Anvil: A General-Purpose Timing-Safe Hardware Description Language

Jason Zhijingcheng Yu*
yu.z@u.nus.edu
Department of Computer Science
National University of Singapore
Singapore

Aditya Ranjan Jha*
adityajha@u.nus.edu
Department of Computer Science
National University of Singapore
Singapore

Umang Mathur
umathur@nus.edu.sg
Department of Computer Science
National University of Singapore
Singapore

Trevor E. Carlson
tcarlson@comp.nus.edu.sg
Department of Computer Science
National University of Singapore
Singapore

Prateek Saxena
prateeks@comp.nus.edu.sg
Department of Computer Science
National University of Singapore
Singapore

Abstract

Expressing hardware designs using hardware description languages (HDLs) routinely involves using stateless signals whose values change according to their underlying registers. Unintended behaviours can arise when the stored values in these underlying registers are mutated while their dependent signals are expected to remain constant across multiple cycles. Such *timing hazards* are common because, with a few exceptions, existing HDLs lack abstractions for values that remain unchanged over multiple clock cycles, delegating this responsibility to hardware designers. Designers must then carefully decide whether a value should remain unchanged, sometimes even across hardware modules. This paper proposes Anvil, an HDL which statically prevents timing hazards with a novel type system. Anvil is the only HDL we know of that guarantees *timing safety*, i.e., absence of timing hazards, without sacrificing expressiveness for cycle-level timing control or dynamic timing behaviours. Unlike many HLS languages that abstract away the differences between registers and signals, Anvil's type system exposes them fully while capturing the timing relationships between register value mutations and signal usages to enforce timing safety. This, in turn, enables safe composition of communicating hardware modules by static enforcement of *timing contracts* that encode timing constraints on shared signals. Such timing contracts can be specified parametric on abstract time points that can vary during run-time, allowing the type system to statically express dynamic timing behaviour. We have implemented Anvil and successfully used it to implement key timing-sensitive modules, comparing them against open-source SystemVerilog counterparts to demonstrate the practicality and expressiveness of the generated hardware.

CCS Concepts: • **Hardware** → **Hardware description languages and compilation**; • **Software and its engineering** → *Context specific languages*; • **Computing methodologies** → *Parallel programming languages.*

Keywords: Hardware description languages; Type systems; Concurrency safety

ACM Reference Format:
Jason Zhijingcheng Yu, Aditya Ranjan Jha, Umang Mathur, Trevor E. Carlson, and Prateek Saxena. 2026. Anvil: A General-Purpose Timing-Safe Hardware Description Language. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 21–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 27 pages. https://doi.org/10.1145/3779212.3790125

1 Introduction

Hardware description languages (HDLs) shape the way people think about and describe hardware designs. Ideally, an HDL should provide easy-to-use abstractions for hardware designers to express their intention precisely and correctly. The concurrent and continuous behaviour of hardware makes this goal challenging to achieve.

Unlike software programs, where values are all persistent (stored either in registers or in memory), hardware designs involve separate notions of *signals* and *registers*. While a register can store persistent values and be assigned new values every cycle, signals are stateless, with their values changing with the registers they depend on. If the hardware designer expects a signal to remain unchanged across multiple cycles, they must explicitly ensure the stored values of their underlying registers do not change. The incorrect timing of register mutation (i.e., change of the stored value in a register) and signal use thus easily introduces invalid or wrong

*Equal contribution.

ASPLOS '26, Pittsburgh, PA, USA.

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790125

values during run-time, and may even expose the hardware design to time-of-check-to-time-of-use (TOCTOU) attacks. We call such problems *timing hazards*. The problem of timing hazards is further exacerbated by the concurrent nature of hardware designs: A hardware design commonly consists of large numbers of modules which are executing in parallel and communicating with each other via shared signals.

Most existing HDLs such as SystemVerilog [22], VHDL [21], and Chisel [3] do not catch timing hazards at compile time, leaving designers to discover these issues only during simulation. Designers frequently seek help on discussion forums simply to pinpoint the origins of the errors [9, 13, 47]. Timing hazards are prevalent even among experienced designers and in widely used open-source hardware components, as is shown by several real-world examples given in Appendix C.

A principled way to eliminate timing hazards is to forbid them in the HDL itself. We call such an HDL that only allows designs without timing hazards *timing-safe*. The key challenge to achieve timing safety while also providing enough expressiveness for writing general-purpose hardware designs. Some existing HDLs can provide timing safety but only at a significant cost of expressiveness, making them only suitable for specific applications. For example, high-level synthesis (HLS) languages [2, 24, 49] offer software-like programming models for hardware design. In these languages, timing hazards are not a concern because they abstract away both cycle latencies and the distinction between wires and registers, effectively treating all values as persistent, similar to variables in software programming. Constructs for expressing cycle-level control and wires are, unfortunately, absent in such languages. Such expressiveness is essential in general-purpose hardware designs, especially where performance is a priority. Consequently, the applicability of HLS languages is commonly limited to speeding up algorithms with programmable hardware (e.g., FPGA). Other timing-safe languages focus only on specific types of hardware designs, such as CPU stages [51] and static pipelines [34, 44].

We present Anvil, the first HDL we know of that **guarantees *timing safety* while maintaining expressiveness for general-purpose hardware design use cases**. Anvil is general-purpose in the sense that the designer retains full control of the cycle-level timing and register states in RTL, unlike HLS languages, and is not limited to design use cases such as CPU-level abstractions and static pipelines. In particular, it allows hardware designers to seamlessly specify cycle-level delays and to express whether a value is stored in a register. It also supports expressing hardware designs with dynamic timing behaviours easily.

Anvil achieves timing safety statically with a novel type system which captures the timing relationships between register mutations and use of signals. It performs type checking that reasons about whether each use of signal takes place in a time window throughout which it carries an unchanging and meaningful value, and rejects code that is not

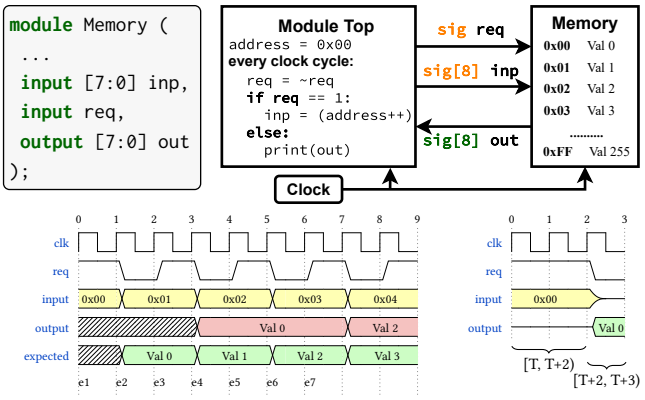

Figure 1. Module `Top` interfaced with `Memory`.

timing-safe. Designs written in Anvil can thus specify precise cycle-level behaviour and register updates. This is in contrast to HLS languages [49] that hide wires and cycles beneath their abstractions. Across hardware modules, Anvil's type system guarantees safe composition by statically checking against *timing contracts*, which specify constraints regarding communicated signals, including constraints about when such signals must be kept unchanged. Although Anvil's type checking is entirely static, it explicitly allows dynamic timing behaviours, i.e., the number of cycles for a behaviour of the hardware design can vary during run-time (e.g., caches). The type system achieves this by capturing time not in terms of an absolute (fixed) number of cycles, but instead as abstract time points that correspond to events that may occur arbitrarily late, for example, the event corresponding to the receipt of data from another module. This is in sharp contrast to recent work [34] in which the proposed type system only allows expressing designs with fixed static timing behaviours.

We have implemented the Anvil compiler (Section 6) which performs type checking and compiles Anvil code to SystemVerilog. Our evaluations highlight the expressiveness and practicality of Anvil (Section 7). Designs written in Anvil can be integrated in existing SystemVerilog code bases, thus allowing incremental adoption and making Anvil immediately useful. We have successfully used Anvil to implement a diverse set of 10 latency-sensitive components ranging from an AES accelerator [38] to a page table walker in a RISC-V CPU [52]. Despite the Anvil compiler being an early-stage prototype, when compared with open-source SystemVerilog implementations, the Anvil implementations show practical overhead averaging 4.50% for area and 3.75% for power. Anvil is open-source at https://github.com/kisp-nus/anvil.

Our Contributions. We introduce Anvil, an HDL with a novel type system that guarantees timing safety without sacrificing expressiveness, e.g., for cycle-level control and dynamic timing behaviours. Anvil allows for general-purpose hardware design use cases and integration with existing SystemVerilog projects.

2 Motivation

The motivation of our work stems from the susceptibility of RTL designs to timing hazards due to limitations of *de facto* standard HDL abstractions.

2.1 Example of a Timing Hazard

Consider the interface of a memory module in SystemVerilog in Figure 1 top left. Unlike software, hardware modules communicate using signals that can be continuously read and updated. Consider an interfacing hardware module (Figure 1, top right), `Top`, which reads a value from a memory module with the same interface. The implementation of `Top` sends an address as a request and expects to read the output in the following cycle. However, the circuit outputs are incorrect, as evident when the system is simulated (Figure 1, bottom left). The culprit is an unexpected timing delay. The module `Top` is written under the assumption that the memory subsystem responds precisely one clock cycle after the `req` signal is set. However, it takes the memory subsystem two cycles to process the lookup request and return the output.

In more detail, the module `Top` requests address `0x00` by setting the `req` signal high during cycle $[0, 1)$. It expects the output in the next cycle, but the memory has not finished dereferencing the input `address`. The memory stops processing since the `req` signal is unset in $[1, 2)$. When `req` is set again in $[2, 3)$ with address `0x01`, the memory is still resolving `0x00`, returning `Val 0` in $[3, 4)$. Meanwhile, the input address changes from `0x01` to `0x02`. When `req` is set again in $[4, 5)$, the memory starts processing `0x02`, skipping `0x01`. As a result, unexpected outputs are observed, and only half of the requested addresses are dereferenced.

The above example illustrates a classic case of a timing hazard, where unintended values are used or values in use are changed unexpectedly. Here, the module `Top` modifies its input while the memory is still processing a request. It also reads the output before it is ready.

2.2 Timing Hazards in Existing HDLs

Timing hazards arise in SystemVerilog and VHDL, two standard and most widely used HDLs, as they lack an abstraction for the designer to express values that are sustained across multiple cycles. These languages also do not provide a mechanism to encode timing constraints pertaining to register assignments and use of signals shared between communicating modules. The abstraction that SystemVerilog and VHDL provide over registers and signals specifies their relationships within a single, non-specific cycle. The designer defines how each register is updated based on the existing register state, and during run-time the signal values are updated accordingly. In other words, signals are essentially pure functions of the current register state; when they are referenced in the code, they simply carry the values of the current moment. Such an abstraction makes it difficult to express intended relationships between signal values across multiple cycles. For example, a SystemVerilog implementation of the `Top` module in Figure 1 does not convey the intent that `req` and `inp` should remain steady for two consecutive cycles, or that `out` should be meaningful only in the following cycle. Other HDLs—including many newer ones that aim to raise the abstraction level for hardware design (e.g., Chisel [3] and SpinalHDL [45])—follow the same fundamental paradigm for describing RTL designs as SystemVerilog and VHDL, and are therefore similarly susceptible to timing hazards.

Some popular HDLs provide different abstractions than SystemVerilog and VHDL but are still unable to avoid timing hazards. Bluespec SystemVerilog (BSV) [4], for example, provides the abstractions of rules and methods. Rules are bundled hardware behaviours that execute atomically. Modules can communicate through invoking each other's exposed methods, which add to the behaviours to be executed. The BSV compiler generates hardware logic to choose rules to execute in each cycle. For example, consider Figure 2. If `Top` reads a value from a cache and enqueues it into a FIFO that only accepts requests when it is not full, the design would typically use two rules: one to invoke the read method of the cache, and another to enqueue the retrieved value into the FIFO. BSV's scheduler ensures that, in each cycle, rules that execute do not conflict (i.e., they do not mutate the same registers), and each rule executes atomically. However, rules only specify operations for the *current* cycle, and scheduling is performed independently for each cycle. BSV does not reason about behaviours that span multiple cycles [4].

In the example, if the module `Top` retrieves a value from a cache and sends it to a FIFO, Anvil enforces the timing contract by detecting violations and guiding the designer towards a timing-safe implementation, as shown in Figure 2 (top). BSV, on the other hand, may still generate a conflict-free schedule that is *timing-unsafe* because it does not capture inter-cycle constraints in its scheduling model.

Root Cause: HDL Abstractions. In summary, the root cause behind the susceptibility of many popular HDLs to timing hazards lies in the abstractions they provide. In particular, their abstractions do *not* express *the designer's intent* concerning when a signal is expected to carry a meaningful value and in which time window the value is expected to remain steady. As such, we provide a novel solution in a new HDL design rather than basing it on existing ones.

2.3 Need for Timing-Safe HDL Abstractions

In this paper, we tackle the problem of timing hazards by creating *timing-safe HDL abstractions* to capture the designer's intent regarding register and signal uses across cycles and in turn prevent timing hazards. An alternative approach is to apply verification techniques to designs expressed in existing HDLs [16, 26, 33, 48]. Such techniques attempt to verify that certain properties about a design (e.g., user-specified

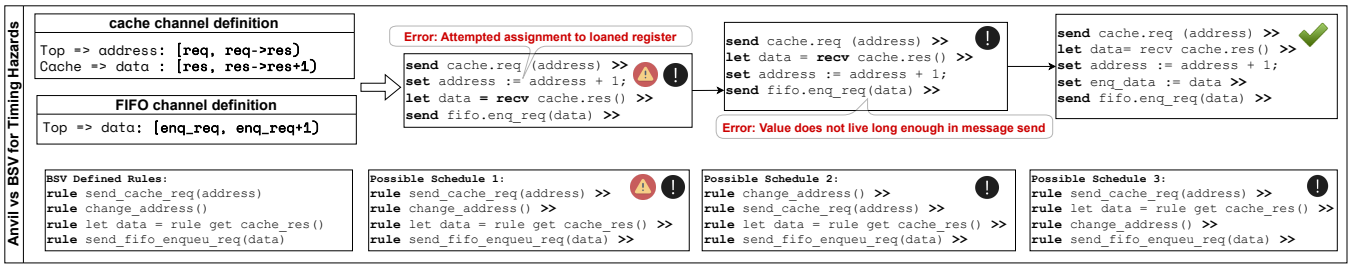

Figure 2. Top: Anvil guiding designer through timing-safe design. Bottom: BSV timing-unsafe schedules.

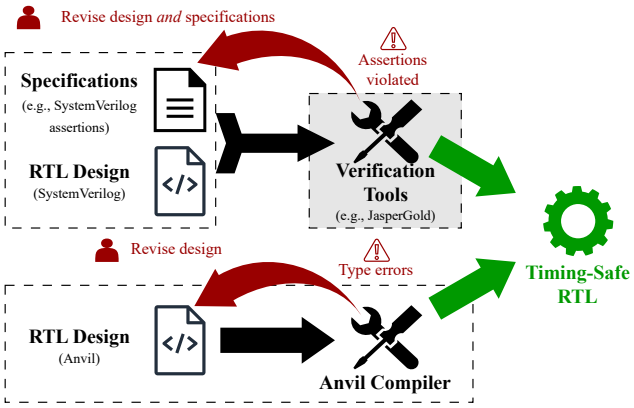

Figure 3. High-level comparison between the flows enabled by verification- (top) and language-based (bottom) approaches. Steps involving manual effort are marked with the *person* icon. White and gray dashed boxes represent design and verification stages, respectively.

SystemVerilog assertions) hold, either statically through formal verifications (e.g., model checking with Cadence JasperGold [7] or Yosys SMT-BMC [50]) or dynamically through testing (e.g., simulation-based verification with UVM [23] or cocotb [11]). This approach is general and may easily extend to other properties about an RTL design beyond timing safety. It is also readily applicable to existing code bases and does not require switching to a new language.

However, we have been motivated to focus on a *language-based* approach because of its unique advantages. As illustrated in Figure 3, a language-based approach can *preclude* designs with timing hazards *during* development. In contrast, verification detects timing hazards only *after the fact*, in a separate verification stage. This allows a faster and more integrated feedback loop. Through a language-based approach, the language abstractions themselves directly express the properties to be checked, for example, as part of a type system. A verification-based approach, on the other hand, requires manually specified, implementation-specific assertions to fill in missing information in the HDL abstraction. These assertions are error-prone and costly to maintain. A language-based approach can also present a more abstract model for reasoning about timing hazards efficiently. This avoids the state explosion problem with verification [10]. For example, bounded model checking may fail to report a violation even at large depths because of the prohibitive size of the model generated from SystemVerilog code. In Appendix B, we present a concrete example comparing Anvil —the language-based solution proposed in this paper—with verification-based methods to illustrate these points further.

2.4 Goal: a Timing-Safe and Expressive HDL

Some existing HDLs do provide timing safety. However, they face challenges in maintaining expressiveness. Some high-level synthesis (HLS) languages [49] provide abstractions of persistent values similar to variables in software programs. They abstract away certain aspects of hardware design such as register placements and cycle latencies. While their abstractions directly prevent timing hazards, they lack the precise timing and register control desired in general-purpose hardware design use cases, especially when the design needs to be latency-sensitive or efficient.

The closest prior work to ours is the Filament HDL [34]. Filament exposes cycle latencies and registers to the designer, and prevents timing hazards through its type system centred around *timeline types*. A timeline type encodes constraints regarding the time window in which each signal carries an unchanging value that can be used. Timeline types also serve to define contracts at module interfaces, allowing for safe composition of modules. Our example memory module can be augmented with such a contract which requires `input` and `req` to remain constant during $[T, T+2)$, and the output to remain constant in $[T+2, T+3)$. Figure 1 (bottom, right) illustrates the output waveform for a system using this contract. However, the timeline type and the contract it represents only capture timing intervals whose duration is fixed to be a statically determined, constant number of cycles. Correspondingly, Filament only aims to support pipelined designs with static timing. This prevents Filament from expressing common hardware designs such as caches and page table walkers that exhibit dynamic timing behaviour.

To see why this is the case, consider a memory subsystem with a cache. Its timing behaviour varies significantly between a cache hit and a cache miss. If the designer chooses a

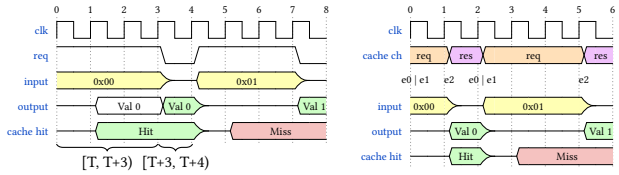

Figure 4. Cache output waveform expressed safely with static (left) and dynamic (right) timing contract.

conservative upper bound statically on the response time to accommodate both cases, the static timing contract would prevent timing hazards but nullify the advantage of caching. Figure 4 (left) illustrates the output waveform for such a system, where the contract uses the worst-case delay. In such cases, one must trade off the flexibility of dynamic latencies for the static guarantee of timing safety.

3 Timing Safety with Anvil

We present Anvil, an HDL with a novel type system that statically guarantees timing safety while retaining the level of expressiveness required for a general-purpose HDL. Unlike HLS languages that abstract away registers and cycle latencies, Anvil gives the designer full control over register mutations and cycle latencies. And unlike Filament [34], Anvil's type system can capture and reason about timing that varies during run-time. Anvil is thus able to enforce *dynamic timing contracts* across modules and precisely express hardware designs with dynamic timing behaviours.

Channels. Anvil models hardware modules as communicating processes [19]. It allows specifying modules with a process abstraction, using the keyword `proc`. A pair of communicating processes can share a bidirectional *channel*, through which they send and receive values. Channels are stateless and both sending and receiving are blocking. Channels are the only way for processes to communicate.

Events. A central concept that enables Anvil to reason about dynamic timing is *events*. Events are abstractions of time which may or may not statically map to a fixed cycle. The start of every clock cycle is an event that is statically known (constant). An example of a dynamic event is when two processes exchange a value through the channel. As described above, sending and receiving values on a channel are blocking. The exchange of the value thus completes at a time both sides agree on: when the sender signals the value is valid and the receiver acknowledges. The completion of this value exchange defines a dynamic event that may correspond to varying clock cycles during run-time. Note that both events and channel-based communication are only abstractions that Anvil provides, and under the hood, do not imply overhead in the resulting RTL design (see Section 6).

Event Graphs. A key observation enables Anvil to reason about events: even though we cannot statically know which exact cycle an event may correspond to, we know of the relationships among events. For example, we can statically obtain that event e_1 corresponds to exactly two cycles after the cycle e_2 corresponds to, and event e_3 corresponds to the first time a specific value is exchanged on a channel after the cycle e_2 corresponds to. Such relationships form an *event graph* (Section 5.3) which serves as the basis for Anvil's type system (Section 5.4 and Appendix D).

Lifetimes and Dynamic Timing Contracts. Anvil's type system uses events to encode the lifetime of a value carried by a signal. The lifetime of a value is identified by a start and an end event, between which the value is expected to remain steady. Channel definitions in Anvil specify the timing contracts for the exchanged values. Since events can be bound to varying concrete clock cycles at runtime, such timing contracts can capture *dynamic* timing characteristics. Enforcement of timing contracts ensures timing-safe composition of two processes when the events mentioned in the timing contract are known to both processes, e.g., when they correspond to value exchanges on the same shared channel.

Example: Anvil in Action. Figure 5 illustrates how Anvil's type system distinguishes between safe and unsafe process descriptions. The description `proc Top_Unsafe` is Anvil's representation (simplified for understanding) of the same circuit `Top` shown in Figure 1. In contrast, `proc Top_Safe` captures the timing characteristics of the memory subsystem with a cache, as depicted in Figure 4 (right). Anvil first derives the action sequence and then verifies whether the process description adheres to the constraints specified by the timing contracts. In our examples, `req` marks the clock cycle when `address` sent by `Top_Unsafe` or `Top_Safe` is acknowledged on the channel. The event `res` marks the clock cycle when `data` sent by the memory subsystem is acknowledged.

For memory without a cache, the expected behaviour is specified in a timing contract, encapsulated in the memory channel definition. This contract requires that `address` remain unchanging and available for two clock cycles after `req` is sent. It also specifies that `data` sent by memory must be available for one clock cycle after `res` is received.

The timing contract is not satisfied by `Top_Unsafe`, and Anvil detects this at compile time. In the HDL code for `Top_Unsafe`, `address` is sent during $[e_0, e_0 + 1)$, but the timing of acknowledgement is uncertain. The output value is used during $[e_0 + 1, e_0 + 2)$, but when `res` will be received is unknown, as it depends on when the memory system responds. As a result, it is unclear whether the next address was sent before the previous output was received and acknowledged. Furthermore, the input address is modified during $[e_0 + 1, e_0 + 2)$, violating the requirement that the address remain unchanging for two cycles after acknowledgement.

The contract for memory with a cache is specified in the cache channel definition. It requires that the `address` sent by `Top_Safe` remain available from the `req` event until the next

occurrence of `res`, written as the lifetime `(req, req->res)`. Similarly, the `data` sent by the memory subsystem has the lifetime `(res, res->res+1)`. As shown in Figure 5 (right), `Top_Safe` satisfies this contract and is therefore deemed safe.

Summary. Anvil is a general-purpose HDL that eliminates timing hazards. It allows designers to specify timing contracts and provides higher-level abstractions to enforce these contracts. The type system ensures that these contracts are respected. Anvil achieves this without sacrificing expressiveness. Dynamic contract definitions make it possible to design circuits with varying timing characteristics. It can capture timing characteristics precisely without introducing performance trade-offs such as additional latency.

Figure 4 shows the simulation output for Anvil's dynamic contracts (right) and static contracts (left). No extra clock-cycle overhead is introduced in either case. In addition, Anvil prevents the generation of unnecessary interface signals through *sync modes*, which specify the frequency at which a sender or receiver exchanges messages. We discuss this in more detail in Section 4.1.

4 Anvil HDL

In this section, we give a tour of novel language primitives in Anvil that are relevant to timing safety.

4.1 Channel

Anvil components communicate by message passing through bidirectional *channels*, which are akin to unbuffered channels in Go [14], where a send and its corresponding receive operations take place simultaneously. Each *channel type definition* in Anvil describes a template for channels, for example:

```
chan mem_ch {
  left rd_req : (logic[8]@#1) @#2-@dyn,
  left wr_req : (addr_data_pair@#1),
  right rd_res : (logic[8]@rd_req) @#rd_req+1-@#rd_req+1,
  right wr_res : (logic[1]@#1) @#wr_req+1-@#wr_req+1
}
```

Messages. The definition specifies the different types of *messages* that can be sent and received over a channel with two *endpoints*, referred to as left and right, respectively. Each type of message is identified by a unique message identifier and annotated with its direction, which is `left` (travelling left, i.e., from the right endpoint to the left endpoint) or `right` (travelling right).

Message Contracts. Each message is also associated with a *message contract*. This contract specifies the data type of the message and indicates the event after which the message content is no longer guaranteed to remain unchanging and should, therefore, be considered *expired*. Depending on the specified event of expiry, a message contract can be either static or dynamic. For example, message `rd_req` in the channel definition earlier has a static contract: It carries 8 bits of data, which expires 1 cycle after the synchronization on the message takes place. In contrast, message `rd_res` has a dynamic contract: It carries 8 bits of data which expires the next time message `rd_req` is sent or received.

Sync Mode. Each message has a *synchronization mode* (*sync mode* for short) for each side of the communication. The sync mode specifies the timing pattern for sending or receiving the message. In a message contract, the sync modes of both endpoints are specified in the format:

```
<left-endpoint-sync-mode>-<right-endpoint-sync-mode>
```

The default sync mode, `@dyn`, specifies that a one-bit signal is used for run-time synchronization. For example, in the channel definition, the message `wr_req` uses this dynamic sync mode on both endpoints. When static knowledge is available about when sending or receiving can occur, the sync mode can encode that information. The left side of the message `rd_req` has the *static sync mode* `@#2`. This specifies that it must be ready to receive the message within at most two cycles after the last time the message was received. For the left endpoint, Anvil statically checks that this constraint holds. For the right endpoint, Anvil uses this knowledge to check that whenever `rd_req` is sent, the receiver will be ready. A sync mode can also be *dependent*. For example, both sides of `wr_res` use `@#wr_req + 1`, meaning the message is sent and received exactly one cycle after `wr_req`.

4.2 Process

Each Anvil component is represented as a *process* defined with the keyword `proc`. A process signature specifies a list of endpoints to be supplied externally when the process is spawned. The process body includes register definitions, channel instantiations, (sub-)process spawning, and threads.

```
proc memory(ep1: left mem_ch, ep2: left mem_ch) {/* ... */}
```

4.3 Thread

Each process contains one or more threads that execute concurrently. Two types of threads exist: *loops* and *recursives*.

Loops. A loop is defined with `loop { t }`, where `t` is an Anvil term (see Section 4.4). This term can represent the parallel and sequential composition of multiple expressions. Each time `t` completes execution, the loop recurses back to the same behaviour. For example, the code below increments a counter every two cycles.

```
loop { set counter := *counter + 1 >> cycle 1 }
```

Recursives. A recursive, defined with `recursive { t }` generalizes loops to allow recursion before `t` completes. Instead, recusion is controlled with `recurse`. As `t` can restart before it completes, multiple threads may execute in an interleaving manner. Such constructs are therefore particularly useful

for expressing simple *pipelined* behaviours. For example, the code below pipelines the logic for handling requests. Specifically, it first waits to receive a request message, then performs two things in parallel: 1) handling the request, and 2) recursing (repeating the process from the beginning, where it starts waiting for the next request) in the next cycle. The direct pipelining support enabled by recursives is comparable to the pipelining support in Filament [34].

```
recursive {
  let r = recv ep.rd_req >>
  { /* handle request */ };
  { cycle 1 >> recurse }
}
```

4.4 Term

Terms are the building block for describing computation and timing control of threads in Anvil. Each term evaluates to a value (potentially empty) and the evaluation process potentially takes multiple cycles. In addition to literals and basic operators for computing (e.g., addition, xor, etc), notable categories of terms include the following.

Message Sending/Receiving. The terms `send e.m (t)` and `recv e.m` send or receive a specified message. The evaluation completes when the message is sent or received.

Cycle Delay. The term `cycle N` evaluates to an empty value after `N` cycles and is used entirely for timing control.

Timing Control Operators. The `>>` and `;` operators are used for controlling timing. See Section 4.5.

4.5 Wait Operator

The wait operator is a novel construct that enables sequential execution by advancing to a time point. In `t1 >> t2`, the evaluation of the first term `t1` must be completed before the evaluation of the second term begins. In contrast, `t1; t2` initiates both term evaluations in parallel. For example, `set r := t` and `set r := t; cycle 1` are equivalent, since register assignment takes one cycle to complete.

This design not only provides a way to advance time by explicitly specified numbers of cycles (e.g., `cycle 2 >> ...`). It also serves as an abstraction for managing and composing concurrent computations, in a way similar to the async-await paradigm for asynchronous programming. A term may represent computation that has not completed. Multiple terms can be evaluated in parallel. When the evaluation result of a term is needed, one can use `>>` to wait for it to complete. For example, the code below waits for messages from endpoints `ep1` and `ep2` and processes the data concurrently.

```
loop {
  let v1 = { let r = recv ep1.rd_req >> /* process r */ };
  let v2 = { let r = recv ep2.rd_req >> /* process r */ };
  v1 >> v2 >> ... /* now v1 and v2 are available */
}
```

4.6 Revisiting the Running Example

Figure 5 includes snippets of Anvil code for the running example introduced in Section 2. The code demonstrates how Anvil exposes cycle-level control and supports expressing dynamic timing behaviours. The code uses the wait operator to control when and how time is advanced. It is clear from the source code when each operation takes place relative to others. In the bottom right timing-safe Anvil code snippet, for example, incrementing `address` and updating `enq_data` take place at the same time (connected with `;`), and sending of `fifo.enq_req` starts one cycle afterwards, when both register updates complete. Such timing control does not have to rely on fixed numbers of cycles. For example, the two register updates discussed above take place after `cache.res` is received, which in turn takes place after `cache.req`. The exact numbers of cycles those operations vary during run-time depending on the interaction between `Top` and `Cache`.

Despite those dynamic timing behaviours that Anvil code can express, Anvil is able to reason about them and ensure timing safety statically, as we will discuss in detail next.

5 Safety of Anvil Programs

Anvil's type system ensures that each process adheres to the contracts defined by the channels it uses. The guarantee the type system provides is as follows: any well-typed process in Anvil can be composed with other well-typed processes without timing hazards at run-time. To provide such guarantees, the type system associates each term with an abstract notion of a *lifetime*, which, intuitively, captures the time window in which its value is unchanging and meaningful. Each register, likewise, is associated with a *loan time*, which describes when it is *loaned*, i.e., needs to remain unchanged. The abstractions of lifetime and loan time form the foundation for ensuring safety in Anvil. Based on them, the type system checks for the following properties for a process —

1. **Valid Value Use:** Every use of a value falls in its associated lifetime.
2. **Valid Register Mutation:** A register mutation does not take place during its loan time.
3. **Valid Message Send:** The time window for which the data sent needs to be live (based on the timing contract) is covered by its associated lifetime. Additionally, such time windows do not overlap for two send operations of the same message type.

A formal presentation of the type system and the safety guarantees of Anvil is available in Section 5.5. We first explain the intuition behind them in this section.

5.1 Events and Event Patterns

Anvil reasons about events which correspond to the times specific terms complete evaluation. Note that such interesting events as sending and receiving of messages and elapse of a number of cycles are naturally included, as the those

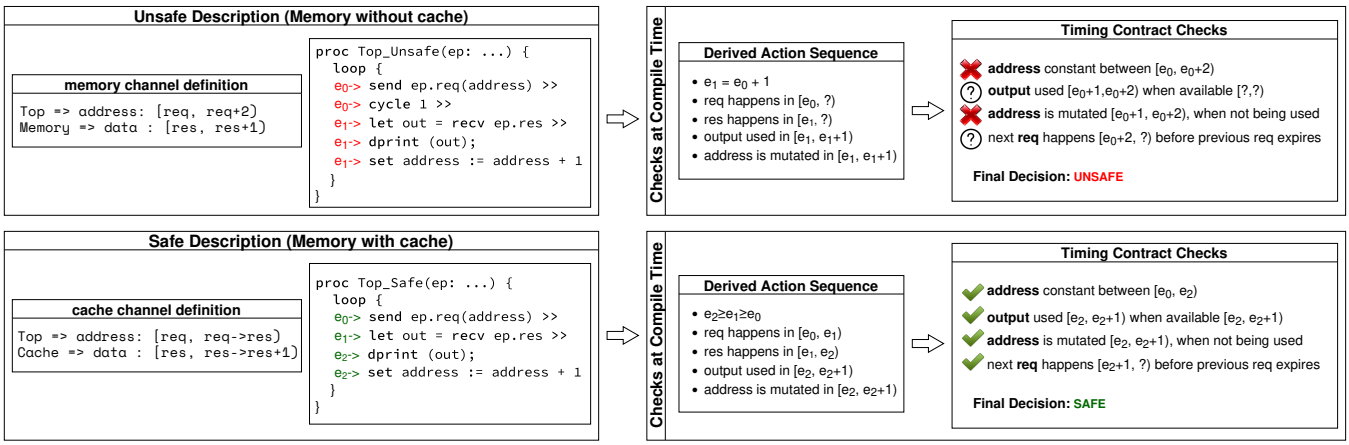

Figure 5. Anvil checking the unsafe version of `Top` interfacing with memory subsystem with and without cache.

operations are all represented as terms (Section 4.4). *Event patterns* can then be defined based on such events. A basic event pattern is of the form $e \triangleright p$, which consists of an existing event e and a *duration* p and specifies the time when a condition specified in duration p is first satisfied after e. The duration can be either static or dynamic. A static duration specifies a fixed number of clock cycles, in the form of $\#N$. A dynamic duration specifies a certain operation ω, in which case $e \triangleright p$ refers to when ω is first performed after e. During run-time, a dynamic duration can correspond to variable numbers of cycles. The typical example of a dynamic duration is the sending or receiving of a specified message type through a channel. In our discussion, this is represented as $\pi.m$, where π is the endpoint name and m is the message identifier. Multiple event patterns can be combined as a set of event patterns $\{e_i \triangleright p_i\}_i$ to form a new event pattern, which refers to the earliest event specified with each $e_i \triangleright p_i$.

5.2 Lifetime and Loan Time

Lifetime. The lifetime represents the interval during which a value is expected to remain unchanging (constant). Anvil infers a lifetime for each value, represented by an interval $[e_{\text{start}}, S_{\text{end}})$, where an event e_{start} and an event pattern S_{end} mark the beginning and end of the interval. During run-time, the events e_{start} and S_{end} will correspond to specific clock cycles. Since each signal carries a value, it inherently has an associated lifetime. At any given instant, a signal is *live* if it falls within its defined lifetime. Conversely, it is *dead*.

Loan Time. Since signals and messages may source values from registers, Anvil tracks the intervals during which a register is loaned to a signal by associating each register with a loan time. The loan time of a register r is a collection of intervals. For each interval included in the loan time, r should not be mutated. Anvil infers the lifetime for all associated values and the loan time for all registers. Consider the example in Figure 6 (left) of a component named `Encrypt`. This component performs encryption on the plaintext received through the endpoint `ch1` using random noise obtained via the endpoint `ch2`. The following are examples of the lifetimes and loan times that Anvil infers:

- The signal `ptext` is bound to a message identified by `enc_req` received on the endpoint `ch1`. Its lifetime is inferred from the channel type definition as $[e_1, e_1 \triangleright \texttt{ch1.enc\_res})$, where e_1 is the event of the message being received.
- The signal `r1_key` is a constant literal and therefore has an *eternal* lifetime, represented with ∞ as its end event. i.e., it can always be used.
- The signal `ctext_out` is used as a value sent as a message from the endpoint `ch1`. Its inferred lifetime begins at the evaluation of the term, represented as e_5, and extends until the message on `ch1` expires, which is $e_9 \triangleright \texttt{ch1.enc\_req}$, where e_9 is the event corresponding to the assignment completion. Therefore, the lifetime is $[e_5, e_9 \triangleright \texttt{ch1.enc\_req})$.
- The signal `(ptext ^ r1_key) + noise` has a lifetime that is the intersection of the lifetimes of `ptext`, `r1_key`, and `noise`, $[e_3, \{e_2 \triangleright \#1, e_1 \triangleright \texttt{ch1.enc\_res}\})$.
- The register `rd2_key` is loaned by a message sent through the endpoint `ch2` and the signal `ctext_out`. Based on the specified timing in the channel type definition `rng_ch`, the lifetime of the message is $[e_5, e_8 \triangleright \#2)$, where e_8 is the event of the message sending completion. Therefore, `rd2_key` has an inferred loan time $[e_5, e_9 \triangleright \texttt{ch1.enc\_req}) \cup [e_5, e_8 \triangleright \#2)$.

See Figure 6 (left) for more examples of inferred lifetimes.

5.3 Event Graph

Events are related to one another by their associated operations. For example, an event e_a may be precisely two cycles after another event e_b. As another example, e_a can refer to the completion of a *specific* message that *starts* at e_b. In general,

```
chan encrypt_ch {
    left enc_req : (logic[8]@enc_res), right enc_res : (logic[8]@enc_req)
}
chan rng_ch {
    left rng_req : (logic[8]@#1), right rng_res : (logic[8]@#2)
}
proc Encrypt(ch1 : left encrypt_ch, ch2 : left rng_ch) {
    /* ... register definitions ... */
    loop {
     e0 let ptext[e1, e1 ▷ ch1.enc_res) = recv ch1.enc_req;
     e0 let noise[e2, e2 ▷ #1) = recv ch2.rng_req;
     e0 let r1_key[e0, ∞) = 25;
     e0 ptext[e1, e1 ▷ ch1.enc_res) >>
     e1 if ptext != 0 {
     e1   noise[e3, e2 ▷ #1) >>
     e3   set rd1_ctext := (ptext ^ r1_key) + noise[e3, {e2 ▷ #1, e1 ▷ ch1.enc_res})
     e1 } else { rd1_ctext := ptext[e1, e1 ▷ ch1.enc_res) };
     e1 cycle 1 >>
     e5 set r2_key := r1_key ^ noise[e6, e2 ▷ #1);
     e5 let ctext_out = *rd1_ctext ^ *r2_key[e5, e9 ▷ ch1.enc_req);
     e5 send ch2.rng_res(*r2_key[e5, e8 ▷ #2)) >>
     e8 send ch1.enc_res(ctext_out[e8, e9 ▷ ch1.enc_req)) >>
     e9 send ch1.enc_res(r1_key[e9, ∞))
    }
}
```

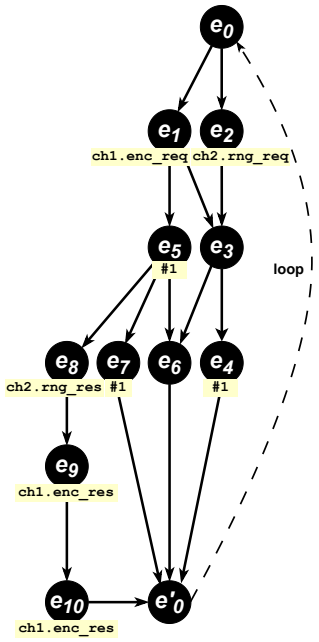

Figure 6. Left: `Encrypt` in Anvil, annotated with timing information. Each blue-shaded annotation marks the event corresponding to the time a term evaluation starts. Each yellow-shaded annotation marks the inferred lifetime associated with the red-circled term next to it. Right: Event graph corresponding to `Encrypt`. Branch-related constructs which exist in the event graph actually used in the type system are omitted for brevity. The operations associated with some of the events are presented in yellow labels.

events and their interrelationships form a directed acyclic graph (DAG), with each node being an event labelled with its associated operation. We call such a DAG an *event graph*. `Encrypt` in Figure 6 (left), for example, has an event graph as shown in Figure 6 (right). The event graph captures the events in one loop iteration only, with event e_0 representing the start of a loop iteration. The event e_0' corresponds to e_0 of the next loop iteration.

An event graph encodes sufficient information to capture all possible timing behaviours in run-time. Intuitively, once we replace each non-cycle operation label (e.g., those associated with e_1, e_2, e_8, e_9, and e_{10} in Figure 6 (right)) with a cycle number that represents the actual amount of time taken to complete the message passing, we can deterministically obtain the exact time (in cycles) each event occurs.

5.4 Safety Checks

Building Blocks: $\leq_G$ and $\subseteq_G$. Based on an event graph G, Anvil compares pairs of events as to the order in which they occur during run-time. In particular, Anvil decides if an event e_a always occurs no later than another event e_b, denoted as $e_a \leq_G e_b$. The simple scenario is when a path exists from e_a to e_b in G and we directly have $e_a \leq_G e_b$. More complex scenarios involve events with no paths between them, which Anvil handles by considering the "worst" cases time gap between when the two events are reached. For example, we have $e_5 \leq_G e_4$, as even in the worst case (receiving `ch2.rng_req` takes 0 cycles), e_4 and e_5 still occur at the same time. We naturally extend the definition of $\leq_G$ to cover event patterns and reuse the notation $S_a \leq_G S_b$.

With $\leq_G$, the Anvil type system can decide that an interval $[e_a, S_a)$ is always fully within another interval $[e_b, S_b)$, denoted $[e_a, S_a) \subseteq_G [e_b, S_b)$, if $e_b \leq_G e_a$ and $S_a \leq_G S_b$. It then decides if the lifetimes and the loan times comply with the three types of constraints. We use the example in Figure 6 to explain them below.

Valid Value Use. Anvil's type system verifies that events at which a signal is used are within its defined lifetime. A use of `ptext` occurs at e_1 in the expression `if ptext != 0 { ... }`, where it has a a lifetime of $[e_1, e_1 \triangleright \texttt{ch1.enc\_res})$. It requires `ptext` to be live for one cycle, i.e., in $[e_1, e_1 \triangleright \#1)$. Anvil checks that $[e_1, e_1 \triangleright \#1) \subseteq_G [e_1, e_1 \triangleright \texttt{ch1.enc\_res})$, which holds in this case. Hence Anvil determines that `ptext` is guaranteed to be live during this read.

However, in `rd1_ctext := (ptext ^ r1_key) + noise`, the signal `(ptext ^ r1_key) + noise` cannot be statically guaranteed to be live. In this case, Anvil compares its lifetime, $[e_3, \{e_2 \triangleright \#1, e_1 \triangleright \texttt{ch1.enc\_res}\})$ with the time when it is used, $[e_3, e_3 \triangleright \#1)$ (the assignment starts at event e_3 and takes one cycle to complete). It cannot obtain $e_3 \triangleright \#1 \leq_G \{e_2 \triangleright \#1, e_1 \triangleright \texttt{ch1.enc\_res}\}$. Intuitively, if it takes more cycles to receive `ch1.enc_req` (e_1) than `ch2.rng_req` (e_2), `noise` will already be dead at e_3 when the assignment happens.

Valid Register Mutation. Anvil ensures that each register value remains constant during its loan time. For the example in Figure 6, the loan time for `r2_key` is $[e_5, e_9 \triangleright \texttt{ch1.enc\_req}) \cup [e_5, e_8 \triangleright \#2)$. To determine if `r2_key` is still loaned when the assignment `r2_key := r1_key ^ noise` takes place, Anvil checks if $[e_5, e_7 \triangleright \#1)$ is guaranteed not to be fully covered by any interval in its loan time, i.e., for every $[e', S')$ in the loan time, either $S <_G S'$ or $e' <_G e$ must hold. Here, e_7 is the event that corresponds to the assignment completion, exactly one cycle after e_5, which corresponds to when the assignment starts. In other words, e_5 and e_7 are adjacent cycles in which the register can carry different values. If an interval in the loan time may contain both e_5 and e_7, at run-time during the interval the register value may change. In the example, $[e_5, e_8 \triangleright \#2)$ potentially (surely in this case) fully covers $[e_5, e_7 \triangleright \#1)$, hence this assignment conflicts with the loan time of `r2_key` and is disallowed by Anvil. Intuitively, a value sourced from `r2_key` is sent through `ch2.rng_res` at e_5, which requires it to be live until two cycles after the send completes. However, `r2_key` already changes one cycle after e_5.

Valid Message Send. In the example in Figure 6, the term `send ch1.enc_res(r1_key)` attempts to send a new message before the previous `enc_res` message sent by the endpoint `ch1` has expired. During run-time on the other end of channel, this can lead to signals received through `enc_res` to change, violating the message contract. Anvil detects such violations by examining whether the required lifetimes of the two send operations are disjoint. The example violates such constraints as $[e_8, e_9 \triangleright \texttt{ch1.enc\_req})$ and $[e_9, e_{10} \triangleright \texttt{ch1.enc\_req})$ are overlapping. Anvil also checks that the lifetimes of sent signals cover the required lifetime specified by the message contract. For example, the send through `ch1.enc_res` at e_9 checks that the lifetime of `r1_key` covers the required lifetime $[e_9, e_{10} \triangleright \texttt{ch1.enc\_req})$. In this case, this check passes as $[e_9, e_{10} \triangleright \texttt{ch1.enc\_req}) \subseteq_G [e_9, \infty)$.

5.5 Formalization

Figure 7 presents the syntax of Anvil. Anvil's type system guarantees that any well-typed Anvil program is timing-safe. Due to space limits, we leave the formal details of the semantics, the type system of Anvil, the safety definitions, and proofs to Appendices D and E.

process definition	$P ::= \texttt{proc}\ p(\pi, \ldots)\{B\}$
process body	$B ::= \emptyset \mid \texttt{reg}\ r : \delta; B \mid \texttt{ch}\ c(\pi, \pi); B$
	$\mid \texttt{spawn}\ p(\pi, \ldots); B \mid \texttt{loop}\ \{t\}\ B$
term	$t ::= \texttt{true} \mid \texttt{false} \mid () \mid \texttt{cycle}\ N \mid x \mid *r$
	$\mid t \Rightarrow t \mid \texttt{let}\ x = t\ \texttt{in}\ t \mid \texttt{ready}\ (\pi.m)$
	$\mid \texttt{if}\ x\ \texttt{then}\ t\ \texttt{else}\ t \mid \texttt{send}\ \pi.m(x)$
	$\mid \texttt{recv}\ \pi.m \mid r := t \mid t \star t \mid \pentagon t$

$\delta \in$ data-types $\quad \star \in$ binary-operators $\quad \pentagon \in$ unary-operators

$\pi \in$ endpoints $\quad x \in$ identifiers $\quad r \in$ registers $\quad m \in$ messages

$c \in$ channels $\quad p \in$ processes $\quad N \in \mathbb{N}$

Figure 7. Anvil syntax.

6 Implementation

We have implemented Anvil in OCaml. The Anvil compiler performs type checking on Anvil code and generates synthesizable SystemVerilog code. We have publicly released the compiler at https://github.com/kisp-nus/anvil.

The compiler uses the event graph as an intermediate representation (IR) throughout the compilation process. It constructs an event graph from the concrete syntax tree of the Anvil source code, performs type checking on it, and lowers it to SystemVerilog. Optimizations are applied to the event graph both before and after type checking. Since event graph construction and type checking follow the type system in a straightforward manner, we focus on the optimization and lowering strategies in this section.

6.1 Event Graph Optimizations

Optimizations aim to reduce the number of events in the event graph while keeping its semantics unchanged. The Anvil compiler performs optimizations in *passes*, with each pass applying a specific optimization strategy. Figure 8 shows examples of such optimization passes. The figure illustrates simplified event graphs during optimization. Edge labels (including their colours illustrated in the figure) describe the timing relationships between events (nodes in the figure). A blue edge from e_a to e_b represents that e_b waits for a fixed delay after e_a, with the number of cycles indicated in #N). When an event waits on multiple other events, i.e., with multiple inbound blue edges, it occurs at the latest of the specified time points. Red edges represent branching. When e_r has red edges to both e_a and e_b, either of them, but not both, occurs in the same cycle as e_r. Orange edges in turn join branches: When e_a and e_b both have orange edges to e_c, e_c occurs in the same cycle when either of them occurs. Triangles represent sets of edges. Recall that events represent abstract time points. In a concrete run, they occur in specific cycles (or are never reached). In general, two events can be merged if they always occur at the same time. Many of the optimization passes are based on identifying such cases.

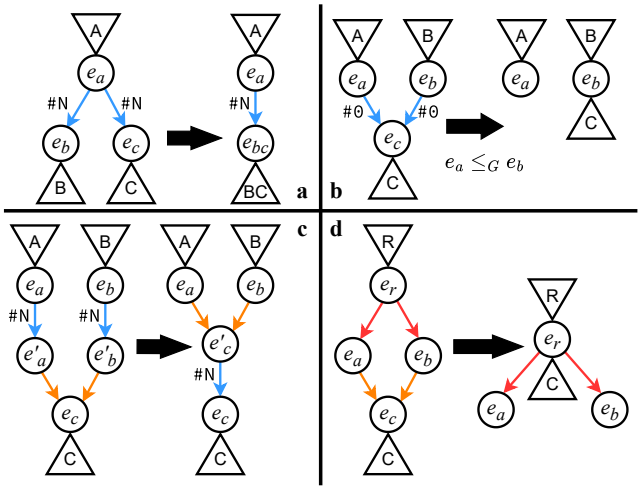

Figure 8. Examples of event graph optimizations. The event graphs are simplified for illustration purposes. Edges describe timing relationships: blue for fixed cycle delays, red for branching, and orange for joining branches. Triangles represent sets of edges.

(a) Merging identical outbound edge labels. This optimization pass merges outbound edges of an event e_a that share the same label. For example, edges labelled with `#N` going to e_b and e_c. The events those edges connect to are merged. A shared label implies that they occur at identical delays from the parent event.

(b) Removing unbalanced joins. This optimization pass removes an e_c with two predecessors when either of its predecessors (e_b) always occurs no earlier than the other (e_a), i.e., $e_a \leq_G e_b$. In this case, e_c is unnecessary, and its outbound edges are migrated to e_b.

(c) Shifting branch joins. When the ending events of the two branches e'_a and e'_b are both derived with N cycles delay after their predecessors e_a, e_b, and have no associated actions (e.g., register assignments or message sends/receives), the event e_c that joins the two can be shifted earlier. Instead of delaying by N cycles and then joining, the branches can join first into e'_c and then delay by N cycles.

(d) Removing branch joins. If an event e_c joins two branches where the ending events e_a and e_b are also the first events of their branches, and both share the same predecessor e_r, then e_c can be merged into e_r. This means that if two branches take no delay, their joining event can be merged into the predecessor. All actions of the joining event are then performed in the predecessor event.

6.2 Code Generation

The Anvil compiler maps each Anvil process to a SystemVerilog module. For each process, it generates module input/output ports for channel communication and a finite state machine (FSM) for control flow based on the event graph. Note that the compiler generates *no* extra code for maintaining lifetimes or enforcing timing safety as it reasons about lifetimes statically and guarantees timing safety through static type checking. As such, they incur no overhead in the generated hardware design.

Message Lowering. Each message in an endpoint maps to up to three module ports: `data`, `valid`, and `ack`. The `data` port carries the communicated data, while `valid` and `ack` are handshake ports for synchronization. The compiler only generates both `valid` and `ack` when the specified sync mode is dynamic for both the sender and the receiver (see Section 4.1). If the sync mode for either side is static or dependent, the compiler omits the corresponding port (`valid` for the sender and `ack` for the receiver). In particular, both handshake ports are omitted for a sync mode that is not dynamic on either side, leaving `data` as the only port generated.

FSM Generation. The compiler generates the FSM based on the event graph structure. For each event, it uses a one-bit wire `current` to indicate if the event has been reached. For some events, the compiler also generates registers to record the current state. Such events include: (a) Joins: which predecessors have been reached; (b) Cycle delays: cycle count; (c) Send/receive events (only those with dynamic sync modes): whether the message has been sent or received.

7 Evaluation

We aim to answer three questions through evaluation:

1. **Expressiveness:** Can Anvil express diverse hardware designs, without incurring any latency[1] overhead?
2. **Safety:** Can Anvil assist the designer to express and meet the implicit timing contracts?
3. **Practicality:** What overheads do Anvil-generated hardware designs incur in synthesis?

Artefacts. We have released the evaluation artefacts publicly for reproducibility. Refer to Appendix A for information.

7.1 Expressiveness

SystemVerilog supports describing circuits with arbitrary latencies. To assess expressiveness, we evaluate designs created in Anvil against open-source designs written in SystemVerilog. We also compare Anvil with Filament [34], which provides specialized abstractions for static pipelines.

Common Cells Benchmarks. Anvil is designed to be a general-purpose HDL. To test this, we implemented various hardware components with different behaviours. Specifically, we implemented a first-in first-out (FIFO) buffer, a spill register, and a passthrough stream FIFO (which allows read and write in the same cycle). These are taken from the Common Cells IP and are highly optimized designs for synthesis [41]. With Anvil, we replicated these designs while ensuring identical functional behaviour through unit tests. Importantly,

[1]Latency refers to clock cycle latency and not propagation delay.

Table 1. Summary of area and power footprints of Anvil and baseline designs in SystemVerilog and Filament. *SV* stands for SystemVerilog and *dyn* indicates dynamically varying cycle latencies.

Hardware Designs	Area (um^2)		Power (mW)		f_{max} (MHz, ±50)		Latency (cycles)	
	Baseline	Anvil	Baseline	Anvil	Baseline	Anvil	Baseline	Overhead
FIFO Buffer (SV)	690	674 (−2%)	1.434	1.403 (−2%)	4062	4156	dyn	0
Spill Register (SV)	165	171 (3%)	0.459	0.469 (2%)	5187	5375	dyn	0
Passthrough Stream FIFO (SV)	679	679 (0%)	1.239	1.264 (2%)	4093	3625	1	0
CVA6 Translation Lookaside Buffer (SV)	5561	5611 (0%)	5.813	5.835 (0%)	2468	2406	dyn	0
CVA6 Page Table Walker (SV)	499	561 (12%)	0.649	0.676 (4%)	3531	3281	dyn	0
AES Cipher Core (SV)	9096	9090 (0%)	0.793	0.972 (22%)	781	1229	dyn	0
AXI-Lite Demux Router (SV)	1318	1469 (11%)	1.351	1.385 (2%)	2437	2125	dyn	0
AXI-Lite Mux Router (SV)	1448	1633 (12%)	1.336	1.324 (0%)	2406	2187	dyn	0
Average overhead compared with SystemVerilog baselines: Area = 4.50%, **Power** = 3.75%								
Pipelined ALU (Filament)	501	404 (−19%)	0.658	0.678 (3%)	3312	4675	1	0
Systolic Array (Filament)	2522	2434 (−3%)	2.533	2.808 (10%)	2437	2862	1	0
Average overhead compared with Filament baselines: Area = −11.0%, **Power** = 6.5%								

Anvil is able to express their dynamic behaviour *without introducing any latency overhead.*

CVA6 MMU. We implemented the translation lookaside buffer (TLB) and the page table walker (PTW), which together form the core of the memory management unit (MMU) in the CVA6 RISC-V core [52]. These units are highly sensitive to dynamic latencies, which static contracts cannot capture. For example, the PTW incurs varying latencies per request due to its dependency on the data cache for fetching page table entries. Anvil replicates the same functional behaviour (verified with baseline RISC-V smoke tests) *without incurring any cycle-level latency overhead over the baselines.*

OpenTitan AES Accelerator [38]. We implemented the unmasked AES cipher core from OpenTitan. This core supports encryption, decryption, and on-the-fly key generation for AES-128 and AES-256. We verified its functional behaviour using unit tests for encryption and decryption of plaintext. The core has a clock-cycle latency proportional to the number of encryption rounds, and it flushes its state during operation. These characteristics make the latency dynamic, and Anvil is able to replicate this behaviour. The original AES core uses an S-box implementation intended for LUT mapping. To stay consistent with this design choice, we used the baseline S-box IP optimized for LUT realization.

AXI-Lite Routers. Anvil abstracts communication interfaces using channels. To demonstrate the utility of this abstraction in real-world components, we implemented the AXI-Lite demux router and AXI-Lite mux router with fair arbitration. The AXI protocol itself is designed to provide a channel-like interface between master and slave components. We verified the correctness of our implementations using unit tests with configurations of 8 slaves and 1 master, and vice versa. These routers can be composed into an AXI crossbar according to the desired configuration. With Anvil, we replicated the same functional behaviour while abstracting away the complexity of handling transaction requests from the user. As in all our experiments, this design also does not incur any additional latency overhead.

Pipelined Designs. Lastly, to demonstrate the ability of Anvil recursives (Section 4.3) to express static pipelined designs, we implemented a pipelined ALU and a pipelined systolic array. We compared these implementations against hardware designs generated by Filament. The evaluation shows that Anvil allows for expressing such designs without incurring any additional penalty.

> **Takeaway.** Anvil provides cycle-level timing control and precise expression of dynamic latency, with no additional cycle latency or throughput overhead.

7.2 Safety

During our evaluation, we observed issues with the stream FIFO. According to the IP documentation, the design goal is clear: Reads are allowed only when the FIFO is not empty, and writes only when it is not full. Additionally, if there is a read and write request in the same cycle and the FIFO is full, it should still allow the write.

However, we noticed that the original FIFO, even with a handshake interface, does not actually prevent writes at all. Instead, it relies on warning assertions (SVA) to alert designers if they run into such cases. This means that unless the design hits a specific overflow condition, no assertion is raised. Moreover, once the overflow happens, there is no further assertion until the FIFO again reaches its full depth. This behaviour is ambiguous and is intended for revision as confirmed with the maintainers [42].

This creates a gap between the documented contract and the actual behaviour. The design does contain possible timing hazards and effectively pushes the responsibility onto the designer to avoid them. In contrast, Anvil enforces these contracts directly, and as we observe, does so without incurring significant overhead. There are several such examples of timing hazards in open-source IPs, where enforcement is

either left to the designer or sometimes not handled at all. We discuss several such instances in Appendix C.

7.3 Practicality

To evaluate the practicality of the generated designs, we synthesized all of them on a commercial 22 nm ASIC process. This shows how well Anvil designs scale during synthesis compared to SystemVerilog, which is widely regarded as the most efficient option for practical hardware. We then provide a detailed analysis of the sources of overhead and efficiency in these designs. Table 1 summarizes the resource consumption of circuits generated with Anvil.

Setup. We report the area, power, the maximum frequencies ($f_{\max}$) at which designs do not violate slack requirements, and the clock cycle latency of the baseline. The area and power are reported at $\min(f_{\max}(\text{Anvil}), f_{\max}(\text{baseline}))/2$.

Propogation Delay. The maximum frequency evaluation shows that Anvil is able to synthesize circuits that are not worse than the baseline in supporting higher frequencies. This is primarily because the critical path is the same in both designs. The additional propagation delay only comes from the extra combinational logic introduced by code generation. For pipelined designs, Anvil achieves higher maximum frequency than the baseline.

Area. Anvil provides constructs that implicitly generate state machines as efficiently as handwritten ones. This is reflected in the area overheads when compared against handwritten baselines. The overhead in non-combinational area across all designs is equivalent to, or in some cases even lower than, the baseline implementations.

For example, in the case of the AXI router, the observed overhead for the AXI demux (1469 vs. 1318) arises entirely from the FIFO component. The FIFO is required to preserve the ordering of transactions on the AW/AR channels relative to their corresponding W/B/R channels. The router instantiates three FIFOs in total. Each FIFO contributes roughly 45 units of area overhead, as the select signal width is only 3 bits. However, as the data width increases, the relative overhead becomes demagnified. This trend is evident in the 32-bit FIFO buffer results reported in Table 1.

A similar observation holds for the PTW. Here, the non-combinational area is comparable (330 vs. 352), while the combinational area shows a modest gap (168 vs. 208). This difference essentially reflects a fixed cost of Anvil's code generation. As a result, the relative overhead is more pronounced for small-area designs but negligible for larger ones.

Power. The power overhead in Anvil arises primarily from bundling signals and flattening data structures. In this representation, the synthesis toolchain may treat the entire bundle as active, even when only a portion is in use. Consequently, switching activity, and thus dynamic power, increases with datapath width, as observed in the AES cipher core. At the same time, the Anvil compiler can reduce leakage power because all signals are explicitly connected, leaving none floating. Additionally, register assignments for explicitly declared registers occur only when the corresponding event is triggered. This behaviour implicitly provides clock gating for some register writes.

Summary. Overall, Table 1 shows that Anvil achieves area efficiency on par with handwritten SystemVerilog, with overheads typically within 12% and averaging 4.50%. Power overheads are more noticeable in wide datapath designs (e.g., the AES cipher core) due to increased switching activity, but remain modest overall (averaging 3.5%). The maximum frequencies are generally on par with handwritten SystemVerilog, and in pipelined cases, even exceed the baseline. Importantly, none of the designs introduce extra cycle latency.

> **Takeaway.** Anvil is practical for creating real-world hardware designs with minimal area/power overheads and seamlessly integrates into existing SystemVerilog designs.

8 Related Work

Timing-Oblivious HDLs. The industry-standard HDLs, SystemVerilog [22] and VHDL [21], describe hardware behaviours with dataflows involving registers and wires within single cycles. This abstract model equips them with low-level expressiveness but is not conducive to time-related reasoning, causing such problems as timing hazards. Embedded HDLs [3, 12, 43, 45] use software programming languages for hardware designs for their better parameterization and abstraction capabilities. They follow the same single-cycle model as in SystemVerilog and VHDL. Bluespec SystemVerilog [5, 35] provides an abstraction of hardware behaviours with sequential firing of atomic rules. It is still limited to describing single-cycle behaviours and does not provide timing safety. Higher-level HDLs, high-level synthesis (HLS) languages, and accelerator description languages (ADLs) [20, 24, 44, 49, 51] specialize in specific applications and abstracts away cycle-level timing and the distinction between stateless signals and registers.

Timing-Aware HDLs. Filament [34] achieves timing safety with timeline types which only support statically fixed delays. As a result, it is limited to designs with static timing behaviours. HIR [31] is an intermediate representation (IR) for describing accelerator designs. It introduces time variables to specify timing, and allows specifying a static delay for each function to indicate when it returns. HIR abstracts away the distinction between signals and registers and does not capture the notion of lifetimes and only supports static timing behaviours. Piezo [27] is an IR that supports specifying both static and dynamic timing through timing guards.

Hazard Prevention. BaseJump [46] and Wire Sorts [8] are type systems deigned to identify combinational loops, a separate concern than timing hazards. ShakeFlow [17] proposes a dynamic control interface to prevent structural hazards

in pipelined designs. Hazard Interfaces [25] generalizes it further to cover data and control hazards as well. Both focus on higher-level notions of hazards than timing hazards on high-level abstractions specialized for pipelined designs.

RTL Verification. Verification techniques focus on more general specifications for RTL designs, e.g., those based on temporal logics [6, 16, 33, 39]. In practice, desired properties are typically specified as assertions in source code, which are verified either through testing [11, 23] or through formal methods such as model checking [7, 48, 50]. Compared with Anvil, verification-based techniques cover more general properties, but suffer from a long feedback loop resulting from a separate verification stage, the extra burden of maintaining implementation-specific specifications, and tractability issues such as state explosion. Section 2.4 compares Anvil with verification-based methods in more detail.

9 Conclusions

In this work, we formalize the problem of timing hazards and present Anvil, a hardware description language that provides timing safety by capturing and enforcing timing requirements on shared values in timing contracts. Anvil ensures safe use of values, guaranteeing that they remain unchanged throughout their lifetimes. Meanwhile, it provides the expressiveness for cycle-level timing control and for describing designs with dynamic timing characteristics.

Acknowledgments

We thank NUS KISP Lab members and anonymous reviewers for their feedback, and Yaswanth Tavva and Sai Dhawal Phaye for their help with infrastructure setup. This research is funded, in part, by Singapore Ministry of Education Tier 2 grants MOE-T2EP20124-0007 and MOE-T2EP20222-0007.

References

[1] Alexforencich. 2022. Tx signals for raw ethernet frame Issue 121 alexforencich/verilog-ethernet. https://github.com/alexforencich/verilog-ethernet/issues/121 Accessed: 2024-11-12.

[2] C. Baay. 2015. *Digital Circuits in ClaSH.* Ph. D. Dissertation. University of Twente. doi:10.3990/1.9789036538039

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference (DAC).* 1216–1225. doi:10.1145/2228360.2228584

[4] Bluespec, Inc. 2008. Bluespec SystemVerilog Reference Guide.

[5] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 243–257. doi:10.1145/3385412.3385965

[6] P. Camurati and P. Prinetto. 1988. Formal Verification of Hardware Correctness: Introduction and Survey of Current Research. *Computer* 21, 7 (1988), 8–19. doi:10.1109/2.65

[7] Candence [n. d.]. Jasper Formal Verification Platform. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html Accessed: 2025-08-21.

[8] Michael Christensen, Timothy Sherwood, Jonathan Balkind, and Ben Hardekopf. 2021. Wire Sorts: a Language Abstraction for Safe Hardware Composition. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI).* 175–189. doi:10.1145/3453483.3454037

[9] Abhishek Chunduri. 2020. '1011' Overlapping (Mealy) Sequence Detector in Verilog. https://electronics.stackexchange.com/questions/505795/1011-overlapping-mealy-sequence-detector-in-verilog Accessed: 2024-11-14.

[10] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. 2012. Model Checking and the State Explosion Problem. In *Tools for Practical Software Verification, LASER 2011, Lecture Notes in Computer Science*, Bertrand Meyer and Martin Nordio (Eds.). 1–30. doi:10.1007/978-3-642-35746-6_1

[11] Cocotb [n. d.]. Cocotb. https://www.cocotb.org/ Accessed: 2025-08-21.

[12] Jan Decaluwe. 2004. MyHDL: a Python-Based Hardware Description Language. *Linux J.* 2004, 127 (2004), 5.

[13] Dimitras-Vtool. 2024. Alu_full_fifo_in_test · Issue #1 · Dimitras-Vtool/ALU. https://github.com/dimitras-vtool/ALU/issues/1. Accessed: 2024-11-14.

[14] Alan A.A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language* (1st ed.).

[15] fpgasystems. 2024. Each completion queue contains 2-cycle burst valid signal | Issue 78 | fpgasystems/Coyote. https://github.com/fpgasystems/Coyote/issues/78 Accessed: 2024-11-12.

[16] Aarti Gupta. 1992. Formal Hardware Verification Methods: A Survey. (1992).

[17] Sungsoo Han, Minseong Jang, and Jeehoon Kang. 2023. ShakeFlow: Functional Hardware Description with Latency-Insensitive Interface Combinators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2.* 702–717. doi:10.1145/3575693.3575701

[18] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (1978), 666–677. doi:10.1145/359576.359585

[19] Charles A. R. Hoare. 2000. *Communicating Sequential Processes* (reprinted ed.).

[20] Steven Hoover and Ahmed Salman. 2018. Top-Down Transaction-Level Design with TL-Verilog. arXiv:1811.01780 [cs.AR] https://arxiv.org/abs/1811.01780

[21] 2009. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (2009), 1–640. doi:10.1109/IEEESTD.2009.4772740

[22] 2018. *1800-2017 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language.*

[23] 2020. IEEE Standard for Universal Verification Methodology Language Reference Manual. *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)* (2020), 1–458. doi:10.1109/IEEESTD.2020.9195920

[24] 2023. IEEE Standard for Standard SystemC® Language Reference Manual. *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)* (2023), 1–618. doi:10.1109/IEEESTD.2023.10246125

[25] Minseong Jang, Jungin Rhee, Woojin Lee, Shuangshuang Zhao, and Jeehoon Kang. 2024. Modular Hardware Design of Pipelined Circuits with Hazards. *Proc. ACM Program. Lang.* 8, PLDI, Article 148 (2024), 24 pages. doi:10.1145/3656378

[26] Christoph Kern and Mark R. Greenstreet. 1999. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems* 4, 2 (1999), 123–193. doi:10.1145/307988.307989

[27] Caleb Kim, Pai Li, Anshuman Mohan, Andrew Butt, Adrian Sampson, and Rachit Nigam. 2024. Unifying Static and Dynamic Intermediate Languages for Accelerator Generators. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 2242–2267. doi:10.1145/3689790

[28] KULeuven-Micas. 2024. Fix ALU valid-ready signal by rgantonio | Pull Request | #163 KULeuven-MICAS/snax_cluster. https://github.com/KULeuven-MICAS/snax_cluster/pull/163/commits/be67fbfd7ab821b7c7928ccceb1801d3e34fb316 Accessed: 2024-11-12.

[29] lowRISC. 2015. Add an INSTR_VALID_ID signal to completely decouple the pipeline stages, LOWRISC/IBEX@F5D408D. https://github.com/lowRISC/ibex/commit/f5d408d7f4523f4f105cf1fe3029bb28dba12d87 Accessed: 2024-11-12.

[30] lowRISC. 2024. Timing issues in FW_OV "Insert Entropy" feature. https://github.com/lowRISC/opentitan/issues/10983. Accessed: 2024-11-12.

[31] Kingshuk Majumder and Uday Bondhugula. 2023. HIR: An MLIR-based Intermediate Representation for Hardware Accelerator Description. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS)*. 189–201. doi:10.1145/3623278.3624767

[32] MITRE. 2024. CWE-1298: Hardware Logic Contains Race Conditions. https://cwe.mitre.org/data/definitions/1298.html. Accessed: 2024-10-26.

[33] Moszkowski. 1985. A Temporal Logic for Multilevel Reasoning about Hardware. *Computer* 18, 2 (1985), 10–19. doi:10.1109/MC.1985.1662795

[34] Rachit Nigam, Pedro Henrique Azevedo De Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 343–367. doi:10.1145/3591234

[35] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. 69–70. doi:10.1109/MEMCOD.2004.1459818

[36] OpenHW Group. 2024. Issue 145: Clarification of valid-ready handshake dependency. https://github.com/openhwgroup/core-v-xif/issues/145 Accessed: 2024-11-12.

[37] OpenHW Group. 2024. Issue 194: Hansdhake rules additional note. https://github.com/openhwgroup/core-v-xif/issues/194 Accessed: 2024-11-12.

[38] OpenTitan [n. d.]. AES - OpenTitan Documentation. https://opentitan.org/book/hw/ip/aes/index.html Accessed: 2024-11-14.

[39] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science (SFCS)*. 46–57. doi:10.1109/SFCS.1977.32

[40] Pulp-Platform. 2016. Add missing w_valid pulp-platform/core2axi@25eba94. https://github.com/pulp-platform/core2axi/commit/25eba94af4a58249cfa65e1c259ed4b4c5bbfd12 Accessed: 2024-11-12.

[41] Pulp-Platform. 2025. GitHub - pulp-platform/common_cells: Common SystemVerilog components. https://github.com/pulp-platform/common_cells Accessed: 2024-11-14.

[42] Pulp-Platform. 2025. Passthrough Stream FIFO correct specification. https://github.com/pulp-platform/common_cells/issues/264 Accessed: 2024-11-14.

[43] Andy Ray, Benjamin Devlin, Fu Yong Quah, and Rahul Yesantharao. 2023. Hardcaml: An OCaml Hardware Domain-Specific Language for Efficient and Robust Design. arXiv:2312.15035 [cs.PL] https://arxiv.org/abs/2312.15035

[44] Frans Skarman and Oscar Gustafsson. 2022. Spade: An HDL Inspired by Modern Software Languages. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 454–455. doi:10.1109/FPL57034.2022.00075

[45] SpinalHDL 2025. Spinal Hardware Description Language — SpinalHDL documentation. https://spinalhdl.github.io/SpinalDoc-RTD/master/index.html Accessed: 2024-11-14.

[46] Michael Bedford Taylor. 2018. Basejump STL: systemverilog needs a standard template library for hardware design. In *Proceedings of the 55th Annual Design Automation Conference (DAC)*. Article 73, 6 pages. doi:10.1145/3195970.3199848

[47] titan. 2014. Synchronizing Multiplier with Adder to Form Mac. https://electronics.stackexchange.com/questions/102746/synchronizing-multiplier-with-adder-to-form-mac Accessed: 2024-11-14.

[48] Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. 2022. A Survey on Assertion-based Hardware Verification. *Comput. Surveys* 54, 11s (2022), 1–33. doi:10.1145/3510578

[49] XLS [n. d.]. XLS: Accelerated HW Synthesis. https://google.github.io/xls/ Accessed: 2024-11-14.

[50] YosysHQ 2025. YosysHQ/Yosys. https://github.com/YosysHQ/yosys Accessed: 2025-08-21.

[51] Drew Zagieboylo, Charles Sherk, Gookwon Edward Suh, and Andrew C. Myers. 2022. PDL: A High-Level Hardware Design Language for Pipelined Processors. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 719–732. doi:10.1145/3519939.3523455

[52] F. Zaruba and L. Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-Nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (2019), 2629–2640. doi:10.1109/TVLSI.2019.2926114

A Artifact Appendix

A.1 Abstract

The artefacts include the Anvil compiler and the case studies used to evaluate Anvil's practicality and expressiveness. The experiments were conducted by re-implementing open-source hardware designs in Anvil, ensuring that each design matches the original behaviour at the cycle level. All designs type check in Anvil, demonstrating that expressing the same behaviour in Anvil does not introduce any additional clock-cycle latency. We also provide the synthesis reports from a commercial 22 nm ASIC flow as part of the artefacts. These reports show that Anvil incurs minimal overhead compared to the baselines and achieves comparable or better maximum critical frequency.

A.2 Artefact check-list (meta-information)

- **Algorithm:** algorithms for type checking and code generation (implemented in the Anvil compiler)
- **Program:** OpenTitan AES accelerator, CVA6 MMU, AXI routers, Common Cells IP (all included)
- **Compilation:** OCaml 5.2, Verilator v5.036
- **Transformations:** Anvil compiler (included)
- **Run-time environment:** Linux/Unix; Docker/Podman (no root access required)
- **Hardware:** 16 GB RAM, 30 GB external storage
- **Execution:** <15 minutes to run all experiments
- **Metrics:** cycle-accurate logs; ASIC synthesis reports
- **Output:** log files; generated SystemVerilog; sample outputs included
- **Experiments:** cycle-accurate behavioural matching; ASIC synthesis evaluation (fully automated "push-button" simulation)
- **How much disk space required (approximately)?:** 30 GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour to build the container image
- **How much time is needed to complete experiments (approximately)?:** 15 minutes to run experiments + manual inspection time
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT License
- **Workflow automation framework used?:** Docker/Podman with scripts
- **Archived (provide DOI)?:** 10.5281/zenodo.18346123

A.3 Description

A.3.1 How to access. The artefacts are available online at https://github.com/kisp-nus/AnvilHDL-Experiments/tree/artefact-asplos-2026. The repository contains the Anvil implementation of the baselines from open source repositories. It also includes all baseline benchmark implementations from open-source projects as Git submodules.

A.3.2 Hardware dependencies. A multicore CPU with at least 16 GB RAM and 30 GB of free disk space is required to reproduce the artefacts.

A.3.3 Software dependencies. The only prerequisite for evaluating the artefact is a working Docker or Podman installation.

A.4 Installation

To begin, clone the GitHub repository:

```
git clone \
    https://github.com/kisp-nus/AnvilHDL-Experiments.git
cd AnvilHDL-Experiments
```

A.5 Experimental Workflow

The quickest way to run the complete experiment suite is to use the provided Dockerfile, which sets up all dependencies including benchmarks, Verilator, the Anvil compiler, and supporting tools. A push-button script automates the entire process—from building the container image to running all experiments. With Docker or Podman installed and a Unix shell, you are ready to proceed.

```
bash run.sh [-r]
```

The `-r` flag forces a rebuild of the container. Without it, the script reuses an existing build (if present) or builds it, and then executes the experiments sequentially, collecting all logs in the `out` directory.

If you prefer to run the workflow locally (without using a container), you can reproduce the same results using:

```
python3 run_artefact.py
```

Individual experiments can also be run independently. Each experiment directory contains its own README with detailed instructions.

A.6 Evaluation and expected results

After running the experiments, the `out` directory contains cycle-accurate print logs for each design. These logs match across Anvil and the corresponding baselines. An explanation of each testbench is provided in the top-level `README` of the repository. A sample output is provided in the `sample_out` directory for their reference.

Synthesis reports are available in the `synthesis_reports` directory. They can be inspected for area, power, and maximum frequency results from the commercial 22 nm ASIC flow.

A.7 Experiment customization

The Anvil compiler is publicly available at https://github.com/kisp-nus/anvil. It is actively maintained and includes detailed instructions for installation, customization, and usage. Comprehensive documentation is available at https://docs.anvil.kisp-lab.org/. In addition, we provide an online playground for editing and simulating simple Anvil designs at https://anvil.kisp-lab.org/.

```
chan ch {
  right data : (logic @res),
  left res : (logic @#1)
}

chan ch_s {
  right data : (logic @#1)
}

proc grandchild(ep : left ch_s) {
  reg cnt : logic[32];
  loop {
    set cnt := *cnt + 32'b1
  }
  loop {
    let v = if *cnt > 32'h100000 { 1'b1 } else { 1'b0 };
    send ep.data(v) >>
    cycle 1
  }
}

proc child(ep : left ch) {
  reg r : logic;
  chan ep_sl -- ep_sr : ch_s;
  spawn grandchild(ep_sl);
  loop {
    set r := ~*r >>
    let d = recv ep_sr.data >>
    send ep.data (*r & d) >>
    let _ = recv ep.res
  }
}

proc Top() {
  chan ep_sl -- ep_sr : ch;
  spawn child(ep_sl);
  loop {
    let d = recv ep_sr.data >>
    cycle 1 >>
    dprint "Value: %b" (d) >>
    cycle 1 >>
    dprint "Value should be the same %b" (d) >>
    cycle 1 >>
    send ep_sr.res (1'b1) >>
    cycle 1
  }
}
```

Listing 1. Example Anvil code.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

```
module grandchild(/* omitted */);
    logic [31:0] cnt;
    logic data_q, data_d;

    assign data_o = data_q;
    assign data_valid_o = 1'b1;
    always_comb begin
        data_d = data_q;
        if (data_ack_i) begin
            data_d = cnt > 32'h100000 ? 1'b1 : 1'b0;
        end
    end

    initial begin
        cnt <= '0;
        data_q <= 1'b0;
    end
    always_ff @(posedge clk_i) begin
        cnt <= cnt + 32'b1;
        data_q <= data_d;
    end
endmodule

module child(/* omitted */);
    /* omitted */
endmodule

module example(input logic clk_i);
    /* omitted */

    enum logic[1:0] { /* omitted */ } state_q, state_d;

    assign data_ack = state_q == RECV;
    assign res_valid = state_q == SEND;
    always_comb begin
        state_d = state_q;
        unique case (state_q)
            /* omitted */
        endcase
    end

    initial state_q <= RECV;
    always_ff @(posedge clk_i) begin
        if (state_q == ST0 || state_q == ST1) begin
            assert(data == $past(data));
        end
        state_q <= state_d;
    end
endmodule
```

Listing 2. Example SystemVerilog code with assertions.

B Comparison with Verification

Consider the Anvil code in Listing 1. The module `Top` receives data and sends back a response to the `child` module. The data received is expected to be usable and keep

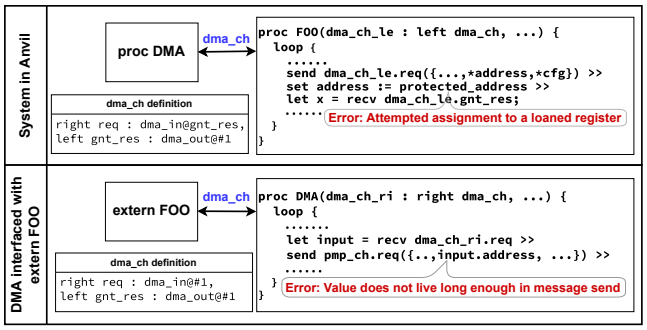

Figure 9. Anvil can assist in preventing bugs.

unchanged in between. The `child` module in turn communicates with grandchild to obtain the data. Anvil's type system rejects this code because the data `child` obtains from `grandchild`, namely `d`, lives only for one cycle, but a value that depends on it (`*r & d`) is sent to `Top` which requires it to stay alive until the response. When fed with this code as input, the Anvil compiler detects this type error and prints the following error message:

```
Value not live long enough in message send!
Top.anvil:29:4:
     29|     send ep.data (*r & d) >>
       |     ^^^^^^^^^^^^^^^^^^^^
```

Now let us consider the same design expressed in SystemVerilog with an assertion for the same property, provided in Listing 2. Comparing the two versions leads us to the following observations:

- The assertion in the SystemVerilog code is tied to the implementation. For example, it already requires the designer to manually and explicitly identify the states where the data is used and needs to remain unchanged. Anvil does not require such manual effort.
- Anvil can check the property individually for each module due to the compositional contracts specified in channel definitions. In the example, Anvil can report the violation by just looking at child module alone. Checking the property in the SystemVerilog code requires reasoning across all three modules.
- Problems may arise if we attempt to apply formal verification techniques to verify the specified property in the SystemVerilog code. For example, bounded model checking on the SystemVerilog code using Yosys SMT-BMC and z3 fails to detect the violation even with large depth limits. This is due to the large number of concrete states. In contrast, the Anvil code provides abstractions that capture more of the hardware designer's intent, allowing the property to be checked more easily.

C Safety Analysis on Real-World Errors

We were motivated to design Anvil by our own frustrating experience implementing an experimental CPU architecture. The frequent timing hazard we encountered during development required significant debugging effort. We demonstrate how Anvil can help designers address the following challenges with minimal effort:

1. Enforcing concrete timing contracts
2. Challenges in implementing timing contracts

Case 1: Enforcing Concrete Timing Contracts. The vulnerability class highlighted in CWE-1298 [32] illustrates a hardware bug from HACK@DAC'21. This bug arose from a missing timing contract in the DMA module of the OpenPiton SoC. The module was intended to verify access to protected memory using specific address and configuration signals. However, it assumed these inputs would remain stable during processing without any mechanism to enforce this assumption. This created a timing vulnerability across module interactions.

If designed in Anvil, the DMA channel definition would explicitly require that input signals remain stable until the request is completed, as shown in Figure 9. Anvil would enforce this stability requirement, ensuring that only compatible modules interact without introducing timing risks. When the DMA module interfaces with non-Anvil modules, Anvil imposes a one-clock-cycle lifetime on external signals. If the DMA implementation does not follow the contract, Anvil triggers an error: "Value does not live long enough…," implying the need to register the signal immediately.

Similarly, designers using custom test benches with open-source hardware often struggle to follow strict timing contracts. This is particularly challenging when there is no mechanism to enforce timing contracts. For instance, in this GitHub issue [1], the designer observed unexpected behaviour during simulation while integrating a Verilog-based Ethernet interface into their module. This Ethernet module required a complex timing contract to be enforced on the interfacing module for proper operation. However, without a language that enforces this contract, the designer struggled to explicitly meet these timing requirements and manage synchronization.

Case 2: Challenges in Implementing Timing Contracts. Designers often face challenges in implementing synchronization primitives and dynamic timing contracts, even when they intend to define them clearly. This difficulty is evident in various open-source project commit histories and issue trackers. For example, in Table 2, we highlight a few instances from GitHub that showcase how designers have struggled with these aspects. Our analysis demonstrates that Anvil could have prevented these issues or helped catch the bugs before compilation.

Even when contracts are explicitly defined, the instructions for compliance can be ambiguous. A case in point is

Table 2. Summary of Issues in some open source repositories

Repository	Issue Analysis	How can Anvil help?
OpenTitan (Issue [30])	In OpenTitan's entropy source module, firmware (FW) is supposed to insert verified entropy data into the RNG pipeline. However, a timing hazard prevented reliable data writing and control over the SHA operation. Solution Proposed in discussion: Add signals for FW to control the entropy source state machine and a ready signal to safely write data into the pipeline.	If implemented in Anvil, FW would inherently control the state machine when asserting data without explicit implementa -tion ensuring synchronization is built-in.
Coyote (Issue [15])	The completion queue has a 2-cycle valid signal burst instead of one cycle. The issue is still open. This happens when a write request is issued on the sq_wr bus, and the cq_wr is observed for completion. The valid signal is high for 2 cycles instead of one. Core Issue: The timing contract was not properly implemented, though the designer defined it. The timing control was deeply embedded within interconnected state machines, making the bug difficult to detect even with a thorough inspection.	Anvil implements the FSM for timing contracts implicitly, providing synchroniza -tion primitives to control the state and ensure an error-free FSM implementation.
ibex (Commit [29])	Commit Message: "Add an instr_valid_id signal to completely decouple the pipeline stages, hopefully, it fixes the exception controller" Commit Summary: Despite the pipeline being statically scheduled, the valid signal was added later to enforce the timing contract only after unexpected behaviour was observed.	In Anvil, even for statically scheduled pipelines, stage-to-stage handshakes are enforced implicitly, ensuring timing contracts are upheld even if the schedule isn't strictly adhered to.

snax-cluster (Commit [28])	Commit changes
	`assign a_ready_o = acc_ready_i && c_ready_i && (a_valid_i && b_valid_i);`
	`assign b_ready_o = acc_ready_i && c_ready_i && (a_valid_i && b_valid_i);`
	
	Commit Summary: Fixes the implementation of the timing contract on the ALU interface by adding the missing valid signal in the handshake.
		Anvil implicitly handles handshake impl -ementation for interfacing signals, ensuring the enforcement of timing contracts.

core2axi (Commit [40])	Commit changes: `w_valid_o = 1'b1;` Commit Summary: Ensure compliance with the timing contract by asserting the missing valid signal when sending a new write request on the bus.	In Anvil, the assertion of valid signals and synchronization is handled implicitly whenever a message is sent

the documentation for CV-X-IF, where one issue [37] reveals the complications involved in adhering to the timing contract. Another issue [36] illustrates that the complexity of a static schedule necessitated additional notes to clarify the implementation guidelines for the interfacing module.

In contrast, Anvil simplifies the implementation of synchronization and finite state machines (FSM) that handle timing contracts. Designers only need to define the contract within the corresponding channel, which can utilize dynamic message-passing events. The synchronization primitives (handshakes) are implemented implicitly and efficiently, ensuring no clock cycle overhead. Additionally, the wait construct allows designers to express the dynamic times required to process a state. In ambiguous process descriptions, Anvil flags the description to make necessary changes to guarantee runtime safety statically.

$$\begin{aligned}
\text{message definitions} \quad & M ::= \{\pi.m : p, \cdots\} \\
\text{message set} \quad & \Sigma ::= \{\pi.m, \cdots\} \\
\text{composition} \quad & \kappa ::= t \mid \kappa \parallel_\Sigma \kappa \\
\text{program} \quad & \mathcal{P} ::= (\texttt{loop}\{t\}, M) \mid \mathcal{P} \parallel_\Sigma \mathcal{P}
\end{aligned}$$

Figure 10. Anvil abstract syntax

D Formalization Details

D.1 Abstract Syntax

For convenience of formal reasoning, we also define an abstract syntax of Anvil programs, shown in Figure 10, allowing us to discuss parallel composition in a style similar to communicating sequential processes (CSP) [18]. The $\parallel_\Sigma$ notation represents parallel composition with the two sides communicating through messages specified in the set Σ. M maps each message to the associated duration requirement.

D.2 Semantics

Execution log. An execution log is simply a sequence $\mathcal{L} = \langle \alpha_0, \cdots, \alpha_k \rangle$, where α_i is represents the set of operations performed during cycle i. Operations can be one of the following — 1. **ValCreate** representing the creation of a new value that depends on a set of registers and existing values, 2. **ValUse**, representing the use of a value, 3. **RegMut**, denoting mutation of a register, 4. **ValSend**, for sending of a value through a message, and 5. **ValRecv**, denoting the receipt of a value through a message. Following this, we define the set of execution logs corresponding to a term, compositions, and finally programs. To capture the non-determinism of message passing and branching in an execution log of a term, we delay each send and receive operation by any non-negative number of cycles and allow each branching term to take either branch. Execution logs of compositions are obtained by combining two execution logs, with the requirement that any send and receive operations for messages in Σ must match and align in pairs, and each pair must use the same value identifier. In the combined execution log, the matching send and receive operations are eliminated. This reflects that they have now become internal details, no longer affecting the semantics of the composition. For programs, we take into consideration the looping semantics of each looping thread. We achieve this by mapping a program to a set of compositions, where each composition is obtained by appending t in each looping thread $\texttt{loop}\{t\}$ arbitrarily many times. Any execution log of any such composition is an execution log of the program. The semantics of those constructs is then defined by their sets of execution logs, which captures all their possible behaviours.

Definition D.1 (Execution log)**.** An execution log consists of a sequence of sets $\mathcal{L} = \langle \alpha_0, \alpha_1, \cdots, \alpha_k \rangle$. The finite set α_i contains the actions in the i-th cycle, each of the following form:

- ValCreate$(v, \{r_1, r_2, \cdots, r_m\}, \{v_1, v_2, \cdots, v_n\})$ (creating a value with name v that depends on registers $r_1, r_2, \cdots, r_m$ and values $v_1, v_2, \cdots, v_n$)
- ValUse(v) (using the value identified by v)
- RegMut(r) (mutating the register identified by r)
- ValSend$(\pi.m, v, p)$ (send a value with name v through message $\pi.m$ with duration p)
- ValRecv$(\pi.m, v, p)$ (receive a value with name v through message $\pi.m$ with duration p)

Definition D.2 (Local execution log)**.** A log $\mathcal{L}$ is a local execution log of a term t if $\Gamma; I, M \vdash t \rightsquigarrow \mathcal{L} \lhd v$, which is defined by the following inference rules.

$$\frac{}{\Gamma; \{v\}, M \vdash \texttt{cycle}\ \#k \rightsquigarrow (\emptyset^{k+1} \circ \langle \{\text{ValCreate}(v, \emptyset, \emptyset)\} \rangle) \lhd v} \quad \text{(E-Cycle)}$$

$$\frac{}{\Gamma; \{v\}, M \vdash n \rightsquigarrow \langle \{\text{ValCreate}(v, \emptyset, \emptyset)\} \rangle \lhd v} \quad \text{(E-Literal)}$$

$$\frac{\begin{array}{c}\Gamma; I_1, M \vdash t_1 \rightsquigarrow \mathcal{L}_1 \lhd v_1 \quad \Gamma; I_2, M \vdash t_2 \rightsquigarrow \mathcal{L}_2 \lhd v_2 \\ I_1 \cap I_2 = \emptyset\end{array}}{\text{shift}(\Gamma, |\mathcal{L}_1| - 1); (I_1 \cup I_2), M \vdash t_1\ \texttt{=>}\ t_2 \rightsquigarrow (\mathcal{L}_1 \circ \mathcal{L}_2) \lhd v_2} \quad \text{(E-Wait)}$$

$$\frac{\Gamma; I_1, M \vdash t_1 \rightsquigarrow \mathcal{L}_1 \lhd v_1 \quad \Gamma, x : (|\mathcal{L}_1| - 1, v_1); I_2, M \vdash t_2 \rightsquigarrow \mathcal{L}_2 \lhd v_2}{\Gamma; (I_1 \cup I_2), M \vdash \texttt{let } x = t_1 \texttt{ in } t_2 \rightsquigarrow (\mathcal{L}_1 \uplus \mathcal{L}_2) \lhd v_2 \quad I_1 \cap I_2 = \emptyset} \text{ (E-Let)}$$

$$\frac{\Gamma(x) = (k, v)}{\Gamma; \emptyset, M \vdash x \rightsquigarrow \emptyset^{k+1} \lhd v} \text{ (E-Ref)}$$

$$\frac{\Gamma; I, M \vdash t \rightsquigarrow \mathcal{L} \lhd v \quad v' \notin I}{\Gamma; I \cup \{v'\}, M \vdash r := t \rightsquigarrow \mathcal{L} \uplus \langle \{\text{ValUse}(v), \text{RegMut}(r)\}, \{\text{ValCreate}(v', \emptyset, \emptyset)\}\rangle \lhd v'} \text{ (E-RegAssign)}$$

$$\frac{\Gamma; I, M \vdash t \rightsquigarrow \mathcal{L} \lhd v \quad v' \notin I \quad k \in \mathbb{N}}{\Gamma; (I \cup \{v'\}), M \vdash \texttt{send } \pi.m(t) \rightsquigarrow \emptyset^{k+1} \circ \langle \{\text{ValSend}(\pi.m, v, M(\pi.m)), \text{ValCreate}(v', \emptyset, \emptyset)\}\rangle \lhd v'} \text{ (E-Send)}$$

$$\frac{k \in \mathbb{N}, u \neq v}{\Gamma; (\{v, u\}), M \vdash \texttt{recv } \pi.m \rightsquigarrow \emptyset^k \circ \langle \{\text{ValRecv}(\pi.m, v, M(\pi.m)), \text{ValCreate}(u, \emptyset, \{v\})\}\rangle \lhd u} \text{ (E-Recv)}$$

$$\frac{\begin{array}{c}\Gamma; I_1, M \vdash t_1 \rightsquigarrow \mathcal{L}_1 \lhd v_1 \\ \Gamma; I_2, M \vdash t_2 \rightsquigarrow \mathcal{L}_2 \lhd v_2 \\ \Gamma; I_3, M \vdash t_3 \rightsquigarrow \mathcal{L}_3 \lhd v_3 \\ I_1 \cap (I_2 \cup I_3) = \emptyset \quad I_2 \cap I_3 = \emptyset\end{array}}{\Gamma; (I_1 \cup I_2 \cup I_3), M \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \rightsquigarrow \mathcal{L}_1 \uplus \mathcal{L}_2 \uplus \langle \{\text{ValUse}(v_1)\}\rangle \lhd v_2} \text{ (E-IfThen)}$$

$$\frac{\begin{array}{c}\Gamma; I_1, M \vdash t_1 \rightsquigarrow \mathcal{L}_1 \lhd v_1 \\ \Gamma; I_2, M \vdash t_2 \rightsquigarrow \mathcal{L}_2 \lhd v_2 \\ \Gamma; I_3, M \vdash t_3 \rightsquigarrow \mathcal{L}_3 \lhd v_3 \\ I_1 \cap (I_2 \cup I_3) = \emptyset \quad I_2 \cap I_3 = \emptyset\end{array}}{\Gamma; (I_1 \cup I_2 \cup I_3), M \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \rightsquigarrow \mathcal{L}_1 \uplus \mathcal{L}_3 \uplus \langle \{\text{ValUse}(v_1)\}\rangle \lhd v_3} \text{ (E-IfElse)}$$

$$\frac{}{\emptyset; \{v\}, M \vdash *r \rightsquigarrow \langle \{\text{ValCreate}(v, \{r\}, \emptyset)\}\rangle \lhd v} \text{ (E-RegEval)}$$

$$\frac{}{\emptyset; \{v\}, M \vdash \texttt{ready}(\pi.m) \rightsquigarrow \langle \{\text{ValCreate}(v, \emptyset, \emptyset)\}\rangle \lhd v} \text{ (E-Ready)}$$

Where $\langle \alpha_0, \alpha_1, \cdots, \alpha_k \rangle \circ \langle \beta_0, \beta_1, \cdots, \beta_l \rangle = \langle \alpha_0, \alpha_1, \cdots, (\alpha_k \cup \beta_0), \beta_1, \cdots, \beta_l \rangle$.

The merge operator $\uplus$ is defined as (without loss of generality, assuming $k \leq l$): $\langle \alpha_0, \alpha_1, \cdots, \alpha_k \rangle \uplus \langle \beta_0, \beta_1, \cdots, \beta_l \rangle = \langle \alpha_0 \cup \beta_0, \alpha_1 \cup \beta_1, \cdots, \alpha_k \cup \beta_k, \beta_{k+1}, \cdots, \beta_l \rangle$.

$\alpha^k = \langle \alpha_0, \cdots, \alpha_{k-1} \rangle$ where for all $i = 0, 1, \cdots, k-1$, $\alpha_i = \alpha$.

The function $\text{shift}(\Gamma, k)$ shifts all delays in Γ by k cycles. Formally,

$$\text{shift}(\emptyset, k) = \emptyset$$
$$\text{shift}((\Gamma, x : (k', v)), k) = \text{shift}(\Gamma, k), x : (\max(0, k' - k), v)$$

Definition D.3 (Compositional execution log)**.** $\mathcal{L}$ is an execution log of a κ if:

- $\kappa = t$ and $\mathcal{L}$ is a prefix of an execution log of t
- $\kappa = \kappa_1 \parallel_\Sigma \kappa_2$, $\mathcal{L}_1, \mathcal{L}_2$ are execution logs of κ_1 and κ_2 respectively, and let $\mathcal{L}_1 = \langle \alpha_0, \cdots, \alpha_m \rangle$, $\mathcal{L}_2 = \langle \beta_0, \cdots, \beta_m \rangle$, the following holds:
  - For all $\pi.m \in \Sigma$, $0 \leq i \leq m$, $\text{ValSend}(\pi.m, v, p) \in \alpha_i$ if and only if $\text{ValRecv}(\pi.m, v, p) \in \beta_i$, and $\text{ValRecv}(\pi.m, v, p) \in \alpha_i$ if and only if $\text{ValSend}(\pi.m, v, p) \in \beta_i$.
  - $\mathcal{L} = \langle \gamma_0, \cdots, \gamma_m \rangle, \gamma_i = \alpha_i \cup \beta_i - \{\text{ValSend}(\pi.m, v, p) \mid \pi.m \in \Sigma\} - \{\text{ValRecv}(\pi.m, v, p) \mid \pi.m \in \Sigma\}$.

Definition D.4 (Concretization). A composition κ is a concretization of program $\mathcal{P}$, written $\mathcal{P} \rightsquigarrow \kappa$, by the following inference rules:

$$\frac{}{(\texttt{loop}\{t\}, M) \rightsquigarrow t} \quad \text{(C-Base)}$$

$$\frac{(\texttt{loop}\{t\}, M) \rightsquigarrow t'}{(\texttt{loop}\{t\}, M) \rightsquigarrow t' \ \texttt{=>}\ t} \quad \text{(C-Extend)}$$

$$\frac{\mathcal{P}_1 \rightsquigarrow \kappa_1 \quad \mathcal{P}_2 \rightsquigarrow \kappa_2}{\mathcal{P}_1 \parallel_\Sigma \mathcal{P}_2 \rightsquigarrow \kappa_1 \parallel_\Sigma \kappa_2} \quad \text{(C-Compose)}$$

Definition D.5 (Program execution log). $\mathcal{L}$ is an execution log of program $\mathcal{P}$ if there exists composition κ such that $\mathcal{P} \rightsquigarrow \kappa$ and $\mathcal{L}$ is an execution log of κ.

D.3 Type System

Event graph. The type system of Anvil is based on the *event graph*. An event graph, denoted $G = (V, E)$, is a directed acyclic graph that describes the time ordering among events in an Anvil process. Each node (i.e., event) is labelled to indicate how its corresponding starting time relates to those of its direct predecessors. Types in Anvil reference the event graph as part of the typing environment to convey timing constraints. We choose this strategy because the timing constraints associated with a term are not always local. Take the example of `send ch.m1 (x) => recv ch.m2`, where `ch.m1` specifies a duration of `ch.m2`. It is necessary to be aware of the first `ch.m2` event that occurs after `ch.m1`. This event does not appear in the expression `send ch.m1 (x)` itself, but rather in the surrounding context in which `send ch.m1 (x)` appears, to ensure that `x` lives long enough.

We choose the event graph as it is a simple structure that captures all the necessary information to reason about such timing constraints. As a shorthand, we use the notation $e_1 \to e_2 \in G$ to say that G contains an edge from event e_1 to event e_2. We use $G(e_2)$ to denote $\langle \omega, \{e_1 \mid e_1 \to e_2 \in G\}\rangle$, which consists of the operation label ω of e_2 as well as the set of all its direct predecessors.

Types. Intuitively, a type encodes a lifetime by referencing the event graph and is a pair:

$$T ::= (e_l, S_d),$$

where e_l is an event graph node that encodes the start time, and S_d is a set of event patterns $e_d \triangleright p$, the earliest match of which defines the end time. An empty S_d indicates that the lifetime is eternal. Each time pointer specifier is a pair of event identifier e_d and duration p, which implies the first time p is matched (the specified number of cycles have elapsed or a specified message is sent or received) after e_d is reached.

Typing Rules. A typing judgment is of the form

$$\Gamma; G, R, M, C, e_c \vdash t : T.$$

The typing environment consists of Γ which maps each let-binding to its type, the event graph G introduced above, R which maps a register to its loan time, M which maps a message specifier (an endpoint and a message identifier, of the form $\pi.m$) to the duration that specifies its lifetime requirement, C which is a set of identifiers associated with all branch conditions that have appeared, and e_c which references a node in G as an abstract specifier of the time at which t is to be evaluated.

The typing rules use the $\leq_G$ and $<_G$ relations to apply timing constraints. Their complete and formal definitions are available in Section D. Intuitively, $a \leq_G b$ if the time specified by a is always no later than that by b in the event graph G, and $a <_G b$ if the time specified by a is always strictly before that by b in G. Here a and b can be nodes or timing patterns in G. In our implementation, we use sound approximations of $\leq_G$ and $<_G$.

$$\frac{\Gamma; G, R, M, C, e_c \vdash t : T}{\Gamma, x : T'; G, R, M, C, e_c \vdash t : T} \quad \text{(T-Weaken)}$$

$$\frac{G(e_l) = \langle \#k, \{e_c\}\rangle}{\emptyset; G, R, M, \emptyset, e_c \vdash \texttt{cycle}\ k : (e_l, \emptyset)} \quad \text{(T-Cycle)}$$

$$\frac{\Gamma; G, R, M, C_1, e_c \vdash t_1 : (e_l, S_d) \quad \Gamma; G, R, M, C_2, e_l \vdash t_2 : T_2 \quad C_1 \cap C_2 = \emptyset}{\Gamma; G, R, M, C_1 \cup C_2, e_c \vdash t_1 \ \texttt{=>}\ t_2 : T_2} \quad \text{(T-Wait)}$$

$$\frac{M(\pi.m) = p \quad G(e_l) = \langle \pi.m, \{e_c\}\rangle}{\emptyset; G, R, M, \emptyset, e_c \vdash \texttt{recv}\ \ m : (e_l, \{e_l \triangleright p\})} \quad \text{(T-Recv)}$$

$$\frac{x : (e_l, S_d) \in \Gamma \quad G(e'_l) = \langle \#0, \{e_c, e_l\}\rangle}{\Gamma; G, R, M, \emptyset, e_c \vdash x : (e'_l, S_d)} \quad \text{(T-Ref)}$$

$$\frac{\begin{array}{c}\Gamma; G, R, M, C, e_c \vdash t : (e_l, S_d) \\ G(e'_l) = \langle \pi.m, \{e_c\}\rangle \\ e_l \leq_G e_c \quad e'_l \triangleright M(\pi.m) \leq_G S_d\end{array}}{\Gamma; G, R, M, C, e_c \vdash \texttt{send}\ \ \pi.m(t) : (e'_l, \emptyset)} \quad \text{(T-Send)}$$

$$\frac{\begin{array}{c}\Gamma; G, R, M, C_1, e_c \vdash t_1 : (e_1, S_1) \\ \Gamma; G, R, M, C_2, e_c \vdash t_2 : (e_2, S_2) \\ G(e'_l) = \langle \#0, \{e_1, e_2\}\rangle \quad C_1 \cap C_2 = \emptyset\end{array}}{\Gamma; G, R, M, C_1 \cup C_2, e_c \vdash t_1 \star t_2 : (e'_l, S_1 \cup S_2)} \quad \text{(T-BinOp)}$$

$$\frac{\begin{array}{c}\Gamma; G, R, M, C, e_c \vdash t : (e_l, S_d) \\ \forall (e, S) \in R(r) : e_c <_G e \vee S \leq_G e_c \\ e_l \leq_G e_c \quad e_c \triangleright \#1 \leq_G S_d \quad G(e'_l) = \langle \#1, \{e_c\}\rangle\end{array}}{\Gamma; G, R, M, C, e_c \vdash r := t : (e'_l, \emptyset)} \quad \text{(T-RegAssign)}$$

$$\frac{\exists (e, S) \in R(r) : e \leq_G e_c \wedge e_c \leq_G S_d \wedge S_d \leq_G S}{\emptyset; G, R, M, \emptyset, e_c \vdash *r : (e_c, S_d)} \quad \text{(T-RegEval)}$$

$$\frac{\begin{array}{c}\Gamma; G, R, M, C_1, e_c \vdash t_1 : (e_1, S_1) \\ \Gamma; G, R, M, C_2, e'_c \vdash t_2 : (e_2, S_2) \\ \Gamma; G, R, M, C_3, e''_c \vdash t_3 : (e_3, S_3) \\ e_1 \leq_G e_c \wedge e_c \leq_G S_1 \\ c \notin C_1 \cup C_2 \cup C_3 \quad C_1 \cap (C_2 \cup C_3) = \quad C_2 \cap C_3 = \emptyset \\ G(e'_c) = G(e''_c) = \langle \&c, \{e_c\}\rangle \quad e'_c \neq e''_c \\ G(e'_l) = \langle \oplus, \{e_2, e_3\}\rangle\end{array}}{\Gamma; G, R, M, C_1 \cup C_2 \cup C_3 \cup \{c\}, e_c \vdash \texttt{if}\ \ t_1\ \ \texttt{then}\ \ t_2\ \ \texttt{else}\ \ t_3 : (e'_l, S_1 \cup S_2 \cup S_3)} \quad \text{(T-Cond)}$$

$$\frac{\begin{array}{c}\Gamma; G, R, M, C_1, e_c \vdash t_1 : (e_1, S_1) \\ \Gamma; G, R, M, C_2, e_c \vdash t_2 : (e_2, S_2) \\ G(e'_l) = \langle \#0, \{e_1, e_2\}\rangle \quad C_1 \cap C_2 = \emptyset\end{array}}{\Gamma; G, R, M, C_1 \cup C_2, e_c \vdash t_1; t_2 : (e'_l, S_2)} \quad \text{(T-Join)}$$

$$\frac{(\pi.m, p) \in M}{\emptyset; G, R, M, \emptyset, e_c \vdash \texttt{ready}(\pi.m) : (e_c, \{e_c \triangleright \#1\})} \quad \text{(T-Ready)}$$

Well-typedness. We define well-typed terms, processes, and programs based on the above.

Definition D.6 (Well-typed Anvil term)**.** An Anvil term t is well-typed under the context M if there exist G, R, e_0, C, and T such that $G(e_0) = \langle 0, \emptyset\rangle$ and $\emptyset; G, R, M, C, e_0 \vdash t : T$.

Definition D.7 (Well-typed Anvil process)**.** Under the context M, we say a process loop $\texttt{loop}\{t\}$ is well-typed if the term $t \Rightarrow t$ is well-typed under M.

Definition D.8 (Well-typed Anvil program)**.** A program $\mathcal{P}$ is well-typed if

- $\mathcal{P} = (\texttt{loop}\{t\}, M)$ and $\texttt{loop}\{t\}$ is well-typed under M.
- $\mathcal{P} = \mathcal{P}_1 \parallel_\Sigma \mathcal{P}_2$, and $\Sigma = M_{\mathcal{P}_1} \cap M_{\mathcal{P}_2}$, where $M_{\mathcal{P}_i}$ is the union of all Ms that appear in $\mathcal{P}_i$.

D.3.1 Auxiliary Definitions.

We define $\leq_G$ and $<_G$ that appear in the typing rules.

Definition D.9 (Timestamp)**.** A function $\tau_G : V \to \mathbb{N}$ is a timestamp function of event graph $G = (V, E)$ if for all $e \in V$:

- If $G(e) = \langle 0, S\rangle$, then $\tau_G(e) = 0$.
- If $G(e) = \langle \#k, S\rangle$, then $\tau_G(e) = \max_{e' \in S}(\tau_G(e') + k)$.
- If $G(e) = \langle \pi.m, S\rangle$, then $\tau_G(e) \geq \max_{e' \in S} \tau_G(e')$
- If $G(e) = \langle \&c, S\rangle \wedge \tau_G(e) = \max_{e' \in S} \tau_G(e')$, then $\forall e' \in V : (e' \neq e \wedge G(e) = \langle \&c, S\rangle) \to \tau_G(e') = \infty$
- If $G(e) = \langle \oplus, S\rangle$, then $\tau_G(e) = \min_{e' \in S} \tau_G(e')$.

It is obvious that for any event graph G, at least one timestamp function exists. We now extend this definition of timestamps to event patterns.

Definition D.10 (Event pattern timestamp)**.** Let G be an event graph and τ_G be a timestamp function of G. We define $e \triangleright p$:

- $\tau_G(e \triangleright \#k) = \tau_G(e) + k$
- $\tau_G(e \triangleright \pi.m) = \min_{G(e')=\langle \pi.m, S\rangle, \tau_G(e) < \tau_G(e')} \tau_G(e')$ (or ∞ if no such e' can be found).

Definition D.11 ($\leq_G$ and $<_G$)**.** Let G be an event graph. We say $e_1 \triangleright p_1 \leq_G e_2 \triangleright p_2$ if for all timestamp functions τ_G of G, it holds that $\tau_G(e_1 \triangleright p_1) \leq \tau_G(e_2 \triangleright p_2)$. Similarly, we say $e_1 \triangleright p_1 <_G e_2 \triangleright p_2$ if for all timestamp functions τ_G of G, it holds that $\tau_G(e_1 \triangleright p_1) < \tau_G(e_2 \triangleright p_2)$.

It is easy to prove the following two lemmas.

Lemma D.12. If $(e_1 \to e_2) \in G$, then $e_1 \leq_G e_2$.

Lemma D.13. $S \cup S' \leq_G S$.

D.4 Safety

Definition D.14 (Register dependency set)**.** We define that the value v has the register dependency set D in the execution log $\mathcal{L}$, written $\mathcal{L} \vdash v \downarrow D$, by the following inference rules:

$$\frac{}{\langle\rangle \vdash v \downarrow \bot} \quad \text{(R-Base)}$$

$$\frac{\mathcal{L} \vdash v \downarrow D}{\mathcal{L} \cdot \langle\emptyset\rangle \vdash v \downarrow D} \quad \text{(R-Empty)}$$

$$\frac{\mathcal{L} \cdot \langle\alpha_i\rangle \vdash v \downarrow D \qquad o \notin \{\text{ValCreate}(v, S_r, S_v) \mid S_r \in 2^{\text{RegId}}, S_v \in 2^{\text{ValId}}\}}{\mathcal{L} \cdot \langle\alpha_i \cup \{o\}\rangle \vdash v \downarrow D} \quad \text{(R-NonCreate)}$$

$$\frac{\begin{array}{c}\mathcal{L} \cdot \langle\alpha_i\rangle \vdash v_1 \downarrow D_1 \quad D_1 \neq \bot \\ \vdots \\ \mathcal{L} \cdot \langle\alpha_i\rangle \vdash v_k \downarrow D_k \quad D_k \neq \bot\end{array}}{\mathcal{L} \cdot \langle\alpha_i \cup \{\text{ValCreate}(v, S_r, \{v_1, \cdots, v_k\})\}\rangle \vdash v \downarrow S_r \cup D_1 \cup \cdots \cup D_k} \quad \text{(R-Create)}$$

Note: $\cdot$ is the normal concatenation operator.
Other auxiliary definitions, assuming $\mathcal{L} = \langle\alpha_0, \alpha_1, \cdots, \alpha_k\rangle$,

- $\text{UseSet}(\mathcal{L}, v) = \{i \mid \text{ValUse}(v) \in \alpha_i \vee \text{ValCreate}(v, S_r, S_v) \in \alpha_i \vee \text{ValRecv}(\pi.m, v, p) \in \alpha_i \vee \text{ValSend}(\pi.m, v, p) \in \alpha_i\}$
- $\text{MutSet}(\mathcal{L}, D) = \{i \mid r \in D \wedge \text{RegMut}(r) \in \alpha_i\}$
- $\text{LtRecv}(\mathcal{L}, v) = \bigcap_{u \in \text{DepSet}(\mathcal{L}, v), \text{ValRecv}(\pi.m, u, p) \in \alpha_i} \text{LtFun}(\mathcal{L}, i, p)$
- $\text{LtSend}(\mathcal{L}, v) = \bigcup_{u \in \text{DeriveSet}(\mathcal{L}, v), \text{ValSend}(\pi.m, u, p) \in \alpha_i} \text{LtFun}(\mathcal{L}, i, p)$
- $\text{LtFun}(\mathcal{L}, i, \pi.m) = [i, w)$ where w is the lowest $j \geq i$, such that $\text{ValSend}(\pi.m, v, p) \in \alpha_j$ or $\text{ValRecv}(\pi.m, v, p) \in \alpha_j$
- $\text{LtFun}(\mathcal{L}, i, \#l) = [i, i + l)$

Defining safety. We first define when an execution log should be deemed safe. This notion, then, can be naturally lifted to define the safety of a term, composition of terms and of an entire Anvil program.

Definition D.15 (Safety of execution log)**.** An execution log $\mathcal{L}$ is safe if for every value v, there exists an interval $[a, b]$ such that $\text{UseSet}(\mathcal{L}, v) \cup \text{LtSend}(\mathcal{L}, v) \subseteq [a, b] \subseteq \text{LtRecv}(\mathcal{L}, v)$, and for D such that $\mathcal{L} \vdash v \downarrow D$, $\text{MutSet}(\mathcal{L}, D) \cap [a, b) = \emptyset$.

$\text{UseSet}(\mathcal{L}, v)$ includes all time points (cycle numbers) at which the value v is used, $\text{LtSend}(\mathcal{L}, v)$ captures when v needs to be live as required by all send operations that involve v or other values that depend on it, $\text{LtRecv}(\mathcal{L}, v)$ captures when v is guaranteed to be live through received messages from the environment, $\mathcal{L} \vdash v \downarrow D$ states that v directly or indirectly depends on the set of registers D, and $\text{MutSet}(\mathcal{L}, D)$ captures when any register in D is mutated. Intuitively, the safety definition above states that all uses of a value v and the lifetime promised to the environment should fall within a continuous time window. During this time window, values received from the environment through *receive* are live, and no register that v depends on is mutated.

Since the set of all execution logs of a term, composition, or program captures all its possible run-time timing behaviours, we define safety for those constructs as follows.

Definition D.16 (Term, composition, and program safety)**.** A term, composition, or program is safe if all its execution logs are safe.

Safety guarantees. We present a sketch of the proof of the safety guarantees of Anvil by providing the key lemmas. The detailed proofs of the lemmas are available in Section E of the Appendix.

First, we show that well-typedness implies safety for terms.

Lemma D.17 (Safety of terms)**.** A well-typed term is safe.

Then, by matching the $\text{LtSend}(\mathcal{L}, v)$ and $\text{LtRecv}(\mathcal{L}', v)$ when obtaining the execution logs of well-typed compositions, we prove that well-typedness implies safety also for compositions.

Lemma D.18 (Safety of compositions)**.** A well-typed composition is safe.

Then, to account for the looping semantics in programs, we show that well-typedness for an Anvil process `loop{`t`}` is sufficient to guarantee that any number of ts joined together by wait ($\Rightarrow$) is also well-typed.

Lemma D.19 (Two iterations are sufficient)**.** Let t be an Anvil term and $t_k, k = 1, 2, \cdots$ be inductively defined as $t_1 = t$ and $t_{k+1} = t_k \Rightarrow t$. If t_2 is well-typed, t_k is well-typed for all $k = 2, \cdots$.

With the results above, the following theorem that describes the main safety guarantees of Anvil easily follows.

Theorem D.20 (Anvil safety guarantees)**.** A well-typed Anvil program is safe.

E Proofs

E.1 Additional Lemmas

Lemma E.1. If a term t is well-typed and $\emptyset; G, R, M, \emptyset, e_0 \vdash t : T$, then for every local execution log $\mathcal{L} = \langle \alpha_0, \cdots, \alpha_k \rangle$ of t, there exists a timestamp function τ_G of G, such that if $\Gamma; G, R, M, C, e_c \vdash t' : (e_l, S_d)$ appears during inference of $\emptyset; G, R, M, \emptyset, e_0 \vdash t : T$, and $\Gamma'; I', M \vdash t' \rightsquigarrow \mathcal{L}' \lhd v$ appears during inference of $\emptyset; I, M \vdash t \rightsquigarrow \mathcal{L} \lhd v_0$, let $\mathcal{L}' = \langle \alpha'_0, \cdots, \alpha'_l \rangle$, then $\forall 0 \le i \le l : \alpha'_i \subseteq \alpha_{i+\tau_G(e_c)}$ and $\tau_G(e_c) + l = \tau_G(e_l)$. And for all $r \in D, \mathcal{L} \vdash v \downarrow D$, there exists $(e, S) \in R(r)$, such that $e \le_G e_l$ and $S_d \le_G S$.

Proof. We first show that such a function τ_G, if it exists, is a timestamp function of G. Consider the sub-terms t' that appear both in typing inference and evaluation. If $\Gamma; G, R, M, C, e_c \vdash t' : (e_l, S_d)$ appears during inference of $\emptyset; G, R, M, \emptyset, e_0 \vdash t : T$, and $\Gamma'; I', M \vdash t' \rightsquigarrow \mathcal{L}' \lhd v$ appears during inference of $\emptyset; I, M \vdash t \rightsquigarrow \mathcal{L} \lhd v_0$, we show that $\tau_G(e_c) + l = \tau_G(e_l)$ is consistent with the timestamp function definition. In addition, we show $\forall (x : (k, v')) \in \Gamma' : \Gamma(x) = (e'_l, S'_d) \rightarrow k = \max(0, \tau_G\left(e'_l\right) - \tau_G(e_c))$. This is shown by considering all possibilities for the rules applied and for each case replacing one constraint for the timestamp with a stricter equation. For example:

- T-Cycle and E-Cycle: $G(e_c) = \langle \#k, \{e_l\} \rangle, l = k$.
- T-Wait and E-Wait: $\tau_G(e_c) + l_1 = \tau_G\left(e'_l\right), \tau_G\left(e'_l\right) + l_2 = \tau_G(e_l), l = l_1 + l_2$.
- T-Ref and E-Ref: $l = k, G(e_l) = \langle \#0, \{e_c, e'_l\} \rangle$.

Let k be the number of all such sub-terms, then there are k linear equations, and each equation involves at least one unique variable. Hence any subset of those equations contain at least as many variables as equations. Therefore, the system of linear equations has at least one solution. In other words, τ_G exists and is a timestamp function of G.

Now we prove that with such a τ_G, $\forall 0 \le i \le l : \alpha'_i \subseteq \alpha_{i+\tau_G(e_c)}$, where $\mathcal{L}' = \langle \alpha'_0, \cdots, \alpha'_l \rangle$. This is shown by induction.

By induction, we can prove that for all $r \in D, \mathcal{L} \vdash v \downarrow D$, there exists $(e, S) \in R(r)$, such that $e \le_G e_l$ and $S_d \le_G S$. □

E.2 Lemma D.17

Proof. Let t be a well-typed Anvil term. From the definition of well-typedness, $\emptyset; G, R, M, \emptyset, e_0 \vdash t : T$. We show that for every local execution log $\mathcal{L} = \langle \alpha_0, \cdots, \alpha_k \rangle$, $\emptyset; I, M \vdash t \rightsquigarrow \mathcal{L} \lhd v_0$, the timestamp function in Lemma E.1 satisfies that for every value v, if $\Gamma'; I', M \vdash t' \rightsquigarrow \mathcal{L}' \lhd v$ appears during inference of $\emptyset; I, M \vdash t \rightsquigarrow \mathcal{L} \lhd v_0$, and $\Gamma; G, R, M, C, e_c \vdash t' : (e_l, S_d)$ appears in during inference of $\emptyset; G, R, M, \emptyset, e_0 \vdash t : T$, let $a = \tau_G(e_l), b = \tau_G\left(\min_{e \triangleright p \in S_d} \tau_G(e \triangleright p)\right)$, then $\text{UseSet}(\mathcal{L}, v) \subseteq [a, b]$ and for all D such that $\mathcal{L} \vdash v \downarrow D$, $\text{MutSet}(\mathcal{L}, D) \cap [a, b) = \emptyset$.

Consider each member $i \in \text{UseSet}(\mathcal{L}, v)$. By induction, it is obvious that one of the following must hold:

- $\text{ValUse}(v) \in \alpha_0'$ by E-IfThen, E-IfElse, and E-RegAssign. By Lemma E.1, $i = \tau_G(e_c)$
- $\text{ValCreate}(v, S_r, S_v) \in \alpha_0'$ by E-RegVal. Similarly, $i = \tau_G(e_l)$
- $\text{ValCreate}(v, S_r, S_v) \in \alpha_k'$ by E-Cycle. In this case, $i = \tau_G(e_l)$

In each case, we get $i \in [a, b]$. Thus $\text{UseSet}(\mathcal{L}, v) \subseteq [a, b]$.

Now we prove for $\mathcal{L} \vdash v \downarrow D, \text{MutSet}(\mathcal{L}, D) \cap [a, b) = \emptyset$. Consider each $i \in \text{MutSet}(\mathcal{L}, D)$. By definition, we have some $r \in D, \text{RegMut}(r) \in \alpha_i$. By Lemma E.1, there must be applications of E-RegAssign and T-RegAssign where $\tau_G(e_c) = i$ and there exists $(e, S) \in R(r)$ such that $e \leq_G e_l$ and $S_d \leq_G S$. Either $e_c <_G e$ or $S \leq_G e_c$. If $e_c <_G e$, by definition of $<_G$ and $\leq_G$, we have $i = \tau_G(e_c) < \tau_G(e) \leq_G \tau_G(e_l) = a$. Hence, $i \notin [a, b)$. If $S \leq_G e_c$, similarly, we have $b = \tau_G(S_d) \leq_G \tau_G(S) \leq_G \tau_G(e_c) = i$. Hence, we also have $i \notin [a, b)$. Therefore, $\text{MutSet}(\mathcal{L}, v) \cap [a, b) = \emptyset$.

By definition of safety, t is safe. □

E.3 Lemma D.18

Proof. Let $\mathcal{L}$ be an execution log of $t_1 \parallel_\Sigma t_2$. By definition, $\mathcal{L}$ can be obtained by combining $\mathcal{L}_1$ and $\mathcal{L}_2$, each an execution log of t_1 and t_2, respectively. Since t_1 and t_2 are well-typed, t_1 and t_2 are safe, and $\mathcal{L}_1, \mathcal{L}_2$ are also safe. By definition of safety, for every value v, there exists a_1, b_1, a_2, b_2, such that $\text{UseSet}(\mathcal{L}_1, v) \cup \text{LtSend}(\mathcal{L}_1, v) \subseteq [a_1, b_1] \subseteq \text{LtRecv}(\mathcal{L}_1, v), \mathcal{L} \vdash v \downarrow D_1, \text{MutSet}(\mathcal{L}_1, D_1) \cap [a_1, b_1) = \emptyset$, and $\text{UseSet}(\mathcal{L}_2, v) \cup \text{LtSend}(\mathcal{L}_2, v) \subseteq [a_2, b_2] \subseteq \text{LtRecv}(\mathcal{L}_2, v), \mathcal{L} \vdash v \downarrow D_2, \text{MutSet}(\mathcal{L}_2, D_2) \cap [a_2, b_2) = \emptyset$.

For $i \in \{1, 2\}$, if a $\text{ValCreate}(v, S_r, S_v)$ appears in $\mathcal{L}_i$, or, if no $\text{ValCreate}(v, S_r, S_v)$ appears in either $\mathcal{L}_i$ or $\mathcal{L}_{3-i}$ but $\text{LtRecv}(\pi.m, v)$ appears in $\mathcal{L}_i$, we say that $\mathcal{L}_i$ owns v. Obviously every v that appears in $\mathcal{L}$ is owned by either $\mathcal{L}_1$ or $\mathcal{L}_2$ but not both. We show that the following a, b satisfies that $\text{UseSet}(\mathcal{L}, v) \cup \text{LtSend}(\mathcal{L}, v) \subseteq [a, b] \subseteq \text{LtRecv}(\mathcal{L}, v), \mathcal{L} \vdash v \downarrow D, \text{MutSet}(\mathcal{L}, D) \cap [a, b) = \emptyset$:

1. If v does not appear in $\mathcal{L}$, then $a = a_1, b = b_1$.
2. If v appears in $\mathcal{L}$, and is owned by $\mathcal{L}_i$, $a = a_i, b = b_i$.

Case 1 is trivial.

For Case 2, by induction on the structure of $\text{DepSet}(\mathcal{L}, v)$, it is easy to obtain that $\text{UseSet}(\mathcal{L}, v) \cup \text{LtSend}(\mathcal{L}, v) \subseteq \text{UseSet}(\mathcal{L}_i, v) \cup \text{LtSend}(\mathcal{L}_i, v)$ and $\text{LtRecv}(\mathcal{L}_i, v) \subseteq \text{LtRecv}(\mathcal{L}, v)$. Therefore, we get $\text{UseSet}(\mathcal{L}, v) \cup \text{LtSend}(\mathcal{L}, v) \subseteq [a_i, b_i] \subseteq \text{LtRecv}(\mathcal{L}, v)$. Now we prove that $\text{MutSet}(\mathcal{L}, D) \cap [a_i, b_i) = \emptyset$. Without loss of generality, we assume $i = 1$.

We use induction on $\text{DepSet}(\mathcal{L}, v)$. Consider the following cases:

1. $\text{DepSet}(\mathcal{L}, v) = \emptyset$. In this case, either $\text{ValCreate}(v, S_r, \emptyset)$ or $\text{LtRecv}(\pi.m, v)$ appears in both $\mathcal{L}_1$ and $\mathcal{L}$. In both cases, $\text{MutSet}(\mathcal{L}, D) = \text{MutSet}(\mathcal{L}_1, D_1)$. Since $\text{MutSet}(\mathcal{L}_1, D_1) \cap [a_1, b_1) = \emptyset$, $\text{MutSet}(\mathcal{L}, D) \cap [a_1, b_1) = \emptyset$.
2. $\text{DepSet}(\mathcal{L}, v) = S_v$. In this case, $\text{ValCreate}(v, S_r, S_v)$ is in both $\mathcal{L}_1$ and $\mathcal{L}$. Consider each $u \in S_v$. Either u is owned by $\mathcal{L}_1$, or it is owned by $\mathcal{L}_2$. Let a', b' be selected such that $\text{UseSet}(\mathcal{L}_j, u) \cup \text{LtSend}(\mathcal{L}_j, u) \subseteq [a', b'] \subseteq \text{LtRecv}(\mathcal{L}_j, u)$ and $\text{MutSet}(\mathcal{L}_j, D') \cap [a', b')$, where $\mathcal{L}_j$ is the owner of u. Let a_0, b_0 be selected such that $\text{UseSet}(\mathcal{L}_1, u) \cup \text{LtSend}(\mathcal{L}_1, u) \subseteq [a_0, b_0] \subseteq \text{LtRecv}(\mathcal{L}_1, u)$ and $\text{MutSet}(\mathcal{L}_1, D_0) \cap [a_0, b_0)$. If $j = 1$, then $a_0 = a_u', b_0 = b_u'$. If $j = 2$, there must be a send operation involving u in $\mathcal{L}_2$ and a matching receive operation in $\mathcal{L}_1$. We have $[a_0, b_0] \subseteq \text{LtRecv}(\mathcal{L}_1, u) \subseteq \text{LtSend}(\mathcal{L}_2, u) \subseteq [a_u', b_u']$. In both cases, we have $[a_0, b_0] \subseteq [a_u', b_u']$. By induction assumptions, $[a', b') \cap \text{MutSet}(\mathcal{L}, D_u) = \emptyset$, hence $[a_0, b_0) \cap \text{MutSet}(\mathcal{L}, D_u) = \emptyset$. Combining all $u \in S_v$, by definition of $\text{LtRecv}(\mathcal{L}_1, v)$, $\text{MutSet}(\mathcal{L}_1, v)$, and $\text{MutSet}(\mathcal{L}, v)$: $[a, b] \subseteq \bigcap_{u \in S_v} \text{LtRecv}(\mathcal{L}_1, u) \subseteq \bigcap_{u \in S_v} [a_u', b_u']$, $\text{MutSet}(\mathcal{L}, v) = \text{MutSet}(\mathcal{L}_1, v) \cup \bigcup_{u \in S_v} \text{MutSet}(\mathcal{L}, u)$. Hence $[a, b) \subseteq \bigcap_{u \in S_v} [a_u', b_u')$, and $[a, b) \cap \text{MutSet}(\mathcal{L}, v) \subseteq \bigcap_{u \in S_v} [a_u', b_u') \cap \bigcup_{u \in S_v} \text{MutSet}(\mathcal{L}, u) = \emptyset$.

By induction, if v is owned by $\mathcal{L}_i$, $\text{UseSet}(\mathcal{L}, v) \cup \text{LtSend}(\mathcal{L}, v) \subseteq [a_i, b_i] \subseteq \text{LtRecv}(\mathcal{L}, v)$ and $[a_i, b_i) \cap \text{MutSet}(\mathcal{L}, D) = \emptyset$. Combining Case 1 and Case 2, we have shown that for all v, there exist such a and b. Therefore, the composition $t_1 \parallel_\Sigma t_2$ is safe. □

E.4 Lemma D.19

Proof. We show that for $k \geq 2$, if t_k is well-typed, t_{k+1} is also well-typed. By induction, this implies that if t_2 is well-typed, $t_k (k = 2, \cdots)$ are all well-typed.

Since t_k is well-typed, we have $\emptyset; G, R, M, \emptyset, e_0 \vdash t_k : T$. Because $t_k = t_{k-1}$ => t, there exists $\emptyset; G, R, M, C_1, e_0 \vdash t_{k-1} : (e_1, S_1)$ and $\emptyset; G, R, M, C_2, e_1 \vdash t : (e_2, S_2)$ which appear during inference. It is obvious that e_1 is a cut vertex in G, i.e., there exists a partition of $V = V_1 \cup V_2 \cup \{e_1\}$, such that all paths between V_1 and V_2 go through e_1, and it can be found such that $V_2 \cup \{e_1\}$ is the set of all nodes that appear in the inference rules used to obtain $\emptyset; G, R, M, C_2, e_1 \vdash t : (e_2, S_2)$. Let G_2 be the subgraph of G with $V' = V_2 \cup \{e_1\}$. Let G_2' be a graph obtained by relabelling nodes of G_2 such that e_1 is relabelled e_2 and nodes in V_2 are relabelled to nodes in V_3, where $V_3 \cap V = \emptyset$. Now let $G' = G \cup G_2'$. Obviously, assuming $<_G$ and $\leq_G$ always hold, we can obtain $\emptyset; G', R', M, C', e_0 \vdash t_{k+1} : T'$ such that the same nodes appear in rules inferring for t_k, and additionally there are rules inferring for t that simply map nodes used inferring $\emptyset; G, R, M, C_2, e_1 \vdash t : (e_2, S_2)$ from $V_2 \cup \{e_1\}$ to $V_3 \cup \{v_2\}$. Therefore, if t_{k+1} is not well-typed, there must be some unattainable $<_G$ or $\leq_G$ that appear in those rules. Consider different cases:

- Some $e_a <_G e_b$ or $e_a \leq_G e_b$, which only involves nodes but not event patterns, does not hold. Obviously, $\{e_a, e_b\} \in V_1 \cup \{e_1\}$ or $\{e_a, e_b\} \in V_2 \cup \{e_1\}$ or $\{e_a, e_b\} \in V_3 \cup \{e_2\}$. This always implies that a corresponding typing judgment does not hold for inferring well-typedness of t_k, contradicting the assumption.
- Some typing judgment that involves $e_a \triangleright p$ does not hold. This similarly imply a contradiction a rule involved in inferring the well-typedness of t_k does not hold.

By contradiction, t_{k+1} is well-typed.

□