CaST: Balancing Accelerator Design

Alborz Jelvani
Rutgers University
USA

Richard P. Martin
Rutgers University
USA

Negin Dehghanchaleshtori
Rutgers University
USA

ABSTRACT

We present *CaST* (Channels and State), a language for designing accelerators that is founded on the paradigm of a graph of concurrent state machines connected by channels. CaST is programmer friendly in order to remove the burden of creating both control and data flow constructs at the register transfer level, as is common with traditional hardware design languages. The goal is to enable making performant accelerators with equivalent, or even less effort, than programming similar functions in high level languages.

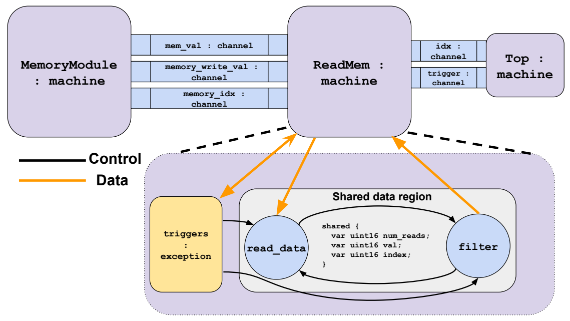
1 INTRODUCTION

The difficulty of building hardware accelerators with existing languages and paradigms is well known. CaST takes the position that most accelerators can be straightforwardly described as state machines connected by channels, and thus it is a programming language that directly supports these two constructs. CaST seeks to be accessible to a wide range of programmers and designers, including those familiar with imperative systems languages such as C, Go, and Zig. It should also be intuitive to designers working with Hardware Design Languages (HDLs) and generators such as Amaranth [1] and Chisel [2].

CaST has a difficult balancing act; on one hand the traditional HLS approach of using widely known languages is accessible, but tends to perform poorly and creates unpredictable designs. Coding at the low-level RTL can be performant but has a high learning curve, and even when mastered is still tedious and error prone. CaST seeks the sweet spot that is simple, intuitive, expressive, and performant.

The core paradigm of a CaST program is a graph of interconnected state machines as shown in the top portion of Figure 1. It shows the filter from Figure 2 built as two state machines connected by channels. CaST channels are similar to those in Go or Anvil [5] — they send and receive structured messages similar to a C language *struct*. Instead of enforcing strict timing contracts, CaST relies on its predictable lowering process which translates states into hardware event loops and channels into latency-insensitive `esi` FIFOs.

CaST is designed to make it straightforward to describe performant accelerators on *networked architectures*. These architectures include a broad class of parallel systems including ASICs, FPGAs, Coarse-Grained Reconfigurable Architectures (CGRAs), and parallel processor arrays such as the Cerebras wafer [4]. We expect CaST programmers to build programs that realize customized structures



```
shared {
  var uint16 num_reads;
  var uint16 val;
  var uint16 index;
}
```

Figure 1: High-level block diagram depicting how the code in Figure 2 maps to hardware.

such as pipelines, systolic arrays, distributed hash tables and radix trees.

By forcing programmers to describe a graph of state-machines, CaST reduces implementation choices compared to RTL. This reduction makes CaST more programmer friendly by making it easier to reason about a design.

CaST differs itself from existing HDLs with the following:

(1) CaST forces programmers to communicate using channels for parallel tasks and shared memory for sequential intra-state logic.
(2) CaST requires control-flow constructs that must be constructed in HDLs: explicit goto's for sequential state transitions and exceptions for non-local control-flow.
(3) CaST channel communication uses blocking and selection within state machines. This forces sequential consistency within state machines.

A CaST program captures designers' intent at an abstraction level higher than RTL while ensuring that the *reduction* (synthesis) remains transparent and predictable, analogous to mapping C to RISC instructions. CaST paradigms map straightforwardly to common physical substrates: state machines reduce to logic and local memory, and structured channels reduce to FIFOs.

CaST also allows a separation of specification from instantiation. Machine and channel types are defined in a separate *instantiation block*. This block is a compiler-time executable program that creates instances of machines, channels, and specifies their interconnections; it is similar to Verilog generate statements. However, CaST's instantiation approach centralizes all the instantiation in one place rather than scattered over the code.

2 THE CAST LANGUAGE

We now show CaST using a small example memory filter shown in Figure 2. Values are read from a memory module and compared against a threshold. If the value is less than 256, a message is printed

LATTE '26, March 23, 2026, Pittsburgh, Pennsylvania, USA

indicating the value is within range. Otherwise, the value stored at that memory index is overwritten with zero.

A CaST design consists of two main sections: *machine* blocks define state machines that execute as independent threads of control and communicate only through typed channels, while the *instantiate* section specifies the number of machines and their connectivity, as shown in Figure 2.

A *machine* block contains several sub-blocks. The *interface* block declares typed channels used for inter-machine communication. The *shared* block defines variables used to pass values between states within a machine. The *states* block contains definitions for each state of the state machine.

The example in Figure 2 contains two states: `read_data` (lines 21–26), which reads a memory value and stores it in the shared variable `val`, and `filter` (lines 27–35), which checks whether the value is less than 256. The two states share variables to pass information between them and repeatedly process memory values.

An *exception* block allows execution to be interrupted at state transition boundaries when data is pending on an exception channel. During an exception, the current state (control and data) of the machine is preserved and control is transferred to the exception state. The exception state is permitted to modify the preserved state data. Every exception state may transition to any other states, but the control flow graph formed by exception states (that is, states reachable from the exception state) *may not* contain cycles. This helps ensure termination of exceptions. Upon termination of an exception, control is transferred back to the excepted state.

instantiate blocks define the layout of the design. These blocks are intended for interpretation at compile time in a traditional von Neumann paradigm. The class of programs in the instantiate block is restricted to ensure termination, so unbouned looping constructs are not allowed.

3 COMPILING CAST TO HARDWARE

CaST is a WIP. It is being implemented as a frontend for the MLIR CIRCT compiler infrastructure. The frontend lowers CaST source into various CIRCT dialects, after which existing CIRCT lowering passes produce SystemVerilog. Figure 3 shows the CaST compilation pipeline.

3.1 Lowering states to hw.module

Each CaST state machine (the blue circles in Figure 1) is lowered to a CIRCT module (`hw.module`) that implements an event loop. We cannot directly use the CIRCT `fsm` dialect because CaST machines support exceptions, which require save and restore logic beyond what `fsm` provides. Instead, we implement the event loop inside an `hw.module`, allowing full control over how exceptions interact with state transitions.

The event loop reacts to two types of events: input messages and state transitions. Input events occur when data becomes available on input channels, which the event loop detects by checking whether the corresponding FIFO has valid data before dequeuing it. State transitions occur when a `goto` statement requests a transition in the previous cycle. The `goto` writes the target state to the state register on the next clock edge, and the event loop dispatches that state in the following cycle. Before dispatching any transition, the

```
 1 enum commands = {CLEAR, PRINT}
 2 // a machine can take compile-time parameters inside [ ]
 3 machine ReadMem [mem_size: uint16] {
 4   // all channels are declared here
 5   interface {
 6     input uint16 idx;
 7     input uint16 mem_val;
 8     input commands trigger;
 9     output uint16 memory_idx;
10     output uint16 memory_write_val;
11   }
12   // these are variables shared between states of a
       single machine
13   shared {
14     var uint16 num_reads;
15     var uint16 val;
16     var uint16 index;
17   }
18   // each state is defined here
19   states {
20     // all paths through a state must end in a goto
21     read_data: index <- idx {
22       memory_idx <- index;
23       val <- mem_val;
24       num_reads++;
25       goto filter;
26     }
27     filter: {
28       if (val < 256) {
29         print("within range");
30       } else {
31         memory_idx <- index;
32         memory_write_val <- 0;
33       }
34       goto read_data;
35     }
36     // exception states are triggered by channels
37     exception triggers: cmd <- trigger {
38       if (cmd == CLEAR) {
39         for(var uint16 i=0; i<mem_size; i++) {
40           memory_idx <- i;
41           memory_write_val <- 0;
42         }
43       } else if (cmd == PRINT) {
44         for(var uint16 i=0; i<mem_size; i++) {
45           memory_idx <- i;
46           var uint16 data <- mem_val;
47           print(data);
48         }
49       }
50     }
51   }
52 }
53 instantiate {
54   mm = MemoryModule(); // defined externally
55   rm = ReadMem[mem_size=1024]();
56   // connect channels
57   rm.mem_val <- mm.mem_val;
58   mm.write <- rm.memory_write_val;
59   mm.mem_index <- rm.memory_idx;
60   rm.memory_idx <- mm.mem_index;
61   // send an input
62   rm.idx <- 10;
63   // trigger an exception
64   rm.trigger <- CLEAR;
65 }
```

Figure 2: An example CaST design that reads data from a memory module.

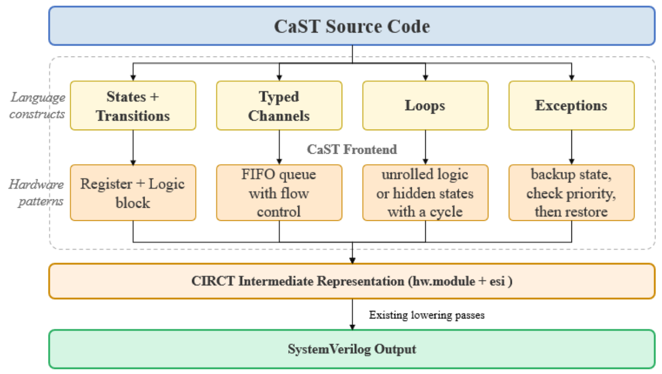


Figure 3: The CaST compilation pipeline.

event loop first checks for pending exception messages, which take priority over normal transitions.

During lowering, each CaST state is translated into four components.

(1) First, the state receives a unique binary encoding so the hardware can identify it.
(2) Second, guard conditions are generated from gating channels to ensure the state fires only when the required input channels contain valid data. When a state does not have an input channel it can execute whenever a goto transitions to that state, but it may still block if there is a channel access inside that state.
(3) Third, the state's logic may be lowered to use hidden internal states to support unbounded loops.
(4) Finally, the `goto` statement determines the next state by writing the target state to the state register (contained in the shared data region in Figure 1) on the next clock edge.

3.2 Lowering channels to `esi` FIFOs

Every CaST channel maps to an `esi` channel type in the CIRCT `esi` dialect [3]. The width matches the channel's declared type width. On the module boundary, enqueue ports are exposed so other modules can send data in. Dequeue ports stay internal to the event loop. Output channels are exposed as ready-valid ports. These channel types are realized into instantiations of actual channels in the CaST *instantiate* block, where they map to `esi.fifo` operations.

In order for a state machine to manipulate channel data, the output of the `esi` channel must be written to a local variable of the same type as the channel type. This requires a handshake.

When a state gates on multiple channels, two modes are supported. A conjunction requires all gated FIFOs to have data before the event loop dequeues them simultaneously. This is an atomic multi-channel receive: either all channels are ready and all are dequeued together, or none are dequeued. A disjunction dequeues from whichever FIFO has data first. If multiple FIFOs have data at the same time under disjunction, a round-robin arbiter selects which one to dequeue to ensure fairness. In addition to gating reads at the state header, channels can also be read inside the body of a state. These body reads behave differently from gating reads. The state has already started executing, but it stalls at the body read until the channel provides data[1].

3.3 Supporting exceptions

The event loop dispatches events in priority order, with exceptions always handled first. Within the exception priority level, multiple pending exceptions are served in round-robin fashion, but the ordering is otherwise undefined. When an exception triggers, the current state register is saved. The handler then runs to completion. Once the handler terminates, the saved state is restored and normal dispatch resumes from exactly where control was excepted. If no exception is pending, the event loop dispatches the current state transition as normal.

For exceptions, the channel FIFO serves as a pending buffer. When data arrives on an exception channel, it sits in the FIFO until the event loop gets to it during exception servicing. This means exceptions are not lost even if the machine is in the middle of executing a state. They are buffered and will be handled at the next state transition boundary. Exceptions are implemented with logic checks at state transition edges. Before every `goto` takes effect, the event loop checks whether any exception FIFOs have pending data. If they do, it saves the current state and transfers control to the handler instead of performing the normal transition.

Our current design means that a blocking channel in the middle of a state (e.g., line 23 in Figure 2) also blocks exceptions. [2]

To give the programmer control over the resource overhead of implementing exceptions, `goto` transitions are permitted to disable certain exceptions. By default, each `goto` checks the exception buffer before a state transition. The cost of supporting exceptions in a machine is a copy of the state register and a priority encoder over the exception FIFOs.

4 CHALLENGES AND FUTURE WORK

As CaST is a WIP we are still exploring the implementation of FFTs, matrix multiplication, and IP packet transformers. We have found that expressing these as a graph of interconnected state machines is a natural way for programmers to conceptualize these accelerators. This is a positive result as ease of design expressibility is a primary goal.

However, several challenges remain. Low throughput from both channel delay and sequential state transitions is one difficulty, as channel semantics hide timing from the programmer.

Another challenge is how to get good performance by reducing the handshaking needed for synchronizing channel sends and receives. Handshakes can be removed at synthesis time if the inter-machine data flow graph contains paths with known timings. We believe this will be applicable to systolic arrays, but may not work for more generic designs.

REFERENCES

[1] Amaranth HDL Project. [n.d.]. Amaranth Hardware Description Language. https://amaranth-lang.org. Accessed 2026.

[1]The design decision to allow blocking channels to also be used inside states is that blocking channels may exist in branches, in which case the state will not always block.
[2]We can fix this by synthesizing a check on the exception channels while blocking in the middle of a state; however we have not implemented this.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th annual design automation conference*. 1216–1225.

[3] John Demme. 2021. Elastic Silicon Interconnects: Abstracting Communication in Accelerator Design. *arXiv preprint arXiv:2111.06584* (2021).

[4] Sean Lie. 2024. Inside the cerebras wafer-scale cluster. *IEEE Micro* 44, 3 (2024), 49–57.

[5] Jason Zhijingcheng Yu, Aditya Ranjan Jha, Umang Mathur, Trevor E Carlson, and Prateek Saxena. 2025. Anvil: A General-Purpose Timing-Safe Hardware Description Language. *arXiv preprint arXiv:2503.19447* (2025).