Elementary Microarchitecture Algebra: Top-Level Proof of Pipelined Microarchitecture

John Matthews and John Launchbury

Oregon Graduate Institute,
P.O. Box 91000, Portland OR 97291-1000, USA
{johnm,jl}@cse.ogi.edu
http://www.cse.ogi.edu/PacSoft/Hawk

Abstract. This is a companion note to *Elementary Microarchitecture Algebra* [1] and outlines an algebraic simplification proof of the pipelined microarchitecture described in that paper.

1 Transforming the Microarchitecture

The laws presented in *Elementary Microarchitecture Algebra* [1], as well as others introduced in this note, can be used for aggressively restructuring microarchitectures while retaining behavioral equivalence. The example we present here contains three levels of forwarding logic, resolves hazards by stalling the pipeline, and performs branch speculation. The block diagram for this microarchitecture is shown in Figure 1.

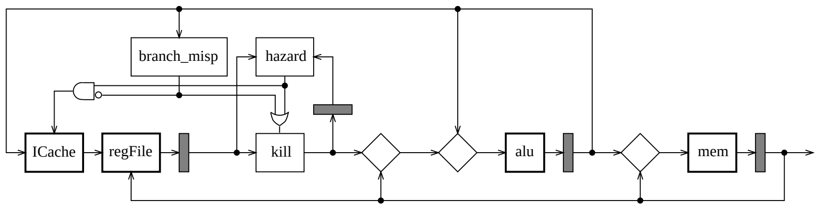

Fig. 1. Microarchitecture before simplification

By just using algebraic laws, we have been able to reduce most of the complexity leaving essentially an unpipelined microarchitecture. We have implemented some of the algebraic laws as a rewrite system in Isabelle. The proof proceeds in stages, according to the geometric goal we are pursuing in each stage.

2 Retiming Stage

We first remove all delay circuits from the main pipeline path, starting at the earliest stage in the pipeline. We accomplish this by repeatedly applying the time-invariance law, and by splitting delays along wires through the circuit duplication and feedback rotation laws.

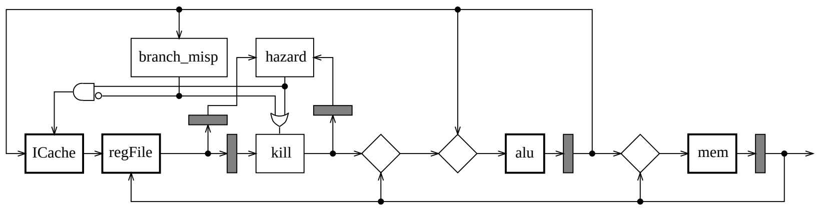

Fig. 2. Split `delay` circuit after `regFile`, using the circuit duplication law

We would now like to move a `delay` through the `kill` circuit, but we can't, since the top input to `kill` does not have a `delay` circuit. To place a `delay` on `kill`'s top input, we will need to move `delay` circuits through the `branch_misp` and `hazard` circuits. This is possible because `branch_misp` and `hazard` are pure combinational circuits that preserve default values (The default value for Booleans is `False`) and are therefore time-invariant.

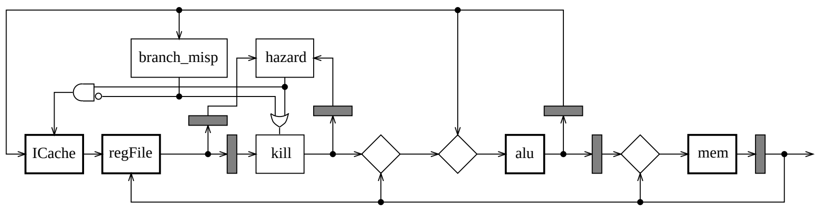

Fig. 3. Split `delay` circuit after `alu`, using the feedback-rotation law

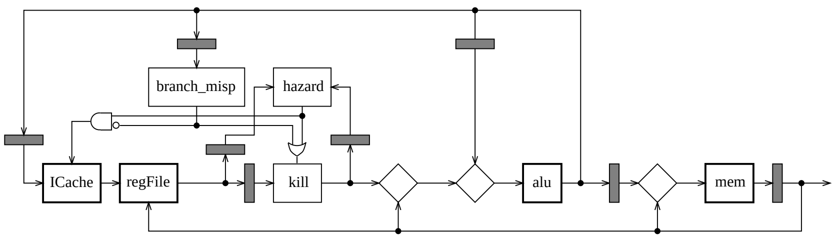

Fig. 4. Split twice the `delay` circuit leading to `branch_misp` and `ICache`, using two applications of the circuit-duplication law

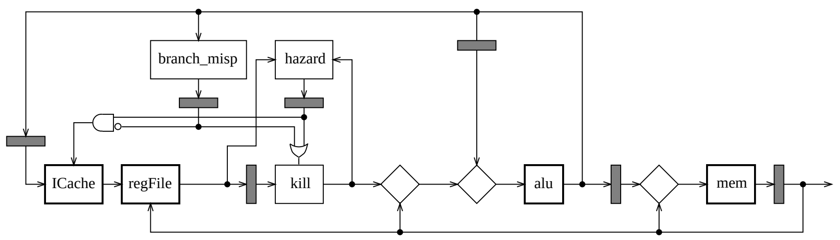

Fig. 5. Move `delay` circuits through the `branch_misp` and `hazard` circuits, using the corresponding time-invariance laws

We can similarly move these `delay` circuits through the `or` and `and` circuits (even though one of the `and` inputs is inverted), since these combinational circuits preserve the default `False` Boolean value. Finally, we can move the original `delay` circuit through the `kill` circuit, since `kill` is a combinational circuit and all of its inputs have delays.

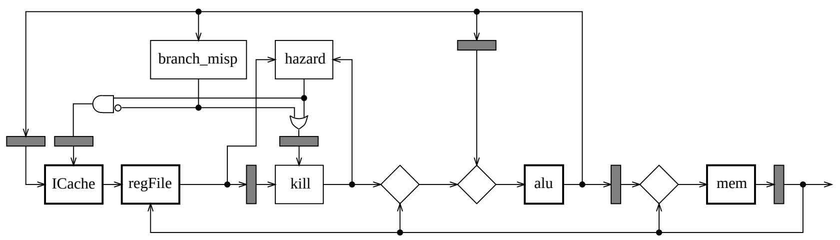

Fig. 6. Move `delay` circuits through the `or` and `and` circuits, using the circuit-duplication law and the corresponding time-invariance laws

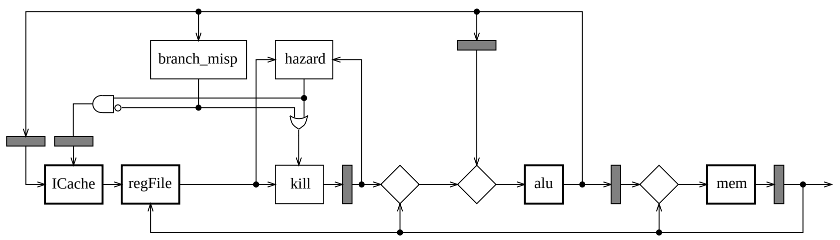

Fig. 7. Move `delay` circuits through the `kill` circuit, using the corresponding time-invariance laws

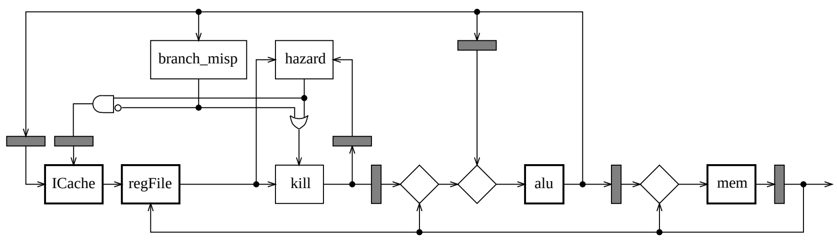

Fig. 8. Split the `delay` circuit after the `kill` circuit, using the circuit duplication law

Once again, we can't move the `delay` circuit past the `bypass` circuit, since the other input to the bypass does not contain a `delay`. Fortunately, the other input originates at the `delay` circuit that is after the `mem` circuit, so we can split that `delay` and move it to the `bypass` input.

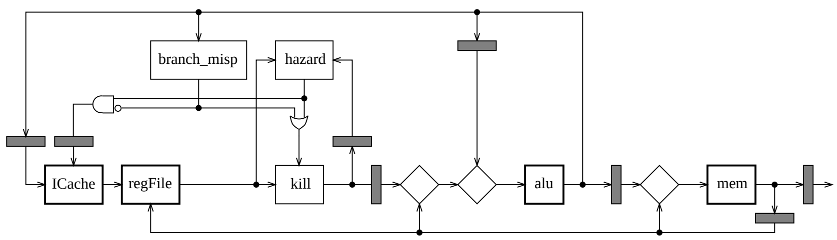

Fig. 9. Split the `delay` circuit after the `mem` circuit, using the feedback rotation law

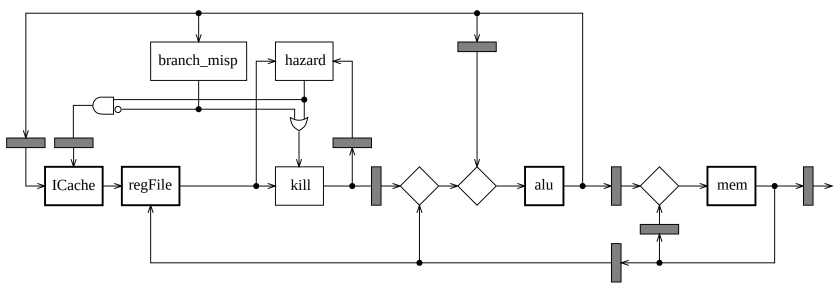

Fig. 10. Split the bottom-most `delay` circuit, using the circuit duplication law

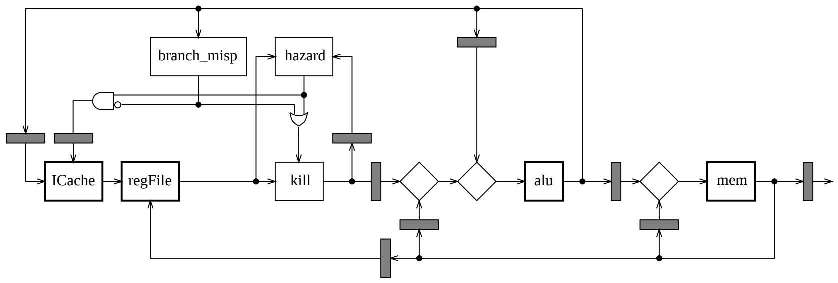

Fig. 11. Split the bottom-most `delay` circuit again, using the circuit duplication law

We can now move our wandering `delay` through the two `bypass` circuits, since bypasses are time-invariant, and they both have `delay` circuits on all inputs.

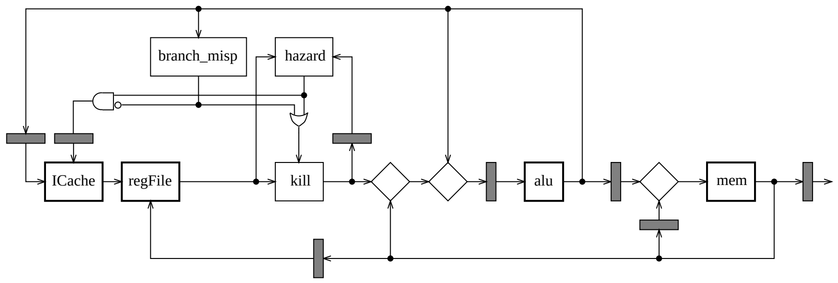

Fig. 12. Move the `delay` circuit before the first `bypass` circuit through the first and second bypasses, using the corresponding time-invariance laws

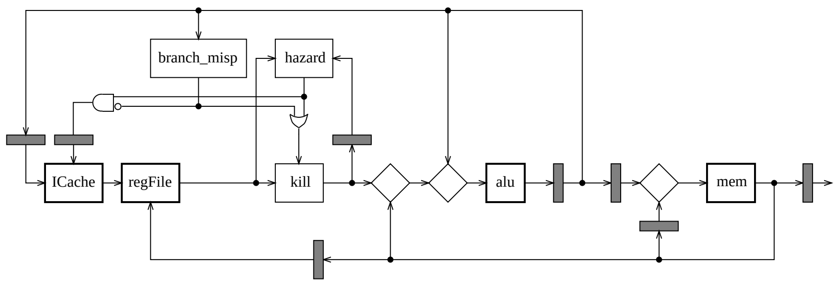

Fig. 13. Move the `delay` circuit through the `alu` circuit using the corresponding time-invariance law

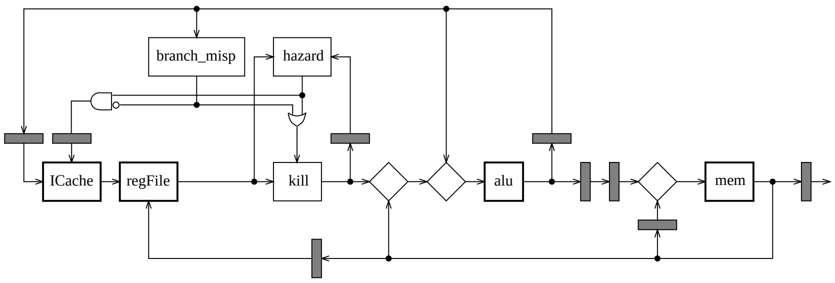

Fig. 14. Split the `delay` circuit after the `alu` circuit using the feedback-rotation law

Now we just have to move the two `delay` circuits before the third `bypass` circuit to the end of the pipeline. Fortunately, both `bypass` and `mem` are time-invariant.

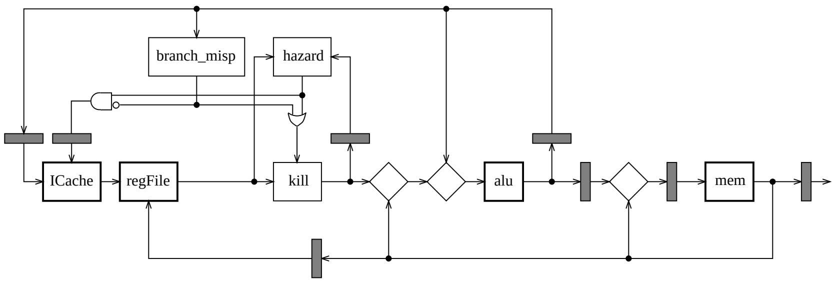

Fig. 15. Move the `delay` circuit through the third `bypass` circuit using the corresponding time-invariance law

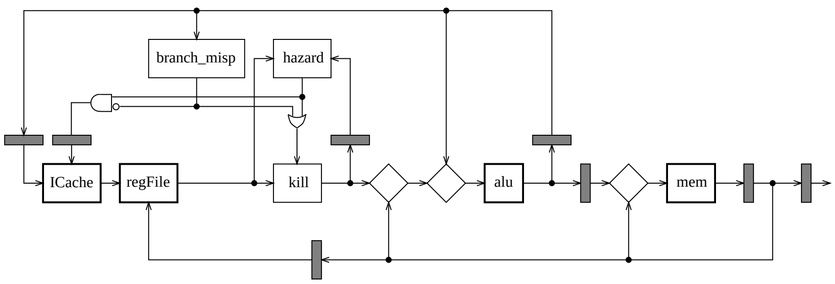

Fig. 16. Move the `delay` circuit through the `mem` circuit using the corresponding time-invariance law

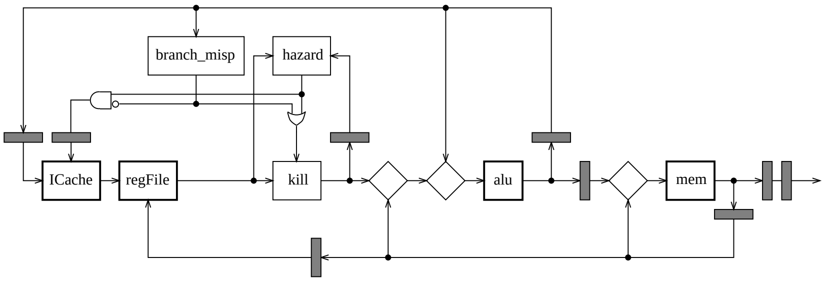

Fig. 17. Split the `delay` circuit after the `mem` circuit, using the corresponding feedback-rotation law

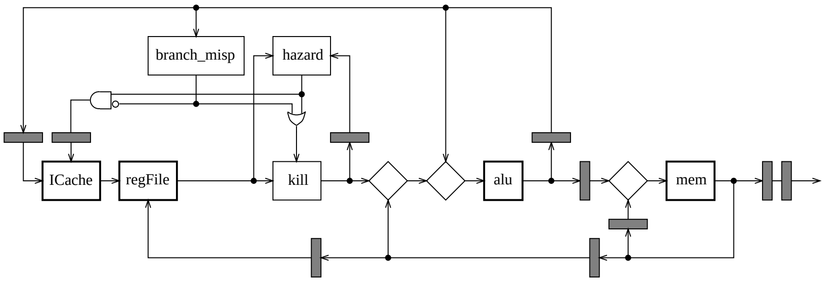

Fig. 18. Split the `delay` circuit below the `mem` circuit, using the corresponding circuit duplication law

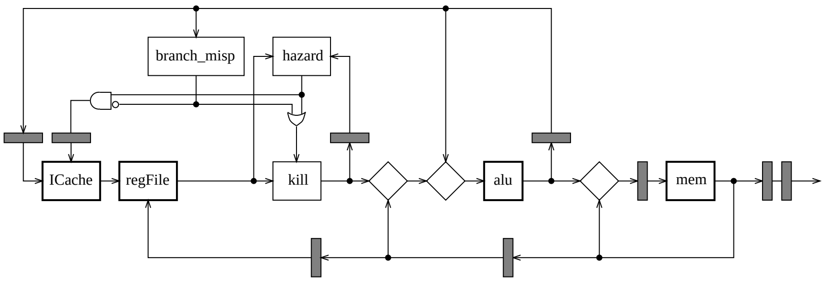

Fig. 19. Move the `delay` circuit through the last `bypass` circuit, using the corresponding time-invariance law

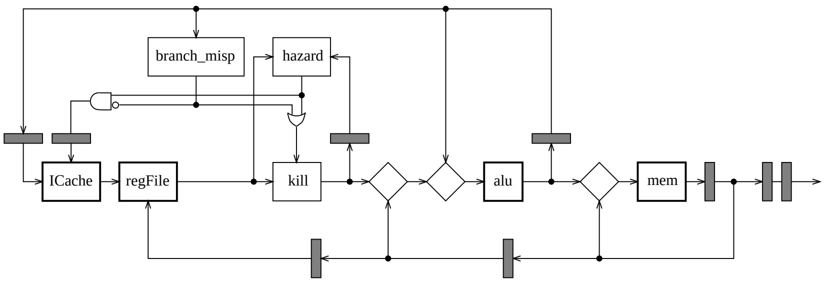

Fig. 20. Move the `delay` circuit through the `mem` circuit, using the corresponding time-invariance law

We'll keep moving this last delay a bit, to set up for the hazard-bypass law later on.

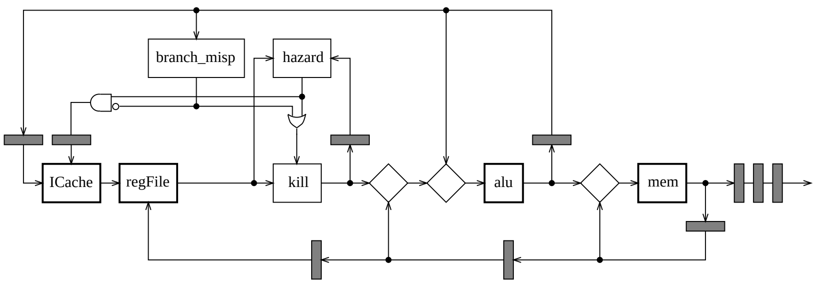

Fig. 21. Split the `delay` circuit after the `mem` circuit, using the feedback-rotation law

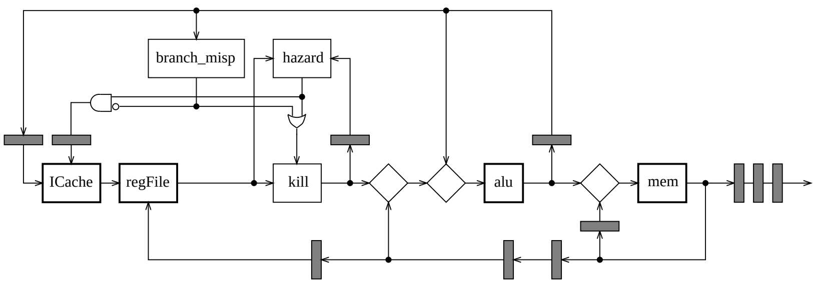

Fig. 22. Split the bottom-rightmost `delay` circuit, using the circuit duplication law

3 Move Control Wires Stage

In this stage we move all wires not directly involved with forwarding logic to either before or after all of the `bypass` circuits. This is to enable the hazard-bypass laws, which we apply in a later step. We move the wires by inserting projection circuits and using the corresponding projection-commutativity laws. While we're at it, we'll also insert `proj_ctrl` circuits on the inputs to the `hazard` circuit, so that we can later on move the register file next to the first bypass.

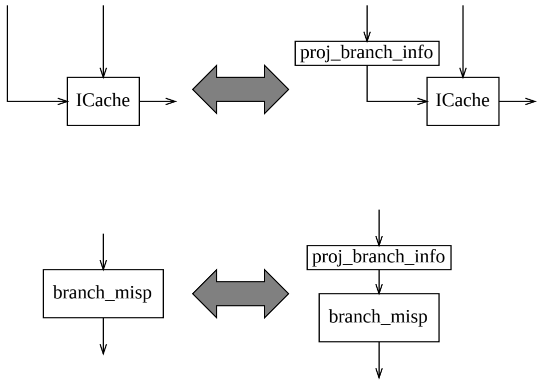

Fig. 23. Projection insertion laws for `proj_branch_info`

The wire we want to move in this case is the feedback wire after the `alu` circuit, which becomes the input to `branch_misp` and `ICache`. The projection that allows us to move the wire is called `proj_branch_info`. On each clock cycle, `proj_branch_info` examines the opcode field of its input transaction. If it is a branch instruction, then it outputs a transaction with the same opcode, destination register name, destination value, and speculative branch target PC fields as the input transaction, but with all other fields (including source-operand register name fields) set to their default values[1]. If the transaction is not a branch instruction, then `proj_branch_info` outputs `nopTrans`. Since the `ICache` and `branch_misp` circuits only examine branch instructions, and in fact only those fields that `proj_branch_info` lets through to its output, then `proj_branch_info` really is an input projection of these two circuits (Figure 23). We thus insert these projections and move them towards the `alu` circuit.

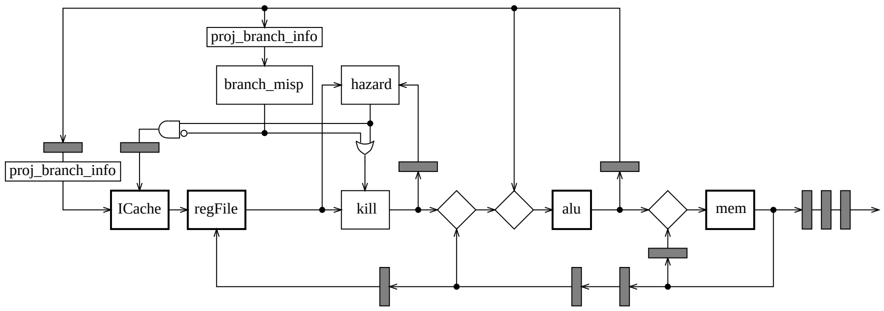

Fig. 24. Insert `proj_branch_info` projection on the inputs to `ICache` and `branch_misp`, using the corresponding projection laws from Figure 23

[1] Our ISA architecture hard-wires register `R0` to zero, so `R0` serves as the default value for register names

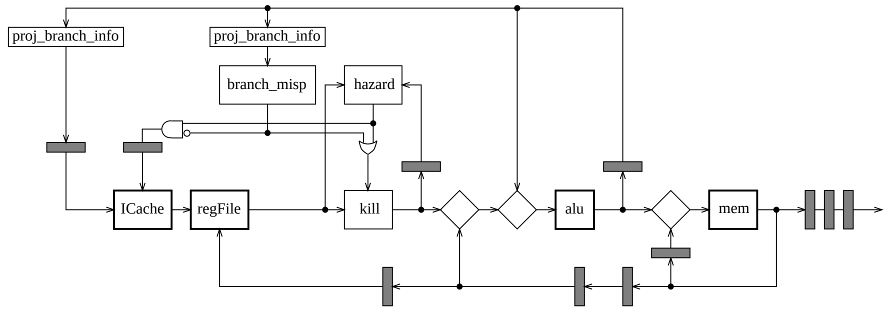

Fig. 25. Move `proj_branch_info` past the left-most `delay`, using the corresponding time-invariance law

To continue moving the `proj_branch_info` projection, we apply the circuit duplication law in reverse, merging the two projections into one.

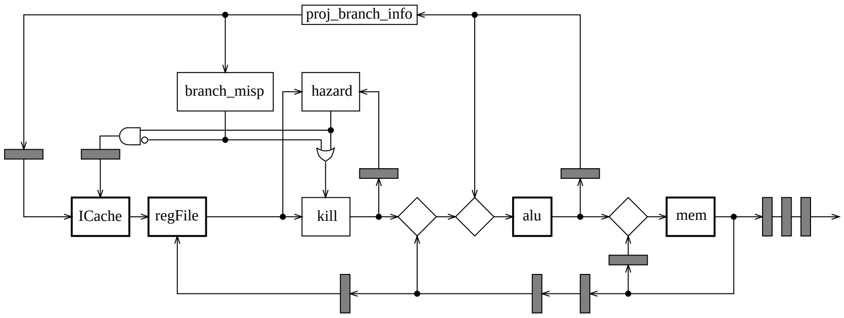

Fig. 26. Merge the two instances of `proj_branch_info`, using the circuit duplication law in reverse

At this point we can't move the `proj_branch_info` circuit any further, since we cannot insert a `proj_branch_info` circuit on the wire leading to the second bypass without changing the functionality of the pipeline. What we do instead is split the `delay` that is to the right of the projection, using the feedback rotation law (and split the feedback wire while we're at it). Once we have duplicated the `delay`, we can continue moving `proj_branch_info` down towards the `alu` circuit.

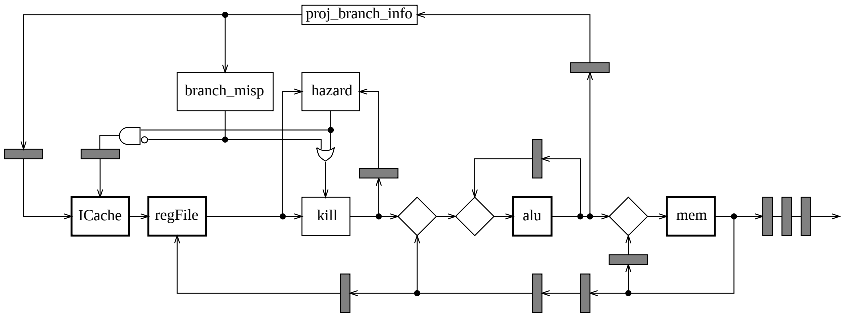

Fig. 27. Split the `delay` circuit ahead of `proj_branch_info`

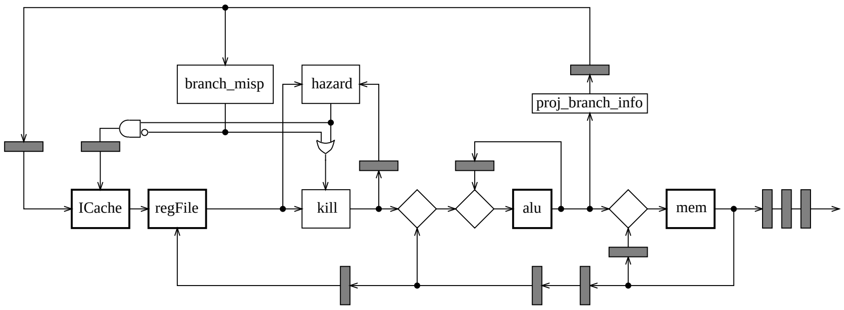

Fig. 28. Move the `proj_branch_info` circuit past the `delay` circuit using the corresponding time-invariance law

Now that `proj_branch_info` is at the output of the `alu` circuit, we can use *projection-invariance* laws to move the projection to the end of the pipeline. Projection-invariance laws act somewhat like commutativity laws, and state that the output of a projection is unchanged when its input signal is moved across another circuit. Figure 29 shows some of the laws for `proj_branch_info`. In particular, we can move the projection past the third `bypass` circuit and the `mem` execution unit of Figure 28, since neither of these circuits alter a transaction's branch information.

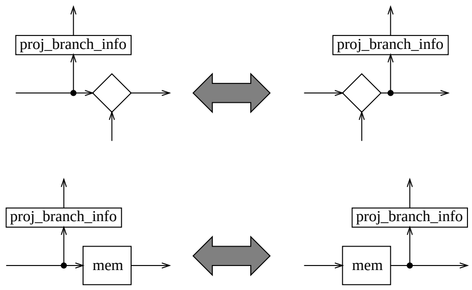

Fig. 29. Projection-invariance laws for `proj_branch_info`

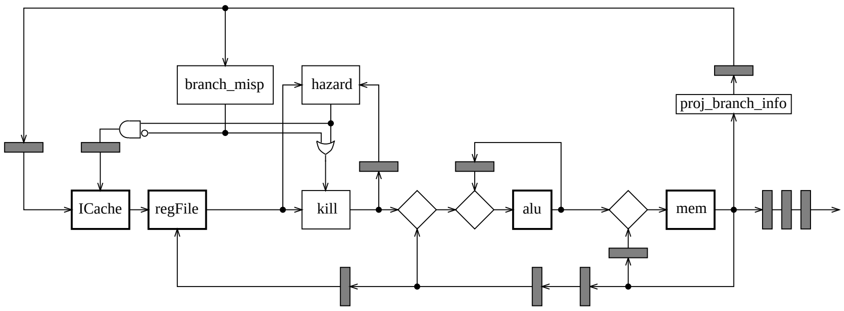

Fig. 30. Move `proj_branch_info` past the third `bypass` and `mem` circuit, using the projection invariance laws from Figure 29

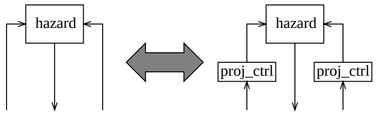

Fig. 31. `proj_ctrl` projection insertion law

To prepare for a future stage, we will also add `proj_ctrl` projections to the inputs of the `hazard` circuit. The `proj_ctrl` circuit passes the opcode, source register name, and destination register name fields of its input transaction through unchanged, but zeros-out all other fields. Since the `hazard` circuit only examines these *control* fields, then the projection insertion law shown in Figure 31 is valid.

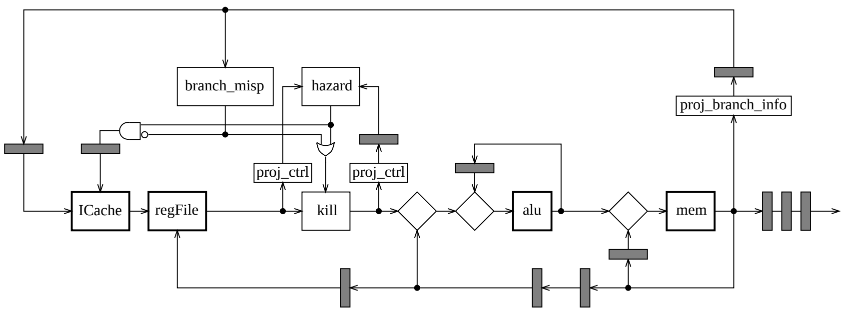

Fig. 32. Add `proj_ctrl` projections to the inputs of the `hazard` circuit using the corresponding projection-insertion laws (Figure 31), and move the right-most `proj_ctrl` circuit past the delay using the corresponding time-invariance law

4 Propagate Hazard Information Stage

At this point we would like to start removing `bypass` circuits using the hazard-bypass law. But this law can only be applied when there are no hazards between the affected stages. So we generate a no-hazard projection at the end of the dispatch stage (which is justified by a projection-absorption law applicable to the kill-circuit complex in that stage), and then move it between the first and second bypass circuits.

The `no_haz` projection insertion law we use at this stage is a slight generalization of the one discussed in the paper, and is shown in Figure 33. This generalized law holds since the `kill` circuit is still guaranteed to "squash" all potential hazards, and in fact may squash other transactions as well. We use this law to insert a `no_haz` circuit after the `kill` circuit in the microarchitecture.

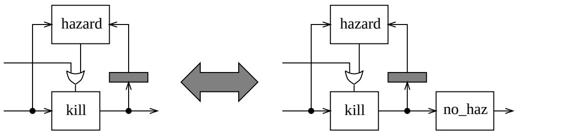

Fig. 33. Generalized `no_haz` projection insertion law

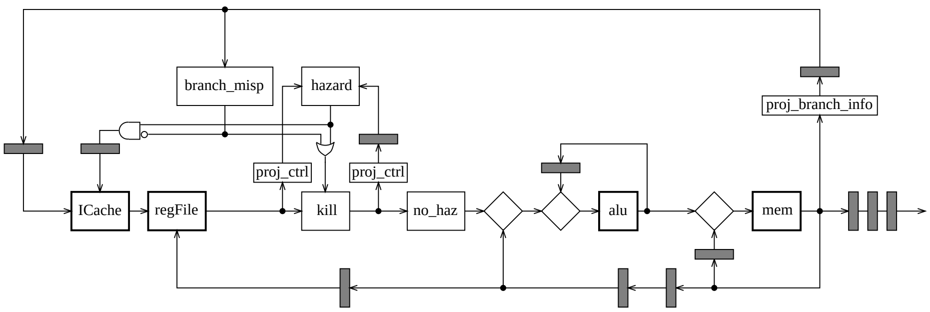

Fig. 34. Insert a `no_haz` projection after the `kill` circuit, using the projection insertion law shown in Figure 33

As mentioned in the paper, the `no_haz` projection commutes with `bypass` circuits. One can see this by noting that `bypass` never changes the transaction fields that `no_haz` examines. Thus `no_haz` will squash the same transactions regardless of whether it is placed before or after the `bypass`. If `no_haz` does squash a transaction by replacing it with `nopTrans`, then `bypass` will not modify the squashed transaction, since `nopTrans` contains no source operands. The `no_haz` circuit acts like an identity on transactions it does not squash, so again it does not matter whether it is placed before or after the `bypass` circuit in this case.

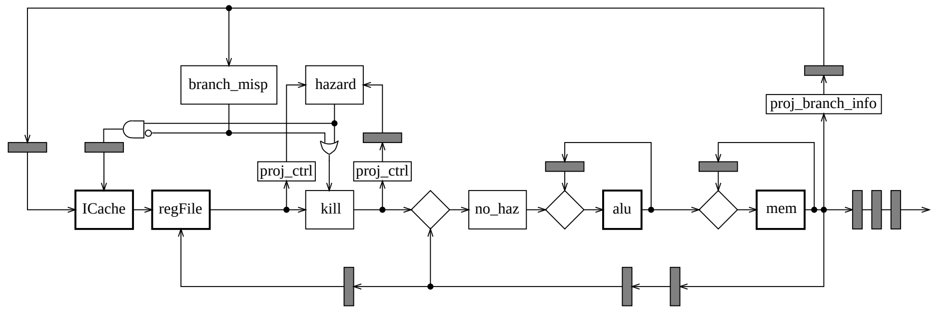

Fig. 35. Commute `no_haz` with the first `bypass`, using the corresponding projection commutativity law (we also reroute the `mem` stage feedback wire)

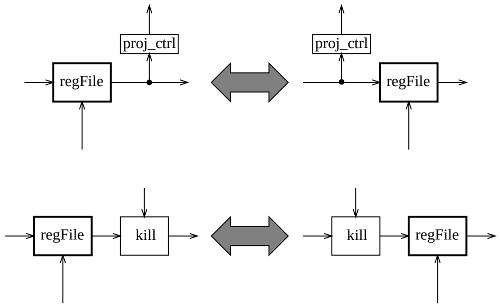

Fig. 36. register file commutativity laws

We will next swap the register file with the `kill` circuitry using the two laws shown in Figure 36, so that the register file is closer to the bypass circuits we want to eliminate. The first law holds since the register file does not modify a transaction's control fields. It is easy to show that the second law holds by performing a case analysis on the Boolean input into `kill`: If the input is `true` at a given clock cycle, then both the left-hand and right-hand circuits output `nopTrans`. If the input is `false`, then the `kill` circuit acts as an identity, so the outputs in both circuits are identical.

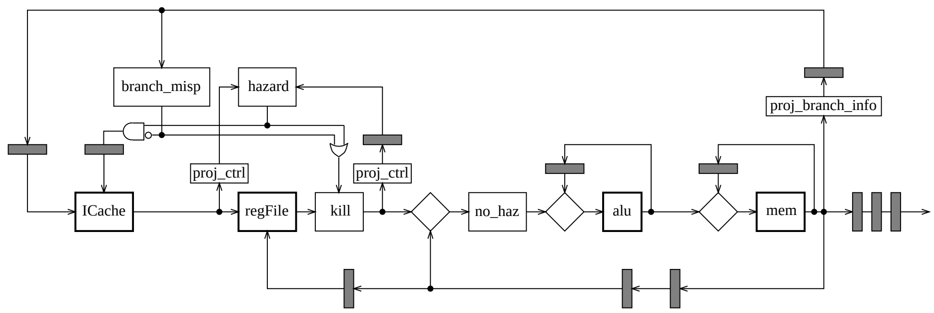

Fig. 37. Commute the first `proj_ctrl` projection with the register file, using the first law of Figure 36

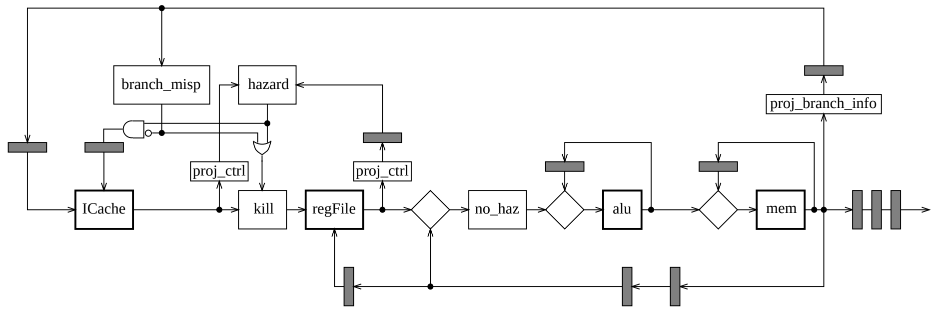

Fig. 38. Commute the register file with the `kill` circuit, using the second law of Figure 36

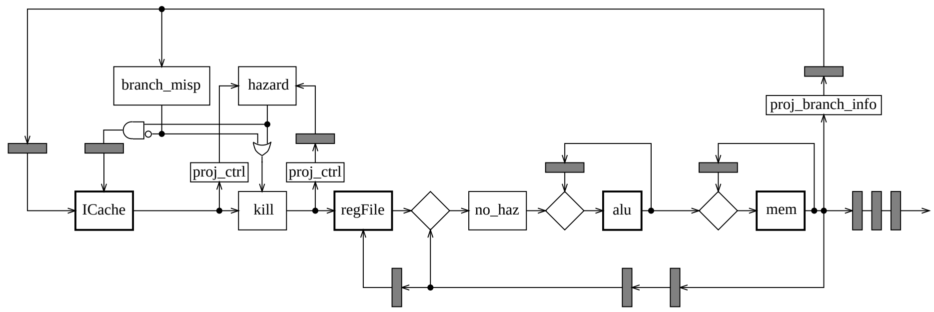

Fig. 39. Commute the second `proj_ctrl` projection with the register file, using the first law of Figure 36

5 Remove Forwarding Logic Stage

We are now in a position to start removing bypass circuits. The first bypass circuit can be removed immediately, due to the register-bypass law:

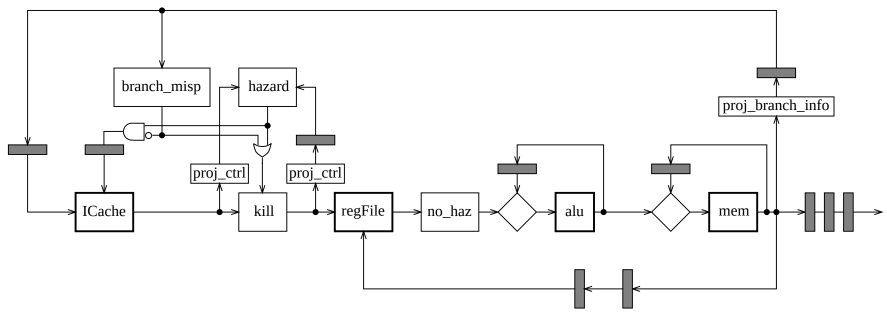

Fig. 40. Use the register-bypass law to remove the left-most `bypass` and the `delay` circuit below it

We can now apply the hazard-bypass law to remove the bypass circuit just prior to the memory unit.

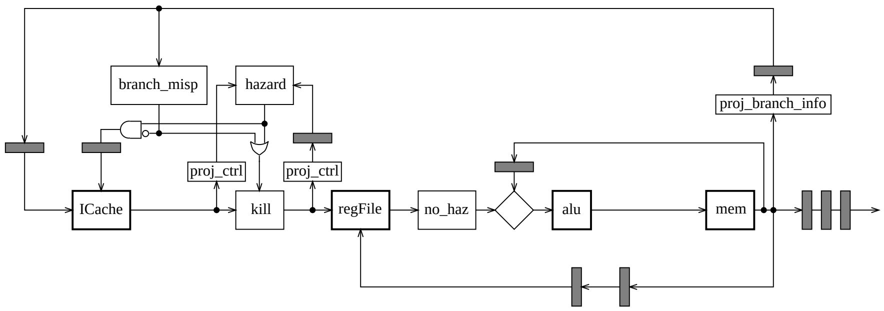

Fig. 41. Remove the right-most `bypass` circuit using the hazard-bypass law

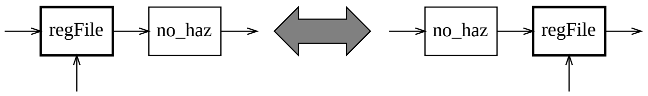

Fig. 42. register file commutes with hazard projection

Next, we can swap the `no_haz` projection with the register file (Figure 42), since the register file never alters its input's control fields, and since the internal state of the register file is only affected by its writeback input, not its data input. Once we have swapped the two components, we can remove the `no_haz` projection by applying the law in Figure 33.

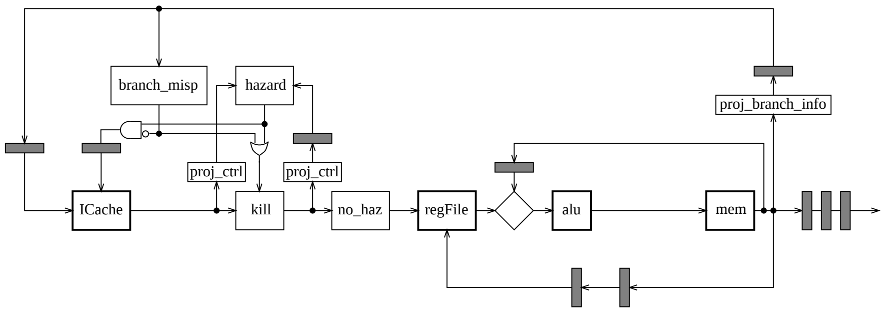

Fig. 43. Swap the register file with `no_haz`, using the commutativity law in Figure 42

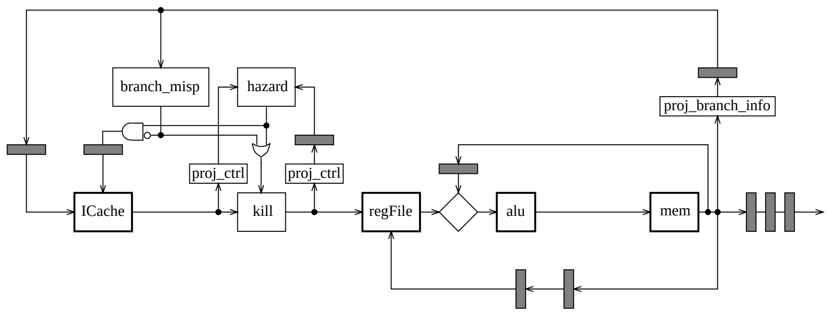

Fig. 44. Remove `no_haz`, using the `no_haz` projection insertion law (Figure 33) in reverse

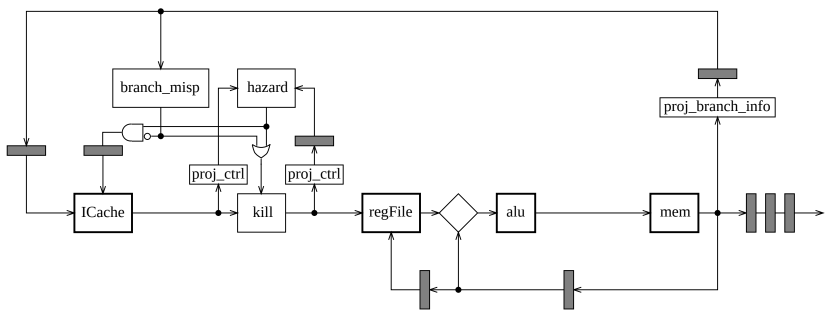

Fig. 45. Merge the `delay` feeding into the remaining `bypass` circuit with the right-bottom-most delay, using the circuit-duplication law in reverse.

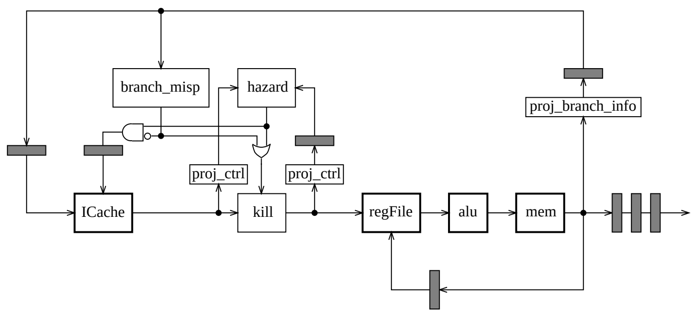

Fig. 46. Remove the last `bypass` circuit, using the register-bypass law

6 Cleanup Stage

The pipeline has now been simplified as much as possible, except that there are still some extra delay components as well as several unnecessary projection circuits. We merge delay components, then move the projection circuits back to their places of origin and remove them using the projection laws in the opposite direction.

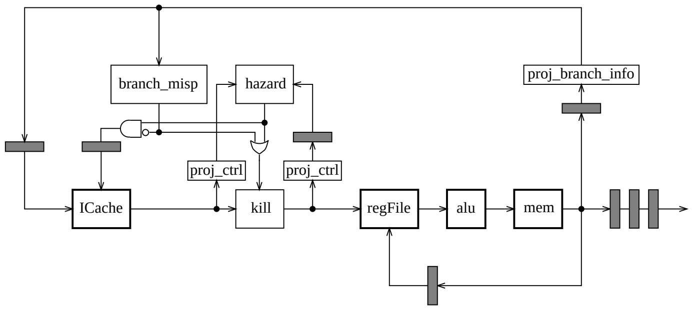

Fig. 47. Swap the `proj_branch_info` projection with the `delay` next to it, using the corresponding time-invariance law.

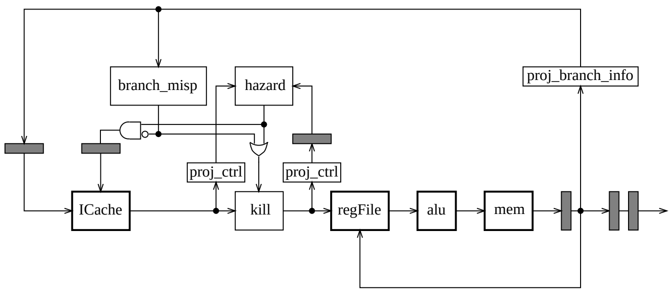

Fig. 48. Merge the three forking `delay` circuits after the `mem` circuit, using the feedback rotation law in reverse.

We would like to remove as many `delay` circuits as possible when simplifying microarchitectures, and there is a way we can merge the `delay` leading into the `hazard` circuit with the `delay` after the `mem` unit. Neither the `alu` nor the `mem` units ever modify the control fields of a transaction, so `proj_ctrl` commutes with both of them (Figure 49).

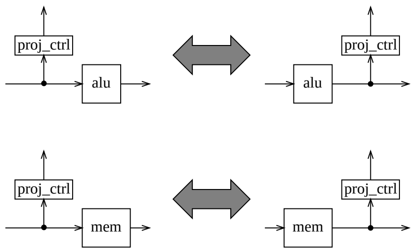

Fig. 49. More `proj_ctrl` projection invariance laws

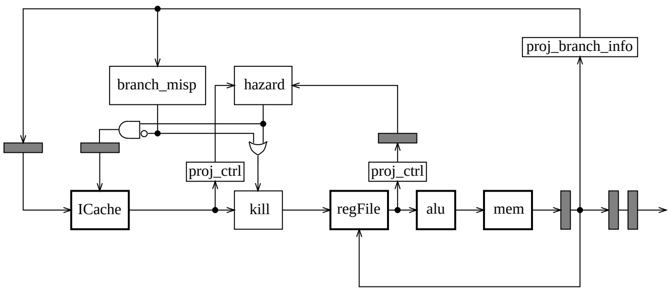

Fig. 50. Move the right-most `proj_ctrl` circuit past the register file, using the first law of Figure 36

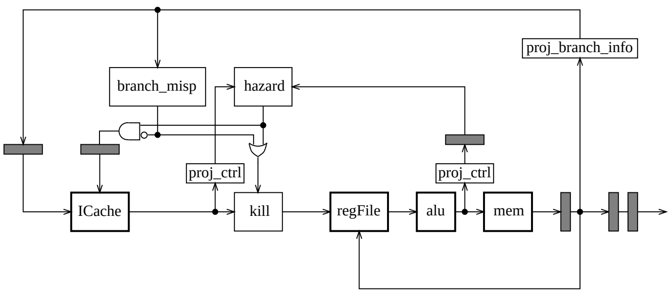

Fig. 51. Move the right-most `proj_ctrl` circuit past the `alu`, using the first law in Figure 49

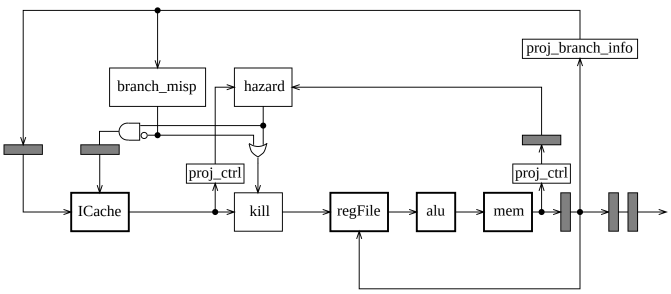

Fig. 52. Move the right-most `proj_ctrl` circuit past the `mem`, using the second law in Figure 49

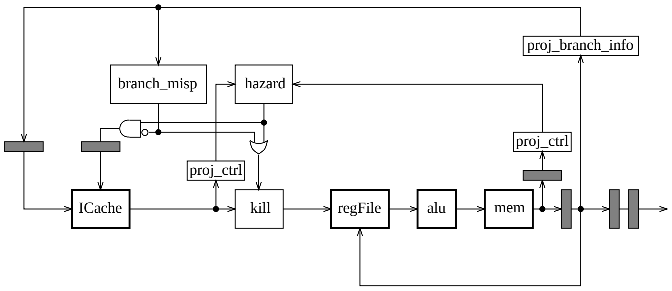

Fig. 53. Swap the right-most `proj_ctrl` circuit with the `delay`, using the corresponding time-invariance law

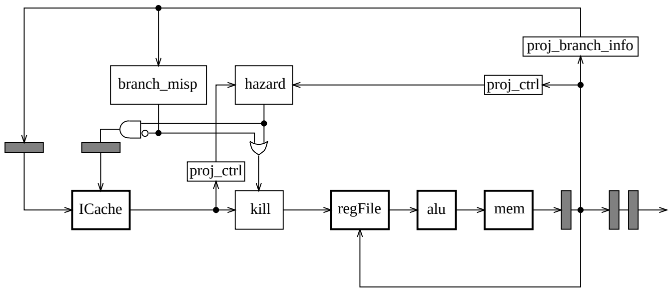

Fig. 54. Merge the `delay` after the `mem` unit with the `delay` below the right-most `proj_ctrl`, using the feedback rotation law in reverse

All that remains now is to absorb the projection circuits back into the circuits they were created from.

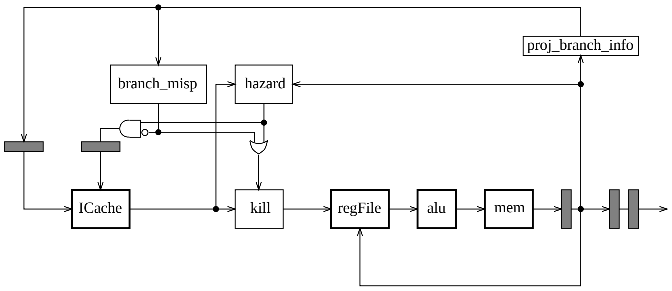

Fig. 55. Remove `proj_ctrl` circuits, using the projection insertion law of Figure 33 in reverse

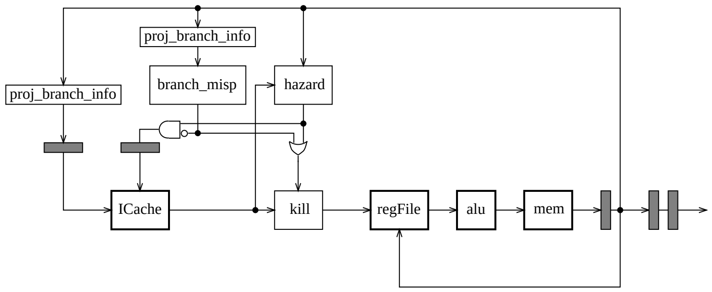

Fig. 56. Split the `proj_branch_info` projection, using the circuit duplication law

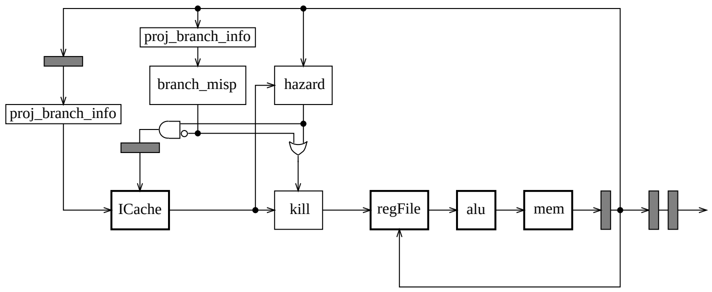

Fig. 57. Swap the left-most `proj_branch_info` projection with the `delay` circuit below it, using the corresponding time-invariance law

After removing the `proj_branch_info` projections, we come to the final microarchitecture in Figure 58. This circuit still outputs exactly the same transaction values, cycle-for-cycle, as the microarchitecture in Figure 1, but is considerably less complex. We can now apply conventional techniques to verify that this microarchitecture is a valid implementation of the ISA.

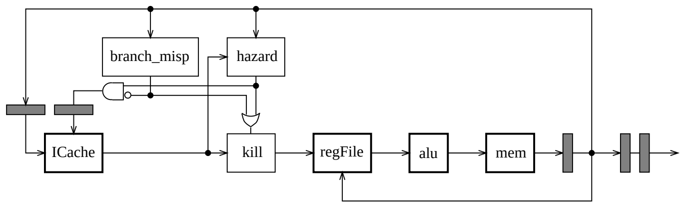

Fig. 58. Remove the `proj_branch_info` projections, using the projection insertion laws of figure 23 in reverse

References

1. Matthews, J., and Launchbury, J. Elementary microarchitecture algebra. In *CAV '99, International Conference on Computer-Aided Verification* (Trento, Italy, July 1999).