CALIFORNIA STATE UNIVERSITY, FRESNO

LYLES COLLEGE OF ENGINEERING

Dept of Electrical and Computer Engineering

SPRING 2025

ECE 298 PROJECT TITLE

COARSE-GRAINED VS. FINE-GRAINED POWER GATING TECHNIQUES FOR RISC-V PROCESSOR

SUBMITTED BY: **Dinesh Munnangi**

ADVISOR: **Dr. Aaron Stillmaker**

THE ECE 298 FINAL PROJECT DISSERTATION IS SUBMITTED TO LYLES COLLEGE OF ENGINEERING IN THE FULFILMENT OF REQUIREMENT FOR THE AWARD OF MASTER OF SCIENCE (MS) DEGREE IN ELECTRICAL AND COMPUTER ENGINEERING.

APPROVED

For the Department of Electrical and Computer Engineering:

We, the undersigned, certify that the project of the following student meets the required standards of scholarship, format, and style of the university and the student's graduate degree program for the awarding of the master's degree.

Dinesh Reddy Munnangi
Project Author

Aaron Stillmaker Electrical and Computer Engineering

Reza Raeisi (Chair) Electrical and Computer Engineering

Zoulikha Mouffak Electrical and Computer Engineering

For the University Graduate Committee:

Dean, Division of Graduate Studies

AUTHORIZATION FOR REPRODUCTION

OF MASTER'S PROJECT

Yes — I grant permission for the reproduction of this project in part or in its entirety without further authorization from me, on the condition that the person or agency requesting reproduction absorbs the cost and provides proper acknowledgment of authorship.

_____ — Permission to reproduce this project in part or in its entirety must be obtained from me.

Signature of project author: Dinesh Reddy Munnangi

ABSTRACT

COARSE-GRAINED VS. FINE-GRAINED POWER GATING TECHNIQUES FOR RISC-V PROCESSOR

Moore's law predicts the continuous increase of transistors on an integrated circuit, while the transistor size decreases to allow this. Power consumption is one of the limitations that impacts the growth rate and transistor scaling. Techniques like power gating enhance circuit efficiency and reduce overall power usage. The power gating method will reduce power consumption by switching off the modules that are not needed at a particular time (low-power mode) and switching them on whenever they are needed (active mode). Power gating is a process of placing sleep transistors in the gated modules. The two types of power gating are fine-grained, where sleep transistors are placed in smaller modules and coarse-grained, where sleep transistors are placed in an entire large module. This project aims to evaluate and compare the impact of fine-grained and coarse-grained power gating techniques on power consumption, performance, and area to provide insights into the best power gating strategy for energy-efficient Reduced Instruction Set Computer (RISC-V) processors. The processor was compiled and simulated with ModelSim, synthesized using Synopsys Design Compiler, and physically designed with Cadence Innovus. The Nangate Open Cell Library was utilized to generate the netlist. This analysis examines which power gating method is suitable for pipelined architecture.

Keywords: Sleep transistors, fine-grained, coarse-grained, RISC-V, pipeline, Nangate Open Cell Library.

Dinesh Reddy Munnangi
May 2025

ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to Dr. Aaron Stillmaker for his constant support, guidance, and encouragement throughout the course of this project. From helping me choose a meaningful and challenging project topic to assisting me at every step, his involvement was truly invaluable. He generously provided access to all the resources he had and took the time to guide me through both technical and academic aspects. His belief in my abilities gave me the confidence to complete this project successfully.

I would also like to thank Dr. Reza Raeisi, our department chair, for stepping in as my official project advisor during Dr. Stillmaker's sabbatical leave. His willingness to support me during this time ensured that I could continue my work smoothly and without interruption.

My sincere thanks go to Dr. Zoulikha Mouffak, our graduate advisor and professor, for her continuous encouragement and motivation. She provided valuable academic guidance and was always there to support us and push us to stay on track.

I am also grateful to my friend Manasa, who assisted me with some tasks during the project. Her help made a difference, especially during the more challenging parts of the process.

Lastly, I would like to express my appreciation to California State University, Fresno, and the Lyles College of Engineering for providing a supportive academic environment, access to necessary resources, and the opportunity to pursue this project. The experience was incredibly rewarding and an important part of my academic journey.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

Page

LIST OF ABBREVIATIONS

Acronym	Definition
ALU	Arithmetic Logic Unit
CCS	Composite Current Source (Library Format by Synopsys)
CMOS	Complementary Metal-Oxide-Semiconductor
CPF	Common Power Format
CTS	Clock Tree Synthesis
DB	Database
DC	Design Compiler
DRC	Design Rule Check
DVFS	Dynamic Voltage and Frequency Scaling
DVS	Dynamic Voltage Scaling
ECSM	Effective Current Source Model (by Cadence)
EDA	Electronic Design Automation
FPGA	Field-Programmable Gate Array
GDSII	Graphic Data System II
GUI	Graphical User Interface
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
IO	Input/Output
LEF	Library Exchange Format
MMMC	Multi-Mode Multi-Corner
MOS	Metal-Oxide-Semiconductor
MTCMOS	Multi-Threshold CMOS
NLDM	Non-Linear Delay Model
NLPM	Non-Linear Power Model
NMOS	N-type Metal-Oxide-Semiconductor
PC	Program Counter
PIPO	Parallel In Parallel Out

Acronym	Definition
PMOS	P-type Metal-Oxide-Semiconductor
PNR	Place and Route
PTM	Predictive Technology Models
RISC-V	Reduced Instruction Set Computer – Fifth generation
RTL	Register Transfer Level
RV32I	RISC-V 32-bit Integer Instruction Set
SDC	Synopsys Design Constraints
SRAM	Static Random Access Memory
TCL	Tool Command Language
UPF	Unified Power Format
VCD	Value Change Dump
VDD	Voltage at the Drain (Positive Power Supply)
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
VSS	Voltage at the Source (Ground)

CHAPTER 1: INTRODUCTION

Power consumption is important in today's electronic devices. Every gadget, such as phones, laptops, or even small sensors, must carefully use energy so that batteries last longer and devices do not overheat. People always want gadgets that are smaller, lighter, faster, and more powerful. To achieve this, engineers must find ways to reduce power usage without lowering device performance. Most of the chip designs have an important design goal as a power budget. If power consumption is more than the design goal, it can be fatal to the design [1]. It increases the overall budget as designers must use high-grade materials, or the designs can be less dependable due to their high-power consumption. With the size of transistors decreasing, more transistors are placed per unit area [2]. The functionality of devices has improved, but it has resulted in high power consumption and leakage. To overcome these types of problems, novel methods that are efficient and sustainable are needed.

Power gating reduces power consumption by turning off parts that are not being used, saving energy and ensuring devices work better for longer periods. Instead of letting those parts waste power, power gating switches them off completely. The RISC-V processor is quicker because it has limited addressing modes and instruction length is fixed [3], this technique can make an enormous difference. In this project, two types of power gating are used. One is coarse-grained and the other is fine-grained; both designs are observed to see how they work for a 5-stage pipelined RISC-V processor to save power.

RISC-V is an open design that anyone can use and change [3]. This makes it perfect for things like small gadgets, smart home devices, or even powerful computers. RISC-V processors start leaking power mainly due to static power dissipation in CMOS transistors, even when the processor is idle. This leakage arises from subthreshold leakage (current flowing between source and drain when the transistor is OFF, but gate voltage is not zero),

gate oxide leakage (electron tunneling through the thin oxide layer), and junction leakage [4]. These effects become more pronounced as transistor sizes shrink in modern technology nodes. All CMOS-based RISC-V designs face these issues. Power gating shuts down those idle parts, so the processor uses less energy overall. This technique is often used in devices that need to run on low power [5]. Without power gating, devices might not last long enough or leak more power [4]. Several low-power digital circuit design techniques can be applied to RISC-V architectures, including sub-threshold circuit design, pipelining, adiabatic logic, asynchronous logic design, clock gating, power gating, and Dynamic Voltage and Frequency Scaling (DVFS) [5]. These techniques help optimize power efficiency while maintaining the flexibility and openness of the RISC-V architecture [5].

The RISC-V 32-bit Integer Base (RV32I) instruction set is used in this design. This has forty-seven instructions that perform basic arithmetic, logical, branching, or jumping operations. A 5-stage pipelined processor splits a task into five steps to make it faster, like an assembly line [4]. These stages are:

1. Fetch: Retrieve the next instruction from memory.
2. Decode: Determine the operation and extract operands.
3. Execute: Perform the operation (e.g., addition, comparison).
4. Memory: Access data memory for load or store instructions.
5. Writeback: Write the result back to the destination register.

Power gating can be applied at various pipeline stages when all units are inactive simultaneously. In fine-grained power gating, specific modules such as the Arithmetic Logical Unit (ALU) or decoder can be turned off if the current instruction only moves data and does not require those components. Coarse-grained power gating may disable an entire stage, such as the memory stage, if no data access is needed. RISC-V architecture supports

power gating flexibility, and designers can selectively gate stages based on workload demands to save energy with minimal performance impact [7]. Since RISC-V is built to be adaptable, power gating makes energy savings for each application, making these devices more efficient and practical [8].

Power gating can significantly reduce energy consumption in real-world applications using RISC-V processors. For instance, in a smart thermostat, which typically remains idle while monitoring temperature, coarse-grained power gating can shut down large sections like the processor core during inactive periods, extending battery life for months without recharging. In a drone that uses a RISC-V chip for camera-based navigation, fine-grained gating can disable unused blocks such as the math unit or memory stage while hovering, improving energy efficiency and flight duration. Similarly, in a wearable fitness tracker, power gating inactive blocks like the display driver when the screen is off helps maximize battery life on compact power sources.

1.1 Motivation for the Project

Rather than power gating on a multi-core processor by switching one of the cores when it is not used, this project uses a RISC-V processor with a single core. The objective is to determine which components within the pipelined architecture can be gated to reduce power consumption without disturbing the pipelined architecture, as it might cause some issues with instruction flow. Therefore, power gates are placed in such a way that the flow is not disturbed, to compare and find which power gating method is useful for these types of pipelined processors. This project compares gating a large block like the memory read and writeback stage and small blocks like the ALU, decoder, multiplier, adder, etc.

This study also considers the trade-off between area overhead and power savings. The aim is to identify an optimal balance where power reduction does not compromise

performance or increase complexity. Finally, the analysis provides insights into the practical implementation of fine-grained versus coarse-grained gating in real-world RISC-V systems.

This report is divided into the following sections:

Section 1: Introduction: This section outlines the project's objectives, purpose, and significance.

Section 2: Background: Provides a brief overview of existing research, concepts, and techniques relevant to this work.

Section 3: Tools and Libraries Used: Lists and describes software tools, technologies, and libraries utilized in the project.

Section 4: Methodology: Details the approach and procedures followed to achieve the project's objectives.

Section 5: Implementation: Explains the practical steps and processes used in developing and deploying the solution.

Section 6: Results and Analysis: Presents and discusses key findings, outcomes, and evaluations derived from the project.

Section 7: Conclusion and Future Work: Summarizes the project's achievements and suggests possible improvements and extensions.

CHAPTER 2: BACKGROUND

2.1 Power Gating

Power gating stops the supply (either VSS or VDD) to switch OFF the modules. Power gates can be added to the whole design or a specific region. The chip can be divided into two or more regions, which are the active region and the power-gated region. The active region is the region that is always ON. The power-gated region is the region that is OFF during the times when it is not needed and ON when it is needed. Power gating is the process of placing additional gates to the existing design in areas that are not needed all the time. Designers use PMOS transistors for the design from VDD to power-gated blocks, which are also called header cells. NMOS cells are placed between VSS and a power-gated block called footer cells. NMOS transistors have higher leakage, making header (PMOS) switches more efficient [4]. The figure below shows the header and footer cells.

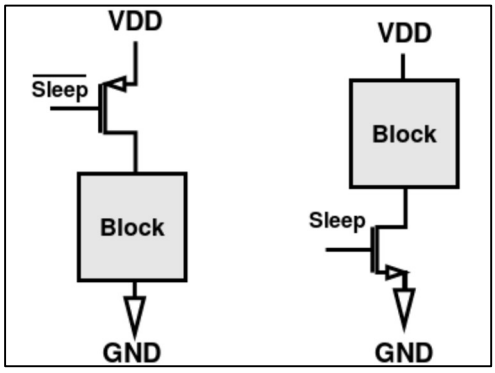

Figure 1: Header (left) and Footer (right) cells [4]

2.1.1 Working

Power gates or switches work in 2 modes; they are

1. Active mode: The power-gated region is connected to the supply through the power gates and works normally.

2. Sleep mode: The control signals make the power gates turn OFF. The supply is disconnected from the power-gated region, and the region goes to sleep mode.
3. Wake-up mode: The control signals make the power gates turn ON by connecting the supply, and the power-gated region goes to active mode again. Retention cells (flip-flops) are used here to avoid delay while turning ON the power gates region [9].

2.1.2 Types

1. Coarse-Grained Power Gating:

Coarse-grained power gating is a technique used to save energy in computer chips by turning off large sections at once. It is like having one main light switch for an entire floor of an office building; when no one is using the floor, you flip that single switch to turn everything off. In a chip, this means disconnecting the power supply from major blocks like a whole processor core, the graphics processing unit, or large memory areas when they are completely inactive. A large group of sleep transistors acts as the main switch for the entire block [8].

The main benefit of this approach is its simplicity. It is easier for engineers to design and manage because there are fewer modules, and the decision-making process is straightforward. This method also tends to take up less extra space on the chip relative to the size of the block it controls and usually does not slow down the block much when it is powered on. However, its major downside is inflexibility; power can only be saved if the entire large block is idle. If even a small part needs to remain active, the whole block stays powered on, missing opportunities to save energy in the portions that are idle [2].

2. Fine-Grained Power Gating:

Fine-grained power gating takes the opposite approach, focusing on turning off very small pieces of the chip individually. This is like having separate switches for every

single desk lamp or small group of lights within an office. On a chip, this means placing many tiny sleep switches around small circuits, like specific calculation units or even smaller logic groups. Whenever one of these tiny parts is not needed, even for a fraction of a second, its individual switch can cut its power, even if neighboring circuits are busy [9].

This method offers many more opportunities to save power because it can target specific idle spots very precisely, potentially leading to greater overall energy reduction if managed effectively. The main drawback is its complexity. Designing and controlling thousands of tiny switches and the logic to decide when to activate them is extremely challenging and takes up significant space on the chip. Furthermore, these tiny switches can sometimes slightly slow down the chip's operation when they are on, and designers must be careful that the energy saved by switching off a tiny part is more than the energy cost of flipping the switch itself (it needs to be off longer than its break-even point).

2.1.3 Challenges

Power gating is simply turning off the circuit, but some issues need to be taken care of. Even if modules are power-gated, there is some power leakage due to the usage of MOS transistors, as they are not idle [10].

1. Physical Design and Verification Processes

Power gates must be strategically placed in the layout to minimize delay and maintain even power distribution during switching events. This can lead to routing congestion and layout complexity. The verification process is also more demanding, as it must account for multiple power states, transitions, and corner cases. Ensuring signal integrity and managing IR drop during power transitions require advanced tools and thorough testing methodologies. Power gating is effective in reducing leakage power, it requires tight coordination across design, physical implementation, and verification phases.

2. Power Controller

Power gating saves energy by completely shutting down sections of a chip when they are idle. However, these transitions introduce latency. If a section is powered down when it is needed shortly, or if wake-up takes too long, overall system performance can suffer [10]. To manage this, a dedicated power controller is used. This control unit determines when to safely power down idle blocks and when to reactivate them efficiently, balancing energy savings with performance requirements.

2.2 Power Gating Implementation Styles

2.2.1 Ring Style

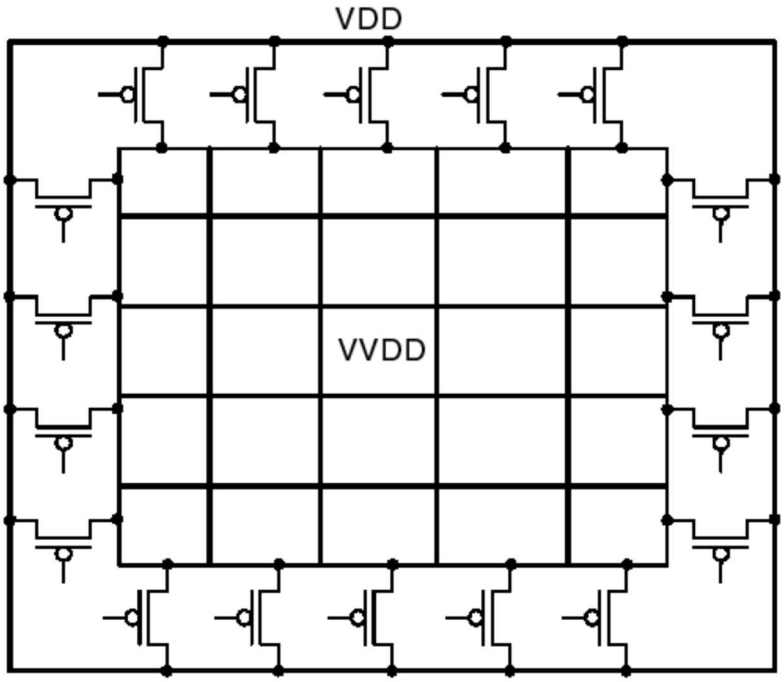

Figure 2: Ring style power gating [14]

The ring style of power distribution for power gating involves creating a continuous power (VDD) ring around the perimeter of the block that needs to be switched ON or OFF [14]. This approach offers simplicity in power network planning and has little

negative impact on the automated Place and Route (PNR) stage of chip design while also allowing always-on cells to be conveniently placed nearby. However, it has significant drawbacks, including the potential for large voltage (IR) drops within big power domains, a higher area cost compared to grid structures, and a crucial limitation in that it cannot support state retention registers for saving information during power down. Despite these issues, the ring style is often the only practical option when adding power gating capabilities to an existing, pre-designed hard block.

2.2.2 Column Style

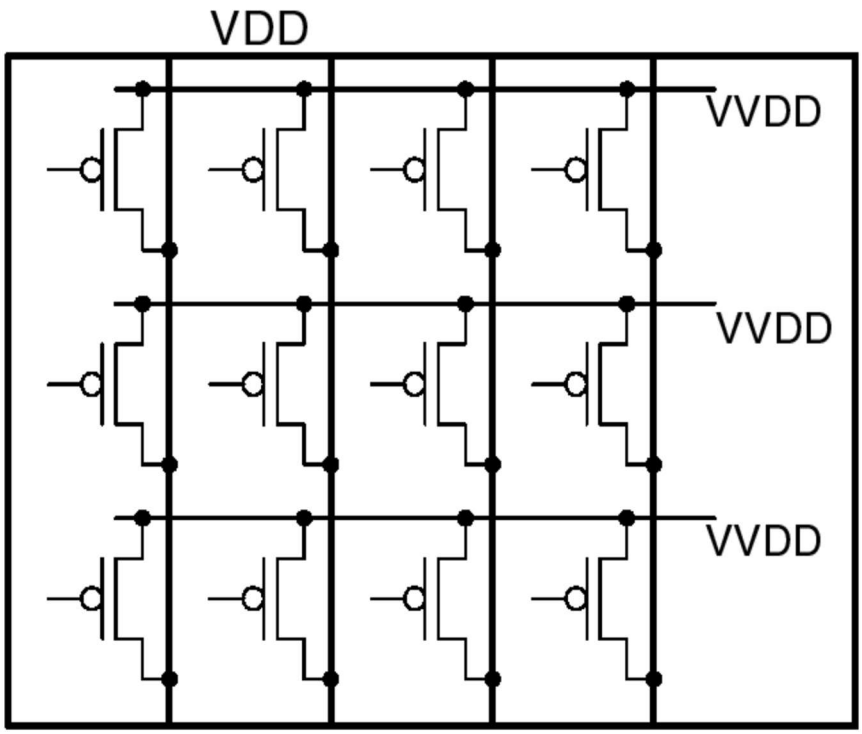

Figure 3: Column style power gating [14]

Alternatively, power gating can be implemented by distributing sleep transistors throughout the power-gated region itself, rather than just around the edge. This approach offers several benefits, including the flexibility to route the virtual power supply using

lower metal layers and requiring fewer sleep transistors compared to the ring style to achieve the same target voltage (IR) drop [14]. It also allows retention registers and always-on buffers to easily connect to the permanent power supply anywhere within the block, aids in managing in-rush current more effectively during wake-up and often leads to lower overall area overhead. The main disadvantages, however, are the increased complexity it introduces, negatively impacting standard cell routing and physical synthesis stages, and resulting in a much more complex power routing network overall.

2.3 Unified Power Format

UPF (Unified Power Format) is essentially a standard language or set of commands used by chip designers specifically to describe how power should be managed in their chip design. Its main purpose is to help designers focus on saving power right from the beginning of the design process. Using UPF, they can specify important power-saving details, like which parts of the chip can have their power turned off completely (power gating) or run at different voltage levels, alongside the regular design description [13]. This standard format is needed because traditional chip design languages were not built to handle these power details well, and relying on different methods from various tool suppliers caused problems with compatibility and potential errors. UPF provides a consistent way for different design tools to understand and implement the intended power-saving strategies, like power switching and clock gating.

UPF is the standard format used to define a chip's power intent; the specific plan for how power will be controlled to save energy. It is an official standard (IEEE 1801) that has been updated over time [15]. UPF uses commands based on the Tool Command Language (TCL) scripting language and works together with common chip design languages like Verilog and VHDL [14]. Within a UPF file, designers can specify important

details such as grouping circuits into power domains that share the same power supply, describing the network of power wires and switches, defining allowed ON/OFF states for different domains, and setting rules for how to protect active circuits from powered-off ones (isolation), how to save information when power is OFF (retention), and how to handle signals moving between areas with different voltages (level shifting) [14]. This complete power plan described in UPF is crucial because it is used by design tools for both checking that the power strategy works correctly (verification) and for building the physical chip with these power-saving features (implementation), directly affecting the chip's layout and wiring.

2.3.1 Elements in Unified Power Intent

1. Power Domain:

Power domain is simply a way to group different parts or instances within a chip design that share the same power supply needs or will be controlled together for power management purposes. Think of it as drawing a boundary around a specific section of the chip to manage its power collectively.

2. Power Supply Network:

This describes the wiring diagram for electrical power within the chip design. It details how the different power supplies (like VDD) and ground connections are distributed to various parts, potentially including power switches that can turn supplies on or off for certain areas.

3. Power States and Transitions:

This defines the different operational modes related to power that a section of the chip (a power domain) or its power supply can be in, such as Run, Sleep, Retention, or

OFF. It also specifies the rules for how the chip is allowed to move from one power state to another.

4. Power Switching Strategies:

This refers to the technique of using special power switch cells, often controlled by specific signals, to completely turn off the power supply going to a particular power domain when it is not needed, primarily to save leakage power. UPF specifies the logical connection of these switches, which signals control them and which supplies they switch, without dictating the exact physical implementation, like coarse-grained or fine-grained switches.

2.4 Related Work

Agnes et al. investigated the integration of clock gating and power gating techniques to address both dynamic and leakage power consumption in digital CMOS circuits [12]. Recognizing that clock networks contribute significantly to dynamic power and that leakage power is a major concern, they explore combining these methods, potentially using clock gating information to drive power gating controls. They analyze different approaches, including AND/FF-based clock gating, and LECTOR/GALEOR power gating, implemented on a parallel in parallel out (PIPO) shift register using 90nm technology [12]. While demonstrating power reductions through this combined approach, the work also acknowledges the practical challenges involved in efficiently integrating the necessary control logic and managing the power and timing overheads associated with using both techniques simultaneously.

Hemant et al. focused on reducing power consumption in standard 6T SRAM memory cells, which are common in portable devices but use considerable power [16]. They specifically investigated the power gating technique, implementing it directly and

exploring variations using header switches and footer switches within the SRAM cell design. Using simulations on a 16nm technology model, they aimed to lower the average power use, particularly leakage power, which increases as technology shrinks, while also ensuring the memory cell remained stable [16]. Their results indicated that the power-gated SRAM cell design performed better than the conventional 6T SRAM and the header/footer variations, achieving significant reductions in average power (around 15-18% less than the variations) and improvements in speed, while also maintaining good stability.

Power gating is a prominent technique for reducing leakage power, particularly significant in deep sub-micron technologies, by deactivating idle circuit blocks via sleep transistors [17, 18]. Various approaches exist, including Multi-Threshold CMOS (MTCMOS), tri-mode gating (which offers a state retention option), using multiple sleep modes via different biasing voltages, and hybrid methods combining several strategies [17, 19]. Implementation aspects such as gate clustering for switch sizing, wakeup scheduling to minimize turn-on delay, and layout-aware methodologies like row-based power gating are also important considerations discussed in the literature [17]. Additionally, specialized techniques have been developed for specific applications, such as dynamically controlled or autonomous gating for FPGAs and handshake-based methods for asynchronous circuits [19].

While effective for leakage reduction, power gating introduces several costs: area overhead due to sleep transistors and potentially required decoupling capacitors (decaps) to manage noise, possible performance degradation from voltage drops across switches, increased dynamic power, and noise generated during power state transitions [18, 20]. Studies, such as [18], show that significant net leakage savings can be achieved, especially as technology scales down, provided these overheads are carefully managed through design choices like sleep transistor sizing and decap allocation [18]. To aid this analysis, figures

of merit have been proposed in [20] to quantify the trade-offs, including metrics like allowable performance degradation, sleep transistor size, leakage savings, power mode transition time, and transition energy overhead [20].

The implementation of power gating generally falls into fine-grained or coarse-grained categories [19, 21]. Fine-grained involves integrating sleep transistors within individual logic cells, which can lead to high area overhead but simplifies control [19, 21]. Coarse-grained uses shared sleep transistors for clusters of cells, often arranged in a ring or grid structure; this typically reduces area overhead but necessitates careful design of the virtual power distribution network and can introduce different timing challenges [19, 21]. One approach proposed in [21] uses a fake-via model during power network synthesis for coarse-grained gating to optimize the placement and number of sleep transistors concurrently with the power grid, achieving substantial reductions in the required sleep transistor area [21].

CHAPTER 3: TOOLS AND LIBRARIES USED

3.1 ModelSim Intel® FPGAs Standard Edition Software Version 20.1.1

Intel ModelSim is a computer program that acts like a virtual test lab for your electronic designs, specifically for special chips called Intel field-programmable gate arrays (FPGA). Before programming the design onto the physical FPGA chip, designers use ModelSim to run the design on the computer. This allows checking if the design works the way expected and helps to find any mistakes or bugs. It works together with the main Intel design software, Quartus Prime. Using ModelSim helps in fixing problems early, which saves time and makes it more likely that the final FPGA chip will work correctly. It supports popular HDLs like VHDL, Verilog, and SystemC, allowing engineers to describe and model digital systems at different levels of abstraction [23].

3.2 Synopsys Design Compiler® Version S-2021.06-SP5-1

Synopsys Design Compiler is a leading software tool used for RTL synthesis, which means it translates high-level hardware descriptions into optimized gate-level designs. Its main advantage is its ability to simultaneously optimize the design for multiple crucial goals: fast performance (timing), small chip size (area), low energy use (power), and making the chip easy to test [24]. Design Compiler features innovative topographical technology that accurately predicts the final timing and area before the physical layout stage, helping designers avoid lengthy iterations and achieve results faster. It is designed to run efficiently on modern multi-core computers and is a central part of the broader Synopsys design flow, working alongside other tools. An advanced version, Design Compiler NXT, offers further improvements like predicting and easing wiring congestion and providing physical guidance to layout tools for even better results [25].

3.3 Cadence® Innovus™ Implementation System Version v20.10-p004_1

Cadence Innovus Implementation System is a key Electronic Design Automation (EDA) tool used in the creation of digital integrated circuits (ICs) [26]. Its main job is physical implementation, which involves taking a synthesized logic design (the output from tools like Design Compiler or Cadence Genus) and turning it into a detailed physical layout [26, 27]. This process includes placement (deciding the exact location of all the standard cells and larger blocks on the chip) and routing (drawing the millions of tiny wires needed to connect everything). Innovus is designed to handle very large and complex chip designs, focusing on achieving critical goals like meeting performance speed targets (timing closure), minimizing power consumption, and optimizing the chip's area, all while aiming for a faster overall design completion time.

3.4 Nangate 45nm Open Cell Library

The Nangate 45nm OpenCell Library is a widely recognized open-source standard-cell library developed by Nangate and made available through organizations like Si2.org [29]. Its primary purpose is not for actual chip fabrication but rather to support academic research, educational use, and the testing and development of Electronic Design Automation (EDA) tools and design methodologies. This library is based on the 45nm FreePDK Base Kit from North Carolina State University (NCSU) and utilizes Predictive Technology Models (PTM) from Arizona State University (ASU) for its characterization. It serves as a common, non-proprietary platform for experimenting with various digital design flows.

The library offers a comprehensive set of standard cells, encompassing around thirty-eight distinct logic functions in its initial releases, ranging from simple buffers and

inverters to more complex elements like scan flip-flops with set/reset capabilities. To cater to different performance requirements, most cell functions are provided in multiple drive strength variants, resulting in over one hundred unique cells in total. To support various stages of the design process, the library includes a full suite of design views: physical layout information in Library Exchange Format (LEF) and Graphic Data System II (GDSII) formats, simulation models in Verilog, VITAL (VHDL), and Spice (both pre-extraction and post-extraction), timing and power information in Liberty (.lib) format, schematics, and Open Access databases [30].

Regarding timing and power characterization, the Nangate 45nm library provides detailed data within its Liberty (.lib) files, supporting several industry-standard models. Specifically, it includes characterization data for the Non-Linear Delay Model (NLDM) and the corresponding Non-Linear Power Model (NLPM). Furthermore, it offers advanced timing models like the Composite Current Source (CCS) Timing and the Effective Current Source Model (ECSM) Timing views. This allows for more accurate timing analysis in modern EDA tools. Characterization data is typically provided across multiple process corners, such as slow, typical, and fast, enabling analysis under different operating conditions.

To facilitate the exploration of modern low-power design techniques, later versions of the Nangate 45nm OpenCell Library were updated to include a specific set of low-power cells. This collection includes key components necessary for implementing strategies like power gating and multi-voltage designs. Among the included low-power cells are always-on cells (designed to remain powered even when their surrounding domain is off), isolation cells (used to manage signals crossing between powered-on and powered-off domains), level-shifter cells (for interfacing between domains operating at different voltages), and

power-switch cells. The inclusion of these cells makes the library suitable for experimenting with and verifying low-power design flows.

CHAPTER 4: METHODOLOGY

4.1 RISC-V Processor

A RISC-V processor has been designed with a 5-stage pipeline architecture. Pipelining is a technique used in processor design to improve instruction throughput; the number of instructions completed per unit of time. It works by overlapping the execution of multiple instructions, breaking down the processing of a single instruction into distinct steps or stages [3]. Each stage manages one part of the instruction, and ideally, each stage completes its work in one clock cycle. The classic 5-stage RISC pipeline, commonly used for RISC-V, consists of the following stages:

4.1.1 Instruction Fetch (IF)

- The primary goal of this stage is to retrieve the next instructions to be executed from memory.
- The Program Counter (PC) register holds the memory address of the current instruction. This address is sent to the instruction memory (or instruction cache).
- The instructions located at that address are fetched from memory.
- Simultaneously, the PC is updated to point to the next instruction. For most instructions, this involves simply incrementing the PC by 4 (since standard RISC-V instructions are 32 bits or 4 bytes). However, if the current instruction is a taken branch or a jump, the PC will be updated with the target address calculated later in the pipeline [6].

4.1.2 Instruction Decode (ID)

- Once an instruction is fetched, it needs to be interpreted or decoded.

- This stage identifies the type of operation the instruction represents (based on its opcode) and determines which registers hold the source operands needed for the operation.
- The required operand values are read from the processor's register file [4].
- Control signals required for the subsequent stages (Execute, Memory, Writeback) are generated based on the decoded instruction. This stage might also include initial logic for detecting potential hazards (dependencies between instructions).

4.1.3 Execute (EX)

- This stage performs the actual computation or operation specified by the instruction.
- For arithmetic or logical instructions (like ADD, SUB, AND, OR), the Arithmetic Logic Unit (ALU) takes the operand values read during the ID stage and performs the calculation.
- For load or store instructions, the ALU typically calculates the effective memory address by adding an offset to a base register value.
- For branch instructions, the ALU might be used to compare register values to evaluate the branch condition, and the target address for the branch is calculated.

4.1.4 Memory Access (MEM)

- This stage handles interactions with the data memory.
- If the instruction is a load, the memory address calculated in the EX stage is used to read data from the data memory (or data cache).
- If the instruction is a store, the data (read from a register during the ID stage) is written to the data memory at the address calculated in the EX stage.
- For instructions that do not involve memory access (like arithmetic/logic operations or branches), this stage might perform no significant action related to data memory.

4.1.5 Write Back (WB)

- This is the final stage where the result of the instruction's execution is written back into the processor's register file.
- For arithmetic/logic instructions, the result comes from the ALU calculation performed in the EX stage.
- For load instructions, the result is the data fetched from memory during the MEM stage.
- This written value can then be used as an operand by subsequent instructions.

By overlapping these stages, the processor can potentially start a new instruction every clock cycle, even though each instruction takes 5 cycles to complete its journey through the pipeline. This significantly increases the overall instruction execution rate compared to a non-pipelined processor, where one instruction must finish before the next begins. However, pipelining introduces complexities like hazards (data hazards, control hazards, structural hazards) that occur when instructions interfere with each other. These require additional hardware mechanisms like forwarding, stalling (inserting bubbles), and branch prediction to manage efficiently.

This design has sub-modules like ALU, controller, coprocessor, decoder, instMemory (instruction memory), memory, PC (program counter), register, and stall. For the five stages, sub-modules are IFID (Instruction Fetch and Instruction Decode), IDIE (Instruction Decode and Instruction Execute), IEME (Instruction Execute and Memory Enable), and MEWB (Memory Enable and Write Back). The top module is the processor. This processor supports RV32I instructions.

The figure below shows the architecture of the RISC-V processor with a 5-stage pipeline. The design is based on this architecture with all the modules connected as shown.

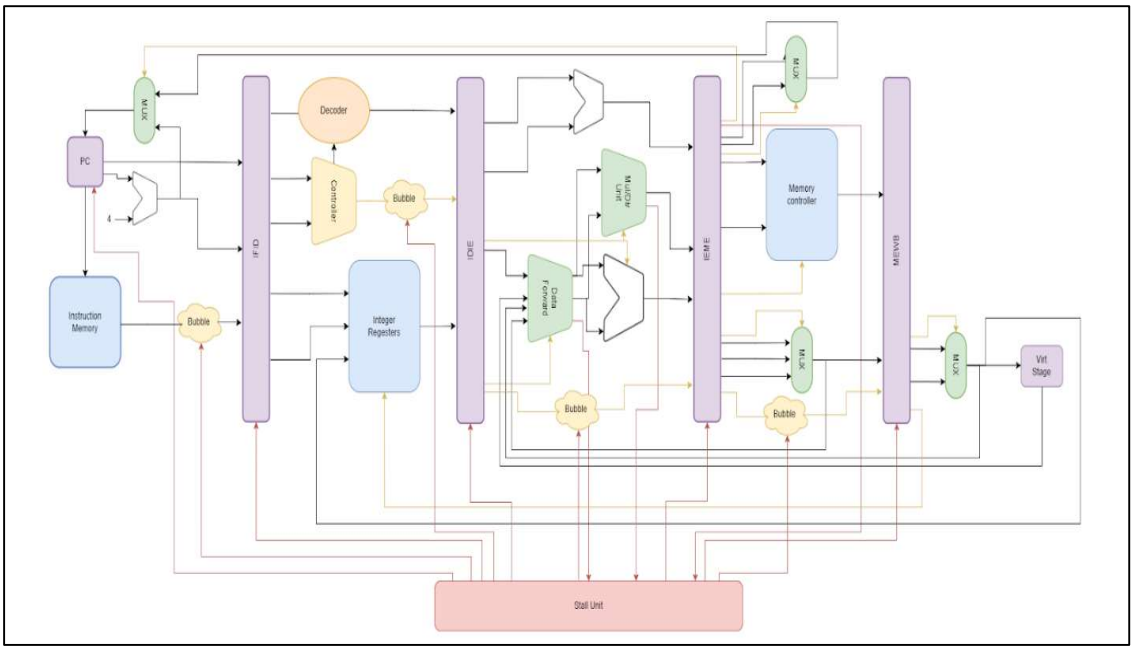

Figure 4: RISC-V processor architecture

4.2 Number of Power Gates

The number of power gates that should be added to the design is determined by the equation-(1) [8]. In this equation, R_{ON} signifies the resistance of the power gate when it is active (R_{ON}). I_{Domain} denotes the current drawn by the circuit section (domain) being controlled by the power gate. Furthermore, the drop target represents the maximum permissible voltage drop (IR drop) across the gate, which is capped at 3% of the supply voltage.

$$\text{Number of Switches} = \frac{(R_{ON} * I_{Domain})}{\text{Drop Target}} \text{----------(1)}$$

4.3 Project Flow

This flowchart outlines the digital design process for creating a power-gated RISC-V processor, starting from the initial Verilog hardware description (RTL). The flow proceeds through simulation/verification. After that synthesis, the power-saving strategy defined in a UPF file is incorporated with the design logic. Finally, it shows the physical implementation stage, where power gates are physically inserted according to the UPF specification, resulting in the final chip layout (GDSII). This flowchart shows the whole process, but deep down each individual step is a long process that is shown in the next chapter.

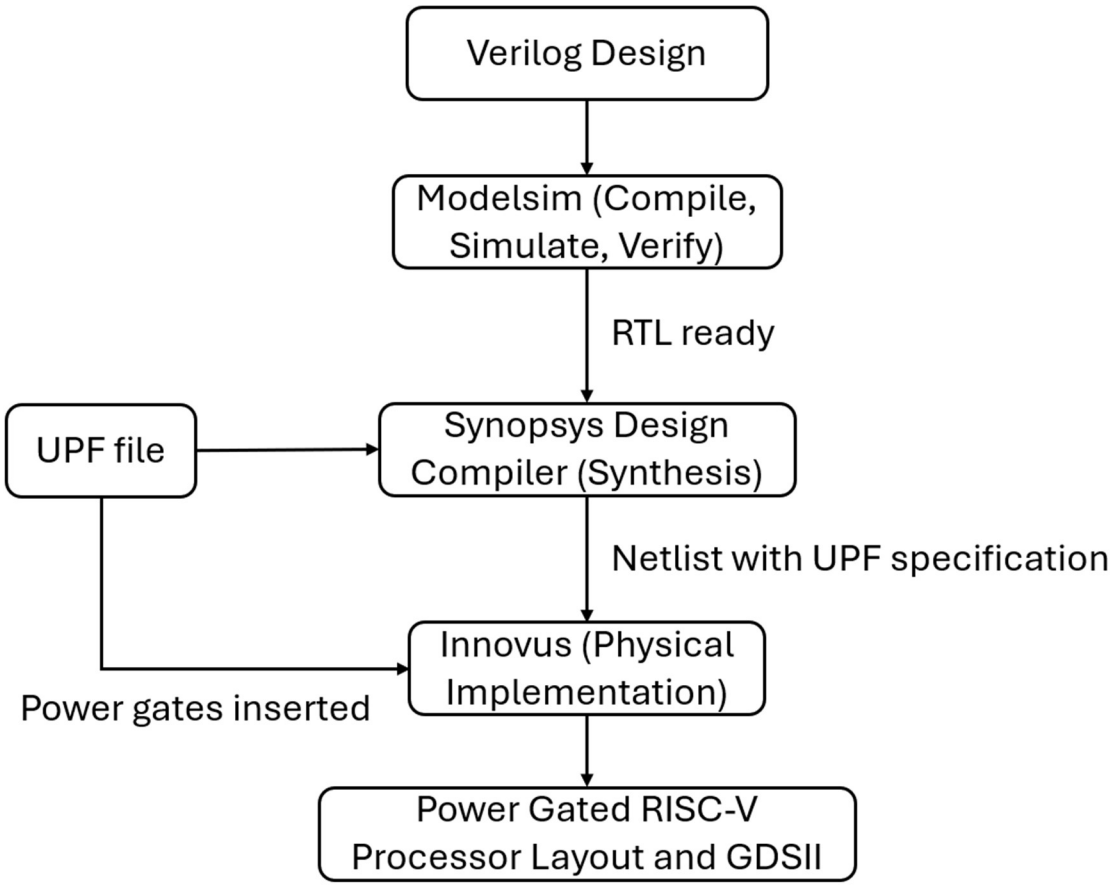

Figure 5: Flowchart for Design Power-Gated Processor

While defining the UPF file, 2 different files have been designed for coarse-grained and fine-grained with different specifications. And then both have the same steps for the rest of the process.

CHAPTER 5: IMPLEMENTATION

5.1 Verilog Files

The design process began with the goal of implementing a modular, scalable, and power-aware 32-bit RISC-V processor architecture based on the RV32I instruction set. A pipelined approach was selected to enhance performance by allowing multiple instructions to be processed simultaneously across different stages.

The top-level file, processor.v, connects the main pipeline stages-IFID.v, IDIE.v, IEME.v, and MEWB.v. These stages follow the standard RISC-V 5-stage pipeline with an optional sixth stage for additional processing in complex instructions. Design decisions such as separating pipeline registers and incorporating clear data/control paths were made to ease debugging, support stalling, and enable future enhancements like hazard handling and branch prediction.

There are some other files:

1. Arithmetic Logic Unit (ALU): ALU is a submodule that performs calculations and handles arithmetic instructions like addition, subtraction, etc., as well as logical operations like AND, OR, NOT, etc.
2. Controller: This module directs the operations; it fetches instructions and makes them execute.
3. Coprocessor: An extra processor unit for specialized tasks like mathematical calculations to speed up the processor.
4. Instruction Memory: This has all the instructions that the processor will fetch and execute.
5. Memory: This acts as a general-purpose memory to store the data.
6. Program Counter: A register that holds the next instruction address; the address is incremented automatically when the instruction is executed.

7. Stall: This is a temporary pause to execution flow whenever there is a hazard to the processor.
8. Stage 6: This is an extra stage for the processor; this is used when the task is too complex.

The process begins with the Instruction Fetch (IF) stage retrieving instructions using the program counter and forwarding them to the Instruction Decode (ID) stage via the IFID register. The ID stage decodes instructions, fetches operands from the register file, and passes all necessary control and data signals to the Execute (EX) stage through the IDIE register. The ALU in the EX stage performs arithmetic or logical operations, and its results are passed to the Memory (MEM) stage via the IEME register for data access. Finally, the results are passed to the Write Back (WB) stage through the MEWB register for updating the register file. This structured handoff between stages enables pipelined execution and allows multiple instructions to be processed concurrently. Potential improvements include optimization of the pipeline to minimize stalls, enhancing memory management, incorporating branch prediction algorithms, and increasing ALU efficiency for broader instruction support.

In the top-level Verilog module, the signal pwr_sig is introduced to control the power gating of specific functional blocks, such as the ALU. This signal is defined as an external input, that is driven by a power management unit. The pwr_sig is then passed as an input to the gated module, allowing it to simulate the behavior of being powered OFF by disabling internal logic operations. Although actual power gating is handled physically through power switches defined in the UPF file, the inclusion of pwr_sig in the RTL enables functional simulation and proper connectivity for synthesis and physical implementation. This setup ensures that the control signal is logically integrated and ready for power-intent mapping during implementation in Design Compiler and Innovus.

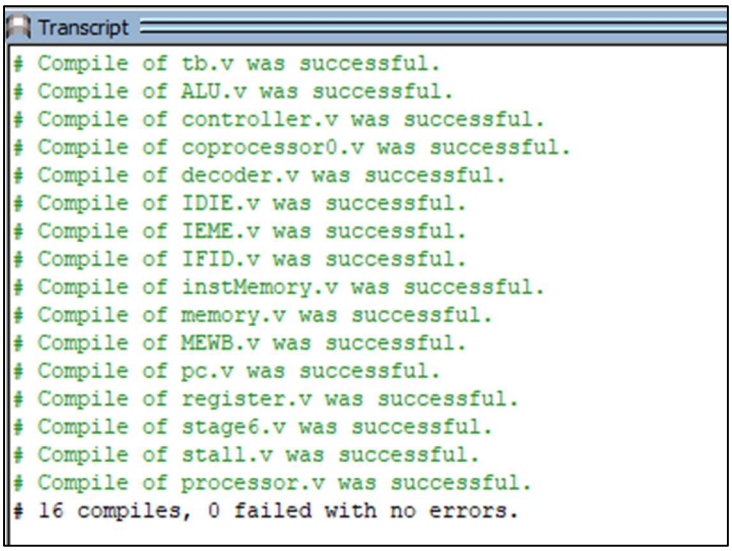

Figure 6: ModelSim Console after compilation

The modules are loaded into the ModelSim and compiled together. After compiling, the RTL is shown in the image below. Using a ModelSim student version does not provide the view RTL option. This RTL diagram is viewed in Intel® Quartus® Prime Standard Edition Design Software, Version 23.1.1.

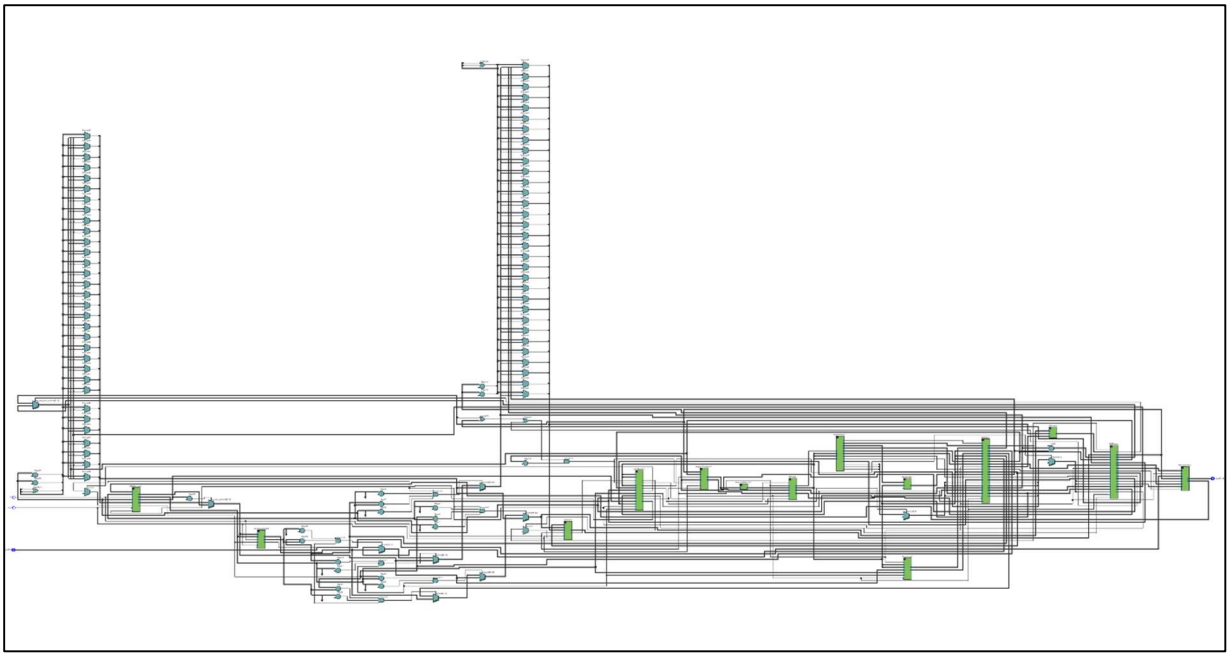

Figure 7: RTL schematic

5.2 UPF File

The Unified Power Format file has been created. This uses Tool Command Language (TCL). Two UPF files have been created; one is for coarse-grained, and the other is for fine-grained.

5.2.1 Power domain creation

It is created by adding some modules that need to be partitioned from the others. For the rest of the modules to be a power domain, the command used is -include_scope. This calls all the remaining modules without specifying each of them.

1. Coarse-Grained:

```
create_power_domain PD_CORE -elements {mewb1}
create_power_domain PD_TOP -include_scope
```

The mewb1 is the instance of MEWB module; this is a larger module separated from the other to power gate for the whole module.

2. Fine-Grained:

```
create_power_domain PD_CORE -elements {alu1 dec1 stall1
coprocessor1 stage6}
create_power_domain PD_TOP -include_scope
```

The alu1, dec1, stall1, coprocessor1, and stage6 are modules, which are additional and not required all the time. They are a bunch of small modules that will be gated.

5.2.2 Power switch creation

```
create_power_switch SW_CORE -domain PD_CORE
```

The power switch has been defined with the above command with all the input and output pins, etc. This has been mapped to the power switch header cells in the low-power

open cell library. Mapping HEADER_X1, X2, etc., will be used at the time of physical design to place the power switch cells directly.

5.2.3 Power state table

The create_pst command in UPF is used to define a Power State Table, specifying the list of power supply nets or ports that the table will describe. Its main purpose, used together with the add_pst_state command, is to define the combinations of states (like voltage levels or on/off status) that are considered valid or legal for those specific supplies to be in simultaneously during the chip's operation.

5.3 Synthesis

To synthesize a design using Synopsys Design Compiler (DC), clear timing and design rules are defined. Design compiler also uses TCL scripting. Using a setup file, dc.setup, all the required paths, libraries, packages, database files, etc., have been declared. The file configures the synthesis environment before running synthesis.

The Design Compiler (DC) template file (.tcl) is loaded into Synopsys Design Compiler, which typically includes commands to read the Verilog files, link the modules, and set constraints by sourcing the .sdc file. The file has information specifying critical parameters like clock periods, input/output delays, false paths, and exceptions. The template guides the synthesis flow through phases such as elaboration, optimization, timing analysis, and finally generating gate-level netlists. The DC template file simplifies synthesis by standardizing the setup, ensuring all constraints from the .sdc file are applied consistently, enabling efficient timing-driven optimization, and reducing potential design errors. This file has all the information related to the design name and synthesis

information. This file has information about which files users want to create after the synthesis, like area, power, timing, cell, and netlist.

For running synthesis, compile_ultra has been used. Unlike the basic compile command, compile_ultra applies aggressive techniques to optimize timing, area, and power. It uses timing-driven mapping, logic restructuring, and register retiming to meet tight timing constraints and improve overall design quality. This command is especially useful for complex or timing-critical designs, as it provides better timing closure at the cost of longer synthesis runtime. The command -gate_clock is used with the compile_ultra command to optimize clock gating during synthesis.

DesignWare Building Block Library	Version	Available
Basic DW Building Blocks	S-2021.06-DWBB_202106.5	*
Licensed DW Building Blocks	S-2021.06-DWBB_202106.5	*

Flow Information	
Flow	Design Compiler WLM

Design Information	Value
Number of Scenarios	0
Leaf Cell Count	39755
Number of User Hierarchies	2
Sequential Cell Count	34522
Macro Count	0
Number of Power Domains	2
Design with UPF Data	true

Figure 8: Summary report while synthesizing

Figure 8 shows the summary report; this report indicates the leaf cell count as 39755 (cell count). Two power domains are present in the design. The design uses UPF for power intent info.

```
clock clk (rise edge)                    4.90       4.90
clock network delay (ideal)              0.00       4.90
clock uncertainty                       -0.25       4.66
CP0outo_reg[31]/CK (DFFR_X1)             0.00       4.66 r
library setup time                      -0.02       4.63
data required time                                  4.63
-----------------------------------------------------------
data required time                                  4.63
data arrival time                                  -4.63
-----------------------------------------------------------
slack (MET)                                         0.00
```

Figure 9: Timing report

Figure 9 shows the timing report. The clock period is set to 4.9 ns for the design to have a "0" slack.

5.4 Physical Implementation

To set up the environment for implementation, three files are needed. First, Verilog files (gate-level netlists) are obtained from synthesis. Second, the Multi-Mode Multi-Corner (MMMC) view file is the setup file for Innovus. This file has timing libraries, RC corners, technology files for extraction, and SDC constraint files. The last file is a globals file; this file includes all the Verilog files, MMMC file, LEF file, and all other TCL design commands. The MMMC view file, and global file use TCL scripting. The design commands, such as floorplan specification, placing, and routing, can be included in this global file, or the commands can be run in the terminal. After loading the global file in Innovus, the model is initialized.

5.4.1 Floor planning

The floor plan of the design should be specified; the core aspect ratio (H/W) is set to 0.99. The core utilization is set to 0.79 (79%). The core margins from the core to the

input/output (IO) boundary are set at 20 micrometers from all the sides. This is so important, as the whole layout depends on the way the space is optimized. The figure below shows the design after floor planning.

Figure 10: Floorplan of the design

5.4.2 Power Intent

The UPF file is loaded into the design by using read_power_intent by adding the type of type at the end. There are three types of power intent files: Common Power Format (CPF), Unified Power Format (UPF), and Database (DB) files. “-1801” should be used for the UPF file at the end. Figure 11 shows that the power intent is loaded into the design.

```
innovus 1> read_power_intent processor.upf -1801
Reading power intent file processor.upf ...
Checking power intent
Checking scoped supply_net connected to top-level supply_net
IEEE1801_RUNTIME: checking scoped supply_net: cpu=0:00:00.00 real=0:00:00.00
Checking supply_set/supply_net
IEEE1801_RUNTIME: checking pst supplies: cpu=0:00:00.00 real=0:00:00.00
Setting boundaryports from port_attr
IEEE1801_RUNTIME: checking port_attribute: cpu=0:00:00.00 real=0:00:00.00
IEEE1801_RUNTIME: checking related supply net: cpu=0:00:00.00 real=0:00:00.00
```

Figure 11: Loading Power Intent

The above command loads and checks the syntax of power intent and proceeds to the next step, executing the power intent. The command to execute power intent is commit_power_intent. This command brings all the power domains, switches, level shifters, and isolation cells that are already defined at the time of synthesis. Executing power intent makes all of them usable in Innovus. The power switches, level shifters, isolation cells, and retention cells are mapped to the respective cells that are present in the low-power open cell library, as shown in the image below.

```
innovus 3> commit_power_intent
IEEE1801_RUNTIME: freeTimingGraph: cpu=0:00:00.08 real=0:00:01.00
IEEE1801_RUNTIME: commit_logic_port/net: cpu=0:00:00.00 real=0:00:00.00
IEEE1801_RUNTIME: commit_supply_net: cpu=0:00:00.00 real=0:00:00.00
IEEE1801_RUNTIME: commit_power_domain: cpu=0:00:00.43 real=0:00:00.00
IEEE1801_RUNTIME: commit_global_connect: cpu=0:00:00.03 real=0:00:00.00
IEEE1801_RUNTIME: commit_power_mode: cpu=0:00:00.00 real=0:00:00.00
IEEE1801_RUNTIME: define low power cells: cpu=0:00:00.03 real=0:00:00.00
Ending "Constraint file reading stats" (total cpu=0:00:00.1, real=0:00:00.0, peak res=
Current (total cpu=0:00:29.8, real=0:05:43, peak res=957.0M, current mem=952.3M)
IEEE1801_RUNTIME: buildTimingGraph: cpu=0:00:00.14 real=0:00:00.00
Cell 'HEADER_OE_X1' has been added to powerDomain PD_CORE connection specification.
IEEE1801_RUNTIME: GNC_connect_existing_iso_shifter: cpu=0:00:00.35 real=0:00:01.00
IEEE1801_RUNTIME: connectAlwaysOnBuf: cpu=0:00:00.00 real=0:00:00.00
IEEE1801_RUNTIME: replaceAlwaysOnAssignBuffer: cpu=0:00:00.01 real=0:00:00.00
Cell 'HEADER_OE_X1' has been added to powerDomain PD_CORE connection specification.
IEEE1801_RUNTIME: commit_power_switch: cpu=0:00:00.00 real=0:00:00.00
IEEE1801_RUNTIME: commit_retention: cpu=0:00:00.00 real=0:00:00.00
INFO: isolation strategy ISO_CORE: added 0 isolation insts
IEEE1801_RUNTIME: commit_isolation: cpu=0:00:00.00 real=0:00:00.00
INFO: level_shifter strategy LS_CORE: added 0 level_shifter insts
IEEE1801_RUNTIME: commit_level_shifter: cpu=0:00:00.00 real=0:00:00.00
IEEE1801_RUNTIME: connectAlwaysOnBuf: cpu=0:00:00.00 real=0:00:00.00
```

Figure 12: Executing Power Intent

The process shown here is for fine-grained power gating. Coarse-grained power gating is done simultaneously but will be shown in the end.

5.4.3 Power Domain

After executing the power intent, the power domain specified is created. But if it does not have a specific area, then the cells will be placed randomly in the design. So, specifying boundaries for the power domain makes all the modules placed in the same area. This makes the power gating easy, as the required region to add power gates stays together. modifyPowerDomainAttr is used to define boundaries for it.

```
innovus 6> modifyPowerDomainAttr PD_CORE -box 50 50 100 100
Power Domain 'PD_CORE'.
          Boundary = 49.9700 49.4200 99.9700 99.4200
           minGaps = T:0.0700 B:0.0700 L:0.0700 R:0.0700
           rsExts  = T:0.0000 B:0.0000 L:0.0000 R:0.0000
           core2Side = T:0.0000 B:0.0000 L:0.0000 R:0.0000
           rowSpaceType = 2
           rowSpacing = 0.0000
           rowFlip = second
           site = FreePDK45_38x28_10R_NP_162NW_34O
```

Figure 13: Power Domain Modification

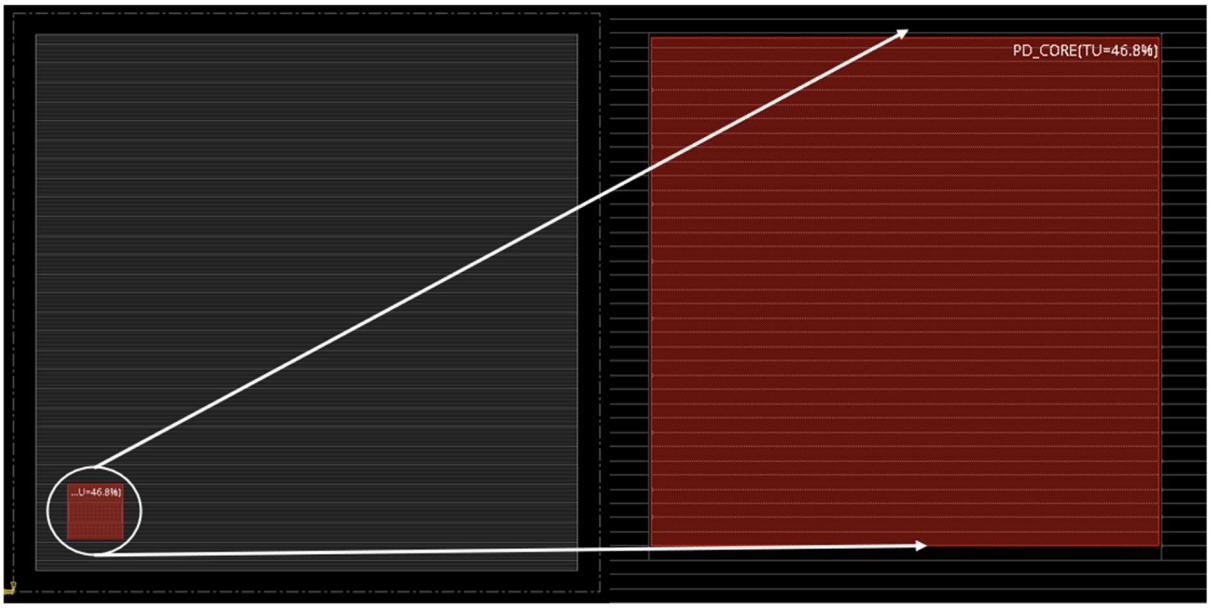

Figure 14: PD_CORE power domain

5.4.4 Power switches:

The power switches are added using the “addPowerSwitch” command. The power switches are already defined in the UPF file. The figure below shows that 140 power switches are added to the PD_CORE power domain. The switch uses the 1801 power switching rule, which brings the characteristics of SW_CORE to HEADER_X1. HEADER_X1 is already mapped to SW_CORE in the UPF file. The switches are added in column style with a horizontal distance of 15 micrometers.

```
innovus 6> addPowerSwitch -powerDomain PD_CORE -globalSwitchCellName HEADER_X1 -
1801PowerSwitchRuleName SW_CORE -column -horizontalPitch 15 -noDoubleHeightCheck

Start building column switches.
Create switch for column at x coord = 49.97
Create switch for column at x coord = 64.97
Create switch for column at x coord = 79.97
Create switch for column at x coord = 94.97
Applying level_shifter rules
INFO: level_shifter strategy LS_CORE: added 0 level_shifter insts

Total number of switches inserted in PD_CORE: 140
Total number of nets inserted in PD_CORE: 0
Total number of columns inserted in PD_CORE: 4

Verifing row coverage by pso switches.
```

Figure 15: addPowerSwitch command

Figure 16 shows the power switches being added to PD_CORE. The switches are highlighted in white color. All the switches are added as 4 columns.

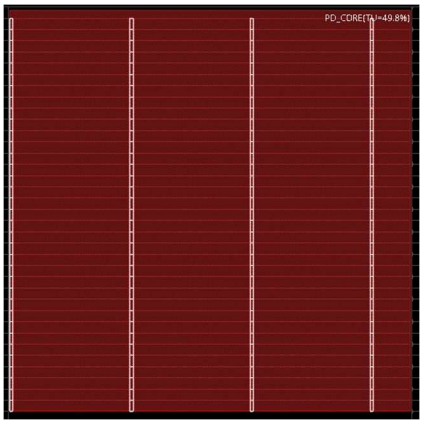

Figure 16: Power switches in PD_CORE

The power switches are added in column style because this is distributed across the power domain. This is flexible and makes retention easier. Even though this acts slightly slower than ring-style implementation, this offers other benefits like uniform switching and easy level shifting.

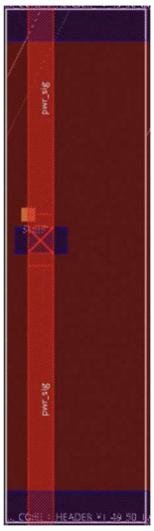

Figure 17: HEADER_X1 cell

This figure shows the individual HEADER_X1 cell added to the design. The label pwr_sig refers to the power control signal used to turn the switch ON/OFF. These cells stop the supply to the power-gated domain whenever they are not in use. The vertical red stripes are wide metal layers carrying VDD_CORE_OUT.

HEADER_X1 is often chosen over larger power switch cells like HEADER_X2 or HEADER_X4 because it provides better placement flexibility and finer granularity in power distribution. Smaller cells like X1 can be inserted more easily between standard cells without causing congestion, making them ideal for dense designs. Although more X1 switches are needed to handle the same current, they help distribute power more evenly across the chip, reducing the chances of voltage drop issues. This makes HEADER_X1 a safer and more efficient choice for maintaining power integrity in low-power and highly integrated designs.

5.4.5 Power Planning

First, power rings are added to the design. The power rings are typical supply voltage metal layers around the design, forming a rectangular loop to provide power uniformly. These rings are added in the GUI of Innovus by selecting the required nets for the design.

Figure 18: Power Rings

Figure 18 shows the Power Rings for the design. The power rings are added to the whole design using Metal 9 for horizontal nets and Metal 10 for vertical nets. The nets added are VDD, VDD_CORE_OUT (supply voltage to power domain PD_CORE) and VSS.

Special routing is a critical step in physical design used to route non-signal nets such as power (VDD, VSS). It is done before regular signal routing to ensure that these essential nets have dedicated, optimized paths with sufficient width, spacing, and metal layer usage. The primary reason for special routing is to handle the higher current demand and timing sensitivity of these power nets, and they must deliver stable voltage with minimal IR drop across the chip. By performing special routing separately, designers can ensure robust power delivery, efficient clock distribution, and reliable connectivity to macros and standard cells, ultimately improving the chip's performance and power integrity.

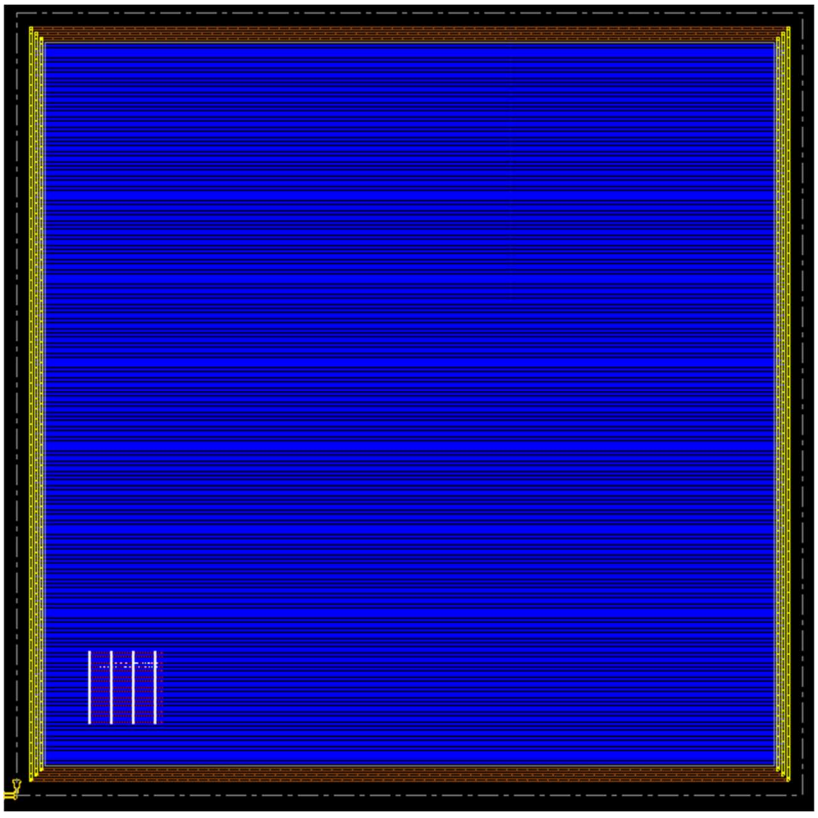

Figure 19: Special Routing

Figure 19 shows the design after special routing. The VDD_CORE_OUT and VSS are added to the power domain PD_CORE. VDD and VSS nets are distributed for the rest.

The next step is to add power strips. This helps deliver power evenly across the chip. Power stripes are wide metal lines that run across the chip and connect to the power rings. They carry the VDD (power) and VSS (ground) signals from the rings into the center of the chip, where all the standard cells are placed. Since the chip has many tiny components that all need power, these stripes make sure that every part gets enough power without too much voltage drop. They also help prevent problems like overheating or parts of the chip not working properly. In simple terms, power stripes are like strong highways that carry electricity to every part of the chip.

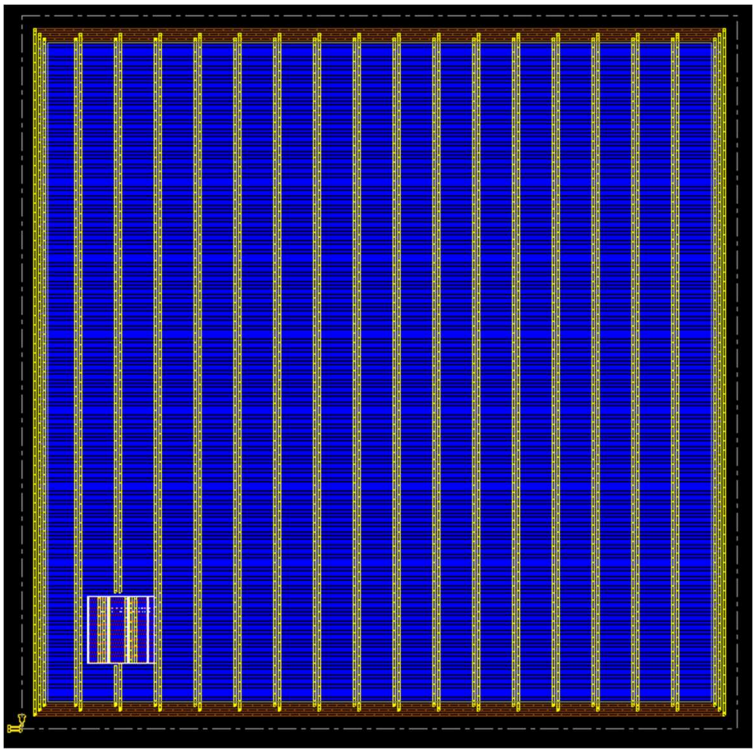

Figure 20: Power stripes

The above figure shows power stripes added to the whole design and the power domain PD_CORE as well. The VDD_CORE_OUT and VSS are added vertically by selecting Metal 10. The gap between each pair of stripes is 12 micrometers. VDD and VSS stripes are added to the rest of the design with a set-to-set distance of 30 micrometers.

5.4.6 Standard Cell Placement

Standard cell placement is the process of arranging small logic blocks like gates and flip-flops inside the core area of a chip. These cells are placed in rows, much like bricks on a wall, to ensure they fit neatly and connect efficiently. The goal is to position the cells in a way that reduces wire length, improves timing, and leaves space for routing, power, and clock connections. During placement, the tool ensures that there are no overlaps and that cells align properly with the power rails. This step is very important because good placement leads to better performance, lower power usage, and easier routing in the next stages of chip design. “place_opt” is used to start the cell placement.

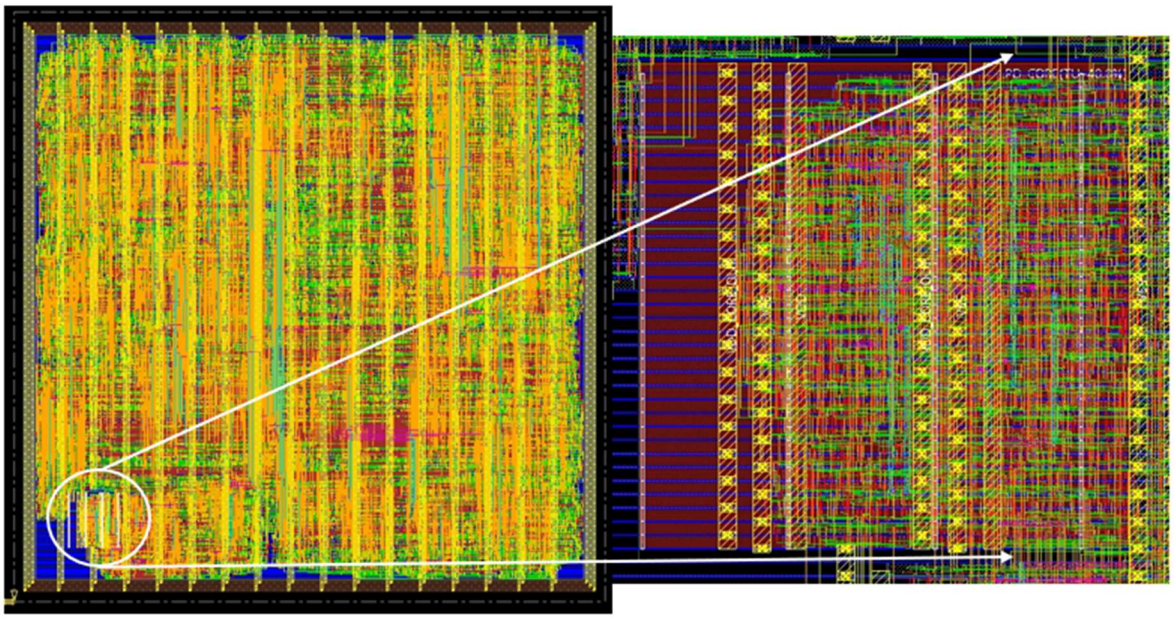

Figure 21: Standard cell placement

The above figure shows that standard cells are neatly arranged in rows inside the core area, ensuring there are no overlaps and that each cell is properly aligned with the power rails. Power stripes and routing tracks are visible running across the layout, enabling efficient power delivery and signal connection.

5.4.7 Clock Tree Synthesis (CTS)

Clock Tree Synthesis (CTS) is a stage in chip design where the tool creates a special network to deliver the clock signal to all the flip-flops in the chip. The goal is to make sure the clock reaches every flip-flop at nearly the same time so that all parts of the chip work in sync. This helps avoid problems like timing errors or glitches. During CTS, the tool adds clock buffers or inverters to balance the path and reduce delay differences, which is called "skew." A well-built clock tree makes the chip faster, more reliable, and efficient in power usage. To synthesize the clock tree, the buffer cells need to be specified. By choosing specific cells, the tool uses those cells, which helps to control area, power, and timing for the design. After that, a clock tree needs to be created; then it can be synthesized to the design.

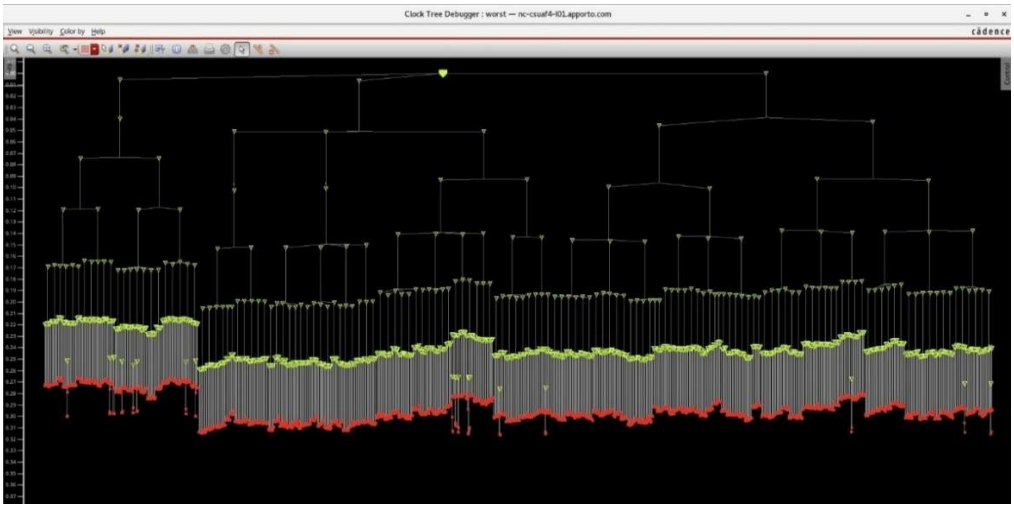

Figure 22: Clock tree

The above figure shows the clock tree. For this design, CLKBUF_X1, X2, and X3 are available to use. The commands to create a clock tree are

```
set_ccopt_property cts_buffer_cells {CLKBUF_X1 CLKBUF_X2 CLKBUF_X3}
create_ccopt_clock_tree_spec
ccopt_design
```

5.4.8 Design Rule Check (DRC)

After placement, a design rule check is performed to check whether the placement is clean and correct from placement or setup issues. This helps prevent routing errors, saves time by catching problems early, and ensures that power planning and floorplan constraints are correctly followed. Fixing issues before routing is much easier and avoids the need for major rework later in the flow.

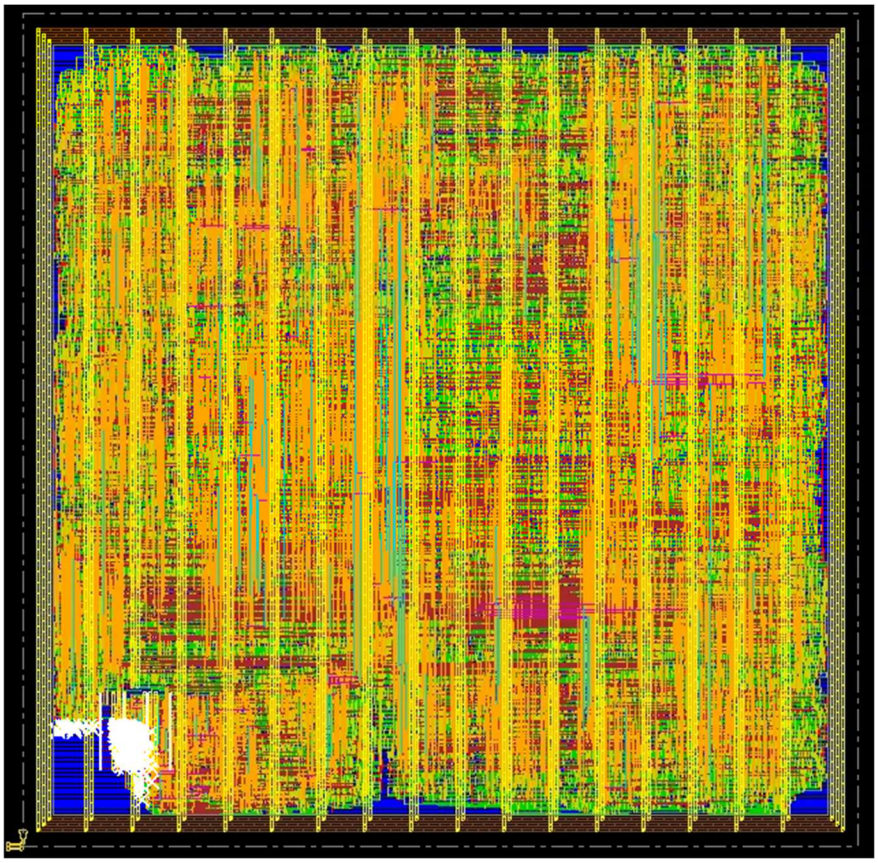

Figure 23: Violations after DRC

The above image shows violations after DRC. A total of 1000 violations are shown in this image from the bottom left side of the design. The DRC is set to stop at a maximum of 1000 violations. Violations need to be rectified. All the violations that occurred will be cleared after performing routing.

5.4.9 Routing

Early global routing is done right after placement to estimate routing congestion and wire delays before detailed routing begins. It helps identify potential routing issues, congested areas, and timing violations early in the design flow. This allows designers and tools to adjust cell placement, sizing, and floorplan to make routing easier and more efficient. It is an important predictive step that saves time and ensures better quality of results in the final routing stage.

Detailed routing is the process of drawing the actual metal wires that connect all parts of a chip after placement. It uses real routing layers and vias to make all the required connections while strictly following manufacturing rules. This step is done using tools like NanoRoute in Cadence Innovus, and it ensures that the chip layout is correct, DRC-clean, and ready for fabrication. It is one of the final and most critical steps in making a working chip. It helps make sure that all connections are correct, there are no errors, and the chip can be manufactured safely. It also works to improve performance by optimizing wire paths and reducing delays.

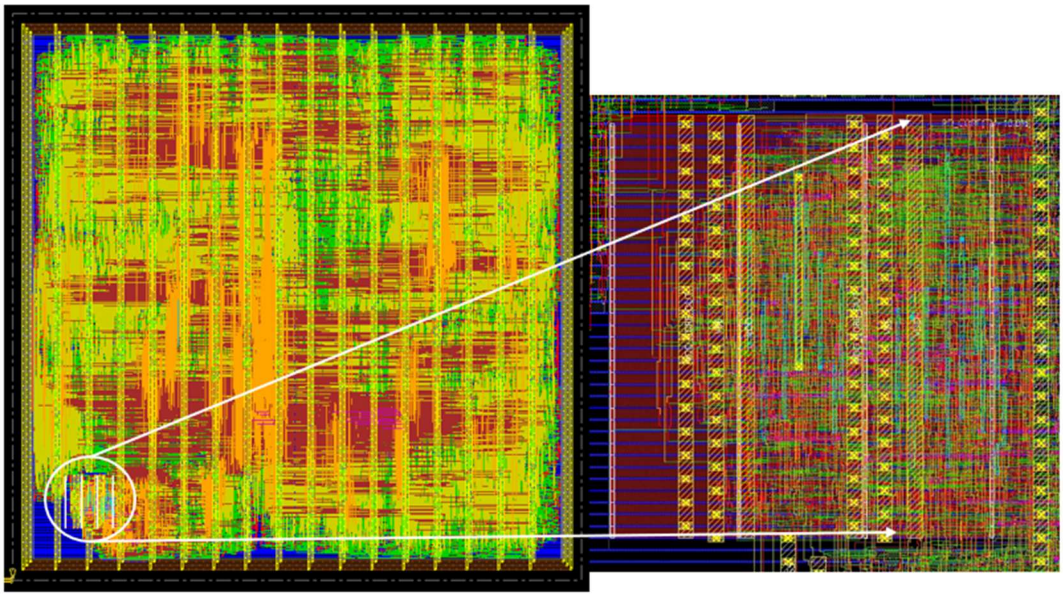

Figure 24: Fine-Grained Detailed routing

Figure 24 shows the layout after detailed routing with 0 violations. The right-side part shows the PD_CORE power domain with power switches after detailed routing. This design is saved as GDSII and can proceed to the next step of fabrication.

The coarse-grained design is synthesized using another UPF file defined earlier. More areas have been used for power domain PD_CORE as it has larger modules. Coarse-grained method is implemented using the same process.

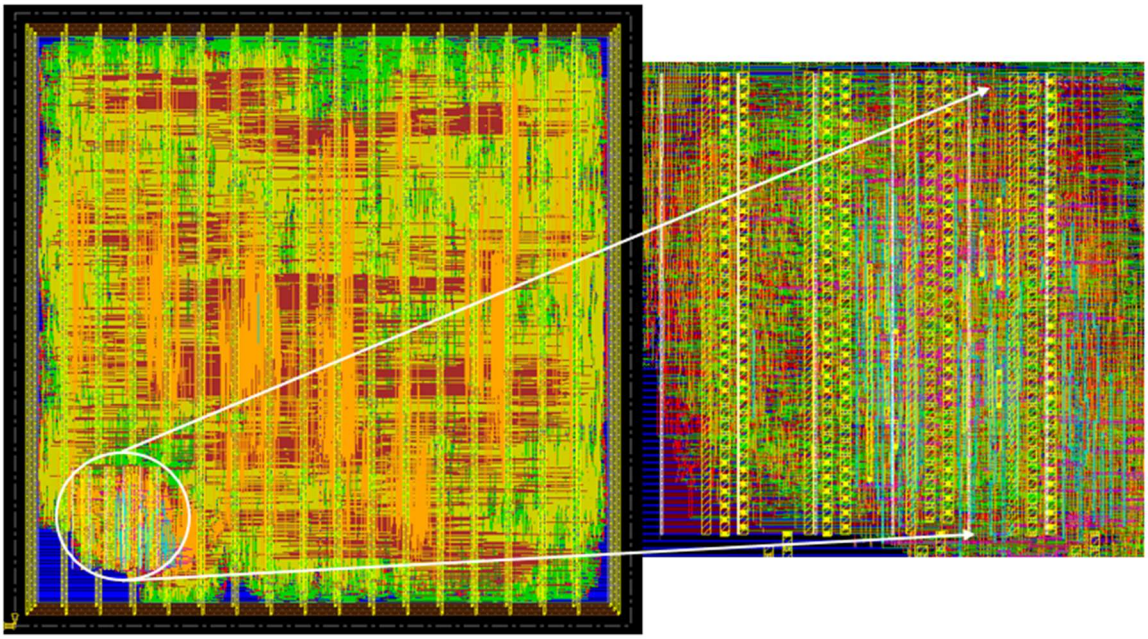

Figure 25: Coarse-Grained Detailed Routing

Figure 25 shows the layout for the coarse-grained power-gated RISC-V processor. The same process has been followed throughout the design, but the modules have been changed. Instead of many small modules, only one large module is gated in this design. More switches have been added to this design. As the size of the power-gated domain is large.

5.4.10 Power Analysis

To implement power gating for specific modules, a control signal named pwr_sig was introduced into the design. This signal was provided as an input to each module that required power gating, enabling or disabling its power supply based on system activity. The pwr_sig signal was also referenced in the Unified Power Format (UPF) file during power switch creation, ensuring proper control and connection of the power gates within the Innovus flow. The top-level module of the design manages the pwr_sig signal, dynamically asserting it to 1'b1 whenever the corresponding module needs to be powered on for operation. To test whether power gating is working or not we need to do power analysis.

Power analysis in Innovus estimates the dynamic, leakage, and total power consumption of a design after placement and routing. A value change dump (VCD) file is created in modelsim using $dumpfile and $dumpvars commands. The generated VCD file was then imported into Cadence Innovus for activity-based power analysis. A switching activity of 0.5 was applied during power analysis to achieve a more accurate estimation of dynamic power based on realistic signal toggling. The following commands are used to load the .vcd file and execute the power analysis.

```
read_activity_file -format VCD ./processor_dump.vcd
set_powerup_analysis -reset
set_dynamic_power_simulation -reset
report_power -outfile ./processor_power.rpt
```

After these commands, a power report was generated. This process was performed to accurately measure the power consumption based on real-time switching activity within the processor.

CHAPTER 6: RESULTS & ANALYSIS

6.1 ModelSim Waveforms

The design has been simulated in ModelSim, and the output is functionally verified. The figure below shows the RISC-V processor output waveforms. The successful execution of instructions and the resulting changes in memory demonstrate that the RISC-V processor is both logically and functionally correct.

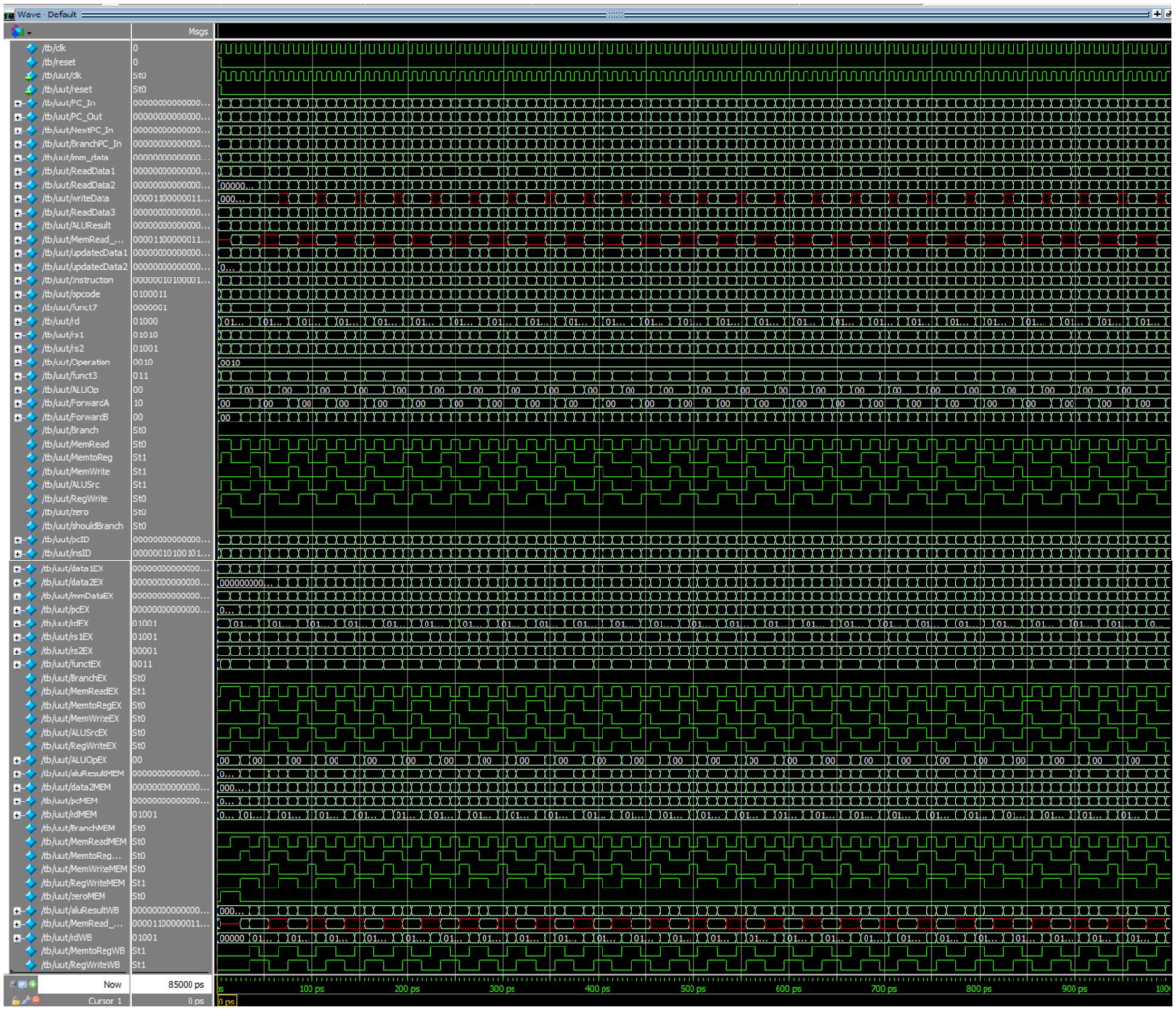

Figure 26: Output Waveforms

6.2 Synopsys Design Compiler Reports

6.2.1 Area Report

The area reports for both designs are shown below. Both have almost the same area after synthesis. The area reports show nearly 30,000 combination cells and 34,000 sequential cells. Total cell area is nearly 196,000.

```
Number of ports:                         4341
Number of nets:                         70193
Number of cells:                        66363
Number of combinational cells:          30604
Number of sequential cells:             34718
Number of macros/black boxes:               0
Number of buf/inv:                       2800
Number of references:                    1086

Combinational area:              39120.088922
Buf/Inv area:                     2203.809964
Noncombinational area:          157164.498788
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:                196284.587709
Total area:                 undefined
```

Figure 27: Fine-Grained RISC-V area report

```
Number of ports:                         4792
Number of nets:                         70805
Number of cells:                        66589
Number of combinational cells:          30822
Number of sequential cells:             34719
Number of macros/black boxes:               0
Number of buf/inv:                       2844
Number of references:                    1081

Combinational area:              39217.444914
Buf/Inv area:                     2235.995963
Noncombinational area:          157167.956788
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:                196385.401701
Total area:                 undefined
```

Figure 28: Coarse-Grained RISC-V area report

6.2.2 Power Report

The below figures show both power reports for coarse-grained and fine-grained designs. From both images, the power for coarse-grained design is higher. After synthesis, the power reports illustrate that fine-grained design is low power compared to the coarse-grained design.

```
  Cell Internal Power  =   4.9437 mW   (83%)
  Net Switching Power  = 984.9554 uW   (17%)
                         ---------
Total Dynamic Power    =   5.9287 mW  (100%)

Cell Leakage Power     =  11.3817 mW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network  1.4543e+03         173.1765          2.2236e+05         1.8499e+03  (  10.69%)
register       3.2205e+03          37.4376          8.7551e+06         1.2013e+04  (  69.40%)
sequential         0.0000           0.0000              0.0000             0.0000  (   0.00%)
combinational    269.1738         774.3347          2.4042e+06         3.4477e+03  (  19.92%)
--------------------------------------------------------------------------------------------------
Total          4.9441e+03 uW      984.9488 uW      1.1382e+07 nW      1.7311e+04 uW
1
```

Figure 29: Fine-Grained RISC-V DC power report

```
  Cell Internal Power  =   9.2944 mW   (83%)
  Net Switching Power  =   1.8771 mW   (17%)
                         ---------
Total Dynamic Power    =  11.1715 mW  (100%)

Cell Leakage Power     =  11.5526 mW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network  1.5764e+03         211.9278          2.2256e+05         2.0109e+03  (   8.85%)
register       7.3248e+03          50.9159          8.9352e+06         1.6311e+04  (  71.77%)
sequential         0.0000           0.0000              0.0000             0.0000  (   0.00%)
combinational    394.3120        1.6142e+03          2.3948e+06         4.4034e+03  (  19.38%)
--------------------------------------------------------------------------------------------------
Total          9.2955e+03 uW      1.8771e+03 uW      1.1553e+07 nW      2.2725e+04 uW
1
```

Figure 30: Coarse-Grained RISC-V DC power report

6.3 Innovus Results

6.3.1 Power Report

The below figures show power reports from Innovus after physical implementation. From these reports, fine-grained design consumes less power than coarse-grained design.

```
Total Power
-------------------------------------------------------------------
---------
Total Internal Power:        1.80853855            50.9520%
Total Switching Power:       1.00481209            28.3086%
Total Leakage Power:         0.73614366            20.7394%
Total Power:                 3.54949427
-------------------------------------------------------------------
---------
```

Figure 31: Fine-Grained RISC-V Innovus power report

```
Total Power
-------------------------------------------------------------------
---------
Total Internal Power:        2.15679377            41.6336%
Total Switching Power:       2.23650611            43.1723%
Total Leakage Power:         0.78711411            15.1940%
Total Power:                 5.18041402
-------------------------------------------------------------------
---------
```

Figure 32: Coarse-Grained RISC-V Innovus power report

The power histograms for both designs are shown below.

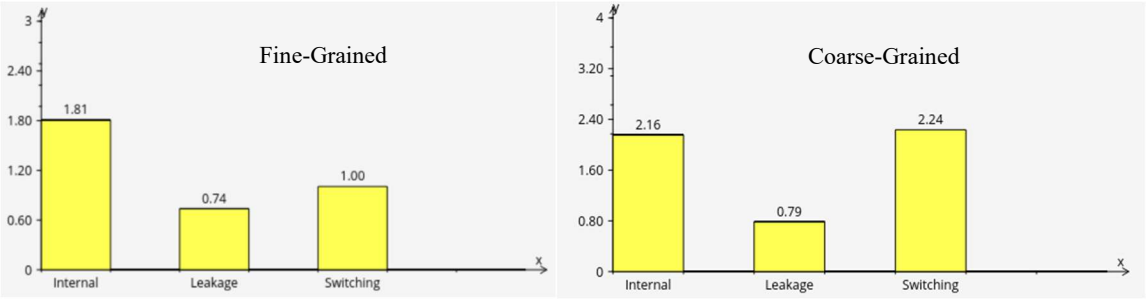

Figure 33: Power histograms for both designs.

6.4 Comparison

Table 1: Comparison of Both Power Gating Methods.

Parameter \ Type	Fine-Grained Power Gating	Coarse-Grained Power Gating
Area	196284.58	196385.40
Power (DC)	17.3 mW	22.7 mW
Power (Innovus)	3.5 mW	5.1 mW
Power Domain Size	2.5 mm^2	8.1 mm^2
Number of Power Gates	140	384

The above table shows a comparative analysis of fine-grained power gating and coarse-grained power gating. The table revealed notable differences in power consumption and implementation metrics based on the evaluated design. The fine-grained technique demonstrated superior power efficiency, registering significantly lower power values in both DC (17.3 mW for fine-grained vs. 22.7 mW for coarse-grained) and Innovus (3.5 mW for fine-grained vs. 5.1 mW for coarse-grained) measurements. While the difference in total area was negligible between the two approaches (196284.58 for fine-grained vs. 196385.40 for coarse-grained).

Fine-grained required substantially fewer power gates (140) compared to coarse-grained (384). This lower number of gates associated with the fine-grained implementation suggests a potentially simpler power management network, which could offer advantages in terms of control logic design and routing complexity. The power domain size is low for the fine-grained technique; it is easy to manage a small power domain compared to a larger one.

CHAPTER 7: CONCLUSION & FUTURE SCOPE

7.1 Conclusion

In this project, the RISC-V processor was successfully power gated using coarse-grained and fine-grained techniques. Both designs have been implemented from RTL design to GDSII, including Verilog files, synthesis, and physical implementation. These methods are designed simultaneously using a UPF file to power gate both designs. Both methods are compared to each other based on metrics like power consumption, area, and number of power gates used.

The comparative analysis between fine-grained and coarse-grained power gating for the evaluated design clearly favors the fine-grained technique. While both approaches resulted in similar silicon area usage, fine-grained demonstrated significant advantages in power efficiency, achieving markedly lower power consumption values. Moreover, fine-grained uses a small number of power gates with less area, pointing towards simpler control network implementation and easier management. Therefore, fine-grained power gating emerges as the more effective strategy with potentially lower implementation complexity.

In conclusion, fine-grained power gating is suitable for the RISC-V processor to overcome the high-power consumption. From the comparison, the coarse-grained approach is not preferred. If it is a regular design, a coarse-grained approach is preferred because it gates larger modules to save power consumption. However, if larger modules like pipeline stages are gated, then they are required to stay ON most of the time. But smaller modules like ALU, decoder, coprocessor, etc., are not required all the time. Gating these modules will help the design to work properly and decrease power compared to coarse-grained.

7.2 Future Work

Future work for this project will focus on the following areas:

While this project successfully demonstrates the effectiveness of fine-grained power gating over coarse-grained power gating in a pipelined RISC-V processor, several enhancements and explorations remain open for future work. One promising direction is the implementation of dynamic power gating control logic based on real-time workload prediction. Currently, power gating techniques are applied statically using UPF. However, adding intelligent control circuitry or machine learning based predictors to selectively gate blocks at runtime. This can further optimize power savings depending on instruction patterns or system idle times.

Another important area for future exploration is the integration of multi-mode power gating. Instead of only ON/OFF gating, introducing intermediate states like retention or sleep modes could help preserve context and reduce wake-up latency. This can be especially beneficial in processors with frequent switching activities or real-time constraints. These modes can be integrated into the fine-grained approach, allowing blocks like ALUs or coprocessors to retain essential states without fully consuming active power.

Additionally, thermal-aware and voltage-aware gating techniques can be studied to complement the power gating approach. By monitoring thermal hotspots or dynamic voltage scaling (DVS) conditions, the power gating mechanism can respond more effectively, providing a balance between performance, reliability, and energy efficiency. Extending the design to include multi-core RISC-V processors with shared and distributed resources can test the scalability of fine-grained strategies and their applicability in more complex processor architectures.

REFERENCES

REFERENCES

[1] V. M. Montabes Jiménez, “Impact of physical low power techniques in a RISC-V processor”, master’s thesis, Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona, Universitat Politècnica de Catalunya, Barcelona, Spain, Jan. 2021.

[2] S. Umapathy, “Design of coarse-grained power gating for a fine-grained many-core processor array”, M.S. thesis, Lyles College of Engineering, California State University, Fresno, CA, USA, Aug. 2018.

[3] H. V. Ravish Aradhya, G. Kanase and V. Y, "RTL to GDSII of Harvard Structure RISC Processor," 2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT), Bangalore, India, 2021, pp. 1-4, doi: 10.1109/CONECCT52877.2021.9622735.

[4] M. S. Gowda, C. Koti and P. Mohanachandran, "Power Optimization Techniques During Synthesis and Physical Design for a Low-Power RISC-V Design," 2024 IEEE International Conference on Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER), Mangalore, India, 2024, pp. 153-158, doi: 10.1109/DISCOVER62353.2024.10750582.

[5] M. Saini, S. Shringi and A. Asati, "An Improved Power Gating Technique with Data Retention and Clock Gating," 2021 International Conference on Control, Automation, Power and Signal Processing (CAPS), Jabalpur, India, 2021, pp. 1-7, doi: 10.1109/CAPS52117.2021.9730489.

[6] G. Kanase and N. M, "ASIC Design of a 32-bit Low Power RISC-V based System Core for Medical Applications," 2021 6th International Conference on Communication and Electronics Systems (ICCES), Coimbatre, India, 2021, pp. 1-5, doi: 10.1109/ICCES51350.2021.9489067.

[7] M. A. Raheem, M. S. Hussain and P. R. Ahmed Khan, "ASIC Flow Implementation on a 32-Bit RISC V Processor using Cadence," 2024 IEEE International Conference of Electron Devices Society Kolkata Chapter (EDKCON), Kolkata, India, 2024, pp. 29-33, doi: 10.1109/EDKCON62339.2024.10870604.

[8] K. Singh, “Designing and Implementing the Multipliers and Making Them Power Efficient Using Power Gating Technique”, M.S. project, Lyles College of Engineering, California State University, Fresno, CA, USA, Aug. 2023.

[9] M. Kondo et al., "Design and evaluation of fine-grained power-gating for embedded microprocessors," 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 2014, pp. 1-6, doi: 10.7873/DATE.2014.158.

[10] S. Eranki and K. Babulu, "Power gating verification of a core in SOC," 2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), Chennai, India, 2016, pp. 1-4, doi: 10.1109/ICCIC.2016.7919648.

[11] P. Verma, A. Noor, A. K. Sharma, and S. K. Vemuri, "Power Gating and Its Repercussions—A Review," in Proc. 1st IEEE Int. Conf. Power Electronics, Intelligent Control and Energy Systems (ICPEICES), Delhi, India, 2016.

[12] R. N. Agnes Shiny, B. Fahimunnisha, S. Akilandeswari and S. J. Venula, "Integration of Clock Gating and Power Gating in Digital Circuits," 2019 5th International Conference on Advanced Computing & Communication Systems (ICACCS), Coimbatore, India, 2019, pp. 704-707, doi: 10.1109/ICACCS.2019.8728370.

[13] A. Gajbhiye and U. Ghodeswar, "Implementation, Validation, and Power Aware Simulation of Low Power 4-bit ALU using Unified Power Format standards," 2024 International Conference on IoT Based Control Networks and Intelligent Systems (ICICNIS), Bengaluru, India, 2024, pp. 737-743, doi: 10.1109/ICICNIS64247.2024.10823168.

[14] J. Biggs, E. Marschner, S. Honnavara-Prasad, D. Cheng, S. Ramachandra, J. Worthington, and N. Dhanwada, "Using UPF for Low Power Design and Verification," presented at Tutorial #2 by members of the IEEE P1801 Working Group, Mentor Graphics, Mar. 3, 2014.

[15] IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems, IEEE Standard 1801-2018 (Revision of IEEE Std 1801-2015), IEEE Computer Society, 2018, pp. 1–284. doi: 10.1109/IEEESTD.2018.8418596.

[16] H. Kumar and S. Saun, "Power Gated Technique to Improve Design Metrics of 6T SRAM Memory Cell for Low Power Applications," ICTACT Journal on Microelectronics, vol. 5, no. 3, pp. 815, Oct. 2019, doi: 10.21917/ijme.2019.0140.

[17] P. Kumar. M. P. and A. S. A. Fletcher, "A Survey on Leakage Power Reduction Techniques by Using Power Gating Methodology," International Journal of Engineering Trends and Technology (IJETT), vol. 9, no. 11, pp. 566, Mar. 2014.

[18] Hailin Jiang, M. Marek-Sadowska and S. R. Nassif, "Benefits and costs of power-gating technique," 2005 International Conference on Computer Design, San Jose, CA, USA, 2005, pp. 559-566, doi: 10.1109/ICCD.2005.34.

[19] A. Kirthanaa and P. Umarani, "Survey on Different Power Gating Techniques," Research Journal of Pharmaceutical, Biological and Chemical Sciences, vol. 7, no. 1, pp. 956, Jan.–Feb. 2016.

[20] B. P. Sharath and K. V. Suhas, "Power Gating to Reduce Leakage Current in Low Power CMOS Circuits," IOSR Journal of VLSI and Signal Processing, vol. 3, no. 3, pp. 19–22, Sep.–Oct. 2013.

[21] B. P. Jyothi and D. S. Suresh, "Power Gating Methods: Comparative Study," International Journal of Computer Science and Technology, vol. 4, no. 2, pp. 588, Apr.–Jun. 2013.

[22] A. Teman, "Power Intent and Low Power Methodology," presented at Emerging Nanoscaled Integrated Circuits and Systems (EnICS) Lab, Bar-Ilan University, Jun. 24, 2020.

[23] ModelSim - Intel® FPGA Edition Simulation Quick-Start*, Intel® Quartus® Prime Standard Edition, Intel Corporation, Updated for Intel® Quartus® Prime Design Suite: Version 18.0, 2018.

[24] Design Compiler® User Guide, Version U-2022.12-SP3, Synopsys Inc., Apr. 2023.

[25] Low-Power Flow User Guide, Version D-2010.03, Synopsys Inc., Mar. 2010.

[26] Innovus Implementation System (Block), Lab Manual, Course Version 18.1, Revision 1.0, Cadence Design Systems, Inc., 2018.

[27] Innovus User Guide, Product Version 23.13, Cadence Design Systems, Inc., Nov. 2024.

[28] End Cap Cells Usage in Innovus Flow, Cadence Design Systems, Inc., 2017.

[29] Nangate Inc., Nangate 45nm Open Cell Library, Version 1.3, Jul. 2009.

[30] W. R. Davis, P. D. Franzon, J. L. Stine, D. Markovic, and Y. Cao, “FreePDK: An Open-Source Variation-Aware Design Kit,” IEEE Micro, vol. 28, no. 2, pp. 60–69, Mar.–Apr. 2008.

[31] CPF to UPF (IEEE-1801) Migration Guide, Product Version 17.04, Synopsys Inc., Aug. 2017.

APPENDICES

APPENDIX A: VERILOG FILES

Top Module- Processor

```
`timescale 1ns / 1ps
`ifndef INCLUDED_MODULES
    `define INCLUDED_MODULES

    `include "memory.v"
    `include "instMemory.v"
    `include "pc.v"
    `include "IFID.v"
    `include "controller.v"
    `include "decoder.v"
    `include "register.v"
    `include "IDIE.v"
    `include "ALU.v"
    `include "IEME.v"
    `include "MEWB.v"
    `include "coprocessor0.v"
    `include "stall.v"
    `include "stage6.v"
`endif

module processor (
    input clk, rst_n, pwr_sig,
    output [11:0] io_out,
    input [15:0] io_in
);
    wire reset;

    wire writeEnable2, writeEnable3, writeEnable4, writeEnable5, writeEnable6;
```

```
wire readMem2, readMem3, readMem4;
wire writeMem2, writeMem3, writeMem4;
wire signExt2, signExt3, signExt4;
wire jumpSignal4;
wire [1:0] aluInputSel2, aluInputSel3;
wire [1:0] jumpSel2, jumpSel3, jumpSel4;
wire [3:0] aluFunc2, aluFunc3;
wire mulSelect2, mulSelect3;
wire [2:0] instrFmt2;
wire [6:0] opcode2, func7_2;
wire [2:0] func3_2, func3_3;
wire [31:0] instr1, instr2;
wire [1:0] writeBackSel4, writeBackSel5, writeBackSel3, writeBackSel2;

wire [31:0] pcPlus4_1, pcPlus4_2, pcPlus4_3, pcPlus4_4;
wire [31:0] pcCurrent1, pcCurrent2, pcCurrent3;
wire [31:0] pcImmAdd3, pcImmAdd4;

wire [31:0] immOut2, immOut3;
wire [31:0] aluOut3, aluOut4;
reg [31:0] aluBInput3;
wire [31:0] writeDataHalf5, writeData6;
reg [31:0] writeData5;
reg [31:0] writeDataHalf4;

wire [31:0] regDataA3, regDataB3, regDataA2, regDataB2;
wire [4:0] regAddrA2, regAddrB2, regAddrA3, regAddrB3;
reg [31:0] forwardedB;
reg [31:0] forwardedA;
```

```
    wire [31:0] forwardedB4;
    wire [4:0] regDest2, regDest3, regDest4, regDest5, regDest6;

    wire [1:0] writeLen2, writeLen3, writeLen4;
    wire [31:0] memOut4, memOut5;

    wire stall_IFID, bubble_IFID, bubble2, bubble_IDIE, stall_IDIE, bubble_IEME,
stall_IEME, bubble_MEWB, stall_MEWB, stall_S6;

    wire [2:0] func3_4;
    wire cp0Success, bubble_CP0;
    wire [31:0] cp0Out3, cp0Out4;

    assign reset = ~rst_n;

    // Stage 1 - Instruction Fetch
    InstructionMem u_IM1(instr1, pcCurrent1);
    programCounter u_PC(pcCurrent1, pcPlus4_1, aluOut4, pcImmAdd4, jumpSel4,
jumpSignal4, clk, reset, stall_IFID);
    IFID u_IFID(pcCurrent2, pcPlus4_2, instr2, bubble2, pcCurrent1, pcPlus4_1,
instr1, bubble_IFID, clk, reset, stall_IFID);

    // Stage 2 - Instruction Decode
    assign opcode2 = instr2[6:0];
    assign func3_2 = instr2[14:12];
    assign func7_2 = instr2[31:25];
    assign regDest2 = instr2[11:7];
    assign regAddrA2 = instr2[19:15];
    assign regAddrB2 = instr2[24:20];
```

```
    controller u_CTRL(writeEnable2, writeBackSel2, readMem2, writeMem2,
signExt2, aluInputSel2,
        jumpSel2, mulSelect2, jumpOpns2, aluFunc2, instrFmt2, writeLen2,
        opcode2, func3_2, func7_2, bubble2);

    decoder u_DEC(immOut2, instrFmt2, instr2, 1'b1);

    Regester u_REG(clk, reset, writeEnable5, regDest5, writeData5,
        regAddrA2, regDataA2, regAddrB2, regDataB2);

    IDIE u_IDIE(pcCurrent3, pcPlus4_3, immOut3, regDataA3, regDataB3, func3_3,
writeEnable3, writeBackSel3, readMem3, writeMem3,
        signExt3, jumpSel3, jumpOpns3, mulSelect3, aluInputSel3, aluFunc3, regDest3,
regAddrA3, regAddrB3, writeLen3,
        pcCurrent2, pcPlus4_2, immOut2, regDataA2, regDataB2, func3_2,
{bubble_IDIE ? 1'b0 : writeEnable2}, writeBackSel2,
        {bubble_IDIE ? 1'b0 : readMem2}, {bubble_IDIE ? 1'b0 : writeMem2},
signExt2, jumpSel2, jumpOpns2, mulSelect2, aluInputSel2,
        aluFunc2, regDest2, regAddrA2, regAddrB2, writeLen2, clk, reset, stall_IDIE);

    // Stage 3 - Execute
    assign pcImmAdd3 = pcCurrent3 + immOut3;

    ALU u_ALU(aluOut3, forwardedA, aluBInput3, aluFunc3);

    cop0 u_CP0(cp0Out3, cp0Success, forwardedA, aluBInput3, func3_3,
{bubble_CP0 ? 1'b0 : mulSelect3}, clk, reset);
```

```
    // Forwarding logic
    always @(*) begin
        if(regDest4 == regAddrA3 && writeEnable4 && (writeBackSel4 == 0) &&
regDest4 != 0)
            forwardedA = writeDataHalf4;
        else if(regDest5 == regAddrA3 && writeEnable5 && regDest5 != 0)
            forwardedA = writeData5;
        else if(regDest6 == regAddrA3 && writeEnable6 && regDest6 != 0)
            forwardedA = writeData6;
        else
            forwardedA = regDataA3;
    end

    always @(*) begin
        if(regDest4 == regAddrB3 && writeEnable4 && (writeBackSel4 != 1) &&
regDest4 != 0)
            forwardedB = writeDataHalf4;
        else if(regDest5 == regAddrB3 && writeEnable5 && regDest5 != 0)
            forwardedB = writeData5;
        else if(regDest6 == regAddrB3 && writeEnable6 && regDest6 != 0)
            forwardedB = writeData6;
        else
            forwardedB = regDataB3;
    end

    always @(*) begin
        case (aluInputSel3)
            PAluImm: aluBInput3 = immOut3;
            PAluPC: aluBInput3 = pcCurrent3;
```

```
            default: aluBInput3 = forwardedB;
        endcase
    end

    IEME u_IEME(pcPlus4_4, aluOut4, pcImmAdd4, cp0Out4, func3_4,
writeEnable4, writeBackSel4, readMem4, writeMem4,
        signExt4, mulSelect4, jumpSel4, jumpOpns4, forwardedB4, regDest4,
writeLen4,
        pcPlus4_3, aluOut3, pcImmAdd3, cp0Out3, func3_3, {bubble_IEME ? 1'b0 :
writeEnable3}, writeBackSel3,
        {bubble_IEME ? 1'b0 : readMem3}, {bubble_IEME ? 1'b0 : writeMem3},
signExt3, mulSelect3, jumpSel3, jumpOpns3,
        forwardedB, regDest3, writeLen3, clk, reset, stall_IEME);

    // Stage 4 - Memory Access
    Memory u_MEM(clk, aluOut4, forwardedB4, writeMem4, readMem4, signExt4,
writeLen4, memOut4, io_out, io_in);

    assign jumpSignal4 = ((jumpOpns4) || (aluOut4[0] ^ func3_4[0])) && (jumpSel4
!= PJumpPc4);

    always @(*) begin
        case (writeBackSel4)
            2'd3: writeDataHalf4 = pcPlus4_4;
            2'd2: writeDataHalf4 = pcImmAdd4;
            default: writeDataHalf4 = mulSelect4 ? cp0Out4 : aluOut4;
        endcase
    end
```

```
   MEWB u_MEWB(writeDataHalf5, memOut5, writeEnable5, writeBackSel5,
regDest5,
      writeDataHalf4, memOut4, {bubble_MEWB ? 1'b0 : writeEnable4},
writeBackSel4, regDest4, clk, reset, stall_MEWB);

   // Stage 5 - Writeback
   always @(*) begin
      case (writeBackSel5)
         2'd1: writeData5 = memOut5;
         default: writeData5 = writeDataHalf5;
      endcase
   end

   // Stage 6 - Virtual forwarding
   s6_Forward u_S6(writeData6, regDest6, writeEnable6, writeData5, regDest5,
writeEnable5, clk, reset, stall_S6);

   // Stall Unit
   stall_Bpred u_STL(bubble_IFID, bubble_IDIE, bubble_IEME, bubble_MEWB,
bubble_CP0, stall_IFID, stall_IDIE, stall_IEME, stall_MEWB,
      regDest3, regDest4, regDest5, regAddrA2, regAddrB2, writeEnable3,
writeEnable4, writeEnable5,
      writeBackSel3, writeBackSel4, writeBackSel5, opcode2, jumpSignal4,
cp0Success);

endmodule
```

APPENDIX B: UPF FILE

```
# processor.upf
set_design_top processor
set_scope .

# Create Power Domains

# Fine Grained- alu1 dec1 stall1 coprocessor1 stage6 modules used
create_power_domain PD_CORE -elements {alu1 dec1 stall1 coprocessor1 stage6}
create_power_domain PD_TOP -include_scope

# Coarse Grained- mewb1 module used
create_power_domain PD_CORE -elements {mewb1}
create_power_domain PD_TOP -include_scope

# Define Power Supplies
create_supply_port VDD
create_supply_port VSS

# Power switch nets.
create_supply_net VDD_CORE_OUT

# Connect Supplies to Domains
create_supply_net VDD
create_supply_net VSS
```

```
set_domain_supply_net PD_TOP \
    -primary_power_net VDD \
    -primary_ground_net VSS

set_domain_supply_net PD_CORE \
    -primary_power_net VDD_CORE_OUT \
    -primary_ground_net VSS

connect_supply_net VDD -ports {VDD}
connect_supply_net VSS -ports {VSS}


set_level_shifter LS_CORE \
  -domain PD_CORE \
  -applies_to outputs \
  -rule low_to_high \
  -location parent

use_interface_cell lvls4 -strategy LS_CORE -domain PD_CORE -lib_cells
{LS_HLEN_X1 LS_HL_X1 LS_LHEN_X1 LS_LH_X1 LS_HLEN_X2 LS_HL_X2
LS_LHEN_X2 LS_LH_X2 LS_HLEN_X4 LS_HL_X4 LS_LHEN_X4 LS_LH_X4}

set_isolation ISO_CORE -domain PD_CORE -applies_to outputs -clamp_value 1 -
isolation_power_net VDD -isolation_ground_net VSS -elements {alu1 dec1}
```

```
set_isolation_control ISO_CORE -domain PD_CORE -isolation_signal rst_n -
isolation_sense high -location self

use_interface_cell isolation1 -strategy ISO_CORE -domain PD_CORE -lib_cells
{ISO_FENCE0N_X1 ISO_FENCE0_X1 ISO_FENCE1N_X1 ISO_FENCE1_X1
ISO_FENCE0N_X2 ISO_FENCE0_X2 ISO_FENCE1N_X2 ISO_FENCE1_X2
ISO_FENCE0N_X4 ISO_FENCE0_X4 ISO_FENCE1N_X4 ISO_FENCE1_X4}

# Power switch for PD_CORE domain
create_power_switch SW_CORE -domain PD_CORE -output_supply_port {V1
VDD_CORE_OUT} -input_supply_port {V2 VDD} -control_port {ms_sel pwr_sig} -
on_state {normal_working V2 {ms_sel}} -off_state {off_state {!ms_sel}}

map_power_switch SW_CORE -domain PD_CORE -lib_cells {HEADER_OE_X1
HEADER_OE_X2 HEADER_OE_X4 HEADER_X1 HEADER_X2 HEADER_X4}

add_port_state VDD -state {ON 0.95}
add_port_state SW_CORE/V1 -state {ON 0.95} -state {OFF 0.00}

# POWER STATE TABLE
create_pst PST -supplies {VDD VDD_CORE_OUT}
add_pst_state PS1 -pst PST -state {ON OFF}
add_pst_state PS2 -pst PST -state {ON ON}
```

APPENDIX C: DESIGN COMPILER FILE

```
#===== Set: make sure you change design name elsewhere in this file
set NameDesign "processor"

#===== Set some timing parameters
set CLK "clk"

#===== All values are in units of ns for NanGate 45 nm library
set clk_period      5.298

set clock_skew      [expr {$clk_period} * 0.05 ]
set input_setup     [expr {$clk_period} * 0.97 ]
set output_delay    [expr {$clk_period} * 0.04 ]
set input_delay     [expr {$clk_period} - {$input_setup}]

# It appears one "analyze" command is needed for each .v file. This works best
# (only?) with one command line per module.
analyze -format verilog processor.v
analyze -format verilog ALU.v
analyze -format verilog controller.v
analyze -format verilog coprocessor0.v
analyze -format verilog decoder.v
analyze -format verilog IDIE.v
analyze -format verilog IEME.v
analyze -format verilog IFID.v
analyze -format verilog instMemory.v
```

```
analyze -format verilog memory1.v
analyze -format verilog MEWB.v
analyze -format verilog pc.v
analyze -format verilog processor.v
analyze -format verilog register.v
analyze -format verilog stage6.v
analyze -format verilog stall.v

add_pg_pin_to_lib
elaborate $NameDesign
current_design $NameDesign
load_upf ./processor.upf
link
uniquify
set_max_area 0.0
if { [sizeof_collection [get_cells * -filter "is_hierarchical==true"]] > 0 } {
  ungroup -all -flatten -simple_names
  }
#===== Timing and input/output load constraints
create_clock $CLK -name $CLK -period $clk_period -waveform [list 0.0 [expr
{$clk_period} / 2.0 ] ]
set_clock_uncertainty $clock_skew $CLK
#set_clock_skew -plus_uncertainty $clock_skew $CLK
#set_clock_skew -minus_uncertainty $clock_skew $CLK
```

```
set_input_delay     $input_delay  -clock $CLK [all_inputs]
#remove_input_delay               -clock $CLK [all_inputs]
set_output_delay    $output_delay -clock $CLK [all_outputs]

set_load 1.5 [all_outputs]

set_voltage 1.25 -object_list VDD
set_voltage 1.25 -min 1.06 -object_list VDD_CORE_OUT
set_voltage 0.0 -object_list VSS

compile_ultra -scan -gate_clock

# Comment "ungroup" line to maybe see some submodules
if { [sizeof_collection [get_cells * -filter "is_hierarchical==true"]] > 0 } {
  ungroup -all -flatten -simple_names
  }
# compile -map_effort medium

#===== Reports
write -format verilog -output processor.vg -hierarchy $NameDesign
write_sdc        processor.sdc
write_sdf        processor.sdf
report_area              > processor.area
report_cell              > processor.cell
report_hierarchy         > processor.hier
```

```
report_net              > processor.net
report_power            > processor.pow
report_timing -nworst 10  > processor.tim
check_timing
check_design
exit
```

APPENDIX D: INNOVUS FILES

MMMC View File

```
# Version:1.0 MMMC View Definition File
# Do Not Remove Above Line
create_library_set -name fast -timing
{/synopsys/Nangate_FreePDK45/NangateOpenCellLibrary_PDKv1_3_v2010_12/Front_
End/Liberty/CCS/NangateOpenCellLibrary_fast_ccs.lib
/synopsys/Nangate_FreePDK45/NangateOpenCellLibrary_PDKv1_3_v2010_12/Low_Po
wer/Front_End/Liberty/CCS/LowPowerOpenCellLibrary_fast_ccs.lib}
create_library_set -name typical -timing
{/synopsys/Nangate_FreePDK45/NangateOpenCellLibrary_PDKv1_3_v2010_12/Front_
End/Liberty/CCS/NangateOpenCellLibrary_typical_ccs.lib
/synopsys/Nangate_FreePDK45/NangateOpenCellLibrary_PDKv1_3_v2010_12/Low_Po
wer/Front_End/Liberty/CCS/LowPowerOpenCellLibrary_fast_ccs.lib}
create_library_set -name worst -timing
{/synopsys/Nangate_FreePDK45/NangateOpenCellLibrary_PDKv1_3_v2010_12/Front_
End/Liberty/CCS/NangateOpenCellLibrary_worst_low_ccs.lib
/synopsys/Nangate_FreePDK45/NangateOpenCellLibrary_PDKv1_3_v2010_12/Low_Po
wer/Front_End/Liberty/CCS/LowPowerOpenCellLibrary_fast_ccs.lib}
create_constraint_mode -name synth -sdc_files {processor.sdc}
create_delay_corner -name fast -library_set {fast}
create_delay_corner -name typical -library_set {typical}
create_delay_corner -name worst -library_set {worst}
create_analysis_view -name fast -constraint_mode {synth} -delay_corner {fast}
create_analysis_view -name typical -constraint_mode {synth} -delay_corner {typical}
create_analysis_view -name worst -constraint_mode {synth} -delay_corner {worst}
```

```
set_analysis_view -setup {worst} -hold {fast}
```

Globals File

```
#set the HDL code that you are trying to place and route, this should be a synthesized
gate-level netlist
set init_verilog {processor.vg}

#set the name of the top cell
set init_top_cell {processor}

#set the input timing file, called a .VIEW file, which should be generated based on timing
libraries and timing constraints from synthesis
set init_mmmc_file {template.view}

#set the power and ground names
set init_gnd_net {VSS}
set init_pwr_net {VDD_CORE_OUT}
set init_pwr_net {VDD}

#set the physical libraries, which Cadence uses the LEF type
set init_lef_file [list
/synopsys/Nangate_FreePDK45/NangateOpenCellLibrary_PDKv1_3_v2010_12/Back_E
nd/lef/NangateOpenCellLibrary.lef
/synopsys/Nangate_FreePDK45/NangateOpenCellLibrary_PDKv1_3_v2010_12/Low_Po
wer/Back_End/lef/LowPowerOpenCellLibrary.lef]
```

```
#generated commands that likely shouldn't be changed unless you know what you are
doing
set ::TimeLib::tsgMarkCellLatchConstructFlag 1
set conf_qxconf_file {NULL}
set conf_qxlib_file {NULL}
set defHierChar {/}
set distributed_client_message_echo {1}
set distributed_mmmc_disable_reports_auto_redirection {0}
set enc_enable_print_mode_command_reset_options 1
set init_design_settop 0
set latch_time_borrow_mode max_borrow
set pegDefaultResScaleFactor 1
set pegDetailResScaleFactor 1
set report_inactive_arcs_format {from to when arc_type sense reason}
set timing_enable_default_delay_arc 1
```