IBTIDA: Fully open-source ASIC implementation of Chisel-generated System on a Chip

Preprint · September 2021

DOI: 10.36227/techrxiv.16663738.v1

5 authors, including:

Muhammad Hadir Khan
University of California, Santa Cruz
5 PUBLICATIONS **4** CITATIONS
SEE PROFILE

Aireen Amir Jalal
University of Arkansas at Fayetteville
5 PUBLICATIONS **20** CITATIONS
SEE PROFILE

Sajjad Ahmed
University of Salento
14 PUBLICATIONS **123** CITATIONS
SEE PROFILE

IBTIDA: Fully open-source ASIC implementation of Chisel-generated System on a Chip

SUBMISSION DATE / POSTED DATE

22-09-2021 / 30-09-2021

CITATION

Khan, Muhammad Hadir; Jalal, Aireen Amir; Ahmed, Sajjad; Ansari, Ali Ahmed; Naqvi, Syed Roomi (2021): IBTIDA: Fully open-source ASIC implementation of Chisel-generated System on a Chip. TechRxiv. Preprint. https://doi.org/10.36227/techrxiv.16663738.v1

DOI

10.36227/techrxiv.16663738.v1

IBTIDA: Fully open-source ASIC implementation of Chisel-generated System on a Chip

Muhammad Hadir Khan, Aireen Amir Jalal, Sajjad Ahmed, Ali Ahmed, and Syed Roomi Naqvi

***Abstract*—Building a System on Chip (SoC) using a fully open-source toolchain requires the availability of open-source tools for RTL simulation, generation, GDS-II conversion, manufacture-able foundry process design kits (PDKs), IP libraries, and I/O blocks. The proposed work shows the methodology of using completely open-source tools and hardware construction language (HCL) to tape-out RISC-V based SoC - Ibtida. The methodology utilizes Chisel (Constructing Hardware in Scala Embedded Language) as the RTL generator, Verilator as the RTL simulator, OpenLANE as the RTL to GDS-II converter, and SKY-130nm Open PDK to manufacture the SoC. Ibtida consists of a 5-stage pipelined 32-bit RISC-V (RV32IM) core with 32 GPIOs, and separate instruction and data memories. The Ibtida design is embedded in a harness on a physical chip. The harness is equipped with a management SoC used as a controller to the Ibtida. Prior to converting the RTL into GDS-II, the cycle-accurate simulation using Verilator and FPGA emulation on Xilinx ARTY A7 has been performed for verification and regression testing. The FPGA implementation utilizes 8650 LUTs, 3356 Slice Registers, 714 flip flops, and 2.5 Block RAM of 36Kb. The ASIC implementation utilizes a 2.5 mm2 area with a density of 37.44 KGate/mm2. The manufacturing of this SoC is provided by Google shuttle program called Open MPW (Multi Project Wafer) in association with Efabless and SkyWater technologies. To the best of our knowledge, this is the first RISC-V based SoC, generated using Chisel and taped-out using fully open-source technologies.**

***Index Terms*—OpenLANE, OpenROAD, Chisel, RTL, FPGA, SoC, Open-source hardware, RISC-V.**

I. Introduction

TODAY, Moore's law is diminishing. The trend of increasing computing capabilities by doubling the number of transistors is coming to a halt [1]. Due to this, we are entering the golden age of computer architecture [2] where the key driving force for the pursuit of increased performance has been other than the only miniaturization. Although, due to the proprietary nature of chip designing, the innovation has been somewhat limited due to the fact that only big companies can design their own processors. This was democratized by the advent of RISC-V Instruction Set Architecture (ISA) [3] which enabled startups and communities to work together in chip designing. Still, there was another barrier for academic researchers, startups, and small companies to actually tape-out their processors, that is, the close nature of Process Design Kit (PDK). From the past, there have been many open-source Electronic Design Automation (EDA) tools (SPICE, Magic, etc.) available for the physical design engineers but the lack of a completely open-source PDK kept the custom hardware design to a handful of large and established companies and well-funded research universities. However, this problem has also been resolved recently in mid-2020, when SkyWater foundry together with Google introduced the first fully open-source PDK the SKY130 process node [4] which is based on a 130nm Complementary Metal Oxide Semiconductor (CMOS) technology.

A. The open-source hardware momentum

Since the arrival of the RISC-V ISA, there has been a boom in the open-source chip designing domain. It proved that like open-source software, open-source hardware can be greatly improved by a collaborative effort between small and big companies complementing each other and not only improving the ISA but also the other tools ecosystem required for hardware designing [5].

The ChipsAlliance [6] established in 2019, takes the aim of open-source hardware designing even further. It provides a commonplace for designers to create innovative solutions using open-source tools. It has renowned companies as members working together to develop reusable open-source IPs. It is also focused on providing tools for open-source physical design. The very ambitious open-source OpenROAD project [7] is also part of the ChipsAlliance that aims to provide 24-hour, No-Human-In-The-Loop layout design for SOC, Package, and PCB with no Power-Performance-Area (PPA) loss, enabling software engineers and people with scarce physical design knowledge to tape-out their own processors.

The availability of everything open-source from RTL to EDA tools still hindered the complete flow of open chip designing due to the nonexistence of a completely open-source PDK. For over twenty years, the PDKs have been kept closed source and required non-disclosure agreements (NDAs), license servers, and password-protected download sites causing the privilege to tape-out designs at the hands of only big established companies [8]. But with the SkyWater foundry opening up their design for a 130nm process together with Google and the Efabless/Google collaboration for providing free tape-out shuttles, presents a huge opportunity for startups, small academic institutes, and even high school students to come up with their custom unique designs and actually get them fabricated.

B. Why hardware should learn from software

Due to the halt in performance even after doubling the number of transistors, the era of domain-specific architecture is booming. The advent of the RISC-V ISA has enabled small teams and startups to develop custom hardware to improve performance and efficiency in terms of power consumption.

However, the process for designing chips has been painfully long and involved a rigid development model that earlier software development followed, known as the waterfall model. The software created in the early days suffered from over-budget, not meeting deadlines, and being abandoned. Making changes to the whole monolithic software project was very difficult as the customer's needs changed. The same goes for hardware projects. In a hardware project, first, the micro-architecture is specified, followed by the RTL design after which the verification happens, and then the complete physical design of the netlist is done. Usually, the physical design is even outsourced to other companies which further increases the timeline of the projects usually ranging from 1-3 years, and if the customer's need changes the whole process needs to be repeated. The agile software methodology [9] emphasizes on working software over detailed documentation, customer collaboration, and being flexible over rigid specifications. It promotes small teams working iteratively on improving working-but-incomplete prototypes and enhancing them until the end result is acceptable. Inspired by this agile software approach, the researchers at the University of California, Berkeley proposed their own "Agile Hardware Manifesto" [10] through which they taped-out eleven processors in a span of five years.

To facilitate this agile hardware development idea by increasing designer productivity, Chisel [11] was created. It is a domain-specific language created on top of Scala which provides all the high-level programming features such as Object-Oriented Programming (OOP) and Functional Programming (FP) to the designer for creating reusable libraries that generate efficient hardware circuits. The idea is to create reusable packages just like in software which provides abstraction and easy-to-use integration opportunities of various verified IPs. Furthermore, the Chisel compiler automatically creates a fast, cycle-accurate C++ software simulator, or low-level synthesizable Verilog that maps to FPGAs or ASIC flows.

C. Previous works

There have been eleven tape-outs based on Chisel utilizing the Rocket-chip generator [12] by the University of California, Berkeley but were based on commercial EDA tools and closed PDKs. Also, a family of striVe SoCs was taped out using the OpenLANE and Skywater 130nm PDK to prove the viability of all open-source EDA tools and the PDK [13]. However, it is written in a traditional low-level hardware description language, Verilog. The Rocketchip generated tape-outs were missing the open-source backend flow to generate the GDS and the striVe family SoCs although mapped on the open PDKs, lacked the frontend design written in a higher-level programming language.

In this paper, we present our contribution by using the abstractness and software programming feel of Chisel to tape-out a 5-stage pipelined RISC-V RV32IM core and a minimal SoC around it with no prior experience in chip designing and passed the generated RTL, Verilog, to OpenLANE [14] to provide a completely open-source RTL-GDS flow which was then mapped onto the fully open-source SkyWater 130nm process design kit through the Google/Efabless MPW Shuttle program [15]. We used Chisel for the ease of programming hardware circuits providing us a quickstart with RTL designing as compared to the low-level Verilog and proved that the generated Verilog can be mapped to the fully open suite of Electronic Design Automation (EDA) tools and can be fabricated on the Skywater 130nm open PDK.

II. Design Methodology and Specification

To prove our work proposed in the paper we followed a methodology on a design specification and analyzed its implementation and results. In the following sub-sections, we will discuss the methodology and specification of the design later delving into other sections for details related to the implementation and analysis of the design.

A. Methodology

Chisel was used as a frontend of the proposed design which is a domain-specific language embedded inside Scala that provides higher functionality of a programming language to design circuits instead of traditional HDLs like Verilog/VHDL [16]. The Chisel front-end generates an Intermediate Representation (IR) called Flexible Intermediate Representation for RTL (FIRRTL) which provides certain transforms and passes based on Scala [17] that runs on top of the Java Virtual Machine (JVM) which transforms the same Chisel code to be used into three different backends: 1) Simulation, 2) FPGA Emulation and 3) ASIC Implementation

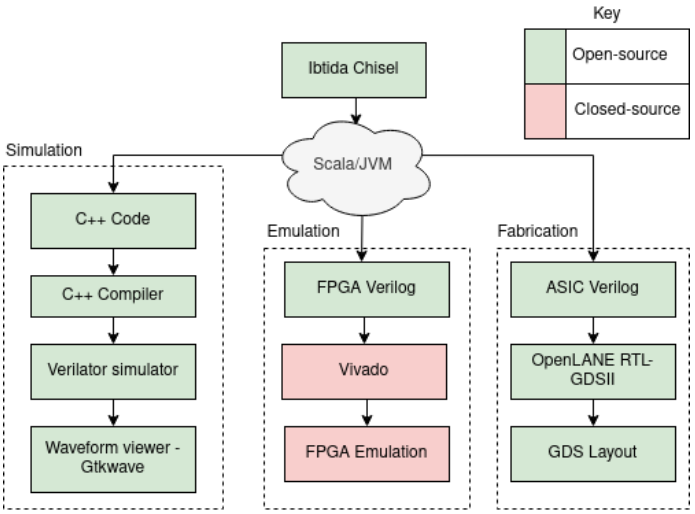

Fig. 1. Overview of different stages used in Ibtida during the tape-out of the design

For simulation, to check the functionality of the design, the Chisel compiler was used to generate a C++ simulator based on the emitted Verilog of the SoC through Verilator [18] and emitted C++ wrapper for providing stimuli to the compiled simulator, finally running the simulator to generate a Value Change Dump (VCD) file that can be viewed on an open-source waveform viewer GTKWave [19].

For emulating on the FPGA, the Chisel generated Verilog was mapped on the Arty A7 FPGA board using Xilinx's

Vivado for synthesizing, placing and routing, and generating the bitstream to be mapped on the board. This is the only closed source path that was used for emulation. However, an open-source alternative for the FPGA implementation exists as well such as the Symbiflow project [20] or OpenFPGA [21] but that is not the scope of this paper.

For the ASIC, the generated Verilog and SkyWater 130 nm PDKs were used along with the OpenLANE flow comprising of various open-source tools for Synthesis, Floorplan, Power Distribution Network (PDN) generation, Place and Route, Design Rule Check (DRC), Layout Versus Schematic (LVS) checks and GDSII generation.

B. Specification

Ibtida is a minimal System on a Chip designed completely with Chisel using the higher programming language features. It consists of four basic elements that every computer has: 1) Compute, 2) Communication, 3) Peripherals, and 4) Storage. The instruction interface has a Point-Point interconnect for fetching instructions and the data interface has a 1xN interconnect that allows the core to either perform loads/stores to the memory or to the GPIO peripheral. Since there is no non-volatile memory present for code storage, a UART controller is designed to accept the program from the host computer and writes it into the ICCM memory every time the board is powered on or a new program needs to be uploaded.

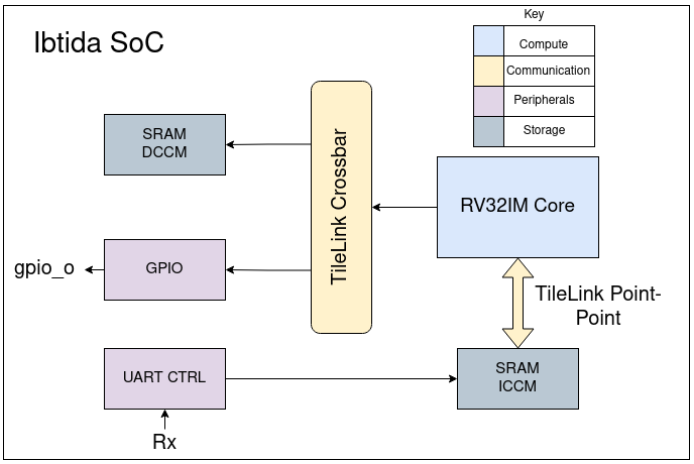

Fig. 2. Ibtida System on a Chip block diagram

The details of each element highlighted in figure 2 above are described below:

1) Compute: It is a 32 bit 5-stage pipelined core compliant with the RISC-V base ISA I-type extension and an additional M-type extension that supports multiply/divide instructions together becoming an RV32IM supported core. It has five pipelined stages: 1) Fetch (F). 2) Decode (D). 3) Execute (E). 4) Memory (M). 5) WriteBack (WB).

a) Fetch: The fetch has a Program Counter (PC) that points to the next instruction to be fetched and an interface to fetch the instructions from the memory. The PC value is updated through a multiplexer that selects the next PC value which can be a simple PC + 4 through an adder or another jump address depending upon the instruction in the Decode stage.

b) Decode: The decode stage consists of a register file with 32 registers x0 to x31 each 32 bits wide as described in the RISC-V ISA. It also has an Immediate Generation unit that extracts the encoded immediate values from the instructions, concatenating and padding them to become 32 bits wide. There is a Control Unit as well that decodes the current instruction using the opcode and enables certain control signals depending upon the type of instruction. There is a Branch Unit that identifies if the current instruction is a branch instruction and calculates the next PC address if the branch is taken. The Branch Unit was kept in the Decode stage to improve the branch miss penalty to 1 cycle if the branch is taken since the fetch would need to be flushed and the new instruction needs to be fetched from the updated PC value. It also has a Hazard Detection logic unit that prevents structural hazards from happening i.e if the register being accessed by the current instruction is also being written at the same time by another instruction in the Write Back stage.

c) Execute: The execute stage has an Arithmetic Logic Unit (ALU) for computation-related tasks and an ALU Control unit indicating the ALU as to which operation needs to be performed. It also has a forwarding unit that is used to provide the ALU with proper operands if there are any data hazards in the pipeline.

d) Memory: The memory stage consists of a store/load unit that performs either stores or loads to the memory or the GPIO peripheral.

e) Write Back: The write back stage consists of a mux that selects the data to be written in the register file which can be either from the ALU output or from the data memory.

2) Communication: The communication mechanism used between the core, peripherals, and memories is TileLink Uncached Lightweight (TL-UL) bus protocol [22]. The miniature version of TileLink, the TL-UL was used since we did not require cache coherency and other complex communication. The fetch stage sends a valid request to the TL-UL Master which then communicates with the TL-UL Slave that is then connected with the instruction memory. This forms a Point-Point interconnection between the core's fetch and instruction memory as shown in figure 2. For load/stores during the memory stage, a 1xN switch is used to connect a single TL-UL Master with multiple TL-UL Slaves which are two in our case. One for the data memory and the other for the GPIO peripheral. The 1xN switch automatically decodes which slave to route the master's request to depending upon the address issued. There is no support for burst accesses. The master can only send one request at a time and wait for the acknowledgment before sending another request. The write back stage consists of a mux that selects the data to be written in the register file which can be either from the ALU output or from the data memory.

3) Peripherals: The SoC contains only one peripheral that is the GPIO connected to the bus. The GPIO has 30 I/O pads going outside to interact with the outside world. Its control and status registers (CSRs) are accessible via TL-UL bus which can be manipulated by the software program running on the

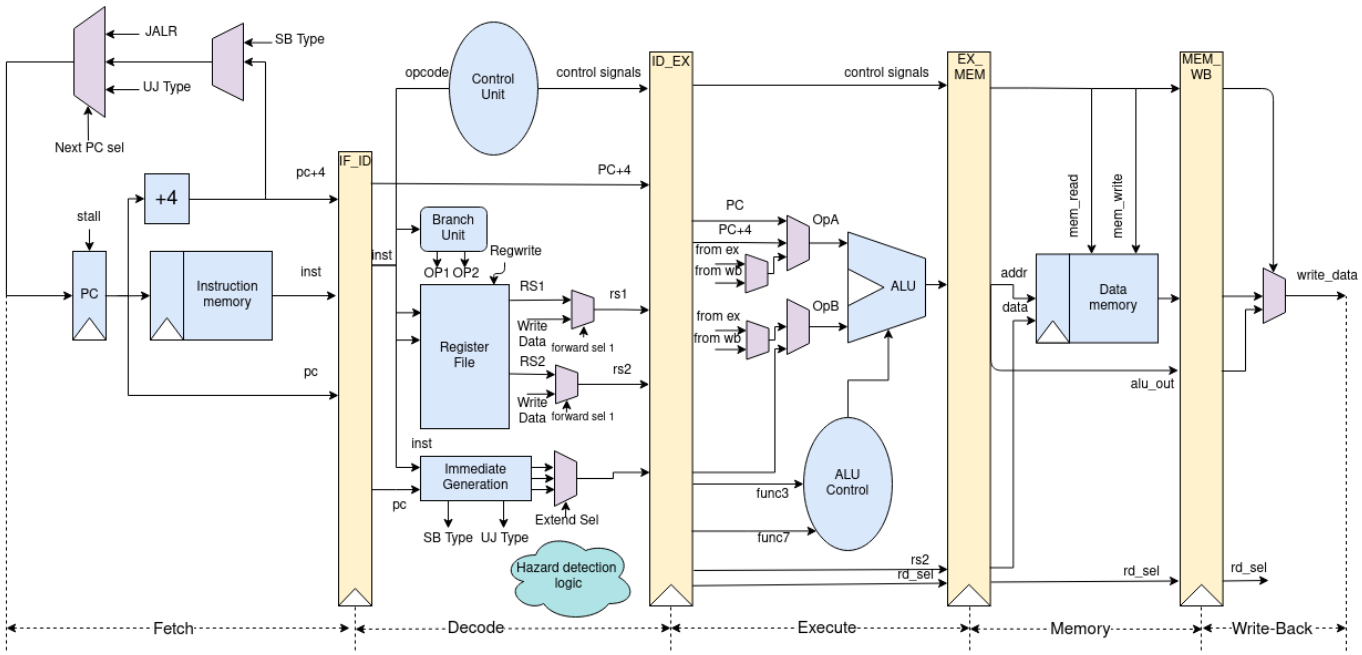

Fig. 3. The RISC-V ISA compliant RV32IM 5-Stage fully pipelined datapath designed from scratch with Chisel

core. The other standalone peripheral is the UART that is used to program the instruction memory.

4) Storage: The SoC has a Harvard Architecture consisting of separate Instruction Closely Coupled Memory (ICCM) and Data Closely Coupled Memory (DCCM). Both of the memories are 1Kbyte in size and are accessible via TL-UL bus.

III. Implementation and Analysis

A. Verilator Simulation

For testing the functionality of the design, Verilator was used to simulate the SoC and each of it's individual components. The listing 1 shows how a 2-way mux can be designed in Chisel.

```
package myMux

class Mux extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val sel = Input(UInt(1.W))
    val o = Output(UInt(8.W))
  })
  io.o := DontCare
  switch(io.sel) {
    is(1.U) {
      io.o := io.b
    }

    is(0.U) {
      io.o := io.a
    }
  }

}
```

Listing 1: Design of a 2-way Multiplexer in Chisel

The verification of the Mux can be done by creating a testbench in Chisel as shown in listing 2.

```
package myMux

class MuxTester(c: Mux) extends PeekPokeTester(c) {
    poke(c.io.a, 2)      // any random value
    poke(c.io.b, 4)      // any random value
    poke(c.io.sel, 0)    // io.a to output
    expect(c.io.o, 2)    // expecting the output
    step(1)              // after one clock edge
    poke(c.io.a, 2)      // any random value
    poke(c.io.b, 4)      // any random value
    poke(c.io.sel, 1)    // io.b to output
    expect(c.io.o, 4)    // expecting the output
}
```

Listing 2: Testbench to verify a 2-way Mux in Chisel

A driver class as shown in listing 3 is used to configure the Scala backend to use verilator for testing and an additional flag is used to generate the VCD trace for waveform view.

Scala build tool (sbt) is utilized to compile the Scala classes and execute them as shown in listing 4 which in turn builds all the verilator files using the testbench and generates a VCD trace to view.

The generated VCD trace can be viewed on GTKWave. In figure 4 the resulting waveform for the mux is depicted.

Similarly, each module within the Ibtida SoC was tested for its correct functionality using Chisel-based testbenches and Verilator based simulation. In table I, a RISC-V assembly program for the sake of testing is shown that is run on the SoC, and figure 6 shows how the instructions passes through the pipeline with only the important signals extracted for ease. The whole test suite run on the Ibtida SoC is present on Github. [23]

Initially, as shown in the figure 6 the UART programmer loads the program into the instruction memory and asserts

```
package myMux

class MuxSpec extends FlatSpec with Matchers {
    behavior of "MuxSpec"
    it should "work correctly" in {
        chisel3.iotesters.Driver.execute(
        Array("--backend-name", "verilator",
              "--generate-vcd-output", "on"),
              () => new Mux) { c =>
            new MuxTester(c)
        } should be(true)
    }
}
```

Listing 3: Wrapper class for building test and creating VCD trace

```
sbt "testOnly myMux.MuxSpec"
```

Listing 4: sbt command to build the verilator simulator and VCD trace of the simulation

TABLE I
RISC-V ASSEMBLY PROGRAM

PC	Machine Code	Program
0x0	0x00300193	addi x3, x0, 3
0x4	0x00200213	addi x4, x0, 2
0x8	0x003202B3	add x5, x4, x3

uart_done high signaling that the memory is loaded. The fetch then sends a valid request with the PC's current value and gets the instruction in the next cycle. Until then a NOP (No operation) instruction is sent to the datapath that does nothing in the pipeline. After this, on each clock cycle, a new instruction is fetched and previous instructions progress through in the pipeline. Finally, the registers get loaded with the values coming from the write back stage.

B. FPGA Emulation

The generated Verilog of Ibtida SoC from Chisel was mapped on the Arty A7 FPGA board. It runs on 8MHz frequency with no total negative slack (TNS) and failing endpoints. Table II shows the timing report of the implemented design.

TABLE II
TIMING REPORT OF IBTIDA SOC

Setup	Hold	Pulse Width
Worst Negative Slack: 1.826ns	Worst Hold Slack: 0.028ns	Worst Pulse Width Slack: 3.000ns
Total Negative Slack: 0.000ns	Total Hold Slack: 0.000ns	Total Pulse Width Negative Slack: 0.000ns

The MMCM primitive was used as the clock generator to provide the clock to the design. The ICCM and DCCM memories were mapped into FPGA Block Rams (BRAMs). The DSP units inside the board were used for efficient multiplication. The resource utilization of the design is given in Table III

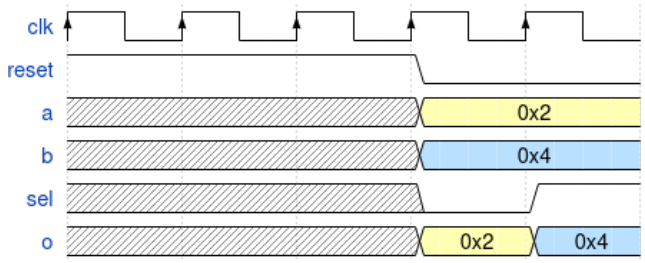

Fig. 4. Waveform illustration of a 2-way Mux

The power consumption of the implementation is given in Table IV.

TABLE III
RESOURCE UTILIZATION REPORT OF IBTIDA SOC

Resource	Utilization	Available	Utilization %
LUT	8650	20800	41.59
FF	3356	41600	8.07
BRAM	2.50	50	5.00
DSP	12	90	13.33
IO	19	210	9.05
MMCM	1	5	20.00

TABLE IV
POWER CONSUMPTION REPORT OF IBTIDA SOC

Elements	Power in Watts (W)	Power in %
Clocks	0.001	1
Signals	0.008	7
Logic	0.006	5
BRAM	<0.001	1
DSP	0.001	1
MMCM	0.106	86
I/O	<0.001	0

Power Type	Power in Watts (W)	Power in %
Dynamic	0.123	66
Static	0.062	34

C. ASIC Implementation

For the ASIC implementation, the Chisel-generated Verilog was integrated inside a testing harness and then hardened through the OpenLANE flow for generating the GDSII layout.

1) Testing Harness: Caravel [24], is a testing harness that acts as a manager of the Ibtida SoC. It has three parts in it: 1) Management Area 2) User Project Area 3) Storage Area as shown in figure 5.

a) Management Area: The management area consists of an SoC built on top of a RISC-V based microprocessor PicoRV32 [25]. It has some peripherals including timers, uart, and gpio. The firmware on the management area can be used

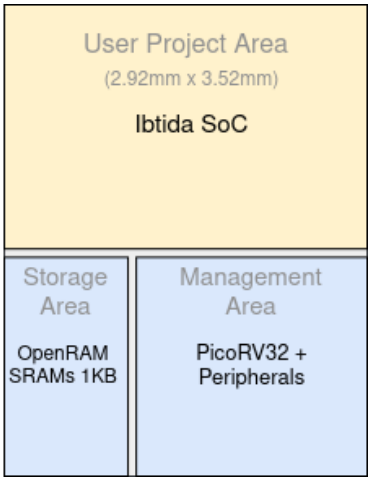

Fig. 5. Internal organization of Caravel test harness

to configure the User Project I/O pads, analyze and control User Project signals through on-chip logic analyzer probes, and control the User Project power supply.

b) User Project Area: This is the space for user design in which the Ibtida SoC was integrated. The silicon area for the User Project is 2.92mm x 3.52mm. It has a fixed number of I/O pads 38 and power pads 4.

c) Storage Area: It consists of two dual port SRAMs of size 1Kbyte generated by OpenRAM [26]. The storage area is only accessible to the management area.

Figure 7 shows the architecture of the Caravel harness. The management area contains peripherals on a Wishbone Bus[27] which are written/read by the PicoRV32. There is also *Chip LA* which is a memory-mapped 128 bits wide logic analyzer on a wishbone bus. It can be configured to read data from the User Project Area or provide any data to it. There is also a Wishbone slave interface inside the User Project Area but we used it only to provide the clock and reset to Ibtida SoC coming via the Wishbone master interface on the management area. The User Project Area has access to the 38 GPIOs after they are configured to be usable by the firmware running on the management core.

2) Integrating Ibtida inside Caravel: Figure 8 shows the configured Ibtida SoC for integrating inside the Caravel User Project Area. The signals prefixed *la* are coming from the logic analyzer. The SRAMs were mapped onto technology-specific flip flop based DFFRAMs.

3) Openlane: RTL to GDS: OpenLANE is an open-source automated RTL-GDSII ASIC design flow based on several components, PDK (Process Design Kit), and IP (Intellectual Property) libraries including standard-cell libraries, that perform steps from RTL synthesis all the way to GDS streaming. It is an aggregation of open-source EDA (Electronic Design Automation) tools explicitly OpenROAD, Yosys[28], Magic[29], Netgen[30], OpenPhySyn[31] and SPEF-Extractor[32]. Furthermore, a custom script is being used for design exploration and optimization. The completely open-source flow was designed in accordance with the open PDK, open-sourced by Google and SkyWater (Sky130 PDK) on a 130nm CMOS technology, but concurrently is generalized to support other technologies. The flow performs full ASIC implementation steps from RTL to GDSII, which includes: 1) Logic Synthesis 2) Floor-Planning 3) Placement 4) Clock Tree Synthesis (CTS) 5) Routing 6) SPEF-Extraction 7) GDSII Generation 8) Physical Verification as shown in figure 9.

The output of synthesis is a gate-level netlist which after floor-planning results in a def (Design exchange format) file, comprising information related to physical layout i.e. pin placement, die area, and the core area of a specific design. During further stages in the flow, the def file gets updated multiple times as standard cells get placed during the placement, and information regarding the coordinates of their placement is added. The final def gets generated after routing where the track information connecting the standard cells is added.

The GDS then gets generated followed by Design Rule Check (DRC) and Layout vs Schematic (LVS) check which is required for the physical verification.

a) Logic Synthesis: The first step towards attaining a hardened Ibtida IP involves logic synthesis. This process specifically focuses on acquiring the RTL along with the standard cell library files. Before the synthesis of Ibtida RTL, the OpenLANE environment needs to be set up as shown in listing 5 followed by the commands shown in listing 6.

```
git clone
> https://github.com/The-OpenROAD-Project/OpenLane.git
cd OpenLane
make mount
```

Listing 5: Setting up the OpenLANE environment

```
./flow.tcl -interactive
prep -design Ibtida
run_synthesis
```

Listing 6: Synthesis script

The flow can be executed in interactive mode, by the *prep -design <design_name>* command which sources the design configuration file; *config.tcl*, where it reads the specified environment variables (VERILOG_INCLUDE_DIRS, SYNTH_STRATEGY, etc.) required for synthesis and merges the relevant library exchange format (LEF) and technology LEF file i.e. technology-specific files information along with the generated Verilog RTL and passes it as input to Yosys and ABC which synthesizes the logic and maps it on to the technology-specific standard cells respectively as seen in figure 10.

The chip area calculated after synthesis is 1.25mm^2. Furthermore, Table V shows the statistics of the generated netlist.

OpenLANE provides a set of design exploration strategies, which enables designers to achieve the design specifications in terms of performance and area. There are four design exploration strategies offered by OpenLANE which have been tested for Ibtida SoC. The strategies provide a trade-off between the area and timing. Strategies 0 and 1 (delay) explicitly focuses on achieving a better performance in terms of timing whereas 2 and 3 (area) strategies focus on getting a better (compact) area. The effect of different design exploration strategies for Ibtida SoC is shown in figure 11.

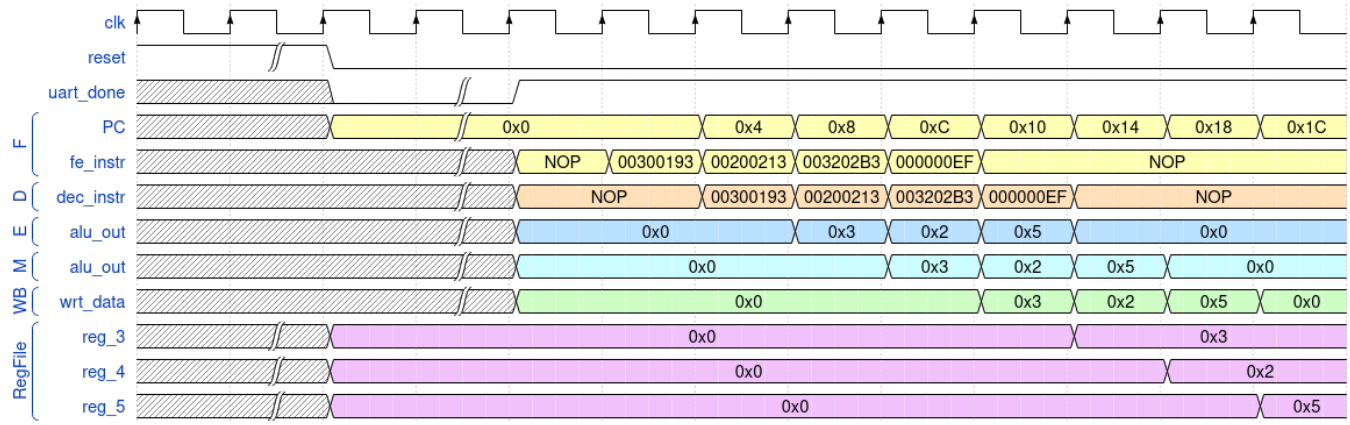

Fig. 6. Waveform showing the program flow in pipeline

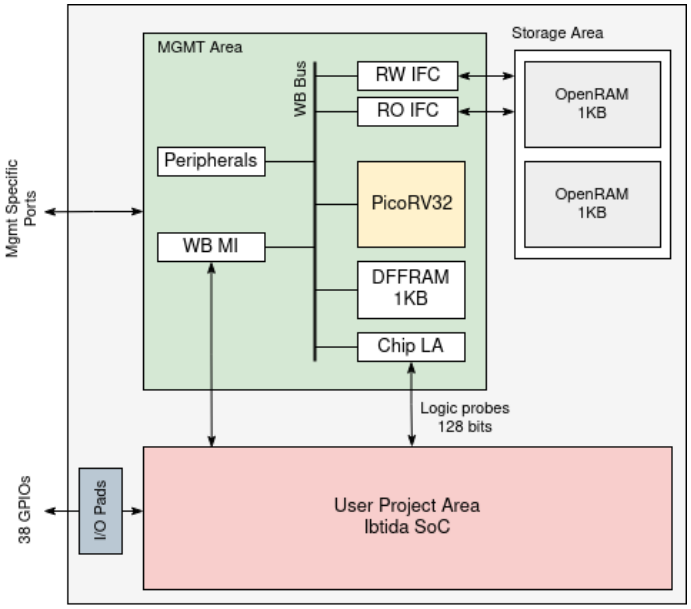

Fig. 7. Caravel architecture overview consisting of management area , user area and storage area

TABLE V
NETLIST REPORT OF IBTIDA SOC AFTER SYNTHESIS

Ibtida SoC	Units
Number of wires	81138
Number of wire bits	81630
Number of public wires	27542
Number of public wire bits	28034
Number of cells	97718

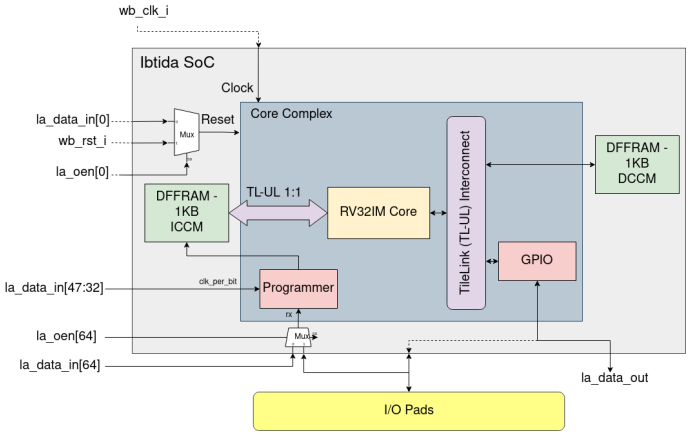

Fig. 8. Caravel architecture overview consisting of management area , user area and storage area

Fig. 9. OpenLANE Flow used for Ibtida SoC

b) Floor-planning: Floor-planning in the OpenLANE flow deals with assigning the die area and core area read from the *config.tcl* and generates the number of rows accordingly, and also involves placement of hard macros if any, in the design space. For Ibtida Soc floor-planning required the gate-level netlist generated through Yosys along with a *pin_order.cfg file*; which includes the name of the pins to be placed in an orderly manner around the SoC with the North Reverse (NR), South (S), East (E) and South Reverse (SR) format.

Floor-planning for Ibtida SoC has been achieved using the listing 7, where *init_floorplan* command floor-plans the netlist

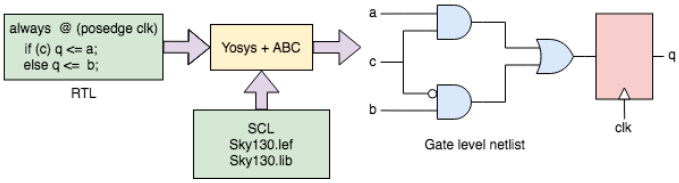

Fig. 10. Logic Synthesis

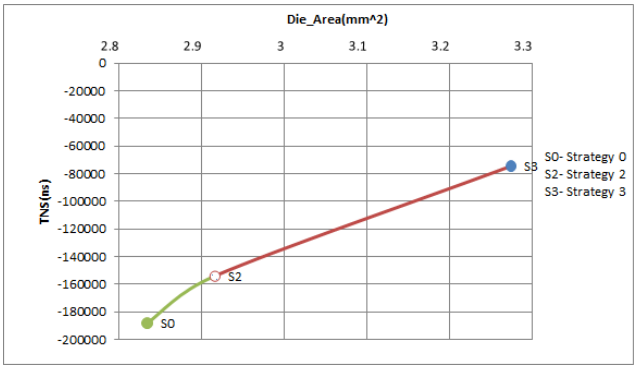

Fig. 11. Design exploration strategies

on a core area of 1620μm x 1590μm with a core utilization of 50%, i.e. half of the core area has been utilized by the standard cells. Table VI shows the coordinates of the core and the die for Ibtida SoC.

TABLE VI
FLOOR-PLANNING REPORT OF IBTIDA SOC

Ibtida SoC	lower x	lower y	upper x	upper y
Die	0.0	0.0	1591.5	1602.2
Core	5.5	10.9	1585.6	1591.2

This is followed by *place_io*, which places the I/Os around the die as shown in figure 12.

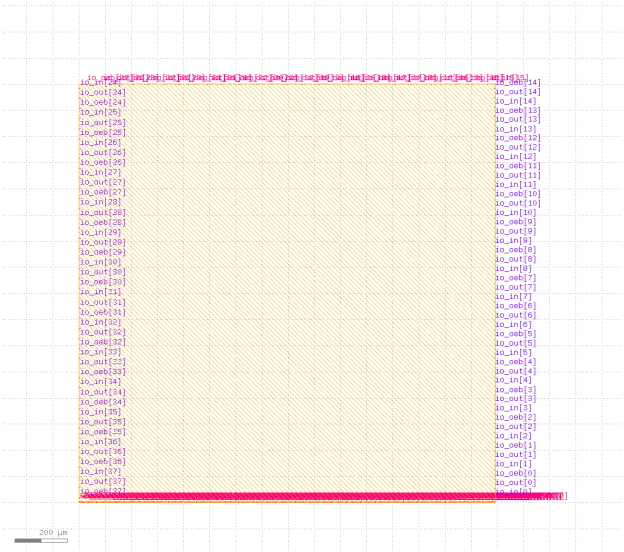

Fig. 12. Layout of Ibtida SoC after I/O placement

```
init_floorplan
place_io
gen_pdn
tap_decap_or
```

Listing 7: Floor-planning Script

Power distribution network (PDN) is then generated using the *gen_pdn* command which creates metal1 and metal4 horizontal rails and vertical straps respectively as shown in figure 13.

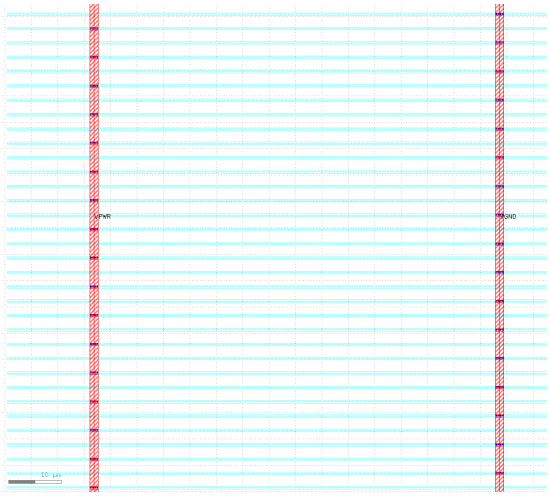

Fig. 13. Layout of Ibtida SoC after PDN generation

Lastly, well tap and decap cells are inserted to prevent latch-up issues between power and ground rails and to maintain a constant voltage supply respectively using the *tap_decap_or* command. The final layout of Ibtida SoC after floor-planning is shown in figure 14.

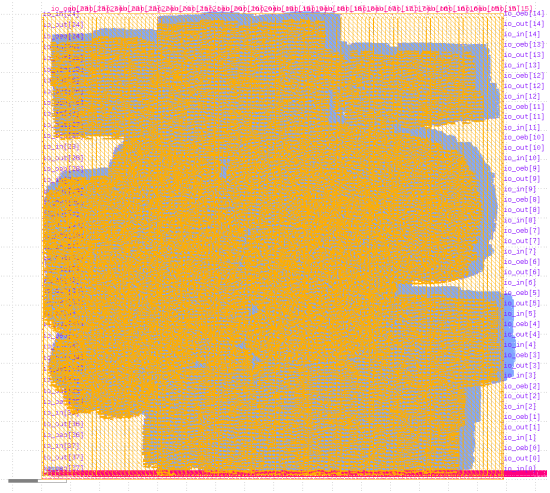

Fig. 14. Layout of Ibtida SoC after floor-planning

c) Placement: OpenLANE provides a set of tools for automatic place and route under the support of OpenROAD. The placement stage involves placing the standard cells onto the rows in the core area, laid out during the floor-planning

stage. The placement of cells is accomplished using two commands; *global_placement* followed by *detailed_placement* as shown in listing 8.

The *global_placement* command, inserts all the standard cells into the core area haphazardly. There is no sequence or order; some standard cells might even overlap each other. Ibtida SoC after global placement is shown in figure 15.

```
global_placement
detailed_placement
```

Listing 8: Placement Script

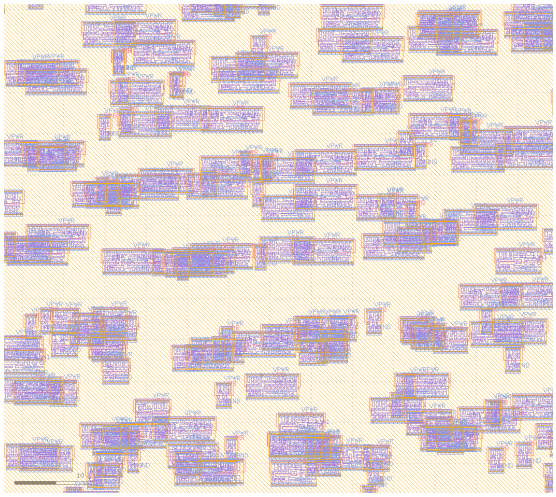

Fig. 15. Layout of Ibtida SoC after Global Placement

The *detailed_placement* command ensures that every cell is placed properly inside the rows. The legalization issues w.r.t. the overlapping cells are catered in this step, enabling the cells to align. Ibtida SoC after global placement is shown in figure 16.

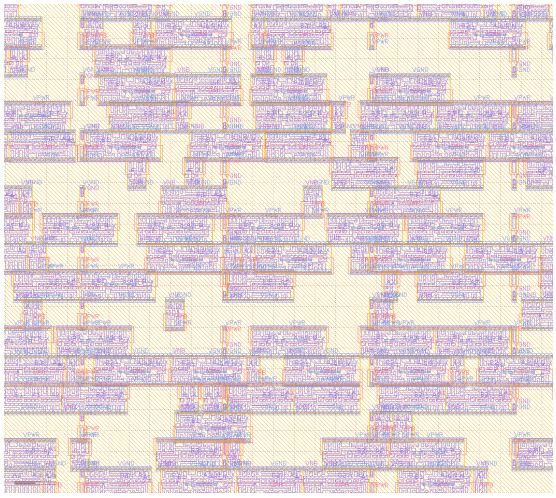

Fig. 16. Layout of Ibtida SoC after Detailed Placement

d) Clock Tree Synthesis (CTS): CTS is the step where the clock tree is established, thereby providing a clock to every sequential element (flip-flop) in the design. The clock tree for Ibtida SoC was generated using the command as shown in listing 9.

```
run_cts
```

Listing 9: Clock Tree Synthesis Script

e) Routing: Routing is a step followed by CTS. In the OpenLANE flow, routing is executed automatically through scripts. The task of the router is to precisely define the paths on the layout surface enabling conductors to carry electrical signals. The conductors are responsible for interconnecting the pins and the standard cells on the layout and thus forming a routing grid. Since the routing grid is quite large, routing is performed using a divide and conquer approach; Global Routing followed by Detailed Routing as shown in listing 10.

```
global_routing
detailed_routing
```

Listing 10: Routing Script

The *global_routing* command abstractly plans the routing guides to outline the implementation of actual routes whereas the *detailed_routing* command enables the wires to follow those routing guides and establish interconnects as shown in figure 17.

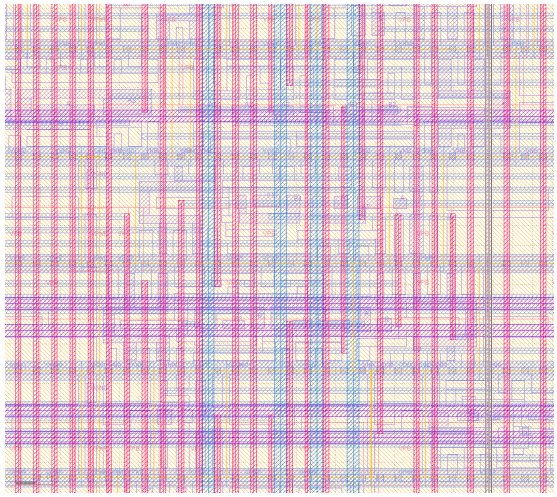

Fig. 17. Layout of Ibtida SoC after Detailed Routing

At this point powered Verilog netlist is generated using the *write_powered_verilog* command as shown in the listing 11.

```
write_powered_verilog
set_netlist $::env(lvs_result_file_tag).powered.v
```

Listing 11: Powered netlist Script

The SPEF extraction step is performed post routing, where resistance and capacitance of networks/devices are extracted from the layout to be utilized for Static Timing Analysis (STA).

f) GDS Generation: The Final steps towards Graphical Data Stream Information Interchange (GDSII) layout generation are ensured through the command as shown in listing 12.

```
run_magic
```

Listing 12: GDS Generation Script

The final Ibtida SoC GDSII layout hardened through OpenLANE flow, can be seen in figure 18

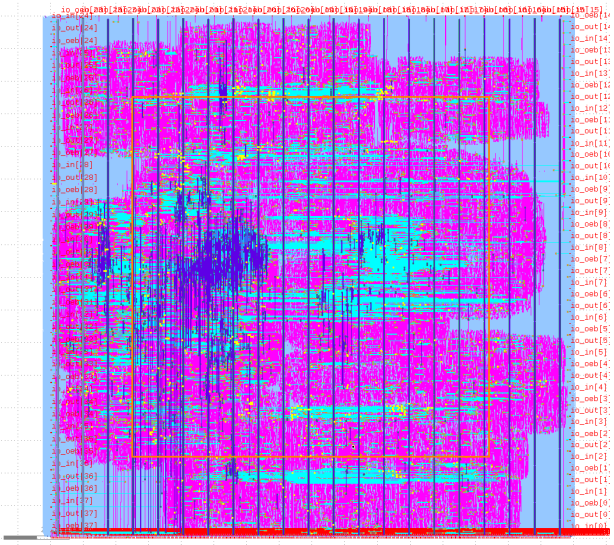

Fig. 18. Final Ibtida SoC GDSII layout

g) Physical Verification: The physical verification step also termed as the sign-off step in the OpenLANE flow is to validate the final layout. Throughout the flow, a series of reports and logs are generated, which usually involves checking the generated def file at each stage of physical design for any design rule violations. This is ensured by the EDA tools; fastroute, for identifying antenna violations, and tritonroute, which checks for any routing violations. The verification step ascertains that the placer and router have correctly placed the cells and routed the grid. The design is checked for any overlapping cells or short circuits and inspects any Layout vs. Schematic (LVS) error that includes any unmatched pins or short/open circuits between nets that should have been connected. Some common Design Rule Check (DRC) errors corresponding to the wire spacing, width and pitch need to be catered as defined in the PDK technology lef (.tlef) file. Some basic errors are shown in figure 19 and 20.

The final DRC and LVS check on the generated Ibtida SoC layout is ensured using the listing 13. For Ibtida SoC design to be considered DRC and LVS clean, it needs to be validated through Magic where *run_magic_drc* command checks the layout for any design rule check errors and reports them if any. Furthermore, a hierarchical SPICE netlist is extracted using the *run_magic_spice_export*. The extracted netlist is then validated through the open-source tool Netgen

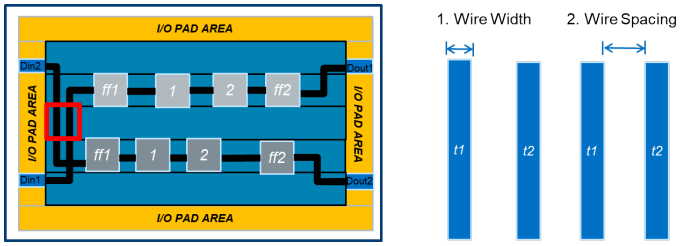

Fig. 19. Some Common DRC's I

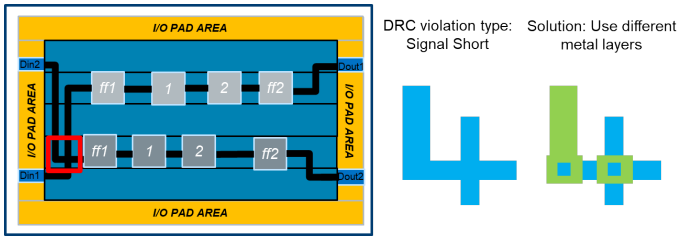

Fig. 20. Some Common DRC's II

by comparing it with the gate-level netlist obtained during the pre-route stage of the flow, and all discrepancies i.e. any mismatched nets and or pins are reported. This is achieved by the *run_lvs* command. Lastly, antenna checks are performed on the generated layout to identify metal tracks where charges are more susceptible to be accumulated during the fabrication process.

```
run_magic_drc
run_magic_spice_export
run_lvs
run_antenna_check
```

Listing 13: Checkers Script

h) Getting aboard on Caravel: The generated GDS of the Ibtida SoC is then placed on the Caravel harness inside the allocated user space area and hardened using the command as shown in the listing 14, and the final GDSII layout after Ibtida SoC caravel integration is shown in figure 21.

```
make ship
```

Listing 14: On-boarding Ibtida SoC on Caravel

IV. Conclusion

In this paper we overviewed a Chisel generated SoC taped-out using the completely open-source toolchain and discussed the different chip designing flows involving RTL simulation, FPGA emulation, and ASIC implementation. Furthermore, we also discussed how the OpenLANE suite allows the automatic place and route of a chip without needing a physical design expert.

Chisel HDL allows software programmers and novice hardware engineers to describe circuits in a higher programming language feel as compared to traditional hardware description languages. The user can abstractly design RTL logic and write

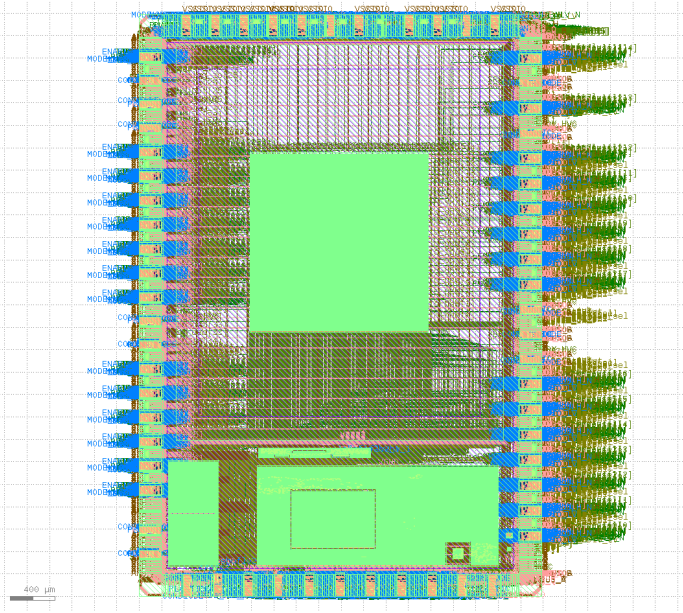

Fig. 21. Ibtida SoC after Caravel Integration

specific FIRRTL transforms for the flow they are targeting i.e Simulation, FPGA emulation, or ASIC implementation. Whereas, OpenLANE allows the design to be taken through the ASIC process to generating a GDSII layout for fabrication.

We believe with the introduction of a completely open-source PDK; SkyWater 130nm, and a completely open-source ISA; RISC-V, combined with HDLs hosted in higher programming languages; Chisel, there exists a great opportunity for undergraduate students, academia and researchers to quickly design, implement and fabricate chips using the agile methodology analogous to the software domain which has been a huge bottleneck in innovation in the hardware design industry.

REFERENCES

[1] Hadi Esmaeilzadeh et al. "Dark silicon and the end of multicore scaling". In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 365–376.

[2] John L. Hennessy and David A. Patterson. "A New Golden Age for Computer Architecture". In: *Commun. ACM* 62.2 (Jan. 2019), pp. 48–60. ISSN: 0001-0782. DOI: 10.1145/3282307. URL: https://doi.org/10.1145/3282307.

[3] Andrew Waterman et al. "The RISC-V instruction set manual, volume I: User-level ISA". In: *CS Division, EECE Department, University of California, Berkeley* (2014).

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

[5] Antmicro. *Open Source Process Design Kit from Google, SkyWater technologies and partners released*. https://antmicro.com/blog/2020/06/skywater-open-source-pdk/. 2020.

[6] *Chips Alliance*. https://chipsalliance.org/.

[7] *OpenROAD*. https://theopenroadproject.org/.

[8] Tim Edwards. "Google/SkyWater and the Promise of the Open PDK". In: (2020).

[9] Kent Beck et al. *Manifesto for agile software development*. https://agilemanifesto.org/. 2001.

[10] Yunsup Lee et al. "An Agile Approach to Building RISC-V Microprocessors". In: *IEEE Micro* 36.2 (2016), pp. 8–20. DOI: 10.1109/MM.2016.11.

[11] Jonathan Bachrach et al. "Chisel: Constructing hardware in a Scala embedded language". In: *DAC Design Automation Conference 2012*. 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[12] Krste Asanović et al. "The Rocket Chip Generator". In: (2016).

[13] 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*. ICCAD '20. Virtual Event, USA: Association for Computing Machinery, 2020. ISBN: 9781450380263. DOI: 10.1145/3400302.3415735. URL: https://doi.org/10.1145/3400302.3415735.

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

[15] *Efabless Open MPW Shuttle Program*. https://www.efabless.com/open_shuttle_program.

[16] Florent Kermarrec et al. *LiteX: an open-source SoC builder and library based on Migen Python DSL*. 2020. arXiv: 2005.02506 [cs.AR].

[17] Martin Odersky et al. "An Overview of the Scala Programming Language". In: 2004.

[18] *Verilator Open Source Simulator*. https://www.veripool.org/verilator/.

[19] *GTKWave Waveform Viewer*. http://gtkwave.sourceforge.net/.

[20] Kevin E. Murray et al. "SymbiFlow and VPR: An Open-Source Design Flow for Commercial and Novel FPGAs". In: *IEEE Micro* 40.4 (2020), pp. 49–57. DOI: 10.1109/MM.2020.2998435.

[21] Xifan Tang et al. "OpenFPGA: An Opensource Framework Enabling Rapid Prototyping of Customizable FPGAs". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 367–374. DOI: 10.1109/FPL.2019.00065.

[22] Henry Cook, Wesley Terpstra, and Yunsup Lee. "Diplomatic design patterns: A TileLink case study". In:

[23] *Verification Tests for Ibtida*. https://github.com/merledu/Caravel-Tests.

[24] *Efabless Caravel "harness" SoC*. https://github.com/efabless/caravel.

[25] *PicoRV32 - A Size-Optimized RISC-V CPU*. https://github.com/cliffordwolf/picorv32.

[26] Matthew R. Guthaus et al. "OpenRAM: An open-source memory compiler". In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2016, pp. 1–6. DOI: 10.1145/2966986.2980098.

[27] *Wishbone Bus*. https://en.wikipedia.org/wiki/Wishbone_(computer_bus).

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

[29] J.K. Ousterhout et al. “Magic: A VLSI Layout System”. In: *21st Design Automation Conference Proceedings*. 1984, pp. 152–159. DOI: 10.1109/DAC.1984.1585789.

[30] Tim Edwards. *Netgen netlist comparison (LVS) and format manipulation*. http://opencircuitdesign.com/netgen/.

[31] A. Agiza and S. Reda. *OpenPhySyn: An Open-Source Physical SynthesisOptimization Toolkit*. https://github.com/scale-lab/OpenPhySyn. 2020.

[32] *SPEF Extractor*. https://github.com/Cloud-V/SPEF_EXTRACTOR. 2020.