Development of a Single-Core Processor with a Verilog System

Using Out-of-Order Instruction Selection and Coherence Guarantees

Jordi Plana
jordi.plana.codina@estudiantat.upc.edu

Alba Soldevilla
alba.soldevilla@estudiantat.upc.edu

PROCESSOR ARCHITECTURE

First Semester 2025-2026

Abstract

This document is intended as a summary of the most common operations in the developed project implemented in Verilog. Each section presents the explanation of a specific instruction type, detailing the most relevant behaviors across the different pipeline stages. Additionally, to improve the understanding of the architecture, two diagrams are provided to illustrate the pipeline stages and their interactions. Figure 1 it is meant to display a high-level view of the entire project. While Figure 2 focuses on architectural components that could not be clearly represented in the first diagram, including the bypass paths, the remove-after-call module, and the generation and propagation of the kill and kill-canceled buses. It also provides a basic representation of how Instruction Window entries are updated during wake-up operations.

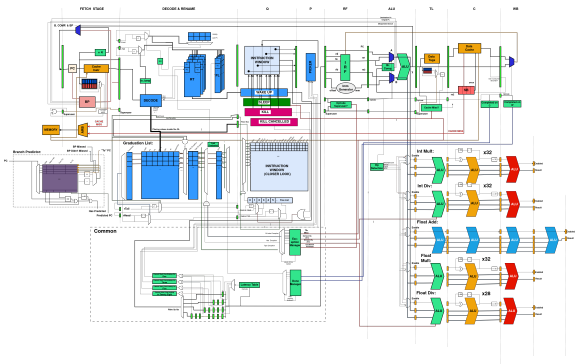

Figure 1: High-level overview of the processor pipeline and stage interactions

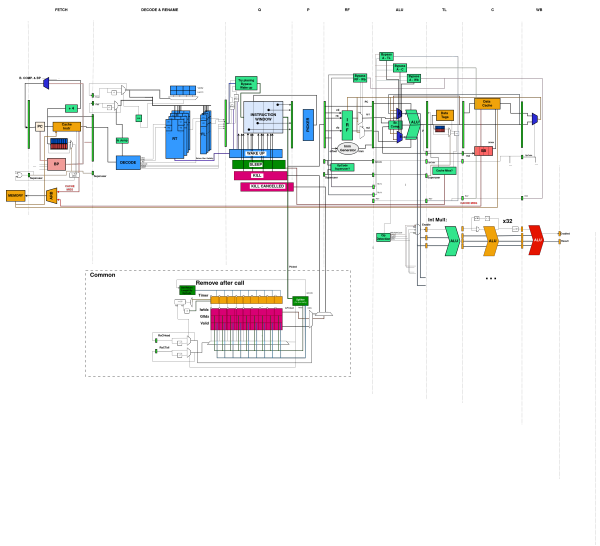

Figure 2: Detailed view of bypass paths, remove-after-call logic, and kill signal propagation

1 Load

Once a given element is picked from the instruction window, if it is a load operation, multiple things can occur depending on the size and the queried address. The common computation that has to be done in the Alu stage for both store and load is the computation of the new address where we are dealing the operation. In both cases we will be able to get a more updated version of the register that adds to the immediate part by bypassing by comparing the used virtual register against the destination virtual register of future stages, to know if some stage ahead of us has the value we need.

1.1 Basic case

The basic case of a load in the cache stage is when the requested lines are present in the cache and none of them have the "present in SB" bit active. This is detected in the "TL" stage. First, it is checked whether the access is in superuser mode; if it is not, the TLB is used to generate the physical address, which is then compared against the cache entries. If this basic case occurs, the instruction continues from the "TL" stage to the "C" stage, where the actual cache lines are stored. There, the load simply acquires the required bytes from at most two lines and continues through the pipeline normally. Another basic case occurs when the elements are still present in the "TL" stage, and it advances to the "C", where it detects a match on an entry in the store buffer. In this case, the newest entry that contains those values is returned. The address and length do not need to match exactly; for example, a byte inside a word that is present in the store buffer can be returned. This is done by validating, for each line, whether it fully fits within any of the elements present. If it does, only the corresponding bits are returned.

1.2 Line Collision

This case does not always occur. It is possible that part of the queried element has been updated in the store buffer, but not all the bytes are present in the element that updates that line. In this situation, the load cannot capture any byte directly from the cache line, because it may contain stale information. In this case, the Cache stage is stalled, allowing the store buffer to write back into the cache, until all the values in the required lines no longer have the "on SB" flag set. At this point, without waiting for the entire store buffer to be cleared, the required element can be directly read from the now-updated line. We do not need to wait until the whole store buffer is empty because we know that no further entries will update the queried lines.

1.3 Line Replacement

We can identify three different situations for line replacement. The first, and most basic one, is when a load requires replacing a line that is not dirty and has no associated store buffer entries modifying it. In this case, a new line is requested from the "ARB" in the "TL" stage, and the pipeline stalls until it is returned. Once received, the line is placed in the "TL" and "C" stages as usual. In the second case, the line to be replaced is dirty. In this situation, the request to the "ARB" is still issued in the "TL" stage. Once the new line is received, the "C" stage sends an eviction notification back to the same ARB along with the evicted line. A particular detail arises when two lines are requested for replacement. Because of the staggered return, the "C" stage outputs two eviction lines: one immediately after receiving the first line, and the other after receiving the second one. The final case occurs when there are entries in the store buffer that must be written back to memory before eviction and replacement. To handle this, a detection is performed in the "TL" stage, and the pipeline waits until the store buffer has written back all elements related to that line. This reduces the situation to the one described in the previous paragraph.

All the instructions of load will be marked as done to the graduation list on the "WB" stage.

2 Store

The steps after being picked are very similar to the ones described on the load. One small different in the "RF" and Alu stage is that meanwhile the Load can only bypass one register, because the other one is marked as not valid. The store can bypass both of them to get a more updated value of the address that will access in addition to the value that will be placed. Then, when a store instruction reaches the "TL" stage, several actions can occur depending on the system state and the type of store request. The first situation is when the TLB entry for that address is not loaded in the stage, which, like in case of a load instruction, will trigger an exception that cancels the rest of the execution. The detailed handling of this case is described in the corresponding section.

Other situations, in order of complexity, are as follows:

2.1 Base Case

If the requested lines are present in the cache, the instruction progresses to the "C" stage, where the tuple '<Data, Address, Size>' is placed into the Store Buffer (SB). Execution stops at this point, and the corresponding completion bit is sent to the Graduation List. If the Store Buffer is full, the pipeline stalls for one cycle to allow the buffer to commit the corresponding line.

This is possible because, at each cycle, the SB proactively returns values that can be placed into the cache. It is the responsibility of the "C" stage, that, when not actively enabled, detect opportunities to consume a value from the Store Buffer, and notify the system so that in the next cycle a new set of values can be taken from the SB. This process continues until the Store Buffer is empty. Then, even if the cache is idle, the stage cannot consume more values, not being more values to consume.

Entries in the Store Buffer are not merged. This is acceptable because entries are placed in order from oldest to youngest. Because the Tail Head structure acts like a FIFO. With this, a younger entry will overwrite an older entry if they share the same address and size when updating the cache line. Or even partially.

For each entry in the SB, the total number of affected lines is computed. This information is used to track when a given line still has pending updates and cannot be replaced.

It is important to note that it is the responsibility of the cache to parse the Store Buffer and determine which cache line each entry corresponds to. One consideration was whether to store, instead of '<Value, Address, Size>', a precomputed tuple '<Value, Address, Size, Line0, Line1, IsLine1Used>' to simplify dirtying words that span multiple lines by preemptively splitting them. However, it was decided that the additional computation should not be added in the Cache stage that has the placement, specially if we have the non-enabled parts of the "C" stage, where it consumes the SB, that the totality of the job is just to dirty the entries from SB, being there the better placement to put this additional computational cost.

2.2 Line Replacement

If the required lines are not present in the cache, the instruction cannot advance to the next stage directly. The correct line must be present in the cache before it is placed into the SB, so that future overwrites can be tracked. In this case, the process defined in the Load section are reused, with the modification that the default base case is applied upon reaching the "C" stage rather than the load-specific one.

Since store instructions complete in the "C" stage instead of "WB" like other instructions, a mechanism is needed to prevent collisions when sending completion signals. If both "TL" and "C" attempt to mark completion in the same cycle, only one signal would be registered by the Graduation List, potentially preventing the instruction from graduating. To avoid this, before marking completion, the "TL" stage checks if "WB" is also signaling completion in the same cycle. If it is, the "TL" stage stalls for one cycle.

3 Arithmetic

Due to the requirement for a multistage pipeline with variable operation latencies, we implemented basic ALU operations that take a single cycle, as well as integer multiplication, integer division, floating-point addition, floating-point multiplication, and floating-point division, including the modulo operation. The "ALU" stage can extend execution time up to 6, 28 or 32 cycles for these multi-cycle operations.

An arithmetic instruction proceeds as follows. After fetching the values of the virtual registers in the "RF" stage, it enters the "ALU" stage. For a basic operation, such as integer addition or any single-cycle instruction, the result is computed, the "TL" stage is notified that the "ALU" cycle is complete and the result is available in the corresponding flip-flops, and the instruction continues normally through the remaining stages.

For multi-cycle operations, such as floating-point addition, the process is different. In the first "ALU" stage, the instruction is identified as a floating-point operation and marked as multi-cycle, so it does not yet enable the "TL" stage. Variables such as "t_isFloat", "t_isMult", and "t_isDiv" are used to determine the specific operation. For floating-point addition, these values are set to "1", "0", "0" respectively. This specific key enables the first floating-point addition module, providing the required input values. The operation then advances through multiple stages, validating the next stage on each sequential cycle. When the final stage is reached, the last enable signal is sent to the "TL" stage, indicating that the multi-cycle ALU operation is complete and the result is available.

3.1 Avoiding collision

It is critical to prevent multiple instructions from finishing their arithmetic computation in the same cycle, as this would cause a collision when enabling the "TL" stage, resulting in undefined behavior. This cannot happen for operations on different paths with the same number of steps because they start on different clock cycles and will finish on different clocks. However, since floating-point addition only takes 6 cycles, a floating-point multiplication could be started 26 cycles earlier, potentially colliding with it. This is prevented by stalling the start of any new ALU operation at key points until the previous operation has completed.

This won't be the only type of collision, this is because not all arithmetic operations are linear sequences of modules. Some are iterative, which means that they are looping a single module multiple times until the required number of cycles is completed. Two operations of the same iterative type cannot execute one after the other, as they would collide within the same module. To prevent this, a module acting like a "semaphore" stores which iterative operations are in use. If another instruction requires the same module, it is stalled until the current operation completes. Multiple different iterative operations, such as floating-point multiplication and division, can run concurrently; the restriction only applies when the same arithmetic operation is used.

3.2 Writing to registers

The end part of most of the arithmetic end by writing to the RF module to update the new computed value. This part is similar as to the not Out Of order architecture, but, in this case we will write into the address set by the virtual registers that were assigned at the decoding stages.

4 Branch Jump

The out-of-order scheme requires a given structure to be able to recover itself from incorrect jumps; in this case, this is due to speculative access. In addition, we have included a branch predictor in the Fetch stage. We will start this explanation considering that the branch predictor is completely empty. Instructions are loaded into the "Decode" stage each clock cycle if nothing stalls or is trapped. During this process, Decode checks whether the instruction is a jump. If it is, we know that the PC may change, but we do not know for sure, nor do we know where it will jump. This information is communicated to the array of rename tables and their free lists, in preparation for possibly having to cancel instructions that will be falsely committed.

We clone the current rename table, and from this point on, all new possibly speculative instructions will only modify the new assigned one.

We continue this process until the instruction containing the jump is picked, and the jump is validated, either as correct or invalidated. If the prediction was correct, nothing needs to be changed. If it was incorrect, we need to "decommit" and restore the previous rename table.

When information about a jump is obtained in the ALU stage, whether it was taken or not, this information, together with the jump address and the target address, is also given to the branch predictor so that it can formulate predictions for that address in the future.

If a prediction exists, it immediately affects the Fetch stage when the program counter is read. The difference this time is that a bit is sent through the entire pipeline indicating whether the branch was predicted as taken or not. If the prediction is correct, the ALU confirmation will indicate that nothing needs to be done. If it is incorrect, it will return the PC without the jump as the place from which execution should continue.

4.1 Jump Predictor

The jump predictor implemented in this project uses a two-bit state machine to express the confidence that a given address will result in a jump. Each time the prediction is validated and results in the correct speculation of the next address, a bit is added to the state machine, unless it is already at the maximum confidence (all bits set to 1).

On startup, the state machine is initialized in the third state, which represents the minimum confidence required to predict that a jump will be taken. This way, once we learn that a given address resulted in a taken jump, it can be predicted as taken the next time it appears. If the initialization were any lower, we would first need to detect that a jump was taken and then observe it being taken multiple additional times before the predictor gained enough confidence. By initializing it just after the first validation threshold, the predictor relies on the assumption that if a given address has jumped once, it is likely to jump again in the future.

4.2 Jump and Link

Jump and link instructions work differently than branches. In this case, there is no condition and the jump is always taken. This would suggest that we could redirect execution in the Fetch stage and avoid loading incorrect subsequent instructions. However, this is not possible because, in the case of JALR, the target address is computed by adding a register value to an immediate. This means that we do not have enough information to make a prediction at this stage. For this reason, these instructions use the same scheme as regular branches. The only difference is that the corrected PC is not simply $pc + immediate$, but rather $rX + immediate \sim (1)$.

5 Exceptions

Because of the out-of-order implementation, a systematic method for handling exceptions will be implemented in the Graduation List module. In each cycle, instructions fetched from memory are placed into the graduation list. A set of values is stored in a table following a "Tail" pointer. With this set of elements, we track a "has_exception" bit and a "type_exception". Initially, these are set to false and NO_EXCEPTION. Normal execution of the stage then proceeds. When an instruction raises an exception, its GLIdx and the type of exception are checked. All exceptions that occur in that cycle are sent to the "Exception Call Resolution" module. This module selects the oldest exception and sends it to the Graduation List.

This step, validating which exception is the oldest using the GLIdx, is preferable to simply checking which instruction is furthest along in the stages, which could be easily done with a chain of if statements. But this would be incorrect, because in cases where two interruptions occur, the out-of-order mechanism might schedule a younger instruction before an older one. If the youngest instruction exception is chosen based solely on stage position, the exception could be reported at the incorrect PC. For example, if DIV 2/0 and DIVF 2/0 occur and DIVF is scheduled first, even if the DIV appears in the code first, the exception would be incorrectly attributed to the second instruction if stage position alone were considered.

5.1 Exception handling on the Graduation List

When a "has_exception" bit is set to true in the graduation list, it waits until all previous instructions are graduated. This ensures that older exceptions are handled first. When it is time to graduate an instruction with an exception, this means that all the other lines and instructions present in the machine are part of the set of younger instructions that must not be committed, so, to assure that, we will clear the Graduation list, and the Instruction Window, the relevant exception information (PC and cause) is sent to the RF, the PC is set to the OS handler, and the superuser bit is enabled. A problem that was detected what that even if no new elements could be picked that were part of the program that experienced a trap, elements that resided still in flip-flop of the remaining stages would try to compute and update values. So, on exception, we also clear all the flip-flop values that are present in between the stages.

Once in the OS handler, we needed a way to comprehend the complete state of the machine and why it resulted in exception. For that, the interruption state can be read with the instruction 'CSRRS x10, mcause, x2'. For reading all the possible characteristics, we implemented the keywords "mepc", "mcause", and "mva" to restore the stage state. Additionally, custom instructions '$TLB.Sx0, x0$', '$TLB.I$', and '$TLB.C$' are used to fill the "RF" stage with the correct virtual page values and update the TLB in the "Fetch" or "TL" stage, the two registers using its 64 bits will be used to fill the virtual, the physical and the flags of the page added. Finally, 'SRET' resumes execution at the address stored in "mepc" and exits superuser mode.

Custom values can be modified using the 'CSRRW' instruction, which takes two registers and a keyword, storing the previous keyword value in the destination register and updating the keyword with the second register's value. Additionally, the 'CSRRWI' is implemented to directly apply to the registers the given immediate.

6 Performance

In this section, we present the results obtained by running several assembly benchmarks and measuring the resulting instructions per cycle (IPC). To put these values into perspective, Table 1 summarizes the main operation arbitrary delays and sources of memory-related overhead that impact performance.

Table 2 reports the total number of executed instructions, the total number of cycles, the execution time, and the resulting IPC for each benchmark case. Finally, Figure 3 provides a visual comparison of the IPC values across all evaluated benchmarks.

Table 1: Operation Delays

Value	**Delay (cycles)**
Arb Query Line	2
Arb Eviction Line	2
Wake Up GL From Pick	1 (Load/Store), 0 (Others)
Floating-point addition	6
Floating-point multiplication	32
Floating-point division	28
Integer multiplication	32
Integer division	32

Table 2: Instruction Timing Cases

Case	**Total Instructions**	**Cycles**	**Total Time**	**IPC**
Memory copy (< 2)	32	137	1,370,000 ps	0.2336
Memory sum (< 2)	17	119	1,190,000 ps	0.143
Perfect case	66	233	2,330,000 ps	0.283
Arithmetic	21	88	810,000 ps	0.259
Large array byte storage	242	725	7,250,000 ps	0.334

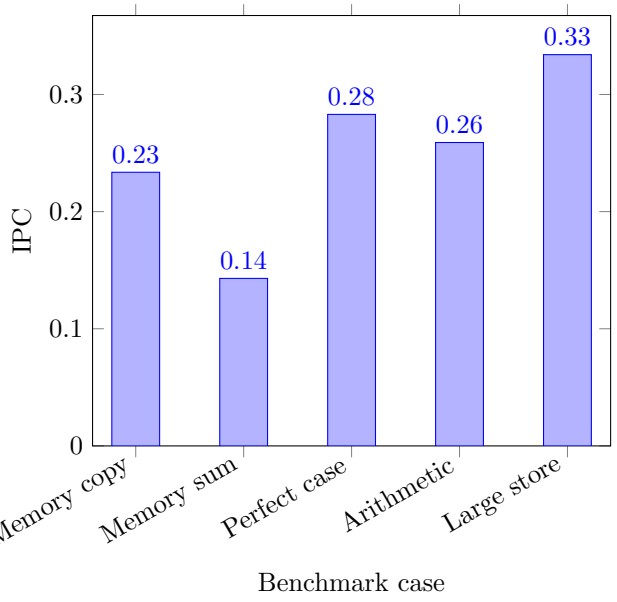

Figure 3: IPC per benchmark case

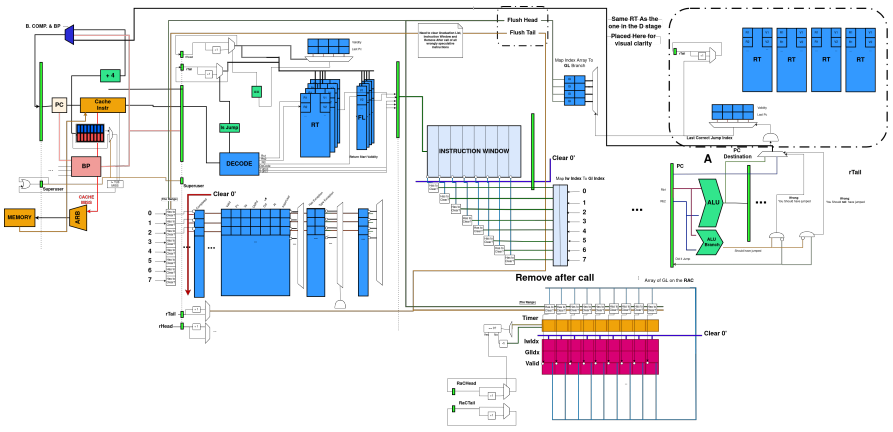

Figure 4: Range of instructions canceled after a branch misprediction, showing the affected entries in the pipeline structures.

7 Annex

7.1 Removing incorrect ranges of instructions from the system

When detailing the characteristics of branch operations, we described the process of decoupling and removing all speculative accesses that populate the system after discovering that a jump was incorrectly taken. We considered it important to further detail the complete procedure, as this part of the system caused the most difficulty during the development of the assignment. In Figure 4, the range of instructions affected by this cancellation is shown.

To explain why each removal step is necessary, we begin with the Graduation List. This is the easiest case to justify: In the lower ranges of the head pointer, there are entries corresponding to incorrect instructions that must not be committed. Since the Graduation List cannot skip entries on a per-cycle basis, these instructions must be removed as soon as it is known that they are invalid. The system guarantees that no incorrectly speculative instruction will be prematurely committed, because committing these younger instructions would require first committing the older branch or jump that caused the misprediction. And, at that point, the ALU stage has already determined that the speculative path is incorrect.

Regarding the Instruction Window, it is necessary to prevent the selection and execution of potentially incorrect instructions that could wrongly update the machine state. Therefore, these entries must also be removed by range. The removal cannot rely directly on Instruction Window indices, since they are decoupled from the Graduation List indices. Instead, a mapping between Instruction Window indices and Graduation List indices is used to identify and remove the corresponding entries.

The final component is the most complex to justify. The system must avoid receiving completion notifications from instructions that have already been removed due to cancellation, as such notifications could incorrectly refer to the newer, correct, instructions. However, resources allocated to these canceled instructions, such as instruction window indices, must still be returned to the free list. To handle this, the return delayed value of now canceled instructions are marked as *Kill Canceled*. These entries do not update the architectural state o the instruction window, but safely return their associated resources to the free list.