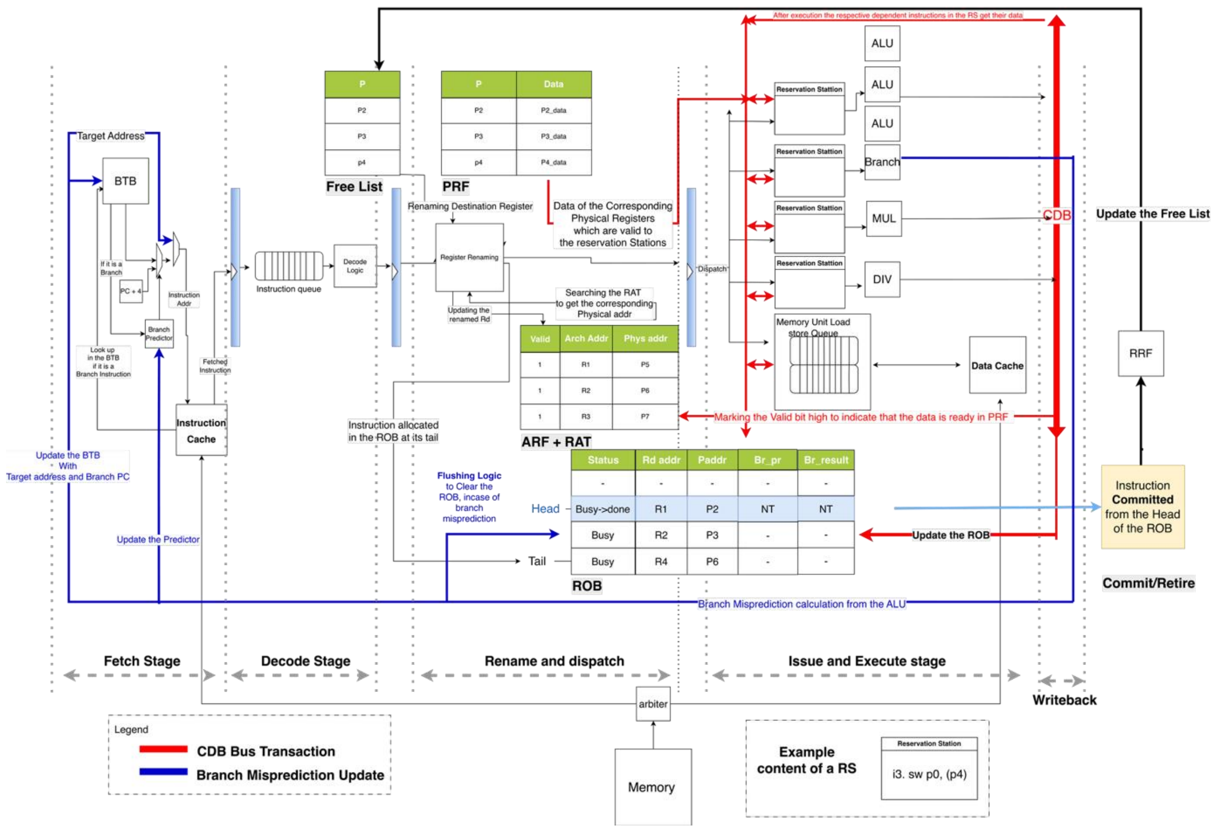
MP_OOO TRISTATE

Introduction and Project Overview:

Implemented a 2-way superscalar, out-of-order processor based on explicit register renaming. The architecture incorporates advanced features like GSHARE branch prediction with a Branch Target Buffer (BTB), a Next Line Prefetcher, pipelined Instruction and Data Caches along with a Split Load Store Queue (LSQ) with out-of-order loads and data forwarding from uncommitted stores.



Milestones:

Checkpoint 1

- Instruction Queue created with parameterized depth and width support
- Cache line Adapter created from BMEM and cache interface
- Cache Arbiter created as a dummy block, separate Instruction Cache and Data Cache to be created with priority table tbd
- Line Buffer created that caches the hit Cache line and serves instruction on each cycle
- Module specific testbench for IQ and Cache line adapter created and tested
- Line buffer behaviour validated with top_tb by passing assembly files and cross-checking committed instructions at ID stage through Verdi
- Block diagram created
- BMEM burst support yet to be fully leveraged

Checkpoint 2

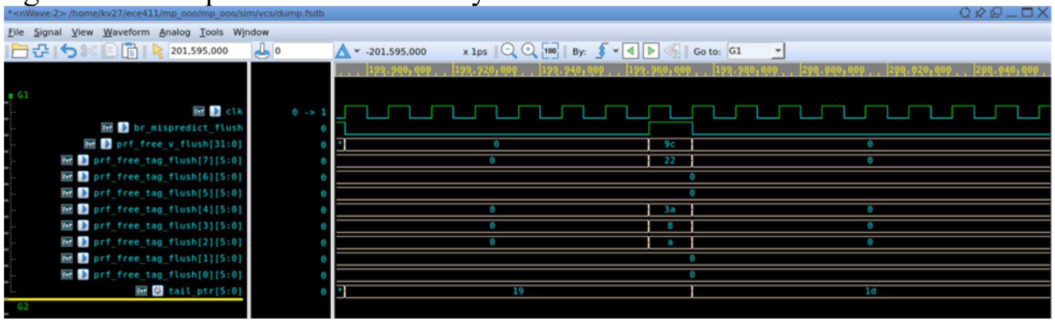
- Verified all RV32I immediate and RV32I register-register ALU instructions. auipc instruction was not included in this checkpoint.
- Integrated the Synopsys MUL/DIV IPs into our execute stage and were able to run all RV32M multiply/divide instructions correctly.
- The design supports out-of-order execution, so independent instructions can run and finish earlier even if older ones are stalled (for example, waiting on MUL/DIV).

Checkpoint 3

- Added support for the remaining RV32IM features needed for full program execution, including memory and branch instructions.
- Integrated both Instruction cache and Data cache with the CPU and connected them to the DRAM model using shared arbitration logic.
- Added support for all control instructions (BR, JAL, JALR) and AUIPC, including correct PC redirect behaviour and flush logic on taken branches and jumps. Planned for the remaining RV32IM features needed for full program execution, including memory and branch instructions.

<u>Advanced Design Features:</u>

1. **2-way Super Scalar**
   - **Fetch and Decode:**
     Two instructions (PC0 and PC1) are fetched per cycle from the instruction cache. Each instruction is at a different index within the same cache block.
   - **Rename Stage:**
     A new physical register is mapped to the architectural destination register of each instruction. Both instructions request for physical registers in the same cycle while performing a two-way rename.
   - **Free List:**
     The free list maintains a pool of free physical registers using a circular queue. It supports multi-pop, meaning two physical registers can be allocated in one cycle. The Free List supports multi-push, so two or more (upon branch mispredict) registers can be pushed in the same cycle.


*<nWave:2> /home/kv27/ece411/mp_ooo/mp_ooo/sim/vcs/dump.fsdb
File Signal View Waveform Analog Tools Window
201,595,000
0
-201,595,000
x 1ps
By:
Go to: G1
199,900,000
199,920,000
199,940,000
199,960,000
199,980,000
200,000,000
200,020,000
200,040,000
G1
clk
br_mispredict_flush
prf_free_v_flush[31:0]
prf_free_tag_flush[7][5:0]
prf_free_tag_flush[6][5:0]
prf_free_tag_flush[5][5:0]
prf_free_tag_flush[4][5:0]
prf_free_tag_flush[3][5:0]
prf_free_tag_flush[2][5:0]
prf_free_tag_flush[1][5:0]
prf_free_tag_flush[0][5:0]
tail_ptr[5:0]
G2
0 -> 1
9c
22
3a
8
a
19
1d

Single cycle flush

Edge Case (Same Destination Register): If both instructions write to the same architectural register, each gets a different physical register. The youngest instruction's mapping becomes visible upon commit.

- **Reorder Buffer (ROB):**
  Every instruction that has been renamed is given a new ROB entry, when two instructions come in from rename, the ROB tail pointer is incremented by two. For execution, instructions execute as soon as their operands are ready. Completion is broadcast on the Common Data Bus (CDB). ROB entries are marked ready independently and out of order. The ROB commits two instructions from the head per cycle. If the first instruction is not ready, the second cannot commit even if it is ready.

Performance Improvement:

Testcase	**IPC (old)**	**IPC (improved)**
coremark	0.094833	0.171152
fft	0.126551	0.329335
merge_sort	0.100001	0.193067
aes_sha	0.102967	0.184368
compression	0.143562	0.193379

Area before superscalar: 26986.790485 um2
Area after superscalar (without any other feature): 29202.3848 um2

2. **Prefetcher**
- A prefetcher helps hide instruction-fetch latency by initiating memory accesses early, so the pipeline is less likely to stall waiting on DRAM. In our system, we observed that even in the absence of branch misprediction flushes, the instruction queue still became empty about 15–20% of the time. This indicated that front-end bubbles were frequently caused by instruction supply limitations rather than control-flow recovery, and that there was meaningful opportunity to overlap instruction fetch with ongoing execution. To exploit this slack, we implemented a next-line prefetcher integrated directly within the I-cache control logic. Under sequential execution, the prefetcher opportunistically requests the next cache-line address ahead of demand. When the BTB predicts a non-sequential path, the prefetcher instead targets the predicted next fetch address, increasing the likelihood that the correct line is already resident in the I-cache when the core redirects the PC.
- Integrating this mechanism introduced a few non-trivial design challenges. First, the cache arbiter required a substantial redesign to support non-blocking BMEM transactions, allowing prefetch requests to be issued and serviced without unnecessarily stalling demand fetches. Second, to avoid polluting the memory interface with redundant reads, we added forwarding and de-duplication logic within the arbiter, ensuring that if a demand request overlaps with an in-flight prefetch (or vice versa), the design can reuse the pending transaction rather than launching an additional BMEM access. Finally, we had to account for SRAM write-delay behaviour

during line fills; to maintain correctness and timing robustness, we introduced line buffers that stage returned cache lines before committing them into the SRAM arrays. Overall, this implementation improves front-end utilization by reducing preventable fetch stalls, and the impact is quantified in the profiling results presented in the following section.

Performance Improvement:

	With Prefetcher			**Without Prefetcher**		
Benchmark	**Hit Rate %**	**AMAT (ns)**	**Avg Miss Resp (cycles)**	**Hit Rate %**	**AMAT (ns)**	**Avg Miss Resp (cycles)**
Coremark	96.97%	1.335	12.253	96.5%	1.447	13.891
AES SHA	86.76%	2.486	12.285	80.78%	3.513	14.084
Compression	100%	1	11.321	99.99%	1.001	14.375
FFT	99.17%	1.092	12.838	97.22%	1.403	15.93
Mergesort	99.73%	1.031	13.126	99.61%	1.047	13.45

Benchmark	**Hit Rate Delta (%)**	**AMAT Improv. (%)**	**Avg Miss Resp Improv. (%)**
Coremark	0.47%	7.74%	11.79%
AES SHA	5.98%	29.23%	12.77%
Compression	0.01%	0.1%	21.25%
FFT	1.95%	22.17%	19.41%
Mergesort	0.12%	1.53%	2.41%
AVERAGE	**1.7%**	**17.55%**	**13.53%**

3. **Pipelined cache:**
To improve throughput and meet timing, we redesigned both the I-cache and D-cache as pipelined caches. In the earlier (non-pipelined) version, the controller often had to return to IDLE before it could accept a new request. That added wasted cycles even when requests were a stream of continuous hits. With the pipelined I-cache and D-cache, a hit can be served in the very next cycle, and the cache can take a new request at the same time, which improves overall throughput. This approach was applied to both caches: the D-cache is 4-way set associative, and the I-cache is a 16-set direct-mapped cache.

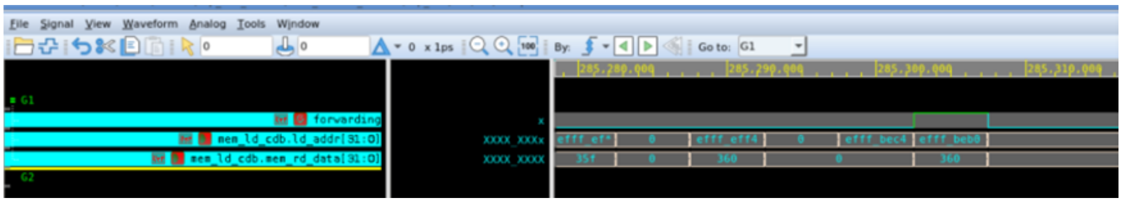
Implementing this introduced a few practical challenges. First, SRAM write update delay could cause timing/correctness issues, so we used internal line buffers to stage data before writing it into the SRAM arrays. Second, we added forwarding for D-cache writes so a load can see the latest store data without waiting for the array update. Third, we changed the UFP request behaviour, so requests are sent as clean one-cycle pulses, making the handshake consistent across pipeline stages. Finally, we ensured the cache can check and serve N-way accesses in one go (parallel tag/way handling), so associativity does not reintroduce extra stall cycles on hits.

Performance improvement:

Test	**Overall Hit Rate (%)**	**AMAT (ns)**	**Miss Penalty (cycles)**
Coremark	99.84%	10.234	15.101
AES SHA	97.86%	13.669	17.131
Compression	99.13%	11.239	14.166
FFT	98.88%	11.631	14.539
Mergesort	98.30%	12.77	16.326
AVERAGE	**98.80%**	**11.909**	**15.453**

4. **Load Store Unit**

- Implemented a split LSQ in which loads can execute out-of-order in between in-ordered stores and can be issued OOO with respect to stores to a different address. The design also supports forwarding of data from uncommitted stores, whose addresses exactly align with the load address.
- Marking load instructions ready – Marking load instructions ready - Load instructions are marked ready if the load address is calculated (valid) and there are no older stores to the same address locations or misaligned address locations and if there are then forwarding should be possible.
- Load queue push pointers – Due to the out-of-order nature of the design, load instructions may complete and exit the queue from non-sequential locations. Hence, a Free List Stack is implemented to track available indices. The "push pointer" for new incoming loads is determined by popping the next available index from this stack.
- The forwarding logic – For every load at the time when it gets pushed into the load queue the store tail pointer relative to it is stored, which is used later in the forwarding logic to check all the stores older than that load and see if the data is ready to be forwarded



Single cycle data transfer due to forwarding

Performance Improvement:

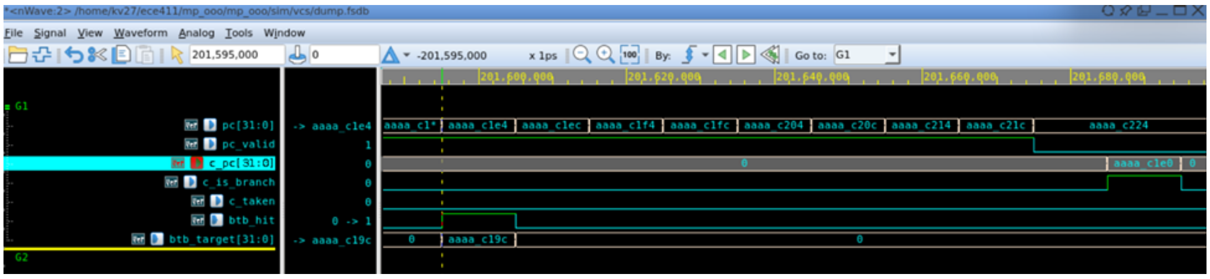
Benchmark	Total loads	Forwarded loads	Percentage
coremark	66,185	1,007	1.52%
fft	184,408	3,692	2.00%
merge_sort	87,292	391	0.45%
aes_sha	235,258	3,033	1.29%
compression	47,124	0	0%

- Forwarding and out of order execution of load is considered a performance improvement because if they executed in order then the other dependent instruction would have been stalled.
- Some Negative effects on the performance:
- The Load Store Unit (LSQ) is the largest single component in the CPU with an area of 173,562 um2 (32% of the CPU's area).
- Combined with superscalar (addition of three ALUs) a Critical path got created from Load ready logic to Reservation Station input through CDB. This prevented us from synthesizing the design at clock frequencies greater than 100Mz.
- Solution to this would have been to reduce the search depth for the store queue, like considering only the top five stores for forwarding.

5. **Branch Target Buffer (BTB)**
   - To minimize fetch stage stalls and offer early target prediction for control flow instructions, the CPU includes a Branch Target Buffer. The program counter's low order bits are used to index the BTB, which then stores the entire program counter as a tag along with the anticipated target address and a valid bit to guarantee accuracy in the event of index aliasing.
   - BTB entries are only changed when a branch is resolved and verified as taken using feedback from the ordering buffer to prevent speculative corruption.
   - Flip flop-based arrays are used to implement the BTB, which is accessed synchronously via a request response interface. This enables pipelining of the lookup and keeps BTB access out of the fetch critical path.

Why flip-flops over SRAM?

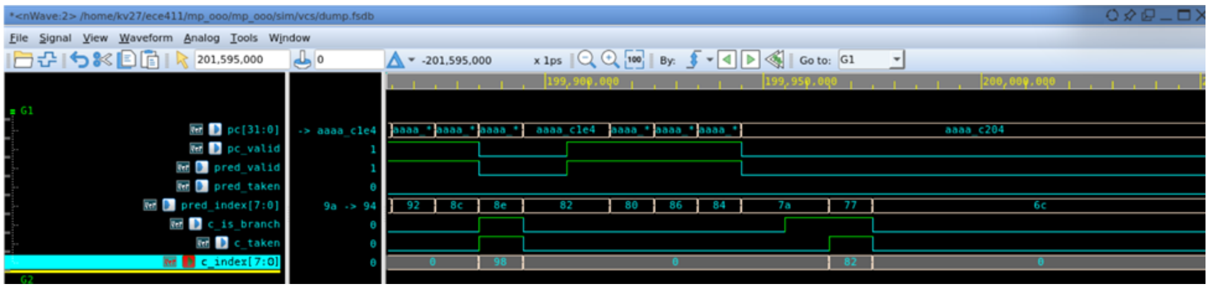
- Flip flop-based storage is used instead of SRAM because it reduces the unnecessary access latency.
- To reduce fetch time, BTB lookup is carried out concurrently with branch direction prediction.
- Using flip flops eliminates SRAM overhead, reducing area and dynamic power for a low-entry count BTB.



BTB hit

6. GSHARE Branch Predictor

- A GSHARE branch predictor is used in the design to increase the accuracy of branch speculation.
- To choose an entry in a 256-entry Pattern History Table made up of two-bit saturating counters, the results of the most recent branches are stored in an eight-bit Global History Register, which is XORed with the branch program counter index.
- Predictor modifications only happen at branch resolution, which makes recovering from mispredictions easier.



Accurate branch prediction

Why flip-flops over SRAM?

- SRAM would add additional access delay, area overhead and extra design complexity.
- Flip-flops were ideal for a timing-critical frontend structure because they offer single-cycle access and make debugging easier.

Performance improvement:

Testcase	Prediction rate	IPC (old)	IPC (improved)
coremark	75.61%	0.406972	0.558020
fft	88.21%	0.556615	0.874472
merge_sort	60.68%	0.496049	0.586789
aes_sha	97.26%	0.435218	0.579752

compression	99.79%	0.577548	0.667850

Average Prediction rate: 84.31%

Conclusion:

- Passed all metrics at 100 MHz but got into a timing violation for higher frequencies.
- Focused too much on implementing the individual advanced features rather than analysing if they were actually feasible in terms of area and other metrics.
- Implemented a parameterized N-WAY superscalar design which we had to drop and stick to 2-way superscalar due to area and timing issues.
- Should have taken an analytical approach during implementation.