P6 4-Way Out of Order Processor

Group 12

Jay Hendricks
Bo Zhang
Fan Zhang
Yilei Xu
Zhen Xu

Heading (Page Number)

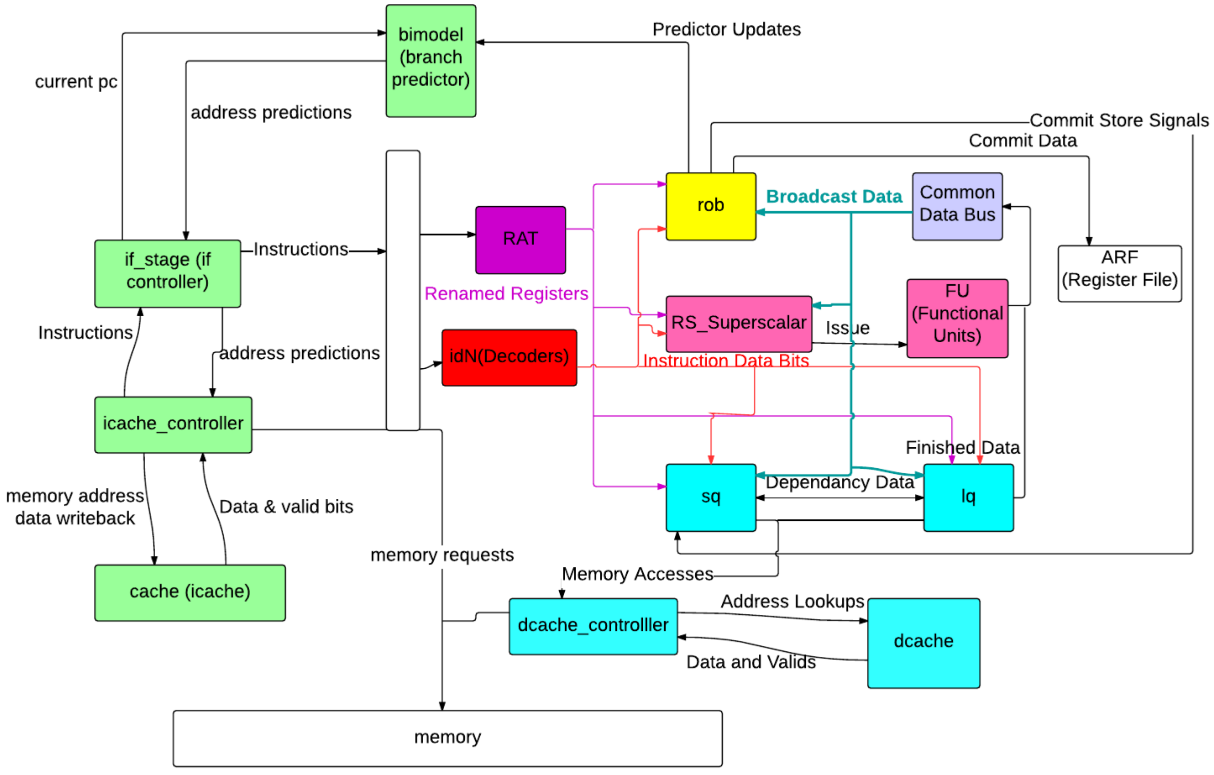
Processor Specification

For our processor, we chose the P6 scheme. This scheme allowed out of order execution without the need for extra complicated modules. In our scheme we have 4 main pipeline sections: and IF stage, and ID stage and EX stage and a writeback stage.

The main advanced feature for our processor was arbitrary way superscaling. For the remainder of the document, any references to “N” or “super N” is the number way superscaling.

In our processor there were three main types of modules: fetch modules, core modules and memory modules. Fetch modules were mainly modules dealing with fetching and prediction, core modules dealt with out of order processing of non-memory instructions and memory modules dealt with memory instructions.

Below is a simple block diagram of the processor, to help with the explanation of the modules:



Out of Order Processor Diagram

The following three sections describe the modules in each category, as well as explaining our pipeline stages. They are meant to give an overview of the processor, as well as explain how it works.

IF

IF controller (if_stage_super.v)

The IF stage controller is a simple pass through of other signals from the IF stage. The module's main purpose is to fetch the correct instructions to feed into an IF/ID register to be decoded and dispatched by the ID stage. It does this by sending predicted instructions addresses from the branch predictor to the cache controller, determining which instructions are valid, then using other signals to determine which to actually deem valid.

This number of instructions sent from the IF stage is limited by the number of available slots in the RS, ROB and LSQ. It takes the smallest number from these available slots, and ideally sends this many instructions.

Chosen instructions from the IF stage are sent to and IF/ID register to be stored. This is the end of the IF stage.

The IF stage was mainly worked on and tested by Jay.

Instruction Cache Controller (icache_controller.v)

The instruction cache controller is a simple adaptation of the basic controller given as a sample. The controller takes in the requested addresses from the IF stage, and sends them to the cache. It then detects the first cache miss in program order, and sends a request for this address to memory. This instruction cache in this way is blocking.

The instruction cache is non-blocking in respect to reads though. The instruction cache controller can read N instructions at a time, up to the first cache missed instruction.

This cache controller was first written by Zhen and Yilei, with Jay helping in the later stages to test.

Branch Predictor (local_hist_2b.v)

The branch predictor is one of a few options. In all, we created several types of branch predictors including bimodal, local history, global, gshare and tournament. Of all these predictors, through testing we found the 1 bit bimodal to be best, considering the IF stage is on the critical path, and the bimodal predictor takes the least amount of time.

In this module, the BTB size and the prediction table size are both parameterizable. In the end, after testing the size of these was found to be best with a BTB of around size 16 and a table size of around 16. Our BTB is fully associative.

Our branch prediction is very conservative, in that the branch prediction updates do not happen until the branches are committed. We felt that committing any earlier may cause negative effects on the branch predictor, because these updates could be non committed instructions.

The branch predictors were written and tested by Jay.

Core

Rename Table (rat.v)

The ID stage begins with the rename table. Our RAT was a simple version of the RAT discussed in class with dependency detection and forwarding. The RAT also serves the purpose of detecting if data or a tag is being sent to the RS, based on the decoder outputs.
The RAT has N outputs for each of the renamed instructions.

The RAT was written and tested by Fan and Bo.

Decoder (id.v)

The decoder was the decoder from the given project 4 skeleton, instantiated N times for N way superscalar instructions. Certain internal signals and outputs were added to facilitate our different modules, but they do not change the overall function and method of the decoders.

The decoders were written, modified and tested by Fan and Bo.

Reorder Buffer (rob.v and rob_single.v)

The reorder buffer was the same as the reorder buffer used in class. There are 32 ROB entries. This number was chosen to keep the ROB as large as possible while keeping it off the critical path.

The way data is loaded into the ROB is with dedicated ROB registers. There are a set of data registers for each ROB that get set for different instructions. The input registers are loaded into the ROB entries when a load in bit is set, and are read and cleared when a load out bit is set. This method for storing data is used within the reservation stations, load store queue and reorder buffer.

We optimized the individual ROB entries to have as much overlap and space saving as possible. In a single entry we currently store the PC of the instruction, and its destination architecture register. With this though, we store very little. We store a few bits for the type of instruction, such as if it is a branch or a store or a load, and a couple of state bits.

The ROB also manages branch prediction updates. The ROB uses internal logic for signaling mispredictions, BTB updates and predictor updates. The ROB has N read ports for incoming instructions, N write ports for outgoing instructions and N write ports for signaling branch prediction updates.

The ROB was written and tested by Yilei and Zhen.

Reservation Stations (rs.v and rs_single.v)

The reservation stations were similar to the ROB. They were the same as the inclass example. There are registers for each reservations station that are read from based on signals into each reservation station, like explained above. The number of RS entries is 16. We chose this in order to keep the reservation stations large enough, while not placing them on the critical path.

Instructions are issued from the reservation station based on the number of multiplies and loads that are finishing this cycle. This logic and reasoning will be explained in the functional unit controller section.

The reservations stations were written and tested by the entire team.

Functional Unit Controller (FU.v)

The functional unit controller was created for simplicity and ease of use. The controller contains N functional units for each type of functional unit (multipliers, ALUs, branch calculators). The multipliers are 8 stage pipelined multipliers. Because of the number of functional units, optimally N single cycle instructions can be quickly issued and completed with no trouble.

The issues here arise when multiplies and loads are issued and then need to complete cycles later. When a multiply or load is finishing this cycle, a signal is sent to the reservation stations to issue less instructions, while instructions are being completed and sent to the CDB’s from the multipliers. This allows instructions to be easily stalled without new logic. The goal here is that as many one cycle instructions issue as possible, only being stalled by multiplies and loads that need to use the CDB.

The functional unit controller was written and tested by Bo and Fan.

Memory Modules

Load and Store Queue (lq.v, lq_single.v, sq.v, sq_single.v, LSQcompartor.v)

For our project, we designed a separated load and store queue for memory based functions, that works parallel to the RS. Because of this, the LSQ works like the LSQ explained in class, but does not take up reservation stations. The LSQ feeds directly into the CDB's and overrides the functional units, essentially performing the dispatching, issuing and execution of the pipeline.

The the load queue and the store queue were both size 8. This was chosen so the LSQ was not on the critical path, while being large enough to execute as many instructions as possible.

As a note, in order for our processor to work, the LSQ size must be larger than the superscalar way. As well, the LSQ size must be a multiple of 2, just like many of our other parameters.

The load queue was written and tested by Bo, and the store queue was written and tested by Fan.

Data Cache Controller

The data cache controller is the EX stage of the LSQ, that performs the loads and stores needed by the LSQ.

The most interesting part about the data cache controller is that it assists in making the data cache non-blocking. The controller takes in all stores and loads and places them into register. In subsequent cycles, the stores and loads are popped from these registers and sent to memory, while writing data to the caches either when the data returns from loads or when the data is written for stores. When these registers are full, stores and loads cannot issue. This creates a cache that can continuously write and read memory requests (non-blocking) without a traditional non-blocking cache.

The data cache controller was written and tested by Zhen and Yilei.

Other modules with no severe special features: CDB and arf (arf.v).

Completion of Base Features

For the project, our group was tasked to create a process with certain features. Below are listed these features and how they were accomplished.

I and D Caches

In our processor we have an I and D cache, with a separate controller for each.
The I cache is blocking, with no prefetching, but allows for N instructions to be read from the cache at a time.

The D cache is non-blocking using the queue system explained above.

Both caches are the same size (32 lines) and are both direct mapped with 64 bit lines, and write through.

Multiple functional units with varying latencies

In our processor we have N ALUs, N branch calculators and N multipliers. The multipliers and the ALUs have varying latency (1 cycle vs 8 cycles).

An out of order implementation

Out of order processing is achieved using the P6 scheme. In order committing is achieved using a reorder buffer.

Branch Prediction with Address Prediction

Our processor implements a branch predictor that fetches instructions using predictions. These predictions are updated on commit by the reorder buffer.

What is Working

All of the processor works for all test cases and all superscalar way 2 through 7. For the processor to work with 8 way super scaling, the LSQ size must be increased to 16.

List of Working Components (Advanced Features)

Below is a list of working advanced features in our processor.

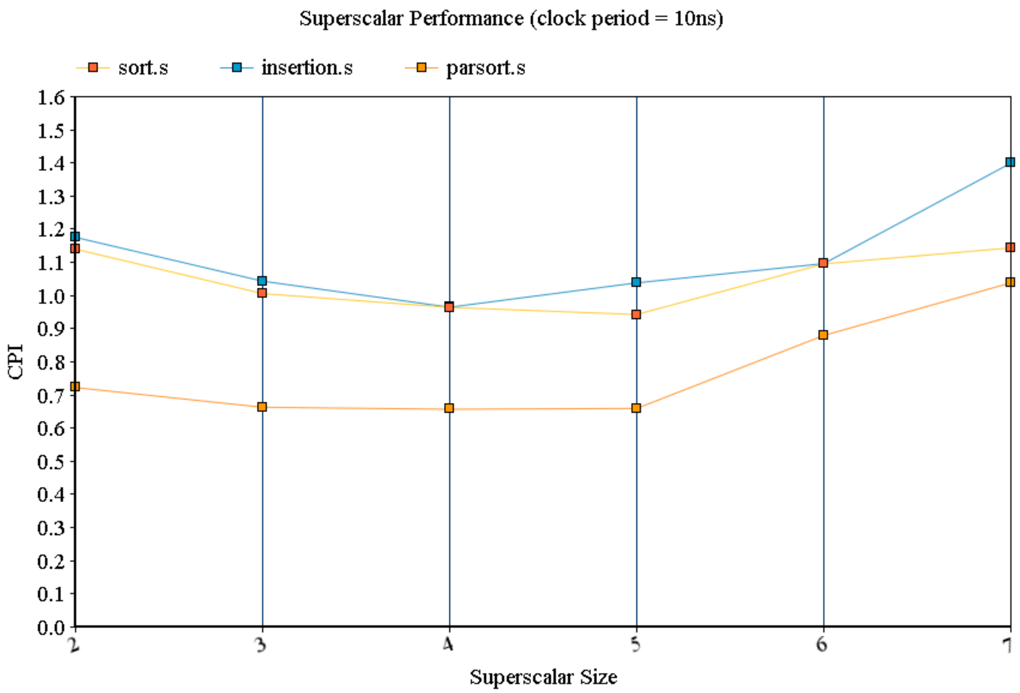
- Arbitrary Way Superscaling
- Data forwarding (LSQ)
- Better branch prediction (branch predictors)
- Out of order memory (LSQ)
- Nonblocking data caches (data cache controller)

In the following section, we will give a simple explanation of the implementation of each of these features, as well as our analysis.

Arbitrary Way Superscaling

The way to accomplish arbitrary way superscaling is to implement modules with parameters or macros. For example in our processor, we have N sets of functional units and N read ports for caches, created with a macro `SUPER_SIZE.

With parameterized modules, we could test the processor by several selected test cases to decide which set of parameters gives the best performance. Below is a CPI comparison when setting clock period equal to 10ns. This is not a 100% fair comparison since for N=2 and N=7 the clock period should be different. (Here we did not consider N=8, because it only works when LSQ size is 16, which would lead to long clock period.)



As we could see, from N=2 to N=3, the CPI drop is large since it allows more instructions to be executed in one cycle. From N=3 to N=4 is less because in this case dependence limits the parallelism. For any N that is larger than 5, the CPI may even increase due to frequently squashing. So the best choice should among N =2, 3, 4, or 5.

After we figured out the exact clock period, we made the following table. After comparing the product of clock period and CPI, we found that the 3 or 4 way superscalar are almost parallel, which one of them performs better really depends on which test cases we choose. For N=3, the process has a shorter clock period. On the other hand, when N=4, it has a smaller CPI.For the consideration of CPI, finally we chose N=4.

N way superscalar	Sort.s CPI	insertion.s CPI	parsort.s CPI	Clock period(ns)
2	1.140104	1.175585	0.722492	9.2
3	**1.005930**	1.043478	**0.663065**	10
4	0.961453	**0.964883**	0.657122	10.8
5	0.843588	0.846154	0.574374	12.5

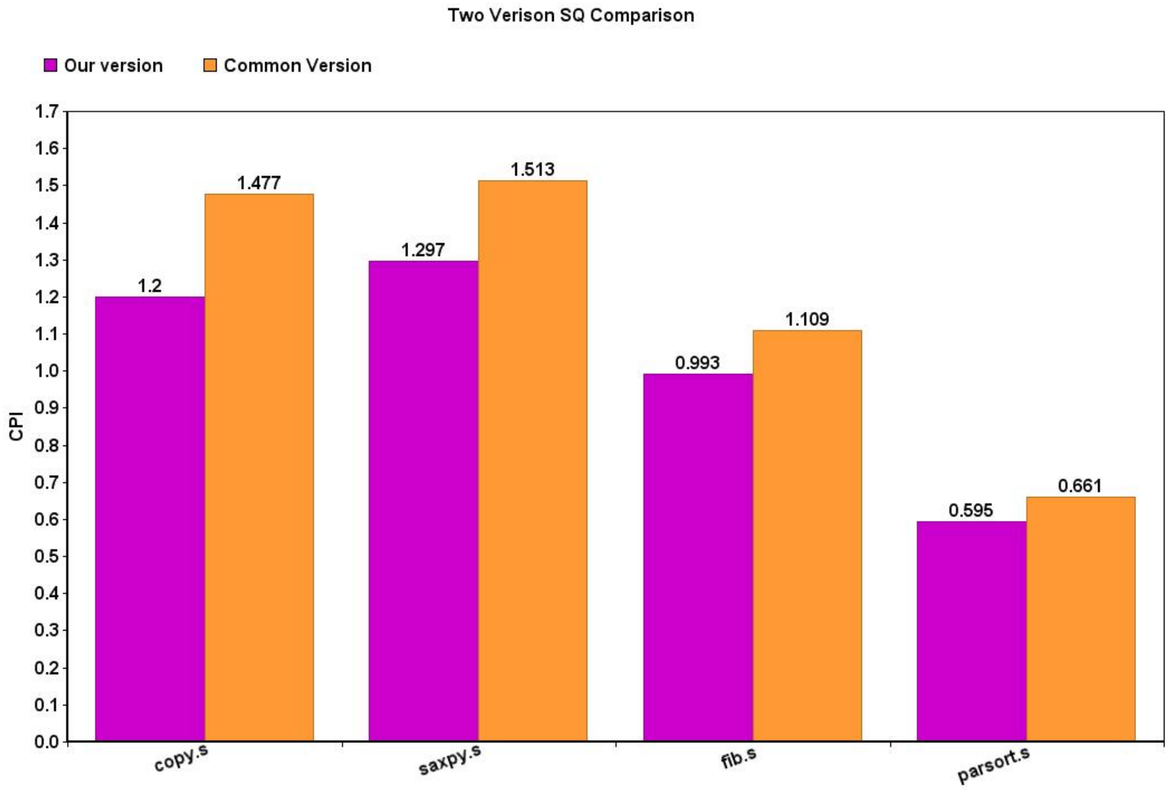
Data forwarding (LSQ)

Data forwarding is achieved using a separate load store queue with comparators. Using the comparators, data addresses can be checked and forwarded from the store queue to the load queue.

Our load store queue is aggressive. Loads will issue before they know for sure whether they are dependent on previous stores. In the case that the loaded data is wrong, the data is overwritten in the load queue

Our store queue has two improved features. One is when ROB commits store, store queue doesn't remove that entry. That entry is only be removed when store queue is full and a new store instruction is dispatched. The other is when branch is mispredicted, store queue only squashes the uncommitted entries and still keeps the committed entries. In this way, the store queue can keep as much data as possible for load queue to forward.

Below is the comparison diagram for two versions of store queue with 4 way super scaling. Our version is the store queue with the above two features, and the common version of store queue always removes a committed entry and squashes the whole store queue when a mispredict occurs.



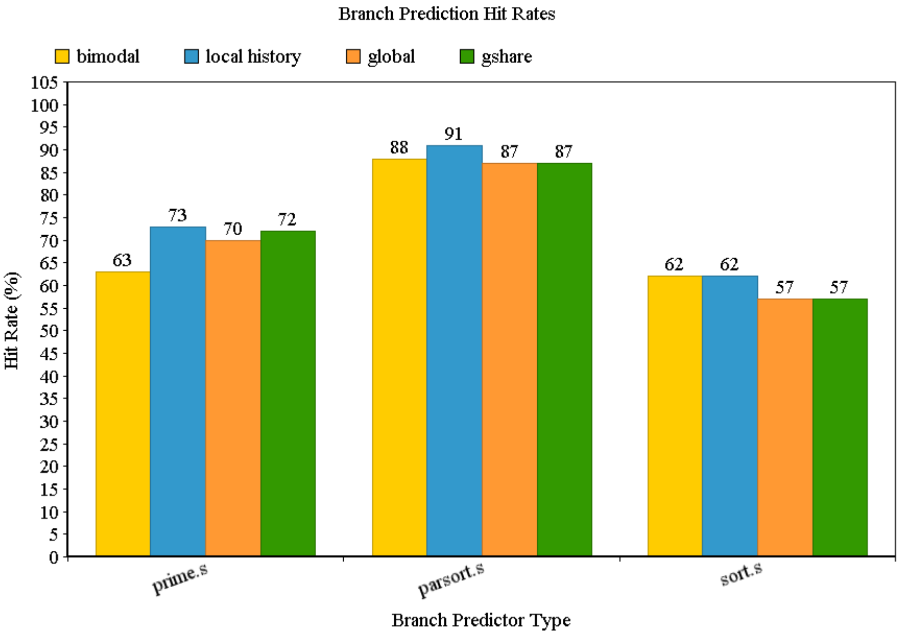
As it can be seen, our improved version using the added features contributes a lower CPI than the common version. This is because our improved version of the store queue can have more data to forward, thus the load queue can get its data from store queue rather than from memory in most cases, which decreases CPI.

Better branch prediction (branch predictors)

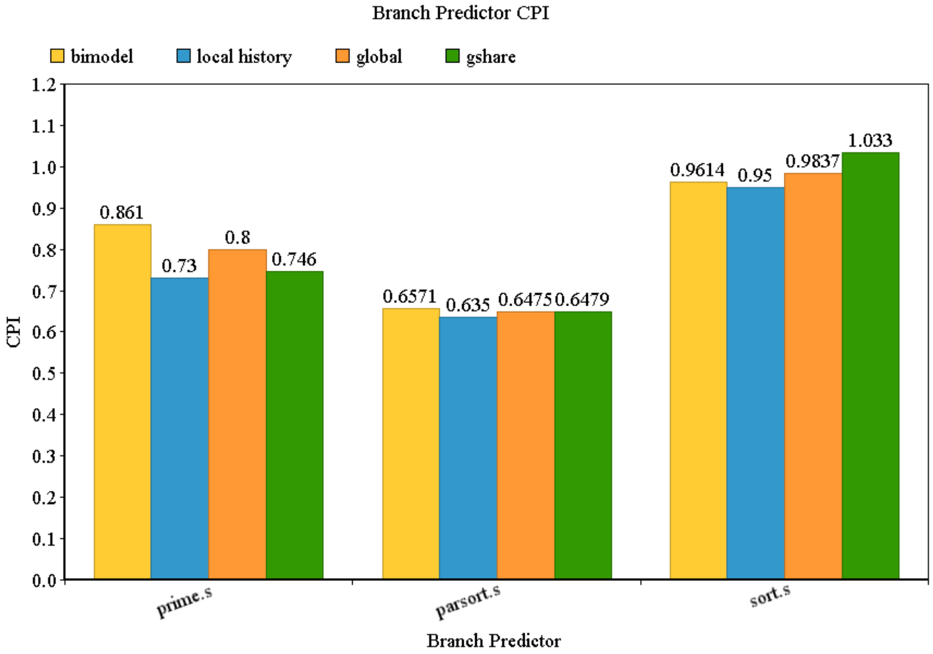
A major feature of our processor was better branch prediction. In our processor, we attempted to implement 4 different branch predictors: a global predictor, a gshare predictor, a local history predictor and a bimodal predictor.

The local history predictor and bimodal predictor both had 16 lines, with the local history having 4 bits for branch history. The global and gshare predictor had a 4 bit registers.

In order to determine which of these predictors was best, we ran the predictors on a few sample test cases and recorded their CPI and prediction hit rate. The results are shown below.



Prediction Rates for all predictors. Hit rates are all similar, with local barely winning in most cases.



CPI numbers for branch predictors. Again, quite similar with local history barely being best in some cases.

From the graphs, it can be seen that in the cases listed, local history is always the best by a small margin and that a higher prediction rate leads to lower CPI. Local history having the lowest CPI was most likely due to a few reasons. The odd patterns of some tests most likely made the local history predictor do well. The 1 bit nature of the bimodal predictor probably caused some issues with its prediction. The little aliasing present in the cases probably made the local history predictor perform better than the global predictors.

What is also interesting about the graphs is the small difference between the predictor values in CPI. This is most likely because the predictions did not cause much issues. Either not a lot of instructions were fetched before a mispredict, or the rate difference was so small it wasn't an issue.

From these results, we chose the Bimodal predictor in our processor. This was because the predictor was on the critical path, and we wanted to keep the clock period as short as possible. Because the CPI difference wasn't large between the predictors, we believe our processor would do best with a fast, slightly worse predictor than a better, slower predictor.

Out of order memory (LSQ)

Out of order memory accesses are achieved with the load store queue. It is a base feature of a load store queue, which has been explained above.

Nonblocking data caches (data cache controller)

The idea is that even if there is a cache miss, the processor can still keep sending requests. No stall is needed to wait for the missing data to come back.

The data cache controller accepts all the load requests and store requests from the LSQ. For stores, the controller puts all the stores in a store buffer. For loads, the controller will first send the load requests to the data cache, and put those load misses into a load buffer.

The store requests take higher priority than the load buffer since the store requests accepted are all guaranteed to be committed. To be specific, the data cache controller will handle all the requests in the store buffer if the store buffer is not empty. Then it will handle the load buffer if there are load requests. The controller will send all the load requests all at once, in consecutive cycles.

The job of the data cache controller is to keep checking the load buffer and store buffer. Using these two buffers, we ensure that all the memory requests are sent in a correct order. More importantly, on every cycle, one new load request or store request will be issued to the memory. Unfortunately, we were unable to see the improvement of implementing a non-blocking data cache because we can not turn the feature off once we implement it. However, the implementation is definitely beneficial, in terms of performance.

Unimplemented, but worked on features

The GUI debugger is working for some processor settings, but not all. The debugger also doesn't give all the information needed. This could be improved. Because of the non-completeness of the debugger, it was not included in the processor files.

Prefetching was in the process of implementation, but is currently not working. With more time this could be implemented, but our group could not implement it in time. The prefetching files were included in pf_ctrl.v.

Team Contributions

Many of the team contributions are listed above with the individual modules, but contained here is a list of what each member did individually. As well, below is a list of who worked on what module. When multiple people are listed as working on a module, it is assumed the work was split equally in that module.

Module Table

Module	**Contributors**
Fetch Stage	Jay
Instruction Cache Controller	Zhen and Yilei
Branch Predictors	Jay
Rename Table	Bo and Fan
Decoders	Bo and Fan
Reorder Buffer	Zhen and Yilei
Reservations Stations	All
Load Queue	Bo
Store Queue	Fan
Data Cache Controller	Zhen and Yilei
Function Units	Bo and Fan
Architected Register File	Bo and Fan

Individual Table

Team Member	Module Name					
Jay	branch predictors	fetch stage	reservation stations			
Bo	Functional units	load queue	ARF	decoders	reservation stations	RAT
Fan	functional units	store queue	ARF	decoders	reservation stations	RAT
Yilei	Data cache controller	instruction cache controller	RoB	reservation stations		
Zhen	Data cache controller	instruction cache controller	RoB	reservation stations		

All of the high level design and formulation was done as a larger group and then realized through smaller groups.

For percentage completion for the project, we chose to set us all at doing around 20% of the work. All people provided equal contribution to the project, in modules and in high level design and testing. We all feel that giving more credit to one person or another would not be right, and that we all did an equal share of the work.