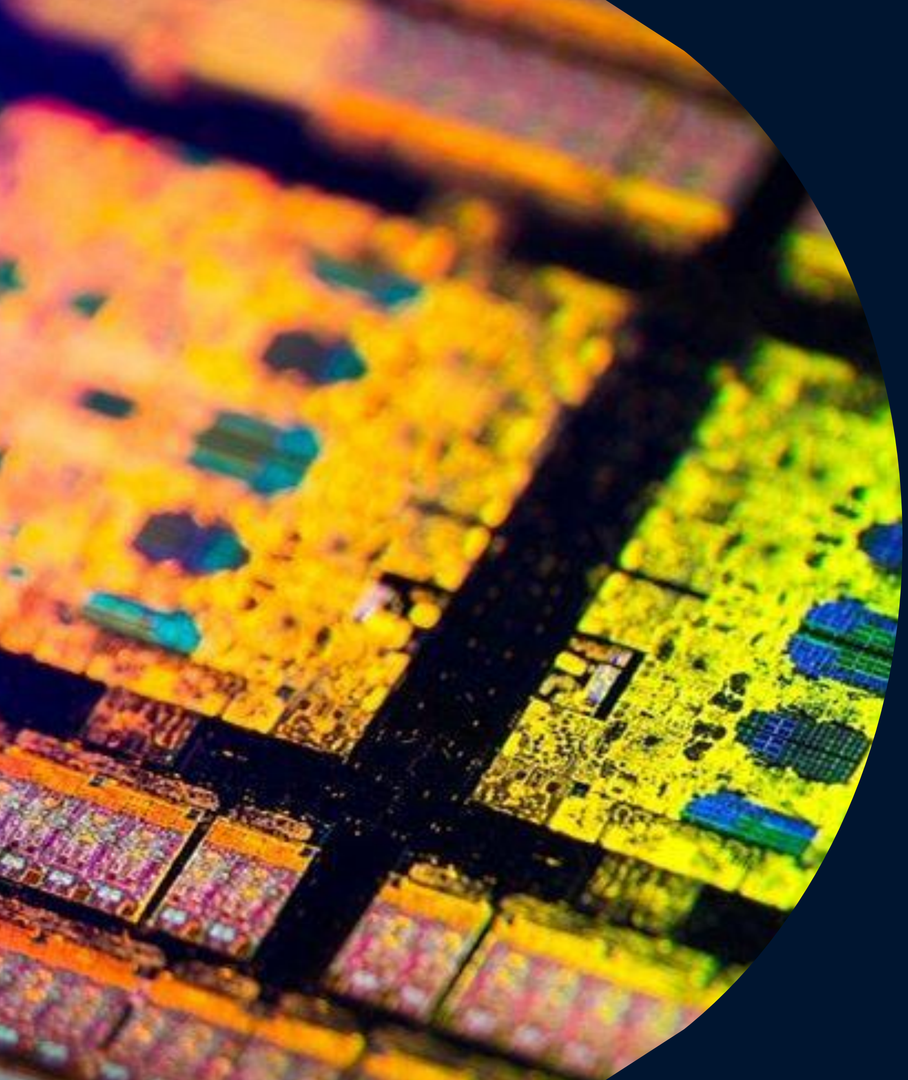
R5SEBA Superscalar Engine and Binary Architecture

EECS 470 Final Project

Presented by

Luke Brzozowski Tong Sing Wu Mahdi Chowdhury
Bureir Alaboudi Michael Kang

April 22, 2025

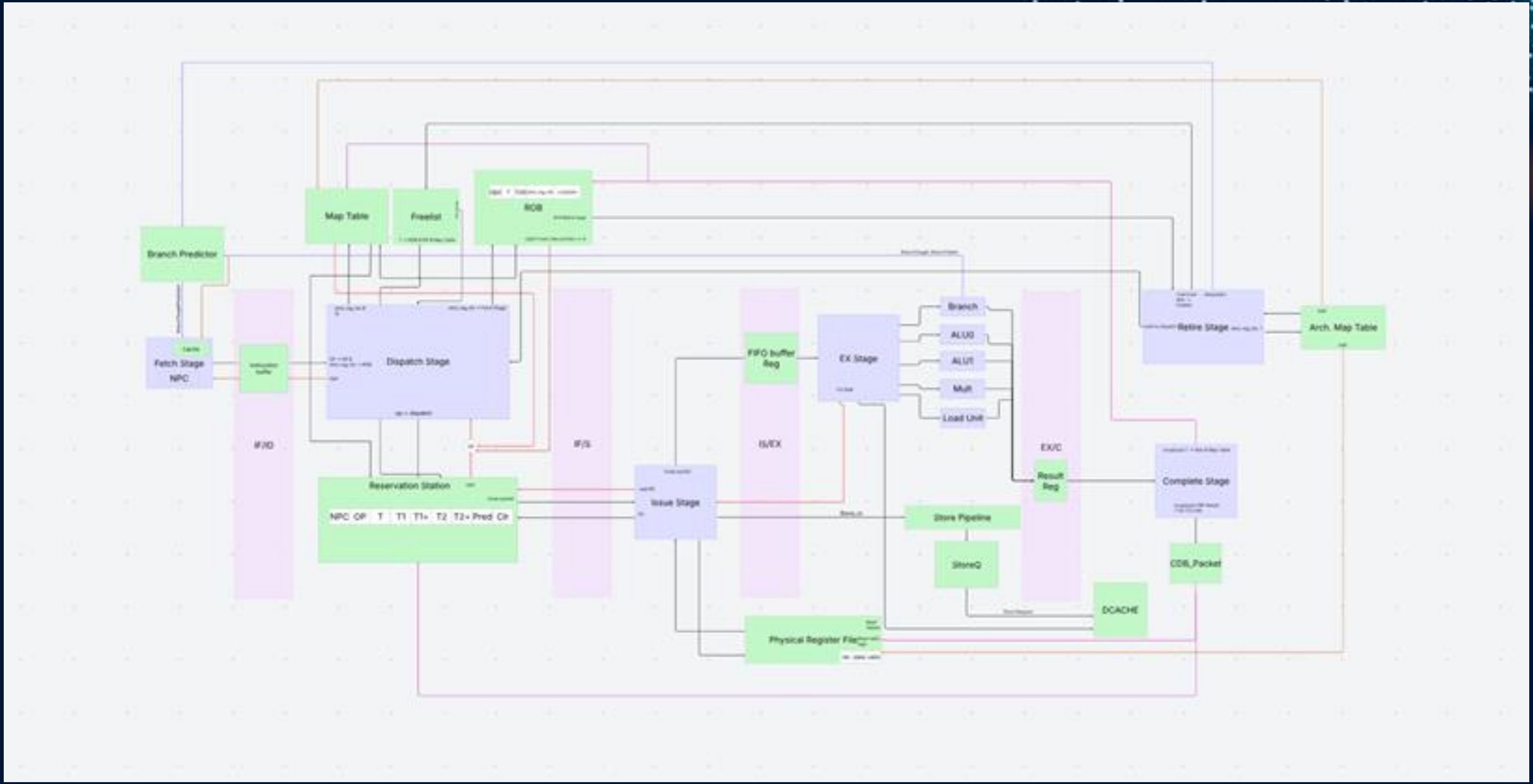


Advanced Features & Unique Implementations

- R10K, 2-Way Superscalar
- Early Tag Broadcasting
- Load Store Queue
- Wb D-Cache
- Cache Buffer

- Store Pipeline in Store Queue
- FIFO Everywhere! (Multiple Designs)
- Freelist Implementation
- Reservation Station without Psel
- Rotating Priority Selector with Load priority

R10K Architectural Map



FIFO Everywhere!

- Cache Buffer (FIFO)
  - Assigns memory requests and transaction tags
  - Handles back pressure from multiple memory requests efficiently

- Issue Buffers (FIFO), per Functional Unit (Load, Store, Mult, ALU, and Branch)
  - Each queue supports 2 functional units for high parallelism
  - Acts as a extension to the RS that handles the CDB backpressure

- Instruction Buffer (FIFO)
  - Handles back pressure when a structural hazard occurs

Also:

- ROB (FIFO)
- LSQ (FIFO) (Has its own slide)
- Freelist (FIFO) (Has its own slide)

Reservation Station Without Priority Selector

- Multi Issue Support
  - Supports 2 dispatches and 2 issues per cycle.

- Memory Dependency Tracking:
  - Uses *UncStatus* flag to stall loads until older store retires

- Backpressure Handling
  - Checks FU FIFO before issuing to prevent stalls.

Early Tag Broadcast

- Have early tag broadcasting for all instructions except load and store
- Arbitrate on the Issue FU FIFO, because it’s an extension of the RS. The head and next head of the FIFO acts as the issue register that is arbitrated on
- This design is unique because this means we don’t get information to arbitrate from the RS so all processing for ETB is unified in Issue Stage

Load Store Queue

- Data Forwarding Support
  - Built-in support for data forwarding, but currently not enabled

- Optimal Load/Store Coordination
  - Load and stores are managed with a design that eliminates arbitration, so they don’t fight for shared resources

- Early Coordination
  - Loads and stores coordinate directly with the Reorder Buffer (ROB) and Reservation Stations (RS) as early as possible, reducing latency and improving efficiency

Store Queue Pipeline

- Divergent Store Pipeline
  - Stores are routed through a separate pipeline, so they don't compete with other instructions for execution resources. This keeps the main pipeline less congested.
- Simple Adder for Store Address Calculation
  - We use a dedicated adder to compute store addresses instead of relying on the ALU. This design avoids ALU bottlenecks (with multiplies) and can shorten store operations by collapsing two cycles into one.
- Matched Store Queue (SQ) & Reorder Buffer (ROB) Sizes
  - By keeping the SQ and ROB the same size, the store pipeline maintains a steady flow with no risk of stalls due to resource limitations.

Rotating Priority Selector

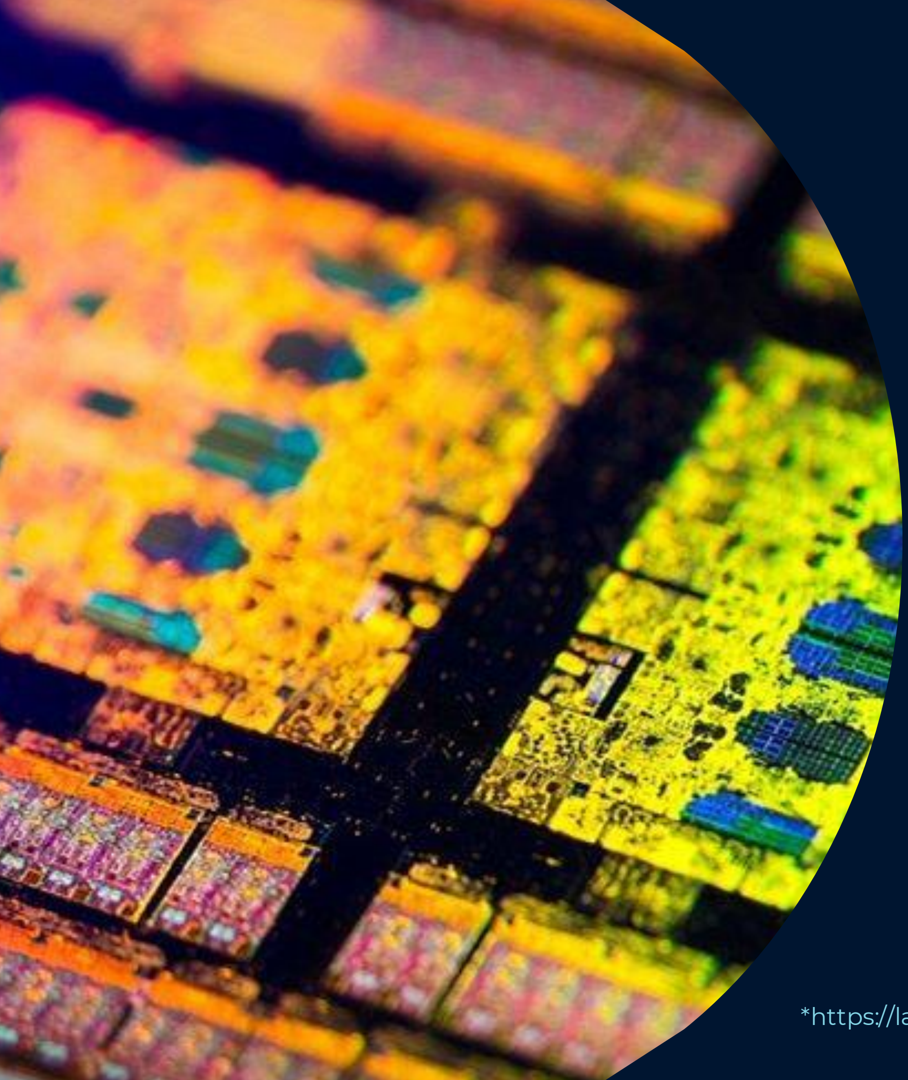
- Rotating PSEL with Load Priority
  - We use a rotating priority selector (PSEL) with hard-coded load priority to ensure loads are always taken care of to avoid starving other instructions.
  - Stores don’t use this pipeline, avoiding unnecessary resource usage

- Parallel Completion with Execute
  - The complete stage operates in parallel with the execute stage, which shortens the critical path and allows timing optimizations by decoupling these processes.

Freelist Implementation

- Freelist as a FIFO
  - Allocates and recycles tags for register renaming using FIFO logic
  - Head pointer: Dispatches (reads) new tags for instructions entering the pipeline
  - Tail pointer: Recycles (writes) tags from retiring instructions

- Branch Misprediction Recovery
  - On misprediction, freelist recovery is instant—tail pointer is set back to the head and size is reset
  - Head → Tail: Free tags (retired instructions)
  - Tail → Head: Tags held by instructions still in flight

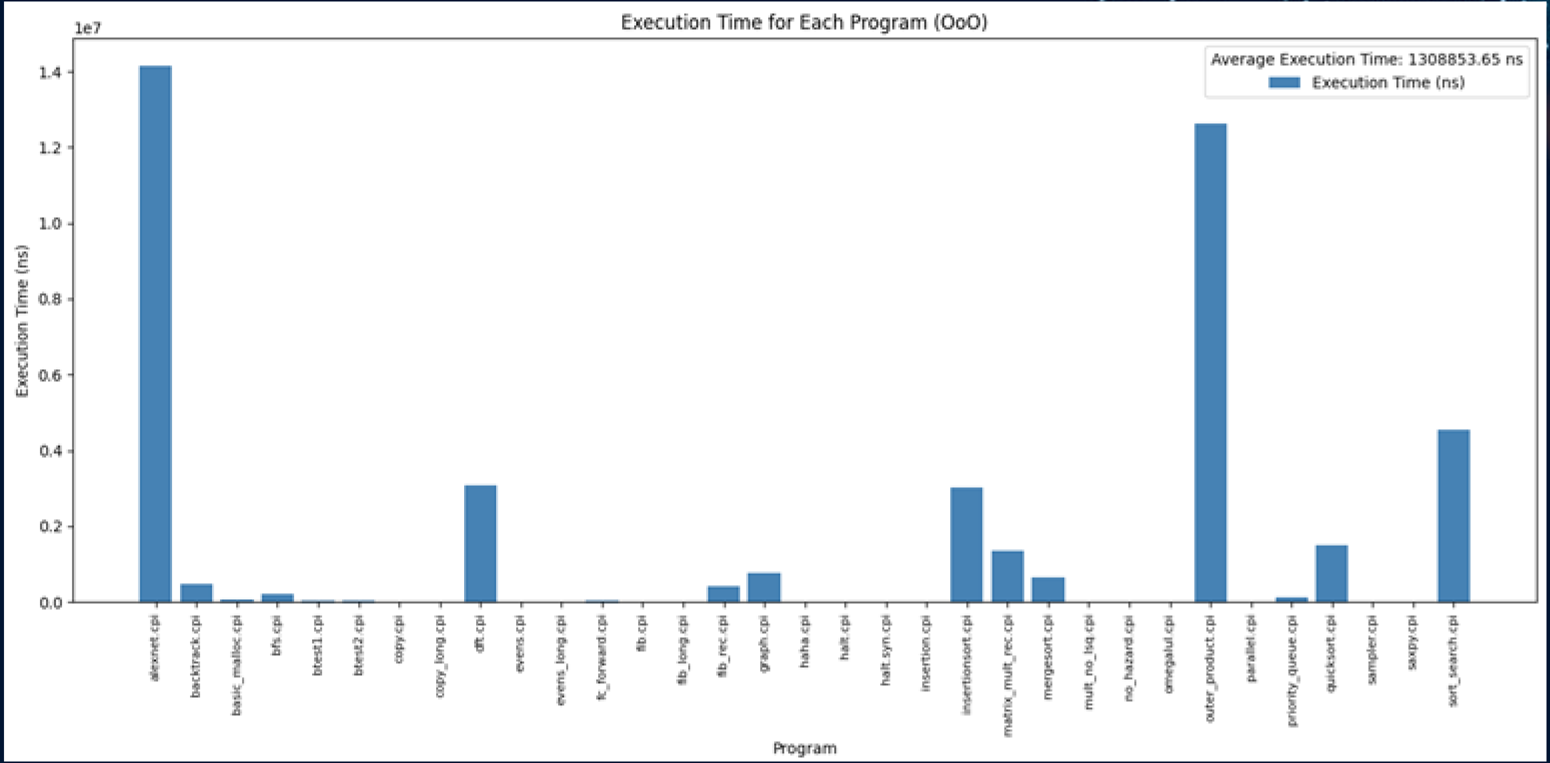


Performance Analysis & Possible Optimization

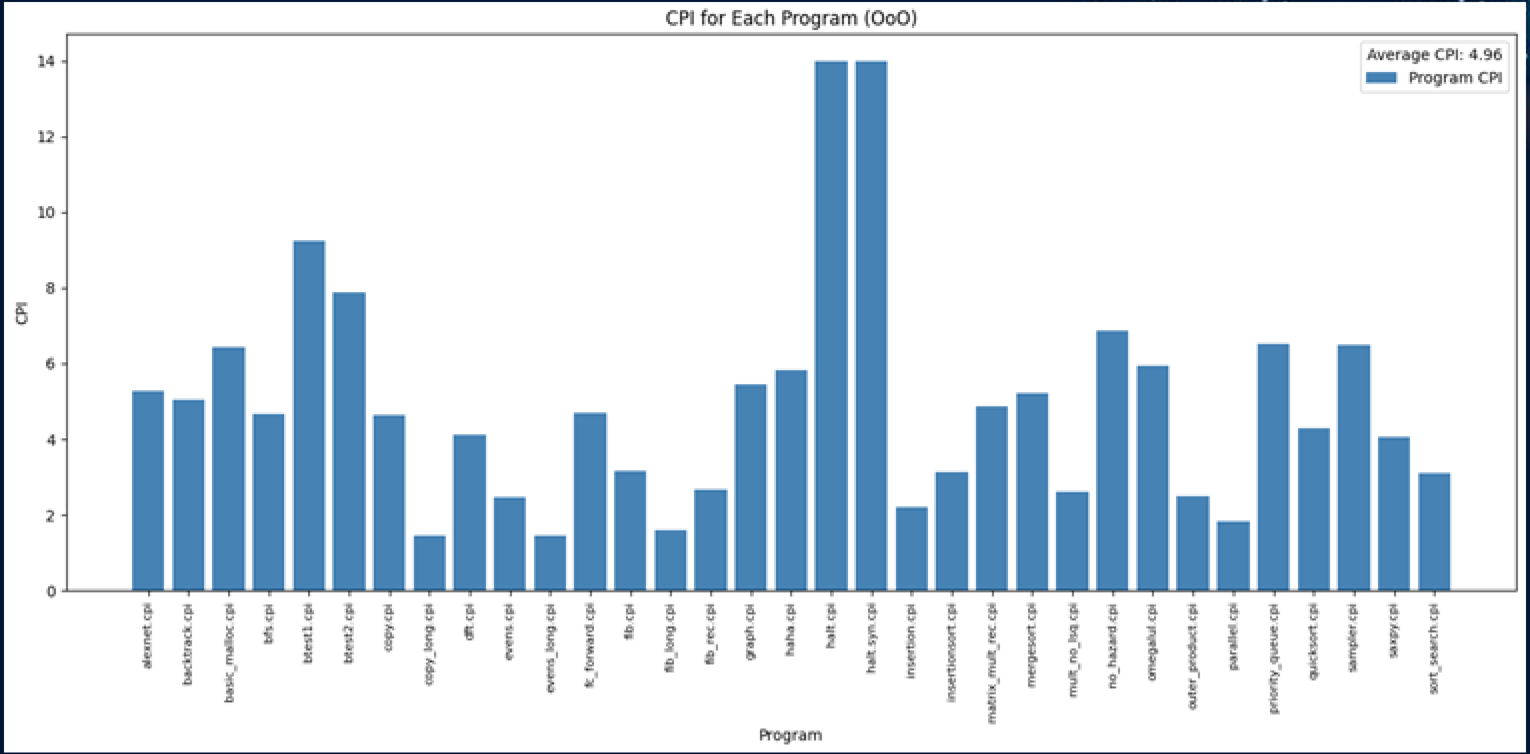
- Clock Period: 12.87 ns
- CPI: 4.96 (On Given Test Cases)
- Critical Path:
  - ROB Retire → Store Queue Retire → DCache

*https://latamfdi.com/semiconductor-industry-in-the-dominican-republic/

Performance Analysis: Execution Time



Performance Analysis: CPI



Bottlenecks & Critical Path

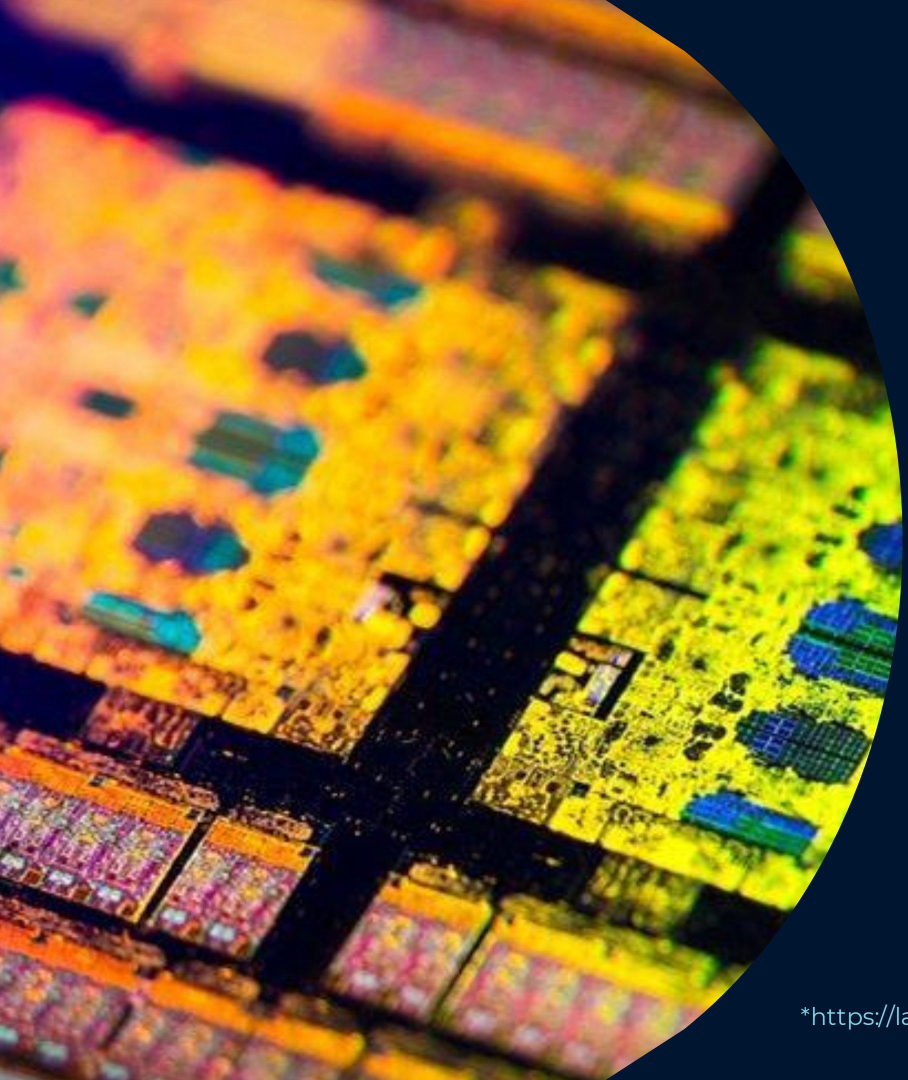
- Critical Path: ROB Retirement to Store Queue
  - When the ROB retires a store, the store queue must also retire the corresponding entry
  - The retiring store triggers a request to the data cache (dcache)
  - Must compare the retiring instruction with others (e.g., next store, JALR, branch) to ensure proper ordering and detect hazards

- Misprediction Complexity:
  - If a misprediction is detected (mispeculated branch), all inflight instructions and the store queue must be squashed
  - This adds logic and timing pressure to the control path, increasing latency

- Multiplier Pipeline:
  - 4-stage multiplier intended for high throughput, but the critical path (above) overshadows the potential CPI gains
  - Result: CPI is still higher than expected, despite pipelining

Possible Optimizations

- 2-Stage Multiplier
  - Implementing a 2-stage multiplier allows for a 13 ns clock period
  - Reduces latency without increasing CPI
- Store Pipeline Simplification
  - Collapse the unique store pipeline stage
  - Register store queue output for dcache access in the next cycle
  - Preserves CPI and significantly reduces clock period
- Control & FIFO Improvements
  - Add support for more functional units per FIFO (if not already two)
- Enhanced Data Forwarding
  - Expand data forwarding support beyond current partial implementation
  - Minimizes stalls and improves overall throughput



Development Methodologies & Testing

- Iterative and adaptive design process
- Comprehensive testing at module level
- Focus on synthesis and validation of hardware

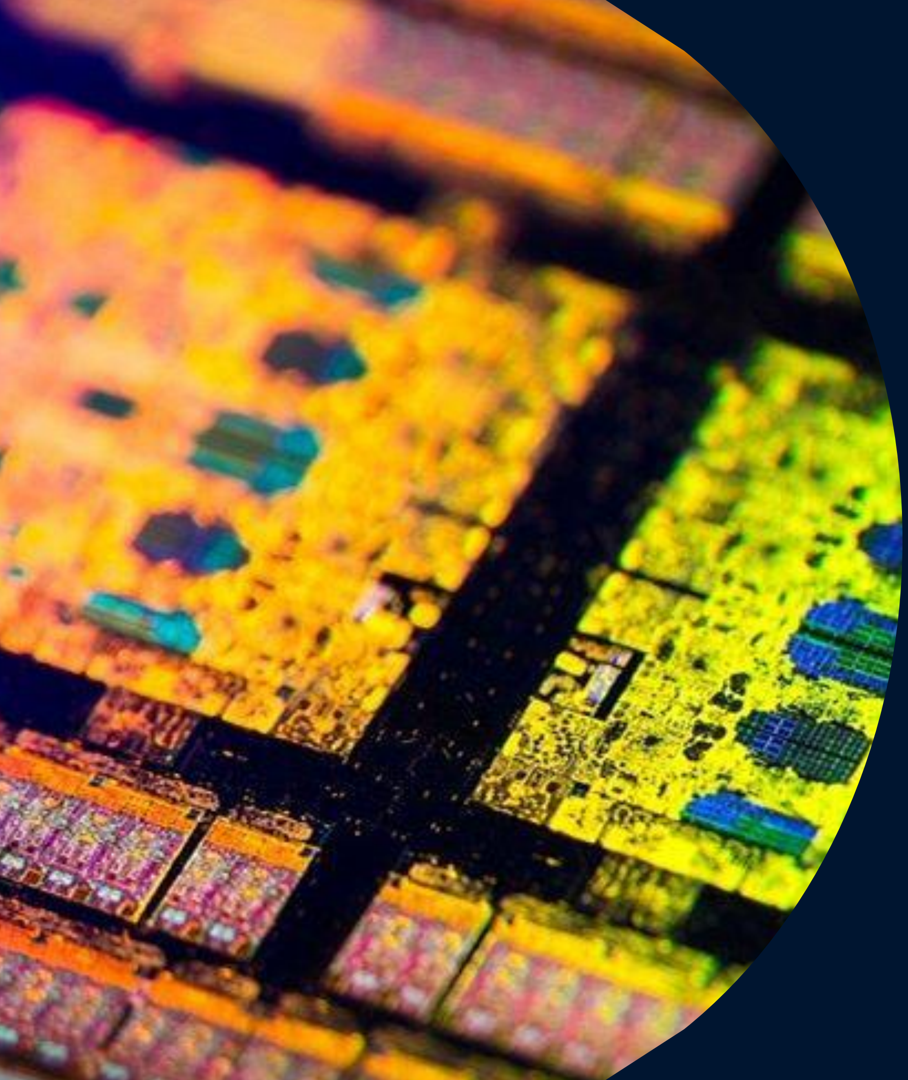
*https://latamfdi.com/semiconductor-industry-in-the-dominican-republic/

Iterative Development & Testing

- Agile Development (At First)
  - Emphasized adaptability, frequent check-ins, and incremental progress
- Spiral Development (post-Spring Break)
  - Switched to a cyclical approach: planning, prototyping, testing, and refining in repeated loops
- Unit & Integration Testing
  - Tested individual modules in isolation to catch bugs early
  - Combined modules and verified their interactions worked correctly
- Fake Fetch
  - Used a fake fetch module early on to simulate instruction flow before the full cpu.sv was ready

Validation and Synthesis Readiness

- Regression Testing
  - After each change, re-ran previous tests to ensure we didn't break existing functionality
- Synthesis Testing
  - Used cpu_synth.out and cpu.rep to check real-world synthesis results
  - Debugged timing loops and latch issues to ensure design could be implemented on hardware
- Continuous Improvement
  - Regularly incorporated test results and synthesis feedback into our development cycle



Challenges & Takeaways

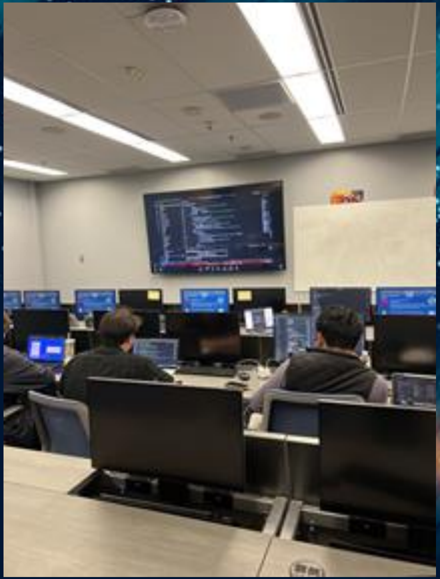
- Initial difficulties starting off
- Crunching to meet deadlines
- Bringing it back to finish strong

Challenges & Difficulties

- Late Start
  - We began the project later than ideal, requiring extra effort and long hours to meet deadlines
- Team Changes
  - During the middle of the semester, we unfortunately had a group member drop, which scattered priorities and we had to have time to adjust for workflow
- Scheduling Struggles In Person
  - At the beginning of the project, it was hard to coordinate meeting times. A lot of the time we relied heavily on working together, but ended up online.
- Team Dynamics
  - Initial difficulties with team dynamics and communication as we learned to collaborate effectively

Takeaways & Learned Learned

- Planning before you code
    - Mapping out to how implement features before jumping in saves time and reduces debugging hours
- Everything takes 2x time than we thought
    - Implementation and debugging usually takes twice as long as you expect it does
- Reach out for help!
    - If you're struggling with an issue or having a hard time starting, then reaching out for help to the staff is such a huge help.
- Make sure to start early!
    - Since we didn't start early, we were constantly playing catch-up till the end of the semester.
- Remember to leave time to synthesize 😭


(11:39pm in BBB yesterday)

THANK YOU!

QUESTIONS?