SimCommand: A High-Performance RTL Testbench API

Vighnesh Iyer, Kevin Laeufer, Koushik Sen, Borivoje Nikolic
UC Berkeley

Testbench APIs

Testbench APIs provide an interface between *testbench logic* and a *simulator backend*.

Simulator backends: Verilator, Icarus, VCS

Native Testbench APIs: C++, SystemVerilog

- Compiled into simulator → performant
- C++ with Verilator does not suppport fork/join
- SystemVerilog verification features (e.g. OOP, SVA) not supported by open source tools

General-Purpose Testbench APIs: cocotb (Python), chiseltest (Scala)

- Portability across simulator backends
- Reuse of host language libraries and ecosystem

Testbench API Primitives

- **poke**: `dut.signal = 100`
- **peek**: `value = dut.signal`
- **step**: `repeat (cycles) @(posedge clk)`
- **fork/join** to express testbench parallelism

Expressing Parallelism

```
module example (input clk, input [31:0] a, b);
```

```
a.poke(1)
clk.step(4)
a.poke(2)
```

```
b.poke(100)
clk.step(2)
b.poke(200)
clk.step(4)
b.poke(300)
```

Manual thread interleaving

```
trait Thread {
  def step(): Unit
  def done(): Boolean
}
class a extends Thread {
  var cycle = 0
  def step() = {
    if (cycle == 0)
      dut.a.poke(1.U)
    else if (cycle >= 4)
      dut.a.poke(2.U)
    cycle = cycle + 1
  }
  def done() = (cycle == 5)
}
val threads = Seq(a(), b())
while (!threads.all(_.done))
threads.forEach(_.step())
```

fork/join threading

```
val a = fork {
  a.poke(1.U)
  clk.step(4)
  a.poke(2.U)
}
  val b = fork {
  b.poke(100.U)
  clk.step(2)
  b.poke(200.U)
  clk.step(4)
  b.poke(300.U)
}
a.join
b.join
```

Threads are synchronized on clock edges

Motivation: Overhead of Fork/Join

Platform	Throughput	Slowdown
Chiseltest with manual thread interleaving	220 kHz	-
Chiseltest with fork/join threading	7.8 kHz	28x
cocotb with async/await	3.8 kHz	108x

DecoupledGCD benchmark on Verilator

We want to design an open-source, **high-performance**, general-purpose testbench API with fork/join support.

Sources of Perf Overhead in chiseltest

- Fork-ing spawns a *new* JVM thread
- Parking and unparking threads is **slow**

A Functional Testbench API

FP Principle: Separate description from interpretation

```
sealed trait Command[R] // R = return type of the Command
case class Peek(signal: I) extends Command[I]
case class Poke(signal: I, value: I) extends Command[Unit]
case class Step(cycles: Int) extends Command[Unit]
```

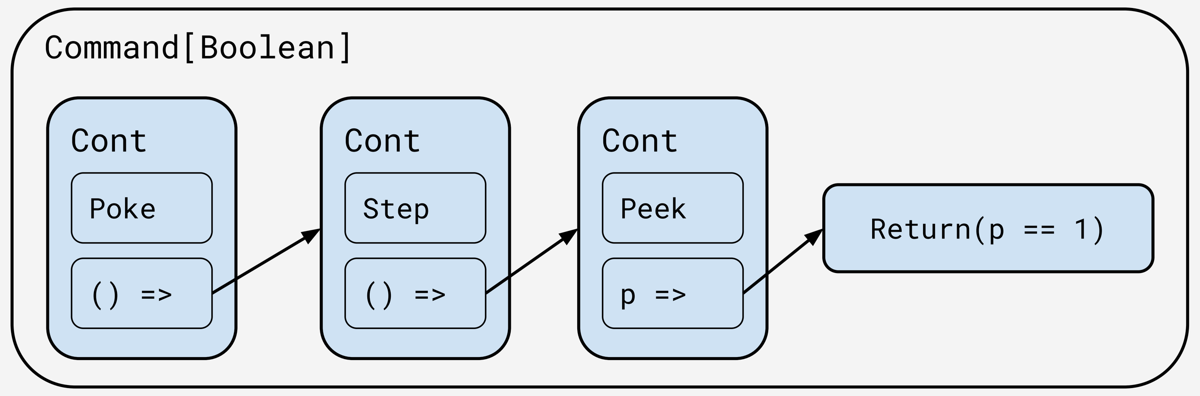
Instances of these classes only *describe* the actions of peeking, poking, or stepping.

Command Continuations

`flatMap` 'unwraps' the return value of a `Command`, and continues with another function.

```
val program: Command[Boolean] = for {
    _ <- Poke(dut.enq.valid, 1.B)
    _ <- Step(1)
    p <- Peek(dut.deq.valid)
} yield p.litValue == 1
```

Scala for-comprehensions desugar to nested `flatMap`s. `program` is a *value*, but *looks* like imperative code.



Command Combinators

Commands are values → can be manipulated by ordinary functions

```
// Run a list of programs sequentially
def concat[R](cmds: Seq[Command[R]]): Command[Unit]

// Run a list of programs and aggregate their results
def sequence[R](cmds: Seq[Command[R]]): Command[Seq[R]]
```

Library of stack-safe functions that emulate imperative loops

```
// Run this program until it returns false
def doWhile(cmd: Command[Boolean]): Command[Unit]

// Step the clock until the signal == value
def stepUntil(signal: I, value: I): Command[Unit]
```

Ready-Valid Example

```
def enqueue(data: T): Command[Unit] = for {
  _ <- poke(io.bits, data)
  _ <- poke(io.valid, true.B)
  _ <- stepUntil(io.ready, true.B)
  _ <- step(1)
  _ <- poke(io.valid, false.B)
} yield ()
```

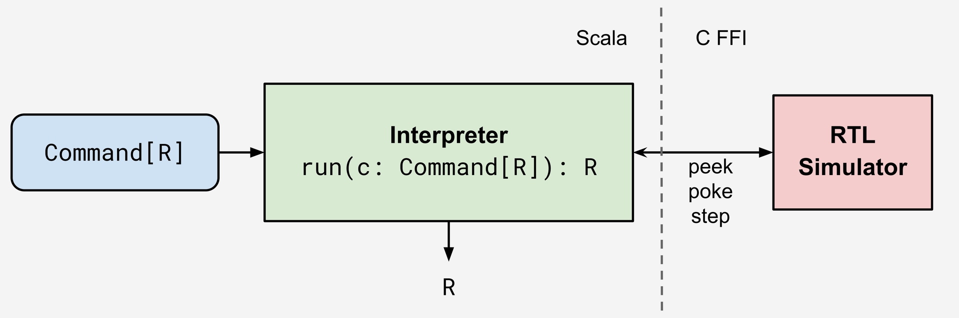
```
def dequeue(): Command[T] = for {
  _ <- stepUntil(io.valid, true.B)
  _ <- poke(io.ready, true.B)
  value <- peek(io.bits)
  _ <- step(1)
} yield value
```

Fork/Join Example

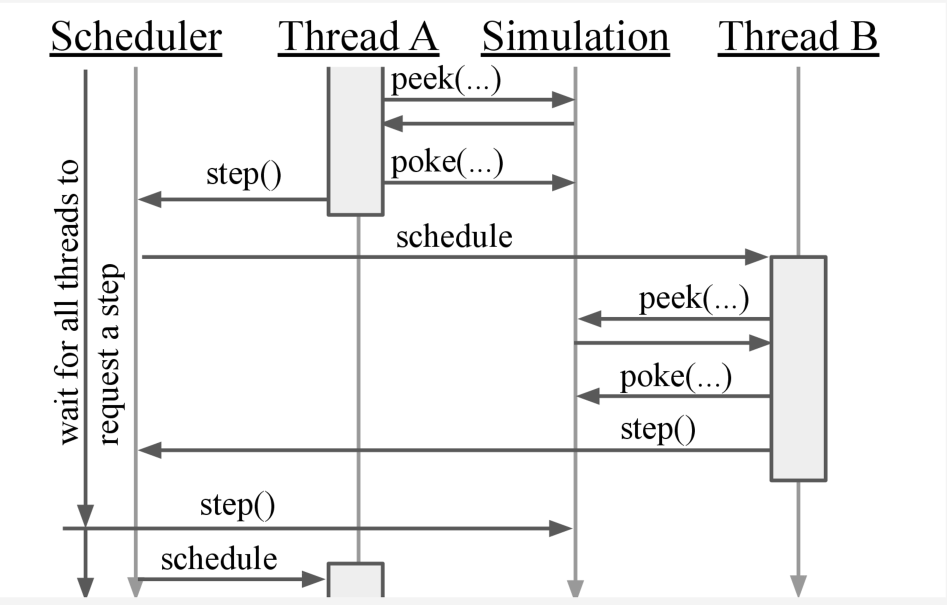
```
val queueTest: Command[Boolean] = for {
    enqThread <- fork(enqueue(100.U))
    deqThread <- fork(dequeue())
    _          <- join(enqThread)
    data       <- join(deqThread)
} yield data.litValue == 100

test(new Queue(UInt(32.W))) { dut =>
  val itWorks: Boolean = run(queueTest)
  assert(itWorks)
}
```

Interpreter / Scheduler Algorithm



The interpreter has full control of a thread (a pointer to a `Command`).



- On each timestep
  - Run every thread until a step, join, or return
  - Collect any new threads spawned
  - Repeat until a fixpoint is reached
- Step the clock
- Repeat until the main thread returns

Results

Platform	DecoupledGCD		NeuromorphicProcessor	
SystemVerilog + commercial simulator	0.40 s	412 kHz	0:30 min	1782 kHz
Chiseltest with manual thread interleaving	0.75 s	220 kHz	2:03 min	432 kHz
Chiseltest with SimCommand	**2.4 s**	**67 kHz**	**5:23 min**	**165 kHz**
Chiseltest with fork/join threading	21 s	7.8 kHz	27:21 min	32.6 kHz
cocotb	43.2 s	3.8 kHz	89:38 min	9.9 kHz

17x faster than cocotb, **5x** faster than chiseltest

Conclusion

Encoding simulation commands as values and using a single-threaded user-level scheduler enables the **fastest general-purpose testbench API with fork/join support**.

Coming up:

- Channels for inter-thread communication
- Better debug capabilities, thread tracing
- TileLink / AXI VIPs
- Performance parity with SystemVerilog