

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

Jason Zhijingcheng Yu*
yu.zhi@comp.nus.edu.sg
National University of Singapore
Singapore

Aditya Ranjan Jha*
arjha@comp.nus.edu.sg
National University of Singapore
Singapore

Umang Mathur
umathur@comp.nus.edu.sg
National University of Singapore
Singapore

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

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

Abstract

Hardware designs routinely use stateless signals which change with their underlying registers. Unintended behaviours arise when a register is mutated even when its dependent signals are expected to remain stable (unchanged). Such *timing hazards* are common because, with a few exceptions, existing HDLs lack the abstraction for stable values and delegate this responsibility to hardware designers, who then have to carefully decide whether a value remains 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* without sacrificing expressiveness for cycle-level timing control or dynamic timing behaviours. Instead of abstracting away differences between registers and signals, Anvil’s type system exposes them fully but captures timing relationships between register mutations and signal usages for enforcing 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 for implementing key timing-sensitive modules in an open-source RISC-V CPU, which demonstrates its expressiveness and practicality.

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 challenging.

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 underlying registers are not mutated. Incorrect timing of register mutation and signal use thus easily introduces invalid or wrong 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 multiple modules executing in parallel, and communicating with each another via sharing signals.

Most existing HDLs such as SystemVerilog [5], VHDL [4], and Chisel [10], fail to catch timing hazards at compile time, leaving designers to discover these issues only during simulation, without any compiler warnings. Indeed, designers frequently seek help on discussion forums [13, 15, 37] simply to pinpoint the origins of the errors. Timing hazards are prevalent even among experienced designers and in widely used open-source hardware components (please see real-world examples in Appendix C).

A principled way to eliminate timing hazards is by guaranteeing *timing safety* in the HDL. The key challenge, though, towards this goal is to simultaneously also provide enough expressiveness for writing general-purpose hardware design use cases. Some existing HDLs have been designed to provide timing safety but at a significant cost of expressiveness, making them only suitable for specific applications. Some high-level synthesis (HLS) languages [2, 6, 9] offer a software-like programming model for hardware design. Timing hazards are not a concern in HLS languages since they abstract away cycle latencies as well as the distinction between wires and registers, making all values stable indefinitely, à la variables are in software programming. This expressiveness for cycle-level control and wires is unfortunately absent in

*Equal contribution.

such languages. This is an essential abstraction in general-purpose hardware designs, especially where performance is a priority. Consequently, the applicability of HLS languages is limited to speeding up algorithms with programmable hardware (e.g., FPGA). Other languages focus only on specific types of hardware designs, such as CPUs [38] and static pipelines [29, 35].

We present Anvil, the first HDL we know of that guarantees timing safety while maintaining expressiveness for general-purpose hardware design use cases. Anvil allows hardware designers to seamlessly specify cycle-level delays and to express whether a value is stored in a register. Anvil 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. Anvil performs type checking that reasons about whether each use of signal takes place in the time window in which it carries a stable and meaningful value and rejects code that is not timing safe. Designs written in Anvil can thus specify precise cycle-level behaviour and register updates. This is in contrast to HLS languages [2] 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 stable. 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 absolute (fixed) number of cycle, 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 [29] which where the type system, meant to enforce timing safety, only allows one to express designs with fixed static timing behaviours.

We have implemented Anvil (Section 6). The Anvil compiler 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 code bases in other HDLs, thus allowing incremental adoption and making Anvil immediately useful. We have successfully used Anvil to implement key latency-sensitive components for an open-source RISC-V CPU implemented in SystemVerilog. Despite our Anvil prototype being an early-stage HDL, the measured area and power footprints of Anvil designs show practical overhead: power overhead between 0.5% and 6.4% and area overhead between 0.6% and 25% compared with hand-coded open-source SystemVerilog implementations. We have released Anvil publicly at <https://github.com/jasonyu1996/anvil>.

```

module Memory (
  ...
  input logic[7:0] inp,
  input logic req,
  output logic[7:0] out
);

```

Figure 1. Interface of a memory module in SystemVerilog.

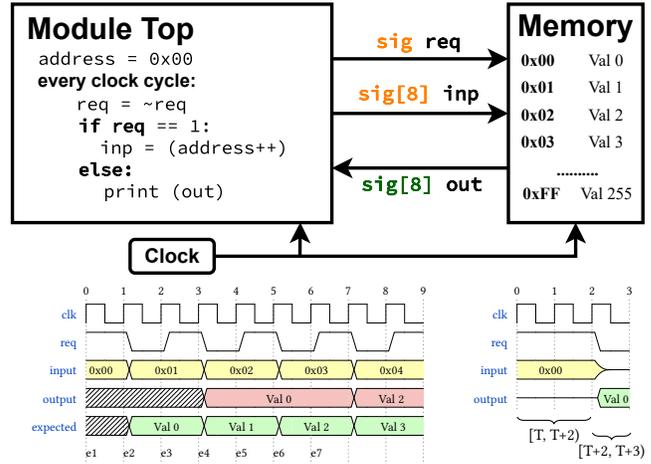


Figure 2. Module Top interfaced with Memory.

Our contributions. We introduce Anvil, an novel HDL to guarantee timing safety without sacrificing expressiveness, e.g., for cycle-level control and dynamic timing behaviours. It features a novel abstractions for specifying timing constraints and type-checking to enforce them. The system allows for general-purpose hardware design use cases and incremental interfacing with existing SystemVerilog code.

2 Motivation

Consider the interface of a memory module in SystemVerilog in Figure 1. Unlike software, hardware modules communicate using signals that can be continuously read and updated. Consider an interfacing hardware module (Figure 2, top), 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 2, 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, the memory subsystem takes 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 $\text{Val } 0$ 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 timing hazard, where unintended values are used or values in use are changed unexpectedly. Here, the module Top modifies its input while the memory still processes the address lookup request. It also reads the output before it is ready.

Timing hazards in existing HDLs. Timing hazards arise in many popular HDLs such as SystemVerilog and VHDL as they lack an abstraction for the designer to express stable values across multiple cycles. Further, these languages also do not provide a mechanism to encode timing constraints pertaining to modification and use of signals shared between communicating modules.

Some existing HDLs are designed to mitigate or prevent timing hazards. Bluespec SystemVerilog (BSV) [3] 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, if Top reads a value from a cache and enqueues it into a FIFO queue that only accepts requests when not full, the design would consist of two rules: one to invoke the cache’s read method and another to enqueue the retrieved value into the FIFO. BSV’s scheduler ensures that rules executed in each cycle do not conflict (i.e., mutate the same registers) and each rule executes atomically. However, rules only describe operations to execute for the current cycle. Scheduling is also performed for each cycle independently. BSV does not reason about behaviours across multiple cycles [3]. Therefore, in the example, the designer still needs to manually ensure that the response from the cache remains stable and avoid timing hazards.

Existing HDLs that provide *timing safety* (i.e., the guarantee of no timing hazards) face challenges in maintaining expressiveness. Some high-level synthesis (HLS) languages [2] provide abstractions of stable 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 Anvil is the Filament HDL [29]. 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 is stable and

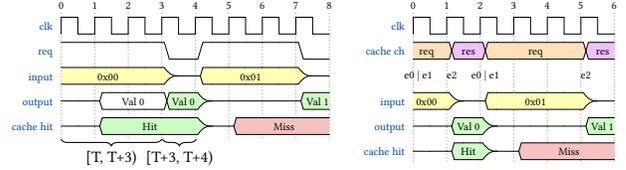


Figure 3. Cache output waveform expressed safely with static (left) and dynamic (right) timing contract.

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 2 (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 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 3 (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 [29], 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 [20]. 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.

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 A).

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. Figure 4 illustrates how Anvil’s type system distinguishes between safe and unsafe process descriptions. The description `proc Top_Unsafe` refers to Anvil’s representation (desugared for understanding) of the same circuit `Top` shown in Figure 2. In contrast, `proc Top_Safe` refers to the description that captures the timing characteristics of the memory subsystem with a cache, as depicted in Figure 3 (right). Anvil first derives the action sequence and verifies whether the process description adheres to the constraints that the timing contracts dictate.

In our examples, the event named `req` marks the clock cycle when the `address` sent by `proc Top_Unsafe` or `Top_Safe` is acknowledged on the channel, whereas the event named `res` refers to the clock cycle when `data` sent by the memory sub-system is acknowledged. For memory without a cache, the expected behaviour is specified in a timing contract, which is encapsulated by the *memory channel definition*. It requires that the `address` sent by `Top_Unsafe` remains stable and available starting from the `req` event for two clock cycles. Similarly, it specifies that the `data` sent by memory must be available for one clock cycle starting from the `res` event.

The timing contract is not satisfied by process `Top_Unsafe`, and Anvil can detect it at compile time. In the HDL code for `proc Top_Unsafe`, the `address` is sent over the channel during $[e_0, e_0 + 1)$, but the timing of the acknowledgement is uncertain. The output value is utilized during $[e_0 + 1, e_0 + 2)$, but the occurrence of the `res` event is unknown, as it depends on whether the memory system has sent it. As a result, it is statically unclear whether the following address was sent before the previous output was received and acknowledged. Furthermore, the input address is modified during $[e_0 + 1, e_0 + 2)$, which violates the requirement for the address to remain stable for two cycles after acknowledgement. The static checker in Anvil enables catching such errors in circuit design.

The contract for the example of memory with cache is specified in the cache channel definition. It implies that the `address` sent by process `Top_Safe` should be available after the event `req`, till the next occurrence of event `res` as specified by its lifetime (`req, req->res`). Similarly, the `data` sent by the memory sub-system has a lifetime from (`res, res->req`). As shown in Figure 4 (right), the process `Top_Safe` definition satisfies this contract, and thus it is deemed safe.

Anvil in action. Anvil is designed as a general-purpose HDL that eliminates timing hazards. It allows the designer to declare timing contracts directly in the interface and provides higher-level abstractions to enforce those contracts. Furthermore, the type system of Anvil ensures that these contracts are respected.

For instance, consider Figure 4 (bottom). The module `Top` retrieves a value from a cache memory subsystem and sends it to a FIFO. Anvil enforces the timing contract by detecting violations and guiding the designer toward a timing-safe implementation that respects the channel contract.

Anvil achieves this without sacrificing expressiveness. It allows capturing timing characteristics precisely without a trade-off in circuit performance (latency). Since channels in Anvil are stateless, message acknowledgements happen instantly. With dynamic contract definitions, designing circuits with varying timing characteristics is possible. Figure 3 shows the simulation output for both Anvil’s dynamic contracts (right) and static contracts (left). It can be seen that no extra clock cycle overhead is introduced.

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*. They are akin to unbuffered channels in Go [16], where the `send` and its corresponding `receive` operations are required to happen instantaneously. Each *channel type definition* in Anvil describes a template for channels. Figure 5 presents an example of a channel type definition.

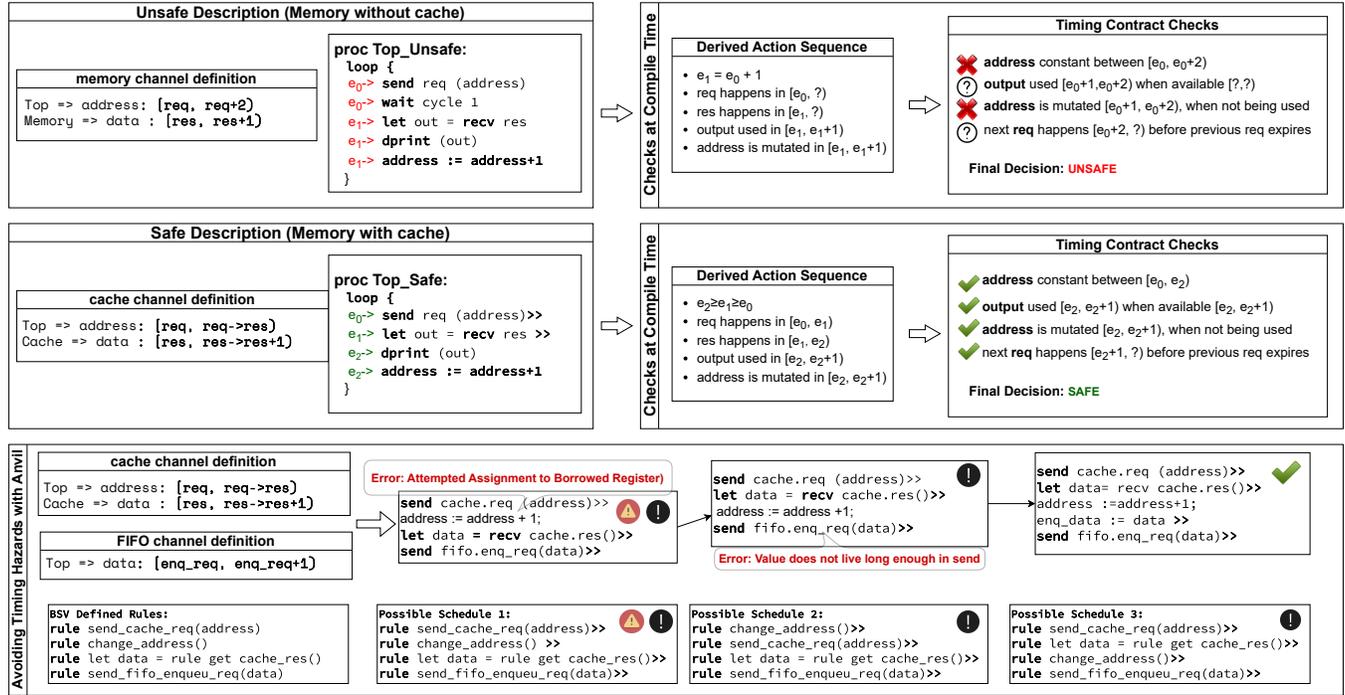


Figure 4. Top & Middle: Verification of unsafe and safe Top modules (memory without vs. with cache); Bottom: Anvil guiding to timing-safe design vs. BSV conflict-free schedules.

Messages. The definition specifies the different types of *messages* that can be sent and received over a channel. Each type of message is identified by a unique message identifier and annotated with its direction, which is left or 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 stable 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 Figure 5 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, which

specifies the timing patterns regarding the sending or receiving of the message. The default sync mode, `@dyn`, specifies that a one-bit signal is used to synchronize during run-time. In the case of `wr_req` in Figure 5, both the sender and the receiver use this dynamic sync mode. When static knowledge is available about when sending or receiving can happen, the sync mode can encode such knowledge. The left side of message `rd_req` has the *static sync mode* `@#2`, which specifies that it is to be ready to receive the message within at most 2 cycles after the last time the message is received. For the left side, Anvil statically checks this is upheld. For the right side, Anvil uses this knowledge to statically check that the every time sending of `rd_req` takes place, the receiver must be ready to receive it. A sync mode can also be *dependent*. Both sides of `wr_res` have `@#wr_req + 1`, which specifies that the message is sent and received exactly one cycle after `wr_req`.

4.2 Process

Each Anvil component is represented as a *process*. A process is 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, other process instantiations, and threads.

```
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
}
```

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

Figure 5. A channel type definition.

4.3 Thread

Each process contains one or more threads that execute concurrently. Two types of threads are available: *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 `loop_term` 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, recursion 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 the request. Recursives provide convenience for pipelining comparable to what Filament [29] provides.

```
recursive {
  let r = recv ep1.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 reg := t` and `set reg := 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 4 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 number 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 ensuring timing safety statically, as we will discuss in detail next.

5 Safety of Anvil Programs

The type system of Anvil ensures that each process implementation adheres to the contracts defined by the channels that 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 stable 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. **Using Stable Values:** 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 the data sent needs

to be live for (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 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 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 fixed 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 stable (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 termed *live* if it falls within its defined lifetime. Conversely, it is deemed *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 $[e_1, e_1 \triangleright \text{ch1}.\text{enc_res})$, 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 $e_9 \triangleright \text{ch1}.\text{enc_req}$, where e_9 is the event corresponding to the completion of the message sending. Therefore, the lifetime is $[e_5, e_9 \triangleright \text{ch1}.\text{enc_req})$.
- The signal $(\text{ptext} \wedge \text{r1_key}) + \text{noise}$ has a lifetime that is the intersection of the lifetimes of ptext , r1_key , and noise , $[e_3, \{e_2 \triangleright \#1, e_1 \triangleright \text{ch1}.\text{enc_res}\})$.
- 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 $[e_5, e_9 \triangleright \text{ch1}.\text{enc_req}) \cup [e_5, e_8 \triangleright \#2)$.

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

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, 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 compare pairs of events as to the order in which they occur during run-time. In particular, Anvil decides if

```

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] = rcv ch1.enc_req;
    e0 let noise [e2, e2 ▷ #1] = rcv 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, ∞) >>
  }
}

```

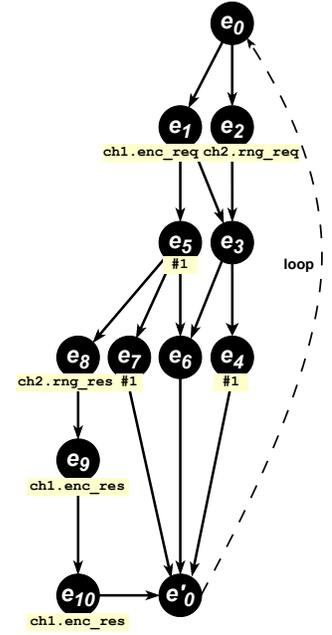


Figure 6. Left: Encrypt in Anvil, annotated with timing information. Each blue-shaded annotation marks the event corresponding to the time a term evaluation starts. Each yellow-shaded annotation marks the inferred lifetime associated with the red-circled term next to it. Right: Event graph corresponding 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.

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.

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

However, in $\text{rd1_ctext} := (\text{ptext} \wedge \text{r1_key}) + \text{noise}$, the signal $(\text{ptext} \wedge \text{r1_key}) + \text{noise}$ cannot be statically guaranteed to be live. In this case, Anvil compares its lifetime of noise , $[e_3, \{e_2 \triangleright \#1, e_1 \triangleright \text{ch1.enc_res}\})$ with the time when it is used, $[e_3, e_3 \triangleright \#1)$. It cannot obtain $e_3 \triangleright \#1 \leq_G \{e_2 \triangleright \#1, e_1 \triangleright \text{ch1.enc_res}\}$. 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 $[e_5, e_9 \triangleright \text{ch1.enc_req}) \cup [e_5, e_8 \triangleright \#2)$. To determine if r2_key is still loaned when the assignment $\text{r2_key} := \text{r1_key} \wedge \text{noise}$ 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. Note that the assignment takes one cycle to update the register value, and e_7 is the event that corresponds to its completion. 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 not allowed by Anvil. Intuitively, a value sourced from r2_key is sent through

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

$\delta \in \text{data-types}$ $\star \in \text{binary-operators}$ $\diamond \in \text{unary-operators}$
 $\pi \in \text{endpointsxinidentifiers}$ $r \in \text{registers}$ $m \in \text{messages}$
 $c \in \text{channels}$ $p \in \text{processes}$ $N \in \mathbb{N}$

Figure 7. Anvil syntax.

`ch2.rng_res` at e_5 , which requires it to be live until two cycle after the send completes. However, the value `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 while they are still expected to be stable according to 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 $[e_8, e_9 \triangleright \text{ch1}. \text{enc_req})$ and $[e_9, e_{10} \triangleright \text{ch1}. \text{enc_req})$ 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 $[e_9, e_{10} \triangleright \text{ch1}. \text{enc_req})$. In this case, this check passes as $[e_9, e_{10} \triangleright \text{ch1}. \text{enc_req}) \subseteq_G [e_9, \infty)$.

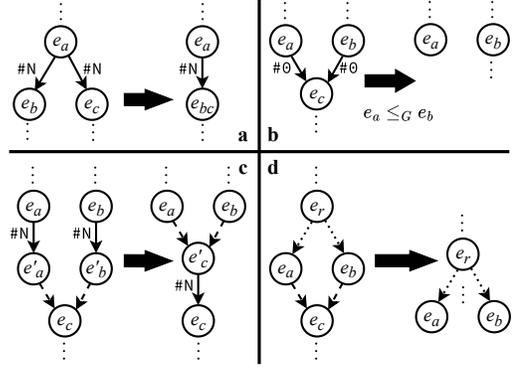
5.5 Formalization

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 A and B.

6 Implementation

We have implemented Anvil in OCaml. The Anvil compiler performs type checking on Anvil code and generates synthesizable SystemVerilog. We have publicly released the compiler at <https://github.com/jasonyu1996/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.


Figure 8. Examples of event graph optimizations. Dotted edges represent branching and dashed edges represent joining of branches.

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*, each applying an optimization strategy. Figure 8 shows example optimization passes. In general, two events can be merged into one if they always occur at the same time. Many of the optimization passes identify special patterns of such cases.

(a) Merging identical outbound edge labels. This optimization pass merges identifiers outbound edges of an event e_a that share the same label (e.g., edges labelled with $\#N$ to e_b and e_c). The events those edges connect to are merged.

(b) Removing unbalanced joins. A *join* is an event that waits on two predecessor events. This optimization pass removes a join e_c when either of its predecessors (e_b) always occurs no earlier than the other (e_a), i.e., $e_a \leq_G e_b$. The outbound edges of e_c are migrated to e_b .

(c) Shifting branch joins. A *branch join* is an event e_c that joins two branches. 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 (i.e., register assignments and message sending or receiving), the branch join can be shifted earlier: instead of delaying by N cycles and join, the branches can join to 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 they share the same predecessor e_r , then e_c can be merged into e_r .

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 used in the synchronization. The compiler only generates both `valid` and `ack` when the specified synchronization mode is dynamic for both the sender and the receiver. If the synchronization 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 synchronization 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) Message send/receive events (only those with dynamic synchronization modes): whether the message has been sent or received.

7 Evaluation

We aim to answer two questions through evaluation:

1. **Expressiveness.** Can Anvil express diverse hardware designs, including ones that cannot be expressed using static timing contracts?
2. **Practicality.** Can Anvil scale to real-world hardware designs and generate reasonably efficient hardware designs compared with existing HDLs?

7.1 Expressiveness

We present four case studies for different atomic characteristics. Three of them cannot be expressed using static timing contracts. We implement each design in Anvil and measure its cycle-level timing behaviours, including throughputs and latencies, against the design goals.

Circuits with State-Dependent Access. Many circuits exhibit timing characteristics that depend on their current state and input. Anvil can express this behaviour without introducing any delay overhead. We implemented an 8-bit FIFO cache with four sets, associativity of four and following a First-In, First-Out (FIFO) replacement policy. The channel definition (`cache_ch`) encodes the contract for both the address and the associated data.

Circuits for Concurrent Request Handling. Circuits must efficiently handle concurrent requests when multiple components share a single resource. The contract should ensure the

resource is accessed only within the granted time interval. A static timing contract cannot represent variable access or waiting times. We implemented an 8-bit FIFO queue with a size of eight. The queue operates with two parallel threads: one for enqueueing requests (Min Latency: 1 cycle) and another for dequeuing (Min Latency: 0 cycles). If the queue is empty, the dequeue thread waits for an enqueue request to complete before acknowledging the subsequent dequeue request. Similarly, when the buffer is full, the enqueue thread waits for a dequeue operation to free space.

Circuits for Asynchronous Processing. Anvil abstracts synchronous processing using message passing on channels between processes. To improve the efficiency of dynamically scheduled circuits, we introduce asynchronous processing using buffers. The timing contract for such circuits must guarantee that a request remains available until serviced, which a static timing contract cannot represent. We implemented an 8-bit priority arbiter connecting four producers to one consumer. The arbiter uses a priority-based scheme to allocate resources. Whenever the consumer is ready to receive a new request, it checks the priority of the currently available producer requests and grants access accordingly. The arbiter interfaces with four endpoints of the channel type `producer_ch` and one endpoint of the channel type `consumer_ch`. The channel definitions encode the necessary contracts, ensuring access rights remain valid until the consumer completes the request and sends back a response.

Circuits with Static Schedules. We use a static timing contract for circuits where schedules can be entirely determined [29]. This contract specifies the lifetime of signals and the corresponding synchronization pattern. While this type of contract does not emphasize dynamic timing safety, we demonstrate that Anvil can still accurately capture its behaviour. We designed a 4-stage pipelined arithmetic processor to showcase Anvil’s expressiveness in capturing static schedules. The processor performs 4 operations: Add, Subtract, AND, and OR. It consists of 4 stages: *Instruction Fetch*, *Decode*, *Execute*, and *Write Back*. Each stage takes one cycle to complete.

Result. Table 1 shows the observed results of the designs implemented in Anvil against the design goals. All implemented designs reach the design goals: they introduce no additional cycle latencies or throughput overhead.

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

7.2 Practicality

We designed Anvil to facilitate our own implementation effort of experimental extensions to CVA6 [39], an open-source RISC-V CPU implemented in SystemVerilog. Such a use case requires not only support for expressing complex

Table 1. Summary of Performance Metrics for Case Studies in Anvil. The design goals are shown in angle brackets (<>). Latency and timing are expressed in clock cycles and throughput in instructions per cycle.

Case Studies	Metrics	Measured
FIFO Cache	Cache Hit Latency <#1>	#1
	Cache Miss Latency <#3>	#3
	Variable Output Latency	Supported
FIFO Queue	Enqueue Latency (Not Full) <#1>	#1
	Dequeue Latency (Not Empty) <#0>	#0
	Variable (Enqueue/Dequeue) Latency	Supported
Priority Arbiter	Request Selection and Read Latency <#1>	#1
	New Request Latency <#1+Access Time>	#1 + (Access Time)
	Minimum Supported Access Time <#1>	#1
Four-Stage Pipelined Arithmetic Processor	Variable Access Time	Supported
	Pipeline Stage Latency <#1>	#1
	Throughput <1>	1

hardware designs, but also integration with an existing SystemVerilog code. We evaluate the practicality of using Anvil in this context in comparison with existing HDLs. Additionally, for static pipelined designs, we evaluate Anvil against Filament [29], which provides specialized abstractions for static pipelines.

Comparison Benchmarks: SystemVerilog. As a baseline, we choose an off-the-shelf open-source implementation of RISC-V CPUs: CVA6 [39] (implemented in SystemVerilog). We selected two key modules of a CPU core: the translation lookaside buffer (TLB) and the page table walker (PTW), which are highly sensitive to dynamic latencies. We re-implemented the same logic in Anvil and verified their functional correctness by replacing the original models in CVA6 with Anvil-generated code. We synthesize both Anvil and baseline hardware designs for a commercial 22 nm ASIC process and target two different clock frequencies of 800 MHz and 2000 MHz. To demonstrate the quality of Anvil-generated code with respect to the baseline designs, we report the area and power footprints at both frequencies.

Comparison Benchmarks: Filament. We also compare with Filament for static pipelined designs, the type of designs Filament is specifically designed to support. For this purpose, we choose a systolic array and a static pipelined arithmetic logic unit (ALU) implemented in Filament. We re-implemented both designs in Anvil using recursives (Section 4.3). The same synthesis targets for clock frequencies and ASIC setup as for SystemVerilog are used here as well.

Remark. We also tried to compare with BlueSpec (BSV). Note that BSV does not provide timing safety guarantees, as is shown in Figure 4 (Bottom), which yields a timing hazard. We implemented the pipelined Page Table Walker (PTW) from another open-sourced RISC-V core called Flute [7] to evaluate expressiveness. However, we observed about a 25% overhead in power and area during synthesis for Anvil. Upon closer inspection, we found that BSV’s efficiency largely came from extensive interactions with an external Verilog module, which is not leveraging BlueSpec. This module was the primary source of power and area consumption in both

implementations. Therefore, we did not consider it in an evaluation of the generated code for BSV and Anvil.

Result. Table 2 summarizes the area and power footprints of each Anvil design compared to its corresponding baseline.

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

8 Related Work

Timing-oblivious HDLs. The industry-standard HDLs, SystemVerilog [5] and VHDL [4], 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 [1, 10, 14, 34] 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 [11, 30] 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) [2, 6, 21, 35, 38] specialize in specific applications and abstracts away cycle-level timing and the distinction between stateless signals and registers.

Timing-aware HDLs. Filament [29] 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 [27] 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 [23] is an IR that supports specifying both static and dynamic timing through timing guards.

Hazard prevention. BaseJump [36] and Wire Sorts [12] are type systems designed to identify combinational loops, a separate concern than timing hazards. ShakeFlow [18] proposes a dynamic control interface to prevent structural hazards in pipelined designs. Hazard Interfaces [22] 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.

9 Conclusions

In this work, we formalize the problem of timing hazards and present Anvil, a new 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 which are guaranteed to remain

Table 2. Summary of area and power footprints of Anvil and baseline designs in SystemVerilog and Filament. The **X** marks indicate slack time violations, so the resulting design is not practical.

Hardware Designs	Power @ 800 MHz (mW)		Area @ 800 MHz (um ²)		Power @ 2000 MHz (mW)		Area @ 2000 MHz (um ²)	
	Baseline	Anvil	Baseline	Anvil	Baseline	Anvil	Baseline	Anvil
CVA6 TLB (SV)	3.771	3.791 (0.53%)	5548	5582 (0.61%)	9.458	9.592 (1.42%)	5914	6145 (3.90%)
CVA6 PTW (SV)	0.433	0.459 (6%)	476	576 (21%)	1.082	1.152 (6.4%)	476	595 (25%)
Pipelined ALU (Filament)	0.169	0.345	733.86	414.81	0.606 X	0.863	1083 X	414
Systolic Array (Filament)	2.073 X	1.812	5667 X	2156	5.282 X	4.543	5856 X	2215

unchanged throughout their lifetimes. While achieving this, it provides the capability of cycle-level timing control and the expressiveness for describing designs with dynamic timing characteristics.

Acknowledgments

We thank NUS KISP Lab members for their feedback, Yaswanth Tavva and Sai Dhawal Phaye for the help with infrastructure setup. This research is supported by a Singapore Ministry of Education (MOE) Tier 2 grant MOE-T2EP20124-0007.

References

- [1] [n. d.]. Spinal Hardware Description Language — SpinalHDL documentation. <https://spinalhdl.github.io/SpinalDoc-RTD/master/index.html>
- [2] [n. d.]. XLS: Accelerated HW Synthesis. <https://google.github.io/xls/>
- [3] 2008. Bluespec SystemVerilog Reference Guide.
- [4] 2009. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (2009), 1–640. <https://doi.org/10.1109/IEEESTD.2009.4772740>
- [5] 2018. *1800-2017 - IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*. IEEE, Place of publication not identified.
- [6] 2023. IEEE Standard for Standard SystemC® Language Reference Manual. *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)* (2023), 1–618. <https://doi.org/10.1109/IEEESTD.2023.10246125>
- [7] 2025. Bluespec/Flute. Bluespec, Inc..
- [8] Alexforencich. [n. d.]. Tx signals for raw ethernet frame Issue 121 alexforencich/verilog-ethernet. <https://github.com/alexforencich/verilog-ethernet/issues/121>
- [9] C. Baay. 2015. *Digital Circuits in ClaSH*. Ph. D. Dissertation. University of Twente, Enschede, The Netherlands. <https://doi.org/10.3990/1.9789036538039>
- [10] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, San Francisco California, 1216–1225. <https://doi.org/10.1145/2228360.2228584>
- [11] 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*. ACM, London UK, 243–257. <https://doi.org/10.1145/3385412.3385965>
- [12] 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 (Virtual, Canada) (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 175–189. <https://doi.org/10.1145/3453483.3454037>
- [13] Abhishek Chunduri. 2020. '1011' Overlapping (Mealy) Sequence Detector in Verilog.
- [14] Jan Decaluwe. 2004. MyHDL: a python-based hardware description language. *Linux J.* 2004, 127 (Nov. 2004), 5.
- [15] Dimitras-Vtool. [n. d.]. Alu_full_fifo_in_test · Issue #1 · Dimitras-Vtool/ALU. <https://github.com/dimitras-vtool/ALU/issues/1>.
- [16] Alan A.A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language* (1st ed.). Addison-Wesley Professional.
- [17] fpgasystems. [n. d.]. Each completion queue contains 2-cycle burst valid signal | Issue 78 | fpgasystems/Coyote. <https://github.com/fpgasystems/Coyote/issues/78>
- [18] 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*. ACM, Vancouver BC Canada, 702–717. <https://doi.org/10.1145/3575693.3575701>
- [19] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677. <https://doi.org/10.1145/359576.359585>
- [20] Charles A. R. Hoare. 2000. *Communicating Sequential Processes* (reprinted ed.). Prentice Hall, New York.
- [21] Steven F. Hoover and Ahmed Salman. 2018. Top-Down Transaction-Level Design with TL-Verilog. *CoRR* abs/1811.01780 (2018). arXiv:1811.01780 <http://arxiv.org/abs/1811.01780>
- [22] 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 (June 2024), 24 pages. <https://doi.org/10.1145/3656378>
- [23] 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 (Oct. 2024), 2242–2267. <https://doi.org/10.1145/3689790>
- [24] KULeuven-Micas. [n. d.]. Fix ALU valid-ready signal by rgantonio | Pull Request | #163 KULeuven-MICAS/snax_cluster. https://github.com/KULeuven-MICAS/snax_cluster/pull/163/commits/be67fbfd7ab821b7c7928ccce1801d3e34fb316
- [25] lowRISC. [n. d.]. Add an INSTR_VALID_ID signal to completely decouple the pipeline stages, LOWRISC/IBEX@F5D408D. <https://github.com/lowRISC/ibex/commit/f5d408d7f4523f4f105cf1fe3029bb28dba12d87>
- [26] lowRISC. [n. d.]. Timing issues in FW_OV "Insert Entropy" feature. <https://github.com/lowRISC/opentitan/issues/10983>. GitHub Issue 10983, accessed: 2024-11-12.
- [27] 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*. ACM, Vancouver BC Canada, 189–201. <https://doi.org/10.1145/3623278.3624767>
- [28] MITRE. 2024. CWE-1298: Hardware Logic Contains Race Conditions. <https://cwe.mitre.org/data/definitions/1298.html>. Accessed: 2024-10-26.
- [29] 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 (June 2023), 343–367. <https://doi.org/10.1145/3591234>
- [30] R. 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, 2004. MEMOCODE '04*. 69–70. <https://doi.org/10.1109/MEMCOD.2004.1459818>
- [31] OpenHW Group. 2024. Issue 145: Clarification of valid-ready handshake dependency. <https://github.com/openhwgroup/core-v-xif/issues/145> Accessed: 2024-11-12.
- [32] OpenHW Group. 2024. Issue 194: Handshake rules additional note. <https://github.com/openhwgroup/core-v-xif/issues/194> Accessed: 2024-11-12.
- [33] Pulp-Platform. [n.d.]. Add missing w_valid pulp-platform/core2axi@25eba94. <https://github.com/pulp-platform/core2axi/commit/25eba94af4a58249cfa65e1c259ed4b4c5bbfd12>
- [34] 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]
- [35] 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. <https://doi.org/10.1109/FPL57034.2022.00075>
- [36] Michael Bedford Taylor. 2018. Basejump STL: systemverilog needs a standard template library for hardware design. In *Proceedings of the 55th Annual Design Automation Conference (San Francisco, California) (DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 73, 6 pages. <https://doi.org/10.1145/3195970.3199848>
- [37] titan. 2014. Synchronizing Multiplier with Adder to Form Mac.
- [38] 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*. ACM, San Diego CA USA, 719–732. <https://doi.org/10.1145/3519939.3523455>
- [39] 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 (Nov. 2019), 2629–2640. <https://doi.org/10.1109/TVLSI.2019.2926114>

message definitions $M ::= \{\pi.m : p, \dots\}$
 message set $\Sigma ::= \{\pi.m, \dots\}$
 composition $\kappa ::= t \mid \kappa \parallel_{\Sigma} \kappa$
 program $\mathcal{P} ::= (\text{loop}\{t\}, M) \mid \mathcal{P} \parallel_{\Sigma} \mathcal{P}$

Figure 9. Anvil abstract syntax

A Formalization Details

A.1 Abstract Syntax

For convenience of formal reasoning, we also define an abstract syntax of Anvil programs, shown in Figure 9, allowing us to discuss parallel composition in a style similar to communicating sequential processes (CSP) [19]. The \parallel_{Σ} 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.

A.2 Semantics

Execution log. An execution log is simply a sequence $\mathcal{L} = \langle \alpha_0, \dots, \alpha_k \rangle$, 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 $\text{loop}\{t\}$ 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 A.1 (Execution log). An execution log consists of a sequence of sets $\mathcal{L} = \langle \alpha_0, \alpha_1, \dots, \alpha_k \rangle$. The finite set α_i contains the actions in the i -th cycle, each of the following form:

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

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

$$\frac{}{\Gamma; \{v\}, M \vdash \text{cycle } \#k \rightsquigarrow (\emptyset^{k+1} \circ \{\{\text{ValCreate}(v, \emptyset, \emptyset)\}\}) \triangleleft v} \text{ (E-CYCLE)}$$

$$\frac{}{\Gamma; \{v\}, M \vdash n \rightsquigarrow \{\{\text{ValCreate}(v, \emptyset, \emptyset)\}\} \triangleleft v} \text{ (E-LITERAL)}$$

$$\frac{\Gamma; I_1, M \vdash t_1 \rightsquigarrow \mathcal{L}_1 \triangleleft v_1 \quad \Gamma; I_2, M \vdash t_2 \rightsquigarrow \mathcal{L}_2 \triangleleft v_2 \quad I_1 \cap I_2 = \emptyset}{\text{shift}(\Gamma, |\mathcal{L}_1| - 1); (I_1 \cup I_2), M \vdash t_1 \Rightarrow t_2 \rightsquigarrow (\mathcal{L}_1 \circ \mathcal{L}_2) \triangleleft v_2} \text{ (E-WAIT)}$$

$$\begin{array}{c}
 \frac{\Gamma; I_1, M \vdash t_1 \rightsquigarrow \mathcal{L}_1 \triangleleft v_1 \quad \Gamma, x : (|\mathcal{L}_1| - 1, v_1); I_2, M \vdash t_2 \rightsquigarrow \mathcal{L}_2 \triangleleft v_2}{\Gamma; (I_1 \cup I_2), M \vdash \text{let } x = t_1 \text{ in } t_2 \rightsquigarrow (\mathcal{L}_1 \uplus \mathcal{L}_2) \triangleleft v_2} \quad (\text{E-LET}) \\
 I_1 \cap I_2 = \emptyset \\
 \\
 \frac{\Gamma(x) = (k, v)}{\Gamma; \emptyset, M \vdash x \rightsquigarrow \emptyset^{k+1} \triangleleft v} \quad (\text{E-REF}) \\
 \\
 \frac{\Gamma; I, M \vdash t \rightsquigarrow \mathcal{L} \triangleleft 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 \triangleleft v'} \quad (\text{E-REGASSIGN}) \\
 \\
 \frac{\Gamma; I, M \vdash t \rightsquigarrow \mathcal{L} \triangleleft v \quad v' \notin I \quad k \in \mathbb{N}}{\Gamma; (I \cup \{v'\}), M \vdash \text{send } \pi.m(t) \rightsquigarrow \emptyset^{k+1} \circ \langle \{\text{ValSend}(\pi.m, v, M(\pi.m)), \text{ValCreate}(v', \emptyset, \emptyset)\} \rangle \triangleleft v'} \quad (\text{E-SEND}) \\
 \\
 \frac{k \in \mathbb{N}, u \neq v}{\Gamma; (\{v, u\}), M \vdash \text{recv } \pi.m \rightsquigarrow \emptyset^k \circ \langle \{\text{ValRecv}(\pi.m, v, M(\pi.m)), \text{ValCreate}(u, \emptyset, \{v\})\} \rangle \triangleleft u} \quad (\text{E-RECV}) \\
 \\
 \frac{\Gamma; I_1, M \vdash t_1 \rightsquigarrow \mathcal{L}_1 \triangleleft v_1 \quad \Gamma; I_2, M \vdash t_2 \rightsquigarrow \mathcal{L}_2 \triangleleft v_2 \quad \Gamma; I_3, M \vdash t_3 \rightsquigarrow \mathcal{L}_3 \triangleleft v_3 \quad I_1 \cap (I_2 \cup I_3) = \emptyset \quad I_2 \cap I_3 = \emptyset}{\Gamma; (I_1 \cup I_2 \cup I_3), M \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightsquigarrow \mathcal{L}_1 \uplus \mathcal{L}_2 \uplus \langle \{\text{ValUse}(v_1)\} \rangle \triangleleft v_2} \quad (\text{E-IFTTHEN}) \\
 \\
 \frac{\Gamma; I_1, M \vdash t_1 \rightsquigarrow \mathcal{L}_1 \triangleleft v_1 \quad \Gamma; I_2, M \vdash t_2 \rightsquigarrow \mathcal{L}_2 \triangleleft v_2 \quad \Gamma; I_3, M \vdash t_3 \rightsquigarrow \mathcal{L}_3 \triangleleft v_3 \quad I_1 \cap (I_2 \cup I_3) = \emptyset \quad I_2 \cap I_3 = \emptyset}{\Gamma; (I_1 \cup I_2 \cup I_3), M \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightsquigarrow \mathcal{L}_1 \uplus \mathcal{L}_3 \uplus \langle \{\text{ValUse}(v_1)\} \rangle \triangleleft v_3} \quad (\text{E-IFELSE}) \\
 \\
 \frac{}{\emptyset; \{v\}, M \vdash *r \rightsquigarrow \langle \{\text{ValCreate}(v, \{r\}, \emptyset)\} \rangle \triangleleft v} \quad (\text{E-REG EVAL}) \\
 \\
 \frac{}{\emptyset; \{v\}, M \vdash \text{ready}(\pi.m) \rightsquigarrow \langle \{\text{ValCreate}(v, \emptyset, \emptyset)\} \rangle \triangleleft v} \quad (\text{E-READY})
 \end{array}$$

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

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

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

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 A.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 $\mathcal{L}_1 = \langle \alpha_0, \dots, \alpha_m \rangle$, $\mathcal{L}_2 = \langle \beta_0, \dots, \beta_m \rangle$, 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 $\text{ValSend}(\pi.m, v, p) \in \beta_i$.
 - $\mathcal{L} = \langle \gamma_0, \dots, \gamma_m \rangle$, $\gamma_i = \alpha_i \cup \beta_i - \{\text{ValSend}(\pi.m, v, p) \mid \pi.m \in \Sigma\} - \{\text{ValRecv}(\pi.m, v, p) \mid \pi.m \in \Sigma\}$.

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

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

$$\frac{(\text{loop}\{t\}, M) \rightsquigarrow t'}{(\text{loop}\{t\}, M) \rightsquigarrow t' \Rightarrow t} \quad (\text{C-EXTEND})$$

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

Definition A.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 .

A.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 $e_1 \rightarrow e_2 \in G$ to say that G contains an edge from event e_1 to event e_2 . We use $G(e_2)$ to denote $\langle \omega, \{e_1 \mid e_1 \rightarrow e_2 \in G\} \rangle$, 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 A. 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 $<_G$.

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

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

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

$$\begin{array}{c}
 \frac{M(\pi.m) = p \quad G(e_l) = \langle \pi.m, \{e_c\} \rangle}{\emptyset; G, R, M, \emptyset, e_c \vdash \text{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 \text{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-REG EVAL}) \\
 \\
 \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) = \emptyset \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 \text{if } t_1 \text{ then } t_2 \text{ 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 \text{ready}(\pi.m) : (e_c, \{e_c \triangleright \#1\})} \quad (\text{T-READY})
 \end{array}$$

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

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

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

- $\mathcal{P} = (\text{loop}\{t\}, M)$ and $\text{loop}\{t\}$ 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 M s that appear in \mathcal{P}_i .

A.3.1 Auxiliary Definitions. We define \leq_G and $<_G$ that appear in the typing rules.

Definition A.9 (Timestamp). A function $\tau_G : V \rightarrow \mathbb{N}$ 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 $\forall e' \in V : (e' \neq e \wedge G(e) = \langle \&c, S \rangle) \rightarrow \tau_G(e') = \infty$
- 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 A.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 A.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 A.12. If $(e_1 \rightarrow e_2) \in G$, then $e_1 \leq_G e_2$.

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

A.4 Safety

Definition A.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:

$$\begin{array}{c}
 \frac{}{\langle \rangle \vdash v \downarrow \perp} \text{ (R-BASE)} \\
 \\
 \frac{\mathcal{L} \vdash v \downarrow D}{\mathcal{L} \cdot \langle \emptyset \rangle \vdash v \downarrow D} \text{ (R-EMPTY)} \\
 \\
 \frac{\mathcal{L} \cdot \langle \alpha_i \rangle \vdash v \downarrow D \quad o \notin \{\text{ValCreate}(v, S_r, S_v) \mid S_r \in 2^{\text{RegId}}, S_v \in 2^{\text{ValId}}\}}{\mathcal{L} \cdot \langle \alpha_i \cup \{o\} \rangle \vdash v \downarrow D} \text{ (R-NONCREATE)} \\
 \\
 \frac{\mathcal{L} \cdot \langle \alpha_i \rangle \vdash v_1 \downarrow D_1 \quad D_1 \neq \perp \quad \vdots \quad \mathcal{L} \cdot \langle \alpha_i \rangle \vdash v_k \downarrow D_k \quad D_k \neq \perp}{\mathcal{L} \cdot \langle \alpha_i \cup \{\text{ValCreate}(v, S_r, \{v_1, \dots, v_k\})\} \rangle \vdash v \downarrow S_r \cup D_1 \cup \dots \cup D_k} \text{ (R-CREATE)}
 \end{array}$$

Note: \cdot is the normal concatenation operator.

Other auxiliary definitions, assuming $\mathcal{L} = \langle \alpha_0, \alpha_1, \dots, \alpha_k \rangle$,

- $\text{UseSet}(\mathcal{L}, v) = \{i \mid \text{ValUse}(v) \in \alpha_i \vee \text{ValCreate}(v, S_r, S_v) \in \alpha_i \vee \text{ValRecv}(\pi.m, v, p) \in \alpha_i \vee \text{ValSend}(\pi.m, v, p) \in \alpha_i\}$
- $\text{MutSet}(\mathcal{L}, D) = \{i \mid r \in D \wedge \text{RegMut}(r) \in \alpha_i\}$
- $\text{LtRecv}(\mathcal{L}, v) = \bigcap_{u \in \text{DepSet}(\mathcal{L}, v), \text{ValRecv}(\pi.m, u, p) \in \alpha_i} \text{LtFun}(\mathcal{L}, i, p)$
- $\text{LtSend}(\mathcal{L}, v) = \bigcup_{u \in \text{DeriveSet}(\mathcal{L}, v), \text{ValSend}(\pi.m, u, p) \in \alpha_i} \text{LtFun}(\mathcal{L}, i, p)$
- $\text{LtFun}(\mathcal{L}, i, \pi.m) = [i, w]$ 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 A.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 A.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 B of the Appendix.

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

Lemma A.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 A.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 $\text{loop}\{t\}$ is sufficient to guarantee that any number of ts joined together by $\text{wait}(\Rightarrow)$ is also well-typed.

Lemma A.19 (Two iterations are sufficient). Let t be an Anvil term and $t_k, k = 1, 2, \dots$ 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 $k = 2, \dots$.

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

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

B Proofs

B.1 Additional Lemmas

Lemma B.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 $\mathcal{L} = \langle \alpha_0, \dots, \alpha_k \rangle$ 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 $\Gamma'; I', M \vdash t' \rightsquigarrow \mathcal{L}' \triangleleft v$ appears during inference of $\emptyset; I, M \vdash t \rightsquigarrow \mathcal{L} \triangleleft v_0$, let $\mathcal{L}' = \langle \alpha'_0, \dots, \alpha'_l \rangle$, then $\forall 0 \leq i \leq l : \alpha'_i \subseteq \alpha_{i+\tau_G(e_c)}$ 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 $e \leq_G e_l$ and $S_d \leq_G S$.

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 $\Gamma'; I', M \vdash t' \rightsquigarrow \mathcal{L}' \triangleleft v$ appears during inference of $\emptyset; I, M \vdash t \rightsquigarrow \mathcal{L} \triangleleft v_0$, we show that $\tau_G(e_c) + l = \tau_G(e_l)$ is consistent with the timestamp function definition. In addition, we show $\forall (x : (k, v')) \in \Gamma' : \Gamma(x) = (e'_l, S'_d) \rightarrow k = \max(0, \tau_G(e'_l) - \tau_G(e_c))$. 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: $\tau_G(e_c) + l_1 = \tau_G(e'_l), \tau_G(e'_l) + l_2 = \tau_G(e_l), l = l_1 + l_2$.
- 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 , $\forall 0 \leq i \leq l : \alpha'_i \subseteq \alpha_{i+\tau_G(e_c)}$, where $\mathcal{L}' = \langle \alpha'_0, \dots, \alpha'_l \rangle$. 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 $e \leq_G e_l$ and $S_d \leq_G S$. \square

B.2 Lemma A.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 $\mathcal{L} = \langle \alpha_0, \dots, \alpha_k \rangle$, $\emptyset; I, M \vdash t \rightsquigarrow \mathcal{L} \triangleleft v_0$, the timestamp function in Lemma B.1 satisfies that for every value v , if $\Gamma'; I', M \vdash t' \rightsquigarrow \mathcal{L}' \triangleleft v$ appears during inference of $\emptyset; I, M \vdash t \rightsquigarrow \mathcal{L} \triangleleft v_0$, 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 $a = \tau_G(e_l)$, $b = \tau_G(\min_{e \triangleright p \in S_d, \tau_G(e \triangleright p)})$, 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 B.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 B.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. \square

B.3 Lemma A.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 $\text{UseSet}(\mathcal{L}_1, v) \cup \text{LtSend}(\mathcal{L}_1, v) \subseteq [a_1, b_1] \subseteq \text{LtRecv}(\mathcal{L}_1, v)$, $\mathcal{L}_1 \vdash v \downarrow D_1$, $\text{MutSet}(\mathcal{L}_1, D_1) \cap [a_1, b_1] = \emptyset$, and $\text{UseSet}(\mathcal{L}_2, v) \cup \text{LtSend}(\mathcal{L}_2, v) \subseteq [a_2, b_2] \subseteq \text{LtRecv}(\mathcal{L}_2, v)$, $\mathcal{L}_2 \vdash v \downarrow D_2$, $\text{MutSet}(\mathcal{L}_2, D_2) \cap [a_2, b_2] = \emptyset$.

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 $\text{MutSet}(\mathcal{L}_j, D'_j) \cap [a', b'] = \emptyset$, 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 $\text{MutSet}(\mathcal{L}_1, D_0) \cap [a_0, b_0] = \emptyset$. 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 $[a, b] \cap \text{MutSet}(\mathcal{L}, v) \subseteq \bigcap_{u \in S_v} [a'_u, b'_u] \cap \bigcup_{u \in S_v} \text{MutSet}(\mathcal{L}, u) = \emptyset$.

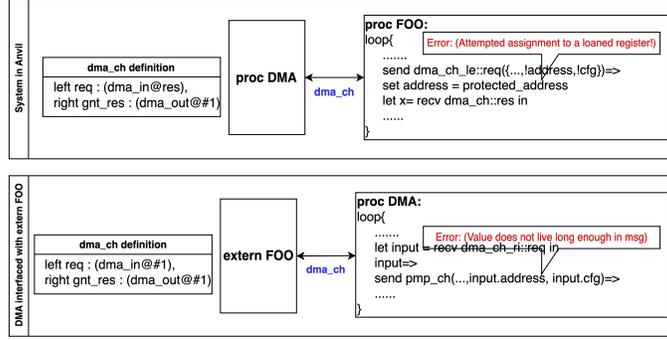


Figure 10. Anvil can assist in picking up bugs.

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 value v , there exists such a and b . Therefore, the composition $t_1 \parallel_{\Sigma} t_2$ is safe. □

B.4 Lemma A.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, $t_k (k = 2, \dots)$ are all well-typed.

Since t_k is well-typed, we have $\emptyset; G, R, M, \emptyset, e_0 \vdash t_k : T$. Because $t_k = t_{k-1} \Rightarrow t$, 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 $V_3 \cup \{e_2\}$. 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. □

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 [28] 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 10. 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

Table 3. Summary of Issues in some open source repositories

Repository	Issue Analysis	How can Anvil help?
OpenTitan (Issue [26])	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 implementation ensuring synchronization is built-in.
Coyote (Issue [17])	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 synchronization primitives to control the state and ensure an error-free FSM implementation.
ibex (Commit [25])	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 [24])	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 implementation for interfacing signals, ensuring the enforcement of timing contracts.
core2axi (Commit [33])	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

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 [8], 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 3, 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 the documentation for CV-X-IF, where one issue [32] reveals the complications involved in adhering to the timing contract. Another issue [31] 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.