



**Senior Design I**  
*90 Page Report Revision*  
**Northrop Grumman Group**

**Authors**

**Davi Dantas**  
*Electrical Engineering*

**Evan Kasky**  
*Computer Engineering*

**Nicolas Sayed**  
*Electrical Engineering*

**Pablo Rodriguez**  
*Computer Engineering*

**Timothy Ogg**  
*Computer Engineering*

**Timothy Ogg**  
*Electrical Engineering*

**Mentor**

**Dr. Mike Borowczak**

**Mentor**

**Dr. Chung Yong Chan**

**Review Committee**

**Aaron Lingerfelt**  
*Northrop Grumman*

**Alberto Reategui**  
*Northrop Grumman*

**Marlon Fuentes**  
*Northrop Grumman*

# Table Of Contents

<b>1</b>	<b><u>EXECUTIVE SUMMARY</u></b>	<b>5</b>
<b>2</b>	<b><u>PROJECT DESCRIPTION</u></b>	<b>6</b>
2.1	PROJECT BACKGROUND AND MOTIVATION	6
2.2	ENGINEERING SPECIFICATIONS	7
2.3	HARDWARE DIAGRAM	7
2.4	SOFTWARE DESIGN DIAGRAM	8
2.5	HOUSE OF QUALITY	8
2.6	EXISTING PRODUCTS USED TO IDENTIFY PROJECT FEATURES	9
2.6.1	GREENWAVES GAP8	9
2.6.2	ESPRESSIF ESP32-C3	10
2.6.3	OTHER PRODUCTS	12
2.6.4	SUMMARY	12
2.7	GOALS AND OBJECTIVES	12
<b>3</b>	<b><u>RESEARCH</u></b>	<b>13</b>
3.1	INTRODUCTION TO TECHNOLOGIES	13
3.2	COMPUTER ARCHITECTURE	16
3.2.1	VON NEUMANN ARCHITECTURE VS. HARVARD ARCHITECTURE	17
3.2.2	MICRO-ARCHITECTURES	18
3.2.3	TEAM DESIGN OPTIONS	23
3.2.4	OPTION A – TEAM DESIGNS 5 STAGE	24
3.2.5	OPTION B – TEAM DESIGNED SINGLE STAGE	26
3.2.6	OPTION C – 5 STAGE W/ HAZARD AND FORWARDING UNITS	26
3.2.7	OPTION D – RIPES.ME	27
3.2.8	MEMORY ARCHITECTURE SELECTIONS	28
3.3	PIPELINE HAZARDS	30
3.3.1	TYPES OF HAZARDS	30
3.3.2	IN-DEPTH ANALYSIS	30
3.3.3	SOFTWARE AND HARDWARE HAZARD CONTROL	32
3.3.4	CUSTOM ASSEMBLER TO MITIGATE CONTROL HAZARDS	35
3.4	DYNAMIC BRANCH PREDICTION	38
3.4.1	BRANCH PREDICTION BUFFER	39
3.4.2	CORRELATING BRANCH PREDICTOR	40
3.4.3	TOURNAMENT PREDICTOR	40
3.5	CACHE AND OPTIMIZATION	40
3.5.1	OPTIMIZATION	40
3.5.2	D-CACHE BLOCK DIAGRAM	42

<b>3.6</b>	<b>INSTRUCTION SET ARCHITECTURES</b>	<b>43</b>
3.6.1	x86	43
3.6.2	ARM	44
3.6.3	MIPS	44
3.6.4	RISC-V	45
<b>3.7</b>	<b>DESIGN LANGUAGES</b>	<b>47</b>
3.7.1	VHDL	47
3.7.2	VERILOG	49
3.7.3	SYSTEM VERILOG	50
<b>3.8</b>	<b>VERIFICATION METHODS AND TIMING</b>	<b>52</b>
3.8.1	CONSTRAINED RANDOM VERIFICATION	52
3.8.2	ASSERTION BASED VERIFICATION	53
3.8.3	COVERAGE DRIVEN VERIFICATION	54
3.8.4	UNIVERSAL VERIFICATION METHODOLOGY	54
3.8.5	TIMING ISSUES IN DIGITAL CIRCUITS	55
<b>4</b>	<b><u>RELATED STANDARDS AND DESIGN CONSTRAINTS</u></b>	<b><u>57</u></b>
<b>4.1</b>	<b>RELATED STANDARDS</b>	<b>57</b>
4.1.1	RISC-V32I ISA	57
4.1.2	IMPACT OF RISC-V32I ISA ON DESIGN	57
4.1.3	IEEE 1800-2017 (SYSTEMVERILOG)	58
4.1.4	IMPACT OF IEEE 1800-2017 (SYSTEMVERILOG) ON DESIGN	58
4.1.5	IEEE 1800.2-2020 (UNIVERSAL VERIFICATION METHODOLOGY)	58
4.1.6	IMPACT OF IEEE 1800.2-2020 (UVM) ON DESIGN	59
4.1.7	FREEPDK45	59
4.1.8	IMPACT OF FREEPDK45 ON DESIGN	59
<b>4.2</b>	<b>SOFTWARE CONSTRAINTS</b>	<b>60</b>
<b>4.3</b>	<b>MANUFACTURABILITY CONSTRAINTS</b>	<b>60</b>
<b>4.4</b>	<b>ECONOMIC CONSTRAINTS</b>	<b>60</b>
<b>4.5</b>	<b>TIME CONSTRAINTS</b>	<b>61</b>
<b>5</b>	<b><u>ARTIFICIAL INTELLIGENCE</u></b>	<b><u>62</u></b>
<b>5.1</b>	<b>CHAT GPT CASE STUDIES</b>	<b>62</b>
5.1.1	CASE STUDY 1	63
5.1.2	CASE STUDY 2	63
5.1.3	CASE STUDY 3	63
5.1.4	CASE STUDY 4	64
<b>6</b>	<b><u>ADMINISTRATION CONTENT</u></b>	<b><u>64</u></b>

<b>6.1</b>	<b>BUDGET AND FINANCING</b>	<b>64</b>
<b>6.2</b>	<b>PHASE ONE MILESTONES</b>	<b>64</b>
<b>6.3</b>	<b>PHASE TWO MILESTONES</b>	<b>65</b>
<b>6.4</b>	<b>WORK DISTRIBUTIONS</b>	<b>66</b>
<b>6.5</b>	<b>INTRODUCTION TO AGILE METHODOLOGY</b>	<b>67</b>
<b>6.6</b>	<b>DELIVERABLE REQUESTS</b>	<b>68</b>
6.6.1	OVERVIEW OF FIRST DELIVERABLE REQUEST	68
6.6.2	RESULTS OF FIRST DELIVERABLE REQUEST	70
6.6.3	INITIAL BLOCK DESIGN DELIVERABLE REQUEST	70
6.6.4	EXPECTATIONS FOR THE TEAM MEETING	70
6.6.5	DIVISION OF LABOR FOR INITIAL BLOCK DESIGN	71
6.6.6	DISCUSSION OF RESULTS FOR INITIAL BLOCK DESIGN, SECOND DELIVERABLE REQUEST	71
6.6.7	OVERVIEW OF RTL DIVISION OF LABOR	71
6.6.8	REVISED TIMELINE (ADJUSTED FOR SETBACKS)	73
6.6.9	KEY MILESTONES AND SPECIFIED DIVISION OF LABOR	74
<b>6.7</b>	<b>OVERVIEW OF IMPLEMENTATION OF AGILE</b>	<b>75</b>
<b><u>7</u></b>	<b><u>APPENDIX A: CITATIONS</u></b>	<b><u>76</u></b>
<b><u>8</u></b>	<b><u>APPENDIX B: AI</u></b>	<b><u>79</u></b>

# 1 Executive Summary

As technology continues to grow and remains a dominant factor that influences everyday life for both consumers and business, the demand for skilled professionals in Application-Specific Integrated Circuit (ASIC) and semiconductor engineering continues to grow annually. This project, sponsored by Northrop Grumman, aims to address the need for skilled ASIC engineers by engaging with a student group to develop a comprehensive digital ASIC tapeout. The main goal of this project is to implement a RISC-V32I core using the FreePDK45 technology node using industry standard Electronic Design Automation (EDA) tools. Through hands-on experience, our group as well as the sponsor aims to give us the practical skills required to enter the semi-conductor industry.

This paper will detail each stage of our design process, which includes RTL design, synthesis, post-synthesis verification, implementation, post-implementation verification, and layout. The paper will document our design processes, including the technologies used, both modern industry standards, as well as technologies the group is already familiar with. It will document the path we took to successfully develop and deliver a digital ASIC tapeout as well as any research, and issues we encountered during our design phases.

At the core of this project is the RAPID microarchitecture, which targets a performance metric of 100 million instructions per second (MIPS). Our team has opted to for a 5-stage RISC-V32I pipeline over simpler architectures such as single-cycle, allowing for a significant improvement in performance by increasing instruction throughput, as well as utilizing parallelism while still meeting the strict performance metrics required. We chose the pipeline architecture as it offers a good balance between the speed we need to operate at, as well as power efficiency.

Throughout our design process, rigorous verification and analysis must be conducted not only to validate our designs integrity but also to ensure our design meets industry standards. To ensure our design is both valid and up to standards, we will use the Universal Verification Methodology (UVM) for most of our testbenches as this standard allows us to reuse and scale testbenches as needed. This offers a robust framework to ensure our design is upheld to the highest regards and meets all our performance metrics.

In summary, this project aims to take our team through the entire ASIC design process, from the early stages of planning and development, all the way through the verification and layout stages of our design. Northrop Grumman's sponsorship provides an invaluable opportunity for our team to gain hands-on experience using industry standard tools and design best practices and prepares us to enter the semi-conductor industry upon graduation. This paper will document our journey as we go through the stages of development, providing insight and critical details about the considerations and decisions made

## 2 Project Description

### 2.1 Project Background and Motivation

Our senior design group is challenged with designing an Application-Specific Integrated Circuit (ASIC), specifically focused on the design and synthesis of a RISC-V32I Central Processing Unit (CPU) core. A complete CPU comprises several computational units, along with various interconnects to external hardware such as RAM and PCIe devices. Our project zeroes in on the core computational unit of the CPU, which is responsible for executing the user's instructions—ranging from logic and arithmetic operations to memory read/write functions. These fundamental tasks lie at the heart of modern digital computing.

Our CPU core will implement the RISC-V32I Instruction Set Architecture (ISA), which defines the interface between software and hardware. The RISC-V32I ISA is an open-source standard that already has established compiler support. This is a key advantage, as it allows us to leverage existing tools such as operating systems, compilers, linkers, APIs, and software. By utilizing a well-supported ISA, we sidestep one of the primary challenges to adopting new CPU architecture: the availability of toolchain support. A powerful CPU architecture can be held back if the necessary development tools—compilers, debuggers, and other software—are not readily available. Our choice of the RISC-V32I ISA ensures that we can tap into a robust ecosystem of existing technologies, facilitating a smoother development process and increasing the potential for adoption.

Semiconductor fabrication and design are integral to the United States' economic stability, national security, and supply chain resilience. A disruption in the steady supply of chips can have significant consequences, as demonstrated by the global semiconductor shortage triggered by the COVID-19 pandemic in 2020. Industries ranging from defense to automotive manufacturing experienced substantial setbacks due to port closures in key production regions. This disruption highlighted the dependency of modern industries on a reliable supply of chips. For example, the automotive sector faced production halts as critical components became unavailable, and the defense industry experienced delays in both maintenance and production of essential equipment. This project aims to address such vulnerabilities by advancing the design of Application-Specific Integrated Circuits (ASICs), contributing to a more secure and self-sustaining semiconductor ecosystem.

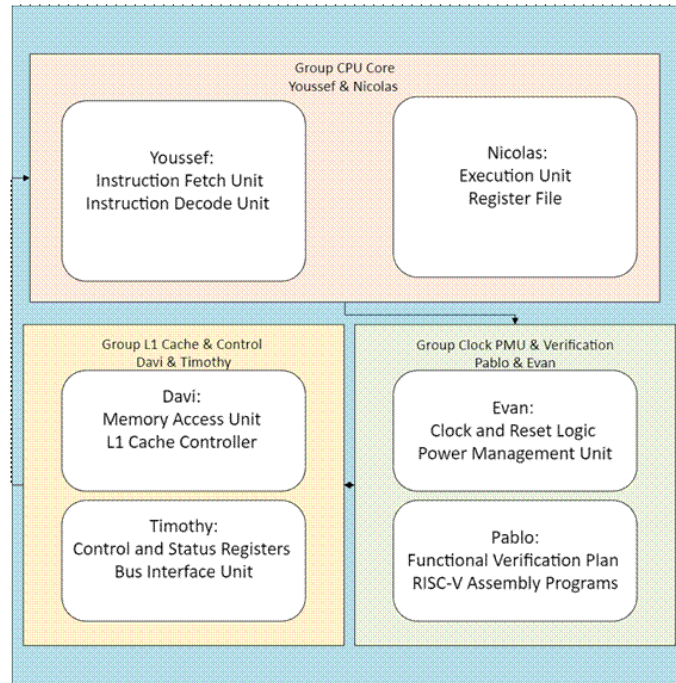
This project aims to contribute to the growth of a more resilient semiconductor industry in the United States by providing UCF students with hands-on, industry-level experience in ASIC design. By equipping students with the skills necessary for ASIC development, this initiative helps cultivate a larger pool of specialized talent within the U.S. semiconductor workforce. Additionally, this project establishes a pipeline of UCF graduates who are well-prepared to pursue careers in ASIC design, strengthening the domestic talent base and supporting the nation's technological leadership.

## 2.2 Engineering Specifications

*Table 2-1: Design Requirements and Specs*

No.	Requirement	Specification	Description	Priority
1	Processing Speed	$\geq 100$ MIPS	The design should process at least 100 MIPS.	H
2	RV32I Compatibility	$\geq 38$ Instr.	Handle at least 38 unique instructions, excluding memory ordering and environment calls.	H
3	Low Power Consumption	$\leq 1$ W	Efficient design meeting specs under 1W power.	H
4	Cache Size	$\geq 16$ MB	Incorporate an L1 cache of at least 16 MB.	H
5	Process Node	$\leq 45$ nm	Implemented with a process size of at most 45nm.	H
6	Operating Voltage Range	0.95 V - 1.25 V	Operate correctly within this voltage range	H
7	Operating Temperature Range	0 °C - 125 °C	Operate correctly within this temperature range.	M
8	IR Drop	$\leq 5\%$	Voltage drops below 5% of expected voltage.	M
9	Timing Failure Count	0 Faults	No timing faults over any operating period.	H

## 2.3 Hardware Diagram



*Figure 2-1: Hardware Design Diagram*

## 2.4 Software Design Diagram

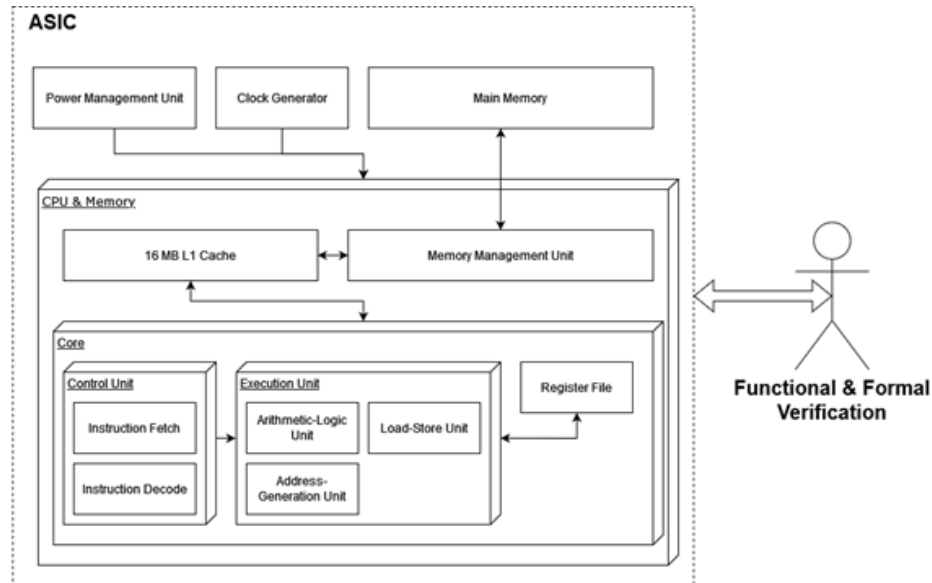


Figure 2-2: Software Design Diagram

## 2.5 House of Quality

<b>Correlations</b> Positive + Negative - No Correlation O		<b>Relationships</b> Strong ● Moderate ○ Weak ▽		<b>Direction of Improvement</b> Maximize ▲ Target ◇ Minimize ▼					
Column #	1	2	3	4	5	6	7	8	9
Direction of Improvement	▲	◇	▼	▲	▼	◇	◇	▼	◇
Customer Requirements (Explicit and Implicit)	Processing Speed	RV32i Compatibility	Power Consumption	Cache Size	Process Node	Operating Voltage Range	Operating Temperature Range	IR Drop	Timing Failure Count
Compatible	▽	●	▽	▽	▽	●	●	▽	▽
Efficient	●	○	●	▽	●	●	▽	○	▽
Performance	●	●	▽	●	▽	▽	▽	○	●
Reliable	▽	▽	▽	▽	▽	●	●	●	●
Target	≥ 100 MIPS	≥ 38 unique instructions	≤ 1W	≥ 16 MB	≤ 45nm	0.95 V - 1.25 V	0 °C - 125 °C	≤ 5%	0 Faults

Figure 2-3: House of Quality



## 2.6 Existing Products Used to Identify Project Features

To better understand our goals and objectives in this project, we need to dive into history to have context around what is trying to be accomplished. In today's world, most computers and phones are based on the Instruction Set Architecture (ISA) of x86. The x86 is a closed-source ISA that requires users to pay to be able to use the software. An x86 can be categorized as a Complex Instruction Set Computer (CISC), so it has a very large set of instructions that can perform complex operations. There are several complex operations that are able to perform such as Load String, Move with Sign-Extension, and Repeat String Move. All these operations would not be as simply written in a Reduced Instruction Set Computer (RISC) architecture.

In x86 architecture, the variable length instructions are between 1 and 15 bytes, which allows for diverse functions in exchange for added complexity to instruction decoding. x86 typically has a smaller number of general-purpose registers which could require more frequent memory access, which would make it slower.

An alternative today to the x86 is the ARM processor which is also a closed source ISA. ARM is a RISC architecture, and it has a fixed-length instruction set. This architecture aims to perform one instruction every cycle for better efficiency. ARM has more general-purpose registers than x86 so it can access memory faster. ARM architecture performs pipelining more efficiently because of its simpler nature.

RISC-V is an open-source ISA, making this software free to design and sell processors with RISC-V ISA without constraints. Regarding architecture, RISC-V is more closely related to ARM. Originally there was a RISC-I created at the University of California Berkeley. There are 32-bit options as well as 64-bit with the option of extending it to 128-bit. The most basic ISA version within every category is the base integer (e.g. RV32I and RV64I) There are extensions to expand the features such as:

- A - Atomic Instruction
- M- Integer Multiplication and Division,
- F - Single-Precision Floating Point
- D - Double-Precision Floating Point
- Q - QuadPrecision Floating-Point
- C - Standard Extension for compressed Instructions.

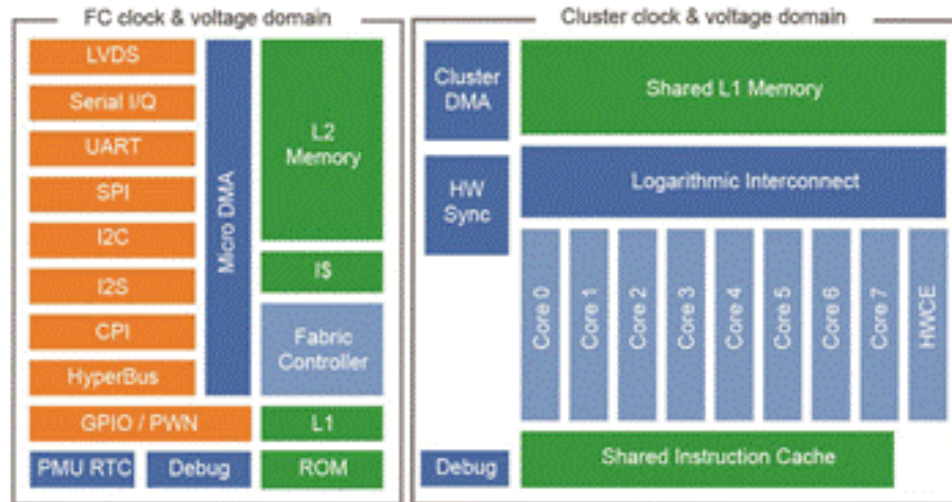
Several commercially available products make use of RISC-V due to it being an open instruction set architecture (ISA) suitable for use in academia, research, and capable of being directly implemented in hardware. The ISA is separated into the RV32I Base Integer Instruction and supports several extensions for additional functionality. RISC-V implementations that only meet the RV32I set are adequate for simple ASICs or educational purposes. These products give an insight into the common characteristics of RISC-V implementations.

### 2.6.1 GreenWaves GAP8

The GreenWaves GAP8 IoT Application Processor is a RISC-V multi-core, low power, battery operated data analysis processor capable of collecting data from image sources,

radar, sound and other signal sources. It is also capable of performing AI-driven workloads and is viable to perform tasks such as image recognition, speech recognition, and sensor monitoring.

This product can operate at up to 175 MHz, within an operating voltage range of 1.0 V to 1.2 V with an operating power of about 70 milliwatts. It is capable of delivering up to 200 MOPS within these limits. It implements a 512 kB L2 cache shared across its cluster of eight cores. It is fabricated with a 55nm process. The GAP8 has found uses in the application areas of drones, surveillance cameras, face detection, robotics, among others.



**GAP8 Block Diagram**

*Figure 2-4: GAP8 Clock and Voltage Domain*



*Figure 2-5: GAP8 Integrated Circuit*

### 2.6.2 Espressif ESP32-C3

The ESP32-C3 is a low-power MCU with a single 32-bit RISC-V processor that incorporates Wi-Fi and Bluetooth wireless communication interfaces. The product has seen success in industrial automatic, consumer electronics, audio processing devices, and sensor data loggers, among others use cases.

The MCU's recommended input voltage lies between 3.0 V and 3.6 V which drives the core up to 160 MHz. It incorporates a 16kB cache and has an average power consumption when active of about 80 milliwatts. This MCU is fabricated with a 40 nm process.

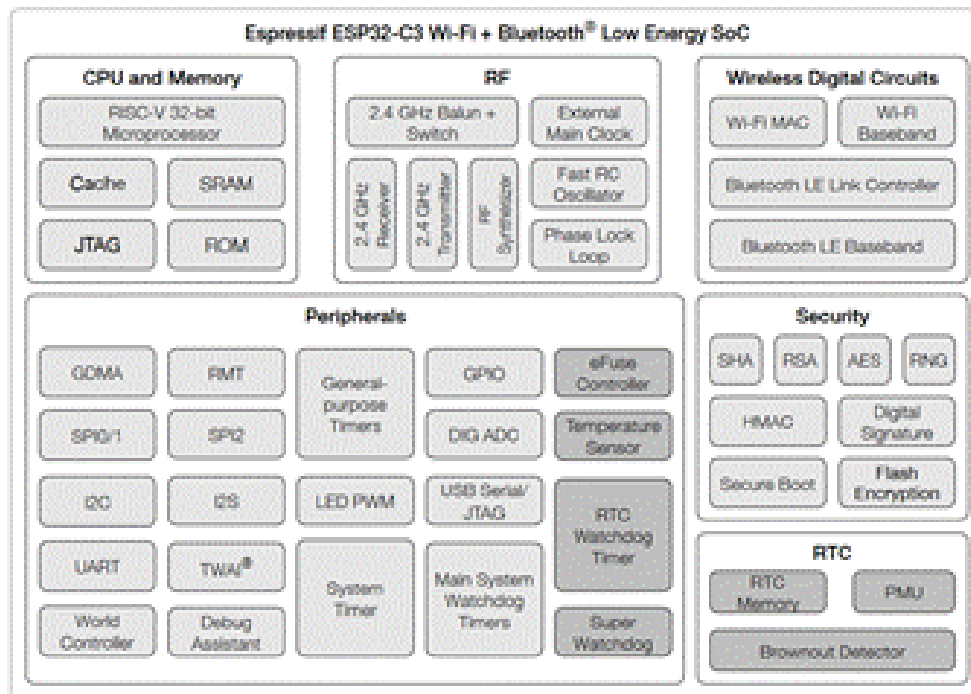


Figure 2-6: ESP32 Architecture



Figure 2-7: ESP32 Integrated Circuit

### 2.6.3 Other Products

Table 2-2: Comparison of Different RISC-V Processors

Model	Fabrication Process	Clock Speed	Miscellaneous Notes
SiFive FE310	180 nm	320 MHz	5-stage pipeline
Renesas RH850/U2B	28 nm	400 MHz	Single issue, 2-stage pipeline
Bouffalo Lab BL602/BL604	22 nm	192 MHz	3-stage pipeline
GigaDevice GD32VF103	-	108 MHz	Single issue, 2-stage pipeline

### 2.6.4 Summary

The aforementioned products reveal some of the common aspects of commercially available RISC-V SoCs. First, they appear to be well-suited for signal processing, so they are a natural choice to pair with sensors for the purposes of data logging. Second, RISC-V SoCs typically operate at clock speeds between 100 MHz and 400 MHz and are able to operate within 100 milliwatts for the lower clock speeds. Third, instruction-level parallelism using pipelines is always implemented in order to achieve higher throughput, however, super-scalar designs are uncommon for this range of processors.

## 2.7 Goals and Objectives

The goal of this project is for students to successfully design an Application Specific Integrated Circuit (ASIC) that implements the RISC-V Instruction Set Architecture, utilizing industry-standard electronic design automation (EDA) tools and relevant Cadence software. The project is divided into two phases: the Fall phase and the Spring phase, with specific objectives outlined for each stage.

In Phase 1 (Fall 2024), the project involves designing an ASIC that implements the RISC-V RV32I base instruction set, excluding memory ordering and environment/system calls. The design must undergo functional and/or formal verification to confirm its functionality, utilizing the Universal Verification Methodology (UVM) as a standardized approach, with a target of achieving a minimum of 85% code coverage. The goal is to successfully tape out an ASIC that implements the RISC-V instruction set, using existing electronic design automation (EDA) tools and relevant design technology libraries, ideally at a process node of approximately 45nm or smaller. The team is expected to gain knowledge about the ASIC design flow, become proficient with Cadence tools, enhance troubleshooting skills, and develop thorough project documentation. To validate the performance and system-level functionality of the design, the team will create and execute test cases in RV32I assembly.

In Phase 2 (Spring 2025), the project will involve conducting static timing analysis (STA), IR drop analysis, and electromigration analysis to ensure that the physical performance requirements are met. The targets for this phase include a maximum IR drop of 5%, no

timing failures, and maintaining the current density within acceptable limits. Additionally, the finalization of functional and/or formal verification of the design will be completed, ensuring that the target code coverage is achieved. Physical design techniques such as design rule checking (DRC) will be used to confirm the manufacturability of the design, and layout versus schematic (LVS) checks will be performed to ensure the correctness of the physical design.

The stretch goals for the project include inserting design for test (DFT) structures to validate the functionality of the manufactured IC. Additionally, gate-level simulations will be performed to verify the design with real clock delays. Logic equivalence checking will also be used to ensure functional equivalence between different stages of the development process.

Regarding deliverables for phase 1 and phase 2 are as follows: Mock tapeout a chip using a maximum 45 nm technology with Cadence digital design, verification, and signoff tools. In Phase 1, the deliverables should include the RTL design, initial verification metrics, and the synthesized netlist. In Phase 2, deliverables should include finalized verification metrics, a GDSII file, a clean timing report, and IR drop and electromigration within specifications. Leverage Cadence resources, including training materials and Rapid Adoption Kits (RAKs), to familiarize students with the ASIC design flow and accelerate project completion.

## 3 Research

### 3.1 Introduction to Technologies

The silicon industry has evolved significantly from the early days of discrete TTL/CMOS logic into the vast field of VLSI (Very Large-Scale Integration), where millions or even billions of transistors are integrated into a single chip. This exponential growth has enabled multiple technological approaches to problem solving, each with its own set of advantages and drawbacks. To better understand the distinctions between these technologies, it's essential to define their unique characteristics.

When faced with a digital problem, a designer may choose to create an Application Specific Integrated Circuit (ASIC). An ASIC is a custom-designed chip built for a single, specific function. The main advantage of this approach is that the designer has complete control over the chip's architecture, allowing for a highly optimized solution that delivers superior performance. No other solution—be it a CPU, GPU, or FPGA—can match the efficiency of an ASIC for its designated task. However, it's rare for businesses to pursue ASIC designs, primarily due to the prohibitively high manufacturing costs.

The cost barrier arises from the complex and time-consuming semiconductor fabrication process. It takes several months for a silicon wafer to complete its journey through the entire fabrication cycle, making it a high-risk investment in today's competitive market.

For instance, companies like Apple release new consumer electronics annually, often featuring higher-performing chips. Their product timelines are narrow, and delays can result in lost market opportunities as competitors may already be working on the next generation of products. Additionally, if an error is found during design or verification, the entire fabrication process must restart—incurring not only significant financial costs but also a potential loss in market share. Customers demand cutting-edge technology, and no one wants to buy last year's model.

This risk is heightened because verifying a chip is completely functional and bug free is nearly impossible. The solution space is vast, far exceeding what current computing power can feasibly explore in a reasonable timeframe. As a result, verification processes are becoming more rigorous and demanding.

A close alternative to the ASIC approach is the Field Programmable Gate Array (FPGA) approach. An FPGA is comprised of look up tables (LUTs), Multiplexers, and switching hubs. Through these relatively simple blocks, any digital logic circuit can be implemented on an FPGA. The usage of FPGA is much higher compared to ASIC design approaches because of the reduced cost. The greatest benefit of FPGA is that it is a reprogrammable circuit. FPGA is the only chips that can receive software updates to fix or add new features to existing products. This is useful for the telecommunications industry, and for their use case it is useful since many updates are released for communication standards and it would be very costly to purchase new hardware for every change in the standards.

One of the strengths of FPGAs compared to other technologies is that it has a high throughput and supports a large degree of parallelism compared to a CPU based platform. FPGA have an impressive throughput rate that can reach into the 17 Terabytes/second region, but they do have limited block RAM which can restrict their performance. This is possible because FPGA are built with large amount of I/O (Input/Output) ports to ingest and output multiple data streams. This is only possible for highly parallel applications and is done by instantizing multiple logic blocks that can process the incoming data streams simultaneously

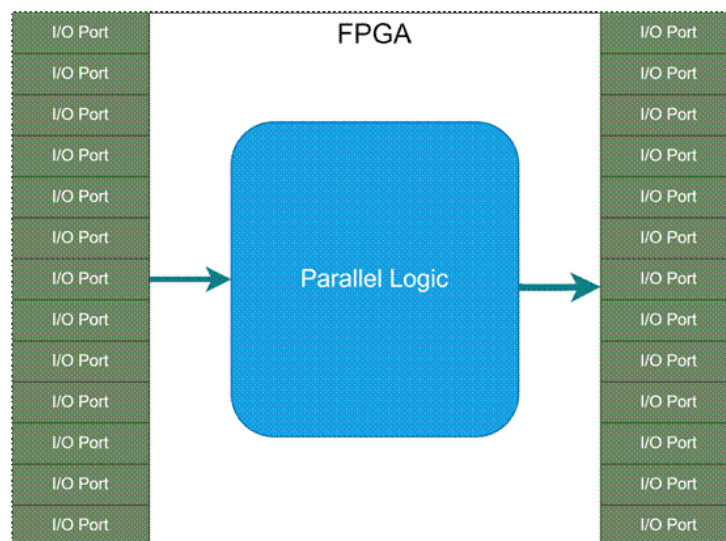
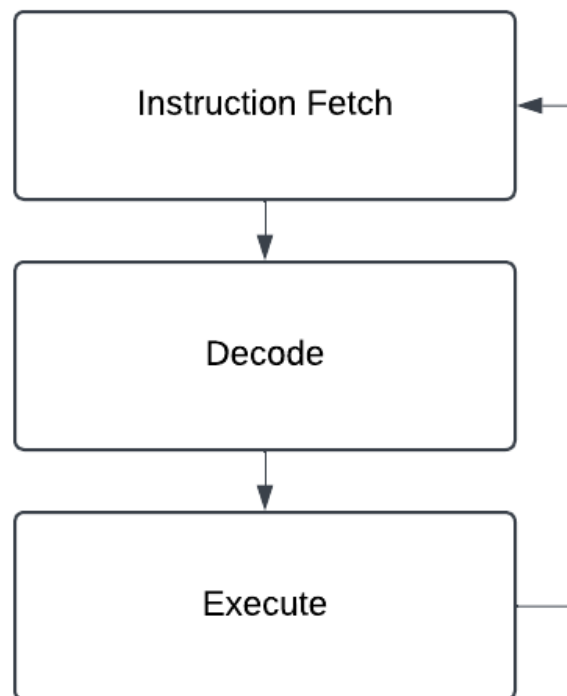


Figure 3-1: I/O Flow

There are two schools of thought when performing computational work. The first of which is spatial computation and the second is temporal computation. The first approach of digital computations was through the use of temporal computation done in the form of a Central Processing Unit or CPU. The idea behind a CPU is that we can split large problems or tasks into smaller pieces which can be broken down into even smaller assembly instructions. This way a problem can be solved using the same piece of hardware which makes CPUs great Von Nouman machines. This was important in the early stages of industry when transistor costs were on the scale of dollars. In those days, silicon design engineers were tasked with reducing the amount of logic gates needed to reduce the cost of fabrication. Reduced logic gates would also reduce the die size which would directly lead to more dies per wafer.

The consequence of CPU-based design is that we are hardware limited, that is, we have queue operations leading to degraded performance. This trade-off is fine for certain scenarios, such as personal computing, sequential tasks, and batch processing computation. The following diagram shows a high-level view of the basic principle behind CPU operation



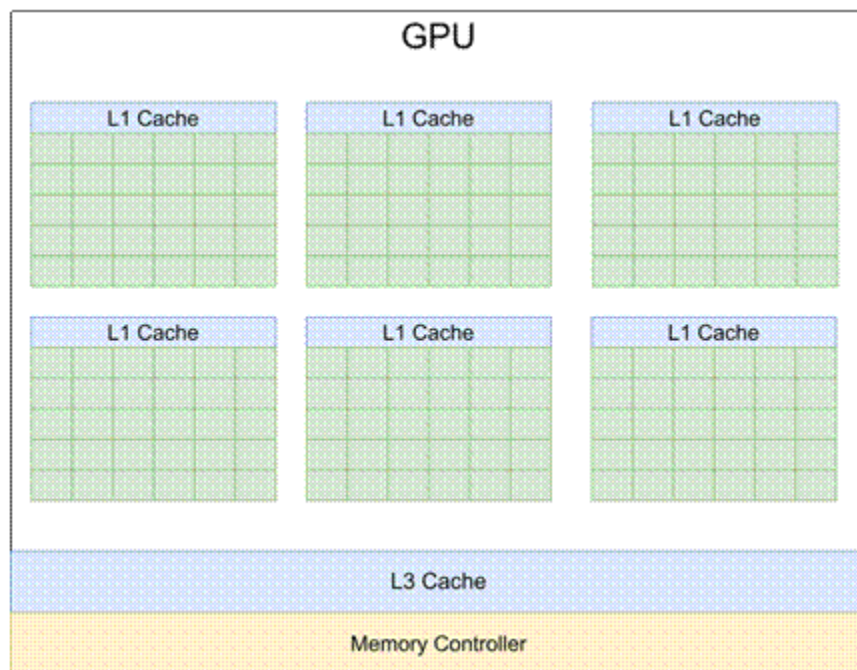
*Figure 3-2: CPU Flow Stages*

Spatial computing is where we have dedicated hardware for each operation in the design problem. For example, in a purely temporal approach a single adder may be used in the execute stage of a CPU but in a spatial approach there can be multiple adders that allow parallelism at the cost of more logic.

A General Processing Unit (GPU) is an example of spatial computation. Unlike a CPU, which excels in handling sequential tasks, the GPU is designed to handle highly parallel workloads. Certain problems can be divided into many smaller, independent tasks that can

be executed simultaneously, making them ideal for GPUs. Modern GPUs consist of thousands of cores, each capable of performing its own computations in parallel. This architecture contrasts sharply with the CPU's temporal approach, where a limited number of cores process tasks in a sequential manner.

GPUs exploit this parallelism to achieve high performance in tasks such as graphics rendering, scientific simulations, and machine learning. Each GPU core operates independently, but they also take advantage of SIMD (Single Instruction, Multiple Data) instructions, which allow multiple data elements to be processed with a single instruction. For example, SIMD enables the GPU to perform a mathematical operation on multiple 32-bit floating-point registers simultaneously, allowing it to process data far more efficiently than a CPU in specific parallel applications. This ability to perform the same operation on multiple data points at once is critical in fields where data throughput is essential, such as rendering or matrix computations.



*Figure 3-3: GPU Memory Diagram*

This architecture, however, also means that GPUs are less flexible than CPUs for general-purpose computing tasks. While the GPU excels at processing large amounts of data in parallel, it is less efficient at handling tasks that require complex control logic or are heavily dependent on sequential processing. This trade-off is what makes GPUs highly specialized, as they perform extremely well in specific domains but are not as versatile as CPUs in a broader range of applications.

## 3.2 Computer Architecture

Computer architecture is a broad term which encompasses the design and organization of computers, but no matter how broad the field is, all the engineering that goes into computer architecture aims to optimize performance, efficiency, and viability. There are a few main



categories which represent computer architecture and those are instruction set architectures, microarchitectures, logic design, and implementation. Depending on the project's needs engineers may abstract these different levels of design, but for our project we will need to understand concepts from the system level down to the transistor level. In this chapter we will cover the information which has led us to how we will design, test, and implement our ASIC.

### 3.2.1 Von Neumann Architecture vs. Harvard Architecture

A computer at a very high level is simply comprised of a processing unit, memory, and I/O. Two architectures which describe how those components interact are the Harvard and Von Neumann architectures. These two models do not particularly differ in how I/O is handled but they do have different approaches regarding how the processor interacts with memory. Depending on the task either model can be used, but in general the Von Neumann architecture is more often used for general purpose computing while the Harvard architecture is more often used for lightweight embedded systems.

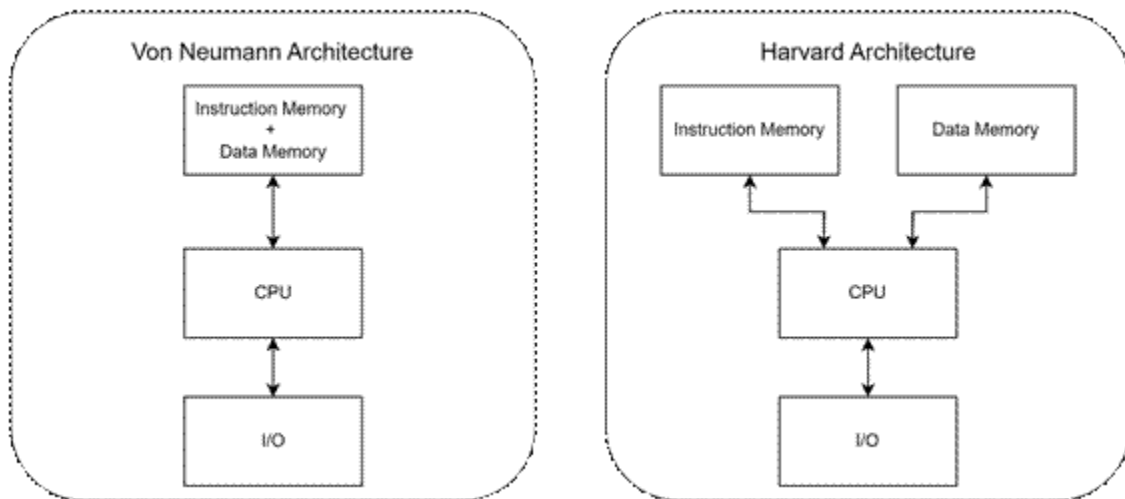


Figure 3-4: Von Neumann vs Harvard Memory Architectures

The Von Neumann architecture is the older of the two designs and was defined in the mid-1940s. In a Von Neumann based processor instructions and data share the same memory space, this means that the same bus is used when fetching instructions and reading or writing data. The Von Neumann design is most often used because only having to work with a single memory space allows for simpler and more flexible programming, but in engineering there is no free lunch. The main issue with the Von Neumann architecture quickly appears when you consider what would happen if you wanted to fetch an instruction and read or write from memory at the same time, the processor must stall for a cycle. This issue can be easily solved but the solution results in a scenario where bus availability becomes one of the main bottlenecks for the processor and in modern computing the largest overhead is generally in memory.

The Harvard architecture offers a solution to the bottleneck caused by the single bus of the Von Neumann architecture by splitting the memory space into instruction memory and data

memory. This change does not necessarily double the processor's speed but by implementing Harvard architecture we are able to increase efficiency while also making physical layout and verification simpler.

Depending on how memory is integrated into a computer system you may also choose to implement a modified Harvard architecture. The modified Harvard architecture is a sort of hybrid between the Von Neumann architecture and the 21 Harvard architecture where the cache is accessed with the separate buses, but other memory units are accessed in a Von Neumann style. Many modern processors such as Intel desktop processors rely on a modified Von Neumann architecture, but for our design we can ignore this as we will be relying on a single, large L1 cache.

*Table 3-1: Von Neumann vs Harvard vs Modified Harvard*

<b>Category</b>	<b>Von Neumann</b>	<b>Harvard</b>	<b>Modified Harvard</b>
<b>Memory Structure</b>	Single memory space which shares data and instructions	Separate memory space for data and instructions	Separate Data and Instruction Cache with a unified main memory space
<b>Bus Structure</b>	Single Bus	Dual bus	Dual Bus & Shared Bus
<b>Simplicity</b>	Simpler hardware	Simpler programming	Middle ground
<b>Cost</b>	Simpler leads to a lower cost	Additional hardware leads to higher cost	Middle ground
<b>Performance</b>	Possible memory bottleneck leads to less performance	Higher memory bandwidth leads to better pipeline	Almost achieves the same performance as standard Harvard

### 3.2.2 Micro-architectures

All processors have several fundamental units that work together to complete an instruction, and these units include the following components: Fetch Unit, Decode Unit, Execution Unit, Memory Management Unit, and Control Unit. These different units can be further split down into individual components but often the main difficulty when targeting performance is how these units work together.

#### Single Cycle

The overall clock cycle of the design in a single cycle processor is determined by the instructions that take the longest time to execute. This is usually a memory access instruction such as load or store. A simple add instruction usually completes quickly, however, since all instructions share the same clock period, the processor cannot take advantage of such quick completion. The control path design for the single cycle is also quite simple since all instructions are executed in one cycle. A control unit is responsible

for generating the control signals depending on the incoming opcodes of the instructions being processed.

In a typical single cycle design the data path would include an Instruction Memory, Program Counter (PC), Register File, ALU, and Data Memory. The Instruction Memory stores the program instructions. The Program Counter points to the current instruction from the list of instructions in memory. The Register File stores the data that the instruction will operate on. ALU performs arithmetic and logic operations. Data Memory handles load and store instructions that interact with external memory, however, every instruction has to pass through this unit in one clock cycle making the data path complex when ensuring proper data flow for every type of instruction.

Regarding the Instruction Set Architecture compatibility of a Single Cycle Architecture, it is typically supported by simpler ISA's such as the RISC-V or MIPS. This is due to the instructions being simpler and having more uniform execution times. They also have fixed-length instructions and less complexities when compared to advanced architectures such as the x86 architecture.

Due to the simplicity of the Single Cycle implementation, there are significant trade-offs between clock cycle time and the throughput of the overall design. If the clock frequency is increased to compensate for the lower throughput, it could result in timing violations overall when performing a place and router, for example, of this design. Hence, why it's important to consider other options such as multi-cycle and pipelined implementations.

### **Multi Cycle**

Multi-cycle Architecture is a processor design that takes multiple clock cycles to complete an instruction, as the name suggests. Such design distributes the workload over several cycles instead of completing it in single cycle like the Single Cycle Architecture. In doing so, it allows for more efficient use of processor resources, since each functional unit operates independently, allowing for maximum resource utilization and minimization of idle time or stall time. Like the single-cycle architecture, the instruction sequence is the same, fetch, decode, execute, memory access, and write-back. The key difference is presented in its flexible cycle time which no longer depends on the longest instruction in the instruction set architecture. Instead, each stage will operate under the appropriate number of clock cycles based on its demands. This allows for simple instructions like register-to-register arithmetic to execute faster than more time-consuming memory access instructions. This will enhance overall design efficiency.

Like single-cycle architecture, the multi-cycle architecture utilizes a finite state machine in the control unit to manage all stages across the varying cycles. <sup>23</sup> Due to the assignment of predetermined control signals for each sub-operation, the control unit allows each instruction to complete using the necessary cycles; no more than necessary. This modularization ensures that the available resources like the ALU, register file, and memory units are not bound by the longest instruction like in single cycle designs. An example of this would be when an instruction fetches in one cycle, decodes in another and performs ALU operations only, when necessary, since the entire design operates in sequence. This allows for the reuse of resources across stages, improving overall hardware efficiency. The

design especially optimizes clock cycle usage since each component completes tasks and returns to an idle state in the following use making it power efficient.

A great benefit of this architecture is the compatibility with RISC-V ISAs, which are reduced, and uniform instruction sets. Typical RISC ISAs enable efficient cycle management and facilitate easier optimizations due to the predictability of the fixed length instructions. Like the discussion on single cycle. Another advantage of this architecture is its ability to have adaptable clock cycle length to the varying instruction types. With shorter instructions like basic arithmetic operations, the design completes faster processing and allows for the relieving of resources faster. Contrary to longer instructions such as memory access, the precedence is slower, but it does not affect the execution of shorter and quicker instructions such as ALU operations. Even though there are presentable advantages, this architecture has limited throughput when compared to a more advanced design such as pipelined architecture where multiple instructions are processed simultaneously. Additionally, if there is a significant concentration of memory access instructions, worst case scenario, the efficiency obtained when compared to a single cycle design may be almost negligible since these instructions would still consume multiple cycle and reduce performance.

### **Pipelined**

The main idea of pipelining is keeping all the parts of the processor busy but executing different instructions simultaneously on each functional unit. It is a form of instruction-level parallelism which increases the processor throughput compared to a multi-cycle processor because, ideally, allows the processor to complete one instruction every cycle even if, individually, each instruction takes longer than a cycle. In perfect conditions, the speedup provided by using a pipeline design is approximately equal to the number of stages of the pipeline because clock cycles are permitted to be faster compared to the single-cycle design.

To design a pipeline processor, it is necessary to identify how to break the execution of an instruction in multiple stages depending on what resource they use. For explanation purposes, a simple RISC pipeline design uses five stages:

- **Instruction fetch (IF):** this unit handles reading the next instruction from memory or cache, it also keeps track of the current position of the program in the program counter register (PC). This stage is characterized for accessing the instruction memory.
- **Instruction decode (ID):** this unit identifies the type of the instruction and sets the rest of control signals required to select the correct register from the register file, as well as identifying hazards or potential for forwarding. In certain cases, this unit is also responsible for branching-related operations. This stage is characterized for accessing the register file.
- **Execute (EX):** this unit encompasses the ALU and other computational components. This stage is characterized by using the ALU.
- **Memory access (MEM):** this unit is responsible for addressing and accessing memory. If the instruction is not a load or store, the instruction simply passes to the

next stage of the unimpeded pipeline. This stage is characterized by using data memory.

- **Writeback (WB):** this unit writes the result to the register file. This stage is characterized by using the register file – note that this can lead to a data hazard with the ID stage.

In a pipeline, data is passed from stage to stage using a set of pipeline registers that retain the data and control signals between each stage. Ideally, each clock cycle advances the pipeline and passes data and signals from one stage to the next. Additionally, each resource (register file, memory port, ALU) can only be used by one pipeline stage at a time, otherwise a structural hazard would occur.

Because of the properties of RISC architectures, the design of a pipeline is greatly simplified. All instructions are the same length and only a few instructions format exist with the location for the operand fields – these properties greatly simplify the instruction fetch and decode stages. Additionally, because of the load/store architecture used by RISC-V, only two kinds of memory access instructions are possible and the operand for arithmetic instructions are not permitted to reference memory – so all data memory accesses are centralized in a single location of the pipeline.

### Comparison Of Architectures

*Table 3-2: Single-Cycle vs Multi-Cycle vs Pipelined Architectures*

Aspect	Single-Cycle	Multi-Cycle	Pipelined
<b>Clock Cycle Time</b>	Longest Instruction	Flexible	Shorter cycles
<b>Efficiency</b>	Low	Moderate	High
<b>Control Complexity</b>	Simple	Moderate	High
<b>Hardware Utilization</b>	Inefficient	Efficient	Very Efficient
<b>Hazard Management</b>	No hazard	No hazard	Complex hazards
<b>Power Efficiency</b>	Low	Moderate	High
<b>Typical Usage</b>	Simple designs	Moderate designs	Complex designs

In single-cycle architecture, the clock cycle is defined by the longest instruction, which leads to inefficiency for the simpler instructions that take less time. Multi-cycle architecture allows for more flexibility in clock length, as each instruction only uses the clock cycles needed to successfully execute resulting in a more efficient clock utilization. The pipelined architecture further optimizes cycle time by overlapping instruction stages, enabling each cycle to handle different stages of multiple instructions in parallel, which leads to much shorter individual cycle times.

Single-cycle processors have low efficiency as every instruction uses the same clock cycle time, which means that simple operations take longer than necessary with a lot of idle time. Multi-cycle architecture improves from the single cycle by allowing each instruction to only occupy the cycles it needs, which reduces idle times. The pipeline architecture maximizes efficiency by running multiple instructions in parallel, which can ideally complete one instruction per cycle.

Single-cycle designs are straightforward, as all the instructions are completed within one cycle, requiring only a basic control unit. Multi-cycle designs add a bit more complexity as they use finite state machines to control different stages that each instruction is used and the cycle time. The pipeline design has a higher level of complexity as each stage in the pipeline needs its own control mechanism to handle data hazards, and stalls.

In a single cycle processor, there is a lot of idle time for components such as the ALU and memory units, as they do nothing while the other stages are completing the instruction. Multi-cycle processors can reuse components across different cycles, making them more efficient. The pipelined processors achieve peak utilization as each stage in the pipeline can be handling a different instruction simultaneously, which allows for maximum usage of the resources per cycle.

In single-cycled designs, hazards are not an issue as each instruction completes before the next one begins. Multi-cycle designs have few hazards to deal with and require basic control strategies to avoid conflicts. Pipelined designs experience a high number of hazards as instructions run in parallel. This means techniques such as data-forwarding, branch prediction, and pipeline stalls are required to ensure the system operates reliably.

Single-cycle architectures have low power efficiency as clock cycles are longer than they need to be for most instructions to accommodate for the instructions that need more time. Multi-cycle architectures improve their power efficiency compared to single cycle by fluctuating the clock cycle time to the shortest time possible. Pipelined architecture has the highest power efficiency as each clock cycle is uniform, and a instruction is ideally completed every clock cycle.

Single-cycle processors are ideal for simple designs, where lower performance is not an issue, and throughput and power efficiency are not important. Multi-cycle processors offer a better balance between efficiency and complexity, making them more suitable for moderately complex systems where performance is important but not critical. Pipelined processors are ideal for complex, high-performance systems where maximum performance and high throughput are essential, where a high rate of instructions must be processed fast

### **Potential Pipeline Optimizations**

Cache and Memory Prefetching can significantly reduce the latency associated with slow memory access, particularly in memory-bound vector operations. Prefetching works by proactively loading data from memory into cache, preparing it for upcoming instructions. This strategy is especially beneficial for vector operations, where performance often suffers due to high data requirements that can exceed cache capacity. When data cannot fit into cache, the operation becomes limited by main memory bandwidth, increasing the need for repeated memory read/writes.

In this case study on AXPY ( $\text{Alpha} * \text{X} + \text{Y}$ ), a standard operation in scientific computing, prefetching techniques have been shown to reduce cycles per instruction (CPI) by nearly half, as evidenced by the following benchmark:

```
1. void DAXPY(int n, double alpha, double *x, double *y) {
```

```

2.     for(int i = 0; i < n; i++) {
3.         y[i] = alpha * x[i] + y[i];
4.     }
5. }

```

Figure 3-5: DAXPY Function

The critical section of this DAXPY function can be translated into the following MIPS instructions, illustrating the execution process at a lower level:

```

1. # Load x[i] into $f2 (64-bit floating-point)
2. LDC1 $f2, 0($s1) # Load x[i] into $f2 (offset is i * 8 for 64-bit)
3. MUL.D $f4, $f0, $f2 # $f4 = alpha * x[i]
4. LDC1 $f6, 0($s2) # Load y[i] into $f6
5. ADD.D $f8, $f4, $f6 # $f8 = alpha * x[i] + y[i]
6. SDC1 $f8, 0($s2) # Store result back into y[i] (64-bit double)

```

Figure 3-6: DAXPY MIPS Translated

Without cache prefetching or instruction reordering, this sequence would follow a simple linear execution path: loading  $x[i]$ , performing the multiplication, then loading  $y[i]$ . In this structure, each load introduces a delay due to memory access times. Prefetching improves efficiency by moving these memory operations earlier in the pipeline, allowing parallel loads and reducing idle cycles where the CPU would otherwise wait for memory.

By leveraging cache prefetching, overall execution time is significantly reduced, as loads occur ahead of arithmetic operations, optimizing CPU cycles and minimizing memory stalls.

### 3.2.3 Team Design Options

After the first two deliverables, initial research phase and initial block design development requests, the team put together a list of options for our microarchitecture of the CPU and memory. These options consist of in-house and open-source designs. This list of options was presented to Northrop Grumman Engineers to comply with our first deliverable deadline. Below is a table demonstrating our options and an explanation of such options. The deadline for this deliverable was 9/27/2024.

Table 3-3: Design Options Overview

Options	Tradeoffs
<b>Option A</b> - Team designed 5 Stage Pipeline model without hardware implemented hazard and forwarding unit.	Would require software implementation of a hazard detection unit through the development of a compiler for RISC code (In the works). The majority of RTL will be developed by team but would be less intensive with the elimination of a forwarding and hazard unit. We would learn more about the first two stages of the entire project but could make phase 2 more difficult to get too.
<b>Option B</b> - Team designed Single Cycle implementation	Simpler design which would allow us to write RTL at a faster rate. The concern is that the throughput is too low for us to

	reach the goal of reaching 100 MIPS due to the decreased throughput. There is more existing documentation for this implementation.
<b>Option C</b> - 5 stage Pipeline with hardware implemented Hardware Detection Unit and Forwarding Unit.	Good amount of documentation. Has increased throughput but is way more complex to implement for the level of experience the team possess.
<b>Option D</b> - Reverse engineering Ripes.me models and code. It would require turning their open-source C++ code into RTL code.	This uses preexisting resources and would require translation of the design into RTL code. The system interactions are already established but would require all RTL code to be from scratch. This could be time favorable assuming translation is efficient and feasible.

### 3.2.4 Option A – Team Designs 5 Stage

Top-level hierarchy design is mostly complete but requires review and some minor polishing. This design would be custom for our chip's requirements and thus the majority of RTL will be original. Although the design would not have hardware level hazard detection, a simple program can be used to edit code to mitigate hazards before compiling. We have developed an RV32I assembler to detect and address hazards in the source code. Examples of some issues which need to be addressed in the design are how the processor will stall and flush and that a custom pipelined design inherently has more risk at all stages.



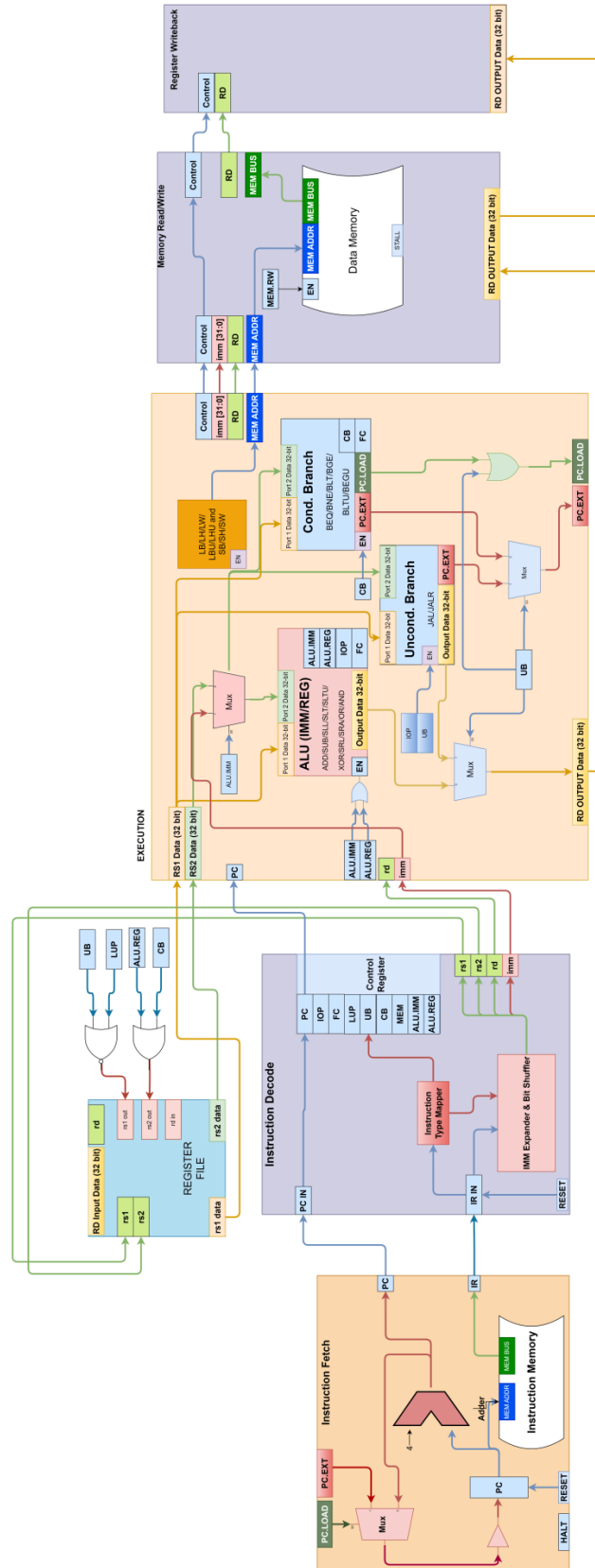


Figure 3-7: 5-stage Pipeline

### 3.2.5 Option B – Team designed Single Stage

Top-level hierarchy design is partially complete but requires review and some work. This design would also be custom for our chip's requirements and thus the majority of RTL will be original. We would not need to worry about hazards when utilizing this design and it is still possible to reach the required MIPS due to the simple instruction set, this would be significantly less powerful than the five stage. The main issue with this design could be that it doesn't hit performance metrics without a high clock rate.

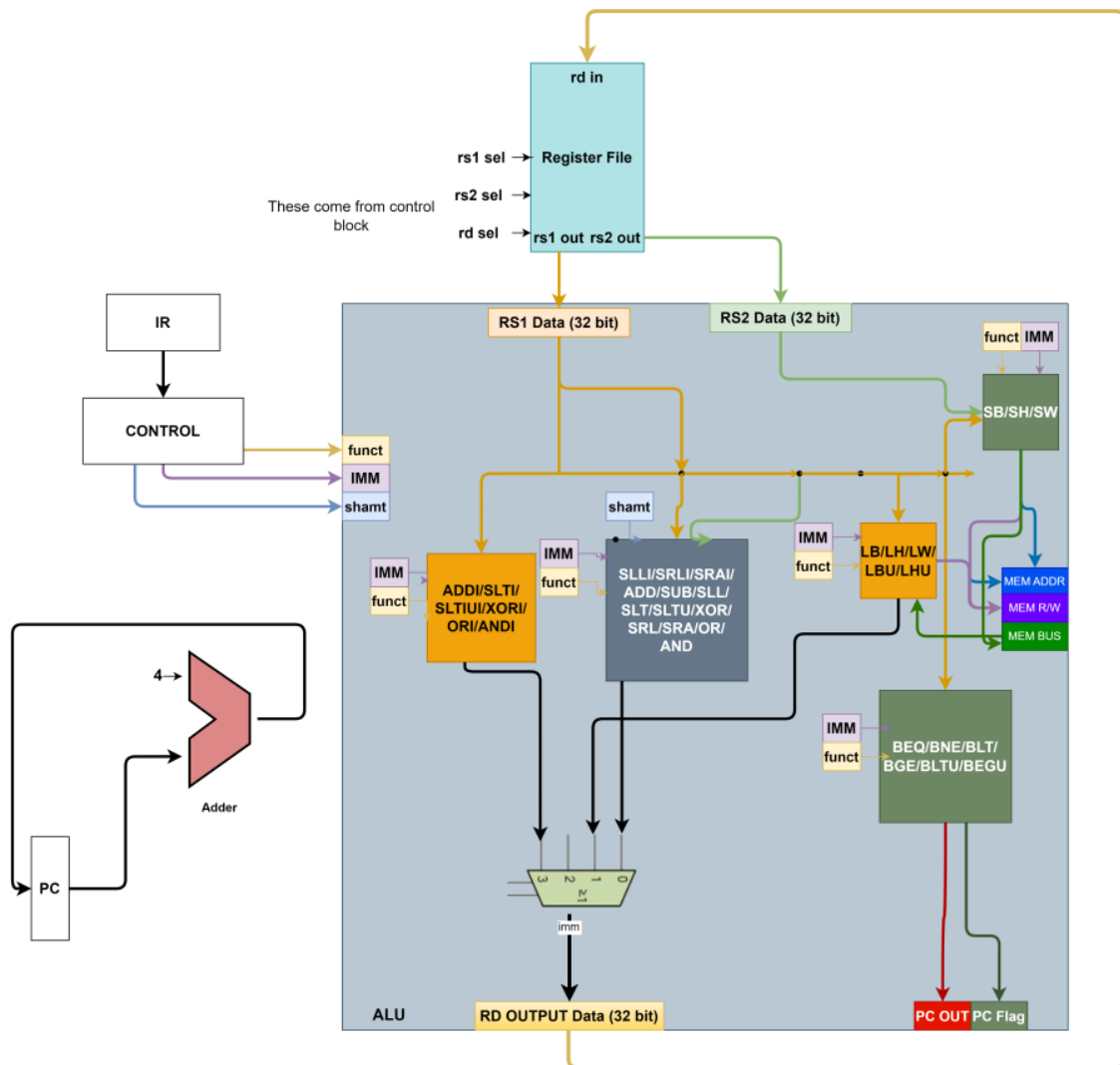


Figure 3-8: Single Stage Pipeline

### 3.2.6 Option C – 5 stage w/ Hazard and Forwarding Units

This implementation would require the design to have a hardware implementation of the hazard and forwarding units. Such requirements would equal more complexity, and more complexity may not allow us to meet our deadlines. The entire team would prefer to consider all other options over this one. Even with open-source resources this implementation would still be complex to implement. Our main concern is that this is a

part of our expected requirements, and we would like to know if other implementations are possible over this one.

### 3.2.7 Option D – Ripes.me

Option D consists of a great dependency on the Ripes graphical interface simulator. Such resource provides multiple iterations of the RISC architecture implementation such as the single cycle, 5 stage and 5 stage with hazard detection and forwarding. Their github is accessible and provides the source code in C++. Our intention when using this resource is to translate the C++ code into RTL. This would give us the opportunity to write our own RTL with an established structure of how the modules are intended to behave. Below are some of the designs they provide:

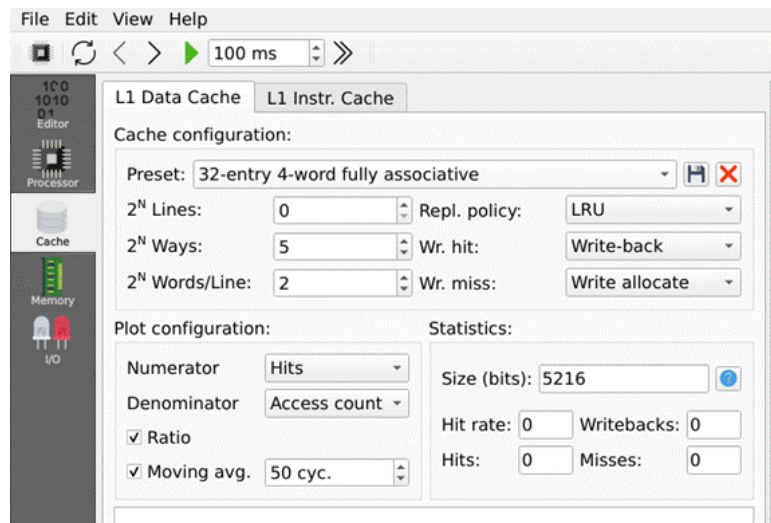


Figure 3-9: L1 Data Cache configuration

The figure above describes the L1 data cache configuration that lists out the main characteristics of the L1 data cache. The implementation of the cache above was used on the design code below, which is also provided by the website. The implementation of the cache above is the “32-entry 4-word fully associative”. An alternative to the fully associative option is the directly mapped option. The direct mapped is much more straightforward to implement while Northrop Grumman Subject Matter Experts have suggested that the fully associative option is what the industry mostly utilizes. We can also observe from the figure above that changing the number of lines, ways and words/line changes the configuration exponentially and it should be modified to best suit the user’s application.

```

Source code      Input type: ☒ Assembly ☐ C
1  # This example demonstrates an implementation of the multiplication of two
2  # complex numbers z = 1 + 3i, w = 5 + 4i.
3
4  .data
5  aa: .word 1 # Real part of z
6  bb: .word 3 # Imag. part of z
7  cc: .word 5 # Real part of w
8  dd: .word 4 # Imag part of w
9  str: .string " + i* "
10
11 .text
12 main:
13     lw a0, aa
14     lw a1, bb
15     lw a2, cc
16     lw a3, dd
17
18     # Do complex multiplication of numbers a0-a3
19     jal complexMul
20     mv t0, a1 # Move imaginary value to t0
21     mv a0, a0 # Move real value to a1
22
23     # Print real value (in a0) by setting ecall argument to 1
24     li a7, 1
25     ecall
26

```

Figure 3-10: Ripes.me Assembly code example with memory map

The Figure above shows the loaded example from a complex multiplication of numbers. This exemplifies how as the assembly code goes through the lines of code, it shows how the available registers are used to hold the values needed to perform the calculation in the assembly code. We have seen by experimenting with the code how a configuration without hazard detection could potentially use an undesired number from the register because there were not enough cycles for the register value to be updated and then used to perform calculations in the assembly code.

### 3.2.8 Memory Architecture Selections

For memory architecture, we are aiming for the simplest design possible so that it meets our needs and has adequate performance. We have determined two options but have identified our preference with our first option; option 1. Below are simple block designs for how we plan on our split cache designs to interact with the main CPU microarchitecture and main memory, in this case RAM. We have also depicted the reason why we do not prefer option 2.

#### Option 1 – Harvard Architecture

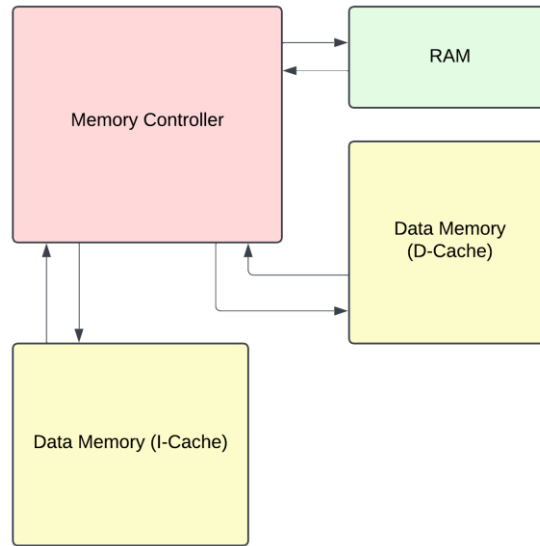


Figure 3-11: Harvard Memory Architecture

The instruction memory or I-Cache will be a part of the 1st stage of the RISC pipeline that we intend to implement. This can be said for data memory or D-cache as well. This split cache technique with a connection to a memory controller for the RAM will be easier to deal with as there will be no back and forth for the I-cache. That is if we load all instructions at the start of the PC and only access RAM when we intend to write to RAM. This will avoid a conflict between the instruction memory and data memory access of the controller.

### **Option 2 – Von Neumann Architecture**

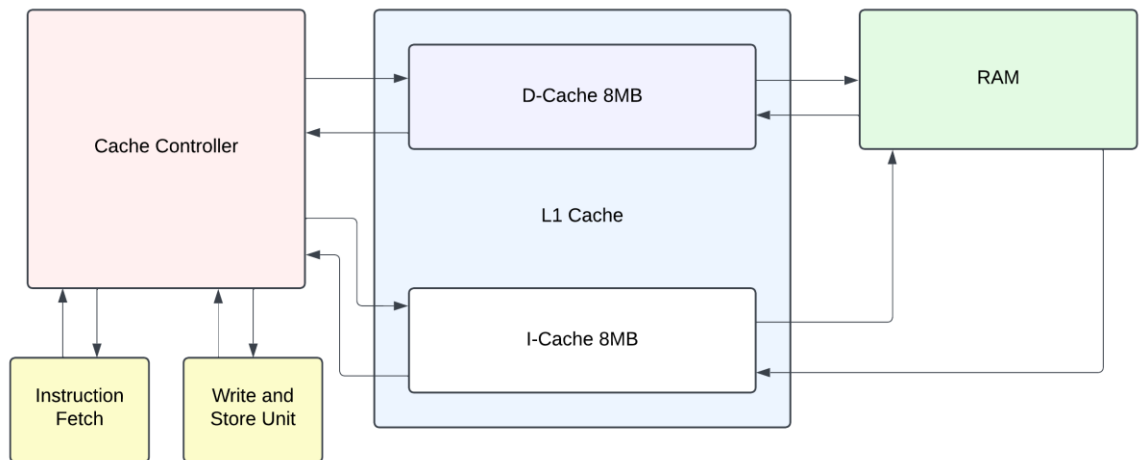


Figure 3-12: Von Neumann Architecture

For this design we use a cache controller and have the cache directly connected to RAM when there is a cache miss. We could also have included a memory controller but that would cause more latency for a simple memory design. It should be noted that the cache has been split into I cache and D cache. The I cache for instructions and the D cache for data.

The potential issues we have identified with this design are that having multiple modules to access the caches would increase our latency. In addition to that having both the instruction fetch and the write and store unit access the cache controller at the same time would cause more latency and then for a consistent number of stalls if not dealt with. Our solution to this is to perform cache line buffering which in turn means that every time we access the D-cache we also access the I cache and pull the next instruction. This would, however, increase the complexity of the design.

### 3.3 Pipeline Hazards

#### 3.3.1 Types of Hazards

In a pipelined datapath architecture, whenever there are multiple instructions in the pipeline at the same time, there exists the possibility for data dependencies and hazards. Hazards present a complicated challenge for the design of the architecture as the hardware must ensure the correct execution and evaluation for all possible combinations that might enter the pipeline sequentially. Generally, there are three different types of hazards that can be recognized by the type of instructions that triggers it:

- Data hazard: this type of hazard occurs when an instruction that writes to a register is ahead of an instruction that reads from the same register in the pipeline; in other words, this is equivalent to a read after write condition, also known as a data dependency. The hardware must detect and correct this condition prior to the evaluation of the lagging instruction.
- Control hazards: this type of hazard occurs when a conditional branch instruction enters the pipeline, and therefore the correct order of subsequent instructions is not known until the conditional branch instruction is fully evaluated. Any instruction that has the potential to alter the program counter can be a source of a control hazard; in RISC-V, only the functions related to branching have this effect.
- Structure hazard: this type of hazards occurs when two instructions need to access the main memory in the same cycle; this requires synchronization of all functional units of the processor that are capable of addressing memory to ensure controlled access and ordering of these memory operations. A dual-port memory, allowing writing and reading on the same cycle can also be used to address this hazard. More generally, this type of hazard is the result of the hardware being incapable of supporting different combinations of instructions simultaneously within the pipeline.

Generally speaking, a processor where hazards can occur will have a higher CPI compared to an ideal processor without hazards. However, in many cases the hardware changes necessary to mitigate a type of hazard may increase the cost or complexity of the processor to an unacceptable amount. Therefore, a pipeline processor design must strike a balance between complexity and the acceptable types of hazards that can occur.

#### 3.3.2 In-depth Analysis

From a processor design perspective, pipelining appears to change the relative timing of instructions by simultaneously executing them. A data hazard occurs when the pipeline

alters the order of reading and writing to a register operand from the order they appear to sequentially execute on a single-cycle processor.

In order to resolve data dependency hazards for pipelined architectures, it is necessary to identify the dependent instruction at the earliest possible time in the pipeline. That is, the architecture must implement a hazard detection unit that operates during the *instruction decode* stage that determines whether a stall must occur in the pipeline or if forwarding could be performed across what functional units.

Data forwarding can only possibly work if the functional unit that produces the necessary data operands completes before the functional unit that consumes said operands commences execution. When an instruction incurs a delay that cannot be mitigated by forwarding, the hazard control unit must ensure a pipeline interlock occurs to maintain the correct order of execution, which most usually results in a pipeline stall.

One scenario in which forwarding cannot be performed immediately after identifying the hazard is when an instruction that reads from a register is immediately following a load instruction that writes to the same register. This type of hazard is also known as a load-use hazard and can only be resolved through a pipeline stall because the result of the load instruction could not possibly be available in time to be forwarded. Depending on the design of a multi-cycle pipeline, other types of hazards might necessitate the use of stalls. During a pipeline stall, all instructions issued before the stalled instruction continue their execution to clear the hazard; while those issued after are stalled.

Regardless of the reason that triggers a pipeline stall, these need to be handled carefully to ensure the pipeline can resume at a later point. Most functional units can be made inert by the hazard detection unit through inserting a bubble – equivalent to a no-op instruction – into the pipeline to delay the preceding instructions without having any side effects, that is not register or memory is written. This bubble will delay execution of the proceeding instructions until the hazard clears or forwarding becomes possible. The instruction fetch unit, however, must retain the current instruction either on its internal instruction register or preventing the program counter from incrementing for the duration of the stall.

### **Structural Hazards in Depth**

The instructions in a pipeline are issued to different functional units within the processor and these must accommodate all possible combinations of instructions in the pipeline. A structural hazard occurs whenever a combination of instructions cannot be simultaneously executed because the processor lacks enough functional units, or they conflict in accessing a resource. For example, if the register file has insufficient ports, it may be impossible for the execution unit and writeback unit to simultaneously execute, therefore creating a structural hazard. This necessitates the processor to stall the pipeline until the resource or functional unit becomes available. When these hazards occur often, the performance of the processor heavily differs from the ideal performance.

### **Control Hazards in Depth**

Control hazards are the consequence of making a branching decision from the results of an instruction that has yet to be fully executed. In the ideal case, an instruction must be fetched

from memory at each cycle to keep the pipeline fed, however in most implementations the result of a branch instruction is not known until the completion of the execution stage several steps down the pipeline. This ambiguity and delay in what instruction to fetch next is called control or branch hazard.

In a RISC architecture, control hazards are especially troublesome because they incur the cost of conditionally changing the program counter and therefore the result of the following instruction fetch. In any case, the most important aspect of dealing with control hazards is that the processor must not change its state until the outcome of the branch is fully known, which might necessitate reverting the state of the processor.

If performance is not a concern, the simplest approach would be to stall the pipeline until the branching decision is determined at which point the pipeline can resume by fetching the next instruction as dictated by the branch. A similarly simple approach would be to predict all branches to always be not taken, which on average halves the cost of control hazards compared to the previous approach. In the case the prediction is wrong, the current instructions in the pipeline should be discarded by directing the control signals to replace all instructions in the pipeline with bubbles and therefore bypass any side effect of the now branched-out instructions. This operation is also known as flushing the pipeline.

As an additional optimization to reduce the delay of branches, some architectures opt to move the branch execution logic earlier in the pipeline – usually directly following the instruction decode stage. The immediate value and the current program counter registers are immediately available by the time the instruction decode stage finishes, and the circuitry required for calculating the target address for the branch is simple enough to be integrated by this stage when needed.

Evaluating the branch decision might necessitate expanding the forwarding and hazard detection unit to bypass operands from the later stages of the pipeline to a comparator integrated in the instruction decode unit. This offloads the comparison for branches from the ALU but requires appropriate forwarding logic from any stage in the pipeline. Additionally, a stall of two or more cycles might be required depending on the instructions preceding the branch to allow such forwarding.

As the penalty and cycles lost from a pipeline flush depend on how long the pipeline is, more advanced designs can make use of techniques such as branch prediction logic to minimize the probability that a pipeline flush occurs. Designs with aggressive pipelines must incorporate an equally aggressive branch prediction logic with additional hardware circuitry.

### 3.3.3 Software and Hardware Hazard Control

Handling pipeline hazards in CPU microarchitecture is essential for ensuring correct execution. Hazards can be addressed either in software, with support from the compiler toolchain, or in hardware logic. Choosing between these two approaches is a critical design decision, especially given the project's specifications and time constraints. The hardware-based approach introduces increased design complexity, particularly in terms of RTL (Register Transfer Level) design. This complexity makes the verification process more challenging, which is a significant concern. Verification is already the most difficult aspect



of this project due to the team's lack of experience and the time required. In terms of performance, the iDesign RISC core must achieve 100 MIPS (Million Instructions per Second) while staying within a defined power budget. A software-based approach, although simpler, reduces the IPC (Instructions Per Clock), leading to a lower MIPS rate. This may not be problematic if the power consumption is low enough to allow for an increased clock frequency, but raising the clock speed could also introduce issues, such as signal IR drop and timing violations. The software approach would result in simpler hardware, easing the verification and RTL design burdens, but it would also reduce the efficiency of the 5-stage pipeline. This method would require the development of custom software tools, such as a RISC assembler tailored to the iDesign microarchitecture, to manage hazards. However, this would also create incompatibility with existing RISC libraries and code, as modern compilers do not optimize for specific microarchitectures.

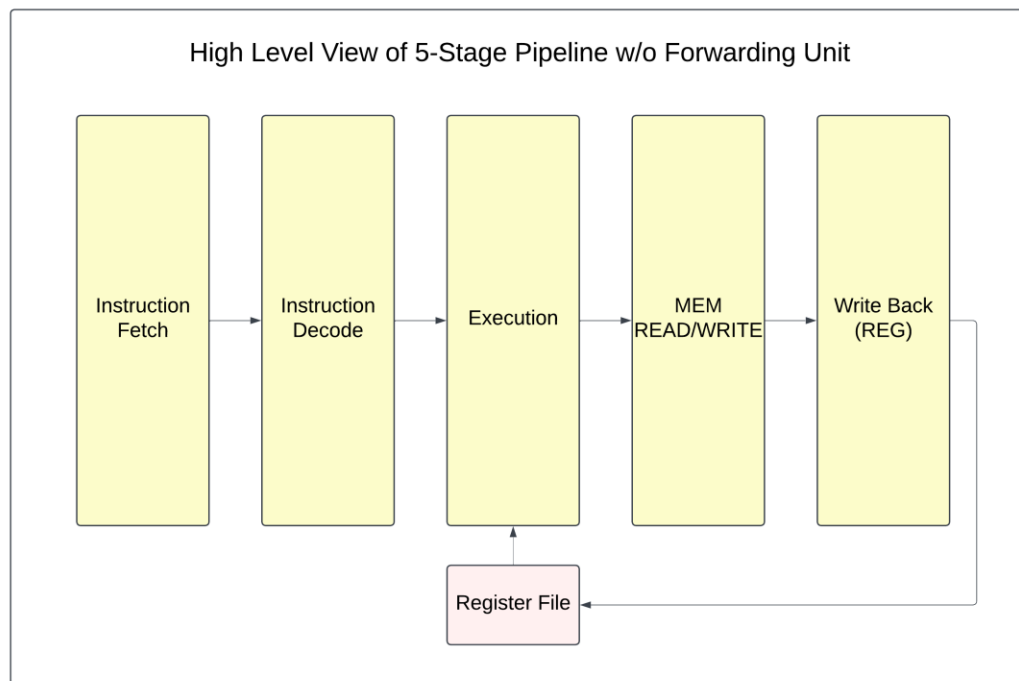
### **Example of Data Hazard and Solution**

To illustrate the hazard issue, consider a simple program that computes the equation  $c = 4 * (a + b)$ . This operation can be split into three instructions, using repeated addition because the R32I ISA does not support multiplication:

*Table 3-4: Example Instructions for Data Hazards*

Instruction #	RISC Assembly	High Level C Abstraction
1	add x7, x6, x2	$x7 = x6 + x2;$
2	add x7, x7, x7	$x7 = x7 + x7;$
3	add x7, x7, x7	$x7 = x7 + x7;$

Each of those instructions will enter the 5-stage pipeline and be executed concurrently as shown in the accompanying pipeline diagram:



*Figure 3-13: 5-Stage Pipeline without Forwarding Unit*

The following table shows the process of execution:

Table 3-5: Instruction Running in Pipeline without Forwarding Unit

IF	ID	EX	MEM	WB
add x7, x6, x2	-	-	-	-
add x7, x7, x7	add x7, x6, x2	-	-	-
add x7, x7, x7	add x7, x7, x7	add x7, x6, x2	-	-
-	add x7, x7, x7	add x7, x7, x7	add x7, x6, x2	-
-	-	add x7, x7, x7	add x7, x7, x7	add x7, x6, x2
-	-	-	add x7, x7, x7	add x7, x7, x7
-	-	-	-	add x7, x7, x7

Red text means wrong value was used in operation

These instructions enter the 5-stage pipeline and execute concurrently. However, as shown in the execution table, both instructions following the first one read the old value of x7. This occurs because the register file only updates at the Write Back (WB) stage at the end of the pipeline, leading to incorrect results. The expected value for x7 is 76, but the actual result is 0.

Table 3-6: Expected vs Actual

Clock Cycle	Instruction at EX	Expected x7	Actual x7
1	add x7, x6, x2	$11 + 8 = 19$	$11 + 8 = 19$
2	add x7, x7, x7	$19 + 19 = 38$	$0 + 0 = 0$
3	add x7, x7, x7	$38 + 38 = 76$	$0 + 0 = 0$

The final result for register x7 would be 0 instead of the correct value of 76. As mentioned above, the simplest solution is to stall the pipeline to clear read-after-write hazards. Both the software and hardware implementation would insert NOP (no operation) instructions into the pipeline; however, the software will have to insert those NOP instructions into code which will increase code size and reduce cache performance since less code can fit into cache memory.

The NOP does exist in the R32I ISA but it can be emulated using the add x0, x0, x0 instruction. The x0 register is hard wired to always output 0, therefore, this instruction will effectively have no effect on the state of the register file but it will occupy a spot in the pipeline. The amount of nop instructions depends on the microarchitecture, in the case of the RAPID microarchitecture, only two nop instructions are needed to clear hazards. The program will have to be adjusted into the following form when using pipeline stalls.

Table 3-7: Example with NOPs to fix incorrect results

Instruction #	RISC Assembly	High Level C Abstraction
1	add x7, x6, x2	$x7 = x6 + x2;$
2	nop (add x0, x0, x0)	No equivalent.
3	nop (add x0, x0, x0)	No equivalent.
4	add x7, x7, x7	$x7 = x7 + x7;$
5	nop (add x0, x0, x0)	No equivalent.
6	nop (add x0, x0, x0)	No equivalent.
7	add x7, x7, x7	$x7 = x7 + x7;$

The hardware would generate these nop instructions at the hardware level after hazard detection and would have no effect on memory size, but the software would be required to insert these instructions directly into the code. The software code had to increase from 12 bytes to 28 bytes, which is a 233% increase in size.

The execution strategy would change into the following:

*Table 3-8: Instruction Running in Pipeline with NOPs*

IF	ID	EX	MEM	WB
add x7, x6, x2	-	-	-	-
nop	add x7, x6, x2	-	-	-
nop	nop	add x7, x6, x2	-	-
add x7, x7, x7	nop	nop	add x7, x6, x2	-
nop	add x7, x7, x7	nop	nop	add x7, x6, x2
nop	nop	add x7, x7, x7	nop	nop
add x7, x7, x7	nop	nop	add x7, x7, x7	nop
-	add x7, x7, x7	nop	nop	add x7, x7, x7
-	-	add x7, x7, x7	nop	nop
-	-	-	add x7, x7, x7	nop
-	-	-	-	add x7, x7, x7

The original code would have only required 7 clock cycles to complete execution whereas the code required to resolve the data hazard requires 13 clock cycles. The IPC decreased from 1.0 I/C to 0.53 I/C which is 46% decrease in IPC. It was assumed that the pipeline was full prior to the execution of this program and the instructions already in the pipeline had no hazards.

### 3.3.4 Custom Assembler to Mitigate Control Hazards

Another hazard is the control hazard, this is introduced in the form of branching. The following program demonstrates how this issue might arise.

Correct execution of this program should branch to the timer label because x3 is greater than x2, however, in the current implementation the PC will jump twice. The first time occurs when the bgt arrives at the execution stage and the second time when the ret instruction arrives at the execution stage which is incorrect behavior. The hardware solution is to reset the previous two stages (Instruction Fetch and Instruction Decode) based on the result of the conditional branch. The software hazard approach can still insert NOPs.

```

1. # Configure initial conditions of registers
2. addi x2, x0, 10
3. addi x3, x0, 20
4. # Branch Instruction
5. bgt x3, x2, timer
6. ret # exit function

```

This section explains a C++ program that implements a custom assembler designed to process R32I assembly code for the RAPID microarchitecture. The assembler addresses

pipeline hazards by injecting NOP (no-operation) instructions where necessary, ensuring smooth execution of the assembly code. It is responsible for four primary tasks:

1. **Detecting Pipeline Hazards:** The assembler identifies potential hazards that could disrupt the flow of execution.
2. **Injecting NOPs:** It inserts the correct number of NOP instructions to mitigate identified hazards.
3. **Converting Assembly Code to Binary:** The assembler translates the assembly instructions into machine-readable binary format.
4. **Instruction Reordering (Optional):** While the assembler is designed with the capability for instruction reordering to reduce stall counts, this feature is not implemented in the current version.

The optional optimization logic is illustrated in the assembler diagram. Although reordering instructions could enhance execution efficiency by avoiding NOPs, the complexity involved has led to its exclusion in this implementation.

The analysis of the assembler reveals its effectiveness in handling pipeline hazards through NOP injection. By systematically identifying hazards and applying the appropriate number of NOPs, the assembler ensures that the execution pipeline remains stable. However, the absence of the instruction reordering feature limits its potential for further optimization. Future implementations could consider integrating this feature to improve performance while weighing the increased complexity it introduces.

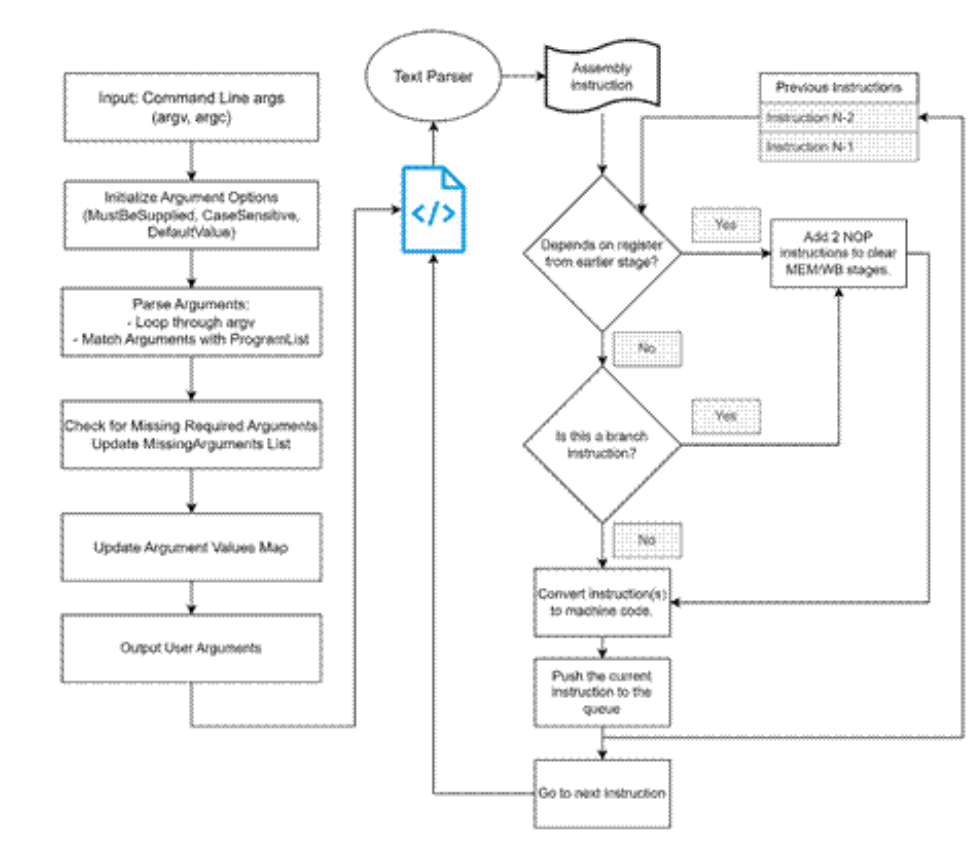


Figure 3-14: Diagram of Custom Assembler

## **Command-Line Argument Processing**

The program begins by defining a structure called ArgOpt to encapsulate options for command-line arguments. This structure includes properties such as whether an argument must be supplied, whether it is case-sensitive, and any default value associated with it. The main function uses the ProcessArguments function to handle command-line input, ensuring that required arguments are provided and default values are assigned where appropriate.

## **Instruction Parsing and Compilation**

The core of the program is centered around parsing assembly instructions. Several functions are defined to handle different aspects of this process:

- **Register Parsing:** The ParseRegister function converts register names (e.g., "x0", "x1") into their corresponding numerical identifiers. It includes error handling for invalid register names.
- **Immediate Value Parsing:** The ParseImmediateValue function processes tokens that may contain immediate values or memory references. It handles syntax checking and converts valid tokens into integers.
- **Instruction Encoding:** The program categorizes instructions into different types (R-type, I-type, B-type, and S-type). Functions like MakeRTypeInstruction, MakeITypeInstruction, and others construct the corresponding binary 43 representation of these instructions using bitwise operations. Each function extracts the necessary fields from the tokenized instruction, including opcodes, function codes, and register identifiers.

## **Instruction Type Identification**

The GetInstructionType function classifies instructions based on their names. This classification is crucial for determining how to encode each instruction correctly. The program maintains a mapping of instruction names to their corresponding types and opcodes, ensuring that the assembler can process a variety of assembly instructions.

## **File Handling**

The OpenFileStream function manages the opening of input files. If the file cannot be opened, it outputs an error message and terminates the program. The CompileFile function reads the assembly code line by line, tokenizing each line and checking for labels, comments, and other syntactical elements. This function also manages instruction dependencies, inserting no-operation (NOP) instructions as necessary to handle potential data hazards.

## **Main Execution Flow**

The main function orchestrates the entire assembly process. After processing command-line arguments, it opens the specified input file and invokes CompileFile to convert the assembly code into machine code. The resulting binary representation is stored in a vector, which can be further utilized or outputted as needed.

## Hardware Implementation

For any hardware implementation, there must be a logic block responsible for detecting hazardous instructions such as read after write (RAW) or write after write (WAW). The simplest solution is to simply stall the pipeline and flush the pipeline. This is the lowest hardware cost and simplest design; it would still decrease the IPC, but it is still better than a single stage design in terms of throughput. The more common approach is to replicate hardware to avoid concurrency issues. This method provides better IPC at the cost of more complexity.

Detection of hazards can be done by adding an additional bit to each register. This bit would be set if the register is being written to in the execution stage and reset at the write-back stage. This would be similar to software interlocking implementation but with reduced code size and much better code compatibility with existing software. This does not address the reduced IPC issue.

To address the IPC reduction, the forwarding unit is required to prevent hazardous situations. Instead of waiting until the result from the execution stage to arrive to write-back stage, we could instead directly send the result back to the next instruction. For this design, we append a bit to the register file to determine data validity.

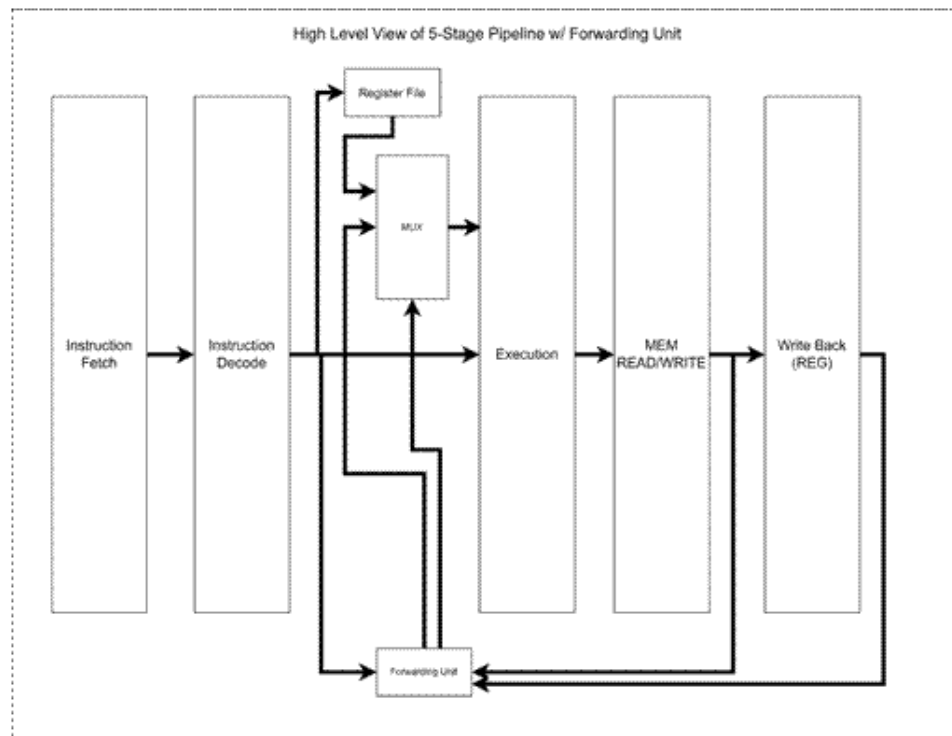


Figure 3-15: General 5-Stage Pipeline with Forwarding Unit

## 3.4 Dynamic Branch Prediction

As the complexity of a pipeline increases, so does the penalty for mis-predicting a branch instruction in terms of instructions and clock cycles wasted. Because of this, static

prediction schemes (such as always predict not taken) are insufficient to achieve good performance. Instead, more complex architectures implement dynamic branch prediction – through which the hardware is able to predict the likely outcome of a branch at runtime using contextual information gathered during the execution of the program.

There exists several dynamic branch predictions schemes and implementation, this paper will discuss some of the most well-established implementations and highlight their advantages and drawbacks.

### 3.4.1 Branch Prediction Buffer

The hardware can keep track of the recent outcome of branch instructions by the use of a branch prediction buffer, also known as branch history table. This is a small amount of memory dedicated to mapping the address of the branch instruction to a state that marks it as recently taken or not taken.

A branch prediction buffer is commonly implemented using a cache-like structure indexed by the lower portion of the address of the branch instruction accessed during the instruction fetch stage (that is, the program counter). If the branch is predicted as taken, the next instruction is fetched as soon as the branch instruction is decoded – otherwise, if the branch is predicted as non-taken, execution proceeds as usual. The pipeline is only flushed only when the outcome of the branch is known and discovered to be inconsistent with the prediction. The state of the branch prediction table is also updated at this time. In addition, the table can also keep track of the decoded destination address for prior executions of a branch by enhancing it with a branch target buffer – this would allow for an additional small optimization in which the processor does not need to wait to decode the branch target after it has seen the instruction at least once and allows the fetching to resume immediately even if the branch is predicted as taken.

The implementation for the prediction scheme can be commonly described by a state machine of two to four states. One or two bits are used to keep track of the state to weakly- or strongly-predict take or not-taken. In certain cases, a 2-bit scheme will make more accurate predictions for conditional loops in situations where a 1-bit predictor would otherwise mis-predict a branch twice in a row.

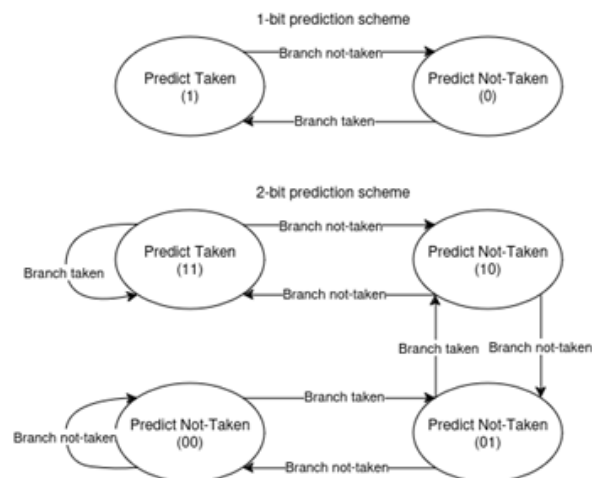


Figure 3-16: Branch Predictor State Machine

### 3.4.2 Correlating Branch Predictor

Several techniques exist to improve branch prediction accuracy beyond branch prediction buffers that make a decision based solely on the history of a single branch. In order to make more accurate predictions, correlating branch predictors work by also taking into account the recent behavior of other branches in the program. By analyzing the global behavior of the program rather than just a single branch, more information is available to decrease the likelihood of mispredictions.

A branch prediction buffer can be enhanced into a correlating predictor by using the history of recent branches as the index into the history table. More accurately, the history of branches can be recorded into a shift register – the value of this shift register is then used to select a branch prediction buffer from a set where each element is a regular branch prediction buffer corresponding to a branch.

### 3.4.3 Tournament Predictor

In general, researchers noted that using both local and global information to predict branches increases performance in situations where otherwise only using local information leads to too many mispredictions. Elaborating on the enhancements granted by combining this information, tournament predictors is a technique where multiple types of predictors are used simultaneously – for example, one using only local information and another using a combination of local and global information. Each predictor is tracked to determine which one has been the most accurate and the tournament predictor chooses selects the right one for each particular branch. Overall, the speedup provided by tournament predictors makes it possible for a processor to make accurate predictions with only a reasonable size requirement by favoring the best predictor based on the best information available to the program.

## 3.5 Cache and Optimization

Choosing an adequate memory hierarchy and cache implementation is crucial for a processor to achieve good performance by taking advantage of spatial and temporal locality of memory accesses. Several factors must be considered when designing a cache implementation, such as cost, access time, and power consumption.

Data is transferred between cache and main memory, or between different levels of cache, in units called blocks. The associativity of a cache defines the mapping between a memory block and the possible locations on the cache where such a block can be placed. The group of cache blocks where a possible memory block can be mapped is called a set.

The two extremes of cache associativity are known as direct-mapped cache, in which each block can only be mapped to only one location in the cache; and fully associative, via which a block can be placed anywhere. The associativity of a cache has a significant impact on the miss rate of the cache, particularly important on the rate of capacity and conflict misses.

### 3.5.1 Optimization

There are several techniques that can be utilized to reduce the average memory access time, which improve one of three aspects of a cache: reducing the miss rate, reducing the miss



penalty, or reducing the time it takes to hit the cache. However, improving one aspect of the cache often has the undesirable effect of worsening some other aspect. So, improving the performance of the cache consists of balancing the tradeoff between the different possible optimizations.

Augmenting the spatial locality of the cache is a simple way to reduce the miss rate. This can be achieved by increasing the block size, which reduces compulsory misses. However, larger block sizes increase conflict misses because it reduces the available number of blocks in the cache, and this results in an increase in the miss penalty as more data has to be transferred from main memory. Determining an adequate block size for the cache requires balancing the acceptable miss rate and miss penalty through optimizing the cache for the underlying memory technology and bandwidth. In general, low latency and low memory bandwidth benefits smaller block sizes; conversely, high latency and high memory bandwidth benefits larger block sizes.

Another technique utilized to reduce the miss rate consists of increasing the cache associativity. In general, researchers have observed that a direct-mapped cache has about the same miss rate as a two-way set associate cache of half its size. The tradeoff for increasing associativity is a likely increase in hit time. Additionally, it has been observed that an associativity greater than eight offers diminishing returns and is practically equivalent to a fully associative cache.

The last simple technique to reduce the miss rate is increasing the capacity of the cache at the expense of increased die size and power. More advanced techniques focus on optimizations beyond the miss rate; such as reducing the miss penalty and hit time. Particularly, because technological improvements in processor clock speeds have far exceeded the clock speed for memory chips, modern processors make use of multi-level caches that range in speed and size. A first-level cache prioritizes speed and minimizes access time as much as possible to keep up with the speed of the processor; while higher level caches trade speed for capacity. In general, lower-level caches prioritize fast hit times while the higher levels prioritize reducing the miss rate.

The following table summarizes how the techniques discussed affect the caches performance. A (+) symbol indicates the technique has a net positive effect on a particular metric, while a (-) symbol indicates it is detrimental to a metric.

*Table 3-9: Techniques that Affect Cache Performance*

<b>Technique</b>	<b>Hit time</b>	<b>Miss penalty</b>	<b>Miss rate</b>	<b>Complexity</b>
<b>Increase block size</b>		-	+	N/A
<b>Increase cache size</b>	-		+	Little complexity
<b>Increase associative</b>	-		+	Little complexity
<b>Multi-level cache</b>		+		Higher complexity

### 3.5.2 D-Cache Block Diagram

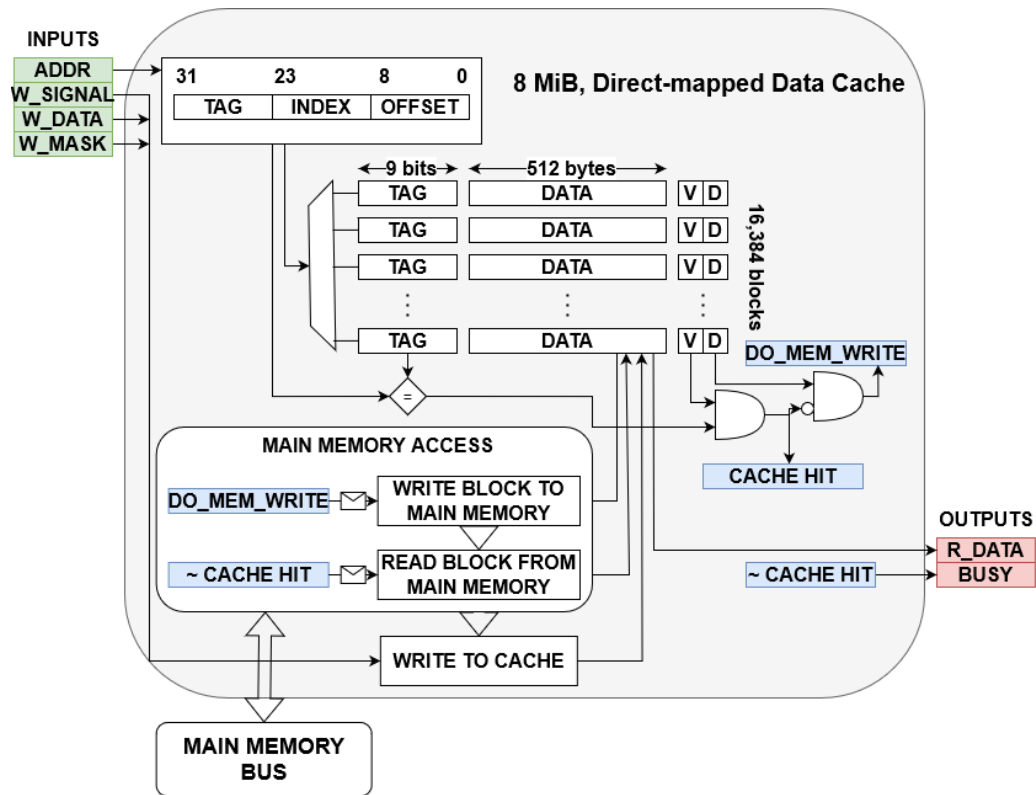


Figure 3-17: D-Cache Block Diagram

The D-Cache implementation would correspond to an 8MiB direct-mapped cache, with 16K blocks of 512 bytes each, but we're open to suggestions about the best block size for the cache. The cache could be implemented with a state machine with four states:

1. If a dirty block is about to be evicted, write it to main memory.
2. On a cache miss, load the corresponding block from main memory.
3. On a write operation, place incoming data from the processor in the cache.
4. On cache hit, give the data back to the processor and clear the busy signal.

Whenever the busy signal is asserted, the processor should pause on the next memory read/write operation to allow time for the current operation to complete.

The I-Cache implementation could follow a similar structure but removing all functionality related to writing to the cache and main memory.

We expect to further expand on the topics below the further we get into the RTL stage:

1. Cache line organization
2. Base address initialization protocol - requires further research
3. Ram organization (shouldn't be too complex like cache)
4. D cache segmentation (due to our requirements it should not be too complex)
5. Note that we are dealing with memory address plus the data within such address.
6. The primary organization is in memory addresses.

## 3.6 Instruction Set Architectures

Above the logic between the units in a processor and below the high-level programming languages used by programmers are instruction set architectures (ISA). The ISA acts as the boundary between software and hardware and in simple terms is the set of instructions that a processor can execute and how instructions are performed at the hardware level. In general, most ISAs are general-purpose register architectures which perform instructions on register and memory locations, but how these instructions are performed is what defines a certain architecture.

There are a few ways to classify ISAs but one of the main differences is the complexity of each instruction. An ISA can be implemented as a complex instruction set computer (CISC) or a reduced instruction set computer (RISC). In a CISC architecture there are many specialized instructions which are executed through the use of microcode. Microcode is not necessarily used for all instructions, but the instructions that require microcode are broken down from entire instructions into small RISC-like instructions. The x86 ISA used by Intel can be considered a CISC architecture even though it has changed to be more like RISC. In contrast, RISC based architectures aim to achieve performance through simplicity with each instruction designed to execute in a single clock cycle without the need for any microcode. While more and more complex instructions are being added to many RISC ISA with certain extensions, these are generally implemented through routines of simple instructions. To put this into perspective, the base instruction set for RISC-V, RV32I, only has 47 instructions while x86 has over 1,500 distinct instructions when considering its many extensions.

Another main way to distinctly classify ISAs is based on how they can access memory. CISC ISAs are often register-memory based which means that many instructions have the ability to access memory such as in x86. Meanwhile, ARM and MIPS, two RISC ISA, are considered load-store which can only access memory with individual load and store instructions rather than having the capability built into most instructions. While this description may make it look like load-store based ISA are weaker than register-memory ISA, this methodology allows for simpler design which can often offer better power efficiency and performance.

### 3.6.1 x86

Seeing as most general-purpose computers are x86 based on x86 it is a topic which needs to be addressed. As mentioned before the x86 ISA is an ISA developed by Intel and based on a CISC architecture. Most modern consumer processors are x86 based such as the Intel core and AMD Zen lineups. The x86 architecture was originally released in the late 1970s as an assembly language for the Intel 8086. Most RISC based ISA have fixed length instructions but x86 has variable length instructions which are between 1 to 15 bytes due to the complex nature of CISC. For 32-bit x86, there are 8 general purpose registers and then special registers such as the flag registers or the instruction pointer register. There are a few more features to addressing in x86 when compared to most RISC based ISA, but overall, the ability to address registers based on an offset as well as directly addressing a register allows for easier programming. A big benefit of the x86 design is the backwards

compatibility, but this could also be considered bloat due to how unwieldy the ISA has become over the past 50 years and would simply add unnecessary instructions for our project. For the purposes of our project, which is to design an efficient and simple ASIC, x86 processors are powerful but not the greatest option for an embedded approach due to the complexity and inefficiency of the architecture. The Intel atom series is Intel's low power, embedded solution which is often used in industrial applications and is a good example of a x86 based embedded CPU. The processors which Intel and AMD offer are really the only possible solutions for x86 however, which is why we will be avoiding an x86 based design even though it is such a popular ISA. This will be further discussed in the constraints section.

### 3.6.2 ARM

Arm could be considered second place in market share for general-purpose processors when compared to x86, but the vast majority of smart phones run Arm, about 99% [4]. This is because the mobile device field is primarily embedded systems where power efficiency matters more than raw performance, so a RISC ISA comes out on top. Of the many Arm based processors, the 32-bit Cortex-M series is often used for low power embedded systems and supports many industry standards such as the Advanced High-Performance Bus (AHB) standard. The Arm architecture is certainly a RISC based architecture, but with the many modern extensions made to the language Arm processors are often close to x86 processors for applications where performance is more valued than efficiency.

Instructions in Arm are generally fixed length 32-bit instructions but there are “thumb” variants of arm which contain 16-bit and 32-bit instructions. As mentioned before, Arm is a load store architecture and most versions of ARM require instructions and data to be memory aligned, so the processing of instructions is relatively simple. Regarding how registers are allocated, 32-bit Arm has 15 general use registers and 37 total registers. The simple instruction list and memory formatting of Arm allows for simpler pipelining as well which is the main reason why RISC based architectures are so efficient. One main issue with Arm though is that there are generally licensing fees, so to develop an Arm based core we would need to pay fees which would not fit within our budget.

### 3.6.3 MIPS

Microprocessor without Interlocked Pipelined Stages (MIPS) is another RISC load store based architecture which was introduced in the early 1980s as a research project at Stanford university. The purpose of MIPS is in the name, the ISA is focused on implementing a simple instruction set which could be easily pipelined, and the pipeline is where you can get the most efficiency out of a processor. MIPS uses 32 registers which are each 32 bits wide and implements 111 unique instructions in a 32 bit field. There are not many other differences between MIPS and RISC-V, but MIPS has a hefty license fee and because of that has been overshadowed by the open source RISC-V ISA for almost every application which would want to use MIPS.

### 3.6.4 RISC-V

The RISC-V architecture is the last ISA which will be discussed in this report and most well meets the requirements of our project. The exact history of RISC-V is not necessarily important, but how the architecture has evolved is important. Currently the RISC-V architecture has a few extensions to the RV32I base ISA which include more complex instructions such as multiplication and division and even support for floating point operations with the inclusion of a floating point ALU. Many embedded systems will not need to implement floating point arithmetic, and some will not even need multiplication or division to be integrated into the instruction set, so to keep the project simple and relevant we plan to stick to the RV32I base instruction set. To add onto this, the fact that the ISA is open source and so well documented is the main reason we chose to use RISC-V over other RISC based ISA.

The RV32I base instruction set consists of 47 instructions, but many of those instructions are not necessary for standard application such as the synchronization and environment instructions like FENCE and EBREAK. Consequently, the scope of this project has been limited to 37 instructions which 51 will be shown in a figure later in the report. Like base MIPS, base RISC-V has 32 general purpose registers which are 32 bits wide. This along with memory ordering makes RV32I relatively simple to pipeline especially when compared to CISC based ISA. All instructions need to be aligned on a four-byte boundary and are divided into 6 instruction formats, Register (R), Immediate (I), Store (S), Upper Immediate (U), Branch (B), and Jump (J) type. There are also immediate types of all these instructions except for the R-type instruction. The limited amount of instruction types makes the decoding stage quite simple as the ALU can simply route all instructions to every internal unit and the control unit can determine which result is to be passed onto later stages of the pipeline. The individual control logic implemented is different from processor to processor, but all processors which implement the RISC-V will need to use this information to be considered RISC-V compatible.

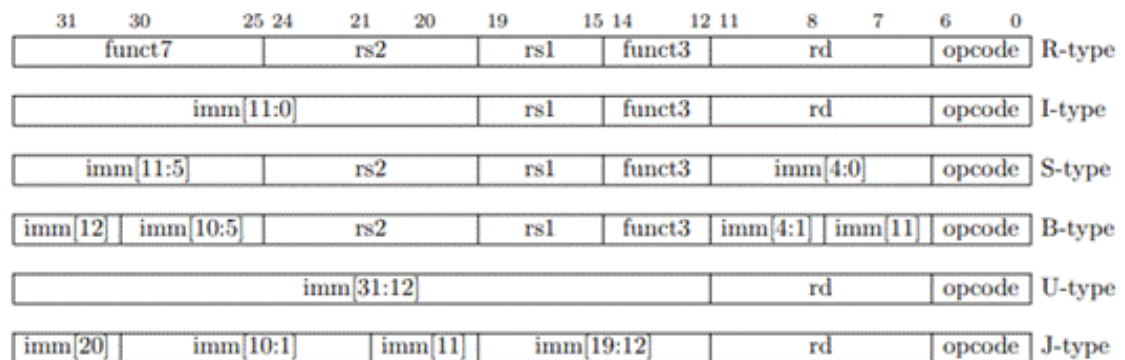


Figure 3-18: Different Types of Instructions

Register-type instructions in RV32I perform certain operations which generally involve a source register, second source register, and a destination register. The operation's result is stored in the destination register and the instruction is complete. Immediate type instructions follow the same methodology but replace a second source register with an

immediate value. The store type has two source register fields but no destination field, this is because the store type acts similarly to the immediate type instruction except the immediate value determines the offset from the rs1 address to find the address which acts as the destination address. Branching and Jump instructions can be lumped together because they perform very similar functions albeit with different conditions. Branch instructions change the program counter to a certain value based on a logical condition while 52 jump instructions do not waste space for a comparison to allow for larger jumps in the memory address. The upper immediate type is unique to LUI and AUIPC but is very similar to immediate type instructions except that LUI can also set the lower 12 bits in the destination register to 0 and that AUIPC edits the value of the program counter after it stores the PC value to rd.

All the instructions in RISC-V follow these formats, but the naming scheme for the fields can be quite opaque when shown without context so the individual bit fields will be explained in this section. The opcode is a 7-bit field which will always be the first part of the instruction evaluated by the processor's decoding unit and determines an instruction's type out of the different instruction categories. However, the opcode alone is not enough to determine which instruction is being written, because you can notice that all the R-type operations share the same opcode, so the funct3 and funct7 further define the operation being performed. For example, when computing addition or subtraction between two registers most implementations allow for the path inside the ALU to be the same but with a control bit on the adder's carry in bit. This bit of information can be extracted from a single bit in the funct7 section.

The other bit fields are very straightforward, any bit field starting with the letter 'r', such as r1 or rd, denotes one of the 32 registers. Immediate fields are also very straightforward, simply include the desired operand in the instruction when programming and the immediate will be used as the data for the operation. The only confusing part of using an immediate based instruction is that the immediate bit field is not always formatted in little endian, for certain instructions, such as Jump-type instructions, the immediate value is split into many sections and shifted around. This is an intentional feature for RISC-V and is used to make most pipelines more efficient and the shifting about of the immediate is not very difficult to implement. There is also another immediate field not represented by imm, and that is the shamt field which is a 5-bit field that stores the amount of positions which a value can be shifted. The shamt is upgraded to a 6-bit field in 64-bit XLEN RISC-V extensions because that then allows for values to be shifted up to 63 bits rather than the allotted 31 bits with a 5-bit field shamt.

The last major part of the RV32I architecture that should be elaborated on is how memory is generally managed for RV32I based processors. Memory management is simple and the LW, LH, and LB instructions along with the SW, SH, and SB instructions handle a majority of the memory management. Since RISC-V is a load store language these load and store instructions are constantly being called so the hardware needs to be very optimized to perform these instructions quickly. The language is byte addressable but since data must be aligned in RISC-V words need to be aligned to a multiple of 4 bytes, halfwords to 2 bytes, and bytes to bytes. As stated multiple times before, this alignment helps with optimization.

## 3.7 Design Languages

Since this project will require the programming for hardware components, we will need to use Hardware Description Language (HDL) to achieve our goals and objectives in this project. Some of the current HDL includes but is not limited to: VHDL (VHSIC Hardware Description Language), Verilog, SystemVerilog, SystemC, AHDL (Altera Hardware Description Language) and Bluespec SystemVerilog. In this section we will take a closer look at SystemVerilog, Verilog and VHDL since these are what we have learned from our past course curriculum and what Northrop Grumman have suggested to be utilized instead of using something completely new and that we have little no understanding of how it works. The reason for not using AHDL is because we have no knowledge of it, and we are also not using Altera.

### 3.7.1 VHDL

Out of the 3 language that will be discussed in this section, VHDL is the oldest one and was developed in the early 1980's. Surprisingly the language was developed by the United States Department of Defense (DoD), they began the program to develop high speed integrated circuit hardware technology. The program developed by the Department of Defense was released to the public in August 1985 which is when it was started to be used commercially. The program was used by the government to improve the documentation available of hardware development that was delivered to the government. Since the military is such a huge enterprise, the goal was to minimize costs of ongoing projects to operate the electronic systems. The military spent \$20 million dollars to develop this program which saw big benefits as in the military such technologies could have a lifespan that goes beyond 20 years. VHDL development was also supported by the industry, which greatly strengthened the language's usability. In 1986 the VHDL's rights were transferred to IEEE as the organization was seeking to develop the standardization of Hardware Description Language. Only in 1987 was the language ratified for industry use, which received the new standards with open arms.

The IEEE VHDL standard, officially recognized as IEEE 1076-1987, is a mandatory requirement for all military electronics contracts that incorporate ASICs (Application Specific Integrated Circuit). As it can be seen from the figure below, IEEE developed a top-down hierarchy which specifies that behavioral structure will need to be captured in new technology developments and because of these new requirements VHDL started being used since it was created for this purpose to fulfill the chart.

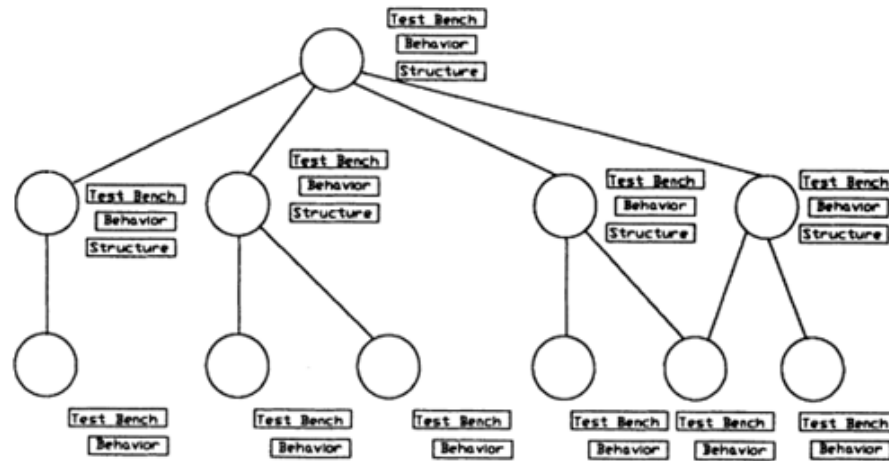


Figure 3-19: Behavioral Structure

In electronic design, there was a shift from traditional bottom-up such as gate level description to more mixed methodologies that reflects the growing complexity of hardware systems. Initially, VHDL allows us to simulate more complex systems before diving into the less granulate implementation details. This alternative topdown approach helps to resolve major issues earlier in the design process. Later in the design process, we incorporate bottom-up elements such as gates and flip flops by using standardized VHDL models and pre-built libraries from chip manufacturers. These resources often come pre-compiled, making them easier to integrate and simulate. This mixed design strategy is crucial for documenting both the structure and behavior of the system accurately and ensures that the final hardware aligns with our initial plans, which will result in more efficient designs.

To effectively simulate electronic hardware using VHDL, you need to have the structural description of the design, behavioral models for each device, stimulus, and configuration details specifying the version of each device model for simulation. For example, a schematic of an electronic circuit might show three NAND gates performing an and-or operation with four inputs. In VHDL, you'd define the architecture and entity with input and output ports and describe how these elements connect structurally and functionally. VHDL also allows you to specify how the circuit will be simulated by giving specifications to the inputs.

When we simulate electronic hardware using VHDL, it's really important to have all the necessary pieces in place. This includes the structural description of the design, behavioral models for each device, stimulus, and configuration details specifying the version of each device model for simulation. For example, when we're working with a schematic of an electronic circuit that shows three NAND gates performing an and-or operation with four inputs, in VHDL, we have to define the architecture and entity with input and output ports, and describe how these elements connect structurally and functionally within the VHDL framework. VHDL's capability to specify how the circuit will be simulated by providing specifications for the inputs allows for a detailed and comprehensive approach to 55 electronic hardware simulation, making sure that the behavior of the designed circuit is accurately represented.



### 3.7.2 Verilog

The second language we will talk about is verilog, standardized as IEEE 1364, also serves as a hardware description language for digital electronics, extensively used in both design and verification. The language was developed by Prabhu Goel, Phil Moorby, Chi-Lai Huang, and Douglas Warmke between 1983 and 1984 at Gateway Design Automations, it includes synthesizable constructs for hardware creation and non-synthesizable constructs for simulations, with its key versions being Verilog-95, Verilog-2001, and Verilog-2005. Some notable characteristics for Verilog include its case sensitivity, Verilog supports single-line (`//`) and block (`/* ... */`) comments, along with input, output, and bidirectional (input) port declarations. Data types such as wire and reg is used depending on various circuit design needs, while its support for concurrent and sequential constructs, a range of operators, and timing constructs like blocking and nonblocking assignments improves its modeling capabilities. Verilog's rich set of features includes functions, tasks, loops, and compiler directives, essential for detailed and effective simulation and synthesis in digital circuit design.

Verilog is considered to be a flexible HDL used in many different design scenarios and can be categorized into three primary coding styles: structural, behavioral, and RTL (Register Transfer Level). Structural design in Verilog focuses on outlining the physical structure of the circuit, employing a hierarchical methodology to connect small-scale digital blocks to realize moderately complex logic, such as a half-adder. On the other hand, behavioral design abstracts away the lower-level details (i.e. gate-level description), describing the operations between inputs and outputs through high-level algorithms, assuming a black-box approach to the underlying hardware. Also, the synthesizable design, often used in RTL coding, utilizes higher-level Verilog constructs to describe feasible device functionality for synthesis, bridging the gap between hardware predictability and software programmability. This last approach allows for translating complex algorithms into hardware implementations, effectively capturing the design intent through synthesizable code, often resembling structural or behavioral models in its formulation.

Combination logic can be modeled using two primary methods: continuous assignments and always procedural blocks. Continuous assignments use the "assign" keyword to directly connect the output variable to current input states without considering the event blocks. This approach is better suited for generating "glue logic" in RTL design, where multiple assignments execute parallel to maintain system efficiency. As a different alternative, the "always" blocks are used to describe combinatorial logic through sensitivity lists, which triggers logic evaluations based on input transitions.

Regarding the operators used, Verilog provides a comprehensive set of operators that handles arithmetic, logical, and comparison operations essential in digital circuit design. These are the arithmetic operators used in Verilog which includes: addition (+), subtraction (-), multiplication (\*), division (/), and modulus (%), facilitate the execution of basic arithmetic functions on binary operands. For logical operations, Verilog includes AND (&&), OR (||), and NOT (!) operators, which are instrumental in constructing combinational logic by returning a singlebit output based on the relationship between binary inputs. There is also: equality (==) and inequality (!=) operators allow for comparing operands, the result

is a Boolean outcome which is used for flow control and decision-making in synthesis processes. These operators form the building blocks for modeling complex digital systems.

An efficient RTL design optimizes logic gates by using constructs to model combinational logic. This involves using "assign" and "always" constructs to achieve desired outputs based on current inputs, which is very important when implementing logic gates like NOT. There are Techniques such as Karnaugh maps, Boolean algebra, and Shannon's expansion theorem used to minimize logic gate usage, this improves the design's performance and reduces its logic density.

Critical approaches for RTL design include using assign statements with conditional expressions to implement 2:1 multiplexers, recognizing multiplexers as a form of universal logic, and applying nested if-else statements for priority logic. The case and case structure is used to model parallel logic within procedural blocks, which covers all possible conditions, even those not explicitly defined. However, synthesis tools do not consider the sensitivity list specified within procedural blocks when modeling combinational logic, which could possibly lead to logic inference issues.

### 3.7.3 System Verilog

Lastly, we will talk about System Verilog. System Verilog is also a Hardware Description Language used to model digital systems. Verilog was developed in the mid 1980's, while System Verilog was introduced in 2002, serving as an extension to the development of the verilog language. The purpose of System Verilog is to increase the language's capabilities in design and verification as the chip complex keeps on increasing as we pursue to make chips smaller and smaller. We will first discuss some differences between the Verilog language and System Verilog, then we will dive deeper into the characteristics of System Verilog.

When Verilog language was created, the chip sizes were nowhere near what we are working with today. Verilog has shown to not be the most effective as complexity increased tremendously, System Verilog is supposed to handle that complexity by achieving many important characteristics desired in the chip 57 complexity such as: multi-core, multi-threading and PPA (Power, performance and Area).

Verilog has more limited data types and the basic ones include wire, reg, integer and real, while System Verilog offers more data types to achieve the more complex goals such as logic, int, shortint, longint, struct, union and string. By having more of these complex data types, the language is able to handle more levels of abstraction and provide more flexibility to the designer and the verification engineer.

The Verilog language does not really have a high-level verification construct and does not support interfaces which makes the task of handling a lot of signal sets very difficult. On the other hand, System Verilog has embedded in the language very powerful constructs such as constrained random testing, functional coverage, and assertions. It also supports UVM (Universal Verification Methodology). System Verilog also introduces the interface construct, which greatly simplifies the handling of connections between blocks.

Lastly regarding the comparisons, Verilog is a more static language which in return produces more static testbenches. Verilog also only has basic support for concurrency

through always blocks and initial blocks that are essential for Hardware Description Language. System Verilog provides more dynamic testbenches which greatly improves the simulation and verification capabilities. System Verilog supports new constructs such as program blocks and clocking blocks.

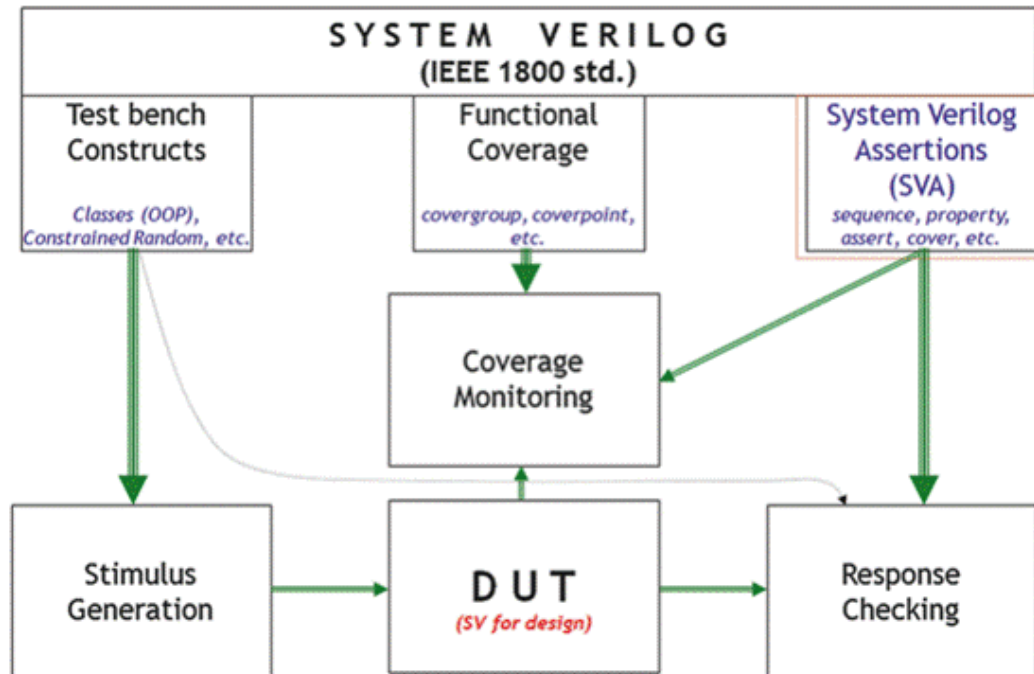


Figure 3-20: SystemVerilog Verification Environment Structure

As the figure above suggests, System Verilog Assertion which is a characteristic of System Verilog, has its own syntax and semantics that are not exactly equal to Universal Verification Methodology. All these different verification methods will be discussed later in the report. In Verilog 2001, there were added features such as Multi-dimensional arrays and auto variables, so functional coverage was consuming more resources in any given project. When the industry recognized the need to create a language that would design and verify a project, Superlog was created and later became what we know as SystemVerilog. At this time in SytemVerilog 3.1, more than 80% of the language had as its purpose for functional verification. The creation of System Verilog Assertions was really a combination from many different languages such as SUGAR from IBM, CBV from Motorola and ForSpec from intel. Many features were derived from these languages to be used in System Verilog Assertions.

System Verilog Assertion purpose is to check the requirements and specifications of the design to ensure the code does not have any violations. SystemVerilog really serves as a superset of the Verilog language, because it includes features such as creating random constraints that is absolutely essential for thorough verification of a complex digital system.

These assertions allow designers to embed verification directly into syntax, this allows the designers to continuously check the correctness during simulation.

After discussing the language options, the language selected is SystemVerilog. This language was chosen due to input from Northrop Grumman as they are the sponsors of the project and have mentioned how it is the most common language utilized in the industry currently. The language also provides a vast list of testbenches that will be analyzed in the next chapter.

## 3.8 Verification Methods and Timing

As we discussed previously, SystemVerilog will provide the most flexibility when it comes to testbenches, especially for complex digital system designs. One of the verification options we will cover is the Constrained Random Verification (CRV) and this method allows engineers to produce random test cases that follow certain constraints. Another option that will be covered is the Assertion Based Verification (ABV), this option was also talked about in the previous chapter and this option involves embedding assertions directly on the design. Assertions in their most simple form are essentially Boolean expressions that evaluate for specific condition or property. Additionally, we will discuss the Coverage Driven Verification (CDV) which is a very important metric when trying to understand how much of the design has been verified. Universal Verification Methodology that is built on top of SystemVerilog, this option is designed to be 63 reusable which makes it very scalable. Lastly, Formal Verification includes using mathematical methods to prove or disprove the correctness of a design.

### 3.8.1 Constrained Random Verification

The Constrained Random Verification was created to increase productivity in order to reduce the time it takes to cover more areas with fewer tests. This test also reduces the time to debug. The principle behind CRV is there will be random input sequences to a Device Under Test (DUT) and the boundaries of these inputs are set by the designer's constraints. The goal is to uncover maybe hidden states that would not be observed with standard testing, since testing every case manually is very impractical.

The CRV promotes much better coverage than directed testing for example, when the designer has to manually give the inputs that will be driven to the Device Under Test. Since the CRV generates gives random inputs, it often finds unexpected bugs. Another benefit of CRV is the automation potential, since you only need to set the constraints, and the program will automatically generate the input to stimulate the Device Under Test. CRV addresses two big problems from the traditional testbench approach: It is not declarative, and it is enumerative rather than comprehensive.

An important feature of CRV is generating simulation vectors, they can be deterministic or random based. It is common for a design to be tested by a mix of these approaches to have directed simulation to obtain known answers and then randomly to see which problems arise. According to our research we found a tool "Simgen". This tool uses Boolean constraints and Binary Decision Diagrams (BDD's) to handle constraints and biases at the same time. This tool seems to be efficient to avoid regular issues such as backtracking.

### 3.8.2 Assertion Based Verification

The second verification method we will look at is the Assertion Based Verification. Here are some characteristics of Assertion Based Verification. First, it needs to be modular which separates detection from action. This modularity makes it easier to reuse verification components, it simplifies code maintenance, and it helps with more advanced features such as error injection. Another principle of ABV is to be clear and target the assertions for simulation, the first part is to focus on the actual design intent and not get distracted with formal verification optimizations. An example of how modularity works is that the assertion-based monitor might detect an error on a bus and rather than taking immediate action on the violation, it simply passes the status to the other testbench components, then the system will take the appropriate actions. This framework allows for advanced functionality and enables more complex interactions and behaviors.

The process to create a reusable verification component begins with a block diagram and interface description are created to list the signals that will be monitored and asserted. After is an overview that describes the main characteristics of the design and uses waveform diagrams to represent the current relationship. The next step is to list all properties in natural language which is later converted to SystemVerilog Assertions. At the end, the assertions are grouped together within a module or interface, which forms an assertionbased monitor with analysis ports. This modularity makes the system interactions to be very efficient.

On-chip buses and standard interfaces play a crucial role in the structure of platform-based System on Chip (SoC) designs, they behave as essential conduits that tie together various IP blocks. Protocols such as the ARM AMBA Advanced High-performance Bus (AHB) and the Open Core Protocol (OCP) aren't just parts of these systems; they are at the heart of the design reuse strategies that drive much of today's digital design.

Arbiters are also part of ABV. Arbiters are important for resource allocation, fairness and deadlock prevention. Arbiters control the entry point of shared resources and ensure only one requestor is granted access at a time. They are also responsible to make sure no requestor are “starving” resources. As the systems get more complex, the demand increases, and the arbiters are there to make sure no catastrophic failure occurs because of the demand. A good way to see if Arbiters are behaving properly is by looking at the mutual Exclusion Assertions, the fairness of assertions and monitoring the allocation patterns to prevent deadlocks.

Controllers are important to manage the logic and operation of different systems. They are responsible for Control Flow Verification, Timing and Synchronization, state management, error handling and recovery, resource management and performance metrics. Controllers are responsible for managing the sequence of operations and ensuring they are executed as expected. Controllers verify that timing requirements are met, this includes setup times, hold times and response times. Error handling and recovery ties back to what we talked about earlier in modularity, since the controllers will identify the error and report back, then the controller initiates the appropriate recovery action.

Datapaths are responsible for the accuracy of data processing, timing verification, throughput optimization and functional coverage. DataPath's perform precise calculations,

data transformation and movements that are essential for digital device performance. Assertions help verify that the data paths meet critical timing requirements, and it checks for data hazard or race conditions.

There are 3 types of different assertions. They are: Immediate assertions, concurrent assertions and deferred assertions. Immediate assertions are nontemporal domain assertions and are executed as in a procedural block. They can be interpreted the same way procedural “if” statement and can only be specified where a procedural statement is specified. Concurrent assertions are temporal domain assertions that allow the creation of complex sequences using clock edges-based semantics. Concurrent assertions are edge sensitive and not level sensitive. Deferred assertions are a type of immediate assertions, so they evaluate immediately without waiting for all the variables to finish your respective combinatorial expression which means they are more prone to glitches.

### 3.8.3 Coverage Driven Verification

Thirdly we will talk about Coverage Driven Verification which is also a powerful tool. CDV is made up of 3 components: stimulus generation, coverage collection, and response checking. The coverage metrics really set this method apart from the other methods. These metrics can include how much RTL code has been exercised by the tests which includes line coverage, toggle coverage, branch coverage and statement coverage. CDV heavily relies on EDA tools, CDV is very efficient and comprehensive. It also uses great automation to generate and analyze results. The disadvantage of CDV is that it is complex and there quite of upfront work to be accomplished.

### 3.8.4 Universal Verification Methodology

Now we deal with Universal Verification Methodology (UVM). UVM is a standardized methodology used for verification purposes of Integrated Circuits (IC) and System on Chip (SoC) design. This method was also developed because of the complexity levels that continue to increase as engineers are looking to put more transistors inside a very tiny chip. UVM is able to provide structure and modular framework to create reusable verification code. UVM reduces the duplication of effort and combines multiple strategies together such as constrained random verification and functional coverage. UVM is great to perform systematic, repeatable verification that leads to a more efficient digital design.

The core principles of UVM are Modularity, Reusability and scalability. UVM is supposed to be modular as in allowing each component such as monitors, drivers to be designed, tested and debugged independently from each other. The UVM in theory is supposed to be able to be used across different teams which would greatly reduce the time and effort to set up testbench. UVM can be used for small scale projects to very complex projects throughout the development cycle. UVM provides a large set of base classes that can be used as a foundation for customized verification components. The following is the UVM testbench hierarchy: Test is the top level, and it manages the configuration and execution of the test scenarios. Environment has multiple agents and components and defines the verification environment for a specific DUT. Agent represents a specific aspect of the DUT's interface which typically contains a sequencer, driver, and monitor.

The sequencer controls the generation stimulus using sequences which are the objects that define a series of transactions to be driven to the DUT. Drivers translate these transactions from the sequencer to signals on the DUT's interface, while the monitor observes the signals on this interface and translates them back to transactions which are then forwarded to coverage collectors.

Lastly, we are going to go over Functional Coverage. The difference between code coverage and functional coverage is that code coverage directly comes from the design code and does not require the designer to change the structure of the code, however code coverage will not catch the intent of the code. A thorough testbench is necessary to actually see if code behaves as it should. The main idea in Functional Coverage is to know the code functions and works as it is intended. Functional coverage is specified by the user, it measures coverage of the design intent, and it is control oriented.

### 3.8.5 Timing Issues in Digital Circuits

As the technology behind digital circuits continues to advance rapidly, the demand for faster and more complex circuits has also increased drastically. Timing has emerged as a critical factor in the design and functionality of digital circuits, particularly in high-speed systems that rely on a sequential clock. Timing issues lead to critical failures, often caused by data corruption due to incorrect signal synchronization. This section explores the fundamental timing challenges that designers must address to ensure reliable operation in today's digital circuits, focusing on clock skew and jitter.

#### **Clock Skew**

Clock skew is the difference in timing when a clock signal reaches various points on an integrated circuit (IC). While the distance the clock signal needs to travel is typically very short, at high frequencies even a minor distance can cause a signal to arrive at a slightly different time. Several factors can cause a skew such as the material the signal is traveling through as well as the physical distance the signal needs to travel to the destination. These timing differences are clock skews and can impact the performance and accuracy of integrated circuits.

#### **Positive Skew**

Positive clock skew occurs when the clock signal arrives at the destination register later than it arrives at the source register. For a synchronous circuit, this means that the clock edge arrives at its destination register later than it arrives at the source register. This allows time for the data to propagate from the source to the destination as the destination is waiting for the clock signal to arrive.

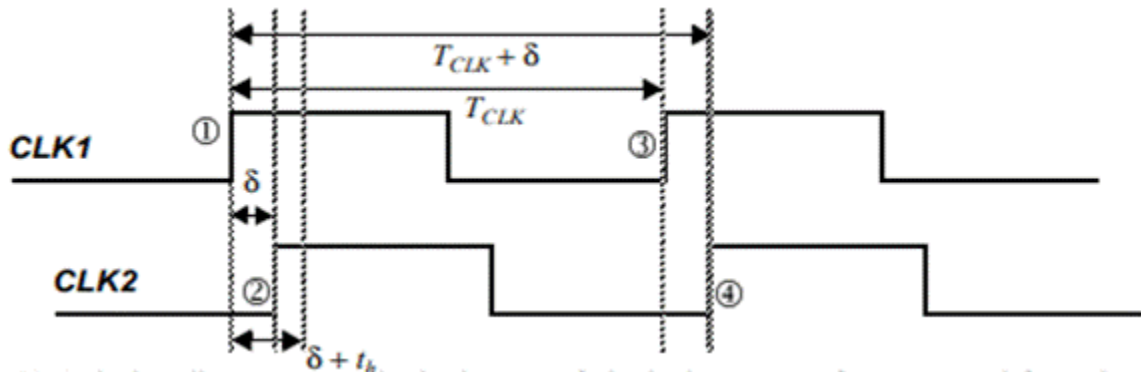


Figure 3-21: Positive Clock Skew and Timing Offset between CLK1 and CLK2.

In an ideal situation without having any skew, CLK1 and CLK2 would tick simultaneously, and they would receive their signals at the same time. However, in the real world the conditions we operate in are not ideal and they can introduce small delays due to materials used, or the physical distance they need to travel. This means a positive skew could be introduced into the system. For example, if the clock period we are running at is 10ns, and due to the physical design of the IC a 2ns positive skew is added, CLK2 would now receive the signal 2ns later giving the data an extra 2ns to propagate from the source to the destination.

### Negative Skew

Negative skew occurs when the clock signal arrives at the destination register earlier than it arrives at the source register. This reduces the amount of time data must propagate from the source register to the destination register.

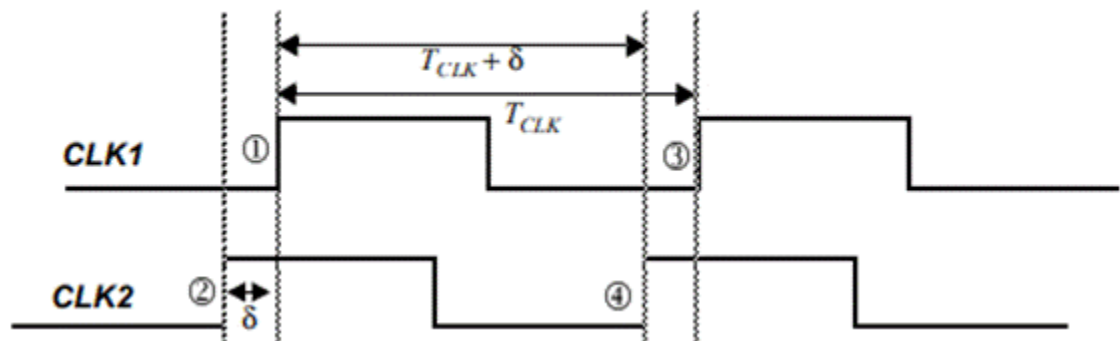


Figure 3-22: Negative Clock Skew and Timing Offset between CLK1 and CLK2.

Similarly to the last example, in an ideal world CLK1 and CLK2 would tick simultaneously, keeping the clock signals in sync, however the physical limitations of the materials used, or the distances traveled can cause the clocks to be out of sync. For example, if the clock period is 10 ns again, but this time there is a 68 negative skew of 2 ns, CLK2 would receive the clock signal 2 ns earlier than CLK1, meaning the data has 2 ns less to propagate from source to destination. This reduced propagation window increases the risk of hold time violations, where the data is not stable when captured by the destination register.



## **Clock Jitter**

Unlike clock skew, which measures the deviation of the clock signal's arrival time between two different points, clock jitter refers to the timing variations of the clock signal at a single point across successive clock cycles. In simple terms, jitter refers to the random shifts when the clock edge arrives, causing the clock signal to sometimes be a bit earlier or a bit later than expected with each cycle.

Clock jitter can be classified into two categories, random and deterministic. Random jitter sounds like the name implies and occur randomly without any real reason for it to occur, meaning the solutions are a bit more complex when dealing with random jitter. Deterministic jitter on the other hand, is a bit more repetitive and occurs periodically throughout the system. While it can still be difficult to pinpoint exact causes for the deviations, because it is repetitive you can usually pinpoint the cause for deterministic jitter.

## **4 Related Standards and Design Constraints**

This section contains information on the standards used, as well as all the constraints that were considered.

### **4.1 Related Standards**

The following standards discussed in this section were found to be useful when developing our digital ASIC tapeout. At this time not all of the standards being used have been fully decided upon, but most of the important ones are listed below. As more standards are considered and implemented we will expand the list as necessary. This section aims to give a compressive overview of the standards we are using, as well as the impacts each standard has on our design process.

#### **4.1.1 RISC-V32I ISA**

RISC-V is an instruction set architecture (ISA) that is becoming increasingly popular in both academia as well as industry due to its robust ecosystem and open-source nature. The ISA was developed in a research lab at the University of California, Berkley in 2010 with simplicity and openness in mind. This allows anyone to use the ISA free of charge unlike most proprietary ISA's typically seen in many modern processors. There are many different versions of the standard and for this project we are sticking to the 32-bit integer implementation of the ISA which allows us to focus on learning the design flow without needing to worry about complex floating-point operations.

#### **4.1.2 Impact of RISC-V32I ISA on Design**

The RISC-V ISA is one of the most influential standards being used for this project as it defines the instructions that our hardware must support. This is pivotal as the entire design process revolves around implementing this ISA into a ASIC design. RISC-V is a good choice as it follows the reduced instruction set computer (RISC) model which means there are far fewer instructions that need to be handled with hardware compared to a complex instruction set computer (CISC) model such as x86. For the scope of our project the main

advantage that the RISC model has over CISC is the number of instructions that need to be implemented into the hardware. In a RISC model, there are far fewer instructions the hardware needs to worry about, leaving it up to the software to handle more of the complex operations. For our team this is a advantage, as we are new to the design process and the simplicity of the RISC model allows us to focus more on the design flow, rather than worrying about a set of more complex instructions the hardware would need to handle. Moreover, we are only using a subset of the RISC-V ISA known as RISC-V32I which is a 32-bit integer version of the standard. This further removes complexity as we are not worried about floating-point instructions which come with more complexity. As a final point, the RISC-V ISA is open-source allowing the team to use it without licensing or royalties which can introduce complexity and be expensive.

#### 4.1.3 IEEE 1800-2017 (SystemVerilog)

The IEEE 1800-2017 standard, also referred to as the SystemVerilog standard aims to provide engineers and developers with a set of unified methods for hardware design and verification. SystemVerilog is a extension of the Verilog hardware description language (HDL) created by Accellera in 2004 with an aim to increase productivity in design and verification of systems. Its main purpose of creation was to combine the system design with verification features into a single language, allowing developers to design and write testbenches in a uniform and consistent way. The wide spread adoption of the SystemVerilog standard enabled more computability across different electronic design automation (EDA) tools, providing interoperability in industry which saves time and money.

#### 4.1.4 Impact of IEEE 1800-2017 (SystemVerilog) on Design

The SystemVerilog standard is another important standard used for this project as it defines the language and framework we will use while designing and verifying our system. SystemVerilog offers a powerful set of tools not only used to describe our hardware but also supports robust verification techniques. The capabilities that SystemVerilog offers will allow our team to ensure that our design meets industry-standard performance metrics while still achieving our target code coverage. Moreover, it also is a widely accepted standard and works well with the Cadence Suite which is the EDA toolkit our team is using for the project. Finally, SystemVerilog supports the Universal Verification Methodology (UVM) which is the standard we will use to verify our design.

#### 4.1.5 IEEE 1800.2-2020 (Universal Verification Methodology)

The Universal Verification Methodology (UVM) is a application programming interface (API) that allows developers to create scalable, modular, and reusable components that are used for the functional verification of designs. The UVM standard was adopted by IEEE in 2017 and revised in 2020, and is maintained by the Universal Verification Methodology Working Group. This standard aims to improve the interoperability for projects and EDA tools by providing a base class framework, allowing a variety of tools to reuse the same testbenches for verification. This approach does not only improve the quality of designs but also reduces the cost of development as the testbenchs can be reused or modified from other projects.

#### 4.1.6 Impact of IEEE 1800.2-2020 (UVM) on Design

The Universal Verification Methodology (UVM) standard will play a significant role in completing and verifying our design. UVM provides a structured way for our team to perform functional verification, which is essential for ensuring the reliability and the accuracy of our design. The UVM standard has a few major impacts on our design process, the first being the reusability that UVM offers. Because UVM tests are broken down into components, this allows the verification team to reuse a lot of the base code for writing testbenches which saves a lot of time. This modular approach not only saves development time, but in production also minimizes costs and redundancy. UVM also offers improved collaboration within the verification team, as different team members can work on different components concurrently while utilizing the same base classes. This streamlines the verification process, and allows for a boost in efficiency. Moreover, UVM was designed to scale as projects get larger and more complex. This may not appear to be important for our project, as our design is relatively simple, however we are following industry standards for our design and want the flexibility to be able to scale up the design's complexity if needed, which UVM will allow. Finally, UVM is widely adopted across a variety of industry standard EDA tools, which makes the code modular and portable across most of the standard tools used in industry.

#### 4.1.7 FreePDK45

The FreePDK45 library is an open source 45nm process design kit (PDK) that was developed at North Carolina State University for academic and research purposes. This PDK provides our team with the resources needed such as design rules, the standard cell, and the layout guidelines that our design must follow to be functional and compliant with industry standards. But unlike proprietary PDKs, which come with a significant licensing cost and often NDAs, FreePDK45 is free to use making it a solid choice for our design. FreePDK45 allows us to develop and test our digital ASIC tapeout using industry standard tools and a node that is similar to industry standard technologies while avoiding the costs associated with a commercial PDK.

#### 4.1.8 Impact of FreePDK45 on Design

The FreePDK45 standard allows our team to use a 45nm node, which is similar to real world manufacturing standards, while being free and open source making it a solid choice for our design. It is a well-documented and widely supported standard for semiconductor research, meaning our team has many resources accessible for learning and troubleshooting. This standard allows us to gain hands-on experience with a technology node that is similar to industry standards even though it is larger than most nodes used today. The open-source nature and standardization of the design rules allows our team to focus on designing and verifying our ASIC without worrying about the costs and licensing agreements that come with proprietary PDKs. The PDK also works within the Cadence Software Suite, which is essential as Cadence is the EDA tool being used for the design of our ASIC.

## 4.2 Software Constraints

The software constraints placed on this project stem from the specialized tools and requirements needed for designing, simulating, and verifying our ASIC. These constraints are influenced by a variety of different sources such as licensing costs, accessibility, and compatibility, which affects the tools our team can use to develop our ASIC. Given that ASIC design is highly specialized, industry standard Electronic Design Automation (EDA) tools are essential for development, but the cost of these tools create limitations. For this reason, the tools and software we are using were provided by the university and project sponsor, complemented by free resources when possible.

Cadence is the primary EDA tool used for this project, offering a compressive suite of tools needed for design, simulation and verification. The Cadence suite provides the team with the industry standard tools that allow for complex ASIC development, including schematic capture, layout design, and functional verification. However, Cadence and other EDA tools come with substantial costs and licensing agreements which can make it a hard barrier to overcome. Fortunately, the university as well as the project sponsor have covered the costs associated with the Cadence EDA suite, allowing us access to the tools we need without the financial burden. This sponsorship is vital for our project, as access to professional EDA tools is essential in developing our ASIC to industry standards. This constraint limits us to using Cadence, as it is the only resource available to us. However, Cadence is the industry standard, making it an excellent choice for this project.

## 4.3 Manufacturability Constraints

Manufacturing an ASIC is a time consuming and expensive process, and is out of the scope of this project. However, as this project aims to teach our team the entire design process of ASICs, so it is important to briefly cover the constraints associated with manufacturing our ASIC and why it is not feasible or even possible. The most pressing issue our design has when it comes to manufacturability is the design node used. FreePDK45 is an open-source PDK node which was designed for academic and research, thus there are no foundries that can actually produce the node. This means that even if we wanted to, there is no foundry that has the capabilities of producing our design. This is not a issue however, as this project is concerned with the design process and does not require us to manufacture our design. In addition to our design node lacking foundries for manufacturing, the costs associated with manufacturing a ASICs are extremely high, and we lack the budget needed to actually manufacture. So, even through manufacturing was not a requirement for our project, the technology constraints as well as financial constraints would prevent us from being able to physically produce our design.

## 4.4 Economic Constraints

When designing an ASIC there are a plethora of economic constraints that need to be considered, especially due to the highly specialized nature of ASIC design such as specialized software, tools, and resources. Many Electronic Design Automation (EDA) tools, such as Cadence which are used for this project, come with a substantial licensing fee which can often cost thousands of dollars per seat annually. The cost of these tools are often out of reach for students and academic projects due to the pure cost of the tools.

Fortunately, our university along with the project sponsor have covered the licensing costs for Cadence EDA, providing us with the industry standard tools needed to develop and verify a industry standard ASIC. This sponsorship allows our team to overcome the financial hurdle associated with ASIC projects, and without it our team would not be able to deliver a industry standard ASIC design.

Another economic constraint considered related to the design was the Process Design Kit (PDK) used. For almost all industry standard ASICs, a proprietary PDK node is used. These proprietary nodes often come with a licensing agreement, which can be costly. Proprietary PDKs also often have more complex regulations attached with them, restricting how they can be used, as well as who you can share them with. By using FreePDK45, an open-source PDK, it eliminates the cost associated with proprietary PDKs, as well as gives us more freedom on how we can use the node in our design while still gaining the experience as if we were using a proprietary node. This choice significantly reduces project costs while still aligning with the objectives of our project.

Moreover, the cost to manufacture a ASIC is probably one of the most expensive costs associated with the process. Fabrication of ASICs often come with significant expenses related to production and testing, which can cost millions of dollars just to get to the production stage. For this reason, our project is limited to the design and verification stages, allowing us to learn the design process all the way up to actual tapeout. This allows our team to learn the skills needed to produce a ASIC, without needing to worry about the manufacturing costs associated with the process. Additionally, as mentioned in the previous section the FreePDK cannot be manufactured.

In summary, there are a considerable number of economic constraints that needed to be considered for this project, mainly the licensing fees for the Cadence EDA Suite. While we are constrained to the tools provided, they are the best tools for the job and used in industry. By relying on the resources provided to us by the university, as well as the sponsor, while also embracing open-source technologies, we are able to mitigate many of the costly expenses that would prohibit this project from being possible.

## 4.5 Time Constraints

Time is one of the most significant constraints that our team faces for this project. ASIC design is known for its long and complex development times even for experienced professionals. As a group of students who is new to ASIC design there is quite a bit we must do even before we start our design. On top of learning, and developing our ASIC for this project we must also balance the workload with our other academic responsibilities, which further limits the amount of time we can dedicate to each phase in the design process. Our timeline is also constrained by the semester, so it's crucial that we establish deadlines for completing the design, verification, as well as documentation for the project.

The first major time constraint our team faced revolves around the training required to learn and use the Cadence EDA tools. Our team is new to the design process, and new to Cadence as well. This introduced a complex learning curve before we could even start developing our ASIC. To learn the tools, Cadence provides a comprehensive set of training videos and lectures to help new engineers learn the tools. However, while these training packs give us the detailed knowledge needed to use the suite, it also takes a significant portion of time.

Each module can take hours to complete, let alone truly understand how these tools work. All together each team member has around 180 hours of training modules that must be completed. This is a huge workload in itself and must be managed accordingly at the risk of falling behind. For this reason, it's important that each team member is constantly working on the training modules, as the knowledge gained is crucial for our final design.

Moreover, the early stages of development required a long and thorough considerations of how our system would operate to meet our performance metrics. We had to evaluate our performance constraints, as well as the technologies we would use to achieve these metrics. This process involved considering high-level choices such as whether to implement a single-cycle, multi-cycle, or a pipelined design, while considering the time we had to implement them as well. These early stages of development required a lot of thought and consideration, even though there was not a lot of code written. It was critical to ensure we had a strong foundation before we proceeded further with the design which took a couple of weeks to figure out.

Another major time constraint our team will face has to do with verification of our design. We will be using the Universal Verification Methodology (UVM), which is a standard used for verification. Our team has never used UVM, so there is going to be a learning curve associated with the concepts, framework and best practices. This requires our verification team to spend additional time to learn the UVM principles, as well as time to develop the testbenches for verifying our design.

Overall, this project has presented a significant time constraint, especially with the added challenge of learning the technologies needed to develop an ASIC. The steep learning curve required for learning these tools, alongside the actual design work, requires that our team stay on top of meeting each deadline, as any delay could set us back substantially. Additionally, we experienced unforeseen time constraints beyond our control, as a week of design and development time was lost due to natural disasters that disrupted school courses. These challenges have highlighted the importance of adaptability and planning, and our team remains committed to managing these constraints effectively to ensure the successful completion of our project.

## 5 Artificial Intelligence

### 5.1 Chat GPT Case studies

During the write up of this assignment, the current version is Chat GPT – 4. While there have tremendous advancements in generative AI in 2024, it is still far from perfect. There are cases which this resource has shown to be helpful and other cases where it may lead you in the wrong direction as we will see in some cases below.

Some of the weakest points from Chat GPT includes doing calculations using advanced mathematics, so those should be done using a more precise software such as Wolfram Alpha or doing them by hand. Chat GPT may also lead you in the wrong direction when the topic being discussed is very novel and the information is not very solid.

### 5.1.1 Case Study 1

**Question:** “What are the different verification methods for digital systems?”

**(See AI Appendix for output)**

**Analysis:** In this example provided by Chat GPT you must have enough knowledge prior to asking a question like case question 1, because it will provide very broad and sometimes answers that are not necessarily pertinent to the task at hand. Also, asking the same questions in different ways may also provide you with answers that are more pertinent to your particular use.

Regarding University-level projects, there may be better resources to learn from and do better research. There are resources such as Google Scholar, which is widely used by Academia and regarded as a reliable source to perform necessary research. A disadvantage from a resource like Google Scholar is that it does not provide a simplistic answer to your inquiry, rather what are the articles and the Academia environment that pertain to your specific inquiry.

### 5.1.2 Case Study 2

**Question:** “what are the different hardware description languages and their benefits?”

**(See AI Appendix for output)**

**Analysis:** On a question with a much smaller scope, we see that Chat GPT performs very well in portraying which are the hardware description languages currently being used in the industry, but it gives a very high-level overview of their benefits. To extract more information from Chat GPT, it is necessary to keep asking follow up questions so that you can dive deeper into each language to make a more informative decision that would best apply to your requirements and needs. We 78 can see that on case where there are not that many different common hardware descriptions languages, it does a good job listing them out, but it performs poorly when you would like more information about their differences and why you would pick one rather than the other because of their similarities.

### 5.1.3 Case Study 3

**Question:** “what is RISC-V architecture?”

**(See AI Appendix for output)**

**Analysis:** When dealing with more definition, Chat GPT performs very well in hitting the main points. Although it does not explain everything in detail, because its answers are limited to a certain amount of content. These main points are what you would be most likely to see as chapter names if there was a book written about RISC-V architecture. We see this example as with the other 2 case studies, that Chat GPT answers lack in depth and one simple question may be answered properly if it is a very basic question, but in topics that require more depth, one question is not sufficient to understand it fully and the user would be required to keep asking question about those main point to try and dive deeper, but it would more beneficial to get a good article from Google Scholar for example.

### 5.1.4 Case Study 4

**Question:** ???

**(See AI Appendix for output)**

**Analysis:** In this example provided by ChatGPT, we can observe that it once again gives a very high-level overview of the main characteristics of each architecture, namely Von Neumann and Harvard. Like the cases above, the researcher would need to keep on investigating the topics that are relevant to make an informative decision to which architecture best applies to the specific application at hand.

## 6 Administration Content

### 6.1 Budget and Financing

This project does not require any hardware, and any software not currently accessible to UCF students will be provided by Northrop Grumman.

### 6.2 Phase One Milestones

**Week 1:** Meeting With Northrop Grumman team

- Student team meets with NG team
- Q&A project goals, objectives and expectations
- Review project timeline and deliverables
- Setup GitHub for all members

**Weeks 2 – 3:** Initial Design Specs and Block Diagram

- Detailed project specifications
- Development of the High-level block diagram of ASIC design
- Development of the High-level block diagram of ASIC design

**Week 4 – 6:** RTL Design Commencement

- Reassess coding standards and documentation
- Begin writing HDL code for the RISC-V base instructions set
- Review RTL among team members

**Week 7:** Functional Verification Plan

- Develop verification plan using SystemVerilog and UVM at the center
- Create testbenches and test cases for RTL functionality verification
- Identify metrics to track verification progress

**Week 8 -10:** RTL Design Finalization

- Completion of RTL for all modules
- Perform code reviews and fix identified issues (bugs)



- Make sure all blocks have proper integration and the design demonstrates effectiveness

#### **Week 11 – 13: Functional Verification Start**

- Use planned functional simulations testbenches on the completed RTL design
- NG labeled goal: Achieve initial functional verification with a focus on reaching 85% code coverage. - Identify and address any bugs or discrepancies.

#### **Week 14: Project Review**

- Progress presentation to Northrop Grumman and advisors
- Presentation must include RTL design, verification results and challenges faced and overcome
- Discuss project plans for next phase

#### **Week 15 -16: Synthesis and Preliminary Timing Analysis**

- Synthesize the RTL design into a gate level netlist using Cadence (Requires training)
- Learn and perform preliminary static timing analysis to identify timing issues
- Prepare Place and route design for physical design in

### **6.3 Phase Two Milestones**

#### **Week 1 – 4: Place and Route Process**

- Begin physical design process using Cadence
- Perform floorplanning, placement and routing of the up to date synthesized netlist
- Ensure that the design meets area, power, and timing constraints
- Work closely with advisors and NG engineers to achieve optimal results via recommendations

#### **Week 5 – 6: Design Rule Checks and Layout vs Schematic**

- Perform Design Rule Checks to ensure manufacturability
- Perform Layout Versus Schematic checks to verify layout correctness
- Address any violations

#### **Weeks 7 – 8: Final Timing and Power Analysis**

- Reperform extensive static timing analysis to ensure the design meets timing requirements
- Conduct IR drop and electromigration analysis using Cadence
- Optimize power distribution and ensure the design operates within specified power limits

#### **Week 9 – 10: Design for testability Insertion and Testing Plan**

- Insert Design for Testability 84 Insertion and Testing Plan (DFT) structures, such as scan chains and BIST.

- Develop a detailed testing plan for post-fabrication validation.
- Simulate the design with real clock delays using gate-level simulations (Stretch Goal).

#### **Week 11 – 12: Final Design Sign Off and Tapeout Prep**

- Ensure all Design Rule Checks, Layout Versus Schematic, timing and power checks are clean (for the most part) and signed off
- Prepare the GDSII file and other required documentation for tapeout
- Review the design with NG team and advisors for approval

#### **Week 13: Final Project Review and Documentation**

- Prepare final presentation with final design, verification results, and tapeout preparation
- Complete and finalize all project documentation which includes the design report, verification reports, and lessons learned by all individual members.

#### **Week 14: Mock Tapeout**

- Perform a mock tapeout of the design to simulate the actual tapeout process (Brainstorm and discuss further on this).
- Ensure everything is ready for the tapeout; this includes all files and docs
- Conduct review of project process
- Meet with NG team and advisors

#### **Week 15 – 16: Project Presentation**

- Final presentation of the project to NG team, advisors and class
- Highlight achievements, challenges, and experience as a whole

## **6.4 Work Distributions**

*Table 6-1: Group 1 members and responsibilities*

<b>Group 1</b>	<b>Responsibilities</b>
<b>Youssef Samwel (ECE)</b> <b>Team 1 lead</b>	Instruction Fetch Unit (IFU)
	Instruction Decode Unit (IDU)
	Develop logic to fetch instructions from memory.
<b>Nicolas Sayed (ECE)</b> <b>Team 1 member</b>	Develop Execution Unit and Register File
	Design the ALU
	Implement Logic to update registers

*Table 6-2: Group 2 members and responsibilities.*

<b>Group 2</b>	<b>Responsibilities</b>
	Designing the Memory Access Unit

<b>Davi Dantas (ECE)</b> <b>Team 2 lead</b>	Ensure load/store functions properly
	Implementation of cache policies
<b>Timothy Ogg (CPE)</b> <b>Team 2 member</b>	Design Control and Status Registers
	Focus on Bus Interface

*Table 6-3: Group 3 members and responsibilities.*

<b>Group 3</b>	<b>Responsibilities</b>
<b>Evan Kasky (CPE)</b> <b>Team 3 lead</b>	Designing the Clock and Reset Logic
	Designing Power Management
	Implement Clock Generation
<b>Pablo Rodriguez (CPE)</b> <b>Team 3 member</b>	Develop functional verification
	Write testbenches to verify each individual component
	Write RISC-V assembly program to test CPU functionalities.

## 6.5 Introduction to Agile Methodology

Why pick Agile over other existing methodologies? Agile gained great usage in many industries especially in software development because it offers several benefits over other traditional project management methods.

The adaptability to change is one of six reasons as to why Agile was chosen. The methodologies iterative system allows for quick adaptability in almost guaranteed changes to requirements and stakeholder feedback. Although our project has a plan, the team is not locked into it thus allowing readjustable priorities, timelines, and deliverables based on feedback and changing needs. The second reason we decided on selecting Agile was due to our expectation on faster delivery. Due to the incrementation nature after each sprint (weeklymanner), this would allow the team to put usable versions of the product together and receive feedback from stakeholders much earlier than other methods that deliver finished products at the end of a long development cycle.

We expect that because of the frequent testing and regular feedback, the team can identify and fix issues early in the development process. In doing so this will help reduce the risk of defects being discovered later in the project; this would result in more reliable and higher quality work, and in time, the final product. Stakeholders are involved throughout the entire process from the initial planning all the way to the last sprint. They would also review every single sprint and provide frequent feedback. With retrospective meetings after each sprint improvement can be more definitive and frequent which fosters an environment for

learning and improvement. Which goes really well with the code of conduct in the team with empowerment and ownership given the control to how we achieve success and how everyone holds their accountability throughout each sprint.

## 6.6 Deliverable Requests

### 6.6.1 Overview of First Deliverable Request

Our first division of labor and research practice milestone requests centered around the initial research phase as well as an general understanding of how to divide the group to specialize on topics and have a good foundation in what we considered to be the basic topics; the must know. Below is the deliverable request and the tables for it.

One of our goals is to be able to understand the research we do and also be able to break it down thoroughly. To achieve this goal we have to do specific group research and general team research. In the process of researching each team member is expected to document everything they find and be able to summarize such findings in their own research journals. Now I know this sounds like a pain, however I have to say that it is effective. These journals should be shared with other team members as well as the resources collected. The next goal of our research is that we hope to be able to start building an initial block design with respect to the behavior of the RV32I ISA. To do this each member should try to create their own block design but put more emphasis on how their respective focus works; if you can make a general design, that works too. Make it basic and refine it. If you can find pipelined information with regards to the RISC-V architecture that is a plus.

#### **Required General Topics for Everyone:**

- RISC-V Basics (RV32I Instruction Set)
  - Core instruction set
  - Instruction formats
  - Immediate value formats
  - Instruction decoding handling
- Pipelining Concepts
  - How it works especially for the 5 stages fetch, decode, execute, memory access, and write-back stages.
  - Pipelining hazard; structural, data, control hazards
  - How to handle hazards
- Cache Architecture and Memory Hierarchy
  - Cache concepts
  - Cache organization
  - Cache coherence
- Basic Computer Architecture
  - Basic CPU components; ALUs, register files, memory and control logic
  - Understand control unit for flow of instructions and data

#### **Team CPU Research Focus:**

- Instruction Fetch Mechanisms
  - Design and optimization of IFU units

- How the program counter PC works
- How branch predictions can be implemented
- Understand instruction prefetching and a plus would be if you learn how to reduce stalls in the pipeline
- Instruction Decoding and Control Signals
  - How does it work especially for extracting opcodes and operands
  - How are control signals generated for different instruction types
  - How does the control unit manage data paths and the execution flow within the CPU
- Handling Control Hazards and Pipeline Stalls
  - How do we handle these control hazards; pipeline flushing, delayed branches, dynamic control management

**Goal:** to understand the fetch and decode stages of the pipeline and learn how the control unit works and how it coordinates the CPU's operations.

#### **Team Memory Research Focus:**

- Memory access and Load/Store operations
  - Research the load and store instructions of RV32I
  - How does the memory access unit handle such ops
  - Coordination of memory access between CPU and L1 cache (memory alignment)
- Cache Architecture
  - Cache design
  - Cache organization
  - Cache line replacement policies and handling cache misses
  - Interactions between L1 cache and memory access unit
- Write Back Mechanisms
  - Research Final Stage of the pipeline where results are written back to the register file.
  - How to manage dependencies in the write-backstage. A plus if you understand it in a pipelined architecture.

**Goals:** Understand Memory access stage and how to design L1 cache as well as know how the write backstage works and how the results are fed back into the register file.

#### **Team Verification Research Focus:**

- ALU Design and Arithmetic Operations
  - Design of ALUs
  - Understand the arithmetic with regards to the RV32I instructions set and how it handles
- Register File Operations
  - How to design and implement a register file especially for multiple reads and writes in single cycle processes
  - How register values are fetched for execution and how are results written back after such execution

- Pipelining and Hazard Handling
  - How does the execution stage fit into a pipelined architecture.
  - How are Pipelined hazards caused
  - How are they resolved
  - Consider a hazard detection unit
  - Study branch prediction and branch delay slots
  - Pipeline stalling

**Goal:** Understand the execution stage of the pipeline, with a focus on ALU operations as well understand the register file and how it is accessed.

### 6.6.2 Results of First Deliverable Request

All team members complied with the soft deadline and produced their research journals on these specified topics. At this point of the project the teams were established but were subject to change after the first sprint. By this point of the project, we were getting ready to transition to the initial block design deliverable which was a continuation of this research phase.

### 6.6.3 Initial Block Design Deliverable Request

Below are the goals, objective and expectations for the team meeting. With the deadline being September 18-19, 2024

**Goal:** To have all team members comprehend the single cycle and 5-stage implementation of the RV32I RISC-V Processor and produce a block design.

**Objective:** All team members will develop a single cycle AND classic 5 stage pipeline block design to the best of their ability. They will document the connections between all units and how certain instruction sets would interact with their design i.e. branch operations. Each protocol should be laid out and formally documented and what each block unit would contain as far as pseudo logic. In addition to that include all misunderstandings and confusions for further assessment. If existing block diagrams are used for inspiration include all resources and reasoning as to why it was used. If your design is predominantly based on an existing design study that design and dissect it with all available resources found and used. Make sure to include all citations and references to all material used. Every type of instruction format should have a set flow meaning that for a specific instruction such instructions would pass through X existing units; such protocols should be communicated in both the single cycle and 5-stage. The reason for developing two block designs is to understand the single cycle to further understand how the 5 stage operates. Break everything down, document it, justify it, reference it, include misunderstandings and have it ready for the weekend meeting.

### 6.6.4 Expectations for the Team Meeting

By the deadline we will have a team meeting discussing our work through 5-10 min presentations. Your work should be included in the team OneDrive for other members to follow along as you present your work. Such work should be a report containing documentation and a block design developed using existing software. Once all work is presented we will all work on merging or choosing the most justified and full proof design

from the group and moving forward from there. All team members should agree on an existing design and have full comprehension of the protocols that such design would implement for varying instruction types. Note that the focus will primarily be on the 5-stage however all members should work on designing and breaking down a single cycle as it is the foundation of the 5 stage. The single cycle should be included in the document. The meeting will be long and could require a follow up meeting to finish selecting the team's presentable design with proper documentation.

### 6.6.5 Division of Labor for initial Block Design

For this deliverable request, everyone was expected to develop their own iterations for the sake of understanding different architectures for the RV32I ISA and to also produce iterations that were a good start for further polishing. Every member either found their own struggles in understanding computer architecture or overcame them by producing their own block design.

### 6.6.6 Discussion of Results for Initial Block Design, Second Deliverable Request

The expectation for this deliverable request was not for every member to produce 91 a block design given the very advanced nature of ISA translation to microarchitecture. Those that provided a foundation for the team's in house design and those that struggled were able to obtain existing open source designs that could be used by the team for learning and for usage. Clear authorization was requested and given in using open source microarchitecture given the nature of the project and complexity of designing our own architecture. By this point, the team decided to use a combination of in house designs and open source reference.

### 6.6.7 Overview of RTL Division of Labor

For this third deliverable request, the team is divided into 3 groups of two. Each team is partitioned a section in our in-house block design and is expected to verify each other's designs. This will make it so that there is 4-6 design verifications for every 2 designers. A ratio that complies with industry in the sense that there are people verifying designs than designing them. By this point there has also been a change in the 2-man groups originally laid out at the beginning of the project from the D&C and 1st deliverable request.

	Team CPU	Team Memory	Team Verification
Main Tasks	Execution Stage & Synchronization	D-Cache/I-Cache/Memory Stage/Write back	Register File/Instruction Fetch/Instruction Decode
Detailed Tasks	<ul style="list-style-type: none"> <li>Design and implement the Execution Unit, including the ALU</li> </ul>	<ul style="list-style-type: none"> <li>Design and implement the memory access unit to handle</li> </ul>	<ul style="list-style-type: none"> <li>Design and implement the Instruction Fetch Unit and the</li> </ul>

	<ul style="list-style-type: none"> <li>• Ensure proper synchronization among stages and memory</li> <li>• Ensure Memory Synchronization</li> <li>• Ensure Hazard handling is present as well as corresponding protocols</li> </ul>	<p>memory loads and stores</p> <ul style="list-style-type: none"> <li>• Implement the write-back unit to write results back to the REG file</li> <li>• Stretch: Design and implement the Hazard detection unit for managing data and control hazards</li> <li>• Integrate the L1 data cache into the memory access pipeline stage</li> </ul>	<p>Instruction Decode Unit</p> <ul style="list-style-type: none"> <li>• Create RTL code for the fetch instruction from cache and decoding them into control signals</li> <li>• Implement the program counter PC, branch handling, and immediate generation</li> <li>• Handle register access using the Register File</li> </ul>
Verification Tasks	<ul style="list-style-type: none"> <li>• Perform self-verification of the execution stage and register file</li> <li>• Cross verify the instruction fetch, decode, memory access, and write back stages from group 1 &amp; 2</li> </ul>	<ul style="list-style-type: none"> <li>• Verification Tasks:</li> <li>• Perform self verification of the memory access, write-back and if completed the hazard detection units</li> <li>• Cross verify the instruction fetch, decode and execution stages from</li> </ul>	<ul style="list-style-type: none"> <li>• Perform self-verification of the IFU and IDU</li> <li>• Cross verifies the execution unit, memory access, and write-back stages from Group 2 and 3</li> </ul>



		Group 1 and Group 2	
--	--	------------------------	--

### 6.6.8 Revised Timeline (Adjusted for setbacks)

Week	Tasks	Milestones/Goals
Oct 7-13	<ul style="list-style-type: none"> <li>- Team setup and task assignment</li> <li>- Finalize pipeline design and component interfaces</li> <li>- Finalize memory interactions for instruction fetch and write-back unit</li> <li>- Start RTL design for individual components</li> </ul>	Team alignment
Oct 14-20	<ul style="list-style-type: none"> <li>- Continue RTL design for instruction fetch, decode, execution</li> <li>- Initial functional verification planning</li> </ul>	Initial component-level design progress
Oct 21-27	<ul style="list-style-type: none"> <li>- Complete RTL for instruction fetch, decode, execution</li> <li>- Begin RTL design for memory access and write-back</li> <li>- Start self-verification on initial modules</li> <li>- Must have complete all System Verilog trainings</li> </ul>	Halfway through RTL design
Oct 28-Nov 3	<ul style="list-style-type: none"> <li>- Complete RTL for memory access and write-back</li> <li>- Cross-verification of completed RTL</li> <li>- Debugging and refining designs</li> </ul>	Major RTL design almost complete
Nov 4-10	<ul style="list-style-type: none"> <li>- Continuation of completion of any RTL designs</li> <li>- Assurance of proper control unit behavior</li> <li>- Debugging and refining designs</li> <li>- Must have completed all verification trainings</li> </ul>	Major RTL design complete

Nov 11-17	<ul style="list-style-type: none"> <li>- Start cross-verification of all modules</li> <li>- Refine functional verification testbenches with System Verilog/UVM</li> </ul>	Start full module verification
Nov 18-24	<ul style="list-style-type: none"> <li>- Conduct full system simulation</li> <li>- Cross-check and debug designs</li> <li>- Prepare for synthesis</li> </ul>	Full RTL verification
Nov 25- Dec 1	<ul style="list-style-type: none"> <li>- Continue verification</li> <li>- Synthesize design with Cadence Genus</li> <li>- Check for synthesis issues</li> <li>- Perform static Timing analysis (STA)</li> </ul>	Optimized and synthesized design (will most likely extend to the following week)
Dec 2-10	<ul style="list-style-type: none"> <li>- Final verification and validation</li> <li>- Prepare presentation</li> <li>- All training packs for the semester should have been completed</li> <li>- If verification and validation has not been completed make sure to make that clear and establish a plan to complete over break</li> </ul>	Final design should be ready or almost ready for phase 2

#### 6.6.9 Key Milestones and Specified Division of Labor

Key Milestones in this Phase	Division of Labor Specification:
<ul style="list-style-type: none"> <li>• Oct 7-13: Task Assignment, setup and finalizing Design specs</li> <li>• Oct 21-27: Completion of major RTL design</li> <li>• Nov 11-17: Cross verification and full system simulation</li> <li>• Nov 18-24: Synthesis and timing analysis with Cadence Genus</li> <li>• Dec 2-10: Final verification and final presentation</li> </ul>	<ul style="list-style-type: none"> <li>• Youssef (Team CPU)</li> <li>• Brandon (Team Memory)</li> <li>• Davi (Team Verification)</li> <li>• Evan (Team Verification)</li> <li>• Pablo (Team Memory)</li> <li>• Nicolas (Team CPU)</li> </ul>

--	--

## 6.7 Overview of Implementation of Agile

The team has maintained consistent communication, meeting, and sprint compliance. In addition to that, the team has been meeting with Northrop Grumman engineers bi-weekly for feedback on our sprints. Our main concerns now is getting back on schedule due to weather delays and completing the RTL phase by the specified deadlines. So far, internal discussions have raised concern with the completion of UVM verification for this semester and our goals have been reevaluated because of it. Our goal now is to complete 25% UVM verification by the end of the semester and will implement other verification methods while the team learns UVM via the Cadence training packs.

## 7 Appendix A: Citations

“Arm Cortex-M4 Processor Technical Reference Manual Revision R0p1.” Accessed September 5, 2024.

<https://developer.arm.com/documentation/100166/0001/Introduction/About-the-processor>.

“The RISC-V Instruction Set Manual Volume I - Version 20240411” Accessed September 5, 2024. <https://github.com/riscv/riscv-isa-manual/releases/download/20240411/unpriv-isa-asciidoc.pdf>

“Greenwaves GAP8 IoT Product Brief” Accessed September 5, 2024.

[https://greenwaves-technologies.com/wp-content/uploads/2021/04/Product-Brief-GAP8-V1\\_9.pdf](https://greenwaves-technologies.com/wp-content/uploads/2021/04/Product-Brief-GAP8-V1_9.pdf)

“ESP32-C3 Series Datasheet Version 1.8” Accessed September 5, 2024. [https://www.espressif.com/sites/default/files/documentation/esp32-c3\\_datasheet\\_en.pdf](https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf)

Muxup. “Commercially Available RISC-V Silicon” Accessed September 5, 2024. <https://muxup.com/2023q1/commercially-available-risc-v-silicon>

D. R. Coelho, *The VHDL Handbook*. Springer Science & Business Media, 2012.

*Digital Logic Design Using Verilog*. Accessed: Oct. 25, 2024. [Online]. Available: <https://link.springer.com/book/10.1007/978-981-16-3199-3>

*System Verilog Assertions and Functional Coverage*. Accessed: Oct. 25, 2024. [Online]. Available: <https://link.springer.com/book/10.1007/978-3-030-24737-9>

“Constraint-Based Verification | SpringerLink.” Accessed: Oct. 25, 2024. [Online]. Available: <https://link.springer.com/book/10.1007/0-387-30784-2>

“IEEE/IEC International Standard–SystemVerilog–Part 2: Universal Verification Methodology Language Reference Manual,” *IEC 62530-2:2023-10 (IEEE Std 1800.2-2020)*, pp. 1–461, Oct. 2023, doi: [10.1109/IEEESTD.2023.10287892](https://doi.org/10.1109/IEEESTD.2023.10287892).

“Intel® 64 and IA-32 Architectures software developer manuals,” Intel. Accessed: Sep. 05, 2024. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>

J. L. Hennessy, D. A. Patterson, and K. Asanović, *Computer Architecture: A Quantitative Approach*. Elsevier, 2012.

“Documentation – Arm Developer.” Accessed: Sep. 14, 2024. [Online]. Available: <https://developer.arm.com/documentation/102202/0300?lang=en>

A. E. Team, "The Official History of Arm," Arm Newsroom. Accessed: Oct. 5, 2024. [Online]. Available: <https://newsroom.arm.com/blog/arm-official-history>

Khosrow Golsha. 2025. ASIC Design Implementation Process. Springer Nature, Switzerland. (978-3-031-58655-2)

UVM (universal verification methodology) working group," Acellera. [Online]. Available: <https://www.accellera.org/activities/working-groups/uvm>

J. M. Rabaey, A. P. Chandrakasan, and B. Nikolić, Digital integrated circuits: A design perspective. Prentice Hall, 2003.

GeeksforGeeks, "Clock Skew in synchronous digital circuit systems," GeeksforGeeks, Oct. 22, 2020. [Online]. Available: <https://www.geeksforgeeks.org/clock-skew-in-synchronous-digital-circuit-systems/>

C. P. Solutions, "How to Eliminate Clock Jitter," Cadence. [Online]. Available: <https://resources.pcb.cadence.com/blog/2022-how-to-eliminate-clock-jitter>

UVM (Universal Verification Methodology) Working Group," Accellera. [Online]. Available: <https://www.accellera.org/activities/working-groups/uvm>

N. B. Harshitha, Y. G. Praveen Kumar, and M. Z. Kurian, "An Introduction to Universal Verification Methodology for the digital design of Integrated circuits (IC's): A Review," in 2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS), IEEE, Mar. 2021, pp. 1710–1713. [Online]. Available: <http://dx.doi.org/10.1109/icais50930.2021.9396034>

S. Qamar, W. H. Butt, M. W. Anwar, F. Azam, and M. Q. Khan, "A Comprehensive Investigation of Universal Verification Methodology (UVM) Standard for Design Verification," in Proceedings of the 2020 9th International Conference on Software and Computer Applications, New York, NY, USA: ACM, Feb. 2020, pp. 339–343. [Online]. Available: <http://dx.doi.org/10.1145/3384544.3384547>

B. Bailey, "What Will That Chip Cost?," Semiconductor Engineering. [Online]. Available: <https://semiengineering.com/what-will-that-chip-cost/>

David A. Patterson and John L. Hennessy. 2013. Computer Organization and Design, Fifth Edition: The Hardware/Software Interface (5th. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

"Ripes." Accessed: Oct. 25, 2024. [Online]. Available: <https://ripes.me/>

IEEE, "1800-2017 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," IEEE Xplore. [Online]. Available: <https://ieeexplore.ieee.org/document/8299595>

Jiwei Lu *et al.*, “The performance of runtime data cache prefetching in a dynamic optimization system,” in *22nd Digital Avionics Systems Conference. Proceedings (Cat. No.03CH37449)*, San Diego, CA, USA: IEEE Comput. Soc, 2003, pp. 180–190. doi: [10.1109/MICRO.2003.1253194](https://doi.org/10.1109/MICRO.2003.1253194).

IEEE, “1800.2-2020 - IEEE Standard for Universal Verification Methodology Language Reference Manual,” IEEE Xplore. [Online]. Available: <https://ieeexplore.ieee.org/document/9195920>

North Carolina State University, “FreePDK45,” NC State EDA. [Online]. Available: <https://eda.ncsu.edu/freepdk/freepdk45/>

## 8 Appendix B: AI

### ChatGPT Case Study 1:

“Verification methods for digital systems are crucial for ensuring that a design meets its intended specifications and requirements without errors. Here are several commonly used methods:

1. **Simulation:**

- a. **Functional Simulation:** This is the process of using software tools to simulate the logical behavior of the digital circuit to check for functional correctness against the specifications.
- b. **Timing Simulation:** After functional verification, timing simulation is used to verify that the circuit meets the required timing specifications such as setup and hold times, and that it operates correctly at the intended clock frequency.

2. **Formal Verification:**

- a. This method uses mathematical techniques to prove or disprove the correctness of algorithms underlying a system. It does not rely on test vectors but uses assertions and property checking to ensure the design meets specified behaviors.

3. **Emulation:**

- a. Emulation involves the use of specialized hardware to mimic the behavior of a digital circuit. This allows for real-time verification and is significantly faster than simulation for complex systems. Emulators can interact with the actual hardware environment and test the integrated system under various conditions.

4. **FPGA Prototyping:**

- a. In FPGA prototyping, the digital design is implemented on an FPGA to validate its functionality in a real-world environment. This allows designers to test the design with actual data and interfaces before final production.

5. **Static Timing Analysis (STA):**

- a. This is a method to analytically verify the timing performance of a digital circuit without needing simulation. STA checks to ensure that all timing paths in a design meet the required constraints under specified operating conditions.

6. **Linting:**

- a. Linting tools are used early in the design process to check for issues in the RTL code, such as syntax errors, semantic errors, and coding style violations. Linting

helps in improving the quality of the RTL code before it goes into deeper verification stages.

7. **Hardware-Assisted Verification:**

- a. This involves the use of specialized hardware to accelerate the verification process. Tools like hardware accelerators and transaction-level modeling can be used to speed up the simulation process for complex system-level verifications.

8. **Coverage Analysis:**

- a. Coverage analysis tools are used to measure the effectiveness of the verification. It ensures that all parts of the design have been adequately tested and helps in identifying areas that have not been covered by the test suite.

Each of these methods has its strengths and fits different stages of the design and verification process. Effective verification often requires a combination of several methods to ensure that the design is robust and performs as expected in all scenarios.” (OpenAI, 2024)

**ChatGPT Case Study 2:**

“Hardware Description Languages (HDLs) are specialized programming languages used to describe the structure, design, and operation of electronic circuits, particularly digital logic circuits. The most commonly used HDLs are Verilog, VHDL, and SystemVerilog, each with distinct features and benefits:

1. **Verilog:**

- a. **Simplicity:** Verilog's syntax and structure are similar to C, which makes it relatively easy to learn, especially for those with a background in software programming.
- b. **Speed:** It is generally faster to compile and simulate due to its simpler syntax and semantics.
- c. **Widely Used:** Popular in industry, especially in the United States, making it a standard language for many ASIC and FPGA design projects.

2. **VHDL (VHSIC Hardware Description Language):**

- a. **Strong Typing:** VHDL has a strong typing system which can prevent many types of bugs early in the design cycle by enforcing strict type checks.
- b. **Rich Syntax:** It supports a wide range of abstract data types and more complex structures, making it suitable for complex and high-reliability systems that require rigorous specifications.



- c. **Popularity:** It is particularly popular in Europe and among industries where military and aerospace applications are involved, due to its robustness and detailed design support.
3. **SystemVerilog:**
- a. **Enhanced Verilog:** It extends Verilog with significant improvements and capabilities particularly in verification through its advanced verification constructs.
  - b. **Object-Oriented Features:** Introduces object-oriented programming and other high-level programming features that facilitate more efficient and effective testbench development.
  - c. **Assertion-Based Verification:** Includes features for assertion-based verification, allowing for more robust and comprehensive testing of digital designs.
4. **SystemC:**
- a. **High-Level Modeling:** Facilitates modeling at higher levels of abstraction than traditional HDLs, including system-level modeling.
  - b. **C++ Based:** As a library of C++, it allows the use of all C++ features and object-oriented design, making it useful for complex system modeling that includes both hardware and software components.
  - c. **Flexibility:** Ideal for early conceptual and architectural exploration before detailed hardware design.
5. **MyHDL:**
- a. **Python-Based:** Uses Python, which is known for its simplicity and readability, allowing hardware designers to leverage powerful software methodologies and libraries.
  - b. **Hardware and Software Co-Design:** Facilitates the integration of hardware and software design flows due to Python's versatility.
  - c. **Testbench Reuse:** Allows designers to write testbenches in Python that can be used to simulate both the MyHDL design and other Python code, promoting reusability and integration.

Each of these languages serves different aspects of the hardware design process and offers benefits tailored to specific types of projects or industry needs. The choice of an HDL often depends on the specific requirements of the project, the design team's familiarity with the language, and the project's verification needs.” (OpenAI, 2024)

### **ChatGPT Case Study 3:**

“RISC-V (pronounced "risk-five") is an open standard instruction set architecture (ISA) based on established reduced instruction set computing (RISC) principles. It was originally developed at the University of California, Berkeley, and is now managed by the non-profit RISC-V International. Here are some key aspects of RISC-V that highlight its significance and features:

#### **Open Standard**

- **Royalty-Free:** RISC-V is available freely to anyone, allowing the design of open-source hardware without the cost associated with proprietary ISAs.
- **Open Source:** Its specifications and implementations can be freely used, modified, and distributed by anyone, fostering a collaborative and innovative environment in both academic and commercial settings.

#### **Modular Design**

- **Extensible:** RISC-V is designed to be highly modular, with a small mandatory base integer ISA, which can be extended with additional optional extensions for applications such as floating-point computations, atomic operations, and vector processing.
- **Scalability:** It supports a wide range of uses from small embedded systems to large-scale cloud servers, making it highly versatile.

#### **Compatibility and Portability**

- **Forward Compatibility:** The architecture is designed to be forward-compatible, meaning that newer implementations are designed to support older versions of the ISA.
- **Multi-platform Support:** It can be implemented in various microarchitectures and platforms, supporting everything from tiny microcontrollers to high-performance computing systems.

#### **Growing Ecosystem**

- **Broad Adoption:** RISC-V has been adopted by many organizations and projects worldwide, ranging from commercial enterprises to educational institutions.
- **Rich Ecosystem:** It benefits from a rapidly growing ecosystem of software tools, operating systems, and libraries, which support development and deployment on RISC-V hardware.

#### **Application Areas**

- **IoT and Embedded Systems:** Its small size and extensibility make it ideal for Internet of Things (IoT) devices and other embedded systems.
- **High-Performance Computing (HPC):** With support for high-throughput and multi-core designs, RISC-V is also suited for use in servers and clusters for high-performance computing tasks.

The open nature and flexible design of RISC-V are disrupting traditional models of processor architecture development, providing a standardized, low-cost, and highly adaptable solution for a wide range of computing needs.” (OpenAI, 2024)

#### **ChatGPT Case Study 4:**

The Von Neumann and Harvard architectures are two fundamental designs used in constructing computer systems. Each has unique characteristics that influence the efficiency and complexity of the computational processes they support. Here’s a breakdown of their main differences:

#### **Von Neumann Architecture**

##### **1. Unified Memory Structure:**

- In a Von Neumann architecture, both data and instructions are stored in the same memory system. There is no physical separation between storage and computational functions, which means that instructions and data share the same communication channel or bus.

##### **2. Memory Bottleneck:**

- Because both data and instructions share the same memory and bus, the system can only access one at a time, which leads to the so-called “Von Neumann bottleneck.” This bottleneck limits the throughput of the system because the processor must wait for data and instructions to be fetched sequentially.

##### **3. Simplicity in Design:**

- The architecture is relatively simple and less costly to implement, as it requires fewer bus lines and less control logic for memory access. This simplicity makes it popular for general-purpose computers and easier to manage and maintain.

#### **Harvard Architecture**

##### **1. Separate Memory Structures:**

- Unlike Von Neumann, the Harvard architecture has distinct memory storage and buses for instructions and data. This means that the CPU can access instructions and read/write data simultaneously, which can significantly speed up processing.

## 2. **Parallel Access:**

- a. The separation of data and instruction memory allows concurrent access to both, eliminating the bottleneck found in Von Neumann systems. This is particularly advantageous in applications where speed and real-time processing are critical, such as in digital signal processing (DSP).

## 3. **Increased Complexity and Cost:**

- a. The need for separate memory banks and buses for data and instructions increases the complexity of the system design. This complexity can lead to higher costs in terms of both hardware and design efforts.

### **Practical Differences and Usage**

- **Performance:** Harvard architecture can offer better performance due to the simultaneous access to instructions and data. However, the performance advantage must be weighed against the increased complexity and cost.
- **Flexibility:** Von Neumann architecture is more flexible and simpler to program because of its unified memory approach. This makes it suitable for a wide range of applications, including most general computing systems.
- **Application Specific:** Harvard architecture is often used in embedded systems and microcontrollers where specific tasks need to be performed quickly and efficiently, such as in DSPs and systems requiring high throughput for algorithm processing.

### **Modified Harvard Architecture**

- To combine the advantages of both architectures, the Modified Harvard Architecture was developed. This approach typically uses a unified memory for storage (like Von Neumann) but separate caches for instructions and data (like Harvard), allowing simultaneous access at the cache level. Many modern processors, including those in personal computers and mobile devices, use this type of architecture to optimize performance without significantly complicating the hardware design.

Each architecture serves its purpose based on the needs of the application, with Von Neumann being more prevalent in general-purpose computing and Harvard favored in specialized and performance-critical environments. (OpenAI, 2024)