Sequential Specifications for Precise Hardware Exceptions

Yulun Yao
Cornell University
Ithaca, NY, USA
yy665@cornell.edu

Drew Zagieboylo
NVIDIA
Westford, MA, USA
dzagieboylo@nvidia.com

Andrew C. Myers
Cornell University
Ithaca, NY, USA
andru@cs.cornell.edu

G. Edward Suh
Cornell University / NVIDIA
Ithaca, NY, USA
suh@ece.cornell.edu

Abstract

Modern processors are difficult to implement because pipelining makes them inherently parallel. A promising new approach, demonstrated in the PDL hardware description language, is to compile a high-level sequential specification into an efficient pipelined implementation. This high-level approach makes design-space exploration and reasoning easier. However, previous work on this approach does not support features needed for operating systems: hardware exceptions like traps and interrupts. The inherently non-sequential nature of these features makes it challenging to give them a sequential specification. They often require flushing the pipeline, writing to control state registers (CSRs), or resetting pipeline state.

In this work, we develop XPDL, which extends PDL to support hardware exceptions. With this extension, logic for precise exceptions can be synthesized from a high-level specification, while maintaining the appealing properties of PDL. Using RISC-V processor designs, we demonstrate that XPDL flexibly supports exceptions with no impact on CPI (Cycles per Instructions), and minor overhead over frequency and area, and argue that the implementation preserves the one-instruction-at-a-time semantics of PDL.

CCS Concepts: • **Hardware → Hardware description languages and compilation**.

Keywords: Pipelining, Interrupts, Exception Handling

ACM Reference Format:
Yulun Yao, Drew Zagieboylo, Andrew C. Myers, and G. Edward Suh. 2026. Sequential Specifications for Precise Hardware Exceptions. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3760250.3762233

ASPLOS '26, March 22–26, 2026, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2165-6/26/03.
https://doi.org/10.1145/3760250.3762233

1 Introduction

Traditional hardware description languages (HDL) capture the concurrent, register-transfer level (RTL) behavior of hardware. They are flexible enough to express high-performance designs but are difficult to reason about when building complex pipelined processors. Pipelined processors are inherently parallel because in a given cycle they execute multiple instructions. However, the Instruction Set Architecture (ISA) specifications that they implement are mostly sequential. Previous work, PDL [47], bridges the gap between parallel microarchitecture and sequential architecture with a high-level, sequential hardware description language (HDL) that targets processor design. PDL provides assurance that the behaviors of the pipeline are consistent with instructions running sequentially, one at a time. This *one-instruction-at-a-time* (OIAT) semantics makes PDL easy to write and reason about. PDL generates processors whose performance is comparable with RTL implementations.

While PDL supports sequential specifications, ISAs are not fully sequential [34, 44]. Non-sequential behaviors include hardware exceptions like interrupts, traps, faults and aborts. PDL needs support for these features in order to design processors usable by operating systems. But these features interact with the pipeline in complex ways: flushing the pipeline, writing to Control and Status Registers (CSRs), rolling back transient effects, and resetting pipeline states. These features can only be implemented with PDL in an awkward and inefficient way.

To address this limitation, we propose XPDL, an extension of PDL that introduces *pipeline exceptions* as a new language feature. Pipeline exceptions can describe final, atomic, and exception-handling operations of a pipeline. Pipeline exceptions are an HDL-level abstraction used to implement a variety of hardware exceptions: traps, interrupts, etc. The

syntax of pipeline exceptions is lightweight, similar to a familiar software-level `try–throw–catch` exception handling mechanism. This language feature is novel in the hardware context; in particular, languages used for high-level synthesis usually do not care about complex control paths [8], and low-level languages cannot reason about high-level behaviors like hardware exceptions [6].

Because the compiler preserves OIAT semantics through static checking, it is easy for programmers to implement hardware exceptions that are *precise* [38]. This is an important property that modern processors aim to satisfy, since precise exceptions ensure the processor remains in a well-defined state, able to safely resume execution after an exception is handled.

We have implemented XPDL on top of the open-source PDL compiler [48], and evaluated performance of the resulting circuits. XPDL generates efficient circuits for a wide range of extra hardware exception logic. Pipelines with hardware exceptions generated by XPDL have the same number of cycles per instruction (CPI), less than 3.3% decrease in maximum frequency, and reasonable overhead in area.

In summary, this paper makes the following contributions:

- XPDL, the first high-level sequential HDL with a safe, expressive abstraction for implementing hardware exceptions.
- Static-checking rules that preserve OIAT semantics and help implement precise exceptions.
- An implementation of pipeline exceptions as a synthesizable translation.
- An evaluation of the performance and programmability of XPDL, on several variations of a five-stage RISC-V processor.

2 Background

2.1 PDL Language Overview

To set the context, we first illustrate the PDL base language with an example. Figure 1 depicts an abbreviated 5-stage pipelined processor. Type information, decoding, and ALU logic are omitted for brevity. The pipeline is written like a function, named `cpu`, with the program counter (`pc`) as the argument. The register file (`rf`), instruction memory (`imem`), and data memory (`dmem`) are memories implemented by connected modules. Code in this module describes the functionality of a processor imperatively as a sequence of stages: instruction fetch and decode, ALU operations, memory operations, and write-back. Triple dashes (`---`) are stage separators that abstract pipeline registers; statements within a stage are combinational. Instructions traverse all ordered stages in issue order. The recursive `call` statement spawns a new instruction, with the provided argument as the next `pc`. Spawning the next instruction in an early stage creates instruction-level parallelism because execution of the next instruction begins immediately.

```
1  pipe cpu(pc)[rf, imem, dmem]{
2    spec_check();
3    acquire(imem[pc], R);
4    insn <- imem[pc]; //non-blocking assignment
5    release(imem[pc]);
6    --- // InsnFetch -[Reg]-> Decode
7    spec_check();
8    s <- spec_call cpu(pc+1);
9    //Some decode logic here
10   acquire(rf[rs1], R); //rs2 omitted
11   alu_arg1 = rf[rs1];
12   release(rf[rs1]);
13   reserve(rf[rd], W);
14   --- // Decode -[Reg]-> Execute
15   spec_barrier();
16   alu_out = alu(args*); //compute logic
17   npc = calc_npc(args*); //branch target
18   if (npc == pc+1){verify(s)}
19   else {invalidate(s); call cpu(npc);}
20   --- // Execute -[Reg]-> MemOps
21   acquire(dmem[alu_out]);
22   if (isStore(op)) {dmem[alu_out] <- data;}
23   if (isLoad(op)) {dmem_out <- dmem[alu_out];}
24   else {dmem_out = alu_out;}
25   release(dmem[alu_out]);
26   --- // MemOps -[Reg]-> WriteBack
27   block(rf[rd]);
28   rf[rd] <- dmem_out;
29   release(rf[rd]);
30 }
```

Figure 1. 5-stage CPU in PDL

PDL uses *pipeline locks* (blue syntax) to govern shared resources and prevent hazards. The `block` and `release` operations define critical sections where only the instruction owning the lock can enter, or else stalls until it is safe to read or write. The order of lock ownership is set by `reserve` operations; in-order reservations preserve the illusion of in-order execution and avoid data hazards. An `acquire` is syntactic sugar for `reserve` and `block` within a single cycle. A `release` operation relinquishes lock ownership and, for write locks, commits pending writes.

Speculative execution can be managed through the *speculation API* (brown syntax). The `spec_call` at line 8 spawns a speculative instruction running the next (`pc+1`) instruction—essentially a next-line predictor. It returns a handle referencing the new speculative instruction. The current instruction can later `verify` or `invalidate` it, depending on whether the prediction was correct. At line 18, a verified instruction becomes non-speculative, whereas an invalidated instruction is killed and a new non-speculative instruction issued using `call`. Operations `spec_check` and `spec_barrier` ask the current instruction to check its speculative state and to terminate on misspeculation. Instructions entering stages

following `spec_barrier` must be non-speculative. PDL automatically generates hardware tables to track speculation.

Through its APIs for pipeline locks and speculation, PDL gives the programmer fine-grained control over pipeline microarchitecture. This programming model facilitates design-space exploration over different pipelined implementations of the same ISA.

2.2 Hardware Exceptions

Operating systems require processors to implement a set of non-sequential behaviors specified in ISAs. These behaviors require the hardware to flush and halt the pipeline, roll back uncommitted effects, and reset the pipeline state. They differ in how they are raised, how they are processed and resolved, and whether they occur synchronously or asynchronously. While these behaviors have diverse semantics, they share a common trait—they disrupt the normal instruction execution flow and are both atomic and final. We refer to these behaviors as hardware exceptions.

The literature uses inconsistent terminology regarding exceptions; Table 1 clarifies our terminology. While support for hardware exceptions varies across different ISAs [1, 17], we adopt and modify the taxonomy from early work in precise interrupts [26], which best distinguishes different types of exceptions.

Badly implemented hardware exceptions can introduce security vulnerabilities: Meltdown [22] is one notorious example. It exploits speculative execution and exception handling to leak privileged memory that should be inaccessible to user-space programs. The attack relies on speculative execution temporarily bypassing access checks, allowing secret data to be extracted via cache side channels. Microarchitectural replay attacks also leverage pipeline flushes from exceptions [37]. These attacks highlight the complexity of hardware exceptions and the need for a clear definition of correctness.

2.3 Precise Hardware Exceptions

Operating systems require *precise exceptions* to be functionally correct [46]. Per Smith et al. [38], a precise exception maintains a state corresponding to a sequential model of execution. Processors implement precise exceptions to enable restarting and recovery of earlier pipeline states, ease of software debugging, and safe virtual memory. With precise exceptions, an exception is considered to occur during the execution of a single instruction, which we call the *exceptional instruction*.

Precise exceptions satisfy three conditions:

1. All instructions preceding the exceptional instruction execute and correctly modify architectural state.
2. Instructions after the exceptional instruction are unexecuted and have no effect on the architectural state.
3. An exceptional instruction behaves atomically—it either completes fully or has not started execution.

A precise exception effectively causes instructions to be inserted into a sequence of regular program instructions, but execution still behaves sequentially with respect to this expanded stream of instructions.

2.4 Motivation

Hardware exceptions are a fundamental, security-critical processor feature, requiring dedicated sections in ISA specifications [4, 36], and even for high-level languages [25]. However, implementing precise hardware exceptions is challenging, and better methods are needed. Traditional implementations of precise exceptions require microarchitectural structures to track the exceptional state of instructions—including reorder buffers, history buffers, future register files, and checkpointing [13, 38]. Their typical RTL implementation is deeply integrated with other control logic, making it more challenging to reason about processor correctness and security.

Traditionally, implementing precise hardware exceptions relies on verification, treating hardware exceptions like other pipeline behaviors. Commonly, behavioral specifications are extracted directly from the RTL implementation, but this method is both costly and error-prone [23]. Model-checking and formal verification frameworks struggle to scale to large designs, demanding significant human expertise [7].

An alternative approach is to raise the level of abstraction with correct-by-design languages. However, it is currently either impossible or awkward to describe hardware exceptions in those languages. For example, while speculative loop pipelining (SLP) [10] exposes commit and rollback to the language level, it serves as an optimization technique that enables speculation in HLS compilation but does not address how to implement exceptional behavior efficiently.

PDL has a similar problem: the only way it can describe non-sequential behavior is using speculation—every instruction could speculate on the absence of exceptions, with exceptions causing misspeculation.

However, this approach would be inefficient because exceptions and speculation have very different area–time performance tradeoffs. PDL handles misspeculations in one cycle but requires area-intensive hardware records for every speculative instruction executing in parallel. While misspeculations are frequent, exceptions are relatively uncommon in most workloads. Exceptions, unlike mispredictions, do not demand low-latency handling in most systems, as their impact on common-case performance is minimal.

Note that we do not make any quantified assumptions about exception frequency; even in syscall-heavy scenarios, the actual exception-triggered behavior is limited to syscall entry and exit, which is minimal compared to the duration of the system call itself. Most of the actual work is performed by software handlers running as regular instructions, without

Category	Characteristics	Example
Aborts and Faults	Synchronous; arise from unexpected instruction behavior. Faults are recoverable (instruction can be retried), whereas aborts are severe and non-recoverable.	*Page fault*: Occurs when a program accesses a memory page not currently mapped in RAM. The OS loads the missing page and retries the instruction.
Traps and System Instructions	Synchronous; raised by specific instructions as expected behavior. Used for control or debugging.	*System call instruction*: The instruction initiates a transfer of control from user space to kernel space.
Interrupts	Asynchronous; triggered by external hardware devices, such as I/O signals, which are independent of instruction execution.	*Keyboard interrupt*: The OS executes an interrupt handler before resuming the original program.

Table 1. Categories of Hardware Exceptions

requiring any special architectural mechanisms. As a result, timing performance for exceptions is generally less critical.

It is awkward—and sometimes impossible—to express the control behavior that exceptions require using only the speculation mechanism. Exceptions trigger diverse behaviors whose description demands an expressive language, whereas speculation is a microarchitectural optimization that should not affect ISA-visible behavior. Mixing exception-handling logic with common-case code in the pipeline body violates separation of concerns.

Modern microarchitectures already treat exceptions and mispredictions differently for these reasons. It motivates adding a separate language-level construct that can express exceptions. Furthermore, integrating hardware exception support into PDL remains promising because its OIAT semantics aid in making exceptions precise. And hardware exceptions enhance PDL's expressiveness, making it more applicable to real-world hardware designs.

```
1  pipe cpu(pc)[rf, imem, dmem, csr]{
2    // InsnFetch Stg
3  ---// InsnFetch (IF) -> Decode (DE)
4    // Decode Stg
5    if (isInvalid(insn)) {throw(ERR_INV);}
6  --- //Decode (DE) -> Execute (EX)
7    // Execute Stg
8  --- //Execute (EX) -> MemOps (MM)
9    // MemOps Stg
10 --- //MemOps (MM) -> WriteBack (WB)
11   block(rf[rd]);
12   rf[rd] <- rd_data;
13 commit: // (CM) if no exception
14   release(rf[rd]);
15   release(dmem[alu_out]);
16 except(error_code): // (RB) if exception
17   // EX1: Save exception info to CSRs
18 --- //ExceptStg1 (EX1) -> ExceptStg2 (EX2)
19   call cpu(handler_pc); // Call handler
20 }
```

Figure 2. Invalid Instruction Example in xPDL

3 Approach

3.1 Overview

Pipeline exceptions are an intuitive extension to the PDL language to support logic for hardware exceptions. They could also be applied to other sequential HDL abstractions such as HLS.

A pipeline exception disrupts the normal execution flow of an instruction. When an exception is raised, the exceptional instruction is replaced with a special pipeline bubble containing information related to the exception. When the bubble reaches the end of the pipeline, preceding instructions have already been committed, and the exceptional instruction is passed to a hardware exception handler. The pipeline is flushed and stopped before this handler executes, and subsequent instructions leave no effect on hardware state. If no exceptions are raised, instructions execute and commit normally, as if no exceptions are implemented.

Figure 2 shows an XPDL program that extends the processor implementation from Figure 1 to support a simple illegal-instruction exception. Figure 3 shows the datapath of the corresponding processor. Conceptually, an XPDL program comprises two pipelines: a main pipeline that executes committing instructions, and a separate pipeline that only processes exceptional instructions.

The decode (DE) stage has a new `throw` statement (line 6) with the appropriate error code as its argument. Executing `throw` marks the current instruction as exceptional. This information, along with the error code, is passed along the

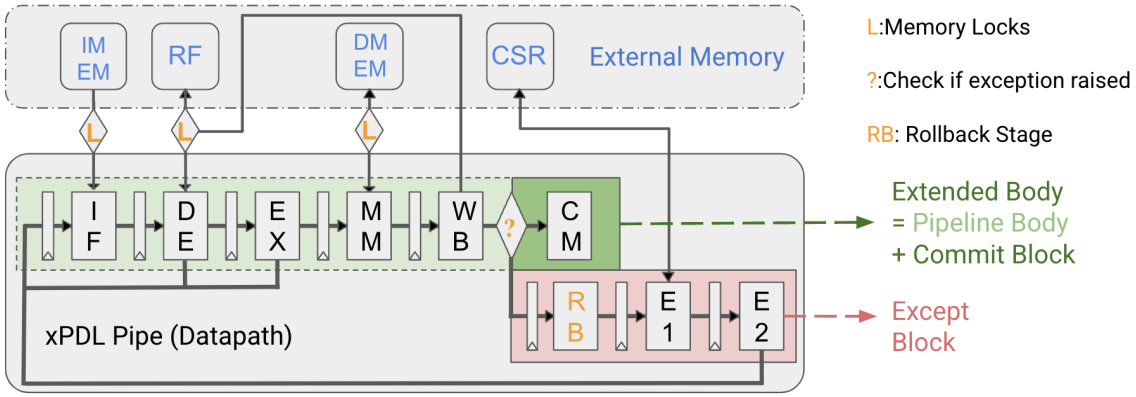

Figure 3. Data path for 5-stage CPU with pipeline exceptions

datapath. At the end of the pipeline body (light green), two code blocks are appended: the *commit block* (dark green) and the *except block* (red). The commit block contains logic for instruction commit, while the except block handles cases where an exception is raised. We call the commit and except blocks *final blocks* because all their effects are final: they cannot be squashed or rolled back. In this example, the commit block contains `release` statements to both write locks, effectively committing the writes, whereas the except block saves information about the exception into the CSRs and spawns a new instruction redirecting the pc address to the software exception handler. Before entering the except block, the pipeline is flushed, and an implicit *rollback* (RB) stage removes all uncommitted changes to architectural state and releases all locks.

3.2 Language

The syntax for pipeline exceptions is lightweight: the programmer only needs to specify 1) a commit block containing a sequence of committing operations, 2) an except block containing all exception handling operations and a signature that defines exception arguments, and 3) a `throw` statement with arguments matching the except-block signature. The `throw` statement does not check the condition of raising an exception, as PDL's base language is expressive enough to describe the condition, including exception priorities. In addition to exception arguments, pipeline arguments and references to module connections are in the environment of the except block, as they are often used for exception handling. Transient pipeline states, except those passed to exception arguments, are not part of the environment.

An XPDL pipeline can have only one commit block and one except block. This facilitates the reuse of rollback logic but does not reduce the expressiveness of the language, as conditionals can be used over different exception argument values. For expressiveness, the pipeline body, the commit block, and the except block can each have an arbitrary number of stages. We refer to the concatenation of the pipeline body and the commit block as the *extended body*. It describes the execution of non-exceptional instructions. Note that stage separators are not required between the pipeline body and the commit block; the first stage of the commit block can be merged into the last stage of the pipeline body, so no microarchitectural change is needed for non-exceptional instructions.

3.3 Synthesis

In XPDL, the synthesis of a pipeline that never uses exceptions is the same as for base PDL; for a pipeline using exceptions, new language syntax is translated into an extended PDL syntax that adds some new internal constructs generated and used only by the compiler. The extended internal syntax is inaccessible to the programmer, because it is dangerous: using it in the pipeline body could lead to inconsistent hardware states.

To represent the exceptional state of instructions and the pipeline at the PDL level, the compiler uses two new expressions: 1) an instruction-specific local exception flag *lef* that marks whether the current instruction is exceptional, and 2) a module-level exception flag *gef* that indicates whether the pipeline is currently in exception-handling mode. Further internal operations are added for flushing the pipeline and rolling back pipeline state: 1) *abort* resets lock states, revoking lock ownership and clearing all uncommitted writes, 2) *pipeclear* clears all pipeline (stage) registers for all stages in the pipeline body; 3) *specclear* resets records for speculative instructions. The semantics of these operations differ from `invalidate` in that they reset the pipeline back to a clean state rather than undo effects of some instructions.

Figure 4 describes our translation strategy formally. A translation rule takes the form $\mathcal{S}[\![a]\!] \triangleq b$. It translates any matching XPDL statement a into a new PDL statement b. $\mathcal{S}[\![\cdot]\!]$ is defined by structural induction on terms, so terms on the right-hand side may be further expanded. For brevity, some side conditions are explained informally in the text.

$$\mathcal{S}[\![c_h \texttt{ --- } c_t]\!] \triangleq \texttt{if } gef \texttt{ skip}$$
$$\texttt{else } c_h$$
$$\texttt{--- } \mathcal{S}[\![c_t]\!] \quad (c_h \textit{ has one stage})$$
$$\mathcal{S}[\![\texttt{commit: } c_c]\!] \triangleq c_c$$
$$\mathcal{S}[\![\texttt{except}(\overline{args})\texttt{:}c_e]\!] \triangleq gef \leftarrow \texttt{true;}$$
$$\texttt{--- skip... --- skip} \quad (n \textit{ times})$$
$$\texttt{--- pipeclear; specclear;}$$
$$\texttt{abort}(M_1)\texttt{; ...; abort}(M_n)\texttt{;}$$
$$\texttt{--- } c_e; gef \leftarrow \texttt{false}$$
$$\mathcal{S}[\![c_b, \texttt{commit:}c_c\texttt{, except}(\overline{args})\texttt{:}c_e]\!] \triangleq$$
$$\mathcal{S}[\![c_b]\!];$$
$$\texttt{if } lef\ \mathcal{S}[\![\texttt{except}(\overline{args})\texttt{:}c_e]\!]$$
$$\texttt{else } \mathcal{S}[\![\texttt{commit: } c_c]\!]$$
$$\mathcal{S}[\![\texttt{throw } (\overline{args})]\!] \triangleq lef \texttt{ = true;}$$
$$earg_1 \texttt{ = } args_1\texttt{; ...;}earg_n \texttt{ = } args_n$$

Figure 4. Translation rules for new syntax

Figure 5 illustrates the translated pipeline for handling the invalid instruction exception in Figure 2, providing a concrete example.

```
1   //..Same as original program
2   //Decode Stg
3   if (isInvalid(insn)) {lef <- True;
4   earg0 <- ERR_INV;}
5   //..Same as original program
6   block(rf[rd]);
7   rf[rd] <- rd_data;
8   if(lef){
9     gef <- True;
10    ---
11    pipeclear; specclear;
12    abort(rf[rd]);
13    abort(dmem[alu_out]);
14    ---
15    // EX1: Save exception info to CSRs
16    ---
17    call cpu(handler_pc); // Call handler
18    gef <- False;
19  } else {
20    release(rf[rd]);
21    release(dmem[alu_out]);
22  }
```

Figure 5. Post Translation Pipeline

The compiler creates *gef* as a 1-bit shared register, and *lef* as a per-instruction boolean variable that is passed along the datapath, becoming one 1-bit register per stage.

In the first rule, the pipeline body is viewed as a sequence of stages, delimited by stage separators, where c_h is the first stage and c_t holds the remaining stages. For each stage in the pipeline body, an additional control path observes *gef* as illustrated in Figure 7. Setting *gef* causes later stages to do nothing. This translation strategy ensures that execution of the except block is mutually exclusive with execution of the main pipeline body.

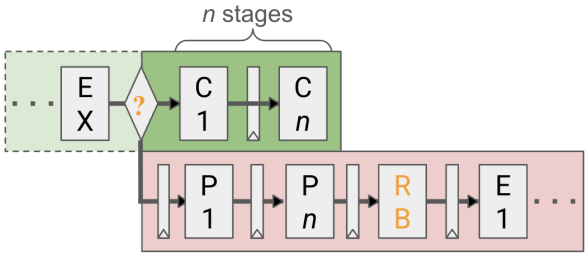

Figure 6. Padding stages translation

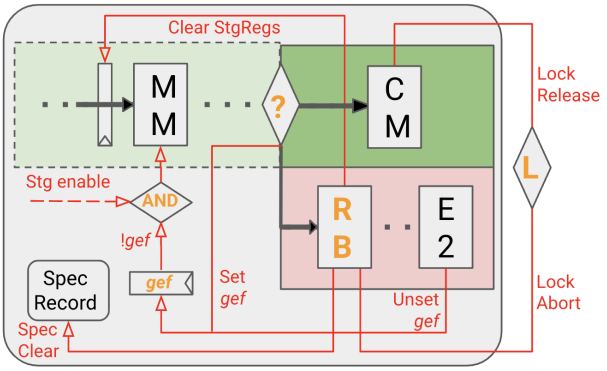

Figure 7. Control path diagram (new logic only)

As shown in Figure 3, final blocks are converted into a single conditional block, dependent on *lef*. If *lef* is not set, statements in the commit block are executed. Otherwise, *gef* is set to enter exception handling mode.

The compiler then generates n padding stages, where n equals the number of commit stages. Figure 6 shows the effect of padding stages: rollback is delayed so all previous committing instructions can finish. The command *skip* is a no-op that does not modify program state. If the commit block has no new stages—the commit block has only one stage and it is merged with last stage of pipeline body—no padding stages will be added. Padding stages are logical constructs−−a simple register guard can be used instead of generating actual pipeline stages.

In the rollback stage (RB), the pipeline is flushed: the XPDL compiler generates a *pipeclear* that captures all pipeline registers before the final blocks and clears them. The compiler also generates *abort* statements for every lock used in the pipeline ($M_i, i \in \mathbb{N}$) and a *specclear* statement that clears the speculative record. Figure 7 shows this extra control path. Following the RB stage, except block statements are executed.

At the end of the except block, *gef* is unset, and the pipeline returns to normal.

Stages in the final block always execute, provided the pipeline registers are non-empty.

The `throw` statements are translated into a set of assignments, where *lef* is set, and values of arguments passed into the `throw` statements are copied to the canonicalized except block arguments ($earg_i, i \in \mathbb{N}$).

3.4 Locks

Locks in PDL are abstractions for memory interfaces equipped with stalling and bypassing logic. They encapsulate pipeline states for different microarchitectural elements.

XPDL's translation strategy simplifies pipeline state management by leveraging locks. Rollback of pipeline states is managed through rollback of locks. Some PDL locks already provide checkpoint and rollback mechanisms for handling misspeculations. However, exceptions are relatively uncommon, whereas misspeculations are frequent. PDL handles misspeculations in one cycle but requires area-intensive hardware records for every speculative instruction executing in parallel. This difference in area–time tradeoff motivates a different rollback implementation.

To address this tradeoff, real processors use different mechanisms, depending upon the frequency of rollback. For instance, rename tables often have a snapshot for each speculative branch, but will use multi-cycle rollback to handle exception instructions [2, 35].

XPDL's new API, *abort*, provides a unified approach for different locks to rollback to a point where precise exceptions can be maintained. Calling *abort* on a lock logically resets any transient microarchitectural state for that lock.

We briefly describe how we extend locks to support *abort*:

Bypass Queue: A bypass queue provides in-order memory writes. Writes are cached and can be passed to later reads. In this case, aborting the state can be achieved by just removing the cached writes; later instructions are squashed and their reads do not affect correctness.

Renaming Register: To support rollback, we store a snapshot of the register mapping table and the free list of available physical registers at checkpoint creation. Since register files are typically small, this approach is feasible despite its higher storage cost. On rollback, we restore the previous register mapping and reclaim any physical registers allocated after the checkpoint, ensuring that previous instructions are committed correctly.

In contrast to the XPDL approach, complex processors often implement more centralized control circuits, relying on scoreboards or, in machines that uses Tomasulo's algorithm [42] reservation stations. Both mechanisms can be modified to add a small field to store exceptional state. Tomasulo's algorithm naturally handles non-committing writes in reservation stations; scoreboard machines rely on a range of mechanisms like write buffers or renaming registers. These approaches are more area-efficient because theye have centralized bookkeeping logic, while in the modular approach of XPDL, a small overhead is added because each memory module does its own bookkeeping.

3.5 Static Checking

All static analyses from base PDL are performed prior to the translation to ensure the correctness of XPDL programs. Most of these checks are generic, such as checking whether the value from a memory read is available at its use. For pipelines that include an except block, XPDL conducts two separate runs on most static analysis passes: one for the extended body and one for the except block. These static checks are exhaustive because all program paths lead to either commit or except. XPDL adds further static checks:

Rule 1: *The except block must be self-contained, so all pending architectural state changes are complete before an instruction exits the block.* This entails three requirements:

a) An instruction may `acquire` a write lock on memory and modify it, but must `release` the lock before exiting the except block.
b) Reads are forbidden from asynchronous memory or other pipelines at the end of the except block, to prevent indefinite stalls.
c) A recursive call statement (which spawns a new instruction in the main pipeline) can only be in the last stage of the except block, to maintain atomicity of the except block.

Rule 2: *Final blocks must be non-speculative and cannot spawn new speculative instructions.* Hence, instructions are never squashed in final blocks. Spawning speculative instructions in final blocks is unnecessary since an instruction has almost finished execution of an instruction and possesses full information for the next pc. Consequently, `spec_check`, `spec_barrier`, and `spec_call` statements are not allowed within the final blocks.

Rule 3: *Write locks acquired in the pipeline body must be released within the commit block and not before.* This rule prevents changes to memory states in the main body. As a result, undecided instructions will not produce any effects, and the order in which instructions take effect aligns with their entry into the final block.

Rule 4: *No stateful operations are allowed in the commit block, with the exception of releasing a lock.* Stateful operations include spawning new instructions, acquiring locks, and speculation-related operations. Combining this rule with our translation strategy gives us a safe *happens-before* [21] relationship between an exceptional instruction and preceding committing instructions: the effects of those instructions must happen before the effects of the exceptional instruction.

XPDL's static checks restrict how pipelines can be structured, which can make certain designs either impossible or

less efficient. These constraints have several implications for processor design:

a) Handling exception while running other instructions is impossible. This is a limiting factor to our processor design; there is latency to interrupt handling because the pipeline must be cleaned up first.
b) Only one exception can be processed at a time. This limits the throughput of control instructions to a maximum of one per pipeline completion (i.e., 1 / number of stages), which could bottleneck workloads with frequent exceptions.
c) Communication between exceptional and non-exceptional instructions is limited. The only communication channel is via architectural state such as datapath memories or registers. Control and status registers (CSRs), for instance, must be modeled as ordinary visible registers—unlike conventional processors which often use separate control paths or microarchitectural signaling.
d) Since there are no side-effects such as resource contention or residual microarchitectural state, exceptional instructions leave no visible trace, thus preventing Meltdown-style vulnerabilities.
e) Exception handling is strictly non-speculative. Misspeculative instructions cannot raise exceptions. In scenarios where exceptions must be handled with minimal latency (e.g., real-time systems), this design can introduce delays if misspeculation keeps happening.

3.6 Interrupts

```
1  volatile pending; //Interrupt pending signal
2  //Non-interrupt pipeline logics omitted
3    if(isIntEnabled && pending != NONE){
4      int_code = get_int_type(pending, mask);
5      //Decide whether and which int to handle
6      throw(int_code);}
7  commit: // No change to commit
8  except(error_code):
9    pending <- NONE;
10   //Unset to acknowledge interrupt
11   CSR[MCAUSE] <- error_code; //Save cause
12   //Logic for different exception types
13   ---
14   call cpu(handler_pc);
```

Figure 8. Interrupt example in xPDL

Figure 8 presents a simple example of interrupt handling in XPDL. The source of the interrupt is the pending register, which every instruction checks to determine whether a pending interrupt is waiting to be processed—provided interrupts are currently enabled. It then reads the mask register and applies precedence logic to decide which interrupt to handle. It then raises an exception through `throw` and resolves the interrupt with user-defined logic in the except block: unset the interrupt pending signal to acknowledge handling of this interrupt; save the error code and possibly other information; call an interrupt handler. XPDL is expressive enough to describe a variety of interrupt-handling logic.

Note that this pending register is marked with a new `volatile` type annotation. While the annotation is reminiscent of the volatile keyword in C, its semantics are not quite the same. There are some similarities: (1) there is at least one external writer whose behavior is unpredictable; (2) all readers must observe the most recent value, with changes becoming visible immediately after the write; if clocks are synchronized, this means the new value is observable in the next cycle. (3) a single read or write is atomic and final.

XPDL uses `volatile` on connected memories that may be modified by external devices, effectively describing device registers. Volatile memories are unique in that instructions in the pipeline cannot lock them. For volatile memories, XPDL limits the placement of reads only to non-speculative, in-order regions of the pipeline (including final blocks), and of writes only to final blocks. A given volatile memory can only be read or written by one instruction at a time. XPDL does not impose a limit on the number of volatile memories that can be connected to a pipeline, allowing programmers to incorporate multiple interrupt sources.

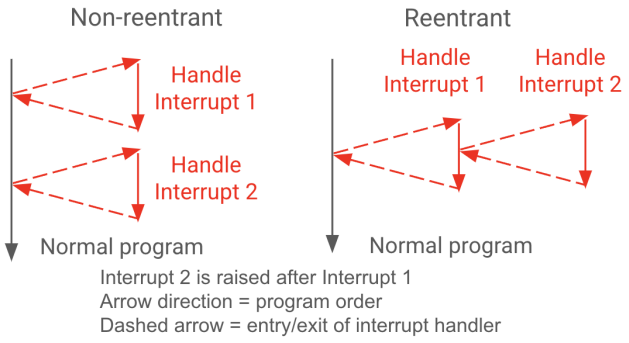

Figure 9. Non-reentrant and Reentrant interrupts

The above rules provide an ordering guarantee: *for any two instructions in program order, the later instruction will always read the more recent value from the volatile memories.* Hence, at any given time, if an instruction captures the current state of the interrupt signal, later instructions cannot observe older states. This guarantees correctness for both non-reentrant and reentrant interrupts (Figure 9) — non-reentrant interrupts are handled strictly in the order they were raised; for reentrant interrupts, the later interrupt can correctly preempt the earlier interrupt.

This ordering guarantee corresponds to sequential consistency [20]: both the XPDL pipeline and external devices observe memory updates in the same global order, and that order respects the program order of both the pipeline and the

external devices. This assumes that external devices access volatile memory in a sequential manner. Under this consistency model, when an instruction in an XPDL pipeline reads from a volatile memory, it observes the most recent committed write according to the global order. This behavior aligns with the semantics of precise exceptions, as instructions never see stale or out-of-order memory updates.

There is no strict requirement for interrupts to be resolved immediately, as they are inherently asynchronous. For example, a timer interrupt is bound to real-time clocks rather than hardware cycles. Additionally, many architectures allow interrupts to be delayed or masked [3, 5].

Since interrupts originate from device-specific hardware, their implementation falls outside the scope of XPDL. However, device-specific hardware, including interrupt controllers, can be implemented as RTL libraries and imported into XPDL as needed.

3.7 Propagating Exceptions

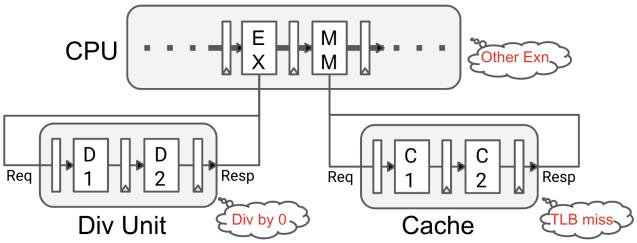

Figure 10. Multi-pipe with decentralized exceptions

An XPDL program describes a single core system, yet may contain multiple pipelines. Figure 10 illustrates a processor that includes a pipelined CPU, a pipelined cache, and a pipelined division unit. The pipelines must be structured as a tree hierarchy, where inter-pipeline calls are blocking—sub-pipelines act as service providers for their callers. Designing exceptions for such a program is an interesting problem because each pipeline can raise its own exceptions. For example, the division unit may raise a division by zero exception, the pipelined cache may trigger a TLB miss, and a cache miss itself can be considered a local exception for the cache.

In XPDL, each pipeline can have its own except block, and exceptions from different pipelines do not interact. This design strikes a balance by ensuring precise exceptions at the local level of each pipeline, while avoiding the overhead of maintaining unnecessary global consistency of exception states across the entire program.

Programmers have the flexibility to choose which exceptions to propagate, while still being able to easily maintain precise exceptions across pipeline boundaries when needed. If an exception can be resolved locally within a pipeline, then it should not be propagated to the parent of this pipeline. For instance: a cache miss can be resolved locally, but an access violation must be propagated to the CPU. In the latter scenario, programmers can explicitly propagate the exceptional state through data responses and raise exceptions in the CPU.

This design decision also benefits processor performance, as programmers may occasionally prefer imprecise exceptions—maintaining precise exceptions under high-latency events are both unnecessary and costly [12].

4 Evaluation

We implemented the XPDL language extension in approximately 2k lines of Scala [29], BlueSpec [28] and Verilog [41] code on top of PDL to extend the compiler and implement new hardware methods. Using software simulation [45], we tested the correctness of our approach using both small test cases, covering different pipeline microarchitectures, and large-scale tests that run real programs with system calls and interrupts on RISC-V processors.

Our evaluation aims to answer the following research questions:

RQ1: Does XPDL allow rich expression of hardware exceptions in an HDL with sequential semantics?

RQ2: Can XPDL support hardware exceptions with a reasonable area and performance overhead?

RQ3: Does XPDL preserve the ease of programming and reasoning of PDL?

4.1 RQ1: Expressiveness

To our best knowledge, XPDL is expressive enough to describe all behaviors specified in the RISC-V privileged ISA manual. This does not rule out XPDL's ability to describe other ISAs—pipeline exceptions are an ISA-agnostic language feature.

To evaluate the expressiveness of XPDL, we implemented several processors that implement various forms of exception handling on top of a baseline implementation. Our baseline is a 5-stage RV32IM processor with speculative execution, renaming registers, a write queue for data memory, and no CSRs. We implemented variants supporting 1) fatal exceptions (illegal instructions and memory accesses); 2) system calls and interrupts that use traps as a handling mechanism; and 3) CSR instructions supporting up to 32 CSRs. And we implemented a processor incorporating all of these exceptions.

We briefly outline how we extended the baseline with system calls and CSR instructions [44]. In the decode stage, an instruction throws an exception whenever it is a system call, a return (exit of a system call), or a CSR instruction. CSR instructions do affect subsequent instructions, but using locks to guard CSRs would be expensive and CSR instructions are rare. Hence, it is practical to implement them as exceptions. The except block supports each instruction according to specification: system calls and return instructions store

exception information to CSRs and redirect pc to a software exception handler or return address. CSR instructions are similar but read or write from data registers. To implement multiple privilege levels, we could store the current privilege level in a CSR, read it every cycle but only write it in the except block.

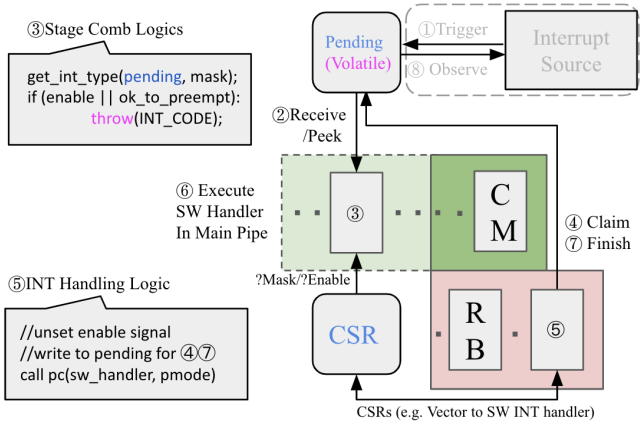

Figure 11. Interrupt handling overview

While there are many advanced hardware interrupt controllers with ever-growing features, interrupt handling follows a general process. Figure 11 generalizes the example from Figure 8, and sketches this interrupt handling process [16, 24, 43] and how programmers use XPDL to provide hardware support. Memory-mapped interrupt controllers are marked `volatile`. In the pipeline body, every instruction reads (interrupts) pending, mask, and enable registers processes it accordingly to specification, and throws an exception if the current pending interrupt can be handled. The remaining hardware implementation is similar to system-call handling—it redirects pc to a software exception handler. The only difference in the except block is disabling/enabling interrupts and writing the interrupt-pending CSR to claim or complete the interrupt.

Appending exception-handling logic to the end of the pipeline has a few implications. First, unless a storage unit only reads or writes at the end of the pipeline, it must be able to manage intermediate state, and support rollback and commit operations—e.g., support checkpointing on their states, or hold writes till commit points. Second, exceptional instructions may take longer to finish, potentially delaying CSR instructions and, more critically, interrupt handling. While many processors poll and handle interrupts at the beginning of the pipeline, this design is not feasible in XPDL. Third, logic currently cannot be shared between the main pipeline and the except block, nor implement *actual* non-local control flow in specific pipeline stages.

4.2 RQ2: Area and Performance Overhead

To evaluate the area overheads of the various XPDL-generated processors, we synthesize hardware [15] and place-and-route [14], with a 163.93 MHz clock and 45 nm FreePDK [39] technology. For performance overhead, we measured the CPI number, maximum frequency, and compilation time of these processors.

We break down the area usage of XPDL implementations to analyze where the overhead comes from and whether it can be mitigated: 1) Register Files and CSRs: the area cost is inevitable as we need to add more control registers to support hardware functionalities; 2) Pipeline registers: our compilation strategy may introduce extra pipeline registers—hand-written hardware can optimize resource usage by reusing existing pipeline registers for exception handling; 3) Combinational logic: cost for combinational logic is mostly intrinsic, as functional logic is required to process exceptions regardless how the hardware is implemented. Comparing against non-XPDL processors is not our goal, as we are evaluating our language exception rather than PDL. Moreover, each non-XPDL processor has a unique microarchitecture, with exception-handling logic deeply intertwined with the control path, making direct comparisons impractical.

Figure 12 presents the results. Each bar is divided into three sections with black lines, where the left section shows the area for Register Files and CSRs, the middle section adds up all stage registers and the right section includes all combinational logic. The numbers on each bar are cumulative: we first implement the first exception (① on the diagram for each group) to the baseline, along with its corresponding handling logic, then add additional exceptions on top (②,③). Within each group, the majority of area differences (up to 65% of the total) are in the area to support CSRs. The meaningful overhead comes from differences in stage registers as we introduce multiple new stages and new local variables, which are propagated through the data path. For CSR instructions, complex decoding logic increases the area by 10%. Nevertheless, there is plenty of logic reuse. For exceptions that share similar handling logic, the cost of supporting additional exceptions is almost negligible, and even for the combined example, the total area cost is still much less than the sum of the areas of each group.

We measured performance for the above examples on three different metrics:

Cycles Per Instructions (CPI): Processors that implement exceptions should not have a worse CPI than those that do not, as long as no exceptions occur during execution. We verify this through RTL simulation [45]. Our benchmark is MachSuite, which we use to generate binaries for RV32IM. Using the same external microarchitecture modules (bypassing and stalling logic, memory interface, and pipeline separator placements), all processors perform equally (1.59 CPI for MachSuite-aes [33]).

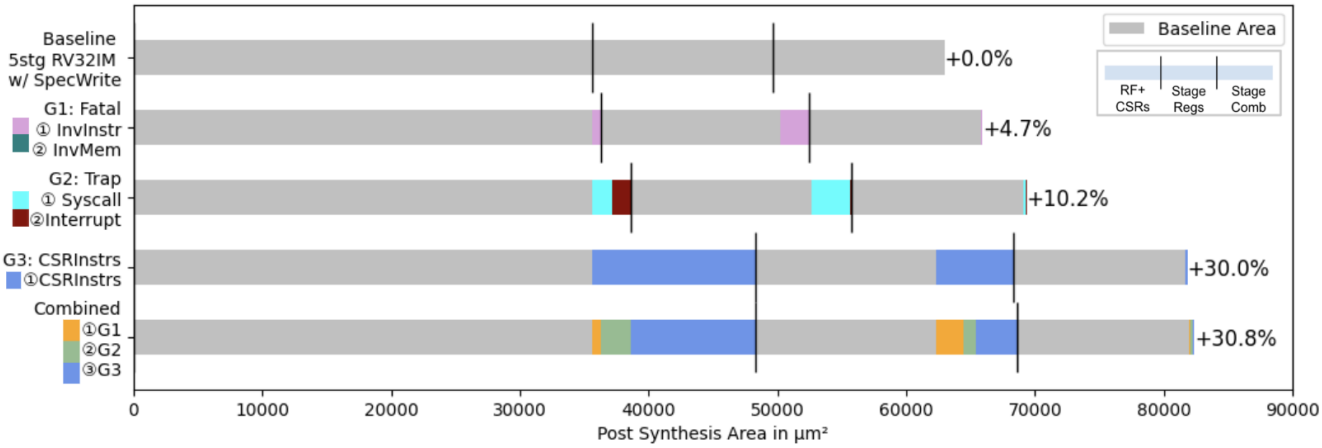

Figure 12. Area of processor implementations in μm^2

Maximum frequency: With the same technology [39] and same configuration for both synthesis and place-and-route, the maximum frequency at which our generated processors can operate reflects the quality of our design, as it is determined by the longest critical paths. Comparing our processor that implements all exceptions to the baseline processor, we observe 3.3% reduction in maximum frequency (169.49 MHz v.s. 163.93 MHz). All experiments used medium-effort synthesis settings, and there is room to further optimize the implementation. A quick test on a Xilinx FPGA also validates result (both close to 65.6 MHz) [11, 32].

Compilation time: To show that XPDL enables fast prototyping of processor pipelines, we measure the end-to-end time to compile an XPDL program down to Verilog on a standard Linux server. This compilation is a two-step process: 1) XPDL to Bluespec; 2) Bluespec to Verilog. For a baseline processor, XPDL compiles the design in 15.34 seconds, with 8.13 seconds for XPDL to Bluespec and 7.21 seconds for Bluespec to Verilog. For a processor that implements all exceptions, compilation takes 15.50 seconds, with 8.17 seconds for XPDL to Bluespec and 7.33 seconds for Bluespec to Verilog. Most time in the compiler is spent in type inference and type checking, which are needed to verify lock placement and correctness of speculation, relying on the Z3 SMT solver [9]. Overall, compilation is fast and supporting exceptions does not add significant overhead.

4.3 RQ3: Ease of Programming

Figure 13 quantifies programmer effort in terms of lines of code (LOC) [27]. This figure has the exact same experimental setup as Figure 12, and can be interpreted similarly. Starting from the left, three bar sections now show LOC in pipeline body and module definition, commit block, and except block, respectively. There are three takeaways: 1) the commit block remains the same for all exceptions implemented over the baseline processor; 2) there is substantial sharing of code in the except block for a group of exceptions with similar handling mechanisms. 3) even with all exceptions implemented, the whole processor takes less than 500 LOC. Note that many of these LOCs are added to describe CSR accesses that are not in the baseline.

Importantly, XPDL preserves OIAT semantics. It does not change the behaviors of a pipeline without pipeline exceptions. For a pipeline with exceptions, any sequence of non-exceptional instructions runs through the extended body, which behaves the same as a simpler pipeline in which all `throw`s are removed along with conditional checks based on exception flags. Therefore, any sequence of non-exceptional instructions adhere to the OIAT semantics enforced by PDL. For a sequence of instructions that contains one or more exceptions, our translation strategy (§3.3) and static-checking rules (§3.4) enforce the three conditions for precise exceptions:

1) *All instructions preceding the exceptional instruction execute and correctly modify architectural state.*
   A well-formed XPDL pipeline has a single final block placed at the end of the pipeline. Therefore, all instructions enter the final block in their issue order. Since instructions cannot be squashed in the final block (Rule 2), commits from instructions before the exceptional instruction must always finish. Additionally, all stateful operations are ordered before rollback happens (Rule 4), so an exceptional instruction cannot roll back the effect of preceding committing instructions.
2) *Instructions after the exceptional instruction are unexecuted and have no effect on the architectural state.*
   Upon entering the final blocks, an exceptional instruction sets the pipeline to exception handling mode, stalling instructions in the pipeline body and preventing succeeding instructions from entering the final block, where memory writes takes effect (Rule 3). In the next stage, it clears

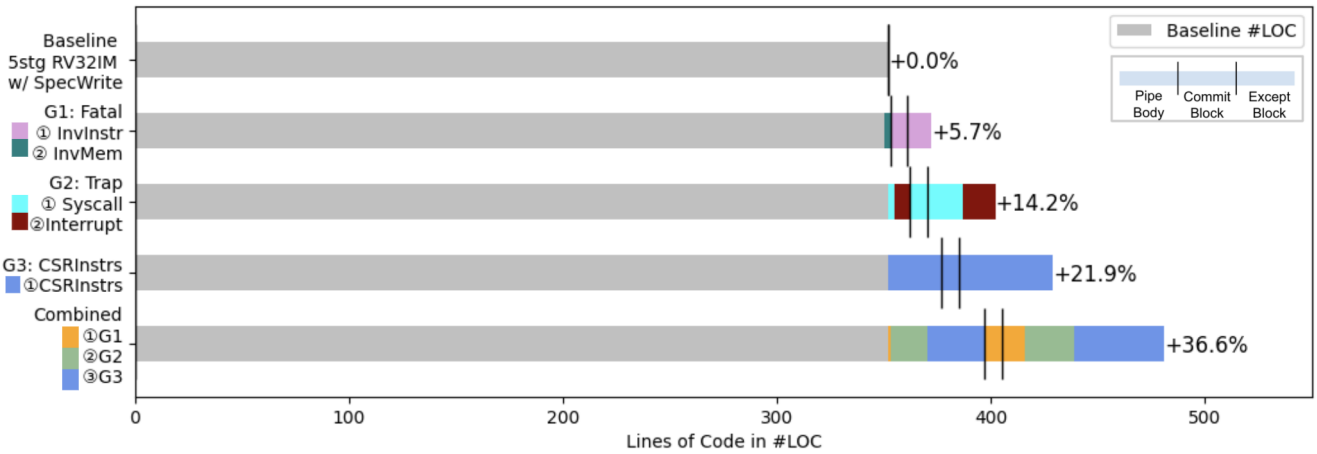

Figure 13. #LOC of xPDL for processor implementations

all uncommitted writes by aborting their locks and making succeeding instructions no-ops by clearing the stage registers.

3) *An exceptional instruction behaves atomically—it either completes fully or has not started execution.* Exceptional instructions are treated as unexecuted, as they cannot commit their changes. Further, they execute alone, and always finish the except block (Rule 1) before any new instructions can be spawned. Any such new instruction executes as if in a fresh pipeline.

Hence, exceptional instructions compose with rest of the instructions while preserving OIAT semantics, which leads naturally to correct implementations of precise exceptions.

5 Related Work

High-level synthesis (HLS) [8] tools focus on modeling algorithms and pipeline functionality. They are easy to write and reason about, but they are insufficient to describe complex control paths. Kanagawa shares a similar goal with PDL and XPDL, but does not provide a simple semantics guarantee like OIAT [31]. It also has difficulty describing the non-sequential behaviors targeted by XPDL. HeteroCL's [19] `break` statement is a shorthand method for producing a priority encoder. C++ style `try-catch` statements in HLS tools do not generate real hardware. They either control simulation or capture the exceptions of HLS tools themselves. Compared to HLS, XPDL provides finer-grained control over pipeline behaviors and easier reasoning.

ISA specification and system modeling languages describe the specifications of hardware rather than the actual implementation. Sail [1] captures exceptions with state monads to achieve precise and finer-grained control over exception handling. SystemC [30] supports events and event handling that are well suited to model hardware exceptions. These languages are used for verification but cannot be synthesized into real hardware without significant effort. By contrast, XPDL is fully synthesizable.

Traditionally, hardware exceptions are implemented at the RTL level, enabling extensive hand-tuning for efficient hardware. However, the low level of RTL makes design-space exploration expensive. Chan et al. [6] abstract checkpoint and rollback as Verilog extensions to simplify their use, but the programmer still has to implement rollback logic for each process state, whereas XPDL automatically generates rollback logic for different process states. Hazard interfaces are a recent idea generalizing valid-ready interfaces that can modularize pipelined circuits with structural, data and control hazards [18]. While it can help in generating complex control logic, XPDL excels on fast prototyping and maintaining high-level semantics, and is more focused on processor pipelines, a challenging and important use case.

Teng and Dubach [40] recently applied continuation-passing style to the generation of hardware-synthesizable exceptions. While their syntactic constructs appear similar (also adopting a try–catch approach), their goal is different: efficiently translating software-level run-time exceptions into circuits, to facilitate porting software applications to hardware. In contrast, XPDL targets native hardware designs and provides novel insights into precise exceptions.

6 Conclusion

XPDL offers a structured and expressive way for high-level hardware languages to support non-sequential behavior, thereby enhancing their practical utility by supporting features needed by OS software. It is, to the best of our knowledge, the first HDL to encapsulate ISA-level hardware exceptions in language abstraction, connecting precise exceptions to OIAT semantics. Although in this work we demonstrate one concrete implementation of this design through our compiler, it is not the only possible approach.

Acknowledgments

We would like to thank Derin Ozturk and Christopher Batten for providing and assisting with ASIC tools, and Hongzheng Chen, Niansong Zhang, and Zhiru Zhang for providing and assisting with FPGA tools. We would also like to thank Google for a grant supporting this work.

References

[1] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, et al. 2019. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *Proc. ACM on Programming Languages* 3, POPL (2019), 1–31.

[2] Görkem Aşılıoğlu, Emine Merve Kaya, and Oğuz Ergin. 2010. Complexity-effective rename table design for rapid speculation recovery. In *Architecture of Computing Systems-ARCS 2010: 23rd Int'l Conf., Hannover, Germany, February 22-25, 2010. Proceedings 23*. Springer, 15–24.

[3] Robert Balas, Alessandro Ottaviano, and Luca Benini. 2024. CV32RT: Enabling fast interrupt and context switching for RISC-V microcontrollers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2024).

[4] Drew Barbier and A. C. Palmer Dabbelt. 2020. RISC-V platform-level interrupt controller specification.

[5] Björn B. Brandenburg, Hennadiy Leontyev, and James H Anderson. 2011. An overview of interrupt accounting techniques for multiprocessor real-time systems. *Journal of Systems Architecture* 57, 6 (2011), 638–654.

[6] Carven Chan, Daniel Schwartz-Narbonne, Divjyot Sethi, and Sharad Malik. 2012. Specification and synthesis of hardware checkpointing and rollback mechanisms. In *Proc. 49th Annual Design Automation Conf.* 1226–1232.

[7] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM on Programming Languages* 1, ICFP (2017), 1–30.

[8] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491.

[9] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proc. Theory and Practice of Software, 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary). Springer-Verlag, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[10] Steven Derrien, Thibaut Marty, Simon Rokicki, and Tomofumi Yuki. 2020. Toward speculative loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 4229–4239.

[11] Tom Feist. 2012. Vivado design suite. *White Paper* 5, 30 (2012), 24.

[12] Siddharth Gupta, Yuanlong Li, Qingxuan Kang, Abhishek Bhattacharjee, Babak Falsafi, Yunho Oh, and Mathias Payer. 2023. Imprecise store exceptions. In *Proc. 50th Annual Int'l Symp. on Computer Architecture*. 1–15.

[13] Wen-mei W. Hwu and Yale N. Patt. 1987. Checkpoint repair for out-of-order execution machines. In *Proc. 14th annual international symposium on Computer architecture*. 18–26.

[14] Cadence Design Systems Inc. 2020. Innovus implementation system. (2020).

[15] Synopsys Inc. 2020. Design compiler RTL synthesis solution. (2020).

[16] SiFive Inc. 2020. SiFive interrupt cookbook. *version* 1.2 (2020).

[17] Intel Corporation. 2011. Intel® 64 and IA-32 architectures software developer's manual. *Volume 3B: system programming guide, part* 2, 11 (2011), 0–40.

[18] Minseong Jang, Jungin Rhee, Woojin Lee, Shuangshuang Zhao, and Jeehoon Kang. 2024. Modular hardware design of pipelined circuits with hazards. *Proc. ACM on Programming Languages* 8, PLDI (2024), 28–51.

[19] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proc. 2019 ACM/SIGDA Int'l Symp. on Field-Programmable Gate Arrays*. 242–251.

[20] Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers* 100, 9 (1979), 690–691.

[21] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: The Works of Leslie Lamport*. 179–196.

[22] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. 2020. Meltdown: Reading kernel memory from user space. *Commun. ACM* 63, 6 (2020), 46–56.

[23] Biruk Wendimagegn Mammo. 2017. *Reining in the functional verification of complex processor designs with automation, prioritization, and approximation.* Ph. D. Dissertation.

[24] Francisco Marques, Manuel Rodríguez, Bruno Sá, and Sandro Pinto. 2024. "Interrupting" the status quo: A first glance at the RISC-V advanced interrupt architecture (AIA). *IEEE Access* 12 (2024), 9822–9833.

[25] Prabhat Mishra, Nikil Dutt, and Alex Nicolau. 2001. Specification of hazards, stalls, interrupts, and exceptions in EXPRESSION. (2001).

[26] Mayan Moudgill and Stamatis Vassiliadis. 1996. Precise interrupts. *IEEE Micro* 16, 1 (1996), 58–67.

[27] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. 2007. A SLOC counting standard. In *Cocomo ii forum*, Vol. 2007. Citeseer, 1–16.

[28] Rishiyur Nikhil. 2004. Bluespec System Verilog: Efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE Int'l Conf. on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.* IEEE, 69–70.

[29] Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in Scala.* Artima Inc.

[30] Preeti Ranjan Panda. 2001. SystemC: A modeling platform supporting multiple design abstractions. In *Proc. 14th Int'l Symp. on Systems Synthesis*. 75–80.

[31] Blake Pelton, Adam Sapek, Ken Eguro, Daniel Lo, Alessandro Forin, Matt Humphrey, Jinwen Xi, David Cox, Rajas Karandikar, Johannes de Fine Licht, et al. 2024. Wavefront threading enables effective high-level synthesis. *Proc. ACM on Programming Languages* 8, PLDI (2024), 1066–1090.

[32] Brent Przybus. 2010. Xilinx redefines power, performance, and design productivity with three new 28 nm FPGA families: Virtex-7, Kintex-7, and Artix-7 devices. *Xilinx White Paper* (2010).

[33] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. Machsuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE Int'l Symp. on Workload Characterization (IISWC)*. IEEE, 110–119.

[34] David Seal. 2001. *ARM architecture reference manual.* Pearson Education.

[35] Dezso Sima. 2000. The design space of register renaming techniques. *IEEE Micro* 20, 5 (2000), 70–83.

[36] Ben Simner, Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Ohad Kammar, Jean Pichon-Pharabod, et al. 2024. Relaxed exception semantics for Arm-A (extended version). *arXiv preprint arXiv:2412.15140* (2024).

[37] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. 2019. Microscope: Enabling microarchitectural replay attacks. In *Proc. 46th Int'l Symp. on Computer Architecture*. 318–331.
[38] James E. Smith and Andrew R. Pleszkun. 1985. Implementation of precise interrupts in pipelined processors. *ACM SIGARCH Computer Architecture News* 13, 3 (1985), 36–44.
[39] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, et al. 2007. FreePDK: An open-source variation-aware design kit. In *2007 IEEE international conference on Microelectronic Systems Education (MSE'07)*. IEEE, 173–174.
[40] Paul Teng and Christophe Dubach. 2025. Hardware synthesizable exceptions using continuations. In *Proc. 30th Asia and South Pacific Design Automation Conf.* 1104–1111.
[41] Donald Thomas and Philip Moorby. 2008. *The Verilog® hardware description language*. Springer Science & Business Media.
[42] Robert M. Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development* 11, 1 (1967), 25–33.
[43] Zynq UltraScale. 2020. Device technical reference manual. (2020).
[44] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. 2014. The RISC-V instruction set manual, volume I: User-level ISA, version 2.0. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54* (2014), 4.
[45] Stephen Williams and Michael Baxter. 2002. Icarus Verilog: Open-source Verilog more than a year later. *Linux Journal* 99 (2002), 3.
[46] Jean Yang and Chris Hawblitzel. 2010. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proc. 31st ACM SIGPLAN Conf. on Programming Language Design and Implementation*. 99–110.
[47] Drew Zagieboylo, Charles Sherk, G. Edward Suh, and Andrew C. Myers. 2022. PDL: a high-level hardware design language for pipelined processors. In *Proc. 43rd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation*. 719–732.
[48] Drew Zagieboylo, Charles Sherk, and Kevin Zhang. 2022. PDL: a hardware pipeline description language. Github repository at https://github.com/apl-cornell/PDL.

A Artifact Appendix

A.1 Abstract

The source code for this work is part of the PDL compiler, available at https://github.com/apl-cornell/pdl. The extension featuring XPDL is located on the `exn` branch. Please use /README file for detailed usage.

Since running the experiment results of our paper relies on licensed software provided by another research group, this artifact focuses on the compilation pipeline and generation of Verilog. Readers are encouraged to inspect the code, run the simulation, and explore the compiler functionality. Reproducing full experimental results requires access to proprietary toolchains (Synopsys Design Compiler and Cadence Innovus).

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

- **Program: A Docker file that contains the compiler is available at the PDL Github repository.**
- **Hardware: (Recommended) A recent Intel CPU.**
- **Output: Verilog files of XPDL generated pipelines**
- **How much disk space required (approximately)?: 5 GB**
- **How much time is needed to prepare workflow (approximately)?: 1 hour**
- **How much time is needed to complete experiments (approximately)?: 1 hour**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: MIT**

A.3 Description

A.3.1 How to access. Both the source code and Dockerfile are available at https://github.com/apl-cornell/PDL/.

A.3.2 Hardware dependencies.

- To match configurations in the paper: compilation was performed with Apple M3 Max.
- Optional: synthesis and place-and-route were done using an Intel processor.

A.3.3 Software dependencies.

- Docker
- A recent Linux distribution
- Java, Scala and sbt (OpenJDK 8)
- Bluespec Compiler (and Haskell)
- Icarus Verilog
- (Optional) Synopsys Design Compiler
- (Optional) Cadence Innovus

A.4 Installation

We recommend installing and running the artifact using Docker: https://www.docker.com/. The Dockerfile is available at: /Dockerfile on the `exn` branch aforementioned. Please download the file from /Dockerfile and place it in a folder with at least 5 GB of available space.

To build and launch the container, execute the following commands in a terminal:

```
# Build the Docker image:
docker build -t pdl-env .

# After the build completes:
docker run -it --rm pdl-env
```

A.5 Evaluation and Expected Results

After executing the `docker run` command, you will be placed in the `pdl` directory. Navigate to the `xpdl-asplos` folder, which contains:

- **tests/**: Demonstrates core language features. A provided shell script (`tests/runall.sh`) compiles and simulates all test cases automatically. All generated Bluespec(BSV)/Verilog files are placed under their respective `output/` subdirectories. The output contains the simulation results of these tests. The expected behaviors are specified at the top of each corresponding `*.pdl` file.
- **risc-pipes/**: Contains RISC-V pipelines extended with hardware exceptions. These examples are mainly for

inspection but they can be modified and played with if readers are interested.

A.6 Experiment Customization

To compile a (X)PDL program with custom arguments, use the following command:

```
pdl gen <input-file.pdl> -o <output-directory> \
    --memInit <memory-name>=<file>
```

Memory input files like `ti` (for instructions) and `td` (for data) can be manually edited to test different runtime scenarios.

Please refer to the /README file for more information.

A.7 Notes

- Running RISC-V pipelines will need manual tweaks to the generated BSV files, as Bluespec compiler cannot identify the exclusivity of rule execution condition and caused false error.