of synthesis techniques for a variety of applications. Every step in the IC design (such as static timing analysis, placement, routing) is already complex and, time and again, establishes its own field of study. The concept phase started out with simulations on the reference design to verify its functionality, performing plenty of regression testing. We even ported the design to an FPGA, in a repetitive manner, upon addition of each feature from the basic design specification in an agile approach.

Our contributions to the basic reference design were to use D flip-flop based memories and 6T cell based memories compiled using OpenRAM[10] in two separate implementations. We modified the clock gating according to the cells available from the PDK that we're using and added a state machine that would wait for a host computer to load a program into the memory over a serial UART channel (separate from the peripheral UART). Also, we integrated it into Caravel[3], a template SoC with on-chip resources and control over the pad-management for GHAZI SoC as well as a chip logic analyzer (LA). We generate the GDS from the netlist using Skywater's free and open-source 130nm production PDK. As per the catalogue, this was the only chip taped out in 130nm PDK using lowRISC IP and was among the 45 designs[8] aboard the open multi-process wafer (Open-MPW) shuttle, also by Google and Efabless[22], that were sent to the Skywater foundry for fabrication.

II. Tools Used

A. SystemVerilog

All of the RTL was written in SystemVerilog. This logic design language (or Hardware Definition Language) has models for RTL that are drastically different from one another and, if care isn't taken, this can lead to code conflicts and code review latency. Since SystemVerilog is also used for verification, both synthesizable and test-bench code can be written which should have a clear distinct line in between and can not be mixed. Since most of the IP we used came from lowRISC, it was their coding style guide[16] that was kept in mind when writing logic for the top level wrapper and memory interfaces. This includes using the begin and end statement, proper indentation and spaces, following the naming conventions for port names of a module, etc. For example, listing 1 shows the definition of the ICCM controller (the state machine that loads the program into the instruction memory over UART), which will be discussed further in the methodology section of this paper.

```
// input and output signals should have '_i'
// and '_o' appended after signal name,
// respectively for input and output and
// '_ni' and '_no' for inverted (active low)
module iccm_controller (
    // Clock and reset
    input               clk_i,
    input               rst_ni,

    // Input from UART Rx
    input               rx_dv_i,
    input        [ 7:0] rx_byte_i,

    // Memory interface
    output logic        we_o,
    output logic [13:0] addr_o,
    output logic [31:0] wdata_o,

    // Reset for system
    output logic        reset_o
);
// ...
endmodule
```

Listing 1: Example port declaration style

B. sv2v: SystemVerilog to Verilog Conversion Tool

GHAZI SoC is a project written in SystemVerilog, but currently Yosys only supports a small subset of SystemVerilog and Icarus Verilog[11] supports verilog only. The sv2v tool[32] is built over a preprocessor and uses abstract syntax tree (AST) representation to make conversion methods more standardised and straightforward. To run our design on OpenLane, and also to be able to verify it inside Caravel, we had to make sure our design was supported by both Yosys and Icarus Verilog for synthesis and simulation purposes respectively. We had to convert our design from SystemVerilog to Verilog using the sv2v tool. In the implementation section, there are details though, on the changes we had to make in the converted RTL afterwards, to make it finally work on Yosys. Running simulations on Icarus Verilog consequently proved that the changes we made along with the whole conversation were functionally verifiable.

C. Icarus Verilog Simulator

As with openlane being open-source, the simulator that we used to validate and test our RTL on was Icarus Verilog, which is another open-source tool. When simulating through Icarus

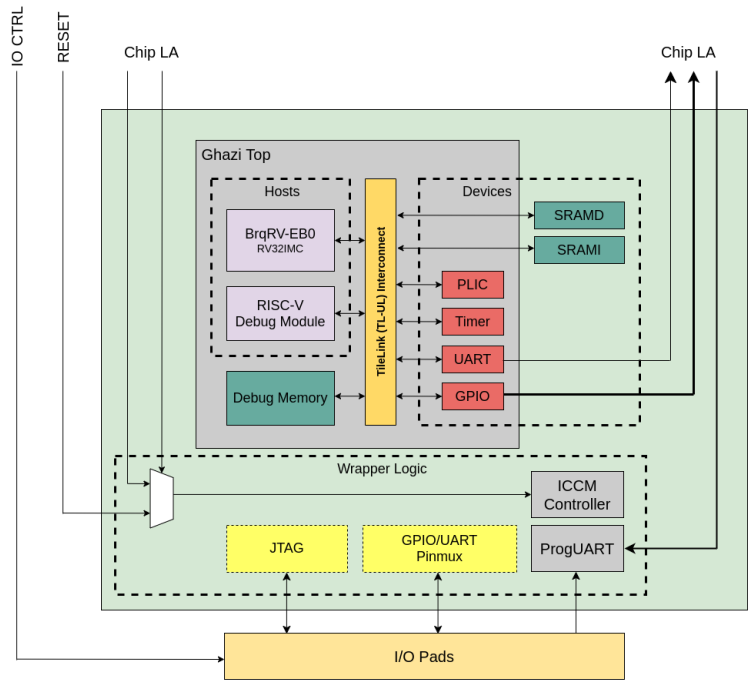


Fig. 2: Top level view of system

Verilog, an internal netlist is generated hence it can double as a synthesis tool. We have only used it to compile the converted Verilog source code via its intermediate form called the "vvp assembly". In this form, the include and define directives have been implemented.

The design netlist along with this intermediate form is a result of an elaboration where the references and instantiations are resolved and expanded respectively[36][11]. The code is combinationally reduced and, is optimized for net-effect zero circuitry and constant propagation. This intermediate form can be executed, successfully testing the functional correctness of the RTL with any desired inputs as injected from the test-bench.

D. Caravel Test Harness

The Caravel chip by Efabless, as mentioned before, is an ASIC implementation, that is based off on the famous full chip PicoSoC reference design[7] of the PicoRV32 RISC-V core. It uses a version of the CPU with the Wishbone Master interface[33], the other versions being a standalone one and one with an AXI4-Lite interface. It comes with the RV32IMC 2-cycle core itself as well as the firmware/software, using GCC in a 32-bit RISC-V cross-compiler to hex file target. The implementation consists of three areas. The first one is a management area that has a flash memory controller over an SPI interface, a Wishbone bus, peripherals and two interfaces over to the user area. The user area is the largest with 10mm2 of die area and access to I/O pads shared with the management area. The last one is a storage area, which has a 1-kilobyte scratchpad SRAM memory and is accessible to the PicoRV32 only.

The two interfaces that come with the management area are the on chip LA for reading and controlling signals internal to the design in the user area, and the Wishbone slave interfaces that let's the design in the user area act a peripheral to the PiocRV32 core. The Wishbone interface has clock and reset signals which the user area can use regardless of the design within needing the rest of the interface. The clock is controlled by the PLL contained within a module that acts as an SPI slave interface accessible from a remote host. Its purpose is to give control over certain system values and "housekeeping" tasks, including adjusting the clock speed of the CPU, and can work even when the CPU is in full reset.

E. RISC-V GNU C Compiler Toolchain

A tool chain that contains a compiler, an assembler, a linker and a disassembler, that allows the developer to compile RISC-V assembly according to the implemented hardware. To use the GCC compiler suite, we first write a linker script that is compatible with our SoC's memory map. The C program that we use to test our SoC with, contains a startup code (the instructions that are executed first). Our C program is then compiled and linked with the above mentioned linker script, to generate the binary. and eventually the hex file that gets loaded in the emulated memory in our RTL.

III. Methodology

Starting with a bottom up approach, the most basic elements of our design were a 32-bit in-order RISC-V core with a 2-stage pipeline that implemented the RV32IMC instruction set architecture, an interconnect bus and separate memories for instructions and data storage. Extending the design meant adding peripherals to the interconnect such as a timer, an

interrupt controller, 32 GPIO and a 2 pin full duplex UART. The design of the whole SoC is shown in figure 2.

The design specification for GHAZI SoC also included a top level module that included logic for multiplexing the input/output (I/O) to the SoC and mapping them to the 38 management controlled pads as shown in figure 3.

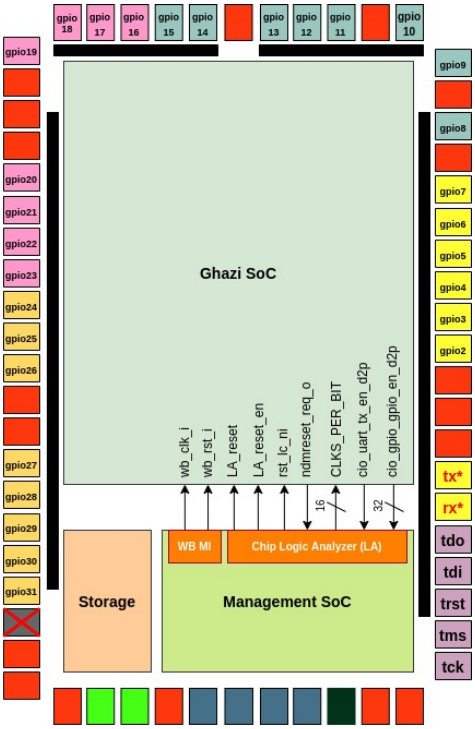


Fig. 3: GHAZI SoC inside Caravel

The output enables of the I/O were also sent to the pad controller as well as, being internal signals to the design, mapped to some of the logic analyzer bits. For clocking the whole design, we had two options, one of which was a user_clock signal and the other one being the clock signal in the Wishbone interface. The reset signal that came with the Wishbone interface was common with the reset to the whole chip, and was used to initially reset the system (including the PicoRV32). Then, with our design being integrated in such a way, the PicoRV32 can act as a reset manager for our design using the chip LA once it has initialized the pads.

One thing that can be observed in our design is that neither memories that we used returns a valid signal when it fetches data out of its cell array to be read by the CPU. A valid signal is used by the core to know that the data on the write bus is correct and it can be saved in the register. For that, we had to write our own logic, shown in Listing 2, for valid that assumes that the data on the bus is correct, and can assert and de-assert the valid signal with each cycle of request from the CPU.

A. Core Microarchitecture - BrqRV-EB0

The BrqRV-EB0 core here is in a very basic sense, Ibex with its pipe-lining configured to be an experimental 3-stage, as shown in figure 4, instead of originally having just a fetch stage and a decode and execute stage. It also has instruction and data memory interfaces modified for the interconnect bus which support bus width transaction sizes as well as masked write operations (byte and 2B).

The fetch stage has a pre-fetch buffer, which includes a FIFO using a feed-through path, to make an instruction available on the output, as soon as it is stored in the FIFO, when empty. Ibex had another experimental feature which was the branch prediction but we decided to keep that off for this version of GHAZI. Also we tried between three different implementations of the multiply-divide unit (although its the multiply operations that varies across them and all three implementations uses the long division algorithm). The single cycle multiplier straight away led to placement overlapping in the APR stages discussed later on in the implementation section. The fast multi-cycle multiplier took three to four cycles whereas the slow multiplier may have took up to 33 cycles but with it's compute additions being done in the arithmetic logic unit (ALU). So we went for the fast multiplier which synthesized and went through place and route along with the design without any issues.

```
always @(posedge wb_clk_i) begin
    if (!rst_ni) begin
        ram_main_instr_rvalid <= 1'b0;
    end else if (ram_main_instr_we
            || ram_prog_instr_we) begin
        ram_main_instr_rvalid <= 1'b0;
    end else begin
        ram_main_instr_rvalid
                <= ram_main_instr_req;
    end
end
```

Listing 2: Logic for valid data signal in memory operations

B. Interconnect Bus

This interconnect bus uses the TileLink Uncached Lite (TL-UL) protocol and hence will be referred to as the TL-UL interconnect throughout the rest of this paper. The official TileLink is a 5-channel bus with features such as point-to-point split transactions, support for multiple hosts and devices, etc. When integrating multiple IPs having different functionalities on a single chip, the design process of the SoC becomes very

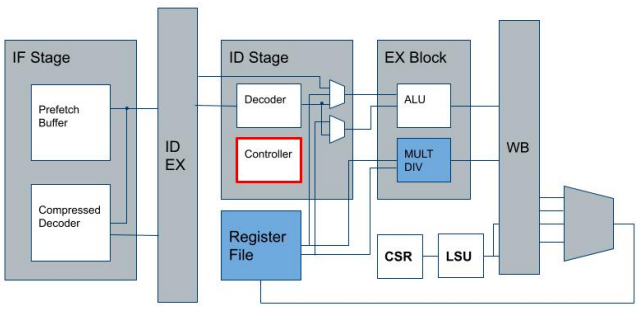


Fig. 4: CPU Core Microarchitecture

complex and controlling the area becomes a key. Hence we used the Uncached Lite version of TileLink, which follows on these features with a 2-channel bus and the added advantage of having a low pin-count overhead.

The design referenced in this paper has a total of 3 hosts, and 7 devices which are arranged in the following manner:

1) The instruction memory interface of the core having access to the instruction memory and the debug memory
2) The data memory interface having access to the data memory, the debug memory and peripherals (Timer, PLIC, GPIO, UART)
3) The system bus access for debug which can connect to the instruction and data memories and all the other peripherals

C. Memories: D Flip-Flop Based

Our initial goal was to use the 6T cell based SRAM model compiled using the OpenRAM memory compiler. This compiler uses a configuration file written in Python to generate files related to the APR (GDSII, verilog behavioral model, SPICE, Liberty, LEF, etc.). We had to place OpenRAM as a macro in our design but in the initial APR of our design, the tool was having trouble with that which resulted in a lot of routing violations. Hence we decided to use flip-flop based memory cells that we synthesized along with our design. Though these memories did end up taking more space on the chip than the OpenRAM generated ones, as we can see in figure 5, the difference between the size of a cell from each memory.

D. ICCM Controller

Finally, for loading the program onto the ICCM, we have another UART receiver and a controller that consecutively takes 4 bytes from the receiver and writes to the ICCM. The controller implements a 4-state finite state machine. The UART that received the instructions uses the same receiver pin as the peripheral UART. The baud rate for the UART is set using the CLKS_PER_BIT value which can be calculated by dividing the frequency of the clock with the desired baud rate. This CLKS_PER_BIT is also interfaced at the chip LA and can be set at the very start of the execution flow right after the PicoRV32 has initialized the I/O pads while the design in the user area is in reset. And as we know, on one of the I/O pads is the UART receiver itself hence the instructions can be loaded as soon as this reset is de-asserted. The last instruction, 0x00000FFF, is used for signalling the end of the program and does not need to be loaded into the instruction memory. This is shown in figure 6.

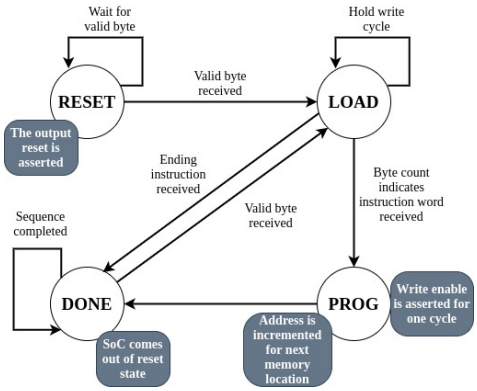
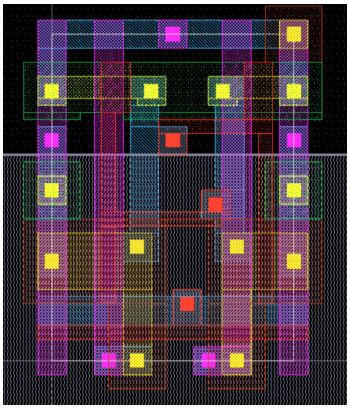


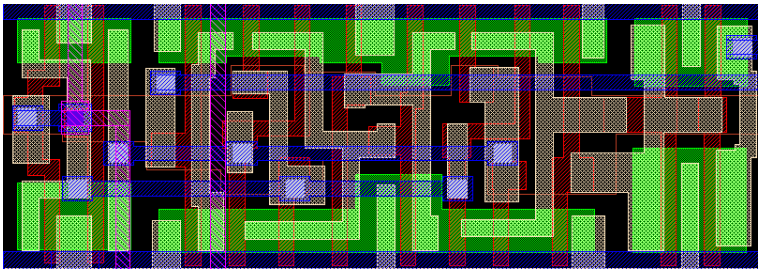
Fig. 6: FSM for loading program

E. Platform Level Interrupt Controller

Three peripherals (Timer, GPIO, UART) in our SoC can be configured to generate interrupt events under certain conditions. The interrupts can be edge triggered, level triggered, etc. To manage these interrupt events, the module that we used implements the RISC-V platform level interrupt controller. All the interrupts are gathered in a single interrupt vector and fed to the PLIC. Based on priorities, our core receives one interrupt line from PLIC which is then queried for the ID of the bit in the interrupt vector indicating which peripheral triggered the event. The communication between the PLIC and CPU core is also done using the system bus.



(a) Standard 6T cell 6.8um x 9.2um



(b) D Flip Flop Based RAM cell

Fig. 5: Comparison of memory cells

F. RISC-V Debug

Back to the top-level view of the system, we included a debug module in the design that can put a request to reset the system (for program debugging and setting up breakpoints) and also has access to the system bus. It has a 5-pin joint test action group (JTAG) interface that can be used for testing and system debug. The pin names are 1) Test Data In (TDI) 2) Test Data Out (TDO) 3) Test Clock (TCK) 4) Test Mode Select (TMS) 5) Test Reset (TRST) and are mapped to the first five pads of the shared GPIO as shown in 3. For debugging the program in actual though, an external translator hardware is also required which would take the remote bit bang sent from OpenOCD, decode it and send it over JTAG. OpenOCD is used for implementing the remote GNU debugger (GDB) protocol.

Also shown in figure 3, are two signals, an input reset to the debug module (which is separate from the system reset) and an output request to reset the system, similar to the debug request that is sent to the core. Both these signals are mapped to an output pin and to an input pin of the chip LA respectively. This reset management was much needed as we not only had the external hard reset, but an internal reset request pin from the debug module that could be read (also using the chip LA) and interpreted to assert the reset to the design.

IV. Design Analysis

To prove our SoC being ready to enter the tapeout stage, we had to make sure that it was a verifiable IP and that it had been synthesized and implemented first to run on a silicon testbench, that is, converted to a bitstream and uploaded on a field-programmable gate array (FPGA). Initial verification of GHAZI SoC SystemVerilog RTL converted to Verilog, was done by simulating it on Icarus Verilog. All the test benches were self-checking in nature but also generated waveforms in a value change dump (VCD) format. The VCD traces can then be viewed to further debug the design after the simulation has completed.

A. Gate-level Simulations

We had to simulate the design of GHAZI SoC furthermore upon being converted to a representation containing wires and gates. The netlist, that is generated multiple times in an APR flow, which will be discussed in the implementation section, may skip some logic for functionality or even whole modules when running it through various optimizations. We verified the functionality as a whole, keeping in mind the following things:

1) Debugging a flattened netlist: The generated netlist in OpenLane is a version of the design where all the modules in the RTL are synthesized along with their respective parent modules and we ended up getting a single file without any hierarchy that was previously there in a design.

2) X-propogations: Non-resettable flip-flops in the design could not been detected in RTL simulations due to X-optimism, a concept where an input don't care (X) value is tied to a known value, by traditional simulators. These X values do propagate in gate level netlist though as it includes the functional Verilog models of the standard cells.

B. Running on an FPGA

The SystemVerilog RTL of the design was mapped on a Xillinx Artix-7 FPGA. The part that came along with the Arty-A7 evaluation board for it contained more than thirty thousand logic cells. FPGA emulation of our design gave us some insight on how our design would would work in actual Silicon as well as on meeting the timing requirements and area constraints. Table II shows the resource utilization of GHAZI SoC for the Artix-7 chip.

	Used	**%**
LUT	9609	46
Flip-flops	4434	11
BRAM	1	2
iO	40	16
MMCM	1	20

TABLE II: GHAZI SoC's resource utilization on Artix-7

Our design didn't use any DSP slice as we went for the 3-cycle fast multiplier and not the single-cycle one. DSPs are considered expensive resource as the equivalent hardware for ASIC implementations takes up a lot of chip area. Also not mentioned with the resource utilization is the amount of global clock tree (BUFG) lines used which was 9% of the total available. This is specific to the clock frequency that was generated by an MMCM primitives as these lines drive the clock and also balance the timing delays throughout the whole design. The timing values for GHAZI SoC are shown in table III. The design was able to run at 40MHz without any total negative slack and failing endpoints.

Setup	
Worst negative slack (WNS)	94.393 ns
Number of failing endpoints	0
Total number of endpoints	11964

(a) Setup time

Hold		**Pulse Width**	
Worst hold slack (WHS)	0.025 ns	Worst pulse width slack (WPWS)	3.000 ns
Number of failing endpoints	0	Number of failing endpoints	0
Total number of endpoints	11964	Total number of endpoints	4541

(b) Hold time and pulse width

TABLE III: GHAZI SoC timing summary

When emulating GHAZI SoC on the FPGA, we used the components available on the evaluation board to map our design I/O to, as well as clock primitive "BUFGC" to replace the clock gating. The JTAG port was mapped to the board headers and the UART was mapped to the onboard FTDI serial to USB interface. The reset was provided with the one of the push button and the rest of the I/O were mapped to switches and other board headers.

V. Implementation

We used SV2V's pre-built binaries to parse all of our SystemVerilog and generate the compatible verilog for the functionality of all the OpenTitan IPs that we were using. When we used SV2V, there was no option for generating code corresponding to the declarative statements in SystemVerilog. Also, the source code/RTL did not contain any include statements which made the tool unable to find declarations and definitions, causing some errors. For this, we created a file just for the purpose of including all the other files containing the source code as shown in listing 3.

```
// Including packages first
`include "src/rv_dm/dm_pkg.sv"
`include "src/prim/prim_pkg.sv"
`include "src/tlul/tlul_pkg.sv"
// ...

// Including rest of the modules
`include "src/xbar.sv"
`include "src/ghazi_top.sv"
`include "src/buraq_core_top.sv"
`include "src/tlul/tlul_adapter_sram.sv"
`include "src/prim/prim_clock_gating.sv"
// ...
```

Listing 3: Caption

Thankfully, sv2v has a macro "SYNTHESIS" which is used for conditional compilation of the RTL to remove any non-synthesizable code. To use the generated verilog, some code still needed to be redone in order to make it work with Yosys. For example, the assert statement is non-synthesizable and as mentioned above, no code would be generated for it. But some assertions are based on conditions for which sv2v will generate empty 'if' statements which Yosys does not accept.

```
module prim_generic_clock_gating (
        clk_i,
        en_i,
        test_en_i,
        clk_o
);
input  clk_i;
input  en_i;
input  test_en_i;
output wire clk_o;
//reg    en_latch;
//always @(clk_i or en_i or test_en_i) begin
//    if (!clk_i)
//        en_latch = en_i | test_en_i;
//    else
//        en_latch = 1;
//    assign clk_o = en_latch & clk_i;
//end
sky130_fd_sc_hd__dlclkp_1 CG(
        .CLK  (clk_i),
        .GCLK (clk_o),
        .GATE (en_i | test_en_i)
);
endmodule
```

Listing 4: Clock gating module

There was one additional change that we had to make for our RTL to be ready to be transformed into GDSII format using Skywater 130nm technology. Our design included a generic clock gating primitive module that could disable the clock signal when the circuit is not in use. The resulting RTL was in the form of a latched circuit that was not library specific. To make it adhere to the guidelines for the Skywater 130nm PDK, we had to replace the logic for the latch, manually, by creating an instance of 'sky130_fd_sc_hd__dlclkp_1' in the clock gating module. This is shown in listing 4.

For the backend design, the flow consists of the following major steps. In this paper, the parameters and results of each step are discussed briefly.

A. Synthesis

It is the very first step of the design where the RTL is mapped to the gate level netlist. The process node sky130 offers different variations of the process like high speed (_hs), high density (_hd), high density low leakage (_hdll), etc. For our GHAZI SoC we have used the high density library [sky130_fd_sc_hd]. OpenLane has four design exploration strategies (DELAY 0, DELAY 1, AREA 0, AREA 1) but due to the size of GHAZI SoC, after the RTL for GHAZI SoC was translated, the synthesis strategy was kept area driven with a fan-out of 4 being used. The DELAY 0 and DELAY 1 strategies along with setting the clock period is targeted more towards meeting the timing for a design, i.e. the total and worst negative slack (TNS and WNS) should be zero which is made sure using static timing analysis (STA) and the tool that was used for this was OpenSTA. The timing strategy for GHAZI SoC, was easily met by setting the clock period to be 80ns with input and output delays being 16ns [20% of the clock]. This is shown in table IV. The resulting netlist was mapped on 124,508 standard cells.

Variable	Value
SYNTH_READ_BLACKBOX_LIB	1
SYNTH_DRIVING_CELL	"sky130_fd_sc_hd__inv_8"
SYNTH_MAX_FANOUT	"4"
SYNTH_STRATEGY	"0"
CLOCK_PERIOD	"80"
CLOCK_PORT	"wb_clk_i"
LIB_SYNTH_COMPLETE	$PDK_ROOT/sky130A/ libs.ref/sky130_fd_sc_hd/ lib/ sky130_fd_sc_hd__ tt_025C_1v80.lib

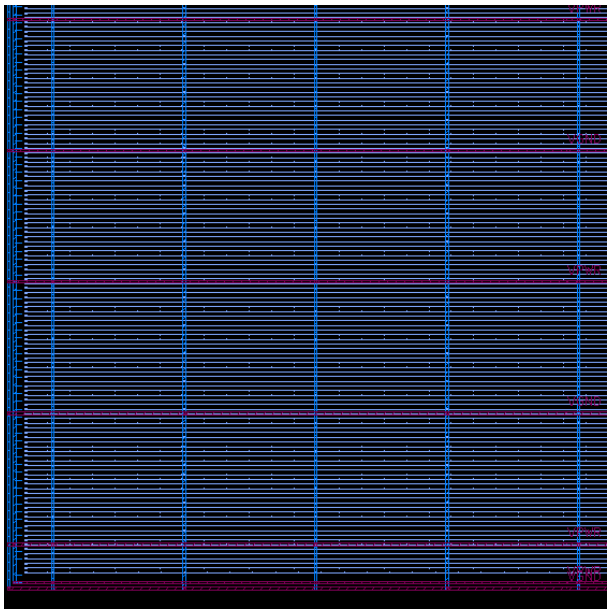
TABLE IV: Synthesis configuration

In the 6T cell based SRAM implementation of the SoC, the SRAM was not synthesized but was placed as a hardened macro, being a black-box to the synthesis tool. The GDSII file and LEF file for that SRAM, provided by OpenRAM instructed the tool according to the appropriate synthesis strategy, to just map the wire connections to the ports of the SRAM. For D Flip-Flop based RAM, we had the option of using it's hardened macro (this one being provided by Efabless) or synthesizing it with the design. Again, as the issue

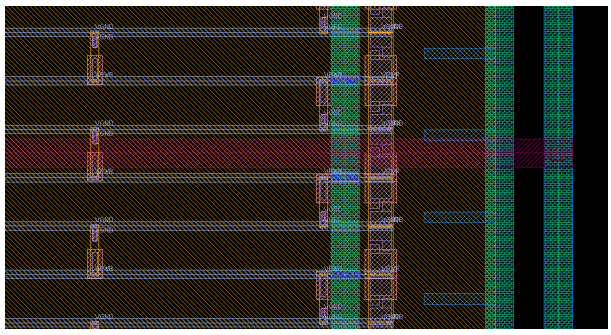
of having routing violations pertained when using the macro, we went for synthesis, but only because we had the option of running it with the design. This also led to an increased area; 0.391mm2 for the hardened macro versus 1.113mm2 (DFFRAM only) when we synthesized it ourselves.

B. Floorplan

The floorplan is the second step of the physical design where the tool defines the area. For the GHAZI SoC the die area was 6.9mm2, and the core area was the user space that is provided in Caravel for MPW shuttles, which is 10mm2. This step required the gate level netlist along with the pin order configuration file for placing the input/output (IO pads) around all the sides of the core area and outputs a design exchange format (DEF) file to be passed onto the placement stage. The step also involves the planning of the power distribution network (PDN) as shown in figure 7a. In GHAZI SoC, we also performed the step of inserting welltap cell to prevent latch-up issue between power and ground rails and decap cells, which act like decoupling capacitors to regulate a constant supply of voltage.



(a) Horizontal straps and vertical rails



(b) Core ring

Fig. 7: Power Distribution Network

1) Power grid: In the PDN, only vertical straps of metal 4 are used and metal 1 is used for the power rails. We did configure our design to purposely have a core ring due to its power requirements. This is shown in figure 7b and was achieved by setting the FP_PDN_CORE_RING environment variable in the config file, despite being hardened as a macro to be used inside the project wrapper for the user space in Caravel.

2) Maximum routing layer: We also prohibited the router from using metal 5 by setting the maximum routing layer to met4 (layer 5). This is done by setting the GLB_RT_MAXLAYER environment variable. We want to save the use of metal 5 afterwards when we would have hardened the macro and would want to place it inside the user project wrapper, to be used exclusively for the core and top level power connections.

3) Macro placement: It is also in this step that the macros are placed, but since the D Flip-flop based implementation of the SoC does not utilize any hardened macros, this is skipped. For the 6T cell based SRAM, the macros are placed after calculation which we took absolute to the core area.

C. Placement

The placement places the standard cells on the core area of the design. The result of GHAZI SoC for the placement stage is shown in figure 8. The cell pad of 8 microns is used during the placement. Cell pad is the hollow space reserved for the future use like insertion of decaps. It took four distinct steps for OpenLane to lay down the standard cells on the pre-defined site rows from the floorplan stage. All tools were invoked within OpenLane itself.

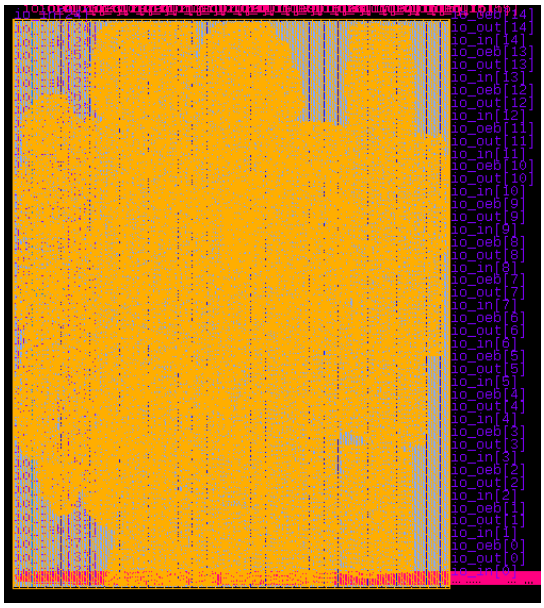


Fig. 8: GHAZI SoC after placement stage

The first one was coarse global placement, done by RePlace, in which the standard cells were placed without any specific order and were allowed to overlap. Resize tool then took up timing information from STA and the gate-level netlist

(timing library used, synthesis strategy, etc.), and resized the cells to meet the timing, power and area targets. Then comes OpenPhySyn to perform optimization and improve the quality of placement. This eventually aligned the cells. Finally OpenDP completed the placement process with the legalized and optimized design.

D. CTS

The fourth step is where the clock is routed throughout the SoC. Due to timing, power and signal integrity implications, different topologies are used to minimize the overheads. Low-latency clock distribution is performed and for the GHAZI SoC H-Tree topology is used. This has low power consumption with lower skew. We had to make sure that the RC delay was balanced which led us to three challenges:

1) Exceeding power usage
2) Large RC value for each net
3) Signal integrity at risk

Though a single tool, called TritonCTS was up to this task, it did follow the detailed placement and used OpenDP internally afterwards to legalize clock buffers inserted during CTS.

E. Routing

Routing connects all the standard cells with each other. We have different layers of metal in the PDK. The Sky130 nm PDK has 6 layers [1 local interconnect and 5 metal layers]. GHAZI SoC utilizes up to metal 4. The parameters that we looked for in global routing were to analyze routing congestion, identify available paths and minimize detouring. Fast route takes LEF and placed-DEF files and defines global routing cells.

For GHAZI SoC, we had to keep an eye out for minimizing the congestion and overflow within the cells that could have occurred anywhere between the process. The tool generated the route guides necessary for detailed routing which is done by TritonRoute[14]. The routed design for GHAZI SoC can be seen in figure 9. Similar to placement, the routing is done in two steps, global and detailed. All the violations were completely resolved in the fifty-seventh iteration for optimizing the legalized routes as shown in listing 5.

We observed from the utilization that metal 2 and metal 3 were used most for routing. Table V shows this.

Routing Layer Name	Utilization [%]
routing_layer1	0
routing_layer2	26.2
routing_layer3	28.25
routing_layer4	2.69
routing_layer5	1.17

TABLE V: Routing layer utilization

F. GDSII Stream Out

This step outputs the GDS file and runs different checks on it. Figure 10 shows the GDSII layout of GHAZI SoC. The design is checked for a number of things like design rule checks, antenna violations, layout vs schematic checks,

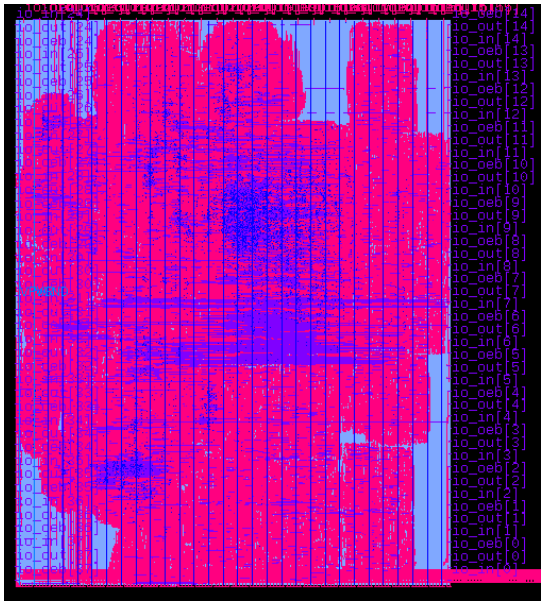


Fig. 9: GHAZI SoC after routing stage

total negative slack, and routing violations. Out of these, TNS is checked throughout all the physical design stages though. The GDS layout of GHAZI SoC was passed on to Magic for extracting the SPICE layout and was found to have zero dissimilarities in the LVS check. The process avoided insertion of blind antenna diodes and used dummy antenna cells to preserve the core utilization of GHAZI SoC. This resulted in about 644 antenna violations and forced us to insert back the real antenna diodes in place of some of the dummy ones.

```
...
start 57th optimization iteration ...
    completing 10% with 0 violations
    elapsed time = 00:00:12,
    memory = 2983.94 (MB)
    completing 20% with 0 violations
    elapsed time = 00:00:25,
    memory = 2983.94 (MB)
    completing 30% with 0 violations
    elapsed time = 00:00:58,
    memory = 2983.94 (MB)
    completing 40% with 0 violations
    elapsed time = 00:01:50,
    ...
    completing 100% with 0 violations
    elapsed time = 00:02:06,
    memory = 2983.94 (MB)
  number of violations = 0
cpu time = 00:08:23, elapsed
time = 00:02:06, memory =
2983.94 (MB), peak = 4331.74 (MB)
total wire length = 9212890 um
total wire length on LAYER
...
```

```
...
li1 = 2191 um
total wire length on LAYER
met1 = 3793617 um
total wire length on LAYER
met2 = 4887007 um
total wire length on LAYER
met3 = 281119 um
total wire length on LAYER
met4 = 248954 um
total wire length on LAYER
met5 = 0 um
total number of vias = 954356
up-via summary (total 954356):
```

Listing 5: Log output of TritonRoute

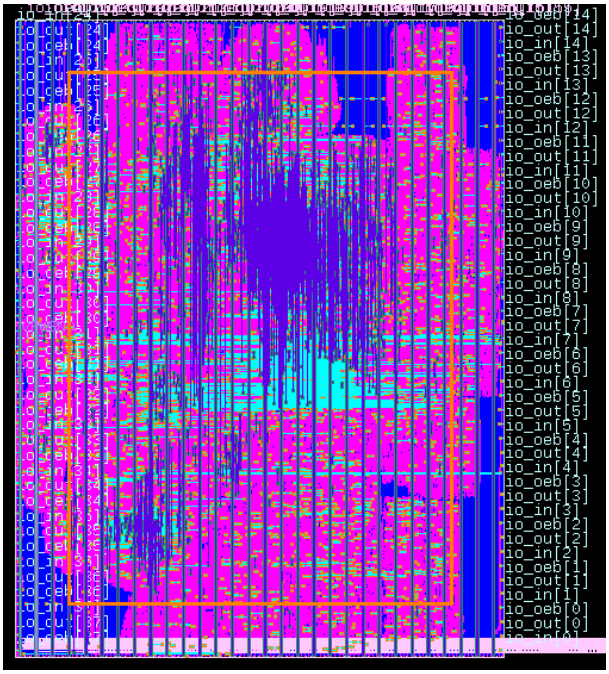


Fig. 10: GDSII layout of GHAZI SoC

For the MPW part, the now hardened GHAZI SoC was placed inside the user project wrapper and routed with the rest of the components (the pre-hardened ones) inside the Caravel design. GHAZI SoC, placed inside caravel is shown in figure 11.

This step generated the final GDS to be sent to the SkyWater foundry, with some preporcessing done by the collaborating team such as inserting filler cells and generating the mask layout of the design.

VI. Conclusion

We have successfully taped out a System on Chip design using all open-source tools all the way from the design stage, where we performed conversions on the RTL and ran simulations on it, to the auto-place and route stage and to the physical verification, the part where we checked the design

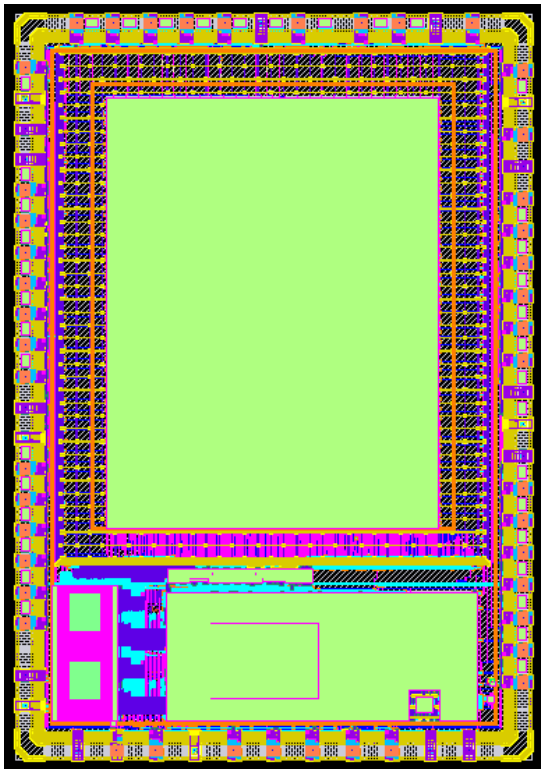


Fig. 11: GHAZI SoC inside Caravel

for DRC violations according to the Sky130A cells, performed LVS on the design and from earlier on where we performed the parasitic extraction. For future possibilities, the scope of this paper suggest that we could do a comparision between OpenLane and commercial EDA tools and how effective it is as commercial EDA tools have timing awareness whereas the synthesis step in OpenLane is not timing driven.

References

[1] Tutu Ajayi and David Blaauw. "OpenROAD: Toward a self-driving, open-source digital layout implementation tool chain". In: *Proceedings of Government Microcircuit Applications and Critical Technology Conference*. 2019.

[2] Tutu Ajayi et al. "Toward an open-source digital flow: First learnings from the openroad project". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–4.

[3] *Caravel Harness*. https://github.com/efabless/caravel.

[4] Nitin Dahad. *Can Open Source Hardware Emulate Linux?* https://www.eetimes.com/can-open-source-hardware-emulate-linux/. 2021.

[5] R. Timothy Edwards. *open_pdks, magic and netgen*. http://opencircuitdesign.com/.

[6] Tim Edwards. *a03 Google/SkyWater and the Promise of the Open PDK*. https://youtu.be/nY6FVsAvrUo/. 2020.

[7] Tim Edwards. *PicoSoC: How We Created A RISC V Based ASIC Processor Using A Full Open Source Foundry Targeted RTL-to-GDS flow*. https://youtu.be/EsEcLZc0RO8/. 2017.

[8] FOSSi Foundation. *45 Chips in 30 Days: Open Source ASIC at its best!* https://www.fossi-foundation.org/2021/03/07/45-chips-in-30-days. 2020.

[9] Michael Gautschi et al. "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), pp. 2700–2713.

[10] Matthew R Guthaus et al. "OpenRAM: An open-source memory compiler". In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2016, pp. 1–6.

[11] *Icarus Verilog*. iverilog.icarus.com.

[12] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language - Redline". In: *IEEE Std 1800-2009 (Revision of IEEE Std1800-2005) - Redline* (2009), pp. 1–1346.

[13] Andrew B Kahng. "Open-Source EDA: If We Build It, Who Will Come?" In: *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*. IEEE. 2020, pp. 1–6.

[14] Andrew B Kahng, Lutong Wang, and Bangqi Xu. "TritonRoute: The Open-Source Detailed Router". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.3 (2020), pp. 547–559.

[15] Josh Lerner and Jean Tirole. "Some simple economics of open source". In: *The journal of industrial economics* 50.2 (2002), pp. 197–234.

[16] *lowRISC Verilog Coding Style Guide*. https://github.com/lowRISC/style-guides/blob/master/VerilogCodingStyle.md. 2020.

[17] *lowRISC: Collaborative open silicon engineering*. https://lowrisc.org/.

[18] Ted Marena. "RISC-V: high performance embedded SweRV™ core microarchitecture, performance and CHIPS Alliance". In: *Western Digital Corporation* (2019).

[19] Ross Miller. "OPEN SOURCE ASICS TAKE A GIANT LEAP FORWARD WITH THE FIRST EVER OPEN FOUNDRY PDK". In: *GSA Global Forum* (2020).

[20] Laurence W Nagel. "SPICE2: A computer program to simulate semiconductor circuits". In: *Ph. D. dissertation, University of California at Berkeley* (1975).

[21] Borivoje Nikolic, Elad Alon, and Krste Asanovic. "Generating the next wave of custom silicon". In: *ESSCIRC 2018-IEEE 44th European Solid State Circuits Conference (ESSCIRC)*. IEEE. 2018, pp. 6–11.

[22] *Open Source Shuttle MPW-ONE*. https://efabless.com/open_shuttle_program.

[23] *Open source silicon root of trust (RoT) — OpenTitan*. https://github.com/lowRISC/opentitan.

[24] *OpenLANE*. https://github.com/The-OpenROAD-Project/OpenLane.

[25] *OpenROAD - Foundations and Realization of Open, Accessible Design*. https://theopenroadproject.org/.

[26] John K Ousterhout et al. "The magic VLSI layout system". In: *IEEE Design & Test of Computers* 2.1 (1985), pp. 19–30.

[27] Anne-Françoise Pelé. *Building an ecosystem for open-source hardware*. https://www.eetimes.com/building-an-ecosystem-for-open-source-hardware/. 2007.

[28] Dominic Rizzo. "OpenTitan at One Year: the Open Source Journey to Secure Silicon". In: *Google Security Blog* (2020).

[29] Pasquale Davide Schiavone et al. "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications". In: *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE. 2017, pp. 1–8.

[30] Mohamed Shalan and Tim Edwards. "Building OpenLANE: a 130nm openroad-based tapeout-proven flow". In: *Proceedings of the 39th International Conference on Computer-Aided Design*. 2020, pp. 1–6.

[31] *SkyWater Open Source PDK*. https://github.com/google/skywater-pdk.

[32] Zachary Snow. *sv2v: SystemVerilog to Verilog*. https://github.com/zachjs/sv2v.

[33] Jennifer E Sowash. *Design of a RISC-V Processor with OpenRAM Memories*. University of California, Santa Cruz, 2019.

[34] *The Free and Open Source Silicon Foundation*. https://www.fossi-foundation.org/.

[35] Andrew Waterman et al. "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1". In: (2016).

[36] Stephen Williams. *How Icarus Verilog Works*. https://github.com/steveicarus/iverilog.

[37] Claire Wolf. *Yosys Open SYnthesis Suite*. http://www.clifford.at/yosys/.