Mixed Topics

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

November 11, 2025

Outline

- Mixed collection of generator ideas
- My last lecture (probably)
- Today, intermediate project presentation and demo
- Next week, another guest
- Week 12: project work
- Week 13: Final project presentation and demonstration

Use a `var` for a Generator

- A `var` can be reassigned
- Just keep the Chisel components connected
- No need to keep the references around

```
class Delay(n: Int) extends Module {
    val io = IO(new Bundle {
        val in  = Input(Bool())
        val out = Output(Bool())
    })
    require(n >= 0)
    var lastConn = io.in
    for (i <- 0 until n)
        lastConn = RegNext(lastConn)
    io.out := lastConn
}
```

Do you want Recursion?

- With a helper function

```
class DelayRec(n: Int) extends Module {
    val io = IO(new Bundle {
        val in  = Input(Bool())
        val out = Output(Bool())
    })
    require(n >= 0)
    def helper(n: Int, lastConn: Bool): Bool =
       {
        if (n == 0) lastConn
        else helper(n-1, RegNext(lastConn))
    }
    io.out := helper(n, io.in)
}
```

Parameterize Number of Ports

- Use a `Vec`

```
class Example(n: Int, w: Int) extends Module {
    val io = IO(new Bundle {
        val in  = Input(Vec(n, UInt(w.W)))
        val out = Output(UInt(w.W))
    })
```

- Next week, a more flexible version
- Decide at runtime even on the type of a port

Use Methods

- Modules and Bundles are Scala classes
- One can define methods on them, like in any Scala class
- We have seen `apply()` on the companion object
- Remember what this gave us?

Factory Method `apply`

- Simpler component creation and use
- Usage similar to built-in components, such as `Mux`

```
val myAdder = Adder(x, y)
```

- A little bit more work on the component side
- Define an `apply` method on the companion object that returns the component (output port)

```
object Adder {
  def apply(a: UInt, b: UInt) = {
    val adder = Module(new Adder)
    adder.io.a := a
    adder.io.b := b
    adder.io.result
  }
}
```

Decoupled Components

- Using the read/valid handshake
- Some flexibility on sending and receiving
- Data is transferred when ready and valid are asserted
- Chisel provides a standard bundle for this: `DecoupledIO`

```
class DecoupledIO[T <: Data](gen: T) extends
    Bundle {
  val ready = Input(Bool())
  val valid = Output(Bool())
  val bits  = Output(gen)
}
```

Methods on `DecoupledIO`

- `fire()` is true if and only if ready and valid are asserted
- `enq(data)` sets data and sets valid to true (no check on ready)
- `noenq()` deasserts valis.
- `deq/nodeq` for the receiver side

```
when(io.in.fire()) {
  // do the transfer
}
```

- Sala convention: methods with no side effects, leaving off the empty parentheses

```
when(io.in.fire) {
  // do the transfer
}
```

Arbiter

- Several components have a request for a shared resource
- Examples: bus, memory port, on-chip network ports
- Needs to select which request gets granted
- Different algorithms to do this
- Simplest is priority, but unfair
- Fairnis, and guaranteed progress, especially in real-time systems, is an issue
- I have a fair arbiter in the Chisel book (10.6.2)

Chisel Arbiter

- Uses `DecoupledIO` for inputs and the output

```
val arb = Module(new Arbiter(UInt(), 2))
arb.io.in(0) <> producer0.io.out
arb.io.in(1) <> producer1.io.out
consumer.io.in <> arb.io.out
```

- `Arbiter` is a simple priority based arbitration
- `RRArbiter` Chosen in round robin
  - No details on implementation
  - True, single cycle, round robin is combinational complex
- `LockingRRArbiter` grant reserved for several clock cycles

Crossbar

- N inputs are connected to M outputs
- Needs M arbitration circuits, each of N inputs
- Need an address for the destination
- Scales only to a moderate N and M
- Add pipelining for larger N and M
- Switch to a Network-on-Chip

FIFO Queue

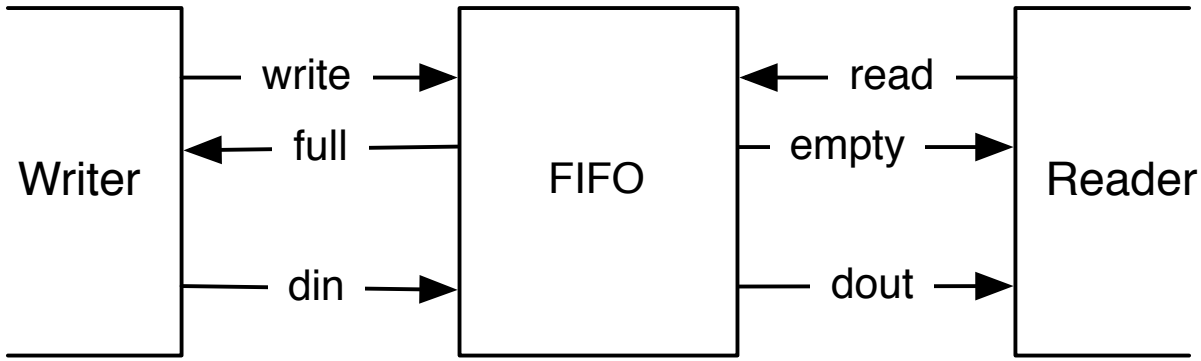

Figure: A writer, a FIFO buffer, and a reader.

- FIFO style queues between a sender and a receiver
- The sender/write enqueues/writes into the Queue
- The receiver/reader dequeues/reads from the Queue
- Can *even out* bursty traffic

Queues

- Used when there is bursty traffic
  - E.g., on a serial port
  - Write faster from CPU than serial can send
  - Receive data when CPU is busy with other stuff
- But can fill up when the sender is continuously *faster*
- Showing implementation variations in the Chisel book
  - Bubble FIFO (was proposed in lab 5)
  - double buffer, memory-based, ...

Bubble FIFO Example

- FIFO interface

```
class FifoIO[T <: Data](private val gen: T)
    extends Bundle {
  val enq = Flipped(new DecoupledIO(gen))
  val deq = new DecoupledIO(gen)
}
```

- Abstract base class (with some common code)

```
abstract class Fifo[T <: Data](gen: T, val
    depth: Int) extends Module {
  val io = IO(new FifoIO(gen))

  require(depth > 0, "Number of buffer
      elements needs to be larger than 0")
}
```

- FIFO code

Bubble FIFO

- Simple, easy to understand
- Uses minimal resources
- However, each buffer stage has to toggle between empty and full
- Maximum bandwidth is one word every two clock cycles
- When full, needs N cycles for a restart
  - The free element *bubbles* towards the input
- Better solutions
  - Double buffer
  - Memory with read and write pointers

Use Inheritance

- Defined an abstract base class for the FIFO and extend it
  - Also use traits to inherit from several sources
- Learn from software development to share code
- We select the implementation by the type

```
class DoubleBufferFifo[T <: Data](gen: T,
    depth: Int) extends Fifo(gen: T, depth:
    Int) {
...
class RegFifo[T <: Data](gen: T, depth: Int)
    extends Fifo(gen: T, depth: Int) {
...
class MemFifo[T <: Data](gen: T, depth: Int)
    extends Fifo(gen: T, depth: Int) {
...
```

- Explore the different implementation details in Section 11.3 of the Chisel book

Chisel `Queue`

- Chisel has a `Queue`
- Interfaces are `DecoupledIO`s
- Specify type and number of entries Queue(UInt(4.W), 8)
- Optional arguments
  - `pipe`: if full, allow concurrent enqueue and dequeue
  - `flow`: if empty, enqueued value available immediately for dequeue
- What is the implementation? Resources? Throughput?

```
val q = Module(new Queue(UInt(), 16))
q.io.enq <> producer.io.out
consumer.io.in <> q.io.deq
```

Connect a Network-on-Chip

- n x n router in bi-thorus configuration
- Each raouter has 4 + 1 ports: North, South, West, East, and local
- Use a helper function `connect(...)`

```
def connect(r1: Int, p1: Int, r2: Int, p2: Int):
    Unit = {
  net(r1).io.ports(p1).in :=
      net(r2).io.ports(p2).out
  net(r2).io.ports(p2).in :=
      net(r1).io.ports(p1).out
}

for (i <- 0 until n) {
  for (j <- 0 until n) {
    val r = i * n + j
    connect(r, EAST, i * n + (j + 1) % n, WEST)
    connect(r, SOUTH, (i + 1) % n * n + j, NORTH)
  }
}
```

Why Open Source?

- Can improve your visibility
- Add your GitHub repos to your CV
- When hiring, I look up GitHub contributions
- You can get help from the community
- I got free translations of my Chisel book
- Why not?
  - You already use open-source tools and libraries
  - So return some to the community
  - Even in a company, if there is no patent
  - Oticon contributes to Zephyr

Open-Source Licensing

- You own the copyright when creating code
  - Hardware is different in DK, so it is blurry
  - Maybe your employer owns the copyright (not at DTU)
- A license grants others to use and change your code
  - Different licenses have different permissions and restrictions
- When releasing, add a license to your repo
  - Have it clearly visible (e.g., as LICENSE in the project root)
  - GitHub will detect it

License Types

- BSD and MIT
  - Commonly used, good for academic
  - Basically, use my stuff, but don't sue me
  - Can also be used in a commercial product (e.g., network stack)
- GPL
  - Commonly used in software
  - Copyleft means you have to make all changes available
  - Some companies have restrictions on using GPL software
- Apache
  - Similar to BDS and MIT
  - Includes explicit patent protection

Documentation

- Summarize the purpose of your project
- Instruct how to use it
- List dependencies
- Good documentation:
  - May attract users
  - May even attract contributors
  - Forces you to rethink your project
- Start early with the documentation

README.md

- Well-suited for small to medium projects
- Immediately visible on GitHub
- Use markdown for simple formatting
- Include figures, best in SVG
- I will consider the README as part of the grade
- Up to now, not much is going on
  - Don’t wait till the final week!

On Chat Tutor

- AI trained on Chisel material
- Are you using it?
- Chattutor
- Marius will come for feedback on Tuesday, November 25th

On the Project

- Today is your presentation and demonstration
  - Let us use reverse the order, e.g., FFT Core first
- No lab
- No report to be written
- But I expect a very detailed README.md
- Explain what it does, how to use

Next Week

- Guest lecture by Hans Jakob on
- Chisel in Research
- Includes a lab exercise

Summary

- Collect ideas for a more productive generator code
- Did you find some Scala tricks that I did not present?
  - You could add it to the course evaluation
  - Easier, add to our shared Google docs
  - Also, what you are missing
- Use ready/valid interfaces (DecoupledIO)
- Use queues, arbitration
- Go open source!