ABSTRACT
In this paper, we present the design of a 2-way super-scalar out-of-order processor core designed and implemented as a part of final class project. Our design of the core is based on what we refer as T3 algorithm. 2-bit local history branch predictor, instruction prefetcher, return address stack, LSQ, non-blocking D-cache and stride prefetcher are advanced features of the core. Our main emphasis has been on the performance and the correctness. Our efforts in these direction have led to a very good clock period and correct functionality of the core for all the benchmarks that were given to us. Effect of various parameter like branch predictor initialization, LSQ size, and number of instruction prefetching is analyzed in this paper.

1. INTRODUCTION
In the following subsections we describe all the features of the core in detail.

1.1 Basic blocks
All the structures required for the T3 are implemented for example PRF, free list, saved free list, RAT and RRAT. The number of the PRF entries is 64 in the final implementation though this is also configurable. The priority selector keeps 2 free physical register numbers every cycle if they are available. If the free physical registers are exhausted the stall is generated. Similarly stall is generated when ROB or RS become full.

1.2 Execution units and CDB
We have used 2 ALUs, 1 pipelined multiplier, 2 CDBs and 16 RSs. We have seen in our benchmarks that only for maximum of 4 percent the stall is generated due to the RSs being full. Also for the arbitration on CDB the load queue is given the highest priority. The CDB is also observed to take care of the "ships passing in the night". This forwarding also happens in the pipeline registers of ID/Alloc stage in case of stalls.

1.3 RAT and RRAT
We have used 64 physical registers. Therefore the table has 64 entries. In addition RRAT is maintained for copying into the RAT in case of a branch squash. Both have standard implementation. The entries for the registers that are assigned free physical register are updated. Entry number 31 always points to physical register 31 which is permanently set to zero. The entries of the RAT are all initialized with this PRn so as to cause renaming of all the new uninitialized registers to PR 31. Thus for such variables always 0 is read from the PRF.

1.4 Load Store Queue
Load and Store are bypassed from RS to go into the special structures namely Load queue and store queue. In that way the queues also play the role of RS for memory operations. Stores from the store queue are removed when the store is going to commit. Loads from the Load queue are launched when the addresses of all the preceding stores are known and are different from the address of this load. In case the address of this load matches with the address of any preceding store the value of the load is taken from that store instead of memory.

Also the size of load and store queues are parameterizable. For the final submission purpose we have kept the load and store queue sizes at 4 each. All the loads that are hit in the Dcache are immediately retired from the queue while the loads that miss are put into the auxiliary queue. This has been done to keep the load queue short thereby reducing the fanout while still having many loads in the waiting. For the purpose of maintaining the load store order the tail of the store queue is kept in the load queue entries. This entry is decremented till the load leaves the load queue or it becomes zero.

Both queues are assigned one dedicated 32 bit adder for calculating the addresses. Only one of the entry is allowed to use the adder once the operands become available. This was done to reduce the overhead of the redundant adders as only one load or store can go to memory at a time. After the calculation of the address it is stored in the queue entry itself for comparison and passing to the memory.

Both of these queues cause the structural hazard once they become full. But the stall is not generated until a load instruction tries to enter the load queue for load queue full case or store instruction tries to enter the store queue.

1.5 Branch Predictor
As part of the core, we have implemented a 32-entry 2-bit local history branch predictor. In the branch predictor module, 32-entry Branch Target Buffer (BTB) is also added and BTB is fully tagged. PC of the branch to predict is supplied by IF stage and BTB and prediction state of the branch predictor is updated whenever a branch commits. If the predictor state is 00 or 01 then the corresponding branch is predicted not taken otherwise branch is predicted taken and a target address is supplied from BTB. For the testcases we have been provided, we have observed that 01 initialization performs marginally better than 00 initialization. So,
prediction state is initialized to 00.

1.6 Return Address Stack
In our implementation of RAS, instructions enter into the stack at the same time when they are issued to Reservation Stations. In assembling of the instructions, jsr instruction is converted into lda and then jsr on the register loaded by jsr and ret $x is used in some of the test programs. All these compication and popping of stack from the if stage makes implementation of RAS little difficult.

1.7 Instruction Prefetcher
Our instruction prefetcher module is very generalized and can be very easily modified to prefetch any number instructions upto 64. Instruction prefetcher’s requests to memory have less preference than IF stage’s requests to memory, which results in priority to ICache miss instruction fetch than the instruction IPrefetch is fetching. Also, at every ICache miss IPrefetch starts prefetching from the missed address. As seen in the analysis section there is little effect of prefetching on CPI more than 4 instruction other than btest1 and btest2. But, btest1 and btest2 show an improvement in CPI and there is no negative effect on other testcases, so we have turned on aggressive instruction (60) prefetching in the code we have submitted.

1.8 Stride Prefetcher
We attempted a data based stride-prefetcher. It has the following characteristics. Whenever a load request is sent to the dcache, it is also sent to the data prefetcher along with its NPC. This load request along with its address is stored in a table. The NPC of this load is used as a marker to distinguish the load from other loads. If this load request comes again, we calculate the difference of the earlier stored address and current address. This is saved along with the NPC as the stride of the load. Then we add the stride to the current address and send a request to memory. Now 2 cases may arise, if next time the same load comes without a stride in its adress then it is put in a rejected load table. But if next time the load comes with the same stride then we send a request again to the memory. One optimization that has been done in the later part of the project in cases of loads with stride is we dont launch a request of address + stride, instead we launch a request of address + 3*stride. This has been done because many times the next load with same NPC comes and is found to be a miss in Dcache. So by early launching such loads we help with the miss.

1.9 Data Cache Controller
We have implemented a non-blocking direct mapped data cache (1 KB in size), that supports “hits under miss” and “miss under miss” with support for up to 15 outstanding misses. The controller returns a response, which is just like the response from memory. Evictions are straightforward as it is a direct mapped cache. The cache line to be evicted is identified in one cycle, its contents stored separately and the store request to memory along with the data is sent in the next cycle. We faced a number of challenges while attempting to accept all requests unless the outstanding misses queue becomes full or the memory returns a response of 0. This is because of the interaction of the incoming data from the memory (of a previous load miss) and the present request, since there is only one read and one write port. To solve this and other challenges, we added on to the response of the dcache-controller, which returns zero in the following cases:

- the memory returns a response of 0
- outstanding misses queue becomes full
- data returned from memory is valid and there is a store request
- evicted data and address needs to be sent in this cycle, and the present request misses in the cache
- there is an outstanding load miss and a store request comes in for the same address

Also, the present request is always given priority over outstanding load misses. Even though this scheme is not fair and might lead to starvation of outstanding requests, we conjecture that such a case, where the outstanding request is not granted even though its data has arrived for more than a few cycles, is extremely rare and does not hurt performance. Our assumption is also backed by the fact that Dcache requests have the highest priority among requests to memory. This helped avoid some complexity.

2. ANALYSIS
The core we have implemented was able to pass all the testcases provided to us and in addition all the testcases provided by Doug Li also ran successfully. Figure 1 provides CPI of all the testcases and in terms of performance parallel.s performed best and CPI is close to ideal number .5. Also, 7 of the testcases have CPI below 1. objsort.s was the worst perfoformer and this could be because we don’t have a set-associative data cache. In the following subsections we describe results and analysis of varying various parameters.

2.1 Branch Predictor Initialization
In this experiment, we analyze the effect of branch predictor initialization. In our implementation we have 2-bit local history branch predictor. Branch predictor can be initialized to either 00 or 01. In the figure 2 CPI of all the provided testcases is plotted with branch predictor initialized to 00 and 01. In almost all the testcases, 01 initialization performs better than or atleast as good as 00 initialization. This can be explained from the fact that when a testcase has branch which is part of a loop and is taken most of the time then 01 initialization will cause the branch to correctly predicted atleast more than 1 time than 00 initialization.
2.2 Instruction Prefetching

In this experiment, we analyze the effect of instruction prefetching. Our design of core has an instruction prefetch unit which can be easily modified to prefetch a different number of instructions for a particular compilation of the core. In our implementation, whenever there is ICache miss we start prefetching from the miss address. Figure 3 shows the effect of prefetching with instruction prefetching up to 2, 4, 8, 16, 32 and 64. On Y-axis CPI is plotted while X-axis shows names of various testcases. If we leave aside btest1 and btest2 testcases then in most of the testcases there is little or no effect of prefetching on CPI after 4. btest1 and btest2 show a continuous decrease in CPI with increased number of instruction prefetching. Since, there is no negative effect of aggressive instruction prefetching on CPI so in the version of the core which we turned in, prefetching is kept at 60.

2.3 Load Store Queue analysis

Referring to Figure 4 we can see that there is gain for all the benchmarks that have significant number of the loads and stores with increase in the sizes of both of these queues. By doubling the size of both of load and store entries the benchmarks, that have consecutive loads or stores, show improvement. But this improvement is not much which in our opinion is attributable to store to load forwarding to some extent and also to the fact that some speculative loads are launched that would not be launched in the original program flow. These speculative loads and stores consume the memory bandwidth as the dcache misses have the highest priority thus reducing the bandwidth for the prefetcher and I-cache misses. We have thus kept the queue sizes of 4 each to reduce the impact on Tclock as the number of comparisons in the load queue will go considerably up and the CDB fanout would also increase.

2.4 Commit Rate Analysis

For the benchmarks with fewer memory references and lot of ILP we can see from Figure 5 that the two-instruction commit rates have been high. In contrast the benchmarks with many branches and memory references thus lesser ILP have fared poorly at the two-instruction commit rates. For such benchmarks the share of zero-instruction commits have correspondingly gone up. In short benchmarks like sort.s with higher share of back to back instruction dependencies the one-instruction commit rates are similar to two-instruction commit rates.

2.5 Stride Prefetcher Analysis

Figure 2: This figure shows the effect of 2-bit local branch predictor initialization on performance.

Figure 3: This figure shows the effect of number of instruction prefetching on the performance of various testcases.

Figure 4: CPI variation with varying LSQ size.

Figure 5: This figure shows the variation of number of instruction committing per cycle.
This is an interesting testcase that we found during our testing. Following is the testcase.

```
lda $r1, 0x200
lda $r4, 0x1
loop : ldq $r3, 0($r1)
addq $r3, 0x1, $r3
mov $r1, $r2
addq $r1, 0x8, $r1
addq $r4, 0x1, $r4
stq $r3, 0($r2)
cmpeq $r4, 0x1f, $r5
bne $r5, loop
callpal 0x555
.align 9
.quad 2, 4, 6, 8
.quad 10, 12, 14, 16
.quad 18, 20, 22, 24
.quad 26, 28, 30, 32
```

In this case there is loop within which a memory address is loaded, 1 is added and is stored back to the same address. In this case there would be 15 load miss and hence the CPI is expected to suffer as consecutive stores would fill up the store queue. But since the machine is out of order, what is observer is that the load that is being processed is 2 iterations ahead of the store that is being processed, so by the time the actual store needs to be done, the value that would be stored has already been calculated. Initially there is a latency of 2 iteration number of cycles, but then since loads that are being processed are ahead of stores that are being processed by 2 iterations, we see that 2 instructions start to commit in 1 cycle and the percentage of 2 commit is more than 50. Even for a small iteration count of 16 the CPI is found to be just more than 1. It can be concluded that due to out of order processing stride based data prefetching had no impact. So, what we observed was that with the stride-based data prefetcher not much gain was found. But when the ROB size was reduced to 8 from 64, so that such a "load store pipeline" cannot be formed then we see a gain of CPI from 1.6 to 1.4.

Following is the distribution of the work among the group members.

**Ankit (26.0%)**: RS module, RS controller, variable priority selector for the FUs, stride prefetcher, D-cache controller, scripting, integration, visual debugger, debugging and synthesis.

**Daya (26.0%)**: ROB, BTB, 2 bit branch predictor, scripting, RAS, 2 priority selector, I prefetcher, load queue, IF & ID stage modifications, debugging and integration.

**Gaurav (22.0%)**: D-cache controller, Instruction prefetcher, top level memory controller, Execution units, CDB arbiter, Dcache, bottom-up synthesis and debugging.

**Kuldeep (26.0%)**: ROB, PRF, RAT, RRAT, free list, saved free list, LSQ, LSQ controller, branch mis-prediction handling, stall handling and debugging.

4. **ATTEMPTED FEATURES**

In the code we have submitted we tried implementing the preID instruction decode for unconditional branches. This didn’t really work for 2 testcases, so we have to turn it off in the final version of the core. One possible explanation for failure of this could be that there is some extra instruction executed because of some problems in valid instructions coming out of IF stage.

5. **CONCLUSIONS**

In this paper, we have presented design and analysis of 2-way out-of-order superscalar processor. Our design synthesized at clock period of 6.7ns and it was able to successfully run all the benchmarks provided to us. We couldn’t achieve a good CPI number in the objsort benchmark because we didn’t implement the set associative cache. In the later part of the project we had to spend a lot of time on DCache controller debugging and that hampered the progress on 2-way data cache. We think that given the time constraint we have been able to come up with a very good working processor implementation.

---

![Hit Rate in Data Cache](image.png)

**Figure 6**: This figure shows the effect of number of instruction prefetching on the performance of various testcases.

3. **GROUP DYNAMICS**