Berkeley
UNIVERSITY OF CALIFORNIA

RISC-V Processor Design

Daniel Min Chang, Vinay Agrawal

EECS 151 Lab

Prof. Bora Nikolic

Electrical Engineering and Computer Sciences

Department of Engineering

Abstract

The goal of this project is to design a simple 3-stage CPU that implements the RISC-V ISA. The primary objective is to write a functional implementation of your processor. Other additional features or architectural choices can be chosen to improve its performance.
While certain testbenches are provided to help verify the functionality of the overall performance of the design, additional testbenches for individual submodules must be created separately. The target implementation technology will be Skywater 130nm. The project highlights the skillsets of designing synthesizable RTL, resolving hazards in a simple pipeline, building interfaces, and approaching system-level optimization.

1 Project Overview

In this project, we implemented a three stage RISC-V CPU. The stages were divided into Instruction Fetch, Decode/Execute, and Memory/Write Back. We modified the 61C pipeline slightly because we needed to reduce the number of stages, handle synchronous reads from memory, and introduce forwarding logic. We implemented synchronous memories by having them act as a pipeline register between stages. In addition, we implemented branch forwarding, alu forwarding, store forwarding, and csr forwarding. In doing so, we expected our critical path to be from our ICACHE to DCACHE because this path involved the most combinational elements along the same path. Our naive branch prediction scheme was always NOT TAKEN. This was sufficient for our design because there was only a one cycle delay for branch misprediction. Our memory accesses were sped up using a write back cache as opposed to a write through cache. Write back caches allowed us to store data directly in the cache and avoid large miss penalties until absolutely necessary.

2 Datapath and Control

The key components of the datapath are as follows. We have a Register File, Immediate Generator, Branch Comparator, ALU, Partial Store, Partial Load, Control Logic Block, Instruction Cache, and Data Cache. For our three pipeline stages, we have IF, DE/EX, and MEM/WB. We implemented these stages with pipeline registers for program counter, instruction, alu, and pcsel. Our branch prediction scheme is not taken. On branch mispredictions, we have a one-cycle delay and on correct predictions, there is no delay. The one-cycle branch mispredict delay is implemented by sending a no-op(no operation) instruction into stage 3 after the branch result is forwarded into stage 3. For memory accesses, the whole CPU is stalled until the mem_req_ready signal is asserted. We implemented our control logic as standard combinational logic because it requires fewer operations, shorter delays, and less area.

2.1 Datapath and Control Signal Diagram

The illustration below is the datapath structure of the CPU. As a reference, the control signals are as follows.

- Stage 2: | instr2 | immsel | CSRWE | PCSel | BrUn | BrEq | BrLT | MEMWE | CSR | ALUSel |
- Stage 3: | instr3 | RegWE | WBSel |
- Forward Logic : | instr2 | ASel | BSel | inst3 |

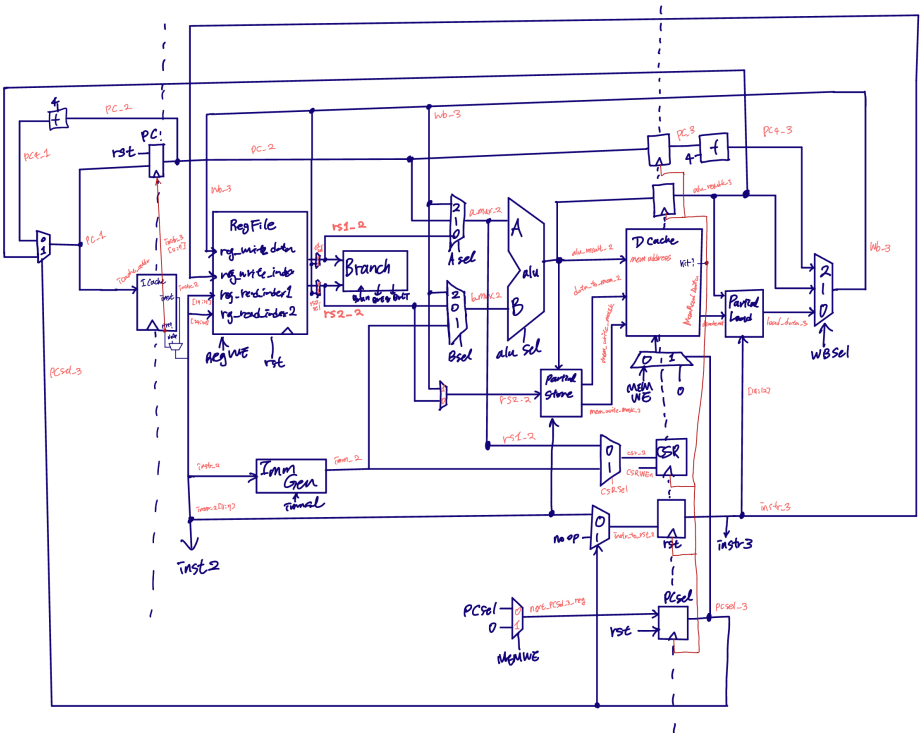

Figure 1: The three-stage pipelined RISC-V Datapath

3 Caches

For this project, we decided to just implement a 4kiB direct mapped cache with write back. The cache has 64 rows of 64B blocks. Our cache was implemented with four parallel data srams and one metadata sram. We modeled our cache as an FSM with four states (Idle, Cache, Read, Write). The metadata sram held two status bits, the valid bit and modified bit, and tag bits. Since one cache row is distributed across four rows of the sram, we indexed into the cache using a combination of index and offset bits. On write back, the cache first writes all data in a 64 Byte block back into main memory. It then loads the requested address into the cache and completes a read or write operation. We decided to use write back because it reduces the number of memory accesses in a program. This improves performance because memory accesses have a long latency. When caches are being accessed, the entire CPU is stalled until the data is written or received.

3.1 Cache Finite State Diagram

Below is a Finite State Diagram of the implemented Write-Back Caching Scheme.

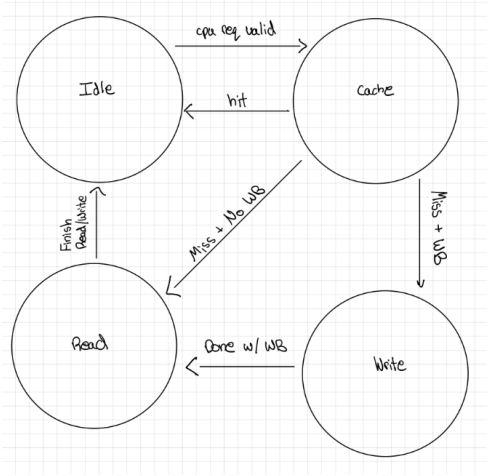

Figure 2: Finite State Diagram

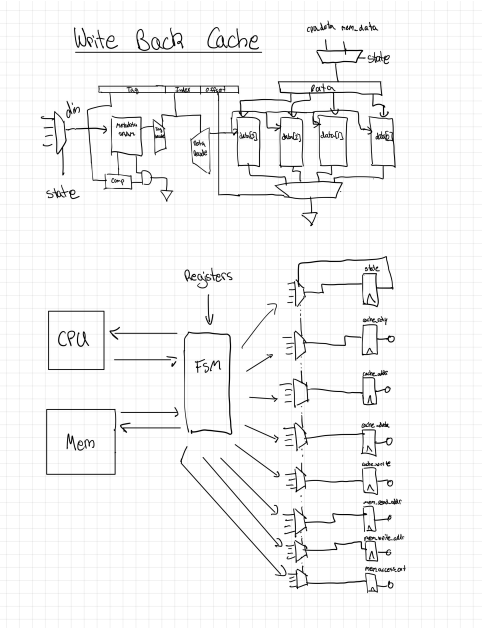

Figure 3: FSM Logic

4 Post Synthesis and Post Place-And-Route Analysis

Below are some of the results from Post Synthesis and Post Place-And-Route(PAR) for the Direct-Mapped Cache CPU implemented.

Chip Design Flow	Critical Path Length / *ns*	Slack / *ns*
Post Syn	42.685	2.315
Post PAR	44.406	0.168

Table 1: Simplified Post Syn and Post PAR Timing Analysis

At the Post Synthesis stage, the critical paths can be explained as follows. The critical paths seen were all from the ICache(Instruction Cache) to the pipeline registers (including DCache or Data Cache) between the 2nd and 3rd stages. This makes sense because the most amount of combinational logic lies within stage 2. The critical paths involved the (regfile + alu) or (regile + partial store). We could have reduced our critical path length by reducing the amount of logic between the stages and optimizing our Verilog Source files. Alternatively, we could split up stage 2 into an Instruction Decode and Execute stage.
Meanwhile, observing in the Post PAR stage, the critical path is from the ICache to the DCache, which is stage 2 in our pipeline. Because it is the longest stage in our pipeline that encompasses instruction decode to the execution stage, it was an expected behavior. In order to further optimize the design, adding another set of pipeline registers in this stage would be helpful.

5 PPA for Direct-Mapped Cache

This section is dedicated to further information collected from PAR.

Clock Period	Operating Frequency	Total Floor Plan Area	Area Density	Total Power
45 ns	22.2 MHz	1093419.3354 um^2	33.75 %	6.00010199 mW

Table 2: Numerical Data from Place-And-Route

6 PAR and the final Floorplan

While there are several methods of performing efficient Place-And-Route, the modules for this project were placed manually for simplicity. If the design had more components that would increase the complexity of the final design, then alternative methods such as the top-down or bottom-up hierarchical placement methods would have been used. Such methods can dramatically accelerate the efficiency of the design flow while meeting the constraints. Such methods will be discussed in a separate report. Below is the image of the floorplan after finishing the manual Place-And-Route process. The congested area of logic is where the 3-stage CPU logic takes place while the surrounding macroblocks are the caches. Amoeba view is also attached for better abstraction of the floorplan.

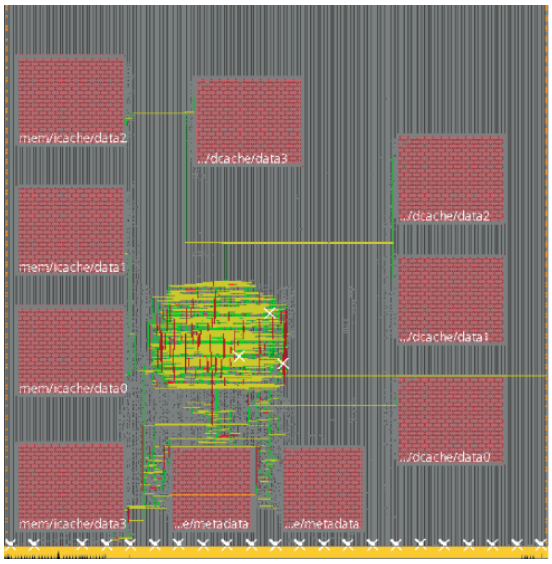

Figure 4: Floorplan Top View

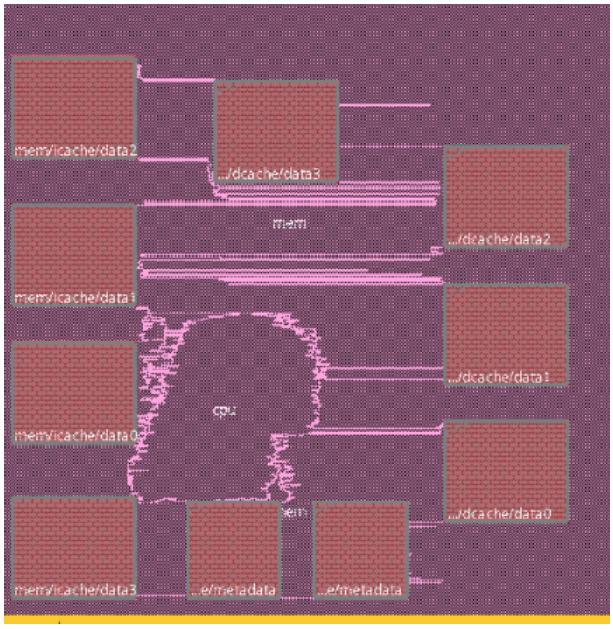

Figure 5: Amoeba View

7 Targeted Analysis

7.1 Clock Tree Debugger View

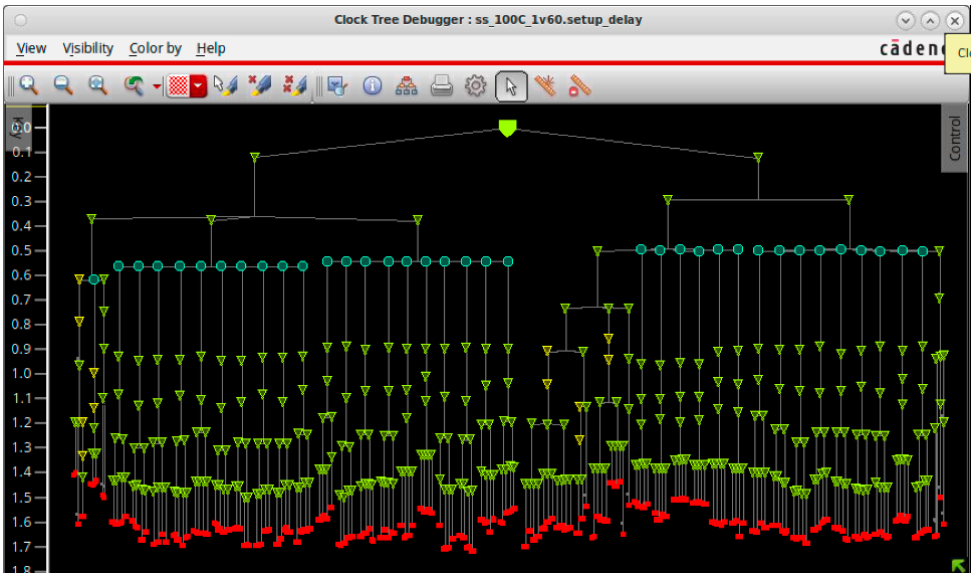

Figure 6: Clock Tree Debugger GUI

The heavily skewed points, or the dips in the tree leave nodes, were mostly the path that begins in the ICache and ends at stage 3, which corresponds to the critical path of our design. Still, considering how the skews of the leaf nodes are not that severe, we can say we still have control over the skew of the clock tree.

7.2 Histogram

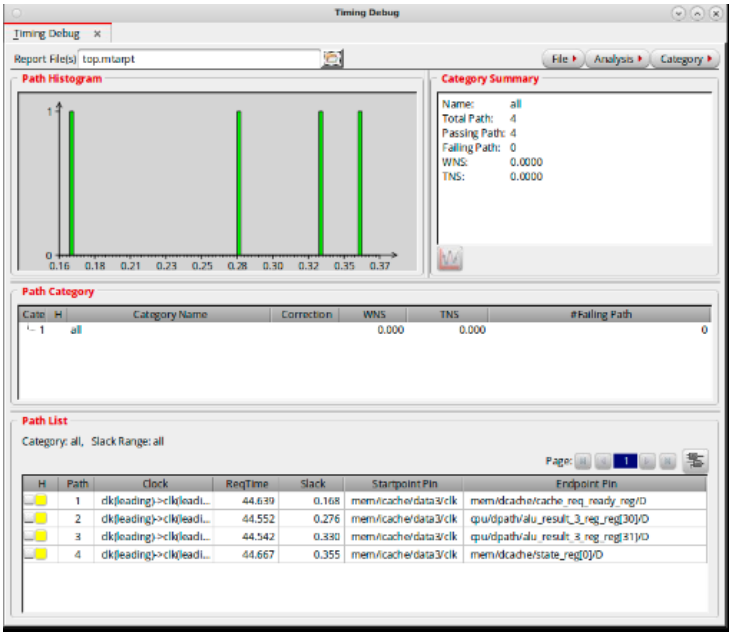

Figure 7: Setup Time

Even though we couldn't see a proper setup diagram due to an innovus bug (since there must be more than just 4 paths), it indicates that our design has low slack. This helps our clock frequency and overall performance of the design.

8 Conclusion

We ultimately did not have time for more optimizations to our CPU, but we wanted to implement a branch taken predictor, which would greatly improve the the cycle count for our benchmarks that are loop heavy. This optimization would greatly improve cycle performance and would not have a significant negative impact on our power or area. We also wanted to reduce unnecessary logic and implement a 4 stage pipeline by splitting up the Instruction Decode and Execute stages. This would allow us to use a shorter clock period. Although a 4 stage pipeline could potentially increase power, we believe the performance improvement would outweigh power concerns as our primary critical path lies in stage 2. We could also improve our cache hit rate by building a set associative cache. With a set-associative cache, our cache area would increase slightly to accommodate further caching logic and an additional SRAM for dual port read from a metadata block. However, our cache hit rate would improve with a set associative cache creating some performance benefit.

8.1 Final Metrics

Benchmarks	Cycles
CacheTest	2,359,234
Final	4,495
Fib	4,197
Sum	13,679,669
Replace	13,679,690
Average	5,945,457

Table 3: Ideal Memory with No Cache

Benchmarks	Cycles
CacheTest	4,817,122
Final	9,093
Fib	8,467
Sum	28,115,954
Replace	28,081,363
Average	12,206,399

Table 4: Direct-Mapped Cache

Benchmarks	Cycles
CacheTest	2.0759
Final	2.0229
Fib	2.0174
Sum	2.0553
Replace	2.0528

Table 5: Ratio Between No-Cache VS. Direct-Mapped

Above are the metrics evaluated via five benchmarks(CacheTest, Final, Fib, Sum, Replace, and Average) provided by the UC Berkeley faculty. The numbers are in cycles it took for each benchmark to run in each environment. The third chart specifically illustrates the cycle ratio between the CPU without cache vs. the CPU with the direct-mapped cache implemented. For instance, if the number of cycles was 10000 and 12345 for the CacheTest benchmark with no-cache and direct-mapped respectively, it would be 1.2345. As a result, the geometric mean of the cycle ratio for all 5 benchmarks was approximately 2.0447. While caches generally improve performance by reducing the average memory access time, the metrics infer that the direct-mapped cache increased the number of cycles. A datapath with a direct-mapped cache might require more cycles to run the same benchmark compared to a datapath without a cache due to several factors related to the behavior of the cache. In a direct-mapped cache, each memory block maps to a specific cache line, leading to conflict misses when different memory addresses compete for the same cache line. These conflict misses force the CPU to fetch data from the slower main memory, incurring additional cycles. Moreover, when a cache miss occurs, the CPU must handle the penalty of fetching and loading the data from memory into the cache, which can significantly delay execution. This situation becomes even more problematic in a pipelined architecture, where cache misses can cause the pipeline to stall, delaying subsequent instructions and increasing the overall cycle count.

Also, if the benchmark's working set, or the set of memory addresses it frequently accesses, is larger than the cache size, the frequent evictions and refills that occur can exacerbate these issues. Without a cache, while each memory access is slower due to direct memory access, the access pattern remains consistent, avoiding the unpredictable delays caused by cache misses and management overhead. Consequently, for certain benchmarks, especially those with access patterns that do not align well with a direct-mapped cache, the overall cycle count can be higher with the cache than without it. Still, if we scale the size of the Cache with an upgraded scheme(such as a 4-way set associative), it is likely to be far more efficient than the datapath without a cache.

Acknowledgements

I am deeply grateful to the faculty and course staff for their time and effort in making this large-scale, partner-based project possible for undergraduate students. I would also like to acknowledge the contributors of Chipyard and Hammer for providing an excellent framework that significantly enhanced our learning experience in VLSI design.