DEVELOPMENT OF CUSTOM RISC-V CENTRAL PROCESSING UNIT (CPU) WITH OPEN-SOURCE GOOGLE + SKYWATER 130NM

By

Md. Shyeem Rahman
21161006

Master of Science in Electrical & Electronic Engineering

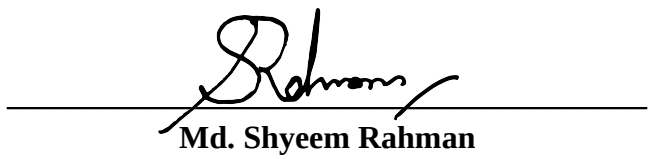
Department of Electrical and Electronic Engineering
BRAC University
September 2025

Declaration

It is hereby declared that

1. The thesis submitted is my own original work while completing degree at BRAC University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. I have acknowledged all the main sources of help.

Student's Full Name & Signature:



Md. Shyeem Rahman
21161006

Approval

The thesis titled “Development of custom RISC-V Central Processing Unit (CPU) with open-source Google + SkyWater 130nm Process Design Kit (PDK)” submitted by

Md. Shyeem Rahman (21161006)

of Spring, 2021 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of Master of Science in Electrical & Electronic Engineering on April 24, 2025.

Examining Committee:

Supervisor:
(Member)

_____

Touhidur Rahman, PhD
Professor, Department of Electrical and Electronic Engineering
BRAC University

Internal Examiner:
(Member)

_____

MG Sorwar Hossain, PhD
Professor, Department of Electrical and Electronic Engineering
BRAC University

External Expert
Examiner:
(Member)

_____

A. B. M. Harun-Ur-Rashid, PhD
Professor and Head of Department,
Department of Electrical and Electronic Engineering
Bangladesh University of Engineering & Technology

Departmental Head:
(Chair)

_____

Md. Mosaddequr Rahman, PhD
Professor and Chairperson,
Department of Electrical and Electronic Engineering
BRAC University

Abstract

The chip design industry relies heavily on secretive methodologies and designs. A significant gap exists between commercial design environment and open-source environment. Although a range of open-source CAD tools has been developed to address aspects such as system design, logic synthesis, place-and-route, test insertion, and verification, there is no established toolchain for open-source library characterization. This thesis aims to develop that toolchain for open-source library characterization and develop a custom standard cell library based on Google + SkyWater 130nm technology. To further verify the toolchain and libraries, a RISC-V CPU is designed with the same library using established open-source Place and Route tools and simulated for result verification.

Therefore, this work aims to tackle issues related to open-source chip design while diving deep into open-source tools to design a RISC V CPU. To achieve this, the gaps which remain between the ASIC design done with commercial design tools and open-source tools are addressed and bridged.

Keywords: open source vlsi tool; library characterization; library characterization; PVT variation; google + skywater 130nm pdk; automatic placement and route; RISC-V; CPU design

Table of Contents

List of Figures

List of Tables

Chapter 1 : Introduction

1.1 Evolution of Open-Source EDA Tools in VLSI Design

In the past, the landscape of electronic design automation (EDA) was dominated by proprietary tools provided by a few major vendors. These commercial EDA tools are powerful but come with high licensing costs and access restrictions, making them less accessible to students, researchers, and small startups. In recent years, a robust open-source EDA ecosystem has begun to emerge to address these issues. A key driver behind this movement is the need for broader accessibility – open-source tools can be freely downloaded and used by anyone, removing the barrier of expensive licenses and enabling wider participation in chip design education [1]. This improved access is a “game changer” for academia, allowing instructors to teach VLSI design using tools that students can continue to use outside the classroom without cost [1].

Another motivation for open-source EDA is to counteract the rising cost and complexity of semiconductor design. Modern system-on-chip (SoC) projects often involve prohibitive upfront costs, partly due to licensing fees for intellectual property (IP) cores and EDA software [1]. For example, traditional ISAs like x86 or ARM are proprietary, which led to the rise of open ISAs such as RISC-V to avoid licensing costs [1]. In a parallel manner, open-source EDA tools are seen to democratize chip design. Indeed, open hardware initiatives like RISC-V have recently cast “a lot of light onto the open-source EDA space” [1], spurring renewed interest in developing free EDA tool flows that anyone can use.

Open-source EDA software is not a completely new idea – some tools have existed for decades. In fact, the open-source EDA community can trace its roots back to the 1970s and 1980s [2]. UC Berkeley’s SPICE, developed in the 1970s, is a prime example: it is a circuit simulation

program that was so successful and easy to use that its name became a verb in chip design ("let's SPICE that circuit") [2]. Likewise, Magic, first released in the 1980s, is an open-source interactive VLSI layout tool that remains in use today [3]. Magic is based on the Mead-Conway design methodology and allows designers to create and modify chip layouts in a hierarchical fashion [3]. These early tools, along with others (such as VIS for verification and Qflow for digital design flow integration), laid the groundwork for an open EDA ecosystem [4] [5]. However, as semiconductor technology advanced rapidly, the open-source EDA community fell behind the state-of-the-art – especially at smaller process nodes – due to limited resources and the immense complexity of modern design requirements.

Recognizing the strategic importance of open hardware and EDA, institutions have started investing in this area. In 2018, the U.S. DARPA launched initiatives (e.g. the OpenROAD project) with significant funding to develop open-source EDA tools, aiming to reduce the cost and time of designing custom integrated circuits [6]. OpenROAD, for instance, seeks to achieve fully automated RTL-to-GDSII digital implementation within 24 hours without human intervention [6]. At the same time, community-driven efforts and university programs have bundled various open tools into usable flows. Projects like OpenLane (by Efabless) and DARPA-funded OpenROAD provide end-to-end digital design flows using open-source components (logic synthesis with Yosys, placement and routing with OpenROAD tools, layout with Magic, verification with OpenVAS, etc.) [6]. These efforts, combined with the momentum of open hardware projects, herald a new era where open-source software can realistically be used to design complex chips.

Perhaps the most groundbreaking development enabling open-source silicon design was the release of an open-process design kit (PDK) for fabrication. In June 2020, Google and SkyWater Technologies announced the availability of the SkyWater 130nm Open PDK, the first commercially manufacturable semiconductor process design kit provided under open-

source terms [7]. Unlike previous PDKs which required NDAs and were restricted, the SkyWater130 PDK is freely accessible, enabling designers to create real chips that can be fabricated in SkyWater's 130 nm CMOS process. This removed a major roadblock in the open silicon flow by providing the community with an NDA-free, industry-supported PDK [8]. Furthermore, Google and partners even sponsored shuttle programs (multi-project wafer runs) to fabricate open-source designs at no cost to the designers [8]. The combination of an open PDK and free fabrication opportunities galvanized the open hardware community, allowing academic and hobbyist projects to move from simulation to physical silicon. It also reinforced the synergy between open-source EDA tools and open hardware designs – for example, many RISC-V chips and other open IPs have since been fabricated on the Sky130 process using fully open-source flows.

In summary, the evolution of open-source EDA tools is tightly coupled with the broader open hardware movement. Open-source EDA addresses critical issues of access and education, allowing students and engineers to learn and innovate without the barriers imposed by proprietary software [1]. With initiatives like the SkyWater 130 nm Open PDK, a complete open-source ASIC design pipeline (from HDL design with open ISAs like RISC-V, through synthesis, verification, place-and-route, and all the way to GDSII) is rapidly becoming a reality. This thesis builds upon this context, specifically tackling one of the remaining missing pieces in the open-source VLSI design puzzle, which is standard cell library characterization.

1.2 The Need for Open-Source Standard Cell Characterization

While open-source tools now exist for many steps of the digital design flow (logic synthesis, simulation, floorplanning, routing, etc.), one critical gap has been the lack of an open-source solution for standard cell library characterization [9]. Standard cell libraries are the bedrock of digital IC design – they provide a catalog of logic gates (AND, OR, flip-flops, etc.) with predefined layouts and known electrical characteristics. These libraries come with data files (typically in Synopsys Liberty .lib format) that specify each cell's timing (delay, output slew) and power consumption under various conditions [10]. Place-and-route tools and static timing analysis engines use this characterization data to ensure that the chip meets timing and power requirements. In a typical industrial setting, foundries or IP vendors provide thoroughly characterized libraries for their processes. However, in the open-source domain, designers often wish to create custom standard cells (for specialized functions or improved performance) or re-characterize existing cells under different conditions – and no fully open-source, mature tool-chain has been available to perform this full library characterization [11].

Characterizing a standard cell means determining how it behaves electrically: how long it takes for outputs to switch after inputs change (propagation delay), how fast the output transition is (slew), how much dynamic and leakage power it consumes, and how its input capacitance loads preceding stages [12]. Importantly, these metrics are not fixed values – they depend on operating conditions and on the context in which the cell is used (e.g., input slope and output load). Therefore, industry characterization tools perform numerous SPICE simulations across various process-voltage-temperature (PVT) corners, input transition times, and output loading conditions to populate multi-dimensional tables (Non-Linear Delay Models, etc.) for each cell [12]. This process must be highly automated and accurate, since a typical library might contain dozens of cells and each cell must be characterized over many conditions.

Historically, library characterization has been the realm of proprietary EDA tools such as Cadence Liberate and Synopsys SiliconSmart. These commercial tools are considered the "gold standard" – for example, Synopsys SiliconSmart generates Liberty timing/power models with *PrimeTime*-level accuracy by using embedded SPICE simulation engines [11]. Cadence's Liberate characterization suite similarly automates the process of cell characterization and validation, which has been used reliably in industry for decades [13]. However, these tools are closed-source and require expensive licenses, putting them out of reach for many academic teams or open-source projects. In fact, even universities that develop their own open-source standard cell libraries have often had to rely on Cadence Liberate for characterization due to the absence of a viable open alternative [14]. This underscores the pressing need for an open-source characterization flow that the academic and open silicon community can use independently.

The difficulty in creating an open characterization toolchain lies in the complexity of the task. The tool must interface with detailed device models (from the PDK), generate input stimuli for all relevant logic transitions of the cell, run a circuit simulation for each scenario, measure the results, and finally format the data into standard library formats (Liberty .lib, and possibly .SPA or ECSM tables for noise, etc.). Additionally, if sequential cells (flip-flops, latches) are involved, the tool must handle timing constraints like setup/hold, clock-to-Q delays, as well as special conditions like asynchronous set/reset (including recovery/removal times). A few research efforts have begun to address this gap. For example, **Libretto**, introduced by Nishizawa and Nakura in 2022, is an open-source cell library characterizer designed to generate timing and power models for standard cells [9]. Libretto uses the open-source NGSPICE simulator as its engine, automatically creating SPICE input decks, running simulations, and collating results into a Liberty format output [9]. This demonstrated that an open tool can indeed perform characterization; however, Libretto in its initial form still had practical

limitations (such as requiring users to manually specify each cell's logic function and pins in configuration files) and did not yet see widespread adoption. Another example is CharLib (2023), which builds on Libretto to provide a Python-based flow for characterization, and research prototypes like LICHEN and SpiceGen have been explored in academia [15]. Despite these efforts, at the time of this writing there remains no fully featured, open-source characterization solution that is seamlessly integrated into a complete digital design flow and proven to be on par with commercial tools. This thesis addresses that gap by developing an open-source characterization methodology and demonstrating its efficacy on real hardware designs.

In conclusion, the introduction of an open-source standard cell characterization flow – validated against industry tools and demonstrated in a real CPU design – is a novel contribution that empowers the academic and hobbyist semiconductor community. It addresses a longstanding gap in the open-source ASIC flow, and its success suggests that we can begin to educate and innovate without limits imposed by proprietary EDA silos. This chapter has outlined the motivation, approach, and outcomes of our work. The subsequent chapters will delve into each aspect in detail: Chapter 2 will describe the design of the characterization flow and custom cells, Chapter 3 will present the RISC-V CPU implementation and integration, Chapter 4 will discuss the schematic, layout and the PnR of the CPU. Chapter 5 will dive into the validation results and comparisons, and Chapter 6 will conclude with insights and future work.

Chapter 2 : Standard Cell Characterization

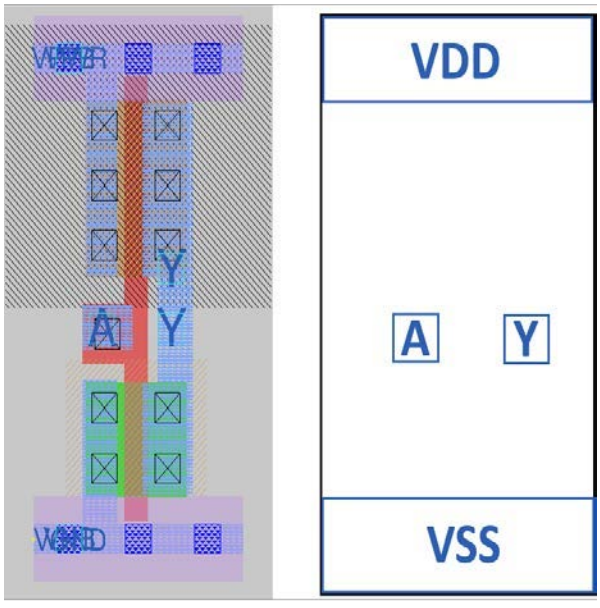
Standard cell characterization is a process of analyzing a circuit using static and dynamic methods to generate models suitable for chip implementation flows [11].

The Automatic Place and Route flow requires the following files as inputs to design the chip.

1. Verilog of the design
2. Standard cell LEF (Library Exchange Format) file
3. Standard cell LIB (Liberty timing) file

2.1 LEF file

Cell LEF file (.lef) contains the abstract information related to each cell present in the standard cell library in separate sections [10]. It contains only the pin and dimension information (Fig. 15. (ii), which is further used in Automatic Placement and Route tools. These tools will use the dimension and the pin information to place the standard cell in chip floorplan.



(i) *(ii)*

Figure 2.1. Inverter (i) Layout and (ii) LEF views

2.2 Liberty file

The .lib file or LIB file is Liberty Timing File. It is an ASCII representation of the timing and power parameters associated with any cell in a particular semiconductor technology [11]. The LIB file varies with each process node and VT condition. A .lib file for a particular cell consists of two sections. The first section contains the common library information and parameters (technology name, unit of calculation and operating condition). The second section contains the timing and power outputs depending on the variation of inputs across different pins of the cell [16].

Depending on the process of characterization, a library can be classified into two models. The first one is Non-Linear Delay Model (NLDM), and the second one is Current Composite (CCS) model. NLDM modeling involves the use of input net transition and output net capacitances as input parameters to generate look-up tables for cell fall, fall transition, cell rise and rise transition. CCS model characterization is done by calculating the current flow through the output capacitor with respect to the input transition [17].

2.3 Library Characterization

Characterization is a process of analyzing a circuit using static and dynamic methods to generate models suitable for chip implementation flows Process of compiling data about the behavior of standard-cells or gates. A regular standard cell liberty file contains the following parameters [17]

1. Timing Parameters
    a. Rise Transition Time
    b. Fall Transition Time
    c. Rise Propagation Delay
    d. Fall Propagation Delay

2. Power Parameters

    a. Rise Power
    b. Fall Power

2.3.1 Timing Parameters

Timing Characterization determines the timing parameters of the gate. Rise transition or rise time is the time taken for the signal to rise from 20% to 80% of the maximum voltage. And the fall transition or fall time is the time taken to fall from 80% to 20% [18]. Equation (1) and (2) represent the calculations and conditions for rise and fall time calculation.

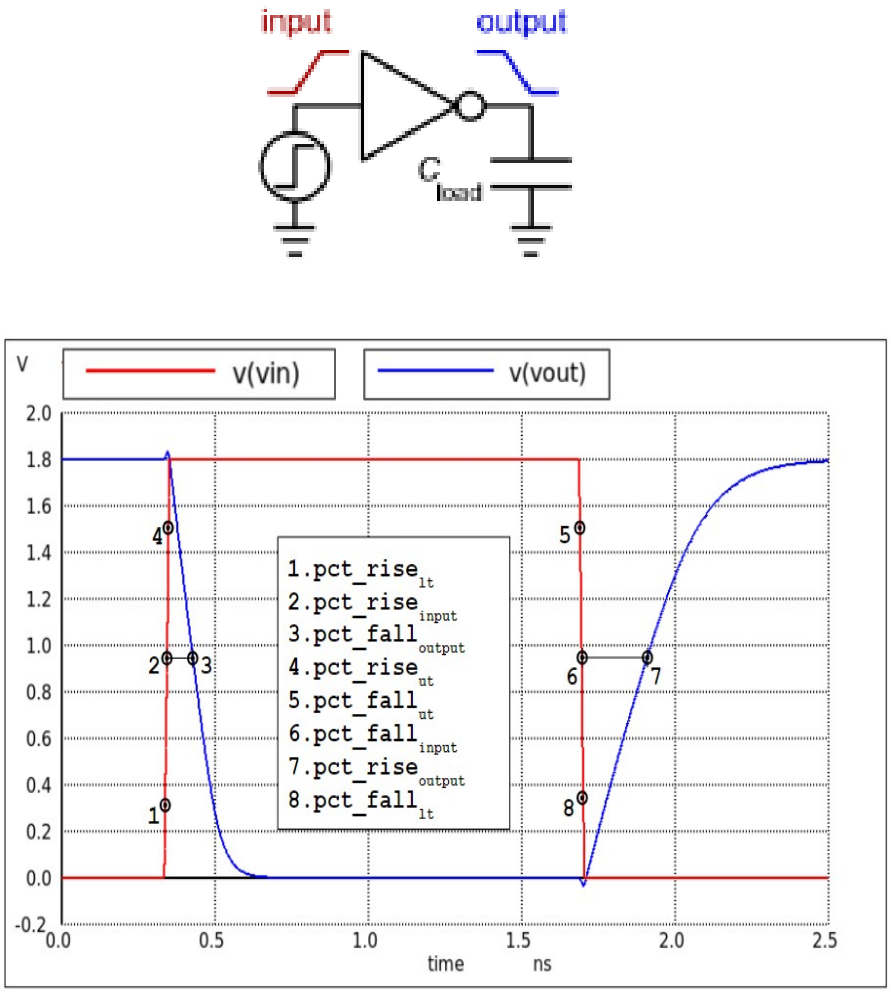


Figure 2.2. Delay Calculation

Equation (1) and (2) represent the calculations and conditions for rise and fall time calculation.

$$cell\_rise = pct\_rise_{ut} - pct\_rise_{lt} \qquad \textit{Equation 2.1}$$

$$cell\_fall = pct\_fall_{ut} - pct\_fall_{lt} \qquad \textit{Equation 2.2}$$

Here, ut = upper threshold and lt = lower threshold

Propagation time for any logic gate is the time taken between 50% of input transition to 50% of fall transition time [18]. Cell fall is the fall propagation time and Cell rise is the rise propagation time respectively. The Equations (3) and (4) show formula for the rising and the falling propagation delays are calculated. Fig. 2.2 shows the different points in the curve from where the values are extracted in the simulation tool.

$$T_{pHL} = pct\_rise_{input} - pct\_fall_{output} \qquad \textit{Equation 2.3}$$

$$T_{pLH} = pct\_fall_{input} - pct\_rise_{output} \qquad \textit{Equation 2.4}$$

Fig 2.3 shows the format of a regular timing library file. The values of transition and propagation times are arranged in look-up tables so that the tool can extrapolate among the values based on the input net transition and output capacitance.

```
lu_table_template ("del_1_7_7") {
    variable_1 : "input_net_transition";
    variable_2 : "total_output_net_capacitance";
    index_1("1, 2, 3, 4, 5, 6, 7");
    index_2("1, 2, 3, 4, 5, 6, 7");
}

timing () {
    cell_fall ("del_1_7_7") {
        index_1("0.0100000000, 0.0230506000, 0.0531329000, 0.1224740000, 0.2823110000, 0.6507430000, 1.5000000000");
        index_2("0.0005000000, 0.0014764100, 0.0043595800, 0.0128730000, 0.0380118000, 0.1122420000, 0.3314310000");
        values("0.0119446000, 0.0137840000, 0.0188149000, 0.0327326000, 0.0729366000, 0.1922578000, 0.5454940000", \
            "0.0157429000, 0.0180991000, 0.0233701000, 0.0374622000, 0.0781416000, 0.1961942000, 0.5456924000", \
            "0.0203785000, 0.0240707000, 0.0324561000, 0.0487044000, 0.0892794000, 0.2076327000, 0.5571236000", \
            "0.0248258000, 0.0307580000, 0.0439276000, 0.0689207000, 0.1156458000, 0.2359077000, 0.5860208000", \
            "0.0262075000, 0.0354142000, 0.0559354000, 0.0961932000, 0.1673476000, 0.2955090000, 0.6455342000", \
            "0.0157468000, 0.0301166000, 0.0619237000, 0.1246900000, 0.2371069000, 0.4263973000, 0.7871234000", \
            "-0.0275597000, -0.0053926000, 0.0434940000, 0.1403033000, 0.3151930000, 0.6122458000, 1.1101468000");
    }
    cell_rise ("del_1_7_7") {
        index_1("0.0100000000, 0.0230506000, 0.0531329000, 0.1224740000, 0.2823110000, 0.6507430000, 1.5000000000");
        index_2("0.0005000000, 0.0014764100, 0.0043595800, 0.0128730000, 0.0380118000, 0.1122420000, 0.3314310000");
        values("0.0175587000, 0.0211484000, 0.0310262000, 0.0584472000, 0.1371815000, 0.3662591000, 1.0435811000", \
            "0.0230691000, 0.0264576000, 0.0360719000, 0.0633862000, 0.1425054000, 0.3734522000, 1.0627602000", \
            "0.0339028000, 0.0388338000, 0.0493056000, 0.0760130000, 0.1545360000, 0.3864530000, 1.0598706000", \
            "0.0498930000, 0.0577606000, 0.0750599000, 0.1075488000, 0.1862726000, 0.4152755000, 1.0963705000", \
            "0.0744968000, 0.0868541000, 0.1145327000, 0.1673453000, 0.2583118000, 0.4867961000, 1.1619753000", \
            "0.1156072000, 0.1340389000, 0.1760385000, 0.2596266000, 0.4057179000, 0.6556388000, 1.3298844000", \
            "0.1928740000, 0.2183706000, 0.2783548000, 0.4048308000, 0.6381117000, 1.0233501000, 1.7170592000");
    }
    fall_transition ("del_1_7_7") {
        index_1("0.0100000000, 0.0230506000, 0.0531329000, 0.1224740000, 0.2823110000, 0.6507430000, 1.5000000000");
        index_2("0.0005000000, 0.0014764100, 0.0043595800, 0.0128730000, 0.0380118000, 0.1122420000, 0.3314310000");
        values("0.0048909000, 0.0069497000, 0.0130525000, 0.0311683000, 0.0847074000, 0.2419201000, 0.7066229000", \
            "0.0069554000, 0.0084546000, 0.0135709000, 0.0312350000, 0.0846748000, 0.2428864000, 0.7101457000", \
            "0.0116927000, 0.0141965000, 0.0197198000, 0.0335758000, 0.0847194000, 0.2435099000, 0.7091378000", \
            "0.0203289000, 0.0240677000, 0.0323348000, 0.0495335000, 0.0902360000, 0.2418217000, 0.7064893000", \
            "0.0354595000, 0.0413981000, 0.0546071000, 0.0793592000, 0.1253289000, 0.2517686000, 0.7084765000", \
            "0.0637251000, 0.0731900000, 0.0941919000, 0.1324136000, 0.2017282000, 0.3250714000, 0.7157009000", \
            "0.1183897000, 0.1329738000, 0.1635937000, 0.2237756000, 0.3292207000, 0.5112171000, 0.8526758000");
    }
    related_pin : "A";
    rise_transition ("del_1_7_7") {
        index_1("0.0100000000, 0.0230506000, 0.0531329000, 0.1224740000, 0.2823110000, 0.6507430000, 1.5000000000");
        index_2("0.0005000000, 0.0014764100, 0.0043595800, 0.0128730000, 0.0380118000, 0.1122420000, 0.3314310000");
        values("0.0102030000, 0.0145665000, 0.0274391000, 0.0655113000, 0.1778623000, 0.5106880000, 1.4869649000", \
            "0.0107758000, 0.0146772000, 0.0273936000, 0.0654823000, 0.1774442000, 0.5091666000, 1.4832895000", \
            "0.0176433000, 0.0206470000, 0.0298357000, 0.0653682000, 0.1774688000, 0.5082211000, 1.4799179000", \
            "0.0287666000, 0.0338285000, 0.0452178000, 0.0722948000, 0.1771081000, 0.5095251000, 1.4848178000", \
            "0.0472331000, 0.0557641000, 0.0742180000, 0.1079812000, 0.1903953000, 0.5072639000, 1.4842672000", \
            "0.0774171000, 0.0903774000, 0.1203311000, 0.1758978000, 0.2700296000, 0.5253227000, 1.4794522000", \
            "0.1325493000, 0.1519492000, 0.1976181000, 0.2847110000, 0.4388531000, 0.6966839000, 1.4978170000");
    }
```

Figure 2.3. Standard Cell Library Format (Timing)

2.3.2 Power Parameters

Fig 2.4 shows the format of the power parameters of a cell in a regular library file. Rise power is defined as the dynamic power drop in the circuit during the signal to rise from 0V of the maximum voltage. The fall power refers to the power drop during the fall from the maximum voltage to 0V. These values are dependent on the amount of current flow during those periods.

```
power_lut_template ("power_outputs_1") {
    variable_1 : "input_transition_time";
    variable_2 : "total_output_net_capacitance";
    index_1("1, 2, 3, 4, 5, 6, 7");
    index_2("1, 2, 3, 4, 5, 6, 7");
}

pin ("A") {
    capacitance : 0.0044590000;
    clock : "false";
    direction : "input";
    fall_capacitance : 0.0042760000;
    max_transition : 1.5000000000;
    related_ground_pin : "VGND";
    related_power_pin : "VPWR";
    rise_capacitance : 0.0046420000;
}
pin ("Y") {
    direction : "output";
    function : "(!A)";
    internal_power () {
        fall_power ("power_outputs_1") {
            index_1("0.0100000000, 0.0230505800, 0.0531329300, 0.1224745000, 0.2823108000, 0.6507428000, 1.5000000000");
            index_2("0.0005000000, 0.0014764110, 0.0043595770, 0.0128730500, 0.0380118100, 0.1122421000, 0.3314308000");
            values("-0.0048729000, -0.0061807000, -0.0104534000, -0.0239314000, -0.0645157000, -0.1847181000, -0.5397786000", \
                "-0.0053546000, -0.0066818000, -0.0108639000, -0.0241652000, -0.0646044000, -0.1847535000, -0.5397978000", \
                "-0.0056889000, -0.0070921000, -0.0113878000, -0.0245474000, -0.0647872000, -0.1848170000, -0.5398257000", \
                "-0.0054172000, -0.0070530000, -0.0115219000, -0.0249378000, -0.0650926000, -0.1849624000, -0.5398857000", \
                "-0.0049315000, -0.0065669000, -0.0113451000, -0.0249480000, -0.0653572000, -0.1851649000, -0.5399634000", \
                "-0.0027297000, -0.0045768000, -0.0098210000, -0.0237645000, -0.0648503000, -0.1852162000, -0.5400208000", \
                "0.0026942000, 0.0005644000, -0.0051128000, -0.0206753000, -0.0628575000, -0.1842632000, -0.5397747000");
        }
        related_pin : "A";
        rise_power ("power_outputs_1") {
            index_1("0.0100000000, 0.0230505800, 0.0531329300, 0.1224745000, 0.2823108000, 0.6507428000, 1.5000000000");
            index_2("0.0005000000, 0.0014764110, 0.0043595770, 0.0128730500, 0.0380118100, 0.1122421000, 0.3314308000");
            values("0.0129073000, 0.0147627000, 0.0198879000, 0.0340719000, 0.0747268000, 0.1938930000, 0.5478985000", \
                "0.0126104000, 0.0143517000, 0.0194124000, 0.0336506000, 0.0744926000, 0.1940106000, 0.5436985000", \
                "0.0125047000, 0.0143132000, 0.0190767000, 0.0331759000, 0.0742976000, 0.1924800000, 0.5446534000", \
                "0.0126884000, 0.0144153000, 0.0189569000, 0.0328509000, 0.0734457000, 0.1937375000, 0.5453426000", \
                "0.0135492000, 0.0149941000, 0.0194349000, 0.0329725000, 0.0731844000, 0.1931974000, 0.5464750000", \
                "0.0149532000, 0.0163193000, 0.0205688000, 0.0337550000, 0.0737759000, 0.1921072000, 0.5425809000", \
                "0.0206515000, 0.0217040000, 0.0253143000, 0.0377926000, 0.0767143000, 0.1949605000, 0.5439724000");
        }
    }
```

Figure 2.4. Standard Cell Library Format (Power)

2.4 VT Variation

Voltage and Temperature variations within a design result in significant changes in the timing analysis of the tools [21]. With the change in the supply voltage, there will be a change in the saturation current resulting in a change in the propagation delay. The speed of the cell is proportional to the supply voltage.

Alternatively, propagation delay increases with the increase in temperature because of the reduction of electron and hole mobility within the silicon [21].

Depending on this, each chip is taped out in multiple design corners to ensure equal reliability and efficiency across different operating conditions. Therefore, flow is verified in three corners

1. Fast (best corner)
2. Slow (worst corner)
3. Typical (nominal corner)

Fig. 2.5 shows the relationship between delay with temperature and voltage variation.

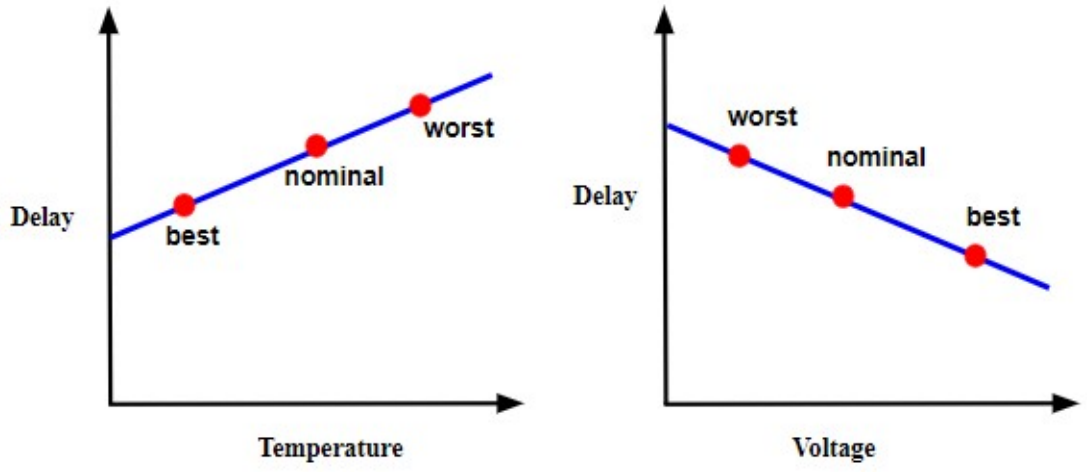


Figure 2.5. VT operating conditions

2.5 Developing a Custom Characterization Flow with Open-Source Tools

This work presents a complete standard cell characterization flow built entirely on open-source tools and the SkyWater 130 nm open PDK. The goal was to create a methodology that enables characterizing custom standard cells (generating their timing and power library data) without any proprietary software, while maintaining compatibility with commercial-grade design flows.

For Liberty or .lib file extraction, the input files are the circuit SPICE netlist generated from the layout and the Sky130 device model. The characterization flow consists of a few steps as described in Fig. 2.4. To simulate each input pin of the gates, a wrapper circuit is added with loads to the circuit under test. With a constant input net transition of 0.01ns to 1.5ns and load different capacitances, the simulation is run on two different flows, one for rise calculation and other for fall calculation. The main reason behind this split is due to the change in input waveform for NgSpice. For rise calculations, the input slope is rising and for the fall calculation the input is falling.

Since PnR tools require a specific format for the .lib file, a template file is added to the flow script. This ensures that the values are placed in a lookup table format. Therefore, the final file obtained from the end of the flow is ready to be used as an input to an Automatic Place and Route or Timing Analysis tools.

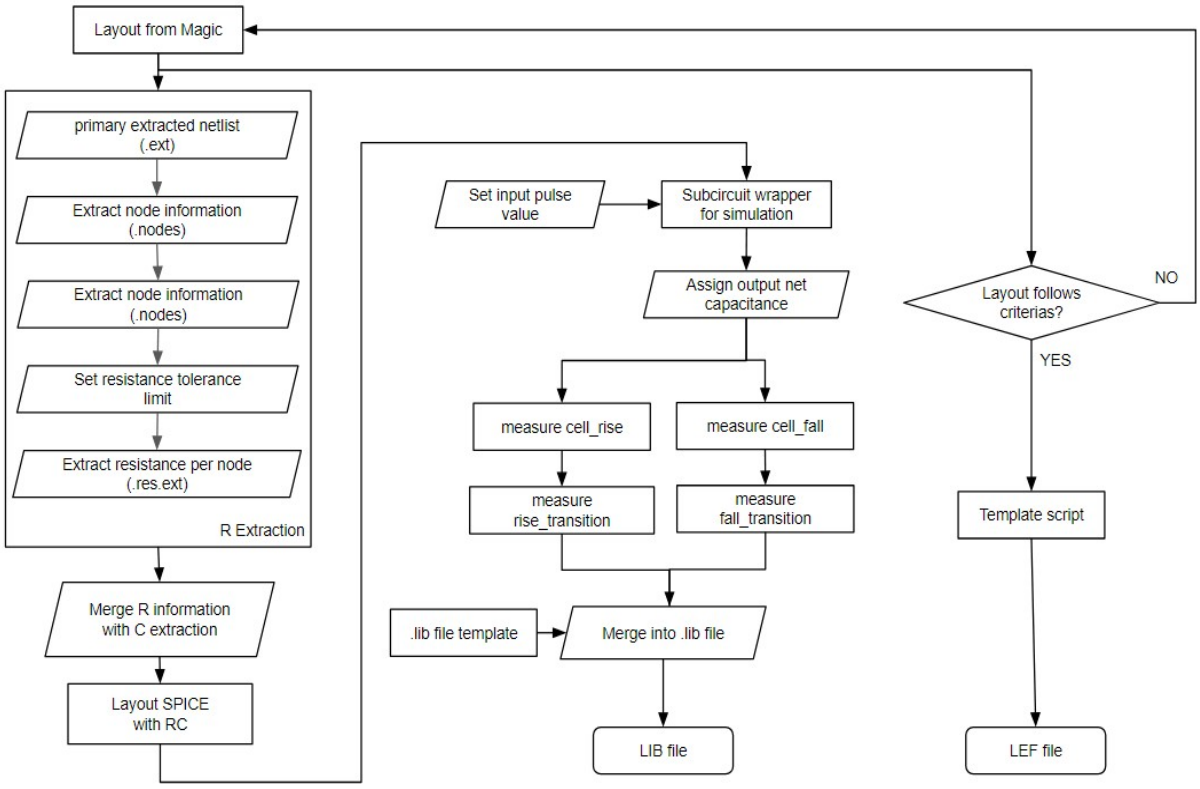


Figure 2.6. Timing Characterization Flow Algorithm

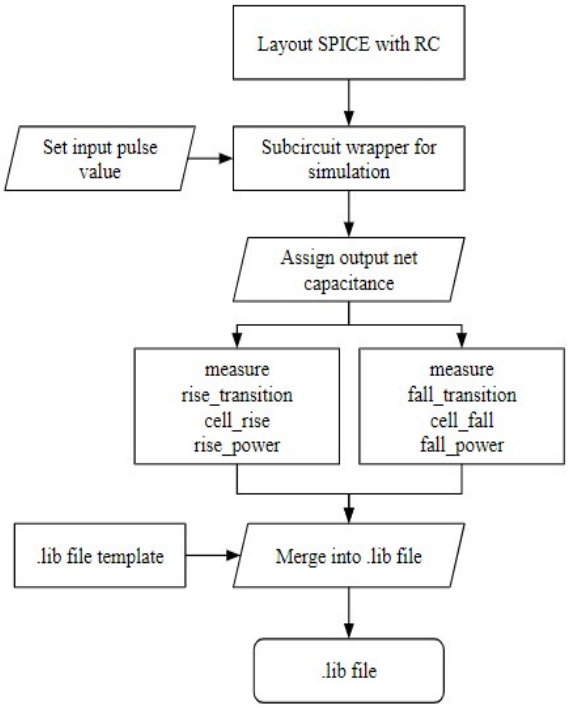


Figure 2.7. Power Characterization Flow Algorithm

2.5.1 Design and Layout of Custom Cells

Custom logic cells are needed for our RISC-V CPU design – including basic gates and certain special cells – using the open source PDK. Schematic capture was done in **Xschem**, an open-source schematic editor, to define the transistor-level circuits for each cell. The physical layouts were drawn in **Magic VLSI**, leveraging Magic's DRC/LVS rules for Sky130 [19]. Magic's friendly interface and real-time design rule checking made it well-suited for crafting cells that obey the foundry rules. All cells were laid out with a common height and pin alignment consistent with a standard-cell methodology (to allow automated place-and-route integration) [19]. After layout, each cell was extracted (to obtain a SPICE netlist with parasitic capacitances) using the Magic tool. The extraction provides the necessary device parameters and parasitics that influence timing.

2.5.2 Simulation of Cell Behavior with NGSPICE

The core of the characterization flow is built around NgSPICE, the open-source SPICE simulator. For each cell, our flow automatically generates a series of SPICE test benches that emulate the standard characterization scenarios (for example, for a 2-input NAND gate, the rise/fall delay for each input controlling the output, under various load capacitances and input edge rates). This automation was implemented with Python scripts that take the cell's netlist and symbol information and produce SPICE decks similar to what a tool like Cadence Liberate would create. NGSPICE then simulates each deck to measure delays, output transition times, and power consumption. The use of an open-source SPICE engine is a critical aspect, since it guarantees that even the simulation step remains transparent and accessible to all users of the flow (no reliance on closed simulators like HSPICE or Spectre).

2.5.3 Timing Characterization

For Liberty or .lib file extraction, the input files are the circuit SPICE netlist generated from the layout and the Sky130 device model. The characterization flow consists of a few steps as described in Fig. 18. To simulate each input pin of the gates, a wrapper circuit is added with loads to the circuit under test. With a constant input net transition of 0.01ns to 1.5ns and load different capacitances, the simulation is run on two different flows, one for rise calculation and other for fall calculation. The main reason behind this split is due to the change in input waveform for NgSpice. For rise calculations, the input slope is rising and for the fall calculation the input is falling.

Since PnR tools require a specific format for the .lib file, a template file is added to the flow script. This ensures that the values are placed in a lookup table format. Therefore, the final file obtained from the end of the flow is ready to be used as an input to an Automatic Place and Route or Timing Analysis tools.

2.5.4 Power Characterization

Power extraction involves two distinct flows: one for rise power and the other for fall power. This division is primarily due to the differences of input slopes. Power extraction involves two distinct flows: one for rise power and the other for fall power. This division is primarily due to the differences of input slopes.

The inverter which was used as the reference circuit to verify the flow, is illustrated in Figure 2.7. During rise time (Fig. 2.7i), the current through PMOS i(V3) is the rise current for the circuit. This rise current charges the load capacitor C1 to the maximum voltage (1.8V). Alternatively, during fall time (Fig. 2.7ii), the NMOS is turned on and the stored charge in the load capacitor discharges to the ground. The discharge current is represented as i(V2). The

equation 5, [20] shows how the values of current can be used to calculate the dynamic power drop for the circuit.

$$P = \frac{1}{2} V_{dd} \int_{t1}^{t2} I(t)dt - \frac{1}{2} C_L {V_{dd}}^2 \qquad \textit{Equation 2.5}$$

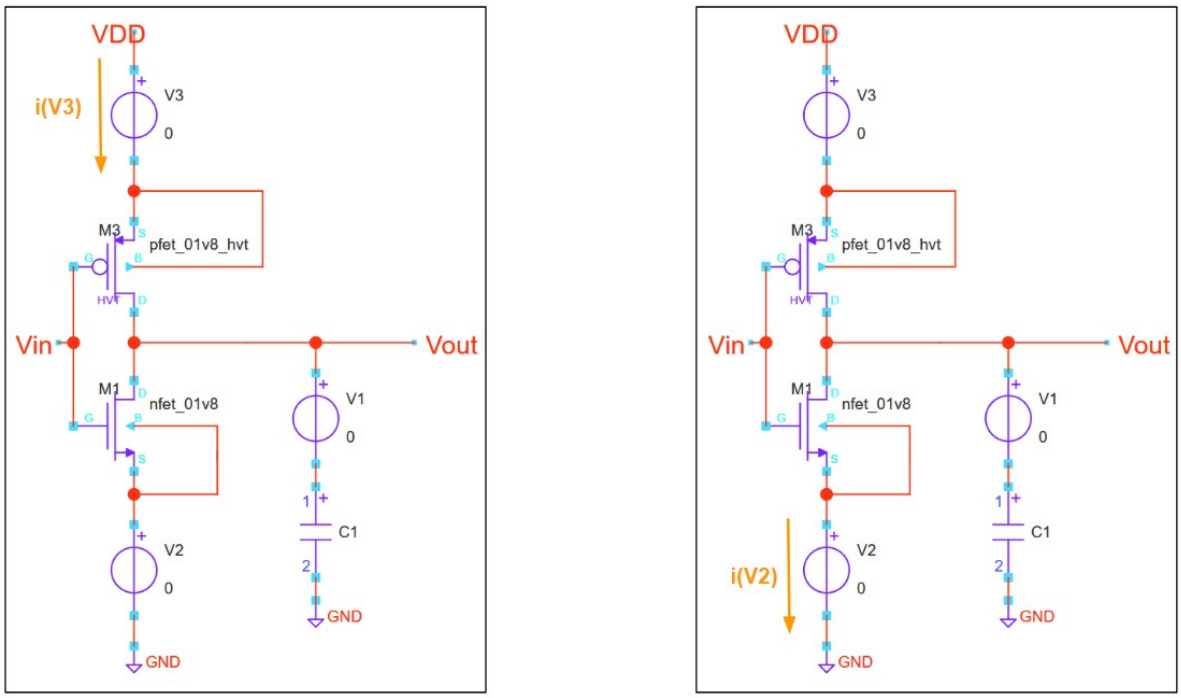


Figure 2.8. Rising Current (ii) Falling Current

2.5.5 Data Processing and Liberty File Generation

Once the NgSPICE simulations are run for all required conditions, the flow parses the output waveforms to extract key metrics. For timing, we measure propagation delays and slew (transition) times at defined voltage thresholds. For power, we integrate currents to compute dynamic energy per transition and record leakage currents. These measurements are then aggregated and formatted into a standard **Liberty (.lib) file** for the cell. The Liberty format is an industry-standard format understood by almost all downstream EDA tools (from open-source timing analyzers like OpenSTA to commercial place-and-route tools). The flow ensures the generated library file includes all the necessary entries (timing arcs, function logic, capacitances, etc.) so that the cells can be used in a digital design flow as drop-in replacements for a foundry-provided library. In essence, the output of our open characterization flow is equivalent to what a commercial tool would produce – a characterized cell library – but produced with free tools.

Crucially, the developed flow was designed to be comprehensive and cover both combinational and sequential cells. This means we can characterize not only simple logic gates but also registers (flip-flops) and other sequential elements which have setup/hold timing constraints. Support for sequential cell characterization often complicates the flow (due to the need for multiple clock phases, measuring setup/hold, etc.), but our methodology incorporates these features so that a full standard cell library (including flip-flops, latches, etc.) can be handled. The flow leverages open-source Python libraries and configuration files to manage the characterization settings (e.g., defining the PVT corners, the range of output loads and input slews to simulate, etc.), making it flexible to adapt to other PDKs or cell designs in the future. By using Magic for layout and extraction, NGSPICE for circuit simulation, and Python for orchestration and data processing, we ensure that every component of the characterization

process is transparent and modifiable by users. This level of openness is particularly empowering for academic use, as it allows students not only to use the tools but to understand and experiment with the characterization process itself, which is often a “black box” when using commercial software.

To design the RISC V CPU the standard cells are characterized under the parameters listed in the following table.

Parameter	Values
Process Technology	130nm
Process Corners	Fast, Typical, Slow
VDD Voltage (V)	1.6, 1.8, 1.95
Temperature (°C)	-40, 25, 100
input_net_transition (ns)	0.0100000000, 0.0230506000, 0.0531329000, 0.1224740000, 0.2823110000
output_net_capacitance (fF)	0.0005000000, 0.0012632100, 0.0031913700, 0.0080627200, 0.0203697000

Table 2.1 Environmental parameters

The following cells are characterized and lib and lef files are generated for the Automated Placement and Route tool to use in the RISC V synthesis and P&R.

1. Buffer
2. D Flip Flop
3. Inverter
4. Mux
5. Nand
6. Nor

To ensure authenticity of the generated libraries, the values are compared against the values obtained from the SKY130 Library characterized using Liberate.

Chapter 3 : Designing the Processor

3.1 Processors (CPU)

A central processing unit (CPU) is the primary component of a computer responsible for executing instructions and processing data. It is often called the “brain” of the computer, performing all fundamental operations and decision-making tasks [22]. The CPU operates by repeatedly fetching instructions from memory, decoding them, executing the specified operations, and then storing the results. This cycle – commonly known as the fetch–decode–execute cycle – underlies all program execution. Modern CPUs implement this cycle in a pipelined fashion, splitting execution into stages (instruction fetch, decode, execute, memory access, and write-back) so that multiple instructions can be processed in parallel (one in each stage) [27]. Each stage has a specific role: the fetch stage retrieves the next instruction from memory, decode interprets the instruction and prepares operands, the execute stage carries out the operation (often via the ALU), a memory stage reads/writes data from memory if needed, and the write-back stage writes the result to a register [27]. This pipelined design greatly improves throughput while maintaining the correct sequential semantics of the program.

At a high level, a CPU consists of several key components that work in unison (Figure 3.1). The Control Unit (CU) orchestrates the overall operation: it reads the current instruction from memory, decodes it, and issues control signals to other components to carry out the instruction [25]. The Arithmetic Logic Unit (ALU) is responsible for performing arithmetic calculations (such as addition, multiplication) and logical operations (such as AND/OR comparisons) on the data [26]. Operands for the ALU are provided by the CPU’s registers, which are small, fast storage locations within the CPU. Registers temporarily hold instruction, operands, addresses, and results; for example, a special register called the Program Counter (PC) tracks the address

of the next instruction, while the Instruction Register (IR) holds the currently executing instruction [24]. General-purpose registers store intermediate values during computations so that the ALU can access them much faster than if it had to fetch data from main memory [26]. Together, these components execute machine instructions and implement the abstract instruction cycle defined by the processor's architecture.

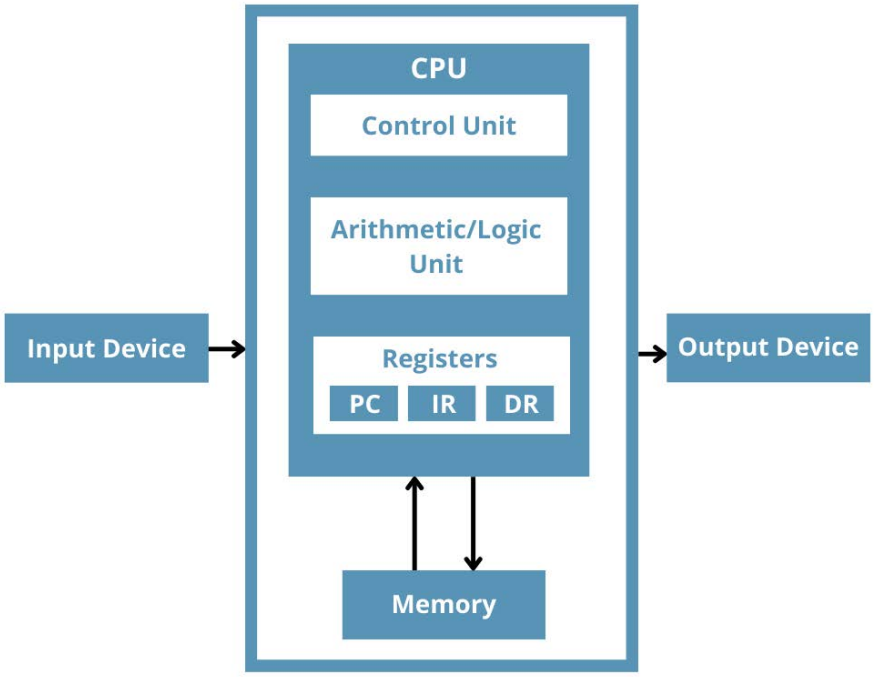


Figure 3.1. Simplified CPU Architecture (von Neumann)

Fig 3.1 shows the Control Unit, Arithmetic/Logic Unit, and registers within the CPU, along with connections to memory and input/ output devices. The Control Unit coordinates data flow between the CPU's internal units and external memory or I/O by sending control signals and managing buses [23]. The ALU performs the actual arithmetic or logic operations on data loaded from registers and then writes results back to registers [25]. Special registers (such as the PC and IR) are used to fetch and hold instructions, enabling the CPU to step through a program sequentially. This organization allows the CPU to repeatedly fetch instructions from memory, decode them, and execute them on the ALU, storing results and then advancing to the next instruction – the essence of the fetch–execute cycle [27].

3.2 Instruction Sets

An instruction set is the collection of machine-language instructions that a processor can execute – essentially, it defines the capabilities of a CPU at the lowest level of software. In processor architecture, the instruction set is a critical part of the interface between software and hardware [30]. It includes the binary encodings of each instruction and their semantic behavior (what operation the CPU will perform, and which registers or memory locations are affected). A well-defined instruction set allows machine code or assembly programs to run on any processor implementing that set, which is crucial for software portability and compatibility across different CPU designs. Instruction sets typically provide a variety of instruction types, each serving different purposes. These can be broadly categorized as follows:

- **Data movement instructions –** transfer data between memory, registers, and I/O. For example, a MOV instruction in x86 copies data from one register or memory location to another, while load/store instructions (e.g., LW/SW in MIPS) read from or write to main memory [28]. These instructions do not modify data values; they simply move operands to where they are needed for computation or store results to memory. Arithmetic and logic instructions – perform computations and evaluations. Examples include ADD or SUB (addition/subtraction of integers), and bitwise logical operations like AND, OR, XOR on registers or memory operands [28]. These instructions are handled by the ALU and often set condition codes or flags based on the outcome (e.g., setting a zero flag if a result is zero).
- **Control flow instructions –** alter the sequence of execution of instructions, enabling loops, conditional execution, and function calls [28]. This category includes jump/branch instructions (e.g., JMP for an unconditional jump, or BEQ – branch if equal – for a conditional jump based on a comparison) and call/return instructions for invoking and returning from subroutines. Control flow instructions update the PC

(program counter) to a new address, instead of simply advancing to the next sequential address.

- **String and block operations –** specialized instructions (mainly in complex instruction set architectures like x86) that operate on sequences of data (often arrays or strings in memory). For instance, x86 has instructions like MOVS/MOVSB to move a block of bytes from one memory location to another, CMPS to compare two-byte strings, or SCAS to scan a string for a value [28]. These instructions implicitly use registers (such as SI/DI in x86) as pointers and can repeat operations on multiple data elements, providing an efficient way to handle arrays or text.
- **System (privileged) instructions –** instructions reserved for operating system use or special hardware control. They perform tasks such as managing processor state, configuring hardware, or handling interrupts [22]. Examples include the INT instruction in x86 (software interrupt/trap, used for system calls) and IRET/RETD (return from interrupt), as well as instructions to manipulate control registers (like loading a page table base register) or to halt the processor (HLT) [28]. These instructions often can only execute in supervisor mode, as they directly affect hardware configuration or interact with the OS kernel.

3.3 Instruction Set Architecture (ISA)

In computer architecture, Instruction Set Architecture (ISA) is the formal definition of the interface between software and hardware. The ISA specifies what a processor can do (the instructions it can execute and the resources available, such as registers and memory addressing modes), without mandating how the processor is implemented internally [30] [31]. In essence, it is an abstract model of the computer, defining the machine's language and behavior as seen by a machine-language programmer or compiler. Key elements of an ISA include the set of opcodes (machine instructions) and their binary encodings, the available data types, the general-purpose and special-purpose registers, the memory addressing scheme, and the expected effects of each instruction on the processor state [22]. By providing this contract between software and hardware, the ISA allows software to be written independently of the specific microarchitecture of the CPU. Different CPUs can implement the same ISA in various ways (e.g. a high-performance out-of-order microarchitecture or a simple in-order design), yet all will execute the same binary instructions correctly if they adhere to the ISA. This standardization is crucial: it enables compatibility across multiple generations or vendors of processors and allows operating systems and compilers to target the ISA rather than each unique hardware design [31]. For example, the x86 ISA has remained largely backward-compatible for decades, so an x86-64 program can run on processors from different manufacturers as they all implement the same ISA specification. One prominent modern ISA is RISC-V, which exemplifies the ISA's role as a hardware-software contract. RISC-V's design is modular: it defines several base ISAs and a set of optional extensions. The base ISA (denoted RV32I, RV64I, or RV128I) provides a core set of integer instructions for a given register width (32-bit, 64-bit, or 128-bit) [32]. Table 1 compares the characteristics of these base ISAs. Each base ISA includes roughly 40–64 fundamental instructions and supports a certain address size and register width [33]. For example, RV32I is the 32-bit base ISA, requiring 32-bit general-

purpose registers and a 32-bit address space; RV64I is the 64-bit base ISA (with 64-bit registers and addresses), and RV128I is a 128-bit address space design. The base ISAs are intentionally minimal, supplying only the essential operations (like arithmetic, logical, load/store, and control flow instructions) needed to function as a complete CPU [32]. Higher functionality is then added via standard extensions (for floating-point, multiplication, atomic operations, etc.) as needed. All RISC-V base instructions are encoded in a fixed 32-bit length for simplicity, with an optional 16-bit compressed format extension available [32]. Importantly, software compiled for a base ISA will run on any hardware that implements that ISA (plus any required extensions), regardless of the microarchitectural details.

Base ISA	Register & Address Size	Approx. Instruction Count	Typical Use Cases
RV32I	32-bit registers; 32-bit address space	~ 40 instructions	32-bit microcontrollers and embedded systems; small-scale processors
RV64I	64-bit registers; 64-bit address space	~ 52 instructions	General purpose 64-bit processors (desktop, server, mobile); mainstream computing
RV128I	128-bit registers; 128-bit address space	~ 64 instructions	Experimental future designs requiring extremely large memory (no commercial implementations as of 2025)

Table 3.1. Comparison of RISC-V Base ISAs (Integer)

Table 3.1 shows the different RISC-V Base ISAs [31] [32] [33] [34]. The primary difference between these base ISAs is the supported data/address width and the number of core instructions. A 128-bit address space (RV128I) would allow addressing astronomically large memory, but in practice there is little need or experience with 128-bit CPUs to date, so the RV128I specification remains provisional [34]. The 32-bit and 64-bit base ISAs (RV32I and RV64I), on the other hand, are ratified and widely used: RV32I is common in microcontroller-class devices and IoT applications, whereas RV64I targets larger systems, providing the benefits of 64-bit computing (such as the ability to efficiently handle large integers and memory

spaces in PCs, servers, and high-performance embedded processors) [34]. Despite their differences, all RISC-V base ISAs maintain software compatibility by using the same core instruction repertoire scaled to different word sizes, and all adhere to the unified RISC-V design principles. In summary, the ISA serves as the stable foundation defining how software communicates with the CPU, and the base ISA concept in RISC-V illustrates how an architecture can be designed to scale across different performance and application domains while preserving that fundamental software/hardware interface contract.

3.4 RISC-V ISA

RISC-V (Reduced Instruction Set Computing - Five) originated from the University of California, Berkeley, and is based on principles of simplicity, modularity, and extensibility. It is designed to support a wide range of computing platforms, from embedded systems to high-performance servers [35]. One of the most significant advantages of RISC-V is its openness, which allows researchers, educators, and developers to innovate without the legal and financial constraints associated with traditional ISAs [36].

RISC-V follows the RISC design philosophy, focusing on a small set of simple instructions that execute in a single cycle. This approach reduces hardware complexity and improves performance through efficient pipelining and parallelism. The base RISC-V ISA is minimal and includes only essential instructions, while additional functionality can be added through standardized or custom extensions [37].

The base integer instruction set, referred to as RV32I for 32-bit systems, includes instructions for arithmetic operations, logical operations, memory access, and control flow. It is designed to be hardware-efficient, allowing straightforward implementation in both simple and complex processors. Extensions such as M (integer multiplication and division), F (single-precision

floating point), and A (atomic instructions) can be optionally implemented, enabling tailored designs for various applications [38].

3.4.1 Benefits of RISC-V

Open Standard

RISC-V is developed under the RISC-V International organization, a global consortium of academia and industry stakeholders. Its open standard ensures that anyone can develop, modify, and implement RISC-V processors without paying royalties or signing restrictive agreements [39].

Modularity and Customization

One of the key strengths of RISC-V is its modular architecture. Designers can choose to implement only the base ISA or add specific extensions as needed. This flexibility makes RISC-V suitable for a wide range of applications, from tiny microcontrollers to advanced multicore processors [40].

Scalability

RISC-V supports both 32-bit (RV32), 64-bit (RV64), and 128-bit (RV128) address spaces, ensuring scalability across different computing domains. Its well-defined privilege modes and support for virtualization make it an attractive option for both embedded and general-purpose computing [41].

Academic and Industry Adoption

Due to its openness and simplicity, RISC-V has been widely adopted in academic institutions for teaching and research. At the same time, industry players such as Western Digital, NVIDIA, and Alibaba have invested in RISC-V for developing custom processors, further validating its practicality and potential [42].

3.5 RISC-V Architecture

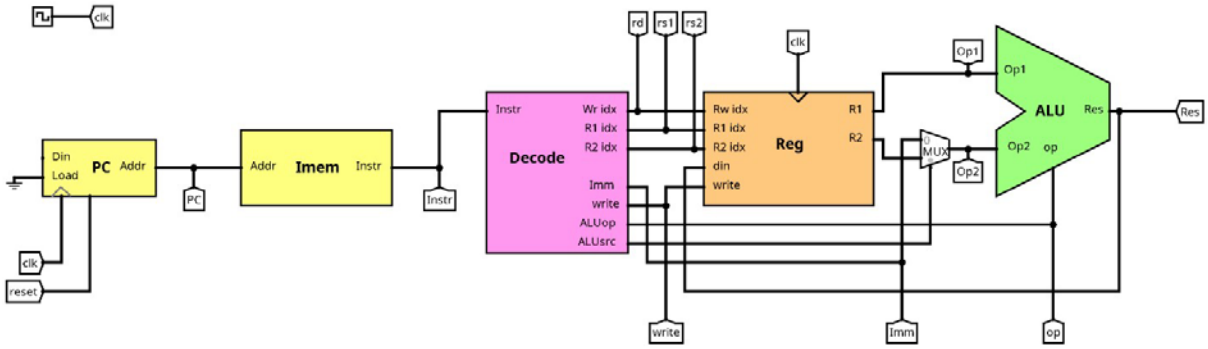


Figure 3.2. RISC-V Architecture Block Diagram

The Instruction Memory Unit will have all the preset instructions. The Instruction Fetch Unit will fetch the instructions one by one with the help of the Program Counter. After fetching the instructions, it will be given to the control unit. The Control Unit will segregate the instruction (analyze the functions and opcode) and it will determine the type of instruction (R, B, U, Jump).

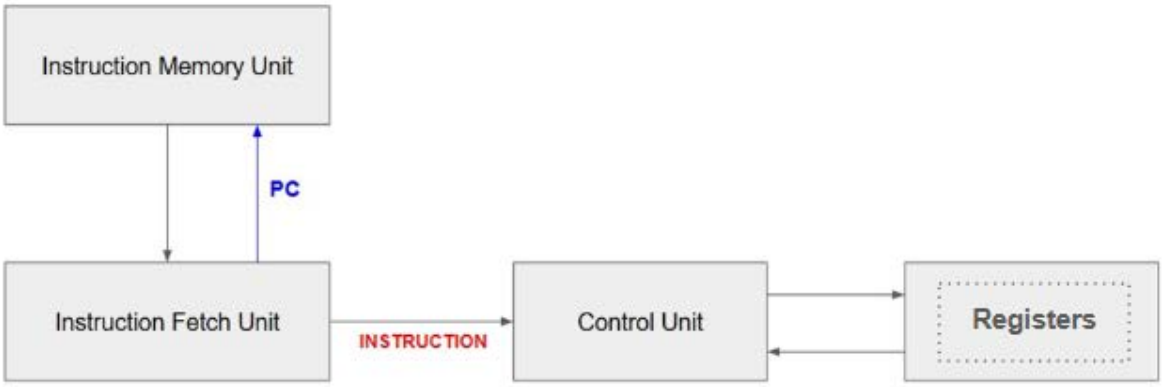


Figure 3.3. RV32 architecture Data Flow Diagram

The Data Path will have 3 units. Since it is 32-bit architecture, there are 32 registers and size of each register is 32. Each register will contain data by default. Data will be taken from registers to ALU and after operation, the results will be stored back to another register.

3.5.1 Register Unit

There are General Purpose Registers (GPB); 31 registers (x1-x31) for holding integer values.

X0 is always 0. In RV32, registers are 32 bits wide. In RV64, registers are 64 bits wide.

1. **X0 (zero):**
    - Hardwired to zero.
    - Reads always return zero, and writes are ignored.
    - Useful for initializing values and as a placeholder in instructions that require a register operand.
2. **X1 (ra - return address):**
    - Used to store the return address for function calls
    - Helps in managing function call stack and returns.
3. **X2 (sp - stack pointer):**
    - Points to the top of the stack.
    - Essential for managing stack operations, local variables, and function call management.
4. **X3 (gp - global pointer):**
    - Points to the global data section.
    - Facilitates access to global variables and constants.
5. **X4 (tp - thread pointer):**
    - Used in multi-threaded programs to point to thread specific data.
    - Helps in efficient thread management.
6. **X5 to X7 (t0 to t2 - temporary registers):**
    - Used for temporary data storage during computations.
    - Not preserved across function calls (caller-saved).

7. **X9 (s1 - saved register):**

    - Used for preserving data across function calls (callee-saved).

8. **X10 to X11 (a0 to a1 - argument registers):**

    - Used for passing arguments to functions.
    - Also used for returning values from functions.

9. **X12 to x17 (a2 to a7 - argument registers):**

    - Additional registers for passing arguments to functions.

10. **X18 to X27 (s2 to s11 - saved registers):**

    - Used for preserving data across function calls (callee-saved).

11. **X28 to X31 (t3 to t6 - temporary registers):**

    - Used for temporary data storage during computations.
    - Not preserved across function calls (caller-saved).

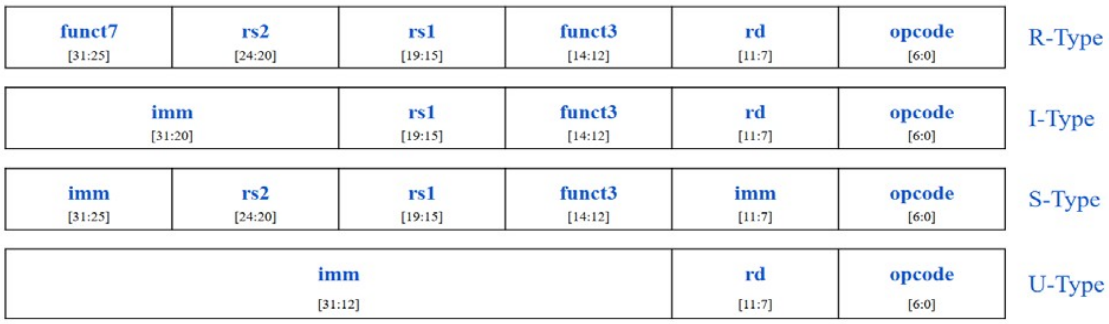


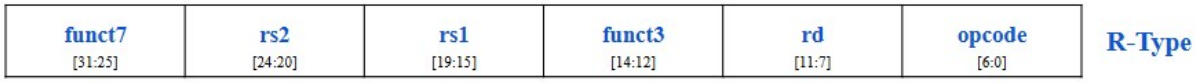
Figure 3.4. Basic Instruction Format in RV32

3.5.2 Instruction Memory Unit

The Instruction Memory Unit in this RISC-V processor has three different types of instructions coded into it.

1. **R-type instruction**

(Register type) instructions in the RISC-V architecture are used for arithmetic and logical operations between registers.



- funct7 (7 bits): Function code that, together with funct3, specifies the exact operation.
- rs2 (5 bits): Address (or index) of the second source register.
- rs1 (5 bits): Address (or index) of the first source register.
- funct3 (3 bits): Function code that, together with funct7, specifies the exact operation.
- rd (5 bits): Address (or index) of the destination register where the result is written.
- opcode (7 bits): Specifies the type of operation (e.g., arithmetic, logical)

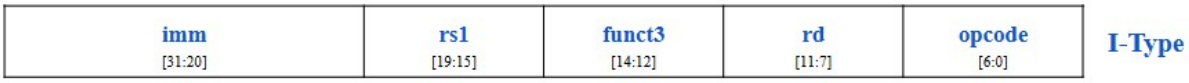
Instruction	Description	Operation
ADD	Addition	rd = rs1 + rs2
SUB	Subtraction	rd = rs1 - rs2
SLL	Shift Left Logical	rd = rs1 << rs2
SLT	Set Less than	rd = (rs1 < rs2) ? 1:0
XOR	Bitwise XOR	rd = rs1 ^ rs2
SRL	Shift Right Logical	rd = rs1 >> rs2 (logical)
SRA	Shift Right Arithmetic	rd = rs1 >> rs2
OR	Bitwise OR	rd = rs1 | rs2
AND	Bitwise AND	rd = rs1 & rs2

Table 3.2. Examples of R-type instructions

2. I-type instruction

In RISC-V architecture, I-type (Immediate type) instructions are used for operations involving an immediate value. These instructions typically perform arithmetic and logical operations where one operand is a constant (immediate) value embedded in the instruction itself.

In R-type instruction, the values are not given directly to the ALU, instead they are stored in the register. Then the ALU accesses the values from the Registers. But in I-type the values are directly given to ALU.



- **imm (12 bits):** Immediate value (only the lower 5 bits are used for shift amount)
- **rs1 (5 bits):** Source register 1 (the register containing the value to be shifted)
- **funct3 (3 bits):** Function code that specifies the operation (SLLI, SRLI, SRAI)
- **rd (5 bits):** Destination register (the register to store the result)
- **opcode (7 bits):** Operation code that identifies the instruction type

Examples of I-type instructions

Example 1: ADDI (Add Immediate)

- Description: Adds an immediate value to a register.
- Instruction:

  ADDI x5, x1, 10
- Immediate (imm): 10 (0x00A)
- Source register (rs1): x1 (1)
- Destination register (rd): x5 (5)

Example 2: SLTI (Set Less Than Immediate)

- Description: Sets the destination register to 1 if the source register is less than the immediate value, otherwise sets it to o.
- Instruction:

  SLTI x5, x1, 10

- This instruction sets x5 to 1 if the value in x1 is less than 10,
- Otherwise, it sets x5 to 0.

Example 3: OR Immediate

- Description: Performs a bitwise OR operation between a register and an immediate value.
- Instruction:

  ORI x5, x1, 0X0F

- This instruction performs a bitwise OR between the value in register x1 and the immediate value 0x0F, and stores the result in register x5.

Example 4: SLLI (Shift Left Logical Immediate)

- Description: This operation shifts the bits of rs1 left by the number of positions specified by immediate value and fills the vacated bits with 0
- Instruction:

  SLLI x10, x5, 3

- This shifts the contents of register x5 left by 3 positions and stores the result in register x10.

Example 5: SRLI (Shift Right Logical Immediate)

- Description: This operation shifts the bits of rs1 right by the number of positions specified by shift amount (immediate value), filling the vacated bits with zeros.
- Instruction:

  SRLI x10, x5, 3

- This shifts the contents of register x5 right by 3 positions (logical shift) and stores the result in register x10.

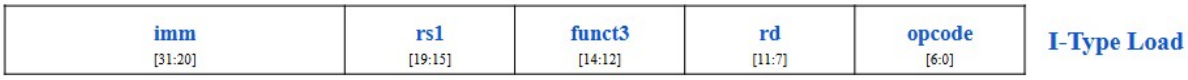
Example 6: SRAI (Shift Right Arithmetic Immediate)

- Description: This operation shifts the bits of rsi right by the number of positions specified by shift amount (immediate value), filling the vacated bits with the sign bit (the most
- significant bit of rs1).
- Instruction:

  SRAI x10, x5, 3
- This shifts the contents of register x5 right by 3 positions (arithmetic shift) and stores the result in register x10.

Example 7: Load Instructions

The load instructions use the I-type format. These instructions load data from memory into a register.



- **imm:**12-bit signed immediate (offset)
- **rs1:** Base address register
- **funct3:** Function code that specifies the operation (LB, LH, LW, LBU, LHU)
- **rd:** Destination register
- **opcode:** Operation code that identifies the instruction type
- Different types of load instructions can be as follows

Instruction	Description
LB	Load byte from memory
LH	Load halfword (16 bits) from memory
LW	Load word (32 bits) from memory
LBU	Load byte (unsigned) from memory
LHU	Load halfword (unsigned) from memory

Table 3.3. Different Load Instructions

3. B-type instruction

B-type instructions in RISC-V are used for conditional branching. They compare two registers and branches to a specified instruction if the comparison is true. B-type instructions use a specific instruction format that includes fields for the opcode, source registers, the branch target offset, and the comparison function.

- **Opcode (7 bits):** Identifies the instruction type.
- **Funct3 (3 bits):** Specifies the comparison operation.
- **rs1 (5 bits):** The first source register.
- **rs2 (5 bits):** The second source register.
- **Imm (12 bits):** The immediate value used for the branch target address, split into multiple parts across the instruction.

Instruction	Description
BEQ	Branch if the values in the two source registers are equal
BNEQ	Branch if the values in the two source registers are not equal
BLT	Branch if the value in rs1 is less than the value in rs2 (signed)
BGE	Branch if the value in rs1 is greater than or equal to the value in rs2 (signed)
BLTU	Branch if the value in rs1 is less than the value in rsz (unsigned)
BGEU	Branch if the value in rs1 is greater than or equal to the value in rs2 (unsigned)

Table 3.4. Different B-tryp instruction

Example : BEQ (Branch Equal To Instruction)

The BEQ (Branch if Equal) instruction is a conditional branch instruction used in RISC-V to control the flow of a program. It compares the values in two source registers, and if they are equal, it branches to a specified target address.

Instruction Format

The BEQ instruction uses the B-type format, which includes the following fields:

- **Opcode (7 bits):** Specifies the type of instruction.
- **Funct3 (3 bits):** Specifies the type of branch comparison.
- **rs1 (5 bits):** Specifies the first source register.
- **rs2 (5 bits):** Specifies the second source register.
- **Immediate (12 bits):** Specifies the branch target address offset

The target address is calculated as:

Target Address = PC + Immediate Target Address

Instruction:

BEQ x5, x6, label

- **x5:** The first source register.
- **x6:** The second source register.
- **label:** (immediate value) The target address, specified as a label in the assembly code.

If the value in x5 is equal to the value in x6, the program will branch to the address specified by label. If not, the next sequential instruction will be executed.

3.5.3 Instruction Fetch Unit

The Instruction Fetch Unit will fetch the instructions one by one from the instruction memory unit.

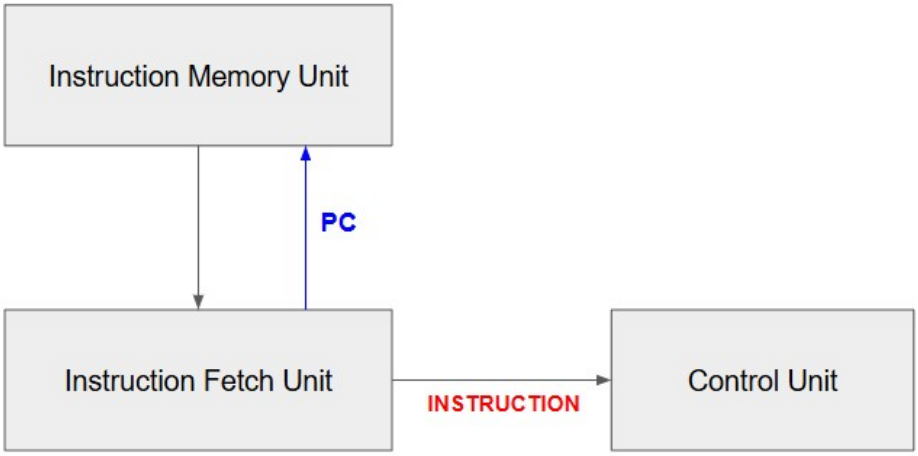


Figure 3.5 IFU connection with IMU

The design is Byte Addressable i.e for every byte (8bits) there will be one address. The PC incrementing logic will be performed by IFU and given to instruction memory.

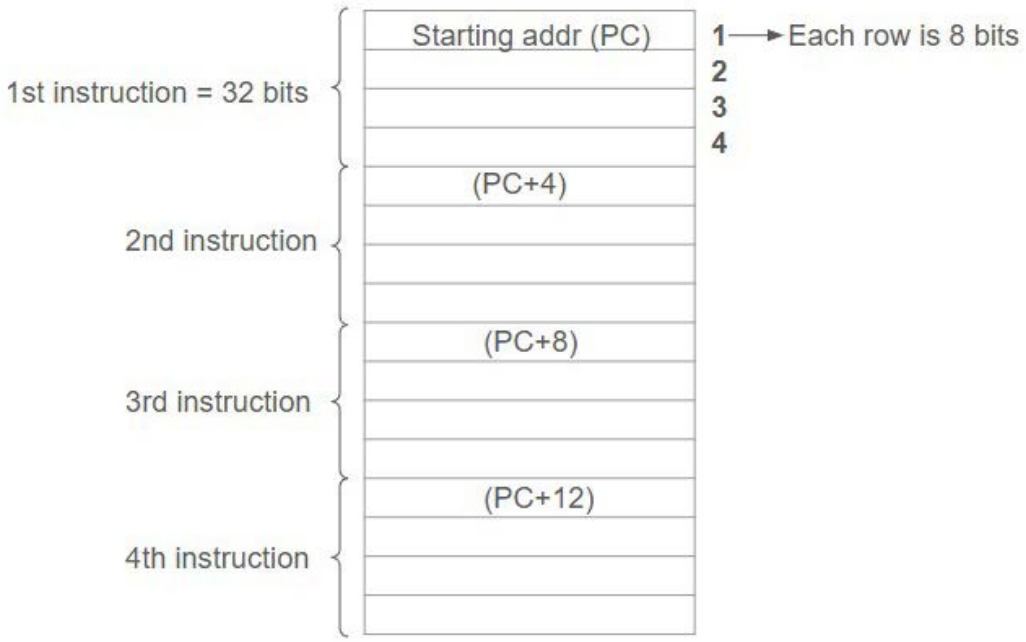


Figure 3.6 Instruction Memory

3.5.4 Control Unit

The ALU control value is dependent on funct3 and funct7 and the control unit determines the operation based on f3 and f7.

opcode	operation	funct3	Instruction	Description
0010011	Immediate calculations	000	ADDI	add immediate
		010	SLTI	set-less-than immediate (signed)
		011	SLTIU	set-less-than immediate (unsigned)
		100	XORI	bitwise-XOR immediate
		110	ORI	bitwise-OR immediate
		111	ANDI	bitwise-AND immediate
		001	SLLI	shift-left logical immediate (shamt in imm[4:0])
		101	SRLI	shift-right logical immediate
0110011	Register Calculations	000	ADD / SUB	funct7[5]=0→ADD, funct7[5]=1→SUB
		001	SLL	shift-left logical
		010	SLT	set-less-than (signed)
		011	SLTU	set-less-than (unsigned)
		100	XOR	bitwise-XOR
		101	SRL / SRA	funct7[5]=0→SRL, funct7[5]=1→SRA
		110	OR	bitwise-OR
		111	AND	bitwise-AND
1100111	Jump	000	JALR	jump and link register

opcode	operation	funct3	Instruction	Description
0000011	Load	000	LB	load byte (signed)
		001	LH	load half-word (signed)
		010	LW	load word (32-bit)
		100	LBU	load byte (unsigned)
		101	LHU	load half-word (unsigned)
0100011	Store	000	SB	store byte
		001	SH	store half-word
		010	SW	store word
1100011	Branch	000	BEQ	branch if equal
		001	BNE	branch if not equal
		100	BLT	branch if less than (signed)
		101	BGE	branch if ≥ (signed)
		110	BLTU	branch if less than (unsigned)
		111	BGEU	branch if ≥ (unsigned)

Table 3.5. Instructions coded in the RISC V CPU

3.5.5 Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) performs the arithmetic and bitwise operations defined by the control unit. This is designed using Verilog. The ALU is designed to perform the following functions.

1. Addition
2. Subtraction
3. XOR
4. OR
5. AND

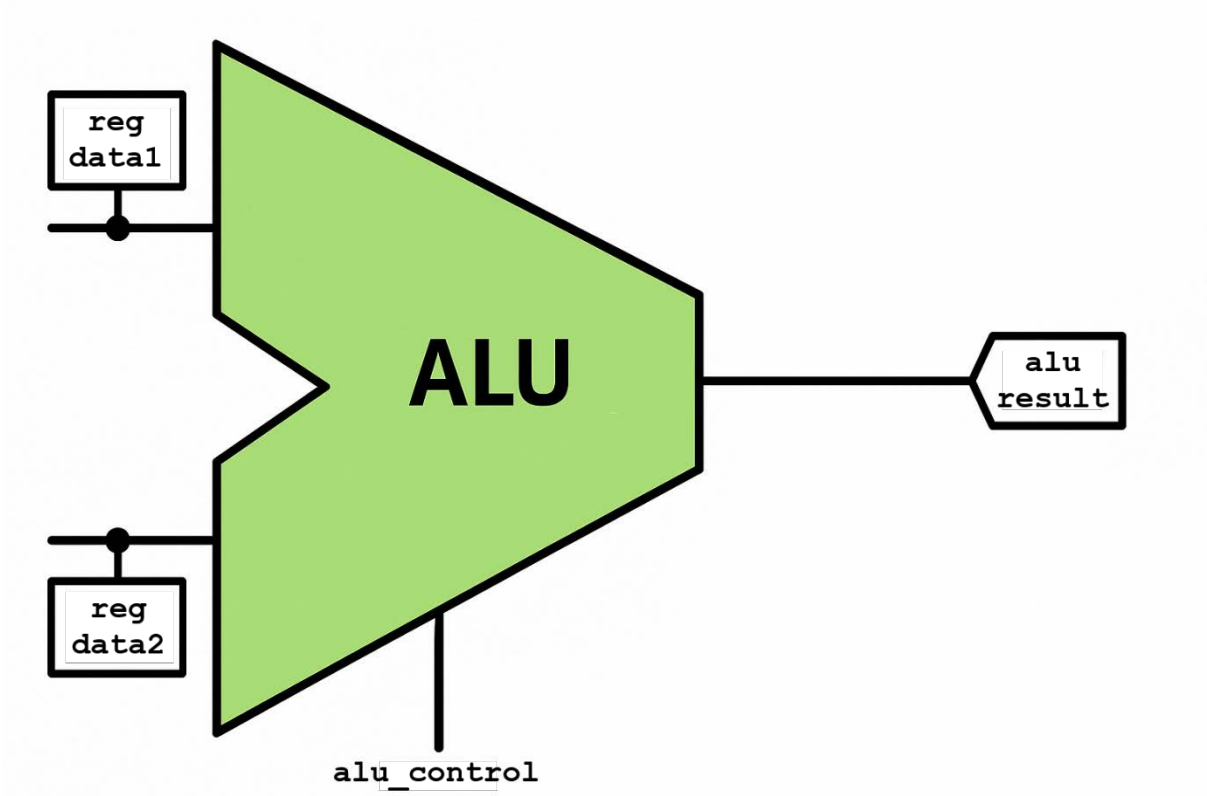


Figure 3.7 Arithmetic Logic Unit (ALU)

Chapter 4 : Physical Design of the CPU

To demonstrate the utility of this open-source characterization flow, it is applied to the development of a custom RISC-V CPU as a case study. RISC-V is a modern open instruction set architecture, and a simple 32-bit RISC-V central processing unit core is designed which is optimized for the sky130 process. Rather than using only pre-existing standard cell libraries, certain custom cells were selected for the CPU's design. These included, for example, tailored arithmetic logic unit (ALU) cells and custom flip-flops that were optimized for specific performance or area targets. Each of these cells was designed from scratch: starting with transistor-level schematics in Xschem, then layout in Magic, followed by verification (DRC/LVS). Once verified, the cells were characterized using the flow proposed in this work to generate accurate timing and power models.

All the standard cells used in the CPU (both the custom-designed ones and the cells borrowed or adapted from an existing open library) were processed through the new characterization flow to ensure consistency. This means a complete Liberty format library for the CPU's cell set is created. The open-source digital flow (synthesis and place-and-route) could then use this library to implement the CPU. Because the library was characterized at the same conditions that would be used for sign-off (and in the same format as expected), it could be seamlessly used with both open-source tools like OpenSTA (for static timing analysis) and with commercial EDA tools if needed. In fact, one of the goals was to ensure that the characterized library would be compatible with commercial place-and-route (PnR) tools. This compatibility is important because it means the open-source library could be used in a hybrid flow (for instance, a university might use open-source tools for design exploration and then use a commercial PnR tool for final implementation).

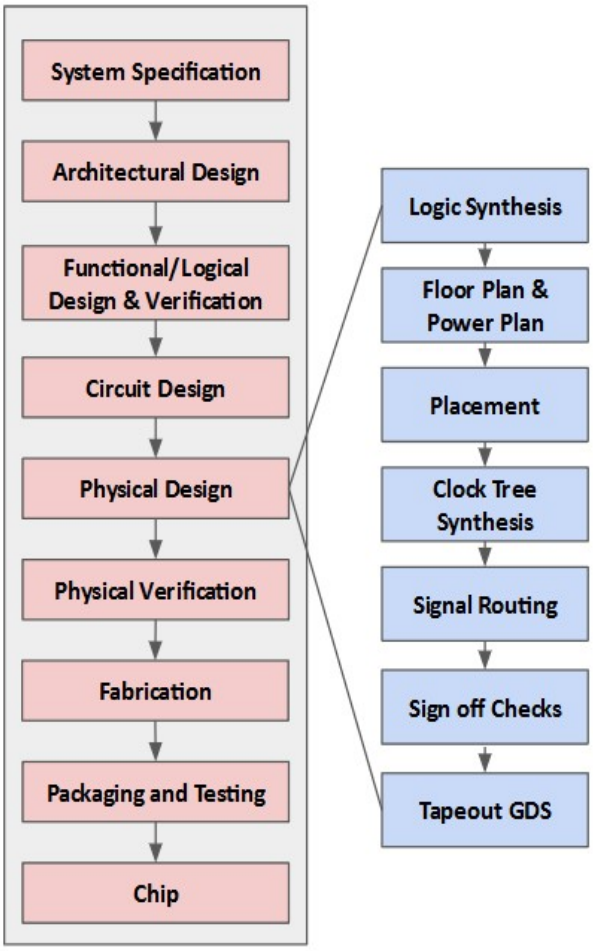


Figure 4.1. ASIC Design Flow

4.1 Standard Cell Design

The following 5 standard cells or gates were selected and designed for the purpose of building the CPU.

1. Buffer
2. Inverter
3. NAND
4. NOR
5. D flip flop

4.1.1 Layout Design

Each schematic was physically implemented in Magic, following SkyWater 130nm design rules. Cells adhere to standard-cell constraints: uniform height, VDD at top, GND at bottom, and metal1 pins aligned to grid [43][46].

4.1.1.1 Inverter

PMOS is placed in n-well at the top and NMOS in p-substrate at the bottom. A vertical poly gate spans both transistors. Source/drain contacts connect to VDD, GND, and output via local interconnect (LI) and metal1. Pins A (input) and Y (output) are routed on metal1, staggered vertically [46]. For the layout with the higher strength, the diffusion layers are widened more and another poly gate is added and the two are shorted together.

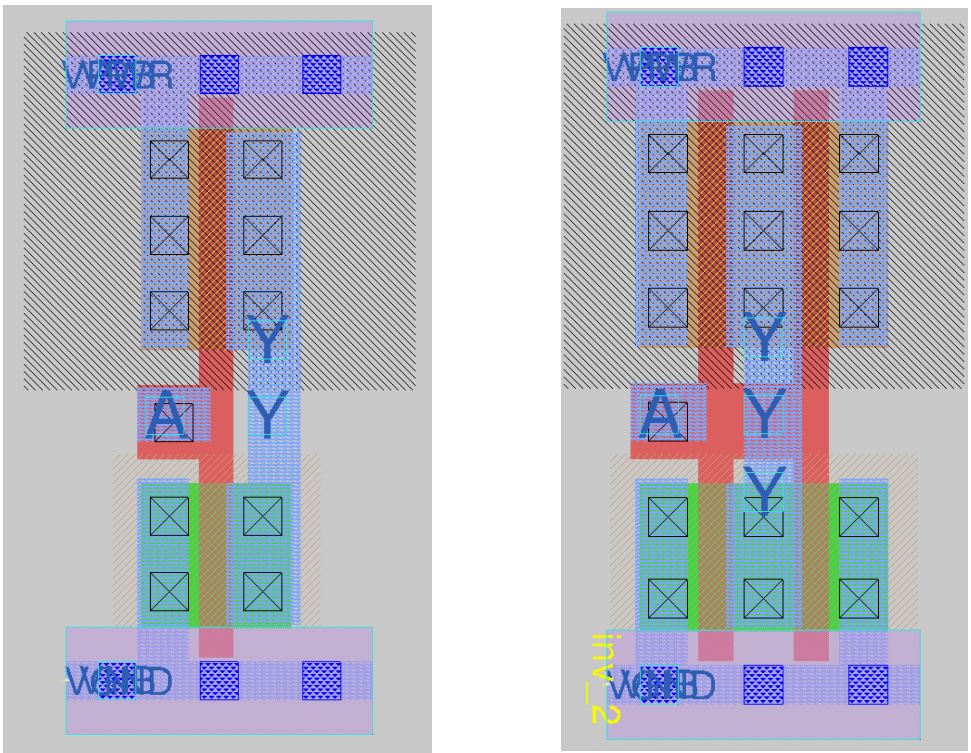


Figure 4.2. Inverter Layouts of two different strengths

4.1.1.2 Buffer

Two inverters are laid out in series. The first inverter drives the second via an intermediate metal connection. PMOS and NMOS are vertically aligned, and power rails run horizontally across the top and bottom. The layout is wider than the inverter but maintains the same cell height. Pins are located at the edges [47]. The higher strength buffer requires another extra poly line for improved performance and reduced delay.

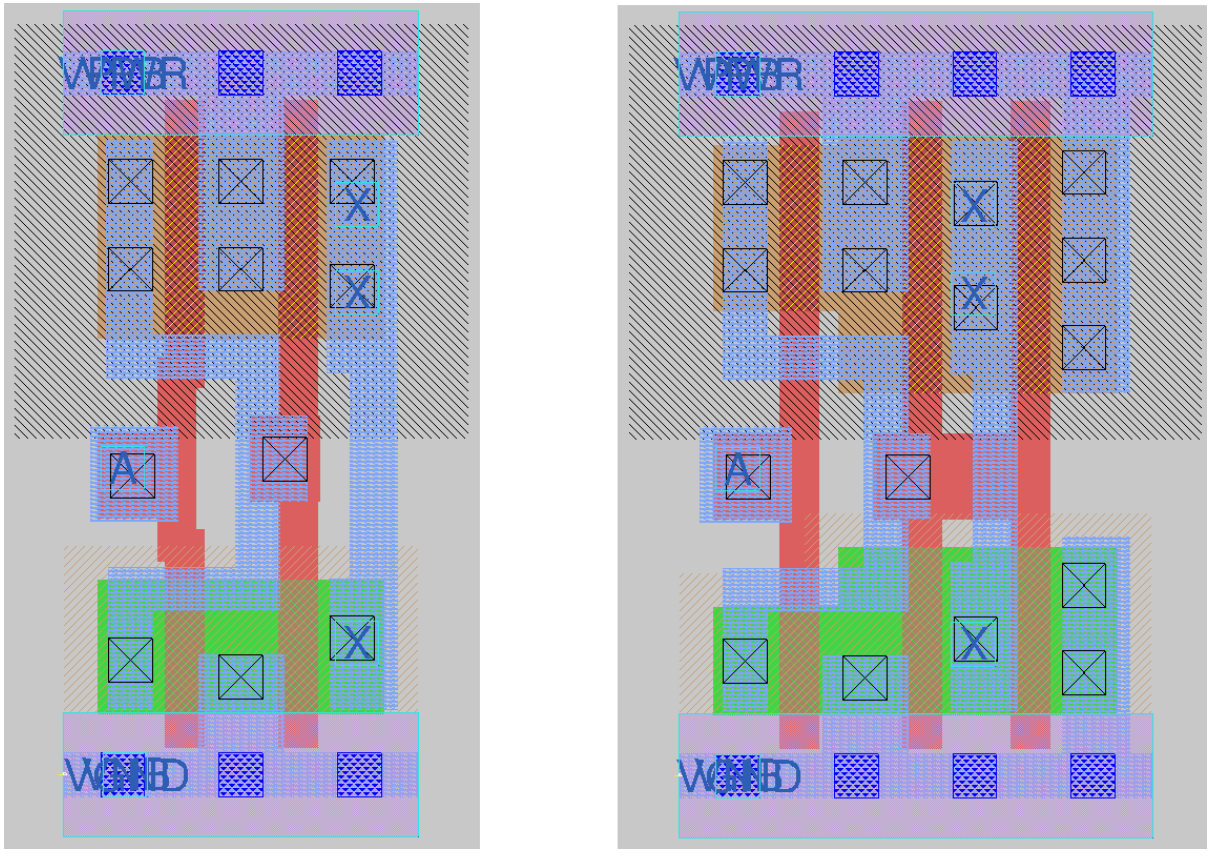


Figure 4.3. Buffers Layouts of two different strengths

4.1.1.3 2-Input NAND

PMOS devices are placed in parallel and NMOS in series. Diffusion sharing is used for NMOS series connection. Inputs A and B drive corresponding gate pairs. Metal1 rails carry power; metal1 and LI route inputs and output. PMOS and NMOS networks are placed for optimal symmetry and density [47]. The higher strength NAND requires another extra 2 poly lines for each input and also a larger diffusion area.

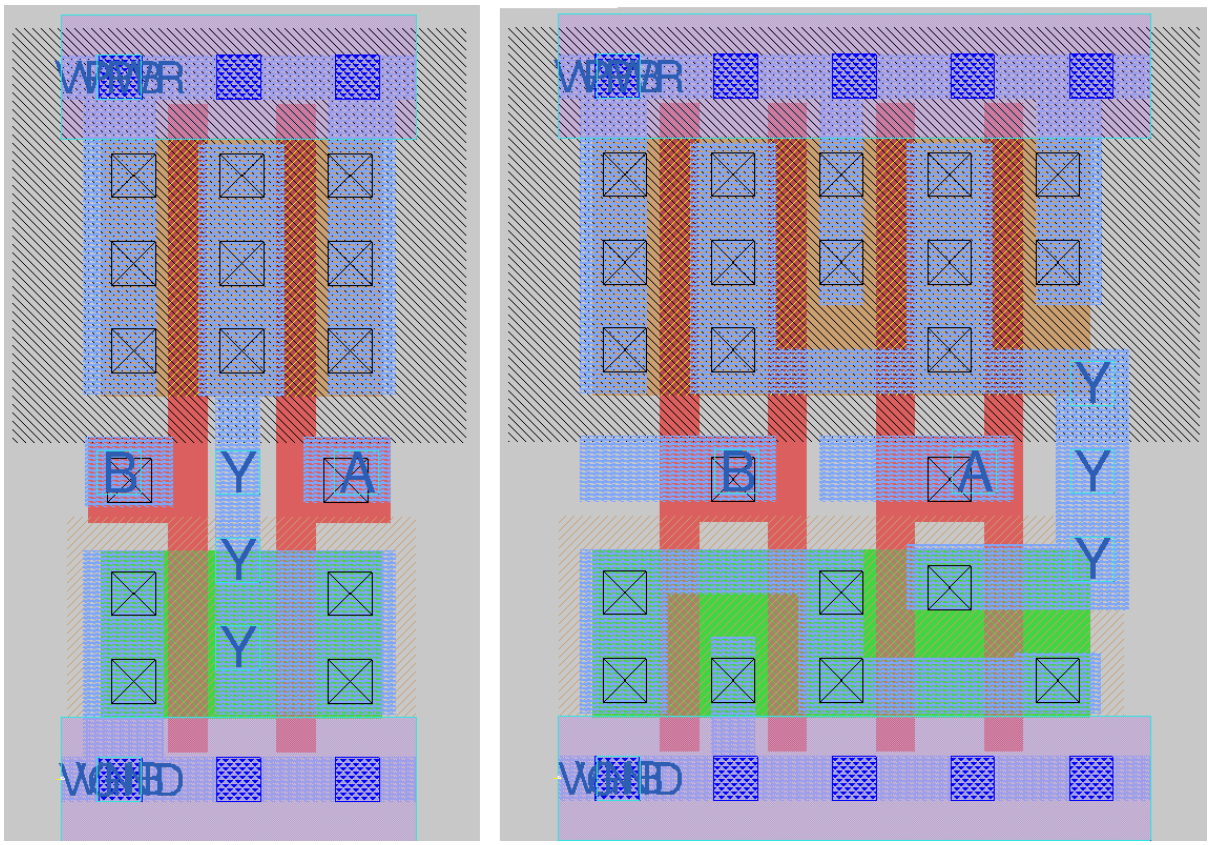


Figure 4.4. NAND Layouts of two different strengths

4.1.1.4 2-Input NOR

Inverted network to NAND; PMOS in series, NMOS in parallel. Layout also mirrors NAND, with shared diffusion between series PMOS and merged diffusion for NMOS output. Inputs connect to staggered gates to ease routing. Power and signal routing follow standard-cell rules [47]. The higher strength NOR gate requires another extra 2 poly lines for each input and also a larger diffusion area.

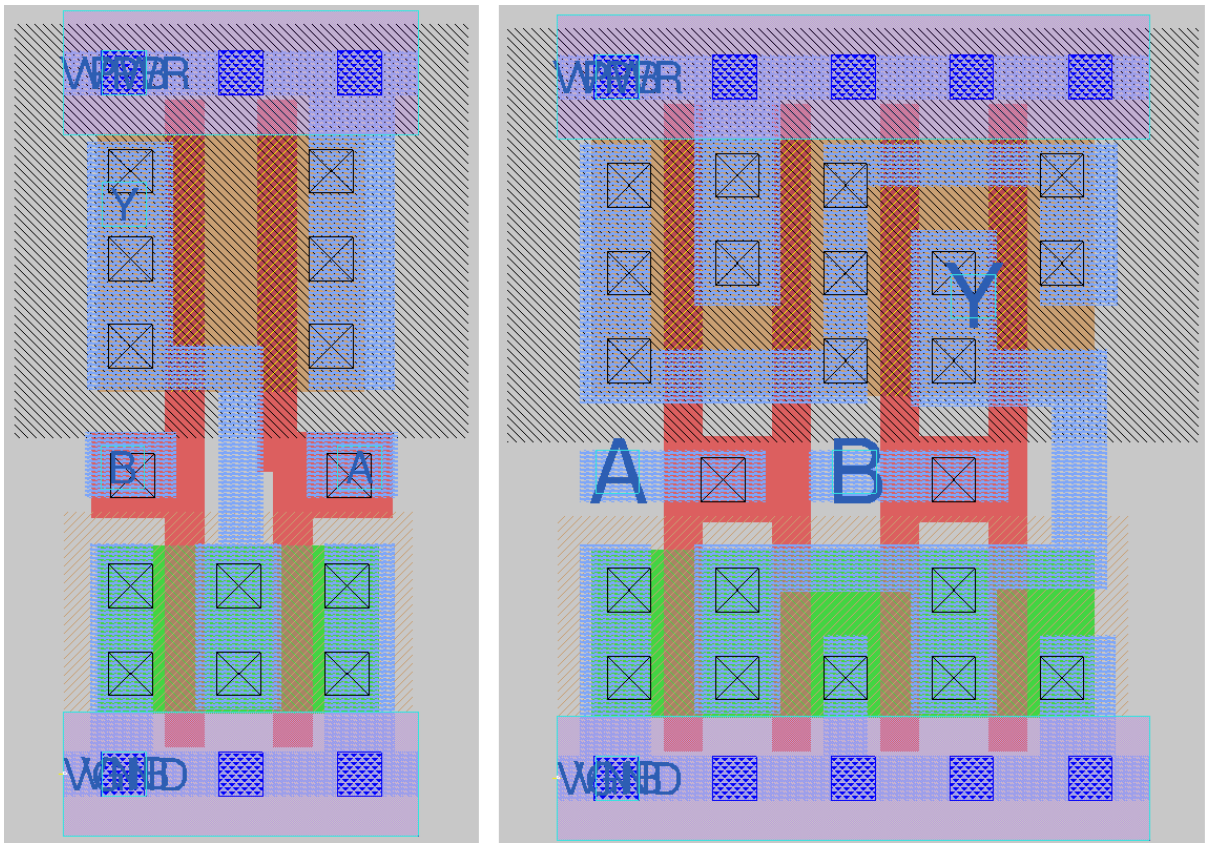


Figure 4.5. NOR Layouts of two different strengths

4.1.1.5 D Flip-Flop

The most complex cell. Layout includes clock inverter, two transmission gates, and inverter pairs for master/slave latches. TGs are implemented with NMOS/PMOS pairs sharing source/drain diffusions. Nodes between TGs and latches are minimized using shared diffusion. CLK and CLK-bar signals route across the cell. Well taps are included at edges. Output Q is routed to metal1 pin at the right edge [48].

For the higher drive strength, the inverter at the end requires and extra poly line and greater diffusion area.

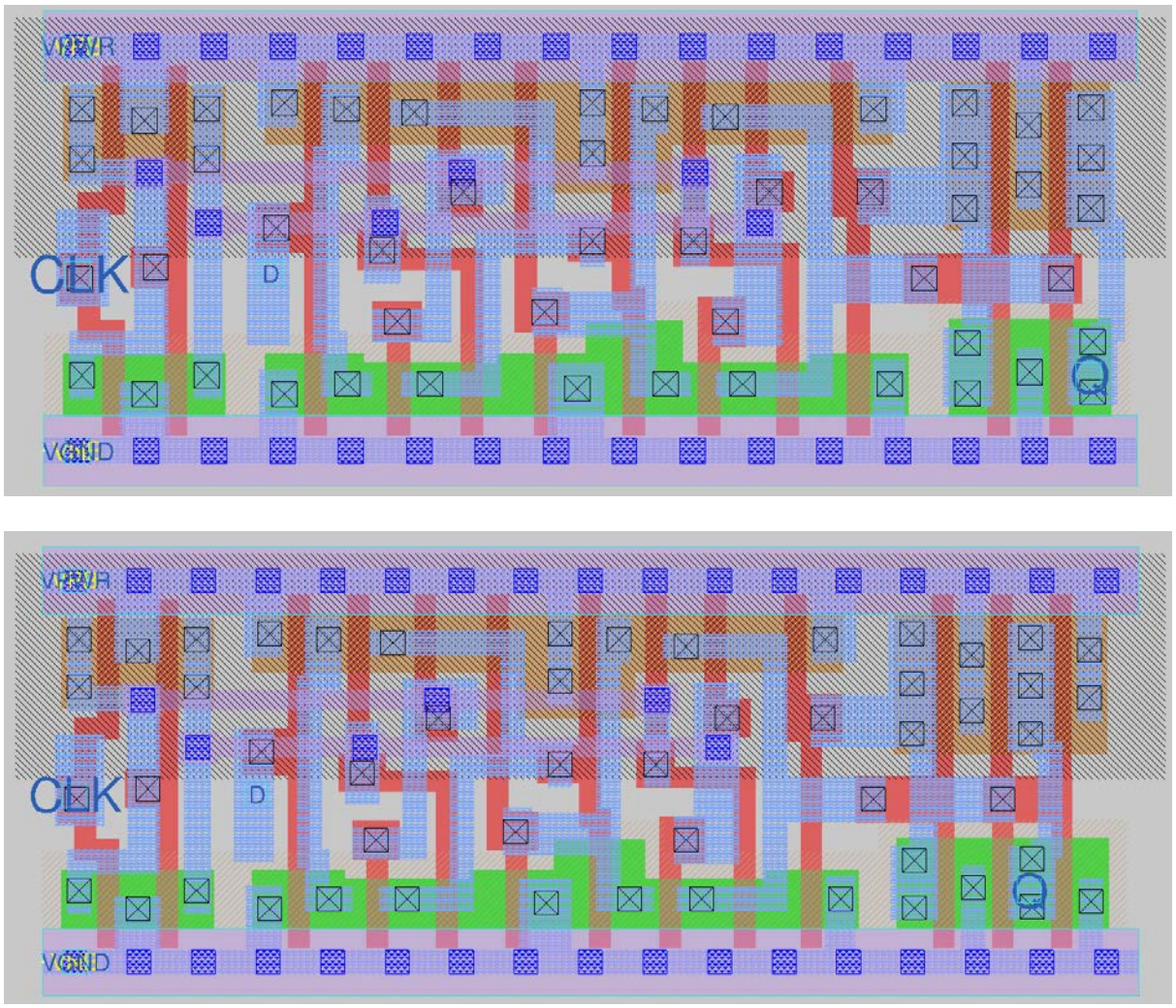


Figure 4.6. D Flip-Flop Layouts of two different strengths

4.1.2 Characterization of the cells

The methodology described in this thesis employs several open-source tools to conduct both the timing and power characterization of the target standard cell. The input for this process is the SPICE-extracted model of the netlist obtained from the opensource Magic layout tool. Fig. 4.6 illustrates the comprehensive flowchart detailing the proposed methodology for library characterization.

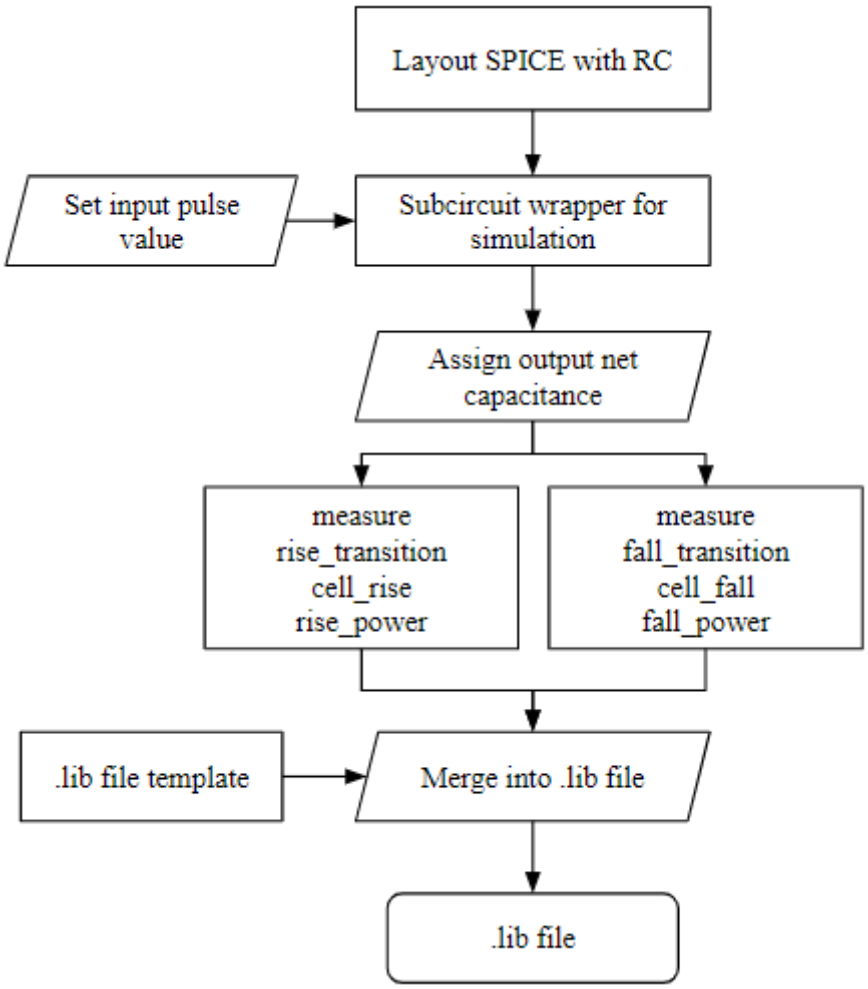


Figure 4.7. Library characterization procedure

In this work, both resistance and capacitance extractions are conducted for the logic gate. Magic can readily extract node capacitances by default without encountering any problems. However, to extract resistance, the tool needs to undergo multiple extraction steps [11] [8]. To extract the Liberty or .lib file, the input files include the circuit SPICE netlist generated from

the layout and the Sky130 device model. The characterization flow comprises several steps outlined in Fig. 4.6. To simulate each input pin of the gates, a wrapper circuit is introduced with connects to the circuit under test. The simulation is then executed employing a constant input net transition of 0.0005000000, 0.0012632100, 0.0031913700, 0.0080627200, 0.0203697000 and varying load capacitances.

Parameter	**Values**
Process Technology	130nm
Process Corners	Fast, Typical, Slow
VDD Voltage (V)	1.6, 1.8, 1.95
Temperature (°C)	-40, 25, 100
input_net_transition (ns)	0.0100000000, 0.0230506000, 0.0531329000, 0.1224740000, 0.2823110000
output_net_capacitance (fF)	0.0005000000, 0.0012632100, 0.0031913700, 0.0080627200, 0.0203697000

Table 4.1. Environmental parameters

Power extraction involves two distinct flows: one for rise power and the other for fall power. This division is primarily due to the differences of input slopes. During rise calculation, the input slope rises, while during fall calculation, the input slope falls. There will only be two states (rising and falling) for the one input pin and one output pin. To accommodate the specific format required by PnR (Place and Route) tools for the .lib file, a template file is integrated into the flow script. This ensures that the values are structured in a lookup table format. Consequently, the final file obtained at the conclusion of the flow is prepared to be directly utilized as an input for Automatic Place and Route or Timing Analysis tools.

4.2 ASIC design of the CPU

Designing a digital ASIC (Application-Specific Integrated Circuit) involves a sequence of steps that transform an RTL hardware description into a manufacturable physical layout. The major stages include RTL design and verification, logic synthesis, floorplanning, placement, clock tree synthesis, routing, and physical verification (sign-off) [52]. In essence, the goal is to start with a Verilog RTL of the RISC-V CPU and end with a GDSII file ready for fabrication. Each stage uses specific EDA tools and leverages the standard cell library of the SkyWater 130 nm process to incrementally build and validate the chip design.

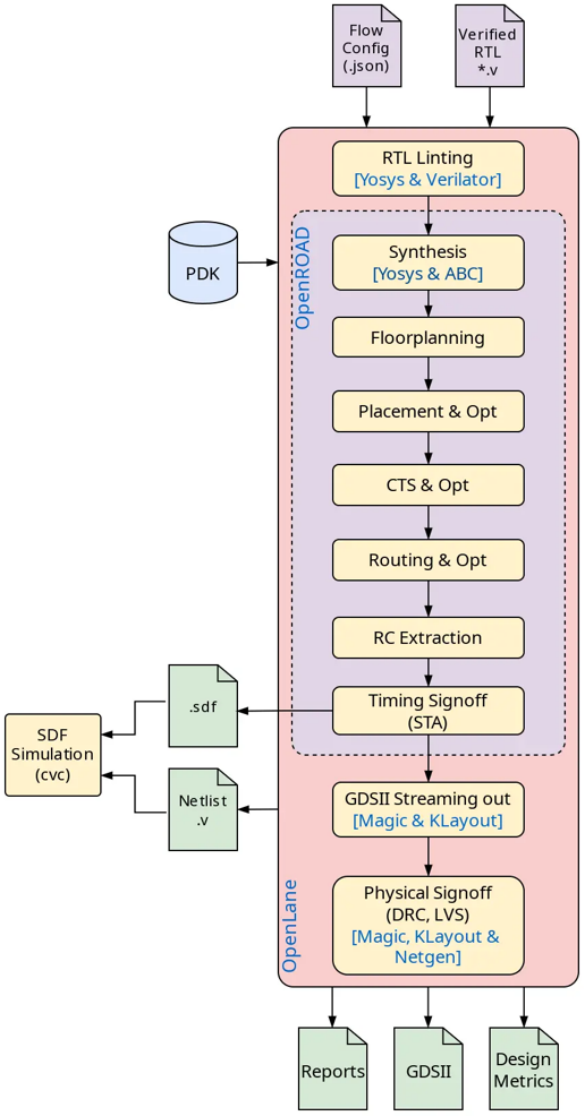


Figure 4.8. OpenLane ASIC Flow [52]

4.2.1 RTL-to-GDSII Implementation Flow with OpenLane

OpenLane orchestrates the entire digital back-end flow for the design. The input to the flow is the Verilog RTL of the RISC-V CPU (assumed functionally verified), along with design constraint files (like timing constraints) and the Sky130 PDK libraries. A single OpenLane configuration (e.g., config.json) controls various options for the run [53]. OpenLane then invokes a series of tools/stages automatically, passing the design through synthesis, floorplanning, placement, CTS, routing, and sign-off, to finally produce the GDSII layout and reports. All the heavy steps from floorplanning through routing are handled inside the integrated OpenROAD tool infrastructure [53], whereas OpenLane also calls external tools for tasks like synthesis and verification.

4.2.1.1 Logic Synthesis (RTL to Gate-Level Netlist)

Logic synthesis is the process of translating the RTL code of the CPU into a gate-level implementation composed of standard cell instances. In OpenLane, this stage is performed by Yosys (for HDL synthesis) followed by ABC (for technology mapping) [53]. The new characterized libraries are used in this stage. Yosys reads the RTL and the cell library's timing and functional models (the .lib timing file and Verilog models for cells) to infer an optimized network of those cells. The output is a synthesized Verilog netlist where each abstract RTL operation is now implemented by a specific standard cell (e.g., NAND, NOR, INV, BUFF, D flip-flop, etc.), all of which exist in the newly characterized library.

During synthesis, the tool also needs to meet timing constraints (like the target clock period) by choosing faster gates or adding pipeline registers if necessary. Static timing analysis using OpenSTA is run on the resulting netlist to evaluate whether timing requirements are met on paths in the design [53]. If not, the synthesis step may be iterated with different constraints or strategies. This allows designers to select a synthesized netlist that best balances performance

and area for the RISC-V core. The primary outputs of the synthesis stage are the gate-level netlist (Verilog) and synthesis reports/logs (cell counts, timing slack, etc.). These serve as inputs to the physical design stages.

It is important to note that the standard cell library's characteristics heavily influence synthesis. Since there are multiple drive strengths and gate variants in the library, Yosys/ABC will pick appropriate cells to meet timing. The .lib timing view of each cell is used by the synthesizer and STA to estimate path delaysschaumont.dyn.wpi.edu.

For example, the two-input NAND in the new library has known propagation delays defined in the library; if a certain path is slow, the synthesizer might swap in with the higher-drive buffer.

4.2.1.2 Floorplanning and Power Planning

Floorplanning is the first stage of physical implementation. Here, the chip die area and the arrangement of major components are decided. Using OpenLane/OpenROAD, the initial floorplan (init_fp) step defines the core region for the design and allocates rows for standard cells and routing tracks across that area [53]. The floorplan must accommodate all the standard cells from synthesis with a target utilization (ratio of cell area to core area) that leaves space for routing and future buffer insertions. OpenLane allows configuring parameters like core aspect ratio, utilization percentage, and cell padding in the floorplan configuration.

If the RISC-V CPU design includes any hard macros (pre-designed blocks such as SRAMs or analog macros), these are placed during floorplanning (often referred to as macro placement). For example, a cache memory macro or a pre-hardened CPU core would be positioned at this stage. Macros are placed near related logic to minimize wiring; for instance, an SRAM macro might be placed adjacent to the CPU core's memory interface. In this work, a standalone CPU core is designed. Therefore, there are no extra hard macros here.

Another critical part of floorplanning is I/O placement. OpenLane uses the IOPlacer utility to place input/output pads or pins around the periphery of the core according to constraints [53]. The user can specify pin locations or let the tool distribute them.

After defining the core area and I/O pins, power planning is performed. The tool PDNgen (Power Distribution Network generator) creates a network of power/ground rails – typically a set of horizontal and vertical metal stripes – that will deliver power (VDD) and ground (VSS) uniformly across the chip [53].

In Sky130, a common scheme is to have vertical power rails on one metal layer and horizontal rails on another, forming a grid. These rails are strapped to the power pins of the standard cells in each row. By having VDD and VSS available nearby, the cells can reliably drive signals without IR drop or ground bounce issues. The inserted power grid prevents excessive voltage droop by providing multiple current paths and nearest connections for the cells. Power planning must also respect electromigration limits and ensure each region of the chip has sufficient power delivery.

Additionally, OpenLane inserts special filler and decoupling capacitor cells during floorplanning via the Tapcell steparmleo-openlane.readthedocs.io. Well taps (tap cells) are placed periodically in the rows to tie the p-type substrate and n-wells to the power rails, ensuring that the substrate/well biases remain stable (this prevents latch-up and maintains transistor threshold conditions) [53]. Dedicated tap cells are used for this purpose (since its standard cells do not have built-in substrate contacts) [53]. Decap cells (on-chip decoupling capacitors) may also be inserted in empty areas to stabilize supply voltage by providing local charge reservoirs. Filler cells are used to fill any leftover gaps between placed cells so that there are no discontinuities in the well and poly geometries – ensuring DRC cleanliness and a contiguous row structure.

The result of floorplanning is an initial DEF (Design Exchange Format) file capturing the floorplan: the core area dimensions, placement of any macros, reserved regions, the coordinates of I/O pins, the power grid geometry, and inserted tap/decap cells. At this stage, large empty areas exist where standard cells will later be placed, but the “framework” for placement is ready.

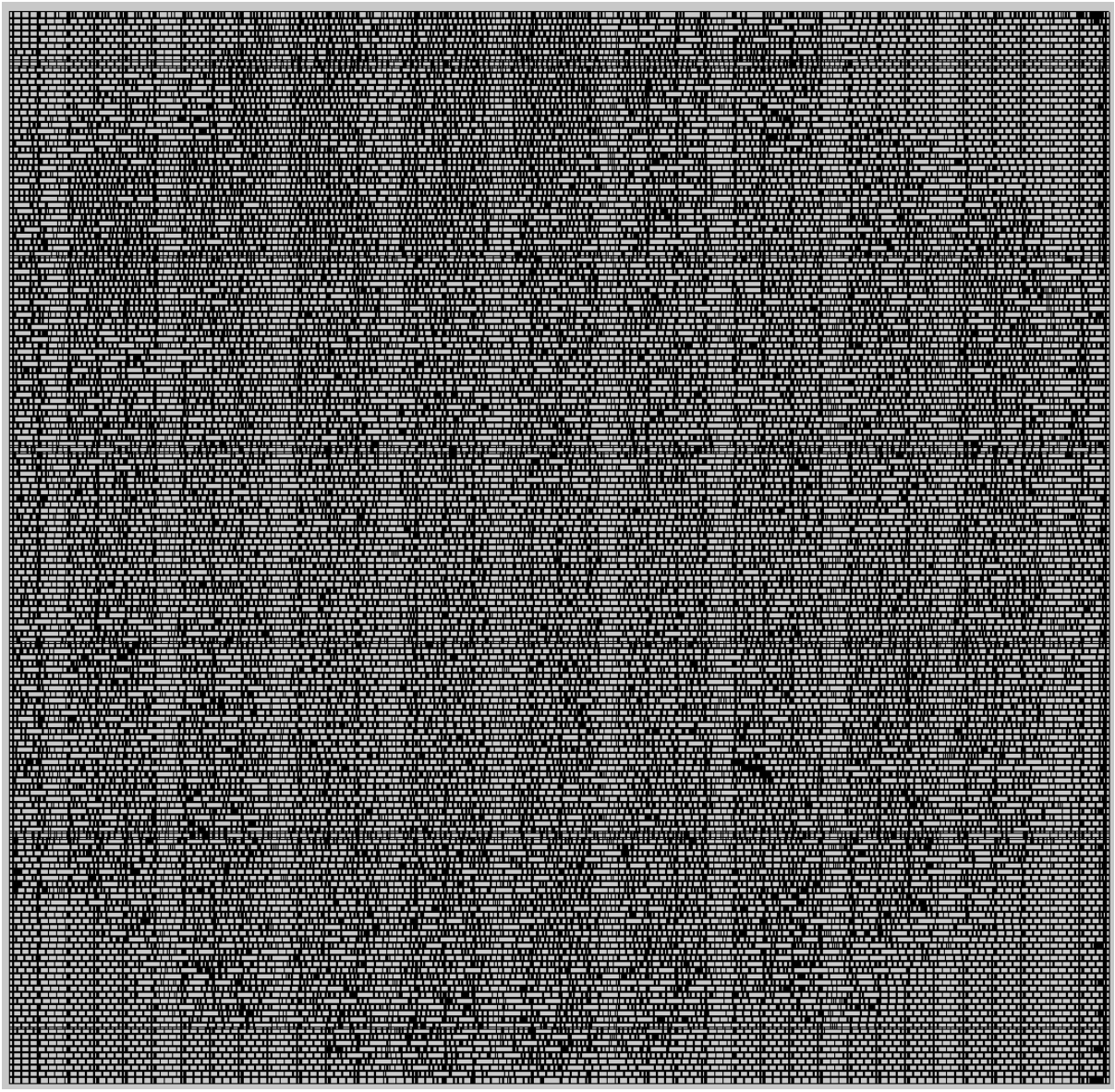


Figure 4.9. Final Floorplan of CPU (with filler cells)

4.2.1.3 Placement of Standard Cells

Once floorplanning and power planning are completed, the next step is placement of the standard cells. Placement entails deciding the exact location of every logic gate (standard cell instance) in the synthesized netlist, within the rows defined by the floorplan. OpenLane employs the OpenROAD placement engine in two phases: global placement followed by detailed placement [53].

Global placement is performed by the RePlAce algorithm (integrated in OpenROAD) [53]. In this phase, all standard cells are treated somewhat like flexible “dots” that can be moved continuously. The placer seeks to minimize a cost function that accounts for wirelength (to shorten interconnect delays and reduce routing congestion) and possibly timing estimates. The cells are spread across the core such that no area is overly congested. At the end of global placement, each cell has an approximate coordinate, but overlaps between cells are allowed (they haven’t been snapped to the discrete site grid yet). In essence, global placement finds an optimized relative positioning of cells without regard to legal spacing rules.

After that, detailed placement is run (using OpenDP in OpenROAD) [53]. Detailed placement takes the outcome of global placement and adjusts cells to exact legal positions on the placement grid (the “sites” in each row) and removes all overlaps. It makes local refinements, possibly swapping or shifting cells, to improve wirelength or timing locally while obeying design rules. The result is a legally placed design: every standard cell sits in a row, aligned to the uniform site grid, with no overlaps and correct spacing between cells. Power pins of each cell align under the pre-laid power rails. Because standard cells in the Sky130 library are designed to abut each other seamlessly (sharing well spacing and power rails) [53], the placed row will be a continuous chain of cells and occasional fillers/taps butting end-to-end.

OpenLane may invoke an optimization step between global and detailed placement as well – for instance, using Resizer/OpenPhySyn to size or swap cells for better timing, or to insert buffers on long nets to fix slew or timing issues [53]. These optimizations can slightly modify the netlist (e.g., buffer insertion), and when they do, OpenLane runs a quick logic equivalence check (LEC) using Yosys to ensure the netlist's logical behavior is unchanged by the optimizations [53]. By the end of placement, the netlist (with any added buffers) is considered final for routing.

The output of the placement stage is an updated DEF file with all standard cell coordinates specified. In addition, OpenLane produces a placed netlist (Verilog) that reflects any changes (like added buffers or inverter pairs for hold fixing). Placement logs and reports will indicate details like total wirelength, any density hotspots, and timing estimates at this stage. A well-executed placement will minimize late surprises in routing by preventing over-congestion in any area of the chip.

4.2.1.4 Clock Tree Synthesis (CTS)

After placement, the design is ready for clock tree synthesis. The clock signal (e.g., the main system clock of the RISC-V core) has to be distributed to all sequential elements (flip-flops, latches) with minimal skew and sufficient drive strength. In the placed netlist, the clock is typically a single net feeding hundreds or thousands of flip-flops – this cannot be routed as a simple wire from one source. CTS addresses this by inserting a clock tree network of buffers and/or inverters that fan-out the clock.

OpenLane uses TritonCTS (an OpenROAD component) to build the clock tree [53]. The algorithm takes the placed design and the clock pin locations of all registers, then hierarchically inserts buffers to create a balanced tree. It starts from the clock source (either an input pad or an internal PLL source) and adds buffers such that the clock load is split and distributed. The

aim is to have the clock arrival times at all leaf nodes (flops) be as equal as possible (low skew) and to minimize the overall insertion delay of the clock network.

TritonCTS will choose buffering points and specific buffer cell types from the standard cell library (Sky130 provides dedicated clock buffer cells like sky130_fd_sc_hd__clkbuf_* which are designed for low delay and low output skew). The user can guide CTS by specifying which buffer cells to use and target skew. The result of CTS is that many new buffer cells are added into the design (these are placed in available gaps in the row nearest to where needed, respecting placement rules). The DEF and netlist are updated with these inserted clock buffers.

OpenLane then often runs post-CTS optimizations. Since adding the clock tree can change timing (especially hold times, because clock insertion can introduce skew intentionally to fix holds or due to process), a common step is to adjust timing on registers by inserting small delay buffers (if needed) to fix any hold time violations that arise post-CTS. The design now has a buffered clock net (or nets, if multiple clocks) connecting from the source through levels of buffers to all flops. A post-CTS static timing analysis is usually performed (OpenSTA can run at this stage) to report the achieved clock skew and any timing violations that need addressing [53].

It’s worth noting that after CTS, OpenLane again can perform a logic equivalence check (LEC) because buffers were added (ensuring no functional change in data paths) [53]and it generates new timing reports. The presence of a well-balanced clock tree is crucial for the CPU to operate at its target frequency reliably.

4.2.1.5 Routing (Global and Detailed Routing)

With cells placed and the clock tree inserted, the next step is routing, which means physically connecting all the nets (signal wires) in the design with metal interconnect on the available metal layers of the Sky130 process. SkyWater130A has 6 routing layers (5 metals plus local interconnect), of which typically Metal1 through Metal5 are used for signal routing (Metal6 often for top-level power grids, etc.). The routing task is split into global and detailed routing phases in OpenLane [54].

Global Routing: OpenLane invokes FastRoute (an open-source global router) to perform global routing [54]. In global routing, the chip is conceptually divided into a grid of rectangular regions, and the router plans an approximate route for each net through these regions. It does not fix exact tracks or coordinates but rather computes an approximate path (like which general direction and which region each connection will go through). The goal is to minimize total wirelength and avoid overloading any region with too many routes (which would cause congestion). FastRoute produces a routing guide file that indicates, for each net, which metal layers and grid regions the net should pass through. This guide is used by the detailed router as a blueprint.

Detailed Routing: The actual wiring is then completed by TritonRoute, the detailed routing tool [54]. TritonRoute takes each connection and assigns specific tracks and via locations to implement the wiring, obeying the design rules of the process. It uses the guides from global routing to confine its search space, which helps ensure that if the global route was congestion-free, the detailed router can finalize all wires without conflict. During detailed routing, TritonRoute must handle complex design rules from the Sky130 PDK (spacing rules, via enclosures, antenna rules, etc.). It may introduce small jogs or multi-layer routing segments for each net. By the end, every net in the design has a physical wire connecting its pins, and no two nets violate spacing or cross-talk rules. The output is an updated DEF with full routing

details for all metals, plus a SPEF (Standard Parasitic Exchange Format) file capturing the parasitic RC values of the routed interconnect.

It's common for the detailed router to also insert antenna diodes or diffusions on long wires to prevent antenna effects (charge accumulation on metal during fabrication). OpenLane has a step to swap in antenna diode cells if needed after routing if the foundry's antenna rules are violated [54]. In Sky130, antenna violations are typically fixed by inserting special diode cells from the standard cell library near the offending net's receiver gate.

OpenLane ensures that the final routed netlist is still functionally equivalent to the pre-routed netlist (again via LEC if any changes like diode insertion occurred) [54]. At this stage, the design is considered fully laid out: all components placed and all connections made. A final LVS (layout vs schematic) check at this point (comparing the routed DEF/netlist to the original netlist) would confirm that no connectivity errors were introduced.

After routing, parasitic extraction is performed on the layout. OpenLane uses OpenRCX to extract RC parasitics from the routed design, generating the SPEF file [54]. These parasitics quantify the capacitance and resistance on each net, which can significantly affect timing. Using the SPEF, a detailed timing sign-off analysis with OpenSTA is run to see if the design still meets the timing constraints when all wire delays are accounted for [54]. If violations are found, engineering change orders (ECOs) may be needed (for instance, resizing a gate or adding a buffer in a critical path and re-routing locally) to fix timing. OpenLane provides mechanisms for ECO loops, but assuming timing is clean, the flow proceeds to final sign-off.

The key outputs after routing are the fully routed DEF file, the SPEF parasitic data, and updated design files (Verilog netlist annotated with any changes). These are used in the final sign-off checks. Logs and reports will include routing statistics (wirelength, number of vias, any DRC violations encountered) and final timing reports. At this point, one can also generate a

visualization of the layout to inspect it. For instance, using KLayout or Magic to view the DEF or the GDSII (to be created next) allows verification that the CPU's floorplan and routing look correct.

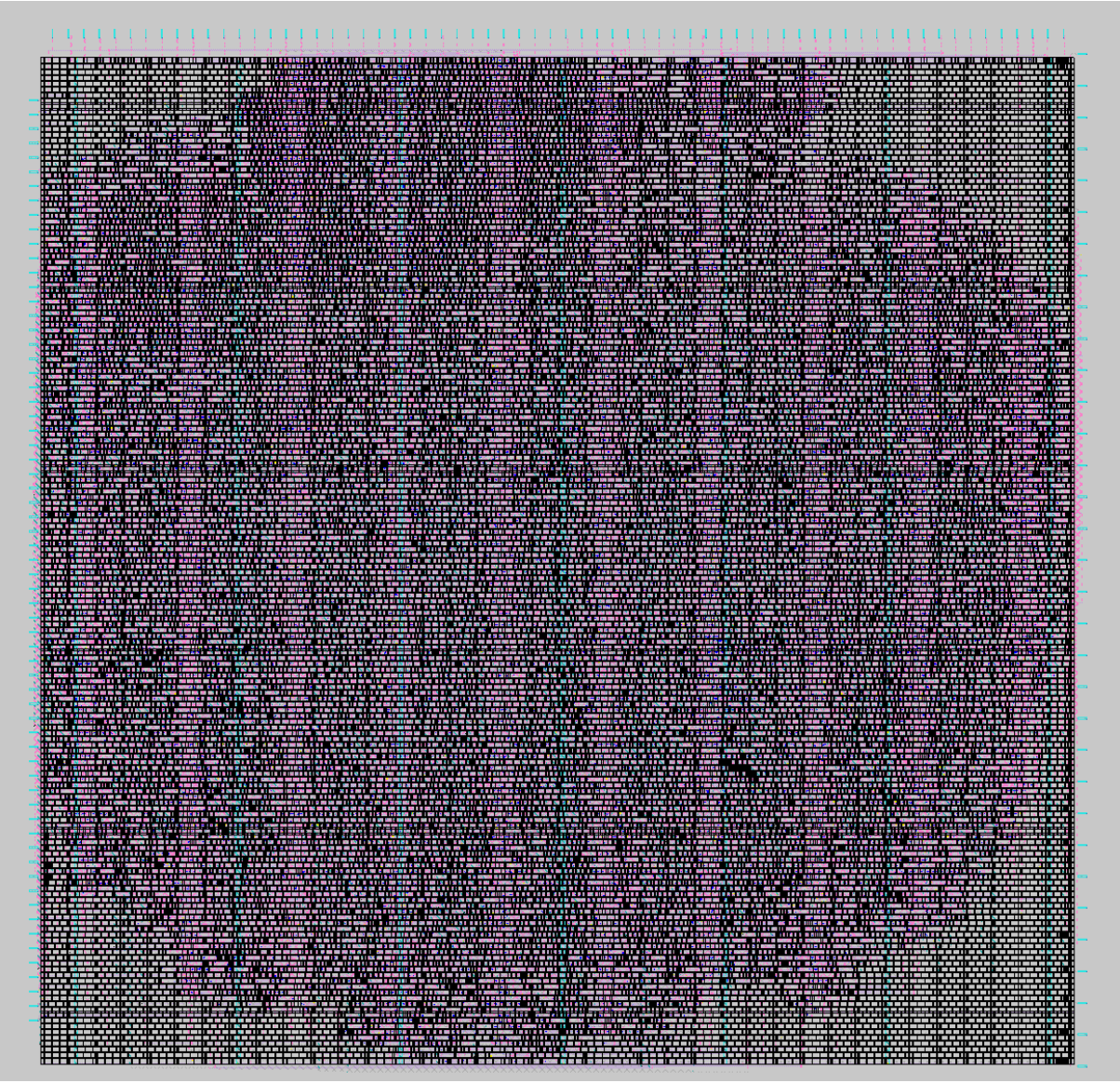


Figure 4.10. Final Routed Design of CPU

4.2.2 GDSII Generation and Physical Verification (Sign-off)

The last stage of the flow is to perform sign-off verifications and export the final layout in the industry-standard GDSII format. In OpenLane, once routing is complete and timing is verified, the flow enters the “Tapeout” and “Signoff” stages [55]:

- **GDSII Streaming Out:** OpenLane uses the Magic VLSI tool to stream out the design to GDSII format [55]. Magic takes the routed DEF (along with the GDS/OASIS layouts of standard cells and macros from the PDK) and merges them to create a full geometric layout. Essentially, Magic knows the shapes of each standard cell (via the cell GDS or CIF) and places those shapes at the coordinates specified in the DEF, then adds all the wiring shapes from the routing. The output is a GDSII file that contains every layer geometry for the entire chip – this is the file sent to the foundry. OpenLane often uses KLayout as a backup to stream out GDS as well [55], ensuring redundancy.
- **Design Rule Check (DRC):** Magic is also used to perform DRC on the final layout [56]. The Sky130 PDK comes with a set of design rules (for distances, widths, spacings, enclosure, etc.) and Magic has a tech file to check them. OpenLane runs Magic’s DRC engine to produce a report of any rule violations. A clean design should have zero DRC errors. If any are present (e.g., minor spacing issues or antenna violations), the flow may attempt automatic fixes or require manual intervention to adjust the layout. In an automated flow, common DRC fixes include adding filler cells or slight layout adjustments. OpenLane’s goal is a DRC-clean GDS, which is crucial because foundries will not fabricate a design with DRC errors.
- **Layout vs. Schematic (LVS):** Using the Netgen tool, OpenLane performs LVS to ensure the final layout’s connectivity matches the original schematic (netlist) [56]. Netgen takes the SPICE extracted netlist from the layout (extracted by Magic or a dedicated extractor) and compares it to the post-synthesis netlist (or a post-routing

netlist). Any mismatches (missing connections, incorrect routing of nets) are flagged. Passing LVS means the layout is functionally identical to the intended design netlist. In practice, if the flow has been correct up to this point, LVS should pass because we haven't intentionally changed connectivity except inserting known equivalent cells (buffers, diodes which are accounted for). OpenLANE's integration of LEC checks earlier in the flow also helps ensure functional equivalence [56]. Nonetheless, LVS is the definitive proof that, for example, every gate output in the schematic connects to the same gates in the layout.

- **Static Timing Analysis (STA):** Although STA is run at intermediate steps, a final sign-off STA is performed with all parasitics to ensure no timing violations remain. OpenSTA (the open-source Static Timing Analyzer) uses the extracted SPEF, the final netlist, and the timing constraints to compute worst-case delays and setup/hold margins. The final timing report will indicate the achieved clock frequency and any remaining slack on critical paths. Ideally, all paths meet the target (e.g., the RISC-V CPU can run at its intended MHz speed in typical conditions). If there are violations, designers might need to backtrack (change floorplan, add pipeline stages, etc.), but assuming all is well, STA is signed off.
- **Power Analysis:** The open-source flow uses tools like IRSIM or custom scripts for a basic IR drop analysis on the power grid, and verify that decoupling is sufficient. However, OpenLane's primary sign-off focuses on geometry (DRC/LVS) and timing.

When all sign-off checks are clean (DRC = 0 errors, LVS = 0 mismatches, timing met), the design is considered tape-out ready. The final GDSII, along with a final netlist and all the documentation (timing reports, area reports, etc.), is produced in OpenLane's results directory.

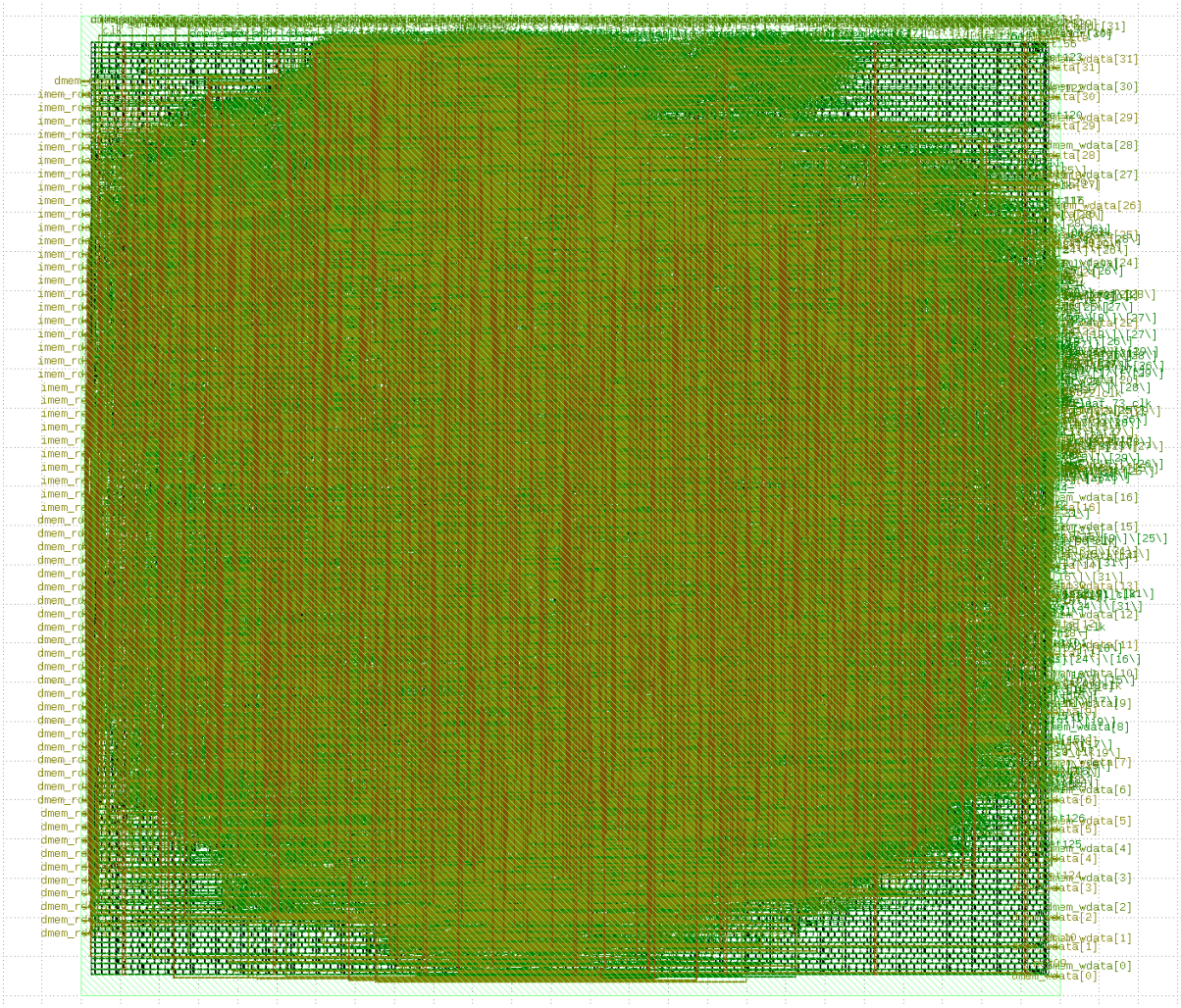


Figure 4.11. Final GDS view of CPU (KLayout)

Chapter 5 : Results and Verification

A critical question in developing any new EDA flow is: **How accurate are the results compared to trusted standards?** In this work, since the characterization flow is intended as an open-source substitute for commercial tools, it is needed to validate that it produces accurate timing and power data. This is addressed by directly comparing the characterization results of the proposed flow against a commercial reference. Specifically, after characterizing the custom cells with our open-source flow, they were re-characterized the same cells using Cadence Liberate – a widely-used commercial library characterization tool – using identical conditions and device models. Then a comparison was made between the key metrics from the two sets of characterization data.

5.1 Liberty file verification

In this work, for the construction of the standard cell/gate liberty (.lib) file, the following parameters are obtained after SPICE simulation of the gate layout SPICE files.

1. Rise Transition
2. Fall Transition
3. Rise Propagation Time
4. Fall Propagation Time
5. Rise Time Power
6. Fall Time Power

Each of the cells used in the CPU design are simulated and the corresponding values are obtained.

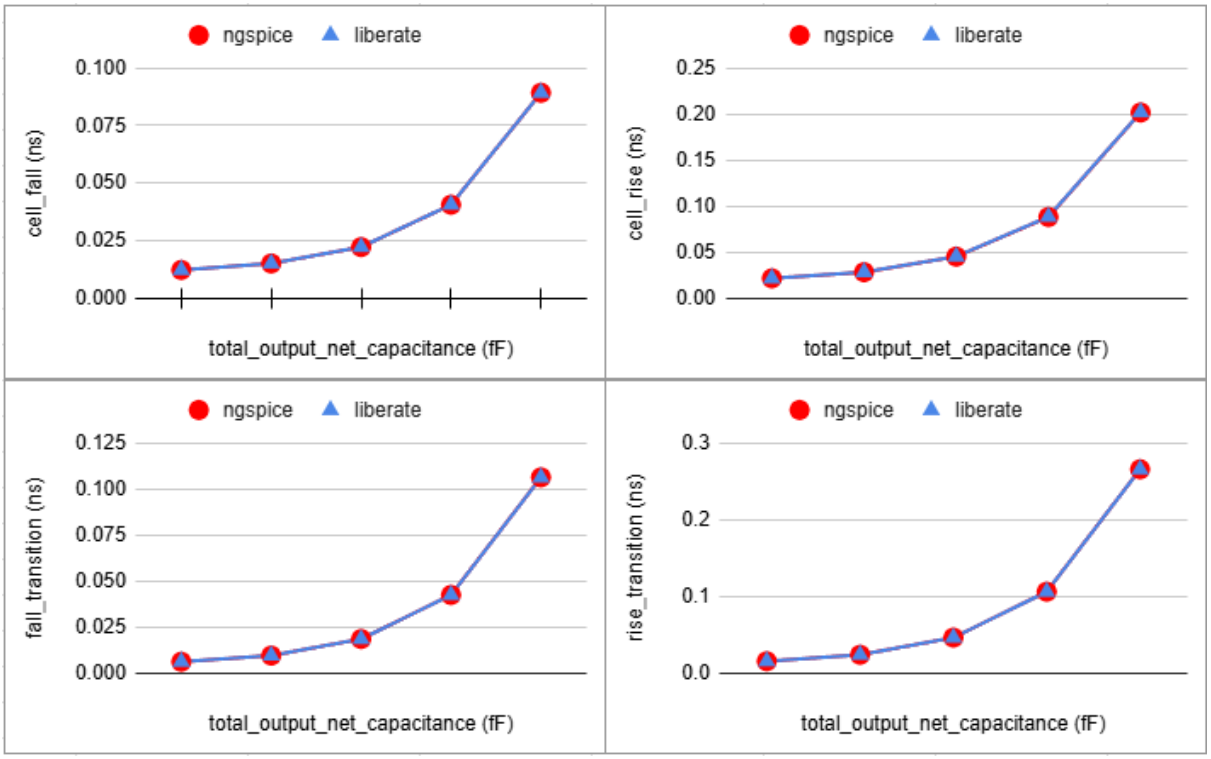
5.1.1 Inverter

Table 5.1 shows the average variation of the comparison of characterization results for two inverter sizes (INVX1, INVX2) between a commercial tool (Liberate) and the proposed thesis characterization flow across three different PVT corners: typical-typical (tt_25C_1.8V), fast-fast (ff_n40C_1.95V), and slow-slow (ss_100C_1.60V). The presented data highlights the percentage variation in key timing parameters (cell fall, fall transition, cell rise, rise transition) and power parameters (rise power, fall power), illustrating the accuracy and reliability of the proposed characterization method relative to industry-standard tool like Liberate.

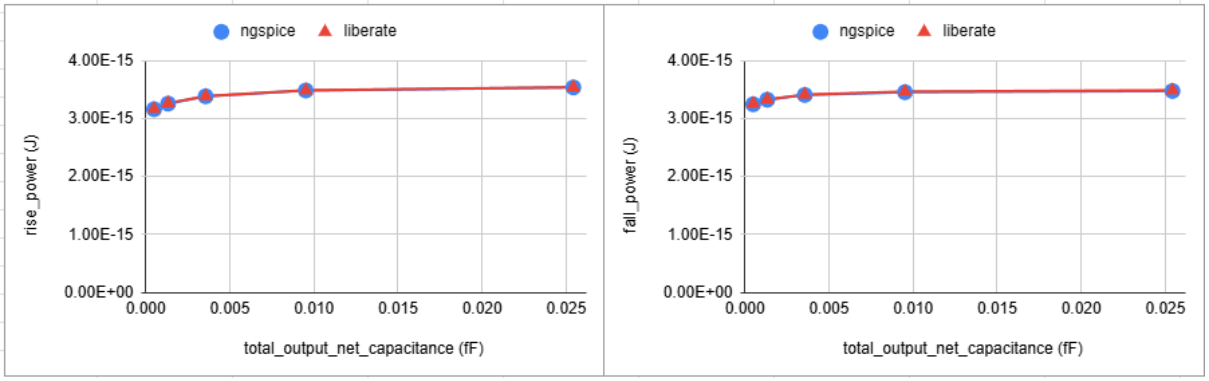
tt_25C_1.8V						
Cell/Gate Name	**Timing Parameters**				**Power Parameters**	
	cell_fall	**fall_transition**	**cell_rise**	**rise_transition**	**rise_power**	**fall_power**
INVX1	0.095 %	0.224 %	0.242 %	0.041 %	0.130 %	0.265 %
INVX2	0.103 %	0.322 %	0.105 %	0.050 %	0.133 %	0.294 %
ff_n40C_1.95V						
Cell/Gate Name	**Timing Parameters**				**Power Parameters**	
	cell_fall	**fall_transition**	**cell_rise**	**rise_transition**	**rise_power**	**fall_power**
INVX1	0.155 %	0.186 %	0.250 %	0.054 %	0.175 %	0.238 %
INVX2	0.120 %	0.717 %	0.158 %	0.077 %	0.108 %	0.419 %
ss_100C_1.60V						
Cell/Gate Name	**Timing Parameters**				**Power Parameters**	
	cell_fall	**fall_transition**	**cell_rise**	**rise_transition**	**rise_power**	**fall_power**
INVX1	0.075 %	0.072 %	0.239 %	0.037 %	0.149 %	0.272 %
INVX2	0.149 %	0.158 %	0.076 %	0.036 %	0.163 %	0.409 %

Table 5.1. Average variation percentage for Inverter

Fig 5.1 represents the graphical visualization of the same data showing that the values completely match each other.



(i)



(ii)

Figure 5.1. Variation plot for Inverter of unit size for (i)Timing Parameters (ii)Power Parameters

5.1.2 Buffer

Table 5.2 shows the average variation of the comparison of characterization data for two buffer cell sizes (BUFX1, BUFX2) between the commercial characterization tool Liberate and the methodology proposed in this thesis. The data reflect percentage variations in timing parameters (cell fall, fall transition, cell rise, rise transition) and power parameters (rise power, fall power) across typical-typical (tt_25C_1.8V), fast-fast (ff_n40C_1.95V), and slow-slow (ss_100C_1.60V) process-voltage-temperature (PVT) conditions. The results emphasize the consistency and accuracy of the proposed characterization flow relative to commercially established tools.

tt_25C_1.8V						
Cell/Gate Name	**Timing Parameters**				**Power Parameters**	
	cell_fall	**fall_transition**	**cell_rise**	**rise_transition**	**rise_power**	**fall_power**
BUFX1	0.144 %	0.322 %	0.050 %	0.075 %	0.101 %	0.515 %
BUFX2	0.064 %	0.295 %	0.053 %	0.069 %	0.258 %	0.340 %
ff_n40C_1.95V						
Cell/Gate Name	**Timing Parameters**				**Power Parameters**	
	cell_fall	**fall_transition**	**cell_rise**	**rise_transition**	**rise_power**	**fall_power**
BUFX1	0.242 %	0.301 %	0.040 %	0.035 %	0.359 %	0.557 %
BUFX2	0.131 %	0.261 %	0.120 %	0.097 %	0.131 %	0.519 %
ss_100C_1.60V						
Cell/Gate Name	**Timing Parameters**				**Power Parameters**	
	cell_fall	**fall_transition**	**cell_rise**	**rise_transition**	**rise_power**	**fall_power**
BUFX1	0.061 %	0.232 %	0.037 %	0.069 %	0.196 %	0.370 %
BUFX2	0.123 %	0.206 %	0.062 %	0.055 %	0.180 %	0.373 %

Table 5.2. Average variation percentage for Buffer

5.1.3 NAND

Table 5.3 shows the average variation of the comparison of characterization data for two buffer cell sizes (NANDX1, NANDX2) between the commercial characterization tool Liberate and the methodology proposed in this thesis. For two input pins, the simulations were done twice.

tt_25C_1.8V							
		Timing Parameters				**Power Parameters**	
Cell/Gate Name	**Pin**	**cell_fall**	**fall_transition**	**cell_rise**	**rise_transition**	**rise_power**	**fall_power**
NANDX1	A	0.156 %	0.057 %	0.187 %	0.040 %	0.222 %	0.261 %
	B	0.079 %	0.072 %	0.182 %	0.061 %	0.343 %	0.103 %
NANDX2	A	0.243 %	0.039 %	0.109 %	0.038 %	0.147 %	0.361 %
	B	0.137 %	0.061 %	0.139 %	0.082 %	0.602 %	0.178 %
ff_n40C_1.95V							
		Timing Parameters				**Power Parameters**	
Cell/Gate Name	**Pin**	**cell_fall**	**fall_transition**	**cell_rise**	**rise_transition**	**rise_power**	**fall_power**
NANDX1	A	0.110 %	0.089 %	0.215 %	0.033 %	0.195 %	0.259 %
	B	0.084 %	0.077 %	0.206 %	0.065 %	0.264 %	0.143 %
NANDX2	A	0.205 %	0.083 %	0.152 %	0.029 %	0.170 %	0.310 %
	B	0.134 %	0.118 %	0.114 %	0.098 %	0.547 %	0.149 %
ss_100C_1.60V							
		Timing Parameters				**Power Parameters**	
Cell/Gate Name	**Pin**	**cell_fall**	**fall_transition**	**cell_rise**	**rise_transition**	**rise_power**	**fall_power**
NANDX1	A	0.151 %	0.057 %	0.103 %	0.052 %	0.183 %	0.226 %
	B	0.090 %	0.061 %	0.110 %	0.078 %	0.306 %	0.176 %
NANDX2	A	0.280 %	0.034 %	0.062 %	0.037 %	0.128 %	0.311 %
	B	0.169 %	0.031 %	0.111 %	0.109 %	0.648 %	0.155 %

Table 5.3. Average variation percentage for NAND

5.1.4 NOR

Table 5.3 shows the average variation of the comparison of characterization data for two buffer cell sizes (NORX1, NORX2) between the commercial characterization tool Liberate and the methodology proposed in this thesis. For two input pins, the simulations were done twice.

tt_25C_1.8V							
		Timing Parameters				**Power Parameters**	
Cell/Gate Name	**Pin**	**cell_fall**	**fall_transition**	**cell_rise**	**rise_transition**	**rise_power**	**fall_power**
NORX1	A	0.019 %	0.334 %	0.019 %	0.045 %	0.052 %	1.417 %
	B	0.201 %	0.062 %	0.096 %	0.047 %	0.043 %	0.099 %
NORX2	A	0.147 %	0.498 %	0.022 %	0.034 %	0.035 %	2.008 %
	B	0.148%	0.201 %	0.040 %	0.044 %	0.063 %	0.225 %
ff_n40C_1.95V							
		Timing Parameters				**Power Parameters**	
Cell/Gate Name	**Pin**	**cell_fall**	**fall_transition**	**cell_rise**	**rise_transition**	**rise_power**	**fall_power**
NORX1	A	0.063 %	0.173 %	0.017 %	0.050 %	0.058 %	1.383 %
	B	0.221 %	0.176 %	0.125 %	0.051 %	0.106 %	0.077 %
NORX2	A	0.098 %	0.294 %	0.014 %	0.035 %	0.033 %	1.886 %
	B	0.169 %	0.222 %	0.118 %	0.036 %	0.037 %	0.159 %
ss_100C_1.60V							
		Timing Parameters				**Power Parameters**	
Cell/Gate Name	**Pin**	**cell_fall**	**fall_transition**	**cell_rise**	**rise_transition**	**rise_power**	**fall_power**
NORX1	A	0.025 %	0.434 %	0.019 %	0.042 %	0.072 %	1.113 %
	B	0.265 %	0.058 %	0.098 %	0.051 %	0.096 %	0.205 %
NORX2	A	0.123 %	0.713 %	0.033 %	0.032 %	0.061 %	3.551 %
	B	0.177 %	0.096 %	0.075 %	0.038 %	0.090 %	0.168 %

Table 5.4. Average variation percentage for NOR

The comparison showed a **very close match** between our open-source flow's results and the commercial tool's results. For example, the propagation delays and output transition times for a representative logic gate were within a few percent of those obtained by Liberate, across all measured load and input slew conditions. Dynamic power measurements (charge/discharge energies per transition) were also closely aligned between the two, and leakage currents differed only by negligible amounts. These small differences can be attributed to minor simulation setting differences (e.g., convergence parameters in SPICE) or interpolation nuances, but importantly, no systematic bias or large error was observed.

This validation is a key novel contribution of this work. Prior open-source efforts like Libretto had introduced the concept of open characterization, but published results did not yet show head-to-head comparisons against industry tools.

5.2 RISC V Simulation

Another aspect of validation involved ensuring that the characterized data functioned correctly in a design flow. The generated library is integrated into an open-source flow (OpenLane with OpenSTA) for the RISC-V CPU design.

Chapter 4 describes the overall architecture and design of the CPU. Following that, this section dives into the functionality of the CPU and its parameters.

5.2.1 CPU Operation

The CPU goes through the following stages during its operation

1. Reset & Initialization
2. Release Reset & First Fetch
3. Fetch instructions and Decode
4. Single-Cycle Execution
5. PC Update

The CPU takes a set of hexadecimal instructions (assembly language) as input to perform operations. This is a replication of an actual CPU function where a program execution occurs.

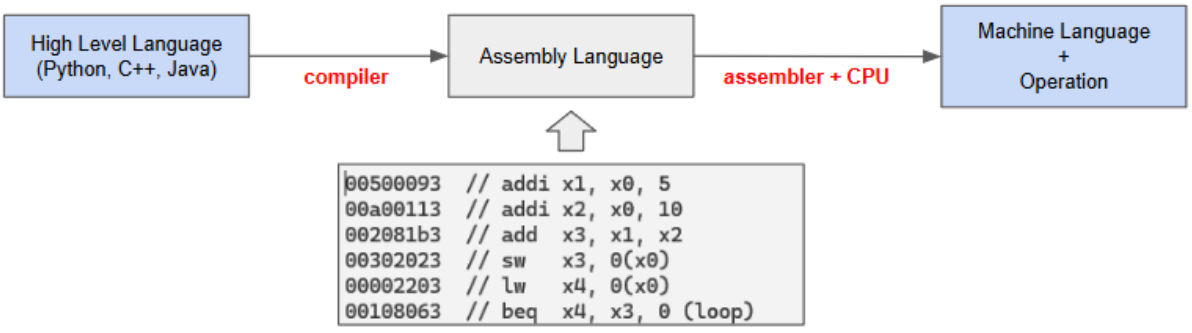
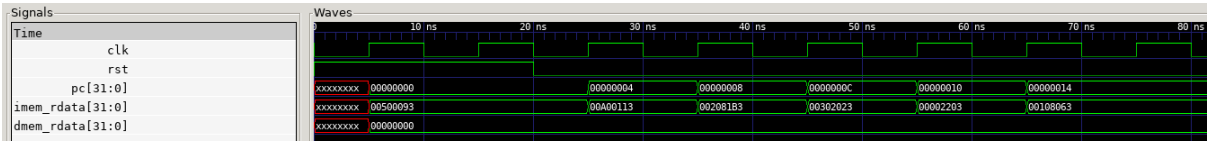
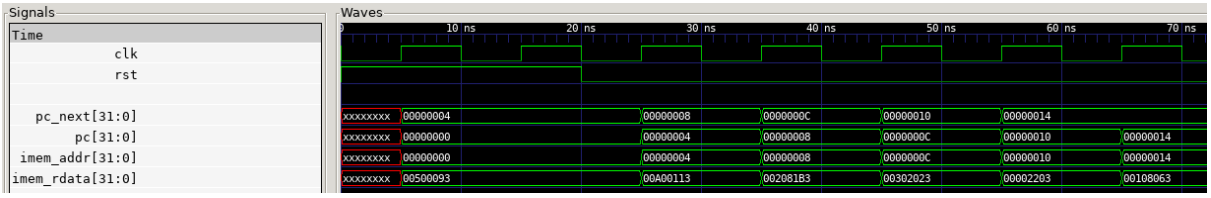


Figure 5.2. Instruction inputs to CPU

1. **Reset & Initialization:**
    - At time 0 ns, **rst = 1**, so the CPU stays in reset
    - On each rising clock (every 10 ns), because **rst** is high, the CPU's **pc** is synchronously forced to 0
    - Meanwhile the testbench has loaded the **imem** array from **imem_tb.hex** and zeroed out dmem
    - At **t = 20ns, rst = 0** so the PC begins count



2. **Release Reset & First Fetch:**
    - After 20 ns, the testbench sets **rst = 0**
    - On the next rising edge (**at 25 ns**), the CPU updates **pc <= pc_next**
    - In reset, **pc_next** was computed as **pc + 4**, so **pc** becomes 4
    - Meanwhile the CPU's fetch logic drives **imem_addr = pc**



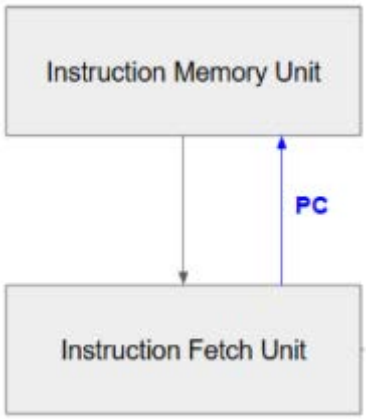


Figure 5.3. Fetching of Instruction using PC

3. Fetch instructions and Decode:

Fetch: CPU presents **imem_addr = pc**

- Meanwhile, testbench provides **instr = imem[imem_addr>>2]**.
- This adds 1 byte to the address to make the space ready for next instruction

Decode: Instruction bits split into the following fields

Field Name	No of bits	Use
funct7	7 bits	Function code that, together with funct3, specifies the exact operation
rs2	5 bits	Address (or index) of the second source register
rs1	5 bits	Address (or index) of the first source register
funct3	3 bits	Function code that, together with funct7, specifies the exact operation
rd	5 bits	Address (or index) of the destination register where the result is written
opcode	7 bits	Specifies the type of operation (e.g., arithmetic, logical)

```
// Decode instruction fields
wire [6:0]  opcode = instr[6:0];
wire [4:0]  rd     = instr[11:7];
wire [2:0]  funct3 = instr[14:12];
wire [4:0]  rs1    = instr[19:15];
wire [4:0]  rs2    = instr[24:20];
wire [6:0]  funct7 = instr[31:25];
```

4. **Single-Cycle Execution:**

Control: Based on opcode, following control signals are generated

- **branch = 1** for BEQ/BNE so we know to compare and maybe jump
- **memread = 1** for LOAD instructions
- **memwrite = 1** for STORE instructions
- **alusrc = 1** if the ALU's second input should come from the immediate instead of register 2
- **regwrite = 1** if we will write back to rd
- **memtoReg = 1** if the data to write back comes from memory rather than the ALU
- aluop a two-bit hint that we pass along to further pick the specific ALU operation

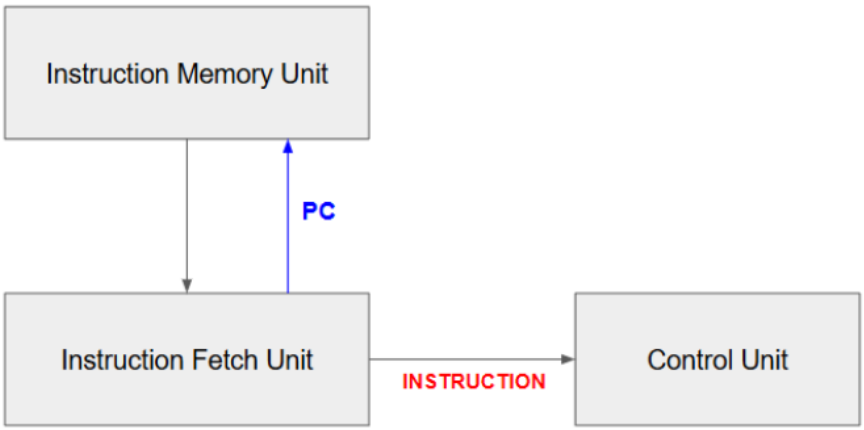


Figure 5.4. Single Cycle execution Map

Register Read

- CPU reads **regs[rs1]** and **regs[rs2]** from the register file.

ALU

- Performs the operation (add, sub, logic, shift, compare) on **reg_data1** and either **reg_data2** or the immediate

Produces **alu_result**

5. **PC Update**

The CPU computes the next address for the new instruction

pc_next = (branch && zero) ? pc + imm : pc + 4;

- and on the next rising clock, loads **pc <= pc_next**

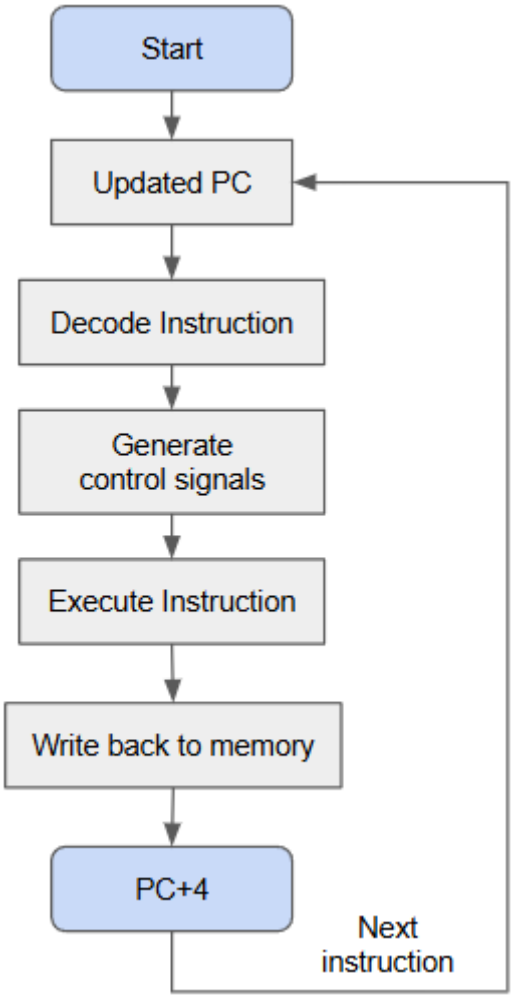


Figure 5.5. Program Counter Update Operation

5.2.2 Assembly Code Simulation

The provided assembly instructions generate a sequence of outputs which utilizes the components of the CPU

```
00500093  // addi x1, x0, 5
00a00113  // addi x2, x0, 10
002081b3  // add  x3, x1, x2
00302023  // sw   x3, 0(x0)
00002203  // lw   x4, 0(x0)
00108063  // beq  x4, x3, 0 (loop)
```

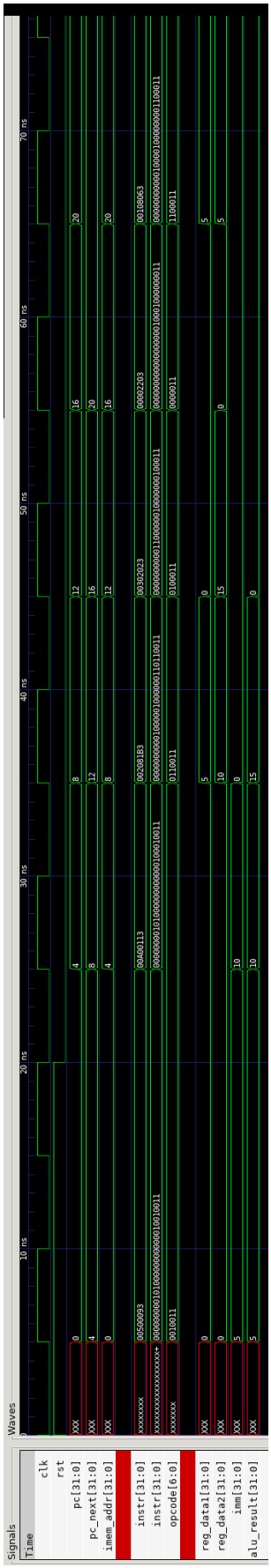
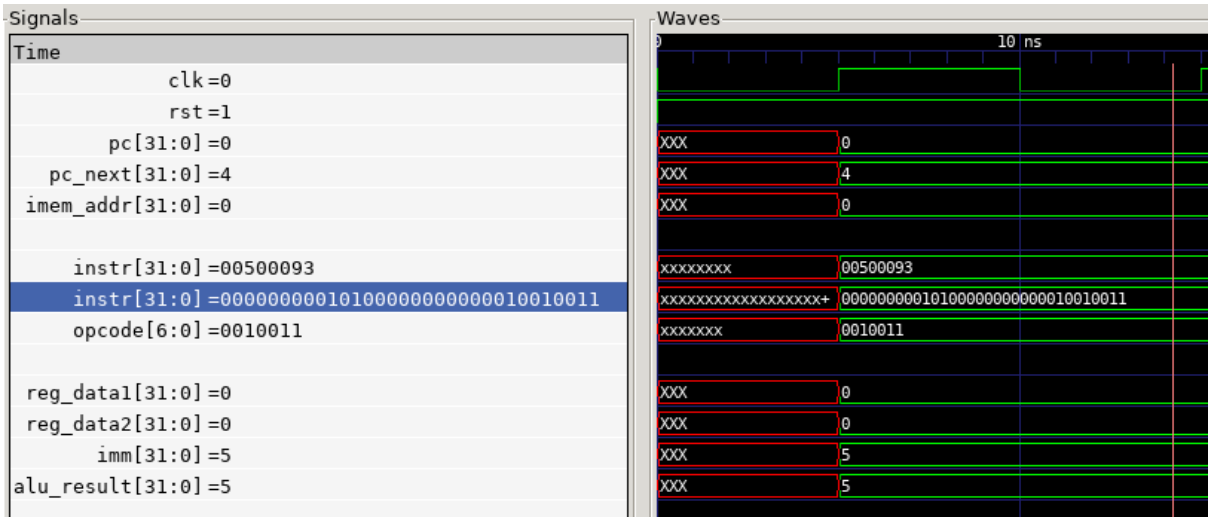


Figure 5.6. TestBench Simulation



Here, BIN instruction = 0000000 0101 00000 000 00001 0010011

Field	Bits	Value	Meaning
opcode	[6:0]	0010011	Immediate opcode
rd	[11:7]	00001	Destination reg (x1)
funct3	[14:12]	000	ADD Immediate
rs1	[19:15]	00000	Source reg (x0)
imm[11:0]	[31:20]	000000000101 (5)	Immediate = +5

Here, Assembly Code

addi x1, x0, 5

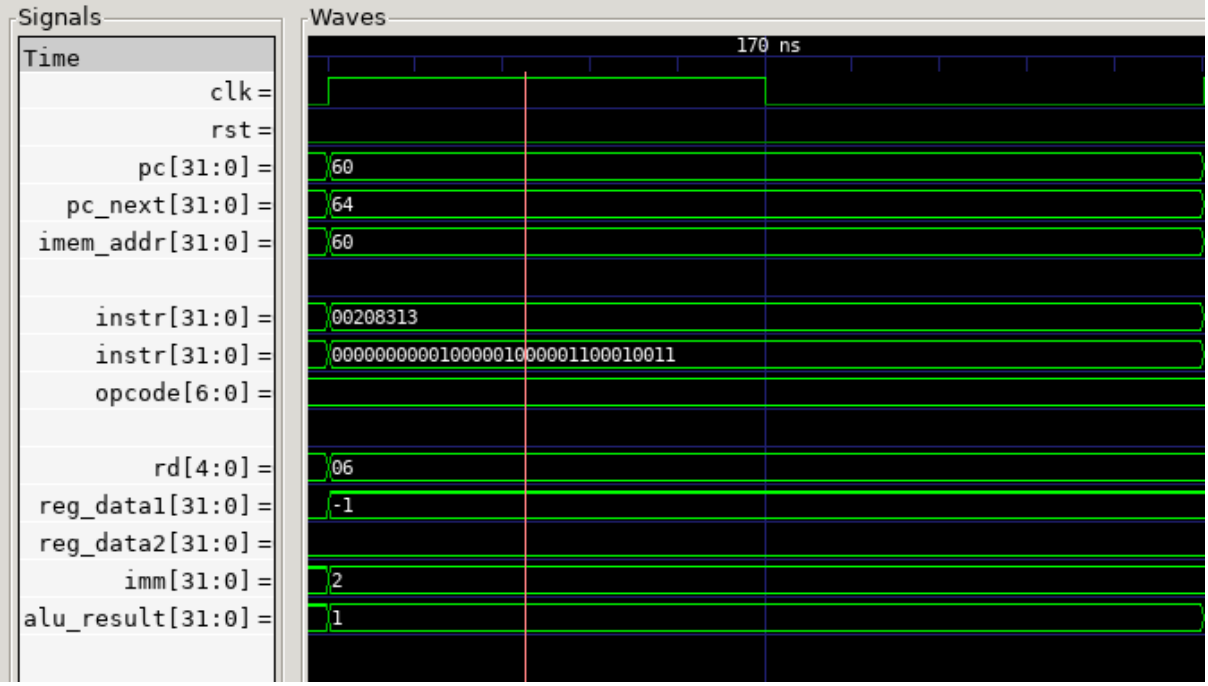
Control Signals

- alusrc=1 (use immediate)
- regwrite=1
- memread=0
- memwrite=0
- branch=0

Data Path

- Reg Read: rs1→x0 = 0
- Imm Gen: imm→5
- ALU: a = 0, b = 5 →
  - alu_result = 5
- Mem: no memory access
- Write-Back: regs[1] ← 5

5.2.3 Standalone assembly code simulation example



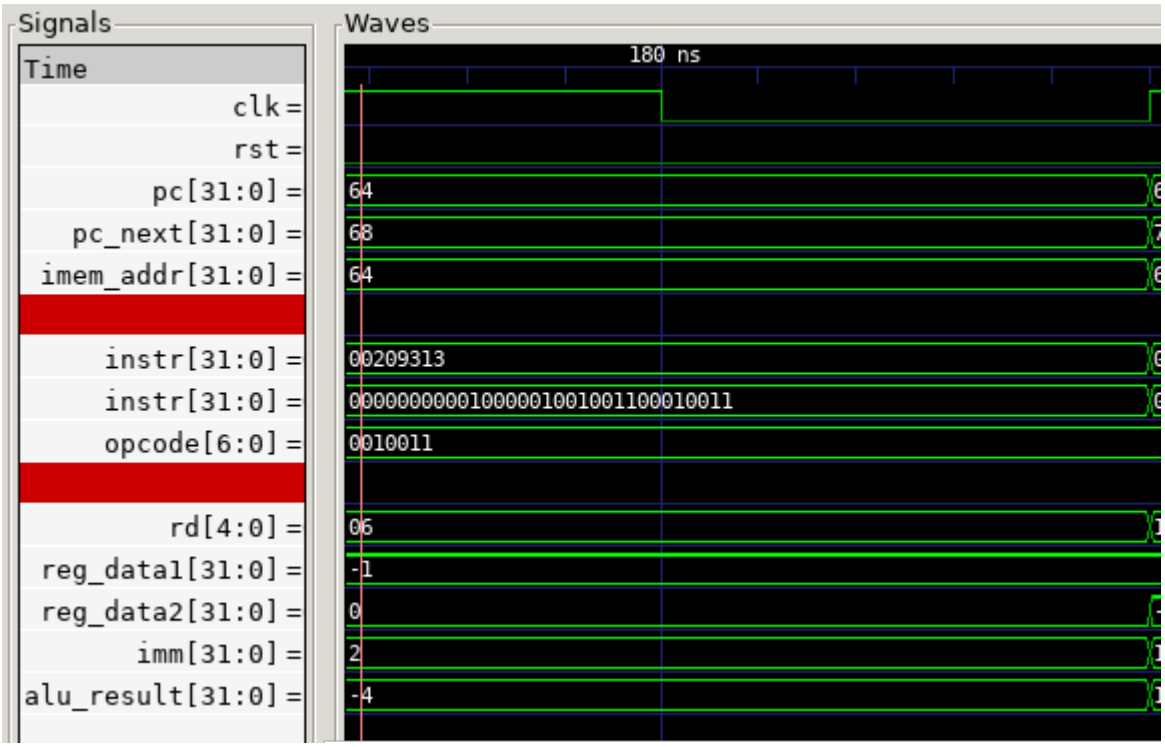
- Assembly Code: **addi x06, x1, 2**
- Binary: 000000000010 00001 000 00110 0010011
- Splitting into separate fields to decode
    - imm[11:0] = 000000000010 (dec value = +2)
    - rs1 = 00001 (x1)
    - funct3 = 000 (ADDI)
    - rd = 00110 (x06)
    - opcode = 0010011 (IMM)
- According to the opcode, this performs **add immediate**

alu_result = reg_data1 + imm

Here, reg_data1 = -1

imm = 2

So, alu_result = 1



- Assembly Code: **slli x06, x1, 2**
- Binary: 000000000010 00001 001 00110 0010011
- Splitting into separate fields to decode
    - imm[11:0] = 000000000010 (dec value = +2)
    - rs1 = 00001 (x1)
    - funct3 = 001 (SLLI)
    - rd = 00110 (x06)
    - opcode = 0010011 (IMM)
- According to the opcode, this performs **shift-left logical immediate**

alu_result = reg_data1 << 2

Here, reg_data1 = -1

(BIN = 11111111111111111111111111111111)

imm = 2

So, alu_result = 11111111111111111111111111111100

= -4

5.3 CPU Design Data

Table 5.5 represents the CPU design parameters obtained from the post Place and Route database

Parameters	Values
CoreArea (um^2)	232535.52
Floorplan Core Utilization	45
Clock Period (ns)	100
Input ports	1121
Outputs ports	1122
Internal Power drop (uW)	0.00601
Switching Power (uW)	0.00426
Leakage Power (uW)	5.56E-08

Table 5.5. Post PnR output parameters

In all cases, the design closure process succeeded using our library, and the resulting CPU met the target frequency without violations when checked with the sign-off tools. This end-to-end test proved not only the numerical accuracy of the characterization, but also the compatibility and completeness of the library data format.

In summary, by cross-verifying with a commercial characterization suite and using the characterized cells in a full chip implementation, this work has shown that the developed open-source flow is practically viable. The close matching values confirm that this approach can serve as a dependable alternative to proprietary tools for standard cell characterization.

5.4 Power optimization of the design

For an optimized design of the CPU using SKY130 PDK, variations were made in the device parameters of the gates used in the design. The available parameters which can be varied in the library are

1. Drain Diffusion area
2. Source Diffusion area
3. Drain Perimeter
4. Source Perimeter

All are used internally by the simulator for accurate parasitic extraction. In this work, the drain and source diffusion areas are varied to achieve better power results. The drain diffusion area (ad) and source diffusion area (as) are physical parameters in a MOSFET (like those in Sky130) that describe the size of the doped regions connected to the drain and source terminals. These areas are crucial for modeling parasitic capacitances and resistances in SPICE simulations.

To demonstrate the effect of the variation, a buffer cell is taken as reference. A buffer cell consists of 4 FETs (2 pFETs and 2 nFETs). For this work, multiple combinations of the parameters are considered, and the power readings are obtained in post-layout power simulations.

Table 5.5 shows the different combinations considered for the design and Table 5.6 shows the power readings of each of the combinations.

No.	Buffer FET combinations	Drain diffusion area (ad)	Source diffusion area (as)
1	sky130_fd_pr__nfet_01v8	0	0
	sky130_fd_pr__pfet_01v8_hvt	0	0
	sky130_fd_pr__pfet_01v8_hvt	0	0
	sky130_fd_pr__nfet_01v8	0	0
2	sky130_fd_pr__nfet_01v8	0.0126	0.0225
	sky130_fd_pr__pfet_01v8_hvt	0.0191	0.0342
	sky130_fd_pr__pfet_01v8_hvt	0.0342	0.0191
	sky130_fd_pr__nfet_01v8	0.0225	0.0126
3	sky130_fd_pr__nfet_01v8	0.0377	0.0676
	sky130_fd_pr__pfet_01v8_hvt	0.0573	0.1027
	sky130_fd_pr__pfet_01v8_hvt	0.1027	0.0573
	sky130_fd_pr__nfet_01v8	0.0676	0.0377
4	sky130_fd_pr__nfet_01v8	0.0566	0.1014
	sky130_fd_pr__pfet_01v8_hvt	0.0859	0.1541
	sky130_fd_pr__pfet_01v8_hvt	0.1541	0.0859
	sky130_fd_pr__nfet_01v8	0.1014	0.0566
5	sky130_fd_pr__nfet_01v8	0.065975	0.1183
	sky130_fd_pr__pfet_01v8_hvt	0.10023125	0.179725
	sky130_fd_pr__pfet_01v8_hvt	0.179725	0.10023125
	sky130_fd_pr__nfet_01v8	0.1183	0.065975
6	sky130_fd_pr__nfet_01v8	0.0707	0.1268
	sky130_fd_pr__pfet_01v8_hvt	0.1074	0.1926
	sky130_fd_pr__pfet_01v8_hvt	0.1926	0.1074
	sky130_fd_pr__nfet_01v8	0.1268	0.0707
7	sky130_fd_pr__nfet_01v8	0.0754	0.1352
	sky130_fd_pr__pfet_01v8_hvt	0.1146	0.2054
	sky130_fd_pr__pfet_01v8_hvt	0.2054	0.1146
	sky130_fd_pr__nfet_01v8	0.1352	0.0754

8	sky130_fd_pr__nfet_01v8	0.0943	0.1690
	sky130_fd_pr__pfet_01v8_hvt	0.1432	0.2568
	sky130_fd_pr__pfet_01v8_hvt	0.2568	0.1432
	sky130_fd_pr__nfet_01v8	0.1690	0.0943
9	sky130_fd_pr__nfet_01v8	0.1131	0.2028
	sky130_fd_pr__pfet_01v8_hvt	0.1718	0.3081
	sky130_fd_pr__pfet_01v8_hvt	0.3081	0.1718
	sky130_fd_pr__nfet_01v8	0.2028	0.1131
10	sky130_fd_pr__nfet_01v8	0.1320	0.2366
	sky130_fd_pr__pfet_01v8_hvt	0.2005	0.3595
	sky130_fd_pr__pfet_01v8_hvt	0.3595	0.2005
	sky130_fd_pr__nfet_01v8	0.2366	0.1320
11	sky130_fd_pr__nfet_01v8	0.1508	0.2704
	sky130_fd_pr__pfet_01v8_hvt	0.2291	0.4108
	sky130_fd_pr__pfet_01v8_hvt	0.4108	0.2291
	sky130_fd_pr__nfet_01v8	0.2704	0.1508

Table 5.6. Combinations of source and drain diffusion areas to determine minimum power consumption

Combination	Internal Power (uW)	Switching Power (uW)	Leakage Power (uW)
1	0.00584	0.00421	5.29E-08
2	**0.00582**	**0.00407**	**5.25E-08**
3	0.00587	0.00411	5.32E-08
4	0.00587	0.00411	5.32E-08
5	0.00586	0.00409	5.30E-08
6	0.00589	0.00412	5.32E-08
7	**0.00582**	**0.00405**	**5.28E-08**
8	0.00588	0.00412	5.29E-08
9	0.00588	0.00413	5.28E-08
10	0.00583	0.00414	5.30E-08
11	0.00585	0.00422	5.31E-08

Table 5.7. Power consumption with varying parameter combinations

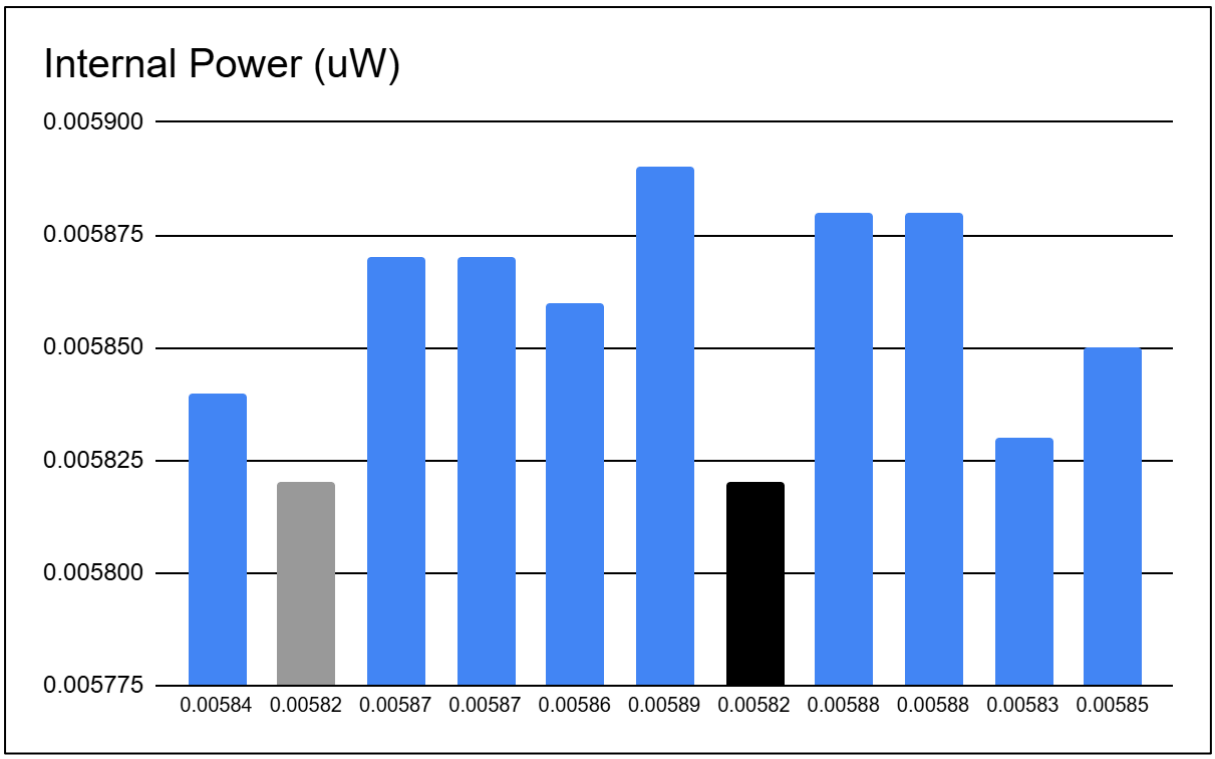


Figure 5.7. Internal Power comparison

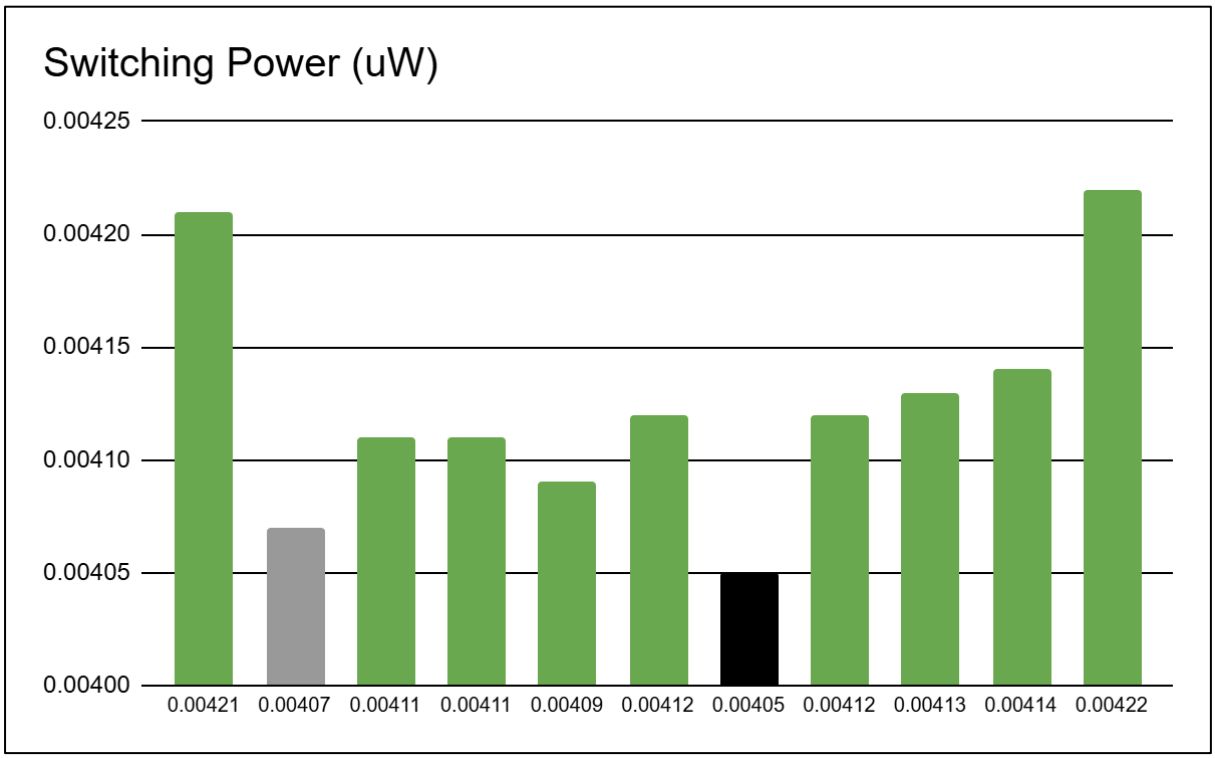


Figure 5.8. Switching Power comparison

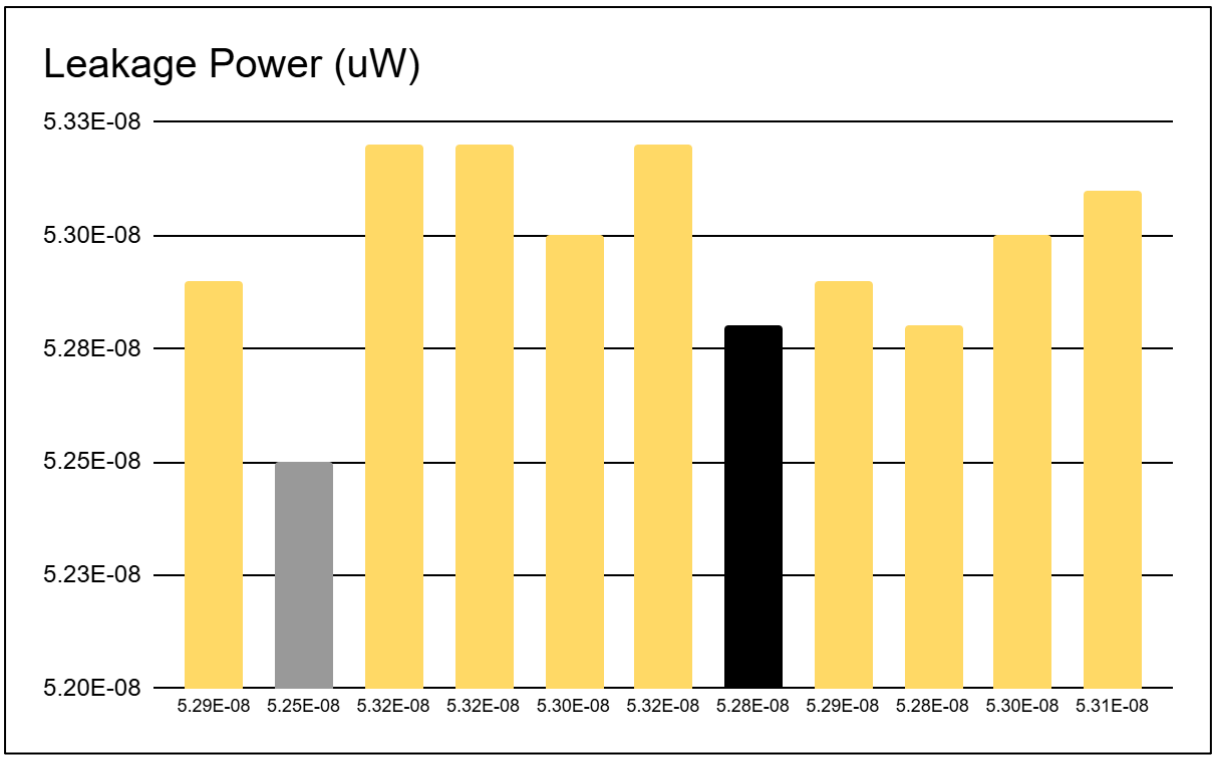


Figure 5.9. Leakage Power comparison

The figures 5.7, 5.8 and 5.9 represent the data in graphical form. From these figures, it is evident that combination 7 provides the best result for both Switching Power. But in the case of Leakage power, 2 provides the best results. Both 2 and 7 have the same Internal Power consumptions.

Therefore, considering the better Leakage power, combination 2 is selected to design the CPU as an improvement over the previous design made using default parameters. Table 5.8 shows the final design parameters with better Leakage power.

Parameters	Initial Design	Improved Design	Percentage Reduction
Internal Power drop (uW)	0.00601	0.00582	3.16%
Switching Power (uW)	0.00426	0.00405	4.93%
Leakage Power (uW)	5.56E-08	5.28E-08	5.04%

Table 5.8. Comparison between initial and improved design

Chapter 6 : Conclusion

The development of a custom RISC-V CPU using entirely open-source tools – and, importantly, the creation of an open-source standard cell characterization flow – carries significant implications for the university and open hardware community. First and foremost, it demonstrates that **academic researchers and students can now undertake the complete ASIC design cycle** (from RTL to GDS, including library creation) without needing access to expensive proprietary EDA tools. This opens new opportunities for hands-on education in VLSI design. Students can learn the intricacies of cell library development; a topic typically glossed over in curricula due to the inaccessibility of tools like Liberate. By using this flow, they can witness how transistor-level characteristics translate into the timing models that digital tools use, thereby gaining a deeper understanding of digital design and timing analysis. Such experience is invaluable for training the next generation of chip designers, and it was previously hard to attain outside of industry. Table 6.1 lists all the open-source tools used in this whole work

Operation	Tool
Verilog Simulation	Verilator
Schematic Simulation	Xschem
Schematic and Layout Design	Magic
SPICE Simulation	NgSpice
Verilog Synthesis	Yosys
Place and Route	OpenLane
Layout Versus Schematic	NetGen
Static Timing Analysis	OpenSTA

Table 6.1 Open-source tools utilized

Throughout the CPU design process, having the ability to create and characterize custom cells proved valuable. It allowed experiments with non-standard cell that potentially reduced logic depth or saved area/power in the CPU, something not possible if one is limited to a fixed foundry library. Moreover, because the characterization is done on demand, the cells were verified across multiple PVT corners to check the CPU's robustness. The experience validated that a small team of researchers or students can undertake the *full custom design cycle (including library development)* with entirely open tools – a methodology that can be replicated for other digital designs in academia.

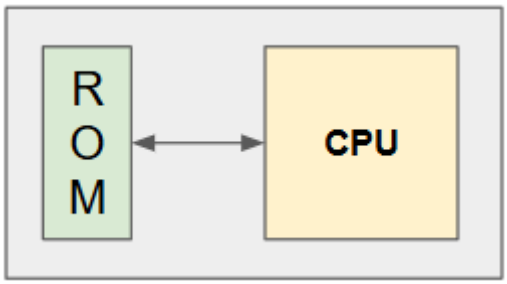
For faculty and researchers, having an open characterization capability means they can pursue more ambitious research projects in circuits and architecture. They can prototype novel standard cells or circuit techniques and immediately characterize and evaluate them in the context of larger systems. This lowers the barrier to innovation at the circuit level – one no longer needs to rely solely on whatever cells the foundry library provides. In the context of research, an open characterization flow also fosters reproducibility. Research results (such as a new cell design or a new low-power methodology) can be shared along with the cell libraries characterized by our flow, enabling other groups to replicate experiments or integrate the new cells into their own designs. This kind of sharing is often hampered when proprietary formats or tools are involved, because not everyone has access to run the same characterization. Open-source tools mitigate that problem, allowing broader collaboration and verification of results.

Another important impact is on the broader open hardware ecosystem. With open ISAs like RISC-V and open PDKs now available, having an open library characterization method completes the toolkit needed for a truly open-source SoC design workflow. This can accelerate the development of open-source hardware IP. Universities, for example, could develop and share optimized cell libraries for specific applications (e.g., ultra-low-power IoT devices or radiation-hardened cells for space applications) characterized by using this flow, thus building

a repository of technology-specific libraries that anyone can use. Since this work used a 130 nm node, one future direction is to apply similar flows to more advanced nodes as open PDKs emerge.

Further testing the reliability of design iterations using the libraries, further experiments were done by varying the device parameters of the gates to achieve 5.04% less Leakage Power in the design.

Also another addition to this work is the design of a memory and added to the CPU using open source memory compilers like OpenRAM [57].



Each success in this area reinforces the feedback loop of open innovation: more open tools lead to more open designs, which in turn attract more contributors and investments into open-source EDA. The significance of this work is that it pushes the boundary of what is achievable with open source in VLSI design, moving the community one step closer to a world where students and engineers can build state-of-the-art chips with nothing more than freely available resources.

References

[1] J. Yih, “The Promise of Open Source EDA Software,” *Asianometry* (online video/article), May 11, 2022.

[2] T. Quarles, D. Pederson, R. Newton, A. Sangiovanni-Vincentelli, and C. Wayne, “The SPICE Home Page,” University of California, Berkeley. [Online]. Available: https://bwrcs.eecs.berkeley.edu/Classes/IcBook/SPICE/.

[3] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, “Magic VLSI Layout Tool,” Open Circuit Design, [Online]. Available: http://opencircuitdesign.com/magic/.

[4] VIS : A System for Verification and Synthesis Robert K. Brayton , Gary D. Hachtel , Alberto Sangiovanni-Vincentelli , Fabio Somenziy , Adnan Aziz , Szu-Tsung Cheng , Stephen Edwards , Sunil Khatri , Yuji Kukimoto , Abelardo Pardo , Shaz Qadeer , Rajeev K. Ranjan , Shaker Sarwary , Thomas R. Shiple , Gitanjali Swamy , Tiziano Villa *Proceedings of the 8th International Conference in Computer-Aided Verification*, January 1996

[5] T. Edwards, “Qflow: An Open-Source Digital Synthesis Flow,” Open Circuit Design. [Online]. Available: http://opencircuitdesign.com/qflow/.

[6] A. B. Kahng and T. Spyrou, “The OpenROAD Project: Unleashing Hardware Innovation,” The OpenROAD Project. [Online]. Available: https://theopenroadproject.org/

[7] Google + SkyWater Technology, “SkyWater SKY130 PDK Documentation,” SkyWater Technology Foundry, [Online]. Available: https://skywater-pdk.readthedocs.io/en/main/

[8] P. Wagner, “Produce your own physical chips. For free. In the Open.” *FOSSi Foundation Blog*, June 30, 2020

[9] S. Nishizawa and T. Nakura, “Libretto: An Open Cell Timing Characterizer for Open Source VLSI Design,” *IEICE Transactions on Fundamentals of Electronics*, vol. E106-A, no. 3, pp. 551–560, Mar. 2023

[10] Team VLSI, “LIB and LEF File in ASIC Design,” Team VLSI, May 8, 2020. [Online]. Available: https://teamvlsi.com/2020/05/lib-and-lef-file-in-asic-design.html

[11] Synopsys, Inc., “What is Library Characterization? – How it Works & Techniques,” Synopsys, [Online]. Available: https://www.synopsys.com/glossary/what-is-library-characterization.html.

[12] Paripath Inc., “Standard Cell Library Characterization,” Paripath Inc. [Online]. Available: https://www.paripath.com/blog/standard-cell-library-characterization

[13] Cadence Design Systems, Inc., “Liberate Trio Characterization Suite,” Cadence Design Systems, Inc. [Online]. Available: https://www.cadence.com/en_US/home/tools/custom-ic-analog-rf-design/library-characterization/liberate-trio-characterization-suite.html

[14] J. Stine, “Interview with James Stine - Open Source Standard Cells,” *Zero to ASIC Course*, Oct. 8, 2022. [Online]. Available: https://www.zerotoasiccourse.com/post/interview-with-james-stine/.

[15] Mellor, Marcus & Stine, James. (2024). CharLib: An Open Source Standard Cell Library Characterizer. 277-281. 10.1109/MWSCAS60917.2024.10658687.

[16] Team VLSI, "*LIB File in Physical Design | Liberty File | .lib file*" Team VLSI, May 8, 2020. [Online]. Available: https://teamvlsi.com/2020/05/lib-and-lef-file-in-asicdesign.html#:~:text=LIB%20file%20is%20an%20ASCII

[17] "NLDM VS CCS," PHYSICAL DESIGN FOR ASIC. https://asicforphysicaldesign.blogspot.com/2015/01/nldm-vs-ccs.html#:~:text=NLDM%3A%20nonlinear%20delay%20models.,driver%20modeling%20(norton's%20theorem).

[18] "Transition Delay and Propagation Delay," ASIC-System on Chip-VLSI Design. https://asic-soc.blogspot.com/2008/12/transition-delay-and-propagation-delay.html

[19] M. S. Rahman, S. H Shuvo, Z. Kamal and T. Rahman "Standard Cell Library Characterization Flow Using Open Source Tools with Google + SkyWater 130nm PDK". 2024 6th International Conference on Electrical Engineering and Information & Communication Technology (ICEEICT) Military Institute of Science and Technology (MIST), Dhaka-1216, Bangladesh.

[20] M. S. Rahman, S. H Shuvo and T. Rahman "Timing and Power characterization of Standard cell with Google + SkyWater 130nm PDK using Open-Source Tools". 12th International Conference on Internet of Everything, Microwave, Embedded, Communication & Networks (IEMECON 2024), University of Engineering & Management, Jaipur, India

[21] "How delay of a standard cell changes with drive strength," How delay of a standard cell changes with drive strength. Available: https://vlsiuniverse.blogspot.com/2017/12/how-delay- ofstandard-cell-changes-with.html (accessed Jan. 30, 2024). 12. Verilator. https://www.veripool.org/verilator/

[22] M. Mano, *Computer System Architecture*, 3rd ed. -
Central Processing Unit (CPU) definition and components. Available: https://www.cise.ufl.edu/~mssz/CompOrg/CDA-proc.html

[23] University of Florida CISE

– *Processor Organization*: Explanation of CPU as active part of computer

(datapath and control). Available: https://www.cise.ufl.edu/~mssz/CompOrg/CDA-proc.html

[24] Red Hat Blog– *The Central Processing Unit*

(CPU): Components and functionality: Description of ALU operations and registers.

Available: https://www.redhat.com/en/blog/cpu-components-functionality

[25] Total Phase Blog
– *What is a Register in a CPU?*: Role of control unit and data flow in CPU. Available: https://www.totalphase.com/blog/2023/05/what-is-register-in-cpu-how-does-it-work/?srsltid=AfmBOooUWiv9BnfZwnv59bAPoTDLR48ucB1qh_QgcAu-nwRvtHbAmBEV

[26] I. Xzy, *Computer Architecture Notes* – Explanation of registers as fast storage in CPU. Available: https://irisxzy.wordpress.com/2016/10/01/cpu-alu-cu-and-registers/

[27] Wikipedia – *Classic RISC Pipeline*: Five-stage pipeline (IF, ID, EX, MEM, WB) in RISC CPUs. Available: https://en.wikipedia.org/wiki/Classic_RISC_pipeline

[28] Wikipedia – *Instruction Set Architecture*: Examples of instruction types (data handling, arithmetic/logic, control flow, etc.)

Available: https://en.wikipedia.org/wiki/Instruction_set_architecture

[29] N. & J. Dale, *Computer Science Illuminated* – Discussion of machine instruction cycle (fetch–decode–execute) and storing results. Available: https://irisxzy.wordpress.com/2016/10/01/cpu-alu-cu-and-registers/

[30] Codasip Glossary
– *ISA Definition*: ISA as interface between software and hardware, abstraction of

instructions and registers. Available: https://codasip.com/glossary/isa/

[31] Semiconductor Engineering
– *ISA Knowledge Center*: Independence of ISA from microarchitecture; standard instruction

categories. Available: https://semiengineering.com/knowledge_centers/compute-architectures/instruction-set-architecture-isa/

[32] Stack Overflow – *RISC-V base ISA explanation*: RV32I, RV64I, RV128I definitions and differences in register width and usage. Available: https://stackoverflow.com/questions/62807066/riscv32-vs-riscv64

[33] RISC-V Foundation – *ISA Spec Table*: Base ISA variants (32, 64, 128-bit), ratification status, and instruction counts. Available: https://en.wikipedia.org/wiki/RISC-V

[34] Wikipedia – *RISC-V Overview*: 32-bit and 64-bit ISA variants are defined; 128 bit ISA remains not frozen due to lack of use cases. Available: https://en.wikipedia.org/wiki/RISC-V

[35] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA," EECS Department, University of California, Berkeley, 2019.

[36] D. Patterson, "Why RISC-V is the Future of Open Source Hardware," IEEE Micro, vol. 40, no. 4, pp. 14–24, 2020.

[37] K. Asanović et al., "The RISC-V Instruction Set Manual Volume II: Privileged Architecture," University of California, Berkeley, 2019.

[38] RISC-V International, “RISC-V Specifications,” [Online]. Available: https://riscv.org/specifications/

[39] RISC-V International, “About RISC-V,” [Online]. Available: https://riscv.org/about/

[40] L. Franchini and F. Conti, "Custom RISC-V Cores for Energy-Efficient Edge AI," in Proc. DATE Conf., 2021.

[41] A. Ardizzone et al., "RISCV: A Modular ISA for High-Performance Computing," ACM Trans. Archit. Code Optim., vol. 16, no. 4, pp. 1–23, 2019.

[42] J. L. Hennessy and D. A. Patterson, "A New Golden Age for Computer Architecture," Commun. ACM, vol. 62, no. 2, pp. 48–60, 2019.

[43] SkyWater PDK, "SkyWater SKY130 PDK Documentation," [Online]. Available: https://skywater-pdk.readthedocs.io/en/main/.

[32] Stack Overflow – RISC-V base ISA explanation: RV32I, RV64I, RV128I definitions and differences in register

width and usage. Available: https://stackoverflow.com/questions/62807066/riscv32-vs-riscv64

[33] RISC-V Foundation – ISA Spec Table: Base ISA variants (32, 64, 128-bit), ratification status, and instruction counts. Available: https://en.wikipedia.org/wiki/RISC-V

[34] Wikipedia – RISC-V Overview: 32-bit and 64-bit ISA variants are defined; 128 bit ISA remains not frozen due to lack of use cases. Available: https://en.wikipedia.org/wiki/RISC-V

[35] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA," EECS Department, University of California, Berkeley, 2019.

[36] D. Patterson, "Why RISC-V is the Future of Open Source Hardware," IEEE Micro, vol. 40, no. 4, pp. 14–24, 2020.

[37] K. Asanović et al., "The RISC-V Instruction Set Manual Volume II: Privileged Architecture," University of California, Berkeley, 2019.

[38] RISC-V International, “RISC-V Specifications,” [Online]. Available: https://riscv.org/specifications/

[39] RISC-V International, “About RISC-V,” [Online]. Available: https://riscv.org/about/

[40] L. Franchini and F. Conti, "Custom RISC-V Cores for Energy-Efficient Edge AI," in Proc. DATE Conf., 2021.

[41] A. Ardizzone et al., "RISCV: A Modular ISA for High-Performance Computing," ACM Trans. Archit. Code Optim., vol. 16, no. 4, pp. 1–23, 2019.

[42] J. L. Hennessy and D. A. Patterson, "A New Golden Age for Computer Architecture," Commun. ACM, vol. 62, no. 2, pp. 48–60, 2019.

[43] SkyWater PDK, "SkyWater SKY130 PDK Documentation," [Online]. Available: https://skywater-pdk.readthedocs.io/en/main/.

[44] Magic VLSI Tool User Guide. [Online]. Available: http://opencircuitdesign.com/magic/

[45] ECE451 Layout Tutorial, Brigham Young University, 2008

[46] Edaboard Forum. "PMOS sizing in CMOS inverters." [Online]. Available: https://www.edaboard.com.

[47] Xschem Documentation. [Online]. Available: https://xschem.sourceforge.io.

[48] Analog Devices Wiki, "CMOS D Latch." [Online]. Available: https://wiki.analog.com.

[49] K. Asanović et al., "The RISC-V Instruction Set Manual Volume II: Privileged Architecture," University of California, Berkeley, 2019.

[50] RISC-V International, “RISC-V Specifications,” [Online]. Available: https://riscv.org/specifications/

[51] RISC-V International, “About RISC-V,” [Online]. Available: https://riscv.org/about/

[52] M. Shalan and T. Edwards, “Building OpenLANE: A 130nm OpenROAD-based Tapeout-Proven Flow: Invited Paper,” *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, San Diego, CA, USA, 2020, pp. 1-6

[53] OpenLane Documentation, efabless.com, 2022. [Online]. Available: https://github.com/The-OpenROAD-Project/OpenLane/blob/master/docs/source/index.rst.

[54] M. Popovich et al., "OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain," in Proc. DAC, 2019. [Online]. Available: https://theopenroadproject.org.

[55] Efabless, “User Guide: OpenLane Design Flow,” [Online]. Available: https://efabless.com/openlane.

[56] S. Raj, "Automated DRC and LVS using Magic and Netgen," GitHub Gist, 2022. [Online]. Available: https://gist.github.com.

[57] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, M. Sarwar, “OpenRAM: An Open-Source Memory Compiler,” Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD), 2016.

Appendix A

Layout SPICE

I. INVX1

```
.subckt sky130_fd_sc_hd__inv_1 A vss vnb vpb vdd Y
X0 Y.t0 A.t0 vss.t0 vnb.t0 sky130_fd_pr__nfet_01v8 ad=0 pd=0 as=0 ps=0
w=0.65 l=0.15
X1 Y.t1 A.t1 vdd.t0 vpb.t0 sky130_fd_pr__pfet_01v8_hvt ad=0 pd=0 as=0
ps=0 w=1 l=0.15
C0 A vpb 0.045062f
C1 vdd vpb 0.054478f
C2 Y vss 0.099841f
C3 vss A 0.040045f
C4 vss vdd 0.033816f
C5 vss vpb 0.009478f
C6 Y A 0.047605f
C7 Y vdd 0.127579f
C8 A vdd 0.037031f
C9 Y vpb 0.017744f
R0 A.n0 A.t1 230.576
R1 A A.n0 158.667
R2 A.n0 A.t0 158.275
R3 vss vss.t0 166.951
R4 Y.n0 Y.t1 235.56
R5 Y Y.t0 152.889
R6 Y Y.n0 2.22659
R7 Y.n0 Y 1.55202
R8 vnb vnb.t0 1612.5
R9 vdd vdd.t0 264.904
R10 vpb vpb.t0 350.853
C10 vss vnb 0.251126f
C11 Y vnb 0.096099f
C12 vdd vnb 0.218922f
C13 A vnb 0.166643f
C14 vpb vnb 0.338976f
.ends
```

II. INVX2

```
.subckt sky130_fd_sc_hd__inv_2 A vss vnb vpb vdd Y
X0 Y.t1 A.t0 vdd.t1 VPB.t1 sky130_fd_pr__pfet_01v8_hvt w=1 l=0.15
X1 vss.t1 A.t1 Y.t2 VNB.t1 sky130_fd_pr__nfet_01v8 w=0.65 l=0.15
X2 Y.t3 A.t2 vss.t0 VNB.t0 sky130_fd_pr__nfet_01v8 w=0.65 l=0.15
X3 vdd.t0 A.t3 Y.t0 VPB.t0 sky130_fd_pr__pfet_01v8_hvt w=1 l=0.15
R0 A.n0 A.t3 212.081
R1 A.n1 A.t0 212.081
R2 A A.n1 189.073
R3 A.n0 A.t1 139.78
R4 A.n1 A.t2 139.78
R5 A.n1 A.n0 61.346
R6 vdd.n0 vdd.t0 262.851
R7 vdd.n0 vdd.t1 259.721
R8 vdd vdd.n0 0.491471
R9 Y.n2 Y.n1 208.965
R10 Y Y.n0 96.8352
R11 Y.n1 Y.t0 26.5955
R12 Y.n1 Y.t1 26.5955
R13 Y.n0 Y.t2 24.9236
R14 Y.n0 Y.t3 24.9236
R15 Y.n3 Y 11.2645
R16 Y Y.n3 6.1445
R17 Y.n3 Y 4.65505
R18 Y Y.n2 2.0485
R19 Y.n2 Y 1.55202
R20 VPB.t1 VPB.t0 248.599
R21 VPB VPB.t1 198.287
R22 vss.n0 vss.t1 169.418
R23 vss.n0 vss.t0 166.787
R24 vss vss.n0 0.491471
R25 VNB.t0 VNB.t1 1196.12
R26 VNB VNB.t0 954.045
C0 VPB Y 0.006097f
C1 vdd Y 0.209105f
C2 A Y 0.089386f
C3 vdd VPB 0.052063f
C4 VPB A 0.074183f
C5 vss Y 0.154601f
C6 vss VPB 0.006491f
C7 vdd A 0.06305f
C8 vss vdd 0.042274f
C9 vss A 0.063754f
C10 vss VNB 0.266187f
C11 Y VNB 0.03316f
C12 vdd VNB 0.246044f
C13 A VNB 0.262807f
C14 VPB VNB 0.338976f
.ends
```

III. BUFX1

```
.subckt sky130_fd_sc_hd__buf_1 A VGND VNB VPB VPWR X
X0 VPWR.t0 A.t0 a_27_47.t1 VPB.t0 sky130_fd_pr__pfet_01v8_hvt ad=0
pd=0 as=0 ps=0 w=0.79 l=0.15
X1 X.t1 a_27_47.t2 VGND.t1 VNB.t1 sky130_fd_pr__nfet_01v8 ad=0 pd=0
as=0 ps=0 w=0.52 l=0.15
X2 X.t0 a_27_47.t3 VPWR.t1 VPB.t1 sky130_fd_pr__pfet_01v8_hvt ad=0
pd=0 as=0 ps=0 w=0.79 l=0.15
X3 VGND.t0 A.t1 a_27_47.t0 VNB.t0 sky130_fd_pr__nfet_01v8 ad=0 pd=0
as=0 ps=0 w=0.52 l=0.15
R0 A.n0 A.t0 260.322
R1 A.n0 A.t1 175.169
R2 A A.n0 156.325
R3 a_27_47.t1 a_27_47.n1 405.735
R4 a_27_47.n1 a_27_47.t0 294.611
R5 a_27_47.n0 a_27_47.t3 254.389
R6 a_27_47.n0 a_27_47.t2 211.01
R7 a_27_47.n1 a_27_47.n0 152
R8 VPWR.n1 VPWR.n0 316.233
R9 VPWR.n0 VPWR.t1 36.1587
R10 VPWR.n0 VPWR.t0 36.1587
R11 VPWR.n1 VPWR 11.7224
R12 VPWR VPWR.n1 1.81142
R13 VPB.t0 VPB.t1 260.437
R14 VPB VPB.t0 192.369
R15 VGND.n1 VGND.n0 203.922
R16 VGND.n0 VGND.t1 33.462
R17 VGND.n0 VGND.t0 33.462
R18 VGND.n1 VGND 10.9695
R19 VGND VGND.n1 1.81142
R20 X.n1 X.t0 368.521
R21 X.n0 X.t1 216.155
R22 X X.n0 78.8791
R23 X.n1 X 10.5563
R24 X X.n1 5.48477
R25 X.n0 X 5.16973
R26 VNB.t0 VNB.t1 1253.07
R27 VNB VNB.t0 897.087
C0 VPWR X 0.089678f
C1 A X 8.48e-19
C2 VGND VPWR 0.028968f
C3 VGND A 0.018425f
C4 VPB VPWR 0.035491f
C5 VPB A 0.052393f
C6 VGND X 0.054627f
C7 VPB X 0.012762f
C8 VPB VGND 0.00505f
C9 A VPWR 0.021545f
C10 VGND VNB 0.207322f
C11 X VNB 0.094114f
C12 VPWR VNB 0.175402f
C13 A VNB 0.164055f
C14 VPB VNB 0.338976f
.ends
```

IV. BUFX2

```
.subckt sky130_fd_sc_hd__buf_2 A VGND VNB VPB VPWR X
X0 VPWR.t1 a_27_47.t2 X.t3 VPB.t1 sky130_fd_pr__pfet_01v8_hvt
ad=0 pd=0 as=0 ps=0 w=1 l=0.15
X1 X.t0 a_27_47.t3 VPWR.t0 VPB.t0 sky130_fd_pr__pfet_01v8_hvt
ad=0 pd=0 as=0 ps=0 w=1 l=0.15
X2 VPWR.t2 A.t0 a_27_47.t0 VPB.t2 sky130_fd_pr__pfet_01v8_hvt
ad=0 pd=0 as=0 ps=0 w=0.64 l=0.15
X3 X.t1 a_27_47.t4 VGND.t1 VNB.t1 sky130_fd_pr__nfet_01v8 ad=0
pd=0 as=0 ps=0 w=0.65 l=0.15
X4 VGND.t0 a_27_47.t5 X.t2 VNB.t0 sky130_fd_pr__nfet_01v8 ad=0
pd=0 as=0 ps=0 w=0.65 l=0.15
X5 VGND.t2 A.t1 a_27_47.t1 VNB.t2 sky130_fd_pr__nfet_01v8 ad=0
pd=0 as=0 ps=0 w=0.42 l=0.15
R0 a_27_47.t0 a_27_47.n2 447.445
R1 a_27_47.n2 a_27_47.t1 301.502
R2 a_27_47.n0 a_27_47.t2 212.081
R3 a_27_47.n1 a_27_47.t3 212.081
R4 a_27_47.n2 a_27_47.n1 160.034
R5 a_27_47.n0 a_27_47.t5 139.78
R6 a_27_47.n1 a_27_47.t4 139.78
R7 a_27_47.n1 a_27_47.n0 61.346
R8 X.n0 X 589.769
R9 X.n1 X.n0 585
R10 X.n3 X.n2 185
R11 X X.n3 81.3181
R12 X.n0 X.t3 26.5955
R13 X.n0 X.t0 26.5955
R14 X.n2 X.t2 24.9236
R15 X.n2 X.t1 24.9236
R16 X.n1 X 15.5613
R17 X.n3 X 5.27109
R18 X X.n1 1.50638
R19 VPWR.n2 VPWR.n1 312.051
R20 VPWR.n0 VPWR.t1 256.243
R21 VPWR.n1 VPWR.t2 55.4067
R22 VPWR.n1 VPWR.t0 34.0906
R23 VPWR.n3 VPWR.n2 24.8476
R24 VPWR.n4 VPWR.n3 9.3005
R25 VPWR.n3 VPWR 6.4005
R26 VPWR.n2 VPWR.n0 6.3848
R27 VPWR.n4 VPWR.n0 0.657345
R28 VPWR VPWR.n4 0.0603958
R29 VPB.t2 VPB.t0 281.154
R30 VPB.t0 VPB.t1 248.599
R31 VPB VPB.t2 192.369
R32 A.n0 A.t0 276.464
R33 A.n0 A.t1 196.131
R34 A A.n0 156.325
R35 VGND.n2 VGND.n1 199.739
```

```
R36 VGND.n0 VGND.t0 161.442
R37 VGND.n1 VGND.t2 51.4291
R38 VGND.n1 VGND.t1 28.7917
R39 VGND.n3 VGND.n2 24.8476
R40 VGND.n4 VGND.n3 9.3005
R41 VGND.n3 VGND 6.4005
R42 VGND.n2 VGND.n0 6.3848
R43 VGND.n4 VGND.n0 0.657345
R44 VGND VGND.n4 0.0603958
R45 VNB.t2 VNB.t1 1352.75
R46 VNB.t1 VNB.t0 1196.12
R47 VNB VNB.t2 925.567
C0 VGND X 0.111264f
C1 VGND A 0.019887f
C2 VGND VPB 0.006417f
C3 X VPWR 0.168994f
C4 A VPWR 0.022108f
C5 X A 0.003062f
C6 VPB VPWR 0.05279f
C7 X VPB 0.004428f
C8 VPB A 0.069727f
C9 VGND VPWR 0.051753f
C10 VGND VNB 0.284185f
C11 X VNB 0.021498f
C12 VPWR VNB 0.249629f
C13 A VNB 0.187471f
C14 VPB VNB 0.427572f
.ends
```

V. NANDX1

```
.subckt sky130_fd_sc_hd__nand2_1 A B vss vnb vpb vdd Y
X0 vdd.t1 A.t0 Y.t2 vpb.t1 sky130_fd_pr__pfet_01v8_hvt ad=0 pd=0 as=0
ps=0 w=1 l=0.15
X1 Y.t1 A.t1 a_113_47.t1 vnb.t1 sky130_fd_pr__nfet_01v8 ad=0 pd=0 as=0
ps=0 w=0.65 l=0.15
X2 a_113_47.t0 B.t0 vss.t0 vnb.t0 sky130_fd_pr__nfet_01v8 ad=0 pd=0
as=0 ps=0 w=0.65 l=0.15
X3 Y.t0 B.t1 vdd.t0 vpb.t0 sky130_fd_pr__pfet_01v8_hvt ad=0 pd=0 as=0
ps=0 w=1 l=0.15
R0 A.n0 A.t0 230.155
R1 A.n0 A.t1 157.856
R2 A A.n0 154.102
R3 Y Y.n0 237.577
R4 Y.n1 Y.t1 140.53
R5 Y.n0 Y.t2 26.5955
R6 Y.n0 Y.t0 26.5955
R7 Y.n1 Y 16.5652
R8 Y Y.n1 9.03579
R9 Y.n1 Y 1.72748
R10 vdd.n0 vdd.t1 256.344
R11 vdd.n0 vdd.t0 254.13
R12 vdd vdd.n0 0.493374
R13 vpb.t0 vpb.t1 248.599
R14 vpb vpb.t0 207.166
R15 a_113_47.t0 a_113_47.t1 49.8467
R16 vnb.t0 vnb.t1 1196.12
R17 vnb vnb.t0 996.764
R18 B.n0 B.t1 229.369
R19 B B.n0 157.927
R20 B.n0 B.t0 157.07
R21 vss vss.t0 158.046
C0 vpb Y 0.006185f
C1 vpb vss 0.004396f
C2 B vpb 0.039072f
C3 A vdd 0.04444f
C4 Y vdd 0.211407f
C5 vss vdd 0.032185f
C6 B vdd 0.047843f
C7 A Y 0.085479f
C8 A vss 0.009489f
C9 vpb vdd 0.050862f
C10 B A 0.050963f
C11 vss Y 0.138901f
C12 B Y 0.048071f
C13 B vss 0.054404f
C14 A vpb 0.037877f
C15 vss vnb 0.23167f
C16 Y vnb 0.055661f
C17 vdd vnb 0.245114f
C18 A vnb 0.143376f
C19 B vnb 0.145827f
C20 vpb vnb 0.338976f
.ends
```

VI. NANDX2

```
.subckt sky130_fd_sc_hd__nand2_2 A B vss vnb vpb vdd Y
X0 vdd.t2 A.t0 Y.t1 vpb.t2 sky130_fd_pr__pfet_01v8_hvt ad=0 pd=0 as=0
ps=0 w=1 l=0.15
X1 Y.t2 A.t1 vdd.t1 vpb.t1 sky130_fd_pr__pfet_01v8_hvt ad=0 pd=0 as=0
ps=0 w=1 l=0.15
X2 vdd.t0 B.t0 Y.t0 vpb.t0 sky130_fd_pr__pfet_01v8_hvt ad=0 pd=0 as=0
ps=0 w=1 l=0.15
X3 a_27_47.t1 B.t1 vss.t1 vnb.t1 sky130_fd_pr__nfet_01v8 ad=0 pd=0
as=0 ps=0 w=0.65 l=0.15
X4 a_27_47.t3 A.t2 Y.t4 vnb.t3 sky130_fd_pr__nfet_01v8 ad=0 pd=0 as=0
ps=0 w=0.65 l=0.15
X5 Y.t3 A.t3 a_27_47.t2 vnb.t2 sky130_fd_pr__nfet_01v8 ad=0 pd=0 as=0
ps=0 w=0.65 l=0.15
X6 Y.t5 B.t2 vdd.t3 vpb.t3 sky130_fd_pr__pfet_01v8_hvt ad=0 pd=0 as=0
ps=0 w=1 l=0.15
X7 vss.t0 B.t3 a_27_47.t0 vnb.t0 sky130_fd_pr__nfet_01v8 ad=0 pd=0
as=0 ps=0 w=0.65 l=0.15
R0 A.n0 A.t0 212.081
R1 A.n1 A.t1 212.081
R2 A A.n2 152.94
R3 A.n0 A.t2 139.78
R4 A.n1 A.t3 139.78
R5 A.n2 A.n0 30.6732
R6 A.n2 A.n1 30.6732
R7 Y.n3 Y.n1 243.68
R8 Y.n4 Y.n0 206.249
R9 Y.n3 Y.n2 205.28
R10 Y.n1 Y.t0 26.5955
R11 Y.n1 Y.t5 26.5955
R12 Y.n2 Y.t1 26.5955
R13 Y.n2 Y.t2 26.5955
R14 Y Y.n3 24.9955
R15 Y.n0 Y.t4 24.9236
R16 Y.n0 Y.t3 24.9236
R17 Y.n4 Y 14.8576
R18 Y Y.n4 0.686214
R19 vdd.n2 vdd.t2 348.959
R20 vdd.n3 vdd.n1 320.976
R21 vdd.n5 vdd.t3 249.362
R22 vdd.n4 vdd.n3 30.8711
R23 vdd.n1 vdd.t1 26.5955
R24 vdd.n1 vdd.t0 26.5955
R25 vdd.n5 vdd.n4 25.977
R26 vdd.n3 vdd.n2 10.9193
R27 vdd.n4 vdd.n0 9.3005
R28 vdd.n6 vdd.n5 9.3005
R29 vdd.n2 vdd.n0 0.572285
R30 vdd.n6 vdd.n0 0.120
R31 vdd vdd.n6 0.0213333
R32 vpb.t1 vpb.t2 248.599
R33 vpb.t0 vpb.t1 248.599
R34 vpb.t3 vpb.t0 248.599
R35 vpb vpb.t3 189.409
```

```
R36 B.n0 B.t0 212.081
R37 B.n1 B.t2 212.081
R38 B B.n2 155.584
R39 B.n0 B.t1 139.78
R40 B.n1 B.t3 139.78
R41 B.n2 B.n0 30.6732
R42 B.n2 B.n1 30.6732
R43 vss vss.n0 214.335
R44 vss.n0 vss.t1 24.9236
R45 vss.n0 vss.t0 24.9236
R46 a_27_47.n0 a_27_47.t3 322.56
R47 a_27_47.n0 a_27_47.t0 188.529
R48 a_27_47.n1 a_27_47.n0 88.3446
R49 a_27_47.n1 a_27_47.t2 24.9236
R50 a_27_47.t1 a_27_47.n1 24.9236
R51 vnb.t2 vnb.t3 1196.12
R52 vnb.t1 vnb.t2 1196.12
R53 vnb.t0 vnb.t1 1196.12
R54 vnb vnb.t0 911.327
C0 vss A 0.017966f
C1 vdd Y 0.398191f
C2 B A 0.074816f
C3 vss vpb 0.005765f
C4 B vpb 0.056795f
C5 A vpb 0.057111f
C6 vdd vss 0.047614f
C7 vss Y 0.020763f
C8 vdd B 0.059991f
C9 B Y 0.062045f
C10 vdd A 0.031531f
C11 A Y 0.177649f
C12 vdd vpb 0.066372f
C13 Y vpb 0.015949f
C14 vss B 0.02902f
C15 vss vnb 0.295829f
C16 Y vnb 0.062962f
C17 vdd vnb 0.309019f
C18 A vnb 0.197653f
C19 B vnb 0.209169f
C20 vpb vnb 0.516168f
.ends
```

VII. NORX1

```
.subckt sky130_fd_sc_hd__nor2_1 A B vss vnb vpb vdd Y
X0 vdd.t0 A.t0 a_109_297.t1 vpb.t1 sky130_fd_pr__pfet_01v8_hvt  ad=0
pd=0 as=0 ps=0 w=1 l=0.15
X1 vss.t1 A.t1 Y.t2 vnb.t1 sky130_fd_pr__nfet_01v8  ad=0 pd=0 as=0
ps=0 w=0.65 l=0.15
X2 a_109_297.t0 B.t0 Y.t1 vpb.t0 sky130_fd_pr__pfet_01v8_hvt ad=0 pd=0
as=0 ps=0 w=1 l=0.15
X3 Y.t0 B.t1 vss.t0 vnb.t0 sky130_fd_pr__nfet_01v8 ad=0 pd=0 as=0 ps=0
w=0.65 l=0.15
R0 A.n0 A.t0 231.835
R1 A.n0 A.t1 157.07
R2 A A.n0 154.012
R3 a_109_297.t0 a_109_297.t1 41.3705
R4 vdd vdd.t0 250.668
R5 vpb.t0 vpb.t1 213.084
R6 vpb vpb.t0 189.409
R7 Y Y.n0 593.34
R8 Y.n2 Y.n0 289.24
R9 Y.n2 Y.n1 176.839
R10 Y.n0 Y.t1 26.5955
R11 Y.n1 Y.t2 24.9236
R12 Y.n1 Y.t0 24.9236
R13 Y Y.n2 11.3699
R14 vss.n0 vss.t0 163.721
R15 vss.n0 vss.t1 161.042
R16 vss vss.n0 0.489856
R17 vnb.t0 vnb.t1 1196.12
R18 vnb vnb.t0 911.327
R19 B.n0 B.t0 230.363
R20 B B.n0 158.4
R21 B.n0 B.t1 158.064
C0 vdd B 0.014836f
C1 vss vdd 0.031443f
C2 vpb vdd 0.044857f
C3 vss B 0.045088f
C4 A vdd 0.052823f
C5 vpb B 0.036697f
C6 Y vdd 0.099513f
C7 A B 0.058413f
C8 Y B 0.087653f
C9 vss vpb 0.004563f
C10 vss A 0.048556f
C11 vss Y 0.154448f
C12 vpb A 0.041461f
C13 vpb Y 0.013918f
C14 Y A 0.047068f
C15 vss vnb 0.263197f
C16 vdd vnb 0.214143f
C17 Y vnb 0.060508f
C18 A vnb 0.14927f
C19 B vnb 0.143121f
C20 vpb vnb 0.338976f
.ends
```

VIII. NORX2

```
.subckt sky130_fd_sc_hd__nor2_2 A B vss vnb vpb vdd Y
X0 Y.t2 B.t0 a_27_297.t1 vpb.t1 sky130_fd_pr__pfet_01v8_hvt d=0
pd=0 as=0 ps=0 w=1 l=0.15
X1 Y.t0 B.t1 vss.t1 vnb.t1 sky130_fd_pr__nfet_01v8 ad=0 pd=0
as=0 ps=0 w=0.65 l=0.15
X2 a_27_297.t2 A.t0 vdd.t1 vpb.t2 sky130_fd_pr__pfet_01v8_hvt
d=0 pd=0 as=0 ps=0 w=1 l=0.15
X3 vss.t3 A.t1 Y.t5 vnb.t3 sky130_fd_pr__nfet_01v8 ad=0 pd=0
as=0 ps=0 w=0.65 l=0.15
X4 Y.t4 A.t2 vss.t2 vnb.t2 sky130_fd_pr__nfet_01v8 ad=0 pd=0
as=0 ps=0 w=0.65 l=0.15
X5 vss.t0 B.t2 Y.t3 vnb.t0 sky130_fd_pr__nfet_01v8 ad=0 pd=0
as=0 ps=0 w=0.65 l=0.15
X6 vdd.t0 A.t3 a_27_297.t3 vpb.t3 sky130_fd_pr__pfet_01v8_hvt
ad=0 pd=0 as=0 ps=0 w=1 l=0.15
X7 a_27_297.t0 B.t3 Y.t1 vpb.t0 sky130_fd_pr__pfet_01v8_hvt ad=0
pd=0 as=0 ps=0 w=1 l=0.15
R0 B.n0 B.t3 212.081
R1 B.n1 B.t0 212.081
R2 B B.n2 182.081
R3 B.n0 B.t2 139.78
R4 B.n1 B.t1 139.78
R5 B.n2 B.n0 30.6732
R6 B.n2 B.n1 30.6732
R7 a_27_297.t0 a_27_297.n1 404.502
R8 a_27_297.n1 a_27_297.t3 301.301
R9 a_27_297.n1 a_27_297.n0 184.905
R10 a_27_297.n0 a_27_297.t1 26.5955
R11 a_27_297.n0 a_27_297.t2 26.5955
R12 Y Y.n0 591.788
R13 Y.n4 Y.n0 585
R14 Y.n3 Y.n1 135.249
R15 Y.n3 Y.n2 98.982
R16 Y.n4 Y.n3 74.7314
R17 Y.n0 Y.t1 26.5955
R18 Y.n0 Y.t2 26.5955
R19 Y.n2 Y.t3 24.9236
R20 Y.n2 Y.t0 24.9236
R21 Y.n1 Y.t5 24.9236
R22 Y.n1 Y.t4 24.9236
R23 Y Y.n4 6.4005
R24 vpb.t1 vpb.t0 248.599
R25 vpb.t2 vpb.t1 248.599
R26 vpb.t3 vpb.t2 248.599
R27 vpb vpb.t3 201.246
R28 vss.n2 vss.t0 287.377
R29 vss.n3 vss.n1 207.965
R30 vss.n5 vss.t2 150.922
R31 vss.n4 vss.n3 32.377
```

```
R32 vss.n1 vss.t1 24.9236
R33 vss.n1 vss.t3 24.9236
R34 vss.n5 vss.n4 24.4711
R35 vss.n3 vss.n2 9.43392
R36 vss.n6 vss.n5 9.3005
R37 vss.n4 vss.n0 9.3005
R38 vss.n2 vss.n0 0.554787
R39 vss.n6 vss.n0 0.120292
R40 vss vss.n6 0.0213333
R41 vnb.t1 vnb.t0 1196.12
R42 vnb.t3 vnb.t1 1196.12
R43 vnb.t2 vnb.t3 1196.12
R44 vnb vnb.t2 968.285
R45 A.n0 A.t0 212.081
R46 A.n1 A.t3 212.081
R47 A A.n2 183.68
R48 A.n0 A.t1 139.78
R49 A.n1 A.t2 139.78
R50 A.n2 A.n0 38.7066
R51 A.n2 A.n1 22.6399
R52 vdd vdd.n0 316.39
R53 vdd.n0 vdd.t1 26.5955
R54 vdd.n0 vdd.t0 26.5955
C0 vdd A 0.041833f
C1 B vpb 0.056648f
C2 Y B 0.179365f
C3 B vss 0.029406f
C4 vdd vpb 0.049962f
C5 Y vdd 0.012692f
C6 vdd vss 0.046681f
C7 A vpb 0.056273f
C8 Y A 0.052333f
C9 vss A 0.059727f
C10 Y vpb 0.009606f
C11 vss vpb 0.006128f
C12 Y vss 0.289148f
C13 B vdd 0.017427f
C14 B A 0.071165f
C15 vss vnb 0.342832f
C16 Y vnb 0.064071f
C17 vdd vnb 0.248551f
C18 B vnb 0.197736f
C19 A vnb 0.206739f
C20 vpb vnb 0.516168f
.ends
```

Simulation SPICE templates

I. Inverter Rise Template

```
Vdd vdd gnd 1.8
Vin a gnd PULSE(0 1.8 5n 16.67p 0.1n 10n 20n)
*Vin a GND PWL(0 gnd 5n gnd 5.01667n 1.8 10n 1.8)
Cload net2 gnd __LOAD__

V1 y net2 0
V2 net1 gnd 0
V3 vdd net3 0

Xinv a net1 net1 net3 net3 y __CELL_NAME__

.tran 0.001n 40n

.meas tran rise_tran     TRIG v(y) VAL=1.44 FALL=1 TARG v(y)
VAL=0.36 FALL=1
.meas tran tpdr          TRIG v(a) VAL=0.9  RISE=1 TARG v(y)
VAL=0.9  FALL=1
.meas tran cur          integ v(v2#branch) from = 0 to = 9n
```

II. Inverter Fall Template

```
Vdd vdd gnd 1.8
Vin a gnd PULSE(1.8 0 5n 16.67p 0.1n 10n 20n)
Cload net2 gnd __LOAD__

V1 y net2 0
V2 net1 gnd 0
V3 vdd net3 0

Xinv a net1 net1 net3 net3 y __CELL_NAME__

.tran 0.001n 40n

.meas tran fall_tran   TRIG v(y) VAL=0.36 RISE=1 TARG v(y)
VAL=1.44 RISE=1
.meas tran tpdf      TRIG v(a) VAL=0.9  FALL=1 TARG v(y) VAL=0.9
RISE=1
.meas tran curf           integ v(v3#branch) from = 0 to = 9n
```

III. Buffer Fall Template

```
Vdd vdd gnd 1.8
Vin a gnd PULSE(0 1.8 5n 16.67p 0.1n 10n 20n)
Cload net2 gnd __LOAD__

V1 y net2 0
V2 net1 gnd 0
V3 vdd net3 0

Xbuf a net1 net1 net3 net3 y __CELL_NAME__

.tran 0.001n 40n

.meas tran rise_tran    TRIG v(y) VAL=0.36 RISE=1 TARG v(y)
VAL=1.44 RISE=1
.meas tran tpdr          TRIG v(a) VAL=0.9  RISE=1 TARG v(y)
VAL=0.9  RISE=1
.meas tran cur         integ v(v3#branch) from = 0 to = 9n
```

IV. Buffer Rise Template

```
Vdd vdd gnd 1.8
Vin a gnd PULSE(1.8 0 5n 16.67p 0.1n 10n 20n)
Cload net2 gnd __LOAD__

V1 y net2 0
V2 net1 gnd 0
V3 vdd net3 0

Xbuf a net1 net1 net3 net3 y __CELL_NAME__

.tran 0.001n 40n

.meas tran fall_tran   TRIG v(y) VAL=1.44 FALL=1 TARG v(y)
VAL=0.36 FALL=1
.meas tran tpdf      TRIG v(a) VAL=0.9  FALL=1 TARG v(y) VAL=0.9
FALL=1
.meas tran curf       integ v(v2#branch) from = 0 to = 9n
```

V. NAND Fall Template

```
Vdd vdd gnd 1.8

Va A gnd PULSE(1.8 0 5n 16.67p 0.1n 10n 20n)
Vb B gnd DC 1.8

V1 y net2 0
V2 net1 gnd 0
V3 vdd net3 0

Cload net2 gnd __LOAD__

X1 A B net1 net1 net3 net3 Y __CELL_NAME__

.tran 0.001n 40n

.meas tran rise_tran   TRIG v(y) VAL=0.36 RISE=1 TARG v(y)
VAL=1.44 RISE=1
.meas tran tpdr      TRIG v(a) VAL=0.9  FALL=1 TARG v(y) VAL=0.9
RISE=1
.meas tran cur            integ v(v3#branch) from = 0 to = 9n
```

VI. NAND Rise Template

```
Vdd vdd gnd 1.8

Va A gnd PULSE(0 1.8 5n 16.67p 0.1n 10n 20n)
Vb B gnd DC 1.8

V1 y net2 0
V2 net1 gnd 0
V3 vdd net3 0

Cload net2 gnd __LOAD__

X1 A B net1 net1 net3 net3 Y __CELL_NAME__


.tran 0.001n 40n

.meas tran fall_tran    TRIG v(y) VAL=1.44 FALL=1 TARG v(y)
VAL=0.36 FALL=1
.meas tran tpdf      TRIG v(a) VAL=0.9  RISE=1 TARG v(y) VAL=0.9
FALL=1
.meas tran curf          integ v(v2#branch) from = 0 to = 9n
```

VII. NOR Fall Template

```
Vdd vdd gnd 1.8

* NOR Gate Inputs
Va A gnd PULSE(1.8 0 5n 16.67p 0.1n 10n 20n)
Vb B gnd DC 0

V1 y net2 0
V2 net1 gnd 0
V3 vdd net3 0

* Load
Cload net2 gnd __LOAD__

X1 A B net1 net1 net3 net3 Y __CELL_NAME__

.tran 0.001n 40n

.meas tran fall_tran     TRIG v(y) VAL=0.36 RISE=1 TARG v(y)
VAL=1.44 RISE=1
.meas tran tpdf          TRIG v(a) VAL=0.9  FALL=1 TARG v(y)
VAL=0.9  RISE=1
.meas tran curf          integ v(v3#branch) from = 0 to = 9n
```

VIII. NOR Rise Template

```
Vvdd vdd 0 1.8

Va A gnd PULSE(0 1.8 5n 16.67p 0.1n 10n 20n)
Vb B gnd DC 0

V1 y net2 0
V2 net1 gnd 0
V3 vdd net3 0

Cload net2 gnd __LOAD__

X1 A B net1 net1 net3 net3 Y __CELL_NAME__


.tran 0.001n 40n

.meas tran rise_tran    TRIG v(y) VAL=1.44 FALL=1 TARG v(y)
VAL=0.36 FALL=1
.meas tran tpdr      TRIG v(a) VAL=0.9  RISE=1 TARG v(y) VAL=0.9
FALL=1
.meas tran cur          integ v(v2#branch) from = 0 to = 9n
```