



Senior Design I

Northrop Grumman Group

<https://github.com/youss2017/NG-ICDesign>

Authors

Davi Dantas
Electrical Engineering

Evan Kasky
Computer Engineering

Nicolas Sayed
Electrical Engineering

Pablo Rodriguez
Computer Engineering

Timothy Ogg
Computer Engineering

Youssef Samwel
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

1	EXECUTIVE SUMMARY	1
2	PROJECT DESCRIPTION	2
2.1	PROJECT BACKGROUND AND MOTIVATION	2
2.2	ENGINEERING SPECIFICATIONS	3
2.3	HARDWARE DIAGRAM	4
2.4	SOFTWARE DESIGN DIAGRAM	4
2.5	HOUSE OF QUALITY	5
2.6	EXISTING PRODUCTS USED TO IDENTIFY PROJECT FEATURES	5
2.6.1	<i>GreenWaves GAP8</i>	6
2.6.2	<i>Espressif ESP32-C3</i>	7
2.6.3	<i>Other Products</i>	8
2.6.4	<i>Summary</i>	9
2.7	GOALS AND OBJECTIVES	9
3	RESEARCH	10
3.1	INTRODUCTION TO TECHNOLOGIES	10
3.2	COMPUTER ARCHITECTURE	13
3.2.1	<i>Von Neumann Architecture vs. Harvard Architecture</i>	14
3.2.2	<i>Micro-architectures</i>	15
3.2.3	<i>Single Cycle</i>	15
3.2.4	<i>Multi Cycle</i>	16
3.2.5	<i>Pipelined</i>	17
3.2.6	<i>Comparison Of Architectures</i>	18
3.2.7	<i>Potential Pipeline Optimizations</i>	19
3.2.8	<i>Memory Architecture Selections</i>	20
3.2.9	<i>Option 1 – Harvard Architecture</i>	21
3.2.10	<i>Option 2 – Von Neumann Architecture</i>	21
3.3	PIPELINE HAZARDS	22
3.3.1	<i>Types of Hazards</i>	22
3.3.2	<i>In-depth Analysis</i>	23
3.3.3	<i>Structural Hazards in Depth</i>	23
3.3.4	<i>Control Hazards in Depth</i>	24
3.3.5	<i>Software and Hardware Hazard Control</i>	25
3.3.6	<i>Example of Data Hazard and Solution</i>	25
3.3.7	<i>Custom Assembler to Mitigate Control Hazards</i>	28
3.4	DYNAMIC BRANCH PREDICTION	31
3.4.1	<i>Branch Prediction Buffer</i>	31
3.4.2	<i>Correlating Branch Predictor</i>	32
3.4.3	<i>Tournament Predictor</i>	33
3.5	CACHE AND OPTIMIZATION	33
3.5.1	<i>Optimization</i>	33
3.5.2	<i>D-Cache Block Diagram</i>	35
3.6	TEAM DESIGN OPTIONS	36
3.6.1	<i>Option A – Team Designs 5 Stage</i>	37
3.6.2	<i>Option B – Team designed Single Stage</i>	41

3.6.3 Option C – 5 stage w/ Hazard and Forwarding Units	41
3.6.4 Option D – Ripes.me	42
3.7 INSTRUCTION SET ARCHITECTURES	47
3.7.1 x86	48
3.7.2 ARM	48
3.7.3 MIPS	49
3.7.4 RISC-V	49
3.8 RISC-V INSTRUCTION IMPLEMENTATION	51
3.8.1 R-Type Instructions	52
3.8.2 I-Type	54
3.8.3 B-Type	57
3.8.4 J-Type	58
3.8.5 U-Type	58
3.8.6 S-Type	59
3.9 DESIGN LANGUAGES	59
3.9.1 VHDL	60
3.9.2 Verilog	62
3.9.3 System Verilog	63
3.10 VERIFICATION METHODS AND TIMING	65
3.10.1 Constrained Random Verification	65
3.10.2 Assertion Based Verification	66
3.10.3 Coverage Driven Verification	67
3.10.4 Universal Verification Methodology	67
3.10.5 Timing Issues in Digital Circuits	68
3.10.6 UVM and Metric-Driven Verification Flow	70
3.11 DIRECT PROGRAMMING INTERFACE (DPI)	71
3.11.1 Purpose of DPI	71
3.11.2 Advantages of DPI	72
3.11.3 DPI Import and Export Syntax	72
3.11.4 Foreign Language Layer	73
3.11.5 Isolation Between Layers	74
3.11.6 Type Compatibility	74
3.12 RANDOM INSTRUCTION GENERATOR FOR UVM TESTING	74
3.12.1 Encoders	75
3.12.2 R-Type Encoder	76
3.12.3 I-Type Encoder	78
3.12.4 S-Type Encoder	80
3.12.5 B-Type Encoder	83
3.12.6 U-Type Encoder	85
3.12.7 J-Type Encoder	87
3.12.8 Example Output from the RIG	90
3.12.9 Connecting the RIG to System Verilog	90
3.12.10 Create a function for DPI	91
3.12.11 Creating a System Verilog Module	91
3.12.12 Connecting the Generator to the UVM Environment	92

3.12.13 Running and Debugging the Testbench	92
3.13 IC DESIGN FLOW AND CHALLENGES	93
4 RELATED STANDARDS AND DESIGN CONSTRAINTS	94
4.1 RELATED STANDARDS	95
4.1.1 RISC-V32I ISA	95
4.1.2 Impact of RISC-V32I ISA on Design	95
4.1.3 IEEE 1800-2017 (System Verilog)	95
4.1.4 Impact of IEEE 1800-2017 on Design	96
4.1.5 IEEE 1800.2-2020 (Universal Verification Methodology)	96
4.1.6 Impact of IEEE 1800.2-2020 (UVM) on Design	96
4.1.7 FreePDK45	97
4.1.8 Impact of FreePDK45 of Design	97
4.2 SOFTWARE CONSTRAINTS	97
4.3 MANUFACTURABILITY CONSTRAINTS	98
4.4 ECONOMIC CONSTRAINTS	98
4.5 TIME CONSTRAINTS	99
5 ARTIFICIAL INTELLIGENCE	100
5.1 CHAT GPT CASE STUDIES	100
5.1.1 Case Study 1	100
5.1.2 Case Study 2	101
5.1.3 Case Study 3	101
5.1.4 Case Study 4	102
6 HARDWARE AND SOFTWARE DESIGN	102
6.1 RAPID CORE DESIGN – PREFACE	102
6.2 SYNCHRONIZATION MODEL	103
6.3 NO OPERATION INSTRUCTION NOP	103
6.4 COMMON REQUIREMENTS	104
6.5 GENERIC CACHE	104
6.5.1 High-Level Description and Requirements	104
6.5.2 The ready-valid protocol	105
6.5.3 Microarchitecture and Finite State Machine	106
6.5.4 Performance and Potential Improvements	107
6.5.5 Cache Subsystem Verification	107
6.6 INSTRUCTION FETCH (IF) STAGE	109
6.6.1 Objective Summary	109
6.6.2 Requirements	109
6.6.3 Block Diagram	109
6.6.4 State Machine	110
6.6.5 Internal Synchronization Flow	111
6.6.6 Internal RESET Configuration	111
6.6.7 State Diagram	111
6.6.8 Design Module I/O Ports	112
6.7 INSTRUCTION DECODER (ID) STAGE	112

6.7.1 Objective Summary	112
6.7.2 4.2 Requirements	112
6.7.3 Block Diagram	116
6.8 STATE MACHINE	116
6.9 INTERNAL SYNCHRONIZATION FLOW	117
6.9.1 Internal RESET Configuration	117
6.9.2 State Diagram	118
6.9.3 Design Module I/O Ports	118
6.9.4 4.6.1 Control Signal Structure	119
6.10 EXECUTE (EX) STAGE	119
6.10.1 Objective Summary	119
6.10.2 Requirements	119
6.10.3 Block Diagram	120
6.10.4 State Machine	120
6.10.5 Internal Synchronization flow	121
6.10.6 Internal RESET Configuration	121
6.10.7 State Diagram	122
6.10.8 Design Module I/O Ports	122
6.11 MEMORY STAGE (MEM)	123
6.11.1 Requirements	123
6.11.2 Block Diagram	124
6.11.3 State Machine	125
6.11.4 Internal Synchronization flow	126
6.11.5 State Diagram	126
6.11.6 Design Module I/O Ports	126
6.12 WRITEBACK STAGE (WB)	127
6.12.1 Objective Summary	127
6.12.2 Requirements	127
6.12.3 Block Diagram	127
6.12.4 State Machine	128
6.12.5 State Diagram	129
6.12.6 Design Module I/O Ports	129
6.13 HAZARD AND FORWARDING UNIT	130
6.14 DATA HAZARD	130
6.15 CONTROL HAZARD	131
6.16 IC POWER PLANNING	132
6.17 TOOL CHAIN	133
7 CORE REDESIGN	135
7.1 RACE CONDITIONS IN RAPID	136
7.2 CONSTRAINTS AND TIME-SENSITIVE REDESIGN	137
7.3 ARCHITECTURAL IMPROVEMENTS IN RAPID-X	137
7.3.1 Rededucation of Redundant States	137
7.3.2 Reorganization of Pipeline Stages	138
7.3.3 Critical Architectural Changes	138
7.3.4 Performance Outcomes	138

7.4	INSTRUCTION FETCH UNIT	139
7.4.1	<i>High-Level Description and Requirements</i>	139
7.4.2	<i>Microarchitecture and Finite State Machine</i>	139
7.4.3	<i>Performance and Potential Improvements</i>	140
7.5	MEMORY UNIT	141
7.5.1	<i>Microarchitecture and Finite State Machine</i>	142
7.5.2	<i>Performance and Potential Improvements</i>	143
7.6	CACHE TO MAIN MEMORY COMMUNICATION	143
7.7	DECODER STAGE	144
7.7.1	<i>Objective Summary</i>	144
7.7.2	<i>Requirements</i>	144
7.8	CONTROL SIGNAL REQUIREMENT	146
7.8.1	<i>State Description</i>	149
7.9	STAGE MODULE I/O PORTS	149
7.10	LOGIC MODULE I/O PORTS	150
7.11	EXECUTE STAGE	150
7.11.1	<i>Objective Summary</i>	150
7.11.2	<i>Requirements</i>	150
7.11.3	<i>State Description</i>	150
7.11.4	<i>State Module I/O Ports</i>	151
8	SYSTEM TESTING AND EVALUATION	152
8.1	VERIFICATION SCOPE	152
8.1.1	<i>Instruction Fetch Testing</i>	153
8.1.2	<i>Verification Software UVM</i>	158
8.2	INTRODUCTION TO BENCHMARKING	166
8.3	RAPID CPI COMPUTATION	167
8.4	CPU CORE BENCHMARKS	168
8.4.1	<i>Cumulative Sum and XOR Operations</i>	168
8.4.2	<i>Justification</i>	168
8.4.3	<i>Program</i>	168
8.4.4	<i>Bit Rotation Test</i>	169
8.4.5	<i>Justification</i>	169
8.4.6	<i>Program</i>	169
8.4.7	<i>Nested Loop with Additions</i>	169
8.4.8	<i>Justification</i>	169
8.4.9	<i>Program</i>	169
8.4.10	<i>Arithmetic Progression Sum</i>	170
8.4.11	<i>Program</i>	170
8.5	DAVI'S EXPERIENCE	172
8.6	EVAN'S EXPERIENCE	175
9	ADMINISTRATIVE CONTENT	177
9.1	BUDGET AND FINANCING	177
9.2	INITIAL PHASE ONE MILESTONES	177
9.3	INITIAL PHASE TWO MILESTONES	178

9.4	WORK DISTRIBUTIONS	179
9.5	INTRODUCTION TO AGILE METHODOLOGY	180
9.6	DELIVERABLE REQUESTS	181
9.6.1	<i>Overview of First Deliverable Request</i>	181
9.6.2	<i>Results of First Deliverable Request</i>	183
9.6.3	<i>Initial Block Design Deliverable Request</i>	183
9.6.4	<i>Expectations for the Team Meeting</i>	183
9.6.5	<i>Division of Labor for initial Block Design</i>	184
9.6.6	<i>Overview of RTL Division of Labor</i>	184
9.6.7	<i>Revised Timeline (Adjusted for setbacks)</i>	186
9.6.8	<i>Key Milestones and Specified Division of Labor</i>	187
9.7	FINAL DELIVERABLE REQUEST FOR SEMESTER	188
9.8	FINALIZED WORK DISTRIBUTIONS ADJUSTED FOR REDESIGNS AND PRODUCTIVITY	190
9.9	PLAN FOR SENIOR DESIGN 2	191
9.10	OVERVIEW OF IMPLEMENTATION OF AGILE	192
CONCLUSION		193

List of Figures

FIGURE 2.1: HARDWARE DESIGN DIAGRAM	4
FIGURE 2.2: SOFTWARE DESIGN DIAGRAM	4
FIGURE 2.3: HOUSE OF QUALITY	5
FIGURE 2.4: GAP8 CLOCK AND VOLTAGE DOMAIN	7
FIGURE 2.5: GAP8 INTEGRATED CIRCUIT	7
FIGURE 2.6: ESP32 ARCHITECTURE DIAGRAM	8
FIGURE 2.7: ESP32 INTEGRATED CIRCUIT	8
FIGURE 3.1: I/O FLOW	11
FIGURE 3.2: CPU FLOW STAGES	12
FIGURE 3.3: GPU MEMORY DIAGRAM	13
FIGURE 3.4: VON NEUMANN VS HARVARD MEMORY ARCHITECTURES	14
FIGURE 3.5: HARVARD MEMORY ARCHITECTURE	21
FIGURE 3.6: VON NEUMANN ARCHITECTURE	21
FIGURE 3.7: 5-STAGE PIPELINE WITHOUT FORWARDING UNIT	26
FIGURE 3.8: DIAGRAM OF CUSTOM ASSEMBLER	29
FIGURE 3.9: GENERAL 5-STAGE PIPELINE WITH FORWARDING UNIT	31
FIGURE 3.10: BRANCH PREDICTOR STATE MACHINE	32
FIGURE 3.11: D-CACHE BLOCK DIAGRAM	35
FIGURE 3.12: INSTRUCTION FETCH UNIT PART OF 5-STAGE PIPELINE	37
FIGURE 3.13: INSTRUCTION DECODE UNIT PART OF 5-STAGE PIPELINE	37
FIGURE 3.14: REGISTER FILE FOR 5-STAGE PIPELINE	38
FIGURE 3.15: EXECUTION STAGE OF THE 5-STAGE PIPELINE	38
FIGURE 3.16: MEMORY READ/WRITE STAGE FOR 5-STAGE PIPELINE	39
FIGURE 3.17: WRITEBACK STAGE FOR 5-STAGE PIPELINE	39
FIGURE 3.18: ENTIRE 5-STAGE PIPELINE DESIGN	40
FIGURE 3.19: SINGLE STAGE PIPELINE	41
FIGURE 3.20: L1 DATA CACHE CONFIGURATION	42
FIGURE 3.21: RIPES.ME ASSEMBLY CODE EXAMPLE WITH MEMORY MAP	43
FIGURE 3.22: LED MATRIX FROM RIPES.ME	43
FIGURE 3.23: SHOWS THE FEATURES OF SWITCHES AND D-PAD	44
FIGURE 3.24: MEMORY MAP FEATURE IN RIPES.ME	44
FIGURE 3.25: SHOWS THE L1 INSTRUCTION CACHE	45
FIGURE 3.26: EXECUTION INFO	46
FIGURE 3.27: DIFFERENT AVAILABLE PROCESSORS IN RIPES.ME	46
FIGURE 3.28: DIFFERENT TYPES OF INSTRUCTIONS	50
FIGURE 3.29: RV32I BASE INSTRUCTION SET	52
FIGURE 3.30: R-TYPE INSTRUCTION BIT FIELDS	52
FIGURE 3.31: SRL EXAMPLE	54
FIGURE 3.32: RA EXAMPLE	54
FIGURE 3.33: IMMEDIATE INSTRUCTION EXAMPLE	54
FIGURE 3.34: IMM_I DECODING	55
FIGURE 3.35: IMMEDIATE SHIFT INSTRUCTION	56
FIGURE 3.36: B-TYPE INSTRUCTION FORMAT	57
FIGURE 3.37: DECODING OF BRANCHING	57

FIGURE 3.38: J-TYPE INSTRUCTION FORMAT	58
FIGURE 3.39: DECODING OF JUMP TYPE	58
FIGURE 3.40: U-TYPE INSTRUCTION FORMAT	58
FIGURE 3.41: UPPER IMMEDIATE DECODING	59
FIGURE 3.42: S-TYPE INSTRUCTION FORMAT	59
FIGURE 3.43: S-TYPE IMMEDIATE DECODE	59
FIGURE 3.44: BEHAVIORAL STRUCTURE	61
FIGURE 3.45: SYSTEMVERILOG VERIFICATION ENVIRONMENT STRUCTURE	64
FIGURE 3.46: POSITIVE CLOCK SKEW AND TIMING OFFSET BETWEEN CLK1 AND CLK2.	69
FIGURE 3.47: NEGATIVE CLOCK SKEW AND TIMING OFFSET BETWEEN CLK1 AND CLK2.	70
FIGURE 3.48: THE 6 TYPES OF RISC-V32I INSTRUCTIONS FROM THE RISC-V DOCS	75
FIGURE 3.49: RANDOM INSTRUCTIONS GENERATED BY CUSTOM GENERATOR	90
FIGURE 3.50: SAME INSTRUCTIONS FROM GENERATOR IN RIPES.ME TO SHOW VALIDITY	90
FIGURE 6.1: FSM FOR CACHE SUBSYSTEM	106
FIGURE 6.2: BLOCK DIAGRAM FOR CACHE	107
FIGURE 6.3: IF UNIT BLOCK DIAGRAM	109
FIGURE 6.4: FSM FOR IF UNIT	111
FIGURE 6.5: ID UNIT BLOCK DIAGRAM	116
FIGURE 6.6: FSM FOR THE ID	118
FIGURE 6.7: BLOCK DIAGRAM FOR EXECUTE STAGE	120
FIGURE 6.8: FSM FOR EXECUTE STAGE	122
FIGURE 6.9: BLOCK DIAGRAM FOR THE WB STAGE	127
FIGURE 6.10: FSM FOR THE WB STAGE	129
FIGURE 6.11: SCHEMATIC OF A GATED CLOCK	133
FIGURE 6.12: XCELIUM DECODER TESTBENCH RESULTS	134
FIGURE 6.13: SIMVISION FAILURE	134
FIGURE 6.14: VIVADO TESTBENCH WITH SIMULATION WAVEFORM	135
FIGURE 7.1: FSM FOR IF UNIT	140
FIGURE 7.2: BLOCK DIAGRAM FOR IF UNIT	140
FIGURE 7.3: BLOCK DIAGRAM FOR THE MEMORY UNIT	143
FIGURE 7.4: OVERVIEW OF PROCESSOR UNIT	144
FIGURE 7.5: I-TYPE IMMEDIATE FORMAT	145
FIGURE 7.6: SHAMT-I FORMAT	145
FIGURE 7.7: B-TYPE IMMEDIATE FORMAT	145
FIGURE 7.8: J-TYPE IMMEDIATE FORMAT	145
FIGURE 7.9: U-TYPE IMMEDIATE FORMAT	146
FIGURE 7.10: S-TYPE IMMEDIATE FORMAT	146
FIGURE 9.1: LISTS SOME OF THE INSTRUCTIONS FROM RV32I	153
FIGURE 9.2: WHOLE WAVEFORM FOR THE INSTRUCTION FETCH	156
FIGURE 9.3: WAVEFORM AT THE START OF THE FETCHING STATE	156
FIGURE 9.4: SHOWS THE HOME SCREEN FROM EDA PLAYGROUND WEBSITE.	158
FIGURE 9.5: SHOWS THE AVAILABLE TOOLS AND SIMULATORS AVAILABLE IN EDA PLAYGROUND.	159
FIGURE 9.6: XCELIUM COMPILER RUN OPTIONS	159
FIGURE 9.7: EXAMPLE WAVEFORM FROM EDA PLAYGROUND	160
FIGURE 9.8: EXAMPLE OF AN OUTPUT USING EDA PLAYGROUND	160

FIGURE 9.9: EXAMPLE OF OUTPUT RUNNING A BASH FILE	161
FIGURE 9.10: FLOW NAVIGATOR FOR THIRD-PARTY NETLIST PROJECT	162
FIGURE 9.11: SYNTHESIS SECTION IN THE FLOW NAVIGATOR	163
FIGURE 9.12: PACKAGE IO PINS	164
FIGURE 9.13: TCL CONSOLE IN VIVADO.	164
FIGURE 9.14: IP CATALOG	165
FIGURE 9.15: I/O PORTS WINDOW	166
FIGURE 9.16: MESSAGE WINDOW IN VIVADO.	166
FIGURE 9.17: CPI COMPARISON	170
FIGURE 9.18: IPC UPLIFT PERCENT	171
FIGURE 9.19: PIPELINE LATENCY	171
FIGURE 9.20: 100 MIPS CLOCK FREQUENCY.	172
FIGURE 9.21: SHOWS THE COMMON FILES USED FOR UVM TESTING	173
FIGURE 9.22: GIVES THE PICTURE OF HOW EXPECTED VALUES WERE CALCULATED IN THE SCOREBOARD	174
FIGURE 9.23: SIMPLE DELAY ASSERTION.	175

List of Tables

TABLE 1: DESIGN REQUIREMENTS AND SPECS	3
TABLE 2: COMPARISON OF DIFFERENT RISC-V PROCESSORS	8
TABLE 3: VON NEUMANN VS HARVARD VS MODIFIED HARVARD	15
TABLE 4: SINGLE-CYCLE VS MULTI-CYCLE VS PIPELINED ARCHITECTURES	18
TABLE 5: EXAMPLE INSTRUCTIONS FOR DATA HAZARDS	25
TABLE 6: TABLE 3 5: INSTRUCTION RUNNING IN PIPELINE WITHOUT FORWARDING UNIT	26
TABLE 7: EXPECTED VS ACTUAL	26
TABLE 8: EXAMPLE WITH NOPs TO FIX INCORRECT RESULTS	27
TABLE 9: INSTRUCTION RUNNING IN PIPELINE WITH NOPs	27
TABLE 10: TABLE 3 9: TECHNIQUES THAT AFFECT CACHE PERFORMANCE	34
TABLE 11: DESIGN OPTIONS OVERVIEW	36
TABLE 12: OR TRUTH TABLE	53
TABLE 13: XOR TRUTH TABLE	53
TABLE 14: AND TRUTH TABLE	54
TABLE 15: FORMULAS USED TO MASK I-TYPE FIELDS	78
TABLE 16: FORMULAS FOR I-TYPE ENCODER SHIFTS	78
TABLE 17: FORMULA FOR ENCODED I-TYPE INSTRUCTION	79
TABLE 18: FORMULAS TO SPLIT THE IMMEDIATE VALUE FOR S-TYPE INSTRUCTIONS	80
TABLE 19: FORMULAS FOR MASKING S-TYPE FIELDS	80
TABLE 20: FORMULAS FOR SHIFTING FIELDS IN S-TYPE ENCODING	81
TABLE 21: FORMULA TO COMBINE ALL S-TYPE FIELDS INTO A VALID INSTRUCTION	81
TABLE 22: MASKING FORMULAS FOR B-TYPE INSTRUCTIONS	83
TABLE 23: SHIFT FORMULAS FOR ENCODING B-TYPE INSTRUCTIONS	84
TABLE 24: FORMULA TO ENCODE A B-TYPE INSTRUCTION	84
TABLE 25: FORMULAS FOR MASKING U-TYPE INSTRUCTIONS	86
TABLE 26: FORMULAS FOR SHIFTING FIELDS FOR A U-TYPE ENCODING	86
TABLE 27: FORMULA FOR ENCODED U-TYPE INSTRUCTION	86
TABLE 28: ENCODING FORMULAS FOR MASKING J-TYPE INSTRUCTIONS	88
TABLE 29: ENCODING FORMULAS FOR SHIFTING J-TYPE FIELDS	88
TABLE 30: FORMULA FOR ENCODING A J-TYPE INSTRUCTION	89
TABLE 31: TRUTH TABLE FOR READ-VALID PROTOCOL	105
TABLE 32: PARTIAL TRUTH TABLE FOR READY-VALID	105
TABLE 33: STATE DESCRIPTION TABLE	110
TABLE 34: STATE TRANSITION TABLE	110
TABLE 35: RESET CONFIGURATION	111
TABLE 36: INSTRUCTION FETCH MODULE I/O PORTS	112
TABLE 37: INSTRUCTION MAPPING TO CONTROL CATEGORY	113
TABLE 38: INSTRUCTION MAPPING TO CONTROL SIGNALS	113
TABLE 39: STATE DESCRIPTION TABLE	116
TABLE 40: STATE TRANSITION TABLE	117
TABLE 41: INTERNAL RESET CONFIGURATION	117
TABLE 42: INSTRUCTION DECODER MODULE I/O PORTS	118
TABLE 43: STATE DESCRIPTION TABLE	120
TABLE 44: STATE TRANSITION TABLE FOR EXECUTE STAGE	121

TABLE 45: INTERNAL REST CONFIG FOR EXECUTE STAGE	121
TABLE 46: EXECUTE MODULE I/O PORTS	122
TABLE 47: BLOCK DIAGRAM FOR MEM STAGE	124
TABLE 48: STATE DESCRIPTION TABLE FOR MEM STAGE	125
TABLE 49: STATE TRANSITION TABLE	125
TABLE 50: FSM FOR MEM STAGE	126
TABLE 51: MEMORY MODULE I/O PORTS FOR MEM STAGE	126
TABLE 52: STATE DESCRIPTION TABLE FOR WB STAGE	128
TABLE 53: STATE TRANSITION TABLE	128
TABLE 54: WRITE BACK MODULE I/O PORTS FOR WB STAGE	129
TABLE 55: INSTRUCTION DECOMPOSITION	146
TABLE 56: CONTROL SIGNALS	147
TABLE 57: STAGE MODULE I/O PORTS	149
TABLE 58: LOGIC MODULE I/O PORTS	150
TABLE 59: 7.11.4 STATE MODULE I/O PORTS FOR EXECUTE	151
TABLE 60: LOGIC MODULE I/O PORTS	151
TABLE 61: GROUP 1 MEMBERS AND RESPONSIBILITIES	179
TABLE 62: GROUP 2 MEMBERS AND RESPONSIBILITIES.	179
TABLE 63: GROUP 3 MEMBERS AND RESPONSIBILITIES.	180
TABLE 64: DETAILED TASKS FOR TEAMS	184
TABLE 65: REVISED TIMELINE	186
TABLE 66: KEY MILESTONES	187
TABLE 67: GROUP 1 UPDATED WORK DISTRIBUTION	190
TABLE 68: GROUP 2 UPDATED WORK DISTRIBUTION	190
TABLE 69: GROUP 3 UPDATED WORK DISTRIBUTION	190

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 RISCV32I 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 RISCV32I Instruction Set Architecture (ISA), which defines the interface between software and hardware. The RISCV32I 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 RISCV32I 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 1: Design Requirements and Specs

No.	Requirement	Specification	Description	Priority
1	Processing Speed	$\geq 100 \text{ MIPS}$	The design should process at least 100 MIPS.	H
2	RV32I Compatibility	$\geq 38 \text{ Instr.}$	Handle at least 37 unique instructions, excluding memory ordering and environment calls.	H
3	Low Power Consumption	$\leq 1\text{W}$	Efficient design meeting specs under 1W power.	H
4	Cache Size	$\geq 16 \text{ MB}$	Incorporate an L1 cache of at least 16 MB.	H
5	Process Node	$\leq 45\text{nm}$	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

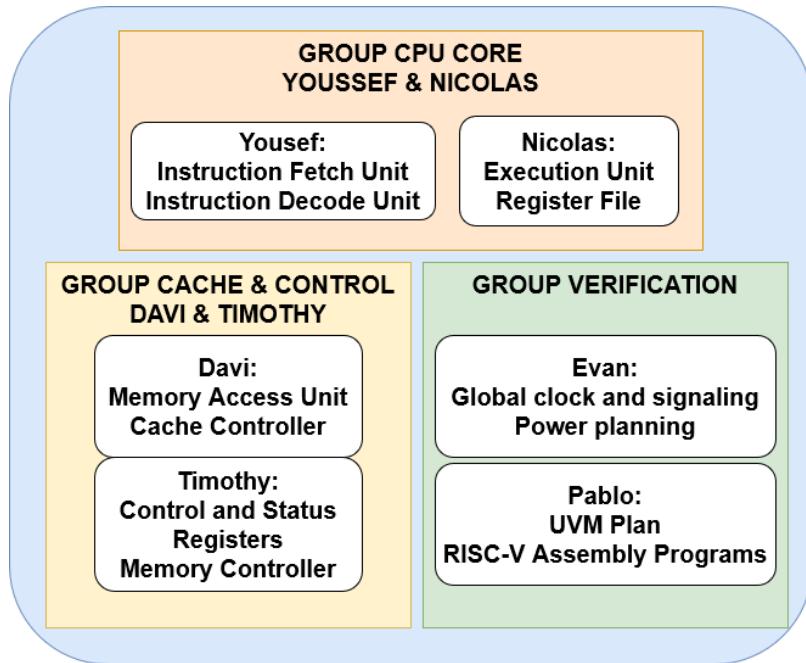


Figure 2.1: Hardware Design Diagram

2.4 Software Design Diagram

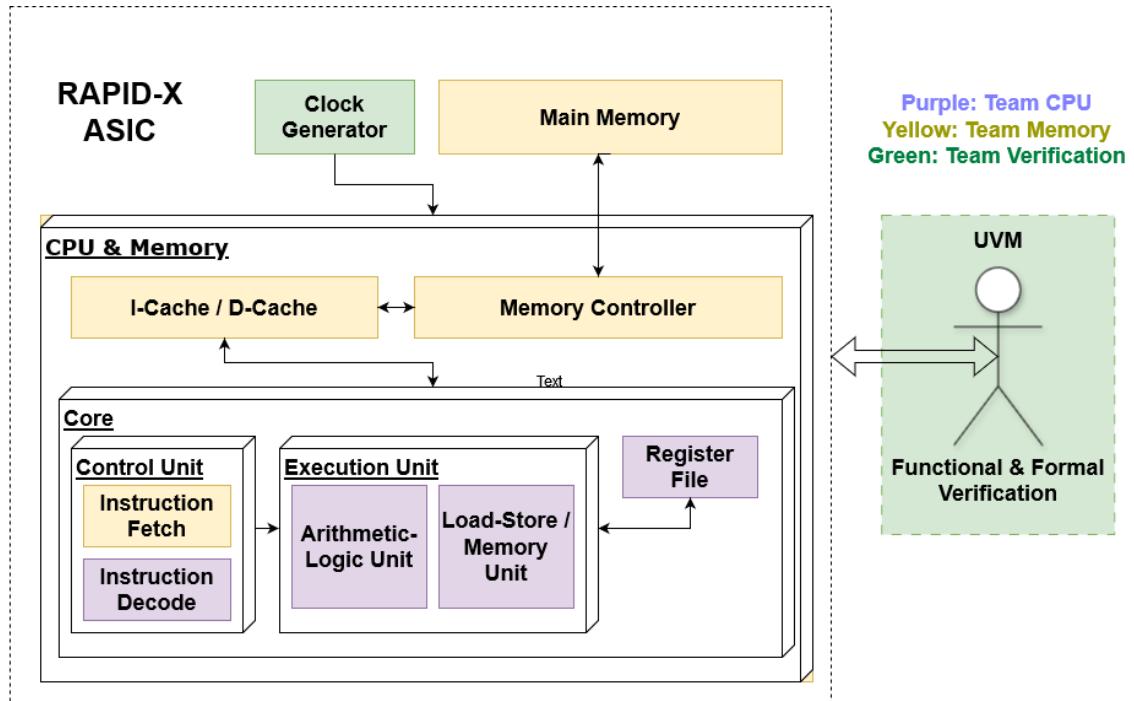


Figure 2.2: Software Design Diagram

2.5 House of Quality

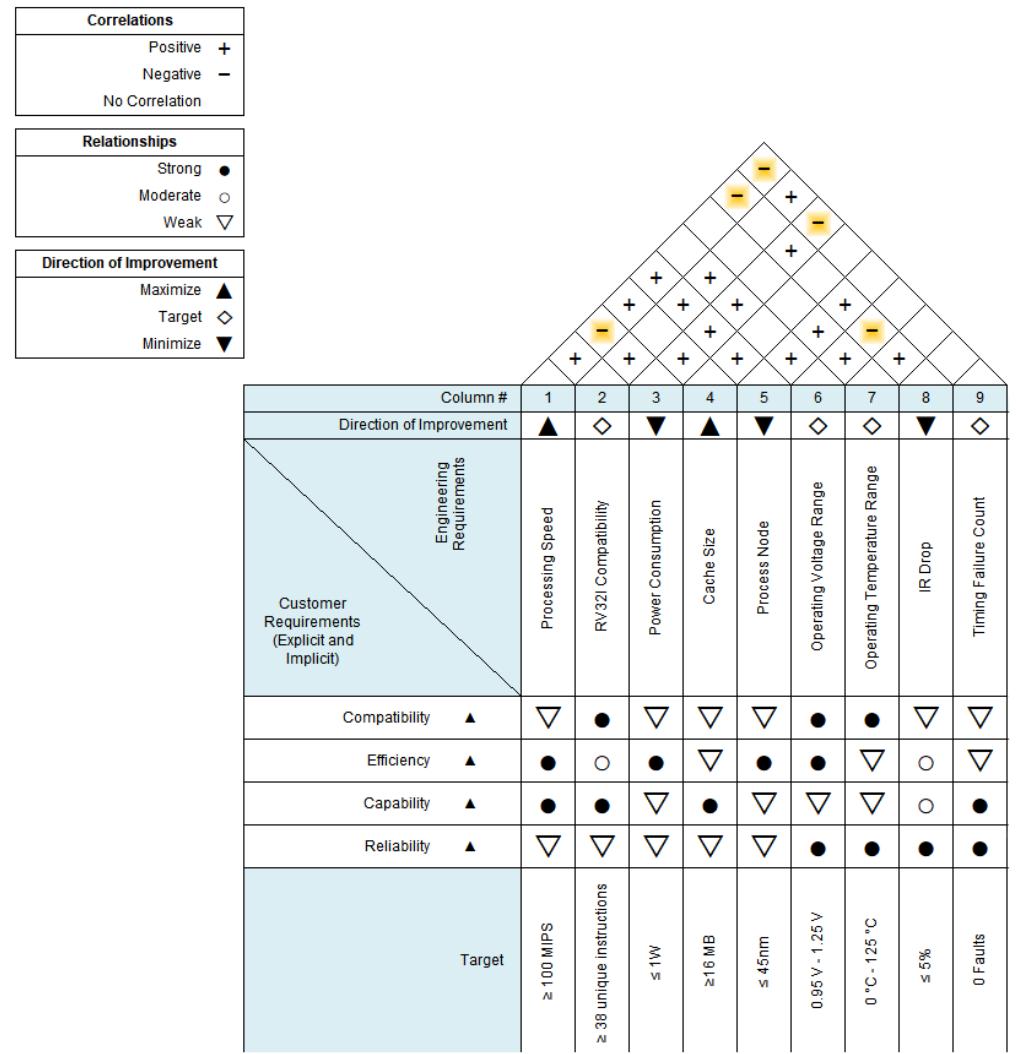


Figure 2.3: House of Quality

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:

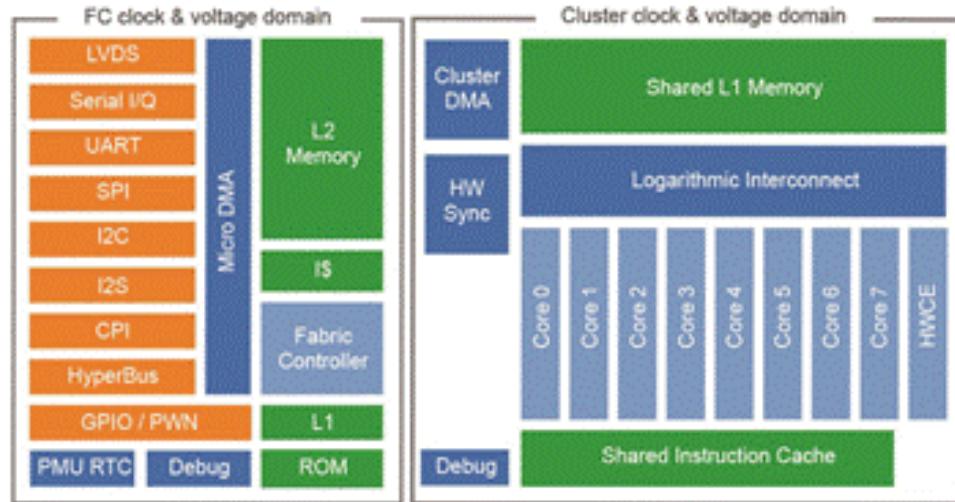
- A - Atomic Instruction
- M- Integer Multiplication and Division,
- F - Single-Precision Floating Point
- D - Double-Precision Floating Point
- Q - QuadPrecision Floating-Point
- C - Standard Extension for compressed Instructions.

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.

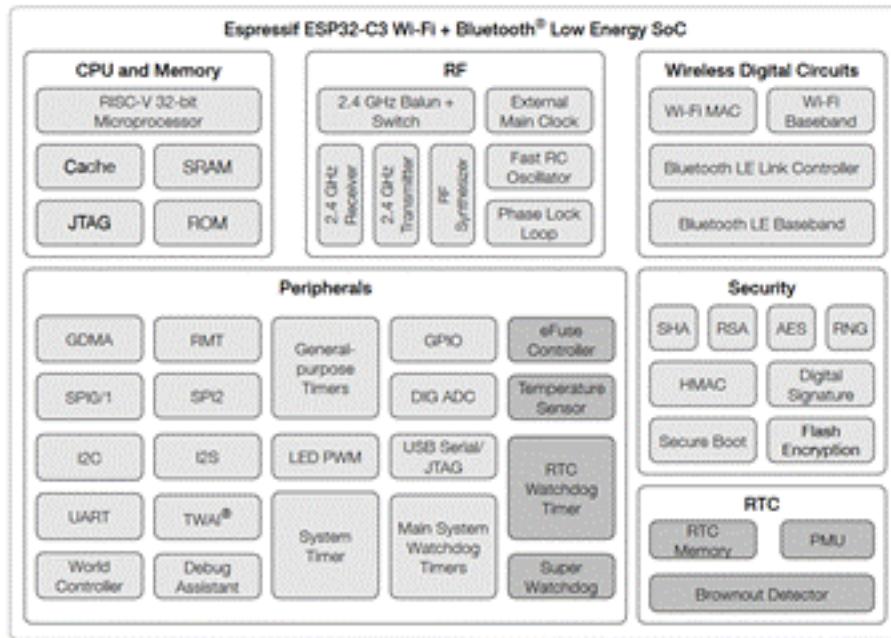


Figure 2.6: ESP32 Architecture Diagram



Figure 2.7: ESP32 Integrated Circuit

2.6.3 Other Products

Table 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—incurred 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

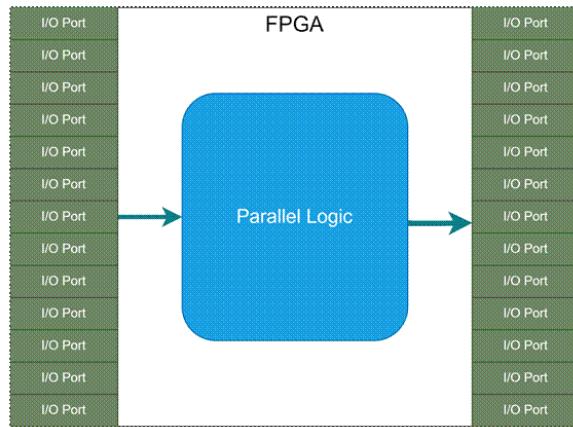


Figure 3.1: I/O Flow

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

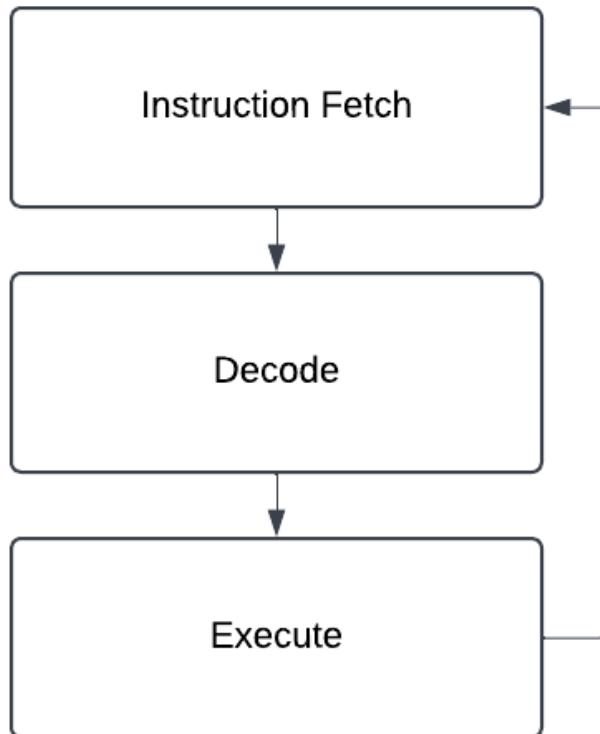


Figure 3.2: CPU Flow Stages

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.

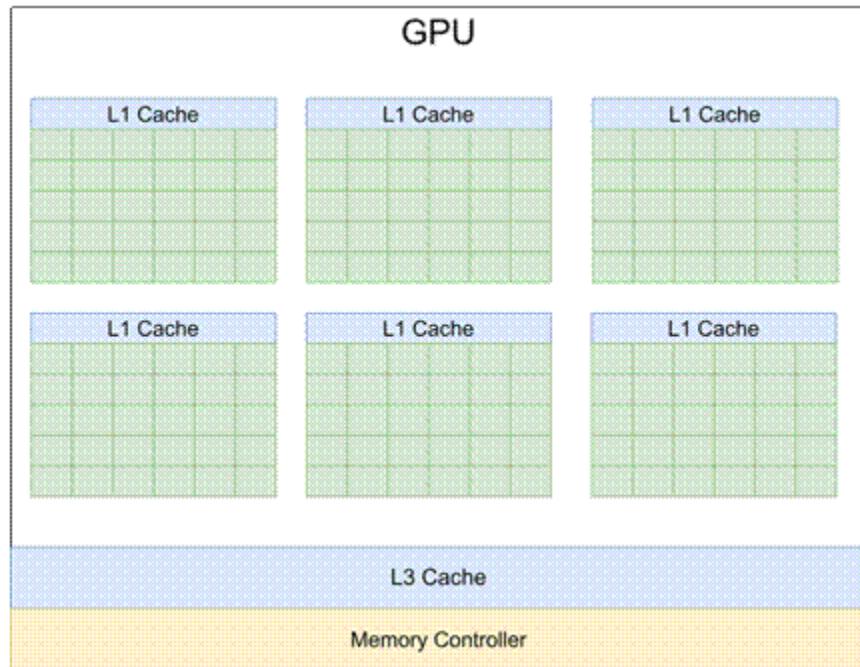


Figure 3.3: GPU Memory Diagram

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.

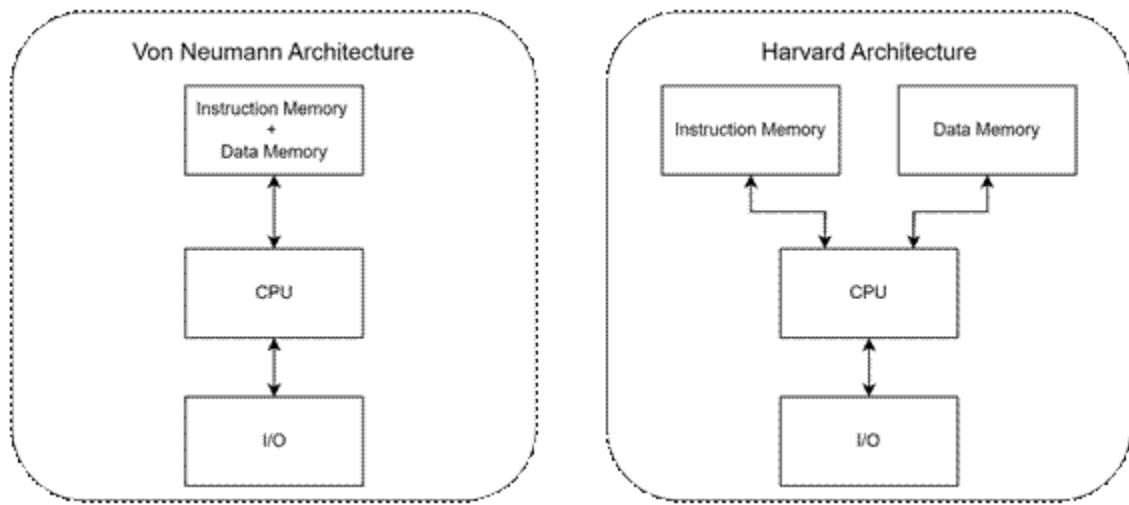


Figure 3.4: Von Neumann vs Harvard Memory Architectures

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: Von Neumann vs Harvard vs Modified Harvard

Category	Von Neumann	Harvard	Modified Harvard
Memory Structure	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
Bus Structure	Single Bus	Dual bus	Dual Bus & Shared Bus
Simplicity	Simpler hardware	Simpler programming	Middle ground
Cost	Simpler leads to a lower cost	Additional hardware leads to higher cost	Middle ground
Performance	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.

3.2.3 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.

3.2.4 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.²³ 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 RSIC 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.

3.2.5 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 throughout 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.

3.2.6 Comparison Of Architectures

Table 4: Single-Cycle vs multi-cycle vs Pipelined Architectures

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

3.2.7 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 (Alpha * X plus 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. }
```

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

```
. # 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)
```

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.

3.2.8 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.

3.2.9 Option 1 – Harvard Architecture

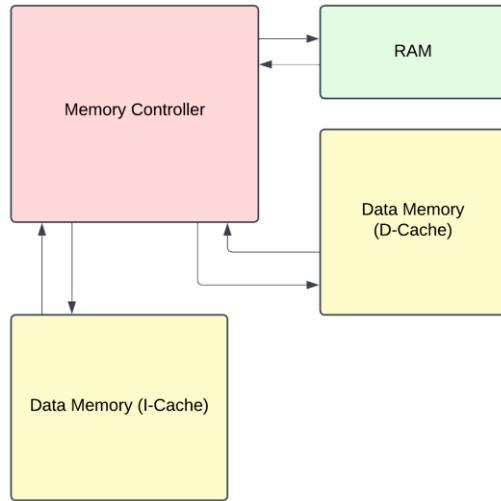


Figure 3.5: Harvard Memory Architecture

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.

3.2.10 Option 2 – Von Neumann Architecture

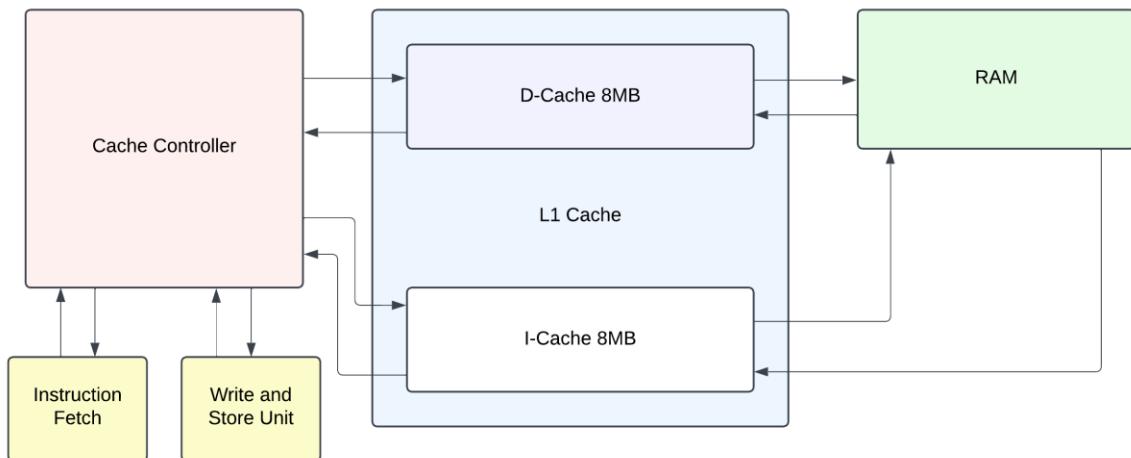


Figure 3.6: Von Neumann Architecture

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.

3.3.3 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.

3.3.4 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.

3.3.5 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.

3.3.6 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 5: 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:

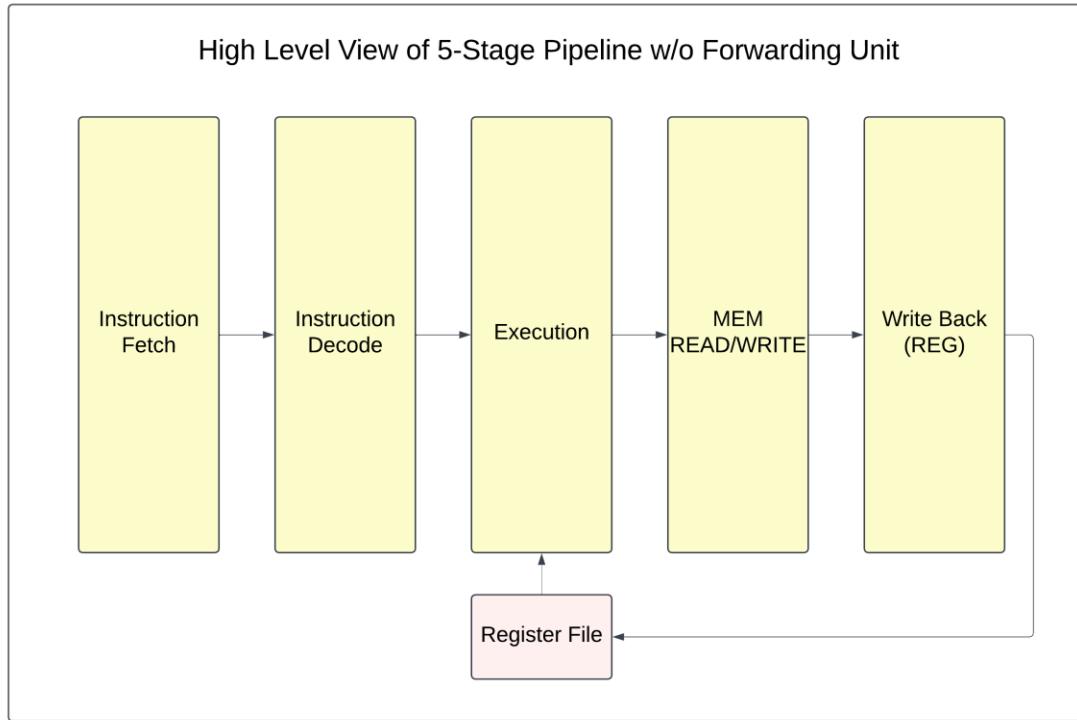


Figure 3.7: 5-Stage Pipeline without Forwarding Unit

The following table shows the process of execution:

Table 6: 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 7: 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 8: 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 9: 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.

3.3.7 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:

- Detecting Pipeline Hazards:** The assembler identifies potential hazards that could disrupt the flow of execution.
- Injecting NOPs:** It inserts the correct number of NOP instructions to mitigate identified hazards.
- Converting Assembly Code to Binary:** The assembler translates the assembly instructions into machine-readable binary format.
- 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.

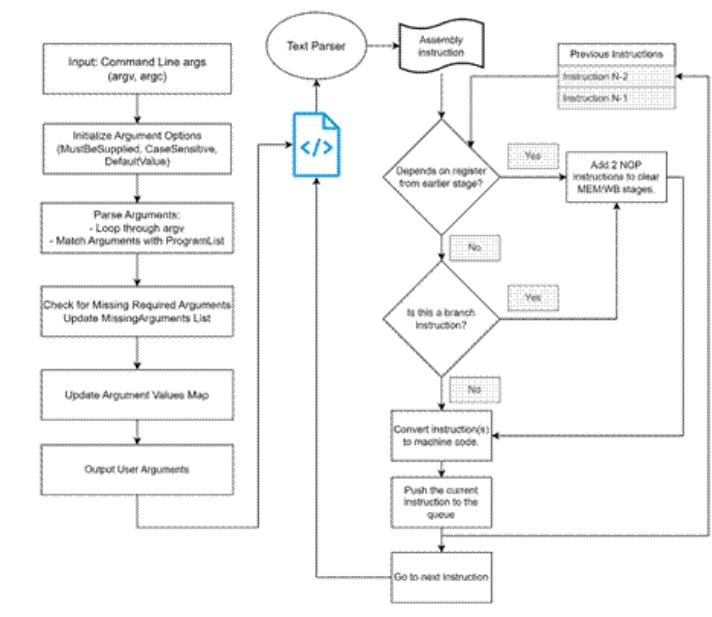


Figure 3.8: Diagram of Custom Assembler

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 MakeRTypelInstruction, MakeITypelInstruction, 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.

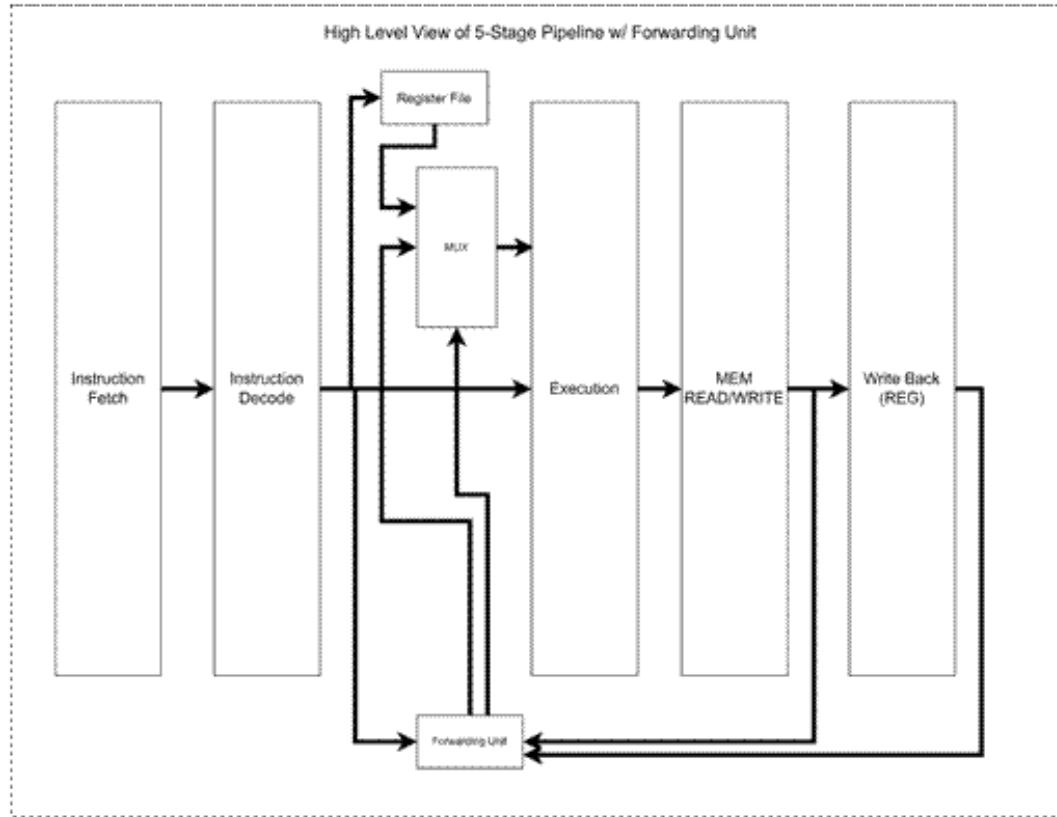


Figure 3.9: General 5-Stage Pipeline with Forwarding Unit

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.

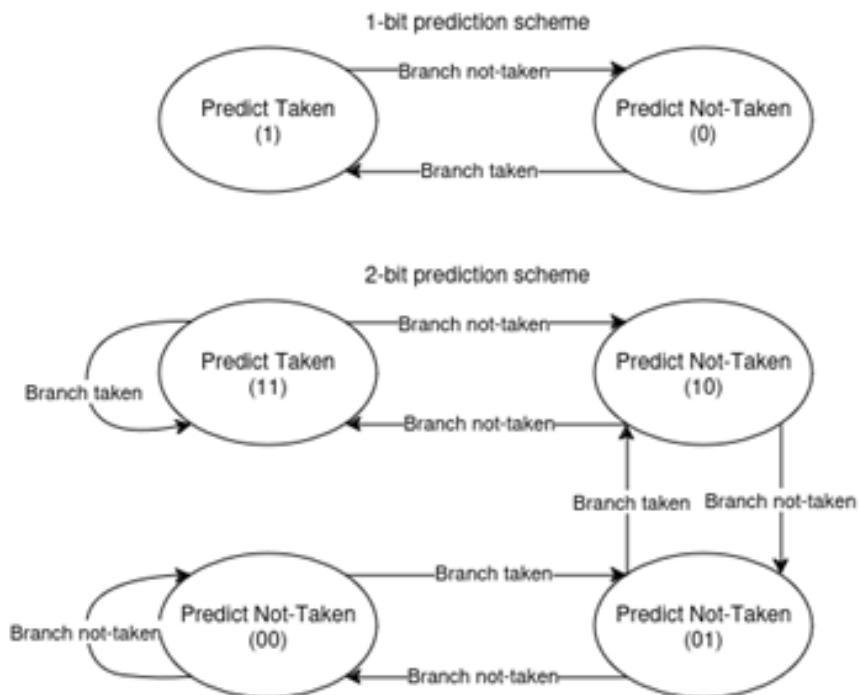


Figure 3.10: Branch Predictor State Machine

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 cache 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 10: Table 3 9: Techniques that Affect Cache Performance

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

3.5.2 D-Cache Block Diagram

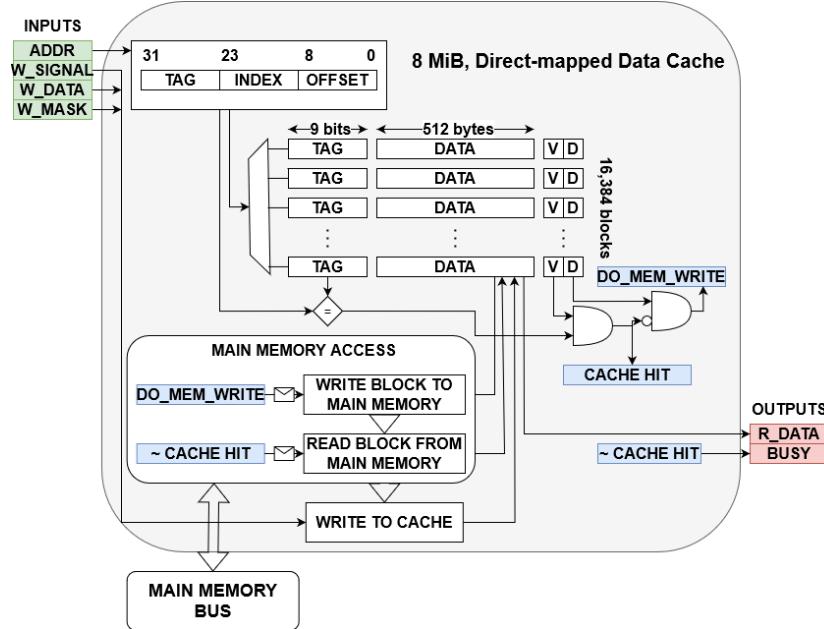


Figure 3.11: D-Cache Block Diagram

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.

- The primary organization is in memory addresses.

3.6 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 11: Design Options Overview

Options	Tradeoffs
Option A - 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.
Option 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.
Option C - 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.
Option D - 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.

3.6.1 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.

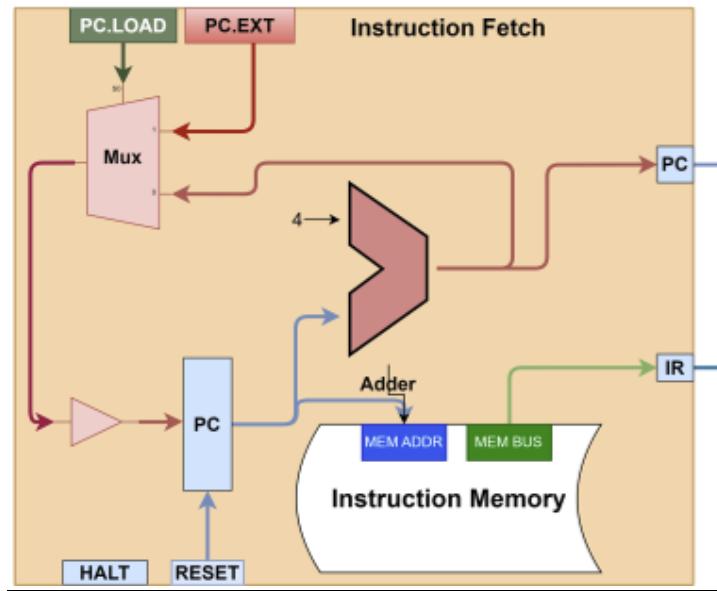


Figure 3.12: Instruction Fetch Unit Part of 5-stage Pipeline

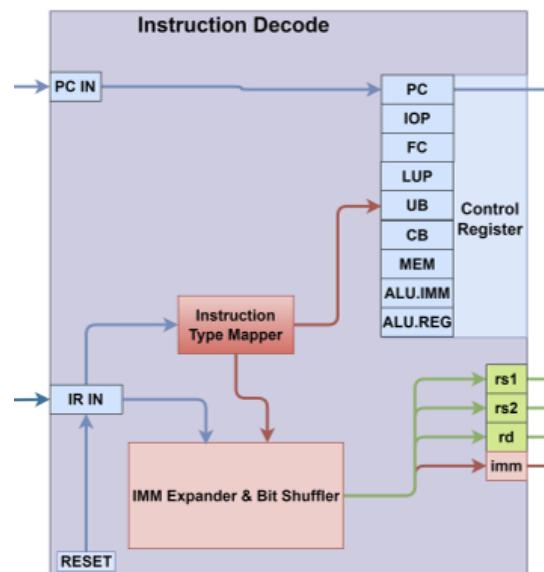


Figure 3.13: Instruction decode Unit Part of 5-stage Pipeline

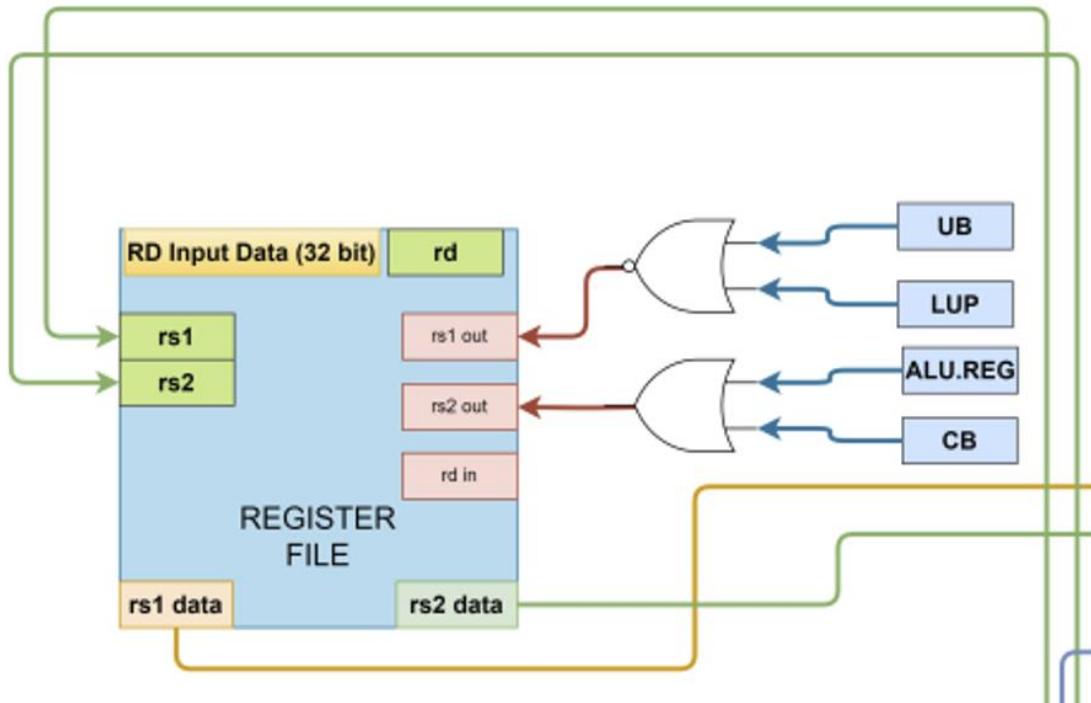


Figure 3.14: Register File for 5-stage Pipeline

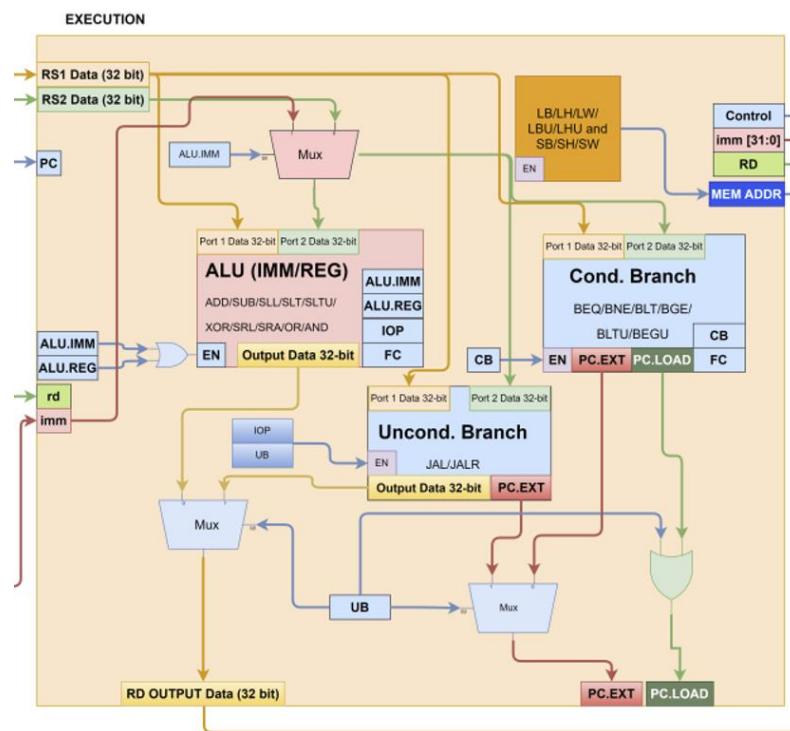


Figure 3.15: Execution Stage of the 5-stage Pipeline

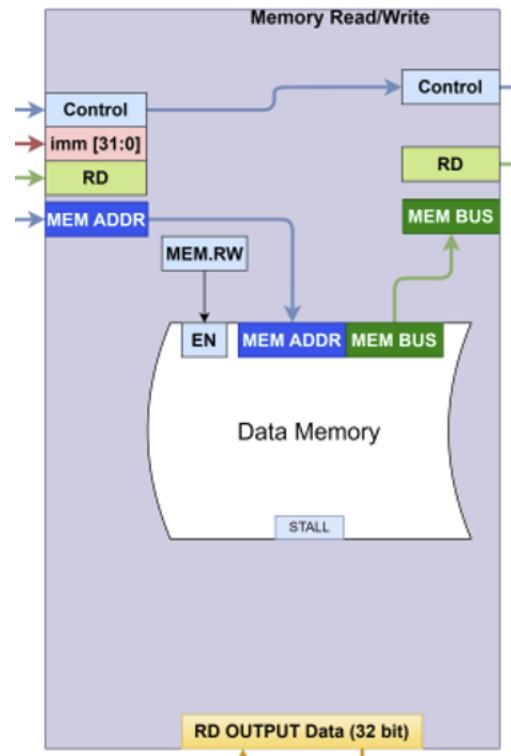


Figure 3.16: Memory Read/Write Stage for 5-stage Pipeline



Figure 3.17: Writeback Stage for 5-stage Pipeline

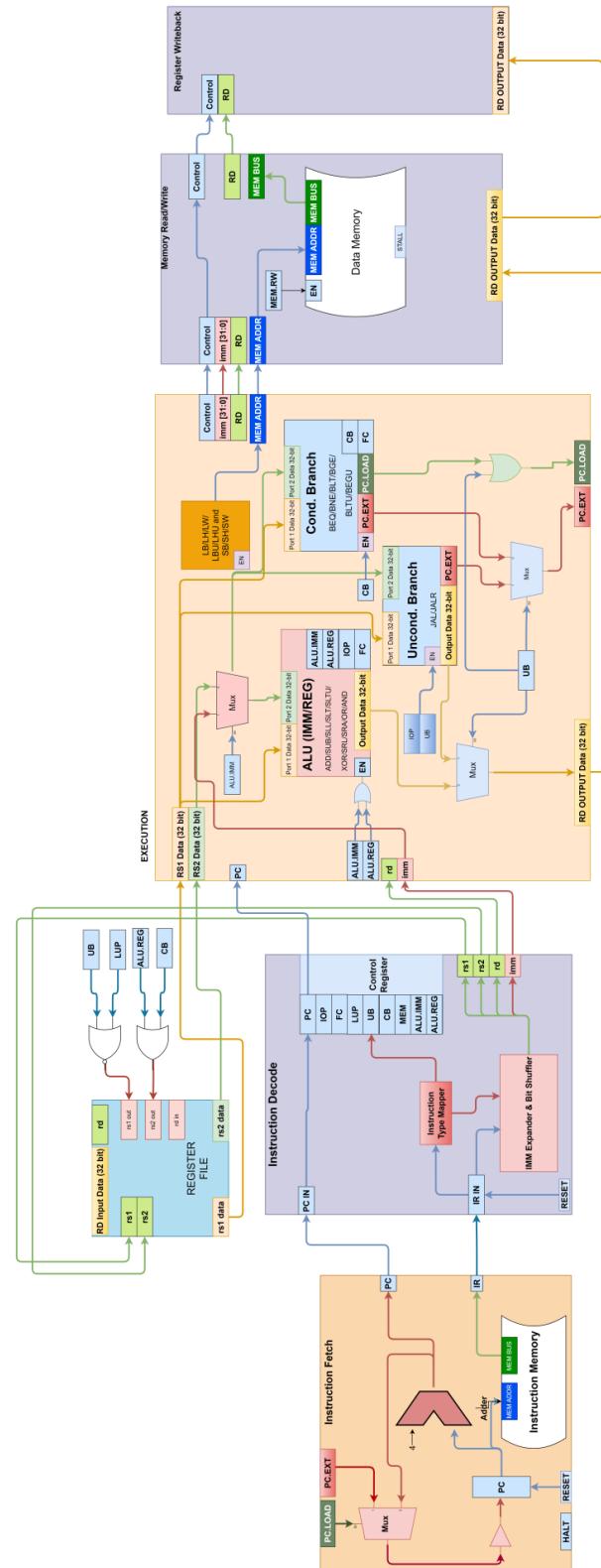


Figure 3.18: Entire 5-stage Pipeline Design

3.6.2 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.

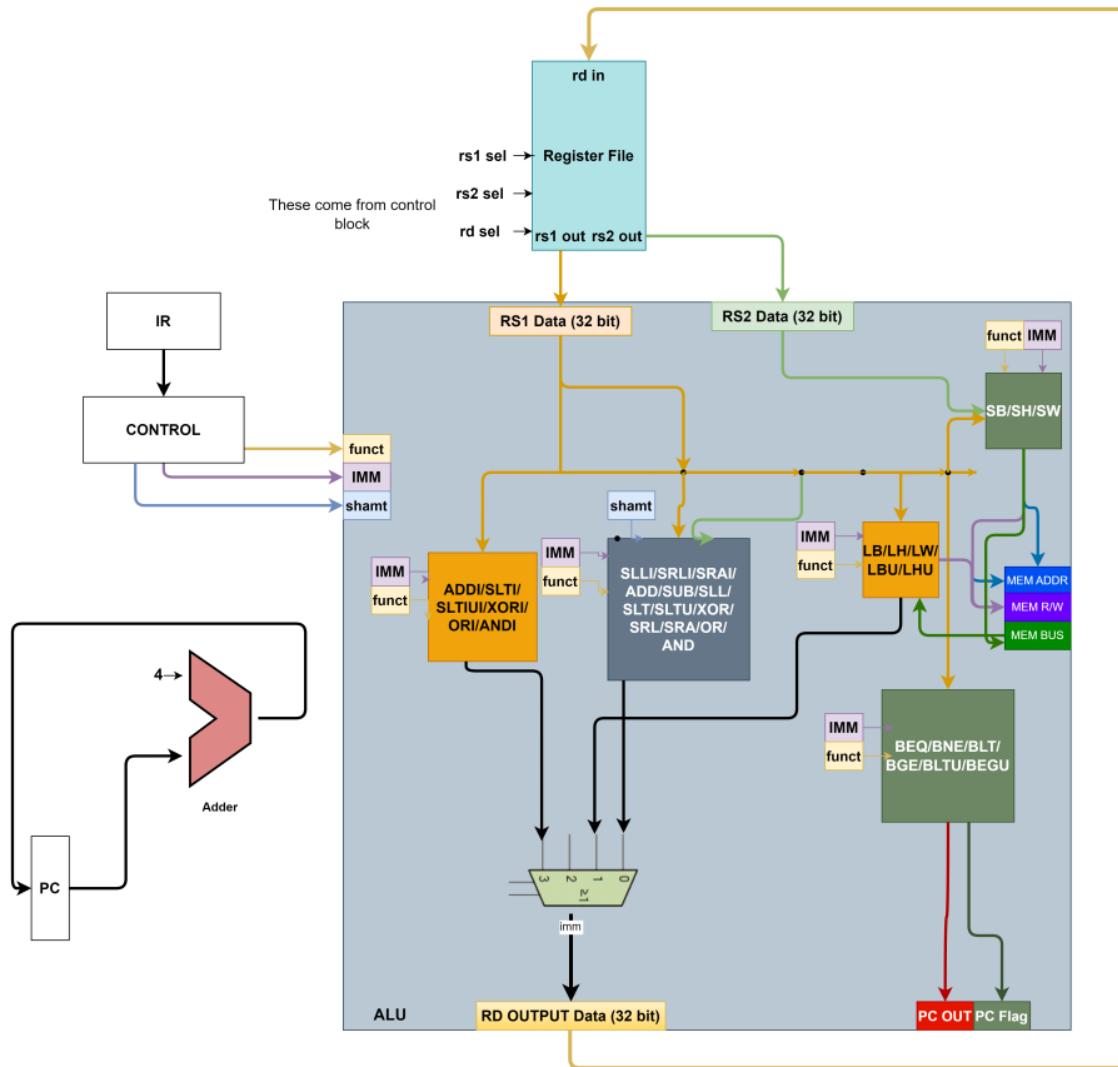


Figure 3.19: Single Stage Pipeline

3.6.3 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 add 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.

3.6.4 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:

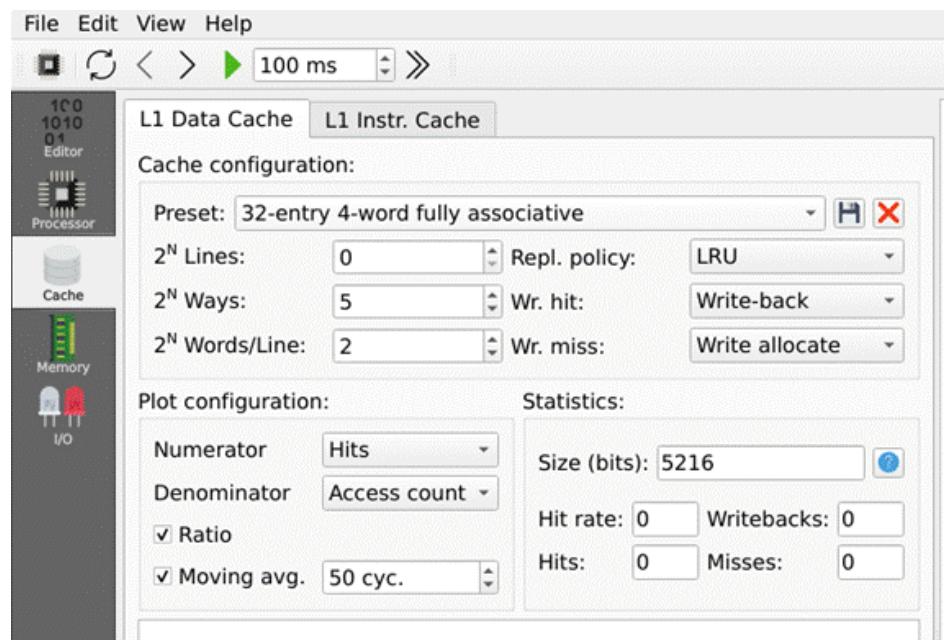


Figure 3.20: L1 Data Cache configuration

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 3.21: Ripes.me Assembly code example with memory map

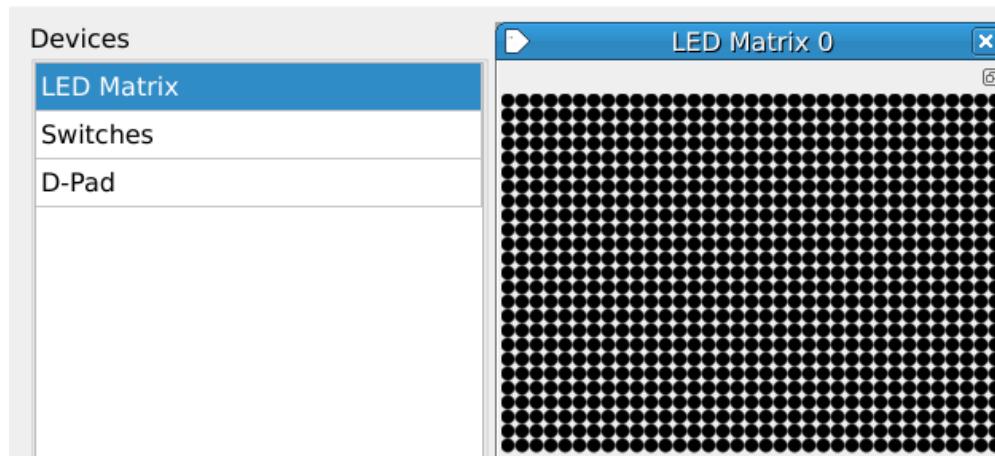


Figure 3.22: LED Matrix from ripes.me

Another nice feature from ripes.me is the LED Matrix components in the simulator. This is a useful for emulating hardware peripherals and they can be controlled straight from the assembly code.

This feature is useful for low-level interactions between software and hardware. Just like the FPGA board BASYS-3 that is used in digital systems. The LEDs were mostly used to show the simple interactions of inputs and logic gates. Visualizing logic gates are a great way to see how the different inputs will affect the output.



Figure 3.23: shows the features of switches and D-Pad

Another peripheral feature that Ripes.me provides for its users are the switches and D-Pad. Just like BASYS 3 FPGA development board that it has switches for most of the LEDs and D-Pad. In Digital systems class, this was mostly used by different states in a Finite State Machine to be achieved via the arrows in the D-Pad.

The designer is able to write in assembly code for the Finite State Machine to go to different states depending on which arrows that it goes, and it may stay in the same state if it is an invalid result. The switches emulate an array that can toggle different thing in the environment such as the LEDs or represent different states.

Memory map		
Name	Size	Range
.text	180	0x00000000 - 0x000000b3
.data	0	0x10000000 - 0x0fffffff
.bss	0	0x11000000 - 0x10ffffff
LED Matrix 0	3500	0xf0000000 - 0xf0000dab
Switches 0	4	0xf0000dac - 0xf0000daf
D-Pad 0	16	0xf0000db0 - 0xf0000dbf

Figure 3.24: memory map feature in Rippes.me

The memory map in the simulator is feature displayed in ripes.me and it provides a good visual outlook for how the different regions of memory are allocated for specific needs of the system. We are able to see how the memory maps for both assembly level language and also the peripherals.

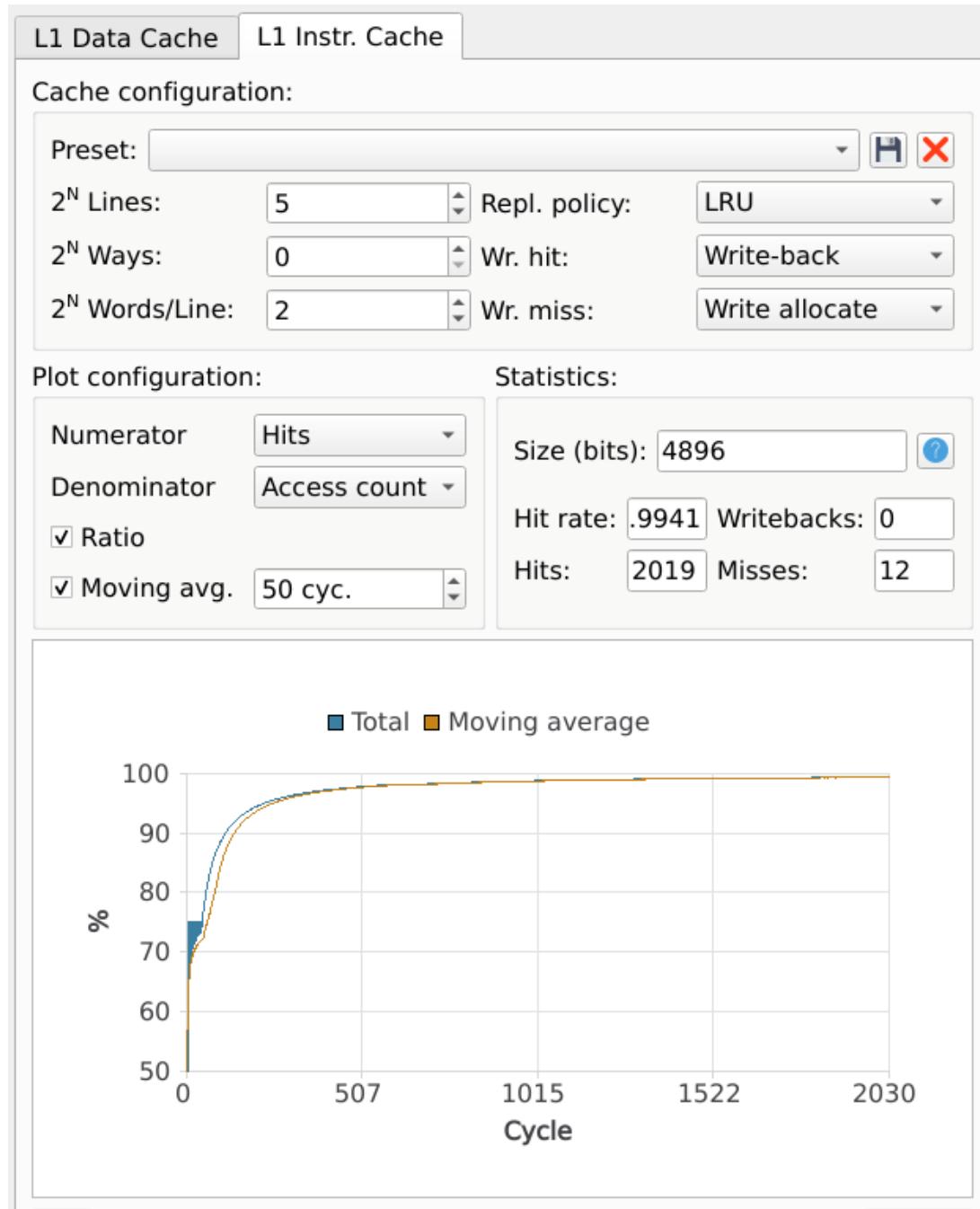


Figure 3.25: shows the L1 Instruction Cache

The L1 Instruction Cache in ripes.me is very useful to provide an in-depth view of how these caching mechanisms behave in the processor architecture. We are also able to see how the L1 Instruction cache can be customized and a lot of it is the same as the L1 Data Cache such as 2^N Lines, 2^N Ways. We are also able to customize the Repl. Policy, write hit, Write miss.

The graph is also a good visual representation to see the hit rate, hits and misses to see how different assembly code will perform differently, depending on the architecture. So if we are trying to increase our hit rate, then we may look into increasing the number of cache lines, increase cache Associativity, modify the replacement policy and adjust cache line size.

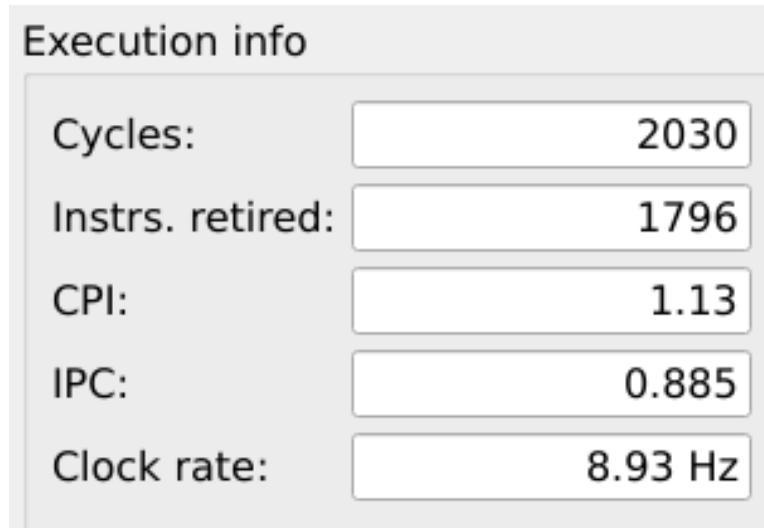


Figure 3.26: Execution Info

Ripes.me also provides this good summary of what the code is doing. These are very good statistical numbers since Northrop Grumman have given us these constraints, so it is necessary to keep an eye out for the efficiency of the program.

An example to how this feature was relevant to our research is that we were able to see our CPI and how inefficient our program ran in the architecture. We were able to observe how we had some architecture flaws such as cache misses, pipeline stalls that required the design team to perform a re-design of the architecture to accommodate these issues and they obtained excellent results as discussed in the appropriate chapter.

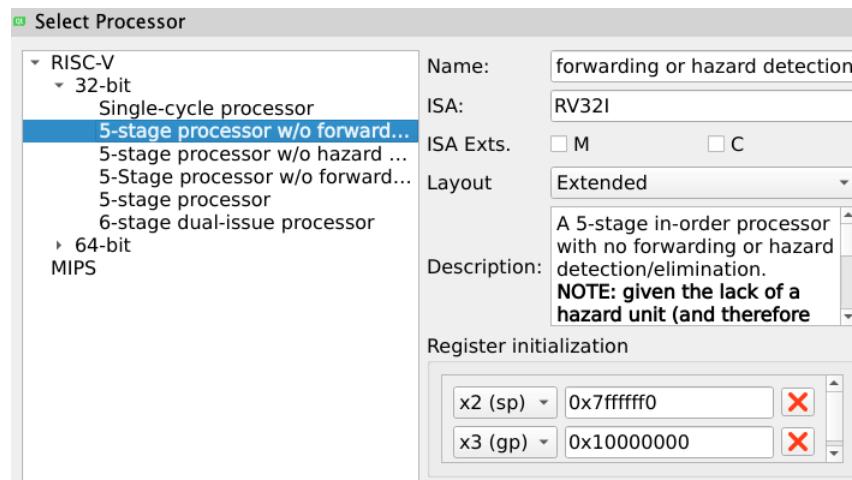


Figure 3.27: Different available processors in Ripes.me

Another very neat feature in ripes.me is how it allows for the designer to choose between many different processors. As talked before in the chapter, we were able to see hoe single-cycle processor, 5 stage processor w/o forward or hazard unit, 5 stage processor and even 6 stage dual processor. Understanding and researching these different architectures were essential for us to implement the correct processor for our needs and be approved by Northrop Grumman.

We are also able to see how ripes.me is also able to implement different Instruction Set Architectures. To our application, the only required extension is I and we discarded M and C as they were not applicable. When selecting the processor, we are also able to customize the register Initialization and visualize the memory map for the register initialization.

3.7 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 49 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.

3.7.1 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.

3.7.2 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 50 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.

3.7.3 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.

3.7.4 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 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.

31	30	25 24	21	20	19	15 14	12 11	8	7	6	0
		funct7		rs2		rs1	funct3		rd		opcode
		imm[11:0]			rs1	funct3		rd		opcode	I-type
		imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
		imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
		imm[31:12]						rd		opcode	U-type
		imm[20]	imm[10:1]	imm[11]	imm[19:12]			rd		opcode	J-type

Figure 3.28: 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.

3.8 RISC-V Instruction Implementation

The methods of how instruction types are decoded and how different modules interact with different instructions is documented in other sections of the paper, but this chapter aims to explain each of the 37 required instructions and how it was implemented within the stages of the design. As a quick review, the ASIC aims to perform 37 unique RV32I instructions which do not include memory ordering functions and environment calls such as FENCE and ECALL. In reference to the figure below, we do not implement FENCE, FENCE.I, ECALL, EBREAK, CSRRW, CSRRS, CSRRC, CSRRWI, CSRRSI, and CSRRCI. The exact control signals are discussed in the specification sheets for RAPID and RAPID-X.

RV32I Base Instruction Set						
imm[31:12]			rd		0110111	
imm[31:12]			rd		0010111	
imm[20:10:1 11:19:12]			rd		1101111	
imm[11:0]		rs1	000	rd	1100111	
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	
imm[11:0]		rs1	000	rd	0000011	
imm[11:0]		rs1	001	rd	0000011	
imm[11:0]		rs1	010	rd	0000011	
imm[11:0]		rs1	100	rd	0000011	
imm[11:0]		rs1	101	rd	0000011	
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	
imm[11:0]		rs1	000	rd	0010011	
imm[11:0]		rs1	010	rd	0010011	
imm[11:0]		rs1	011	rd	0010011	
imm[11:0]		rs1	100	rd	0010011	
imm[11:0]		rs1	110	rd	0010011	
imm[11:0]		rs1	111	rd	0010011	
0000000	shamt	rs1	001	rd	0010011	
0000000	shamt	rs1	101	rd	0010011	
0100000	shamt	rs1	101	rd	0010011	
0000000	rs2	rs1	000	rd	0110011	
0100000	rs2	rs1	000	rd	0110011	
0000000	rs2	rs1	001	rd	0110011	
0000000	rs2	rs1	010	rd	0110011	
0000000	rs2	rs1	011	rd	0110011	
0000000	rs2	rs1	100	rd	0110011	
0000000	rs2	rs1	101	rd	0110011	
0100000	rs2	rs1	101	rd	0110011	
0000000	rs2	rs1	110	rd	0110011	
0000000	rs2	rs1	111	rd	0110011	
0000000	pred	succ	00000	000	00000	0001111
0000	0000	0000	00000	001	00000	0001111
0000000000000			00000	000	00000	1110011
0000000000001			00000	000	00000	1110011
csr			rs1	001	rd	1110011
csr			rs1	010	rd	1110011
csr			rs1	011	rd	1110011
csr			zimm	101	rd	1110011
csr			zimm	110	rd	1110011
csr			zimm	111	rd	1110011

Figure 3.29: RV32I Base Instruction Set

3.8.1 R-Type Instructions

If we follow the implementation in the execute module logic then it makes sense to start with the register type instructions. The register type instructions are broken up into the following 6 bit-fields, funct7, rs2, rs1, funct3, rd, and the opcode. The instruction bit layout is shown in the figure below. The R-type instructions which we need to implement are add, sub, sll, slt, sltu, xor, srl, sra, or, and and.

31	30	25 24	21	20	19	15 14	12 11	8	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode		R-type

Figure 3.30: R-Type Instruction Bit Fields

The add instruction adds rs2 to rs1 and stores the value in rd and the sub instruction does the same thing except we subtract rs2 from rs1. To implement this in the execute stage we differentiated between the instruction with a single inverse operation control signal because it represents how subtraction can be performed by using an adder by simply inverting rs2 and setting the carry-in value to the adder to 1. This is equivalent to performing the 2's complement of the value in rs2 and then adding that 2's complement to the value in rs1.

The next two instructions are SLT and SLTU, which stand for set less than and set less than unsigned. These two instructions set the value of rd to 1 if the value of rs1 is less than the value of rs2, but SLTU considers the inputs to be unsigned and SLT considers the values to be signed. Our execution stage does not actually differentiate between the two of these instructions because our decode stage handles the sign extension so there is no need to use the System Verilog `$sign()` cast.

XOR and OR are next, which are the exclusive or and standard or operators which are often represented as `^` and `|` in most programming languages respectively. This instruction was very simple to implement as we just needed to use the `^` and `|` operators built into Verilog. The single bit truth tables for the functions are shown below, these operators are performed bitwise and thus are not signed. The resulting value is stored in register rd.

Table 12: OR Truth Table

A	B	A B
0	0	0
0	1	1
1	0	1
1	1	1

Table 13: XOR Truth Table

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

AND is the next bitwise operator and is performed by using the Verilog bitwise & operator. This operation is also unsigned because it also bitwise like XOR and OR and the truth table for a single bit of the operator is shown below.

Table 14: AND Truth Table

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

The shift instructions are a bit unique when compared to the other R-Type instructions because of how they alter the bit fields. The shift operators are SLL, SRL, and SRA, which stand for Shift Left Logical, Shift Right Logical, and Shift Right Arithmetic. What is unique about these instructions compared to other R-Type instructions is that they only use the 5 least significant bits of the register rs2. This is because with RV32I the XLEN of our registers is 32 bits, so if you shift a register's value 32 bits to the left it is guaranteed to be 0 and the same can be said for shifting a value to the right. SLL works by shifting in a zero for the least significant bit, SRL works the same way but instead shifts in a 0 to the most significant bit, but SRA works by shifting in a sign bit. An example of SRL vs SRA is shown in the figures below. In System Verilog the difference in syntax is `>>` vs `>>>` for performing SRL and SRA respectively.

SRL	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
rs1	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	1	1	1	0	1	1	0	0	1	1	1	1
rs2	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	1	1	1	0	1	1	0	0	1	1	1	1
rd	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	0

Figure 3.31: SRL Example

SRA	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
rs1	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	1	1	1	0	1	1	0	0	1	1	1	1	
rs2	1	1	0	0	0	0	0	1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	1	1	1	0	1	1	0	0	1	1	1	1	
rd	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	0

Figure 3.32: RA Example

3.8.2 I-Type

The next instruction family are the immediate type family. The immediate family shares a lot of instructions with the register type family except that the immediate type of family replaces rs2 with the immediate value encoded within the instruction. Due to this difference all the immediate instructions are sign extended. This gives the immediate value a range of -2048 to 2047, which is less than the possible values for a normal register, but the LUI function described later solves one of these issues. The I-Type instructions that we are going to implement are as follows, jalr, addi, slti, sltiu, xori, ori, andi, slli, srli, and srai, lb, lh, lw, lbu, lhu.

imm[11:0]	rs1	funct3	rd	opcode	I-type
-----------	-----	--------	----	--------	--------

Figure 3.33: Immediate Instruction Example

The immediate values are decoded in an interesting way when compared to r-type instructions, and the shift instructions are also a bit strange. Below are two figures which outline how these values are divvied up.

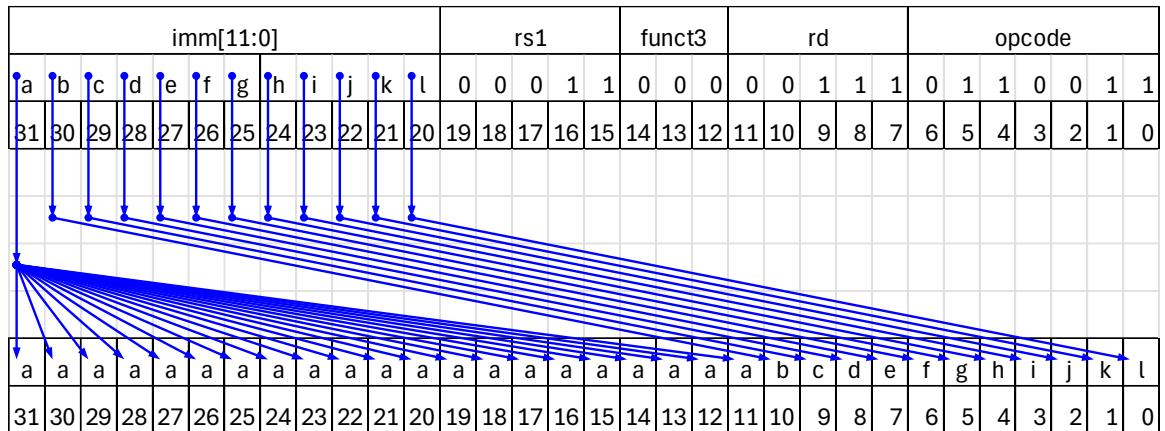


Figure 3.34: imm_i decoding

The addi instruction adds the immediate value to rs1 and then stores the value into the destination register rd. This is almost exactly the same as the add instruction but what is interesting about the add immediate instruction is that subtraction is already built into the function. This is due to the nature of immediate instructions always having a signed immediate value. This also means that there is no corresponding subi instruction because it would simply be the exact same instruction with a flipped immediate.

The next two instructions are SLTI and SLTIU, which stand for set less than immediate and set less than immediate unsigned. These two instructions work the same as the register based SLT and SLTU, but rs2 is swapped with the value of the immediate. If rs1 is less than the immediate then the destination register is set to 1. What is interesting about SLTI and SLTIU is that in the System Verilog it takes advantage of the fact that if there is an unsigned value in an operation then the entire operation is treated as unsigned, so all you need to do with SLTI and SLTIU is set the register as signed or unsigned in the instruction.

XORI and ORI are next, which are the exclusive or and standard or operators which are often represented as \wedge and \vee in most programming languages respectively. These instructions can still be performed bitwise because of the sign extension which takes place when you decode the immediate value. It may seem as if though the value would be ruined if you ORI a negative rs1 and a positive immediate value, but that is just how negative OR works so there is no need to worry about that. These functions are not often used however because when you use bitwise operators you are generally working on an entire 32-bit number. In the execute stage we perform this instruction by changing the value for port 2 to the immediate value if the IOP is enabled.

ANDI is a similar situation to XORI and ORI where the negative sign is preserved throughout operations, the destination register is set to the bitwise and of rs1 and the immediate value.

The shift instructions are yet again unique when compared to the other instructions in this family. We only use the 5 least significant bits of the immediate value just like before, but we also take bit 30 and use it to determine whether we perform SRAI vs SRLI. SLLI works the same but there is no need to distinguish between SLAI and SLLI. The lower 5 bits which we use for the jump are often denoted as the shamt.

imm[11:0]										rs1			funct3			rd			opcode												
0	b	0	0	0	0	h	i	j	k	l	0	0	0	1	1	0	0	0	0	1	1	1	0	1	1	0	0	1	1		
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 3.35: Immediate Shift Instruction

Now for the instructions that do not have register type counterparts, which include jalr, lb, lh, lw, lbu, and lhu. These instructions are relatively simple when compared to other instructions, but the implementation required in the other modules is what makes them so difficult to perform.

JALR stands for jump and link register and is one of the many jumps within RV32I. This instruction sets the destination register to the value of the current program counter plus four and then jumps the program counter to rs1 plus the immediate value. One facet of this is that the program counter can never be an odd address, so the instruction always sets the least significant bit of the jumping address to zero no matter what rs1 plus the immediate value actually equals. From the name JALR sounds as if it would be a register-based instruction but it is only named that due to the fact that there is a jump and link which does not even include a rs1 and only includes a larger immediate value. Some elaboration on why the instruction links the program counter plus four is because in most processor implementations the program counter increases by four always even if another value is added to the program counter for a jump or branch.

The load instructions are relatively simple for what they do, the goal of each load instruction is to set the destination register to the value of the fetched data from memory, but each of these instructions does it slightly differently. LB stands for load byte which fetches a sign extended byte from the memory address given by the sum of rs1 and the immediate value. LBU stands for load byte unsigned which fetches a zero extended byte from the memory address given by the sum of rs1 and the immediate value. LH stands for load half word, which for RV32I is a 16 bit half word, where a sign extended half word is fetched from the memory address given by the sum of rs1 and the immediate value. LHU is the same as LH but instead of being sign extended the half word is zero extended. LW returns the sign extended full word from the memory address which is the sum of rs1 and the immediate value. There is no need for an unsigned load word because the sign bit at the end is always included when the full word is loaded.

3.8.3 B-Type

The next instruction family are the branch type instructions which use an even immediate value as a jump value based on the condition type of the branch. The list of branch instructions are as follows: BEQ, BNE, BLT, BGE, BLTU, BGEU. What is interesting about the branch functions is that they contain both an immediate value a rs1 value and a rs2 value because the immediate value is used solely used to change the value of the program counter. Branch instructions are decoded in a very strange way, so the figure below shows how it is done in detail. The range of these branches is actually slightly less than -4096 to 4095 because the bottom bit of the jump value is always less 0, so we cannot get the extra bit of jumping space. This is also because we always want the branch location to be even because the pc counter always must be even. For the original RAPID design we often chose not to branch if we could because when branching we needed to completely fill the pipeline with noops (addi x0, x0, 0), but we support efficient branching in the current design.

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
---------	-----------	-----	-----	--------	----------	---------	--------	--------

Figure 3.36: B-Type Instruction Format

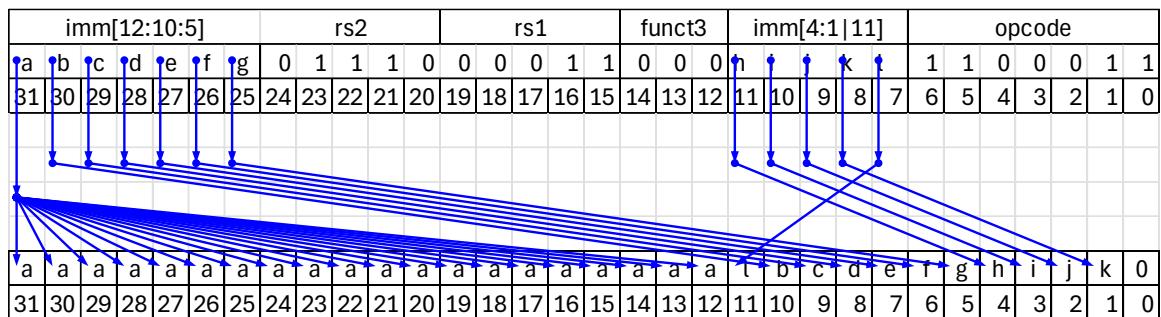


Figure 3.37: Decoding of Branching

The way the branches are evaluated is very simple, compared rs1 to rs2 based on which instruction. BEQ stands for branch on equal and allows for a branch when the value of rs1 is equal to the value of rs2. BNE stands for branch not equal and allows for a branch when the value of rs1 is not equal to rs2. BLT stands for branch less than and allows for a branch when the value of rs1 is less than the value of rs2. BGE stands for branch greater than or equal to and branches when the value of rs1 is greater than or equal to rs2. BLTU stands for branch less than unsigned, which means that the branch is allowed if the value of rs1 is less than the value of rs2, but both registers are treated as though their values are unsigned. BGEU stands for branch greater than or equal to unsigned and branches if the unsigned value of register 1 is greater than or equal to the unsigned value of register 2. For our processor we make sure to decode whether the register should be signed or unsigned when branching so we use the same comparisons for the unsigned operations as the signed operations.

3.8.4 J-Type

Similar to branches are jumps, but jumps are unconditional. The only J type instruction in the RV32I instruction set is jal, but jalr is pretty close. JAL has similar rules to branching and jalr where the least significant bit of the jump is always zero because the jumping address must be even. The immediate value for a J-Type instruction is 20 bits, but due to the last bit always being set to zero we effectively have a 21-bit immediate value. This gives us a jump range of -1048576 to 1048574 which is easily the biggest jump range out of all the instructions. Although jal is mainly just used for jumping, the instruction also write the program counter plus four is written to the destination register before the program counter jumps to the immediate value plus four.

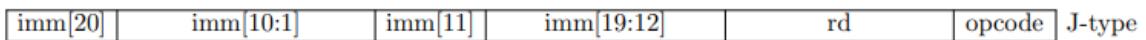


Figure 3.38: J-Type Instruction Format

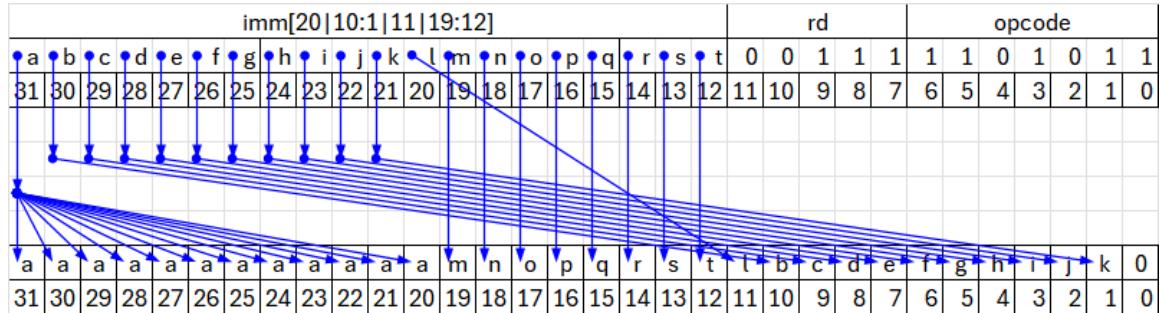


Figure 3.39: Decoding of Jump Type

3.8.5 U-Type

The U-type instructions are the upper immediate types which include the instructions lui and auipc. These two instructions take a 20-bit immediate along with a destination register and perform an operation. LUI sets the destination register to the upper immediate decoded immediate value while auipc adds the current program counter to the upper immediate decoded immediate value and then stores it in the destination register. These two instructions are very useful even if they seem confusing at first because they allow the user to possibly jump to higher ranges than the 21 bits allowed by jal and also allow the user to easily load values into registers which are bigger than the allowed 11 bits from addi. These two functions operate differently for a register size of 64 bits, but for our applications this does not even matter.



Figure 3.40: U-Type Instruction Format

imm[31:12]																rd				opcode											
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	0	0	1	1	1	0	1	0	0	1	1	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	0	0	0	0	0	0	0	0	0	0	0	

Figure 3.41: Upper Immediate Decoding

3.8.6 S-Type

The S-type instruction format is similar to the load instructions from the immediate family, but instead of loading data from the memory into a register the S-type family loads a register into a certain memory address. Store type instructions come in three different flavors, the SB instruction sets the address given by the sum of rs1 and the immediate value to the lower 8 bits of rs2, the SH instruction sets the address to the lower 16 bits, and SW sets the address to the entire value of rs2. These instructions are incredibly important to the data handling of the ASIC's pipeline, and it would be very difficult to perform a complex program without the combination of the load instructions and store instructions. The formatting of the immediate value for S-type instructions is not the most complicated, but a diagram is shown in the figure below to explain how the instruction is decoded.

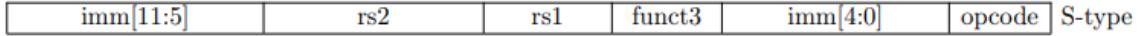


Figure 3.42: S-Type Instruction Format

imm[11:5]								rs2				rs1				funct3			imm[4:0]					opcode							
a	b	c	d	e	f	g	0	1	1	1	1	0	0	0	1	1	0	0	0	0	1	1	0	1	0	0	0	1	1		
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	b	c	d	e	f	g	h	i	j	k	l

Figure 3.43: S-Type Immediate Decode

3.9 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.

3.9.1 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.

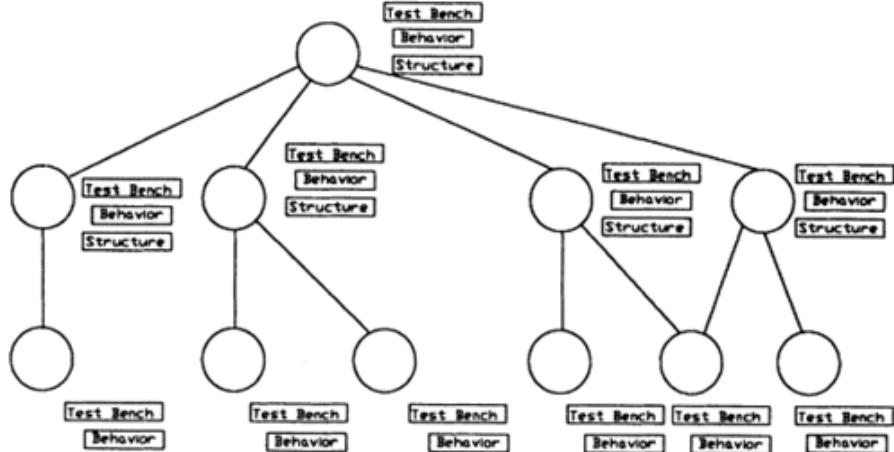


Figure 3.44: Behavioral Structure

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 granular 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.

3.9.2 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.

3.9.3 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.

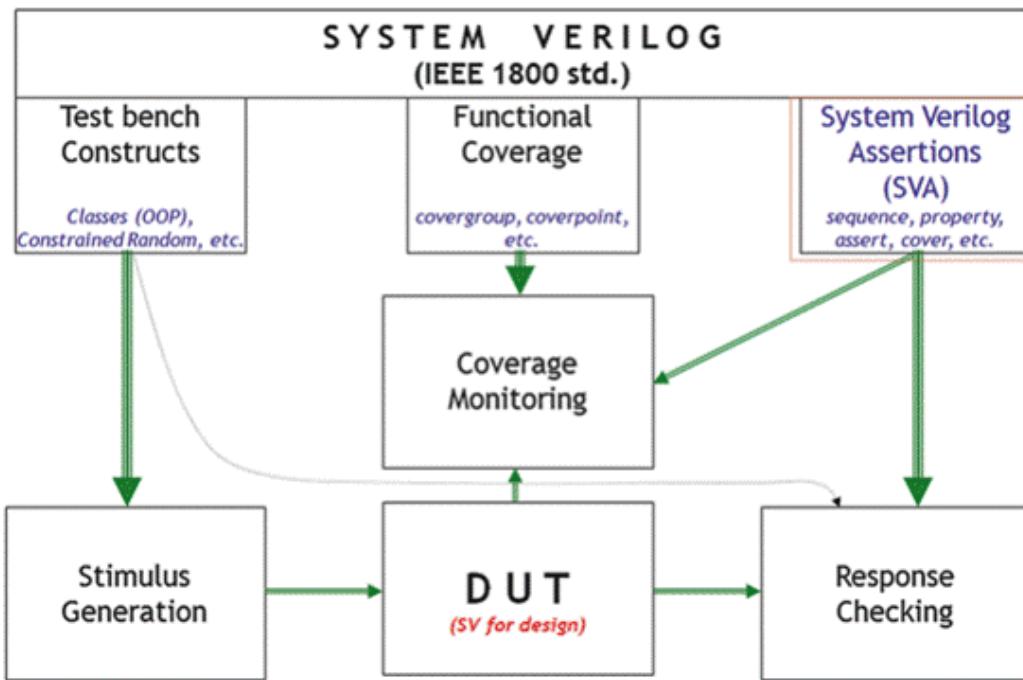


Figure 3.45: SystemVerilog Verification Environment Structure

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 SystemVerilog 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.

3.10 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.

3.10.1 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.

3.10.2 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 65 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.

3.10.3 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.

3.10.4 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.

3.10.5 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.

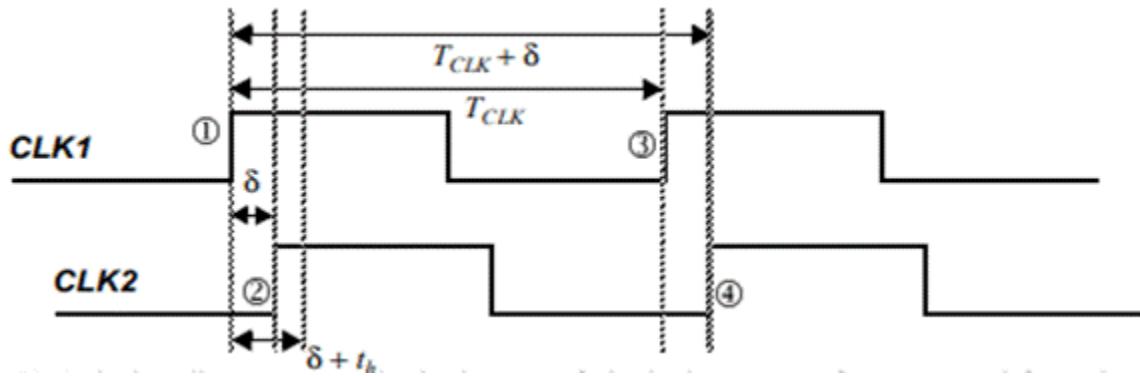


Figure 3.46: Positive Clock Skew and Timing Offset between CLK1 and CLK2.

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.

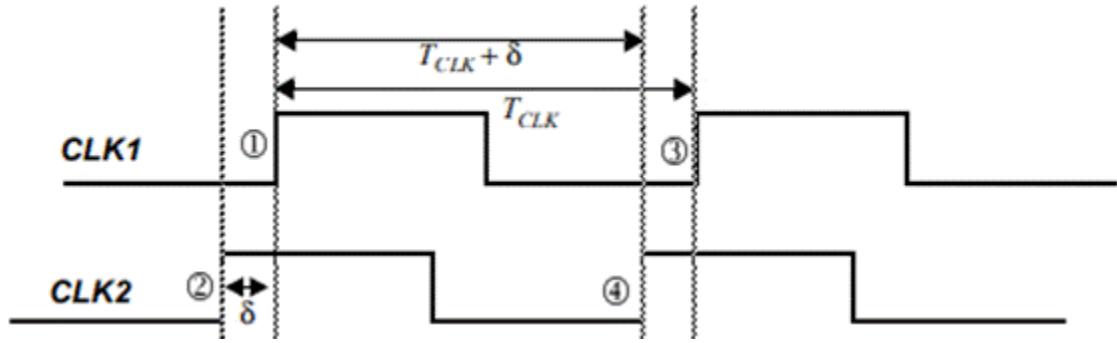


Figure 3.47: Negative Clock Skew and Timing Offset between CLK1 and CLK2.

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.

3.10.6 UVM and Metric-Driven Verification Flow

Functional verification serves to demonstrate the functional correctness of the design and attempts to find error and show that the design implements the specification. Additionally, it verifies functionality and timing characteristics. There are two main methods for verifications of integrated circuits: dynamic verification through simulation, and static verification through formal methods. UVM verification workflow based on a metric-driven, dynamic verification strategy.

The metric driven verification (MDV) flow is a closed loop process consisting of: planning an executable verification plan, constructing a MDV testbench, and measuring and analyzing the results. Engineers make use of the following components integrated in their

MDV workflow: verification plan, automatic generation of stimulus, coverage measurement, and definition of assertion. The last three components define three cornerstones of a modern verification flow: randomization, coverage, and assertions.

Coverage measures the progress against the verification plan, it quantifies what has and what has not been tested. It provides feedback and confirmation that verification goals have been reached. Together with verification automation, it seeks to provide randomized stimulus data, analyze coverage data, generate checker reports, and provide direction when goals are not met. Overall, coverage aims to approach verification completeness.

UVM is a library providing verification building blocks and automation. It is formally known as the IEEE 1800.1 standard. Moreover, it is also a methodology scalable from block-level to system-level verification. Engineers use UVM because it augments productivity, predictability, and quality of the design flow. The UVM workflow starts with a verification plan (or vPlan), which captures the foals of the verification process, defines criteria, and milestones. It must be measurable and executable, because it should be able to measure the progress towards the project's closure. The most important aspect of a vPlan is that it defines 'what' aspect of the design to verify, rather than 'how' to verify.

Finally, formal verification methods can be used to complement dynamic verification. The formal method is based on formal mathematical proofs, some examples are logic equivalence checking (LEC) and functional formal verification (formal analysis). Formal properties include assertions and assumptions (constraints). These are defined in a language known as SVA. In contrast with simulation, formal verification is exhaustive (that is, it hits all states of the design) and does not require preparing testbenches ahead of time. Formal property verification (FPV) becomes infeasible for large projects, but engineers can use semi-formal verification to help discover bugs. Some times include the aforementioned FPV, cycle swarm, state swarm, and guidepointing. These are controlled through metrics gathered to maximize coverage.

3.11 Direct Programming Interface (DPI)

The direct programming Interface (DPI) is a System Verilog library that allows System Verilog code to integrate with foreign programming languages such as C and C++ but it can also integrate with SystemC as well. This library allows engineers and designers to leverage a high-level programming language while running hardware verification code and testbenches. It also allows developers to utilize existing codebases and complex functions that can be difficult to write in pure System Verilog. This section goes over the basics of the direct programming interface, the architecture that DPI uses, how to import and export functions with the DPI library, and the compatible data types.

3.11.1 Purpose of DPI

The Direct Programming Interface which is a library offered with System Verilog aims to bridge the gap between System Verilog and foreign programming languages, particularly C and C++. This interface allows for direct function calls between System Verilog code and the mentioned languages above allowing for seamless integration and interoperability between high-level general programming

languages and hardware description languages. By allowing the System Verilog code to call functions written in C and C++, DPI allows for more flexibility in the verification process of hardware designs. This becomes extremely beneficial when dealing with existing codebases that may already exist in C or C++ as you can directly use the code within your testbench mitigating the need to translate the code into System Verilog. Not only does this save time by utilizing code that already exists, but some algorithms can be complex to write in System Verilog which makes it useful for mitigating complexity as well. DPI was designed to abstract away the complexities involved with interoperability the languages, allowing designers to focus purely on functionality without worrying about the complexities involved in combining the languages together.

3.11.2 Advantages of DPI

The Direct Programming Interface (DPI) is not the first interface that aimed to bridge the gap between System Verilog code and C and C++ code, but it is the most straightforward and intuitive way to connect your code together. The original method, the Programming Language Interface (PLI) from the mid 1980s provided designers a way to interface with external C functions and create custom tasks to aid in verification processes. However, PLI requires comprehensive knowledge on the linking process which can be complex and adds unnecessary steps which distract from the testing process. To address the issues the Verilog Procedural Interface (VPI) was developed part of the IEEE 1364 standard which simplified the process slightly while also allowing for object-oriented methodology to access the internal data structures of Verilog. However, it still required too much knowledge in the simulation process and distracted from actual verification code. That is why the Direct Programming Interface (DPI) was created to seamlessly integrate between System Verilog code and C and C++. This ease of integration allows for more flexibility in the type of code that can be integrated into testbenches.

The Direct Programming Interface (DPI) offers several advantages that can help enhance both the design stages as well as the verification process of hardware designs. One of the key advantages is code reusability. DPI allows designers to reuse existing C and C++ code directly into any System Verilog environment, which allows developers to utilize code that has already been written without needing to spend time reimplementing the code into System Verilog. This does not only save time when writing hardware designs but also allows for consistency across different designs. In addition, C and C++ code is highly optimized and can execute task more efficiently than a traditional System Verilog testbench which can lead to faster execution times for testbenches. It also has the advantage of being much simpler than traditional integration methodologies such as PLI. This simplicity reduces the learning curve required to use DPI while also reducing the errors due to verification setup.

3.11.3 DPI Import and Export Syntax

As covered in the previous section, DPI bridges the gap between System Verilog code and C++ code. It achieves this by using import and export syntax allowing for

direct function calls. If you want to make a function call from an external C++ program you would use the following syntax to import the function.

```
1. import "DPI-C" function return_type function_name (input_type input_arg, ...);
```

This tells the System Verilog compiler about the existence of an external C++ function and allows it to then be used in the System Verilog files as if they were native System Verilog functions. It also works the other way around, meaning if you would like to call a System Verilog function within a C++ program you would use the following export syntax.

```
1. export "DPI-C" function function_name;
```

This allows the System Verilog function to be called from within the C++ program. By using the import and export syntax you can easily bridge the gap between System Verilog and C++ and use them as native functions. This enables efficient integration between System Verilog and foreign programming languages which can simplify the verification process of hardware designs.

3.11.4 Foreign Language Layer

The Direct Programming Interface (DPI) allows for two different foreign languages to be integrated with System Verilog, C and C++. The foreign language layer allows designers to write code in C and C++ extending the verification capabilities of the System Verilog language. This is the layer where the complex tasks, functions and algorithms are written that need to be later be called from within the System Verilog testbench. For this integration to happen the C style functions need to be written in a very specific way to ensure the System Verilog simulator can find, use and execute the functions. The DPI library uses the standard C style linkage which is strictly enforced. It is extremely important when using C++ code as C++ compilers will mangle the namespace of functions to account for overloaded functions. This is not allowed with DPI as the simulator looks for the function name and if it gets mangled by the compiler it will not be recognized and will fail to link. To ensure that the compiler adheres to the standard C style linking your functions must use the following format.

```
1. extern "C" void func(int arg) {  
2.     ...  
3. }
```

The `extern C` keyword ensures that the compiler uses the standard C style naming while linking. It's also important that the data types used for the functions and their parameters have corresponding data types within System Verilog to prevent any data loss or unintended data manipulation.

The foreign language layer plays a critical role in leveraging C and C++ code within a System Verilog program and enhancing the functionality and efficiency of hardware verification environments. This layer enables the execution of specialized and computationally intensive tasks that might be less efficient to

implement directly in SystemVerilog. By utilizing existing libraries, algorithms, and toolchains developed in C and C++, designers can reduce development time and increase the accuracy of their testbenches. The foreign language layer is particularly useful for integrating third-party software tools or complex models, directly into the hardware verification process. This integration not only expands the capabilities of SystemVerilog but also ensures a smooth workflow by maintaining compatibility between the two programming environments. With the foreign language layer's ability to combine high-level programming with hardware-specific testing, DPI allows engineers to build robust, flexible, and scalable verification environments for modern hardware designs.

3.11.5 Isolation Between Layers

The Direct Programming Interface (DPI) maintains a clear and concise separation between the System Verilog layer and the foreign language layer. As seen in previous sections, to use an external function it must use the import or export syntax. Simply linking to the foreign language layer is not enough. It needs direct imports and exports for proper usage. You also cannot write your C/C++ and System Verilog within the same file. It requires a separate C file as well as a System Verilog file. This allows for easier maintainability as all changes are made in the foreign language layer are independent from the design and visa versa. This helps simplify the debugging process and also helps with code reusability as the same function can be imported/exported into many different files.

3.11.6 Type Compatibility

When working with DPI it's important that the data types used have a matching data type in both System Verilog and C/C++. System Verilog offers a wide range of the basic data types that most languages offer such as int, byte, and long int, all of which C and C++ offer. There are no issues when using data types that are part of both languages as they can transfer between the two languages. The issues arise when using more complex data types especially in C/C++ such as user defined structures which do not translate easily to types in System Verilog. To ensure that all code functions properly ensure that you reference the DPI documentation to ensure the types of your functions return as well as the parameters have a corresponding type in System Verilog and C/C++.

3.12 Random Instruction Generator for UVM Testing

The Random Instruction Generator (RIG) is a critical component for verifying the hardware design of our RISC-V core. Its primary function is to produce random but valid RISC-V32I instructions, encompassing a wide range of possible operations and edge case scenarios. By leveraging the pseudorandom nature of random number generators, the RIG enables the efficient creation of a wide array of input scenarios, ensuring the correctness of our design while meeting specified performance metrics.

One of the RIG's key objectives is to simulate real-world operations that a processor core might encounter. For our design, which supports 37 base

instructions, the RIG was developed to randomly generate all supported instruction types, including logical operations, arithmetic, and memory access. The RIG was also designed with extendibility in mind. The flexibility allows our generator to allow for future expansion of the types of instructions it can generate. Simply adding a new function and adding it to the map is all you need to do to add support for new instruction generation.

This dynamic design makes the RIG valuable not only for our testing purposes, but also for other projects or teams who also may need the ability to generate lots of random RISC-V32I instructions quickly. Additionally, because System Verilog offers the DPI library, this code can be directly used within System Verilog with very little modification, which further simplifies the verification process, as you no longer need to worry about the logic needed to ensure your inputs are valid. This will allow our UVM testbench to generate the random instructions we need for testing as if it were a native System Verilog function.

The Random Instruction Generator (RIG) was designed to create valid RISC-V32I instructions by using structured randomization and encoding techniques. Each instruction that the RIG produces is compliant with the RISC-V32I ISA and can be seen in the table below.

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7					rs2		rs1		funct3		rd		opcode	R-type
	imm[11:0]					rs1		funct3		rd		opcode	I-type	
	imm[11:5]				rs2		rs1		funct3	imm[4:0]		opcode	S-type	
	imm[12 10:5]				rs2		rs1		funct3	imm[4:1 11]		opcode	B-type	
		imm[31:12]								rd		opcode	U-type	
		imm[20 10:1 11 19:12]								rd		opcode	J-type	

Figure 3.48: The 6 types of RISC-V32I Instructions From the RISC-V Docs

This table from the RISC-V documentation shows most of the formats needed to encode each instruction the processor is capable of. While our processor only covers 37 of the RISC-V32I instructions the generator can easily be expanded to support all types of instructions, including from other variants such as RISC-V32M or RISC-V32A. This section will go over how the generator works to produce valid and random instructions.

3.12.1 Encoders

The encoders are the most important component of the Random Instruction Generator (RIG), as they are responsible for producing the 32-bit binary words required for the processor to interpret instructions correctly. For this project, six different encoders were implemented, corresponding to the six RISC-V instruction types our core supports: R, I, S, B, U, and J types. Each of the encoders adheres to its own unique format as seen in the figure above to ensure that all instructions are in proper specification to ensure the validity of the instructions being generated.

While each instruction type has its own distinct format, all encoders share a common goal which is to transforming random values generated by a pseudorandom number generator into properly structured 32-bit instructions. These encoders manipulate the generated random values, assigning them to the appropriate fields such as opcode, funct3, funct7, and registers (rs1, rs2, and rd). This process ensures that each field is correctly positioned and conforms to the required bit-length and functionality.

The following sections will provide a detailed look at each of the six encoders, highlighting their roles, structure, and how they maintain compliance with the RISC-V ISA to produce valid instructions.

3.12.2 R-Type Encoder

The R-Type Encoder is responsible for generating instructions that require two source registers, and a destination register. These instructions are typically arithmetic or logical operations such as ADD, SUB, AND, or OR. The R-Type encoder has the following fields:

1. **Opcode (bits 0–6)** – Positioned at the 7 least significant bits and specifies which type of instruction is being performed
2. **Rd (bits 7-11)** – The next 5 bits correspond to the destination register
3. **Funct3 (bits 12-14)** – Specifies the operation type within the R-Type category
4. **Rs1 (bits 15-19)** – Specifies the first source register to be used in the instruction
5. **Rs2 (bits 20-24)** – Specifies the second source register to be used in the instruction
6. **Funct7(bits 25-31)** – Provides additional details about the instruction operation

For an R-Type instruction, each field has a specific bit-width that it must adhere to, but each of the parameters listed above starts as a 32-bit integer. That is why it is important that each field is masked so that only the relevant bits are kept before being placed into their respective positions within the 32-bit instruction word.

The masking process ensures that no unintended bits from the input values overflow into other fields, which preserves the integrity of the encoded instruction. Each field is then shifted to its designated position using bitwise operations, and all fields are combined using bitwise OR (|) to produce the final 32-bit instruction. Below is an example of how an ADD instruction is encoded:

Step 1 Mask:

$$\begin{aligned} \text{Opcode}_{mask} &= 0b000000000000000000000000000000110011 \& 0x7F \\ &= 0b000000000000000000000000000000110011 \end{aligned}$$

$$\begin{aligned} \text{rd}_{mask} &= 0b000000000000000000000000000000101 \& 0x1F \\ &= 0b000000000000000000000000000000101 \end{aligned}$$

```


$$funct3_{mask} = 0b00000000000000000000000000000000 \& 0x7$$


$$= 0b00000000000000000000000000000000$$



$$rs1_{mask} = 0b00000000000000000000000000000011 \& 0x1F$$


$$= 0b00000000000000000000000000000011$$



$$rs2_{mask} = 0b00000000000000000000000000000011 \& 0x1F$$



$$funct7_{mask} = 0b00000000000000000000000000000000 \& 0x7F$$


```

Step 2: Left Shift:

Step 3: Combine using Bitwise OR:

$$instr = funct7_{shift} \mid rs2_{shift} \mid rs1_{shift} \mid funct3_{shift} \mid rd_{shift} \mid opcode_{shift}$$

$$= 0b000000001100011000001010110011$$

The example above is used for all R-Type instructions to ensure that all the randomly generated values are aligned to their proper bit-widths, ensuring the validity and correctness of the instruction. This process is not unique to R-Type instructions, and all the encoders follow a similar pattern to ensure proper alignment with the RISC-V instruction format.

For each instruction type (I-Type, S-Type, B-Type, U-Type, J-Type), the fields are masked to retain only the relevant bits, shifted to their designated positions, and then combined into a single 32-bit binary word. And while the position for each field is different for every instruction type, the approach of mask, shift, and OR remains consistent. For this reason each of the subsequent sections will only give the formula used, without the example.

By adhering to this encoding process, the Random Instruction Generator (RIG) guarantees that every instruction it generates, regardless of instruction type, is both valid and ready for use in hardware verification and testing. The accuracy of the instruction generator is imperative to ensure that the instructions being used for testing are valid so we can abstract away the worry about having good valid input for the testbenches.

3.12.3 I-Type Encoder

The I-Type Encoder is responsible for generating instructions that involve an immediate value and a source register. These instructions typically include arithmetic operations that include an immediate value in them such as ADDI, or SUBI. It also covers bitwise operations such as ANDI or ORI as well as memory operations such as LW or LB. The I-Type format includes the following fields:

1. **Opcode (bits 0–6)** – Specifies the I-Type format
2. **Rd (bits 7–11)** – Specifies the destination register
3. **Funct3 (bits 12–14)** - Specifies the specific operation type within the I-Type category
4. **Rs1 (bits 15–19)** – Specifies the source register used for the instruction
5. **Immediate (bits 20–31)** – A 12-bit constant value used in the instruction

The encoding process for I-Type instructions ensures that all fields are correctly positioned within the 32-bit instruction word. Similar to the R-Type encoder, the I-Type encoder involves three main steps: masking, shifting, and combining.

Step 1: Masking

Each field is masked to keep only the necessary bits required for the instruction. This ensures that extra bits in the 32-bit integers do not interfere with the final encoding. The table below shows the proper formulas for masking I-Type fields.

Table 15: Formulas used to Mask I-Type Fields

I-Type Masking Formulas	
Field	Formula
$opcode_{mask}$	$opcode \& 0x7F$
rd_{mask}	$rd \& 0x1F$
$funct3_{mask}$	$funct3 \& 0x7$
$rs1_{mask}$	$rs1 \& 0x1F$
imm_{mask}	$imm \& 0xFFFF$

Step 2: Shifting

Once the fields are masked, they are shifted to their proper bit positions within the 32-bit instruction word. The table below shows the formulas required to properly align each field to its appropriate section of the 32-bit word.

Table 16: Formulas for I-Type Encoder Shifts

I-Type Shift Formulas	
Field	Formula
$opcode_{shift}$	$opcode_{mask}$
rd_{shift}	$rd_{mask} \ll 7$
$funct3_{shift}$	$funct3_{mask} \ll 12$

imm_{shift}	$imm_{mask} \ll 20$
---------------	---------------------

Step 3: Combining

The fields are combined using the bitwise OR (`|`) operator to produce the final 32-bit binary instruction.

Table 17: Formula for Encoded I-Type Instruction

Encoded I-Type Instruction Formula
$instr = imm_{shift} \mid rs1_{shift} \mid funct3_{shift} \mid rd_{shift} \mid opcode_{shift}$

Example of I-Type Encoding:

This example shows the proper use of the above formulas to derive an ADDI instruction with the following fields:

<i>opcode</i> = 0b0010011	I-Type
<i>rd</i> = 0b00101	Register x5
<i>funct3</i> = 0b000	ADDI operation
<i>rs1</i> = 0b00011	Register x3
<i>imm</i> = 0b000000000101	5

Step 1: Mask Fields

opcode = 0b0010011 & 0x7F = 0b0010011

$$rd = 0b00101 \& 0x1F = 0b00101$$

funct3 = 0b000 & 0x7 = 0b000

$$rs1 = 0b00011 \& 0x1F = 0b00011$$

$$imm = 0b000000000101 \& 0xFFFF = 0b000000000101$$

Step 2: Shifting

opcode = 0b00000000000000000000000000000010011

`funct3 = 0b00 << 12 = 0b00000000000000000000000000000000`

$rs1 = 0b00011 \ll 15 = 0b00000000000000000000000011000000$

imm = 0b000000000101 << 20 = 0b0000000000000001010000000000000000000000

Step 3: Combining

instr = *imm* | *rs1* | *funct3* | *rd* | *opcode* = 0000010100110000000000101100011

The I-Type encoder can encode instructions with immediate values by following a process of masking, shifting, and combining fields. It can encode a wide range of I-Type instructions, making it an essential component of the Random Instruction Generator (RIG). This structured approach ensures that all generated instructions are valid and ready for use in processor verification and testing.

3.12.4 S-Type Encoder

The S-Type Encoder is responsible for encoding instructions that involve storing data from a register into memory. Unlike the R and I type instructions, there is no destination register, as it will be moving data from registers to memory. Instead, it requires the following fields:

1. **Opcode (bits 0–6)**: Specifies the S-Type format.
2. **Immediate (bits 7–11, 25–31)**: A 12-bit signed value split across two segments.
3. **Funct3 (bits 12–14)**: Specifies the operation type within the S-Type category.
4. **Rs1 (bits 15–19)**: The base address register.
5. **Rs2 (bits 20–24)**: The source register containing the data.

You can see from the bits above that the immediate field is split into two different sections of the 32-bit instruction word. This happens because there is no continuous number of bits in a S-Type instruction that can fit all 12-bits needed together. This adds an extra step in the encoding process, as we must first split the immediate value before, we can do the mask, shift and combine operations. Below are the steps required to properly encode a S-Type instruction.

Step 1: Split the Immediate value

Before we can start the typical encoding process of masking, shifting, and combining, we must first properly extract the upper and lower bits for the immediate field, as they are not continuous within a the encoded 32-bit S-Type instruction. The formulas below show how to properly split the immediate value:

Table 18: Formulas to Split the Immediate Value for S-Type Instructions

Formulas to split the Immediate value in S-Type Instructions	
imm_{lower}	$(imm \gg 5) \& 0x7F$
imm_{upper}	$imm \& 0x1F$

Step 2: Masking

Now that the immediate value has been properly split, we can continue with the normal encoding process that begins with masking. The formulas below show the proper way to mask each field for an S-Type Instruction:

Table 19: Formulas for Masking S-Type fields

Formulas for Masking S-Type Fields

$opcode_{mask}$	$opcode \& 0x7F$
$imm_{lowerMask}$	imm_{lower}
$imm_{upperMask}$	imm_{upper}
$funct3_{mask}$	$funct3 \& 0x7$
$rs1_{mask}$	$rs1 \& 0x1F$
$rs2_{mask}$	$rs2 \& 0x1F$

Step 3: Shifting

With each field now masked so only the proper bits are kept, we can now shift them into the correct position of a 32-bit word by using left shifts which will prep them to be combined into one instruction for the next step. The table below shows the formulas used to align each field to its proper position for a S-Type instruction.

Table 20: Formulas for shifting fields in S-Type encoding

Formulas for Shifting S-Type Fields	
$opcode_{shift}$	$opcode_{mask}$
$imm_{lowerShift}$	$imm_{lowerMask} \ll 7$
$imm_{upperShift}$	$imm_{upperMask} \ll 25$
$funct3_{shift}$	$funct3 \& 0x7 \ll 12$
$rs1_{shift}$	$rs1 \& 0x1F \ll 15$
$rs2_{shift}$	$rs2 \& 0x1F \ll 20$

Step 4: Combining

The Final step for encoding a S-Type instruction uses the bitwise-OR operation to take each field, which has been properly masked and shifted to combine them into one valid RISC-V instruction. The formula to combine all the fields is shown in the table below.

Table 21: Formula to combine all S-Type fields into a valid instruction

Encoded S-Type Instruction Formula
$instr = imm_{upperShift} rs2_{shift} rs1_{shift} funct3_{shift} opcode_{shift}$

Example of S-Type Encoding:

To better understand how the S-Type encoding works, and how the immediate field is split before masking, this section will cover an example of how the S-Type encoder would encode a SW instruction.

$opcode = 0b0100011$	S-Type Instruction
$funct3 = 0b010$	SW Instruction
$rs1 = 0b00101$	Random base register address
$rs2 = 0b0011$	Random source register (x3)

imm = 0b000000001101 Random immediate value (13)

Step 1: Split the immediate value

$$imm_{lower} = (0b000000001101 \gg 5) \& 0x7F = 0b01101$$

$$imm_{upper} = 0b000000001101 \& 0x1F = 0b0000000$$

Step 2: Masking

$$opcode_{mask} = 0b0100011 \& 0x7F = 0b0100011$$

$imm_{upperMask} = imm_{upper} = 0b00000000$

$$imm_{lowerMask} = imm_{lower} = 0b01101$$

$$funct3_{mask} = 0b010 \& 0x7 = 0b010$$

$$rs1_{mask} = 0b00101 \& 0x1F = 0b00101$$

$$rs2_{mask} = 0b0011 \& 0x1F = 0b00011$$

Step 3: Shifting

$opcode_{shift} = opcode_{mask} = 0b00000000000000000000000000000000100011$

$imm_{lowerShift} = imm_{lowerMask} \ll 7 = 0b00000000000000000000000000101000000$

$$imm_{upperShift} = imm_{upperMask} \ll 25 = 0b000000000000000000000000000000$$

$funct3_{shift} = funct3_{mask} \ll 12 = 0b0000000000000000100000000000$

$$rs1_{shift} = rs1_{mask} \ll 15 = 0b000000000000010100000000000000$$

$rs2_{shift} = rs2_{mask} \ll 20 = 0b00000000000110000000000000000000$

Step 4: Combine

$$instr = imm_{upperShift} \mid rs2_{shift} \mid rs1_{shift} \mid funct3_{shift} \mid imm_{lowerShift} \mid opcode_{shift} \\ = 0b00000000000011000101001010100011$$

The S-Type encoder ensures that all the fields are properly aligned within the 32-bit instruction word to produce a valid RISC-V instruction. The splitting of the immediate value into `imm_lower` and `imm_upper` makes the S-Type instruction unique among other types of instructions and it must be handled accordingly. By adding the extra step of splitting the immediate value before the typical mask, shift, and combine operation ensures that the S-Type instruction will be properly formatted into a valid RISC-V instruction.

3.12.5 B-Type Encoder

The B-Type Encoder is responsible for encoding branch instructions, which compare two register values and conditionally alter the program counter based on the result. These instructions are responsible for implementing control flow within a program, such as conditional loops or if-else statements. The B-Type format uses the following fields:

1. **Opcode (bits 0–6)**: Specifies the B-Type instruction format.
2. **Immediate (bits 7–11, 25–31)**: A 12-bit value split across two segments, representing the branch offset.
3. **Funct3 (bits 12–14)**: Specifies the type of branch operation
4. **Rs1 (bits 15–19)**: The first source register used in the comparison.
5. **Rs2 (bits 20–24)**: The second source register used in the comparison.

Very similar to the S-Type instructions, the B-Type also splits its Immediate field in two separate non-continuous portions of the encoded instruction. This is for the same reason as the S-Type where there just aren't enough continuous bits for the Immediate value so it must be split. This adds the same additional step that the S-Type instructions have where we must first split the immediate value before we can encode. However, because we already included the split in its own step for S-Type I will combine it with the mask step to save time as it is not strictly necessary for it to have its own step. Below are the steps required to properly encode a B-Type instruction.

Step 1: Mask

The first step to encoding the random values into a B-Type instruction is to split and mask all the fields so only the important bits are left. This will prevent any unintended bits from being set in the final instruction. The formulas to mask each field for a B-Type instruction are shown in the table below.

Table 22: Masking Formulas for B-Type Instructions

Formulas for Masking B-Type Fields	
$opcode_{mask}$	$opcode \& 0x7F$
$imm_{lowerMask}$	$(imm \gg 1) \& 0xF$
$imm_{upperMask}$	$(imm \gg 5) \& 0x7F$
imm_{12}	$(imm \gg 11) \& 0x1$
imm_{11}	$(imm \gg 1) \& 0x1$
$funct3_{mask}$	$funct3 \& 0x7$
$rs1_{mask}$	$rs1 \& 0x1F$
$rs2_{mask}$	$rs2 \& 0x1F$

Step 2: Shifting

Now that all of the fields have been properly masked, and all of the bits that are not needed have been removed, you can now use the left shift operation to move

each field to the appropriate bit position. This way later we will easily be able to combine them together. The following table shows the formulas used to ensure each field is located correctly in the 32-bit word.

Table 23: Shift Formulas for Encoding B-Type Instructions

Formulas for Shifting B-Type Fields	
$opcode_{shift}$	$opcode_{mask}$
$imm_{lowerShift}$	$imm_{lowerMask} \ll 8$
$imm_{upperShift}$	$imm_{upperMask} \ll 25$
$imm_{12Shift}$	$imm_{12} \ll 31$
$imm_{11Shift}$	$imm_{11} \ll 7$
$funct3_{shift}$	$funct3_{mask} \ll 12$
$rs1_{shift}$	$rs1_{mask} \ll 15$
$rs2_{shift}$	$rs2_{mask} \ll 20$

Step 3: Combining

Now that all the bits are in the correct positions in their own variables, and all the unnecessary bits have been removed by the masking, you can now use the bitwise-OR operations to combine all the fields into one valid B-Type instruction. The formula to combine all fields into one instruction is shown in the table below.

Table 24: Formula to Encode a B-Type Instruction

Encoded B-Type Instruction Formula
$instr = imm_{12Shift} imm_{upperShift} rs2_{shift} rs1_{shift} funct3_{shift} imm_{11Shift} opcode_{shift}$

Example B-Type Encoding:

To better understand how the B-Type encoding process works, this section will go through a BEQ instruction example with the following fields:

$opcode = 0b1100011$	B-Type Instruction
$funct3 = 0b000$	BEQ instruction
$rs1 = 0b00001$	Random register (x1)
$rs2 = 0b00010$	Random register (x2)
$imm = 0b000000000110$	Random Offset (6)

Step 1: Masking

$$opcode_{mask} = 0b1100011 \& 0x7F = 0b1100011$$

$$imm_{lowerMask} = (0b000000000110 \gg 1) \& 0xF = 0b0011$$

$$imm_{upperMask} = (0b000000000110 \gg 5) \& 0x7F = 0b0000000$$

$$imm_{12} = (0b000000000110 \gg 11) \& 0x1 = 0b0$$

$imm_{11} = (0b000000000110 \gg 1) \& 0x1 = 0b0$

$$func3_{mask} = 0b000 \& 0x7 = 0b000$$

$$rs1_{mask} = 0b00001 \& 0x1F = 0b00001$$

$$rs2_{mask} = 0b00010 \& 0x1F = 0b00010$$

Step 2: Shifting

$opcode_{shift} = opcode_{mask} = 0b000000000000000000000000000000001100011$

$imm_{upperShift} = imm_{upperMask} = 0b000000000000000000000000000000$

$imm_{12Shift} = imm_{21} \ll 31 = 0b000000000000000000000000000000$

$imm_{11Shift} = imm_{11} \ll 7 = 0b000000000000000000000000000000$

$funct3_{shift} = funct3_{mask} \ll 12 = 0b00000000000000000000000000000000$

$$rs1_{shift} = rs1_{mask} \ll 15 = 0b0000000000000000100000000000000$$

$$rs2_{shift} = rs2_{mask} \ll 20 = 0b00000000001000000000000000000000$$

Step 3: Combine:

instr

```
= imm12shift | immupperShift | rs2shift | rs1shift | funct3shift | imm11shift | opcodeshift
= 0b0000000000010000010000110000110011
```

Overall, the B-Type encoder allows the RIG to generate a valid random B-Type instruction by properly splitting all of the fields into their appropriate bit-widths and uses bitwise operations to make sure they all combine nicely into one valid instruction. This encoder is similar to the others in that it follows the same basic steps, but because the immediate bits are split across the encoded instruction more than the S-Type, it can be a bit trickier to keep track on what is happening. The RIG handles this for us so we will not need to worry about this in our testbenches.

3.12.6 U-Type Encoder

The U-Type Encoder is responsible for handling both the LUI (Load Upper Immediate) and AUIPC (Add Upper Immediate to PC) instructions within our RISC-V processor core. Unlike other instruction types, U-Type instructions do not use source registers or require the immediate field to be split. This eliminates the additional complexity of managing a split immediate value, but the encoder still adheres to the same methodology of masking, shifting, and combining fields to produce valid instructions. The simplicity of U-Type instructions, with fewer fields,

means that there fewer formulas to apply during encoding. Below is a breakdown of the fields utilized in a U-Type instruction:

1. **Opcode (bits 0–6)**: Specifies the U-Type instruction format.
2. **Rd (bits 7–11)**: Specifies the destination register.
3. **Immediate (bits 12–31)**: A 20-bit constant value.

Step 1: Mask

Although U-Type instructions involve fewer fields, the encoding methodology remains consistent. The first step involves masking each field to ensure that any irrelevant or unnecessary bits are removed, preserving only the bits required for a valid instruction. This prevents invalid or unintended behavior caused by data in a bit position that should not be there in a field. The table below shows the proper masking equations to encode a U-Type instruction.

Table 25: Formulas for Masking U-Type Instructions

Formulas for Masking U-Type Fields	
$opcode_{mask}$	$opcode \& 0x7F$
rd_{mask}	$rd \& 0x1F$
imm_{mask}	$imm \& 0xFFFF$

Step 2: Shifting

Similar to all the other encoders, the second step in the encoding process for a U-Type instruction involves using the left shift operation to properly align all bits in the correct position. The table below shows the formulas used for each field in a U-Type encoding to ensure each field has its bits in the proper position.

Table 26: Formulas for Shifting Fields for a U-Type Encoding

Formulas for Shifting U-Type Fields	
$opcode_{shift}$	$opcode_{mask}$
rd_{shift}	$rd_{mask} \ll 7$
imm_{shift}	$imm_{mask} \ll 12$

Step 3: Combine

Now that all the bits have been properly masked and shifted to the proper locations for U-Type instructions, we can now simply apply a bitwise-OR operator to all of the fields to get one encoded U-Type instruction. The table below shows the proper formula to achieve a encoded U-Type instruction.

Table 27: Formula for Encoded U-Type Instruction

Encoded U-Type Instruction Formula

$$instr = imm_{shift} \mid rd_{shift} \mid opcode_{shift}$$

Example U-Type Instruction:

To get a better understanding of how U-Type Encoding works this section will cover a LUI instruction encoding example with the following field values

<i>opcode</i> = 0b0110111	U-Type (LUI)
<i>rd</i> = 0b00010	Random destination register (x2)
<i>imm</i> = 0b000000000000010100000	Random immediate value (0xA00)

Step 1: Masking

$$opcode_{mask} = 0b01101111 \& 0x7F = 0b0110111$$

$$rd_{mask} = 0b00010 \& 0x1F = 0b00010$$

$$imm_{mask} = 0b0000000000000010100000 \& 0xFFFFF \\ = 0b0000000000000010100000$$

Step 2: Shifting

$opcode_{shift} = 0b0000000000000000000000000000110111$

$rd_{shift} = 0b00010 \ll 7 = 0b000000000000000000000000100000000$

$$imm_{shift} = 0b00000000000010100000 \ll 12 \\ \equiv 0b0000000000001010000000000000000$$

Step 3: Combing

$$instr = imm_{shift} \mid rd_{shift} \mid opcode_{shift}$$

$$\equiv 0b00000000000001010000001000110111$$

While the U-Type encoder follows the same design steps as all encoders, the reduced number of fields cuts back on the number of equations needed to properly encode a U-Type instruction. The simplicity makes it a good example to follow the logic as there are far fewer equations being used to get to the encoded instruction. The U-Type instruction encoder plays a vital role in ensuring our testbenches can cover all possible instructions and ensures the validity of the instruction to the testbench does not need to worry about it.

3.12.7 J-Type Encoder

The final encoder that our instruction generator will handle is the J-Type Encoder. The J-Type Encoder is responsible for encoding jump instructions, such as JAL which modifies the program counter and store the return address in a destination register. The J-Type instruction is important for implementing control flow within a program such as function calls and unconditional jumps. The J-Type encoder like

all the others follows the same three tasks: mask, shift and combine, however like other formats, the immediate field is broken into 4 non-continuous bits for the encoded instruction. Below are the required fields for a valid J-Type instruction.

1. **Opcode (bits 0–6)**: Specifies the J-Type instruction format.
2. **Rd (bits 7–11)**: Specifies the destination register to store the return address.
3. **Immediate (bits 12–31)**: A 21-bit signed offset value split into multiple segments.

Step 1: Masking

The first step in encoding a J-Type instruction is to split the immediate field into its respective segments and mask each field to preserve only the relevant bits. This ensures that the final instruction only contains valid bits and eliminates any unintended data. The table below shows the masking formulas for J-Type instructions.

Table 28: Encoding Formulas for Masking J-Type Instructions

Formulas for Masking J-Type Instructions	
Field	Formula
$opcode_{mask}$	$opcode \& 0x7F$
rd_{mask}	$rd \& 0x1F$
imm_{20}	$(imm \gg 20) \& 0x1$
imm_{10}	$(imm \gg 1) \& 0x3FF$
imm_{11}	$(imm \gg 11) \& 0x1$
imm_{19}	$(imm \gg 12) \& 0xFF$

Step 2: Shifting

After masking the fields so only the proper bits are set for a J-Type instruction, we can then shift all the bits into their correct locations so combining later becomes a much simpler process. The formulas used to properly align each field into the correct bit position are given in the table below:

Table 29: Encoding Formulas for Shifting J-Type Fields

Formulas for Shifting J-Type Instructions	
Field	Formula
$opcode_{shift}$	$opcode_{mask}$
rd_{shift}	$rd_{mask} \ll 7$
$imm_{20shift}$	$imm_{20} \ll 31$
$imm_{10shift}$	$imm_{10} \ll 21$
$imm_{11shift}$	$imm_{11} \ll 20$
$imm_{19shift}$	$imm_{19} \ll 12$

Step 3: Combining

Once all the fields have been properly masked, so that the only data in each field is valid, and we have properly shifted everything so all the fields are in their correct positions for a J-Type instruction, we can now simply use a bitwise-OR operation to combine all the fields into one encoded 32-bit instruction word. The formula to combine all the fields into one encoded instruction are given in the table below.

Table 30: Formula for Encoding a J-Type Instruction

Encoded U-Type Instruction Formula					
$instr = imm_{20shift}$	$imm_{19shift}$	$imm_{11shift}$	$imm_{10shift}$	rd_{shift}	$opcode_{shift}$

Example of J-Type Encoder:

To better understand how all these formulas work to encode a J-Type instruction we will go through a JAL instruction example with the following fields.

<i>opcode</i> = 0b1101111	JAL Instruction Opcode
<i>rd</i> = 0b00001	Random register (x1)
<i>imm</i> = 0b0000000000000010101010101010101011	Random Imm (0x15555)

Step 1: Masking

$$opcode_{mask} = 0b1101111 \& 0x7F = 0b1101111$$

$$rd_{mask} = 0b00001 \& 0x1F = 0b00001$$

$imm_{20} = (0x15555 \gg 11) \& 0x1 = 0b1$

$$imm_{10} = (0x15555 \gg 1) \& 0x3FF = 0b1010101010$$

$$imm_{11} = (0x15555 \gg 11) \& 0x1 = 0x1$$

$$imm_{19} = (0x15555 \gg 12) \& 0xFF = 0b00010101$$

Step 2: Shifting

$opcode_{shift} = 0b000000000000000000000000000000001101111$

$$rd_{shift} = rd_{mask} \ll 7 = 0b000000000000000000000000000010000000$$

$imm_{20shift} = imm_{20} \ll 31 = 0b10000000000000000000000000000000$

$imm_{10shift} = imm_{10} \ll 21 = 0b0000001010101010000000000000000$

$$imm_{11shift} = imm_{11} \ll 20 = 0b00000000000000001000000000000000$$

$$imm_{19shift} = imm_{19} \ll 12 = 0b000000000000000010101000000000000$$

Overall the J-Type encoder follows the same basic principles as all 6 of the encoders, while having the additionally handle splitting the immediate value for proper encoding.

3.12.8 Example Output from the RIG

The image below shows the output of running the instruction generator as a standalone C++ program along with the same instructions inside Ripes.me to show that they are valid RISC-V instructions. The next section will go over how we converted the C++ code to work with System Verilog.

Assembly Instruction	Encoded (Hex)	Encoded (Binary)
bge x21, x3, 316	123ade63	00010010001110101101111001100011
jal x25, -360148	92ca8cef	10010010110010101000110011101111
lui x7, 319352	4df783b7	01001101111011110000011101101111
bgeu x13, x8, 1166	4886f763	01001000100001101111011101100011
lh x0, 1319(x3)	52719003	0101001001110001100100000000011
addi x19, x18, 127	7f90993	0000011111110010000100110010011
bltu x19, x17, 736	2f19e063	0010111100011001111000001100011
jalr x15, x4, -1929	877207e7	1000011101110010000001111100111
lh x29, 65(x14)	4171e83	00000100000101110001111010000011
lh x27, 760(x12)	2f861d83	0010111100001100001110110000011

Figure 3.49: Random Instructions Generated by Custom Generator

0:	123ade63	bge x21 x3 316	IF
4:	92ca8cef	jal x25 -360148	
8:	4df783b7	lui x7 0x4df78	
c:	4886f763	bgeu x13 x8 1166	
10:	52719003	lh x0 1319 x3	
14:	07f90993	addi x19 x18 127	
18:	2f19e063	bltu x19 x17 736	
1c:	877207e7	jalr x15 x4 -1929	
20:	04171e83	lh x29 65 x14	
24:	2f861d83	lh x27 760 x12	

Figure 3.50: Same instructions from generator in Ripes.me to show validity

3.12.9 Connecting the RIG to System Verilog

Integrating the custom instruction generator which is written in C++ with System Verilog involves utilizing the Direct Programming Interface (DPI). This will allow our System Verilog UVM testbench to make a call to the C++ function to obtain a valid random instruction. Below are the steps taken to convert the pure C++ to work with the System Verilog UVM test benches.

3.12.10 Create a function for DPI

The first step to ensure that the instruction generator code works in our UVM testbenches is to ensure that all the functions that will be called from System Verilog are marked as extern “C”. This prevents name mangling which occurs in a typical C++ compilation process. Name mangling occurs to prevent conflicts with overloaded functions with the same names, but this is not allowed with DPI nor is it needed. Marking the function as extern “C” uses the traditional naming conventions during compilation. This allows it to keep its original function name which System Verilog is looking for. To make sure we only get the encoded instruction we made a gen_rand_instr_dpi function dedicated for integration with System Verilog.

```
1. extern "C" unsigned int gen_rand_instr_dpi() {
2.     static bool initialized = false;
3.     if (!initialized) {
4.         initialize_register_map();
5.         initialized = true;
6.     }
7.
8.     std::pair<std::string, uint32_t> instr = gen_rand_instr();
11.    return instr.second;
12. }
13.
```

This function first ensures that all the register maps have been initialized because the instruction generator relies on the correct mappings of all the registers (x0-x31). This initialization only needs to happen once, and it is a costly function to map thus the static variable keeps track of the initialization state. It then calls the gen_rand_instr method which handles generating a random instruction and stores the encoded instruction as well as the assembly string version. We don't care about the assembly string so we disregard it and only return the encoded instruction. Note that the function definition is marked as extern “C” to prevent name mangling.

3.12.11 Creating a System Verilog Module

The next step is to create a module that will import the function that generates a random instruction. DPI function imports must occur within a module which is why we create a dedicated module for interfacing between the C++ code and the System Verilog code. This module is very simple as its only purpose is to import the function for use. Below is a image of a module that correctly imports the function.

```
1. module instr_gen_module;
2.
3. import "DPI-C" function int unsigned dpi_test();
4.
5. endmodule: instr_gen_module
6.
```

3.12.12 Connecting the Generator to the UVM Environment

The next step is to integrate the instr_gen_module into the UVM environment to allow the testbench to use the random instruction generator seamlessly. This involves creating a connection between the UVM driver and the imported DPI function so that instructions can be retrieved and applied to the design under test (DUT). Here's how this step is implemented.

The virtual interface is used to connect the instr_gen_module to the UVM driver class. This interface allows the driver to call the gen_rand_instr_dpi function via the imported module. The interface is declared below:

```
1. interface instr_gen_interface;
2.   logic [31:0] instruction;
3.   import "DPI-C" function int unsigned gen_rand_instr_dpi();
4.
5.   task generate_instruction();
6.     instruction = gen_rand_instr_dpi();
7.   endtask
8. endinterface: instr_gen_interface
9.
```

You can then use this virtual interface within the driver class to make calls to generate random instructions during the simulation.

3.12.13 Running and Debugging the Testbench

The final step in integrating the random instruction generator into the UVM testbench is to compile and run the simulation while validating that everything functions correctly. This process involves proper compilation, runtime configuration, and debugging to ensure the generated instructions are applied seamlessly to the design under test (DUT). Here's how this step is carried out:

Compiling the C++ Instruction Generator

Before running the simulation, the C++ code for the instruction generator must be compiled into a 32-bit object file. This is required by the Cadence Xcelium simulator which requires 32-bit object files for DPI integration. You can use the following command to compile the C++ code:

```
g++ -m32 -c -o instr_gen.o instr_gen.cpp -fPIC -I{path_to_svdpi.h}
```

- -m32 – Compiles the file as 32-bit
- -c – Compile only does not link
- -o – Names the output file
- -fPIC – Allows the file to be dynamically loaded
- -I{path_to_svdpi} – Ensure you give the proper includes for the dpi library

Linking the C++ Object File with the Simulator

- Once compiled, the object file must be linked to the simulator during simulation. You can do this in a couple of different ways, but for our tests I linked them inside our run.sh file which runs our UVM testbench. Below is an example script that properly links the compiled object file with the simulation.

```
• 1. xrun -Q -unbuffered \
• 2. -timescale 1ns/1ns -sysv \
• 3. +inccdir+$UVM_HOME/src \
• 4. rapid_pkg.sv \
• 5. interface.sv sequence_item.sv sequence.sv \
• 6. sequencer.sv driver.sv monitor.sv agent.sv scoreboard.sv \
• 7. env.sv test.sv \
• 8. instr_gen.o \ # Link the compiled C++ object file
• 9. +SHM_PACKED_LIMIT=115200 \
• 10. -access +rw \
• 11. -uvmnoccndnsextra -uvmhome $UVM_HOME \
• 12. design.sv testbench.sv
• 13.
```

With all the proper setup complete, the simulation is now ready to be executed and debugged. Running the testbench will integrate the C++ instruction generator with the UVM environment, allowing for the generation of random RISC-V instructions to the DUT. This setup ensures a robust testing framework where any issues can be efficiently debugged by leveraging the assembly string outputs, simulation logs, and UVM coverage metrics.

3.13 IC Design Flow and Challenges

An IC design is created in three stages: system, which defined specifications and requirements and the overarching architecture; logical, the development, simulation, and synthesis of RTL; and physical, the placement and routing which leads to the tapeout and manufacturing of the device. Commonly, the stages of system and logical are known as the “front-end” of the design, while the physical phase is known as the “back-end”.

The system stage is composed of the definition of the design specification and description of the microarchitecture; the logical stage consist of the writing and synthesizing the RTL for the design; and the physical stage consists of the place and route of the design. Each phase is divided in and implementation and verification parts, which occur parallel and simultaneously. Implementation translates the idea to hardware and logic, while verification ensures the functionality and design integrity. The main challenges for an ASIC design include size, complexity, power, performance, and choosing the adequate IP.

An ASIC design begins with the design specification. A design specification is a reference module that species interfaces, functional blocks, IP used, as well as design requirements. High-level decisions affect the system and the environment

it will operate on. The microarchitecture provides details for the internal components and its based on the block specification. Mid-level decisions create the microarchitecture of the internal blocks of the design. Register-Transfer-Level, or RTL, describes the digital circuit that forms the design. It's a module used to create the logic, interfaces, blocks, and simulations. RTL codifies the blocks of the microarchitecture. Low-level decisions affect the design of these RTL blocks.

The implementation is done via means of hardware description languages (HDL), which are used to model the behavior, structure, and timing of electronics circuits. It allows for design reuse (IP) and its independent of vendor and implementation technology. HDL can express a design in different layers of abstraction across the entire lifetime of the design project. The four predominant layers of abstraction of IC design are: behavioral, RTL, gate-level, and switch level. Engineers write the RTL by reasoning the behavioral synthesis of the specification. A synthesis tool performs logic synthesis on the coded RTL to generate a gate-level description of the design, also known as a netlist. Finally, automated tools, with direction from engineers, perform place and route to layout the design in terms of primitives provided by the underlying technology.

Verification must account for the size and complexity of the design, testbenches environments, exceptions, corner cases, and testing strategies adequate for the design. Some concerns for the implementation phase include: size and complexity, power, timing and design closures, and yield. The quality of the RTL is determined by its timing, area, and power. If the quality does not meet the specification after testing, synthesis constraints or parts of the RTL would need to be modified. Timing specifications must be met across all stages of implementation, and engineers can use several strategies, like static timing analysis, to validate timing specifications. Timing closure is the process of iterating through different possible implementations to find one that meets the timing requirements of the design using the actual placed routes. In any case, modification of the RTL should be last resort when it comes to solving timing problems.

Design for tests (DFT) are techniques used and components included in the design to make applying manufacturing tests easier. DFT is one of the requirements that affect the RTL. Particularly, engineers can decide to implement one of, or a combination of, built-in self tests (BIST) and joint-test action group (JTAG) for testing and debugging the manufactured design. For placement purposes, boundary-seam architectures (such as JTAG), allows to test the connectivity of the I/O pins without affecting the core logic of the device without impacting the core logic. This is useful for chain integrity testing.

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 (System Verilog)

The IEEE 1800-2017 standard, also referred to as the System Verilog standard, aims to provide engineers and developers with a set of unified methods for hardware design and verification. System Verilog 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 widespread adoption of the System Verilog 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 on Design

The System Verilog 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. System Verilog offers a powerful set of tools not only used to describe our hardware but also supports robust verification techniques. The capabilities that System Verilog 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, System Verilog 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 testbenches 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 re-usability 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 designs 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 of 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 scope of this project. However, as this project aims to teach our team the entire design process of ASICs, 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 though 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 finical 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, and 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 compressive 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 amount 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 due 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 learning 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 been tremendous advancements in generative AI in 2024, it is still far from perfect. There are cases in 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 include 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

This case study aimed to gather general knowledge about the various verification methods used in digital design. We tasked Chat GPT with explaining verification techniques, their significance, and their application in ensuring the reliability of

digital systems. We provided the following prompt to see what information we could learn from the model: “What are the different verification methods for digital systems?”.

The output generated by ChatGPT for this question can be found in **Appendix A**. ChatGPT provided a high-level overview of the different verification methodologies including simulations, formal verification, and emulation.

In this example provided by Chat GPT you must have enough knowledge prior to asking a question like such as this one, 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?”

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?”

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: “What is the difference between Von Neuman vs Harvard”

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 Hardware and Software Design

6.1 RAPID Core Design – Preface

This specification document was created to support the RTL design process and enhance coordination between the CPU and Memory teams. Initially, both teams faced challenges due to overlapping efforts on the same tasks, resulting in inconsistencies that impeded development. Following a group meeting involving all members of both teams, it was decided to develop this document to centralize design objectives and streamline collaboration.

The original goal for the RISC-V architecture implementation, referred to as the RAPID microarchitecture, was to achieve a straightforward and efficient design. However, the initial approach encountered significant challenges, which are discussed throughout this project. An earlier design attempt was abandoned after consultation with Northrop Grumman project advisors, who highlighted unconventional aspects of our approach and potential long-term issues. Two major concerns from these initial efforts—timing and synthesis—are addressed in this document.

During the early stages of the project, the team faced a critical decision: whether to use a pre-designed RTL core or to develop a custom core. While a pre-designed core could have reduced the project's scope, particularly in UVM verification, the team was drawn to the challenge of designing a core from scratch. This decision was motivated by a strong interest in RTL design, as many team members aspired to work in this field post-graduation.

Despite concerns about whether a custom core could be completed within the given timeframe—and the potential impact on UVM verification efforts—the team justified their choice by emphasizing the simplicity of the proposed design and its manageable verification requirements. Additionally, some advisors recommended incorporating pre-built designs to streamline the verification process, but the

team's passion for the project ultimately drove the decision to pursue the more challenging path of custom core development.

The first major challenge stemmed from synthesis, as misinformation led to the belief that behavioral RTL could not be used for this project. This misunderstanding increased complexity, requiring manual implementation of fundamental logic components such as adders, shifters, and comparators. While these elements are covered in VLSI design coursework, their manual implementation introduces intricate trade-offs in design complexity and performance. For example, a simple shift operation can vary in execution time from a single cycle to up to 32 cycles, depending on the hardware configuration.

The second issue arose from premature timing optimization. Concerns about timing stability prompted efforts to overcompensate by allocating excessive computation time early in the design process. However, this approach proved counterproductive. The team's inexperience made early timing optimization difficult, introducing timing bugs and race conditions during testing. These issues ultimately necessitated the development of the RAPID-X microarchitecture, which addressed the flaws of the initial design.

This specification documents the rationale behind key decisions, lessons learned, and the revised approach. It serves as a framework for achieving our engineering goals while highlighting the team's dedication to pursuing challenging yet rewarding design methodologies.

6.2 Synchronization Model

In this synchronization model, all pipeline stages progress simultaneously, simplifying both operations and timing coordination. However, this design limits the CPU's capabilities because all stages essentially stall until the pipeline is ready to continue execution. The performance impact of this architecture choice is limited for our case because RAPID Core has a streamlined ISA, which does not support complex instructions such as floating-point operations, integer division, or environmental control instructions. Exception handling is also omitted, which, while reducing debug flexibility, simplifies the instruction flow through the pipeline. This trade-off in performance is minimal, as the primary advantage of asynchronous execution typically applies to lengthy operations. In our design, any stage with a longer execution time stalls the entire pipeline. Fortunately, with the RI32V ISA's short instruction execution times, this effect is limited, and memory access is expected to be the primary bottleneck.

6.3 No Operation Instruction NOP

A no operation or NOP instruction is an instruction that does not alter the state of the CPU, for the implementation of the RTL, a NOP will be implemented in the hardware as the following: add x0, x0, x0. The x0 is hardwired to always return 0 so this instruction does not alter any state of the CPU.

6.4 Common Requirements

Each stage outputs a “done” flag, indicating it has reached a stable state or completed execution. When this occurs, the stage transitions to the “Wait for Pipeline” state and holds until it receives a “WAIT” signal from the top module. This “WAIT” signal is generated by a 5-bit AND gate that combines the “done” signals from all five pipeline stages. When the CPU is reset, all stages are changed to the state before the “WAIT” state and the internal states for each stage are configured for a “nop” instruction. This behavior allows all stages to simultaneously enter the “WAIT” state the subsequent clock cycle.

Each stage also outputs its current and next states to facilitate verification. When each stage receives the “RESET” signal, it should reset to the processing state to a “nop” instruction. The ability to view the current/next state in the simulation waveform proved to be an invaluable piece of information in the process of gaining insight of the actual execution of RTL.

6.5 Generic Cache

6.5.1 High-Level Description and Requirements

One of the goals for the design of the cache subsystem is to implement a generic cache implementation in RTL that is able to be parameterized for different sizes and data bus widths. The RTL code should be optimized such that the synthesis tool can infer block RAM for the underlying array of cache lines, but be versatile enough such that it can be switched for an optimized RAM library later during development.

The purpose of generalizing the cache is to be able to use the same cache implementations between both the instruction fetch and memory units. This way, both units would benefit from an improvement to the cache subsystem. Also, it makes it possible to switch and try different combinations to identify which one gives the better performance for each usage scenario.

The cache subsystem should be implemented as a SystemVerilog module such that it can be instantiated anywhere it is needed – such as inside the instruction fetch unit and the memory unit. The team tasked with designing the memory architecture has decided that the following requirements should be met:

- The cache subsystem must use a ready-valid protocol to communicate with the connected processor unit.
- The processor unit and cache subsystem exchange information in fixed sized words, the size of which is parameterized when the cache subsystem is instantiated.
- The cache subsystems and main memory exchange information one cache line at a time, the size of which is 16 bytes.
- The cache subsystem must accept a write mask that indicates which bits of the incoming word are used to overwrite bits in the cache line. (Required to retain bytes not affected in a store byte, or store half-word instruction).

6.5.2 The ready-valid protocol

The ready-valid protocol is a popular hardware protocol to send data between producer and consumer components used in ASIC or FGPA design. It serves as a coordination protocol to manage the timing of the data sent between two units. In addition to the data bus, two binary signals – ready and valid – are required to implement this protocol. Valid is asserted by the producer to send data, and ready is asserted by the consumer to indicate it is able to receive data. In its most basic form, examining the values of the signals at a particular clock edge allows the modules to determine whether data has been sent, as described by the following table:

Table 31: Truth Table for Read-Valid Protocol

Valid	Ready	Data successfully transferred?
0	0	No
0	1	No
1	0	No
1	1	Yes

In this case, the producer is the processor unit and the consumer is the cache subsystem. More specifically, the protocol used is the “ready-then-valid” variation for signal ordering in which the producer (processor unit) waits for the consumer (cache subsystem) to assert the ready signal, then places the address for the memory access (and write values, if applicable) on its respective bus and holds valid high. If and only if, during a particular clock edge both valid and ready are asserted then the processor unit knows that the cache has received the request.

For our implementation the protocol is extended with a reverse valid signal, which is asserted by the cache to indicate that the data requested during a read request is stable on the cache’s output port. The following table summarizes the most important aspects of the protocol.

Table 32: Partial Truth Table For Ready-Valid

Valid	Ready	Reverse-Valid	Comment
X	1	X	Cache is ready to receive the next request
0	X	1	Data in cache output port is valid.
X	0	X	Cache is busy.
1	1	X	The request has been received by the cache.

The signals from the processor unit to the cache (valid, address, write data, write mask, write enable) and from the cache to the processor unit (ready, reverse valid, data) are defined in a SystemVerilog interface to facilitate reusability of the code.

6.5.3 Microarchitecture and Finite State Machine

The current version of the cache implements a direct-mapped, write-back cache, with a size that is defined by a parameter during instantiation. The cache must keep track of valid and tag values for each line stored in it. To this end, the cache uses two inferred block RAM components: memory data (the cached lines), and metadata (tag and valid bits). The cache is implemented using a four cycle finite state machine with the following states:

- IDLE: cache is ready to receive next request (ready signal is asserted)
- COMPARE: metadata is accessed and compared for a match; on cache hit, the memory data is connected to the output port (reverse valid is asserted).
- ALLOCATE: a block is fetched from main memory and replaces (evicts) block from data memory.
- WRITE BACK: a dirty, about-to-be-evicted block is written to main memory.

The following is a block diagram of the FSM as well as the signals that might trigger a transition.

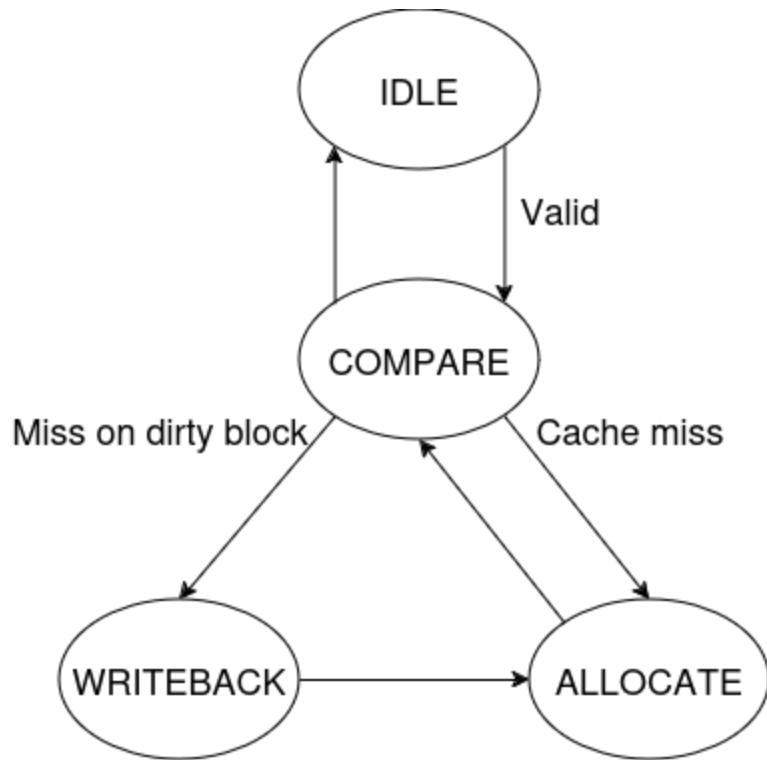


Figure 6.1: FSM For Cache Subsystem

The following is a high-level overview of the block diagram of the microarchitecture of the cache.

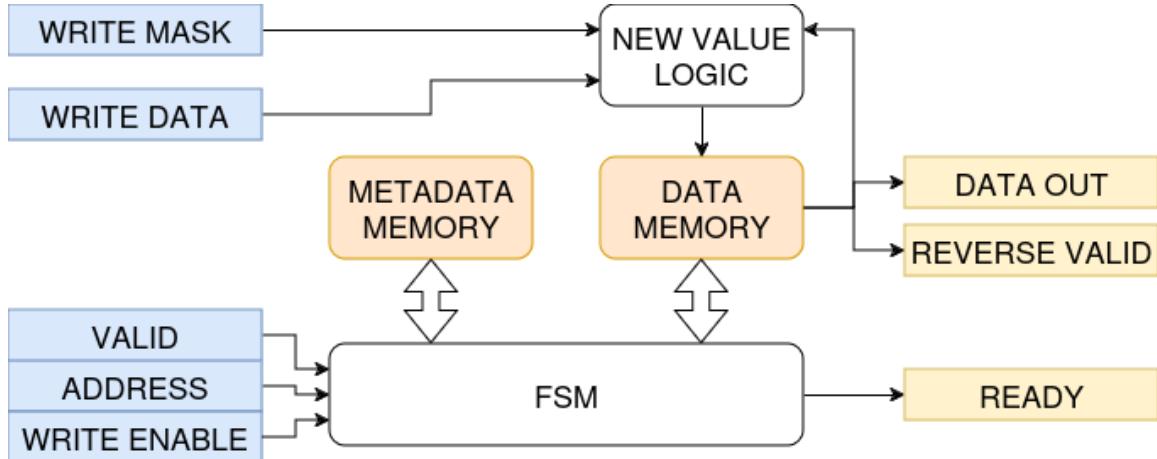


Figure 6.2: Block Diagram for Cache

6.5.4 Performance and Potential Improvements

The cache is an important part of the system as two instances of it are integrated in the instruction fetch and memory units of the processor. As it stands, the timing performance of the cache is acceptable for our purposes and the cache is able to provide data in one clock cycle on a cache hit. That is, if a valid read or write request occurs prior to clock edge N (and it's a cache hit) then the output data is stable by clock edge N+1, and the cache is ready for the next request by clock edge N+2. On a cache miss, the cache is allowed to take an indefinite amount of time to send and retrieve data from main memory.

Hit and miss rate of the cache remains to be evaluated by the team. In the case the team finds the miss rate is too high, the cache system is flexible enough to be revised to incorporate associativity and different cache replacement policies. Similarly, if the team finds the inferred block RAM by the synthesizer tool to be inadequate, then the open-source OpenRAM memory compiler can be used as per recommendation by Northrop. OpenRAM is a framework to create layouts, netlists, routing, placement, and power models for static RAM for ASIC design.

6.5.5 Cache Subsystem Verification

Because the cache subsystems are instantiated both in the Instruction Fetch unit and the Memory Access unit, it is very important to verify the behavior of the cache as any bugs will propagate up the module hierarchy and potentially make debugging and UVM diagnostics more difficult at the time the team tests the pipeline. A robust verification methodology for the cache is necessary to complement other verification methods and obtain adequate coverage metrics.

In particular, the cache subsystem is comprised of a finite state machine (FSM), and memory modules (either inferred block RAM as determined by the synthesis tool, or IP from a dedicated static RAM library like OpenRAM). In addition, the hierarchy of the cache is complex because it separates the metadata (such as the tag and valid bits) with the primary data (the actual cached values), however access to the primary data memory are determined by the validity derived

comparisons from comparisons of the metadata memory with the incoming requested memory address.

The verification of the cache subsystem is divided in four stages, the first one is to verify the functionality and transitions of the FSM for all possible stages and cache access outcome (cache hits, compulsory miss, capacity miss, and conflict miss). This verification is performed by carefully analysing the FSM at each clock cycle during the simulation and comparing its state transition with what is expected to happen by the testbench. Several bugs were encountered by this. One such example was finding a latent bug caused by incorrect usage of input and intermediate signals in the code were causing race conditions if the incoming signals were not stable. This resulted in a significant change of the cache design early on in the design phase. Now, the cache utilizes the ready-valid protocol to better synchronize timing with the units of the CPU pipeline, in addition, the cache latches the incoming requests to an internal register before processing it. This permits the cache to be less sensitive to timing violations.

The second cache subsystem verification stage is verification of the internal memories. As discussed previously, the cache maintains two memory blocks for the metadata and primary data. Naturally, it is important to tests that these memory implementation work as expected and that its timing meets the requirements to access the cache very fast - this point is of significant importance in our design because we only have one single level of cache. In general, the verification for the cache memories validates that the cache can perform reads in one clock cycle, and writes in a maximum of two clock cycles. Verification must be replicated across both the metadata and primary data memories, hence the use of SystemVerilog modules makes the code reusable and verification straightforward to implement.

The third element of verification for the cache subsystem is verifying the combinatorial and comparator logic. This is the part that is responsible for determining the cache output (hit or miss), and makes the decision of which block to evict from the active cache line. Due to its combinatorial behavior, this is the easiest part to verify as formal verification methods can be applied to check validate the logic takes the correct decision under all possible circumstances.

Finally, the fourth stage consists of verifying the communication between the cache subsystem and the top level memory. The interface between the cache and the memory is well defined through the memory controller intermediary, and simulation tools can be used to create a “fake memory” that records the transactions between it and the cache. This transaction log can later be examined to see if it corresponds to reads originating by the cache during a cache miss and writes during cache evictions.

The following waveform shows the cache subsystem in action, with the transition between states as the cache determines the outcome of the memory accesses and the communication signals between it and the top level main memory. This particular example shows the behavior of the cache the moment before a miss on-dirty block occurs and a block eviction to main memory occurs. The FSM of the cache can be seen transitioning from the COMPARE state (checking the metadata

and deciding the outcome of the request), to the WRITE_BACK state (evicting the block to main memory), and ALLOCATE (reading the new block from memory).

6.6 Instruction Fetch (IF) Stage

6.6.1 Objective Summary

To manage the program counter and interface with instruction cache block.

6.6.2 Requirements

1. Communicate with I-Cache interface
2. Allow for loading PC from external sources.
3. Handle RESET signal by:
 - a. Changing the current state to fetch and,
 - b. Adjusting the internal state to read from RESET vector.
4. Implement the RAPID synchronization model.
5. Halt execution upon encountering an unknown instruction.
6. Ability to halt execution through an external signal.

6.6.3 Block Diagram

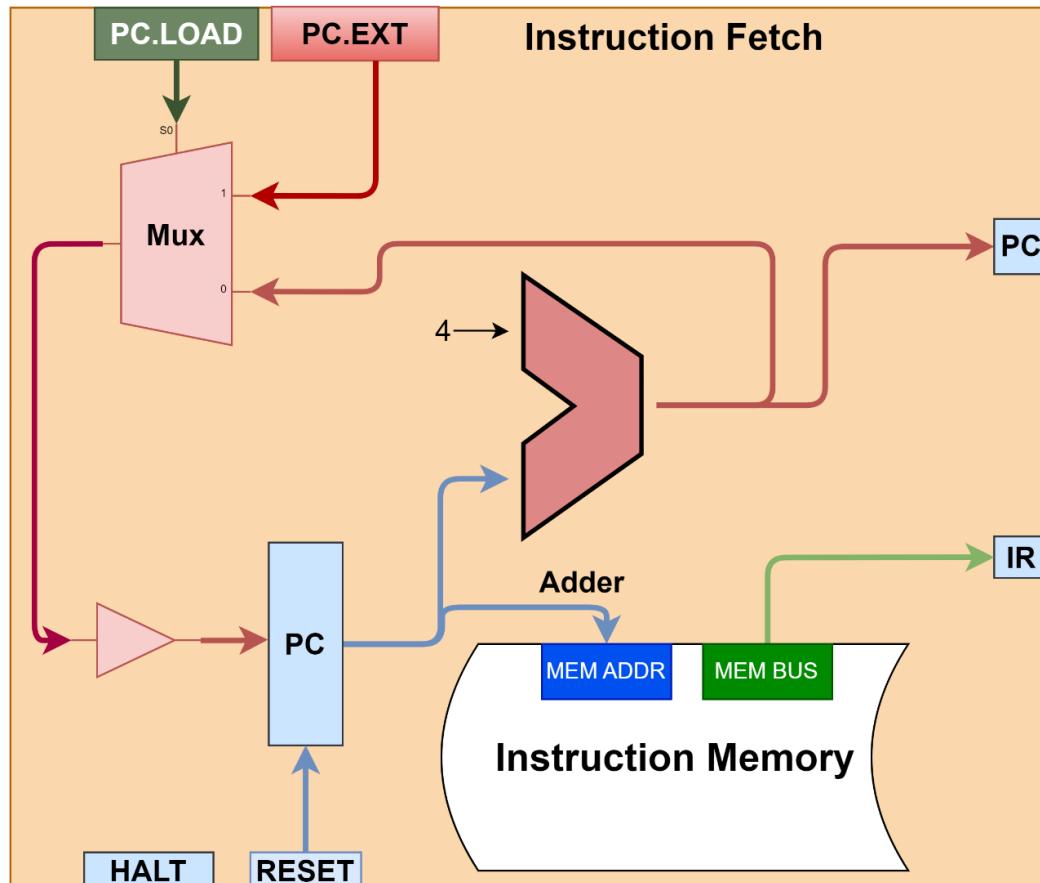


Figure 6.3: IF Unit Block Diagram

6.6.4 State Machine

There are 5 states in the instruction fetch stage, the table below summarizes the objective of each state.

Table 33: State Description Table

State Description Table		
State #	State	Description
1	Fetch Instruction	Updates the memory address based on PC value IF stage waits in this state until memory read is complete. The stage is responsible for updating the output port.
2	Wait for Pipeline	IF moves to this state only from state #1 if pipeline ready flag is 0 and remains in this state until flag changes to 1.
3	Choose next PC value	Chooses between PC <- PC+4 OR PC <- EXT_PC based on PC_LOAD flag and then moves to state #4. Set done flag to false.
4	Halt	This state is entered only by EXT_HALTI flag and the IF stage does not exit this stage unless RESET flag is high.
5	Reset	Reset the PC value to the reset vector value defined rapid_pkg and jump to IF_FETCH.

Table 34: State Transition Table

State Transition Table		
Current State	Next State	Condition
1 (FETCH)	1 (FETCH)	REST=0, HALT=0, MEM_READY = 0
1 (FETCH)	2 (WAIT FOR PIPELINE)	REST=0, HALT=0, MEM_READY =1, PIPELINE_READY=0
1 (FETCH)	3 (NEXT PC)	REST=0, HALT=0, MEM_READY =1, PIPELINE_READY=1
2 (WAIT)	2 (WAIT)	REST=0, HALT=0, PIPELINE_READY =0
2 (WAIT)	3 (NEXT PC)	REST=0, HALT=0, PIPELINE_READY =1
3 (NEXT PC)	1 (FETCH)	REST=0, HALT=0
4 (HALT)	4 (HALT)	REST=0
X	5 (RESET)	REST=1
5 (RESET)	1 (FETCH)	RESET=0

6.6.5 Internal Synchronization Flow

1. In "FETCH" once memory is loaded, signal "o_done" to high and move to "WAIT" stage.
2. Once "i_pipeline_ready" is high, set "o_done" to low and move to "NEXT" stage.

6.6.6 Internal RESET Configuration

Table 35: RESET Configuration

State Variable	Value
pc	RESET_VECTOR (defined in rapid_pkg)
next_state	IF_FETCH

6.6.7 State Diagram

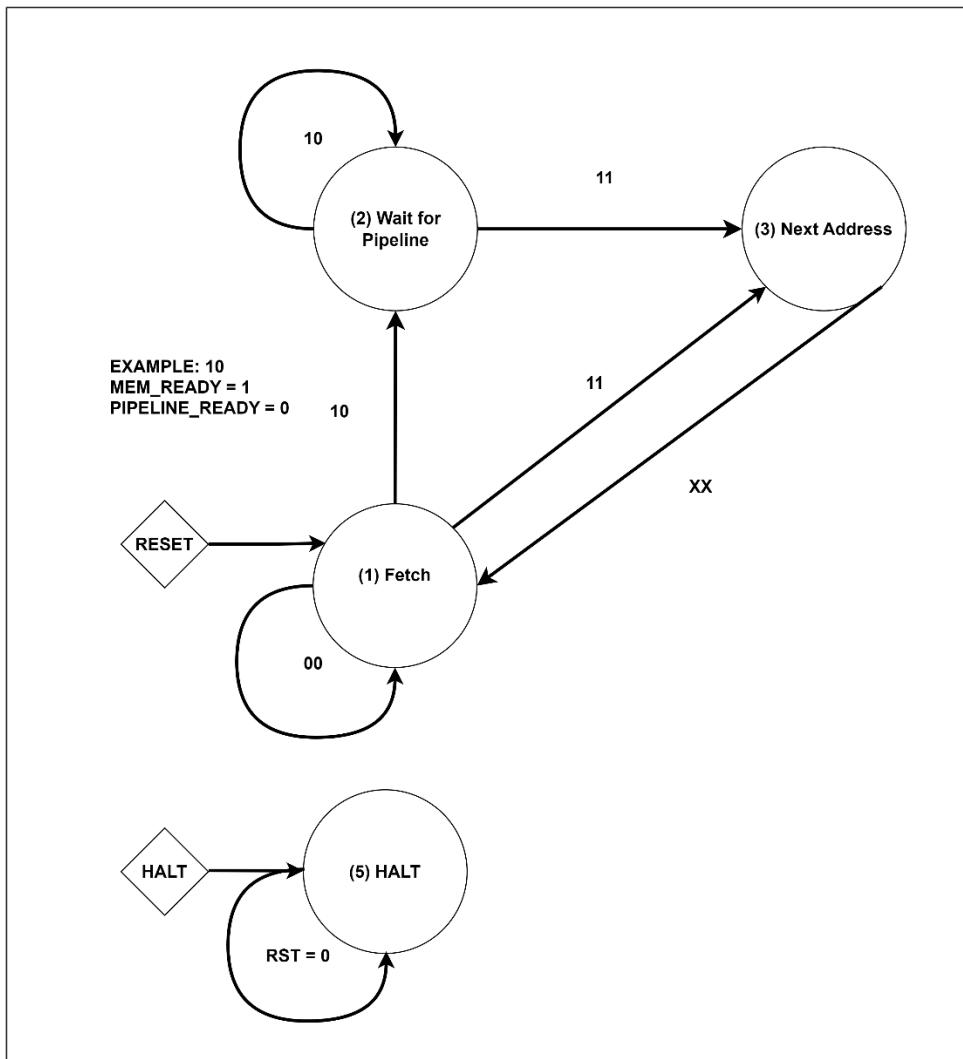


Figure 6.4: FSM for IF Unit

6.6.8 Design Module I/O Ports

Table 36: Instruction Fetch Module I/O Ports

Instruction Fetch Module I/O Ports		
Type	Port	Summary
input	i_clk [1-bit]	Clock signal
input	i_reset [1-bit]	Reset signal
input	i_pipeline_ready [1-bit]	Synchronization signal w/ other pipeline stages.
input	i_ext_pc_load [1-bit]	External flag to load i_ext_pc into pc.
input	i_ext_pc [32-bit]	External pc value used to branch pc.
output	o_pc [32-bit]	Output PC value (set when i_pipeline_ready = 1)
output	o_instruction [32-bit]	Output Instruction
output	o_done [1-bit]	Output flag to signal done.
output	o_current_state [3-bit]	Output current state, WAIT or DECODE.
output	o_next_state [3-bit]	Output next state, WAIT or DECODE.

6.7 Instruction Decoder (ID) Stage

6.7.1 Objective Summary

Decompose the instruction from the instruction fetch (IF) stage into control signals and sign extending the immediate bit field to a 32-bit. This stage is also responsible for properly reordering the bitfields for U, J, I, B, S, and R instruction types.

6.7.2 4.2 Requirements

1. The decoder must decode the 37 instructions in the RAPID microarchitecture which fall into the following 6 categories:
 - a. Load Upper Immediate (LUP)
 - b. Unconditional Branch (UB)
 - c. Conditional Branch (CB)
 - d. Memory Load/Store (MEM)
 - e. Immediate Arithmetic and Logic (IMM)
 - f. Register Arithmetic and Logic (REG)
2. Handle RESET signal by:
 - a. Changing the current state to execution and,
 - b. Adjusting the internal state to perform a “nop”.
3. Implement the RAPID synchronization model.

The control category is selected by a Finite Control Signal (FCS); however, some several instructions share the FCS signal because they have similar functionality but may be doing the exact opposite operation such as ADD and SUB instructions; for these types of instructions, the Inverse Operation (IOP) flag is used to select the correct operation.

The following is the grouping of instructions with the proper control category.

Table 37: Instruction Mapping to Control Category

Instruction Mapping to Control Category	
Category	RV32I Instructions
Load Upper Immediate (LUP=000)	LUI, AUPIC
Unconditional Branch (UB=001)	JAL, JALR
Conditional Branch (CB=010)	BEQ, BNE, BLT, BGE, BLTU, BGEU
Memory Load/Store (MEM=11)	LB, LH, LW, LBU, LHU, SB, SH, SW
Immediate Arithmetic and Logic (IMM=100)	ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI
Register Arithmetic and Logic (REG=101)	ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND

Table 38: Instruction Mapping to Control Signals

Instruction Mapping to Control Signals		Control Category	Finite Control Signals	Inverse Op	Contr ol Signal
Instruction	Opcode				
LUI	01101	LUP (LOAD UPP IMM)	000	1	LUP-000-1
AUIPC	00101	LUP (LOAD UPP IMM)	000	0	LUP-000-0
JAL	11011	UB (UNCOND.BRANCH)	000	0	UB - 000-0
JALR	11001	UB (UNCOND.BRANCH)	000	1	UB - 000-1
BEQ	11000	CB (COND.BRANCH)	000	0	CB - 000-0
BNE	11000	CB (COND.BRANCH)	001	0	CB - 001-0
BLT	11000	CB (COND.BRANCH)	100	0	CB - 100-0
BGE	11000	CB (COND.BRANCH)	101	0	CB - 101-0

BLTU	11000	CB (COND.BRANCH)	110	0	CB - 110-0
BGEU	11000	CB (COND.BRANCH)	111	0	CB - 111-0
LB	00000	MEM (MEM LOAD/STORE)	000	0	MEM- 000-0
LH	00000	MEM (MEM LOAD/STORE)	001	0	MEM- 001-0
LW	00000	MEM (MEM LOAD/STORE)	010	0	MEM- 010-0
LBU	00000	MEM (MEM LOAD/STORE)	100	0	MEM- 100-0
LHU	00000	MEM (MEM LOAD/STORE)	101	0	MEM- 101-0
SB	01000	MEM (MEM LOAD/STORE)	000	1	MEM- 000-1
SH	01000	MEM (MEM LOAD/STORE)	001	1	MEM- 001-1
SW	01000	MEM (MEM LOAD/STORE)	010	1	MEM- 010-1
ADDI	00100	IMM (ALU IMM)	000	0	IMM- 000-0
SLTI	00100	IMM (ALU IMM)	010	0	IMM- 010-0
SLTIU	00100	IMM (ALU IMM)	011	0	IMM- 011-0
XORI	00100	IMM (ALU IMM)	100	0	IMM- 100-0
ORI	00100	IMM (ALU IMM)	110	0	IMM- 110-0
ANDI	00100	IMM (ALU IMM)	111	0	IMM- 111-0
SLLI	00100	IMM (ALU IMM)	001	0	IMM- 001-0
SRLI	00100	IMM (ALU IMM)	101	0	IMM- 101-0
SRAI	00100	IMM (ALU IMM)	101	1	IMM- 101-1
ADD	01100	REG (ALU REG)	000	0	REG- 000-0
SUB	01100	REG (ALU REG)	000	1	REG- 000-1

SLL	01100	REG (ALU REG)	001	0	REG-001-0
SLT	01100	REG (ALU REG)	010	0	REG-010-0
SLTU	01100	REG (ALU REG)	011	0	REG-011-0
XOR	01100	REG (ALU REG)	100	0	REG-100-0
SRL	01100	REG (ALU REG)	101	0	REG-101-0
SRA	01100	REG (ALU REG)	101	1	REG-101-1
OR	01100	REG (ALU REG)	110	0	REG-110-0
AND	01100	REG (ALU REG)	111	0	REG-111-0

6.7.3 Block Diagram

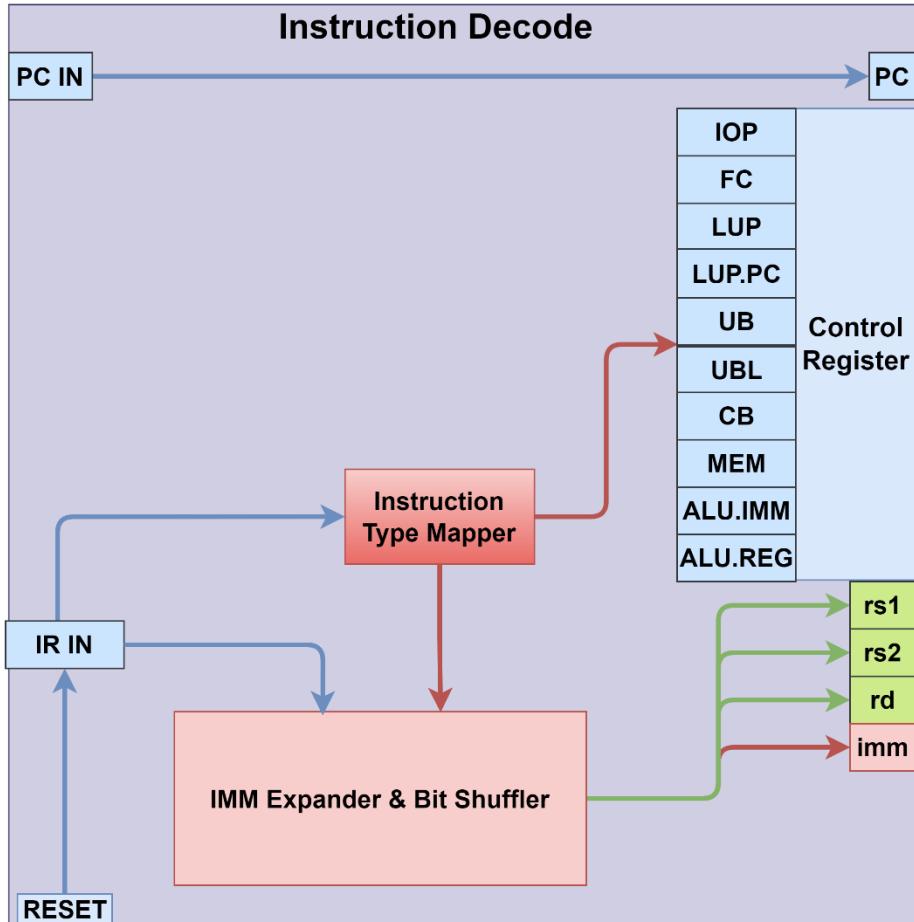


Figure 6.5: ID Unit Block Diagram

6.8 State Machine

There are 2 states in the instruction decode (ID) stage, the table below summarizes the objective of each state.

Table 39: State Description Table

State Description Table		
State #	State	Description
1	Wait for Pipeline	ID moves to this state only from state #1 if pipeline ready flag is 0 and remains in this state until flag changes to 1. When ID moves to this state, it also updates the local state from the input ports.
2	Decode Instruction	Load instruction from input port and parses the instruction to the desired control signals. This stage also updates the output ports in a always_comb block.

3	Reset	Resets the instruction decoder by configuring the internal state to a “nop” instruction and jumping to decode instruction.
---	-------	--

Table 40: State Transition Table

State Transition Table		
Current State	Next State	Condition
1 (WAIT)	1 (WAIT)	P=0
1 (WAIT)	2 (DECODE)	P=1
2 (DECODE)	1 (WAIT)	N/A
X	3 (RESET)	RESET=1
3 (RESET)	2 (DECODE)	N/A

6.9 Internal Synchronization Flow

1. The “WAIT” stage will set “o_done” to low when moving to “DECODE” stage.
2. The “DECODE” stage will set “o_done” to high when moving to “WAIT” stage.
3. The WAIT stage will load data from input ports if “o_done” is high.

6.9.1 Internal RESET Configuration

The RESET state will change the following internal variables to the specified values.

Table 41: Internal RESET Configuration

State Variable	Value
pc (program counter)	RESET_VECTOR (defined rapid_pkg)
instruction	NOP_INSTRUCTION (defined rapid_pkg)
Next_state	DE_DECODE

6.9.2 State Diagram

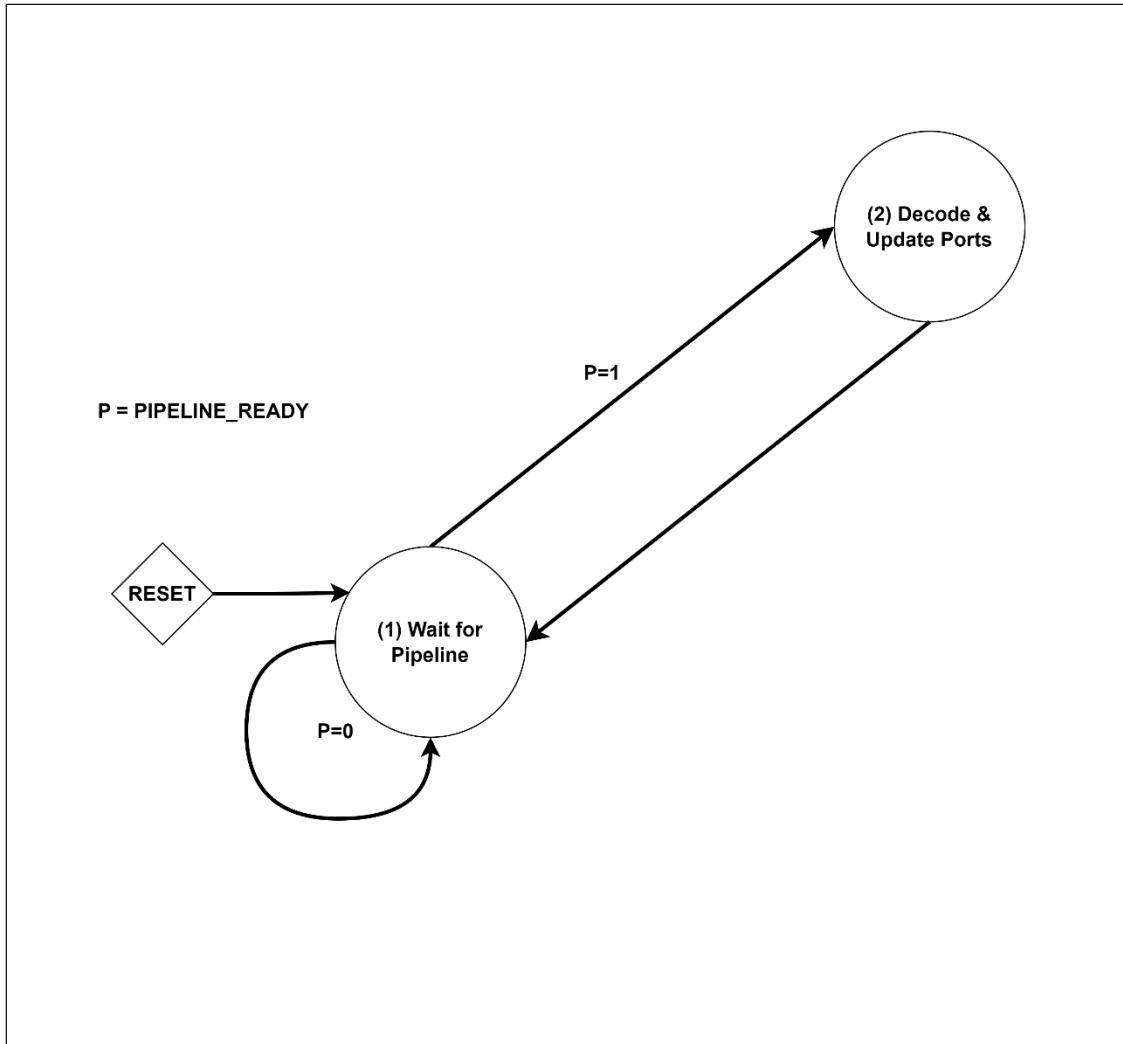


Figure 6.6: FSM for the ID

6.9.3 Design Module I/O Ports

Table 42: Instruction Decoder Module I/O Ports

Instruction Decoder Module I/O Ports		
Type	Port	Summary
input	i_clk [1bit]	Clock signal
input	i_reset [1-bit]	Reset signal
input	i_pipeline_ready [1-bit]	Synchronization signal w/ other pipeline stages.

input	i_pc [32-bit]	External flag to load i_ext_pc into pc.
input	i_instruction [32-bit]	Input instruction from IF stage.
output	o_pc [32-bit]	Output PC value (set when i_pipeline_ready = 1)
output	o_control_signal	Struct that contains the Control Category, FCS, and IOP signals.
output	o_rs1 [5-bit]	Register index for input register #1.
output	o_rs2 [5-bit]	Register index for input register #2.
output	o_rd [5-bit]	Register index for destination register.
output	o_imm [32-bit]	Sign extended value from instruction.
output	o_done [1-bit]	Output flag to signal done.
output	o_current_state [2-bit]	Output current state, FETCH, WAIT, NEXT, HALT.
output	o_next_state [2-bit]	Output next state, FETCH, WAIT, NEXT, HALT.

6.9.4 4.6.1 Control Signal Structure

- CC – Control Category, 3-bit wide
- FCS – Finite Control Signal, 3-bit wide
- IOP – Inverse Operation, 1-bit wide

6.10 Execute (EX) Stage

6.10.1 Objective Summary

Performs logical and arithmetic computations including branching for all 37 instructions. For the memory load or store operations, it computes the read/write address for the subsequent memory stage.

6.10.2 Requirements

1. Given a decoded instruction via a control signal structure:
 - a. Compute memory addresses for all memory operations.
 - b. Perform register and immediate level arithmetic and logical instructions.
 - c. Perform conditional and unconditional branching.
2. Handle RESET signal by:
 - a. Changing the current state to execution and,
 - b. Adjusting the internal state to perform a “nop”.
3. Implement the RAPID synchronization model.

6.10.3 Block Diagram

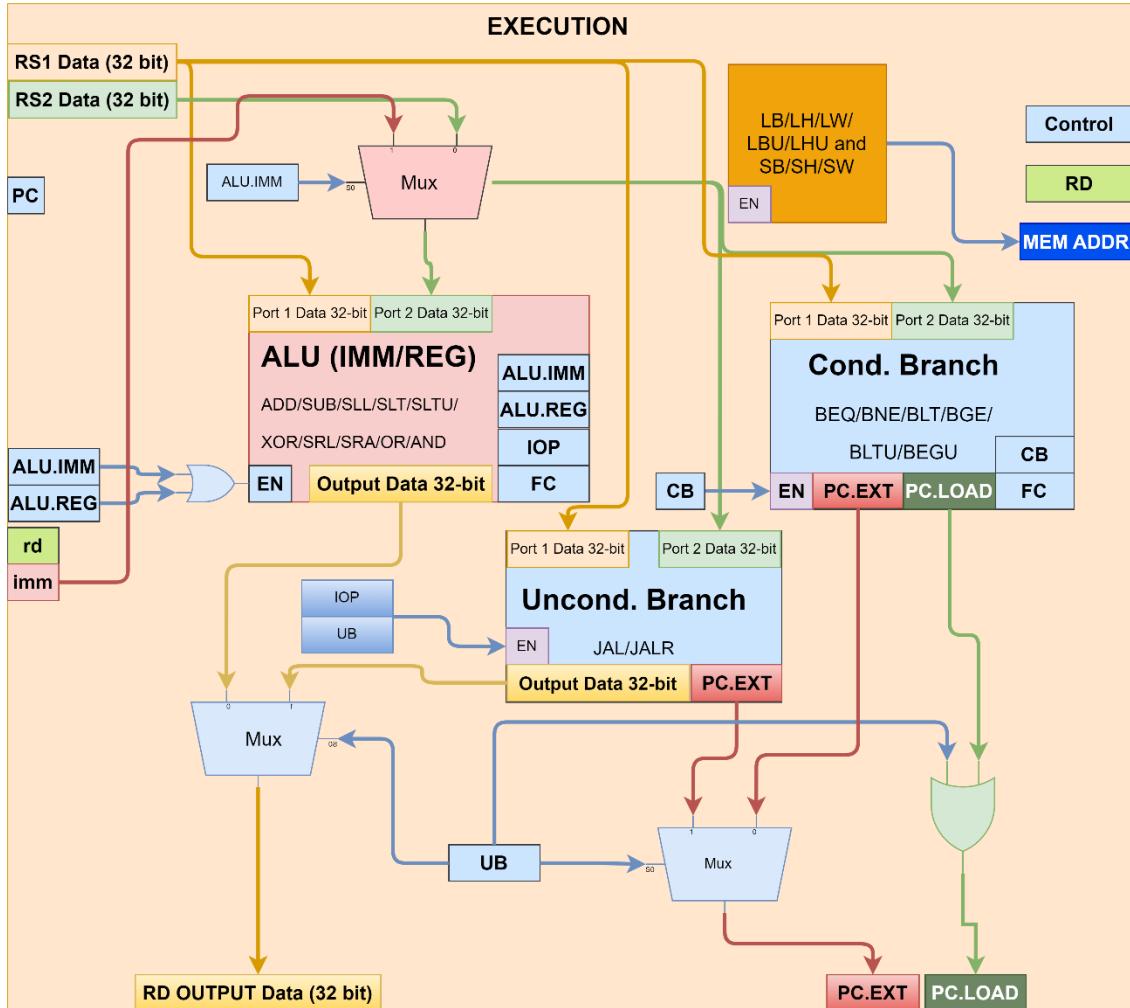


Figure 6.7: Block Diagram for Execute Stage

6.10.4 State Machine

There are 2 states in the execute (EX) stage, the table below summarizes the objective of each state.

Table 43: State Description Table

State Description Table		
State #	State	Description
1	Wait for Pipeline	EX moves to this state only from state #1 if pipeline ready flag is 0 and remains in this

		state until flag changes to 1. When EX moves to this state, it also updates the local state from the input ports.
2	Read Register	This stage resolves a race condition in the architecture.
3	Execute Instruction	Performs computation or branching based on the control signal structure and updates output ports. Note, the update happens in an always_comb block.
4	Reset	Configures internal state to perform a “nop” and jumps to Execute Instruction.

Table 44: State Transition Table for Execute Stage

State Transition Table		
Current State	Next State	Condition
1 (WAIT)	1 (WAIT)	P=0
1 (WAIT)	2 (REGISTER)	P=1
2 (REGISTER)	3 (EXECUTE)	N/A
3 (EXECUTE)	1 (WAIT)	N/A
X	4 (RESET)	RESET=1
4 (RESET)	3 (EXECUTE)	N/A

6.10.5 Internal Synchronization flow

1. The “WAIT” stage will set “o_done” to low when moving to “EXECUTE” stage.
2. The “EXECUTE” stage will set “o_done” to high when moving to “WAIT” stage.
3. The WAIT stage will load data from input ports if “o_done” is high.

6.10.6 Internal RESET Configuration

Table 45: Internal REST Config for Execute Stage

State Variable	Value
control_s	load_upper_imm: '0, uncond_branch: '0, cond_branch: '0, mem: '0, alu_imm: '0, alu_reg: '0, iop: '0, rs1_out: '0, rs2_out: '0,

	fcs_opcode: '0, rs1: '0, rs2: '0, rd: '0
next_state	EX_EXECUTE

6.10.7 State Diagram

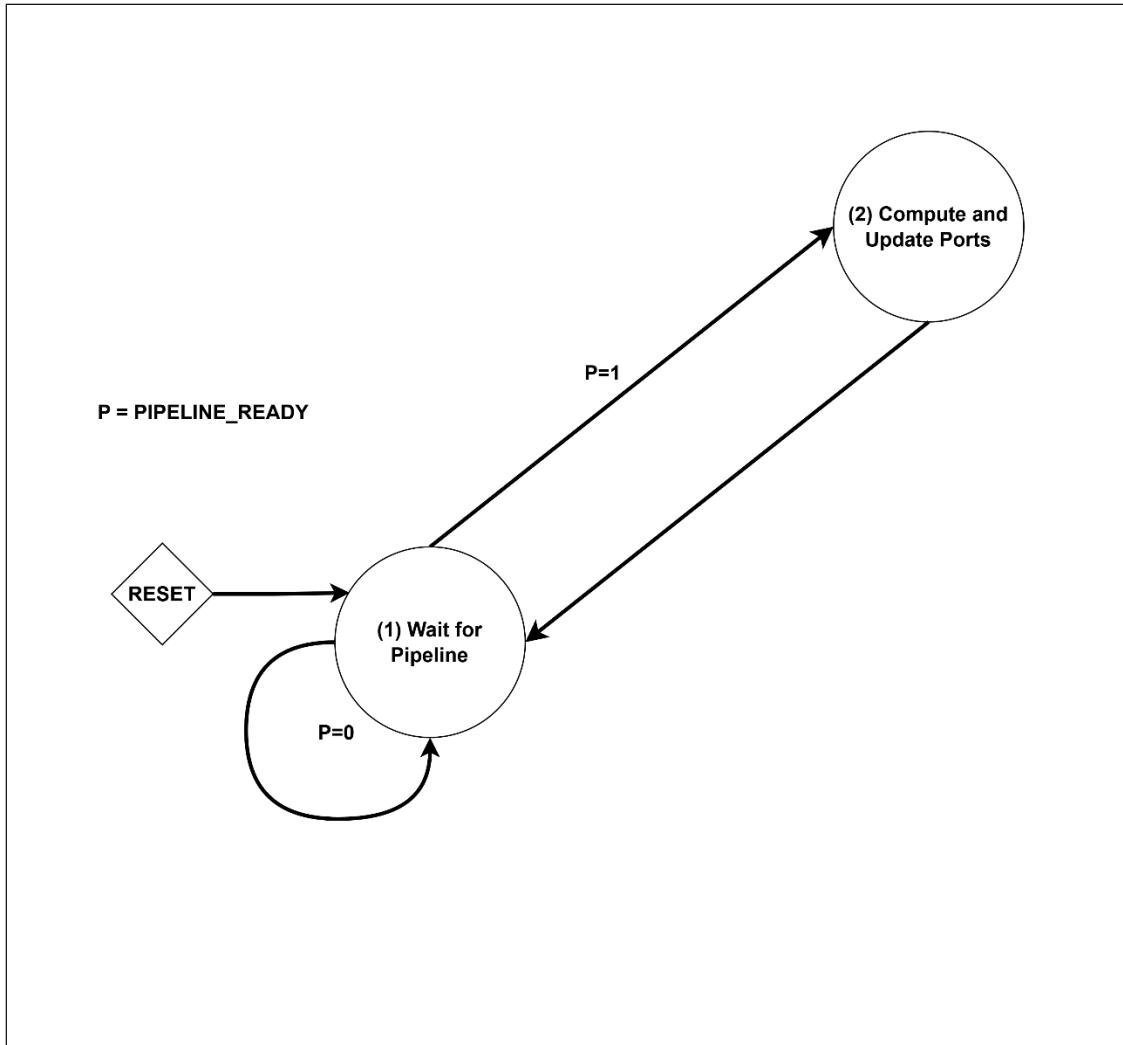


Figure 6.8: FSM for Execute Stage

6.10.8 Design Module I/O Ports

Table 46: Execute Module I/O Ports

Execute Module I/O Ports		
Type	Port	Summary
input	i_clk [1bit]	Clock signal
input	i_reset [1-bit]	Reset signal

input	i_pipeline_ready [1-bit]	Synchronization signal w/ other pipeline stages.
input	i_pc [32-bit]	External flag to load i_ext_pc into pc.
input	i_control_sig [struct]	Decoded control signals from instruction.
input	i_rs1 [32-bit]	Rs1 register value from register file.
input	i_rs2 [32-bit]	Rs2 register value from register file.
input	i_imm [32-bit]	Sign extended immediate value from instructoin.
output	o_control_signal [struct]	Control signal from input.
output	o_pc_ext [32-bit]	Computed program counter value to jump to.
output	o_pc_load [1-bit]	Branch flag for IF stage.
output	o_rd [5-bit]	Register index for destination register.
output	o_rd_output [32-bit]	Value to write to destination register, this value is also used to specify memory address for load/store operation. The meaning depends on the control signal.
output	o_done [1-bit]	Output flag to signal done.
output	o_current_state [EX_stage_t]	
output	o_next_state [EX_stage_t]	

6.11 Memory Stage (MEM)

The purpose of this stage is to connect to the memory interface (and indirectly the data cache “D-Cache”) to perform memory loads and memory writes. Specifically, for memory loads, the result will be saved to the register file in the subsequent pipeline stage: Write Back Stage (WB).

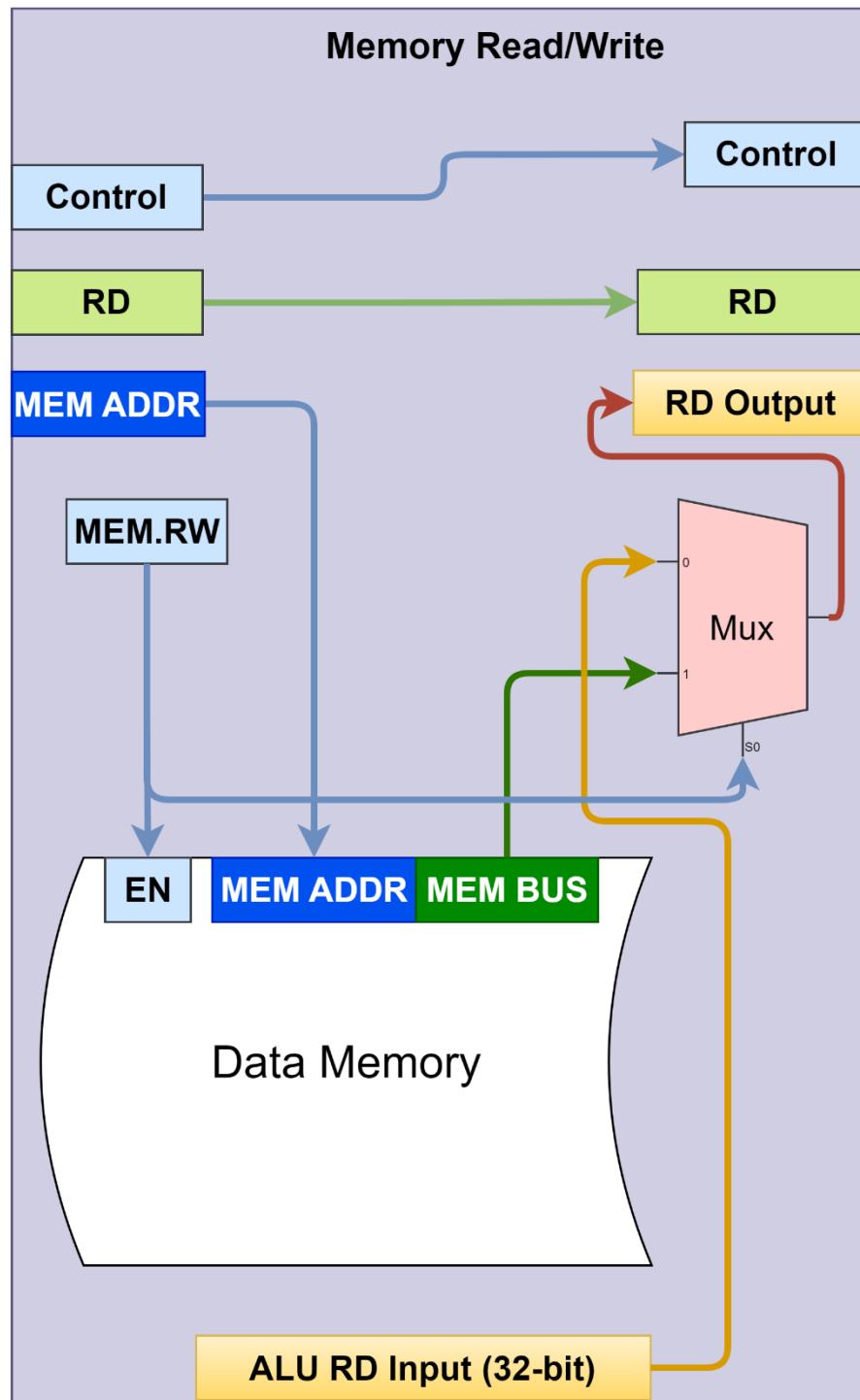
6.11.1 Requirements

1. Given an input memory address (and corresponding control signals), communicate with memory interface to:
 - a. Load data from RAM
 - i. Followed by a register write in the WB stage.
 - b. Store data into RAM (data will be provided from the EX-stage).
2. Handle RESET signal by:
 - a. Changing the current state to execution and,
 - b. Adjusting the internal state to perform a “nop”.
3. Implement the RAPID synchronization model.

Output current and next state

6.11.2 Block Diagram

Table 47: Block Diagram for MEM stage



6.11.3 State Machine

Table 48: State Description Table for MEM Stage

State Description Table		
State #	State	Description
1	Wait for Pipeline	<p>Waits for pipeline before loading next instruction, once P=1, updates the current state with the next instruction (set “o_done”=0).</p> <ol style="list-style-type: none"> 1. If the current instruction is a not memory read or write, then just forward the current instruction to WB stage. 2. If the operation is memory read, then jump to READ state. 3. If operation is memory write, then jump to WRITE state.
2	Memory Read	Updates the memory address and R/W flag of memory interface and waits in this state until read is complete, afterwards, it jumps to “WAIT” stage and sets “o_done”=1.
3	Memory Write	Updates the memory address and R/W flag of memory interface and waits in this state until write is complete, afterwards, it jumps to “WAIT” stage and sets “o_done”=1.

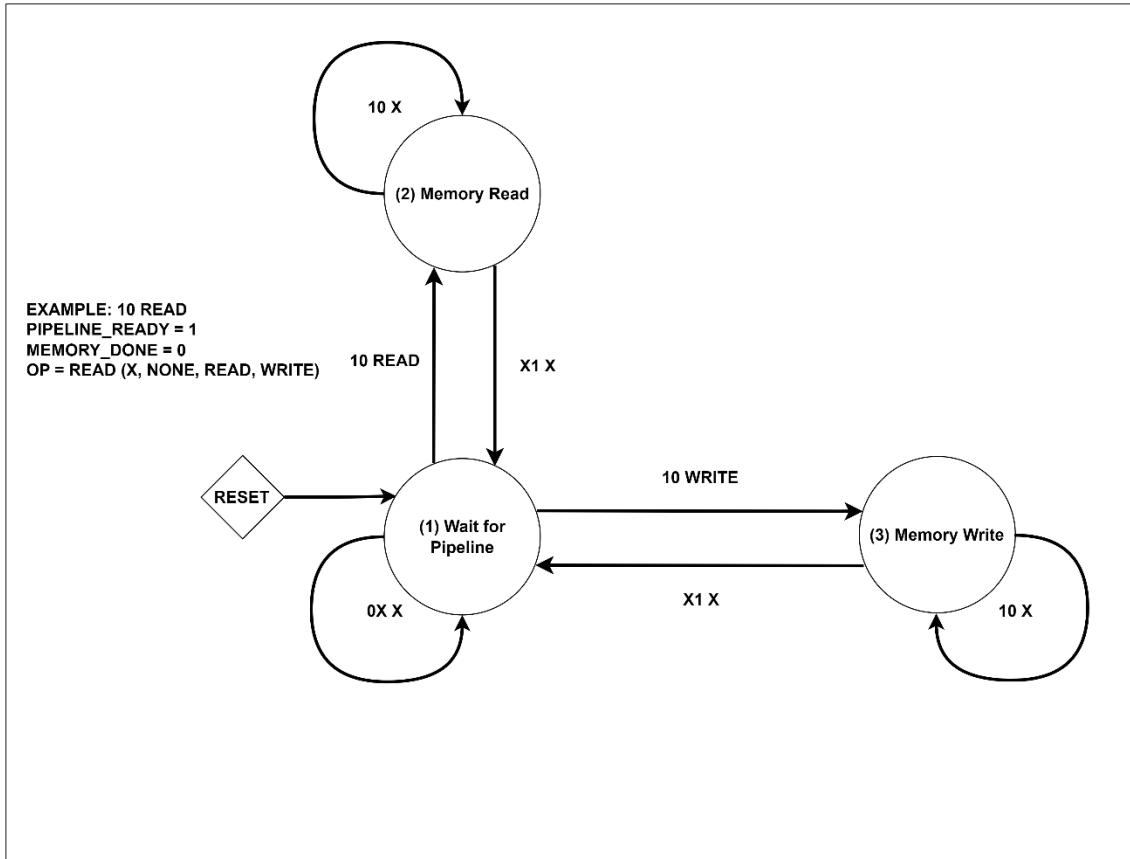
Table 49: State Transition Table

State Transition Table		
Current State	Next State	Condition
1 (WAIT)	1 (WAIT)	P=0, OP=X
1 (WAIT)	1 (WAIT)	P=X, OP=None
1 (WAIT)	2 (READ)	P=1, OP=READ
1 (WAIT)	3 (WRITE)	P=1, OP=WRITE
2 (READ)	2 (READ)	MEM_DONE=0
2 (READ)	1 (WAIT)	MEM_DONE=1
3 (WRITE)	3 (WRITE)	MEM_DONE=0
3 (WRITE)	1 (WAIT)	MEM_DONE=1

6.11.4 Internal Synchronization flow

6.11.5 State Diagram

Table 50: FSM For MEM Stage



6.11.6 Design Module I/O Ports

Table 51: Memory Module I/O Ports for MEM stage

Memory Module I/O Ports		
Type	Port	Summary
input	i_clk [1bit]	Clock signal
input	i_reset [1-bit]	Reset signal
input	i_pipeline_ready [1-bit]	Synchronization signal w/ other pipeline stages.
input	i_pc [32-bit]	External flag to load i_ext_pc into pc.
input	i_control_sig [struct]	Decoded control signals from instruction.
input	i_data_in [32-bit]	Either the computed data from EX stage or memory

		address for memory I/O operation.
output	o_control_signal [struct]	Control signal from input.
output	o_rd_output [32-bit]	Value to write to destination register, this value is also used to specify memory address for load/store operation. The meaning depends on the control signal.
output	o_done [1-bit]	Output flag to signal done.
output	o_current_state [MEM_state_t]	
output	o_next_state [MEM_state_t]	

6.12 Writeback Stage (WB)

6.12.1 Objective Summary

The only stage to perform commits writes to the register file.

6.12.2 Requirements

1. Manage memory rights with register file.
2. Handle RESET signal by:
 - a. Changing the current state to execution and,
 - b. Adjusting the internal state to perform a “nop”.
3. Implement the RAPID synchronization model.
4. Output current and next state.

6.12.3 Block Diagram

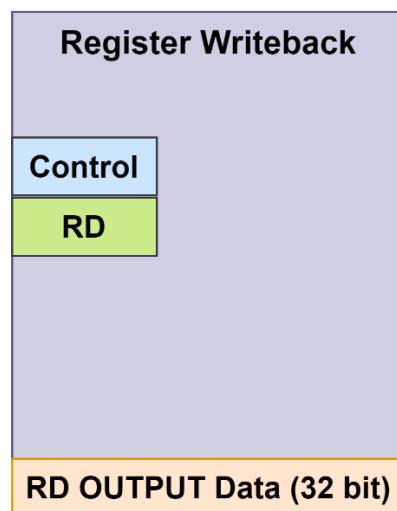


Figure 6.9: Block Diagram for the WB stage

6.12.4 State Machine

Table 52: State Description Table for WB Stage

State Description Table		
State #	State	Description
1	Wait for Pipeline	Waits for pipeline before loading data to the designated register in the register file.
2	Register Write	Updates the register address and waits in this state until the register file is complete, afterwards, it jumps to “WAIT” stage and sets “o_done”=1.

Table 53: State Transition Table

State Transition Table		
Current State	Next State	Condition
1 (WAIT)	1 (WAIT)	P=0
1 (WAIT)	2 (WRITE)	P=1
2 (WRITE)	1 (WAIT)	P = X

6.12.5 State Diagram

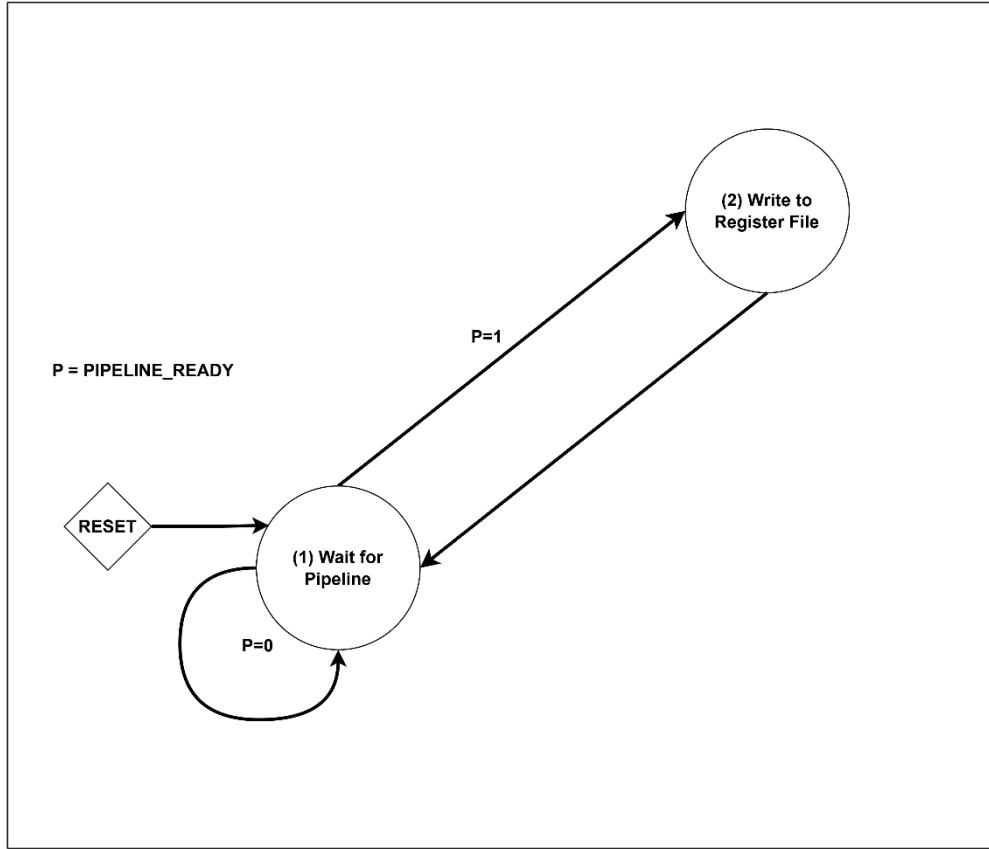


Figure 6.10: FSM for the WB Stage

6.12.6 Design Module I/O Ports

Table 54: Write back Module I/O Ports for WB Stage

Write back Module I/O Ports		
Type	Port	Summary
input	i_clk [1bit]	Clock signal
input	i_reset [1-bit]	Reset signal
input	i_pipeline_ready [1-bit]	Synchronization signal w/ other pipeline stages.
input	i_control_signal [control_s]	Decoded control signals from instruction.
input	i_data_in [32-bit]	Either the computed data from EX stage or memory address for memory I/O operation.
output	o_control_signal [control_s]	Control signal from input.
output	o_rd_output [32-bit]	Value to write to destination register.

output	<code>o_rd [5-bit]</code>	The register index to write to. If current instruction does not write to the register file then the register index value is 0.
output	<code>o_done [1-bit]</code>	Output flag to signal done.
output	<code>o_current_state [WB_state_t]</code>	
output	<code>o_next_state [WB_state_t]</code>	

6.13 Hazard and Forwarding Unit

The hazard and forwarding units were not implemented in this version of the architecture. Instead, the team adopted a software-based approach to manage data and control hazards. A custom RV32 assembler was developed by the CPU team to mitigate these issues programmatically. This section outlines the types of hazards encountered and the strategies employed to address them.

The approaches outlined in this section represent a compromise between project complexity and performance. By relying on the assembler to address hazards, the team avoided the additional hardware complexity associated with forwarding units and advanced branch handling. However, this came at the cost of reduced efficiency and increased CPI. These trade-offs highlight the challenges of balancing simplicity and performance in architectural design and underscore the importance of continuous improvement in future revisions.

6.14 Data Hazard

Data hazards occur when the pipeline attempts to read a register value that has not yet been written back due to the sequential flow of the pipeline. In this architecture, register reads are performed during the transition between the decode and execute stages, while register writes occur at the end of the pipeline in the writeback stage. This creates a dependency issue when two or more consecutive instructions operate on the same register, as the result of the earlier instruction is not available until it completes the writeback stage.

A common hardware solution to data hazards involves implementing a forwarding unit, which dynamically routes the latest value of a register from later stages of the pipeline (such as memory or writeback) to earlier stages (such as execute). The forwarding unit is designed to ensure that the correct and "newest" value of a register is used by the subsequent instruction, effectively bypassing the delay introduced by the sequential pipeline flow. This requires the forwarding unit to identify potential hazards, resolve dependencies, and route the correct value. In cases where multiple consecutive instructions target the same register, the forwarding unit must select the correct value based on instruction order.

While effective, implementing a forwarding unit adds significant complexity to the pipeline design. In this architecture revision, the forwarding unit was omitted due to concerns about this complexity, and a software-based solution was employed instead.

The CPU team's custom assembler addresses data hazards by automatically inserting **NOP** (no-operation) instructions into the user's code. These NOP instructions introduce delays between dependent instructions, ensuring that the result of the earlier instruction is written to the register file before the subsequent instruction attempts to read it. Specifically, the assembler inserts two NOP instructions whenever a data hazard is detected. This delay of two clock cycles ensures that the pipeline has sufficient time to propagate the correct register values.

While effective in resolving data hazards, the assembler-based solution introduces two significant drawbacks:

1. **Increased Code Size:** The insertion of NOP instructions increases the overall size of the program. This inflated code size reduces the cache hit ratio, as the data becomes more dispersed in RAM. Consequently, the CPU spends more time waiting for data to be fetched from memory, which further impacts performance.
2. **Higher CPI (Cycles Per Instruction):** The additional NOP instructions increase the average number of clock cycles required per instruction, thereby reducing overall throughput.

These drawbacks have a cascading effect on performance metrics. For example, the increased CPI makes it more difficult to meet the 100 MIPs performance requirement for the project. Achieving this target would require raising the clock frequency, which introduces additional challenges, such as higher power consumption and tighter timing constraints.

Some optimizations can mitigate these drawbacks. For instance, users can manually optimize their code to reduce the occurrence of data hazards, and the assembler or compiler can employ instruction reordering to minimize the impact of dependencies. Instruction reordering is a widely used technique in modern CPUs and compilers to improve performance by arranging instructions in a way that minimizes pipeline stalls and maximizes parallelism. These optimizations, while helpful, cannot fully eliminate the performance penalties associated with the lack of a forwarding unit.

6.15 Control Hazard

Control hazards arise from branch instructions, whether conditional or unconditional, that disrupt the normal flow of the instruction pipeline. In this architecture, the execute stage determines both the branch condition and the target address. However, by the time the branch instruction is processed in the

execute stage, the fetch and decode stages may have already loaded subsequent instructions that are no longer valid. This requires flushing the pipeline to prevent the execution of incorrect instructions.

To address control hazards, the CPU team's assembler inserts two NOP instructions immediately after each branch instruction, regardless of whether it is conditional or unconditional. These NOPs effectively flush the fetch and decode stages of the pipeline, ensuring that no invalid instructions are executed.

Although this approach resolves control hazards, it introduces similar drawbacks to those encountered with data hazard solutions:

1. **Pipeline Stalling:** The insertion of NOP instructions delays subsequent instruction execution, which reduces overall pipeline efficiency.
2. **Increased CPI:** The additional NOPs increase the average number of cycles per instruction, further compounding the challenge of meeting the project's performance targets.

The increased CPI caused by control hazards exacerbates the same challenges faced in addressing data hazards. The reliance on NOP instructions significantly impacts throughput, making it more difficult to meet performance goals such as the 100 MIPs requirement.

While this assembler-based solution was chosen to simplify hardware design for this revision, more advanced techniques could be explored in future iterations to address control hazards more efficiently. For instance, branch prediction or speculative execution could be implemented to minimize the need for pipeline flushing. These techniques are widely used in modern CPUs to improve performance by speculatively executing instructions beyond a branch point, reducing the impact of control hazards.

6.16 IC Power Planning

Another important aspect to the design as the project moves its final phase is power planning. In short, power planning is creating the global power network of the device. It seeks to minimize IR drop (drops in voltages across the network) and electromigration (mechanical failures due to wires being too close together). Inaccurate power distribution can lead to complete chip failure. Adequate connections to ground and power supply using enough number of power pads, power rings, trunks, and grids are necessary when designing the power network. Sometimes, multiple supply voltages or domains can also be used to reduce power consumption.

Power consumption in CMOS semiconductors is characterized as either static power or dynamic power. Static power is the leakage current of the underlying electronics. The design engineers can not address that static current in RTL. However, dynamic power can be reduced by minimizing switching activity through clock-gating. Clock gating prunes the clock tree to “turn-off” the clock network to a

section of the circuitry and has to be coded in the RTL. To implement a gated clock, the RTL must describe a circuit equivalent to the diagram below:

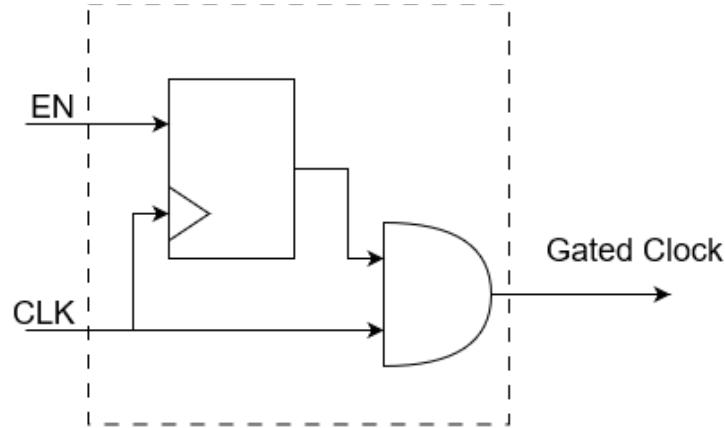


Figure 6.11: Schematic of a gated clock

Because clock gating is described as part of the RTL, it makes it highly error prone and makes verification more difficult. Because of this, formal verification is often needed to validate the gated clock does not change the functionality of the design thorough equivalence checking methods (LEC). Further power optimization should be tested by low-power simulation, which mimics behavior the target technology in a reduced power environment to ensure correct functionality of port isolation, state retention, and state loss.

6.17 Tool Chain

Because our ASIC is developed using the System Verilog standard there are many software suites which can simulate, synthesize, and implement the hardware description language. Cadence, Synopsys, and Xilinx all have their own programs but for this project we chose to utilize the Cadence toolchain which includes Xcelium for simulation and Genus for synthesis. However we ran into issues when using the VLSI servers provided by UCF, here those toolchain issues will be highlighted along with the solutions.

Xcelium is a useful tool for running testbenches as it can run assertions and all the other useful implementations in the System Verilog standard. Shown in the figure below is an example of a simulation testbench for the decoder module. This module tests the accuracy of the decoder state, and we check that the output instructions are as expected based on our input instructions. After the testbench compiled and was built the Xcelium displays the result of our test. In this case the testbench was successful. To access our Cadence tools we used SSH to have direct access to the server terminal. Throughout the term this SSH method has been relatively reliable, however there have been a few instances where simulations could not be run due to the server being down or the UCF VPN not being available at the moment.

```

[nl581013@vlsi CadenceTB] cat tb_decoder_top.f
rapid_pkg.sv
decoder_state_tb.sv
decoder_state.sv
[nl581013@vlsi CadenceTB] xrun -f tb_decoder_top.f
TOOL: xrun 23.09-s006: Started on Nov 25, 2024 at 20:40:58 EST
xrun: 23.09-s006: (c) Copyright 1995-2024 Cadence Design Systems, Inc.
Recompiling... reason: file './decoder_state_tb.sv' is newer than expected.
    expected: Mon Nov 25 20:39:52 2024
    actual: Mon Nov 25 20:40:21 2024
file: decoder_state_tb.sv
    module worklib.decoder_state_tb;
        errors: 0, warnings: 0
        Caching library 'worklib' ..... Done
    Elaborating the design hierarchy:
    Top level design units:
        rapid_pkg
        $unit_0x84de3129
        decoder_state_tb
        .i_pc_load();
    xmelab: *W,CUVMW ./decoder_state_tb.sv,20|12): port sizes differ in port connection(32/1) for the instance(decoder_state_tb).
    Building instance overlay tables: ..... Done
    Generating native compiled code:
        worklib.decoder_state_tb:sv <0x45f3c480>
            streams: 5, words: 4885
    Building instance specific data structures.
    Loading native compiled code: ..... Done
    Design hierarchy summary:
        Instances Unique
        Modules: 2 2
        Verilog packages: 1 1
        Registers: 16 16
        Scalar wires: 3 -
        Vectored wires: 8 -
        Always blocks: 1 1
        Initial blocks: 1 1
        Cont. assignments: 2 -
        Pseudo assignments: 3 -
        Assertions: 2 2
        Compilation units: 1 1
        Process Clocks: 1 1
        Simulation timescale: 1ps
        Writing initial simulation snapshot: worklib.decoder_state_tb:sv
    Loading snapshot worklib.decoder_state_tb:sv ..... Done
xcelium> source /home/net/cadence/install/xCELUM2309/tools/xcelium/files/xmsimrc
xcelium> run
Starting decoder_state_tb...
=) --> Finished simulation without bugs.
Simulation complete via $finish() at time 1 US + 0
./decoder_state_tb.sv:44          $finish;
xcelium> exit
TOOL: xrun 23.09-s006: Exiting on Nov 25, 2024 at 20:40:58 EST (total: 00:00:00)
[nl581013@vlsi CadenceTB]

```

Figure 6.12: Xcelium Decoder Testbench Results

The main issue with the Xcelium installation on the UCF servers was how Simvision was implemented. Below is an example of what happens when you attempt to use Simvision with Xcelium. The fix for this is currently a work in progress, but the transfer over to using Cadence Genus for synthesis and then Cadence Virtuoso for layout should not cause any issues due to the standardized nature of System Verilog.

```

SimVision/Verisim Debug process terminated before a connection could be established.
xmsim: *E,STRPIN: Could not initialize SimVision connection: SimVision/Verisim Debug process terminated before a connection was established.
TOOL: xrun 23.09-s006: Exiting on Nov 25, 2024 at 21:09:39 EST (total: 00:00:02)
[nl581013@vlsi CadenceTB]

```

Figure 6.13: Simvision Failure

Early on in the project we would mainly use Xcelium for simulation, but when more complicated timing issues began to pop up with more complicated modules such as the execution stage or the forwarding unit, we realized we needed to find a workaround. At this point there was a bit of a split in software choices, our design team decided to use Xilinx Vivado, which could run the Xcelium simulator and while also having a simulation waveform. Meanwhile, our verification team decided that EDA playground would be useful because of the many pre-established UVM examples while also still using the Xcelium simulator. The main reason the design team had chosen to use Vivado was that the environment was much friendlier for synthesis and implementation when compared to EDA playground. That same simulation from before can be seen in Vivado in the figure below, but with a waveform that certainly confirms that our test bench is working as expected.

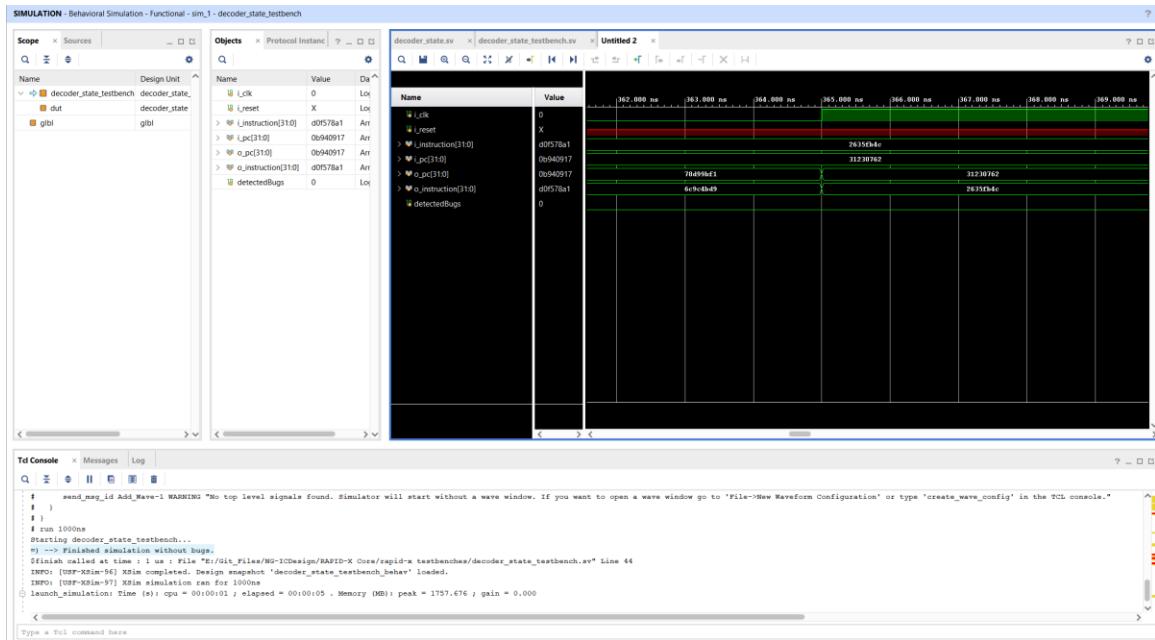


Figure 6.14: Vivado Testbench with Simulation Waveform

7 Core Redesign

Most senior design groups begin revising their projects in the second semester, but our team was fortunate to start experimentation early. This head start provided invaluable insights into our initial design, allowing us to identify and address critical issues sooner. Originally, our plan was for the RTL team to design the core and memory architecture while the UVM team focused on verifying the implementation. However, delays in verification prompted the RTL team to develop and utilize testbenches for testing and experimentation.

Early experimentation revealed several expected bugs in the RTL implementation but also exposed unforeseen issues that had significant implications. The most impactful discovery was the decision to forgo hardware solutions for data and control hazards, which caused severe performance degradation of 300–400%. It became evident that without a significant redesign, we would fail to meet the engineering goals specified by our customers.

Additionally, the limitations of our fabrication technology presented challenges. While increasing the system clock frequency was theoretically possible, real-world constraints such as IR drop, and timing delays made this impractical. These issues could potentially be resolved by a more technically skilled team, but optimizing the microarchitecture proved to be a more viable solution for our team.

The primary issue addressed by the RAPID-X architecture is the performance bottleneck caused by the lack of forwarding and control hazard units. The initial pipeline latency was also excessively high for a 5-stage pipeline, with 12 clock cycles required to complete an instruction, which could present challenges for real-time applications. Another significant inefficiency was the excessive use of registers in each stage to store input, internal, and output states, which could be optimized by retaining only the registers necessary for internal state.

7.1 Race Conditions in RAPID

As experimentation with the RAPID architecture progressed, we identified several race conditions that highlighted design flaws in our initial implementation.

The first race condition arose when all modules were connected, causing the CPU to remain indefinitely in the "WAIT" stage. Each pipeline stage waited for the others to complete execution in "WAIT," which depended on a pipeline-ready flag controlled by the "o_done" signal from each stage. The solution involved modifying the reset logic and changing the initial state from "WAIT" to an equivalent "EXECUTE" state, allowing the pipeline to function as intended.

A second race condition occurred during the transition between the decoder and execute stages. The decoder was responsible for determining the indices for the rs1 and rs2 registers. However, a conflict emerged during the register file access, as the output ports would not update until the next rising clock edge. This caused the execute stage to read incorrect data while the decoder updated the register file indices.

Several solutions were considered to resolve this issue:

Read the register file in the decoder: The decoded values could be stored in additional registers until the pipeline was ready for transition.

Move the register file to the execute stage: This would allow the execute stage to directly access the register file. However, it introduced complications with the writeback stage, which would also need register file access, potentially conflicting with future implementations like a forwarding unit.

Introduce an additional state in the execute stage: The execute stage would wait for an extra clock cycle before reading rs1 and rs2 values. This solution would stall the entire pipeline temporarily but resolve the data conflict.

Ultimately, Option 3 was chosen as the most straightforward and effective solution.

While the performance results of the second iteration of the RAPID architecture were disappointing, the challenges and issues we faced provided a valuable learning experience. Identifying these problems early in the senior design project timeline gave us the opportunity to refine our design and avoid similar pitfalls in the future. The lessons learned from the RAPID architecture directly informed the development of the RAPID-X architecture, enabling us to address critical performance bottlenecks and improve overall system efficiency. Even though the

process involved significant setbacks, these early iterations laid the foundation for a more robust and effective design moving forward.

7.2 Constraints and Time-Sensitive Redesign

When deciding to re-design the CPU architecture, our team faced significant time constraints. We needed to have a fully functional CPU core ready by the end of the week to showcase at both the senior design demo and in the senior design paper. Compounding this challenge, we had less than 12 hours from the time the decision to re-design was made to prepare and present our new architecture to Northrop Grumman engineers for approval. Their feedback and approval were essential to proceed, as this was the last opportunity this semester to finalize the architecture before submission.

Given these constraints, we aimed to reuse as much of the original code as possible in the new design. This approach allowed us to accelerate development and focus on resolving the most critical issues without starting from scratch. While this limited our ability to implement advanced optimizations, it ensured we could meet the immediate deadlines for the semester.

Although this iteration had to be the final revision for this semester due to time constraints, we acknowledged that further optimizations could be pursued in future revisions. The primary goal was to deliver a functional and demonstrable CPU architecture within the tight timeline, ensuring the project met the expectations for the senior design demo and paper.

This time-sensitive redesign process highlighted the importance of balancing engineering improvements with practical deadlines. It also reinforced the value of modular design and code reusability, which allowed us to adapt and iterate quickly under pressure. While the timeline restricted the scope of optimizations, it laid the groundwork for more sophisticated enhancements in subsequent iterations of the architecture.

7.3 Architectural Improvements in RAPID-X

The RAPID-X architecture represents a significant improvement over the original RAPID design, with changes specifically targeting the issues that hindered performance in the previous iteration. The following architectural modifications were implemented to enhance efficiency and address key bottlenecks:

7.3.1 Reeducation of Redundant States

One of the primary changes involved reducing the number of states in the finite state machines (FSMs) by leveraging combinational logic more effectively. The original design relied heavily on sequential states to handle transitions and logic, which added unnecessary complexity and increased execution time. By shifting to a design that maximized the use of combinational logic, we were able to streamline the control flow, reducing the latency associated with state transitions.

7.3.2 Reorganization of Pipeline Stages

Another significant improvement was the reorganization of the pipeline stages to address the race conditions inherited from the original architecture. These changes optimized the flow of data between stages and resolved timing issues that caused stalls and incorrect outputs.

7.3.3 Critical Architectural Changes

The following were the most impactful changes implemented in the RAPID-X architecture:

Removal of the Writeback Stage

By eliminating the writeback stage, the pipeline was reduced from five stages to four. This change directly improved the pipeline latency, decreasing the number of clock cycles required to complete instruction execution.

Separation of State and Logic Modules

Each pipeline stage was restructured into distinct state and logic modules. This separation allowed for a more modular design and enabled better optimization of each stage's functionality. Additionally, the reduction of excessive FSM states contributed to a significant reduction in cycles per instruction (CPI), bringing it down from approximately 3 to 1.

Implementation of Forwarding and Control Hazard Units

A critical upgrade in RAPID-X was the addition of forwarding and control hazard units. These units effectively eliminated the need for NOP instructions, which previously caused performance degradation. By resolving data dependencies and controlling hazards at the hardware level, the pipeline achieved a smoother and more efficient operation.

7.3.4 Performance Outcomes

These architectural improvements had a profound impact on the performance of the RAPID-X architecture. The reduction in pipeline stages decreased the overall latency, while the enhanced modularity and removal of redundant states streamlined execution. The inclusion of forwarding and control hazard units not only improved the throughput but also made the architecture more robust for handling a variety of workloads.

The RAPID-X redesign demonstrates the effectiveness of iterative refinement in addressing architectural bottlenecks. By focusing on simplification, modularity, and targeted performance enhancements, we were able to deliver an architecture that significantly outperforms its predecessor while maintaining a manageable design complexity.

7.4 Instruction Fetch Unit

7.4.1 High-Level Description and Requirements

The instruction fetch (IF) unit is responsible for fetching the current program instruction from main memory and pushing it into the pipeline. Because the IF unit is the first unit in the five-stage pipeline, any latency incurred at fetching instruction will have a ripple effect in the performance of the system. Because of this, it is important for the IF unit to operate as efficiently as possible.

The following signals are passed into the IF unit from the pipeline:

- *Pipeline ready* signal
- External program counter and corresponding set flag

The following signals are the outputs of the IF unit to the pipeline:

- Current program counter
- Current instruction

The following are the requirements for the IF unit:

- The IF unit shall keep track of the current program pointer, incrementing it by four for each valid instruction pushed into the pipeline.
- The IF unit shall reset the program pointer to the value of the reset vector upon assertion of the reset signal.
- The IF unit must be capable of externally setting the program counter to a value specified by the ALU, for operations such as jumping or branching.
- The IF unit must always push an instruction per clock cycle into the pipeline if and only if the *pipeline ready* signal is asserted; if a valid instruction is not available, it shall push must a NOP.

7.4.2 Microarchitecture and Finite State Machine

The current version of the IF unit is implemented using a four-state finite state machine to coordinate the actions of increasing or externally setting the program counter and scheduling fetches from the underlying cache system. The separation of different actions taken by the IF unit into distinct FSM states makes it straightforward to implement. The four cycles are described as followed:

- RESET: initializes the program counter to the hard coded reset vector.
- WAIT FOR PIPELINE: waits for the *pipeline ready* signal and increments the program counter (or sets it from an external source).
- SCHEDULE CACHE: waits for the cache to be ready to accept a request, then schedules a cache access with the program counter as the address.
- WAIT FOR CACHE: waits for the cache to return the requested data.

The following is a diagram of the logic of the FSM as well as what signals can trigger a state transition.

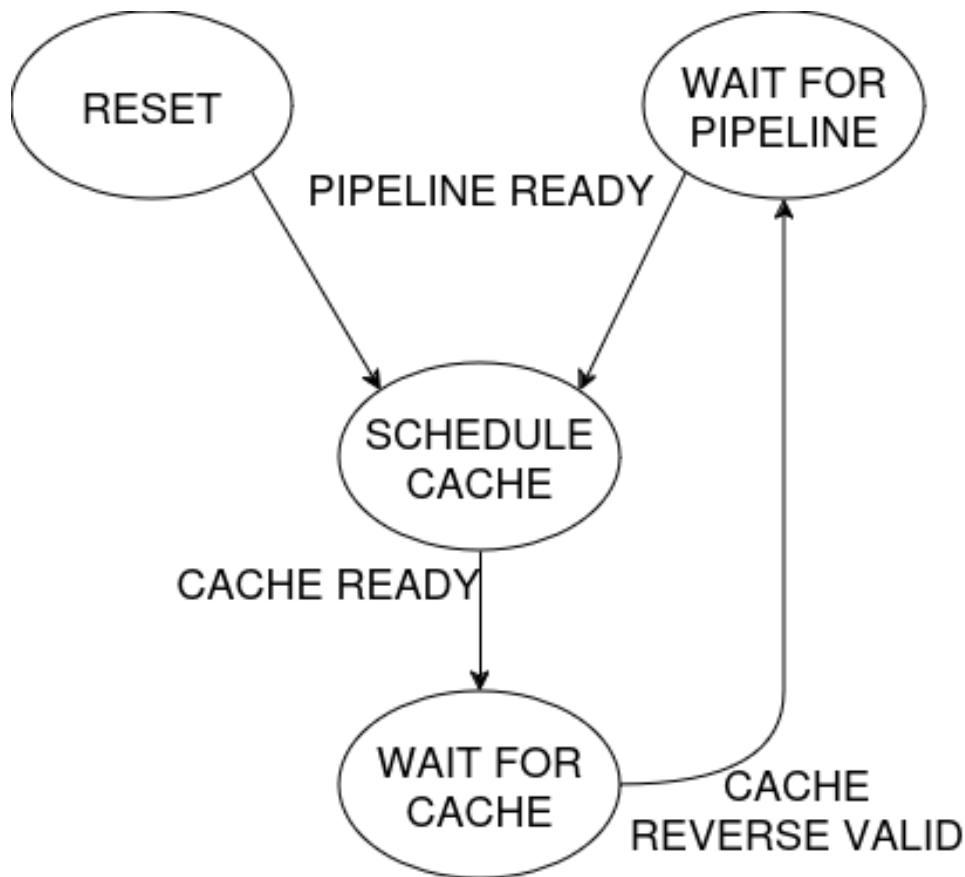


Figure 7.1: FSM for IF Unit

Internally, the IF unit must keep track of the internal program counter. The following is a high-level overview of its block diagram and internal subsystems:

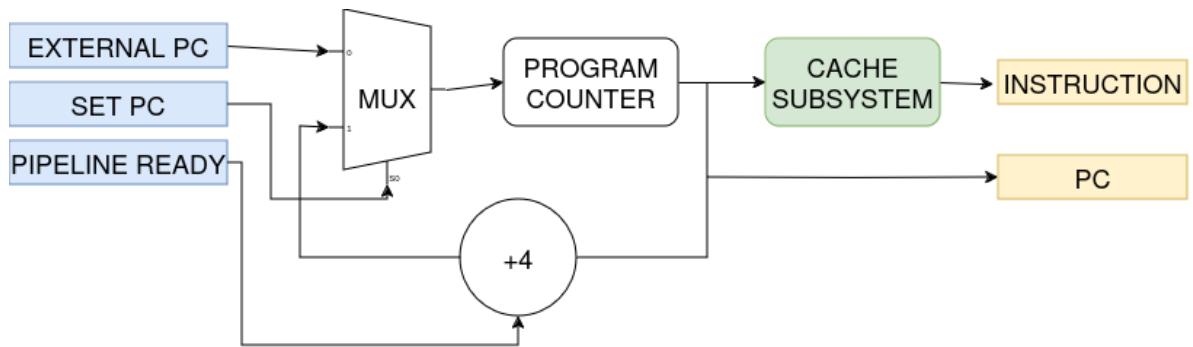


Figure 7.2: Block Diagram for IF Unit

7.4.3 Performance and Potential Improvements

As discussed previously, the performance of the IF unit highly impacts the rest of the pipeline. In the current implementation, the use of a 3-step cycle in the FSM results in delays when fetching an instruction. The advantage of this design is that

it is easy to test and diagnose problems, as behavioral errors or incorrect states can be easily detected during simulation and verification. The disadvantage is that it is suboptimal for sequential memory access (that is, accessing adjacent memory locations in order). In particular, if the *pipeline ready* signal becomes asserted before clock edge N, then the output of the unit is not stable until clock edge N+3 even if the operation resulted in a cache hit. In practice, this means that the IF unit inserts at least three NOPs in-between each pair of real instructions on perfect cache behavior – which means that the current pipeline implementation throughput could be improved with a better IF unit.

The team currently has several ideas for improving the IF unit. The method that appears to be most reliable is to increase the data bus width between the IF unit and the underlying cache subsystem, therefore allowing the IF to fetch several words at a time and store them in an internal register to queue the instructions for immediate output; while simultaneously scheduling a cache access for the next group of instructions – branching and cache misses notwithstanding, this improvement could result in up to 4x increase in throughout.

7.5 Memory Unit

The memory unit (MEM) unit is responsible for coordinating access to the main memory for both read and write access via an intermediary cache. Not all instructions in the RISC-V ISA access main memory, so the MEM unit should only be active when needed. In particular, it should just pass through the control signals to the following stage if the instruction is not a memory access.

The following signals are passed to the MEM unit from the pipeline:

- Memory enable: asserted if the instruction is memory access (store or load).
- Address: an absolute address.
- Register in: value of the register to write to memory.
- Finite control signal: determines different memory access modes (byte, half-word, or word), and sign-extend requirements.
- Other control signals.

The following are outputs from the MEM state to the next unit in the pipeline:

- Control signals passthrough
- Register out: value obtained from memory for load instruction, passthrough in all other cases.
- Done signal: indicates that the memory access is complete and the pipeline can advance.

The following are the requirements for the MEM unit:

- The MEM unit stores and loads in units of words.
- The MEM unit must passthrough signals in one clock cycle for non-memory instructions.
- The MEM unit must perform sign-extend when needed.

- The MEM unit must handle partial word reads and writes (byte, half-word) from and to registers (which are always full words).

The last point is important, since it implies two things (1) the memory unit must be able to translate any memory byte address to a word-aligned address before interface with the cache and (2) the memory unit must read or write the only the requested data. For example, when loading an unsigned half-word, the lower sixteen register bits must be set to the value from memory while the higher sixteen bits are reset to zero. This must always be the case independently of the size of the underlying memory bus to the cache and any permitted address. To this end, the memory unit performs two operations when the target memory address is not word-aligned (1) shifts the incoming and outgoing cache data to their proper position relative to where the memory address lies within a word and (2) sets a write mask to only overwrite the appropriate values from an existing word in the cache. This example illustrates it better:

Current data in cache:	AAAAAAAA'BBBBBBBB'CCCCCCCC'DDDDDDDD
Write mask:	00000000'00000000'11111111'11111111
Write value:	xxxxxxxx'xxxxxxxx'EEEEEEEE'FFFFFFF
<hr/>	
New data saved:	AAAAAAAA'BBBBBBBB'EEEEEEEE'FFFFFFF

This approach permits the memory unit to transparently handle all memory access instructions in a consistent manner. Should the underlying cache technology change for any reason, the memory unit can be adjusted to present the same behavior without altering any other pipeline units.

7.5.1 Microarchitecture and Finite State Machine

The current version of the MEM unit is implemented using a simple two-state finite state machine to coordinate the actions of interacting with the underlying cache.

- STANDBY: passes through all signals to the next pipeline stage for all non-memory instructions and keeps done signal asserted. If the instruction is a memory access, it schedules a cache request, clears the done signal, and goes to the ACCESS state.
- ACCESS: waits for the cache access to complete to connects it to the memory unit output, at which point it asserts the done signal to advance the pipeline and goes back to the STANDBY state.

Internally, the MEM unit must certain combinatorial operations to generate the write mask and shift register values for input and output. The following is a high-level overview of its block diagram and internal subsystems.

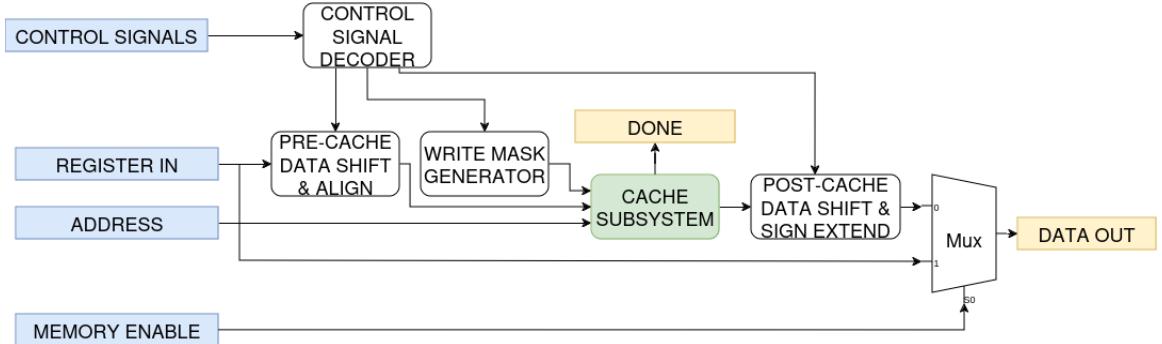


Figure 7.3: Block Diagram for the Memory Unit

7.5.2 Performance and Potential Improvements

Given that the memory unit uses a simple FSM, and almost all the logic is combinatorial and passes right through to the output or the underlying cache, the performance of the memory unit is mostly dependent on the underlying cache. Therefore, the memory unit will benefit from any cache improvements that might be implemented later.

As it stands, the two-state pipeline implementation would not permit two back-to-back memory operations to occur on two consecutive clock edges. But this is irrelevant, as the underlying cache takes two clock edges to become ready for the next request anyways. So, a cache hit memory access takes one cycle to complete – but there must be at least one clock cycle in between consecutive access (for example by making the next instruction a non-memory-access) otherwise the memory unit will stall the pipeline for one clock cycle.

7.6 Cache to main memory communication

To simplify the design of the cache, it was determined that the cache shall transfer data in units of one cache line at a time to and from the main memory controller. In this context, “controller” means a simulated device that can coordinate accesses from two or more caches to a single memory space. The current cache line size is 16 bytes or 128 bits, but this can be modified with minimal effort.

For simulation purposes, this controller is implemented as a SystemVerilog module that loads its initial data from disk to provide a “fake memory”. This fake memory behaves like the top most main memory of the processor: all reads and writes originating from both the data and instruction cache end up in the fake memory. The contents of which can later be examined for verification purposes. Depending on the scope of the project, the memory controller can be made to interface with a Wishbone or AXI bus through which main memory or peripherals would be connected. In general, making the cache subsystem interface with a memory controller in a well-defined manner simplifies both implementation and verification by allowing the use of fake memories during the simulation and the possibility to use a bus should the project grow larger.

The following diagram illustrate a high-level overview of the processor units that connect, directly or indirectly, to main memory.

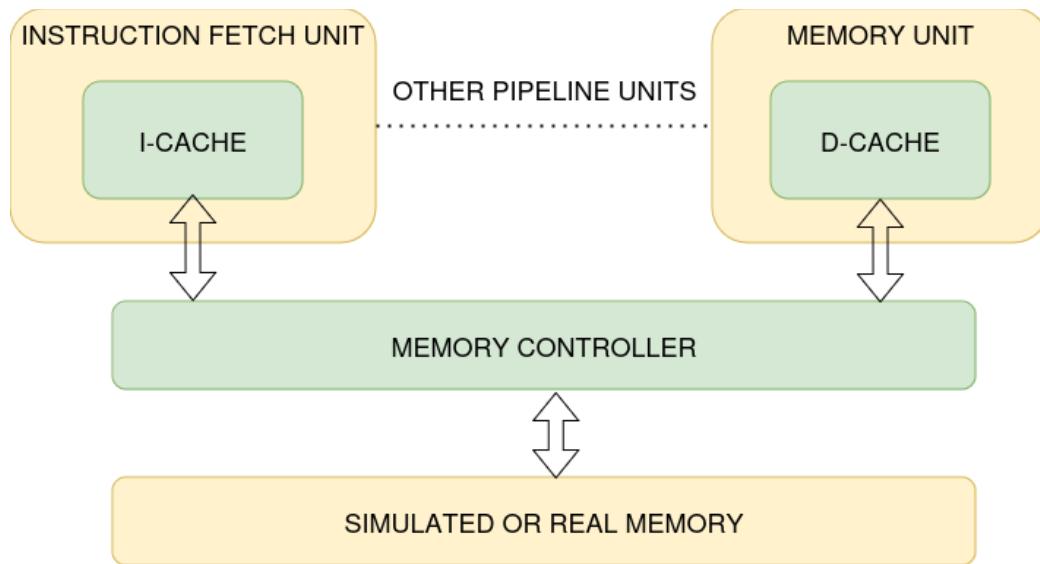


Figure 7.4: Overview of Processor Unit

7.7 Decoder Stage

7.7.1 Objective Summary

Decode and prepare instructions for execution in the 4-stage pipeline.

7.7.2 Requirements

The decoder stage is responsible for two critical tasks in the pipeline:

1. The generation of the proper control signals for each instruction and the,
 - a. Grouping instructions with similar functionality and operations into the families.
 - b. Provide the ability to differentiate between instructions with exact opposite operations such as addition and subtraction.
2. Arrange the immediate values from the instructions with sign extensions to a 32-bit output immediate field with support for the following formats:
 - a. (12-bit) immediate I-type
 - b. (20-bit) immediate U-type
 - c. (12-bit) immediate B-type
 - d. (12-bit) immediate S-type
 - e. (12-bit) immediate J-type
 - f. (20-bit) Shamt I-type
3. Split the decoder stage into two modules,
 - a. A state module containing the input program counter and input instruction.
 - b. A combinational logic module to perform the decoding process.

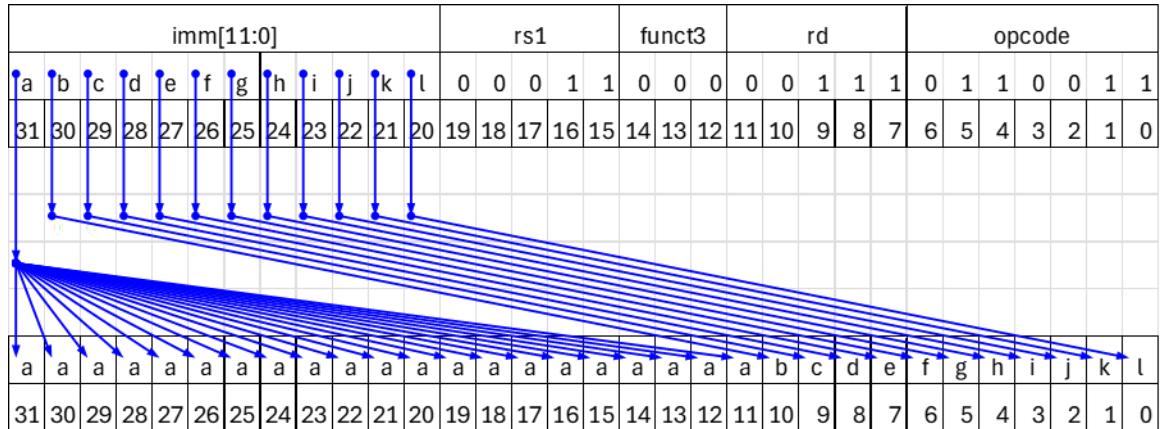


Figure 7.5: I-type Immediate Format

imm[11:0]												rs1			funct3			rd			opcode										
0	b	0	0	0	0	0	h	i	j	k	l	0	0	0	1	1	0	0	0	0	0	0	0	1	1	0	0	1	1		
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 7.6: shamt-I format

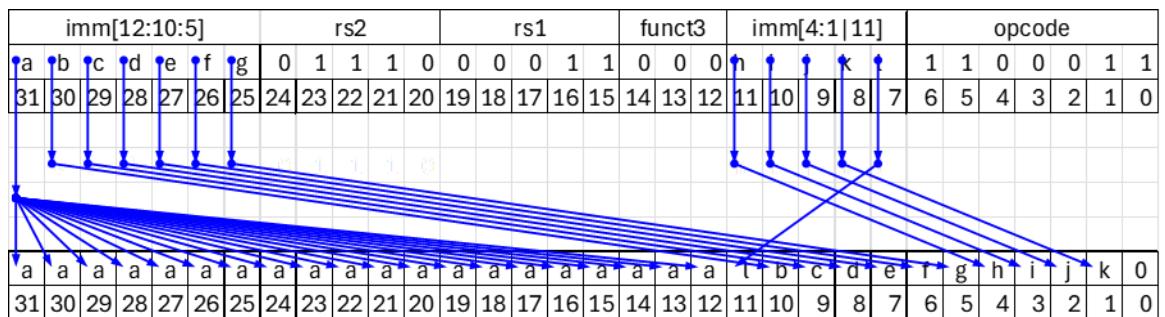


Figure 7.7: B-type Immediate Format

imm[20 10:1 11 19:12]																rd			opcode												
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	0	0	1	1	1	1	0	1	0	1	1	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 7.8: J-Type Immediate Format

imm[31:12]																rd				opcode											
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	0	0	1	1	1	0	1	0	0	0	1	1
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

imm[31:12]								rd				opcode							opcode												
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	0	0	1	1	1	0	1	0	0	0	1	1
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 7.9: U-Type Immediate Format

imm[11:5]								rs2				rs1				funct3			imm[4:0]				opcode								
a	b	c	d	e	f	g	h	0	1	1	1	1	0	0	0	1	1	0	0	0	0	0	1	1	0	1	0	0	0	1	1
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	b	c	d	e	f	g	h	i	j	k	l	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 7.10: S-Type Immediate Format

7.8 Control Signal Requirement

The following reference table can be used to aid the process of decoding the signal and verifying the validity of the decoded signals.

Table 55: Instruction Decomposition

Instruction	OpCode	Funct3	Funct7	I/O
LUI	0110111	N/A	N/A	imm+rd
AUIPC	0010111	N/A	N/A	imm+rd
JAL	1101111	N/A	N/A	imm+rd
JALR	1100111	000	N/A	imm+rd+rs1
BEQ	1100011	000	N/A	imm+rs1+rs2
BNE	1100011	001	N/A	imm+rs1+rs2
BLT	1100011	100	N/A	imm+rs1+rs2
BGE	1100011	101	N/A	imm+rs1+rs2
BLTU	1100011	110	N/A	imm+rs1+rs2
BGEU	1100011	111	N/A	imm+rs1+rs2
LB	0000011	000	N/A	imm+rs1+rs2
LH	0000011	001	N/A	imm+rd+rs1
LW	0000011	010	N/A	imm+rd+rs1
LBU	0000011	100	N/A	imm+rd+rs1
LHU	0000011	101	N/A	imm+rd+rs1
SB	0100011	000	N/A	imm+rs1+rs2

SH	0100011	001	N/A	imm+rs1+rs2
SW	0100011	010	N/A	imm+rs1+rs2
ADDI	0010011	000	N/A	imm+rs1+rd
SLTI	0010011	010	N/A	imm+rs1+rd
SLTIU	0010011	011	N/A	imm+rs1+rd
XORI	0010011	100	N/A	imm+rs1+rd
ORI	0010011	110	N/A	imm+rs1+rd
ANDI	0010011	111	N/A	imm+rs1+rd
SLLI	0010011	001	0000000	imm+rs1+rd
SRLI	0010011	101	0000000	imm+rs1+rd
SRAI	0010011	101	0100000	imm+rs1+rd
ADD	0110011	000	0000000	rd+rs+rs2
SUB	0110011	000	0100000	rd+rs+rs2
SLL	0110011	001	0000000	rd+rs+rs2
SLT	0110011	010	0000000	rd+rs+rs2
SLTU	0110011	011	0000000	rd+rs+rs2
XOR	0110011	100	0000000	rd+rs+rs2
SRL	0110011	101	0000000	rd+rs+rs2
SRA	0110011	101	0100000	rd+rs+rs2
OR	0110011	110	0000000	rd+rs+rs2
AND	0110011	111	0000000	rd+rs+rs2

The following table was developed from analysis from the previous table. Key aspects of the following control signal table,

1. FCS opcode is mostly linked to the funct3 parameter in the original instructions; however, it has been introduced to other instructions that do not naturally have this parameter to reduce the required number of signals.
2. IOP is a flag which indicates to do an inverse operation for example for the memory load operations a load or a store would be determined by the IOP flag. IOP is inspired by the funct7 field in RV32 ISA, but it is extended to apply to other instructions as well.

Table 56: Control Signals

Instruction	OpCode	Control Category	Finite Control Signals	Inverse Op	Control Signal
LUI	01101	LUP (LOAD UPP IMM)	000	1	LUP-000-1
AUIPC	00101	LUP (LOAD UPP IMM)	000	0	LUP-000-0
JAL	11011	UB (UNCOND.BRANCH)	000	0	UB -000-0

			UB			
JALR	11001	(UNCOND.BRANCH)	000	1	UB -000-1	
BEQ	11000	CB (COND.BRANCH)	000	0	CB -000-0	
BNE	11000	CB (COND.BRANCH)	001	0	CB -001-0	
BLT	11000	CB (COND.BRANCH)	100	0	CB -100-0	
BGE	11000	CB (COND.BRANCH)	101	0	CB -101-0	
BLTU	11000	CB (COND.BRANCH)	110	0	CB -110-0	
BGEU	11000	CB (COND.BRANCH)	111	0	CB -111-0	
LB	00000	MEM (MEM LOAD/STORE)	000	0	MEM-000-0	
LH	00000	MEM (MEM LOAD/STORE)	001	0	MEM-001-0	
LW	00000	MEM (MEM LOAD/STORE)	010	0	MEM-010-0	
LBU	00000	MEM (MEM LOAD/STORE)	100	0	MEM-100-0	
LHU	00000	MEM (MEM LOAD/STORE)	101	0	MEM-101-0	
SB	01000	MEM (MEM LOAD/STORE)	000	1	MEM-000-1	
SH	01000	MEM (MEM LOAD/STORE)	001	1	MEM-001-1	
SW	01000	MEM (MEM LOAD/STORE)	010	1	MEM-010-1	
ADDI	00100	IMM (ALU IMM)	000	0	IMM-000-0	
SLTI	00100	IMM (ALU IMM)	010	0	IMM-010-0	
SLTIU	00100	IMM (ALU IMM)	011	0	IMM-011-0	
XORI	00100	IMM (ALU IMM)	100	0	IMM-100-0	
ORI	00100	IMM (ALU IMM)	110	0	IMM-110-0	
ANDI	00100	IMM (ALU IMM)	111	0	IMM-111-0	
SLLI	00100	IMM (ALU IMM)	001	0	IMM-001-0	
SRLI	00100	IMM (ALU IMM)	101	0	IMM-101-0	
SRAI	00100	IMM (ALU IMM)	101	1	IMM-101-1	
ADD	01100	REG (ALU REG)	000	0	REG-000-0	
SUB	01100	REG (ALU REG)	000	1	REG-000-1	
SLL	01100	REG (ALU REG)	001	0	REG-001-0	
SLT	01100	REG (ALU REG)	010	0	REG-010-0	
SLTU	01100	REG (ALU REG)	011	0	REG-011-0	
XOR	01100	REG (ALU REG)	100	0	REG-100-0	
SRL	01100	REG (ALU REG)	101	0	REG-101-0	
SRA	01100	REG (ALU REG)	101	1	REG-101-1	

OR	01100	REG (ALU REG)	110	0	REG-110-0
AND	01100	REG (ALU REG)	111	0	REG-111-0

The following are the abbreviations used for the table above:

1. LUP – Load Upper Immediate Family
2. UB – Unconditional Branch Family
3. CB – Conditional Branch Family
4. MEM – Memory Load Store Family
5. IMM – ALU Immediate
6. REG – ALU Register

7.8.1 State Description

The state module shall be a distinct and separate from the logic module and must retain,

1. Program Counter Value and,
2. Instruction from the instruction fetch stage.

The state module must also output the internal state of PC and instruction value via continuous assignment, to enable proper execution of the combinational logic block.

7.9 Stage Module I/O Ports

Table 57: Stage Module I/O Ports

Type	Port	Summary
input	i_clk [1bit]	Clock signal
input	i_reset [1-bit]	Reset signal
input	i_pc_load[1-bit]	Flag from the execute stage to determine branching. If flag is high, then the output values are redirected to a NOP instruction and the output PC value is changed to a 0.
input	i_instruction [32-bit]	Input instruction from instruction fetch stage.
input	i_pc [32-bit]	Input PC value from instruction fetch stage.
output	o_pc [32-bit]	Output PC value.
output	o_instruction [32-bit]	Output instruction from internal state.

7.10 Logic Module I/O Ports

Table 58: Logic Module I/O Ports

Type	Port	Summary
input	i_instruction [32-bit]	Input instruction from instruction fetch stage.
output	o_control_signal [control_ex_s]	Decoded control signal given an instruction.
output	o_imm [32-bit]	Decoded immediate value from the instruction.

The control_ex_s structure is defined as the following:

1. Load upper immediate family [1-bit]
2. Unconditional branch family [1-bit]
3. Conditional branch family [1-bit]
4. Memory family [1-bit]
5. ALU Immediate family [1-bit]
6. ALU Register family [1-bit]
7. IOP Flag [1-bit]
8. FCS OpCode [3-bit]
9. rs1 index [5-bit]
10. rs2 index [5-bit]
11. rd index [5-bit]

7.11 Execute Stage

7.11.1 Objective Summary

The execution stage is responsible for performing arithmetic, bitwise operation, and branching.

7.11.2 Requirements

Must support the following operations,

1. Addition and Subtraction
2. Arithmetic and Logical shifting
3. Conditional branching,
 - a. Support signed conditional branching
 - b. Support unsigned conditional branching
4. Support unconditional branching.
5. Support setting upper bits of register value.

7.11.3 State Description

The state module shall be a distinct and separate from the logic module and must retain,

1. Control signal (execute) structure
2. Rs1 register value.
3. Rs2 register value.
4. Immediate value.
5. Program Counter value.

The state module must also output the internal state of PC and instruction value via continuous assignment, to enable proper execution of the combinational logic block.

7.11.4 State Module I/O Ports

Table 59: 7.11.4 State Module I/O Ports for Execute

Type	Port	Summary
input	i_clk [1bit]	Clock signal
input	i_reset [1-bit]	Reset signal
input	i_pc_load[1-bit]	Flag from the executive stage to determine branching. If the flag is high, then the output values are redirected to a NOP instruction and the output PC value is changed to a 0.
input	i_pc [32-bit]	Input instruction from instruction fetch stage.
input	i_control_signal [32-bit]	Input PC value from instruction fetch stage.
input	i_rs1 [32-bit]	Output PC value.
input	i_rs2[32-bit]	Output instruction from internal state.
input	i_imm	Immediate value input from decoder.
output	o_pc	
output	o_control_signal [control_ex_s]	
output	o_rs1	
output	o_rs2	
output	o_imm	

Table 60: Logic Module I/O Ports

Type	Port	Summary
input	i_pc [32-bit]	Input instruction from instruction fetch stage.
input	i_control_signal [control_ex_s]	Decoded control signal given an instruction.

input	i_rs1[32-bit]	Forwarded rs1 register value.
input	i_rs2 [32-bit]	Forwarded rs2 register value.
input	i_imm [32-bit]	Decoded immediate value from decoder stage.
output	o_control_signal [control_mem_s]	Trimmed control signal for memory stage.
output	o_pc_load [1-bit]	Branch signal
output	o_pc_ext [32-bit]	Branch address
output	o_memory_data [32-bit]	Data for memory store operations.
output	o_rd_output [32-bit]	Data for writing to the register file.

The control_mem_s structure is defined as the following:

1. Memory family [1-bit]
2. IOP Flag [1-bit]
3. FCS OpCode [3-bit]
4. rd index [5-bit]

8 System Testing and Evaluation

8.1 Verification Scope

The verification scope for the RISC-V RV32I processor focuses on ensuring the functionality of its 5-stage architecture discussed in the previous chapters and they are: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), and Write Back (WB) stages. This pipeline was designed to process instructions. The verification process seeks to assess each stage module to identify potential issues in the code. By performing this verification, we hope to ensure the accuracy and efficiency of our design.

RV32I Base Instruction Set					
imm[31:12]			rd	0110111	LUI
imm[31:12]			rd	0010111	AUIPC
imm[20 10:1 11 19:12]			rd	1101111	JAL
imm[11:0]	rs1	000	rd	1100111	JALR
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011 BEQ
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011 BNE
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011 BLT
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011 BGE
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011 BLTU
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011 BGEU
imm[11:0]	rs1	000	rd	0000011 LB	
imm[11:0]	rs1	001	rd	0000011 LH	
imm[11:0]	rs1	010	rd	0000011 LW	
imm[11:0]	rs1	100	rd	0000011 LBU	
imm[11:0]	rs1	101	rd	0000011 LHU	
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011 SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011 SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011 SW
imm[11:0]	rs1	000	rd	0010011 ADDI	
imm[11:0]	rs1	010	rd	0010011 SLTI	
imm[11:0]	rs1	011	rd	0010011 SLTIU	
imm[11:0]	rs1	100	rd	0010011 XORI	

Figure 8.1: Lists some of the instructions from RV32I

The primary features to be verified from our design is the comprehensive set of instructions supported by RV32I instruction set architecture as some examples can be seen in the image above. We will exclude as part of our design and verification the “FENCE” instructions and Control and Status Register (CSR) operations. The goal is to have each single instruction verified in many different ways to confirm the correct execution with different constraints.

To achieve a thorough verification and achieve our objectives, the verification team will be using a combination of directed-testing, random-testing, and coverage-driven verification. We will also be utilizing an advanced verification framework in Universal Verification Methodology. The team will implement robust tests to simulate all the different operations and we hope this approach will have a 85% coverage or higher as Northrop Grumman has set that goal for us.

8.1.1 Instruction Fetch Testing

In this section we will discuss the individual testing of the instruction fetch unit from the RAPID-X Core. The `cpu_ifetch_unit` module has five inputs and four outputs. Of those inputs we have the `i_clk`, `i_reset`, `i_ext_pc`, `i_pc_load`, and

`i_pipeline_ready`. As for outputs we have `o_pc`, `o_instruction`, `mem_req` and `mem_res`; `mem_req` and `mem_res` pertain to the communication between the memory interface package that contains relevant signals between cache and core communication. The `i_ext_pc` is an external program counter for jumps or branches, and `i_pc_load` is for control signals to load a new program count. `I_pipeline_ready` indicates pipeline readiness for the next instruction. `O_pc` is the output for the current program count, `o_instruction` is the fetched instruction and structured signals for memory requests `mem_req` and responses `mem_res` to interface with the cache and memory controller. It's important to review such inputs and outputs for further discussion of the testbench, simulation results, and expected unit behavior. The unit includes an instantiation of the `dcache_interface` and `dcache_dm1cycle`. Note that although it is called `dcache`, it is an instantiation of the same direct mapped architecture and is not connected to the instantiation located in the memory stage. This is in correspondence with our split cache design.

The fetch unit utilizes a finite state machine to determine memory communication and overall functionality in case of reset, pipeline readiness, and instruction validity; this FSM is central to the design. The module maintains registers for the current program counter, known as PC, and its next value, `pc_next`, and the current and next states of the finite state machine. The FSM installed governs the fetch process requiring proper testing and signal observation in the simulation waveform. The design also includes a `dcache_interface` object that is used for the cache communication, with a workaround signal `iface_valid`. Constants such as `RESET_VECTOR` which is the PC's initial value, and `NOOP_INSTRUCTION` which is a default instruction sent to the pipeline during stalls support the functionality of the module. This is really good for debugging as it communicates well to the tester that there is proper interaction among multiple modules in the system meaning that we know when stalls occur and how long.

Four states dictate the instruction fetch logic driven by the FSM. The first state is the `RESET` state, where the PC is initialized to the `RESET_VECTOR`. The module then waits for the pipeline to signal readiness. The `SCHEDULE_CACHE` state, the second state, starts the fetch request to the cache using the `iface_valid` signal; this sets the cache address to the present program count. This state serves an additional purpose in handling edge cases where a new program count is loaded in the middle of an operation which restarts the catch fetch process. Once the cache is ready, the FSM moves on to the `WAIT_FOR_CACHE` state. In this state, it waits for the cache to provide the requested instruction and handles middle operation PC updates accordingly. In the final state, `WAIT_FOR_PIPELINE` state, the module outputs the fetched instruction and PC to the pipeline. Note that when the pipeline is ready, the PC increments by 4 or is updated with the `i_ext_pc` if a new PC is loaded dictated by other stages of the pipeline like the decoder and control signals. Then the FSM will transition back to `SCHEDULE_CACHE` for the upcoming instruction fetch.

The module's cache subsystem includes a `dcache_interface` for organized communication and a `dcache_dm1cycle` for the direct-mapped data cache design used. These components are responsible for the handling of memory requests

from the variables mem_req and mem_res; request and response. This allows for efficiency and timely instruction retrieval. The control signals, such as iface.addr, iface.valid and iface.rvalid, are coordination signals for the cache operations. Note that the module always performs read operations and iface.rw is set to 0.

Before performing any advanced verification methods such as UVM, it was important for a simple testbench to be created that can later be further developed. The goal of such testbench was to validate the functionality of the instruction fetch unit as well as its ability to interact with a cache and memory subsystem while also delivering accurate instruction data to the pipeline. The established simulated environment ensured the FSM, PC logic, and cache-handling mechanisms work as intended. The results demonstrate that the module fetches the correct instruction in sequence and handles control signals like pipeline readiness. The testbench in question creates a controlled testing scenario where a sequence of expected instructions is preloaded into the memory subsystem; which in this case was loaded into our fake_memory module. Such a module is connected to the cache via memory_controller_interface in this case mem_res and mem_req. Both of these signals are declared at the top of the testbench and used in both the fake_memory instantiation and cpu_ifetch_unit instantiation. In the actual initial begin portion we reset all signals and registers such as i_pc_load, i_pipeline_ready, reset and then proceed to set i_pipeline_ready to 1. In setting this value to a constant 1, we can sequentially fetch in ideal conditions while also testing expected stalling behavior. The expected stall behavior causes the unit to push a no operation instruction and initial program counter while the unit waits for a response from memory. This no operation and initial program counter does not reflect the real program counter and operation that the unit has yet to validate from memory. Reference Appendix C to see the instruction fetch testbench.

A key aspect of this testbench is its ability to monitor and validate interactions between the cache as we can observe memory requests and responses. This allows for the determination of correct scheduling of cache operations and data retrieval. FSM transitions are also verified as RESET, SCHEDULE_CACHE, WAIT_FOR_CACHE, and WAIT_FOR_PIPELINE, in expected order has proper sequencing of fetch operations. The testbench also simulates edge cases such as mid-operation PC updates, in which the unit has to restart the cache operation with a new PC; this counters unexpected control signal changes. The testbench also assesses the module's ability to handle the synchronization between instruction fetching and the rest of the pipeline demonstrating a good amount of system level verification.

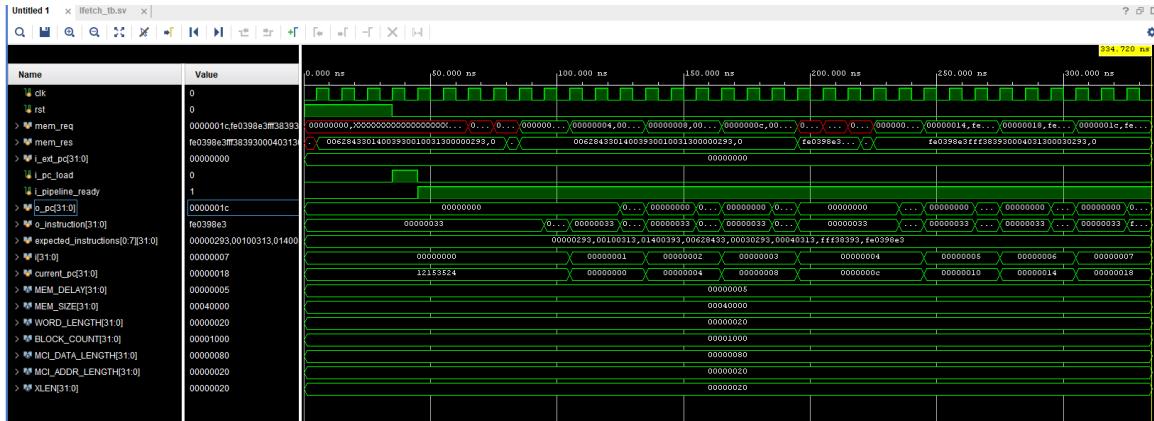


Figure 8.2: Whole waveform for the instruction fetch

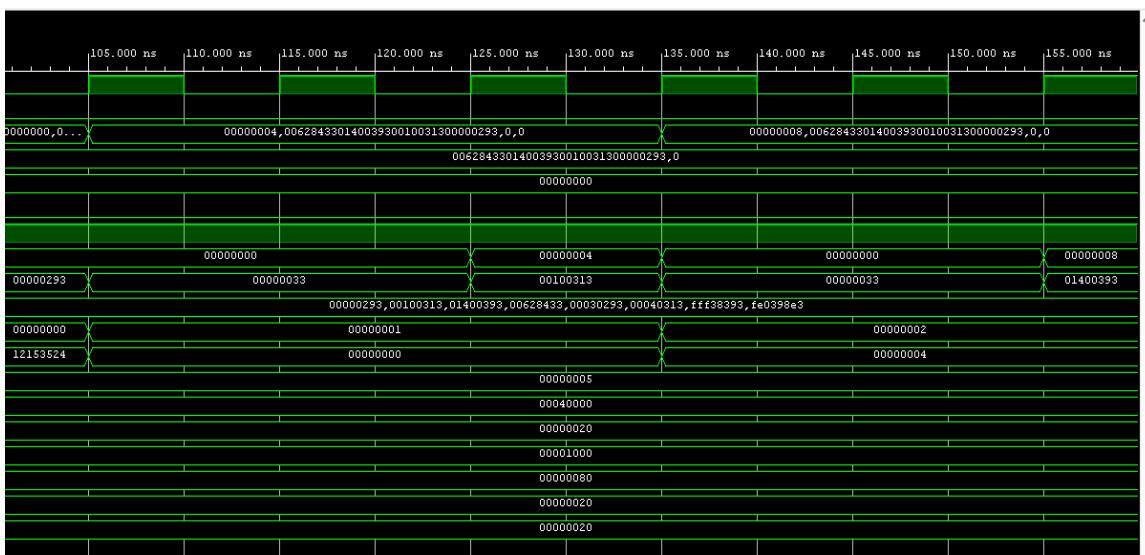
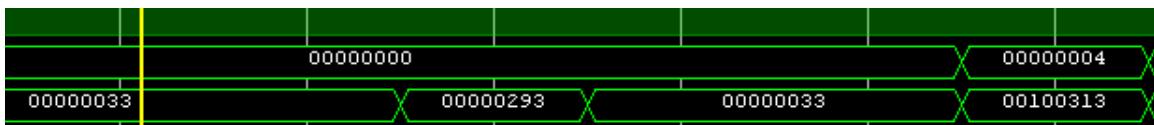
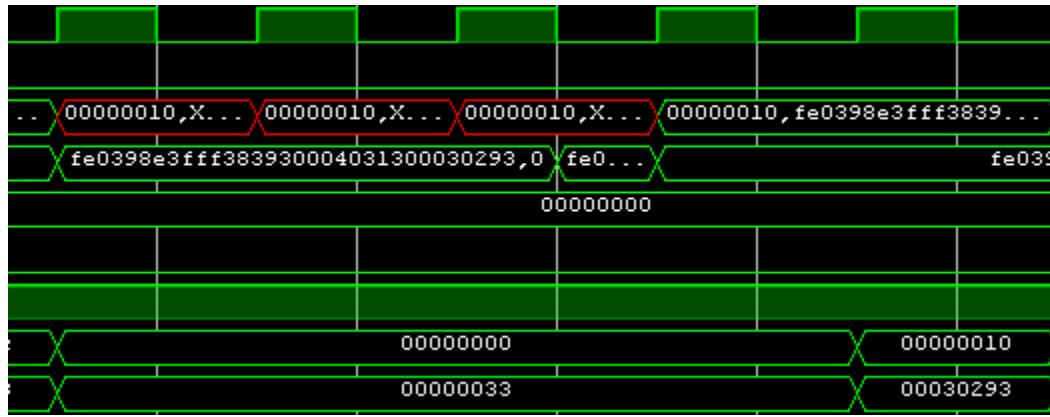


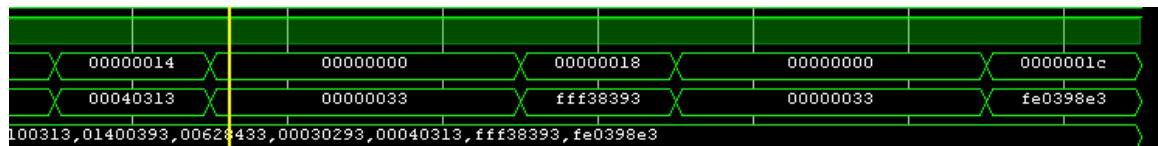
Figure 8.3: Waveform at the start of the fetching state



Here we see the initial program count at “00000000” and the no operation “00000033” and concurrent values. At the start we see the program count be the same for the first instruction and for no operations. The program counter also changes to “00000004” when loading the second instruction; this is also expected behavior.



Here the fetch unit is pushing no operations and initial program count when a cache line transition occurs in memory as depicted by the memory response and memory request signals in red. The loading of the next instruction is also depicted by the values “00000010” for the 5 instructions “00030293”.



```
# run 1000ns
Starting Instruction Fetch with Cache Testbench...
Fetching and validating instructions:
[FAKEMEM] ReadReq block [00000000]
PC [00000000]: Fetched Instruction [00000293], Expected [00000293]
PC [00000004]: Fetched Instruction [00100313], Expected [00100313]
PC [00000008]: Fetched Instruction [01400393], Expected [01400393]
PC [0000000c]: Fetched Instruction [00628433], Expected [00628433]
[FAKEMEM] ReadReq block [00000010]
PC [00000010]: Fetched Instruction [00030293], Expected [00030293]
PC [00000014]: Fetched Instruction [00040313], Expected [00040313]
PC [00000018]: Fetched Instruction [ffff38393], Expected [ffff38393]
PC [0000001c]: Fetched Instruction [fe0398e3], Expected [fe0398e3]
$finish called at time : 335 ns : File "C:/Users/prime/OneDrive/De:
```

In the simulation results, we observe that the logged outputs confirm that the fetched instructions match the expected values. The logs above demonstrate instructions stored in the fake_memory module that are subsequently loaded into the cache. The ifetch unit fetches each instruction from the cache in the same order they were read from the text file loading into memory; fake_memory module.

The detailed testing and simulation of the instruction fetch unit from the RAPID-X Core depicts functionality and seamless integration with the cache and memory subsystem modules. The testbench demonstrated that it was able to validate key aspects of the design, including the main FSM transitions, proper program counter

updates, and handling of pipeline readiness as well as cache responsiveness. The results highlight the unit's ability to manage edge cases, like the mid-operation program counter updates, while making sure proper and accurate instruction delivery to the pipeline.

8.1.2 Verification Software UVM

For the software testing of our pipeline and different modules, we have decided to utilize EDA playground website, since it is a very comprehensive tool that was also recommended by the review committee as a good resource. This tool has also come in a good time as there have been numerous issues trying to utilize the cadence tools available in the VLSI server which became very troublesome and unreliable.

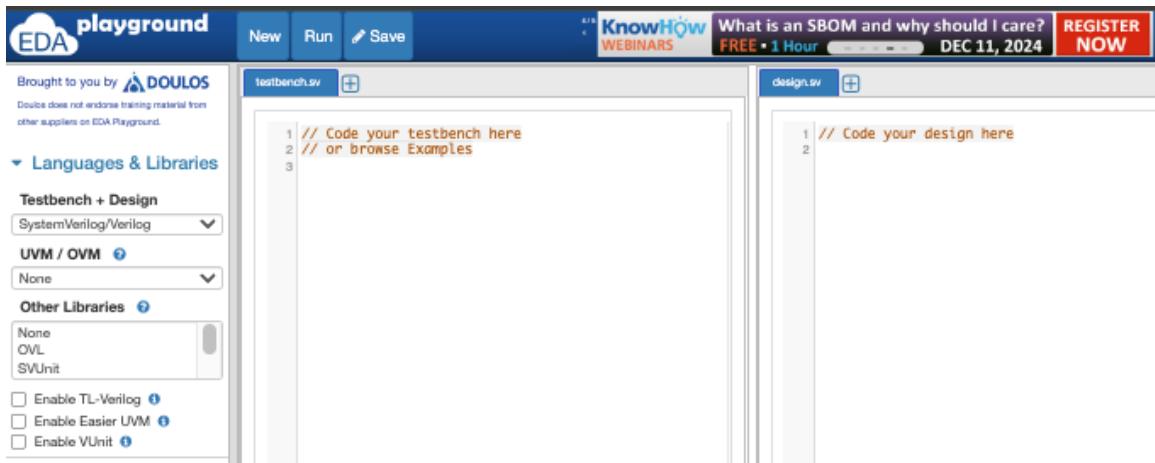


Figure 8.4: Shows the home screen from EDA playground website.

EDA Playground is an online platform that can be used as a tool for learning, designing and simulating hardware description languages such as System Verilog as it was the language used for this project. There are great benefits in using EDA Playground such as no installation required, since it can be used straight from the website, so it works both on Windows and macOS which was useful for the variety of systems the group's team members possess.

From figure 9.1, we can also observe that there are different libraries and UVM/OVM that can be used in EDA playground. This goes to show yet another feature by this website that can accommodate the different necessities a user may have. In this website you are also able to make copies of links that are sent to you so that you can work on a code without changing the original code and you are also able to save the project you are currently working at to pick up where you left off at a different time.

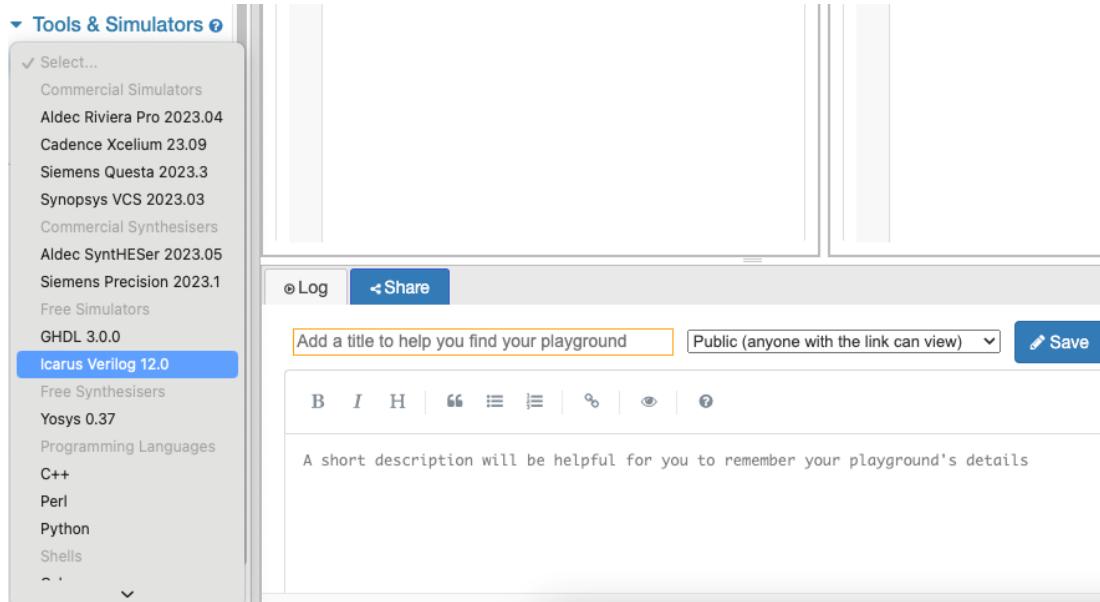


Figure 8.5: shows the available tools and simulators available in EDA Playground.

EDA Playground is able to accommodate different hardware description languages as well such as Verilog and VHDL. Another great benefit of using EDA Playground is that it offers many different software-based tools from different companies such as Cadence, Siemens and Synopsys. This way, we are exposed to a variety of different platforms and can learn in different platforms.

From the picture above we can also see the log tab which the space is utilized in the website where a lot of important information will be displayed. Once there is a design and a testbench inserted in the proper places, the log will be able to inform the errors that the current code is facing. This log is also very useful to see the print that the designer uses in the code statements.

When particularly looking at UVM, there are a lot of UVM_INFO and UVM_ERROR through the code which flags to the designer where there potential errors that needs to be fixed and all of that is displayed in the log tab.

Cadence Xcelium

The Xcelium xrun command is used, so all of these options can be either Compile or Run Options.

Option	Description
-abvcoveron	Enable cover directives
-abvevalnochange	Revert back expression change optimization
-abvrecordoverall	Record all finishes for cover directives
-access <+/-rwc>	Turn on read, write and/or connectivity access
-allowredefinition	Allow multiple files to define the same object
-assert	Enable PSL language features

Figure 8.6: Xcelium Compiler Run Options

Another nice feature about using EDA Playground is that you are able to see how to do different options in the compile options sections. This page does a very good job explaining what are all the options available in EDA Playground and what are their descriptions

A very good example of how a compile option can be used is the option of setting the seed for the random number generator for the SystemVerilog \$random function which is very useful for UVM as it is a core part to obtain different inputs so that different outputs can be compared.

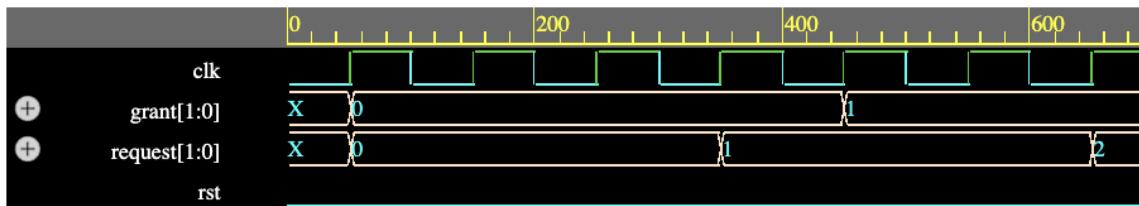


Figure 8.7: Example Waveform from EDA Playground

In the example above shows an example from a code created that had a logic code when to grant access when requested and when not to grant when requested. Just like in a software like Vivado we can see that the EDA Playground platform is also able to show the waveform which is a great thing when we want to better understand using visuals to what is happening in the code.

```
** Report counts by severity
UVM_INFO : 152
UVM_WARNING : 0
UVM_ERROR : 0
UVM_FATAL : 0
** Report counts by id
[COMPARE] 149
[RNTST] 1
[TEST_DONE] 1
[UVM/RELNOTES] 1
```

Figure 8.8: Example of an output using EDA Playground

The figure above is an example of how using Universal Verification Methodology for an ALU can look like. This quick snapshot of the figure above can be used to have a quick overview of how the code is behaving during verification. When using

UVM for a different design code, we obtained very different results that included warnings and errors.

The structure that most people use for UVM is very much standardized, this way people can quickly identify what each message is trying to relate to the designer. These are called UVM Messaging Macros and are widely used by designers to provide essential information on the code that will be useful to understand the state it is currently in and also the simulation.

The UVM warning is supposed to relate issues when unexpected conditions that do not necessarily stop the simulation occur, but may require the designer's attention to fix it later. The UVM_error indicates critical errors that require immediate attention for the code to compile.

There are also different verbosity levels that can be used in UVM, they are as follows: UVM_NONE, UVM_LOW, UVM_MEDIUM, UVM_HIGH, UVM_FULL and UVM_DEBUG. In order I listed them, this is their description: No messages to be printed, low level information message, medium verbosity includes standard info, high verbosity is detailed message, full verbosity is maximum level of detail and debug level includes all messages. The verbosity level is set in the testbench whenever the designer thinks it is appropriate.

The verification strategy for the RISC-V RV32I core processor will leverage the UVM to create a basic skeleton structure to be used across all modules since the goal for Universal Verification Methodology is to have a versatile and flexible structure, that will be the verification team goal.

A main characteristic for using UVM will be to have very similar modules. So the verification team will create the Driver, Sequencer, and the monitor to be used almost identical in every situation. The sequencer will organize the sequences of transactions. The monitor will watch the Device Under Test behavior and activity as we stimulate the inputs.

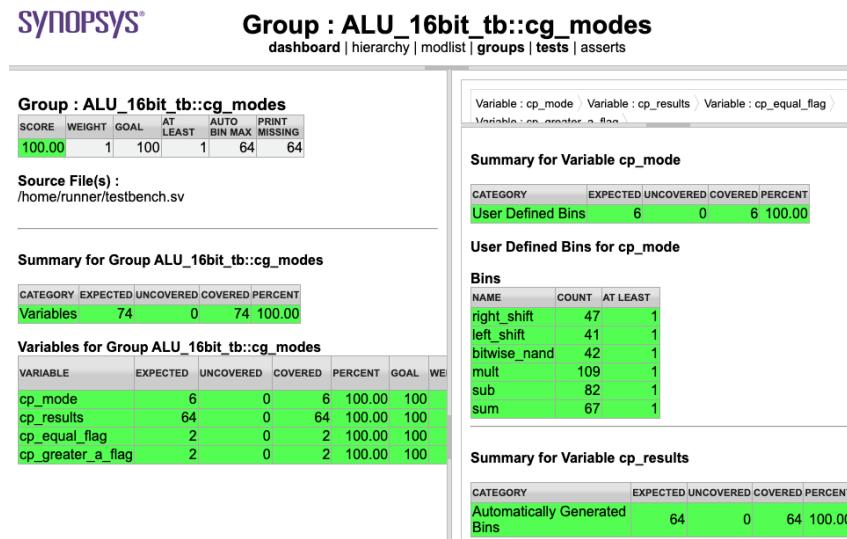


Figure 8.9: Example of output running a bash file

The picture above exemplifies a powerful way that EDA Playground can be used. The photo shows a verification performed for an ALU. The testbench was about checking which modes were used from the ALU, which can be seen on the bins section on the right side. Each bin represents a unique task performed by the ALU such as multiply, subtract, add, right shift and left shift. We can have a good visual by the green cells identifying that these operations were performed at least once during the test verification.

On the left side of the photo, we can also see the variables were identified from the design. This specific ALU design had flags to identify when variable a was equal to variable b and there was also another flag to identify when variable a was the same as variable b. We also have the green cell saying that at least once in our code these flags were high which is important when making sure that we have good coverage from the design code.

Good coverage from the code means that, realistically, you want all your code to be tested but it is very unlikely due to design complexity. Since this was a simple example, we were able to see that these flags were high at least once. We can also observe from the bottom right part of figure 9.3 that I separated bins for different values for the variables. This is important because the inputs a and b were 32 bits long, they can obviously be in a very large range, so I split into different bins so we can see the behavior of inputs a and b at different ranges.

Our group collectively with our advisor and Northrop Grumman have agreed to do work on Vivado Suite for our design because of technical issues discussed in a different chapter regarding accessing VLSI server remotely to access Cadence Xcellium and other tools. Below we will investigate the power and the features that Vivado brings to the table and why it is such a powerful tool to be used.

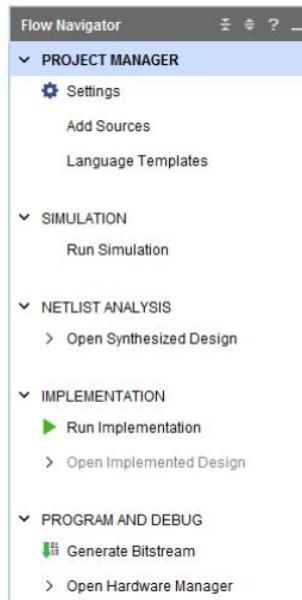


Figure 8.10: Flow Navigator for Third-Party Netlist Project

This section once it is complete, the designer is able to open the designs and to analyze the results and apply the necessary constraints. The tools such as Open Elaborated Design, Open Synthesized Design and Open implemented design are crucial and essential to the development of the design.

The Open Elaborated Design features allows a very high-level view of the design once all the hierarchies are implemented. Then, Open Synthesized Design, allows the designer to look deeper in post-synthesis, this is the place that high level logic is translated to gate level netlist. This is a very important process to our design because the netlist is actually a deliverable requested by Northrop Grumman and we were only able to design a core that is synthesizable. Lastly Open Implemented Design has the detailed view of the design after place-and-route.

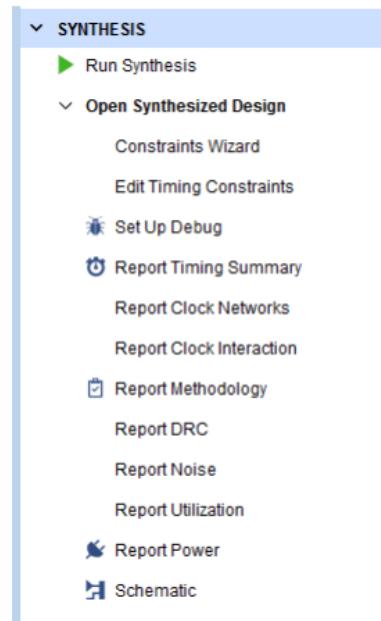


Figure 8.11: Synthesis Section in the Flow Navigator

Looking at the capabilities from the Vivado Suite, this is the method used by Vivado to transition the design from a Hardware Description Language to gate level representation. The starting point of this process is to firstly run synthesis, which checks for any syntax errors in the code, it follows design hierarchies to check for flow and it will also apply the constraint file which once is complete, it is ready for further optimization as we will observe above.

The constraint wizard is a helpful tool that simplifies the management of constraints which can be very cumbersome with a complex design. This tool will essentially guide the designer through the set up of timing, I/O standards. For beginners, which is the case of all the members comprising this group, to create a comprehensive constraint management.

Edit the timing constraints allows the designer to directly edit the timing constraints, that particularly includes clock periods, Input and Output delays as well as multi-

cycle path. These timing constraints are essential to meet design operational frequency and avoids timing violations.

Set Up Debug is a tool in Vivado Suite that integrates debugging logic to capture and analyze all the internal signals for the duration of the operation. According to its functionality, it is helpful when it comes to diagnosing functional or timing issues in the synthesized design because it provides visibility into the behavior of what is happening internally.

Report Timing Summary is the report that provides high level overview of the design timing performance which includes how it achieved clock frequencies and the slack values. This report will show the designer in which modules require greater optimization.

Report Clock Interaction will be looking into how different clock domains and will provide more information about what synchronization issues such as there are some domain crossings that could require additional logic to work properly.

Name	Available	Prohibit	Ports	I/O Std	Dir	Vcco	Bank	Bank Type
All Pins (1156)								
> I/O Bank 0 (23)	0	<input type="checkbox"/>						Dedicated
> I/O Bank 14 (56)	50	<input type="checkbox"/>						High Performance
> I/O Bank 15 (57)	50	<input type="checkbox"/>						High Performance
> I/O Bank 16 (56)	22	<input type="checkbox"/>				1.800		High Performance
> I/O Bank 17 (56)	10	<input type="checkbox"/>				1.800		High Performance

Figure 8.12: Package IO Pins

When designing something in Vivado that requires the designer to know which bank has the input or output that the designer could be looking for. For example, in the ZCU10 Ultrascale + board, if the designer is looking to use the Gigabit Transceivers channel from the board itself, then the user would investigate the documentation and find out that bank 230 are the connections of the GTH interface with the FPGA board.

```

Tcl Console
synth_design -rtl -name rtl_1
Command: synth_design -rtl -name rtl_1
Starting synth_design
Using part: xc7u035-fbv900-2-e
Top: bft
-----
Starting RTL Elaboration : Time (s): cpu = 00:00:03 ; elapsed = 00:00:03 . Memory (MB): peak = 2774.691 ; gain = 0.000
-----
INFO: [Synth 8-638] synthesizing module 'bft' [C:/work/project_bft/project_bft.srcs/sources_1/imports/Sources/bft.vhd:52]
WARNING: [Synth 8-614] signal 'wbInputData' is read in the process but is not in the sensitivity list [C:/work/project_bft/project_bft.srcs/source_1/imports/Sources/bft.vhd:122]
WARNING: [Synth 8-614] signal 'wbInputDataStage0' is read in the process but is not in the sensitivity list [C:/work/project_bft/project_bft.srcs/source_1/imports/Sources/bft.vhd:122]
INFO: [Synth 8-638] synthesizing module 'round_1' [C:/work/project_bft/project_bft.srcs/sources_1/imports/Sources/bftLib/round_1.vhd:50]
INFO: [Synth 8-638] synthesizing module 'coreTransform' [C:/work/project_bft/project_bft.srcs/sources_1/imports/Sources/bftLib/core_transform.
Parameter DATA_WIDTH bound to: 16 - type: integer

```

Figure 8.13: TCL console in Vivado.

The tool command language Console in Vivado is used for many purposes such as command execution, automation, customization, debugging, creating clocks and real time feedback. In the TCL console, the designer is able to perform actions, run synthesis, implementation, generate reports. In the TCL console the user is also able to automate tasks by writing TCL scripts as some designers use these scripts to set up constraints, generate bitstream.

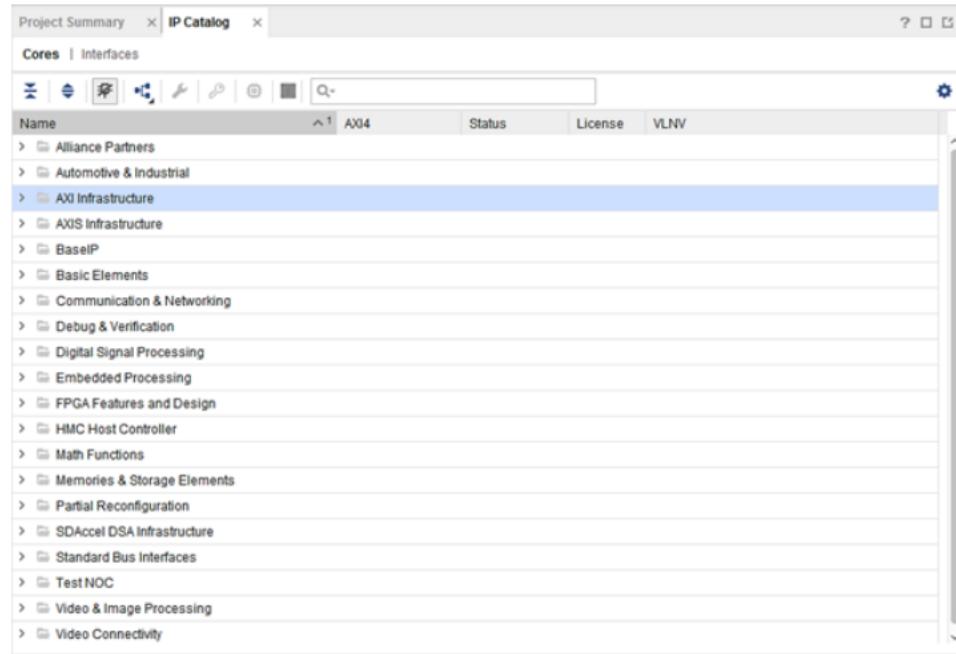


Figure 8.14: IP catalog

The IP catalog in Vivado is a huge asset. The IP catalog is a previously designed Intellectual Property (IP) cores that are often integrated in FPGA design. There are many different components in the IP catalog such as arithmetic modules, communication protocols, memory controllers and all of them serve to improve the development of our design.

As an example, designers are able to include Ethernet MACs, PCIe interfaces, or AXI components directly to the project. There are some useful IP cores that have built in testbenches which is very useful since the all the parts have been tested, which means the designer is able to use the current testbench and modify it to its own needs.

Name	Direction	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type	Pull Type
All ports (71)											
> wbInputData (32)	IN			<input checked="" type="checkbox"/>	65	LVCMS18	1.800			NONE	
> wbOutputData (32)	OUT			<input checked="" type="checkbox"/>	64	HSLVDCI_15	.800				
						HSLVDCI_18					
Scalar ports (7)											
bftClk	IN		AD16	<input checked="" type="checkbox"/>	64	HSTL_I	.800			NONE	
error	OUT		AC14	<input checked="" type="checkbox"/>	64	HSTL_II	.800			NONE	
reset	IN		AA19	<input checked="" type="checkbox"/>	64	HSTL_II_18	.800			NONE	
wbClk	IN		AE16	<input checked="" type="checkbox"/>	64	HSTL_I_12	.800			NONE	
wbDataForInput	IN		AB19	<input checked="" type="checkbox"/>	64	HSTL_I_18	.800			NONE	
wbDataForOutput	OUT		AD14	<input checked="" type="checkbox"/>	64	HSTL_I_DCI	.800			NONE	
wbWriteOut	IN		Y16	<input checked="" type="checkbox"/>	64	LVCMS18	1.800			NONE	

Figure 8.15: I/O Ports Window

The input and output ports are a very critical component in the FPGA design, and they interface between FPGA and the external device. This IO Ports is very useful because if all the logic is synthesized but the ports are chosen incorrectly, then the sys will simply not work. The IO potts will display all of the ports declared in the code and in this window, you are able to quickly identify all the ports and assign in to the correct port. In this window you are also able to see the voltage and the designer also has the chance see the voltage operating the port.

Messages											
<input type="button" value="Show All"/> <input type="checkbox"/> Warning (571) <input type="checkbox"/> Info (927) <input type="checkbox"/> Status (229)											
Synthesis (526 warnings, 838 infos)											
[Common 17-1460] Use of init.tcl in C:/Users/dpeascoe/AppData/Roaming/Xilinx/Vivado/init.tcl is deprecated. Please use Vivado_init.tcl [Common 17-349] Got license for feature 'Synthesis' and/or device 'xc7v585t-fgg1157' > [Synth 8-638] synthesizing module 'top' [top.v:49] (101 more like this) > [Synth 8-256] done synthesizing module 'IBUFG' (#1) [unisim_comp.v:14894] (101 more like this) > [Synth 8-155] case statement is not full and has no default [mgtTop.v:994] (10 more like this) > [Synth 8-3848] Net rom in module/entity ROCKETIO_WRAPPER_TILE_GT_FRAME_GEN does not have driver. [rocketio_wrapper_tile_gt_frame_gen.v:94] (99 more like this) > [Synth 8-3936] Found unconnected internal register 'rx_data_ram_r_reg' and it is trimmed from '80' to '32' bits. [rocketio_wrapper_tile_gt_frame_check.v:389] > [Synth 8-6014] Unused sequential element RXENMCOMMADET_OUT_reg was removed. [rocketio_wrapper_tile_gt_frame_check.v:335] (99											

Figure 8.16: Message Window in Vivado.

Another great feature from Vivado that is heavily used by all the members to see what the program is trying to tell the designer. The message log which can also be found in files inside the program folder will say all the information and warnings regarding the code. Mostly, designers will look for the errors that will the designer what are the most urgent fixed he needs to make in order to make the code synthesizable.

8.2 Introduction to Benchmarking

Benchmarking is a critical process when it comes to evaluating and comparing the performance of the existing and redesigned CPU cores. The approach to benchmarking requires the running of carefully designed programs that test

specific aspects of the core's functionality, such as arithmetic operations, bit manipulation, and branch handling. In these cases tested, optimal case testing, benchmarks like cumulative sum and XOR operations are used because they lack data dependencies and allow for sustained instruction throughput. It has been determined by the design team that these tests emphasize the best-case performance of the core by maintaining continuous utilization of the pipeline with minimal stalls. In a similar fashion the arithmetic progression summations and bit rotation tests focus on predictable workloads that leverage specific functional units of the CPU while avoiding complex control flow or dependencies. This enables a clearer understanding of the microarchitecture's efficiency under ideal conditions and also demonstrates the need for further verification on worst case scenarios and more complex instruction handling.

Unfortunately, optimal cases alone do not give us the full scope of performance. The worst-case scenario testing is equally important to reveal the limiting factors and bottlenecks in both the RAPID and RAPID-X cores. The worst-case benchmarks should introduce scenarios that really stress the core by having frequent stalls, hazards, and dependencies. This can be done with programs with nested dependencies, and memory bottlenecks. An example of this would be a nested loop with additions that can be expanded to include data dependencies between inner and outer loops to simulate workloads that challenge pipeline interlocking mechanisms and hazard scenarios. In testing scenarios that involve large frequent memory accesses or random access patterns the team could expose inefficiencies in memory hierarchies and the cache subsystem.

To implement worst-case testing, benchmarks would need to include varying instruction mixes. This includes high branching frequencies, and deliberate NOP insertions to create pipeline hazards. By comparing the performance metrics, such as CPI execution time, and power consumption across these worst-case workloads, the benchmarking process provides insights into the robustness of the original and redesigned cores. We should expect these findings to enable more targeted optimizations to address inefficiencies and improve overall system performance, making benchmarking a comprehensive tool for architectural evaluation. Below, we will discuss our initial benchmarking and our results. Such results contain the programs used and the graphs comparing performance metrics such as CPI. Worst-case scenario testing is expected to further develop deeper in the verification phase.

8.3 RAPID CPI Computation

To calculate the **Cycles Per Instruction (CPI)** for a pipelined CPU, we use the following formula:

$$CPI = \frac{\text{Total Clock Cycles}}{\text{Total Instructions}}$$

1. For the RAPID microarchitecture, an instruction is consumed every 3 clock cycles which is the pipeline initiation interval
2. And each instruction takes 12 clock cycles to complete execution (latency of pipeline); excluding memory operations.

In a pipelined processor, multiple instructions overlap in execution, so the CPI is determined primarily by the initiation interval; this is not the full latency. Therefore:

$$\text{CPI} = \text{Initiation Interval}$$

$$\text{CPI} = 3 \text{ (RAPID CORE)}$$

$$\text{CPI} = 1 \text{ (RAPID-X CORE)}$$

However, the CPI computation is not as simple as shown above due to the addition of NOP instructions complicating the computation because they inflate the instruction count with respect to clock cycles. The impacts of NOP are based on the current executing program. To provide a better perspective on the actual performance of the microarchitecture, we ran several programs to compare the performance of both microarchitecture as well as the reference microarchitecture.

8.4 CPU Core Benchmarks

8.4.1 Cumulative Sum and XOR Operations

This program computes a cumulative sum and XOR for a range of integers, demonstrating arithmetic and bitwise operations.

8.4.2 Justification

1. Involves a mix of addition and XOR, two commonly used operations in digital systems.
2. No data dependencies between instructions make it suitable for performance testing.
3. Provides a large loop iteration count for sustained instruction flow.

8.4.3 Program

```

1.      li t0, 0          # Initialize sum to 0
2.      li t1, 0          # Initialize XOR result to 0
3.      li t2, 10000       # Loop counter (large number)
4.      li t3, 1          # Increment value
5. loop: add t0, t0, t3   # Add increment to sum
6.      xor t1, t1, t3   # XOR current value with t1
7.      addi t3, t3, 1    # Increment current value
8.      addi t2, t2, -1   # Decrement loop counter
9.      bne t2, x0, loop  # Repeat if counter not zero

```

8.4.4 Bit Rotation Test

This program rotates bits in a register by shifting and performs OR operations for aggregation.

8.4.5 Justification

1. Focuses on bit-level manipulation and logical operations.
2. Exercises the shift unit in the CPU pipeline extensively.
3. Avoids complex data dependencies, enabling isolated performance evaluation.

8.4.6 Program

```
1.      li t0, 0x80000000 # Initial value with high bit set
2.      li t1, 0           # Accumulator for rotated values
3.      li t2, 1000        # Loop counter
4.      li t3, 1           # Shift amount
5. loop: sll t4, t0, t3  # Logical shift left t0 by t3
6.      or t1, t1, t4    # Accumulate result with OR
7.      addi t3, t3, 1    # Increment shift amount
8.      andi t3, t3, 31   # Wrap around shift to 0-31
9.      addi t2, t2, -1   # Decrement loop counter
10.     bne t2, x0, loop  # Repeat if counter not zero
```

8.4.7 Nested Loop with Additions

This program uses nested loops to accumulate sums, simulating workload patterns seen in some real-world nested computations.

8.4.8 Justification

1. Provides a mix of inner and outer loops for evaluating branch prediction.
2. Tests the handling of nested data dependencies and instruction execution order.
3. Generates a significant number of instructions for performance measurement.

8.4.9 Program

```
1.      li t0, 0           # Outer sum accumulator
2.      li t1, 10          # Outer loop counter
3.      li t2, 0           # Inner sum accumulator
4.      li t3, 1000         # Inner loop counter
5.      li t4, 1           # Increment value
6. outer: li t5, 1000      # Reset inner loop counter
7. inner: add t2, t2, t4   # Add increment to inner sum
8.      addi t5, t5, -1    # Decrement inner loop counter
9.      bne t5, x0, inner  # Repeat inner loop
10.     add t0, t0, t2      # Add inner sum to outer sum
11.     addi t1, t1, -1    # Decrement outer loop counter
12.     bne t1, x0, outer  # Repeat outer loop
```

8.4.10 Arithmetic Progression Sum

Calculates the sum of an arithmetic progression in a loop using basic additions.

1. Uses simple addition operations that are representative of many real-world applications.
2. Predictable data flow and low instruction complexity allow for clear performance insights.
3. A large loop ensures sustained load on the instruction pipeline.

8.4.11 Program

```
1.      li t0, 0          # Initialize sum to 0
2.      li t1, 1          # Starting value of progression
3.      li t2, 10000       # Number of terms
4.      li t3, 1          # Increment value
5. loop:   add t0, t0, t1    # Add current term to sum
6.           add t1, t1, t3    # Increment term
7.           addi t2, t2, -1    # Decrement loop counter
8.           bne t2, x0, loop  # Repeat if counter not zero
```

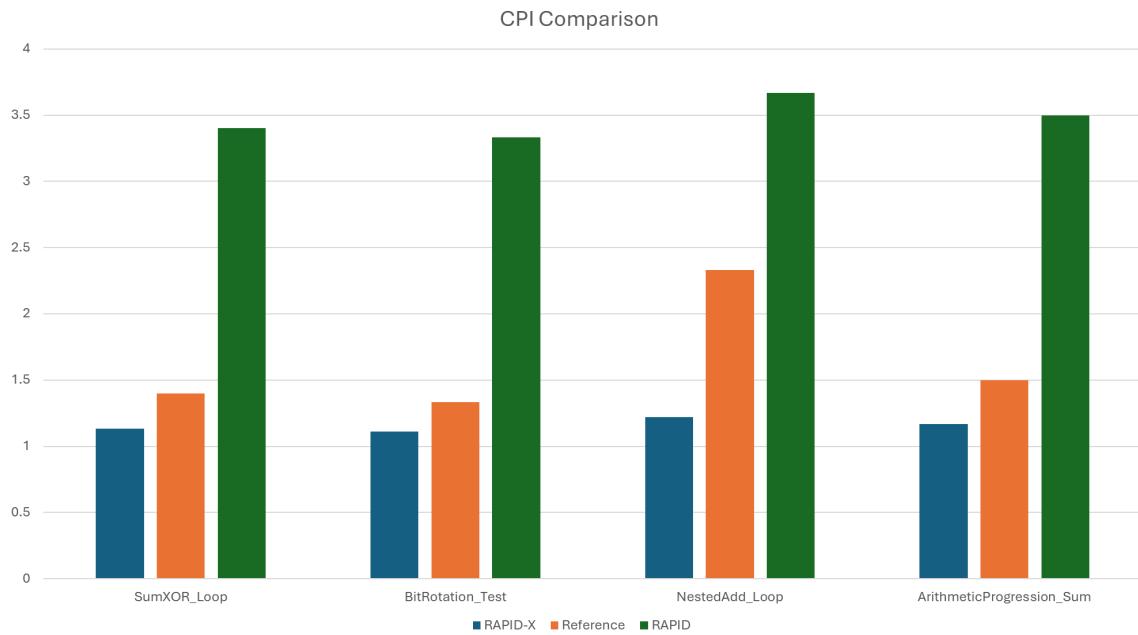


Figure 8.17: CPI Comparison

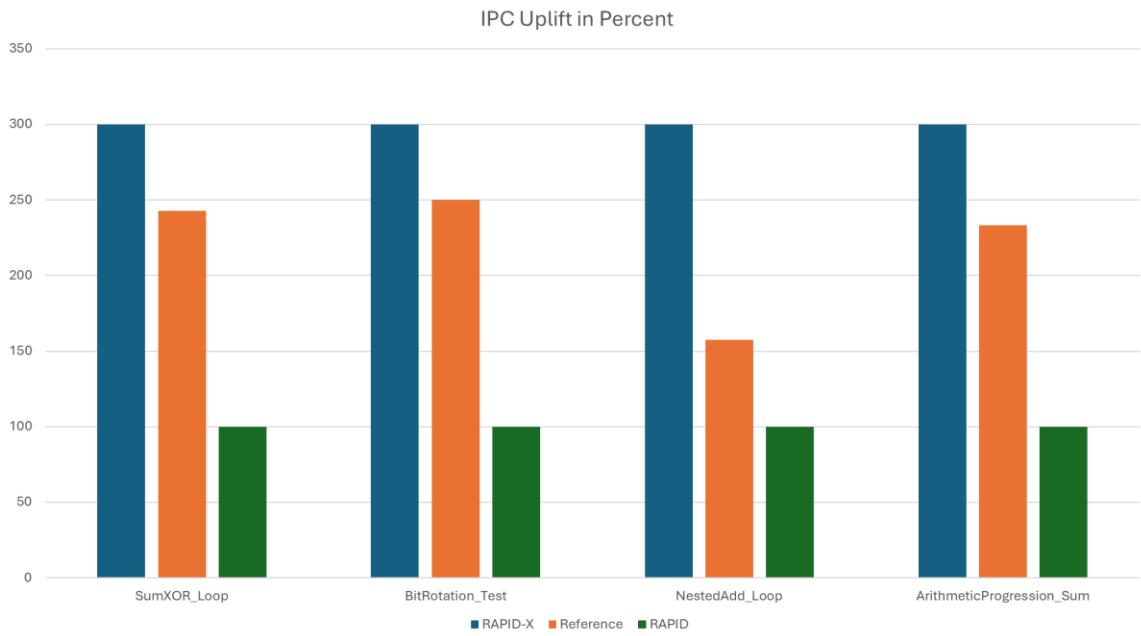


Figure 8.18: IPC Uplift Percent

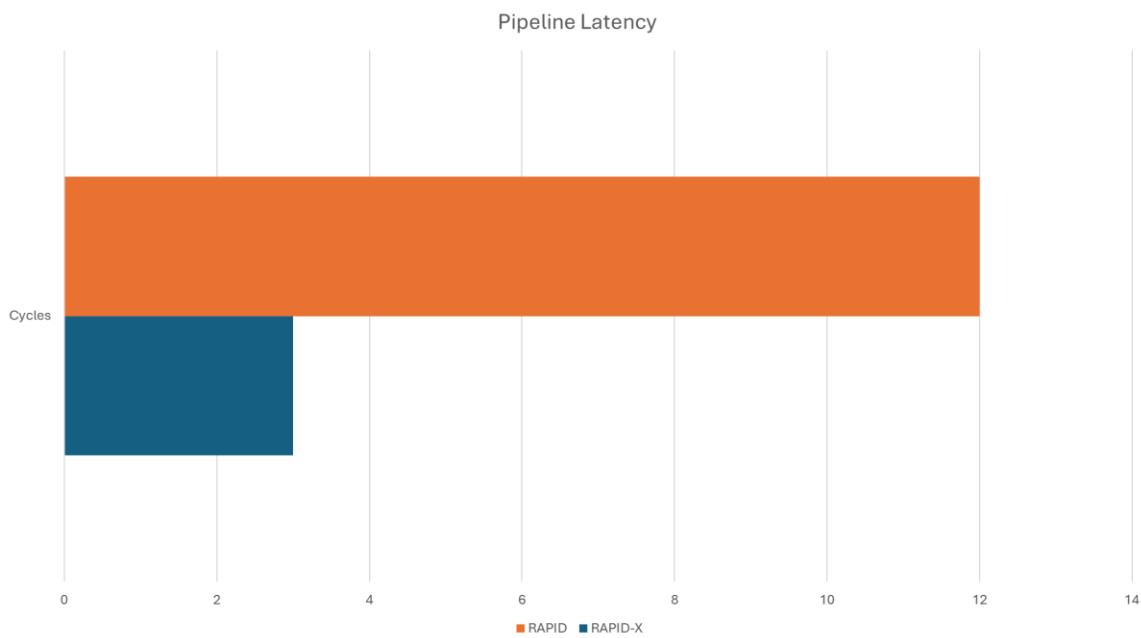


Figure 8.19: Pipeline Latency

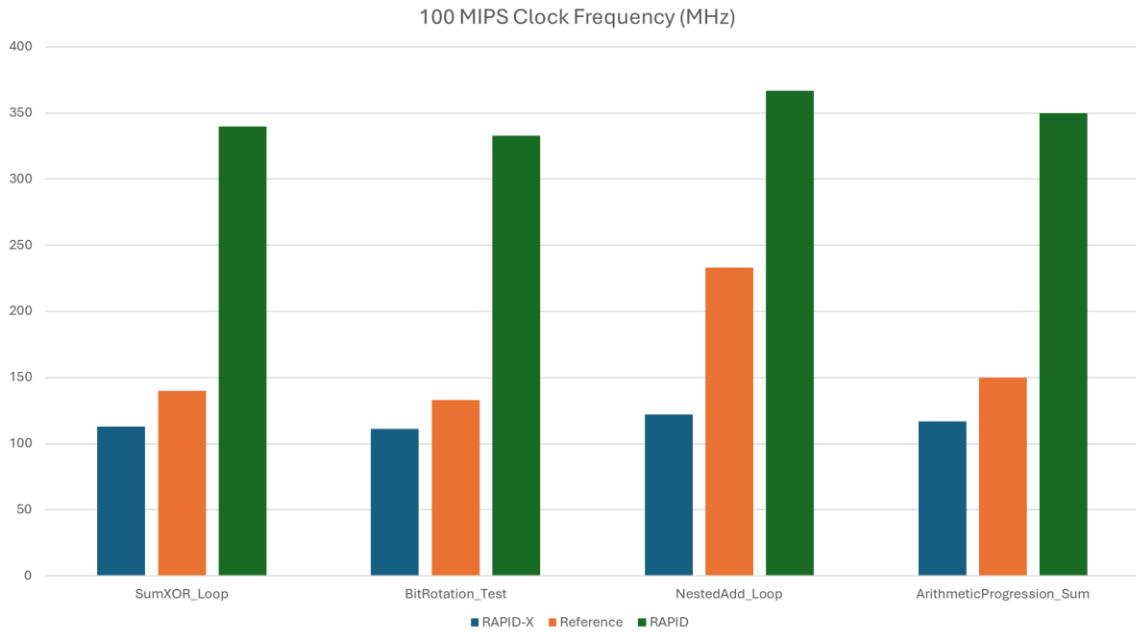


Figure 8.20: 100 MIPS Clock frequency.

8.5 Davi's experience

To better start comprehending the process of how Universal Verification Methodology works, I needed to go back and understand what Object-Oriented Programming is all about and better understand its core concepts.

Object Oriented Programming is a programming paradigm that is centered around “objects”, which are the instances of classes that have inside data and how the data will behave.

Unlike Procedural programming, which is more focused on the sequential execution of tasks, OOP was designed to organize the design around the interaction of these objects. The purpose of OOP is to make the code more modular, reusable and easier to maintain just like UVM.

When studying OOP, I needed a better understanding on some key principles such as encapsulation, inheritance, polymorphism and abstraction. Altogether, these principles enable the designers to create flexible software.

The first concept I needed to better understand is Inheritance. Inheritance is the mechanism that allows for one class to inherit the properties and behaviors. The class that inherits is a “subclass” and the class that inherits is called “superclass” or “base class”. This concept is essential for the reusability and establishes the connection between the classes.

Regarding Inheritance, the most important thing to keep in mind is that you are creating sub classes that will have the same characteristics as the parent. The sub class may have its own characteristics as well.

Polymorphism enables the object to be treated as instances of their parents' class rather than its own class. Encapsulation ensures that the object's internal state is hidden from outside influences, by only making observable in a well defined interface. Lastly, Abstraction is used to simplify complex systems by modelling the classes to the problem and focusing on which data and behaviors are the most relevant and hiding the unnecessary details.

Diving a little deeper into classes and objects which are the building blocks of OOP. Classes are the blueprint that defines the structure and behavior of the object. The class is the data and the properties of the object. When a class is defined, the designer can establish new data types and is able to create multiple objects inside.

An object is an instance of a class. When the class is initially defined, there is no memory associated to it until an object is created. Each object maintains its own state and is able to interact with other objects

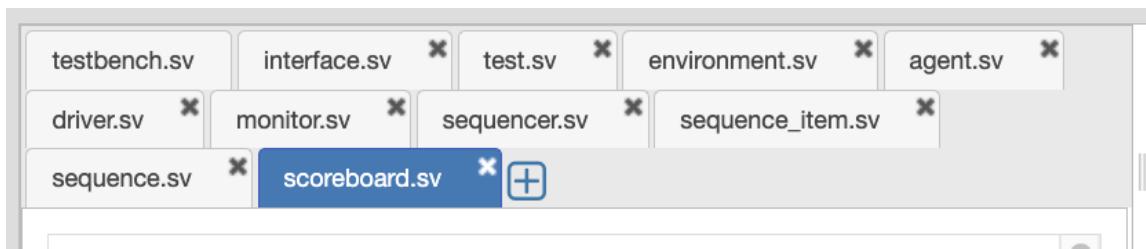


Figure 8.21: Shows the common files used for UVM testing

These tabs seen above are the accolade of our research. The research tabs in order are: testbench, interface, test, environment, agent, driver, monitor, sequencer, sequence_item, sequence and scoreboard

In UVM, the agents are the components responsible for handling the specifics of the verification process. The agent in our case has three subclasses: the driver, the sequencer and the monitor. By dividing into different subclasses, we can enhance the modularity of the verification environment.

The Driver within the UVM agent will be tasked with stimulating the Device Under Test by driving the input signals to the data from the design. The driver will also translate the high-level transaction generated by the sequencer into low level signal activities that the Device Under Test can interpret and respond to.

The sequencer will serve as the traffic controller within the Agent, and it organizes the flow of transactions that the Driver will apply to the Device Under Test. The sequencer will also manage the sequence of transactions, in our case the sequencer will be very simplistic.

The monitor is also very important as it observes the interactions between the Device Under Test and the testbench. The primary role from the monitor is to collect the information that will be used for analysis later in the scoreboard, so the monitor will be tracking the signal activities and will provide the necessary information to identify possible issues with the design code.

The environment in UVM is the top container that integrates all the Agents and manages their interactions with DUT. Within the environment is the verification setup. The environment also handles the interface interactions and provides the structure for the seamless integration of the different parts.

In the process after the skeleton for the UVM was created was to instantiate the Device Under Test in the testbench. The testbench was also the place to import uvm_pkg that will be used in other classes for the testbench. In the testbench I also instantiated the clock that the verification process will utilize.

In the interface I declared all the signals that shall be used in the verification. At first, When I tried to include the rapid.pkg into the design there were some issues with EDA Playground and how it was interacting and to solve that issue, I need to include all the signals from the package to be inside the design module.

Another big issue that needed to be resolved were the signals among the different classes. At times they were disagreeing with each other as far as data type, so I needed to ensure that the data types and lengths were matching across different classes.

```

81
82 // Compute expected value based on the operation
83 case (curr_trans.fcs_opcode)
84     3'b000: expected = curr_trans.iop ? ($signed(curr_trans.rs1) -
$signed(curr_trans.rs2)) : ($signed(curr_trans.rs1) +
$signed(curr_trans.rs2));
85     3'b001: expected = curr_trans.rs1 << curr_trans.rs2;
86     3'b010: expected = ($signed(curr_trans.rs1) <
$signed(curr_trans.rs2)) ? 1 : 0;
87     3'b011: expected = ($unsigned(curr_trans.rs1) <
$unsigned(curr_trans.rs2)) ? 1 : 0;
88     3'b100: expected = curr_trans.rs1 ^ curr_trans.rs2;
89     3'b101: expected = curr_trans.iop ? ($signed(curr_trans.rs1) >>
$signed(curr_trans.rs2)) : ($signed(curr_trans.rs1) >>
$signed(curr_trans.rs2));
90     3'b110: expected = curr_trans.rs1 | curr_trans.rs2;
91     3'b111: expected = curr_trans.rs1 & curr_trans.rs2;
92     default: expected = 32'hx;
93 endcase

```

Figure 8.22: gives the picture of how expected values were calculated in the scoreboard

In the image above we can see how the expected values were calculated in the scoreboard. It takes whatever the inputs were on the Device Under Test and it actually implements its own logic to obtain a result. Once the expected value is calculated, the scoreboard will compare this value to what the monitor was observing the outputs were in the design.

In the process of building the Universal Verification Methodology I needed to learn more about System Verilog Assertions which is still a powerful feature from SystemVerilog language that should be helpful in improving the verification process.

```

property hash_delay_p;
  @(posedge clk) a ##2 b;
endproperty

hash_delay_chk: assert property (hash_delay_p);

```

Figure 8.23: Simple delay assertion.

The figure above described as property hash_delay_p will check if signal a is asserted high on each positive edge of the clock cycle. Once input a has been high for 2 clock cycles represented by ##2.

In this example above, there are two situations in which the assertion will fail which is any cycle which input a is not high. The second way it will fail is if after 2 cycles of input a being high and b is not high every other 2 clock cycles.

8.6 Evan's Experience

This semester's main goal was not focused on testing, but we were able to get a jump start on testing our processor core. This includes using traditional testbenches as well as UVM testbenches to validate our design. While the primary focus of this semester was focused on designing the core, I did begin early stages of verification for the instruction decode logic portion of the pipeline. My testing goal for this semester included gaining a strong foundation in the UVM framework as well as trying to complete the initial stages for the instruction decode testbench.

The UVM methodology offers a structured approach for validating hardware designs by offering comprehensive error detection, while having a focus on reusability. Even for a relatively simple core like ours which is only focused on integer-based instructions, being able to correctly analyze the signals coming in and out of the decoder logic state is vital to ensure the proper functioning of our core. This methodology supports randomization and coverage-driven verification which will allow us to test all cases including edge cases which can be easily overlooked. While we are in the early stages of UVM testing, it has provided valuable insights into the bugs within our designs.

Initially, my main focus was to gain a deep understanding of how the methodology works. This includes gaining a comprehensive understanding of the framework and all of the components involved in writing a comprehensive UVM test. This includes the core concepts such as drivers, monitors, environments, scoreboards and more. This was a new technology for me so there was a long learning curve that involved many tutorials as well as delving into the documentation for the framework and gradually I applying what I have learned to the instruction decoder testbench. I then focused on writing a comprehensive random instruction generator which was a bit tricky than I expected.

Although the UVM tests are in the early stages, significant progress has been made in both the learning phase as well as the initial stages of the instruction decoder testbench. Although not complete, the testbench shows promising signs for our design. However, this is still much work needed in the testing phase to ensure our design is comprehensive. Overall, we got a jump start on UVM testing for this semester and have gained a solid foundation in how the UVM framework works.

Learning the UVM Framework

Learning the UVM framework was one of the most critical and time-consuming aspects of the testing process this semester. As someone who is new to UVM, I first started by understanding the core problems UVM aims to address. This includes structured testing, reusability, and coverage-driven verification. After familiarizing myself with the UVM environment conceptually, I found tutorials to bridge the gap between the theory of the framework and a practical test. One of the most helpful tutorials Davi shared with me included writing a UVM testbench for a simple arithmetic logic unit (ALU). This tutorial walked me through the entire UVM design process including drivers, monitors, agents, sequencers, scoreboards. By following this tutorial, I was able to gain a much deeper understanding of how all the pieces of the methodology come together to form a comprehensive testbench.

With this new knowledge, I then started to develop an initial UVM testbench for the instruction decode logic stage of our processor. This testbench required integrating the random instruction generator with the UVM environment, creating sequences to supply test scenarios, and monitoring outputs to verify the decoder's accuracy. The early stages of this testbench highlighted the decoder's ability to correctly parse instructions and assign control signals, but progress was slowed due to a redesign of the processor core. The redesign heavily altered the signals required in the stage that I needed to restart from scratch. This was however necessary as the original design was not working as intended.

Despite these setbacks, the learning process was still very useful as I can reuse a lot of the initial set up for the new design which is what I am currently doing now. It provided not only a solid introduction to UVM principles but also shows the iterative nature of testing in hardware design. While the instruction decode testbench is still in its early stages, the groundwork laid this semester positions the project for more robust and comprehensive testing in the next phase and puts us ahead for the next semester.

9 Administrative Content

9.1 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.

9.2 Initial 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

9.3 Initial 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 Desing 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

9.4 Work Distributions

Table 61: Group 1 members and responsibilities

Group 1	Responsibilities
Youssef Samwel (ECE) Team 1 lead	Instruction Fetch Unit (IFU) Instruction Decode Unit (IDU) Develop logic to fetch instructions from memory.
Nicolas Sayed (ECE) Team 1 member	Develop Execution Unit and Register File Design the ALU Implement Logic to update registers
	Responsibilities Designing the Memory Access Unit Ensure load/store functions properly Implementation of cache policies Design Control and Status Registers Focus on Bus Interface

Table 62: Group 2 members and responsibilities.

Group 2	Responsibilities
Davi Dantas (ECE)	Designing the Memory Access Unit Ensure load/store functions properly

Team 2 lead	Implementation of cache policies
Timothy Ogg (CPE) Team 2 member	Design Control and Status Registers

Table 63: Group 3 members and responsibilities.

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

9.5 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 adjustable 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 (weekly manner), 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. After retrospective meetings after each sprint improvement can be more definitive and frequent which fosters an environment for learning and improvement. Which goes 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.

9.6 Deliverable Requests

9.6.1 Overview of First Deliverable Request

Our first division of labor and research practice milestone requests centered around the initial research phase as well as a 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.

9.6.2 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.

9.6.3 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.

9.6.4 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.

9.6.5 Division of Labor for initial Block Design

The expectation for this deliverable request was not for every member to produce 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.

9.6.6 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.

Table 64: Detailed Tasks for Teams

	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"> • Design and implement the Execution Unit, including the ALU 	<ul style="list-style-type: none"> • Design and implement the memory access unit to handle memory 	<ul style="list-style-type: none"> • Design and implement the Instruction Fetch Unit and the Instruction

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

		instruction fetch,decode and execution stages from Group 1 and Group 2	
--	--	--	--

9.6.7 Revised Timeline (Adjusted for setbacks)

Table 65: Revised Timeline

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

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

9.6.8 Key Milestones and Specified Division of Labor

Table 66: Key Milestones

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

<ul style="list-style-type: none"> • Dec 2-10: Final verification and final presentation 	<ul style="list-style-type: none"> • Nicolas (Team CPU)
---	--

9.7 Final Deliverable Request for Semester

The following deliverable request outlines the deliverables, objectives, and team responsibilities for the full RTL Verification and Debugging phase of the project. The goal was to ensure that functional correctness and integration issues within RTL models were addressed before moving on to further verification and synthesis testing. Throughout this deliverable request we were able to find race conditions, perform core redesigns and fix bugs across multiple modules. The key activities for this deliverable include cross-verification of modules across the three groups, full system simulations, debugging and optimizations. The teams focused on verifying the redesigned 4-stage and ongoing 5 stage pipeline models addressing critical design flaws such as any found race conditions and ensuring proper functionality. The deliverables that are expected are initial UVM cases, detailed documentation of the identified and resolved issues, and simulation results that validate pipeline behavior. As for the responsibilities, each distributed among each group, and each tasked with specific design, testing and debugging roles to ensure as comprehensive as possible and collaborative approach to the verification process and the redesigns. The milestone checkpoints are set to track progress with November 15 targeted for initial integration and testing, and November 23 for completing full system simulations and debugging. Here is a copy of the deliverable request formatted in the same manner it was sent to the group on November 5, 2025:

Deliverable Request: Full RTL Verification and Debugging
Deadline: November 24, 2024

The goal of this deliverable is to ensure all RTL models are initially verified and debugged. In doing so we address the functional correctness and integration issues before continuing to synthesis testing.

Objectives:

Complete Cross-Verification: All teams will swap pick 2 units non-matching for team memory and team verification. Team core will focus on debugging core units like decode and execute	All teams must conduct cross-verification of their respective modules for the redesigned core. This includes the 4-stage update. If the 5 stage is still being tested in initial testbenches, then save testbenches and results and work on the 4 stage redesign testbenches and UVM case setup.
--	---

stages. Verification will tackle all.	
Full System Simulation Team Core simulates. Team core and Team verification will simulate individual unit behavior.	Run full system simulations on both the 5 stage and 4 stage redesign. Make sure to document that the 4 stage corrects the race condition issues and a number of bugs. Expect bugs for the 4 stages as well and document them.
Debugging and Refinement Team core and Team memory will focus on debugging and making the small changes to design.	Debug the issues found from simulations and make sure the design team is consulted. Record each bug in mini reports or save simulations and logs for future reference.

Deliverable: Initial UVM cases, initial debug stage of the RTL redesign with detailed documentation of identified and resolved issues. Simulation results that demonstrate the working pipeline behavior and output for the test cases.

Milestone Checkpoints:

November 15, 2024 for the initial system integration and testing

November 23, 2024 complete full system simulation and debugging

END OF DELIVERABLE REQUEST

It's important to note that this deliverable request was made with the collaboration of two other team leads Davi Dantas and Youssef Samwel and discussed with the rest of the group. This deliverable request was developed on a dynamic approach of the team's development across the months of November and end of October. This showcases how a project like this was subject to changing requirements per its development. The result of the deliverable was accomplished with initial UVM cases being produced, full core simulation and debugging, and effective redesigns. A big triumph for the team in the final stages of the project for the semester.

9.8 Finalized Work Distributions Adjusted for Redesigns and Productivity

Table 67: Group 1 Updated Work Distribution

Group 1	Responsibilities
Youssef Samwel (ECE) Team 1 lead	Instruction Fetch Unit (IFU)
	Instruction Decode Unit (IDU)
	Control Signal Unit System level design
Nicolas Sayed (ECE) Team 1 member	Initial test benching for all modules designed
	Execute Stage Unit
	Register File Unit

Table 68: Group 2 Updated Work Distribution

Group 2	Responsibilities
Davi Dantas (ECE) Team 2 lead	Execute Stage test benching
	UVM case setup for Execute Stage
	Memory stage cross verification
Evan Kasky (CPE) Team 3	UVM Case setup for Rapid and Rapid X core units
	Memory stage cross verification
	Memory stage Decoder Test Benching

Table 69: Group 3 Updated Work Distribution

Group 3	Responsibilities
Timothy Ogg (CPE) Team 2 member	Instruction Fetch System Level Testbenches
	Memory Stage System Level Testbench
	Initial UVM case setup for both stages above
Pablo Rodriguez (CPE) Team 3 Lead	Cache Subsystem Designer for I cache and D Cache. Integrate cache systems to Ifetch unit and Memory stage.

9.9 Plan for Senior Design 2

The goal for Senior Design 2 is to complete the development, verification and demonstration of a fully functional RV32I pipelined RISC-V processor. Here we transition from RTL design to a tapeout ready implementation. As of now, our extensive debugging and redesigns have increased the integrity of our design. However, further verification is required. The project will be divided next semester into 5 stages, ironically.

The first stage which will be the initial preparation will be between January 8-20, 2025. Here we will focus on reorienting the team and aligning efforts after winter break. The team should review deliverables from Senior Design 1, in this case the existing core and cache subsystem, to address incomplete modules and debugging, and to finalize UVM training and middle point to end point for UVM testing. This will ensure all members are proficient in UVM testing, a key development for us as students and for the project. Cadence tools such as Genus, Innovus, and Xcelium will be utilized along the simulation environments and will be the main tool chain for this stage. Weekly sprint tasks will continue and will be further outlined to create, once again, a detailed timeline. By the end of this stage, the team should expect to deliver an updated project timeline, with finalized RTL design, and evidence of UVM testing completion.

For stage two, advanced verification, January 21- February 29, 2025 will focus on completing functional verification using UVM and resolving functional and timing issues that are expected to arise in the RTL design. The team is expected to develop UVM testbenches for all pipeline stages (fetch, decode, execute and memory; writeback omitted due to redesign), ensuring coverage of edge cases, hazard testing, and pipeline stall scenarios. Then full pipeline integration testing will follow, including verification of the memory hierarchy and load/store unit interactions. All identified issues should be debugged, and optimization of the core should follow. Deliverables include comprehensive UVM testbenches, verification reports, and debugged RTL design.

In the third stage, Synthesis and Floorplanning, March 1 - March 18, 2025 will focus on synthesizing the RTL design into a gate-level netlist and initiating physical design with chip floorplanning. The team will perform RTL-to-GDS synthesis using Cadence Genus. The team is expected to optimize for area, power, and performance while also making sure we pay close attention to synthesis warnings and timing violations. Using Cadence Innovus, the floor planning process will define the die size, IO placement, and power grid design, including clock tree distribution and signal routing. We must also perform

preliminary power and timing analysis that will assess power consumption and timing performance. Deliverables will include a synthesized gate-level netlist, an initial floor plan layout, and a power and timing analysis report.

The fourth stage, Physical Design and Static Timing Analysis, March 18 - April 1 , 2025 involves completing the physical design and validating timing through Static Timing Analysis known as STA. The team should have standard cells place, completed signal routing, and verified connectivity. STA should validate timing performance across all paths, and any setup or hold time violations must be resolved. In addition to this power grid distribution will be analyzed and optimized, with additional focus on leakage and dynamic power analysis. Deliverables are a fully placed and routed design layout, and STA report and a power analysis report.

For the fifth and final stage, Verification and Sign-off, April 1 - 20, 2025 will focus on verifying the complete design against project requirements and finalizing the tapeout-ready design. The team must have full-system simulations that validate functionality, timing, and power performance at the gate-level. The final GDSII file will be generated for tapeout, and should be accompanied by thorough documentation of design protocols and performance metrics. The team should document similar to how documentation was performed in Senior Design 1; probably even more extensively. Additionally, the team will prepare for a project presentation to faculty and sponsors, which will include a live demonstration of the processor's functionality. Deliverables will include a validated GDSII file, a presentation slide deck and a demonstration setup.

With this plan we comply with the Agile methodology, and continue our structured and efficient approach to completing the RV32I RISC-V processor. From verification to final tapeout, while also meeting academic and industry level standards. Work distributions are expected to be developed the first week of the semester.

9.10 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 concern 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.

Conclusion

The team of 6 members embarked on a comprehensive journey to design and synthesize our very first Application-Specific Integrated Circuit (ASIC) implementing the RISC-V RV32I instruction Set Architecture (ISA). The end goal is yet to be reached, but it is proving to be hands-on industry-level experience in ASIC design by using state of the art electronic design automation tools while maintaining industry level standards. Not only was our cause and effort an academic exercise but a response to our passions and to the broader context of elevating the semiconductor industry talent in the United States. As of now our core can run the 37 instructions in the RV32I ISA, and according to our best case scenario benchmarks we can achieve 100 MIPS with a reasonable clock frequency. Additionally due to our parameterizable cache, we are able to reach the 16 MB cache requirement.

Throughout the multiple projects' phases, the team engaged in computer architecture, ISA breakdown, RTL design development, synthesis, and advanced verification. Due to our pre-selected RISC-V RV32I ISA, the team was able to review existing implementations and leverage existing technologies to determine ways of creating our novel implementation. By using a well-supported ISA, the project had smoother development and increased potential for adoption. The initial design constraints targeted a performance metric of 100 million instructions per second (100 MIPS), with an operating voltage range of 0.95 to 1.25 V and temperatures ranging from 0 degrees C to 125 degrees celsius while also consuming a maximum of 1 W of power. Originally, to meet these requirements the team opted for a modified 5 stage RISC pipeline over a single stage implementation. This was done to allow a higher throughput and more efficient instruction processing which is essential for achieving the desired MIPS rating.

In the process of researching existing products like the GreenWaves GAP8 and the Espressif ESP32-C3 and multiple open source github implementations we gained integral insight into common practices of commercial and open source implementations. These research dives allowed us to make decisions on cache subsystem design, cache organization, pipeline stages, and control unit design; the most complex areas for novel implementations. The incorporation of a 16 MB L1 Cache, for instance, was a given in the original project specifications but how we were to establish our cache for efficiency was up to us. Leaving us with our choice for Harvard Implementation, a split cache design, as our final choice. In selecting this design choice we enhanced performance by reducing memory access times.

One of the more complex aspects of the project was addressing inherent pipeline architecture interactions. For example, the handling of pipeline hazards such as data, control , and structural hazards. Team needed to fully understand in-depth what each hazard was, where they came from, and what caused them. In order to handle them our original designs used software implementations to handle

hazards but after multiple core redesigns the team did a hardware implementation of the forwarding unit and control unit designs across the entire pipeline and cache subsystem to handle improper behavior. A feat that was not expected at the start due to high complexity and considered a stretch goal but was accomplished. This allowed for an even more efficient core known as RAPID-X. The team's latest iteration of our novel core implementation.

Verification is an ongoing cornerstone of the project, with the team employing various verification methods such as Constrained Random Verification, Assertion-Based Verification, Coverage-Driven Verification, and the Universal Verification Methodology. Due to the complexity of these varying verification methods, verification has yet to fully conclude though significant progress has been performed, enough for the team to decide on multiple core redesigns. These techniques collectively ensured that we have yet to meet the required specifications and lacked reliability under diverse scenarios. This emphasis on verification was not only a commitment to the industry standards and practices, but an essential part in determining our core's functionality. Fortunately, multiple redesigns have allowed us to reach a point where the core can complete simple programs with expected and accurate behavior.

Time constraints were a huge challenge for the team as all members needed to complete extensive training on Cadence EDA tools and Rapid Adoption Kits. The team's proactive approach to undergo trainings for tool proficiency and focusing on actual project development was important. Deciding what trainings to focus on was also a requirement since there was not enough time to do all the trainings and also focus on project responsibilities. The team mitigated this by having specific members focus on trainings that focused on their responsibilities in the project allowing for a specialized approach in development and still meeting project milestones. This highlighted the importance of efficient time management and adherence to project milestones, essential skills for a fast-paced semiconductor development team.

The exploration of ChatGPT's capabilities and limitations offered a modern perspective on integrating artificial intelligence tools into engineering workflows such as ASIC development. ChatGPT proved useful in providing overviews and summarizing concepts, however, was limited in providing any sort of feedback for advanced verification and system level pipeline design. This is why the team relied so heavily on supplementary research and training packs as well as critical evaluation with subject matter experts from Northrop Grumman. This emphasizes the role of AI in engineering and why human expertise is still critical in the development of ASICs at present.

The project successfully achieved its goals of designing an ASIC implementing the RISC-V RV32I ISA while meeting three pivotal core accomplishments. Running all 37 instructions, reaching 100 MIPS and a parameterizable cache that can reach a size of 16MB. This experience provided team members with a very practical

experience in modern CPU architecture development, design methodologies, and verification techniques. There is still work to be done in verification and debugging as well as the additional stages in preparation for our mock tapeout. So far the team has hit most milestones listed in our timelines and is on track. The journey from conceptualization to a mock tapeout is halfway there and this was possible with consistent decision-making, problem-solving, and adherence to industry standards. Without such qualities navigating the challenges of technological choices, verification complexities, and design constraints, we would not have been able to produce a novel implementation of the RV32I ASIC.

Appendix A ChatGPT

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:

- a. 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:

- a. 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:

- a. 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:

- a. 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)

Appendix B References

- [1] **Arm Cortex-M4 Processor Technical Reference Manual Revision R0p1.** Accessed September 5, 2024. <https://developer.arm.com/documentation/100166/0001/Introduction/About-the-processor>.
- [2] **The RISC-V Instruction Set Manual Volume I - Version 20240411.** Accessed September 5, 2024. <https://github.com/riscv/riscv-isamain/releases/download/20240411/unpriv-isa-asciidoc.pdf>.
- [3] **Greenwaves GAP8 IoT Product Brief.** Accessed September 5, 2024. https://greenwaves-technologies.com/wp-content/uploads/2021/04/Product-Brief-GAP8-V1_9.pdf.
- [4] **ESP32-C3 Series Datasheet Version 1.8.** Accessed September 5, 2024. https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf.
- [5] **Muxup. Commercially Available RISC-V Silicon.** Accessed September 5, 2024. <https://muxup.com/2023q1/commercially-available-risc-v-silicon>.
- [6] Coelho, D. R. *The VHDL Handbook*. Springer Science & Business Media, 2012.
- [7] **Digital Logic Design Using Verilog.** Accessed October 25, 2024. [Online]. Available: <https://link.springer.com/book/10.1007/978-981-16-3199-3>.
- [8] **System Verilog Assertions and Functional Coverage.** Accessed October 25, 2024. [Online]. Available: <https://link.springer.com/book/10.1007/978-3-030-24737-9>.
- [9] **Constraint-Based Verification | SpringerLink.** Accessed October 25, 2024. [Online]. Available: <https://link.springer.com/book/10.1007/0-387-30784-2>.
- [10] **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, October 2023. doi: [10.1109/IEEESTD.2023.10287892](https://doi.org/10.1109/IEEESTD.2023.10287892).
- [11] **Intel® 64 and IA-32 Architectures Software Developer Manuals.** Intel. Accessed September 5, 2024. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.
- [12] Hennessy, J. L., Patterson, D. A., and Asanović, K. *Computer Architecture: A Quantitative Approach*. Elsevier, 2012.
- [13] **Documentation – Arm Developer.** Accessed September 14, 2024. [Online]. Available: <https://developer.arm.com/documentation/102202/0300?lang=en>.

- [14] A. E. Team. "The Official History of Arm." *Arm Newsroom*. Accessed October 5, 2024. [Online]. Available: <https://newsroom.arm.com/blog/arm-official-history>.
- [15] Golsha, Khosrow. *ASIC Design Implementation Process*. Springer Nature, Switzerland, 2025. ISBN: 978-3-031-58655-2.
- [16] **UVM (Universal Verification Methodology) Working Group**. Accellera. [Online]. Available: <https://www.accellera.org/activities/working-groups/uvm>.
- [17] Rabaey, J. M., Chandrakasan, A. P., and Nikolić, B. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 2003.
- [18] **Clock Skew in Synchronous Digital Circuit Systems**. GeeksforGeeks, October 22, 2020. [Online]. Available: <https://www.geeksforgeeks.org/clock-skew-in-synchronous-digital-circuit-systems/>.
- [19] **How to Eliminate Clock Jitter**. Cadence. [Online]. Available: <https://resourcespcb.cadence.com/blog/2022-how-to-eliminate-clock-jitter>.
- [20] Patterson, D. A., and Hennessy, J. L. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. 5th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.
- [21] **Ripes**. Accessed October 25, 2024. [Online]. Available: <https://ripes.me/>.
- [22] **FreePDK45**. North Carolina State University. Accessed October 25, 2024. [Online]. Available: <https://eda.ncsu.edu/freepdk/freepdk45/>.
- [23] **What is In-Memory Computing? IBM Technology**. February 22, 2022. Accessed September 8, 2024. [Online Video]. Available: <https://www.youtube.com/watch?v=BTnr8z-ePR4>.
- [24] **SystemVerilog DPI Tutorial**. Doulos. [Online]. Available: <https://www.doulos.com/knowhow/systemverilog/systemverilog-tutorials/systemverilog-dpi-tutorial/>.
- [25] **SystemVerilog DPI Verification Guide**. [Online]. Available: <https://verificationguide.com/systemverilog/systemverilog-dpi/>.
- [26] Sutherland, Stuart. "The Verilog PLI Is Dead (Maybe)... Long Live The SystemVerilog DPI!" [Online]. Available: https://sutherland-hdl.com/papers/2004-SNUG-presentation_Verilog_PLI_versus_SystemVerilog_DPI.pdf.
- [27] Munteanu, A. I. "How to Call C-functions from SystemVerilog Using DPI-C." AMIQ Consulting. [Online]. Available: <https://www.consulting.amiq.com/2019/01/30/how-to-call-c-functions-from-systemverilog-using-dpi-c/>.

Appendix C Code

Instruction Fetch Testbench:

```
1. `timescale 1ns / 1ps
2.
3. module tb_ifetch_with_cache;
4.
5.     import memory_controller_interface::*;
6.     import rapid_pkg::*;
7.
8. //Parameters
9.     localparam MEM_DELAY = 5;
10.    localparam MEM_SIZE = 262144; //1MiB
11.    localparam WORD_LENGTH = 32;
12.    localparam BLOCK_COUNT = 4096;
13.    localparam MCI_DATA_LENGTH = 128;
14.    localparam MCI_ADDR_LENGTH = 32;
15.    localparam XLEN = 32;
16.
17. //Clock and Reset
18.     logic clk, rst;
19.
20.
21. //Cache-Memory Communication Signals
22.     mci_request_t mem_req;
23.     mci_response_t mem_res;
24.
25. //CPU Instruction Fetch Signals
26.     logic [XLEN-1:0] i_ext_pc;
27.     logic i_pc_load, i_pipeline_ready;
28.     logic [XLEN-1:0] o_pc, o_instruction;
29.
30.
31. fake_memory memory (
32.     .i_clk(clk),
33.     .mem_req(mem_req),
34.     .mem_res(mem_res)
35. );
36.
37.
38. cpu_ifetch_unit ifetch (
39.     .i_clk(clk),
40.     .i_reset(rst),
41.     .i_ext_pc(i_ext_pc),
42.     .i_pc_load(i_pc_load),
43.     .i_pipeline_ready(i_pipeline_ready),
44.     .o_pc(o_pc),
45.     .o_instruction(o_instruction),
46.     .mem_req(mem_req),
47.     .mem_res(mem_res)
48. );
49.
50.
51. //Expected Instruction Sequence
52. logic [XLEN-1:0] expected_instructions[0:7];
53. initial begin
54.     expected_instructions[0] = 32'h00000293; //addi x5, x0, 0
55.     expected_instructions[1] = 32'h00100313; //addi x6, x0, 1
56.     expected_instructions[2] = 32'h01400393; //addi x7, x0, 20
57.     expected_instructions[3] = 32'h00628433; //add x8, x5, x6
58.     expected_instructions[4] = 32'h00030293; //addi x5, x6, 0
```

```

59.      expected_instructions[5] = 32'h00040313; //addi x6, x8, 0
60.      expected_instructions[6] = 32'hfff38393; //addi x7, x7, -1
61.      expected_instructions[7] = 32'hfe0398e3; //bne x7, x0, loop
62. end
63.
64. initial clk = 0;
65. always #5 clk = ~clk;
66. int i;
67. int current_pc;
68.
69. initial begin
70.     $display("Starting Instruction Fetch with Cache Testbench...");
71.
72.
73.     rst = 1;
74.     i_ext_pc = 32'h0000_0000; //PC 0
75.     i_pc_load = 0;
76.     i_pipeline_ready = 0;
77.     current_pc = $random;
78.
79.     @(posedge clk);
80.     @(posedge clk);
81.     @(posedge clk);
82.     @(posedge clk);
83.     rst = 0;
84.
85.     $display("Fetching and validating instructions:");
86.     i_pc_load = 1;
87.     @(posedge clk);
88.     i_pc_load = 0;
89.     //i_pipeline_ready
90.     i_pipeline_ready = 1;
91.
92.
93. end
94.
95. always @ (posedge clk) begin
96.
97.
98.     if (o_pc != current_pc && o_instruction !== 32'bx && o_instruction != 32'h00000033) begin
99.         current_pc = o_pc;
100.        $display("PC [%h]: Fetched Instruction [%h], Expected [%h]", o_pc, o_instruction, expected_instructions[i]);
101.
102.        if (o_instruction !== expected_instructions[i]) begin
103.            $error("Mismatch at PC [%h]: Fetched [%h], Expected [%h]", o_pc, o_instruction, expected_instructions[i]);
104.            //$/stop;
105.        end
106.        i++;
107.        if (i == 8) begin i_pc_load = 1; i_ext_pc = 0; $finish; end
108.    end
109. end
110.
111. endmodule
112.

```