ECE/CS 552 Spring 2025: Final Project Report

Due date: May 2, 2024, 11:59pm on Canvas

Team Members: Srivibhav Jonnalagadda, Josh Mlaker, Sandeep Sankar, Kailan Kraft

1. Overview

Our final project is a five-stage pipelined CPU implementing the WISC-S25 ISA. The pipeline consists of instruction fetch, decode, execute, memory, and writeback stages. Both instruction and data memories are accessed via 2KB, 2-way set-associative caches, each employing a Least Recently Used (LRU) replacement policy. The pipeline and memory subsystems are designed to support efficient instruction throughput, issuing one instruction per cycle under ideal conditions. Additionally, our Phase-3 processor design incorporates an implicit dynamic branch predictor based on 2-bit saturating counters, which initializes to "weak not taken" upon reset. It also utilizes a Branch Target Buffer (BTB) to cache the targets of recently taken branches, improving branch prediction accuracy and execution efficiency.

Submission Folder Description

The submission folder is organized in the following format:

/Submission

├── cpu.png # Top-level block diagram of the pipelined CPU in PNG file format

├── TestPrograms/ # Directory consisting of test cases used to verify the design

├── EC_pre/ # Directory containing pre-area-optimized extra-credit design and synthesis files

│ ├── designs/ # Directory containing pre-synthesis design files + synthesized .vg netlist of proc.v

│ ├── outputs/ # Directory containing post synthesis test output files and synthesis reports

│ └── tests/ # Directory containing all related testbench files used for testing

├── EC_post/ # Directory containing post-area-optimized extra-credit design and synthesis files

│ ├── designs/ # Directory containing pre-synthesis design files + synthesized .vg netlist of proc.v

│ ├── outputs/ # Directory containing post synthesis test output files and synthesis reports

│ └── tests/ # Directory containing all related testbench files used for testing

├── Phase-3/ # Directory containing Phase-3 files with dynamic branch prediction and caches

│ ├── designs/ # Directory containing Phase-3 design files

│ ├── outputs/ # Directory containing top level generated test output files

│ └── tests/ # Directory containing all related testbench files used for testing

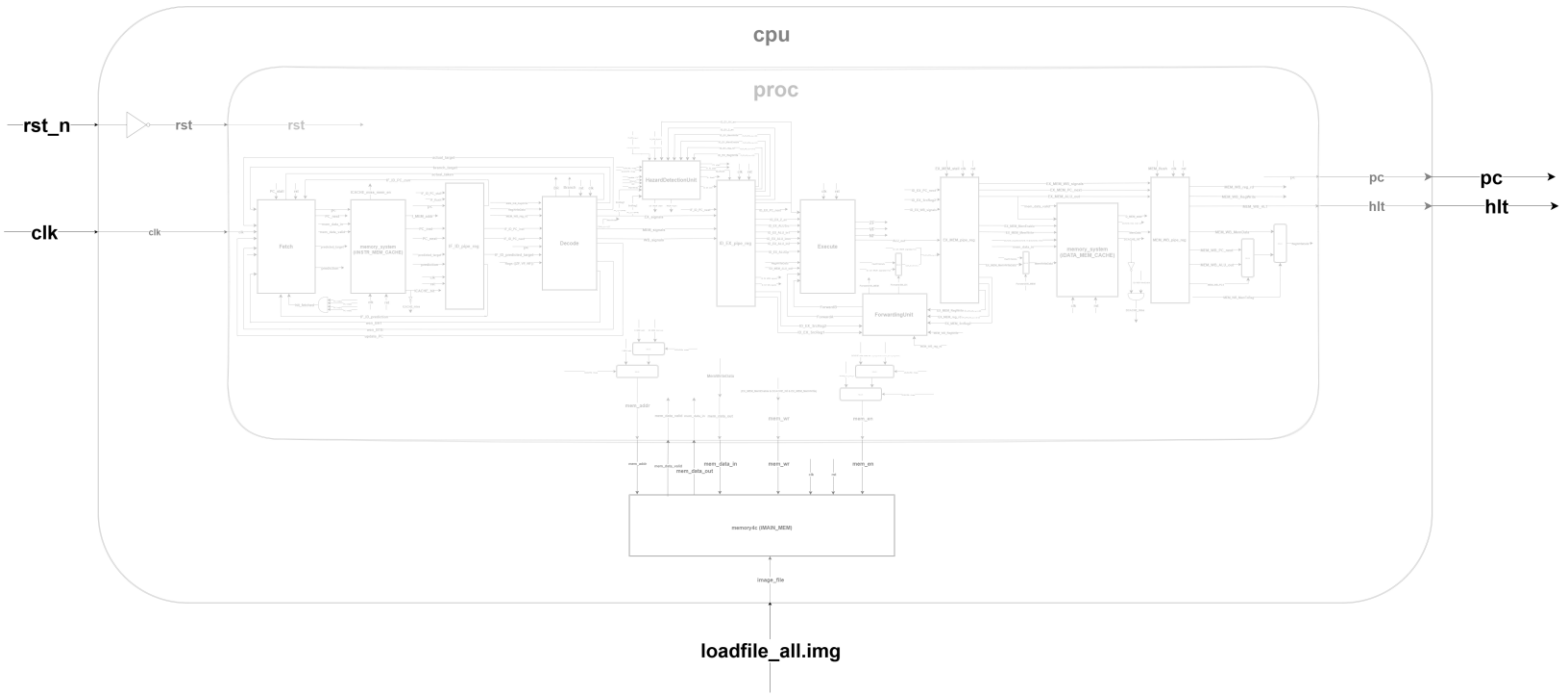
├── Scripts/ # Directory containing scripts for automating workflows

Top-Level Block Diagram Description

The module hierarchy is illustrated below:

- cpu.v: Top-level module integrating the processor core and main memory
    - proc.v: Implements the pipelined CPU core
        - (Internally includes additional design files such as Fetch.v, memory_system.v (I-Cache), IF_ID_pipe_reg.v, Decode.v, HazardDetectionUnit.v, ID_EX_pipe_reg.v, ForwardingUnit.v, Execute.v, EX_MEM_pipe_reg.v, memory_system.v (D-Cache), MEM_WB_pipe_reg.v)
    - memory4c.v: Implements the unified instruction and data memory that performs a single cycle write and a multi-cycle read

The diagram below illustrates how the various modules interact with one another: (A zoomable picture is attached to the submission of the project)



2. Responsibility/Task Breakdown

PHASE\MEMBERS	SRIVIBHAV	JOSH	SANDEEP	KAILAN
PHASE-1	25% Designed ALU and related submodules, RegisterFile, and helped with testing	25% Helped with testing and top-level integration	25% Designed the ControlUnit and helped with testing	25% Helped with top-level integration and testing
PHASE-2	75% Designed Fetch, Decode, and Execute modules, IF_ID, ID_EX, EX_MEM, MEM_WB pipeline registers, ForwardingUnit, and HazardDetectionUnit modules	5% Helped with top-level testing	15% Helped with designing the DynamicBranchPredictor, integration and top level testing	5% Helped with top level testing
PHASE-3	50% Designed memory system, instruction and data caches, and cache controllers	25% Helped with testing and top level integration	25% Helped with testing and top level integration	—
Extra-Credit	50% Synthesized pre-optimized code, automated workflows for faster integration and testing	50% Synthesized optimized code and verified functionality post-synthesis	—	—

3. Special Features

- We used Karnaugh Maps (K-Maps) to optimize Boolean logic expressions, reducing area and improving synthesis quality.
- We implemented the caches as **64 sets of 2 blocks each**, rather than **128 blocks per cache**, reducing the hit cycle time from **2 to 1 cycle**.
- Initially, our 2-bit Branch History Table (BHT) was implemented with all entries initialized to **strongly not taken**. This design meant that the predictor required two consecutive mispredictions to fully correct itself, introducing an additional cycle of latency after a branch misprediction. To improve performance, we updated the initialization to **weakly not taken**, allowing the predictor to adapt more quickly and reducing misprediction recovery latency by one cycle.
- In addition to optimizing the BHT, we implemented a **Branch Target Buffer (BTB)**. The BTB allowed us to predict the target address of branch and jump instructions without waiting for full instruction decode. By using the BTB, we were able to fetch the next instruction immediately after a predicted taken branch, rather than stalling for a cycle to compute the branch target. This significantly reduced control hazards associated with taken branches and improved overall pipeline efficiency.
- Alongside area optimization, we focused on **IPS (Instructions Per Second) optimization** by minimizing pipeline stalls, reducing branch misprediction

penalties, and improving cache access times. These efforts collectively increased instruction throughput, shortened simulation cycle counts, and enabled the CPU to operate at significantly higher frequencies, improving synthesis frequency from **440 kHz to 10 MHz**

- While we did not encounter major bottlenecks during the initial design implementation, optimizing synthesis posed challenges. Specifically, although we identified the critical path, determining the most effective method to optimize for higher synthesis frequencies required careful analysis.

4. Completeness

- We verified that our design fully meets the requirements outlined in **Phases 1, 2, and 3**.
- Comprehensive testing revealed no functional bugs, and we successfully submitted a fully operational CPU module for each phase's deadline.
- No post-submission modifications to the design files were necessary, as all submitted versions functioned correctly.

5. Testing

- We developed a **Python**-based test automation framework integrated with a **Makefile**, utilizing make run targets to execute various **.img test files**. These files were assembled using the provided assembler and loaded into the CPU's instruction memory.
- Verification was conducted using a combination of **self-checking tasks**, **SystemVerilog packages**, and a **scoreboard CPU class**, with detailed output logs generated through ModelSim to ensure comprehensive functional validation.
- The test environment incorporated:
    - **Memory dump verification**
    - **Register dump verification**
    - **Dynamic Branch Predictor dump verification**
    - **Self-checking of internal signals** by comparing DUT behavior against the expected outputs generated by the scoreboard class
- All test files, test benches, and supporting verification infrastructure are included with the submission.

6. Results

The following table summarizes the output generated for various test cases across each phase, where a ✅ indicates a correct result:

PHASE\TEST	TEST-1	TEST-2	TEST-3	TEST-4
PHASE-1	✅ Cycle Count: 14	✅ Cycle Count: 15	✅ Cycle Count: 22	✅ Cycle Count: 156
PHASE-2	✅ Cycle Count: 18	✅ Cycle Count: 20	✅ Cycle Count: 28	✅ Cycle Count: 211
PHASE-3	✅ Cycle Count: 46	✅ Cycle Count: 61	✅ Cycle Count: 56	✅ Cycle Count: 331

We did not modify any design files after the submission deadlines, as they were verified to be correct prior to submission.

7. Extra Credit Synthesis

The following metrics summarize the **post-synthesis** results and testing outcomes for the **Extra-Credit** pre-area-optimized design:

- **Cycle Time (post-synthesis):** 2240 ns
- **Synthesis Tool:** Synopsys Design Compiler
- The following table summarizes the output generated for various test cases across the pre-area-optimized synthesized netlist, where a ✅ indicates a correct result:

EXTRA-CREDIT	TEST-1	TEST-2	TEST-3	TEST-4
PRE-AREA OPTIMIZED SYNTHESIS	✅ Cycle Count: 46	✅ Cycle Count: 61	✅ Cycle Count: 56	✅ Cycle Count: 331

A post-synthesis testbench is included in the submission files and was used to generate these results. Post-synthesis verification confirmed that the synthesized proc.vg netlist correctly initialized the program counter (PC) to 0x0000 upon reset. The output logs clearly showed that the processor properly activated memory access on instruction cache and data cache misses and asserted the 'hlt' signal when the **HLT** instruction was encountered in the corresponding test cases.

8. Extra Credit Features

We implemented the following extra credit features:

- **1. Synthesis and Post-Synthesis Verification:**
  Successfully synthesized the design using Synopsys Design Compiler, extracted the netlist, and verified functionality through post-synthesis simulation using the custom-made test bench.

- **2. Optimization for Low Area:**
  We optimized the design using techniques such as **Karnaugh Maps (K-Maps)** to minimize Boolean expressions. We redesigned key modules, including the **CLAs**, **Shifters**, **Caches**, the **ALU**, and the **Register File**. Restructuring these modules for simplicity and efficiency enabled the synthesis tool to generate more optimized hardware, avoiding the inference of unnecessarily complex logic and improving overall synthesis results. As a result, the **total area** after synthesis was significantly reduced — from approximately **878,520 μm^2** before optimization to **611,206 μm^2** after optimization — reflecting a notable improvement in resource utilization.

- **4. Optimization for High IPS (Instructions Per Second):**
  We optimized the design for high **Instructions Per Cycle (IPC)** and post-synthesis frequency. Specifically, we restructured the cache architecture into a 64-set, 2-block configuration, reducing cache hit latency from 2 cycles to 1 cycle and improving overall throughput. Furthermore, after applying these optimizations, we re-ran the synthesis flow on the redesigned CPU. We significantly improved the clock frequency**, increasing it from approximately 440 kHz to 10 MHz**. In addition, we **restructured the cache** implementation to use higher-level coding constructs rather than explicit D flip-flop (DFF) instantiations, enabling the synthesis tool to optimize the design for timing and area better.

- **7. Branch Prediction:**
  We implemented a **2-bit branch predictor** initialized to **weak not taken**, replacing the previously used **static predict-not-taken** scheme. This **dynamic prediction** approach enabled earlier correct fetching of taken branches and more effective prediction adjustments, reducing stall cycles. We also implemented a **Branch Target Buffer (BTB)** to further improve control flow to store predicted branch targets. The BTB allowed the processor to immediately fetch the correct target address for taken branches without stalling for target computation, further reducing control hazards and enhancing overall **pipeline performance**.

Extra Credit Feature 1: Synthesis and Post-Synthesis Verification

1. Description of the Feature:

We successfully synthesized our CPU design using **Synopsys Design Compiler**, extracted the gate-level netlist, and performed **post-synthesis verification**. This ensured that the synthesized hardware implementation behaved identically to the RTL design. We synthesized the design to meet a **clock frequency of 440 kHz**.

2. Task Breakdown:

TASK\MEMBERS	SRIVIBHAV	JOSH	SANDEEP	KAILAN
PRE-AREA OPTIMIZED SYNTHESIS	50% Made a post-synthesis testbench and automated workflows for faster integration and testing	50% Synthesized pre-optimized code and analyzed reports	—	—

3. Special Features:

- No explicit special features were developed specifically for this extra credit feature. Instead, we created a **synthesis script** and ran it against our **Phase-3 design** to establish a baseline for further optimization.

4. Completeness:

- Through **synthesis**, verification that our design met **timing** at **440 kHz** with **positive slack**, satisfaction of **minimum delay** requirements, and successful completion of **post-synthesis tests**, we can certify that our design meets the specifications for all three phases of the project and fully complies with the **WISC-S25 ISA**.

5. Description of Test Cases:

- We created a **post-synthesis testbench** and ran it on the pre-synthesis RTL to establish baseline outputs.
- After synthesis, we simulated the post-synthesis netlist using the same testbench to verify that the outputs matched the baseline.
- During **post-synthesis verification**, we specifically:
    - Verified that the **Program Counter (PC)** was correctly initialized to **0x0000** after reset.
    - Verified the interaction between **main memory** and the processor, particularly during **cache misses** and **memory writes**.
    - Confirmed that the processor correctly raised the '**hlt**' signal once it encountered the **HLT** instruction.
    - **Section 7** (above) illustrates the **test output results** for the **pre-area-optimized synthesized design**.

No functional mismatches were observed between pre-synthesis and post synthesis simulations.

Extra Credit Feature 2: Optimization for Low Area

1. Description of the Feature:

We focused on optimizing our CPU design for **low area**. Using **Synopsys Design Compiler**, we successfully synthesized the design, extracted the **gate-level netlist**, and performed **post-synthesis verification** to ensure that the synthesized hardware matched the behavior of the original RTL design. This process allowed us to achieve significant area reductions while preserving functional correctness.

2. Task Breakdown:

TASK\MEMBERS	SRIVIBHAV	JOSH	SANDEEP	KAILAN
AREA OPTIMIZED SYNTHESIS	50% Identified key modules to optimize and reduce overhead	50% Synthesized optimized code and re-analyzed reports	—	—

3. Special Features:

- In developing this feature, we focused on optimizing the design for better **area efficiency**. Specifically, we recoded all **Carry Lookahead Adders (CLAs)** to use standard **"+" and "-" operators**, and similarly optimized the **shifters** and **decoders**. Additionally, we identified a major bottleneck in the provided **DataArray** and **MetaDataArray**. To address this, we refactored these components using **higher-level constructs**, which allowed the synthesis tool to optimize the design more effectively and improve synthesis time.

4. Completeness:

- Through **synthesis** and verification that our design, optimized for **area**, still met all timing and functional requirements, we can certify that our design meets the specifications for all three phases of the project and fully complies with the **WISC-S25 ISA**. Specifically, the **total area** after synthesis was significantly reduced — from approximately **878,520 μm^2** before optimization to **611,206 μm^2** after optimization — and the design passed all **post-synthesis tests** successfully.

5. Description of Test Cases:

- We utilized the same **post-synthesis testbench** that was created for synthesizing our baseline CPU. The following table summarizes the output generated for various test cases across the area-optimized synthesized netlist, where a ✅ indicates a **correct result**.
- Detailed **reports** are included as part of the submission.

EXTRA-CREDIT	TEST-1	TEST-2	TEST-3	TEST-4
AREA OPTIMIZED SYNTHESIS	✅ Cycle Count: 46	✅ Cycle Count: 61	✅ Cycle Count: 56	✅ Cycle Count: 331

No functional mismatches were observed between the pre-optimized and post-optimized designs. Through this optimization, we were able to synthesize with a reduced area, providing more flexibility for incorporating additional features in the future.

Extra Credit Feature 3: Optimization for High IPS (Instructions Per Second):

1. Description of the Feature:

We focused on optimizing our CPU design for **high IPS** (Instructions Per Second). To achieve this, we implemented various performance optimizations in critical paths of the pipeline, including improving the **instruction fetch** and **execution stages**, reducing latency, and minimizing pipeline stalls. By refining our design, we were able to increase the overall throughput of the CPU. After synthesizing the design with **Synopsys Design Compiler**, extracting the **gate-level netlist**, and performing **post-synthesis verification**, we ensured that the synthesized hardware matched the behavior of the original RTL design, achieving higher IPS while maintaining functional correctness.

2. Task Breakdown:

TASK\MEMBERS	SRIVIBHAV	JOSH	SANDEEP	KAILAN
CLOCK CYCLE OPTIMIZATION	50% Identified key modules to optimize and reduce overhead	50% Synthesized optimized code and re-analyzed reports	—	—

3. Special Features:

- This feature combines results from **area optimization** and **dynamic branch prediction**, as described in the following sections. During the area optimization process, we identified the **DataArray** as the primary bottleneck, with a critical path originating from it that severely limited our ability to further optimize for higher synthesis frequencies. While initially aimed at reducing area, this effort also led us to restructure the **cache design**, which ultimately allowed us to improve the synthesis frequency from **440 kHz** to **10 MHz.** Additionally, from a **simulation**

perspective, the inclusion of **dynamic branch prediction** (as described in the following section) with a **2-bit Branch History Table (BHT)** and a **Branch Target Buffer (BTB)** significantly improved the **execution time** of each test case, enhancing overall performance.

4. Completeness:

- Through **synthesis** and verification that our design, optimized for **high frequency** and reduced **simulation cycles**, still met all timing and functional requirements, we can certify that our design meets the specifications for all three project phases and fully complies with the **WISC-S25 ISA**. Specifically, the **synthesis frequency** was improved from **440 kHz** to **10 MHz**, and the design passed all **post-synthesis tests** successfully, demonstrating its performance improvements in speed and functionality.

5. Description of Test Cases:

- We utilized the same post-synthesis testbench initially developed for synthesizing and verifying our baseline CPU. The following table summarizes the output generated for various test cases across the **cycle-optimized synthesized netlist**, where a ✅ indicates a correct result. These results are compared against the pre-optimized, default **predict-not-taken** pre-synthesized test outputs to highlight the improvements achieved through our optimizations.
- Detailed **reports** are included as part of the submission.

EXTRA-CREDIT	TEST-1	TEST-2	TEST-3	TEST-4	TEST-7
PRE CYCLE OPTIMIZED PRE-SYNTHESIS	✅ Cycle Count: 46	✅ Cycle Count: 61	✅ Cycle Count: 56	✅ Cycle Count: 346	✅ Cycle Count: 172
CYCLE OPTIMIZED SYNTHESIS	✅ Cycle Count: 46	✅ Cycle Count: 61	✅ Cycle Count: 56	✅ Cycle Count: 331	✅ Cycle Count: 170

No functional mismatches were observed between the pre-optimized and post-optimized designs. By identifying critical paths and reducing stall cycles, we were able to achieve both higher **throughput** and improved **synthesis frequency**.

Extra Credit Feature 4: Branch Prediction:

1. Description of the Feature:

We focused on optimizing our CPU design by implementing **dynamic branch prediction** to improve **Instructions Per Second (IPS)**. A key aspect of this optimization was the introduction of a **2-bit Branch History Table (BHT)**, which tracks branch outcomes and predicts future branches based on historical behavior. We also integrated a **Branch Target Buffer (BTB)** to store taken branch targets, allowing immediate fetching of branch target

addresses. By leveraging the BHT and BTB, we reduced the time spent waiting for branch address computations and minimized pipeline stalls, improving instruction throughput.

2. Task Breakdown:

TASK\MEMBERS	SRIVIBHAV	JOSH	SANDEEP	KAILAN
DYNAMIC BRANCH PREDICTION	85% Developed the 2-bit BHT and BTB	—	15% Optimized code and reduced cycle time further	—

3. Special Features:

- Through optimization, including integrating dynamic branch prediction and improvements for higher throughput, we significantly reduced pipeline stall cycles and flushes. These optimizations allowed for smoother instruction flow and faster execution, ensuring the design met all functional requirements.
- Initially, we set the branch prediction to "strong not taken," which resulted in higher cycle counts. However, after changing the initialization to "weak not taken," we observed a notable improvement in cycle counts, further enhancing the overall performance and reducing execution time.

4. Completeness:

- By implementing dynamic branch prediction and optimizations for higher frequency, we successfully reduced pipeline stall cycles and flushes, resulting in a more efficient CPU design. These improvements allowed the design to meet all timing and functional requirements. We can confirm that the design satisfies the specifications for all three project phases and fully adheres to the WISC-S25 ISA. The optimization significantly reduced cycle counts and enhanced overall performance, ensuring the design's effectiveness and compliance.

5. Description of Test Cases:

- We utilized a custom cpu_tb.sv testbench to functionally verify the design. Test cases (included in the submission) were developed specifically to evaluate the performance of programs with loops. The following table summarizes the output generated for various test cases comparing the predict-not-taken scheme with the dynamic branch predictor scheme, where a ✅ indicates a **correct** result.
- Detailed reports are included as part of the submission.

EXTRA-CREDIT	TEST-1	TEST-2	TEST-3	TEST-4	TEST-7
BRANCH PREDICT NOT TAKEN	✅ Cycle Count: 46	✅ Cycle Count: 61	✅ Cycle Count: 56	✅ Cycle Count: 346	✅ Cycle Count: 172
DYNAMIC BRANCH PREDICTION	✅ Cycle Count: 46	✅ Cycle Count: 61	✅ Cycle Count: 56	✅ Cycle Count: 331	✅ Cycle Count: 170

No functional mismatches were observed between the predict-not-taken and dynamic branch prediction designs. By integrating dynamic branch prediction, we were able to reduce pipeline stall cycles and flushes, leading to improved instruction throughput and faster program execution.

9. Verilog Files

All required Verilog files are included in the submission, organized according to the directory structure described in Section 1.