Formulation and Implementation of Speculation v1

Spring Semester 2025
05.05.2025

SUBMITTED TO

ETH Zurich
D-ITET
DYNAMO
20 Credit Project
Emmet Murphy

SUBMITTED BY

Shun Katsumi
shundroidk@gmail.com
Mobility Student at D-ITET

Contents

I. Speculation Basics

In this chapter, I begin by explaining the motivation for speculation (Section 1), followed by an overview of our current formulation using typical examples (Section 2) and relevant background on speculation and dataflow circuits (Section 3). For readers looking for a quick start, I then introduce how to try out speculation (Section 4), along with a discussion of its limitations (Section 5).

1. Introduction

(a)

```
float d=0.0f, x=100.0f; int i=0;
do {
  d = a[i] + b[i];
  c[i] = d;
  i++;
} while (d < x);
```

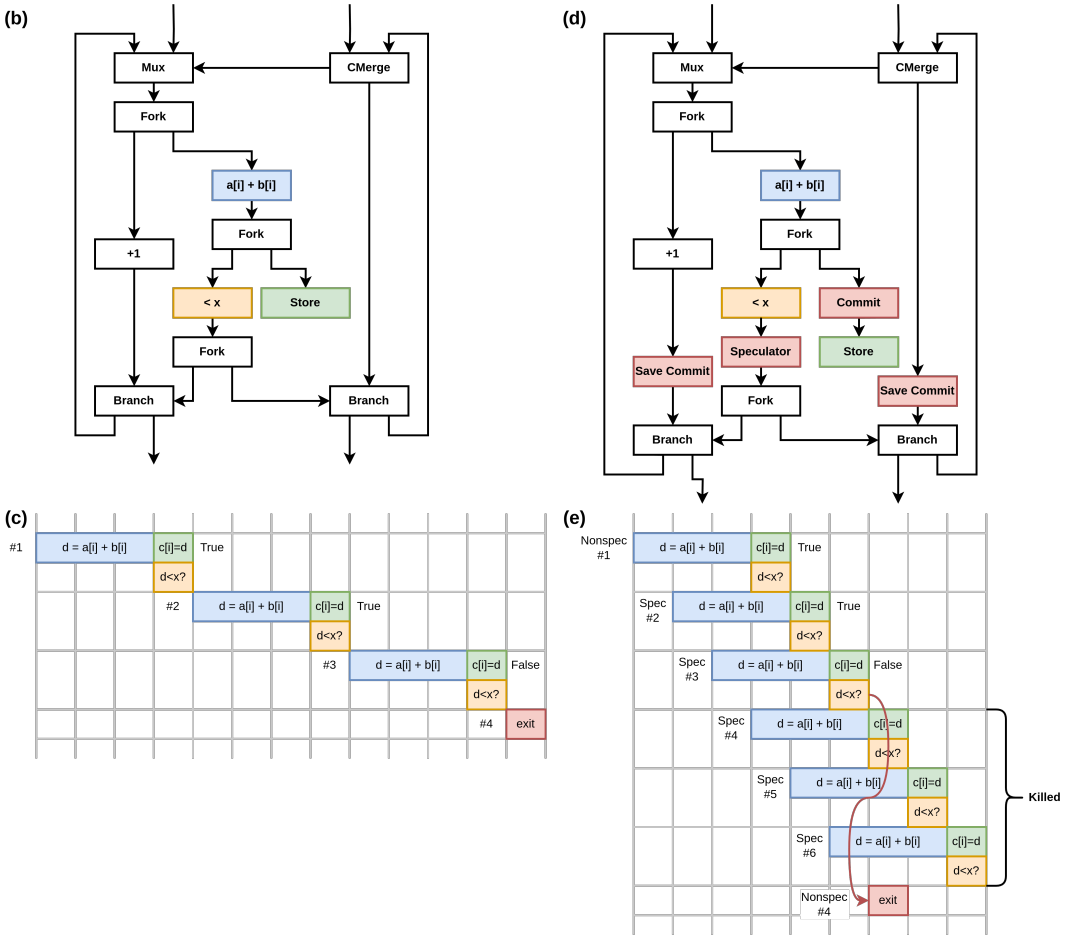


Figure 1: Speculation mitigates data dependencies by predicting values before they arrive. (a) Example program with a loop condition dependency. (b) Corresponding dataflow circuit. (c) Original schedule, where iterations cannot overlap. (d) Speculative circuit, incorporating speculator, save-commit, and commit units. (e) Updated schedule with speculation applied, achieving II = 1. Buffers are omitted from the circuit diagrams for clarity.

Dynamically scheduled circuits often suffer from performance bottlenecks due to data dependencies. Speculation helps address this by predicting values before they are available, effectively breaking these dependencies.

Within loops, speculation can remove dependencies and allow better overlap between iterations, significantly improving the initiation interval (II). Two main types of dependencies impact II:

1. **Loop condition dependency**: The program cannot advance to the next iteration without resolving the loop condition, severely impacting performance. Speculating on the loop condition, known as **condition speculation**, addresses this issue. This is the case shown in Figure 1, which will be discussed later.
2. **Inter-iteration variable dependency**: Variables dependent on prior iterations can limit throughput. Speculation on these variable values is referred to as **data speculation**.

Speculation in dynamically scheduled circuits was first introduced in Josipović et al. [1], which proposed key units like the speculator, commit, and save-commit. However, scenarios involving speculation within loops were not fully addressed in its placement algorithm.

Subsequent student projects extended this work. Haoran Zhao [2] implemented speculative units and refined the placement algorithm from Josipović et al. [1], introducing the **snapshot** concept. He validated his approach across several benchmarks, though his implementation relied on an older LLVM-based infrastructure.

Aleix Seguí Ugalde [3] reimplemented the placement algorithm within a newer MLIR-based infrastructure, successfully generating speculative circuit IRs. However, the backend for converting these IRs into hardware designs did not yet support speculation, preventing practical evaluation.

In my work, I extended the backend to support speculation, allowing for the debugging and testing of the implementation in [3]. This process uncovered and resolved several bugs and limitations in both the placement algorithm and the RTL design of speculative units. As a result, the conception of speculation has been revised, and these updates are detailed throughout this report. Finally, I re-evaluated benchmarks from Haoran's thesis, successfully reproducing most results using the updated MLIR-based flow.

2. Speculation v1

We refer to our current approach as **Speculation v1**. Its main strength lies in its generality: speculation can be applied to **any** edge within a loop where latency becomes a bottleneck—whether due to loop condition dependencies or inter-iteration variable dependencies.

While Josipović et al. [1] explored speculation beyond loops, including non-loop benchmarks, Speculation v1 focuses solely on speculation **within** loops, which is practically beneficial for improving throughput.

2.1. Condition Speculation

Let's take a closer look at the program shown in Figure 1 (a). Its corresponding dataflow circuit is shown in Figure 1 (b), and the original schedule is illustrated in Figure 1 (c). In this case, the loop condition depends on the result of a long-latency operation, d = a[i] + b[i], which limits the loop's throughput.

To enable speculation, we introduce several components as shown in Figure 1 (d), which we refer to as **speculative units**:

- **Speculator**: Sends a predicted token before the actual input arrives.
- **Save-Commit**: Restores the circuit state if the speculation turns out to be incorrect.

- **Commit**: Ensures predicted tokens do not cause side effects (e.g., memory stores, program exits) before the speculation resolves.

With these speculatiive units in place, new iterations can be launched before the actual loop condition is resolved, significantly enhancing loop pipelining. The updated schedule is shown in Figure 1 (e).

In this example, the speculator always predicts that the loop will continue, and it emits **speculative tokens** representing the loop condition. Once the actual loop condition arrives at the speculator, it compares the prediction with the actual result. If correct, execution proceeds normally. If incorrect, the speculator emits a **non-speculative token** with the actual condition. The save-commit units then resend the appropriate tokens from the corresponding iteration, effectively restoring the circuit's prior state. Any speculative tokens are invalidated by the speculator, save-commit, and commit units.

For the memory store, the commit unit before the store unit holds the speculative token until speculation is resolved. If the prediction is incorrect, it discards the token, preventing incorrect memory writes.

In this example, speculation allows the loop to achieve an initiation interval (II) of 1, with proper buffering.

This case demonstrates how speculation resolves **loop condition dependencies**.

2.2. Data Speculation

Let's now consider a different example involving **inter-iteration variable dependencies**:

```
int j = 0;
for (int i = 0; i < 10; i++) {
  if (v[j] > 10.0f)
    j = e[j];
  else
    j = e[j + 1];
}
return v[j];
```

Listing 1: Tree-based beam search.

This program performs a tree-based beam search, with the traversal path determined by the value of the current vertex `j` (`v[j]`). Assume this `if` statement is converted to a ternary expression, following Dynamatic [4]: `j = (v[j] > 10.0f) ? e[j] : e[j + 1];`.

Figure below illustrates a partial dataflow circuit corresponding to the program, showing the computation of the variable `j` and `i`:

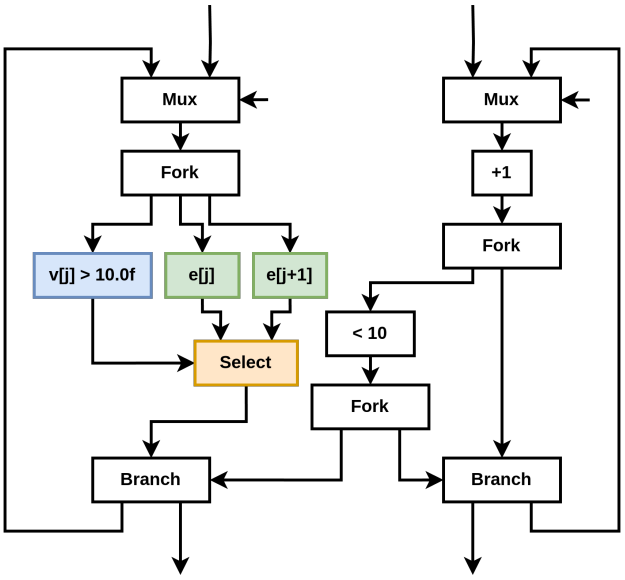


Figure 2: The partial dataflow circuit of Listing 1 showing the computation of `j` (left) and `i` (right). Buffers are omitted.

The throughput of this loop is limited by the dependency of `j` across iterations. Here is the original schedule:

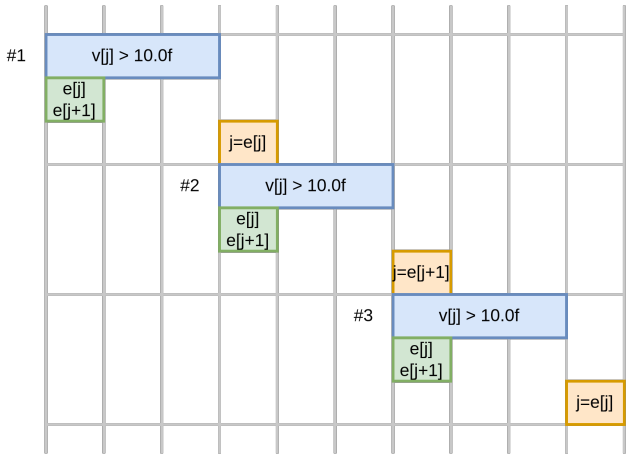


Figure 3: The schedule of Listing 1.

Suppose we know that `v[j]` is usually greater than `10.0f`. In that case, we can speculate that the next value of `j` will typically be `e[j]`. To do this, we place a speculator on the condition edge of the `select` unit, always speculating the condition as true, along with the necessary save-commit units.

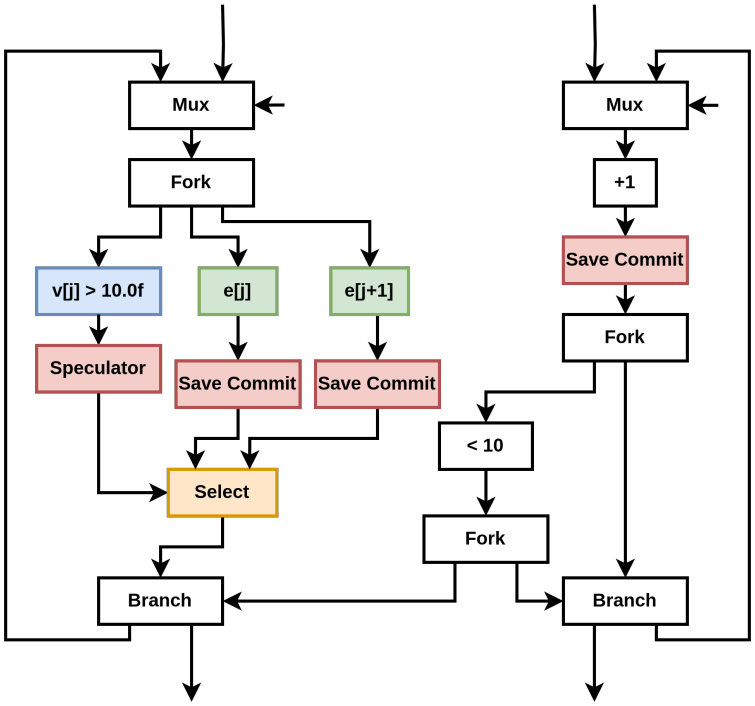


Figure 4: The partial speculative dataflow circuit of Listing 1. Buffers are omitted.

This allows the schedule to be transformed as follows, with proper buffering, achieving II = 1:

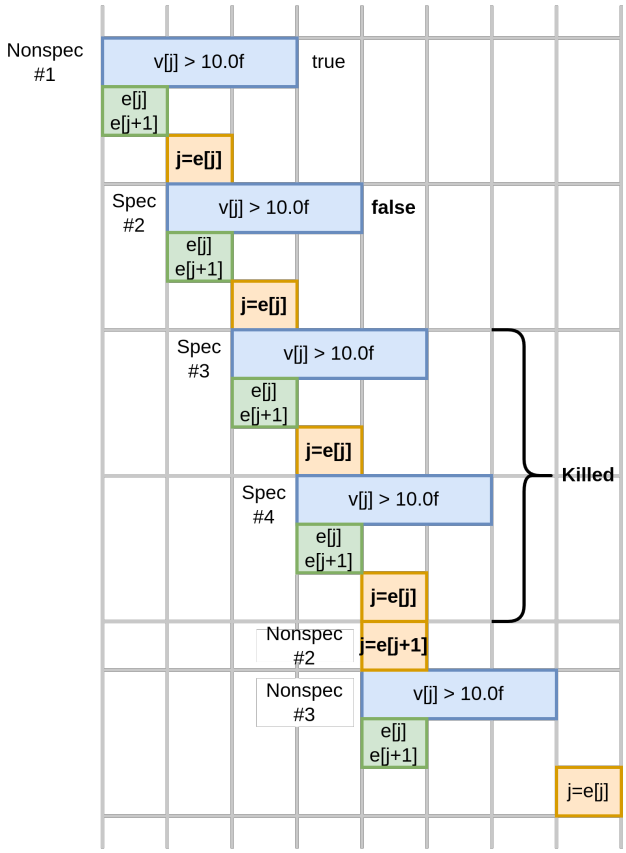


Figure 5: The schedule with speculation of Listing 1.

When the speculation is incorrect, the speculator emits the correct condition, and the save-commit units restore the proper state while invalidating incorrectly speculated tokens.

2.3. Speculative Bit

Edges that may carry speculative tokens are referred to as **speculative edges**. These are marked with a **speculative bit**, or **spec bit**, which indicates whether a token is speculative or not. This bit is primarily used by the commit unit and the speculator, which need to distinguish speculative tokens from non-speculative ones.

The figure below (Figure 6) shows a part of the circuit where the spec bit is carried. The bit is consumed by the commit unit and is not propagated to the downstream store unit.

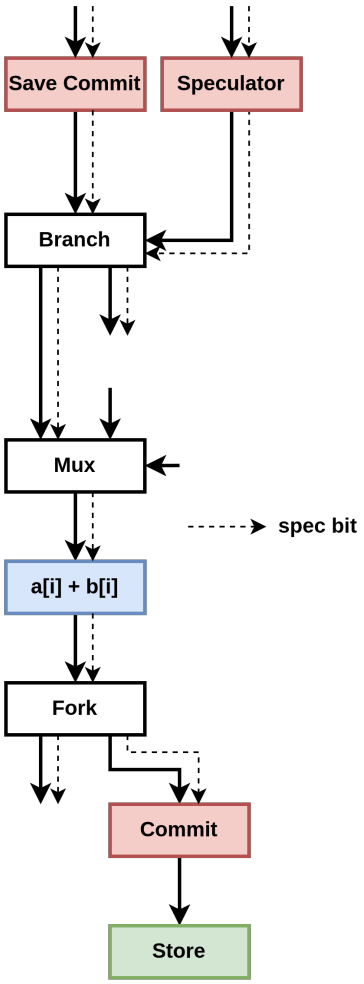


Figure 6: Part of circuit with spec bit.

Supporting the spec bit required significant changes in both the backend and the compiler's IR, which are detailed in Section 12 and Section 14.

2.4. Speculative Region

The **speculative region** of a circuit refers to the area where speculative tokens may flow. All edges within this region are **speculative edges**.

This region can be visualized as an unfolded, acyclic representation of the loop (Figure 7). It starts at the speculator and save-commit units, and ends at the speculator, save-commit units, and commit units.

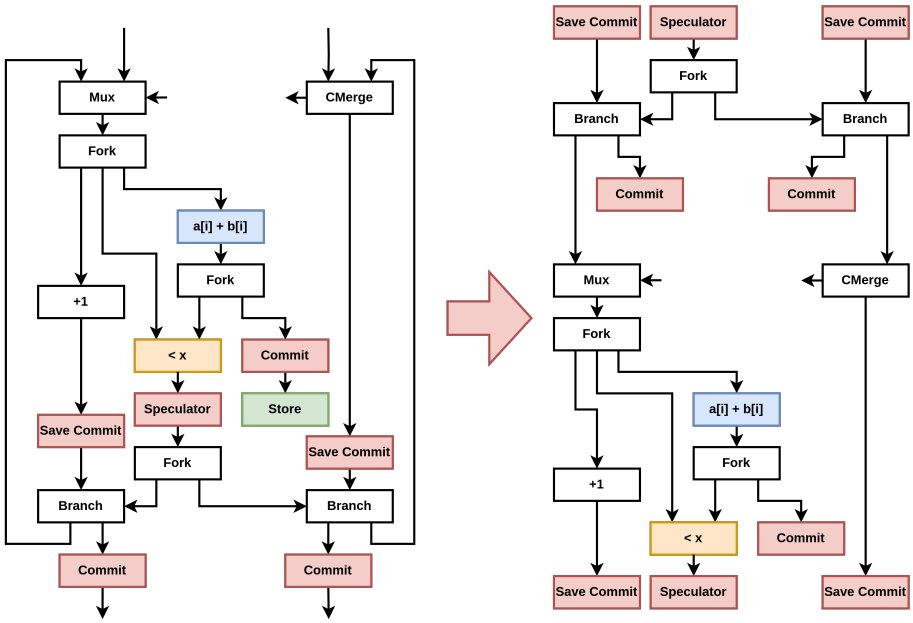


Figure 7: Speculative region example.

The speculative region is a useful concept for several reasons:

- **Ensuring validity**: A speculative circuit is ensured to be valid if a well-formed speculative region can be drawn.
- **Algorithm design**: Several speculation algorithms rely on traversing all speculative edges. Having a clear notion of the speculative region aids in designing these algorithms.

3. Background

3.1. Speculation in Computing

Speculation is a common technique in computer systems used to overcome dependencies and allow execution to continue before the actual outcome is known. When the prediction is correct, performance improves. When it is incorrect, the system restores the state and resumes execution—without worse performance than simply waiting, assuming efficient recovery.

3.1.1. Speculation in CPUs

In modern pipelined processors, speculation is essential for high performance. The most well-known example is **branch prediction**, which uses highly trained predictors to guess whether a branch will be taken. Other examples include memory dependence prediction and cache level prediction.

A notable distinction is that speculation mechanisms in CPUs are typically specialized for each task. In contrast, Speculation v1 is generalized—it can be applied to any edge in a dataflow circuit where latency is a concern.

It's also worth noting that speculation has been a source of various security vulnerabilities in modern CPUs (e.g., Spectre [5], Meltdown [6]).

3.1.2. Speculation in Static HLS

Speculation might appear to be a feature exclusive to dynamically scheduled circuits. This intuition is understandable, as dynamic schedules naturally allow speculative execution.

However, there are research efforts aiming to bring speculation into **static** high-level synthesis (HLS) as well.

For example, Gorius et al. [7] proposed a method to enable speculation in unmodified commercial HLS toolchains through source-to-source transformation. This approach primarily targets **data speculation**, and showed that speculation can be effective, especially when the speculated path is usually taken and leads to shorter latency.

The key insight enabling speculation in static HLS is that the schedule remains **fixed** within each iteration. Speculation chooses one path deterministically, allowing the circuit to be statically scheduled even if the original control flow was conditional.

Still, speculation in dynamic HLS is generally more flexible. For instance, it allows conditional logic before commit units, and benefits from various optimizations that take advantage of dynamic scheduling.

3.2. Dataflow Circuit

Dynamatic [4], the compiler used in this work, targets dynamically scheduled HLS and generates **dataflow circuits**. These circuits—also known as **elastic circuits**—are composed of latency-insensitive components such as adders, forks, branches, and more. Each component communicates via a handshake protocol, enabling bottom-up, locally coordinated dynamic scheduling.

Speculative units are also implemented to follow these same latency-insensitive protocols.

3.2.1. In-Order Control Network

A notable feature of Dynamatic's dataflow circuit construction is its use of an **in-order control network**. This network prevents tokens from arriving out-of-order at merge points between basic blocks (Figure 8). It propagates a **control token**—a token that carries only handshake signals, not data—to indicate which basic block is executing.

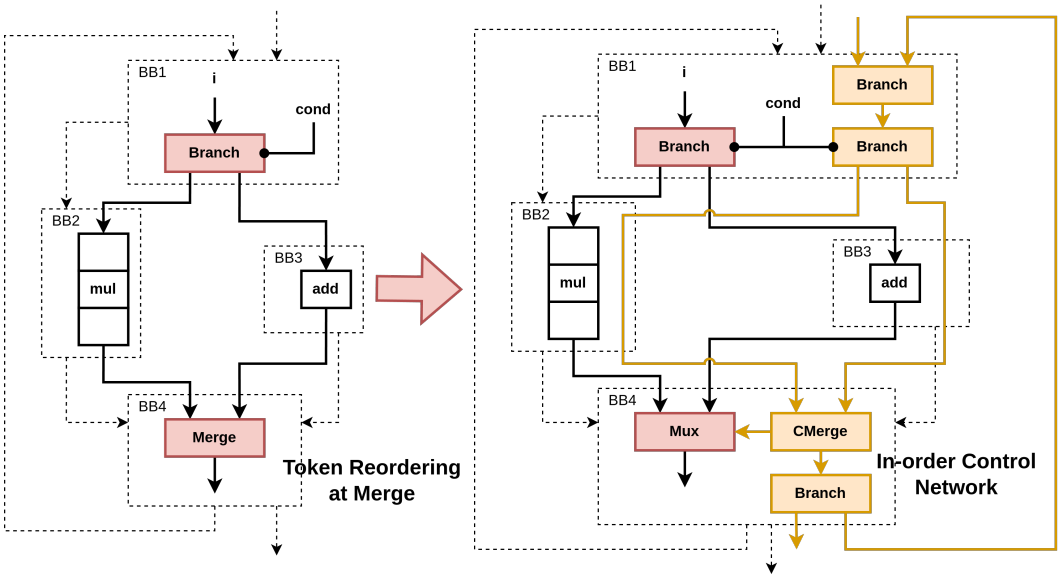


Figure 8: In-order control network to prevent token reordering. The control token indicates the currently executing basic block. Adapted from [8].

Beyond ensuring in-order execution, the control token is also used to track the execution of basic blocks. Speculation relies on this control token to precisely determine when to launch

a new speculation. Specifically, the **trigger** signal to the speculator is derived from the control network, allowing it to initiate speculation whenever the corresponding basic block is executed. This helps prevent token mismatches and is introduced by Haoran Zhao's work [2] (see Section 6.4.7).

The in-order control network maintains exactly one control token at a time to consistently indicate the currently executing basic block. However, with speculation, a control token needs to be resent upon detecting a misspeculation. This can lead to conflicts, so it must be handled with great care (see Section 7.3.1).

3.3. MLIR

MLIR (Multi-Level Intermediate Representation) supports a wide variety of dialects, including custom ones, in contrast to LLVM's single-layer IR design. Dialects in MLIR can define operations (e.g., `add`), types (e.g., `i32`), attributes (e.g., predicates for comparisons), and more.

For speculation, we primarily leverage MLIR's **custom type system** to attach spec bits to values in the circuit. For example, a 32-bit channel carrying a value with a spec bit might be typed as `<i32, [spec: i1]>`. The type system ensures that these spec bits are neither suddenly lost nor incorrectly introduced within the circuit (see Section 12.2.1).

Beyond types, MLIR's verification framework allows us to define consistency rules across the IR. In systems with many concepts and passes, where no one person can track everything, we believe encoding design intentions through verification is a clear and robust way to ensure correctness.

The technical details of how spec bits are handled within MLIR are discussed in Section 12.

4. How to Use Speculation

Speculation is currently available on the main branch of Dynamatic [4].

The speculation integration tests are benchmarks designed to evaluate the effectiveness of speculation. These tests are based on Haoran Zhao's master's thesis [2].

Unlike other integration tests, these require manual modifications to the programs and IR to ensure that speculation is applied correctly.

This section explains how to run the speculation integration tests and outlines the necessary manual modifications.

This guide assumes you have already cloned the Dynamatic repository and built the project. For setup details, please refer to the documentation in the repository [4].

4.1. Running the Tests

There are eight speculation integration tests in the `integration-test` folder:

- `single_loop`
- `loop_path`
- `subdiag`
- `subdiag_fast`
- `fixed`
- `sparse`
- `nested_loop`
- `if_convert` (data speculation)

These are adopted from Haoran Zhao's master's thesis [2]. The `newton` benchmark is excluded because it contains branches within the loop, which is not supported in the current implementation (see Section 5.1).

Since these tests require manual modifications and a custom compilation flow, we have provided a ready-to-run script. You can execute the speculation integration tests (covering compilation, HDL generation, and simulation) with a single command:

```
$ python3 tools/integration/run_spec_integration.py single_loop
```

Requirement: Python **3.12** or later is needed to run the script.

You can run a test without speculation using the custom compilation flow:

```
$ python3 tools/integration/run_spec_integration.py single_loop --disable-
spec
```

To visualize and confirm the initiation interval, you can simply use the Dynamatic interactive shell:

```
$ ./bin/dynamatic
> set-src integration-tests/single_loop/single_loop.c
> visualize
```

4.2. Manual Work Is Required

Running speculation on your own code still requires some significant manual steps:

- **CFG Modification**: Due to limitations in our frontend (currently under development) and the requirement for a single basic block loop for speculation (discussed in Section 5.1).
- **Specifying Speculator Position**: Manually define the position of the speculator.
- **Specifying FIFO Depths of Speculator and Save-Commits**: Manually set the FIFO depths for these units. Details in Section 13.3.
- **Buffer Placement to Prevent Deadlocks and Gain Performance Improvement**: Details in Section 13.1 and Section 13.2.

The actual steps are highly specific to the codebase. For detailed instructions, please refer to the document in the repository: `docs/Speculation/IntegrationTests.md`.

4.3. Custom Compilation Flow for Speculation

The full details of the custom compilation flow can be found in the Python script: `tools/integration/run_spec_integration.py`. Below is a summary of the process:

- Compilation begins with the `cf` dialect, as modifications to the CFG are required.
- The speculation pass (`HandshakeSpeculation`) runs **after** the usual buffer placement pass.
- A **custom buffer placement pass** follows the speculation pass, just before the `HandshakeToHW` pass, ensuring the necessary buffers for speculation are in place.
- A **Python-based, generation-oriented beta backend** is used, which supports spec bits.

5. Limitations

The current implementation of speculation comes with several limitations.

5.1. Single Basic Block Loops

At present, speculation is only supported within loops that consist of a single basic block. This restriction stems from the current design of the placement algorithm—particularly for save-commit units—where a single-block loop greatly simplifies the logic.

That said, the algorithm could likely be extended to handle multi-block loops with moderate effort.

This restriction also implies that we currently only support speculation in **innermost** loops, which is typically where speculation yields the most benefit.

5.1.1. Manual CFG Rewriting

Manual CFG rewriting is required for two reasons:

1. The current frontend generates overly complex CFGs for irregular loop kernels.
2. Speculation only supports single-basic-block loops.

The first issue is actively being addressed. However, even after improvements to the frontend, some manual CFG transformations may still be necessary. This is because the transformations described in the following subsections are somewhat unconventional and may not be supported.

5.1.1.1. Converting `while` Loops to `do-while` Loops

If a loop is guaranteed to execute at least once, you can convert it into a `do-while` structure. This helps eliminate the need for a separate basic block that handles the initial condition, making the loop structure simpler and compatible with speculation.

Before:

```
while (cond) {
  // Executed at least once
}
```

After:

```
do {
  // Executed at least once
} while (cond);
```

5.1.1.2. Merge Tail `break` Statement

We eliminate the `break` statement by incorporating its condition into the loop's termination condition. This transformation is also applied to `for` loops when applicable.

Before:

```
for (int i = 0; i < N; i++) {
  // Body
  if (cond) break;
}
```

After:

```
int i = 0;
bool break_flag = false;
do {
  // Body
  i++;
```

```
  break_flag = cond;
} while (i < N && !break_flag);
```

5.2. Nested Loop: II=1

As discussed later in Section 7.3.1 and Section 7.3.2, commit units must be placed at the convergence point of basic blocks to prevent token reordering. This requirement can limit the effectiveness of speculation, particularly in nested loops. However, if the innermost loop can achieve an initiation interval (II) of 1, the commit units can be safely removed (see Section 7.3.2).

Currently, our placement algorithm does not analyze whether a loop can achieve II = 1. Instead, it assumes this as a limitation and always removes the commit unit on the backward edge of the innermost loop.

5.3. Load Store Queue

Our speculation methodology does not yet support loops whose memory operations are connected to load-store queues (LSQs). This limitation arises because LSQ allocation might occur in iterations that are later determined to be misspeculated, and we currently lack a mechanism to handle this scenario. As discussed in Josipović et al. [1], implementing speculative loads and speculative LSQ is necessary to support such cases.

II. Speculative Units

In this chapter, I explain the details of speculative units: first, their design (Section 6); second, how they are placed within the circuit (Section 7); and finally, how they are connected to the speculator to function correctly (Section 8).

6. Speculative Units Design

This section describes the design of each speculative unit in detail.

6.1. Speculating Branch

First, I'll explain the **speculating branch**, which is used internally by other speculative units or to connect with the speculator. It differs from regular branches in one key aspect: it branches the data token based on the **spec bit** of the condition token (Figure 9). The data payload of the condition token is ignored.

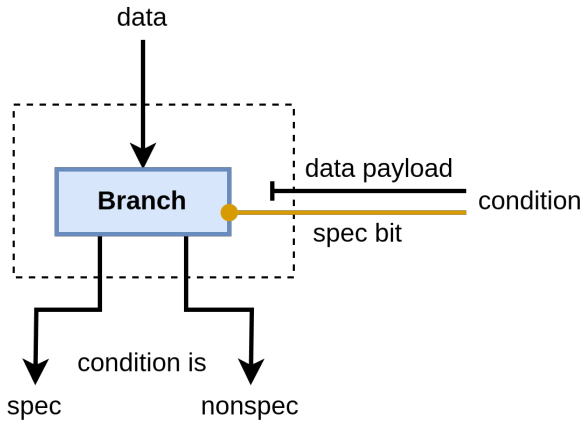


Figure 9: Internal structure of the speculating branch.

In some cases, the data and condition come from the same upstream unit. When this happens, the data channel is branched based on the spec bit of the **data token itself** (Figure 10).

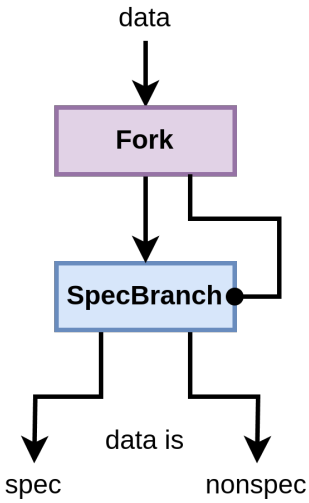


Figure 10: Speculating branch connected to the same upstream.

I occasionally shorten the name of the speculating branch to **spec branch**, especially in figures. However, it is **completely different** from the "speculative branch" in Josipović et al. [1], which refers to a branch speculator unit.

6.2. Commit

The commit unit delays the propagation of speculative tokens until the corresponding speculation resolves, to prevent misspeculated tokens from causing side effects (e.g., memory store, program exit).

Whether the speculation resolved or not is noticed to commit units via control signal from the speculator.

Here are inputs and output of commit units:

Inputs:

- **`dataIn`**: the speculative or non-speculative data token.
- **`ctrl`**: control signal from the speculator indicating whether the speculative token should be passed (**`PASS`**) or discarded (**`KILL`**).

Output:

- **`dataOut`**: emits the forwarded data token.

6.2.1. Internals of the Commit Unit

The internal structure is shown below:

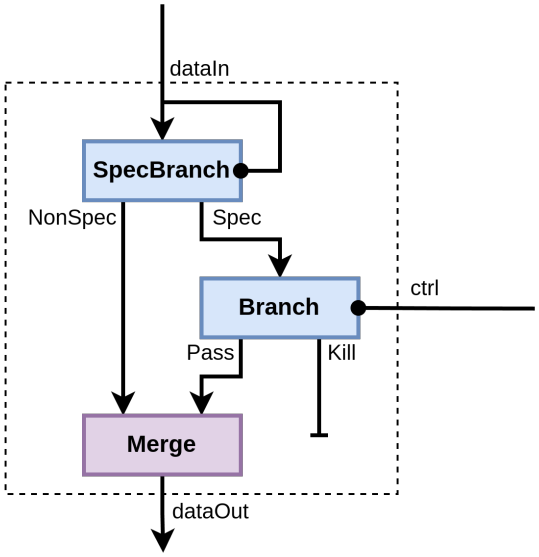


Figure 11: Internal structure of the commit unit.

The unit consists of three elastic components:

1. A **speculating branch** (see Section 6.1), which branches `dataIn` based on its spec bit:
   - **Non-speculative** tokens are forwarded directly to the merge, and then to `dataOut`.
   - **Speculative** tokens are passed to the second, regular branch.
2. The second branch is conditioned on the `ctrl` signal from the speculator:
   - If it is `PASS`, the token is allowed through.
   - If it is `KILL`, the token is discarded.

3. A merge, which joins the paths of non-speculative tokens and speculative tokens that have been permitted to proceed.

6.3. Save-Commit

Save-commit units are responsible for restoring the loop state by storing tokens from unresolved iterations and resending them when speculation is determined to be false.

The inputs and outputs of the save-commit unit are as follows:

Inputs:

- **`dataIn`**: Any channel (including control channels) that the save-commit unit manages for token restoration.
- **`ctrl`**: The control signal from the speculator that directs the unit's behavior.

Output:

- **`dataOut`**: Emits the token from the `dataIn` channel with the restoration functionality.

6.3.1. Control Signals

The control signals from the speculator govern the behavior of the save-commit unit, determining whether tokens are sent out or stored in memory.

The following control signals are sent from the speculator:

- **`PASS`**: Sends the token received in `dataIn` as a **speculative token** (setting the spec bit to 1) and stores it in internal memory.
- **`NO_CMP`**: Sends the token received in `dataIn` as a **non-speculative token** (setting the spec bit to 0) without storing it in memory.
- **`KILL`**: Removes the oldest token from internal memory.
- **`PASS_KILL`**: A combination of `PASS` and `KILL`. Sends the token received in `dataIn` as a **speculative token**, stores it in memory, and removes the oldest token from memory.
- **`RESEND`**: Sends the oldest token in internal memory as a **non-speculative token** and removes it from memory.

6.3.2. Save-Commit as Co-Speculator

The term **save-commit** originates from earlier understanding that combined the previous save unit (used to restore tokens) and the commit unit, as introduced in Josipović et al. [1]. However, Haoran Zhao's work [2] significantly revised its internal structure to meet updated requirements. We've further refined its functionality, and now consider the save-commit unit as a "co-speculator" rather than a combination of save and commit units.

The key feature of the save-commit unit is its synchronization with the speculator. When the speculator sends out a speculative token, the save-commit unit will also send out a speculative token (upon receiving `PASS` or `PASS_KILL`), even if it received a non-speculative token. Similarly, when the speculator sends out a non-speculative token (via `RESEND`), the save-commit unit sends out a non-speculative token. This synchronization is not inherently part of the save or commit units themselves.

In contrast, there are key differences between the save-commit unit and the speculator:

- **No speculation**: When the speculator sends a `PASS` or `PASS_KILL` signal, the save-commit unit simply passes the **received** token, setting the `spec` bit to 1. It does not perform any speculation.

- **Restoration**: When the speculator sends a `RESEND` signal, the save-commit unit emits the **oldest token** from its internal memory, which is guaranteed to be correct, and sets the `spec` bit to 0.

6.3.3. Internals of Save-Commit

Save-commit unit is built around a ring buffer that holds tokens, with three pointers managing its behavior:

- **`Head` (H):** Points to the oldest valid token—i.e., the most recently confirmed correct token. This is the token emitted when receiving a `RESEND`.
- **`Current` (C):** Points to the oldest token not yet sent out.
- **`Tail` (T):** Points to the next available slot for incoming tokens.

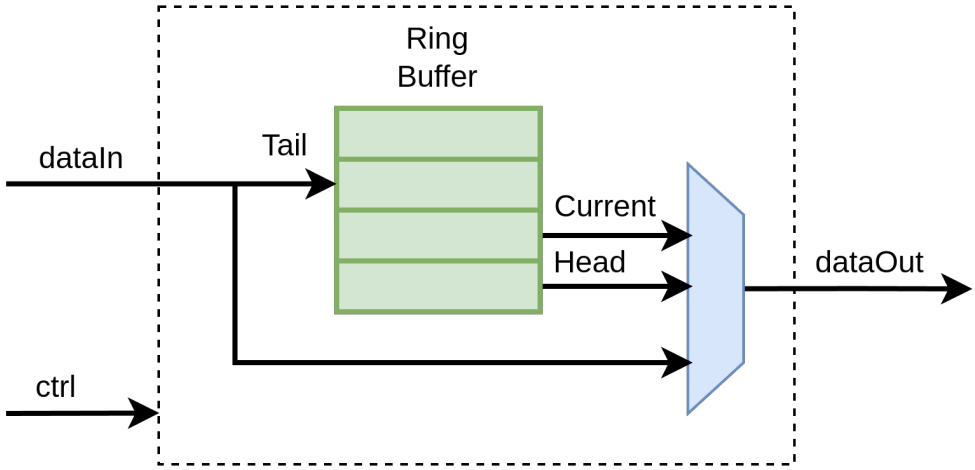


Figure 12: Internal structure of the save-commit unit. Adapted from [2].

Save-commit units accept new tokens continuously unless the buffer is full. However, tokens are only emitted in response to appropriate control signals from the speculator.

6.3.3.1. Pointer Behavior

Initially, `Head` = `Current` = `Tail` holds. Thereafter, we maintain `Head` $\leq$ `Current` $\leq$ `Tail`. This relation is not automatic and must be enforced explicitly (see next subsection).

- **`Head`** is incremented when the oldest token is either:
  - **Killed** (on receiving `KILL` or `PASS_KILL`)
  - Emitted as **non-speculative** (on receiving `RESEND` or `NO_CMP`)
- **`Current`** is incremented when:
  - A **speculative token** is emitted (on receiving `PASS` or `PASS_KILL`), or
  - In special cases where it's necessary to preserve `Head` $\leq$ `Current` — specifically on `NO_CMP`, or occasionally on `KILL`.
- **`Tail`** is incremented on every new token received.

6.3.3.2. Guaranteeing `Head` $\leq$ `Current` $\leq$ `Tail`

To ensure **`Current`** $\leq$ **`Tail`**, save-commit units only accept control signals when `Current` $<$ `Tail` — a condition that is eventually met in every iteration.

To ensure **`Head`** $\leq$ **`Current`**, we make an exception by incrementing both **`Head`** and **`Current`** when receiving `NO_CMP` or in some cases `KILL`.

- **Why `NO_CMP` moves both**: This signal is sent by the speculator when no speculation has been made, or when all speculation is resolved (i.e., `Head` = `Current`). If only `Head` were incremented, we would end up with `Head` > `Current`, breaking the invariant.
- **Why `KILL` sometimes moves both**: When misspeculation is detected, the speculator sends `KILL` signals to clear all tokens. In most cases, on the final `KILL`, `Head` = `Current` holds because the token to be removed hasn't been sent out (see example in the next subsection). If only `Head` were incremented, the `Head` ≤ `Current` invariant would be violated.
  - However, this is not always the case. If a speculative token has already exited the loop before misspeculation is realized, then the last token has been sent out. In this case, on the final `KILL`, it is sufficient to increment only `Head` (another example is shown in the next subsection).
  - Therefore, on receiving a `KILL`, the save-commit unit increments `Current` **only when `Head` = `Current`**.

6.3.3.3. Running Examples

This section illustrates how save-commit works in a concrete example. In this scenario, four speculations are made: the first two are correct, and the third is incorrect.

The sequence of control tokens received by the save-commit unit is: `PASS`, `PASS`, `KILL`, `PASS`, `PASS_KILL`, `RESEND`, `KILL`, `KILL`.

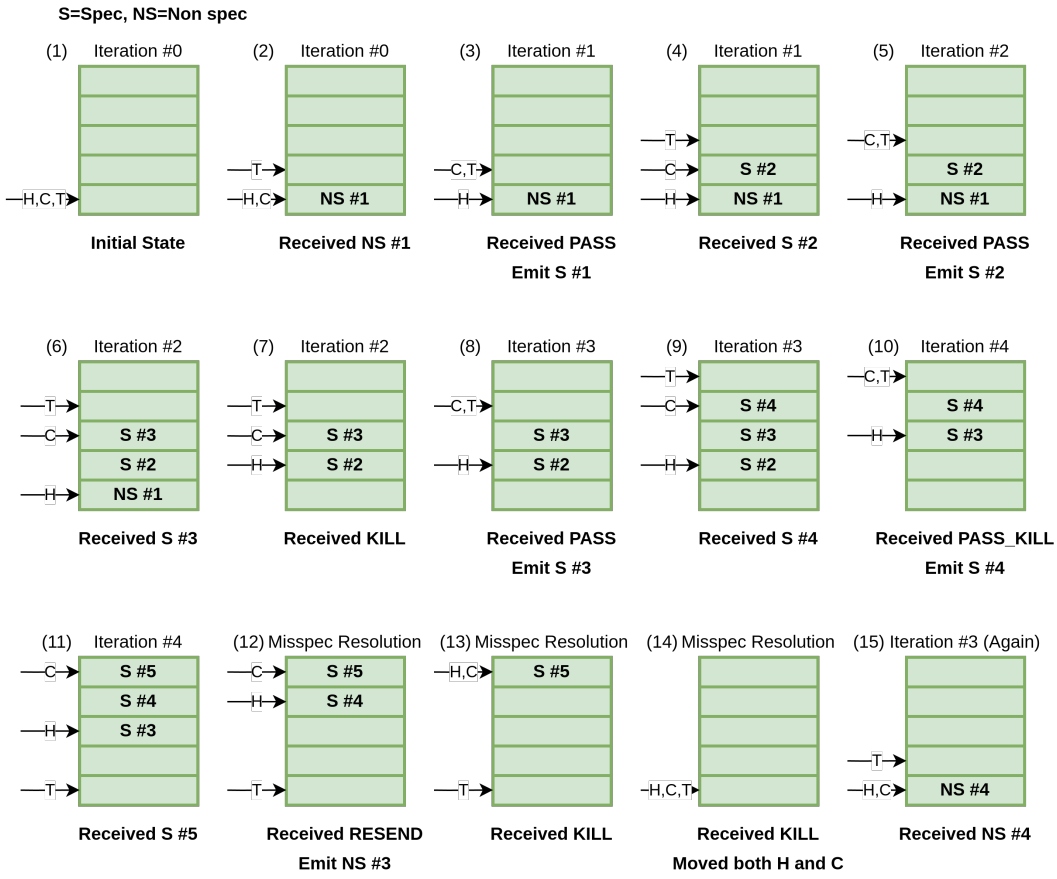


Figure 13: Running example of save-commit behavior.

1. Initially, `Head` = `Current` = `Tail`.
2. Save-commit receives a non-speculative token in Iteration #0.

3. Upon receiving the first `PASS`, it emits the token as a **speculative** token, marking the start of Iteration #1.
4. The speculative token from Iteration #1 (S#1) traverses the loop and returns to save-commit as S#2.
5. On receiving another `PASS`, S#2 is sent out, and Iteration #2 begins.
6. S#3 is received.
7. The first speculation is confirmed correct, and the speculator sends a `KILL`. Note: (6) and (7) may occur in either order.
8. Repeat (3).
9. Repeat (4).
10. When speculation is confirmed correct and a new one begins simultaneously, `PASS_KILL` is received — save-commit emits a new speculative token and kills the oldest one.
11. S#5 is received. At this point, the first two speculations are resolved; the other two are still pending.
12. The third speculation fails, and a `RESEND` is received. Save-commit sends out token #3 as **non-speculative** (NS#3) and increments `Head`.
13. A `KILL` is received; only `Head` is incremented.
14. Another `KILL` is received; this time, **both `Head` and `Current`** are incremented.
15. NS#3 traverses the loop and returns as NS#4. Note: (15) may occur before (13) or (14).

In this example, all tokens sent out by save-commit eventually return (e.g., S#1 returns as S#2, NS#3 as NS#4). However, it's also possible for a token to exit the loop — either speculatively or non-speculatively — and never return.

For instance, suppose token S#5 exits the loop. The execution path from step (11) changes:

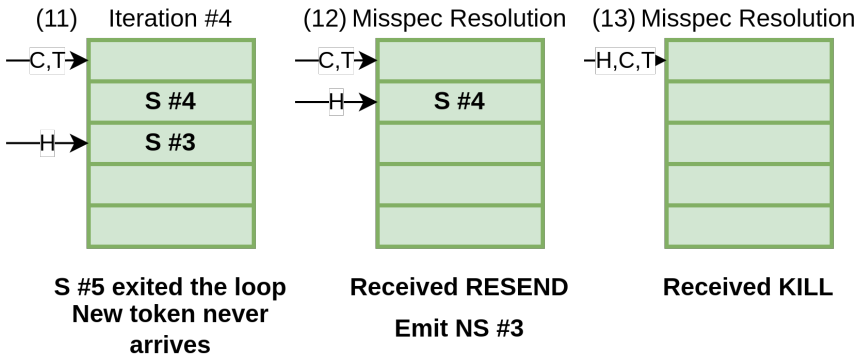


Figure 14: Modified example when S#5 exits the loop.

In this case, misspeculation is realized only after the token has exited, at step (12). At step (13), upon receiving the last `KILL`, only `Head` is incremented. Whether the resent token NS#3 reaches save-commit or not depends on the non-speculative loop condition — both outcomes are valid.

6.4. Speculator

The speculator is the central unit that both initiates speculation and coordinates the behavior of other speculative units. Our implementation builds upon Haoran Zhao's design [2], with key updates.

Inputs:

- `dataIn`: The channel carrying the actual token to be compared against the predicted value after speculation.
- `trigger`: A control signal that initiates the next speculation. Renamed from `enable` for clarity.

Outputs:

- `dataOut`: The channel carrying either the predicted or actual token, depending on the speculation state.
- Internal control signals that coordinate with other internal units.

Internally, the speculator is a set of *semi*-elastic components as shown in Figure 15.

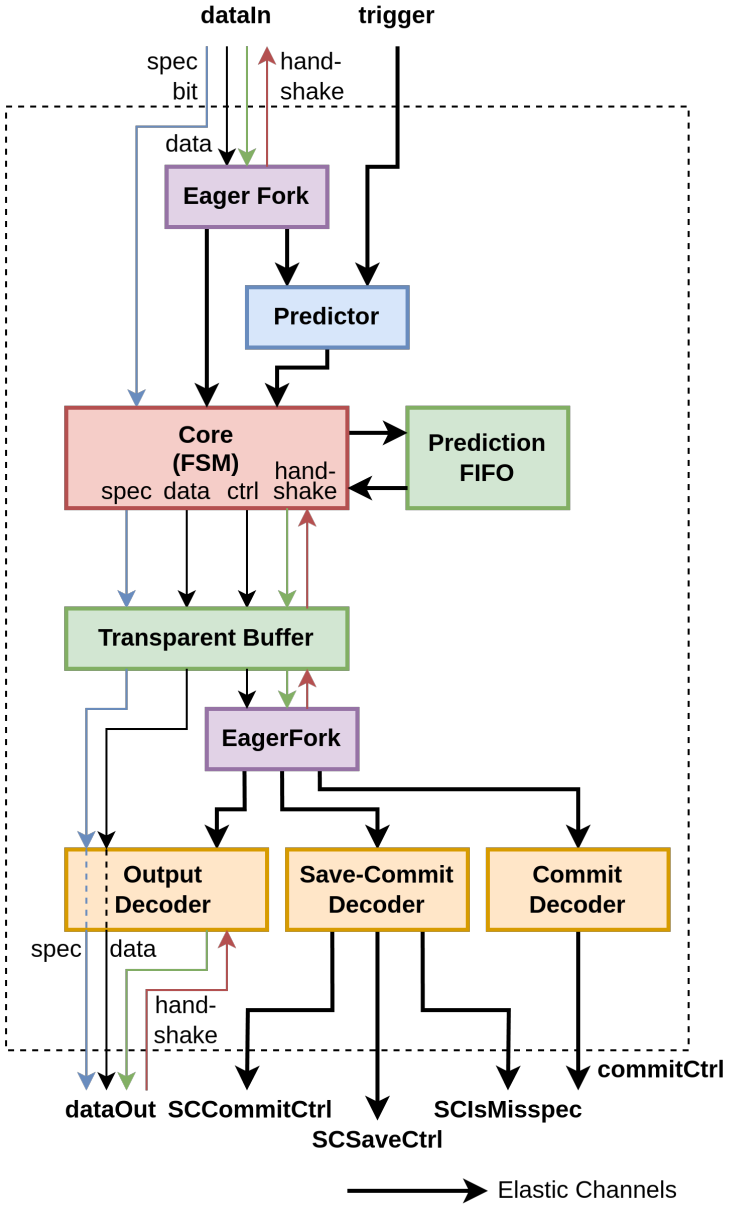


Figure 15: Internal structure of the speculator.

Internal Components:

- **Predictor:** Generates a predicted value for each incoming trigger token. By default, it uses the most recent `dataIn` value, but the prediction strategy can be customized as needed.
- **Prediction FIFO:** Stores the predicted values in the order they were generated, enabling comparison with actual data when it arrives.
- **Core:** A finite state machine that controls the speculation process. It functions as a Mealy machine—its outputs (data, control signal, and handshake signals) are computed combinationally based on both inputs and internal state. The Core issues the **internal control signal**, interacts with the Prediction FIFO, and generates the data payload and spec bit of `dataOut`. Details are discussed in the next section.
- **Transparent Buffer:** Prevents handshake protocol violations. Discussed later in Section 6.4.5.
- **Decoders:** Interpret internal control signals generated by the Core and issue appropriate signals to other speculation units. Each decoder responds to specific internal control signals. Details are discussed later in Section 6.4.4.

6.4.1. Internal States of Core

The Core has three states: `IDLE`, `KILL`, and `KILL_ONLY_DATA`. Figure 16 below illustrates the states and transitions of the Core.

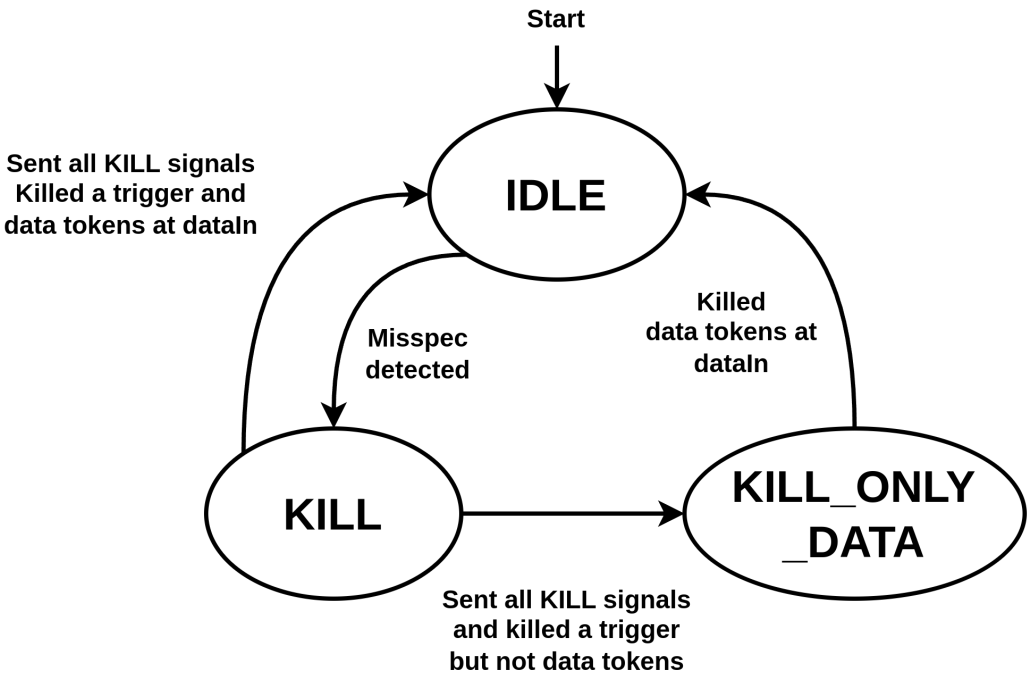


Figure 16: States and transitions of Speculator Core.

- **`IDLE`**: The default state, where most speculation functionality is active.
- **`KILL`**: This state is triggered when misspeculation is detected. The speculator compares the received `dataIn` to the oldest item in the Prediction FIFO. If they don't match, it enters the `KILL` state. Here, the speculator invalidates both the speculative trigger and data tokens received after misspeculation. The Prediction FIFO is popped, and a `KILL` control signal is issued for each item in the FIFO to kill matching speculative tokens across other speculation units. Once the FIFO becomes empty and both a trigger and data tokens are killed, the speculator returns to the `IDLE` state. If the trigger is killed but not the data tokens, the speculator transitions to the `KILL_ONLY_DATA` state.

- **KILL_ONLY_DATA**: In this state, the speculator waits for the misspeculated data tokens to be killed. An important feature of this state is that the speculator can make new predictions, as the misspeculated trigger has been invalidated. New speculations are safe because the actual data token for the new speculation is guaranteed to arrive after the misspeculated data tokens. This state improves throughput by allowing new speculation while still processing pending misspeculated data tokens.

6.4.2. Invalidation of Misspeculated Trigger and Data Tokens

When misspeculation is realized, in addition to clearing its internal FIFO, the speculator also kills incoming misspeculated trigger and data tokens.

For the trigger, the speculator kills at most one. If the loop has already exited, the control token doesn't reach the speculator and is instead killed by a downstream commit unit.

For data tokens, the speculator may discard multiple tokens—these originate from iterations that were speculatively executed after the misspeculated one.

The arrival of a **non-speculative** data or trigger token—sent upon a `RESEND` by the speculator or a save-commit unit—indicates that all earlier misspeculated data tokens or trigger have been successfully killed, assuming in-order token delivery. Then the speculator can safely return to the `IDLE` state.

6.4.3. Internal Control Signals

Each internal control signal represents an **action** taken by the Core, and is interpreted by other speculative units to coordinate with the speculator. The meaning of each signal depends on the receiving unit.

The following describes how each internal signal reflects an action of the Core:

- **SPEC**: The speculator makes a new speculation and emits a speculative token.
- **NO_CMP**: When the speculator has not yet made a speculation and receives actual data token, it emits the actual data token (instead of the predicted one) upon receiving a trigger token.
- **CMP_CORRECT**: The speculator compares the actual data with its oldest prediction (from the FIFO). If they match, it emits this signal.
- **CORRECT_SPEC**: A combination of `SPEC` and `CMP_CORRECT` that enhances throughput. The speculator receives a trigger and makes a new speculation in addition to confirming the correctness of the previous prediction.
- **RESEND**: When the actual data differs from the oldest prediction in the FIFO, the speculator emits the actual data it just received.
- **KILL**: In the `KILL` state, when the speculator pops one item from the Prediction FIFO, it emits one `KILL` signal.

All signals except for `KILL` are emitted during the `IDLE` state. The following table details the behavior of the Core during the `IDLE` state.

Inputs				Meaning	Outputs			
Prediction (Trigger)	**DataIn**	**FIFO**	**DataIn == FIFOHead**		**Internal Signal**	**DataOut**	**FIFO**	**Next State**
Valid	Invalid	Not Full	Don't care	Speculate	`SPEC`	Prediction	Push	`IDLE`
Valid	Valid	Empty	Don't care	Forward real data	`NO_CMP`	Actual	-	`IDLE`
Invalid	Valid	Not Empty	True	Correct speculation	`CMP_CORRECT`	Invalid	Pop	`IDLE`
Valid	Valid	Full	True					
Valid	Valid	Not Empty & Not Full	True	Correct spec. + speculate	`CORRECT_SPEC`	Prediction	Push & Pop	`IDLE`
Don't care	Valid	Not empty	False	Misspeculation + forward real data	`RESEND`	Actual	-	`KILL`

Table 1: Behavior of the Core during the `IDLE` state, detailing input-output interactions.

6.4.4. Decoder Behavior

The decoders translate internal control signals into forms usable by other speculative units. Table 2 shows the behavior of each decoder. "-" in the table indicates that the decoder ignores the corresponding control token.

Core Control	Commit	Save-Commit			Output		
		SCSave	**SCCommit**	**Misspec**	**Data**	**Spec Bit**	**Valid**
`SPEC`	-	`PASS`	-	-	Prediction	Spec	True
`NO_CMP`	-	`NO_CMP`	-	-	Actual	NonSpec	True
`CMP_CORRECT`	`PASS`	`KILL`	-	False	-	-	False
`CORRECT_SPEC`	`PASS`	`PASS_KILL`	-	False	Prediction	Spec	True
`RESEND`	-	`RESEND`	-	-	Actual	NonSpec	True
`KILL`	`KILL`	-	`KILL`	True	-	-	False

Table 2: Decoder behavior.

For commit units, `PASS` signals are sent when speculation succeeds, and `KILL` signals are issued according to the speculator's internal FIFO clearance.

For save-commit units, the control signals `SCSave`, `SCCommit`, and `Misspec` are involved (their roles are explained in detail in Section 8.1). All internal signals are translated into these controls. Notably, a `KILL` signal resulting from `CMP_CORRECT` is sent via `SCSave`, whereas a `KILL` signal directly from the Core is sent via `SCCommit`. This distinction helps prevent token mismatches and is further discussed in Section 8.1.

The output decoder is used directly by the speculator. It forwards the `dataOut` and spec bit from the Core (via the transparent buffer), and generates the handshake logic for `dataOut` based on the internal control signal. Specifically, it detects control signals that trigger data emission—such as `SPEC` or `NO_CMP`—and sets the valid bit of `dataOut` accordingly.

6.4.5. Transparent Buffer Prevents Handshake Violation

The speculator contains an internal transparent buffer that holds the data output, spec bit, and internal control signals from the speculator Core (Figure 15). This buffer is essential to prevent handshake violations that may occur when the speculator's outputs are stalling.

The Core can update its outputs even before they have been accepted. For example, suppose the inputs satisfy the condition to emit a `SPEC` signal (see Table 1), and the speculator FIFO is empty. If `dataIn` arrives in a later cycle before the `SPEC` is accepted, the Core switches its output from `SPEC` to `NO_CMP`. Similarly, a transition from `CMP_CORRECT` to `CORRECT_SPEC` may occur if the FIFO is not full and a trigger token is later received.

These changes can result in a handshake violation, where the data payload is modified while the valid bit remains high. However, this is not just a protocol issue—it leads to **token mismatches**. For instance, in circuits with multiple save-commit units, the `SCSaveCtrl` output first connects to an **eager fork**, which then distributes the control signals to the save-commit units (Figure 17, top). Eager forks allow one output to complete handshaking while others do not, for performance reasons. In such cases, the eager fork does not assert the ready signal of its input until all outputs are accepted, meaning the handshake is still incomplete from the speculator's perspective.

This setup relies on *data persistency*—the protocol guarantee that the payload remains stable until all output handshakes are complete. However, in the speculator's case, if the control signal changes mid-handshake, the eager fork may deliver inconsistent control values to different save-commit units (Figure 17, top).

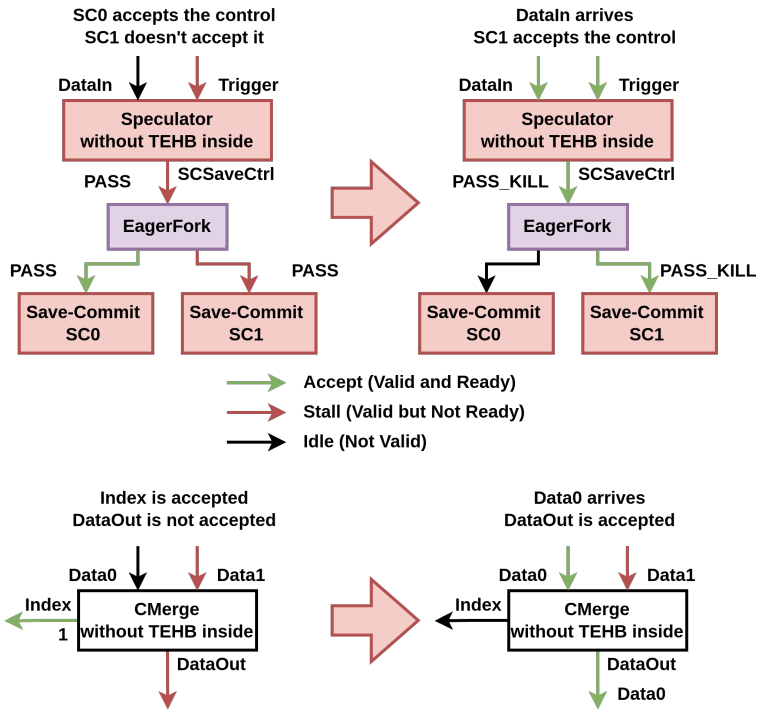


Figure 17: Top: Without a persistent data payload, different save-commit units receive inconsistent control values. Bottom: A similar issue occurs in the classical case with a CMerge unit.

This is identical to the classical and well-known issue with a `CMerge` unit without an internal transparent buffer. Internally, it uses an eager fork to distribute both the index and data. If one of the outputs is stalled, and a new data input with a lower index arrives, the fork may deliver mismatched index and data.

To guarantee persistency, we insert a **transparent buffer** (**TEHB**) on the *impersistent* channel before the eager fork. This buffer typically forwards a data token immediately, but when the output is stalled, it holds the input until the output is ready. This ensures the data payload remains stable and prevents handshake violations. The same approach is also used for `CMerge` units.

In the case of the speculator, it also internally contains an eager fork to broadcast control signals to the decoders (Figure 15), distinct from the eager fork placed before the save-commit units in Figure 17. To preserve consistency among the decoders, we place a transparent buffer before this internal fork.

Moreover, the output decoder requires not only control signal persistency, but also that of the `dataOut` and the spec bit. Therefore, these values are also latched in the transparent buffer.

In Haoran Zhao's original design [2], this issue was addressed by maintaining separate FSM states to handle the stalling of each control signal. However, this led to a large and complex FSM. During the bug fix process, I re-designed this logic using a transparent buffer—providing a unified and cleaner solution, consistent with the `CMerge` approach.

6.4.6. Recovering Is Not Optimized

After a misspeculation, the speculator must first flush all tokens from its internal FIFO (Section 6.4.1) and then wait for the arrival of a non-speculative trigger (Section 6.4.2) before it can initiate a new speculation. In practice, this FIFO clearance takes a noticeable number of cycles.

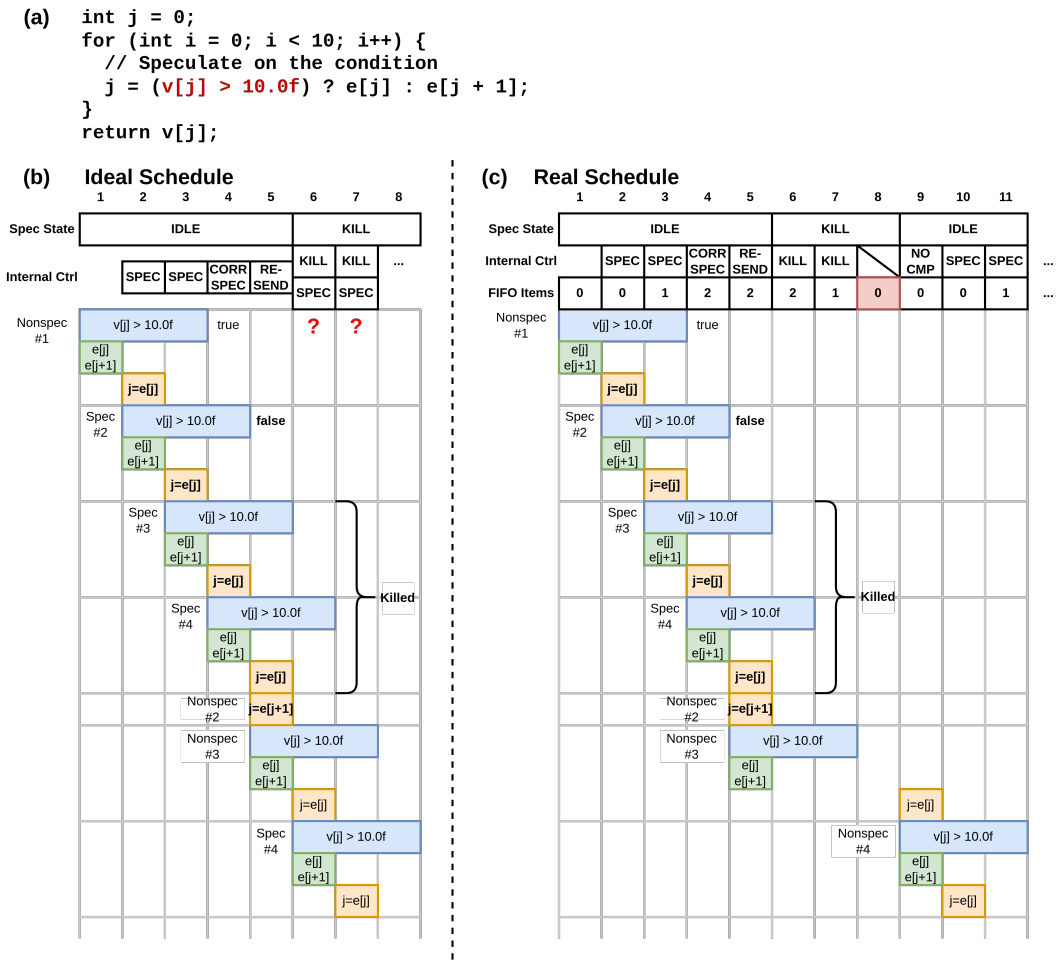


Figure 18: The recovery schedule of the data speculation example (Section 2.2). (a) Program snippet. (b) Ideal scheduling, where a new speculation begins one cycle after misspeculation is detected. (c) Actual scheduling, where a new speculation is delayed until FIFO clearance. `CORR_SPEC` stands for `CORRECT_SPEC`.

Figure 18 illustrates the recovery schedule for the misspeculation example in Section 2.2, which speculates on the condition `j = (v[j] > 10.0f) ? e[j] : e[j + 1]`. Ideally, as shown in Figure 18 (b), the speculator would resume speculation in Cycle 6, just one cycle after the misspeculation is detected in Cycle 5. Achieving this, however, would require internal modifications to the speculator's structure. Specifically, the `KILL` and `SPEC` signals must be overlapped during the `KILL` state. Currently, while the `CORRECT_SPEC` signal enables this in

the IDLE state, it is not supported in the KILL state. Additionally, new FIFO entries are pushed during KILL before the FIFO is fully cleared—supporting this would require tracking the head pointer of the FIFO at the start of the KILL state, which is not currently implemented.

The actual behavior is shown in Figure 18 (c). By Cycle 8, the FIFO has been cleared and all KILL signals have been sent. Due to limitations in the current FSM design, an extra cycle is required after the clearance of the FIFO before transitioning back to the IDLE state, resulting in a four-cycle delay before the next iteration begins in Cycle 9. Notably, when II = 1, the non-speculative token arrives in the same cycle (Cycle 8) that the FIFO is cleared, following the evaluation of `v[j] > 10.0f` from the Nonspec #3 iteration. Consequently, in Cycle 9, the speculator emits the received token with the internal signal NO_CMP. The speculator resumes speculation in Cycle 10.

In summary, recovery is blocked by the delay required to clear the FIFO. When II = 1, the first token output after a misspeculation is always a NO_CMP, effectively wasting one speculation opportunity.

Note that the KILL_ONLY_DATA state (Section 6.4.1) can mitigate part of the delay only when II > 1, where non-speculative tokens arrive more slowly. However, it does not eliminate the delay caused by FIFO clearance.

Optimizing recovery behavior could yield significant performance gains—especially in loops that are executed multiple times or in workloads involving data speculation—but is left for future work.

6.4.7. One Data Token Emission per Trigger Consumption

In Josipović et al. [1], speculation was performed unconditionally, regardless of whether the basic block was actually executed. However, we observed that emitting more data tokens than triggers can lead to deadlock—especially when other tokens have already exited the loop, leaving only the predicted token and PASS signals to the save-commit units.

Haoran Zhao's work [2] introduced the trigger signal to address token mismatches. While this mitigated the issue, it wasn't sufficient. Each token emitted on dataOut must correspond to one trigger consumption. This is why we updated the speculator to consume a trigger even on events like NO_CMP.

Notably, on RESEND, the speculator emits the real token *without* receiving a new trigger. This is because RESEND can occur after the control token has left the loop. In such cases, the previously misspeculated trigger is discarded by the downstream commit unit. When the control token is still within the loop, it is killed by both the save-commit unit and the speculator itself (as a misspeculated trigger).

7. Speculative Units Placement

Here, I explain how the speculative units are placed to enable the speculation functionality.

7.1. Speculator

The speculator can be placed on any edge within a loop's basic block, which is typically a throughput bottleneck and is predictable in its behavior.

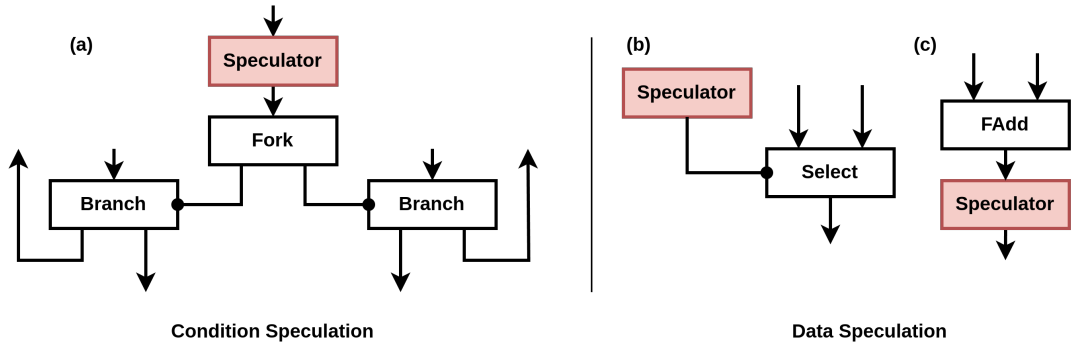


Figure 19: Different placements of the speculator.

In most scenarios, it is placed before a fork that leads to the condition channel of all loop branches. This is referred to as a **condition speculation** case (Figure 19 (a)).

Other scenarios are categorized as **data speculation**. Data speculation can sometimes involve predicting the condition of an if statement (Figure 19 (b)). Additionally, more general speculation methods are supported, including non-boolean predictions (Figure 19 (c)).

7.2. Save-Commit: Graph Cut

The flexibility of Speculation v1 stems from its ability to restore the circuit state when misspeculation occurs.

Misspeculated tokens that are distributed across the circuit need to be killed, and correct tokens must be sent to completely roll back to the most recent non-speculative, correct state. This cannot be handled solely by the speculator. At this point, save-commit units assist the speculator by covering every path from the loop entry to the loop exit. In other words, the speculator and save-commit units together form a cut of the loop DFG (Figure 20).

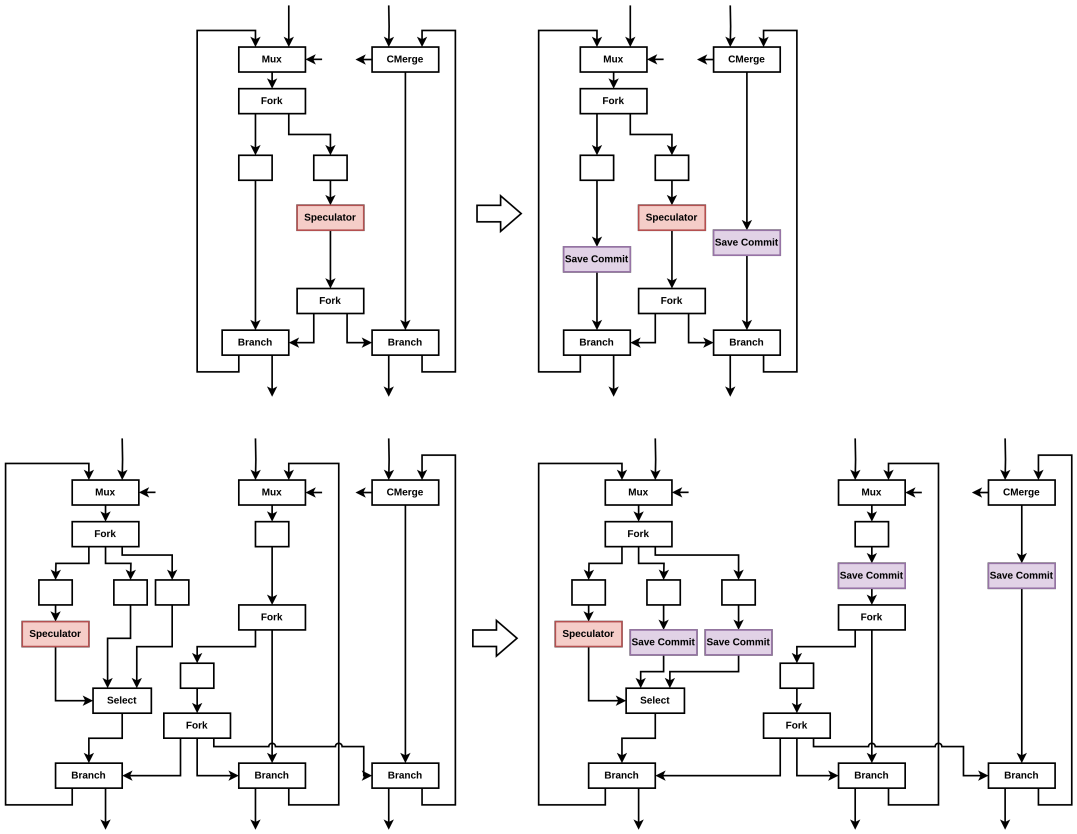


Figure 20: Place save-commits to cut every path in the loop.

In the following sections, we assume the loop consists of a single basic block (refer to Section 5.1). Scenarios where the loop has multiple exits or internal control flow branches are not well addressed for placing save-commit units and will be explored in future work.

Based on this assumption, we also exclude cases where the speculative region contains inner loops. Therefore, the speculated loop body is acyclic, and we refer to a unit as being “above” or “below” another unit within the loop’s DFG.

7.2.1. Flexibility and Constraints in Save-Commit Placement

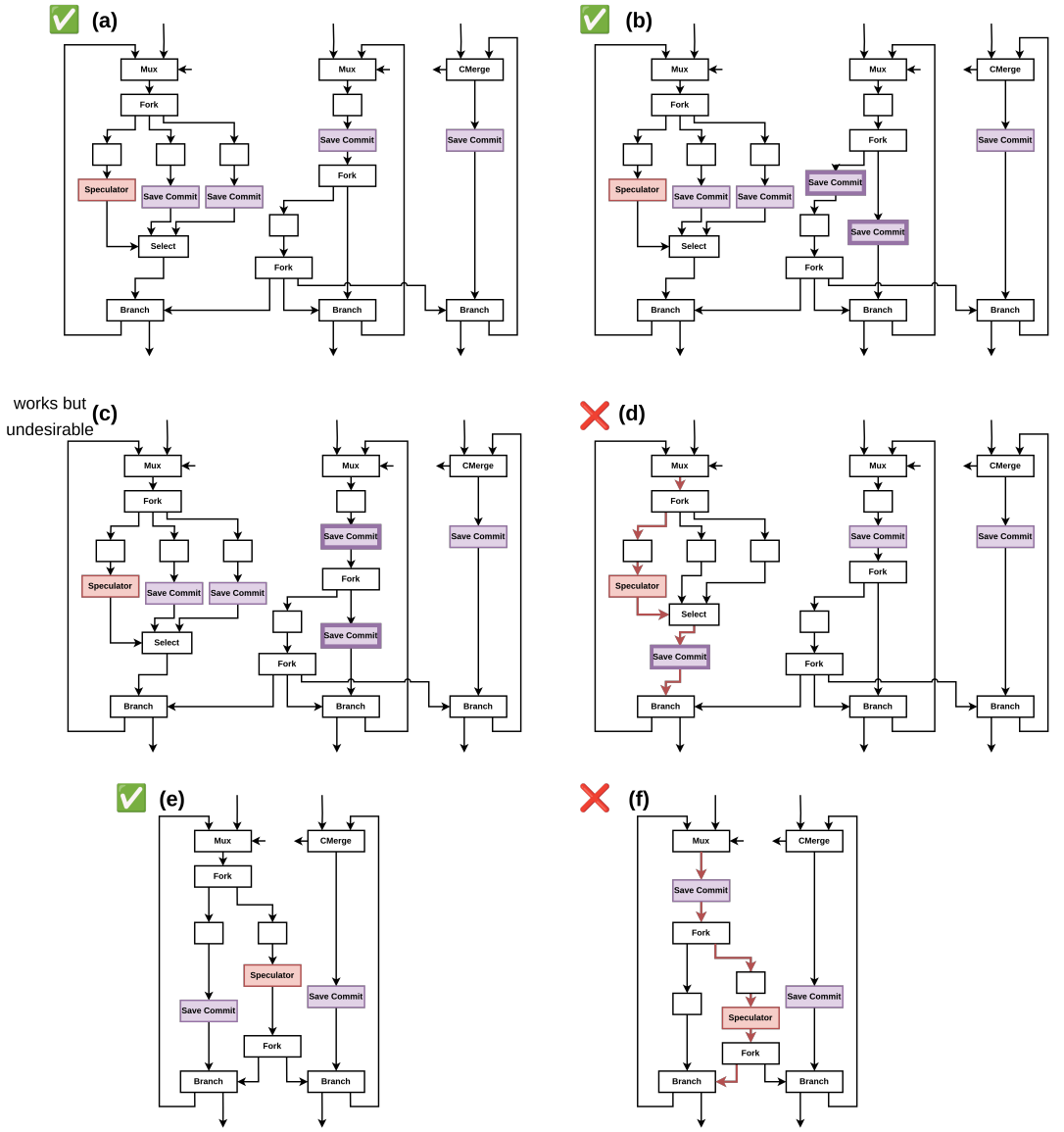


Figure 21: Variants of save-commit placements.

While save-commit units must collectively form a cut together with the speculator in the loop's dataflow graph (DFG), their specific placement allows for some flexibility. In Figure 21 (a), we show the default configuration (identical to the bottom of Figure 20). In Figure 21 (b), one save-commit unit has been repositioned below the fork, and another is added to maintain the cut. This is a valid alternative.

However, as shown in Figure 21 (c), chaining save-commits in series—though functionally correct—is not recommended. This can complicate speculation behavior and lead to redundancy or confusion (e.g., the speculative region from Section 2.4 cannot be drawn). Worse, as seen in Figure 21 (d), placing a speculator and a save-commit along the same path introduces a correctness issue: upon misspeculation, the save-commit replaces the correct token from the speculator with the resent token, causing misbehavior.

Figure 21 (e) mirrors the configuration from the top of Figure 20. In contrast, Figure 21 (f) fails for the same reason as (d): the save-commit erroneously intercepts the corrected token from the speculator.

The key rule is that **each path from loop entry to loop exit should be cut exactly once**, by either a speculator or a save-commit unit. This rule avoids overlap and ensures correct speculation behavior. Theoretically, if a path is cut more than once, it's always possible to adjust the placement using local rewrite rules (e.g., repositioning units to less overlapping paths), as illustrated in the figure below.

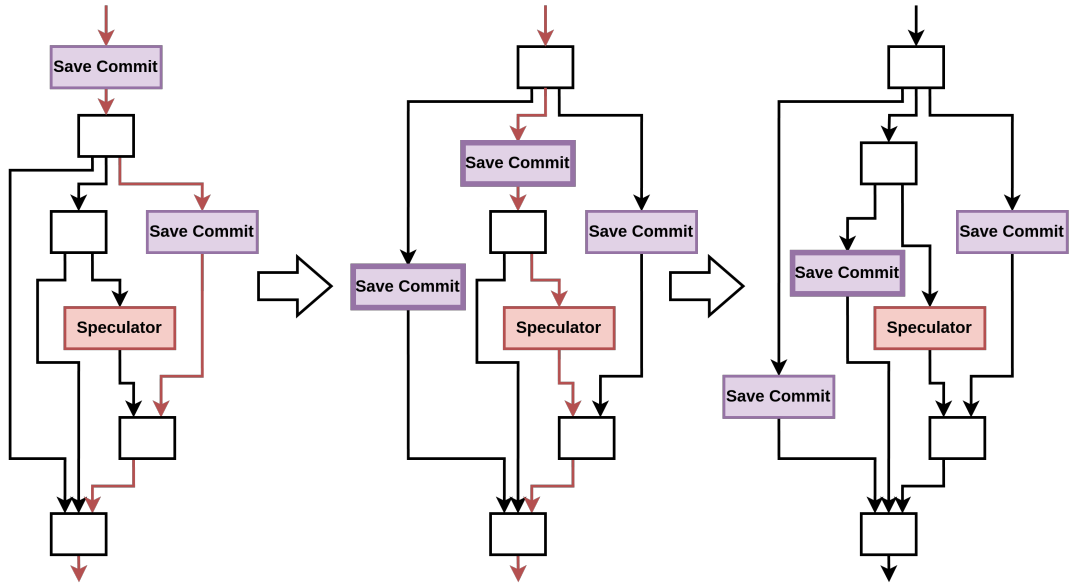


Figure 22: Rewriting placements to ensure each path is cut exactly once.

Note that we previously omitted the trigger signal of the speculator for simplicity. However, when placing save-commit units, it's essential that the **trigger input to the speculator is taken from above the save-commit**. This avoids deadlock during loop instantiation, and ensures correct alignment between control and data tokens.

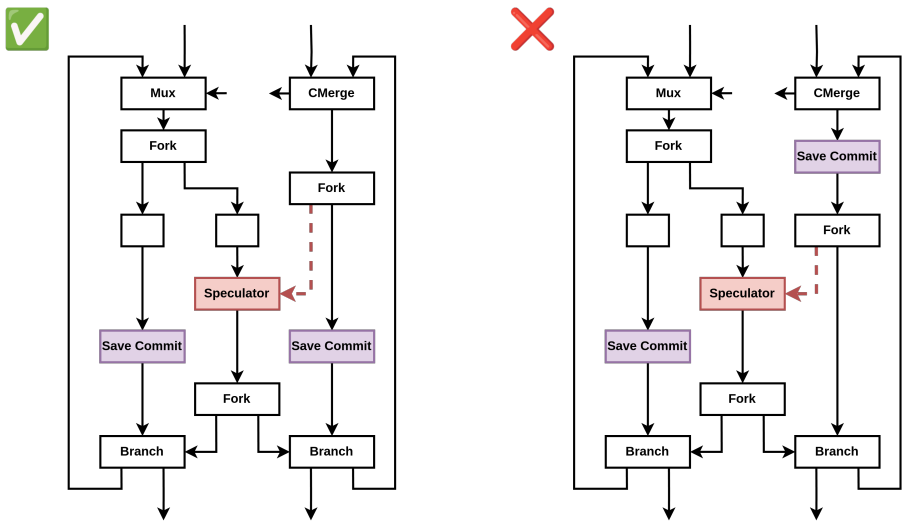


Figure 23: The trigger signal must be sourced from above the save-commit unit.

7.2.2. Terminus Cut

Loops may contain terminus nodes, such as `StoreOp`, which don't have output edges. The question is whether these units are cut by the speculator or save-commit units.

The answer is that sometimes, save-commit units and the speculator will cut these units, and other times they won't.

At the top of Figure 24, the `StoreOp` is fully cut by both the speculator and save-commit units, allowing speculation to work correctly on this circuit.

At the bottom of Figure 24, the `StoreOp` is located above the speculator. On the bottom left, the save-commit unit is positioned above the `StoreOp`. This placement causes problems: in the first loop *iteration*, a speculative token from the save-commit combines with a non-speculative token, and if misspeculated, it cannot be restored. In general, tokens from different *executions* of the speculative region should never meet.

We assume the position of the speculator is fixed because its placement is tightly coupled with its functionality.

To resolve the problem, there are two possible solutions: we can move the save-commit unit below the terminus, which prevents the terminus from being *partially* cut (as shown in the bottom middle of Figure 24). Alternatively, we can adjust the placement of commit units to ensure that tokens from different executions are not combined. The configuration at the bottom right of Figure 24 is valid—where one commit unit is cut while the other is not, yet both are connected to the same terminus. We refer to this as a *skewed terminus*. Although the two commit units receive tokens from different executions, they manage their tokens correctly, so the subsequent node receives properly matched tokens. In this way, speculation only concerns the nodes before the commit units.

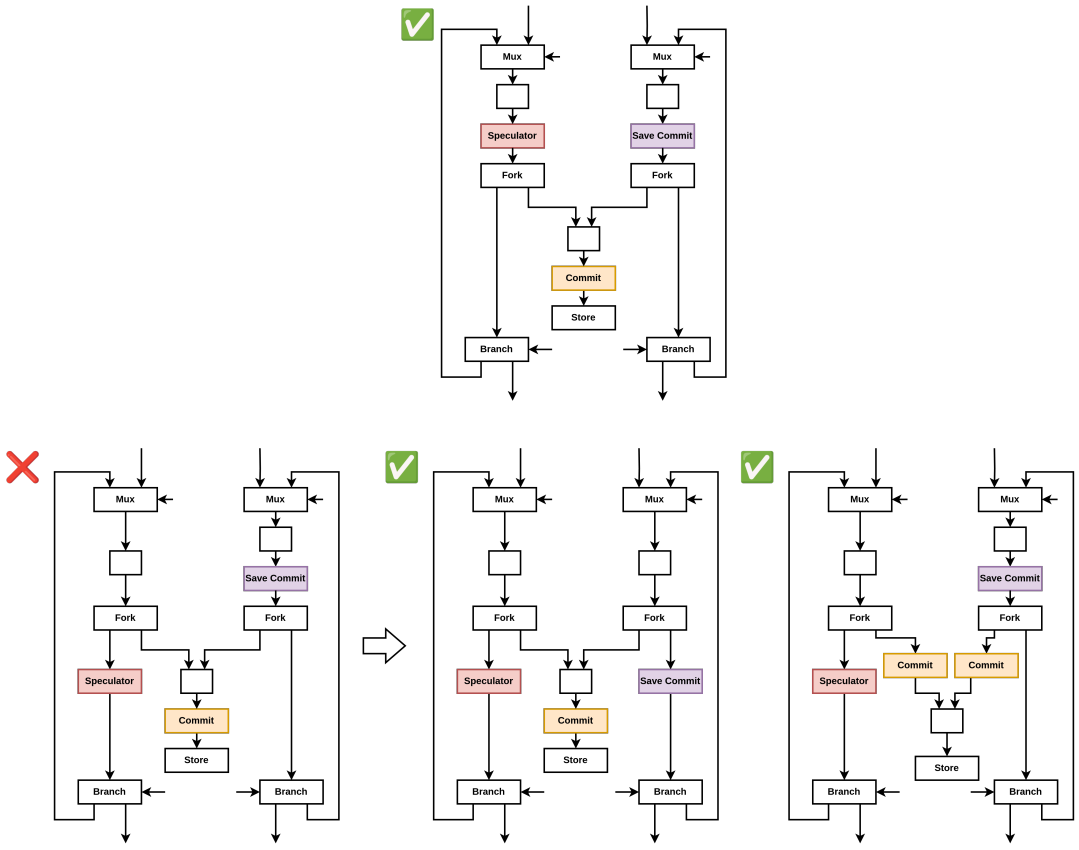


Figure 24: Different cut configurations for terminus units.

7.3. Commit: Preventing Side Effects and Token Reordering

Commit units must be inserted before operations with side effects—such as `StoreOp`, `EndOp`, and `MemoryControllerOp`, as is intuitively required. Note that we do not place commit units before `LoadOp`; memory loads are performed speculatively. As a result, all commit units are placed on acyclic paths that lead to these terminus operations.

In fact, additional commit units may be needed, as noted in Haoran Zhao's master's thesis [2], where multiple basic blocks within the speculative region converge. We elaborate on this in the next subsection.

We refer to commit units that prevent side effects as **regular** commit units, in contrast to those placed at the convergence points.

7.3.1. Commit Units at Convergence Points

When control flow converges within a speculative region, commit units are necessary to prevent token reordering, as discussed in Haoran Zhao's master's thesis [2].

During circuit state restoration, a save-commit unit may resend a control token, causing a temporary situation where **two** control tokens exist simultaneously. In the example shown in Figure 25, if the speculative token takes the slower path while the resent (non-speculative) token takes a faster one, the non-speculative token may overtake the speculative one—leading to misordered execution.

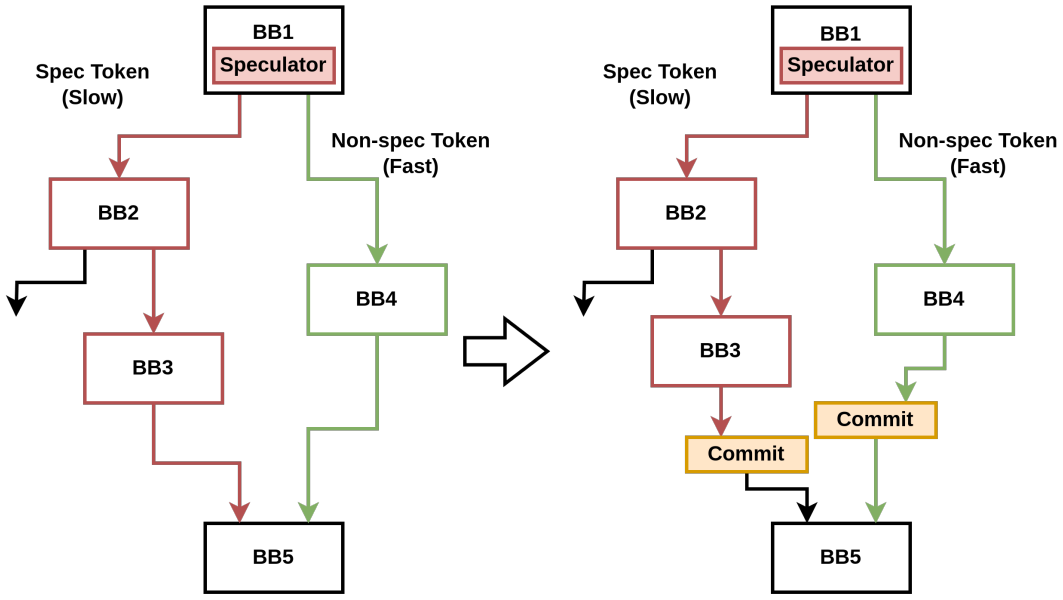


Figure 25: Commit units are needed at basic block convergence points.

This reordering of speculative and non-speculative tokens introduces issues in the circuit, specifically for the speculator and save-commit units. The speculator treats the arrival of a non-speculative data token as the signal to finish killing incoming tokens (see Section 6.4.2), and the save-commit unit does not check the spec bit of incoming tokens, killing them based solely on their arrival order (see Section 6.3.1).

To prevent reordering, commit units must be placed at the convergence point (BB3 and BB4 in Figure 25). These commit units should be placed as far from the speculator as possible to maximize the effectiveness of speculation.

Note: With the integration of Fast Token Delivery techniques [9], the need to place commit units at the convergence points may be eliminated, as also discussed in Haoran Zhao's master's thesis [2].

7.3.2. Removing Inter-BB Commit Units in Nested Loops with II = 1

In nested loop scenarios, the previous discussion (Section 7.3.1) would imply placing commit units on the **top of the innermost loop**, which can significantly reduce the benefit of speculation.

However, if the innermost loop achieves an **initiation interval (II) of 1**, such commit units can be safely removed from the inner loop edges.

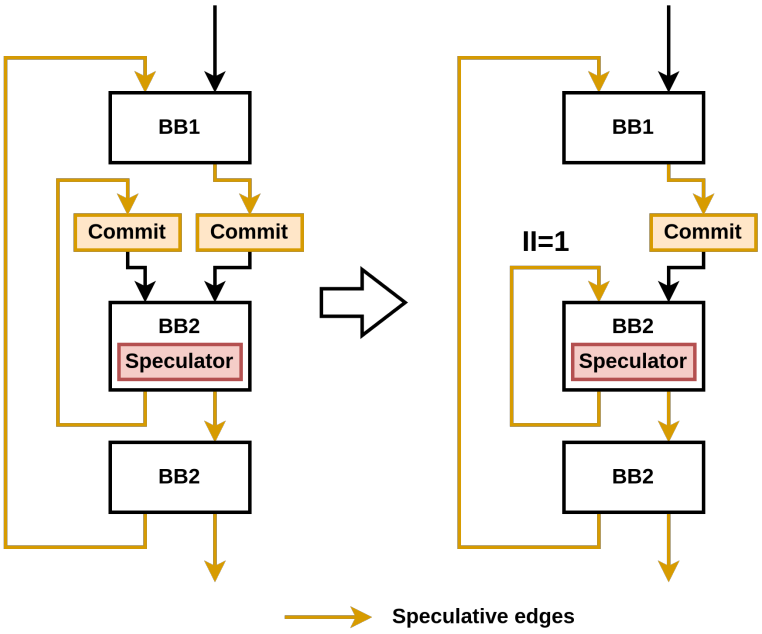


Figure 26: Commit units on inner edges can be removed when II = 1.

Here's why: When II is 1, a control token enters the control merge every cycle. Reordering can only occur if a speculative control token is still in flight after a misspeculation is realized. Suppose the speculator incorrectly predicts an "inner loop continue" and later resends a corrected "exit" token. Due to II = 1, all speculative tokens have already passed through the control merge by the time the misspeculation is realized. Only the resent token arrives afterward, preserving in-order behavior.

On the other hand, if the speculator incorrectly predicts an "inner loop exit" when it should have been a "continue," the speculative token is caught by the commit unit placed at the loop header—again preventing reordering.

7.3.3. Convergence Inside Loops Is Not Well Considered

We currently assume loops consist of a single basic block, as required by our algorithm (Section 5.1). Under this assumption, basic block convergence does not occur within the loop.

The case where convergence does occur inside the loop has not been thoroughly addressed. In such scenarios, commit units must be placed on cyclic paths to prevent token reordering. This results in a speculator and a commit unit being chained in series—similar to the chaining of a speculator and a save-commit unit discussed in Section 7.2.1—raising concerns about the speculative region becoming fragmented or ill-defined. More critically, it can cause token

mismatches. Duplicated kill tokens for the same input token are issued, once by the commit unit and again by the speculator and save-commit units.

This issue is, in fact, one of the reasons why I modified Aleix Seguí Ugalde's algorithm [3] to avoid placing commit units before `LoadOp`, inside the loop. Further investigation into these cases is left for future work.

8. Connection with Speculator

The speculation decisions made by the speculator must be communicated to each speculative unit via control signals, as discussed in previous sections. However, this connection is not straightforward—the signals must pass through a specialized handshake network to avoid token mismatches.

8.1. Save-Commit

Recall that three control signals from the speculator interact with the save-commit units: `SCSaveCtrl`, `SCCommitCtrl`, and `SCIsMisspec` (in Section 6.4.4).

Core Control	SCSaveCtrl	SCCommitCtrl	SCIsMisspec
SPEC	PASS	-	-
NO_CMP	NO_CMP	-	-
CMP_CORRECT	KILL	-	False
CORRECT_SPEC	PASS_KILL	-	False
RESEND	RESEND	-	-
KILL	-	KILL	True

Table 3: Decoder behavior related to save-commit units.

Also recall that there are two separate cases where a `KILL` signal is issued:

- One occurs under `CMP_CORRECT`, passed via `SCSaveCtrl`,
- The other under `KILL`, passed via `SCCommitCtrl`.

The reason for routing the latter through `SCCommitCtrl` is to handle it differently, as it relates to discarding misspeculated tokens that might have already exited the loop.

The save-commit units are connected to the speculator as shown below, specifically in single-basic-block loops (Section 5.1):

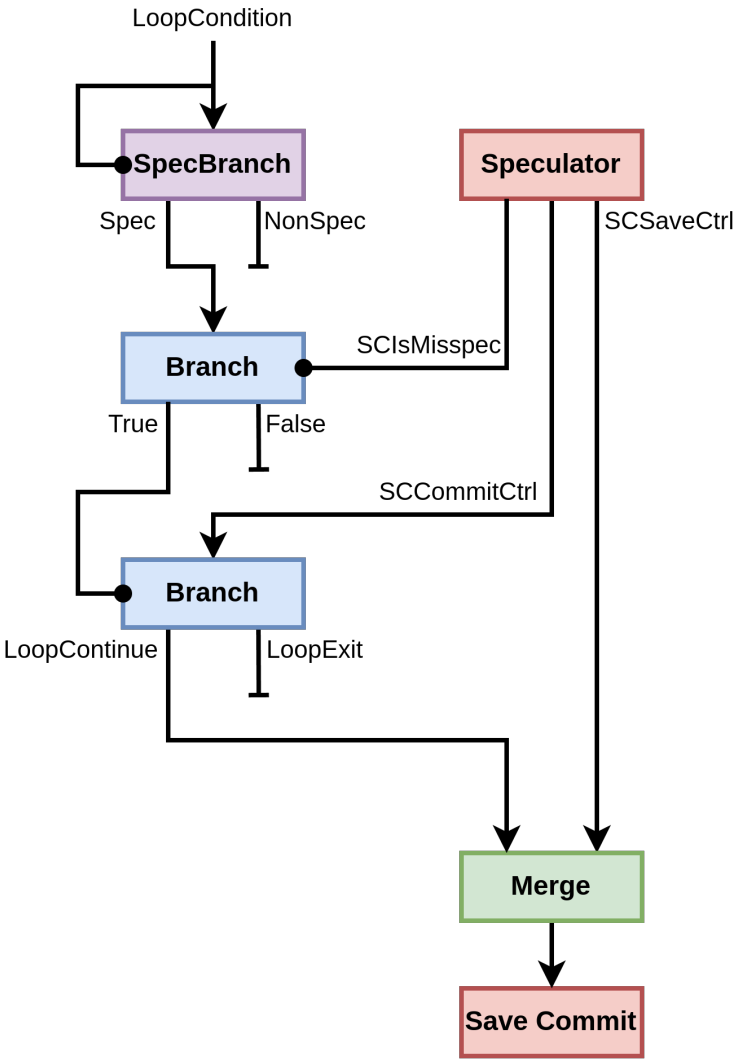


Figure 27: Connection between speculator and save-commit units, in single-basic-block loops. Note that the merge unit has an ordering issue, which will be discussed in Section 8.1.1.

The `SCCommitCtrl` path must discard certain tokens to avoid mismatches. Each `KILL` signal kills one token in the save-commit's internal buffer. If a speculative token did not reach the save-commit (i.e. it has already exited the loop), its corresponding `KILL` is still sent by the speculator, but it should not reach the save-commit.

To discard the token, we use a sequence of three branch-like units. The first is a **speculating branch** (see Section 6.1); both the data and condition inputs are set to the loop condition itself, allowing non-speculative loop conditions to be discarded. The second is a regular branch conditioned on `SCIsMisspec`, so only **misspeculated** loop conditions pass through. Finally, this result gates the `SCCommitCtrl` signal through a third branch. With this setup, `SCCommitCtrl` is propagated to save-commit units only when the **misspeculated** loop condition indicates "**loop continue**".

When the loop consists of multiple basic blocks and has multiple exits, it appears that all loop conditions must branch the `SCCommitCtrl` signal. Note that cases involving branching and convergence within the loop are not thoroughly addressed in this work.

If you revisit the example in Figure 14, you'll notice that the save-commit unit receives one fewer `KILL` signal compared to Figure 13, even though the speculation behavior is identical.

This is because the speculator still sends the same number of `KILL` signals, but the last one is discarded by the `SCCommitCtrl` branch logic.

8.1.1. Buffer Placement Restriction

Placing a buffer before a merge unit can lead to reordering of control signals, which compromises correct execution. For this reason, we impose a restriction: **buffers must not be inserted between the speculator and a merge**.

8.1.2. `SCCommitCtrl` May Be Unnecessary

The `SCCommitCtrl` signal isn't strictly required. Just as the speculator detects that all misspeculated tokens have been cleared upon receiving a non-speculative token (see Section 6.4.2), save-commit units could adopt the same approach: **kill tokens until a non-speculative one arrives**.

This change would make save-commit units stateful but would significantly simplify the control signal network. As a result, the need for a `merge` unit would be eliminated. Exploring this approach is left as future work.

8.2. Commit

Although the speculator's `commitCtrl` channel emits a `PASS` or `KILL` signal for every prediction made by the speculator, not all corresponding tokens necessarily reach the commit unit. Therefore, forwarding `commitCtrl` to commit units requires a slightly more sophisticated network.

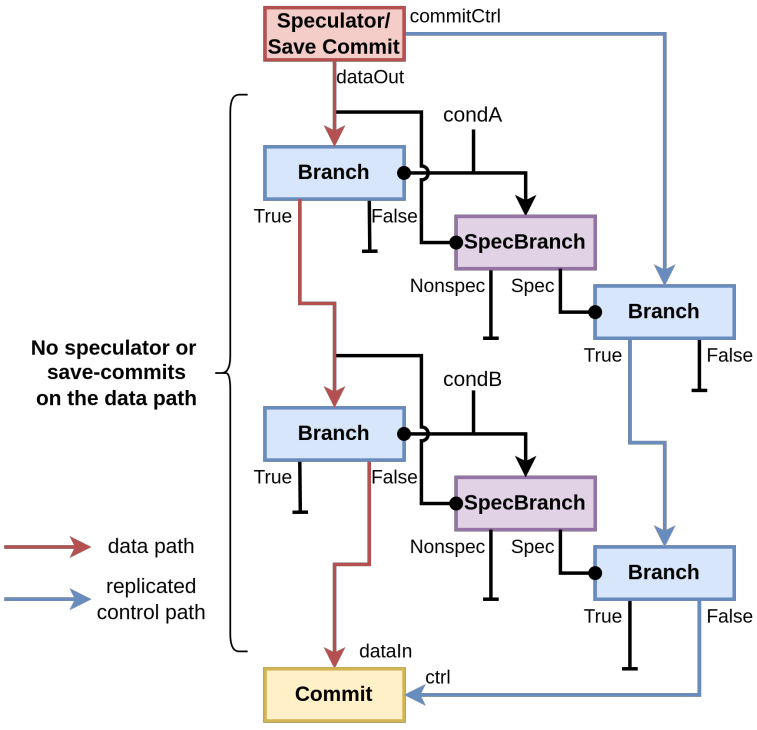


Figure 28: Commit control network. Fork units are omitted for clarity.

Figure 28 shows how `commitCtrl` signals are routed to the appropriate commit units. Each commit unit connects to the speculator or a save-commit unit—sometimes to multiple. Consider a path from a **source** (speculator or save-commit) to a given commit unit. This path must be confined to the speculative region—that is, it must not pass through any other speculator or save-commit unit—as doing so can cause token mismatches.

Along the path, there may be branches both inside and outside the loop. Note that multiple paths may exist from a source to the commit unit. However, since all such paths traverse the same number of branches with identical conditions, it is sufficient to examine just one.

At each branch encountered on the path, the `commitCtrl` network replicates the same branching structure. However, a subtlety arises: while branch conditions (like `condA` and `condB` in Figure 28) apply to **all** tokens (speculative or not), `commitCtrl` signals correspond **only** to speculative tokens. To prevent mismatches, we insert speculating branches (see Section 6.1) along the `commitCtrl` path. These filter the branch conditions, ensuring they apply only when the corresponding token is speculative.

Thanks to this replication mechanism, each commit unit receives control tokens only for speculative tokens that actually arrive—ensuring correctness without mismatches.

8.2.1. No Need to Consider Convergence

You might wonder why the commit control network only replicates branches, and not convergence structures like muxes or cmerges. Shouldn't those also affect how `commitCtrl` is routed?

The reason we can ignore them is that commit units are **already placed** at convergence points between basic blocks within the speculative region (see Section 7.3.1). This placement ensures that tokens from different paths do not reach the same commit unit, eliminating the need for additional convergence handling in the control network.

9. Units No Longer Needed

Some units are no longer needed in the current design. This section discusses these units and their removal.

9.1. Save

In previous work [1], [2], [3], "save unit" was introduced. These units were placed at the non-speculative edges where they meet a speculative edge.

While save units are useful for speculation in non-loop programs, they can impair the speculation effect within loops, much like commit units. In the example shown in Figure 29, a save unit is placed on the non-speculative operand of the `Add` unit, because the other operand is speculative. However, this placement unintuitively disrupts loop pipelining, as save units only forward new tokens after the previous speculation has been resolved (at least under the current specification). Here, since the value is constant, the save unit is unnecessary and can be removed.

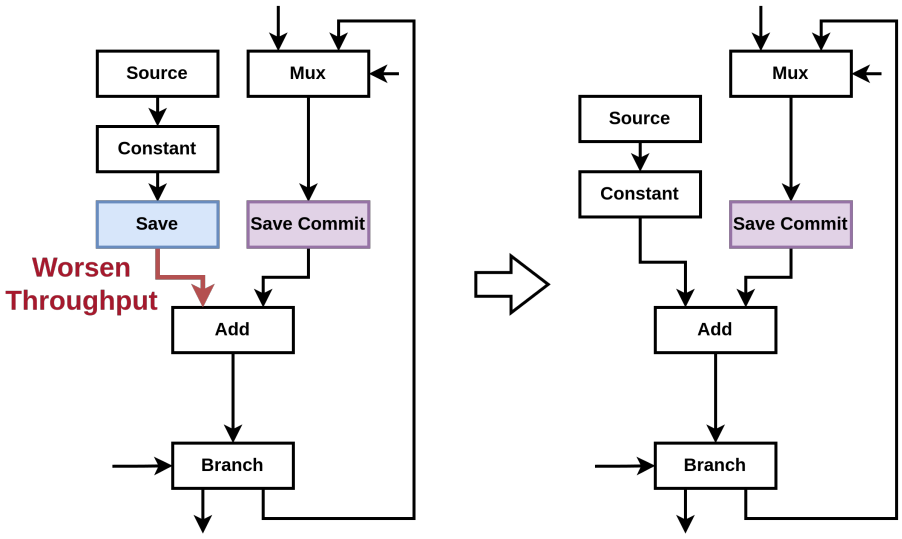


Figure 29: Placing a save unit affects II.

So, do we need to place save units at all? In general, they are required where non-speculative edges meet speculative ones. However, excluding constant edges, which do not require save units, non-speculative edges are usually not present within speculative loops. The only possibility for non-speculative edges comes from commit units. There are two kinds of commit units (see Section 7.3):

- For commit units preventing side effects, the operations that cause side effects (e.g., stores) are all terminus nodes and commit units are not placed inside the loop. (Note that we do not place commit units before loads.)
- For commit units preventing token reordering, our assumption of single basic block loops (Section 5.1) ensures there are no convergence points inside the loop.

Therefore, under our assumption for implementation, save units are unnecessary, and token restoration can be fully managed by the save-commit units.

Without the single basic block loops assumption, there might be convergence points inside the loop, and commit units are placed there (Section 7.3.3). In such cases, save units may appear to function correctly, but they still fall short in addressing the kill signal mismatch problem (Section 7.3.3): both the commit unit and the speculator/save-commit issue kill signals for the same token. The save unit still fails to emit a sufficient number of matching tokens, leaving the issue unresolved.

9.2. Synchronizer

The synchronizer, introduced in Haoran Zhao's master's thesis [2], was designed to prevent the issue where only some tokens in a single iteration are speculative while others are non-speculative. This could lead to misbehavior, particularly at commit units. However, with the recent update to save-commit units, they now always send out speculative tokens upon receiving a `PASS`, regardless of whether the token was originally speculative or non-speculative. This update effectively prevents the aforementioned issue, eliminating the need for the synchronizer.

10. Updates from Haoran Zhao's Work

10.1. Speculator

The original implementation contained several bugs, which were uncovered during benchmarking, code review, and unit testing (unit tests are still in draft form and have not been merged or submitted as pull requests).

While I have resolved some of these issues, I believe there may still be unresolved bugs. These have not been addressed due to time constraints. Replicating each bug by preparing specific test signal patterns is a time-consuming process, so to develop a random signal generator and employ fuzzing techniques will be the future work, although this is a low-priority task. Suspected problem lines have been commented, and I assume these unresolved bugs are limited to edge cases.

Below are the issues I've resolved so far:

- Enforced a single data token emission per trigger consumption (Section 6.4.7):
  - Killed the trigger when misspeculated to prevent deadlock.
  - NoCmp now consumes a trigger.
- Correctly handled the case where the FIFO is full.
- Several fixes to the handshaking logic.
- Redesigned the FSM for better clarity and simplicity while working on the above fixes.

10.2. Save-Commit

The original implementation of the save-commit unit had several bugs that were identified and fixed.

- Incrementing the `Current` Pointer upon receiving `NO_CMP` and Certain `KILL`s.
- Several fixes to the handshaking logic.
- Always Emitting Speculative Tokens on `PASS`: This change removes the need for a synchronizer, as discussed in Section 9.2.
- Correctly handled the case where the FIFO is full.
- Redesigned the combinational `process` for better clarity and simplicity while working on the above fixes.

10.3. Commit

- Removed Internal Buffers: Haoran Zhao's work [2] included two FIFOs inside the commit unit: one for speculative tokens between the speculating branch and the regular branch, and one for control signals. Buffers are better placed **outside** the commit unit for clarity and control. Therefore, I removed the internal FIFOs.

III. Implementation

So far, I've described how speculative units are placed and connected. In this chapter, I present how these placements and connections are implemented in practice, outlining the algorithms behind them—and more broadly, how speculation is realized in Dynamatic.

Figure 30: The final stages of Dynamatic flow modified by speculation. Stages related to speculation are marked in red.

The figure above shows the Dynamatic flow with speculation enabled. Speculation modifies the final stages of the flow, including speculation-specific passes and RTL generation.

The sections in this chapter follow this compilation flow. Section 11 explains how the speculation pass works, Section 12 highlights the distinctive features of the IR it produces, Section 13 discusses the manual buffering applied afterward, and Section 14 describes the backend that generates the final circuit.

11. Algorithms in Speculation Pass

A handshake-level transform pass enables speculation in the circuit, handling tasks such as the placement of speculative units and their connections. This pass is located in the `experimental/lib/Transforms/Speculation/` folder, and is currently performed after the buffering pass.

The procedure for this pass is as follows:

1. Determine the placement of speculative units.
2. Place the speculative units.
3. Connect the speculative units to the speculator.
4. Mark speculative edges (Section 12).

Note that the current implementation assumes the loop consists of a single basic block for the sake of algorithmic simplicity. Generalizing this approach is left for future work.

This pass builds on Aleix Seguí Ugalde's previous work [3], with significant modifications.

11.1. Speculation as a Post-Buffering Pass

The speculation pass is currently executed after the buffering pass for the following reasons:

- Timing and latency models for speculative units are not yet available.
- The handshaking network for save-commit control signals restricts placing buffers on certain edges (see Section 8.1.1).

Speculation enables a better II, which may require additional buffering for improved pipelining. As a result, making the speculation pass occur before buffering is an ideal goal, but will be left for future work.

However, other types of buffers are still necessary to prevent deadlock. These buffers are not properly handled by the existing buffering pass, as discussed in Section 13.1.

11.2. Determine Placements

Before placing the speculative units, we first determine their positions. This allows for potential repositioning or conversion of the units as needed.

The algorithm is built on our previous understanding. For instance, it identifies the positions of save units—units that are never actually placed—and later converts them into save-commit units. The algorithm still provides valid placements, though it is not always optimal. Given this, updating the algorithm entirely is not a priority at the moment[1].

The procedure for this stage is as follows:

1. Identify save positions.
2. Identify save-commit units.
3. Identify regular commit units (those placed to prevent side effects).
4. Identify commit units between basic blocks (those placed to prevent token reordering).

Note: The position of the speculator is **manually specified**.

Throughout this section, we use the following kernel as a running example:

```
void kernel(int a[], int b[]) {
  int x;
  for (int i = 0; i < N; i++) {
    int x1 = f1(x);
    int x2 = f2(x);
    a[x1] = x2;
    // Speculate on cond(x)
    x = cond(x) ? x1 : x2;
    b[i] = x;
  }
}
```

Listing 2: Example kernel used to illustrate the placement algorithm.

In this kernel, assume that `cond(x)` is the performance bottleneck. We aim to speculate on its outcome.

The dataflow circuit is shown below, with the speculator already manually placed.

[1] I have a draft of the new placement algorithm in `NewPlacementFinder.cpp` based on the graph cut in the repository, but it has not been thoroughly tested or documented yet.

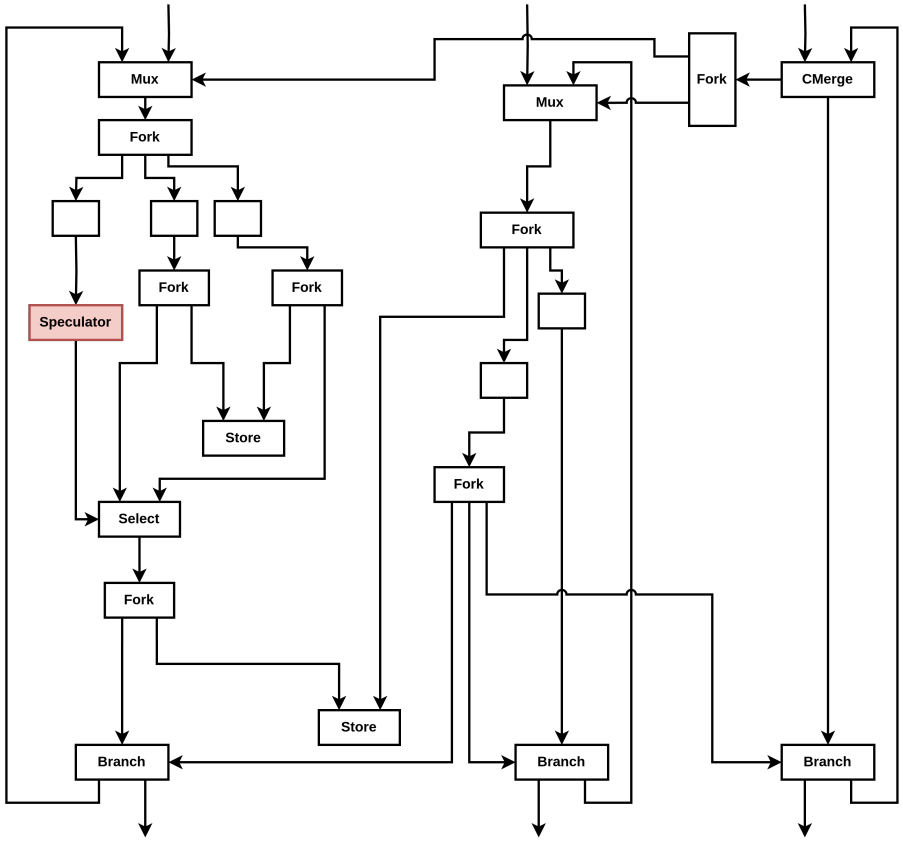


Figure 31: Initial circuit with the speculator manually placed.

11.2.1. Identify Save Positions

The algorithm begins by identifying the positions of the (previously used) save units. However, since these will be converted into save-commit units in the following stage, this stage can be considered as "identifying a subset of save-commit positions."

To do this, the algorithm traverses the loop's dataflow graph (DFG), starting from the manually specified location of the speculator and proceeding forward—up to but not through branches. This means the traversal avoids backedges. All visited edges are marked as ***directly* speculative edges**.

Then, for each unit in the loop, the algorithm checks whether the unit receives inputs from both directly speculative edges and non-speculative ones. If so, it places save units on the non-speculative edges.

After this stage, the circuit looks like the following: four save positions have been identified.

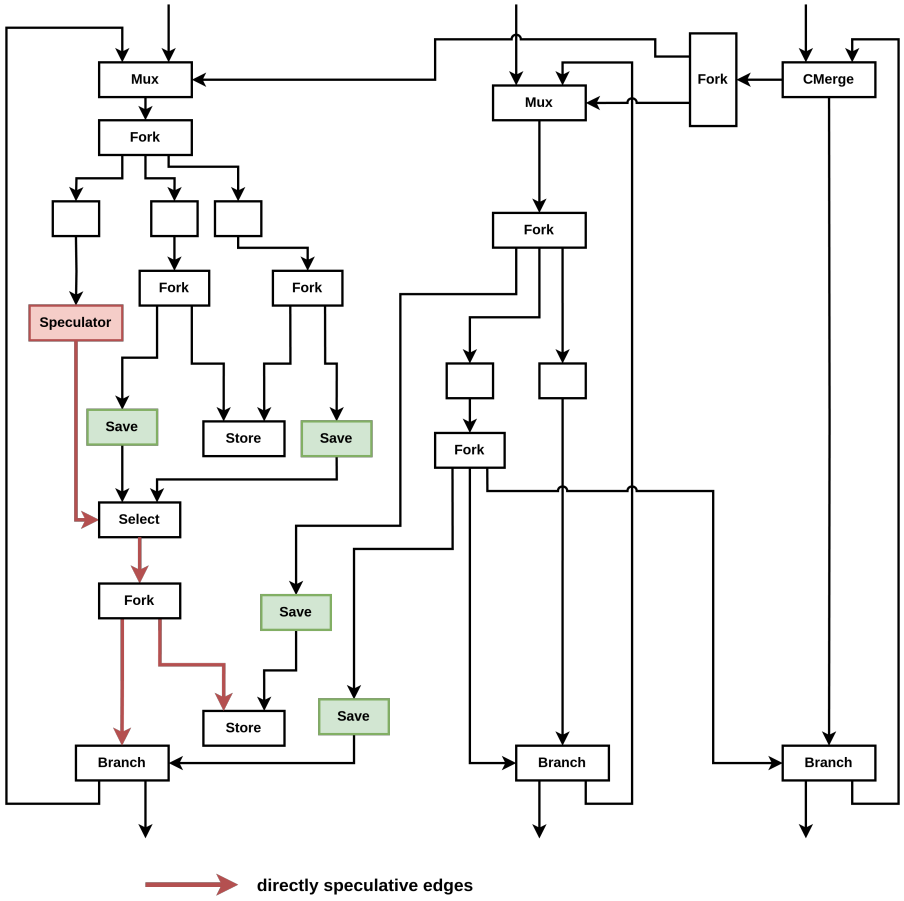


Figure 32: Save positions are determined.

11.2.2. Identify Save-Commit Units

The algorithm now identifies all save-commit units. It traverses the entire loop DFG starting from the control merge (CMerge), ensuring all paths are cut by the speculator or save-commit unit. During the traversal:

- If it encounters the speculator or already determined save-commit positions, the traversal stops because the path is cut by them.
- If it reaches a branch, it indicates that the traversed path is not cut by the speculator or save-commit unit. In this case, **a save-commit unit is placed in front of the branch**.
- If it encounters a save position, **it converts it to a save-commit unit position** and stops the traversal.
  - Here, we reuse the legacy save unit positions to avoid placing speculator and save-commit units in series (which should be prevented, as discussed in Section 7.2.1). Without this adjustment, save-commit units would always end up placed before branches.

By the end of this step, **all save units within the loop are converted to save-commit units**, and **all save-commit positions are determined**.

The example circuit now looks as follows: It has determined a total of eight save-commit positions, including four positions converted from save positions.

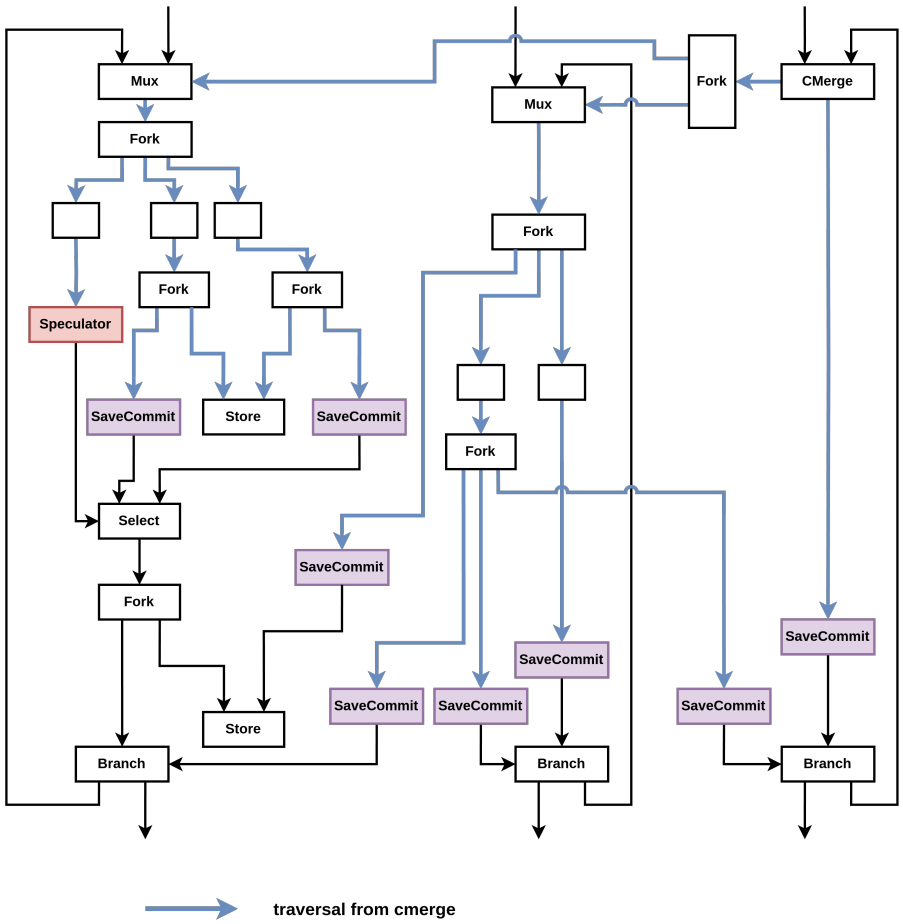


Figure 33: Save-commit positions are determined.

11.2.3. Identify Regular Commit Units

The algorithm next identifies regular commit units, needed to prevent side effects. It performs DFS traversals, starting from positions of the speculator or save-commit units. This time, the traversal passes over branches and explores the entire speculative region, even outside the loop.

- If the algorithm encounters a speculator or save-commit unit, it stops traversal, as this marks the end of the speculative region (see Section 2.4).
- If the algorithm encounters units with side effects (e.g., `StoreOp`, `MemoryControllerOp`, or `EndOp`), it places a commit unit and stops the traversal.

As a result, the example circuit now looks as follows: four commit units have been placed.

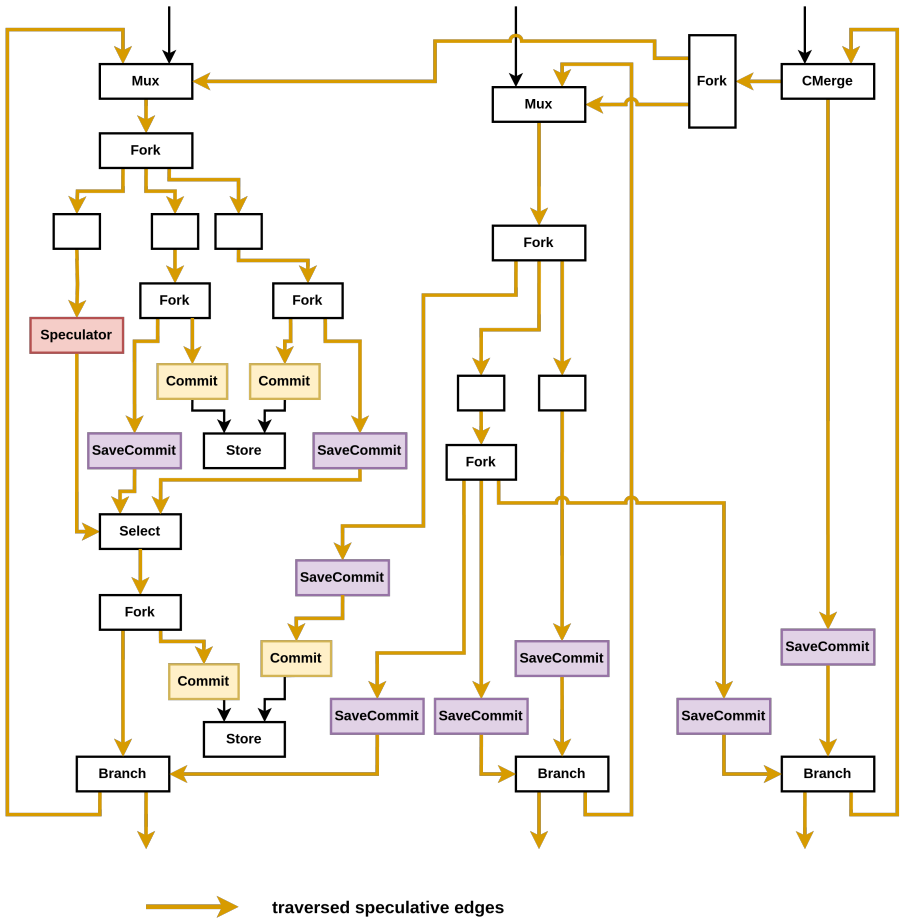


Figure 34: Regular commit positions are determined.

11.2.4. Identify Commit Units Between Basic Blocks

Finally, the algorithm identifies commit units between basic blocks to prevent token reordering.

It performs the same DFS traversal starting from the speculator or save-commit units, exploring the entire speculative region, and marking speculative edges.

Next, it uses a library function in `CFG.cpp` to find edges between basic blocks. For any set of edges going to the same basic block, if **two or more** edges from **different basic blocks** are speculative, it places commit units on all speculative edges in that set (Figure 35).

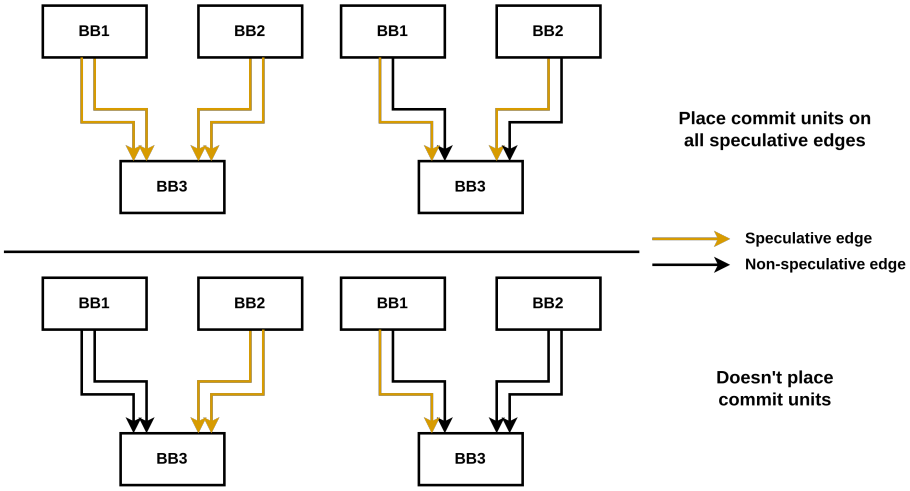


Figure 35: Commit units are placed between basic blocks based on specific conditions.

Loop backedges, including those going to and from the same basic block, are considered edges between basic blocks. However, we make an exception by skipping the placement of commit units on the innermost loop backedges, to maintain high throughput in nested loop cases where II = 1 (Section 5.2). It results in the placement of Figure 36.

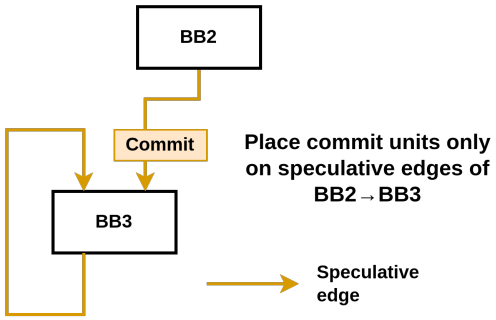


Figure 36: Commit units are not placed on the innermost loop backedges.

After placing commit units between basic blocks, regular commit units may now be outside the speculative region. We then perform the same traversal again to remove all commit units that are not reached.

11.2.5. Placement Is Not Optimized

You may notice that the resulting circuit in Figure 34 is not optimized, given that save-commits are relatively expensive. The circuit includes a large number of them.

Figure 37 demonstrates better placements. At the top of the figure, we see the circuit resulting from the algorithm, while the circuits at the bottom reduce the number of save-commit units. The bottom (b) reduces it by 4, but with this optimization the circuit requires recalculating values on misspeculation, which increases power consumption or latency. The bottom (c) represents a more balanced approach, targeting save-commit units that share the same tokens, reducing the number of save-commit units by 2.

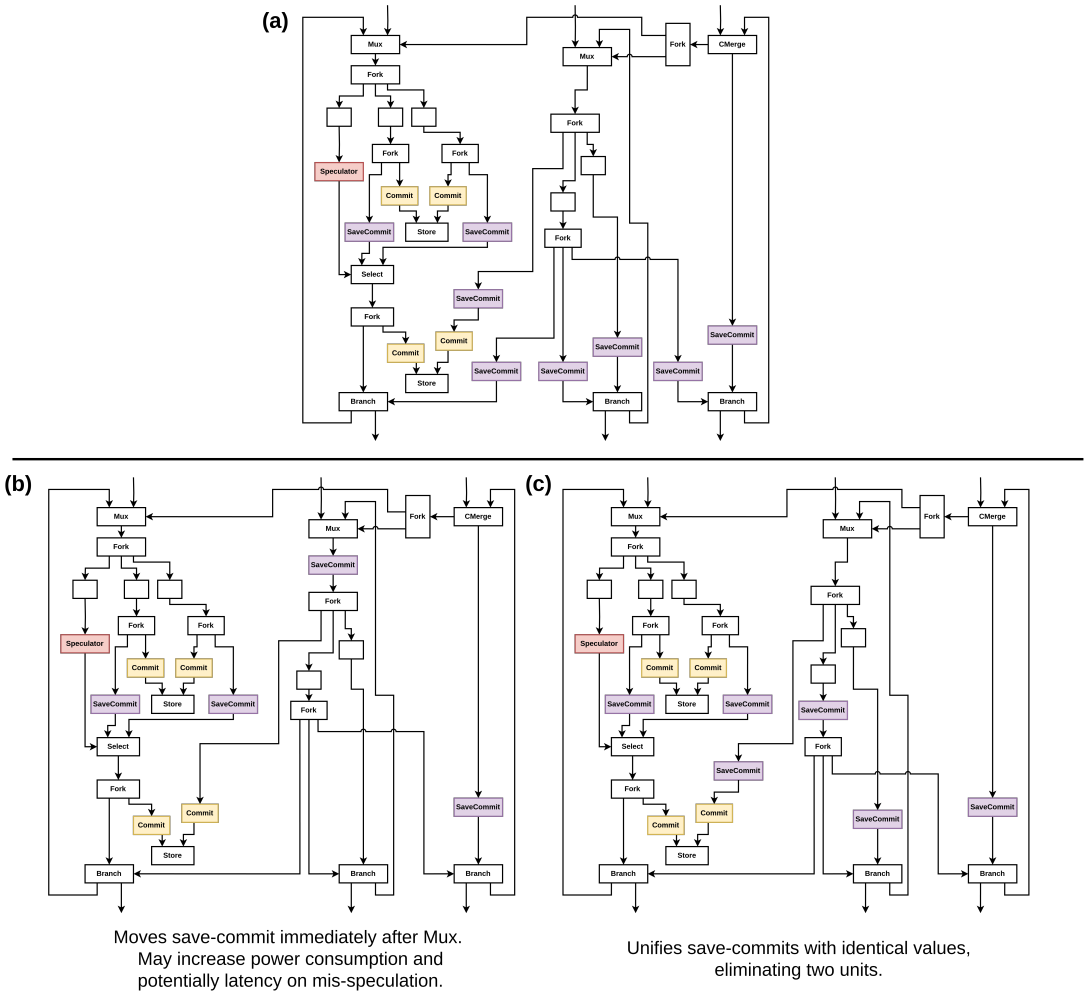


Figure 37: Optimized placements of save-commit units.

These optimizations are not yet implemented in the current algorithm and are left as future work.

11.3. Speculative Units Placement and Connection

Once the placements are finalized, we proceed to place the speculative units and connect them to the speculator. The overall procedure is as follows:

1. Place the speculator.
2. Compute the save-commit control network.
3. Place the save-commit units.
4. Place the commit units.
5. Route the commit control signals.

11.3.1. Order of Unit Placement

The placement order is important and must be carefully followed:

- The **speculator** is placed first, as other units need to reference it during connection.
- The **save-commit units** must be placed before the **commit units**. This is due to a technical detail: we specify the **operand** (i.e., the input edge of an operation) as the placement position. If both a save-commit and a commit unit are placed on the same operand, the one placed first will appear above the other. Only the configuration where

the save-commit unit is above the commit unit is valid, so we place save-commit units first.

11.3.2. Connection with Speculator

The save-commit control network is constructed to match the structure shown in Figure 27, as described in Section 8.1. This is relatively simple (without recursion), and the control value is shared across all save-commit units, unlike the commit control network. This is why the save-commit control network is calculated before placement.

For commit control, following the approach discussed in Section 8.2, we construct the control network by replicating the branching structure of the dataflow graph. The traversal strategy is as follows:

- Start from the speculator and all save-commit units.
- Replicate any branch nodes encountered.
- Stop traversal if another speculator or save-commit unit is encountered (to avoid token mismatches as noted in Section 8.2).
- Connect the control signal and stop traversal at commit units.

A naive DFS that replicates branches immediately upon encountering them may produce redundant units, if the traversal path doesn't ultimately reach any commit unit. To avoid this, we delay creating branch units until reaching a commit unit. We maintain a stack of encountered branch information and only instantiate the corresponding branches once a commit unit is found.

12. Representation of Spec Bit

Speculative units need to determine whether an incoming token is speculative. To support this, the **spec bit**—introduced in Section 2.3—is used to annotate edges within the speculative region. This section explains how the spec bit is represented within the MLIR framework, and how the algorithm attaches it to specific edges.

We begin by reviewing key MLIR concepts as they relate to Dynamatic, with a particular focus on its type system. Next, we explain the **extra signal system**, which extends Dynamatic's existing type system. The spec bit is then introduced as a specific use case of this system. These parts are intentionally technical to help ground the explanation in concrete examples. Finally, we describe the algorithm for attaching spec bits to edges in the speculative region.

12.1. MLIR Concepts

To set the stage for how spec bits are represented in MLIR, we introduce the fundamental MLIR concepts as used in our compiler—focusing especially on the type system.

12.1.1. Operations in Dynamatic

Operations represent primitive computations in the IR, such as `add` or `branch`. From the perspective of the dataflow graph, they correspond to its vertices. In our Handshake IR, each operation (e.g., `MergeOp`, `ConstantOp`) maps to an individual RTL module. During RTL generation, these modules are instantiated at the netlist and are connected according to the structure of the IR.

In this section, we reference concrete examples from the codebase, linking to relevant TableGen files that declaratively define the operations (e.g., `HandshakeOps.td`, `HandshakeArithOps.td`).

Each operation has **arguments**, which fall into three categories: operands, attributes, and properties. Here, we focus solely on **operands**. Operands represent the edges in the dataflow graph and, at the RTL level, correspond to the input channels of the corresponding module.

Note: For simplicity, we do not distinguish between operands and **values** here, assuming the IR is materialized.

For example, `ConditionalBranchOp` has two operands: one for the condition and one for the data (see the code).

```
let arguments = (ins ChannelType:$conditionOperand,
                     HandshakeType:$dataOperand);
```

Some operands are **variadic**, meaning they can have a variable number of inputs. For example, the data operand of `MuxOp` is variadic (see the code).

```
let arguments = (ins ChannelType:$selectOperand,
                     Variadic<HandshakeType>:$dataOperands);
```

For more information on operation arguments, please refer to the MLIR documentation: https://mlir.llvm.org/docs/DefiningDialects/Operations/#operation-arguments

Each operation also has **results**, which represent output channels at the RTL level. For instance, `ConditionalBranchOp` has two results, corresponding to the "true" and "false" branches (see the code).

```
let results = (outs HandshakeType:$trueResult,
                    HandshakeType:$falseResult);
```

Just like operands, some results are variadic (e.g., outputs of `ForkOp`).

For more information on operation results, please refer to the MLIR documentation: https://mlir.llvm.org/docs/DefiningDialects/Operations/#operation-results

12.1.2. Types in Dynamatic

Types define the kind of operands and results an operation accepts or returns (e.g., `i32`, `i1`, `f32`). In the previous section, you may have noticed that operands and results were declared with types like `HandshakeType` or `ChannelType`. These are **custom types** used in Dynamatic's Handshake IR to describe the kind of RTL channels.

There are three main types in the Handshake IR:

- **`ChannelType`** – Represents a data port with **data + valid + ready** signals.
- **`ControlType`** – Represents a control port with **valid + ready** signals.
- **`HandshakeType`** – Can be either a `ChannelType` or a `ControlType`.

These types are defined in `HandshakeTypes.td`.

When an operation is instantiated, the actual operands have concrete **instances** of these types. For example, an operand of `AddIOp` (integer addition) has a `ChannelType`, meaning its actual type will be:

- `!handshake.channel<i32>` (for 32-bit integers)
- `!handshake.channel<i8>` (for 8-bit integers)
- and so on.

You can think of `ChannelType` as similar to a *type constructor* or a *type class* in other languages —it takes another type (like `i32`) as a parameter to specify the kind of data it carries.

12.1.3. Traits

Traits are attached to operations for a variety of purposes—categorizing ops, indicating certain features, or enforcing invariants. Here, we focus on their role in **type verification**.

For example, In `CompareOp`, the `lhs`/`rhs` operands must have the same type instance (e.g., `!handshake.channel<i32>`). However, simply specifying `ChannelType` for each is not enough—without additional constraints, the operation could exist with mismatched types, like:

- `lhs: !handshake.channel<i8>`
- `rhs: !handshake.channel<i32>`

To enforce type consistency, we apply the **`AllTypesMatch`** trait (see the code):

```
AllTypesMatch<["lhs", "rhs"]>,
```

This guarantees that both operands share the exact same type instance.

Note that this type verification is *continuous* with ordinary type checking. Most languages check whether the types of `a` and `b` match when `a = b`, or whether a certain relationship holds between them. Similarly, when `c ? a : b` is evaluated, the type relationship between `a` and `b` is also considered. The trait can be seen as a more general form of this, allowing us to specify arbitrary relationships between the types of operands and results.

Traits are sometimes called **multi-entity constraints** because they enforce relationships across multiple operands or results.

In contrast, types (or type constraints) are called **single-entity constraints** as they enforce properties on individual elements.

It's worth noting that we sometimes use traits even in single-entity cases for consistency. For example, `IsIntChannel<channelName>` ensures the channel's data type is `IntegerType` (e.g., `i32` or `i8`).

More on constraints: https://mlir.llvm.org/docs/DefiningDialects/Operations/#constraints

12.2. Extra Signals

We extend the type system described above to support **extra signals**—an array of name–type pairs.

Extra signals can be added to both `ChannelType` and `ControlType`. For example:

- `!handshake.channel<i32, [spec: i1]>` — a channel carrying `i32` data and an extra `spec` signal of type `i1`.
- `!handshake.control<[spec: i1, tag: i8]>` — a control channel with two extra signals: `spec` (`i1`) and `tag` (`i8`).

12.2.1. Verification

Since extra signals are part of the type, they naturally become subject to type verification. This allows us to ensure that extra signals are correctly propagated through the circuit, or processed properly by individual operations.

To support this, we introduced a set of **custom traits** that provide verification of extra signal behavior. For example:

- **`AllDataTypesMatch`** — Ensures data types match while ignoring differences in extra signals.
- **`AllExtraSignalsMatch`** — Ensures the extra signals match while ignoring the data type.

These traits make our type system more expressive and robust, allowing precise control over extra signal handling.

12.2.2. Applying Traits to Operations

Next, we explore how these traits are applied to different operations to enforce correct handling of extra signals.

12.2.2.1. Regular Cases

Most operations are expected to have **consistent extra signals** across all their inputs and outputs, including `AddIOp`, `ForkOp`, or `ConditionalBranchOp`.

To further specify the meaning of "consistent extra signals across all their inputs and outputs", we provide an example: if one of the inputs to `AddIOp` carries an extra signal, such as `spec: i1`, then the other input and the output must also have the same extra signal, `spec: i1`.

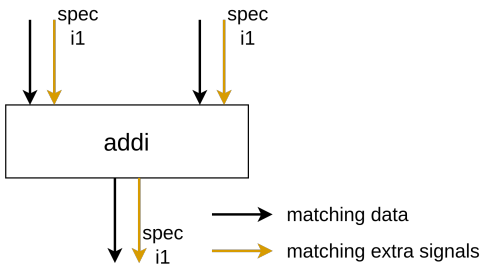


Figure 38: `AddIOp` has consistent extra signals.

This is enforced for the following reasons:

- To reduce variability in these operations, simplifying RTL generation.
- To impose a built-in constraint: we aim to enforce the `AllTypesMatch` trait as much as possible. This special built-in trait simplifies the IR format under the declarative assembly format and enables a simpler builder.

Note that this constraint applies to the **presence** of the extra signals—not to their **values**. For instance, in the `addi` example, one input's `spec` signal might hold the value `1`, while the other input's `spec` signal could hold `0`. The RTL implementation of `addi` must account for and handle these cases appropriately.

In practice, most operations enforce this constraint using the `AllTypesMatch` trait, which ensures the entire type consistency across channels. However, in cases where operands and results differ in their base data types, we instead use the `AllExtraSignalsMatch` trait to enforce consistency only for the extra signals.

For example, consider `ConditionalBranchOp`, which accepts a condition of type `i1` and a data operand of arbitrary type. Its constraints are declared as follows (see the code):

```
  AllTypesMatch<["dataOperand", "trueResult", "falseResult"]>,
  AllExtraSignalsMatch<["conditionOperand", "dataOperand", "trueResult",
"falseResult"]>,
  IsIntSizedChannel<1, "conditionOperand">,
```

Here, `AllTypesMatch` doesn't apply to `conditionOperand`, while `AllExtraSignalsMatch` ensures that `conditionOperand`, `dataOperand`, `trueResult`, and `falseResult` all share the same set of extra signals, regardless of their data types.

12.2.2.2. MuxOp and CMergeOp

For both `MuxOp` and `CMergeOp`, the index channel is **simple**, meaning it does not carry any extra signals. Based on current needs, extra signals of the index channel are unnecessary. Therefore, we chose not to support them at this stage. Support can be added easily in the future if needed.

All data inputs and the data output must carry the same set of extra signals. As a result, `MuxOp` and `CMergeOp` follow the structure shown below:

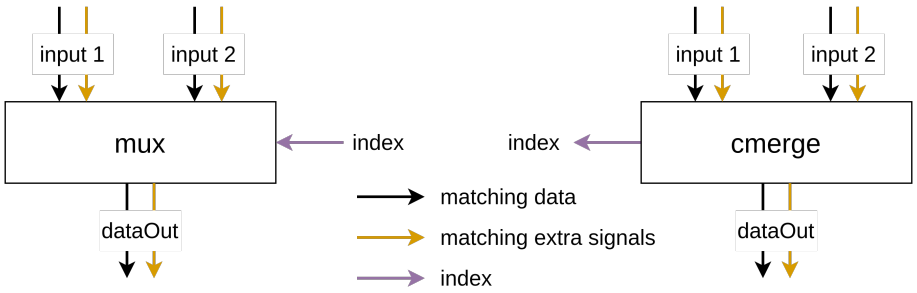


Figure 39: Type constraints for `MuxOp` and `CMergeOp`.

12.2.2.3. MemPortOp (Load and Store)

The `MemPortOp` operations, such as load and store, communicate directly with a memory controller or a load-store queue (LSQ). The channels connected to these operations must be simple.

This design ensures that the memory controller can focus solely on managing memory access, while the responsibility for handling extra signals lies with the `MemPortOp`.

For the load operation, the structure is as follows:

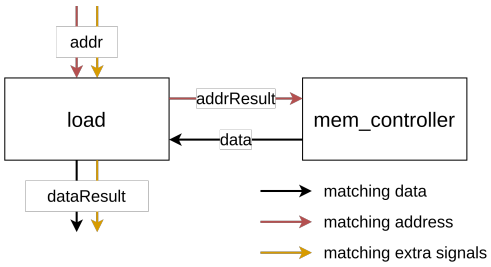


Figure 40: Type constraints for `LoadOp` operation.

- The `addrResult` and `data` channels, used to communicate with the memory controller, must be simple.
- The `addr` and `dataResult` channels must carry the same set of extra signals.
- The `addr` and `addrResult` channels must have the matching data type.
- The `data` and `dataResult` channels must have the matching data type.

For the store operation, the structure is:

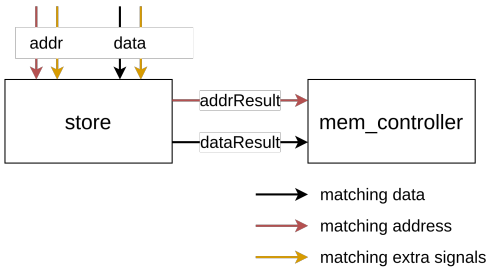


Figure 41: Type constraints for `StoreOp` operation.

- The `addrResult` and `dataResult` channels, which interface with the memory controller, must also be simple.
- The `addr` and `data` channels must have matching extra signals.
- The `addr` and `addrResult` channels must have the matching data type.
- The `data` and `dataResult` channels must have the matching data type.

At the code level, the following constraints are enforced:

- `AllExtraSignalsMatch` – Ensures extra signals match across corresponding channels.
- `IsSimpleHandshake` – Ensures that channels connected to the memory controller do not carry extra signals.
- `AllDataTypesMatch` – Maintains consistency between `addr`/`addrResult` and `data`/`dataResult` data types.

12.2.2.4. ConstantOp

While this operation falls under the regular category (Section 12.2.2.1), it's worth highlighting due to the non-trivial way it handles control tokens with extra signals that trigger the emission of a constant value.

`ConstantOp` has one input (a `ControlType` to trigger the emission) and one output (a `ChannelType`). Like other operations, the extra signals of the input and output should match.

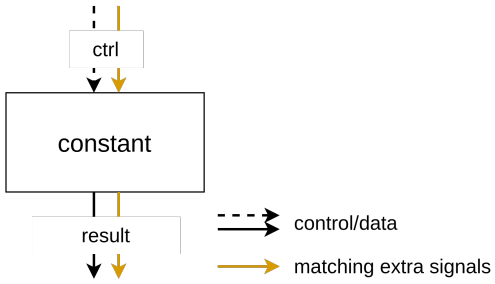


Figure 42: Type constraints for `ConstantOp` operation.

To ensure consistency for succeeding operations, `ConstantOp` must generate an output with extra signals. For example, if an adder expects a `spec` tag, the preceding `ConstantOp` must provide one.

However, since control tokens can now carry extra signals, a control token with extra signals may trigger `ConstantOp` (e.g., in some cases, a token from the basic block's control network is used).

Therefore, we decided to forward the extra signals from the control input directly to the output token, rather than discarding them and hardcoding constant extra signal values in `ConstantOp`.

In other words, `ConstantOp` does not generate extra signals itself—this responsibility typically falls to a dedicated `SourceOp`, which supplies the control token for the succeeding `ConstantOp`. The values of these extra signals depend on the specific signals being propagated and are not discussed here.

12.3. Spec Bit as an Extra Signal

So far, we have discussed the extra signal system in general. As seen in earlier examples, the spec bit is represented as an extra signal for the operand type.

We introduced traits specific to the spec bit to ensure the validity of how it is handled across the circuit.

Here's the general handling of the spec bit within a speculative region:

- For speculator and save-commit units, both the data input and output carry the spec bit.
- For commit units, the spec bit is consumed. The data input carries the spec bit, while the data output does not.
- The save-commit and commit control networks do not carry the spec bit:
  - The control outputs of the speculator do not carry any extra signals (i.e., they are **simple**).
  - The control input of the save-commit and commit units is also simple.
  - Speculating branches used to build the control network (see Section 6.1) consume the `spec` bit of operands, without propagating any extra signals to the results.

12.3.1. Technical Details

We have introduced several traits specific to the spec bit to ensure its correct handling. These traits are applied to speculative units.

Traits:

- `HasValidSpecTag<string name>`: Verifies that the operand or result specified by the name has a valid `spec` extra signal of type `i1`.
- `LacksSpecTag<string name>`: Ensures that the operand or result specified by the name does not have an extra signal named `spec`.
- `AllExtraSignalsMatchExcept<string except, list<string> names>`: A general trait that ensures all extra signals match across the specified operands or results, except for the one specified by `except`.

The following explains how these traits are applied to speculative units:

Speculator:

- `HasValidSpecTag<"dataIn">`, `HasValidSpecTag<"dataOut">`: Since the speculator is placed within a loop, both `dataIn` and `dataOut` must carry a valid `spec` extra signal.
- `AllTypesMatch<["dataIn", "dataOut"]>`: Ensures that the types of `dataIn` and `dataOut` match, including the `spec` extra signal.
- `IsSimpleHandshake<"...">`: The control outputs of the speculator must be simple, meaning they do not carry any extra signals.

SaveCommit: Same as the speculator.

- `HasValidSpecTag<"dataIn">`, `HasValidSpecTag<"dataOut">`
- `AllTypesMatch<["dataIn", "dataOut"]>`
- `IsSimpleHandshake<"ctrl">`: The control input does not carry any extra signals.

Commit:

- `HasValidSpecTag<"dataIn">`: The `dataIn` operand must carry a valid `spec` extra signal.
- `LacksSpecTag<"dataOut">`: The `dataOut` operand must not carry the `spec` extra signal.
- `AllDataTypesMatch<["dataIn", "dataOut"]>`: Ensures that the data types of `dataIn` and `dataOut` match.
- `AllExtraSignalsMatchExcept<"spec", ["dataIn", "dataOut"]>`: Ensures that the extra signals of `dataIn` and `dataOut` match, except for the `spec` extra signal.
- `IsSimpleHandshake<"ctrl">`: The control input does not carry any extra signals.

SpeculatingBranch:

Since the speculating branches are used for the speculative control network, the outputs do not carry any extra signals, while the inputs carry `spec` bits.

The `SpeculatingBranch` has two operands, `dataOperand` (data) and `tagFromOperand` (condition using the spec bit), and two results, `trueResult` and `falseResult`.

- `HasValidSpecTag<"tagFromOperand">`, `HasValidSpecTag<"dataOperand">`: Both operands must carry a valid `spec` extra signal.
- `IsSimpleHandshake<"trueResult">`, `IsSimpleHandshake<"falseResult">`: Both results must not carry any extra signal.
- `AllDataTypesMatch<["dataOperand", "trueResult", "falseResult"]>`: Ensures that the data types of `dataOperand`, `trueResult`, and `falseResult` match.

12.4. Further Details

The MLIR documentation might be complex, but it covers the key concepts well. You can check out the following links for more details:

https://mlir.llvm.org/docs/DefiningDialects/Operations

https://mlir.llvm.org/docs/DefiningDialects/AttributesAndTypes

12.5. `addSpecTag` Algorithm

This section outlines the algorithm used to add the spec bit to operands and results within a speculative region. This algorithm is implemented in the speculation pass and performed as the final step.

While type verification ensures that the circuit correctly carries extra signals, including the spec bit, it does not automatically update or infer them. Therefore, an explicit algorithm is required to handle this task.

The algorithm is explained using the example circuit in Figure 43.

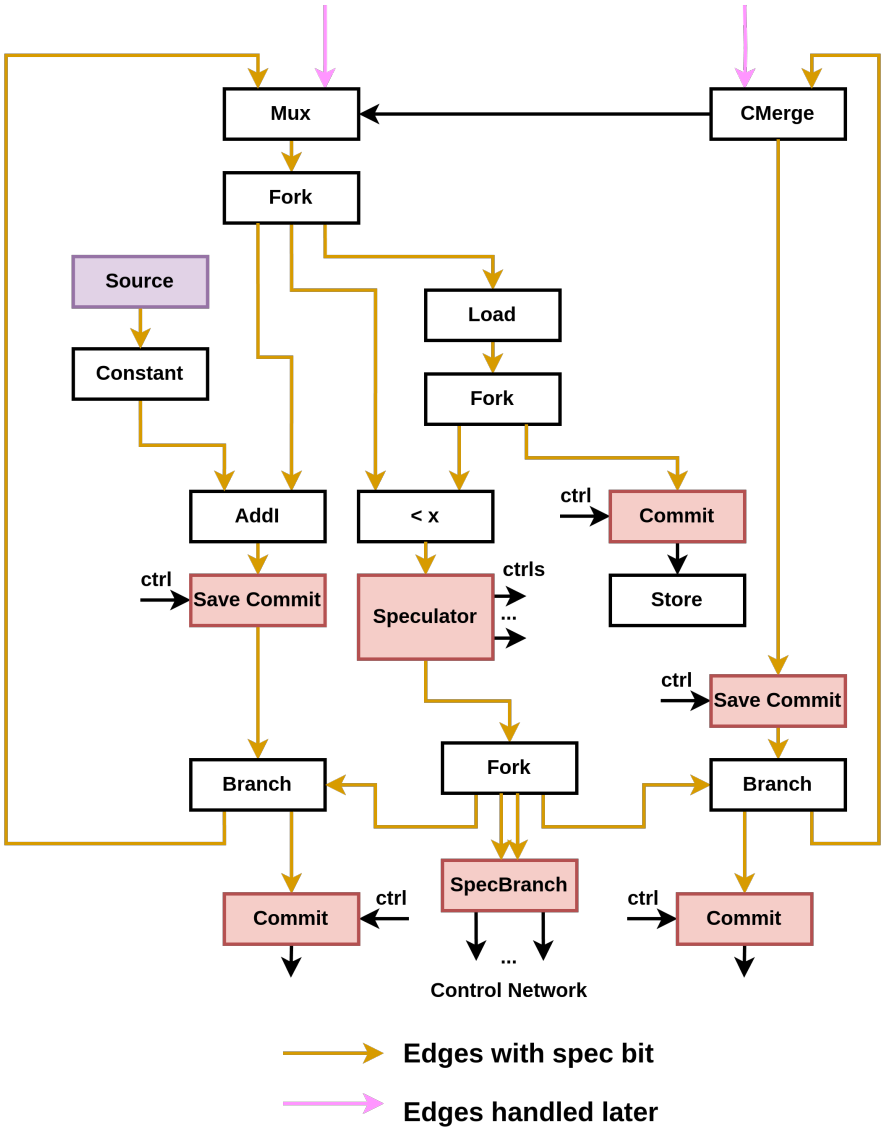


Figure 43: Example circuit illustrating the `addSpecTag` algorithm.

The traversal in this algorithm explores the entire speculative region, similar to previous algorithms, but with key differences:

- The traversal starts **only** from the speculator.
- It performs both **upstream** and **downstream** traversals.
- It **passes through** save-commit units.

This approach is necessary due to the presence of **source units** within speculative regions. These units are omitted in most of the earlier figures but are usually present before constant units, to provide constant control signals. Source units have no predecessors and also define the entry point of the speculative region.

To satisfy the type constraints for consistent extra signals (see Section 12.2.2.1) at the `AddI` unit in Figure 43, we must traverse source units and mark them to generate control tokens with a spec bit. Upstream traversal ensures that source units are reached (Figure 44); without it, handling source units gets complicated.

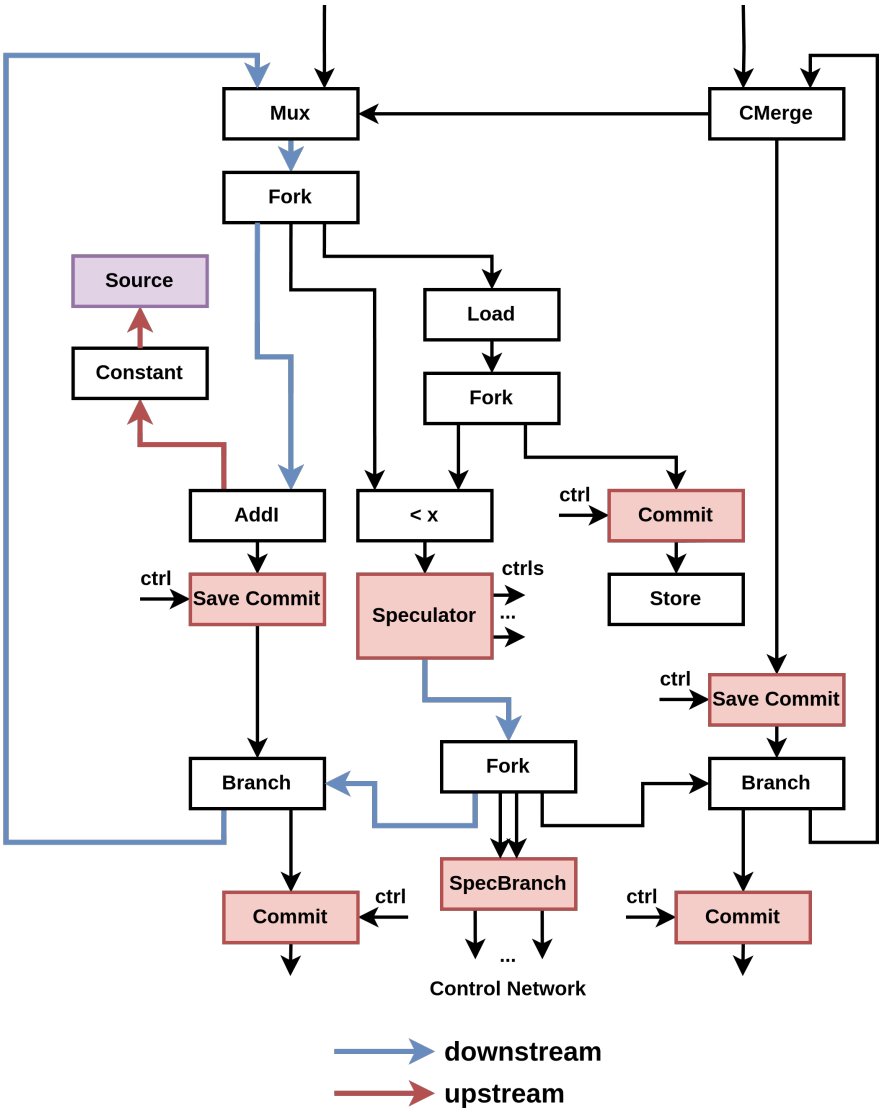


Figure 44: Example of a traversal reaching the source unit.

Here are other characteristics of the algorithm:

- Downstream traversal stops at the commit unit or the speculating branch (which forms the start of the control network).
- Upstream traversal stops at the `CMerge` or `Mux` units (as some operands originate outside the speculative region, and will be handled later). The loop backedge is covered by downstream traversal from the branches.
- The control network for save-commit and commit units is not traversed.

Finally, to ensure consistency of extra signals at `Mux` and `CMerge` units (Section 12.2.2.2), a unit called `NonSpec` is placed on their data operands that are not traversed (Figure 45). `NonSpec` is a simple unit that accepts an input without a spec bit and outputs a token with a spec bit. The data is forwarded unchanged, while the added spec bit is always set to `0`, indicating that the token is not speculative.

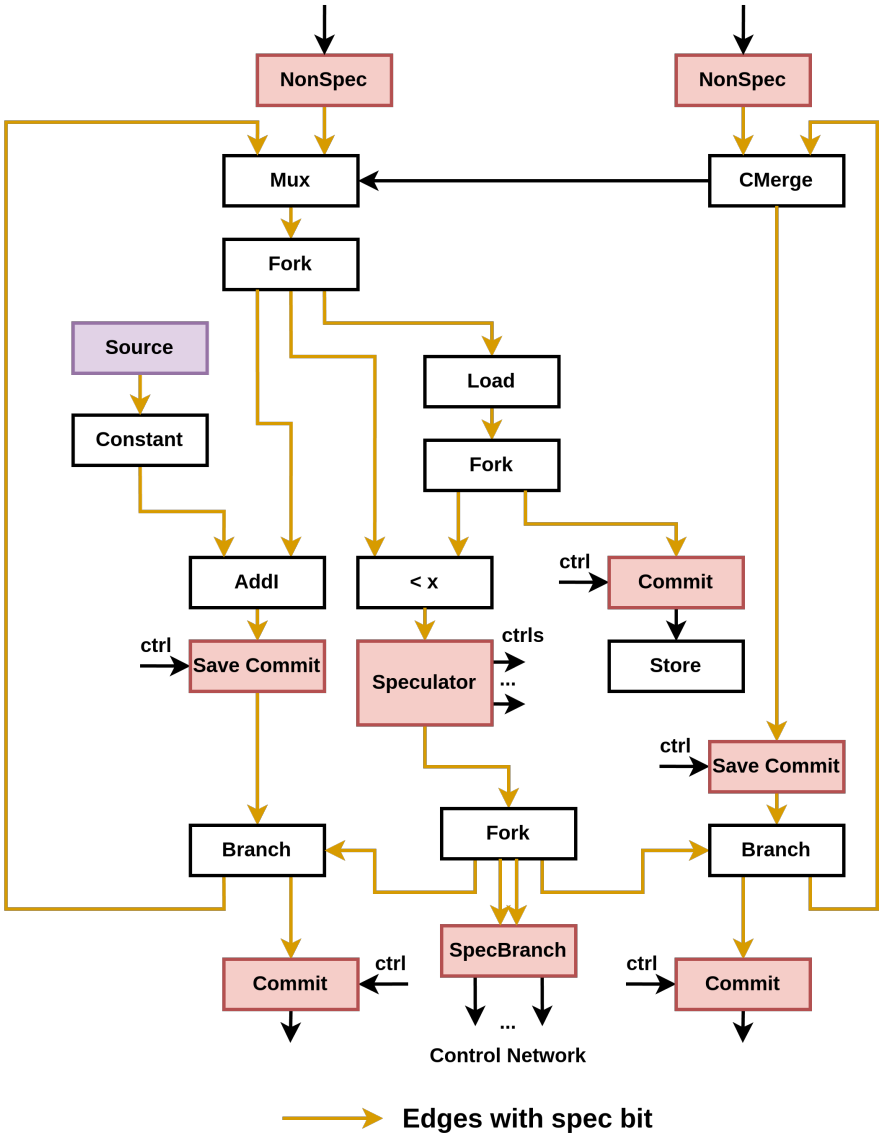


Figure 45: Final circuit with spec bit satisfying the type constraints.

13. Manual Buffer Placement

After the speculation pass is applied, we manually insert additional buffers. This process involves three categories of buffer placement:

1. Buffers required to prevent deadlock
2. Buffers for throughput improvement (speculation-specific)
3. Buffers for throughput improvement (general)

If the speculation pass runs before buffer placement, the existing buffer placement pass can handle category 3. However, categories 1 and 2 would require a separate process tailored to speculation. The automation of these works is left as future work.

Assume we've successfully inserted speculative units in the circuit shown below (Figure 46). This example expands on Figure 1 with additional details, including buffer slots, a speculator trigger, and a precise `StoreOp` interface.

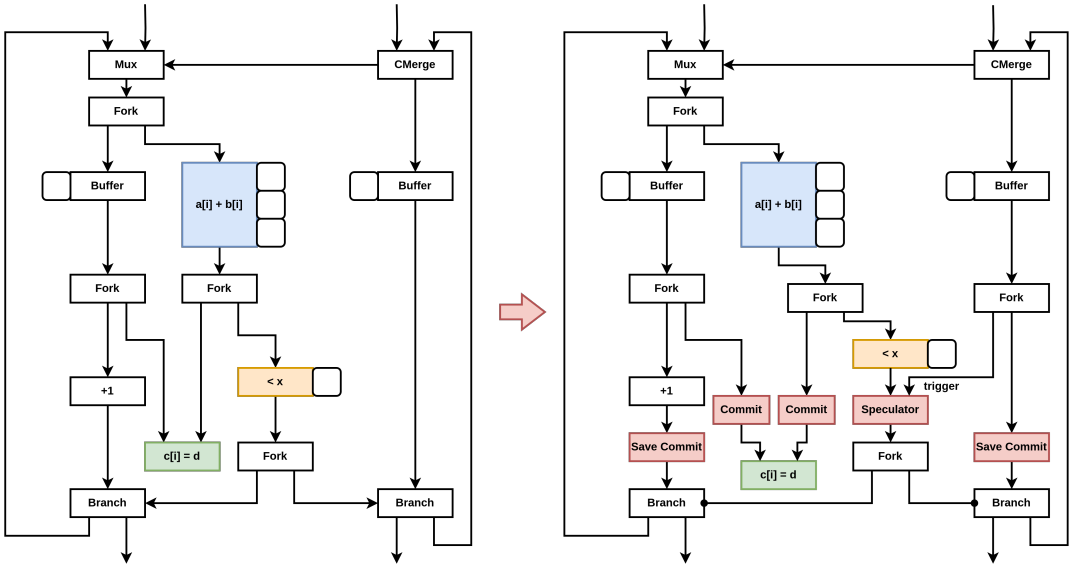


Figure 46: Example circuit for illustrating buffer placement.

Some operation slots are also shown—for instance, the three boxes beside `a[i] + b[i]` indicate that this operation takes three cycles and is pipelined.

Note that the trigger is taken from the fork below the buffer to align the timing between the speculator making a speculation and the save-commit units passing a token.

13.1. Buffers Needed to Prevent Deadlock

As it stands, the circuit in Figure 46 does not work with speculative units alone. To understand why, let's examine the control network connecting the save-commit and commit units (Figure 47).

In this network, there are problematic branches. For the commit units, after the speculating branch, the loop condition joins `commitCtrl` at a regular branch. Whereas the loop condition is generated every time the speculator is triggered, `commitCtrl` arrives later—only after the speculator receives the actual token. Similarly, for the save-commit units, the loop condition joins the `isMisspec` signal, which also arrives late. Because of this, these branch units are unable to accept the loop condition immediately, causing the speculator's `dataOut` to stall.

Worse still, when `dataOut` stalls, **the speculator cannot send its next control signals**. Internally, the speculator's Core cannot emit its next internal control signal until the previous one is accepted. Therefore, `commitCtrl` and `isMisspec` are never sent, even after the actual token arrives, leading to a real deadlock scenario.

To prevent this, we need to insert buffers. Importantly, the number of required buffer slots depends on two factors:

- The number of cycles between when speculation is triggered and when it is resolved.
- The target II with speculation.

Suppose that only one buffer slot is placed, and the actual token does not arrive in the next cycle. A deadlock occurs if the speculator is triggered again before the first speculation is resolved.

Assuming the circuit achieves an II of 1, three buffer slots are needed to store tokens until their corresponding speculations are resolved (Figure 48).

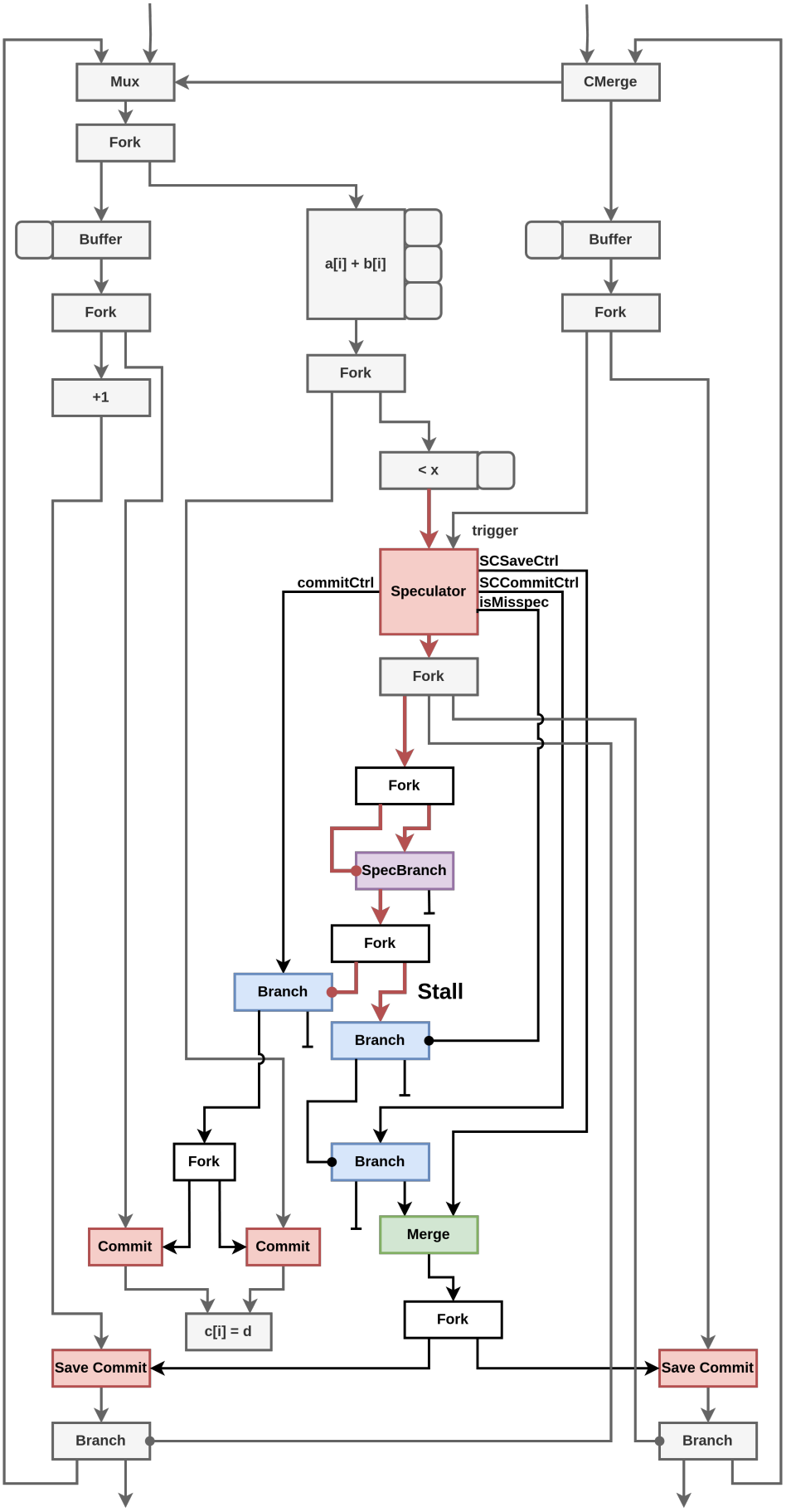


Figure 47: Control network of the speculative units in the example circuit.

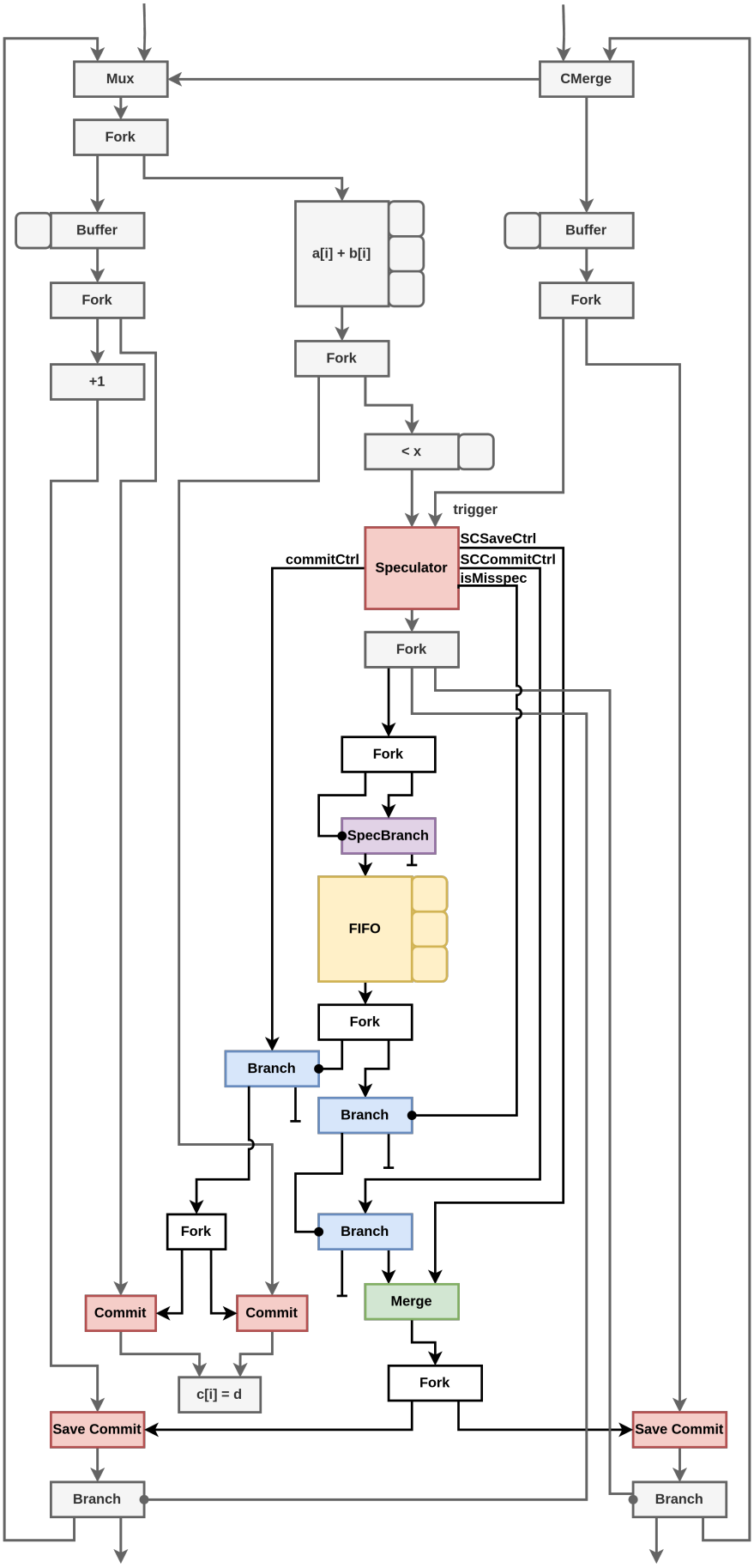


Figure 48: Required buffers inserted to prevent deadlock with II = 1.

If the circuit only achieves an II of 2, then $\lceil \frac{3}{2} \rceil = \lceil 1.5 \rceil = 2$ buffer slots are sufficient.

13.1.1. Redesigning of Speculator

We can prevent deadlocks caused by insufficient buffer placement by redesigning the speculator. The root issue is that, although the speculator emits various control signals on different output channels, internally it relies on a single monolithic control bus. If the signal indicating the comparison outcome could bypass the signals related to token emission, this issue could be resolved.

13.2. Buffers Needed to Gain Performance Improvement

The second type of buffer placement focuses on improving performance. In general, when targeting a lower II, additional buffers are often required. While the existing buffer placement algorithm may help with this, some speculation-specific handling is still necessary.

Suppose we've already placed the necessary buffers in the control network to prevent deadlock. Even so, the circuit on the right in Figure 46 cannot achieve an II of 1. The bottleneck lies in the commit units: they can only accept incoming speculative tokens after receiving control signals from the speculator.

In Figure 49, we insert sufficient buffers before the commit units. As before, the number of buffers depends on both the speculation resolution latency and the target II. In this case, to achieve II = 1, we balance the buffer slots between before the speculator and before the commit units.

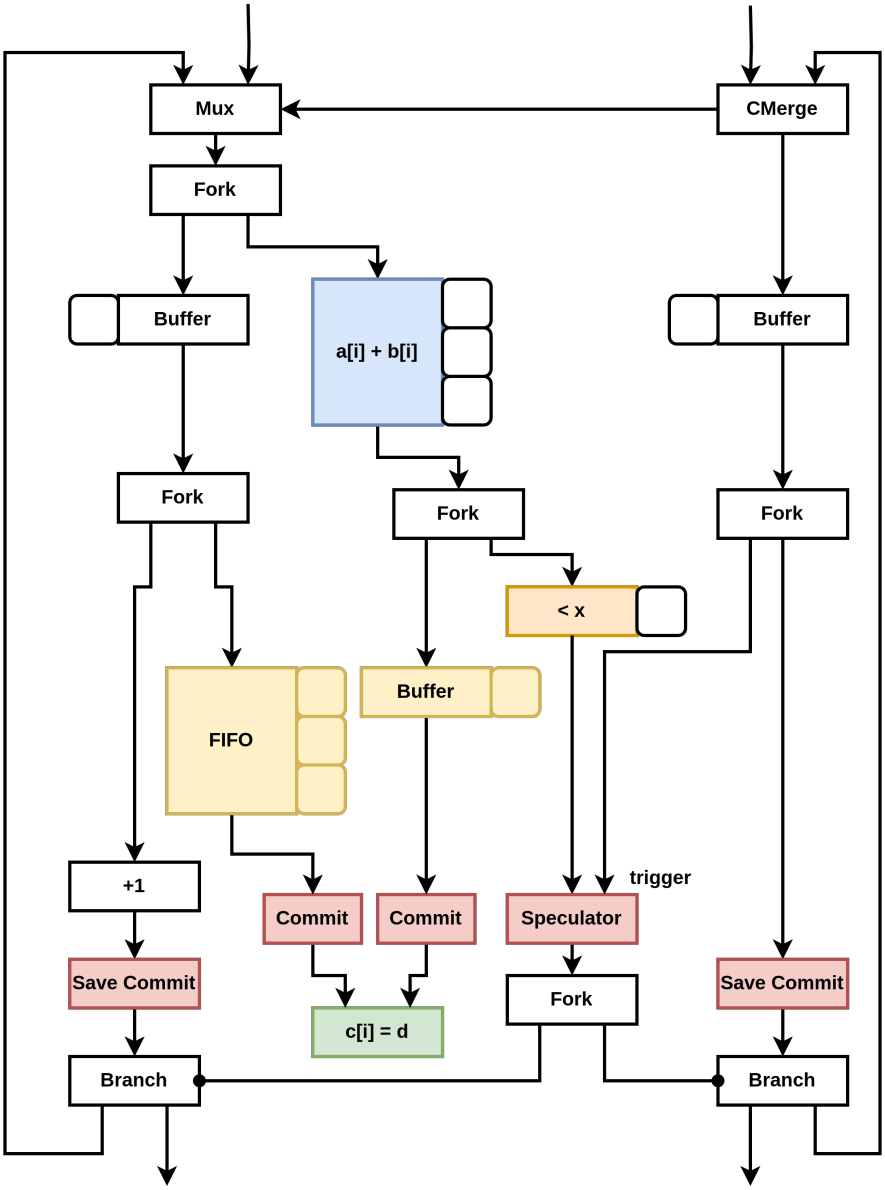


Figure 49: Placed buffers before commit units.

With these buffers, the circuit can now reach the desired II of 1.

Note that in some cases, buffers also need to be placed on the `ctrl` channel of save-commit units to reach optimal throughput. However, that's not necessary here—since the trigger is taken from below the buffer, the timing of making a speculation and passing tokens through the save-commit units is already aligned.

13.2.1. Buffers Before Commit Units Also Help Prevent Deadlock

In the example, the circuit *happens to work* correctly—albeit with a degraded initiation interval—without sufficient buffering before the commit units. This is because a single buffer exists between the speculator and the commit units, and further speculation is naturally stalled by backpressure from the commit units.

However, in some cases, buffers before the commit units are also essential to prevent deadlock. In this sense, **the roles of buffers in deadlock prevention and performance improvement are not entirely distinct**.

The core issue is a circular dependency involving `commitCtrl`. Without appropriate buffering, some tokens must wait for a `commitCtrl` token to be accepted, but the `commitCtrl` token itself cannot be issued until those tokens are accepted.

This circularity is not confined to the control network—the case discussed in Section 13.1. In Figure 50, the `dataOut` from the speculator cannot be accepted until `commitCtrl` is issued, without sufficient buffering. However, `commitCtrl` cannot be issued until `dataOut` is accepted, due to the speculator's internal control bus.

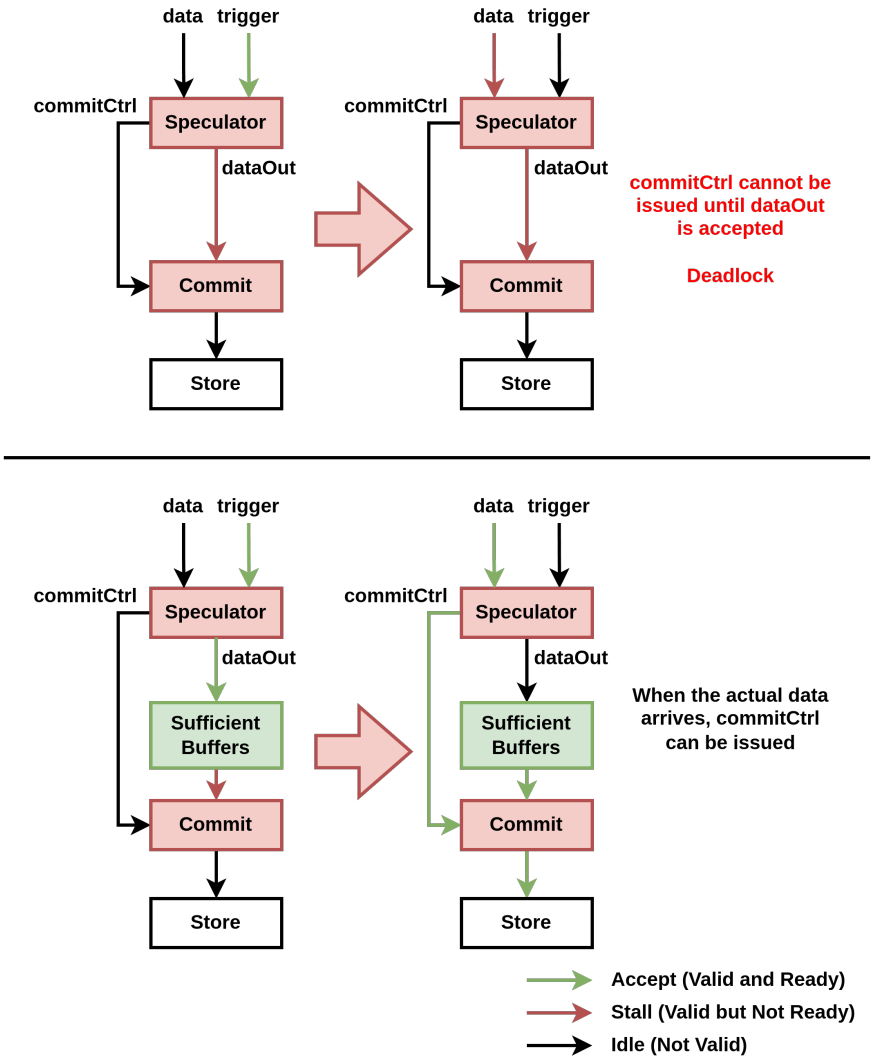


Figure 50: Buffers needed at `commitCtrl` convergence points to prevent circular dependencies.

Ultimately, deadlock prevention relies on redesigning the speculator's internal control bus, as discussed in Section 13.1.1.

13.3. Buffers Inside Speculator and Save-Commits

The speculator has an internal prediction FIFO to store unresolved predictions, and the save-commit units use a ring buffer to store unresolved speculative tokens. The size of these buffers is manually specified, and similar to other buffers, it involves the target II and speculation resolution latency. Typically, these buffers are the same size.

In the example circuit, the buffer size should be 4. Although the resolution latency is 3 cycles, we need one additional empty slot, as is typical with ring buffers.

Note that a **deadlock may occur** if the speculator has sufficient slots available but the save-commit units do not. This happens because the speculator may issue `PASS_KILL` tokens —combinations of `PASS` and `KILL`—to the save-commit units. They can only accept such tokens if they have empty slots, due to the behavior of `PASS`. When the save-commit units are full, they require the `KILL` functionality of `PASS_KILL` to free up space—but since they cannot accept the control token in the first place, the `KILL` cannot be processed, resulting in a deadlock.

The solution remains to **separate the signal buses** used for resolving previous speculation (`KILL`) and initiating new speculation (`PASS`), as previously discussed in Section 13.1.1. This enables independent handling of `KILL` and `PASS`.

14. Generative Backend

Finally, we describe how the circuit is generated from the IR extended with extra signals. To support these additions, we developed a new backend specifically designed to handle extra signals, including the `spec` bit. This section also outlines the design choices made during this development.

14.1. Propagation of the spec Bit

Thanks to the algorithm introduced in Section 12.5, every edge within the speculative region carries a `spec` bit. This bit indicates whether the associated token is speculative, defined as follows:

spec bit	Meaning
0	Non-speculative
1	Speculative

Table 4: Meaning of the `spec` bit.

Now, how is the `spec` bit propagated through each unit, especially when multiple inputs with `spec` bits are present?

This logic is handled at the RTL level. When a unit has multiple inputs, we apply a **bitwise `OR`** to all `spec` bits. In most cases, speculator and save-commit units synchronize their `spec` bits (see Section 6.3.2), so all `spec` bit inputs tend to carry the same value. However, as discussed in Section 12.5, source units can also initiate speculative regions. These source units always emit control signals with `spec = 0`, indicating non-speculative tokens. Therefore, an `OR` operation is required to combine speculative and non-speculative signals correctly.

More advanced analysis could allow us to selectively use a `spec` bit from a single input —ignoring those generated by source units—thus eliminating the need for an `OR`. This optimization is left for future work.

14.2. How Should We Handle Extra Signals?

There are two main approaches to handling extra signals in the IR:

- Treat extra signals as independent from the original op
- Treat extra signals as part of the original op

In the first approach, extra signals are introduced independently. For example, a unit like `addi` is initially instantiated with inputs and outputs that include extra signals (e.g., `spec: i1`). Then, in a separate pass, the op is rewritten to remove the extra signal from `addi`, with the signal handled independently by auxiliary ops.

However, since these extra signals do not carry the handshaking signals, this introduces a **signal-level representation** into the IR—contrasting with the **handshake-level representation** used by default. This shift not only complicates the IR specification, but also breaks assumptions made by other passes. For instance, the existing buffering pass attempts to choose buffer positions from nearly all edges, but edges lacking handshake logic do not support buffering.

An alternative would be to define a new IR specifically for signal-level representation. However, we consider this a high-cost solution with limited benefit. From a practical standpoint, we also want to maintain a direct mapping between Handshake IR ops and RTL modules, which simplifies both simulation and debugging.

Based on these considerations, we chose to treat extra signals as part of the original op. In this model, the extra signals are not manipulated at the IR level, but instead are handled inside the RTL implementation of the op—just like the op's core logic.

This design decision worked well in the development of our new backend. However, we recognize that this touches on broader discussions about the semantics and role of ops in the Handshake IR. Some have raised concerns about the *qualitative* explosion of RTL unit variants—not *quantitative* (which is already unbounded due to the bitwidth)—due to the combinatorial space of extra signal configurations.

Our current stance is the following:

- The handshake-level abstraction should remain the finest granularity for IR ops.
- Each op should internally manage any signal-level details.
- Even handshake steering modules, such as rigidifiers, can be formulated as handshake-level modules.
- While we produce RTL modules in a generative fashion (as discussed in the following sections), the generation logic is clearly defined (as seen later in Listing 4) and remains constrained by type system verification (see Section 12.2.1), which enforces correctness.

14.3. Difference from the Existing Backend

The existing backend relies primarily on a **template-based** approach, where pre-written RTL files are just copied into the output folder, with a few exceptions where code generation is necessary. These RTL templates are written in a parameterized style—allowing customization of attributes like bitwidth or buffer size using VHDL or Verilog's **generics**.

While this works well for simple numeric parameters, it is already limiting when structural variants exist. For example, units such as `fork` are instantiated differently depending on whether they operate on data or control channels. Control channels omit the `data` signal in their entity declarations, so we maintain two separate RTL templates—one for the data variant and another for the control variant (referred to as `dataless`).

After lowering to HW IR (see Figure 30), which is a hardware-oriented intermediate representation, RTL generation proceeds via a **matching logic**. This logic is designed for the template-based approach. As shown in Figure 51, a `fork` with a data signal is matched to a specific RTL file (`fork.vhd`) and the bitwidth is specified as a generic parameter. A control-only `fork` matches to a separate file without data handling.

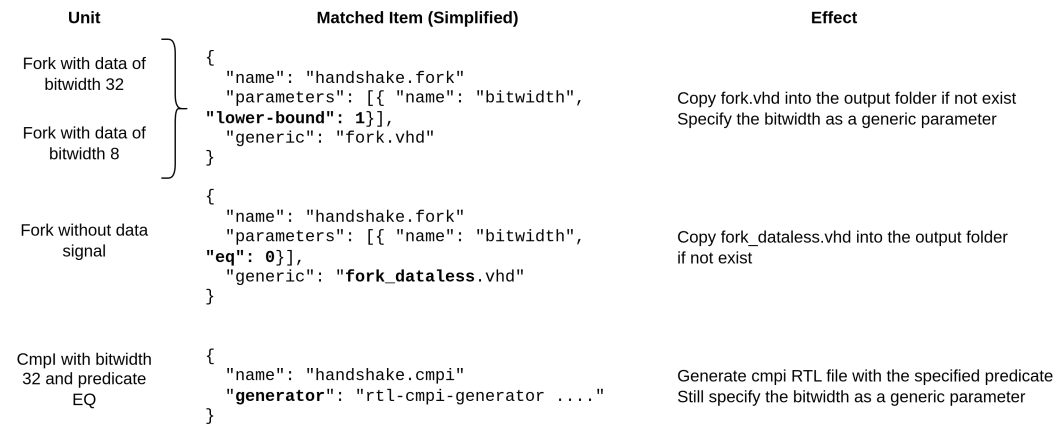


Figure 51: Example of matching logic for RTL generation.

Even within this system, certain units require actual RTL generation. For example, `cmpi` depends on a comparison predicate (like `EQ` or `LT`), which is not well-suited for generics. Instead of maintaining a file for every predicate variant, the backend calls a generator to produce the appropriate RTL on demand.

In those cases, a single matching item is present for this kind of units, which maps to a generator function instead of a template (the bottom of Figure 51).

Our new backend adopts this same generation-based strategy globally. Since extra signal configurations cannot be expressed through generic parameters, we chose to generate RTL modules for **all** units. Each unit has a single corresponding matching item that invokes our generator. The matching logic framework remains unmodified, enabling seamless integration with our extended backend.

14.4. Generator

As explained in the previous section, no changes were made to the matching logic. Instead, our changes are primarily focused on the generators.

We implemented a Python-based generator that works for every unit. This approach is largely inspired by Gioele Gottardo's ongoing work on the SMV backend, where generative methods are required due to SMV's limited expressibility. Building on this, we defined how parameters are passed to the RTL generators from the RTL exporter and developed a signal manager to handle the extra signals.

We chose Python as the language for the generator for several reasons:

- **Text Processing**: RTL generation involves extensive text processing, and Python excels at this task, with features like f-strings for syntax handling and flexible string join methods.
- **Accessibility**: Python is widely used and accessible, making it easier for others to understand and contribute to the code compared to MLIR (which has limited documentation and is more complex to work with).

The generator is structured in an organized way, as shown below:

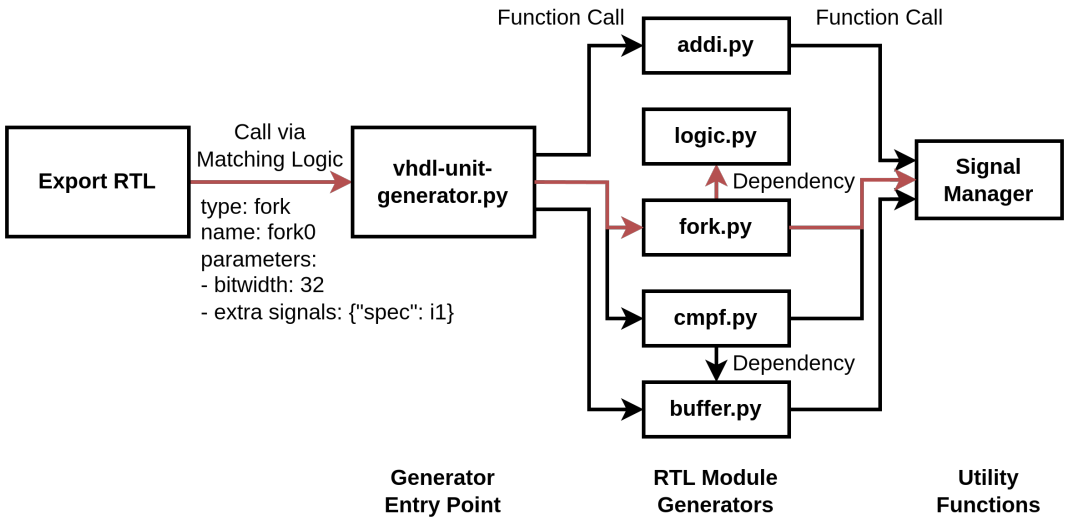


Figure 52: RTL Generator structure.

The generator consists of three main parts. The entry point and RTL module generators are largely inspired from Gioele Gottardo's ongoing work.

Entry Point:

The entry point is invoked through the matching logic, for example, `"generator": "python vhdl-unit-generator.py ..."`. It receives parameters necessary to generate each unit, which will be discussed further in the next section.

Note that all units share this entry point, and the entry point calls a separate generator file for each unit based on the `type` parameter, which specifies the unit type. As a result, the matching logic remains nearly identical across all units, with the only difference being the specific parameters passed to the corresponding RTL module generator:

```
{
  "name": "handshake.addi",
  "generator": "python vhdl-unit-generator.py --type addi ..."
},
{
  "name": "handshake.fork",
  "generator": "python vhdl-unit-generator.py --type fork ..."
},
```

Listing 3: Matching logic is almost identical.

The entry point itself is concise: it simply parses the arguments and branches the processing based on the unit type specified in the argument:

```
def generate_code(name, mod_type, parameters):
  match mod_type:
    case "addf":
      return addf.generate_addf(name, parameters)
    case "addi":
      return addi.generate_addi(name, parameters)
    case "andi":
      return andi.generate_andi(name, parameters)
    # ...
```

Listing 4: Entry point of the generator.

RTL Module Generators:

Responsible for generating each RTL module. Some modules may depend on others, and in such cases, the module generator is allowed to call other module generators. Note that the RTL modules also include those that do not follow the handshaking protocol (e.g., `logic.py` used by `fork.py`). These modules are exclusively used as dependencies by other modules and are not called directly from the entry point. The management of these dependencies is discussed in a later section.

Utility Functions:

Some aspects of the generation process are common across different units. To streamline this, we provide utility functions, especially for generating signal managers that handle the extra signals. Since most modules share the same implementation for the signal manager, they rely on a helper function to generate it. The specifics of this will be covered in a later section.

14.5. Parameter Passing

The parameters required to generate the RTL modules are passed from the C++ side to the Python generator as arguments. These parameters include:

- **`type`**: The name of the handshake operation (e.g., `fork` or `addi`).
- **`name`**: The unique name of the operation instance in the circuit (e.g., `fork0` or `addi1`). This is used as the module name in the generated RTL file.
- **`parameters`**: These vary depending on the module type:
  - **`bitwidth`**: The bitwidth of the data type (e.g., `32` or `8`). Passed as a number.
  - **`extra_signals`**: The extra signals of the operation, serialized as a Python `dict[str, int]` (e.g., `{"spec": 1}` meaning a `spec` signal with a bitwidth of 1).

These parameters are extracted from the HW IR right before the RTL generation step, bypassing all previous passes to prevent any unintended effects.

For operations with traits such as `AllTypesMatch`, `AllDataTypesMatch`, or `AllExtraSignalsMatch` (see Section 12.1.3 and Section 12.3.1), specifying only one `bitwidth` or `extra_signals` is sufficient. In some cases, additional parameters are required. For example, the `load` operation requires two bitwidths (`addr_bitwidth` and `data_bitwidth`). The necessary parameters for each RTL module are clearly defined within the respective module generator files.

Note that we decided to pass the minimum information necessary to steer the RTL generation, rather than the entire operation specification. If redundant parameters were passed to the generator (e.g., two channels with the same bitwidth), it would be the generator's responsibility to choose the appropriate one. We believe this information extraction, which we refer to as "**parameter analysis**," should be handled within the MLIR framework, where type system enforcement is available to ensure correctness.

14.6. Dependency Resolution

Dependency handling between RTL modules is mainly managed on the generator's side. This approach contrasts with the existing backend, where the matching logic is responsible for resolving dependencies. However, the existing matching logic was not powerful enough for our new backend.

The dependency resolution in the matching logic only applies to **generic** modules, which are simply copied into the output folder. Specifically, it simply enumerates the file names of dependencies for each matching item without addressing the actual parameters of the modules. Parameters are specified as generic values when instantiating the module. In our

new backend, however, we need to **call** the generator with specific **parameters** for each dependency.

Inside the generator, dependencies are resolved by calling the generator function of another RTL module file as a Python function call.

A notable aspect of dependency resolution in our new backend is its tree-like structure (Figure 53). When one RTL module depends on another, the content of the dependent module is appended to the same file as the caller. Even if multiple RTL modules share the same dependency, the content is redundantly appended to each file, unlike the original matching logic, which checks if the dependency file has already been copied. While this results in larger RTL files, it does not pose an issue during synthesis.

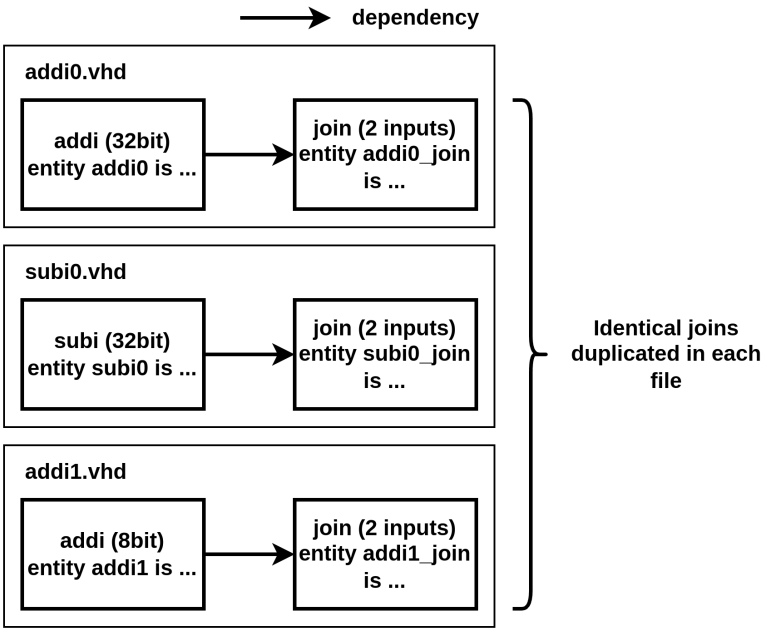


Figure 53: Dependency resolution in the generator.

The redundancy of this approach can lead to **name collisions**. To mitigate this, we adopt a tree-like naming convention for dependency units. For instance, if `addi0` depends on a two-input join, it generates the dependency with the name `addi0_join` (see Figure 53), using the caller's name as a prefix to avoid collisions. This idea is adopted from Gioele Gottardo's ongoing work.

However, redundancy remains problematic in certain cases:

- **External Floating Pointer Library RTL**: This introduces large file sizes and name collisions, as the RTL library has a fixed name.
- **Type Aliases (VHDL)**: We define a 2D array for variadic channels. In VHDL, two type aliases referring to the same type are treated as distinct, making duplicated definitions problematic.

For these cases, we revert to the dependency resolution from the matching logic. These files are not generated but instead copied directly. In the future, once the matching logic is phased out, the generator will handle these file copies when necessary.

14.7. Signal Manager

The signal manager serves as a wrapper for each RTL module to handle extra signals. Since most operations share the way of handling extra signals, we provide common signal managers as utility functions. If the generation parameter includes extra signals, the RTL module generator calls this utility to create the signal manager and makes the RTL module itself a dependency of the signal manager (as shown at the top of Figure 54). In this setup, the signal manager is named based on the top-level name passed as an argument from the C++ side, while the inner RTL module uses `<name>_inner` as its identifier.

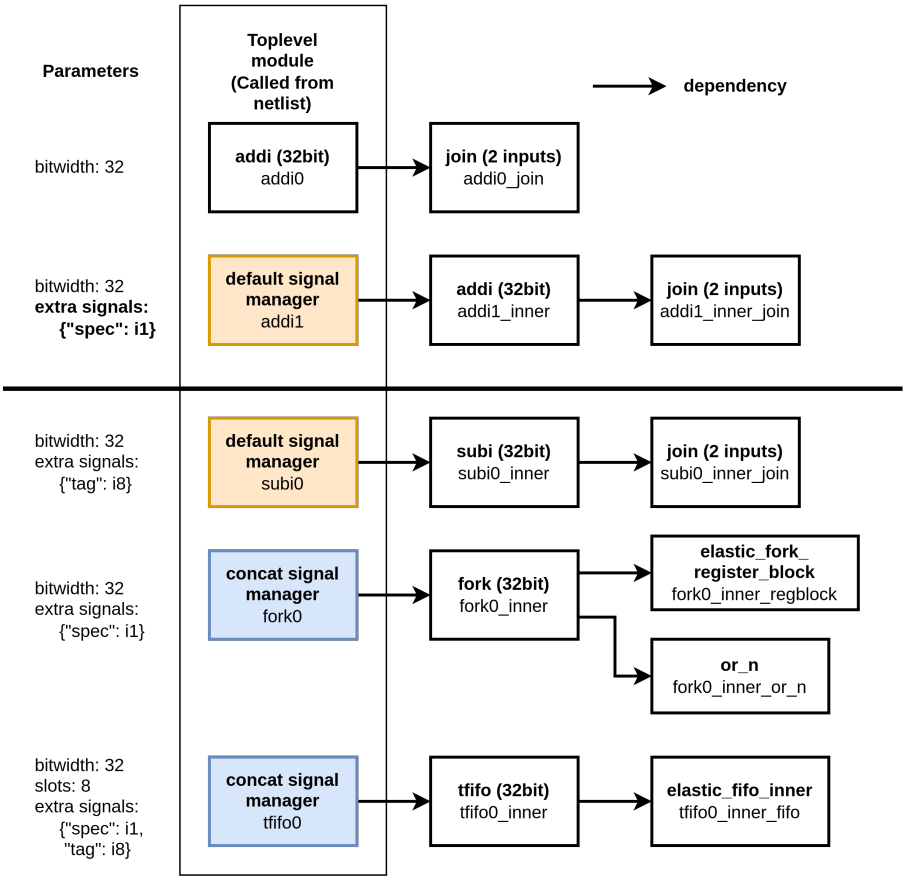


Figure 54: Top: Dependency with signal manager. Bottom: Common signal managers shared across RTL modules.

The bottom of Figure 54 shows the same signal manager function is invoked by different RTL modules.

Next, I will describe three typical signal managers that are commonly used by the RTL modules.

14.7.1. Default Signal Manager

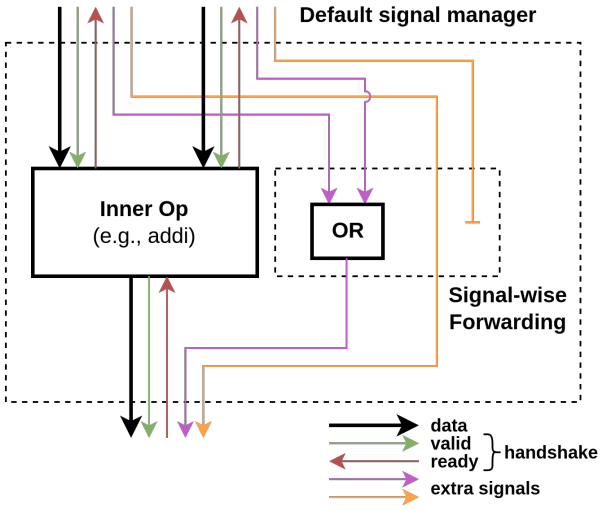


Figure 55: Default signal manager.

The default signal manager forwards the extra signals in a signal-wise manner. Each signal is forwarded through a function that maps all input signals to one output signal. In the figure, the purple signal performs an `OR` operation, while the orange signals select one value. If there are multiple outputs, the forwarded signal value is shared.

For the `spec` signal, we specifically perform an `OR` operation, as discussed in Section 14.1.

14.7.2. Buffered Signal Manager

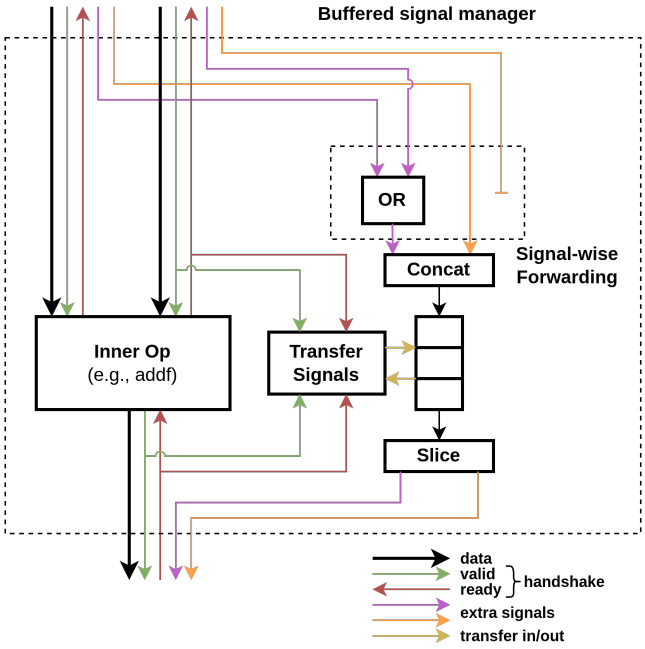


Figure 56: Buffered signal manager.

The buffered signal manager also forwards the extra signals in a signal-wise manner but includes buffers to store extra signals after forwarding. This is intended for units that introduce latency, such as floating-point additions. The buffer size is specified as a parameter to match the latency of the internal module.

To control the buffer push and pop operations, we calculate **transfer signals**. The `transferIn` (push) signal is generated by performing an `AND` operation between the ready and valid signals of one input, while the `transferOut` (pop) signal is generated similarly for one output.

14.7.3. Concat Signal Manager

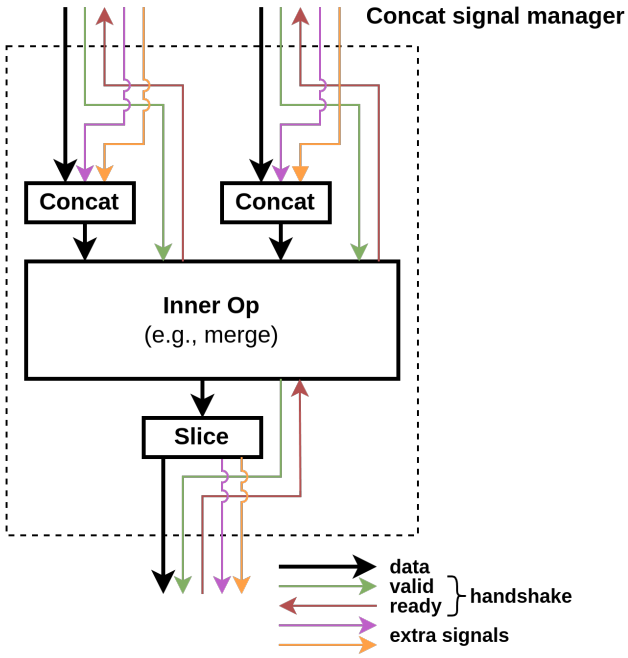


Figure 57: Concat signal manager.

The concat signal manager concatenates the extra signals with each data signal. This results in an expanded bitwidth for the data signal, and the internal module is instantiated with the appropriate bitwidth as a dependency.

This unit is used for operations that do not rely on the actual content of the value, such as `fork` and `buffer`.

14.7.4. Select: Example of Custom Signal Manager

While we provide common signal managers, some RTL modules require custom signal managers. For example, the `select` operation uses a custom signal manager.

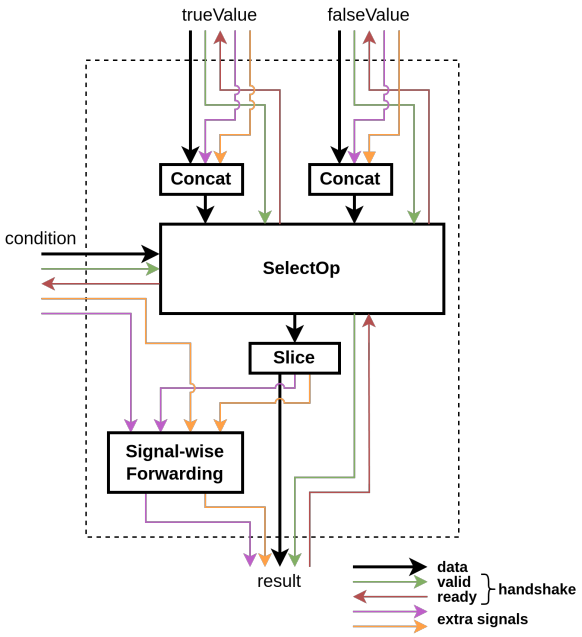


Figure 58: Select signal manager.

The custom manager of `select` combines aspects of both the default and concat signal managers. The `trueValue` and `falseValue` are concatenated and passed to the inner `select` operation, and the result is sliced. The extra signals from the sliced result and those from the `condition` input are then forwarded to the output.

14.7.5. Design Decision

- The forwarding behavior varies from signal to signal, with each signal manager using the common signal-wise forwarding pattern. By defining the forwarding semantics in **a single place**, we ensure consistency and reuse across all signal managers.
- Instead of extending the few existing signal managers, we chose a strategy of **reinventing** new ones. For example, while the default signal manager could be considered a zero-latency version of the buffered signal manager, we defined them separately. This approach emerged from our experience working throughout this project—extending the existing things often results in highly parameterized, monolithic designs that are difficult to modify and understand. In contrast, our approach encourages modularity and simplicity, leading to clearer, more maintainable designs. We implemented small, concrete helper functions, designed to work like Lego bricks. While it might seem repetitive to reinvent signal managers, these helper functions (such as the one for signal-wise forwarding) handle the tedious parts, keeping the custom signal manager implementation manageable.

14.8. Future Work for the Backend

Although our backend is functional, there are still several issues to address. Some of the key areas for improvement include:

- **Execution Time**: Our backend currently performs slower than the existing one. While the exact impact depends on the circuit size, I estimate it takes about 4 seconds to generate the circuit, while the existing backend is almost instantaneous. The performance bottleneck needs to be investigated, and potential optimizations to speed up Python execution should be considered.

- **Dependency Resolution**: While the duplicated dependency resolution doesn't pose any issues during synthesis, the generated code would benefit from being more compact.
- **Redesigning the C++ side**: Our current implementation of the new backend is still somewhat provisional, especially regarding its integration with the existing C++ infrastructure. For example, the parameter analysis is done right before RTL generation (see Section 14.5), but due to limited access to the original Handshake IR, we extract this information in a rather fragile manner, specifically via the channel *index*. Additionally, our reliance on the existing matching logic is temporary. As we transition to a more refined design on the C++ side, the backend will become more robust and better organized.

15. Evaluation

To assess the effectiveness of speculation in the current implementation, I ran benchmarks from Haoran's thesis [2]. These benchmarks are available as speculation integration tests in the `integration-test` folder (see Section 4.1).

We executed the tests following the procedure outlined in Section 4 and Section 5, particularly the manual modification of the program and CFG to adhere to the single-basic-block loop limitation (Section 5.1).

We compared the initiation interval (II) with and without speculation. Measuring clock period (CP) or resource usage will be left for future work.

Below are the results for each benchmark. Overall, we were able to replicate Haoran Zhao's results [2] regarding the achieved II.

15.1. `single_loop`

The loop condition depends on the multiplication of two int variables, a computationally heavy operation that hindered pipelining. Therefore, condition speculation was performed.

15.1.1. Original Program

```
void single_loop(in_int_t a[N], in_int_t b[N], inout_int_t c[N]) {
  int i = 0;
  int bound = 1000;
  int sum = 0;
  while (sum < bound) {
    sum = a[i] * b[i];
    c[i] = sum;
    i++;
  }
}
```

15.1.2. Modified Program

The while loop was converted to a do-while loop to create a single basic block (1-BB) loop.

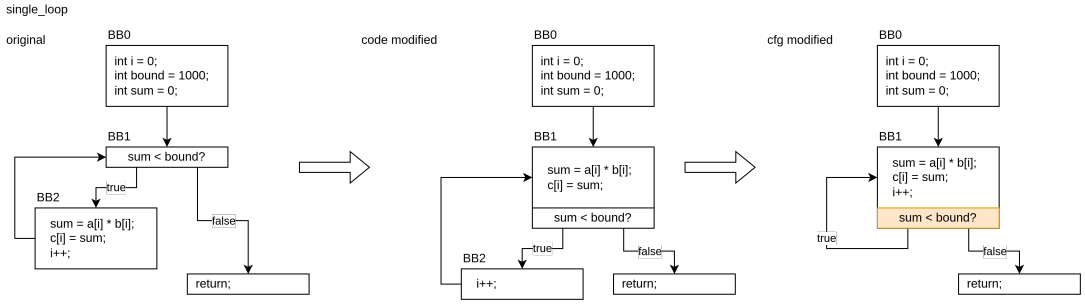
```
void single_loop(in_int_t a[N], in_int_t b[N], inout_int_t c[N]) {
  int i = 0;
  int bound = 1000;
  int sum = 0;
  do {
    sum = a[i] * b[i];
    c[i] = sum;
    i++;
```

```
  } while (sum < bound);
}
```

15.1.3. CFG

The modified do-while loop still forms a 2-BB loop due to limitations in the MLIR scf dialect. As a result, I manually modified the CFG to eliminate the second basic block (BB 2).

The figure below highlights the speculated condition. The speculator is placed in the same BB as the condition.



15.1.4. Results (Code and CFG modified)

The result matched that of Haoran's thesis.

	No Speculation	Speculation
II (Haoran's thesis [2])	6	1
II	6	1
Cycles (Testbench)	3013 (End: 3011)	516 (End: 511)

15.2. `loop_path`

The bottleneck in the original program was the condition of the `if` statement, which was speculated by Haoran Zhao's work [2]. However, due to the single-basic-block limitation, I manually modified the loop condition and removed the `break` statement while maintaining the execution equivalence. As a result, the problem is similar to the `single_loop` case.

15.2.1. Original Program

```
void loop_path(in_int_t a[N], in_int_t b[N], inout_int_t c[N]) {
  int i;

  for (i = 0; i < N; i++) {
    int temp = a[i] + b[i];
    int x = 5;
    c[i] = temp;
    if ((1000 - temp) <= x * temp) {
      break;
    }
  }
}
```

15.2.2. Modified Program

I introduced the `break_flag` variable as an alternative to the `break` statement.

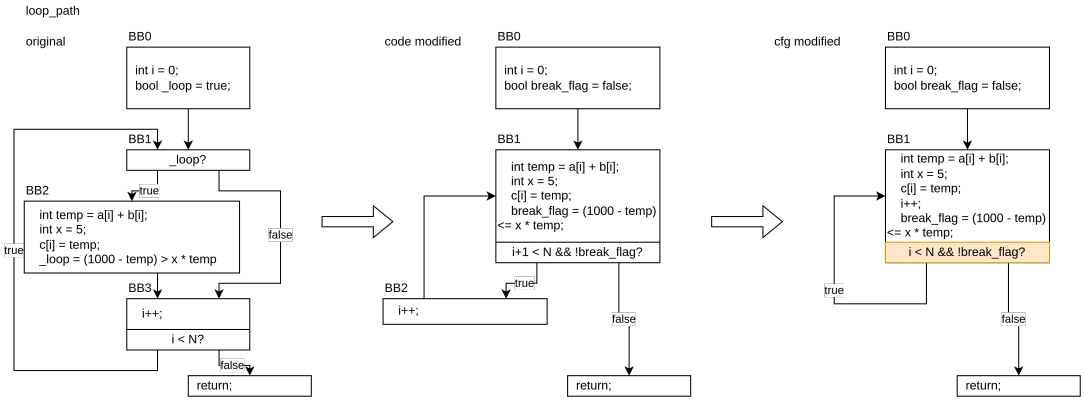
```
void loop_path(in_int_t a[N], in_int_t b[N], inout_int_t c[N]) {
  int i = 0;
  bool break_flag = false;
```

```
  do {
    int temp = a[i] + b[i];
    int x = 5;
    c[i] = temp;
    i++;
    break_flag = (1000 - temp) <= x * temp;
  } while (i < N && !break_flag);
}
```

15.2.3. CFG

Due to the limitations in the MLIR `scf` dialect, the current frontend cannot generate the CFG corresponding a loop with a `break` statement. Instead, it continues the loop even after the `break` is encountered. By using a `do-while` loop and introducing the break_flag variable, the CFG is straightforward and can be further manually manipulated.

Another observation of the CFG generated by the current frontend: When closely examining the CFG of the "modified code" (in the middle), the loop condition is `i+1<N`, meaning that `i+1` is calculated both in BB 1 and in BB 2, which introduces redundancy.



15.2.4. Results (Code and CFG modified)

The speculation case achieved an II of 1, matching the results in Haoran's thesis.

The II is only 2, even in the non-speculation case. This is because the break condition `(1000 - temp) <= x * temp` is optimized in the current frontend, eliminating the multiplication.

	No Speculation	Speculation
II (Haoran's thesis [2])	6	1
II	2	1
Cycles (Testbench)	341 (End: 339)	175 (End: 173)

15.3. subdiag

The code is essentially the same as `loop_path`. However, since it uses float variables, a larger improvement in II was expected.

15.3.1. Original Program

```
int subdiag(in_float_t d[N], in_float_t e[N]) {
  int i;

  for (i = 0; i < N_DEC; i++) {
```

```
    float dd = d[i] + d[i + 1];
    float x = 0.001;
    if ((e[i]) <= x * dd)
      break;
  }

  return i;
}
```

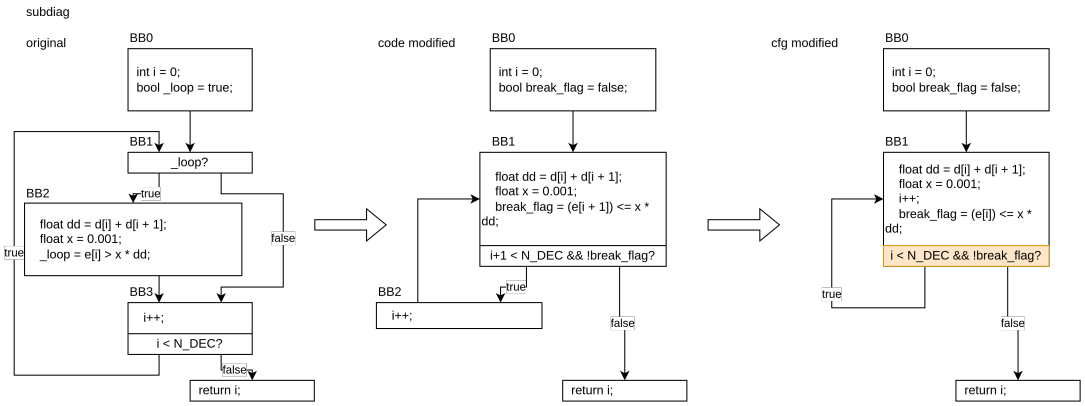
15.3.2. Modified Program

The same transformation was applied as in the `loop_path` case.

```
int subdiag(in_float_t d[N], in_float_t e[N]) {
  int i = 0;
  bool break_flag = false;

  do {
    float dd = d[i] + d[i + 1];
    float x = 0.001;
    i++;
    break_flag = (e[i]) <= x * dd;
  } while (i < N_DEC && !break_flag);
  return i;
}
```

15.3.3. CFG



15.3.4. Results (Code and CFG modified)

The II could not achieve 1, because of the two load operations on `d` array inside the loop (`d[i] + d[i + 1]`). I also created an alternative benchmark, replacing `d[i] + d[i + 1]` with `d1[i] + d2[i + 1]`, which successfully achieved an II of 1 (see the next benchmark).

Additionally, the current implementation of `cmpf` seems to generate output within the same cycle, which is unrealistic. While the change of the `cmpf` implementation does not affect the result in the speculation case, it may worsen the II in the non-speculation case.

	No Speculation	Speculation
II (Haoran's thesis [2])	15	1
II	16	2
Cycles (Testbench)	1623 (End: 1621)	230 (End: 224)

15.4. `subdiag_fast`

I replaced `d[i] + d[i + 1]` with `d1[i] + d2[i + 1]` in the `subdiag` benchmark to enable simultaneous memory access.

15.4.1. Modified Program

```
int subdiag_fast(in_float_t d1[N], in_float_t d2[N], in_float_t e[N]) {
  int i = 0;
  bool cond_break = false;
  do {
    float dd = d1[i] + d2[i + 1];
    float x = 0.001;
    i++;
    cond_break = (e[i]) <= x * dd;
  } while (i < N_DEC && !cond_break);
  return i;
}
```

15.4.2. Results (Code and CFG modified)

It successfully achieved an II of 1.

	No Speculation	Speculation
II	15	1
Cycles (Testbench)	1522 (End: 1520)	126 (End: 120)

15.5. `fixed`

15.5.1. Original Program

Condition speculation is still performed, but since the variables `x0` and `x1` depend on values from previous iterations, a poor II is expected.

```
float fixed(in_float_t y) {
  float c = 1000.0f;
  float x1 = 0.0f;
  float x0 = 1.0f;
  float a = 0.00000001f;
  while (c >= a) {
    x1 = x0 * y;
    c = x0 - x1;
    x0 = x1;
  }
  return x1;
}
```

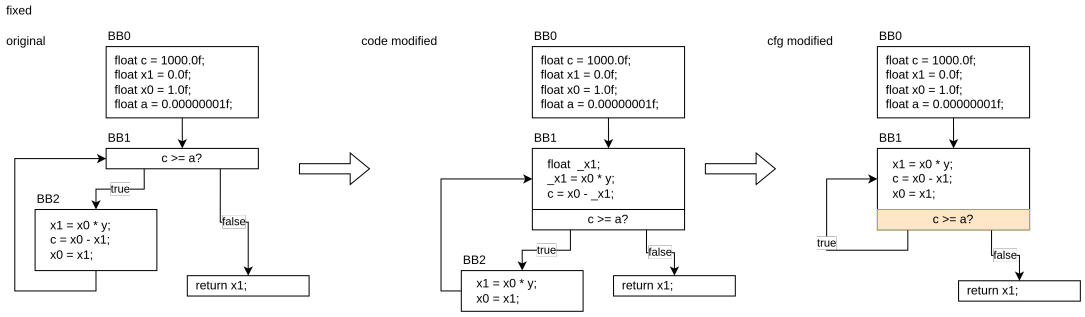
15.5.2. Modified Program

```
float fixed(in_float_t y) {
  float c = 1000.0f;
  float x1 = 0.0f;
  float x0 = 1.0f;
  float a = 0.00000001f;
  do {
    x1 = x0 * y;
    c = x0 - x1;
    x0 = x1;
  } while (c >= a);
```

```
  return x1;
}
```

15.5.3. CFG

Observation of the CFG generated by the current frontend: In the middle of the figure below, variables spanning loop iterations can only be updated in `BB2` due to the restrictions of the MLIR `scf` dialect, leading to awkward code.



15.5.4. Results (Code and CFG modified)

As expected, the speculation case did not achieve a good II, though it successfully eliminated the latency of `subf` and `cmpf`.

	No Speculation	Speculation
II (Haoran's thesis [2])	16	6
II	14	5
Cycles (Testbench)	705 (End: 704)	275 (End: 270)

15.6. `sparse`

In Haoran's thesis [2], data speculation on `sum` with very low precision was performed. However, since the prediction of floating-point variables is not supported yet, I decided to speculate on the loop condition instead. This resulted in a situation essentially similar to the `fixed` case, due to the loop dependency `sum += mul`.

15.6.1. Original Program

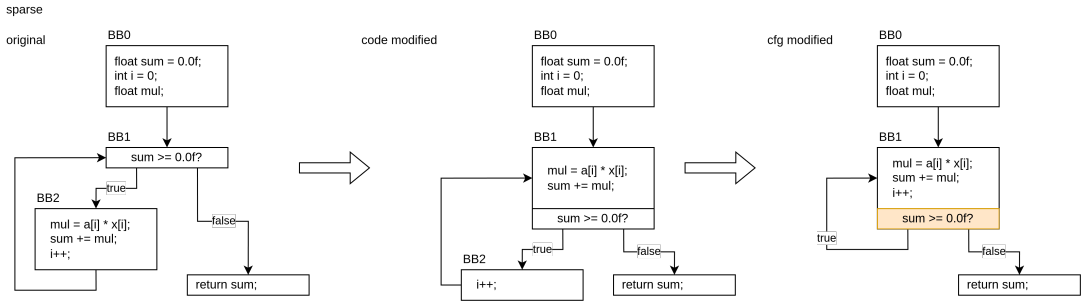
```
float sparse(in_float_t a[N], in_float_t x[N]) {
  float sum = 0.0f;
  int i = 0;
  float mul;
  while (sum >= 0.0f) {
    mul = a[i] * x[i];
    sum += mul;
    i++;
  };
  return sum;
}
```

15.6.2. Modified Program

```
float sparse(in_float_t a[N], in_float_t x[N]) {
  float sum = 0.0f;
  int i = 0;
  float mul;
  do {
```

```
    mul = a[i] * x[i];
    sum += mul;
    i++;
  } while (sum >= 0.0f);
  return sum;
}
```

15.6.3. CFG



15.6.4. Results (Code and CFG modified)

Although Haoran's thesis attempted data speculation and achieved an II of 1, it seemed to be somewhat imprecise, and the total cycle count may not have improved in that case.

	No Speculation	Speculation
II (Haoran's thesis [2])	16	1
II	15	10
Cycles (Testbench)	1237 (End: 1235)	844 (End: 841)

15.7. `nested_loop`

Unlike the other benchmarks, this one requires the loop to be executed repeatedly. The inner loop is similar to the `single_loop` benchmark, with speculation applied to its loop condition.

15.7.1. Original Program

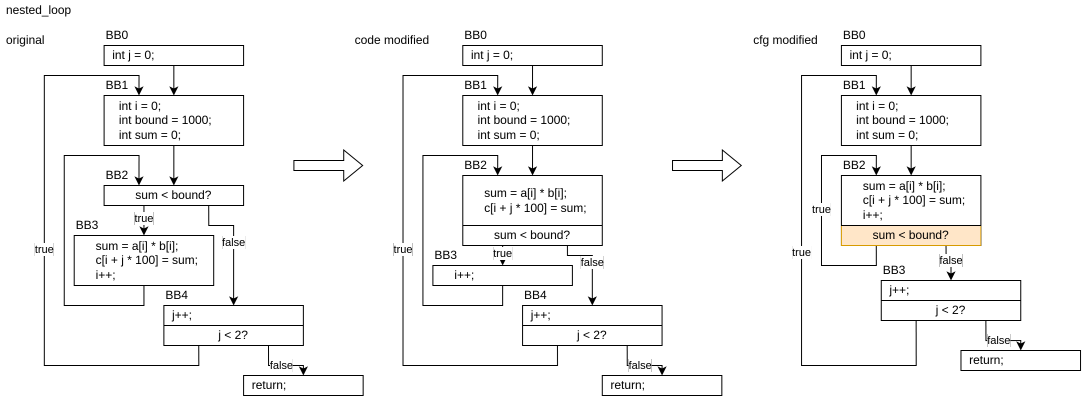
```
void nested_loop(in_int_t a[N], in_int_t b[N], inout_int_t c[N]) {
  for(int j = 0; j < 2; j++){
    int i = 0;
    int bound = 1000;
    int sum = 0;
    while (sum < bound) {
      sum = a[i] * b[i];
      c[i + j * 400] = sum;
      i++;
    }
  }
}
```

15.7.2. Modified Program

```
void nested_loop(in_int_t a[N], in_int_t b[N], inout_int_t c[N]) {
  for(int j = 0; j < 2; j++){
    int i = 0;
    int bound = 1000;
    int sum = 0;
    do {
      sum = a[i] * b[i];
```

```
      c[i + j * 400] = sum;
      i++;
    } while (sum < bound);
  }
}
```

15.7.3. CFG



15.7.4. Results (Code and CFG modified)

The result matched that of Haoran's thesis.

	No Speculation	Speculation
II of the inner loop (Haoran's thesis [2])	6	1
II of the inner loop	6	1
Cycles (Test Bench)	2423 (End: 2421)	443 (End: 439)

15.8. `if_convert`

The final benchmark speculates on the condition of `if` statement inside the loop, which involves data speculation.

15.8.1. Original Program

```
void if_convert(in_int_t a[N], inout_int_t b[N]) {
  int i = 1;
  while (i < N2) {
    int tmp = a[i];
    if (i * tmp < 10000) {
      i++;
    }
    i++;
    b[i] = 1;
  }
}
```

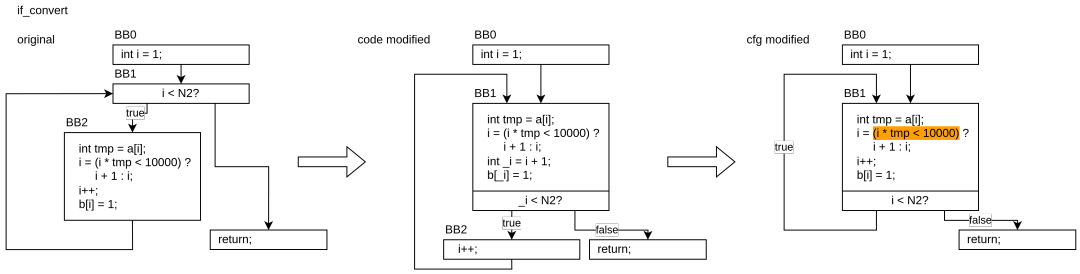
15.8.2. Modified Program

```
void if_convert(in_int_t a[N], inout_int_t b[N]) {
  int i = 1;
  do {
    int tmp = a[i];
    if (i * tmp < 10000) {
      i++;
    }
```

```
    i++;
    b[i] = 1;
  } while (i < N2);
}
```

15.8.3. CFG

The condition of the ternary operator is speculated, leading to data speculation of `i`.



15.8.4. Results (Code and CFG modified)

The optimal II was achieved with speculation.

	No Speculation	Speculation
II (Haoran's thesis [2])	7	1
II	6	1
Cycles (Test Bench)	1129 (End: 1127)	309 (End: 307)

16. Conclusion

This report presented the formulation and implementation of Speculation v1 within the current Dynamatic ecosystem. While there is still considerable room for improvement—from the speculator design and speculation algorithm to the backend—this version now being part of Dynamatic's main branch opens the door for broader use. I welcome any feedback and suggestions for improvement, and I hope this report serves as a helpful reference for future work on speculation.

17. Appendix: Speculation Might Impair Dynamism

Dynamic scheduling enables variable throughput and higher performance, as it isn't bottlenecked by occasional long-latency iterations. However, speculation can potentially undermine this advantage.

One concern is deadlock due to insufficient buffering. With speculation enabled, extra buffers are required—primarily within the control network—to hold unresolved speculative tokens (see Section 13.1). When speculation resolution is occasionally delayed (e.g., by control flow or memory hierarchy), the buffers must be sized for the worst-case scenario to prevent deadlock.

A more fundamental problem is that speculation can degrade the circuit's dynamic behavior. A transient stall may propagate through the speculative logic and recur in subsequent cycles, effectively locking the initiation interval (II) at a suboptimal value. The following example illustrates this effect.

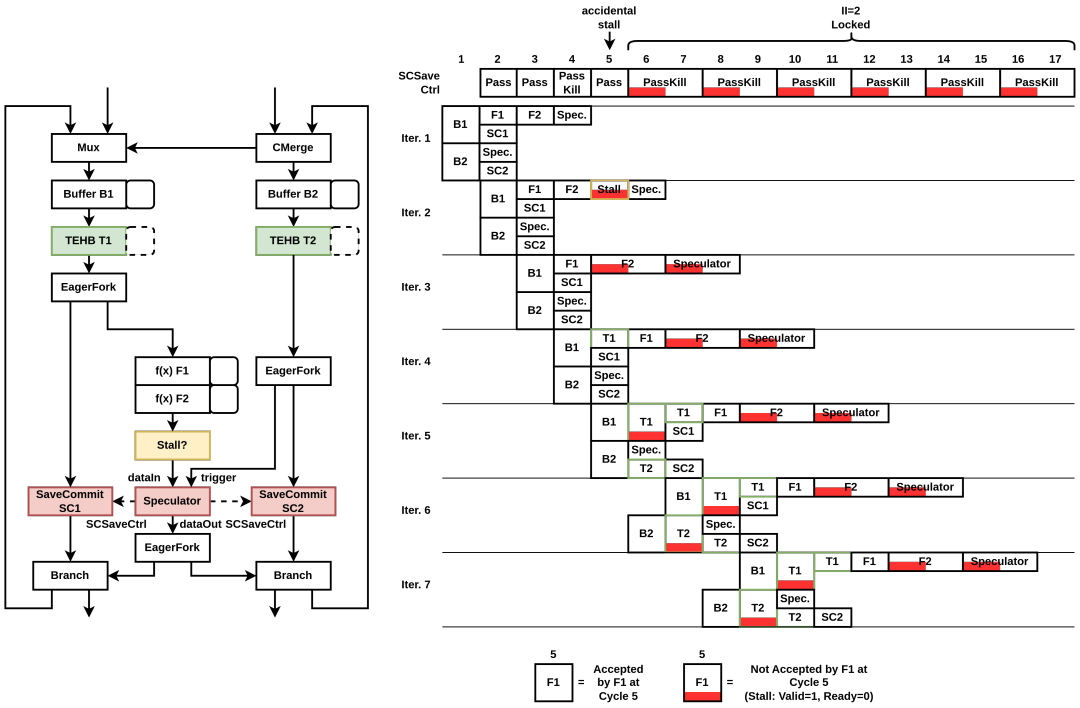


Figure 59: A speculative circuit and its schedule. An accidental stall occurs at the `Stall?` unit in cycle 5. This stall propagates through the circuit and recirculates, preventing recovery to the original II=1. As a result, the circuit locks into a degraded II=2.

Figure 59 shows the example circuit and its schedule. A stall can occur just before the speculator `dataIn`, caused by factors such as a control-flow decision (including one made by a `select` node) or, in the future, a cache miss. Buffers are inserted using the default `on-merges` strategy in Dynamatic [4]. The resulting circuit states from cycles 5 to 8 are shown in Figure 60.

At cycle 5, a stall occurs, preventing the speculator from resolving the token of iteration 2. This in turn stalls the `f(x)` unit just upstream, blocking the token from iteration 3 from advancing through the pipeline. The token from iteration 4 is *absorbed* by the transparent buffer T1. Due to the eager fork, the save-commit unit SC1 accepts the token from iteration 4, even though `f(x)` is stalled. The speculator is still triggered and sends a `PASS` token to the save-commit units (not a `PASS_KILL`, since the actual data token from iteration 2 remains unresolved).

At cycle 6, `f(x)` and the speculator both accept the next token. However, T1 now stalls because the token it holds is being read by the eager fork and `f(x)`, propagating the stall from the previous cycle. As a result, the token from iteration 4, previously stored in buffer B1, cannot proceed to either `f(x)` or SC1. The speculator is triggered again and this time issues a `PASS_KILL`. However, SC1 cannot accept the control token because it has not received the corresponding data token. SC2 also cannot accept the token due to the Mux not accepting the index signal from the CMerge. Notably, B2 does accept the token, thanks to the downstream T2 and the eager fork logic inside the CMerge.

At cycle 7, the speculator's `dataIn` stalls due to the earlier stall at the speculator's outputs. (The speculator also has an internal transparent buffer.) This resembles the earlier stall scenario from cycle 5. The token from iteration 5 is now absorbed by T1. At this point, SC1 successfully receives its data token, and both SC1 and SC2 accept the token. However, T2 cannot accept the token, mirroring T1′s behavior in the previous cycle.

At cycle 8, SC1 does not receive a token, replicating the situation from cycle 6. From this point onward, the circuit enters a repeating pattern of cycles 7 and 8—effectively halving the throughput and locking the circuit into an initiation interval (II) of 2.

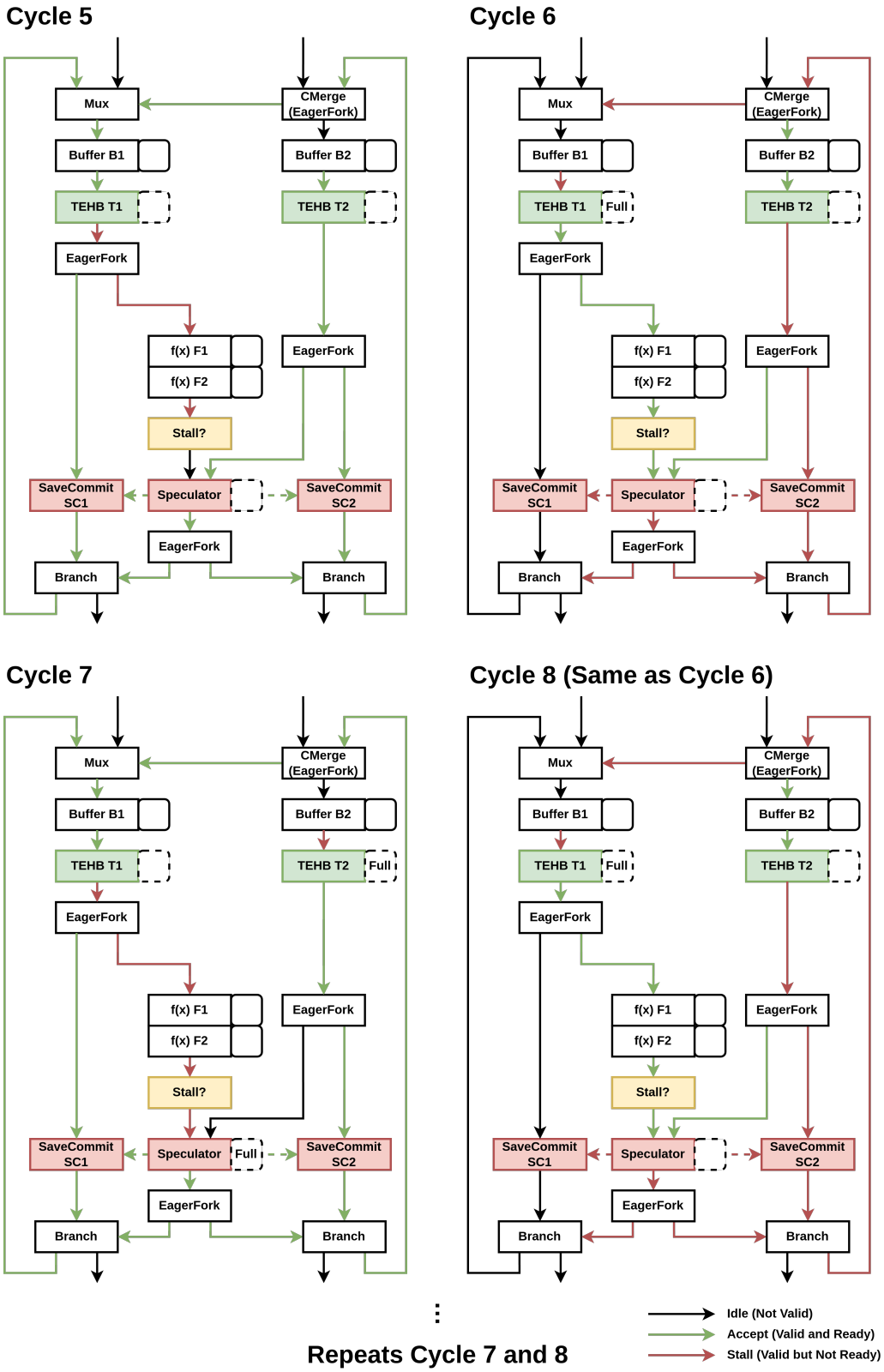


Figure 60: Circuit states from cycles 5 to 8. Once the stall occurs at cycle 5, the circuit enters a repeating pattern (cycles 7 and 8), failing to recover the original high-throughput schedule (II=1).

This problem does not occur in circuits without speculation. When a stall accidentally happens on a cyclic path without speculation, it does not circulate—only a single iteration experiences increased latency (Figure 61, left). This is because there is no *medium* for the stall to propagate: without speculation, only one token exists on a cyclic path at any time. Note that even when the stall occurs on an acyclic path (Figure 61, right)—which is structurally more similar to the speculative case in Figure 59—the stall still does not propagate.

Therefore, this stall circulation behavior is **unique to speculation**, where multiple tokens can exist simultaneously on cyclic paths.

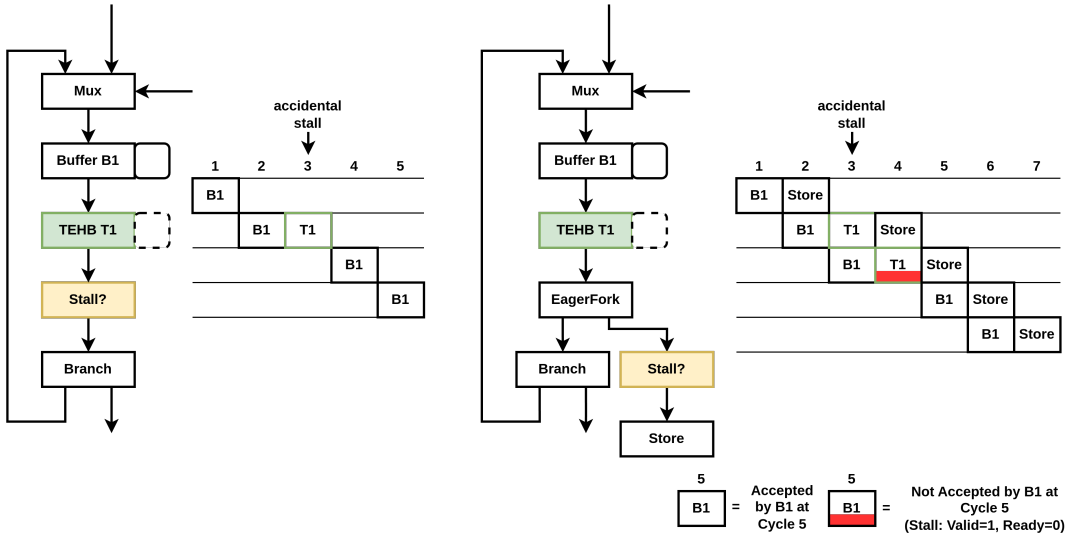


Figure 61: Accidental stalls in circuits without speculation. In both cases—inside and outside a cyclic path—the stall does not propagate or cause sustained throughput degradation.

One way to prevent this issue is to provision sufficient buffering so that stalls never occur. However, this might not be practical, as it seems difficult to predict when and where stalls might arise. Moreover, this problem is likely not limited to the current *implementation* of speculation, so future designs should also account for this potential behavior.

Bibliography

[1] L. Josipović, A. Guerrieri, and P. Ienne, "Speculative Dataflow Circuits," in *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 2019, pp. 162–171.

[2] H. Zhao, "Speculative Dataflow Circuits (Master Thesis)," Apr. 2023.

[3] A. S. Ugalde, "Speculative Dataflow Circuit Generation Using a MLIR-based Compiler," Jan. 2024.

[4] "Dynamatic: An open-source high-level synthesis toolchain for dataflow circuits. github.com/EPFL-LAP/dynamatic." 2023.

[5] P. Kocher *et al.*, "Spectre Attacks: Exploiting Speculative Execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[6] M. Lipp *et al.*, "Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[7] J.-M. Gorius, S. Rokicki, and S. Derrien, "SpecHLS: speculative accelerator design using high-level synthesis," *IEEE Micro*, vol. 42, no. 5, pp. 99–107, 2022.

[8] L. Josipović, A. Guerrieri, and P. Ienne, "From C/C++ code to high-performance dataflow circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 7, pp. 2142–2155, 2021.

[9] A. Elakhras, A. Guerrieri, L. Josipović, and P. Ienne, "Unleashing parallelism in elastic circuits with faster token delivery," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 253–261.