Speculative Out of Order Processor

Varun Govind and Samuel Cheung

May 17, 2020

Contents

1 Introduction

In this project, the goal was to create a speculative and pipelined (i.e. somewhat modern) processor capable of fast instruction execution with high throughput as a result of the efficient use of processor resources. Throughout this project, we implemented many advanced features, including a modified version of Tomasulo's original algorithm. The most notable advanced features that our processor implements are super-scalar/out-of-order speculative instruction execution, single-cycle recovery, pipelined level one caches, dependency-based issue queues for a small unified reservation station, and a multi-ported register file. In this report, we discuss our core and memory architecture in detail and additionally cover some of the specific implementation details regarding the frontend and backend of the pipeline. We also cover the memory subsystem architecture and its respective hierarchy and relationship with the core pipeline.

2 Architecture Overview

Our processor consists of three major components: the core frontend and backend and the memory subsystem. The frontend and the backend the instruction execution, while the memory subsystem handles all the movement of data. The figure below (Figure 13) shows a high-level architectural overview of our processor. The memory subsystem is similar in that there are split level one caches for instructions and data, an arbiter, a level two cache, and a burst adapter.

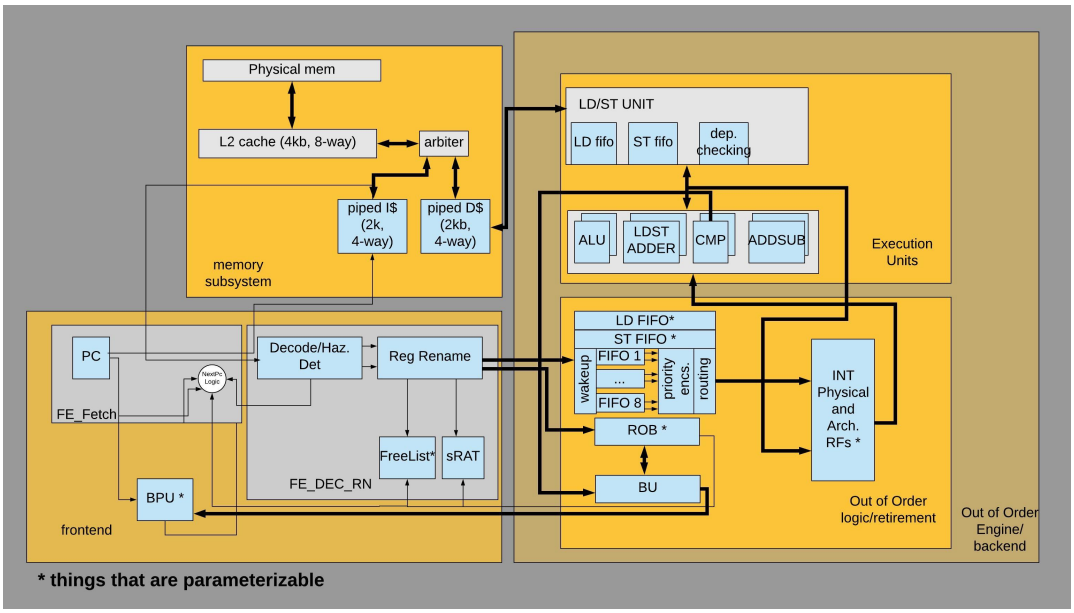


Figure 1: Processor High-level Overview

2.1 Core Architecture

For our core architecture, the main components are the frontend pipeline and the backend pipeline. The frontend pipeline handles several vital functions, including instruction fetching, dependency checking and handling, explicit register renaming, dispatching, and additionally, the retirement of instructions. The backend of the pipeline handles instruction issue, data fetch from the register file, instruction ordering through the Reorder Buffer (ROB), and finally write-back.

2.2 Memory Architecture

Our memory subsystem has several levels in the hierarchy, as shown in Figure 2. On the level closest to the core, we first have split, pipelined level 1 (L1) caches. We split the L1 caches into instruction and data caches, respectively. The instruction cache (I-cache) only fetches instructions, so there is no functionality needed to write and to write-back. Hence, its functionality is more simplified than the L1 Data Cache (D-cache).

In contrast, the data cache handles both reads and writes and more complex functionality with write-backs and memory loads. Both L1 caches are 4-way 16-entries (approximately 2KB) with a pseudo-least recently used (PLRU) replacement policy. An arbiter is also required to arbitrate between requests to the L2 cache from both L1 caches. The arbiter prioritizes an I-cache request if there is contention between the I-cache and D-cache. The arbiter then connects to an L2 Cache, which is 8-way 16-entries (4KB). The L2 cache has a 3-cycle hit response while misses vary according to the operation done (this is discussed in detail later). One thing to note is that the cacheline sizes of all caches used have a size of 32 bytes (or 256-bits). Finally, the last part of the memory subsystem consists of a cacheline adaptor to interface with a burst port memory. The adaptor allows us to reduce the number of IO pins and to optimize timing further.

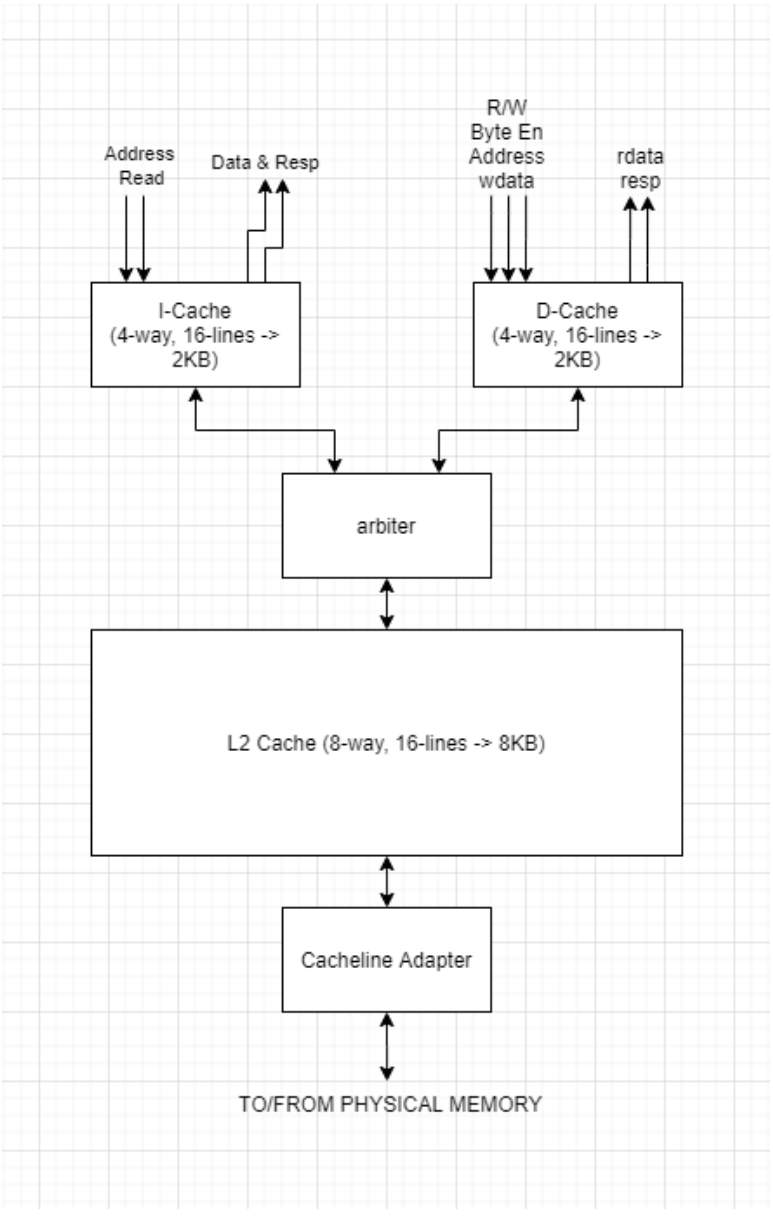


Figure 2: Memory Subsystem Hierarchy

3 Detailed Core Architecture

3.1 Frontend

The frontend has two main responsibilities: the first is to fetch two instruction words, and the second is to rename the instructions according to older-previous instructions to preserve dependencies and to avoid data hazards. The first is dubbed the fetch pipe, and the latter is the rename pipe.

3.1.1 Fetch Pipe

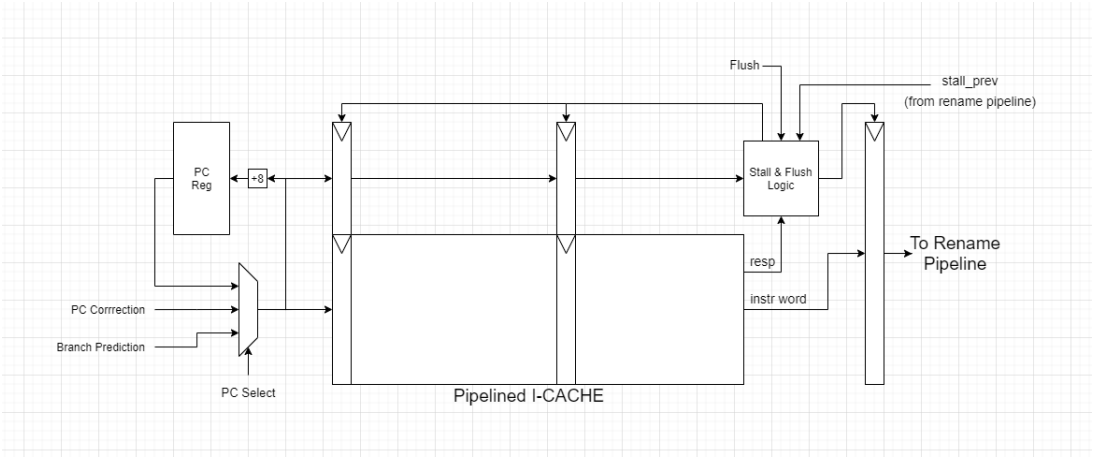


Figure 3: Fetch Pipeline

The fetch pipeline mainly consists of the PC register, routing to feed the PC into the I-cache to fetch the correct instructions, metadata registers to pipeline the fetch requests, and stalling/flushing logic (as shown in Figure 3). The way instruction stalling works is that if there is a valid memory access being performed (i.e., the read signal was sent to the cache), this signal would be set in the metadata registers. If the signal is high in the second stage and there is no response, this would set the stall condition high. The stall condition would prevent loading to the PC and would additionally hold the previous registers while still allowing the next stage to proceed like normal.

The flushing hardware is relatively simple. In addition to the valid memory access signal, there would be *instruction valid* signals in the metadata registers. The *instruction valid* signal would be propagated throughout the frontend pipeline, all of which are used in conditions like dependency checks and in setting write signals for other modules like the rename table. The flush signal sets the *instruction valid* signals to zero, and as a result, nothing is executed in the pipeline. One thing to note is that in the fetch pipe, a stall can still occur even if there is a memory response and a valid request in the second stage. This other stall can come from the rename pipeline. As a result, an additional stall input is needed for the I-cache, which tells it to hold its pipeline registers.

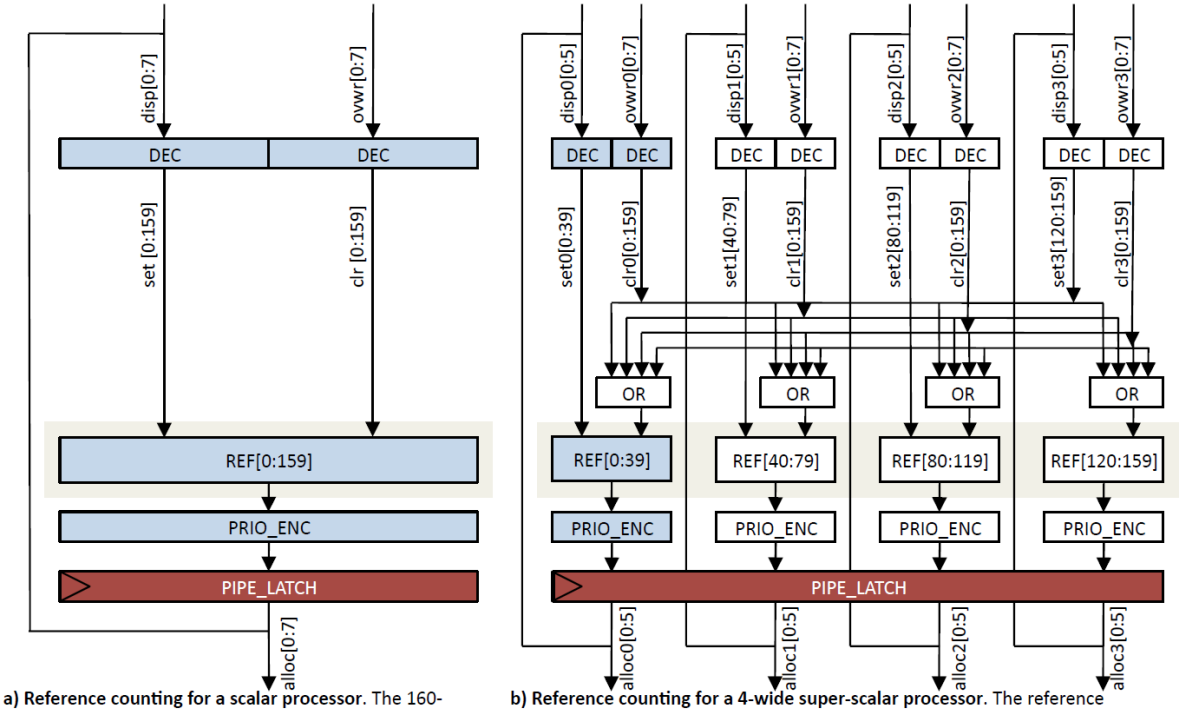
3.1.2 Rename Pipe

Once we have the two instruction words from the fetch pipe, the decoding begins. The decode stage simultaneously detects dependencies if it is a valid instruction and sets relevant signals to dequeue from the free list and write to the register alias table (RAT, also called the rename table). The free list register structure and the RAT structure are complicated structures, each with recovery mechanisms so that when an exception occurs, the processor can easily recover the architectural state by merely reverting the register mappings. Since we do explicit renaming and have a novel checkpointing mechanism, it is easy to recover the state within a single cycle. After the renaming finishes, the next step is to decide where we need to "steer" the instruction. In the backend, we implement dependence based issue queues from S. Palacharla et al. 1997 [1]. By keeping track of dependent instructions, we can organize them in lists by inserting them in different queues (FIFOs). This allows us to check the head of the FIFO for the readiness of operands, which reduces the amount of hardware for reservation stations. In the rename pipeline, we must figure out which instructions are dependent on each other, and one way to do this is to use a

dependence map. The dependence map is organized with the rows being logical registers and the entries being the FIFO id. It essentially keeps track of where instructions are according to what logical registers they write to (i.e., their logical destination registers). If any subsequent instructions use that destination register as a source, it will be placed in the same FIFO. The map is then updated with the new FIFO for the new instruction (the destination register is updated, not the sources). Once the FIFO steering has been done, the final step of the rename pipeline is to dispatch to the issue queues and the re-order buffer. If there is no space available in either the re-order buffer or the issue queues, the entire rename pipeline will stall. It will not finish stalling until there is space available for both instructions to queue up.

Free List

In order for the free list to work with superscalar, a queue like structure can be used. However, the amount of overhead scales up non-linearly. In a queue-like structure, there are many cases to consider when retiring instructions and freeing physical registers and simultaneously using other free physical registers to rename existing in-flight instructions in the pipeline. In addition to this, it is hard to pipeline and make it fast. Therefore, we use a method called *Register Reference Counting* [2] to manage used register and free registers efficiently. The method is highly scalable to N-way superscalar processors and is very fast and efficient for large out-of-order cores.



a) Reference counting for a scalar processor. The 160-register reference count vector supports one bit set (from dispatch) and one bit clear (from commit/overwrite) per cycle. Registers are allocated using a priority encoder. The allocated register is the dispatched destination register.

b) Reference counting for a 4-wide super-scalar processor. The reference counting vector is divided into 4 disjoint segments. 4 ¼ width priority encoders allocate 4 registers, 1 from each segment, per cycle. Because the dispatched destination registers are the allocated registers, those decoders are ¼ wide as well. The 4 decoders for the overwritten registers are full width.

Figure 4: Free List Architecture

In the figure above (Figure 4), the *disp* is the dispatched physical register or the physical register that we want to un-free. The *ovwr* is the committed physical register or the register that we want to free (used when we retire the instruction). The *alloc*, towards the bottom (coming out of the pipe-latch), are the free registers that are available to be used. The **REF** is the bit-vector (using SR-latches) that represents our free list. Using decoders and priority encoders, we can set and detect bits that represent free and not-free physical registers, which are then sent to the rename pipe to be used.

Register Alias Table and Copy Free Checkpointing

To support quick recovery and fast renaming for a two-wide superscalar core, we had to think of a good way to recover the processor state in a single cycle. With some research, we found a good paper by K. Aasaraai et al. 2009 [3] that implements a method of checkpointing without any copying of data. The

way it works is that we have an array of dirty flag bits where the row represents the architectural register, and the column represents the checkpoint number, as shown below in Figure 5. This data structure is also called the Dirty Flag Array (DFA).

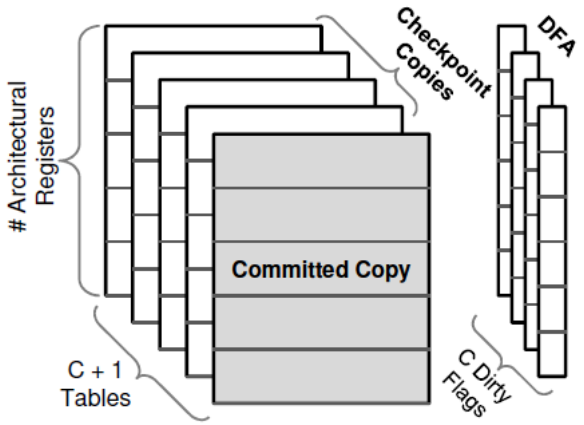


Figure 5: Structure of the register alias table arrays

The way that this data structure works is whenever we modify the RAT for renaming a destination register, we then mark the dirty bit for the respective checkpoint that we are currently located. This checkpoint or speculative state that we are executing instructions at is the head pointer of the DFA. When we want to advance the checkpointed state, we move the tail by the number of checkpoints we want to throw away.

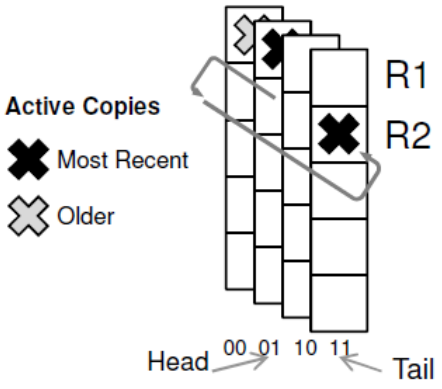


Figure 6: Dirty Flag Array

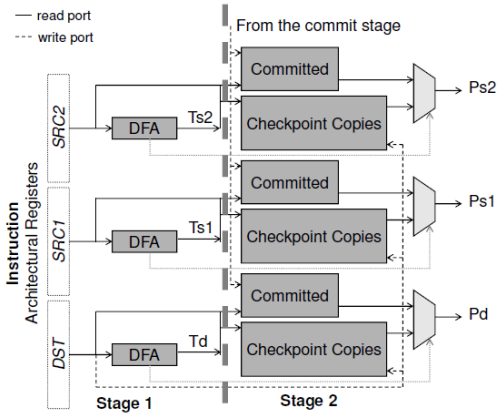


Figure 7: Copy Free Checkpointing Architecture

By marking the bit, we are able to tell which checkpoint has the most recent mapping. Using the head and tail pointer, we can do an age-based search on the DFA to find the most recent mapping. An example is shown in Figure 6. The head is the current checkpointed state, whereas the tail is the oldest. If the entry in the entire DFA is empty, meaning there are no dirty bits marked for a register, we use the value from the committed copy. In Figure 7, we see how this would work for a 1-way scalar processor. To extend this design to a superscalar design, we simply have to add more ports for the committed and checkpointed copies. With this copy-free checkpointing design for the RAT, we were able to maintain a high frequency for our core design and additionally have a quick recovery scheme.

3.2 Backend

Simply put, the backend executes instructions, retires them and signals for recovery on a branch misprediction. It's worth noting that our register renaming scheme uses a rename buffer [4] that's a merged architectural and rename register file (figure 8). Thus each physical register will either be in one of four

states: available (not committed), architectural register, renamed and invalid, and renamed and valid. It's also worth mentioning that the backend ended up using an additional 8KB in BRAM.

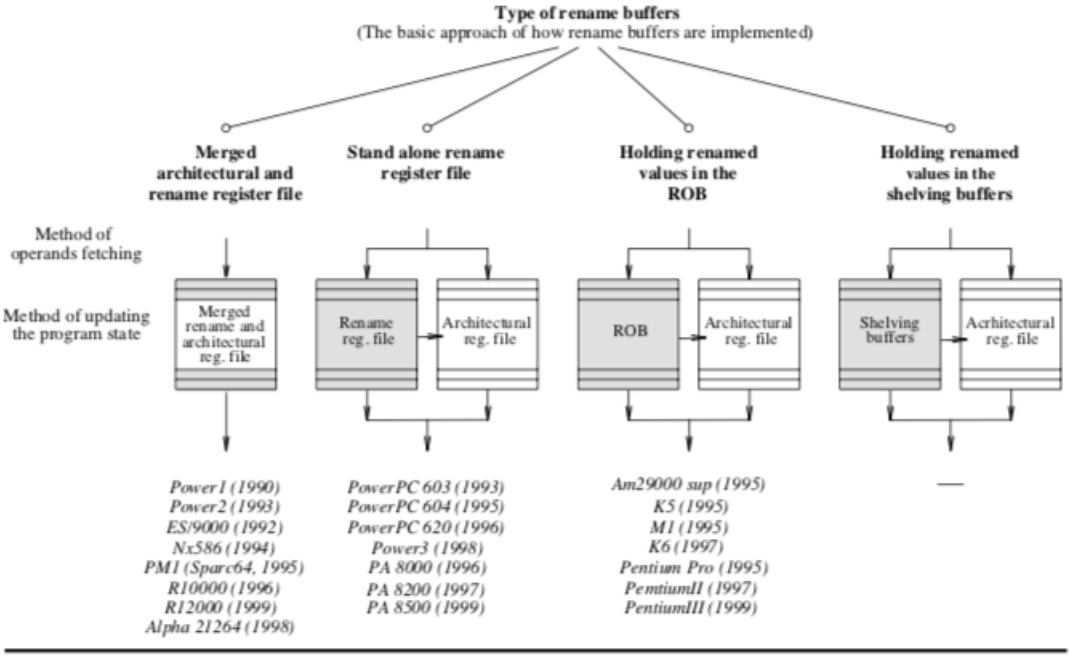


Figure 8: Generic Types of Rename Buffers

3.2.1 Instruction Queues

The Instruction queues are where all instructions (not including nops) first reside upon entering the backend. It consists of 10 Quartus IP FIFOs that accept dispatched instructions from the frontend. Two are for memory (one for load and one for store) and eight for all other instructions. The memory packets are 39 bits while for the other instructions, their FIFOs are 83 bits wide. For the regular packets we store the PC of the instruction (32 bits), the remapped sources and destination (21 bit) of the instruction, two ready flags, a 3 bit instruction type, 20 bit immediate value, funct7, funct3, and a valid flag. This is quite a bit of information to store, however we use BRAM for our FIFOs and as will be explained, don't perform a CAM based tag check. The memory packets will be detailed out below. When queued, ready bits of a bit array that correspond to ready source operands (these are the remapped physical registers) are marked and checked when dequeuing an instruction. We only dequeue from the head of each FIFO, knowing that dependent instructions have been steered to the same FIFO and independent ones to a seperate FIFO. This "complexity-effective" wakeup-select mechanism was taken from [3] and is not pipelined as can be found in [5] or speculative [6]. The intent was to remove the long combinatorial loop that exists within the typical wakeup-select logic, say found in the Alpha 21264 (figure 9) [6]. There are five "channels" that feed instructions of specific types into one of the following execution units: Add/Sub, Branches, Ld/St, U-Type, and ALU.

Add/sub unit takes care of all Reg-Reg and Immediate add/sub instructions as well as the JALR instruction. Branches and Jumps are evaluated and the resulting address is stored in another structure called the Backend Control Unit. One register is used to save the jump address after evaluating an unconditional branch through either the u-type adder (JAL) or the add/sub adder (JALR). The branch control unit will communicate with the frontend to update the BTB immediately when a branch condition evaluates and to jump when a jump is ready to retire from the ROB. Ld/St instructions enter into their own load/store buffer in order. These instructions will be discussed in detail below. U-Type encompasses AUIPC, LUI, and JAL instructions are executed in the same unit. At this point all other instructions will be either an ALU instruction or a nop.

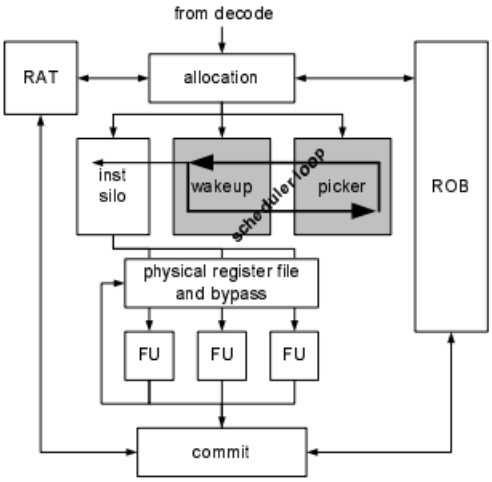


Figure 9: Combination Wakeup Select loop

3.2.2 Reorder Buffer

The ROB is responsible for ordering and committing the instructions as they execute out of order with respect to one another. Each field is 17 bits wide. It contains the 5 bit original architectural register, 7 bit physical register, and 5 flags (jump, branch, checkpointed instruction, flush, and done). Since we do 2 instruction fetch/decode we retire 4 instructions to avoid ROB structural hazards that could lead to stalls. When a branch resolves its ROB entry is updated so that when a control instruction can be retired the correct PC is used.

3.2.3 Ld/St Unit

Once sources are ready and both the address and any data is available a load or a store will enter a load or store buffer. We prioritize loads over stores (and only request from the head of the load or store buffer). Additionally, since our cache is pipelined we must pipeline our access. The load and store buffers each have unique entries. Loads have to track the address, data, rob entry index, funct3, remapped destination register, and three flags (valid, commit, store match). Store entries are similar but don't have the remapped destination register and st match bit. Additionally, memory alignment occurs here as well. I will detail the queuing process first. A major component of the ld/st unit is the CAM based search upon queueing an instruction. Since we assume the memory instruction has its address and is in order (according to their type), we need to ensure that any new load with matching stores to the same address gets the value from the youngest store that's older than the load. Similarly, when a store enters you must update all loads that are younger than you with your data. This process is tricky since we implement the buffer with head and tail pointers. A valid load or store at the head will be selected for memory. Therefore, after we match to all stores we must shift the result by the tail amount and use a priority encoder to extract the correct matching store. For stores it's much simpler. When a new store enters it will update all younger matching loads since at this point in time it is the youngest store in the store buffer. The store match field for loads is also set when addresses match.

For dequeuing, loads only have to wait until they receive their data, however stores must wait until it retires from the ROB. However there are a few subtle points. You cannot dequeue the load until you know that there aren't anymore older stores either in the store buffer, on the way to the store buffer or at the head of the store queue. Additionally, you have to have data which can either come from a store or the memory. It's important to note that the accesses are pipelined since the cache is pipelined. So a stall signal is needed that accounts for both the memory response from L1 and if you matched with a store (store match field). Once either one at the load head is found to be true we release the load and update the load head pointer. For stores we must wait until we retire from the ROB to release the store from the head. It should also be mentioned that upon release from the store queue we mark the corresponding ROB entry as done. The ROB will calculate a retire mask and if it includes a store in the mask the store buffer will mark the commit bit and dequeue stores.

3.2.4 Control Instructions

When control instructions exit the instruction FIFOs, they are calculated, their addressed are saved and their ROB entry is updated. When the ROB retires a control instruction it will signal the frontend to correct if necessary. The entire backend flushes if so (except the register file).

4 Detailed Memory Architecture

4.1 Hierarchy

The memory hierarchy consists of pipelined L1 caches, an arbiter, L2 cache, and a cacheline adapter (as shown in Figure 2). As described previously, the L1 caches are 4-way 16-entry caches with 256-bit (32 byte) cachelines. This means our L1 caches collectively are 4KB. These sizes are not standard of what is typically done in the industry. However, we were limited to 8KB total for our memory subsystem. Typically, the collective L1 size is smaller than the L2 cache. Our design makes the combined size of the L1 caches the same as our L2 cache because it would maximize our BRAM usage and because the way sizes and entries are all even powers of two. Our arbiter prioritizes any I-cache request unless there is an ongoing request with the D-cache in progress. In that case, the I-cache must wait for the request to be satisfied before the I-cache's request is handled. The L2 is a much simpler cache with a state machine as a controller. We keep the cacheline sizes the same throughout the memory subsystem, so this allows us to simplify tag matching. The last part of the memory architecture is the burst adapter. The burst adapter takes an entire cacheline and outputs split sections of the cacheline. In our case, it was 64-bits or 8-byte sections, which amounts to 4 bursts for a 32-byte cacheline. A state machine also controlled the burst adapter. The number of bursts can additionally be modified for larger cacheline sizes; however, because of time constraints, we thought it was simpler to leave the cacheline sizes at 32-bytes.

4.2 L1 Caches

We made a single all-purpose pipelined cache for both the I-cache and the D-cache. This simplified our design process to simply focus on what the main functionality of the D-cache is (because the D-cache functionality also encapsulates the functionality of the I-cache). The pipelined cache uses BRAM M9Ks to avoid using LUTs, ALMs, and MLABs because a lot of on-chip resources would be used. Additionally, M9Ks use less power and are faster than all of the other options provided.
As shown in the figure above (Figure 10), the pipelined cache has two stages: the read stage, and the hit stage. The read stage is where the array access happens, and the data is routed to the next pipeline register. In order to constrain timing to meet our specifications, we had to implement some early hit check and some early hazard logic (not shown). These early hit checks allowed us to reduce the critical path in the pipelined cache such that the slack is more evenly spread among both stages. As mentioned before, one thing to keep in mind is that there are data hazards when pipelining the cache. Suppose the pipeline makes two sequential memory requests to the same address (i.e., cacheline block), and the first older request is at the hit-stage. If the cache loads the new cacheline block (from memory) into the data arrays, on the next cycle, you will miss and perform another load because the cache did not do any hazard checking. Therefore, we reduce the amount of lower-level memory accesses by keeping track of the last memory operation done. We check the last memory operation with the current memory request in the hit stage to resolve any hazards. Once the request is satisfied, we update the hazard logic.

To handle lower-level memory operations like loading and write-back, we use status registers to keep track of the most recent operation done. These status registers act as a state machine that stalls the rest of the cache pipeline when a lower-level memory access is being performed. One problem that became clear when adding the BRAM was that simultaneous accesses when reading and writing to the same block, do not return with the most recently written value (like how the old data arrays work). Due to this, we had to construct additional forwarding logic to correct the data given to the next pipeline register. This is the forwarding logic shown in the bottom left of Figure 10. The forwarding logic keeps track of any written data to the arrays and compares the write index to the read index. If they are equal, this sets the mux signals which rout the corrected data stored in the forward buffers to the second pipeline register.

The L1 cache uses 32-bytes in one cacheline and is a 4-way set associative cache with 16 entries per way. The replacement policy we used was the PseudoLRU (PLRU) policy because it was simple and

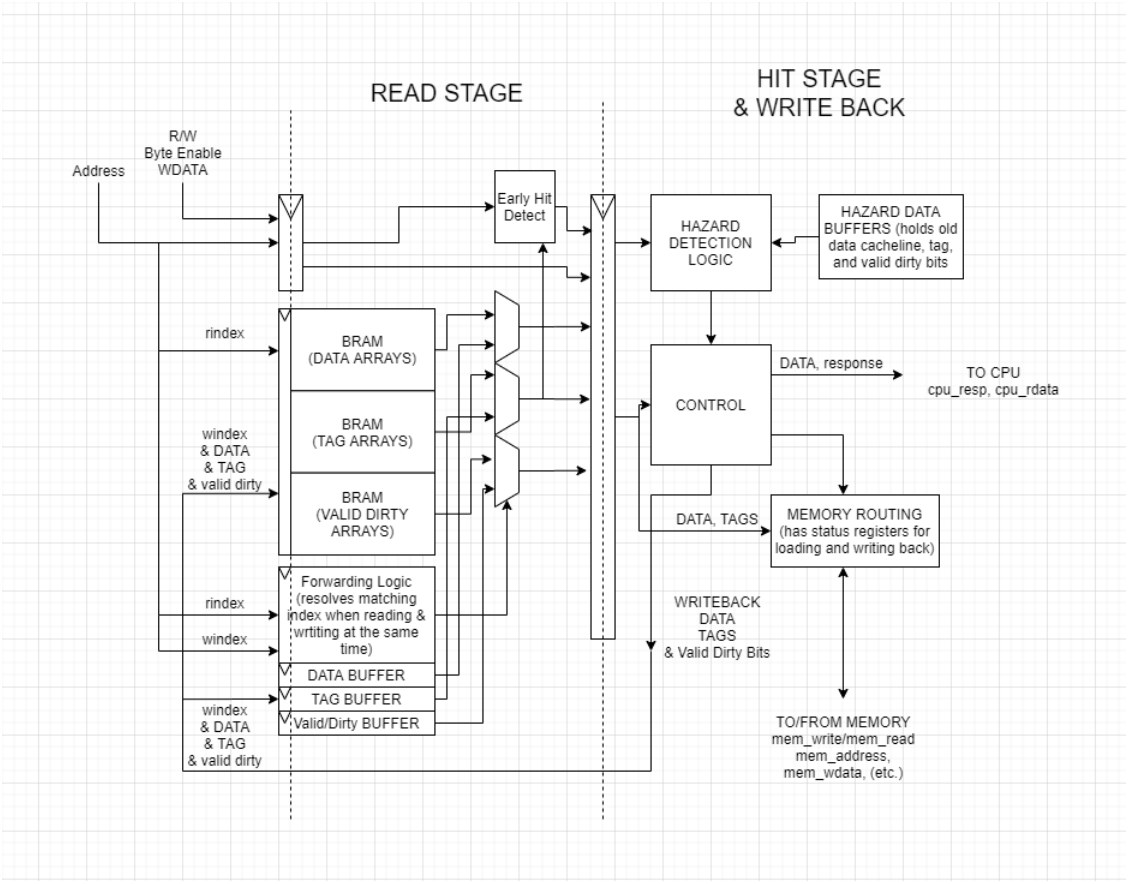


Figure 10: Pipelined Cache

had low latency. However, because it is not entirely accurate, there may be some performance hit due to evicting the wrong cachelines. PseudoLRU uses a tree-like structure to keep track of the age, and because of the simple concept, it is effortless to implement and has a lot less overhead than trueLRU. The trade-off between the two methods is that PseudoLRU has much less overhead and is much faster than what TrueLRU does for large set-associative caches. The total size of one pipelined cache is approximately 2KB (slightly larger with the metadata bits in mind).

4.3 L2 Cache

The L2 cache is very similar to L1 caches, with the only difference being that is is not pipelined and that it uses a state-machine for its control. The L2 cache takes longer to respond because extra registers and states were added to meeting timing requirements. Although they were not required, it allowed us to focus on optimizing other parts of the memory subsystem and the core. The L2 cache (like the L1) checks for hits early and stores the output in a register, when the state machine reaches the hit-check state, it will read this register. There are four main operations that the L2 cache does. They are: Read/Write Hit, Read not-dirty Miss, Read dirty Miss, and Write Miss. The read and write hits are simple, it takes two cycles to read, and one cycle to respond to the hit. The read not-dirty miss does a lower level load memory operation. We also implemented a form of load forwarding so that the L2 will respond at the same time as the lower-level memory response. The read dirty miss is the operation that takes the longest. The cache must first store the dirty line, then load the new line from memory, this amounts to two memory ops that the L2 is waiting for. The write miss is another simple operation; if the line is dirty, the cache can store the dirty line to memory, and while it is doing this, store the cacheline into the data arrays and respond. If the previous cacheline is not dirty, then the cache can store it directly without doing a memory operation.

The L2 also uses PseudoLRU for its replacement policy. We chose to do it this way because it was relatively easy to extend the PseudoLRU module from L1 caches. If we were to implement the TrueLRU

replacement policy for an 8-way set-associative cache, the overhead would be far too great, and there is little benefit in terms of speed and resource cost. Therefore, we thought that it would be best to take the performance hit and do the simplest solution to the replacement policy.

5 Performance

5.1 Memory Subsystem

The pipelined caches each have a hit-response of 2 cycles while the number of cycles for misses vary. The arbiter takes one cycle for the request to propagate while the response is immediate. In addition, the L2 takes a varying amount of cycles based on hits or misses. For example, for a hit, it takes three cycles to respond. However, for a miss and a write, it would be four cycles because our state-machine takes three cycles to check the hit, and while it is writing back to memory, the L2 cache will store the word and respond without waiting for memory in the next cycle. This way, we do not have to load and wait for two memory responses. The worst-case scenario for the L2 is if we read and we need to write-back a dirty cacheline. In our case, we would store the dirty cache-line first and then load in a new cacheline taking two memory ops. A better way that we could have done this is if we loaded the new cacheline first and responded while storing the dirty line back. This way, we would only wait for one memory operation and then respond faster. In hindsight, we could have more performance if we had reduced the number of ways in the L2 and increased the cacheline size for the L2. This way, whenever the I-cache missed, it would not have to load a new cacheline every time.

6 Challenges

6.1 Memory Subsystem

One of the interesting challenges of the memory system was how to verify the correctness of the caches. What we did was modify the existing magic memory and shadow memory to monitor all memory accesses. We then made a testbench to do random memory accesses. However, one important caveat of the shadow memory is that it only checks the correct data when the testbench asserts a read operation. So two simple tests were made, one was a read after write test, and the other was a random read or write test. The read after write (RAW) test would write to a random memory location and then read right afterward to check if the write was done correctly. The other test was a random read or write (RAND) test so that we could cover more of the unusual cases. This worked pretty well after we implemented randomness between either of the tests, meaning we would additionally randomly choose which test to do (either the RAW test or the RAND test). This testing methodology was a good system that uncovered many bugs in the memory system. One interesting thing that we found was that we would find bugs much faster when we swept over a smaller memory space. For example, if we tested over a memory space with an address range of 0x60 to 0x7000, we would find a bug when the simulation time was at 1 million nanoseconds. However, if we swept over an address range of 0x60 to 0x300, we would find the bug at 10 thousand nanoseconds. This is important because all of our caches were implemented using BRAM arrays, and simulation using BRAM arrays takes a significantly larger amount of time to simulate than the regular arrays. So if we stopped the longer simulations early, we would consequently miss some bugs because there were not any errors. Therefore it was important to cover a smaller address space to hit more of the edge cases in the memory subsystem.

6.2 Frontend

There were not any serious difficulties in the frontend because we had a system of unit testing, which guaranteed that certain parts worked when combining the parts. If there were any one-difficult thing, it would be the stalling hardware. While flushing was relatively simple, stalling was more complicated than we initially thought. This is because one thing we wanted to do was while stalling, we only wanted to stall previous stages and not the entire frontend pipeline. There were certain components like the rename table that were also pipelined that needed to be stalled as well. It was a challenge trying to figure out which registers needed to be held stable and which ones were allowed to continue. There was not any specific verification method used to unit-test; it was a combination of both simulation tracing and test benches with random stimuli.

6.3 Backend

6.3.1 Branch prediction

I was able to design and test as a DUT the branch prediction subsystem but did not incorporate into the final product. Therefore it's worth noting this work. I will also detail here the complete control architecture we adopted. Insipired by BOOM [7], we have a 1024 entry direct mapped BTB and a 4096 entry PHT (2 bit counters) for a 13 bit history gshare predictor. It's important to highlight a couple of things from the frontend Branch logic.: managing BTB, how to make superscalar fetch and prediction, "shadow" updates, and resetting on misprediction. I hash a selection of PC bits by doing a series of adds and xors to keep it single cycle. This is used to index into the BTB and perform a tag/offset match. When a branch is resolved, the BTB is immediately updated. Every cycle it is consulted for the next PC and if there is no tag match or the entry is invalid next PC is taken. One issue with Branch prediction is what history to consider in the global history register. We use the OR-reduction BOOM v2 [7] uses for the GHR update. When making a prediction you must use a GHR that is speculatively updated using predictions for known decoded branches. However you always want the GHR to reflect the outcome of all previous branches to the best of its ability, so snapshots are saved in a branch ROB and used to recover from when a branch is retired and found to be incorrectly predicted.

6.4 Other Features

We also considered other additional things detailed below, but they weren't added (however they were designed and tested).

6.4.1 Victim Cache

In managing the victim cache [8] the following cases must be considered: (1) Processor reference to memory misses in both the L1 cache and the victim cache and (2) processor reference to memory misses the direct-mapped cache but hits the victim cache . In the first case the required block is fetched from main memory (or the L2 cache if present) and placed into the L1 cache. Then the replaced block in the main cache is moved to the victim cache. There is no replacement algorithm. With a direct-mapped cache, the line to be replaced is uniquely determined. Finally, the victim cache can be viewed as a FIFO queue or, equivalently, a circular buffer. The item that has been in the victim cache the longest is removed to make room for the incoming line. The replaced line is written back the main memory if it is dirty (has been updated). In the second cases the block in the victim cache is promoted to the direct-mapped cache. Then the replaced block in the main cache is swapped to the victim cache. We also made the Pseudo LRU 8-way.

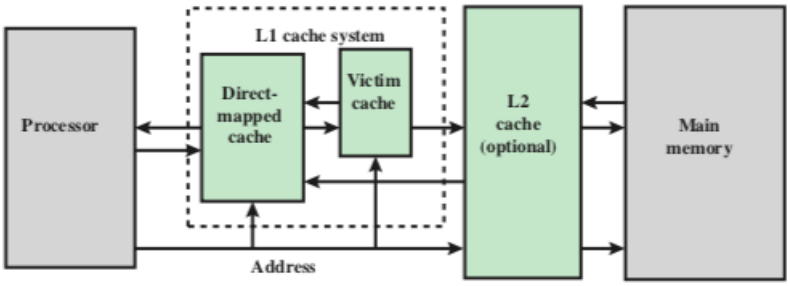


Figure 11: High Level VC

6.4.2 Eviction Address Filter

We want to prevent two things in a small cache 1) cache pollution, i.e., blocks with low reuse evicting blocks with high reuse from the cache, and 2) cache thrashing, i.e., blocks with high reuse evicting each

other from the cache. The implementation and design comes from [9]. It's a fairly simple structure and is described succinctly in the figures below (Figures 11 and 12).

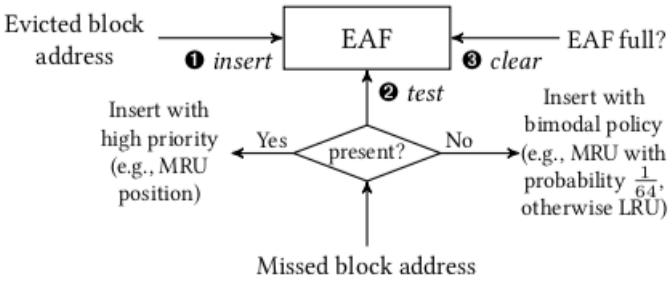


Figure 12: EAF mechanism

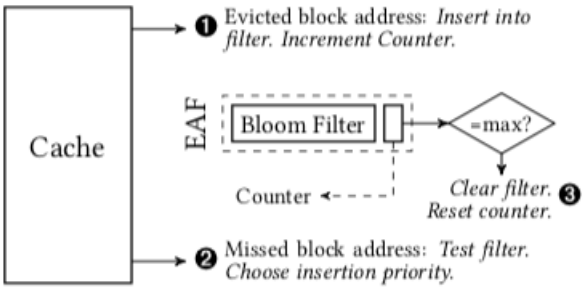


Figure 13: EAF implementation using a Bloom filter and a counter. The figure shows the three events that trigger operations on the EAF

7 Conclusion

7.1 Team Member Contributions

Varun Govind - Implemented the frontend pipeline, the free list, the copy-free-checkpointing rename table, fifo map table, all of the memory subsystem including the pipelined caches, and the multi-ported register file.

Samuel Cheung - Implemented the backend pipeline (designed but unincorporated Victim Cache, Eviction Address filter, BTB, and GSHARE). Personally, I enjoyed the journey of ironing out the nuances. If a week more was given I would've planned on finishing our verification, connected our prexisting advanced features (branch predictor, BTB, victim cache, EAF), and added in some additional features (extensions and cache improvements).

References

[1] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Conference Proceedings. The 24th Annual International Symposium on Computer Architecture*, pages 206–218, 1997.

[2] S. Battle, A. D. Hilton, M. Hempstead, and A. Roth. Flexible register management using reference counting. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, Feb 2012.

[3] K. Aasaraai and A. Moshovos. Towards a viable out-of-order soft core: Copy-free, checkpointed register renaming. In *2009 International Conference on Field Programmable Logic and Applications*, pages 79–85, Aug 2009.

[4] D. Sima. The design space of register renaming techniques. *IEEE Micro*, 20(5):70–83, 2000.

[5] M. D. Brown, J. Stark, and Y. N. Patt. Select-free instruction scheduling logic. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 204–213, 2001.

[6] J. Stark, M. D. Brown, and Y. N. Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pages 57–66, 2000.

[7] C. Celio, P. Chiu, K. Asanović, B. Nikolić, and D. Patterson. Broom: An open-source out-of-order processor with resilient low-voltage operation in 28-nm cmos. *IEEE Micro*, 39(2):52–60, 2019.

[8] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[9] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 355–366, 2012.