Shanghai Jiao Tong University

Boom Lite CPU Report

Ruihang Zhang Yuetian Wang

January 22, 2026

1 Overview

In the past month we have implemented, as the course project of Computer Architecture, a fully out-of-order RISC-V CPU named Boom Lite[1].

The CPU's architecture is inspired by the Berkeley Out-of-Order Machine (BOOM)[1], though we made considerable simplifications and modifications to the original design to fit our time constraints. The final result is a CPU with more than 10 stages that accomplishes:

1. Full support for the RV32I instruction set, including `JAL`, `JALR`, and bit-masked memory access.
2. Full support for the M extension, using Wallace tree multipliers and restoring division algorithms.
3. Unified memory system based on DRAM, along with ICache and DCache support.
4. Multi-hierarchy branch predictor, combining a two-bit saturating BTB and a RAS.
5. RAT-based OoO implementation to cut down on data movements.
6. Partial OoO memory access as well as MMIO IO support.

The CPU is written in Chisel, and has been synthesized to Verilog using Chisel's built-in FIRRTL compiler. The CPU achieved a clock frequency of over 517MHz on ASAP7 Demo, occupying under 15,000 um^2 of area.

[1]The source code is available at `https://github.com/RayZh-hs/chisel-boom/`.

2 Architecture

2.1 Frontend Design

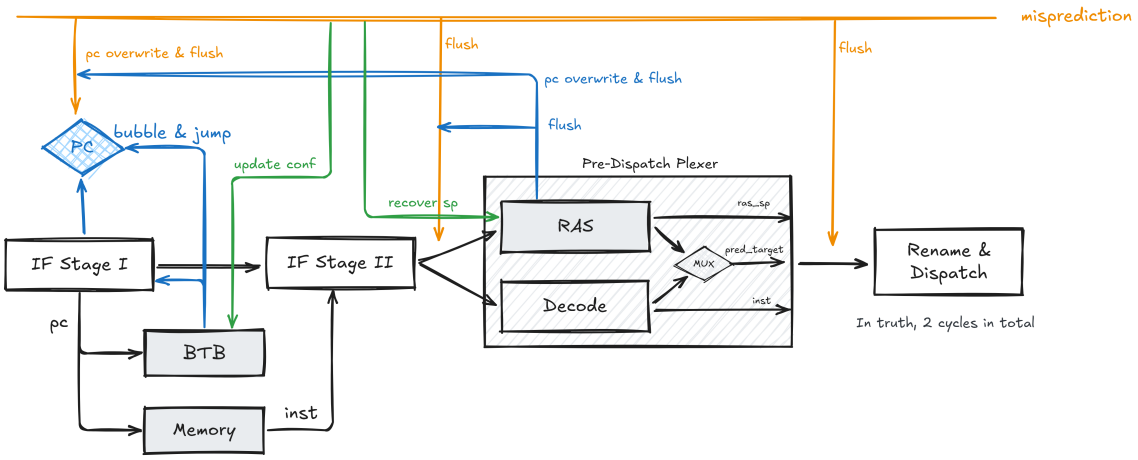

Figure 2.1: Boom Lite Frontend Diagram

The frontend of Boom Lite consists of 5 stages, as is shown in Figure 2.1. We will provide a brief overview of the stages below.

2.1.1 Instruction Fetch

The Fetch stage spans two cycles, the first of which fetches the instruction from ICache as well as consult the low-latency BTB for branch prediction. The second Fetch Stage gathers the fetched instruction and BTB feedback. If BTB predicts a taken branch, the PC is updated and the next-cycle fetch result will be killed.

2.1.2 RAS & Instruction Decode

The instruction is then passed on to RAS (Return Address Stack) Adaptor and the Decoder simultaneously. The Decoder simply performs combinational logic to retrieve fields from the instruction. The RAS Adaptor picks out `JAL` and `JALR` instructions, and takes action on the RAS. According to common ABI patterns, `CALL` and `RET` commands are speculated and the RAS modified thus. The component produces surprisingly sound prediction results, as was discovered in the profiling phase. If the predicted destination conflicts with BTB predictions, the former is disregarded and the frontend before the decoder flushed. The results from the two components are merged in the PDP (Pre-Dispatch Plexer), and the PC, Decoded Bundle, Predicted Target and RAS Stack Pointer are passed on to Rename & Dispatch phases.

2.1.3 Rename & Dispatch

The Rename phase consults the Free List for currently available physical registers and updates the Map Table (i.e. RAT, the Register Alias Table) accordingly, producing

physical register indices for source and destination registers as well as documenting the stale pdst. The renamer also tracks the ROB ID for each instruction (which is deterministic since ROB is in-order enqueued).

The result is piped into a selective router known as the Dispatch Router, routing instructions to various Issue Buffers according to their types. We will cover the four issue buffers in the Backend Design section.

It is worth noting that the PC is **not** stored in the ROB. In fact, unless routed to the Branch Unit, the PC is discarded after the Dispatch phase. This design choice makes our ROB and normal issue buffers extremely small and efficient, having no need to pass around 32-bit information around.

2.1.4 Notes on Branch Prediction

As is mentioned above, the RAS stack pointer is passed down to the Dispatcher. This pointer is only passed to the Branch Unit, which, on misprediction recovery, will flag the old stack pointer to be restored. This will not eliminate the possibility of RAS being modified between the mispredicted branch and the recovery, but since it is speculative the approach is acceptable. We referenced various sources on the design and recovery of RAS[2][4] before agreeing on this design.

Note that there are three cycles between Fetch Stage 1 and Decode, both sides included. This gives time for a full-fledged Global History Lookup Predictor to be implemented, and reusing the wiring of the plexer, but we had not yet implemented it. Had we done so the frontend would have resembled closely to a full TAGE predictor, which is the pattern adopted in BOOM.

2.2 Backend Design

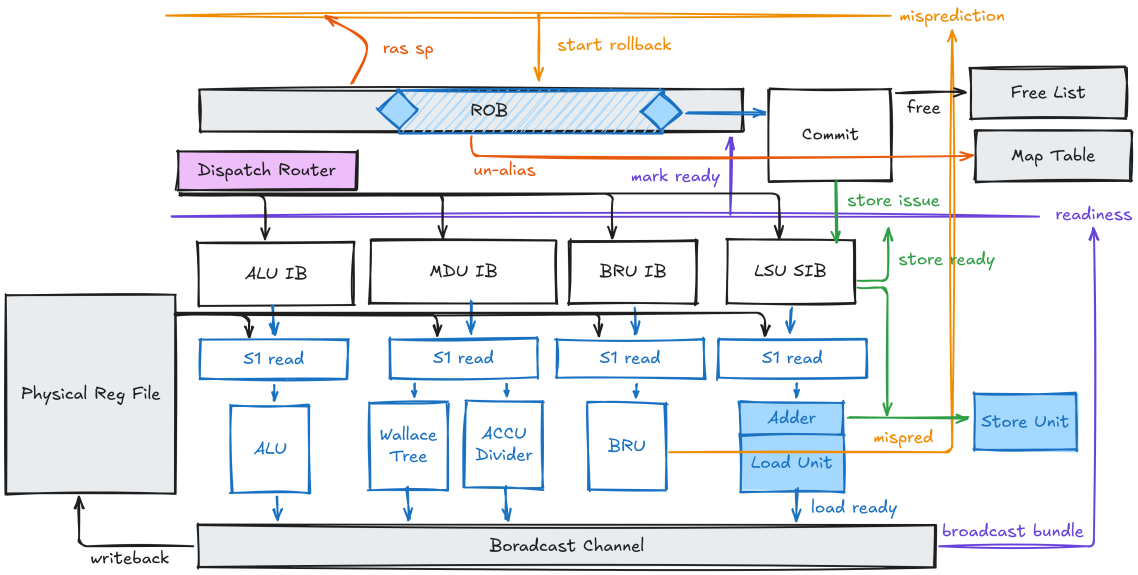

Figure 2.2: Boom Lite Backend Diagram

The backend of Boom Lite follows a fairly standard out-of-order architecture, as is shown in Figure 2.2.

2.2.1 The Re-Order Buffer

In our Boom Lite design, the ROB is a circular buffer whose entries have minimal information.

```
class ROBEntry extends Bundle {
    val ldst = UInt(5.W)
    val pdst = UInt(PREG_WIDTH.W)
    val stalePdst = UInt(PREG_WIDTH.W)
    val isStore = Bool()
    val ready = Bool()
}
```

Considering that there should be around 64 Physical Registers, the entire entry only takes up 19 bits, making it extremely area-efficient.

On misprediction, the ROB rolls back 2 entries at a time, comparing entries it contains with the mispredicted branch's ROB ID. Note that since everything is decoupled, all other units are still functional, alleviating the performance penalty.

2.2.2 Issue Buffers & Functional Pipelines

There are four issue buffers in Boom Lite, each catering to different instruction types.

1. **ALU (Arithmetic Logic Unit) Pipeline**: Handles RV32I arithmetic and logic instructions. Functional units have a 1-cycle latency.
2. **MDU (Multiply Divide Unit) Pipeline**: Handles RV32M multiplication and division instructions. Multiplication is done using a Wallace tree multiplier, taking 3 cycles. Division is done using a restoring division algorithm where two bits are produced per cycle, taking 16 cycles in total.
3. **BRU (Branch Unit)**: Handles all branch instructions. Similar to ALU, the functional unit has a 1-cycle latency. The unit signals misprediction on sight to the Misprediction Line, triggering flush and ROB rollback.
4. **LSU (Load Store Unit) Pipeline**: This unit has been replaced later with LSQ (Load Store Queue) to support OoO memory access. It controls all memory access instructions.

Before the execution of every instruction by the functional unit, all Issue Buffers share a common Data extraction trait which reads data from the Physical Register File. Nothing is stored in the issue buffer itself but is directly passed on the the functional unit for processing.

2.2.3 Broadcasting & Commit

All functional units broadcast results to the CDB, which is responsible for writing back to the Physical Register. It adopts a round robin arbitration scheme among functional units.

Commit is done asynchronously, the same in standard BOOM Design[2]. When a store instruction is committed, in the in-order memory version of Boom Lite, the Committer alerts the front of the Memory Unit to perform the memory write. We will discuss the OoO memory design in the next subsection.

2.2.4 Out-of-Order Memory Access

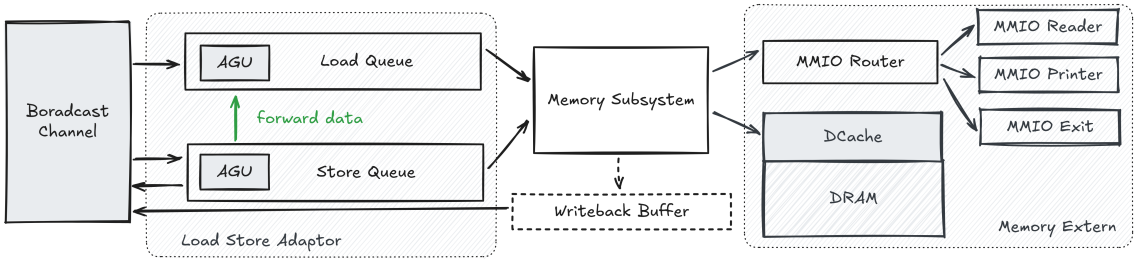

Figure 2.3: OoO Memory Access Diagram

This is an experimental design refactor that has not yet been merged into the main branch. View it on the `feature/lsq` branch[3].

Our Load Store Queue implements partial out-of-order execution and features store-

[2]An overview of standard BOOM architecture can be found here: `https://docs.boom-core.org/en/latest/sections/intro-overview/boom-pipeline.html`.

[3]`https://github.com/RayZh-hs/chisel-boom/tree/feature/lsq`.

to-load forwarding. It consists of a Load Queue and a Store Queue, connected via the Load Store Adaptor.

The Store Queue functions as a circular buffer following a Reservation Station style. Unlike other issue buffers, it captures and stores data upon broadcast. The Address Generation Unit (AGU) within the SQ calculates the effective address once the base is ready to facilitate store-to-load forwarding. A store is marked ready to commit when both address and data are available. It is first broadcast to the ROB for commit; once the ROB confirms the commit, the store is sent to the DCache.

The Load Queue is a non-ordered Reservation Station style buffer that issues loads speculatively. Upon enqueue, it records the current Store Queue tail to track program order.

Every cycle, the unit selects a ready load and checks for collisions against prior stores. Data bypassing is performed, in the process of which memory overlap issues are resolved.

2.2.5 Memory Mapped I/O

Our design fully employs MMIO to handle peripheral interactions. The stable release supports the `Exit` and `Put` devices, while a new `Get` device has been added in the `feature/mmio` branch[4].

Before making their way into DCache and therefore DRAM, memory accesses are checked against the MMIO address map, and those in the target range are sent to the MMIO Router. The router then dispatches the requests to the corresponding device modules, returning data to the CPU if necessary.

[4] `https://github.com/RayZh-hs/chisel-boom/tree/feature/mmio`. This branch is spawned from `feature/lsq`.

3 Performance

We tested Boom Lite on a suite of benchmarks, among which are CPU Simulator Testcases from PPCA. An overview of the performance is shown below[5].

Category	Metric	Value
Branch Prediction	Total Branches	1,015,143
	Total Mispredictions	98,814
	Misprediction Rate	**9.73%**
IPC & Timing	Total Instructions	5,466,223
	Total Cycles	14,414,332
	Average IPC	**0.7010**
Queue Depths (Avg)	Fetch Queue	1.15
	Issue ALU	4.19
	Issue BRU	2.48
	LSU Buffer	5.23
	ROB	21.94
	Avg. Rollback Time	13.26 cycles
Speculation	Total Dispatched	7,850,082
	Squashed Instr.	2,383,859
	Writeback Rate	39.99%
	ROB Commit Rate	37.92%

Table 3.1: General Performance Statistics

Note that the average IPC is calculated as the average number of IPC instead of the ratio of total instructions to total cycles. The few testcases with the most cycles tend to have abnormally low IPC, skewing the overall ratio, and dominating the instruction count. In fact, we observed that the average of per-cycle IPC is around 0.7010, while the overall ratio is only 0.3792, whereas only 2 testcases have IPC below 0.4.

Branch Prediction results outperform our expectations. The addition of the RAS significantly reduced mispredictions, resulting in a drop from 14.83% to 9.73%. We believe that by introducing a global/local history predictor, the rate can be further reduced to around 5%.

We think the overall performance is acceptable for a single-issue OoO CPU, although the IPC can still be improved. In order to identify the bottleneck, we profiled the stages of the pipeline, and the results are shown below.

[5]Benchmarked on stable commit `d27697ced3bfbb90460d24d4375a2dbd9fc8dbca`. All optimizations except OoO memory access applied.

Pipeline Stage	Event / Stall Reason	Count / Cycles	Utilization
Fetcher	*Total Active*	14,415,034	100.00%
	Stall: Buffer	4,215,922	29.25%
Decoder	*Total Active*	12,233,822	84.87%
	Stall: Dispatch	4,548,627	31.55%
Dispatcher	*Total Active*	7,850,082	54.46%
	Stall: FreeList	1,557,644	10.81%
	Stall: ROB	1,095,108	7.60%
	Stall: Issue	1,831,193	12.70%
Issue-ALU	*Total Active*	1,782,905	12.37%
	Stall: Operands	7,610,318	52.79%
Issue-BRU	*Total Active*	1,016,658	7.05%
	Stall: Operands	8,787,193	60.96%
Issue-Mult	*Total Active*	606	0.00%
	Stall: Operands	37,285	0.26%
LSU (Memory)	*Total Active*	9,331,346	64.73%
	Stall: Commit	666,588	4.62%
ROB Commit	*Total Active*	5,466,223	37.92%

Table 3.2: Pipeline stage utilization and stall analysis

After identifying that memory operations are the bottleneck for some testcases, we implemented the OoO memory access design described in the Architecture section. In addition, we applied several optimizations, which we will further discuss in the next major section. Several memory-intensive benchmarks, including the Matrix Multiplication testcase, saw significant IPC improvements.

3.0.1 Matrix Multiplication

Category	Metric	Value
Branch Prediction	Misprediction Rate	10.46%
IPC & Timing	IPC	0.5396
Queue Depths (Avg)	Fetch Queue	1.38
	Issue ALU	1.44
	Issue BRU	0.09
	LSU Buffer	5.43
	ROB	7.41

Table 3.3: Matrix Multiplication: General Performance Statistics

Matrix Multiplication is a highly memory-intensive benchmark, involving tight-looped loads and stores. With the OoO memory access design and optimizations applied, we

observed a significant IPC improvement from 0.3722 to 0.5396.

3.0.2 Add to 100

Category	**Metric**	**Value**
Branch Prediction	Misprediction Rate	2.73%
IPC & Timing	IPC	0.5566
Queue Depths (Avg)	Fetch Queue	1.20
	Issue ALU	0.44
	Issue BRU	0.17
	LSU Buffer	5.89
	ROB	4.28

Table 3.4: Add to 100: General Performance Statistics

Add to 100 is a simple benchmark that sums numbers from 1 to 100. It is not memory-intensive, and the overall performance exceeds that of Matrix Multiplication.

Note that the branch misprediction rate is significantly lower here since a simple BTB suffices for the large-constant loop structure.

4 Synthesis

We utilized Silicon Compiler[6][3] to synthesize our design from Verilog generated by Chisel. The synthesis was performed on the ASAP7 Demo, a 7nm process node. It yielded the following results.

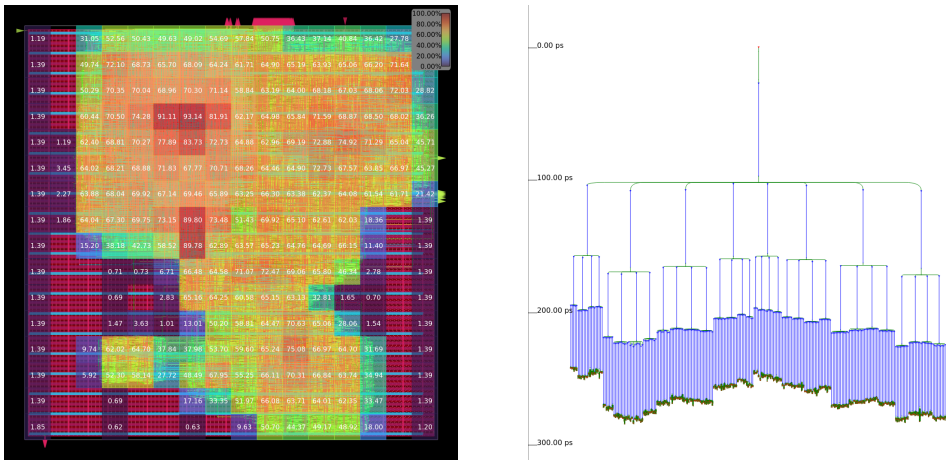

Figure 4.1: Placement Density and Clock Tree on ASAP7 Demo (w. DRAM)

Metric	**Syn**	**Floorplan**	**Place**	**CTS**	**Route**	**Signoff**
Area & Utilization						
Cell Area (μm^2)	5954.06	5637.14	6196.85	6353.44	6353.44	6353.44
Utilization (%)	—	37.97%	41.74%	42.79%	42.79%	42.79%
Timing & Frequency						
Setup WNS (ns)	—	-10.675	-1.253	-0.848	-0.905	-0.934
Fmax (MHz)	—	—	443.92	541.09	524.93	**517.17**
Power						
Peak Power (mW)	—	29.77	32.25	35.38	36.08	36.19
Leakage (mW)	—	0.006	0.007	0.007	0.007	0.007
Design Complexity						
Cells	47,854	44,608	46,281	47,256	47,256	47,256
Registers	7,558	7,558	7,558	7,558	7,558	7,558
Nets	48,288	43,236	44,909	45,769	45,769	45,769
Runtime						
Task Time (s)	34.30	50.67	346.32	950.37	419.15	311.67

Table 4.1: BoomCore Physical Design Flow Summary (ASAP7)

[6] `https://www.siliconcompiler.com/`

5 Optimization

Throughout the development of Boom Lite, we have applied several optimizations to improve performance and efficiency. Some succeeded whereas others did not yield expected results. Below is a summary of some notable ones we have tried.

5.1 Branch Prediction

The original design consists of a naive BTB predictor, where the first jump of a hashed PC determines all the jumps thereafter. This approach has a high misprediction rate. In harder tests it can reach up to 25%.

Test Case	**Misprediction Rate**		
	Original	**2-Bit Sat.**	**BTB + RAS**
fibonacci	30.51%	34.45%	15.18%
matmul_8x8	33.76%	29.70%	29.14%
superloop	11.14%	8.11%	7.97%
hanoi	23.75%	26.35%	18.23%
bulgarian	18.34%	16.58%	6.82%
queens	29.52%	24.92%	24.95%
AVERAGE	**24.50%**	**23.35%**	**17.05%**

Table 5.1: Branch Predictor Accuracy Comparison

Introducing a 2-bit saturating counter for each BTB entry improved the accuracy only slightly, but the addition of RAS significantly reduced mispredictions, especially in recursive testcases like `fibonacci` and `hanoi`.

5.2 Cache

After migrating to unified DRAM, read and write has become more expensive when they cover consecutive bytes. The DRAM + Cache system does not improve performance in the simulation since all the memory accesses are 1-cycle only in the simulator, but in real-world scenarios the cache would greatly reduce memory access latency. In sight of this, we only verified the correctness of the cache system without further optimizations.

5.3 Load Store Queues

Before Out-of-Order Memory Access, the bottleneck of multiple testcases have been identified as the Load Store Unit (later the Memory Subsystem). Introducing OoO MA brought up the IPC considerably.

To make the most out of OoO MA, we attempted to create a bypassing unit between Load and Store Queues, which takes into account memory overlaps. This approach did increase the IPC further (but not as much as we had expected), but it had brought correctness issues into the system that we have yet to diagnose and resolve[7].

A curious optimization is that we discovered that our initial assumption that the Broadcast Channel seldom fills up, thus seldom blocks issue buffers, is actually wrong in the context of memory access. This happens when Store Memory Access commands awaiting Commit block the route to the CDB, causing operand-ready memory actions to pile up inside the issue queue. We resolved this issue by adding a small FIFO buffer before the CDB, which actually boosted IPC by a noticeable margin.

5.4 Re-Order Buffer

As is mentioned in the Architecture section, our ROB design is extremely compact, containing only essential information. That we keep PC information and Physical Register data minimal greatly reduces the area of our core.

An optimization we applied to the ROB is to decouple its rollback from the rest of the pipeline. On misprediction, the ROB rolls back independently while other units continue functioning; that is, all other units have a one-cycle delay in flushing only. This reduces the performance penalty of mispredictions.

In addition, we implemented a dual-flush mechanism in ROB, where two consecutive commands can be simultaneously flushed. This reduces the rollback time on mispredictions by nearly half, a considerable boost to performance.

We selected instruction-level flushing as opposed to snapshot-based flushing because we would have to stall if the snapshot buffer is full. By adopting our decoupled approach, we would not need to stall, though the rollback time is longer in our case.

[7] Currently the `feature/lsq` branch is still under debugging and development, since it is known to fail on deep recursion testcases like `bulgarian` and `queens`.

References

[1] Christopher Celio, David A. Patterson, and Krste Asanović. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical Report UCB/EECS-2015-167, Jun 2015.

[2] Veerle Desmet, Yiannakis Sazeides, Constantinos Kourouyiannis, and Koen De Bosschere. Correct alignment of a return-address-stack after call and return mispredictions. In *Proceedings of the 4th Workshop on Duplicating, Deconstructing and Debunking (WDDD)*, June 2005.

[3] Andreas Olofsson, William Ransohoff, and Noah Moroze. A distributed approach to silicon compilation: Invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, page 1343–1346, 2022.

[4] Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, and Douglas W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 259–271. IEEE Computer Society, November 1998.