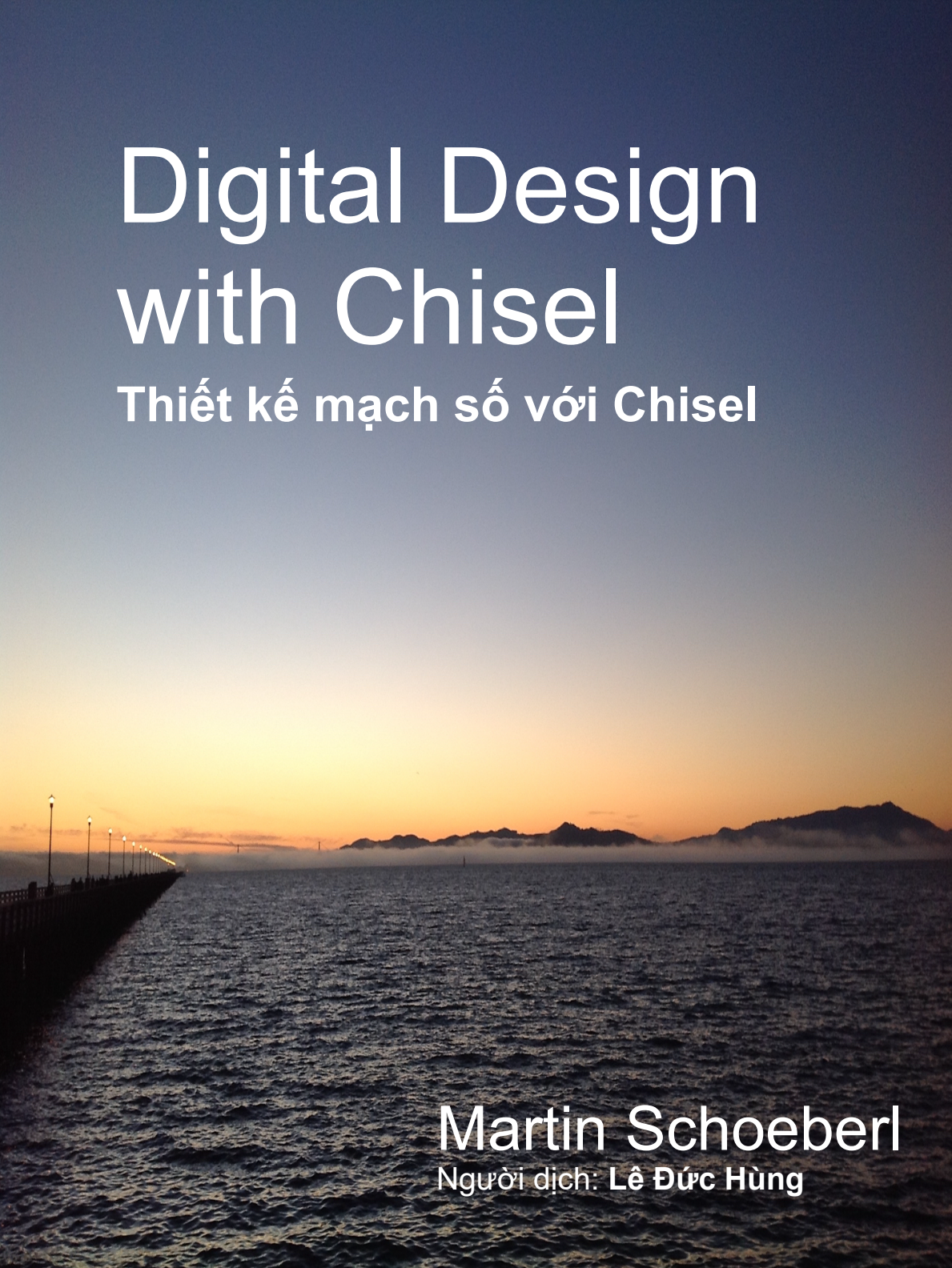
Digital Design with Chisel

Thiết kế mạch số với Chisel



Martin Schoeberl

Người dịch: **Lê Đức Hùng**

Digital Design with Chisel
Thiết kế mạch số với Chisel

Second Edition
Ấn bản lần 2

Digital Design with Chisel
Thiết kế mạch số với Chisel

Second Edition
Ấn bản lần 2

Tác giả: Martin Schoeberl

Người dịch: Lê Đức Hùng

Email: martin@jopdesign.com
Visit the source at https://github.com/schoeberl/chisel-book

Published 2019 by Kindle Direct Publishing,
https://kdp.amazon.com/

Library of Congress Cataloging-in-Publication Data

Schoeberl, Martin

Digital Design with Chisel
Martin Schoeberl
Includes bibliographical references and an index.
ISBN 9781689336031

Manufactured in the United States of America.
Typeset by Martin Schoeberl.

Mục lục

Danh sách hình vẽ

Danh sách bảng

Listings

Lời tựa

Đây là thời điểm thú vị để tham gia vào thế giới thiết kế mạch số. Với sự kết thúc của Dennard Scaling và sự chậm lại của Định luật Moore, có lẽ chưa bao giờ nhu cầu đổi mới trong lĩnh vực này lại lớn hơn như vậy. Các công ty bán dẫn tiếp tục vắt kiệt mọi hiệu suất mà họ có thể, nhưng chi phí của những cải tiến này đang tăng lên một cách đáng kể. Chisel làm giảm chi phí này bằng cách cải thiện năng suất. Nếu các nhà thiết kế có thể xây dựng nhiều hơn trong thời gian ngắn hơn, đồng thời khấu hao chi phí kiểm tra thông qua tái sử dụng, các công ty có thể dành ít chi phí hơn cho Kỹ thuật không định kỳ (NRE - Non-Recurring Engineering). Ngoài ra, cả sinh viên và những người đóng góp cá nhân đều có thể tự mình sáng tạo dễ dàng hơn.

Chisel không giống như hầu hết các ngôn ngữ ở chỗ nó được nhúng trong một ngôn ngữ lập trình khác, đó là Scala. Về cơ bản, Chisel là một thư viện có các lớp và các hàm đại diện cho các nguyên tố cần thiết để biểu diễn các mạch số, mạch đồng bộ. Thiết kế Chisel thực sự là một chương trình Scala *tạo ra* một mạch điện khi nó thực thi. Đối với nhiều người, điều này có vẻ phản cảm: "Tại sao không làm cho Chisel trở thành một ngôn ngữ độc lập như VHDL hoặc SystemVerilog?" Câu trả lời của tôi cho câu hỏi này như sau: thế giới phần mềm đã chứng kiến một lượng đổi mới lớn trong phương pháp thiết kế trong vài thập kỷ qua. Thay vì cố gắng điều chỉnh những kỹ thuật này sang một ngôn ngữ phần cứng mới, chúng tôi có thể chỉ cần *sử dụng* một ngôn ngữ lập trình hiện đại và đạt được những lợi ích đó miễn phí.

Một lời chỉ trích lâu nay đối với Chisel là nó "khó học". Phần lớn nhận thức này là do sự phổ biến của các thiết kế lớn, phức tạp được tạo ra bởi các chuyên gia để giải quyết nhu cầu nghiên cứu hoặc thương mại của riêng họ. Khi học một ngôn ngữ phổ biến như C++, người ta không bắt đầu bằng cách đọc mã nguồn của GCC. Thay vào đó, có rất nhiều khóa học, sách giáo khoa và các tài liệu học tập khác phục vụ cho người mới học. Trong cuốn sách *Digital Design with Chisel*, Martin đã tạo ra một nguồn tài nguyên quan trọng cho bất kỳ ai muốn học Chisel.

Martin là một nhà giáo dục có kinh nghiệm, và điều đó thể hiện trong cách tổ chức cuốn sách này. Bắt đầu với việc cài đặt và các thành phần nguyên bản, anh ấy gầy dựng sự hiểu biết cho người đọc như xây một tòa nhà, từng viên gạch một. Các bài tập đi kèm là lớp vữa củng cố sự hiểu biết, đảm bảo rằng mỗi khái niệm sẽ hiện rõ trong tâm trí người đọc. Cuốn sách đạt đến đỉnh cao với *các bộ tạo phần cứng* giống như một mái nhà

cho phần còn lại của cấu trúc. Cuối cùng, người đọc kết thúc cuốn sách với kiến thức để xây dựng một thiết kế đơn giản nhưng hữu ích: đó là bộ xử lý RISC.

Trong cuốn sách *Digital Design with Chisel*, Martin đã đặt nền móng vững chắc cho thiết kế mạch số hiệu quả. Những gì các bạn xây dựng được với nó thì tùy thuộc vào các bạn mà thôi.

Jack Koenig
Người bảo trì Chisel và FIRRTL
Kỹ sư nhân viên, SiFive

Lời người dịch

Cuốn sách này được dịch sang Tiếng Việt phục vụ cho cộng đồng trong lĩnh vực thiết kế mạch số nói riêng và lĩnh vực kỹ thuật điện tử máy tính nói chung. Người dịch chỉ đóng vai trò dịch thuật từ nội dung sách của tác giả sang ngôn ngữ Tiếng Việt, không chỉnh sửa về cách trình bày và văn phong của tác giả. Cuốn sách này không trình bày về ngôn ngữ lập trình Chisel/Scala, mà là về thiết kế mạch số dùng ngôn ngữ Chisel/Scala theo chuyên môn của tác giả. Vì vậy, nội dung cuốn sách khá kén người đọc. Để hiểu nội dung sách, người đọc cần có kiến thức nâng cao, biết về mạch số, một ngôn ngữ lập trình bất kỳ, và kiến trúc máy tính càng tốt.

Các nội dung sau chưa được hoàn thiện: Listing vẫn được giữ nguyên tiếng Anh vì chưa Việt hóa được trong gói Latex; các từ Tiếng Việt trong các Listing chưa được Việt hóa vì các chữ có dấu không hiển thị được sau khi biên dịch (đang tìm các khắc phục). Một số từ thuật ngữ tiếng Anh chuyên môn hay sử dụng, người dịch để từ nguyên gốc để dễ hiểu. Người dịch sẽ cố gắng Việt hóa một số thuật ngữ tiếng Anh cần thiết còn lại và hoàn thiện những thiếu sót ở những phiên bản tiếp theo.

Nội dung dịch thuật còn nhiều thiếu sót và chưa hoàn thiện. Người dịch mong muốn nhận được sự phản hồi và đóng góp để cập nhật sách vào những phiên bản sau.

Lê Đức Hùng
Phòng thí nghiệm DESLAB
Khoa Điện tử - Viễn thông
Trường Đại học Khoa học Tự nhiên, ĐHQG-HCM
Email: leduchung@gmail.com

Lời nói đầu

Cuốn sách này giới thiệu về thiết kế mạch số với trọng tâm là sử dụng ngôn ngữ xây dựng phần cứng Chisel. Chisel đưa những tiến bộ từ kỹ thuật phần mềm, chẳng hạn như ngôn ngữ lập trình hướng đối tượng và chức năng, vào trong thiết kế mạch số.

Cuốn sách này nhắm đến các nhà thiết kế phần cứng và kỹ sư phần mềm. Các nhà thiết kế phần cứng, với kiến thức về Verilog hoặc VHDL, có thể nâng cấp năng suất thiết kế với một ngôn ngữ hiện đại cho các thiết kế ASIC hoặc FPGA tiếp theo của mình. Các kỹ sư phần mềm, với kiến thức về lập trình hướng đối tượng và chức năng, có thể tận dụng kiến thức của mình để lập trình phần cứng, ví dụ các bộ tăng tốc phần cứng trên FPGA thực thi trên "đám mây".

Cách tiếp cận của cuốn sách này là trình bày các thành phần phần cứng điển hình có kích thước nhỏ đến trung bình để khám phá về thiết kế mạch số với Chisel.

Lời nói đầu cho ấn bản lần 2

Vì Chisel cho phép thiết kế phần cứng nhanh, nên việc truy cập mở và in theo yêu cầu cũng cho phép việc xuất bản cuốn sách nhanh. Chưa đầy 6 tháng sau ấn bản đầu tiên của cuốn sách này, tôi có thể đưa ra ấn bản thứ hai được cải tiến và mở rộng.

Bên cạnh những sửa chữa nhỏ, những thay đổi chính trong phiên bản thứ hai như sau. Phần thử nghiệm đã được mở rộng. Chương xây dựng các khối mạch tuần tự chứa nhiều ví dụ mạch điện hơn. Chương mới về xử lý ngõ vào giải thích sự đồng bộ hóa ngõ vào, chỉ ra cách thiết kế mạch gỡ lỗi và cách lọc tín hiệu ngõ vào bị nhiễu. Chương các thiết kế ví dụ đã được mở rộng để cho thấy các cách thực hiện khác nhau của FIFO. Các loại biến thể FIFO cũng cho thấy cách sử dụng các tham số kiểu và tính kế thừa trong thiết kế mạch số.

Lời nói đầu cho ấn bản lần 3

Chisel đã có những thay đổi trong năm ngoái, vì vậy đã đến lúc có ấn bản mới cho cuốn sách Chisel.

Chúng tôi đã thay đổi tất cả các ví dụ thành phiên bản mới nhất của Chisel (3.4.1) và đề xuất phiên bản Scala (2.12.12).

Lời cám ơn

Tôi muốn cảm ơn tất cả những người đã làm việc trên Chisel vì đã tạo ra một ngôn ngữ xây dựng phần cứng tuyệt vời. Thật hào hứng khi sử dụng Chisel và do đó cũng xứng đáng để viết một cuốn sách về nó.

Tôi cảm ơn toàn thể cộng đồng Chisel, những người luôn chào đón, thân thiện và tôi không bao giờ mệt mỏi khi trả lời các câu hỏi về Chisel.

Tôi cũng muốn cảm ơn các sinh viên của tôi trong những năm cuối của khóa học Kiến trúc máy tính nâng cao, nơi hầu hết họ đã chọn Chisel cho dự án cuối cùng. Cảm ơn các bạn đã ra khỏi vùng an toàn của mình và bắt đầu hành trình học hỏi và sử dụng ngôn ngữ mô tả phần cứng tối tân. Nhiều câu hỏi của các bạn đã giúp hình thành nên cuốn sách này.

1 Giới thiệu

Cuốn sách này giới thiệu về thiết kế hệ thống số sử dụng ngôn ngữ xây dựng phần cứng hiện đại, Chisel [2]. Trong cuốn sách này, chúng tôi tập trung vào mức trừu tượng bậc cao hơn thông thường trong các sách thiết kế số khác, nhằm cho phép người đọc xây dựng các hệ thống số tương tác, phức tạp hơn trong thời gian ngắn hơn.

Cuốn sách này và Chisel nhắm đến hai nhóm phát triển: (1) những người thiết kế phần cứng và (2) các lập trình viên phần mềm. Những người thiết kế phần cứng thông thạo VHDL hoặc Verilog và sử dụng các ngôn ngữ khác như Python, Java hoặc Tcl để tạo phần cứng có thể chuyển sang một ngôn ngữ xây dựng phần cứng duy nhất, ở đó việc tạo ra phần cứng là một phần của ngôn ngữ. Các lập trình viên phần mềm trở nên quan tâm đến thiết kế phần cứng, ví dụ: các chip trong tương lai của Intel sẽ bao gồm phần cứng khả trình để tăng tốc các chương trình. Việc các bạn sử dụng Chisel làm ngôn ngữ mô tả phần cứng đầu tiên là hoàn toàn ổn.

Chisel đưa những tiến bộ từ kỹ thuật phần mềm, chẳng hạn như ngôn ngữ lập trình hướng đối tượng và chức năng, vào trong thiết kế mạch số. Chisel không chỉ cho phép biểu diễn phần cứng ở mức-chuyển-thanh-ghi (Register-Transfer Level) mà còn cho phép viết các bộ tạo ra phần cứng.

Phần cứng hiện tại chủ yếu được mô tả bởi ngôn ngữ mô tả phần cứng. Thời đại vẽ các thành phần phần cứng, ngay khi dùng các công cụ CAD, đã qua rồi. Một số bản vẽ mức cao có thể cho một cái nhìn tổng quan về hệ thống nhưng không nhằm mục đích mô tả hệ thống. Hai ngôn ngữ mô tả phần cứng phổ biến nhất hiện nay là Verilog và VHDL. Cả hai ngôn ngữ đều đã cũ, chứa nhiều di sản và có một dòng chuyển động về những cấu trúc của ngôn ngữ có thể tổng hợp được với phần cứng. Đừng hiểu sai ý tôi: VHDL và Verilog có thể mô tả hoàn hảo một khối phần cứng được tổng hợp thành một ASIC. Với thiết kế phần cứng trong Chisel, Verilog đóng vai trò là ngôn ngữ trung gian để tổng hợp và kiểm tra mạch.

Cuốn sách này không giới thiệu tổng quan và các cơ sở về thiết kế phần cứng. Nên để được giới thiệu những cơ bản trong thiết kế mạch số, chẳng hạn cách xây dựng cổng logic từ các transistor CMOS, bạn đọc hãy tham khảo các sách thiết kế mạch số khác. Tuy nhiên, cuốn sách này dự định hướng dẫn thiết kế mạch số ở cấp độ trừu tượng, đó là các bài thực hành hiện đại để mô tả ASIC hoặc nhắm mục tiêu vào thiết kế trên FPGAs.[1]

[1] Vì tác giả quen thuộc với FPGA hơn là ASIC như công nghệ đích, nên một số tối ưu hóa thiết kế được trình

Như điều kiện tiên quyết cho cuốn sách này, chúng tôi giả định người đọc đã có kiến thức cơ bản về Đại số tuyến tính và Hệ thống số nhị phân. Hơn nữa, cũng giả định là người đọc đã có một số kinh nghiệm lập trình với bất kỳ ngôn ngữ lập trình nào. Không cần có kiến thức về Verilog hoặc VHDL. Chisel có thể là ngôn ngữ lập trình đầu tiên của bạn để mô tả phần cứng số. Vì quy trình xây dựng trong các ví dụ dựa trên `sbt` và `make`, nên các kiến thức về tương tác bằng dòng lệnh trên terminal (Unix/Linux) là hữu ích.

Bản thân Chisel không phải là một ngôn ngữ lớn. Các cấu trúc cơ bản của nó được trình bày chỉ trong một trang và có thể học được trong vòng một vài ngày. Vì vậy, cuốn sách này không phải là một cuốn sách lớn. Chisel chắc chắn nhỏ hơn VHDL và Verilog, mang nhiều di sản. Sức mạnh của Chisel đến từ việc nhúng Chisel vào bên trong Scala, mà chính nó là ngôn ngữ biểu diễn. Chisel kế thừa các đặc điểm từ Scala là "ngôn ngữ phát triển nhờ có bạn" [12]. Tuy nhiên, Scala không phải là chủ đề của cuốn sách này.

Chúng tôi cung cấp một phần ngắn về Scala cho các nhà thiết kế phần cứng. Cuốn sách được viết bởi Odersky và các cộng sự [12] cung cấp giới thiệu tổng quan về Scala. Cuốn sách này hướng dẫn về thiết kế mạch số và ngôn ngữ Chisel; nó không phải là một tài liệu tham khảo về ngôn ngữ Chisel, cũng không phải là một cuốn sách về thiết kế chip hoàn chỉnh.

Tất cả các ví dụ mã code chương trình được trình bày trong cuốn sách này đều được rút trích từ các chương trình hoàn chỉnh đã được biên dịch và thử nghiệm. Do đó, chương trình sẽ không có bất kỳ lỗi cú pháp nào. Các ví dụ có sẵn từ kho GitHub của cuốn sách này. Bên cạnh việc trình bày các mã code Chisel, chúng tôi cũng cố gắng trình bày các thiết kế hữu ích và nguyên lý của phong cách mô tả phần cứng tốt.

Cuốn sách này được tối ưu để đọc trên máy tính xách tay hoặc máy tính bảng (ví dụ như iPad). Chúng tôi đưa vào các liên kết để đọc thêm trong văn bản đang chạy, chủ yếu là các chủ đề trên Wikipedia.

1.1 Cài đặt Chisel và các công cụ FPGA

Chisel là một thư viện Scala, và cách dễ nhất để cài đặt Chisel và Scala là với `sbt`, một công cụ xây dựng Scala. Bản thân Scala phụ thuộc vào việc cài đặt Java JDK 1.8. Vì Oracle đã thay đổi cấp giấy phép cho Java, nên việc dễ dàng hơn là cài đặt OpenJDK từ AdoptOpenJDK.

bày trong cuốn sách này nhắm đến công nghệ FPGA.

1.1.1 macOS

Cài đặt Java OpenJDK 8 từ AdoptOpenJDK. Trên HĐH Mac OS X, với trình quản lý gói Homebrew, `sbt` và git có thể được cài đặt như sau:

```
$ brew install sbt git
```

Cài đặt GTKWave và IntelliJ (phiên bản cho cộng đồng). Khi nhập (Import) một dự án, **c**họn JDK 1.8 đã được cài đặt trước (không phải Java 11!)

1.1.2 Linux/Ubuntu

Cài đặt Java và các công cụ hữu ích khác trên Ubuntu với lệnh:

```
$ sudo apt install openjdk-8-jdk git make gtkwave
```

Với Ubuntu, vốn được dựa trên Debian, các chương trình thường được cài đặt từ một tập tin Debian (.deb). Tuy nhiên, tại thời điểm viết bài này, `sbt` không còn có sẵn gói để cài đặt. Do đó, quá trình cài đặt sẽ tốn nhiều bước hơn một chút như sau:

```
echo "deb https://dl.bintray.com/sbt/debian /" | \
  sudo tee -a /etc/apt/sources.list.d/sbt.list
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 \
  --recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823
sudo apt-get update
sudo apt-get install sbt
```

1.1.3 Windows

Cài đặt Java OpenJDK từ AdoptOpenJDK. Chisel và Scala còn có thể được cài đặt và được sử dụng trên Hệ điều hành (HĐH) Windows. Cài đặt GTKWave và IntelliJ (phiên bản cho cộng đồng). Khi nhập một dự án, **C**họn JDK 1.8 đã được cài đặt trước (không phải Java 11!), `sbt` có thể được cài đặt với chương trình cài đặt trên Windows, xem thêm: Cài đặt sbt trên Windows. Cài đặt chương trình con git.

1.1.4 Các công cụ FPGA

Để xây dựng phần cứng cho FPGA, bạn cần công cụ tổng hợp mạch. Hai hãng cung cấp FPGA lớn, Intel[2] và Xilinx, cung cấp các phiên bản miễn phí cho các công cụ hỗ trợ các

[2] tiền thân là Altera

FPGA có kích thước nhỏ và trung bình. Các FPGA có kích thước trung bình này đủ lớn để xây dựng các bộ vi xử lý kiểu RISC đa lõi. Intel cung cấp Quartus Prime Lite Edition và Xilinx cung cấp Vivado Design Suite, WebPACK Edition. Cả hai công cụ này đều đều có sẵn trên HĐH Windows và Linux, nhưng chưa có cho HĐH macOS.

1.2 Hello World

Mỗi cuốn sách ngôn ngữ lập trình đều bắt đầu với một ví dụ tối giản, thường được gọi là ví dụ *Hello World*. Đoạn mã sau là cách tiếp cận đầu tiên:

```
object HelloScala extends App{
  println("Hello Chisel World!")
}
```

Biên dịch và thực thi chương trình ngắn này với `sbt`

```
$ sbt run
```

dẫn đến kết quả ngõ ra mong đợi của một chương trình Hello World:

```
[info] Running HelloScala
Hello Chisel World!
```

Tuy nhiên, đây có phải là Chisel? Phần cứng này có được tạo ra để in một chuỗi không? Không, đây là mã Scala đơn giản và không phải là chương trình Hello World đại diện cho một thiết kế phần cứng.

1.3 Hello World trong Chisel

Như vậy tương đương với chương trình Hello World cho thiết kế phần cứng là gì? Thiết kế hữu ích và hiển thị tối thiểu? Một đèn LED nhấp nháy là phiên bản phần cứng (hoặc thậm chí phần mềm nhúng) của Hello World. Nếu đèn LED nhấp nháy, thì chúng ta sẵn sàng giải quyết các bài toán lớn hơn!

Listing 1.1 biểu diễn một đèn LED nhấp nháy, được mô tả bởi Chisel. Cũng không quan trọng phải hiểu chi tiết về ví dụ của đoạn mã này. Chúng tôi sẽ trình bày những điều đó trong những chương sau. Chỉ lưu ý rằng mạch điện thường được cấp xung clock ở tần số cao, ví dụ: 50MHz, và chúng ta cần một mạch đếm để lấy chu kỳ thời gian ứng với khoảng tần số trong dải Hz để có thể thấy được sự nhấp nháy của đèn. Trong ví dụ trên, chúng ta đếm từ 0 lên đến 25000000-1, sau đó chuyển đổi tín hiệu nhấp nháy (`blkReg`

```
class Hello extends Module {
  val io = IO(new Bundle {
    val led = Output(UInt(1.W))
  })
  val CNT_MAX = (50000000 / 2 - 1).U;

  val cntReg = RegInit(0.U(32.W))
  val blkReg = RegInit(0.U(1.W))

  cntReg := cntReg + 1.U
  when(cntReg === CNT_MAX) {
    cntReg := 0.U
    blkReg := ~blkReg
  }
  io.led := blkReg
}
```

Listing 1.1: Phần cứng Hello World trên Chisel

`:= ~blkReg`) và khởi tạo lại mạch đếm (`cntReg := 0.U`). Phần cứng này sẽ nhấp nháy đèn LED ở tần số khoảng 1 Hz.

1.4 Công cụ IDE cho Chisel

Cuốn sách này không đưa ra giả định nào về môi trường lập trình hoặc trình soạn thảo mà bạn sử dụng. Việc học các kiến thức cơ bản sẽ trở nên dễ dàng chỉ bằng cách sử dụng `sbt` ở dòng lệnh và một trình soạn thảo mà bạn chọn. Theo truyền thống của các cuốn sách khác, tất cả các lệnh mà các bạn nhập vào shell/terminal/CLI đều được bắt đầu bằng ký tự `$`, ký tự mà các bạn không được gõ vào. Ví dụ, đây là lệnh `ls` trên Unix, liệt kê các tập tin trong thư mục hiện tại:

```
$ ls
```

Điều đó nói rằng, một môi trường phát triển tích hợp (IDE), nơi một trình biên dịch đang chạy trong nền, có thể tăng tốc độ viết mã code. Vì Chisel là một thư viện Scala nên tất cả các IDE hỗ trợ Scala cũng là các IDE tốt cho Chisel. Các công cụ như IntelliJ và Eclipse có thể tạo một dự án từ việc cấu hình dự án sbt trong lệnh `build.sbt`.

Trong IntelliJ, các bạn có thể tạo một dự án mới từ các tập tin nguồn hiện có: *File - New - Project from Existing Sources...* và sau đó chọn tập tin `build.sbt` từ dự án.

Trong Eclipse, bạn có thể tạo một dự án từ lệnh:

```
$ sbt eclipse
```

và nhập dự án đó vào trong Eclipse.[3]

Visual Studio Code là một lựa chọn khác cho Chisel IDE. Phần tiện ích mở rộng Scala Metals cung cấp việc hỗ trợ cho Scala. Ở trên thanh bên trái chọn *Extensions*, tìm kiếm *Metals* và cài đặt *Scala (Metals)*. Để nhập một dự án dựa trên `sbt`, mở thư mục bằng cách vào *File - Open*.

1.5 Truy cập nguồn và các đặc điểm của eBook

Cuốn sách là mã nguồn mở và được đặt tại GitHub: chisel-book. Tất cả các ví dụ mã code của Chisel, được trình bày trong cuốn sách này, được đưa vào trong kho lưu trữ (repository) trên GitHub. Các đoạn mã được biên dịch với phiên bản mới của Chisel, và nhiều ví dụ mẫu cũng có chứa testbench. Chúng tôi thu thập các ví dụ Chisel lớn hơn trong kho lưu trữ kèm theo chisel-examples. Nếu các bạn tìm thấy lỗi hoặc lỗi đánh máy trong sách, cách thuận tiện nhất là yêu cầu kéo (pull request) dữ liệu về từ GitHub để kết hợp với các cải tiến của các bạn. Các bạn cũng có thể cung cấp phản hồi hoặc nhận xét để cải thiện bằng cách gửi vấn đề trên GitHub hoặc gửi email.

Cuốn sách này được cung cấp miễn phí dưới dạng sách điện tử PDF và ở dạng in truyền thống. Phiên bản sách điện tử liên kết đến các tài nguyên khác và các mục trên Wikipedia. Chúng tôi sử dụng các mục trên Wikipedia cho thông tin cơ bản (ví dụ: hệ thống số nhị phân) không phù hợp trực tiếp với cuốn sách này. Chúng tôi đã tối ưu hóa định dạng của sách điện tử để đọc trên máy tính bảng, chẳng hạn như iPad.

1.6 Đọc thêm

Đây là danh sách đọc thêm cho thiết kế mạch số và Chisel:

- Digital Design: A Systems Approach, bởi tác giả William J. Dally and R. Curtis Harting, là sách giáo khoa hiện đại về thiết kế mạch số. Sách có sẵn ở hai phiên bản: sử dụng Verilog hoặc VHDL như một ngôn ngữ mô tả phần cứng.

Tài liệu chính thức của Chisel và các tài liệu khác có sẵn trực tuyến:

- Trang chủ Chisel là điểm khởi đầu chính thức để tải và học Chisel.

[3] Chức năng này cần plugin Eclipse cho sbt.

- Trang Chisel Tutorial cung cấp một dự án thiết lập sẵn chứa các bài tập nhỏ với các bộ kiểm tra và bài giải.
- Trang Chisel Wiki chứa hướng dẫn sử dụng ngắn về Chisel và các liên kết đến các thông tin khác.
- Trang Chisel Testers có kho lưu trữ riêng chứa tài liệu Wiki.
- Trang Generator Bootcamp là khóa học Chisel tập trung vào các bộ tạo phần cứng, dạng sổ ghi chép Jupyter
- Trang Chisel Style Guide bởi Christopher Celio.
- Trang chisel-lab chứa các bài tập Chisel cho khóa học "Digital Electronics 2" tại trường Technical University of Denmark.

1.7 Bài tập

Mỗi chương kết thúc với một bài tập thực hành. Với bài tập ở chương giới thiệu, chúng ta sử dụng một bo mạch FPGA để làm cho một đèn LED nhấp nháy.[4] Bước đầu tiên tiến hành clone thư mục chisel-examples từ GitHub. Ví dụ Hello World nằm trong thư mục `hello-world`, được thiết lập như một dự án nhỏ. Các bạn có thể khám phá mã Chisel của đèn LED nhấp nháy trong `src/main/scala/Hello.scala`. Biên dịch chương trình đèn LED nhấp nháy theo các bước sau:

```
$ git clone https://github.com/schoeberl/chisel-examples.git
$ cd chisel-examples/hello-world/
$ sbt run
```

Sau khi tải xuống một số thành phần Chisel ban đầu, chương trình sẽ tạo ra tập tin Verilog `Hello.v`. Khám phá tập tin Verilog này. Các bạn sẽ thấy nó chứa hai ngõ vào `clock` và `reset` và một ngõ ra `io_led`. Khi các bạn so sánh tập tin Verilog này với mô-đun của Chisel, các bạn lưu ý rằng mô-đun Chisel không chứa tín hiệu `clock` hoặc `reset`. Những tín hiệu đó được tạo ra một cách ngầm định, và trong hầu hết các thiết kế, thật tiện lợi khi không cần phải xử lý những chi tiết ở mức thấp này. Chisel cung cấp các thành phần thanh ghi, và chúng được kết nối một cách tự động với `clock` và `reset` (nếu cần thiết).

[4] Nếu hiện tại các các bạn không có bo mạch FPGA nào, hãy cứ tiếp tục đọc vì chúng tôi sẽ cho các bạn xem một phiên bản mô phỏng ở cuối bài tập.

Bước tiếp theo là thiết lập tập tin trong dự án FPGA cho công cụ tổng hợp mạch, gán các chân, biên dịch[5] chương trình Verilog, và cấu hình FPGA với tập tin dùng để nạp. Chúng tôi không thể cung cấp chi tiết các bước thực hiện này. Vui lòng tham khảo hướng dẫn sử dụng công cụ Intel Quartus hoặc Xilinx Vivado mà các bạn đang sử dụng. Tuy nhiên, các ví dụ mẫu có chứa một số dự án Quartus có sẵn để sử dụng trong thư mục `quartus` cho các bo mạch FPGA phổ biến (ví dụ: DE2-115). Nếu các ví dụ này có các dự án hỗ trợ cho các bo mạch FPGA mà các bạn đang có, thì chỉ việc chạy Quartus, mở dự án, biên dịch bằng cách nhấn nút *Play*, và cấu hình cho bo mạch FPGA với nút *Programmer* và một trong số các LED trên bo mạch sẽ nhấp nháy.

Chúc mừng! Bạn đã thực hiện được thiết kế đầu tiên của mình bằng ngôn ngữ Chisel chạy trên FPGA!

Nếu đèn LED không nhấp nháy, thì hãy kiểm tra trạng thái của chân reset. Trên cấu hình của bo mạch DE2-115, tín hiệu ngõ vào reset được nối vào SW0.

Bây giờ thay đổi tần số nhấp nháy đèn LED thành giá trị nhanh hơn hay chậm hơn và chạy lại quy trình biên dịch và cấu hình FPGA. Tần số nhấp nháy và các mẫu nhấp nháy cũng báo hiệu "các trạng thái" khác nhau. Ví dụ: các tín hiệu LED nhấp nháy chậm báo hiệu mọi thứ đều ổn, các tín hiệu LED nhấp nháy nhanh báo hiệu trạng thái cảnh báo. Hãy khám phá giá trị tần số nào biểu diễn tốt nhất hai trạng thái đó.

Xem như phần mở rộng với nhiều thử thách hơn cho bài tập, hãy tạo mẫu nhấp nháy sau: đèn LED sáng mỗi 200 ms. Đối với mẫu này, các bạn có thể tách sự thay đổi của đèn LED nhấp nháy bằng việc reset lại bộ đếm. Các bạn sẽ cần một hằng số thứ hai thay đổi trạng thái của thanh ghi `blkReg`. Vậy loại trạng thái nào mà mẫu này tạo ra ? Nó là tín hiệu báo động hay giống tín hiệu sự sống hơn ?

Nếu chưa có bo mạch FPGA, thì các bạn vẫn có thể chạy ví dụ đèn LED nhấp nháy. Các bạn sẽ sử dụng công cụ mô phỏng Chisel. Để tránh thời gian mô phỏng quá lâu, thay đổi tần số xung clock trong đoạn mã Chisel từ 50000000 xuống 50000. Thực thi lệnh sau để mô phỏng đèn LED nhấp nháy:

```
$ sbt test
```

Lệnh này sẽ thi hành chương trình kiểm tra, chạy trong một triệu chu kỳ xung clock Tần số nhấp nháy phụ thuộc vào tốc độ mô phỏng, tức là phụ thuộc vào tốc độ máy tính của bạn. Vì vậy, các bạn phải cần thử nghiệm một chút với tần số giả định để thấy đèn LED nhấp nháy được mô phỏng.

[5] Quy trình thực được chi tiết hóa hơn với các bước sau: tổng hợp mạch logic, thực hiện đặt và đi dây, thực hiện phân tích thời gian, vào tạo ra tập tin để nạp. Tuy nhiên, với mục đích của ví dụ trong chương giới thiệu này, chúng tôi chỉ đơn giản gọi là "biên dịch" chương trình.

2 Các thành phần cơ bản

Trong phần này, chúng tôi giới thiệu các thành phần cơ bản cho thiết kế mạch số: Các mạch tổ hợp và Flip-flop. Các thành phần thiết yếu này có thể được kết hợp để hình thành các mạch điện lớn hơn và thú vị hơn.

Các hệ thống mạch số nói chung được xây dựng với việc sử dụng tín hiệu nhị phân, có nghĩa là một bit đơn hoặc tín hiệu chỉ có thể có một trong hai giá trị. Các giá trị này thường được gọi là 0 và 1. Tuy nhiên, chúng tôi cũng sử dụng các thuật ngữ sau: mức thấp/cao, giá trị sai/đúng và phủ định/khẳng định. Các thuật ngữ này có nghĩa là hai giá trị có thể giống nhau của tín hiệu nhị phân.

2.1 Các loại tín hiệu và hằng số

Chisel cung cấp ba kiểu dữ liệu để mô tả tín hiệu, logic tổ hợp và các thanh ghi: `Bits`, `UInt`, và `SInt`. `UInt` và `SInt` mở rộng `Bits`, và tất cả ba kiểu này biểu diễn véc-tơ của các bit. `UInt` cung cấp cho véc-tơ bit này ý nghĩa của một số nguyên không dấu và `SInt` của một số nguyên có dấu.[1] Chisel sử dụng Số bù 2 biểu diễn số nguyên có dấu.

Sau đây là định nghĩa các kiểu khác, một kiểu `Bits` 8-bit, một số nguyên không dấu 8-bit, và một số nguyên có dấu 10-bit:

```
Bits(8.W)
UInt(8.W)
SInt(10.W)
```

Độ rộng của véc-tơ bit được định nghĩa bởi kiểu độ rộng của Chisel (`Width`). Biểu thức sau chuyển đổi số nguyên Scala `n` thành `width` kiểu Chisel, được sử dụng cho định nghĩa của véc-tơ `Bits`:

```
n.W
Bits(n.W)
```

[1] Kiểu `Bits` trong phiên bản hiện tại của Chisel đang thiếu các phép toán vì vậy không hữu ích cho đoạn mã của người sử dụng.

Các hằng số có thể được định nghĩa bằng cách sử dụng số nguyên Scala và đổi nó thành kiểu Chisel:

```
0.U  // defines a UInt constant of 0
-3.S // defines a SInt constant of -3
```

Các hằng số còn có thể được định nghĩa bởi độ rộng, bằng cách sử dụng kiểu độ rộng của Chisel:

```
3.U(4.W) // An 4-bit constant of 3
```

Nếu các bạn thấy khái niệm của 3.U và 4.W hơi buồn cười, hãy xem nó như một biến thể của một hằng số nguyên có kiểu. Khái niệm này tương tự với 8L, biểu diễn một hằng số nguyên dài trong ngôn ngữ C, Java, and Scala.

Những lỗi có thể xảy ra: Một lỗi có thể xảy ra khi định nghĩa các hằng số với độ rộng định sẵn bị thiếu thông số `.W` cho giá trị độ rộng. Ví dụ: `1.U(32)` sẽ *không* định nghĩa hằng số rộng 32-bit biểu diễn số 1. Thay vào đó, biểu thức `(32)` được hiểu như trích xuất bit từ vị trí 32, dẫn đến một hằng số bit duy nhất là 0. Đó có thể không phải là ý định ban đầu của người lập trình.

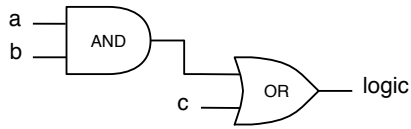
Chisel hưởng lợi từ những suy luận của các kiểu dữ liệu Scala và ở nhiều chỗ, kiểu thông tin có thể bị bỏ sót. Điều tương tự cũng đúng cho độ rộng bit. Trong nhiều trường hợp, Chisel tự động suy ra độ rộng chính xác. Vì vậy, mô tả phần cứng của Chisel súc tích hơn và dễ đọc hơn so với VHDL hoặc Verilog.

Đối với những hằng số được định nghĩa trong những cơ số khác với hệ thập phân, hằng số được định nghĩa trong một chuỗi với ký tự `h` đứng trước cho hệ thập lục (cơ số 16), `o` cho hệ bát phân (cơ số 8), và `b` cho hệ nhị phân (cơ số 2). Ví dụ sau cho thấy định nghĩa của hằng số 255 với những cơ số khác nhau. Trong ví dụ này, chúng tôi bỏ qua độ rộng bit và Chisel sẽ suy ra độ rộng tối thiểu để phù hợp với các hằng số, trong trường hợp này là 8-bit.

```
"hff".U         // hexadecimal representation of 255
"o377".U        // octal representation of 255
"b1111_1111".U // binary representation of 255
```

Đoạn mã trên cho thấy cách sử dụng dấu gạch dưới để nhóm các chữ số trong chuỗi để biểu diễn một hằng số. Dấu gạch dưới được bỏ qua.

Để biểu diễn các giá trị logic, Chisel định nghĩa kiểu `Bool`. `Bool` có thể biểu diễn giá trị *true* hoặc *false*. Đoạn mã sau cho thấy định nghĩa của kiểu `Bool` và định nghĩa của hằng số `Bool`, bằng cách đổi hằng số Boolean Scala `true` và `false` thành hằng số Chisel `Bool`.



Hình 2.1: Mạch logic cho biểu thức `(a & b) | c`. Các đường nối dây có thể là bit đơn hoặc nhiều bit. Biểu thức trong Chisel, và bản vẽ mạch là như nhau.

```
Bool()
true.B
false.B
```

2.2 Mạch tổ hợp

Chisel sử dụng các toán tử Đại số Boole, như đã được định nghĩa trong C, Java, Scala, và trong nhiều ngôn ngữ lập trình khác, ví dụ cổng logic trong mạch tổ hợp được mô tả như sau: `&` là toán tử AND và `|` là toán tử OR. Dòng mã sau định nghĩa mạch điện kết hợp các tín hiệu `a` và `b` dùng cổng *and* và kết hợp kết quả ngõ ra với tín hiệu `c` dùng cổng *or*.

```
val logic = (a & b) | c
```

Hình 2.1 biểu diễn mạch điện của biểu thức mạch tổ hợp này. Lưu ý rằng mạch điện này có thể dành cho vectơ các bit và không chỉ các dây đơn được kết hợp với các mạch AND và OR.

Trong ví dụ này, chúng ta không định nghĩa kiểu dữ liệu, cũng không phải độ rộng của tín hiệu `logic`. Cả hai đều được suy ra từ kiểu và độ rộng của biểu thức. Các phép toán logic chuẩn trong Chisel là:

```
val and = a & b // bitwise and
val or = a | b  // bitwise or
val xor = a ^ b // bitwise xor
val not = ~a    // bitwise negation
```

Các phép toán số học sử dụng các toán tử chuẩn:

```
val add = a + b // addition
val sub = a - b // subtraction
val neg = -a    // negate
```

```
val mul = a * b // multiplication
val div = a / b // division
val mod = a % b // modulo operation
```

Độ rộng kết quả của phép toán là độ rộng tối đa của các toán tử cho phép cộng và phép trừ, tổng của hai độ rộng cho phép nhân và thường là độ rộng của tử số cho phép chia và phép toán chia lấy dư.[2]

Một tín hiệu ban đầu có thể được định nghĩa dưới dạng một số kiểu như `Wire`. Sau đó, chúng ta có thể gán một giá trị với dây nối bởi toán tử cập nhật `:=`.

```
val w = Wire(UInt())

w := a & b
```

Một bit đơn có thể được trích xuất như sau:

```
val sign = x(31)
```

Một trường con có thể được trích xuất từ vị trí đầu đến cuối:

```
val lowByte = largeWord(7, 0)
```

Các trường bit được ghép nối bởi `Cat`.

```
val word = Cat(highByte, lowByte)
```

Bảng 2.2 trình bày danh sách đầy đủ của các toán tử (xem thêm tại các toán tử nội tại). Mức độ ưu tiên của toán tử Chisel được xác định bởi thứ tự đánh giá của mạch, theo thứ tự ưu tiên của toán tử Scala. Nếu còn nghi ngờ, việc sử dụng dấu ngoặc đơn luôn là một phương pháp hay.[3]

Bảng 2.2 trình bày các hàm khác nhau được định nghĩa trên và cho các kiểu dữ liệu Chisel

2.2.1 Mạch đa hợp

Mạch đa hợp là mạch chọn một trong nhiều ngõ vào đưa tới một ngõ ra (mạch đa hợp

[2] Chi tiết chính xác có trong Đặc tả FIRRTL.

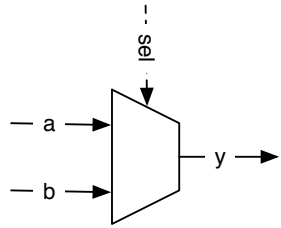
[3] Độ ưu tiên toán tử trong Chisel là một tác dụng phụ của việc chi tiết hóa phần cứng khi cấu trúc cây của các nút phần cứng được tạo ra bằng cách thực thi các toán tử Scala. Toán tử ưu tiên trong Scala tương tự nhưng không giống với Java/C. Verilog có cùng độ ưu tiên toán tử như C, nhưng VHDL lại có độ ưu tiên khác. Verilog có thứ tự ưu tiên cho các phép toán logic, nhưng trong VHDL các toán tử đó có cùng thứ tự ưu tiên và được đánh giá từ trái sang phải.

Toán tử	Mô tả	Kiểu dữ liệu
`* / %`	nhân, chia, chia lấy dư	UInt, SInt
`+ -`	cộng, trừ	UInt, SInt
`=== =/=`	bằng, không bằng	UInt, SInt, returns Bool
`> >= < <=`	so sánh	UInt, SInt, trả về Bool
`« »`	dịch trái, dịch phải (mở rộng dấu cho SInt)	UInt, SInt
`~`	NOT	UInt, SInt, Bool
`& | ^`	AND, OR, XOR	UInt, SInt, Bool
`!`	logic NOT	Bool
`&& ||`	logic AND, OR	Bool

Bảng 2.1: Các toán tử phần cứng được định nghĩa trên Chisel.

Hàm	Mô tả	Kiểu dữ liệu
`v.andR v.orR v.xorR`	AND, OR, XOR rút gọn	UInt, SInt, trả về Bool
`v(n)`	trích xuất một bit đơn	UInt, SInt
`v(end, start)`	trích xuất trường bit	UInt, SInt
`Fill(n, v)`	sao chép chuỗi bit, n times	UInt, SInt
`Cat(a, b, ...)`	ghép các trường bit	UInt, SInt

Bảng 2.2: Các hàm phần cứng được định nghĩa trên Chisel, được gọi trên `v`.



Hình 2.2: Mạch đa hợp cơ bản 2:1.

còn được gọi là mạch dồn kênh, mạch ghép kênh). Ở dạng cơ bản nhất, mạch gồm có 2 ngõ vào. Hình 2.2 biểu diễn mạch đa hợp 2:1, hay gọi là mux cho ngắn gọn. Tùy thuộc vào giá trị của tín hiệu lựa chọn (tín hiệu `sel`), tín hiệu y sẽ biểu diễn tín hiệu a hoặc tín hiệu b.

Mạch đa hợp có thể được hình thành từ các cổng logic. Tuy nhiên, vì đa hợp là một phép toán chuẩn nên Chisel cung cấp mạch đa hợp,

```
val result = Mux(sel, a, b)
```

trong đó `a` được chọn khi `sel` là `true.B`, ngược lại `b` sẽ được chọn. Kiểu dữ liệu `sel` là Chisel `Bool`; các ngõ vào `a` và `b` có thể là bất kỳ kiểu cơ số Chisel hoặc kiểu tập hợp (bundles hoặc vectors) miễn là chúng cùng kiểu.

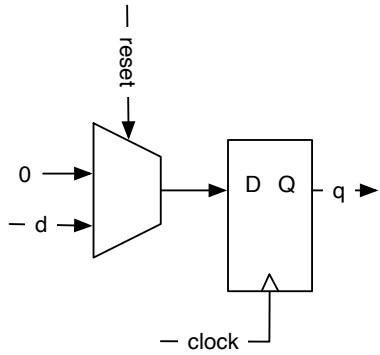
Với các phép toán logic và số học và đa hợp, mỗi một mạch đa hợp có thể được mô tả. Tuy nhiên, Chisel cung cấp thêm các thành phần và điều khiển trừu tượng để mô tả một cách thanh thoát hơn về mạch tổ hợp, sẽ được mô tả trong chương sau.

Thành phần cơ bản thứ hai cần để mô tả mạch số là các thành phần trạng thái, hay còn được gọi là thanh ghi, sẽ được mô tả ở phần tiếp theo.

2.3 Các thanh ghi

Chisel cung cấp thanh ghi, là tập hợp của các flip-flop D. Thanh ghi được kết nối ngầm với xung clock toàn cục và được cập nhật trạng thái ở cạnh lên xung clock. Khi giá trị khởi tạo được cung cấp lúc khai báo thanh ghi, nó sử dụng reset đồng bộ được nối với tín hiệu reset toàn cục. Một thanh ghi ghi có thể là bất kỳ kiểu Chisel nào có thể được biểu diễn ở dạng tập hợp các bit. Đoạn mã sau định nghĩa một thanh ghi 8-bit, được khởi tạo bằng 0 lúc reset:

```
val reg = RegInit(0.U(8.W))
```



Hình 2.3: Flip-flop D dựa trên thanh ghi với reset đồng bộ về 0.

Ngõ vào được nối với thanh ghi bởi toán tử cập nhật `:=` và ngõ ra của thanh ghi có thể được sử dụng chỉ với tên trong biểu thức :

```
reg := d
val q = reg
```

Một thanh ghi còn có thể được kết nối với chính ngõ vào của nó ở định nghĩa sau:

```
val nextReg = RegNext(d)
```

Hình 2.3 biểu diễn mạch điện của định nghĩa thanh ghi với xung clock, tín hiệu reset đồng bộ với `0.U`, ngõ vào `d`, và ngõ ra `q`. Các tín hiệu toàn cục `clock` và `reset` được kết nối ngầm với mỗi thanh ghi được định nghĩa.

Một thanh ghi còn có thể được kết nối với ngõ vào của nó và một hằng số như giá trị khởi tạo ở định nghĩa sau:

```
val bothReg = RegNext(d, 0.U)
```

Để phân biệt giữa các tín hiệu biểu diễn mạch logic tổ hợp và thanh ghi, một thực tế phổ biến là đặt hậu tố các tên thanh ghi bởi `Reg`. Một thực tế khác, đến từ Java và Scala, là sử dụng camelCase cho việc định danh bao gồm nhiều từ. Quy ước là bắt đầu các hàm và biến bằng chữ thường và bắt đầu các lớp (các kiểu) bằng chữ in hoa.

2.3.1 Đếm

Đếm là một hoạt động cơ bản trong các hệ thống số. Nó có thể đếm các sự kiện. Tuy nhiên, đếm thường được sử dụng hơn để xác định một khoảng thời gian. Đếm chu kỳ xung clock và kích hoạt một hoạt động khi khoảng thời gian đã hết.

Một cách tiếp cận đơn giản là đếm lên một giá trị. Tuy nhiên, trong khoa học máy tính, và thiết kế mạch số, việc đếm bắt đầu từ 0. Do đó, nếu chúng ta muốn đếm đến 10, thì chúng ta đếm từ 0 đến 9. Đoạn mã sau đây cho thấy một mạch đếm như vậy: đếm đến 9 và quay về 0 khi đạt đến 9.

```
  val cntReg = RegInit(0.U(8.W))

  cntReg := Mux(cntReg === 9.U, 0.U, cntReg + 1.U)
```

2.4 Cấu trúc với Bundle và Vec

Chisel cung cấp hai cấu trúc để nhóm các tín hiệu liên quan với nhau: (1) `Bundle` để nhóm các tín hiệu có kiểu khác nhau và (2) `Vec` để biểu diễn tập hợp các tín hiệu cùng kiểu có thể truy xuất được. Các `Bundle` và `Vec` có thể được lồng vào nhau tùy ý.

Một bundle trong Chisel nhóm nhiều tín hiệu. Toàn thể bundle có thể được tham chiếu toàn bộ chung, hoặc các trường riêng rẽ có thể được truy cập bởi tên của chúng. Chúng ta có thể định nghĩa một bundle (tập hợp của các tín hiệu) bằng cách định nghĩa một lớp mở rộng `Bundle` và liệt kê các trường như `val`s trong khối cấu trúc.

```
class Channel() extends Bundle {
  val data = UInt(32.W)
  val valid = Bool()
}
```

Để sử dụng bundle, chúng ta tạo nó bởi từ khóa `new` và bọc (wrap) nó thành một kiểu `Wire`. Các trường được truy cập bởi dấu chấm:

```
  val ch = Wire(new Channel())
  ch.data := 123.U
  ch.valid := true.B

  val b = ch.valid
```

Ký hiệu dấu chấm phổ biến trong các ngôn ngữ hướng đối tượng, với `x.y` có nghĩa `x` là một tham chiếu đến đối tượng và `y` là trường của đối tượng đó. Vì Chisel là hướng đối

tượng, nên chúng ta sử dụng ký hiệu dấu chấm để truy cập các trường trong bundle. Bundle tương tự với `struct` trong C, `record` trong VHDL, hoặc `struct` trong SystemVerilog. Bundle cũng có thể được tham chiếu một cách tổng thể:

```
val channel = ch
```

`Vec` trong Chisel biểu diễn tập hợp các tín hiệu có cùng kiểu (véc-tơ). Mỗi thành phần có thể được truy cập bởi một chỉ số. `Vec` trong Chisel tương tự với cấu trúc mảng dữ liệu trong các ngôn ngữ lập trình khác.[4] `Vec` được tạo bằng cách gọi cấu trúc với hai tham số: số phần tử và kiểu phần tử. `Vec` tổ hợp cần được bọc thành kiểu `Wire`.

```
val v = Wire(Vec(3, UInt(4.W)))
```

Các thành phần riêng rẽ được truy cập bởi `(index)`.

```
v(0) := 1.U
v(1) := 3.U
v(2) := 5.U

val idx = 1.U(2.W)
val a = v(idx)
```

Một véc-tơ được bọc thành `Wire` là mạch đa hợp. Chúng ta có thể bọc một véc-tơ thành một thanh ghi để định nghĩa một mảng các thanh ghi. Ví dụ sau định nghĩa một tập thanh ghi cho một bộ xử lý; 32 thanh ghi, mỗi thanh ghi 32-bit như trong bộ xử lý 32-bit RISC, giống phiên bản 32-bit của RISC-V.

```
val registerFile = Reg(Vec(32, UInt(32.W)))
```

Một phần tử của tập thanh ghi đó được truy cập bằng một chỉ số và được sử dụng như một thanh ghi bình thường.

```
registerFile(idx) := dIn
val dOut = registerFile(idx)
```

Chúng ta có thể tự do trộn bundle và véc-tơ với nhau. Khi tạo một véc-tơ với kiểu bundle, chúng ta có thể chuyển một nguyên mẫu của các trường véc-tơ. Sử dụng `Channel`, đã được định nghĩa ở trên, chúng ta có thể tạo véc-tơ của các kênh với:

```
val vecBundle = Wire(Vec(8, new Channel()))
```

Một bundle cũng có thể chứa một véc-tơ:

[4] Tên `Array` đã được sử dụng trong Scala.

```
class BundleVec extends Bundle {
  val field = UInt(8.W)
  val vector = Vec(4,UInt(8.W))
}
```

Khi chúng ta muốn một thanh ghi của kiểu bundle cần một giá trị reset, đầu tiên cần tạo `Wire` của bundle đó, đặt giá trị các trường riêng lẻ nếu cần, và sau đó chuyển bundle này thành `RegInit`:

```
val initVal = Wire(new Channel())

initVal.data := 0.U
initVal.valid := false.B

val channelReg = RegInit(initVal)
```

Với sự kết hợp của `Bundles` và `Vecs` chúng ta có thể định nghĩa các cấu trúc dữ liệu riêng của mình, đó là những sự trừu tượng mạnh mẽ.

2.5 Chisel tạo phần cứng

Sau khi tham khảo một số mã code ban đầu của Chisel, chúng ta thấy nó tương tự như các ngôn ngữ lập trình cổ điển như Java hay C. Tuy nhiên, Chisel (hoặc bất kỳ các ngôn ngữ mô tả phần cứng khác) định nghĩa các thành phần phần cứng. Trong một chương trình phần mềm, một dòng code chỉ được thi hành sau khi dòng khác được thi hành, trong khi đó trong phần cứng tất cả các dòng code đều được *thực thi song song*.

Điều quan trọng cần nhớ là mã Chisel tạo ra phần cứng. Hãy thử tưởng tượng, hoặc vẽ trên một tờ giấy, các khối riêng lẻ được tạo ra bởi mô tả mạch điện Chisel của các bạn. Mỗi lần tạo thành phần sẽ thêm phần cứng; mỗi câu lệnh gán tạo ra các cổng và/hoặc các flip-flop.

Về mặt kỹ thuật, khi Chisel thực thi mã của các bạn, nó sẽ chạy như một chương trình Scala, và bằng cách thực thi các câu lệnh Chisel, nó *thu thập* các thành phần phần cứng và kết nối các nút đó lại với nhau. Mạng lưới các nút phần cứng này là phần cứng, có thể thành mã Verilog để tổng hợp ASIC hoặc FPGA hoặc có thể được kiểm tra bằng trình kiểm tra Chisel. Mạng các nút phần cứng được thực thi hoàn toàn song song.

Đối với một kỹ sư phần mềm, hãy tưởng tượng sự song song to lớn này có thể tạo ra trong phần cứng mà không cần phân vùng ứng dụng thành các luồng và nhận khóa chính xác cho giao tiếp.

2.6 Bài tập

Trong phần giới thiệu, các bạn đã thực hiện đèn LED nhấp nháy trên bo mạch FPGA (ở thư mục chisel-examples), đó là một ví dụ *Hello World* hợp lý về phần cứng. Nó chỉ sử dụng các trạng thái bên trong, một đèn LED ngõ ra, và không có ngõ vào. Chép dự án này vào một thư mục mới và mở rộng nó bằng cách thêm một số ngõ vào ở `io Bundle` với `val sw = Input(UInt(2.W))`.

```
val io = IO(new Bundle {
  val sw = Input(UInt(2.W))
  val led = Output(UInt(1.W))
})
```

Với những công tắc này, các bạn còn cần gán tên chân cho bo mạch FPGA. Các bạn có thể tìm ví dụ gán chân trên tập tin dự án Quartus qua dự án ALU (Ví dụ: trên bo mạch FPGA DE2-115).

Khi các bạn đã định nghĩa các ngõ vào đó và gán chân, hãy bắt đầu với một bài kiểm tra đơn giản: loại bỏ tất cả logic nhấp nháy khỏi thiết kế và kết nối một công tắc với ngõ ra LED; biên dịch và cấu hình thiết bị FPGA. Các bạn có thể gạt công tắc đèn LED bật hay tắt không? Nếu có, bạn đã có các ngõ vào có sẵn. Nếu không, bạn cần gỡ lỗi cấu hình FPGA của mình. Việc gán chân cũng có thể được thực hiện với phiên bản GUI của công cụ.

Bây giờ sử dụng hai công tắc và thực hiện một trong những hàm mạch tổ hợp cơ bản, ví dụ: hai công tắc cho hai ngõ vào AND và hiển thị kết quả trên đèn LED. Thay đổi chức năng. Bước tiếp theo liên quan đến ba công tắc ngõ vào để thực hiện mạch đa hợp: một công tắc hoạt động như tín hiệu chọn và hai công tắc còn lại là hai ngõ vào cho mạch đa hợp 2:1.

Bây giờ, các bạn đã có thể thực hiện các hàm của mạch tổ hợp đơn giản và kiểm tra chúng trong phần cứng thực sự trên FPGA. Bước tiếp theo, chúng ta sẽ xem làm thế nào quá trình xây dựng hoạt động để tạo cấu hình cho FPGA. Hơn nữa, chúng ta cũng sẽ khám phá một khung sườn kiểm tra đơn giản từ Chisel, cho phép các bạn kiểm tra mạch điện mà không cần cấu hình FPGA và chuyển đổi công tắc.

3 Xây dựng quy trình và kiểm tra

Để bắt đầu với các đoạn mã Chisel thú vị hơn, trước tiên chúng ta cần học cách biên dịch các chương trình Chisel, cách tạo mã Verilog để thực thi trong FPGA và cách viết các bài kiểm tra để gỡ lỗi và kiểm tra rằng các mạch điện của chúng ta có hoạt động đúng hay không.

Chisel được viết bằng Scala, vì vậy bất kỳ quá trình xây dựng nào hỗ trợ Scala đều có thể thực hiện được với dự án Chisel. Một công cụ xây dựng phổ biến cho Scala là sbt, viết tắt của công cụ xây dựng tương tác Scala (Scala interactive Build Tool). Bên cạnh việc thúc đẩy quá trình xây dựng và kiểm tra, `sbt` cũng sẽ tải xuống phiên bản đúng của thư viện Scala và Chisel.

3.1 Xây dựng dự án với sbt

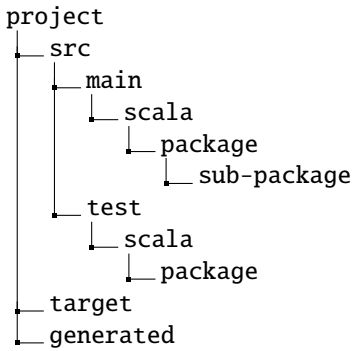
Thư viện Scala đại diện cho Chisel và trình kiểm tra Chisel được tự động tải xuống trong quá trình xây dựng từ kho lưu trữ Maven. Các thư viện được tham chiếu bởi `build.sbt`. Có thể cấu hình `build.sbt` bằng `last.release` để luôn luôn sử dụng phiên bản Chisel cập nhật nhất. Tuy nhiên, điều này có nghĩa là trên mỗi lần build (xây dựng), phiên bản được tra cứu từ kho lưu trữ Maven. Việc tra cứu này cần có kết nối Internet để việc build thành công. Tốt hơn hãy sử dụng phiên bản Chisel chuyên dụng và tất cả các thư viện Scala khác trong `build.sbt` của bạn. Có lẽ đôi khi cũng tốt khi có thể viết mã code phần cứng và kiểm tra nó mà không cần kết nối Internet. Ví dụ, thật tuyệt khi thiết kế phần cứng ở trên máy bay.

3.1.1 Tổ chức nguồn

`sbt` kế thừa các quy ước nguồn từ công cụ build tự động Maven. Maven còn tổ chức các kho thư viện Java mã nguồn mở.[1]

Hình 3.1 biểu diễn tổ chức của cấu trúc cây nguồn (source tree) của một dự án Chisel tiêu biểu. Thư mục gốc của dự án là thư mục nhà (thư mục home) của dự án, chứa tập

[1] Đó cũng là nơi các bạn đã tải xuống thư viện Chisel trong lần build đầu tiên của mình: `https://mvnrepository.com/artifact/edu.berkeley.cs/chisel3`.



Hình 3.1: Cấu trúc cây nguồn của một dự án Chisel (sử dụng `sbt`)

tin `build.sbt`. Nó còn bao gồm tập tin `Makefile` cho quy trình build, tập tin README, và một tập tin LICENSE. Thư mục `src` chứa tất cả mã nguồn. Từ đó, nó được tách giữa thư mục `main`, chứa nguồn phần cứng và thư mục `test` chứa trình kiểm tra. Chisel thừa hưởng từ Scala, vốn kế thừa từ Java cấu trúc nguồn trong các gói. Các gói tổ chức mã Chisel của bạn trong các không gian tên. Các gói còn có thể chứa các gói con. Thư mục `target` chứa các tập tin lớp và các tập tin khác được tạo ra. Tôi khuyên là cũng nên sử dụng một thư mục chứa các tập tin Verilog được tạo ra, thường gọi là thư mục `generated`.

Để sử dụng các phương diện trong không gian tên của Chisel, các bạn cần khai báo lớp/mô-đun được định nghĩa trong gói, trong ví dụ này `mypack`:

```
package mypack

import chisel3._

class Abc extends Module {
  val io = IO(new Bundle{})
}
```

Lưu ý rằng trong ví dụ này, chúng ta thấy việc nhập của gói `chisel3` để sử dụng các lớp Chisel.

Để sử dụng mô-đun `Abc` trong một ngữ cảnh khác (gói không gian tên), các thành phần của gói `mypack` cần phải được nhập vào. Dấu gạch dưới (_) đóng vai trò là ký tự đại diện, nghĩa là tất cả các lớp của `mypack` được nhập.

```
import mypack._
```

```
class AbcUser extends Module {
  val io = IO(new Bundle{})

  val abc = new Abc()

}
```

Cũng có khả năng là không nhập tất cả các kiểu từ `mypack`, nhưng sử sụng tên đủ điều kiện `mypack.Abc` để đưa đến mô-đun `Abc` trong gói `mypack`.

```
class AbcUser2 extends Module {
  val io = IO(new Bundle{})

  val abc = new mypack.Abc()

}
```

Cũng có thể nhập vào chỉ một lớp duy nhất và tạo một thực thể của nó:

```
import mypack.Abc

class AbcUser3 extends Module {
  val io = IO(new Bundle{})

  val abc = new Abc()

}
```

3.1.2 Chạy chương trình sbt

Một dự án Chisel có thể được biên dịch và thi hành với một lệnh `sbt` đơn giản:

```
$ sbt run
```

Lệnh này sẽ biên dịch tất cả các mã Chisel của bạn từ cấu trúc cây nguồn và tìm kiếm các lớp chứa `object` bao gồm phương pháp `main`, hoặc đơn giản hơn là mở rộng `App`. Nếu có nhiều hơn một đối tượng, thì tất cả các đối tượng được liệt kê và và một đối tượng có thể được chọn. Các bạn còn có thể xác định trực tiếp đối tượng sẽ được thi hành như là một tham số đối với `sbt`:

```
$ sbt "runMain mypacket.MyObject"
```

Theo mặc định, `sbt` chỉ tìm kiếm phần `main` của cấu trúc cây nguồn, chứ không tìm kiếm phần `test`.[2] Tuy nhiên, các trình kiểm tra Chisel, như đã được mô tả ở đây, chứa `main`, nhưng sẽ được đặt trong phần `test` của source tree. Để thực thi `main` trong cấu trúc cây của trình kiểm tra, sử dụng lệnh `sbt` sau:

```
$ sbt "test:runMain mypacket.MyTester"
```

Bây giờ chúng ta biết được cấu trúc cơ bản của một dự án Chisel và cách để biên dịch và chạy nó với `sbt`, chúng ta có thể tiếp tục với một khung sườn kiểm tra đơn giản.

3.1.3 Quy trình công cụ

Hình 3.2 trình bày quy trình công cụ của Chisel. Mạch số được mô tả trong một lớp Chisel được trình bày trong `Hello.scala`. Trình biên dịch Scala sẽ biên dịch lớp này, cùng với các thư viện Chisel và Scala, và tạo ra lớp Java `Hello.class` có thể được thực thi bởi Máy ảo Java (JVM) chuẩn.

Việc thực thi lớp này với trình điều khiển Chisel tạo ra cái gọi là dạng biểu diễn trung gian linh hoạt cho RTL (FIRRTL), một dạng biểu diễn trung gian của các mạch số. Trong ví dụ, tập tin biểu diễn trung gian là `Hello.fir`. Trình biên dịch FIRRTL thực hiện các phép biến đổi trên mạch điện.

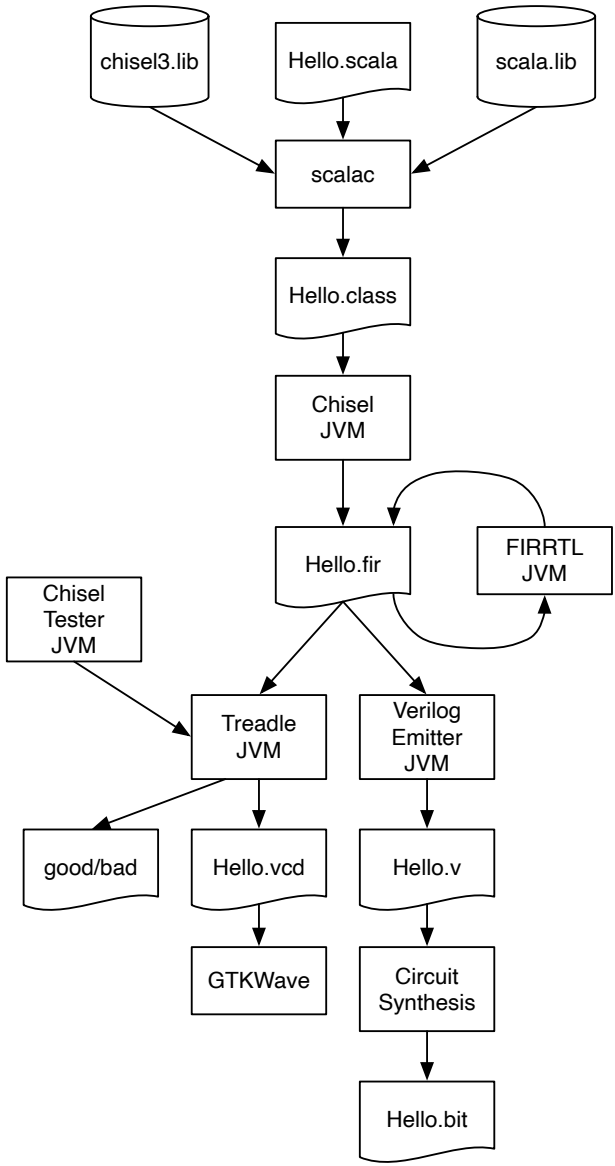
Treadle là một trình thông dịch FIRRTL để mô phỏng mạch điện. Cùng với trình kiểm tra Chisel, nó có thể được sử dụng để gỡ lỗi và kiểm tra các mạch điện của Chisel. Với các giá trị ngõ vào, chúng có thể đưa ra các kết quả kiểm tra. Treadle cũng có thể tạo ra các tập tin dạng sóng (`Hello.vcd`), chúng có thể được xem bằng chương trình xem dạng sóng (Ví dụ: các chương trình xem miễn phí như GTKWave hoặc Modelsim).

Một phép biến đổi FIRRTL, hay bộ phát Verilog, sẽ tạo mã Verilog để tổng hợp mạch (`Hello.v`). Một công cụ tổng hợp mạch (Ví dụ: Intel Quartus, Xilinx Vivado hoặc một công cụ ASIC) sẽ tổng hợp mạch điện. Trong quy trình thiết kế FPGA, công cụ tạo tập tin bitstream FPGA được sử dụng để cấu hình cho FPGA, ví dụ: `Hello.bit`.

3.2 Kiểm tra với Chisel

Các kiểm tra cho các thiết kế phần cứng thường được gọi là testbench. Testbench khởi tạo các thiết kế cần được kiểm tra (Design under Test - DUT), điều khiển các ngõ vào, quan sát các ngõ ra, và so sánh chúng với các giá trị mong đợi.

[2] Đây là dạng quy ước của Java/Scala rằng thư mục test chứa các bài kiểm tra đơn vị chứ không phải các đối tượng với `main`.



Hình 3.2: Quy trình công cụ của hệ sinh thái Chisel.

3.2.1 PeekPokeTester

Chisel cung cấp các testbench dưới dạng `PeekPokeTester`. Một điểm mạnh của Chisel là nó có thể sử dụng toàn bộ sức mạnh của Scala để viết các testbench đó. Ví dụ, một người có thể viết mã với chức năng mong đợi của phần cứng trong một chương trình mô phỏng phần mềm và so sánh kết quả mô phỏng của phần cứng với chương trình mô phỏng phần mềm. Phương pháp này rất hiệu quả khi kiểm tra việc thực hiện một bộ xử lý [6].

Để sử dụng `PeekPokeTester`, các gói sau cần được nhập vào (import):

```
import chisel3._
import chisel3.iotesters._
```

Việc kiểm tra một mạch điện có (ít nhất) ba thành phần: (1) thiết bị được kiểm tra (thường được gọi là DUT), (2) giá trị logic để kiểm tra, còn được gọi là testbench và (3) các đối tượng của trình kiểm tra có chứa `main` để bắt đầu việc kiểm tra.

Đoạn mã sau đây trình bày thiết kế đơn giản của chúng ta đang được thử nghiệm. Nó chứa hai cổng ngõ vào và một cổng ngõ ra, tất cả đều có độ rộng 2-bit. Mạch điện thực hiện phép toán AND bit-wise để trả về giá trị ngõ ra:

```
class DeviceUnderTest extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(2.W))
    val b = Input(UInt(2.W))
    val out = Output(UInt(2.W))
  })

  io.out := io.a & io.b
}
```

Testbench cho DUT này mở rộng `PeekPokeTester` và có DUT như là tham số cho hàm tạo:

```
class TesterSimple(dut: DeviceUnderTest) extends
    PeekPokeTester(dut) {

  poke(dut.io.a, 0.U)
  poke(dut.io.b, 1.U)
  step(1)
  println("Result is: " + peek(dut.io.out).toString)
  poke(dut.io.a, 3.U)
  poke(dut.io.b, 2.U)
```

```
  step(1)
  println("Result is: " + peek(dut.io.out).toString)
}
```

`PeekPokeTester` có thể đặt giá trị ngõ vào với `poke()` và đọc ngược các giá trị ngõ ra với `peek()`. Trình kiểm tra tăng quá trình mô phỏng thêm một bước (= một chu kỳ xung clock) với `step(1)`. Chúng ta có thể in các giá trị ngõ ra bởi `println()`.

Kiểm tra được tạo và chạy với chương trình kiểm tra chính sau:

```
object TesterSimple extends App {
  chisel3.iotesters.Driver(() => new DeviceUnderTest()) { c =>
    new TesterSimple(c)
  }
}
```

Khi các bạn chạy kiểm tra, các bạn sẽ thấy kết quả được in ra ở terminal (bên cạnh các thông tin khác):

```
[info] [0.004] SEED 1544207645120
[info] [0.008] Result is: 0
[info] [0.009] Result is: 2
test DeviceUnderTest Success: 0 tests passed in 7 cycles
taking 0.021820 seconds
[info] [0.010] RAN 2 CYCLES PASSED
```

Chúng ta thấy rằng 0 AND 1 cho kết quả là 0; 3 AND 2 cho kết quả là 2. Bên cạnh việc đánh giá các kết quả xuất ra theo cách thủ công, vốn là một điểm khởi đầu tuyệt vời, chúng ta cũng có thể tbiểu diễn kết quả mong đợi của mình trong chính testbench với `expect()`, có cổng ngõ ra và giá trị mong đợi dưới dạng các tham số. Ví dụ sau trình bày việc kiểm tra với `expect()`:

```
class Tester(dut: DeviceUnderTest) extends PeekPokeTester(dut) {

  poke(dut.io.a, 3.U)
  poke(dut.io.b, 1.U)
  step(1)
  expect(dut.io.out, 1)
  poke(dut.io.a, 2.U)
  poke(dut.io.b, 0.U)
  step(1)
  expect(dut.io.out, 0)
}
```

Thực thi việc kiểm tra này không xuất ra bất kỳ giá trị nào từ phần cứng, nhưng tất cả bài kiểm tra đều được PASSED khi tất cả giá trị mong đợi đều đúng.

```
[info] [0.001] SEED 1544208437832
test DeviceUnderTest Success: 2 tests passed in 7 cycles
taking 0.018000 seconds
[info] [0.009] RAN 2 CYCLES PASSED
```

Kiểm tra không thành công, khi DUT hoặc testbench có lỗi, sẽ tạo ra một thông báo lỗi mô tả sự sai biệt giữa giá trị kỳ vọng và giá trị thực tế. Trong phần sau, chúng ta thay đổi testbench để kỳ vọng kết quả là 4, đó là một lỗi:

```
[info] [0.002] SEED 1544208642263
[info] [0.011] EXPECT AT 2   io_out got 0 expected 4 FAIL
test DeviceUnderTest Success: 1 tests passed in 7 cycles
taking 0.022101 seconds
[info] [0.012] RAN 2 CYCLES FAILED FIRST AT CYCLE 2
```

Trong phần này, chúng ta mô tả phương tiện kiểm tra cơ bản với Chisel cho các kiểm tra đơn giản. Tuy nhiên, trong Chisel, toàn bộ sức mạnh của Scala có sẵn để viết các trình kiểm tra.

3.2.2 Sử dụng ScalaTest

ScalaTest là một công cụ kiểm tra dành cho Scala (và Java), mà chúng ta có thể sử dụng để chạy các chương trình kiểm tra Chisel. Để sử dụng nó, bao gồm thư viện trong `build.sbt` của các bạn với dòng lệnh sau:

```
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.5"
    % "test"
```

Các bài kiểm tra thường được tìm thấy trong `src/test/scala` và có thể được chạy bởi lệnh:

```
$ sbt test
```

Một kiểm tra tối thiểu (kiểm tra Hello World) để kiểm tra cộng số nguyên trong Scala:

```
import org.scalatest._

class ExampleSpec extends FlatSpec with Matchers {
```

```
  "Integers" should "add" in {
    val i = 2
    val j = 3
    i + j should be (5)
  }
}
```

Mặc dù kiểm tra trên Chisel nặng nề hơn kiểm tra đơn vị của các chương trình Scala, chúng ta có thể kết hợp kiểm tra Chisel thành một lớp ScalaTest. Đối với `Tester` được hiển thị trước đây là:

```
class SimpleSpec extends FlatSpec with Matchers {

  "Tester" should "pass" in {
    chisel3.iotesters.Driver(() => new DeviceUnderTest()) { c =>
      new Tester(c)
    } should be (true)
  }
}
```

Lợi ích chính của bài tập này là có thể chạy tất cả các trình kiểm tra với một lệnh `sbt test` đơn giản (thay vì chạy `main`). Các bạn có thể chạy chỉ một trình kiểm tra duy nhất với `sbt` như sau:

```
$ sbt "testOnly SimpleSpec"
```

3.2.3 Dạng sóng

Các chương trình kiểm tra, như đã đề cập ở trên, làm việc tốt với các thiết kế nhỏ và kiểm tra đơn vị, vì nó phổ biến trong phát triển phần mềm. Một tập hợp các bài kiểm tra đơn vị cũng có thể dùng cho kiểm tra hồi quy.

Tuy nhiên, để gỡ lỗi các thiết kế phức tạp hơn, người ta muốn xét nhiều tín hiệu cùng lúc. Một cách tiếp cận cổ điển để gỡ lỗi các thiết kế mạch số là hiển thị các tín hiệu dưới dạng sóng. Ở dạng sóng, các tín hiệu được hiển thị theo thời gian.

Các chương trình kiểm tra có thể tạo ra một dạng sóng bao gồm tất cả các thanh ghi và tất cả các tín hiệu vào ra IO. Trong các ví dụ sau, chúng ta hiển thị chương trình kiểm tra dạng sóng cho `DeviceUnderTest` từ ví dụ trước (hàm AND 2-bit). Với ví dụ sau, chúng ta nhập các lớp sau:

```
import chisel3.iotesters.PeekPokeTester
```

```
import chisel3.iotesters.Driver
import org.scalatest._
```

Chúng ta bắt đầu với một trình kiểm tra đơn giản, lấy giá trị của các ngõ vào và tăng xung clock bằng lệnh `step`. Chúng ta không đọc bất kỳ ngõ ra nào hoặc so sánh nó với `kết quả mong đợi`.

```
class WaveformTester(dut: DeviceUnderTest) extends
    PeekPokeTester(dut) {

  poke(dut.io.a, 0)
  poke(dut.io.b, 0)
  step(1)
  poke(dut.io.a, 1)
  poke(dut.io.b, 0)
  step(1)
  poke(dut.io.a, 0)
  poke(dut.io.b, 1)
  step(1)
  poke(dut.io.a, 1)
  poke(dut.io.b, 1)
  step(1)
}
```

Thay vì chúng ta gọi `Driver.execute` với các tham số để tạo các tập tin dạng sóng (tập tin .vcd).

```
class WaveformSpec extends FlatSpec with Matchers {
  "Waveform" should "pass" in {
    Driver.execute(Array("--generate-vcd-output", "on"), () =>
        new DeviceUnderTest()) { c =>
      new WaveformTester(c)
    } should be (true)
  }
}
```

Các bạn có thể xem dạng sóng với các công cụ miễn phí như GTKWave hoặc ModelSim. Chạy GTKWave và chọn *File – Open New Window* và chuyển đến thư mục mà trình kiểm tra Chisel đặt tập tin `.vcd`. Theo mặc định, các tập tin được tạo ra ở trong thư mục `test_run_dir` sau đó tên của chương trình kiểm tra được đánh thêm số ở phía sau. Ở trong thư mục này, các bạn có thể thấy tập tin `DeviceUnderTest.vcd`. Các bạn có thể lựa chọn các tín hiệu từ bên trái và kéo chúng vào cửa sổ chính. Nếu các bạn muốn lưu

cấu hình của các tín hiệu, các bạn thực hiện bằng cách vào *File – Write Save File* và sau đó có thể mở chúng lại bằng cách vào *File – Read Save File*.

Liệt kê rõ ràng tất cả các giá trị ngõ vào không chia tỷ lệ. Do đó, chúng ta sử dụng một số mã Scala để điều khiển DUT. Trình kiểm tra sau liệt kê tất cả các giá trị có thể có cho 2 tín hiệu ngõ vào 2-bit.

```
class WaveformCounterTester(dut: DeviceUnderTest) extends
    PeekPokeTester(dut) {

  for (a <- 0 until 4) {
    for (b <- 0 until 4) {
      poke(dut.io.a, a)
      poke(dut.io.b, b)
      step(1)
    }
  }
}
```

Chúng ta thêm đặc tả ScalaTest cho trình kiểm tra mới này

```
class WaveformCounterSpec extends FlatSpec with Matchers {

  "WaveformCounter" should "pass" in {
    Driver.execute(Array("--generate-vcd-output", "on"), () =>
        new DeviceUnderTest()) { c =>
      new WaveformCounterTester(c)
    } should be (true)
  }
}
```

và thực thi với lệnh

```
sbt "testOnly WaveformCounterSpec"
```

3.2.4 Gỡ lỗi với printf

Một hình thức gỡ lỗi khác là cái gọi là "gỡ lỗi printf". Hình thức này đơn giản chỉ là đặt các câu lệnh `printf` trong C để in ra giá trị các biến chúng ta quan tâm trong quá trình thực thi chương trình. Gỡ lỗi printf này cũng có sẵn trong quá trình kiểm tra các mạch điện Chisel. Việc in các giá trị xảy ra ở cạnh lên của xung clock. Câu lệnh `printf` có thể được chèn vào bất kỳ đâu trong định nghĩa mô-đun, như được trình bày trong phiên bản gỡ lỗi printf của DUT.

```
class DeviceUnderTestPrintf extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(2.W))
    val b = Input(UInt(2.W))
    val out = Output(UInt(2.W))
  })

  io.out := io.a & io.b
  printf("dut: %d %d %d\n", io.a, io.b, io.out)
}
```

Khi kiểm tra mô-đun này với trình kiểm tra dựa trên mạch đếm, lặp lại trên tất cả các giá trị có thể, chúng ta nhận được kết quả ngõ ra sau, xác minh rằng hàm AND là đúng:

```
Circuit state created
[info] [0.001] SEED 1579707298694
dut: 0 0 0
dut: 0 1 0
dut: 0 2 0
dut: 0 3 0
dut: 1 0 0
dut: 1 1 1
dut: 1 2 0
dut: 1 3 1
dut: 2 0 0
dut: 2 1 0
dut: 2 2 2
dut: 2 3 2
dut: 3 0 0
dut: 3 1 1
dut: 3 2 2
dut: 3 3 3
test DeviceUnderTestPrintf Success: 0 tests passed in 21 cycles
  taking 0.036380 seconds
[info] [0.024] RAN 16 CYCLES PASSED
```

Lệnh printf Chisel hỗ trợ định dạng như trong C và Scala.

3.3 Bài tập

Với bài tập phần này, chúng ta sẽ xem lại bài đèn LED nhấp nháy từ chisel-examples và phám khá kiểm tra Chisel.

3.3.1 Dự án tối thiểu

Đầu tiên, chúng ta hãy cùng tìm hiểu dự án Chisel tối thiểu là gì. Khám phá các tập tin trong ví dụ Hello World. `Hello.scala` là tập tin nguồn phần cứng duy nhất. Nó chứa mô tả phần cứng của đèn LED nhấp nháy (`class Hello`) và `App` tạo mã Verilog.

Mỗi tập tin bắt đầu với việc nhập Chisel và các gói liên quan:

```
import chisel3._
```

Sau đó theo mô tả phần cứng, như đã trình bày trong Listing 1.1. Để tạo mô tả Verilog, chúng ta cần một ứng dụng. Một đối tượng Scala `extends App` là một ứng dụng ngầm tạo hàm chính nơi ứng dụng đó khởi động. Hành động duy nhất của ứng dụng này là tạo ra một đối tượng `HelloWorld` mới và chuyển nó vào trình điều khiển Chisel `execute` hàm. Đối số đầu tiên là một mảng Strings, nơi các tùy chọn cho việc build có thể được đặt (ví dụ, thư mục ngõ ra). Đoạn mã sau sẽ tạo ra tập tin Verilog `Hello.v`.

```
object Hello extends App {
  (new chisel3.stage.ChiselStage).emitVerilog(new Hello())
}
```

Chạy quá trình tạo ví dụ theo cách thủ công với lệnh

```
$ sbt "runMain Hello"
```

và khám phá tập tin `Hello.v` được tạo ra với một trình soạn thảo. Mã Verilog được tạo ra có thể không đọc được nhưng chúng ta có thể tìm hiểu một số chi tiết. Tập tin bắt đầu với mô-đun `Hello`, có cùng tên với mô-đun Chisel của chúng ta. Chúng ta có thể xác định cổng LED là ngõ ra `output io_led`. Tên các chân là tên Chisel với `io_` ở trước. Bên cạnh chân LED, mô-đun còn chứa các tín hiệu ngõ vào là `clock` và `reset`. Hai tín hiệu này được thêm vào tự động bởi Chisel.

Hơn nữa, chúng ta có thể xác định định nghĩa của hai thanh ghi `cntReg` và `blkReg`. Chúng ta còn có thể thấy reset và cập nhật các thanh ghi này ở cuối định nghĩa của mô-đun. Lưu ý rằng Chisel tạo reset đồng bộ.

Để `sbt` có thể gọi trình biên dịch Scala và thư viện Chisel đúng, chúng ta cần lệnh `build.sbt` sau:

```
scalaVersion := "2.12.12"

scalacOptions := Seq("-deprecation", "-Xsource:2.11")

resolvers ++= Seq(
  Resolver.sonatypeRepo("snapshots"),
  Resolver.sonatypeRepo("releases")
)

libraryDependencies += "edu.berkeley.cs" %% "chisel-iotesters"
    % "1.5.1"
libraryDependencies += "edu.berkeley.cs" %% "chiseltest" %
    "0.3.1"
// Chisel 3.4.1 is loaded as a dependency on the testers
```

Lưu ý rằng trong ví dụ này, chúng ta có số phiên bản Chisel cụ thể để tránh kiểm tra phiên bản mới ở mỗi lần chạy (sẽ không thực hiện được nếu không có kết nối với Internet, ví dụ: khi thiết kế phần cứng trên máy bay). Thay đổi cấu hình `build.sbt` để sử dụng phiên bản Chisel mới nhất bằng cách thay đổi lệnh trong phần thư viện như sau

```
libraryDependencies += "edu.berkeley.cs" %% "chisel3" %
    "latest.release"
```

và chạy lại build với lệnh `sbt`. Xem có phiên bản Chisel mới hơn không và nó có được tải xuống tự động không?

Để thuận tiện, thư mục dự án cũng chứa một tập tin `Makefile`. Trong tập tin này chỉ có lệnh `sbt`, vì vậy chúng ta không cần nhớ nó và có thể tạo mã Verilog với lệnh:

```
make
```

Bên cạnh tập tin `README`, thư mục dự án cũng chứa các tập tin dự án cho bo mạch FPGA khác nhau. Ví dụ, trong quartus/altde2-115 các bạn có thể thấy hai tập tin dự án để định nghĩa một dự án Quartus cho bo mạch DE2-115. Các định nghĩa chính (tập tin nguồn, thiết bị, gán chân) có thể được tìm thấy trong tập tin văn bản hello.qsf. Khám phá tập tin và xem chân nào được nối với tín hiệu nào. Nếu các bạn cần chuyển dự án sang một bo mạch khác, thì ở đó các thay đổi được cập nhật. Nếu bạn có chương trình Quartus đã cài đặt, mở dự án đó, biên dịch với nút *Play* xanh lá, và sau đó cấu hình cho FPGA.

Lưu ý rằng *Hello World* là dự án Chisel tối thiểu. Các dự án thực tế hơn có các tập tin nguồn được tổ chức thành các gói và chứa các trình kiểm tra. Bài tập tiếp theo sẽ khám phá một dự án như vậy.

3.3.2 Bài tập kiểm tra

Trong bài tập của chương trước, các bạn đã mở rộng ví dụ về đèn LED nhấp nháy với một số ngõ vào để build cổng AND và mạch đa hợp, và chạy phần cứng này trên FPGA. Bây giờ chúng ta sẽ sử dụng ví dụ này và kiểm tra chức năng bằng trình kiểm tra Chisel để tự động kiểm tra và cũng độc lập với FPGA. Sử dụng thiết kế của các bạn từ chương trước và thêm trình kiểm tra Chisel để kiểm tra chức năng. Cố gắng liệt kê tất cả các giá trị ngõ vào có thể và kiểm tra ngõ ra bằng `exception()`.

Kiểm tra trong Chisel có thể tăng tốc độ gỡ lỗi thiết kế của các bạn. Tuy nhiên, các bạn nên tổng hợp thiết kế của mình cho FPGA và chạy kiểm tra với FPGA. Ở đó, các bạn có thể thực hiện kiểm tra thực tế về kích thước thiết kế của mình (thường là số lượng LUT và Flip-flop) và hiệu suất thiết kế của bạn ở tần số xung clock tối đa. Như một điểm để tham khảo, một bộ xử lý RISC đường ống theo kiểu sách giáo khoa có thể chiếm khoảng 3000 LUT 4-bit và có thể hoạt động ở khoảng tần số 100 MHz trên FPGA có giá thành thấp (Intel Cyclone hoặc Xilinx Spartan).

4 Các thành phần

Một thiết kế mạch số lớn hơn được cấu trúc thành một tập hợp các thành phần, thường theo cách phân cấp. Mỗi thành phần có một giao tiếp với các dây ngõ vào và ngõ ra, thường được gọi là cổng. Chúng tương tự như các chân ngõ vào và ngõ ra trên mạch tích hợp (IC). Các thành phần được kết nối với nhau bằng cách nối dây các ngõ vào và ngõ ra. Các thành phần có thể chứa các thành phần con để xây dựng hệ thống phân cấp. Thành phần ngoài cùng, được kết nối với các chân vật lý trên chip, được gọi là thành phần cấp cao nhất (top-level).

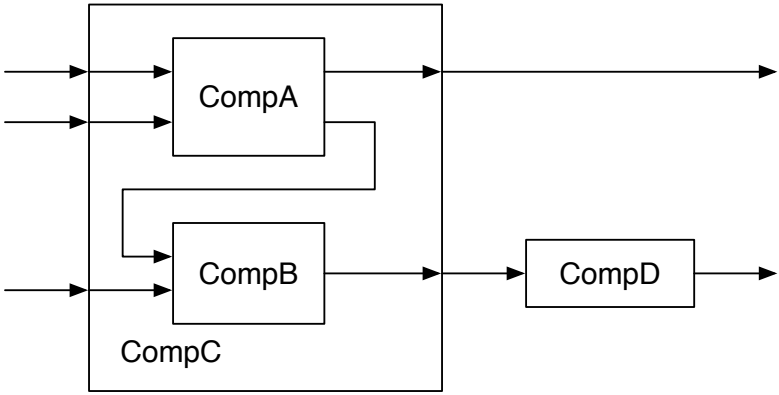
Hình 4.1 biểu diễn một thiết kế ví dụ. Thành phần C có ba cổng ngõ vào và hai cổng ngõ ra. Bản thân thành phần này được ghép lại từ hai thành phần con: B và C, được kết nối với các ngõ vào và ngõ ra của C. Một ngõ ra của A được kết nối với ngõ vào của B. Thành phần D ở cùng cấp phân cấp với thành phần C và được kết nối với nó.

Trong chương này, chúng ta sẽ giải thích cách các thành phần được mô tả trong Chisel và cung cấp các ví dụ về các thành phần chuẩn. Các thành phần tiêu chuẩn đó phục vụ hai mục đích: (1) chúng cung cấp các ví dụ về mã Chisel và (2) chúng cung cấp một thư viện gồm các thành phần sẵn sàng được sử dụng lại trong thiết kế của các bạn.

4.1 Các thành phần trong Chisel là mô-đun

Các thành phần phần cứng được gọi là mô-đun trong Chisel. Mỗi mô-đun mở rộng lớp `Module` và chứa trường `io` cho việc giao tiếp. Giao tiếp được định nghĩa bởi `Bundle` được gói thành một lệnh gọi đến `IO()`. `Bundle` chứa các trường để biểu diễn các cổng ngõ vào và ngõ ra của mô-đun. Chỉ hướng vào ra được đưa ra bằng cách đưa một trường vào lệnh gọi `Input()` hoặc `Output()`. Chỉ hướng là từ góc nhìn của chính thành phần đó.

Listing 4.1 biểu diễn định nghĩa của hai thành phần ví dụ A và B từ Hình 4.1. Thành phần A có hai ngõ vào, được đặt tên là `a` và `b`, và hai ngõ ra, được đặt tên là `x` và `y`. Với những cổng của thành phần B chúng ta chọn các tên là `in1`, `in2`, và `out`. Tất cả các cổng sử dụng số nguyên không dấu (`UInt`) với độ rộng bit là 8. Vì đoạn mã ví dụ này đề cập về kết nối các thành phần và xây dựng cấu trúc phân cấp, nên chúng ta không triển khai bất kỳ thực hiện nào trong các thành phần. Việc triển khai thành phần được viết ở nơi các chú thích khai báo “Hàm của X”. Vì chúng ta không có hàm nào được liên kết với



Hình 4.1: Một thiết kế gồm các thành phần phân cấp.

```
class CompA extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val x = Output(UInt(8.W))
    val y = Output(UInt(8.W))
  })

  // function of A
}

class CompB extends Module {
  val io = IO(new Bundle {
    val in1 = Input(UInt(8.W))
    val in2 = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })

  // function of B
}
```

Listing 4.1: Định nghĩa của thành phần A và B

```
class CompC extends Module {
  val io = IO(new Bundle {
    val in_a = Input(UInt(8.W))
    val in_b = Input(UInt(8.W))
    val in_c = Input(UInt(8.W))
    val out_x = Output(UInt(8.W))
    val out_y = Output(UInt(8.W))
  })

  // create components A and B
  val compA = Module(new CompA())
  val compB = Module(new CompB())

  // connect A
  compA.io.a := io.in_a
  compA.io.b := io.in_b
  io.out_x := compA.io.x
  // connect B
  compB.io.in1 := compA.io.y
  compB.io.in2 := io.in_c
  io.out_y := compB.io.out
}
```

Listing 4.2: Thành phần C

các thành phần của ví dụ này, nên chúng ta sử dụng các tên cổng chung. Với một thiết kế thực sự, chúng ta sử dụng các tên cổng mô tả, chẳng hạn như `data`, `valid`, hoặc `ready`.

Thành phần C, được biểu diễn trong Listing 4.2, có ba cổng ngõ vào và hai ngõ ra. Nó được xây dựng từ các thành phần A và B. Chúng ta chỉ ra cách A và B được kết nối với các cổng của C và cũng là kết nối giữa cổng ngõ ra của A và cổng ngõ vào của B.

Thành phần C được tạo bởi `new`, ví dụ: `new CompA()`, và cần được gói trong một lệnh gọi tới `Module()`. Tham chiếu đến mô-đun đó được lưu trong biến cục bộ, trong ví dụ này là `val compA = Module(new CompA())`.

Với tham chiếu này, chúng ta có thể truy cập đến các cổng IO bằng cách xóa tham chiếu trường `io` của mô-đun và các trường riêng lẻ của `Bundle` IO.

Thành phần đơn giản nhất trong thiết kế, như đã biểu diễn trong Listing 4.3, chỉ có một cổng ngõ vào, được đặt tên là `in`, và một cổng ngõ ra được đặt tên là `out`.

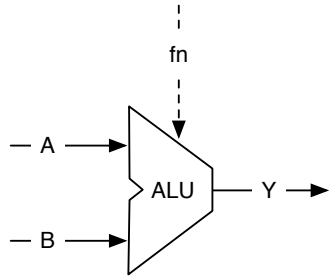
Phần còn thiếu cuối cùng của thiết kế ví dụ là thành phần mức cao nhất (top-level), chính nó được ghép nối từ các thành phần C và D, như được biểu diễn trong Listing 4.4.

```
class CompD extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })

  // function of D
}
```

Listing 4.3: Thành phần D

```
class TopLevel extends Module {
  val io = IO(new Bundle {
    val in_a = Input(UInt(8.W))
    val in_b = Input(UInt(8.W))
    val in_c = Input(UInt(8.W))
    val out_m = Output(UInt(8.W))
    val out_n = Output(UInt(8.W))
  })

  // create C and D
  val c = Module(new CompC())
  val d = Module(new CompD())

  // connect C
  c.io.in_a := io.in_a
  c.io.in_b := io.in_b
  c.io.in_c := io.in_c
  io.out_m := c.io.out_x
  // connect D
  d.io.in := c.io.out_y
  io.out_n := d.io.out
}
```

Listing 4.4: Thành phần cao nhất



Hình 4.2: Đơn vị logic số học, hoặc ghi tắt là ALU.

Thiết kế các thành phần tốt tương tự như việc thiết kế tốt các hàm hoặc phương pháp trong thiết kế phần mềm. Một trong những câu hỏi chính là chúng ta sẽ đưa bao nhiêu chức năng vào một thành phần và một thành phần phải lớn như thế nào. Có hai thái cực: các thành phần nhỏ chẳng hạn như mạch cộng và các thành phần khổng lồ chẳng hạn như một bộ vi xử lý đầy đủ.

Những người mới bắt đầu thiết kế phần cứng thường bắt đầu với các thành phần nhỏ. Vấn đề là sách thiết kế mạch số sử dụng các thành phần nhỏ để biểu diễn các nguyên lý. Nhưng kích thước của các ví dụ (trong những cuốn sách đó và cả trong cuốn sách này) nhỏ để vừa với một trang và không làm người đọc xao nhãng bởi quá nhiều chi tiết.

Với các thành phần nhỏ, như mạch đếm, Chisel cung cấp một cách nhẹ nhàng hơn để mô tả chúng như các hàm trả về phần cứng.

Giao tiếp với một thành phần hơi dài dòng một chút (với các kiểu, tên, hướng, cấu trúc IO). Theo nguyên tắc chung, tôi đề xuất rằng lõi của thành phần, của hàm, ít nhất phải dài bằng giao tiếp của thành phần.

4.2 Đơn vị Logic số học

Một trong những thành phần trung tâm của mạch điện tính toán, ví dụ như bộ vi xử lý, là đơn vị logic số học, hay ghi tắt là ALU. Hình 4.2 biểu diễn biểu tượng của một ALU.

ALU có hai ngõ vào, đặt nhãn là `A` và `B` như trong hình, một ngõ vào chức năng `fn`, và một ngõ ra, đặt nhãn là Y. ALU hoạt động dựa trên giá trị `A` và `B` và đưa kết quả ở ngõ ra. Ngõ vào `fn` chọn phép toán cho `A` và `B`. Các phép toán thường là dạng số học, chẳng hạn như phép cộng và phép trừ, và một số phép toán logic như and, or, xor. Đó là lý do tại sao nó được gọi là ALU.

Ngõ vào chức năng `fn` chọn phép toán. ALU thường là mạch tổ hợp không có bất kỳ phần tử trạng thái nào. Một ALU còn có thể có ngõ ra bổ sung để báo hiệu các đặc tính

của kết quả như cờ zero, cờ dấu.

Đoạn mã sau đây biểu diễn một ALU có ngõ vào và ngõ ra 16-bit hỗ trợ các phép toán: cộng, trừ, OR, AND và phép toán khác được chọn bởi tín hiệu `fn` 2-bit.

```
class Alu extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(16.W))
    val b = Input(UInt(16.W))
    val fn = Input(UInt(2.W))
    val y = Output(UInt(16.W))
  })

  // some default value is needed
  io.y := 0.U

  // The ALU selection
  switch(io.fn) {
    is(0.U) { io.y := io.a + io.b }
    is(1.U) { io.y := io.a - io.b }
    is(2.U) { io.y := io.a | io.b }
    is(3.U) { io.y := io.a & io.b }
  }
}
```

Trong ví dụ này, chúng ta sử dụng một cấu trúc Chisel mới, cấu trúc `switch/is`, để mô tả bảng chọn ngõ ra của ALU. Để sử dụng chức năng tiện ích này, chúng ta cần nhập (import) một gói Chisel khác:

```
import chisel3.util._
```

4.3 Các kết nối khối

Để kết nối các thành phần có nhiều cổng IO, Chisel cung cấp toán tử kết nối khối (bulk connection) <>. Toán tử này kết nối các phần của bundle theo cả hai hướng. Chisel sử dụng tên của các trường lá (leaf field) để kết nối. Nếu thiếu tên, nó không được kết nối.

Ví dụ, hãy giả sử chúng ta xây dựng một bộ xử lý đường ống (pipeline). Giai đoạn nạp lệnh có giao tiếp sau:

```
class Fetch extends Module {
  val io = IO(new Bundle {
```

```
    val instr = Output(UInt(32.W))
    val pc = Output(UInt(32.W))
  })
  // ... Implementation of fetch
}
```

Giai đoạn tiếp theo là giai đoạn giải mã lệnh.

```
class Decode extends Module {
  val io = IO(new Bundle {
    val instr = Input(UInt(32.W))
    val pc = Input(UInt(32.W))
    val aluOp = Output(UInt(5.W))
    val regA = Output(UInt(32.W))
    val regB = Output(UInt(32.W))
  })
  // ... Implementation of decode
}
```

Giai đoạn cuối cùng của bộ xử lý đơn giản của chúng ta là giai đoạn thực thi.

```
class Execute extends Module {
  val io = IO(new Bundle {
    val aluOp = Input(UInt(5.W))
    val regA = Input(UInt(32.W))
    val regB = Input(UInt(32.W))
    val result = Output(UInt(32.W))
  })
  // ... Implementation of execute
}
```

Để kết nối tất cả ba giai đoạn, chúng ta cần hai toán tử <>. Chúng ta cũng có thể kết nối cổng của mô-đun con với mô-đun mẹ.

```
  val fetch = Module(new Fetch())
  val decode = Module(new Decode())
  val execute = Module(new Execute)

  fetch.io <> decode.io
  decode.io <> execute.io
  io <> execute.io
```

4.4 Các thành phần nhẹ dùng các hàm

Mô-đun là cách tổng quát để cấu trúc việc mô tả phần cứng của các bạn. Tuy nhiên, có một số mã code viết sẵn khi khai báo một mô-đun và khi khởi tạo và kết nối nó. Một cách gọn nhẹ để cấu trúc phần cứng của các bạn là sử dụng các hàm. Các hàm Scala có thể nhận các tham số Chisel (và Scala) và trả về phần cứng đã được tạo. Với một ví dụ đơn giản, chúng ta tạo một mạch cộng:

```
def adder (x: UInt, y: UInt) = {
  x + y
}
```

Chúng ta có thể tạo hai mạch cộng bằng cách đơn giản gọi hàm `adder`.

```
val x = adder(a, b)
// another adder
val y = adder(c, d)
```

Lưu ý rằng đây là *bộ tạo phần cứng*. Các bạn không thực hiện bất kỳ thao tác thêm nào trong khi build, nhưng hãy tạo hai mạch cộng (thực thể phần cứng). Mạch cộng là một ví dụ nhân tạo đơn giản. Chisel đã có sẵn hàm tạo mạch cộng, chẳng hạn như `+(that: UInt)`.

Các hàm, như là bộ tạo phần cứng nhẹ, cũng có thể chứa trạng thái (bao gồm thanh ghi). Ví dụ sau trả về một phần tử trễ một chu kỳ đồng hồ (một thanh ghi). Nếu một hàm chỉ có một câu lệnh duy nhất, thì chúng ta có thể viết nó trong một dòng và bỏ qua dấu ngoặc ().

```
def delay(x: UInt) = RegNext(x)
```

Bằng cách gọi hàm với chính hàm làm tham số, điều này tạo ra độ trễ hai chu kỳ xung clock.

```
val del0ut = delay(delay(delIn))
```

Một lần nữa, đây là một ví dụ rất ngắn nhưng hữu ích, vì `RegNext()` đã là hàm tạo thanh ghi cho độ trễ.

Các hàm có thể được khai báo như một phần của `Module`. Tuy nhiên, các hàm sẽ được sử dụng trong các mô-đun khác nhau thì tốt hơn là được đặt trong một đối tượng Scala để lấy các hàm tiện ích.

5 Các khối xây dựng mạch tổ hợp

Trong chương này, chúng ta khám phá các mạch tổ hợp khác nhau, các khối xây dựng cơ bản mà chúng ta có thể sử dụng để xây dựng các hệ thống phức tạp hơn. Về nguyên tắc, tất cả các mạch tổ hợp có thể được mô tả bằng phương trình đại số Boole. Tuy nhiên, thông thường, mô tả dưới dạng bảng sẽ hiệu quả hơn. Chúng ta sẽ để công cụ tổng hợp trích xuất và tối thiểu hóa các phương trình đại số Boole. Hai mạch cơ bản, được mô tả tốt nhất dưới dạng bảng, là mạch giải mã và mạch mã hóa.

5.1 Các mạch tổ hợp

Trước khi mô tả một số khối xây dựng mạch tổ hợp tiêu chuẩn, chúng ta sẽ khám phá cách các mạch tổ hợp có thể được biểu diễn trong Chisel. Dạng đơn giản nhất là biểu thức đại số Boole, có thể được gán với một tên:

```
val e = (a & b) | c
```

Biểu thức đại số Boole được đặt tên (`e`) bằng cách gán nó với một giá trị Scala. Biểu thức có thể được tái sử dụng lại trong những biểu thức khác:

```
val f = ~e
```

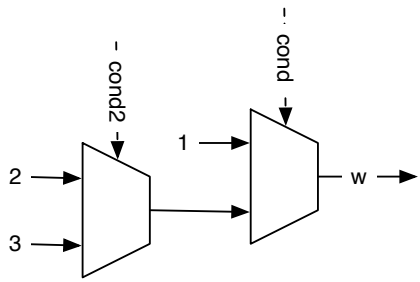
Một biểu thức nhu vậy được xem như là cố định. Một phép gán lại cho `e` bởi `=` sẽ dẫn đến lỗi trình biên dịch Scala: `reassignment to val`. Chúng ta thử với toán tử Chisel `:=`, được biểu diễn như sau,

```
e := c & b
```

sẽ dẫn đến một ngoại lệ runtime: `Cannot reassign to read-only`.

Chisel cũng hỗ trợ mô tả các mạch tổ hợp với các cập nhật có điều kiện.

Một mạch điện như vậy được khai báo là `Wire`. Sau đó, các bạn sử dụng các phép toán điều kiện, như `when`, để mô tả logic của mạch điện. Đoạn mã sau khai báo `Wire w` kiểu `UInt` và gán một giá trị mặc định `0`. Khối `when` kiểu `Bool` trong Chisel và gán lại giá trị `3` cho `w` nếu giá trị của `cond` là `true.B`.



Hình 5.1: Chuỗi các mạch đa hợp.

```
val w = Wire(UInt())

w := 0.U
when (cond) {
  w := 3.U
}
```

Logic mạch điện là mạch đa hợp, với hai ngõ vào là hằng số `0` và `3` và tín hiệu chọn điều kiện `cond`. Nên nhớ rằng chúng ta mô tả các mạch phần cứng chứ không phải chương trình phần mềm được thực thi có điều kiện.

Cấu trúc điều kiện `when` trong Chisel còn có dạng *else*, nó được gọi là `otherwise`. Với việc gán giá trị trong bất kỳ điều kiện nào, chúng ta có thể bỏ qua việc gán giá trị mặc định:

```
val w = Wire(UInt())

when (cond) {
  w := 1.U
} .otherwise {
  w := 2.U
}
```

Chisel còn hỗ trợ chuỗi các điều kiện (chuỗi if/elseif/else) với `.elsewhen`:

```
val w = Wire(UInt())

when (cond) {
  w := 1.U
} .elsewhen (cond2) {
```

```
    w := 2.U
  } .otherwise {
    w := 3.U
  }
```

Chuỗi `when`, `.elsewhen`, và `.otherwise` tạo thành chuỗi các mạch đa hợp. Hình 5.1 biểu diễn chuỗi các mạch đa hợp này. Chuỗi đó đưa ra mức độ ưu tiên, ví dụ: khi `cond` là đúng, các điều kiện khác sẽ không được đánh giá.

Lưu ý '.' trong `.elsewhen` cần thiết với các phương pháp chuỗi trong Scala. Những nhánh `.elsewhen` đó có thể dài ngẫu nhiên. Tuy nhiên, nếu chuỗi điều kiện phụ thuộc vào một tín hiệu duy nhất, thì tốt hơn là sử dụng lệnh `switch`, được giới thiệu trong phần sau với mạch giải mã.

Đối với các mạch tổ hợp phức tạp hơn, thực tế là gán giá trị mặc định cho `Wire`. Phép gán mặc định có thể được kết hợp với khai báo đi dây với `WireDefault`.

```
val w = WireDefault(0.U)

when (cond) {
  w := 3.U
}
// ... and some more complex conditional assignments
```

Người ta có thể đặt câu hỏi là tại sao lại sử dụng `when`, `.elsewhen` và `else` khi Scala có `if`, `else if` và `else`? Các câu lệnh đó là để thực thi có điều kiện mã Scala, không tạo ra phần cứng Chisel (mạch đa hợp). Các điều kiện Scala đó có công dụng trong Chisel khi chúng ta viết các trình tạo mạch, lấy các tham số để tạo các thực thể phần cứng *khác* một cách có điều kiện.

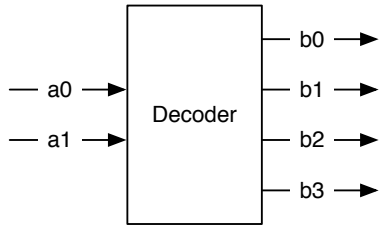
5.2 Mạch giải mã

Mạch giải mã đổi số nhị phân n bits thành tín hiệu m-bit, với $m \leq 2^n$. Ngõ ra được mã hóa one-hot (trong đó chính xác một bit bằng 1).

Hình 5.2 biểu diễn mạch giải mã 2-bit ra 4-bit. Chúng ta có thể mô tả chức năng của mạch giải mã bằng bảng trạng thái, như Bảng 5.2.

Câu lệnh Chisel `switch` mô tả logic dưới dạng bảng trạng thái. Câu lệnh `switch` không phải là một phần của ngôn ngữ Chisel cốt lõi. Vì vậy, chúng ta cần đưa vào các thành phần của gói `chisel.util`.

```
import chisel3.util._
```



Hình 5.2: Mạch giải mã 2-bit ra 4-bit.

a	b
00	0001
01	0010
10	0100
11	1000

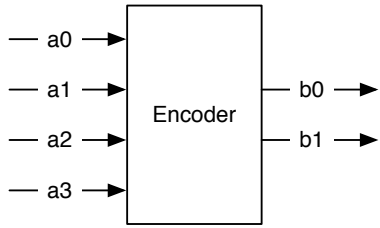
Bảng 5.1: Bảng trạng thái cho mạch giải mã 2 ra 4.

Đoạn mã sau giới thiệu lệnh `switch` của Chisel để mô tả mạch giải mã:

```
result := 0.U

switch(sel) {
  is (0.U) { result := 1.U}
  is (1.U) { result := 2.U}
  is (2.U) { result := 4.U}
  is (3.U) { result := 8.U}
}
```

Câu lệnh `switch` ở trên liệt kê tất cả giá trị có thể có của tín hiệu `sel` và gán giá trị giải mã cho tín hiệu `result`. Lưu ý rằng ngay cả khi chúng ta liệt kê tất cả các giá trị ngõ vào có thể có, Chisel vẫn cần chúng ta gán cho một giá trị mặc định, giống như chúng ta thực hiện bằng cách gán 0 cho `result`. Việc gán này này sẽ không bao giờ kích hoạt và do đó được tối ưu hóa bởi công cụ đầu cuối (backend). Nó nhằm tránh các tình huống có phép gán chưa hoàn tất cho các mạch tổ hợp (trong Chisel là `Wire`) sẽ dẫn đến các mạch chốt không mong muốn trong các ngôn ngữ mô tả phần cứng như VHDL và Verilog. Chisel không cho phép các phép gán chưa hoàn tất.

Trong ví dụ trước, chúng ta đã sử dụng số nguyên không dấu cho các tín hiệu. Một



Hình 5.3: Mạch mã hóa 4-bit thành 2-bit.

biểu diễn rõ ràng hơn của mạch mã hóa sử dụng ký hiệu nhị phân:

```
switch (sel) {
  is ("b00".U) { result := "b0001".U}
  is ("b01".U) { result := "b0010".U}
  is ("b10".U) { result := "b0100".U}
  is ("b11".U) { result := "b1000".U}
}
```

Một bảng cung cấp một biểu diễn rất dễ đọc của chức năng giải mã nhưng cũng hơi dài dòng. Khi xem bảng, chúng ta thấy một cấu trúc thông thường: giá trị 1 được dịch sang trái bởi số được biểu diễn bởi `sel`. Do đó, chúng ta có thể biểu diễn một mạch giải mã bằng phép toán dịch trong Chisel «.

```
result := 1.U << sel
```

Mạch giải mã được sử dụng như một khối xây dựng cho mạch đa hợp bằng cách sử dụng ngõ ra là tín hiệu điều khiển với cổng AND cho ngõ vào dữ liệu của mạch đa hợp. Tuy nhiên, trong Chisel, chúng ta không cần phải xây dựng một mạch đa hợp, vì `Mux` có sẵn trong thư viện lõi. Mạch giải mã cũng có thể được sử dụng để giải mã địa chỉ và sau đó các ngõ ra được sử dụng như các tín hiệu chọn, ví dụ: các linh kiện IO khác nhau được kết nối với bộ vi xử lý.

5.3 Mạch giải mã

Mạch mã hóa đổi tín hiệu ngõ vào one-hot thành tín hiệu ngõ ra được mã hóa nhị phân. Mạch mã hóa hoạt động ngược lại với mạch giải mã.

Hình 5.3 biểu diễn mạch mã hóa ngõ vào 4-bit one-hot thành ngõ ra nhị phân 2-bit, và Bảng 5.3 biểu diễn bảng trạng thái chức năng mã hóa. Tuy nhiên, mạch mã hóa chỉ

a	b
0001	00
0010	01
0100	10
1000	11
????	??

Bảng 5.2: Bảng trạng thái cho mạch mã hóa 4 thành 2.

hoạt động như mong đợi khi tín hiệu ngõ vào được mã hóa one-hot. Đối với tất cả các giá trị ngõ vào khác, giá trị ngõ ra là không xác định. Vì chúng ta không thể mô tả một hàm có ngõ ra không xác định, nên chúng ta sử dụng một phép gán mặc định để bắt tất cả các mẫu ngõ vào không xác định.

Mã Chisel sau gán giá trị mặc định là 00 và sau đó sử dụng câu lệnh switch cho các giá trị ngõ vào hợp lệ.

```
b := "b00".U
switch (a) {
  is ("b0001".U) { b := "b00".U}
  is ("b0010".U) { b := "b01".U}
  is ("b0100".U) { b := "b10".U}
  is ("b1000".U) { b := "b11".U}
}
```

5.4 Bài tập

Mô tả mạch tổ hợp để chuyển đổi ngõ vào nhị phân 4-bit thành giá trị mã hóa hiển thị LED 7-đoạn. Các bạn có thể định nghĩa mã cho các chữ số thập phân, đây là cách sử dụng để khởi tạo ban đầu của hiển thị LED 7-đoạn, hoặc bổ sung thêm định nghĩa mã cho các bit còn lại để có thể hiển thị tất cả 16 giá trị của chữ số trong hệ thập lục phân. Nếu bo mạch FPGA của các bạn có LED 7-đoạn, hãy kết nối ngõ vào của mạch với 4 công tắc gạt hoặc nút nhấn và nối ngõ ra với LED 7-đoạn.

6 Các khối xây dựng mạch tuần tự

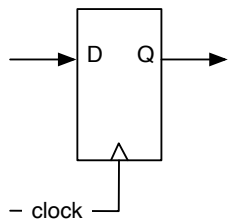
Mạch tuần tự là mạch điện mà giá trị ngõ ra phụ thuộc vào giá trị ngõ vào *và* giá trị trước đó. Vì chúng ta quan tâm đến thiết kế đồng bộ (thiết kế có xung clock), nên chúng ta muốn nói đến mạch tuần tự đồng bộ khi chúng ta nói về mạch tuần tự.[1] Để xây dựng các mạch tuần tự, chúng ta cần các phần tử có thể lưu trữ trạng thái: được gọi là thanh ghi.

6.1 Các thanh ghi

Các thành phần cơ bản để xây dựng các mạch tuần tự là thanh ghi. Một thanh ghi là tập hợp gồm nhiều flip-flop D. Một flip-flop D giữ lại giá trị ngõ vào của nó ở cạnh lên của xung clock và lưu trữ nó ở ngõ ra. Ngoài ra, nói cách khác: thanh ghi cập nhật giá trị ngõ ra của nó với giá trị của ngõ vào ở cạnh lên của xung clock.

Hình 6.1 biểu diễn biểu tượng trong sơ đồ mạch điện của thanh ghi. Nó chứa ngõ vào `D` và ngõ ra `Q`. Mỗi thanh ghi còn chứa tín hiệu ngõ vào `clock`. Vì tín hiệu xung clock toàn cục này được kết nối với tất cả các thanh ghi trong mạch đồng bộ, nên nó thường không được vẽ trong sơ đồ mạch điện. Hình tam giác nhỏ ở dưới tượng trưng cho ngõ vào xung clock và cho chúng ta biết rằng đây là một thanh ghi. Chúng ta bỏ qua tín hiệu xung clock trong các sơ đồ mạch điện sau. Việc bỏ qua tín hiệu xung clock toàn cục

[1] Chúng ta cũng có thể xây dựng các mạch tuần tự với logic và hồi tiếp bất đồng bộ, nhưng đây là một chủ đề cụ thể và không thể diễn đạt bằng Chisel.



Hình 6.1: Thanh ghi dựa trên Flip-flop D.

cũng được phản ánh trong Chisel khi không cần có kết nối rõ ràng tín hiệu với ngõ vào xung clock của thanh ghi.

Trong Chisel, một thanh ghi với ngõ vào `d` và ngõ ra `q` được định nghĩa bởi:

```
val q = RegNext(d)
```

Lưu ý rằng chúng ta không cần kết nối xung clock với thanh ghi, Chisel đã ngầm thực hiện điều này. Ngõ vào và ngõ ra của thanh ghi có thể là các kiểu phức tạp tùy ý được tạo ra từ sự kết hợp của các véc-tơ và các bundle.

Một thanh ghi cũng có thể được định nghĩa và sử dụng theo hai bước:

```
val delayReg = Reg(UInt(4.W))

delayReg := delayIn
```

Đầu tiên, chúng ta định nghĩa thanh ghi và cho nó một cái tên. Thứ hai, chúng ta kết nối tín hiệu `delayIn` với ngõ vào của thanh ghi. Cũng lưu ý rằng tên của thanh ghi chứa chuỗi `Reg`. Để dễ dàng phân biệt giữa mạch tổ hợp và mạch tuần tự, thông thường chúng ta sử dụng nhãn `Reg` như một phần của tên. Ngoài ra, lưu ý rằng các tên trong Scala (và do đó cũng trong Chisel) thường ở là CamelCase. Tên biến bắt đầu bằng chữ thường và các lớp bắt đầu bằng chữ hoa.

Một thanh ghi cũng có thể được khởi tạo khi reset. Tín hiệu `reset`, giống như tín hiệu `clock`, ẩn chứa trong Chisel. Chúng ta cung cấp giá trị cho reset, ví dụ là 0, như là tham số cho bộ tạo thanh ghi `RegInit`. Ngõ vào cho thanh ghi được kết nối với một lệnh gán trong Chisel.

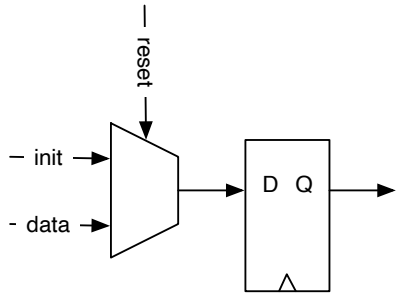
```
val valReg = RegInit(0.U(4.W))

valReg := inVal
```

Việc thực hiện mặc định reset trong Chisel là reset đồng bộ[2]. Với reset đồng bộ, không cần thay đổi trên flip-flop D, chỉ cần thêm mạch đa hợp[3] vào ngõ vào để chọn giữa giá trị khởi tạo khi reset và giá trị dữ liệu.

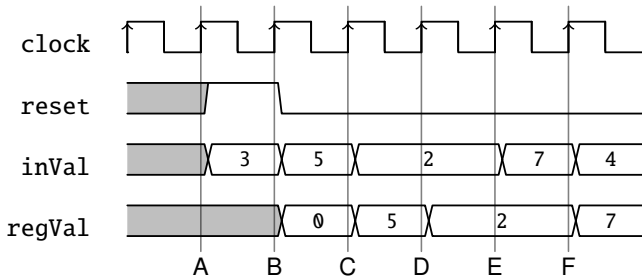
Hình 6.2 biểu diễn sơ đồ mạch điện của một thanh ghi với reset đồng bộ trong đó reset sẽ điều khiển mạch đa hợp. Tuy nhiên, vì reset đồng bộ được sử dụng khá thường xuyên, nên các flip-flop FPGA hiện đại chứa ngõ vào reset (và set) đồng bộ để không lãng phí tài nguyên LUT cho mạch đa hợp.

[2] Hỗ trợ cho reset bất đồng bộ hiện đang được phát triển

[3] Các flip-flop trong FPGA hiện tại chứa ngõ vào reset đồng bộ. Do đó, không cần thêm tài nguyên nào cho mạch đa hợp.



Hình 6.2: Thanh ghi dựa trên flip-flop D với reset đồng bộ.

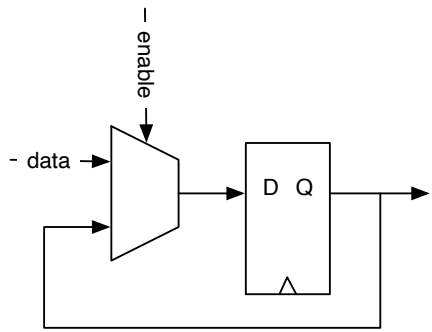


Hình 6.3: Dạng sóng của thanh ghi với tín hiệu reset.

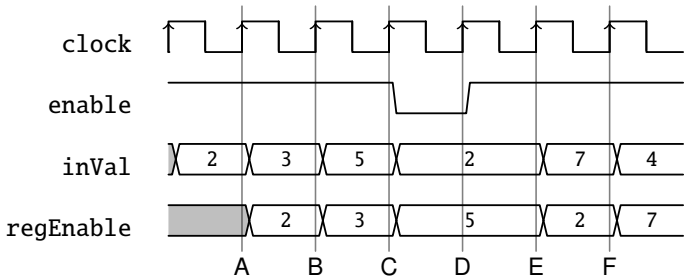
Các mạch tuần tự thay đổi giá trị của chúng theo thời gian. Do đó, hành vi của chúng có thể được mô tả bằng một biểu đồ hiển thị các tín hiệu theo thời gian. Biểu đồ như vậy được gọi là dạng sóng biểu đồ thời gian.

Hình 6.3 biểu diễn dạng sóng của thanh ghi với tín hiệu reset và một số dữ liệu ngõ vào được áp vào nó. Thời gian tăng dần từ trái sang phải. Ở phía trên hình, chúng ta thấy xung clock điều khiển mạch điện. Trong chu kỳ xung clock đầu tiên, trước khi reset, nội dung thanh ghi là không xác định. Trong chu kỳ đồng hồ thứ hai, reset được đưa lên ở mức cao và ở cạnh lên của chu kỳ xung clock này (được gán nhãn B), thanh ghi nhận giá trị ban đầu `0`. Ngõ vào `inVal` bị bỏ qua. Trong chu kỳ xung clock tiếp theo, `reset` bằng `0` và giá trị của `inVal` được ghi lại ở cạnh lên tiếp theo (được gán nhãn C). Từ đó trở đi `reset` vẫn giữ nguyên giá trị là `0`, và ngõ ra thanh ghi đi sau tín hiệu ngõ vào với độ trễ một chu kỳ xung clock.

Dạng sóng là một công cụ tuyệt vời để xác định hành vi của mạch bằng đồ họa. Đặc biệt là trong các mạch phức tạp hơn, nơi nhiều phép toán tiến hành song song và dữ liệu



Hình 6.4: Thanh ghi dựa trên flip-flop D với tín hiệu cho phép.



Hình 6.5: Biểu đồ dạng sóng của thanh ghi với tín hiệu cho phép.

di chuyển theo đường ống (pipeline) xuyên suốt mạch, biểu đồ thời gian trở nên thuận tiện. Các trình kiểm tra Chisel cũng có thể tạo ra các dạng sóng trong quá trình kiểm tra, dạng sóng có thể được hiển thị bằng công cụ xem dạng sóng và được sử dụng để gỡ lỗi.

Một mẫu thiết kế tiêu biểu là một thanh ghi có tín hiệu cho phép. Chỉ khi tín hiệu cho phép là `true` (mức cao), thanh ghi mới giữ lại giá trị ngõ vào; ngược lại, nó vẫn giữ nguyên giá trị cũ. Tín hiệu cho phép có thể được thực hiện, tương tự như tín hiệu reset đồng bộ, với một mạch đa hợp ở ngõ vào của thanh ghi. Một ngõ vào cho mạch đa hợp là hồi tiếp ngõ ra của thanh ghi.

Hình 6.4 biểu diễn sơ đồ mạch điện của một thanh ghi có tín hiệu cho phép. Vì đây cũng là một mẫu thiết kế phổ biến, các flip-flop trong FPGA hiện đại chứa ngõ vào cho phép dành riêng và không cần thêm tài nguyên.

Hình 6.5 biểu diễn một dạng sóng ví dụ cho thanh ghi có tín hiệu cho phép. Hầu hết thời gian, bật tín hiệu cho phép ở mức cao (`true`) và thanh ghi đi sau ngõ vào với độ

trễ một chu kỳ xung clock. Chỉ trong chu kỳ xung clock thứ tư, `enable` ở mức thấp, và thanh ghi giữ giá trị của nó (giá trị bằng 5) ở cạnh lên của nhãn D.

Thanh ghi với tín hiệu cho phép có thể được mô tả trong một vài dòng của mã Chisel với bản cập nhật có điều kiện:

```
val enableReg = Reg(UInt(4.W))

when (enable) {
  enableReg := inVal
}
```

Thanh ghi với tín hiệu cho phép có thể được reset:

```
val resetEnableReg = RegInit(0.U(4.W))

when (enable) {
  resetEnableReg := inVal
}
```

Thanh ghi cũng có thể là một phần của một biểu thức. Mạch điện sau đây dò cạnh lên của tín hiệu bằng cách so sánh giá trị hiện tại của nó với giá trị từ chu kỳ xung clock sau cùng.

```
val risingEdge = din & !RegNext(din)
```

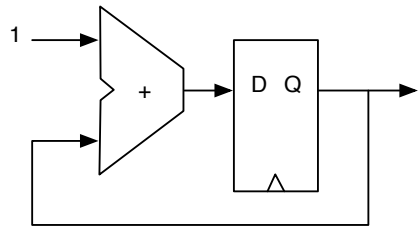
Bây giờ chúng ta đã khám phá tất cả các cách sử dụng cơ bản của một thanh ghi, chúng ta sử dụng tốt các thanh ghi đó và xây dựng các mạch tuần tự thú vị hơn.

6.2 Mạch đếm

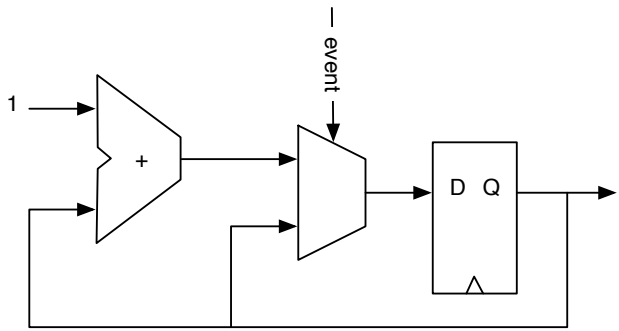
Một trong những mạch tuần tự cơ bản nhất là mạch đếm. Ở dạng đơn giản nhất, mạch đếm là một thanh ghi nơi mà ngõ ra được kết nối với một mạch cộng và ngõ ra của mạch cộng được kết nối với ngõ vào của thanh ghi. Hình 6.6 biểu diễn một mạch đếm chạy tự do (free-running counter).

Mạch đếm chạy tự do có một thanh ghi 4-bit đếm từ 0 đến 15 và sau đó lại quay về 0 chạy tiếp. Một mạch đếm cũng phải được reset về một giá trị đã biết.

```
val cntReg = RegInit(0.U(4.W))

cntReg := cntReg + 1.U
```



Hình 6.6: Mạch cộng và kết quả thanh ghi trong mạch đếm.



Hình 6.7: Các sự kiện đếm.

Khi chúng ta muốn đếm các sự kiện, chúng ta sử dụng một điều kiện để tăng giá trị cho mạch đếm, như trong Hình 6.7 và trong đoạn mã sau.

```
val cntEventsReg = RegInit(0.U(4.W))
when(event) {
  cntEventsReg := cntEventsReg + 1.U
}
```

6.2.1 Đếm lên và đếm xuống

Để đếm lên một giá trị và sau đó khởi động lại với giá trị `0`, chúng ta cần so sánh giá trị mạch đếm với một giá trị hằng số tối đa, ví dụ: với câu lệnh điều kiện `when`.

```
val cntReg = RegInit(0.U(8.W))

cntReg := cntReg + 1.U
when(cntReg === N) {
  cntReg := 0.U
}
```

Chúng ta còn có thể sử sụng mạch đa hợp cho mạch đếm:

```
val cntReg = RegInit(0.U(8.W))

cntReg := Mux(cntReg === N, 0.U, cntReg + 1.U)
```

Nếu đang ở trạng thái đếm xuống, thì chúng ta bắt đầu bằng cách đặt lại thanh ghi mạch đếm với giá trị cực đại và reset mạch đếm về giá trị đó khi đạt giá trị 0.

```
val cntReg = RegInit(N)

cntReg := cntReg - 1.U
when(cntReg === 0.U) {
  cntReg := N
}
```

Khi chúng ta đang viết mã code và sử dụng nhiều mạch đếm hơn, chúng ta có thể định nghĩa một hàm với một tham số để sinh tạo mạch đếm cho chúng ta.

```
// This function returns a counter
def genCounter(n: Int) = {
  val cntReg = RegInit(0.U(8.W))
```

```
    cntReg := Mux(cntReg === n.U, 0.U, cntReg + 1.U)
    cntReg
  }

  // now we can easily create many counters
  val count10 = genCounter(10)
  val count99 = genCounter(99)
```

Câu lệnh cuối cùng của hàm `genCounter` là giá trị trả về của hàm, trong ví dụ này là thanh ghi đếm `cntReg`.

Lưu ý rằng trong tất cả các ví dụ, mạch đếm có các giá trị giữa `0` và `N`, bao gồm `N`. Nếu chúng ta muốn đếm 10 chu kỳ xung clock, chúng ta cần đặt `N` là 9. Đặt `N` bằng 10 là ví dụ cổ điển về lỗi logic off-by-one.

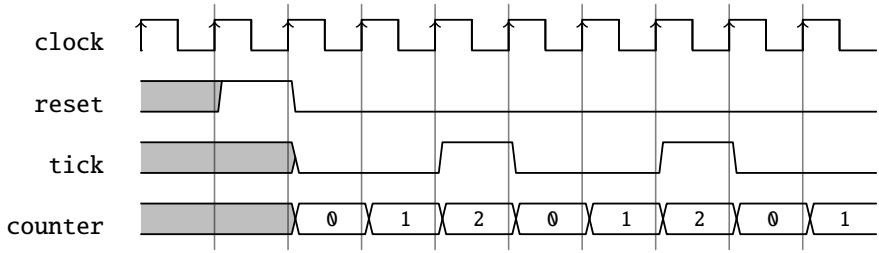
6.2.2 Tạo thời gian với mạch đếm

Bên cạnh việc đếm các sự kiện, mạch đếm thường được sử dụng để tạo ra khái niệm về thời gian (thời gian giống như thời gian trên đồng hồ treo tường). Mạch điện đồng bộ chạy với xung clock đồng hồ có tần số cố định. Mạch sẽ hoạt động trong những thời điểm ứng với chu kỳ xung clock này. Không có khái niệm thời gian trong mạch số ngoài việc đếm các thời điểm với xung clock. Nếu chúng ta biết tần số xung clock, chúng ta có thể sinh ra các mạch tạo các sự kiện định thời, chẳng hạn như nhấp nháy đèn LED ở một số tần số như đã trình bày trong ví dụ "Hello World".

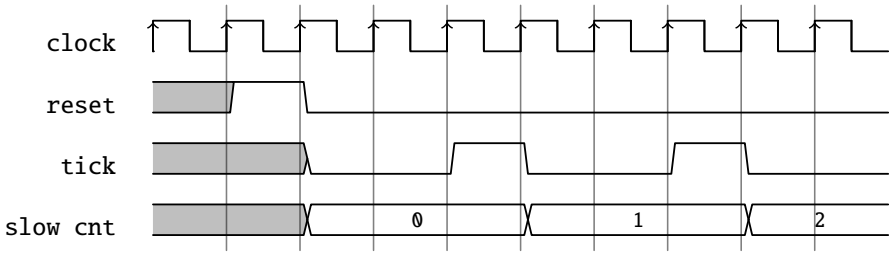
Một thực tế phổ biến là để tạo một chu kỳ đơn *tick* với tần số f_{tick} mà chúng ta cần nó trong mạch của mình. Tín hiệu tick đó xảy ra sau mỗi n chu kỳ xung clock, trong đó $n = f_{clock}/f_{tick}$ và tick có độ dài chính xác một chu kỳ xung clock. Tín hiệu tick này *không* được sử dụng như một xung clock được dẫn xuất, nhưng là một tín hiệu cho phép các thanh ghi trong mạch sẽ hoạt động hợp lý ở tần số f_{tick}. Hình 6.8 biểu diễn một ví dụ về một tín hiệu tick được tạo ra sau mỗi 3 chu kỳ xung clock.

Trong mạch điện sau, chúng ta mô tả một mạch đếm sẽ đếm từ `0` đến giá trị cực đại `N - 1`. Khi đạt đến giá trị cực đại, `tick` là `true` cho một chu kỳ đơn, và mạch đếm được reset về `0`. Khi chúng ta đếm từ `0` đến `N - 1`, chúng ta tạo một tick hợp lý mỗi chu kỳ xung clock `N`.

```
  val tickCounterReg = RegInit(0.U(4.W))
  val tick = tickCounterReg === (N-1).U

  tickCounterReg := tickCounterReg + 1.U
  when (tick) {
    tickCounterReg := 0.U
```



Hình 6.8: Sơ đồ dạng sóng để tạo một tick tần số chậm.



Hình 6.9: Sử dụng tick tần số chậm.

```
}
```

Thời gian logic của một tick với mỗi n chu kỳ xung clock sau đó có thể được sử dụng để cải tiến các phần khác của mạch với xung clock logic chậm hơn này. Trong đoạn mã sau, chúng ta chỉ sử dụng một mạch đếm khác tăng dần lên 1 sau mỗi n chu kỳ xung clock.

```
val lowFrequCntReg = RegInit(0.U(4.W))
when (tick) {
  lowFrequCntReg := lowFrequCntReg + 1.U
}
```

Hình 6.9 biểu diễn dạng sóng của tín hiệu tick và mạch đếm chậm tăng lên mỗi tick (n clock cycles).

Ví dụ về việc sử dụng xung clock *logic* chậm hơn này là: nhấp nháy đèn LED, tạo tốc độ baud (baud rate) cho bus nối tiếp, tạo tín hiệu cho đa hợp hiển thị LED 7 đoạn và lấy mẫu con các giá trị ngõ vào để chống dội các nút và công tắc.

Mặc dù suy luận về giá trị độ rộng sẽ hình thành kích thước các thanh ghi, nhưng tốt

hơn nên chỉ định rõ ràng giá trị độ rộng với kiểu khi định nghĩa thanh ghi hoặc với giá trị khởi tạo. Định nghĩa giá trị độ rộng rõ ràng có thể tránh được những bất ngờ khi giá trị reset của `0.U` dẫn đến kết quả mạch đếm có độ rộng là một bit.

6.2.3 Mạch đếm Nerd

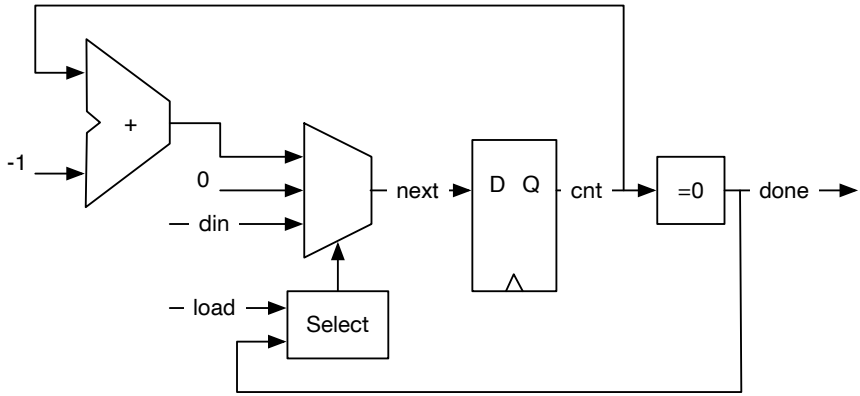
Đôi khi, nhiều người trong chúng ta cảm thấy mình giống như một người chú tâm về một vấn đề nào đó (nerd). Ví dụ: chúng ta muốn thiết kế một phiên bản tối ưu hóa cao cho việc tạo mạch đếm/tick. Một mạch đếm chuẩn cần các tài nguyên sau: một thanh ghi, một mạch cộng (hoặc mạch trừ) và một mạch so sánh. Chúng ta không thể làm gì nhiều về thanh ghi hoặc mạch cộng. Nếu đếm lên, chúng ta cần so sánh với một số, đó là một chuỗi bit. Mạch so sánh có thể được xây dựng từ các cổng đảo cho các số 0 trong chuỗi bit và một cổng AND lớn. Khi đếm xuống đến 0, mạch so sánh là một cổng NOR lớn, có thể đơn giản hơn mạch so sánh một chút với một hằng số trong ASIC. Trong FPGA, nơi cổng logic được xây dựng từ các bảng tra, không có sự khác biệt giữa việc so sánh với 0 hoặc 1. Yêu cầu về tài nguyên là như nhau đối với mạch đếm lên và đếm xuống.

Tuy nhiên, vẫn còn một mẹo nữa mà một nhà thiết kế phần cứng thông minh có thể thực hiện. Cho đến nay, đếm lên hoặc đếm xuống cần phải so sánh với tất cả các bit đếm. Điều gì sẽ xảy ra nếu chúng ta đếm từ N-2 xuống -1? Một số âm có bit MSB (Most Significant Bit) được đặt thành 1 và một số dương với bit này được đặt thành 0. Chúng ta chỉ cần kiểm tra bit này để phát hiện rằng mạch đếm của chúng ta đạt đến -1 hay chưa. Ở đây, mạch đếm được tạo bởi một nerd:

```
val MAX = (N - 2).S(8.W)
val cntReg = RegInit(MAX)
io.tick := false.B

cntReg := cntReg - 1.S
when(cntReg(7)) {
  cntReg := MAX
  io.tick := true.B
}
```

6.2.4 Bộ định thời

Một dạng định thời khác mà chúng ta có thể tạo, là bộ định thời one-shot. Bộ định thời one-shot cũng giống như bộ định thời trong nhà bếp: các bạn đặt số phút và nhấn nút bắt đầu. Khi hết khoảng thời gian được chỉ định, âm báo sẽ phát ra. Bộ định thời kỹ thuật số



Hình 6.10: Bộ định thời one-shot.

được nạp với thời gian theo chu kỳ xung clock. Sau đó, nó đếm xuống cho đến khi đạt đến giá trị 0. Ở giá trị 0, bộ định thời xác nhận với tín hiệu *done*.

Hình 6.10 biểu diễn sơ đồ khối của một bộ định thời. Thanh ghi có thể được nạp với giá trị `din` bởi tín hiệu xác nhận `load`. Khi tín hiệu `load` không được xác nhận, việc đếm xuống được lựa chọn (bằng cách chọn `cntReg - 1` như ngõ vào cho thanh ghi). Khi mạch đếm đạt giá trị `0`, tín hiệu `done` được xác nhận và mạch đếm dừng đếm bằng cách chọn ngõ vào của mạch đa hợp cung cấp giá trị `0`.

Listing 6.1 biểu diễn mã Chisel cho bộ định thời. Chúng ta sử dụng một thanh ghi 8-bit `reg`, được reset về `0`. Giá trị boolean `done` là kết quả của việc so sánh `reg` với `0`. Đối với mạch đa hợp ngõ vào, chúng ta đưa vào dây nối trung gian `next` với giá trị mặc định là `0`. Khối `when/elsewhen` đưa vào hai ngõ vào khác với chức năng lựa chọn tín hiệu. Tín hiệu `load` có mức độ ưu tiên qua độ lựa chọn giảm dần. Dòng cuối cùng kết nối mạch đa hợp, được biểu diễn bởi `next`, với ngõ vào của thanh ghi `reg`.

Nếu chúng ta nhắm đến mã chương trình ngắn gọn hơn một chút, thì chúng ta có thể gán trực tiếp các giá trị của mạch đa hợp cho thanh ghi `reg`, thay vì sử dụng dây trung gian `next`.

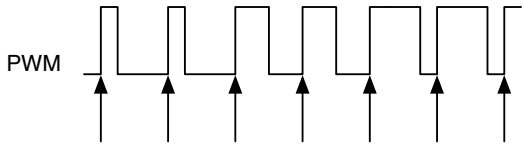
6.2.5 Điều biến độ rộng xung

Điều biến độ rộng xung (Pulse-width modulation - PWM) là tín hiệu có chu kỳ không đổi và điều biến thời gian tín hiệu là *mức cao* trong chu kỳ đó.

Hình 6.11 biểu diễn một tín hiệu PWM. Các mũi tên trỏ đến điểm bắt đầu các khoảng thời gian của tín hiệu. Phần trăm thời gian tín hiệu ở mức cao, còn được gọi là chu kỳ

```
val cntReg = RegInit(0.U(8.W))
val done = cntReg === 0.U

val next = WireDefault(0.U)
when (load) {
  next := din
} .elsewhen (!done) {
  next := cntReg - 1.U
}
cntReg := next
```

Listing 6.1: Bộ định thời one-shot



Hình 6.11: Điều biến độ rộng xung.

nhiệm vụ (duty cycle). Trong hai chu kỳ đầu tiên, chu kỳ nhiệm vụ là 25 %, trong hai chu kỳ tiếp theo 50 %, và trong hai chu kỳ cuối là 75 %. Độ rộng xung được điều biến giữa 25 % và 75 %.

Thêm một mạch lọc thấp qua vào tín hiệu PWM sẽ cho ra một bộ chuyển đổi số sang tương tự đơn giản. Mạch lọc thấp qua có thể ở dạng đơn giản chỉ với điện trở và tụ điện.

Ví dụ đoạn mã chương trình sau sẽ tạo dạng sóng 3 chu kỳ xung clock mức cao mỗi 10 chu kỳ xung clock.

```
def pwm(nrCycles: Int, din: UInt) = {
  val cntReg = RegInit(0.U(unsignedBitLength(nrCycles-1).W))
  cntReg := Mux(cntReg === (nrCycles-1).U, 0.U, cntReg + 1.U)
  din > cntReg
}

val din = 3.U
val dout = pwm(10, din)
```

Chúng ta sử dụng hàm cho bộ tạo PWM để cung cấp một thành phần nhẹ, tái sử dụng

được. Hàm có hai tham số: một số nguyên Scala cấu hình PWM với số chu kỳ xung clock (`nrCycles`), và một dây nối Chisel (`din`) để cung cấp giá trị chu kỳ nhiệm vụ (pulswidth) cho tín hiệu ngõ ra PWM. Chúng ta sử dụng một mạch đa hợp trong ví dụ này để biểu diễn mạch đếm. Dòng cuối cùng của hàm: so sánh giá trị mạch đếm với giá trị ngõ vào `din` để trả về tín hiệu PWM. Biểu thức cuối cùng trong hàm là giá trị trả về, trong trường hợp của chúng ta, dây được nối với hàm so sánh.

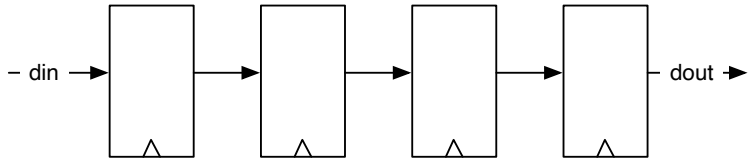
Chúng ta sử dụng hàm `unsignedBitLength(n)` để xác định số bit cho mạch đếm `cntReg` cần để biểu diễn các số không dấu lên đến (và bao gồm) n.[4] Chisel còn có hàm `signedBitLength` để cung cấp số bit nhằm biểu diễn cho một số có dấu.

Một ứng dụng khác là sử dụng PWM để làm mờ đèn LED. Trong trường hợp đó, mắt đóng vai trò như bộ lọc thông thấp. Chúng ta mở rộng ví dụ trên để hướng quá trình tạo PWM bằng một hàm tam giác. Kết quả là một đèn LED có cường độ sáng thay đổi liên tục.

```
val FREQ = 100000000 // a 100 MHz clock input
val MAX = FREQ/1000  // 1 kHz

val modulationReg = RegInit(0.U(32.W))

val upReg = RegInit(true.B)

when (modulationReg < FREQ.U && upReg) {
  modulationReg := modulationReg + 1.U
} .elsewhen (modulationReg === FREQ.U && upReg) {
  upReg := false.B
} .elsewhen (modulationReg > 0.U && !upReg) {
  modulationReg := modulationReg - 1.U
} .otherwise { // 0
  upReg := true.B
}

// divide modReg by 1024 (about the 1 kHz)
val sig = pwm(MAX, modulationReg >> 10)
```

Chúng ta sử dụng hai thanh ghi cho điều biến: (1) `modulationReg` để đếm lên và đếm xuống và (2) `upReg` như cờ để xác định là đếm lên hay đếm xuống. Chúng ta đếm lên tần số ngõ vào xung clock (100 MHz trong ví dụ), cho ra kết quả tín hiệu 0.5 Hz. Độ dài biểu thức `when/.elsewhen/.otherwise` xử lý việc đếm lên hay đếm xuống và chuyển hướng.

[4] Số bit để biểu diễn một số không dấu n ở dạng nhị phân là $\lfloor log_2(n) \rfloor + 1$.



Hình 6.12: Thanh ghi dịch 4 tầng.

Vì PWM chỉ đếm lên đến phần 1000 của tần số để tạo ra tín hiệu 1 kHz, chúng ta cần chia tín hiệu điều biến cho 1000. Vì phép chia số thực rất tốn kém trong phần cứng, chúng ta chỉ đơn giản dịch 10 lần sang phải, điều này tương đương một phép chia cho $2^{10} = 1024$. Vì chúng ta đã định nghĩa mạch điện PWM như một hàm, nên có thể đơn giản khởi tạo mạch điện đó bằng cách gọi hàm. Dây nối `sig` biểu diễn tín hiệu PWM đã được điều biến.

6.3 Thanh ghi dịch

Thanh ghi dịch là tập hợp của các flip-flop được kết nối thành một chuỗi. Mỗi ngõ ra của một thanh ghi (flip-flop) được nối với ngõ vào của thanh ghi kế tiếp. Hình 6.12 biểu diễn thanh ghi dịch 4 tầng. Mạch điện *dịch* dữ liệu từ trái sang phải ở mỗi tick xung clock. Trong dạng đơn giản này, mạch điện thực hiện độ trễ 4 lần từ `din` đến `dout`.

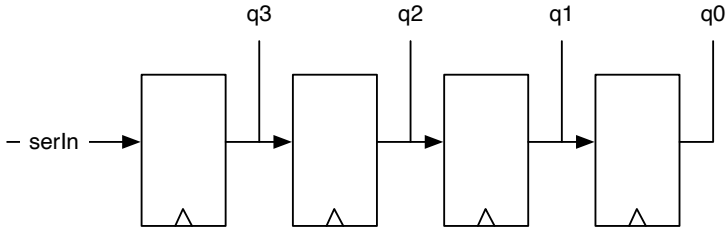
Mã Chisel cho thanh ghi dịch đơn giản này thực hiện: (1) tạo một thanh ghi 4-bit `shiftReg`, (2) ghép nối 3-bit thấp của thanh ghi dịch với ngõ vào `din` cho ngõ vào tiếp theo vào thanh ghi và (3) sử dụng bit MSB (Most Significant Bit) của thanh ghi làm ngõ ra `dout`.

```
val shiftReg = Reg(UInt(4.W))
shiftReg := Cat(shiftReg(2, 0), din)
val dout = shiftReg(3)
```

Thanh ghi dịch thường được sử dụng để chuyển đổi dữ liệu từ nối tiếp sang song song hoặc dữ liệu từ song song sang nối tiếp. Phần 11.2 trình bày cổng nối tiếp sử dụng các thanh ghi dịch dùng cho chức năng nhận và gửi dữ liệu.

6.3.1 Thanh ghi dịch với ngõ ra song song

Cấu hình vào-nối-tiếp ra-song-song của thanh ghi dịch chuyển đổi dòng dữ liệu ngõ vào nối tiếp thành các từ song song. Điều này có thể được sử dụng trong cổng nối tiếp



Hình 6.13: Thanh ghi dịch 4-bit với ngõ ra song song.

(UART) cho chức năng nhận dữ liệu. Hình 6.13 biểu diễn thanh ghi dịch 4-bit, trong đó mỗi ngõ ra flip-flop được nối với một bit ngõ ra. Sau 4 chu kỳ xong clock, mạch điện chuyển một từ (word) với 4-bit dữ liệu nối tiếp thành một từ với 4-bit dữ liệu song song có sẵn trong q. Trong ví dụ này, chúng ta giả sử bit 0 (LSB) được gửi trước và do đó nó đến ở tầng cuối cùng khi chúng ta muốn đọc toàn bộ từ.

Trong đoạn mã Chisel sau, chúng ta khởi tạo thanh ghi dịch `outReg` với giá trị 0. Sau đó chúng ta dịch vào từ bit MSB, có nghĩa là dịch phải. Kết quả song song, q, chỉ là việc đọc giá trị thanh ghi `outReg`.

```
val outReg = RegInit(0.U(4.W))
outReg := Cat(serIn, outReg(3, 1))
val q = outReg
```

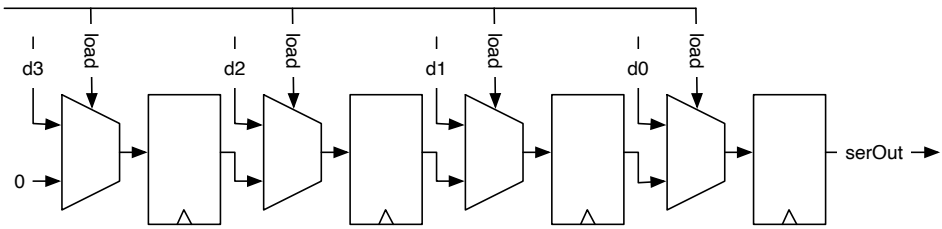
Hình 6.13 biểu diễn thanh ghi dịch 4-bit với hàm ngõ ra song song.

6.3.2 Thanh ghi dịch với tải song song

Cấu hình vào-song-song ra-nối-tiếp của thanh ghi dịch chuyển đổi dòng dữ liệu các từ (bytes) ngõ vào song song thành dòng dữ liệu ngõ ra nối tiếp. Điều này có thể được sử dụng trong cổng nối tiếp (UART) cho chức năng gửi dữ liệu.

Hình 6.14 biểu diễn thanh ghi dịch 4-bit với chức năng tải song song. Mô tả Chisel của chức năng đó tương đối dễ hiểu như sau:

```
val loadReg = RegInit(0.U(4.W))
when (load) {
  loadReg := d
} otherwise  {
  loadReg := Cat(0.U, loadReg(3, 1))
}
```



Hình 6.14: Thanh ghi dịch 4-bit với tải song song.

```
val serOut = loadReg(0)
```

Lưu ý rằng bây giờ chúng đang dịch sang bên phải, nên điền vào các số 0 ở MSB.

6.4 Bộ nhớ

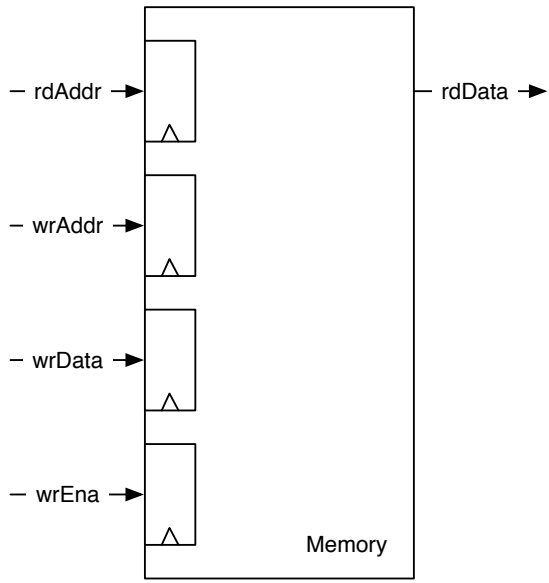
Một bộ nhớ có thể được tạo ra từ một tập hợp các thanh ghi, trong Chisel, một `Reg` của `Vec`. Tuy nhiên, điều này là tốn kém về phần cứng và cấu trúc bộ nhớ lớn hơn được xây dựng dưới dạng SRAM. Đối với ASIC, một trình biên dịch bộ nhớ dùng để xây dựng các bộ nhớ. FPGA chứa các khối bộ nhớ trên chip, còn được gọi là các RAM khối. Các khối bộ nhớ trên chip đó có thể được kết hợp để tạo ra các bộ nhớ lớn hơn. Các bộ nhớ trong FPGA thường có một cổng đọc và một cổng ghi hoặc hai cổng có thể được chuyển đổi giữa đọc và ghi trong thời gian chạy.

FPGA (và cả ASIC) thường hỗ trợ bộ nhớ đồng bộ. Bộ nhớ đồng bộ có các thanh ghi ở ngõ vào (địa chỉ đọc và ghi, dữ liệu ghi và cho phép ghi). Điều đó có nghĩa là dữ liệu đọc có sẵn trong một chu kỳ xung clock sau khi thiết lập địa chỉ.

Hình 6.15 biểu diễn sơ đồ mạch của một bộ nhớ đồng bộ. Bộ nhớ có hai cổng với một cổng đọc và một cổng ghi dữ liệu. Cổng đọc có một ngõ vào duy nhất, địa chỉ đọc (`rdAddr`) và một ngõ ra, dữ liệu đọc (`rdData`). Cổng ghi có ba ngõ vào: địa chỉ (`wrAddr`), dữ liệu được ghi (`wrData`), và chân cho phép ghi (`wrEna`). Lưu ý rằng đối với tất cả các ngõ vào, có một thanh ghi trong bộ nhớ hiển thị hành vi đồng bộ.

Để hỗ trợ bộ nhớ trên chip, Chisel cung cấp hàm tạo bộ nhớ `SyncReadMem`. Listing 6.2 biểu diễn một thành phần `bộ nhớ` thực thi 1 KiB bộ nhớ với dữ liệu ngõ vào và ngõ ra có độ rộng 1 byte (8-bit) và một chân cho phép ghi.

Một câu hỏi thú vị là giá trị nào được trả về từ một lần đọc khi trong cùng một chu kỳ xung clock, một giá trị mới được ghi vào ở cùng địa chỉ được đọc ra. Chúng ta quan tâm đến hành vi đọc-trong-quá-trình-ghi của bộ nhớ. Có ba khả năng: giá trị được ghi mới, giá trị cũ hoặc không xác định (có thể là sự kết hợp của một số bit từ giá trị cũ và một số



Hình 6.15: Bộ nhớ đồng bộ.

```
class Memory() extends Module {
  val io = IO(new Bundle {
    val rdAddr = Input(UInt(10.W))
    val rdData = Output(UInt(8.W))
    val wrEna = Input(Bool())
    val wrData = Input(UInt(8.W))
    val wrAddr = Input(UInt(10.W))
  })

  val mem = SyncReadMem(1024, UInt(8.W))

  io.rdData := mem.read(io.rdAddr)

  when(io.wrEna) {
    mem.write(io.wrAddr, io.wrData)
  }
}
```

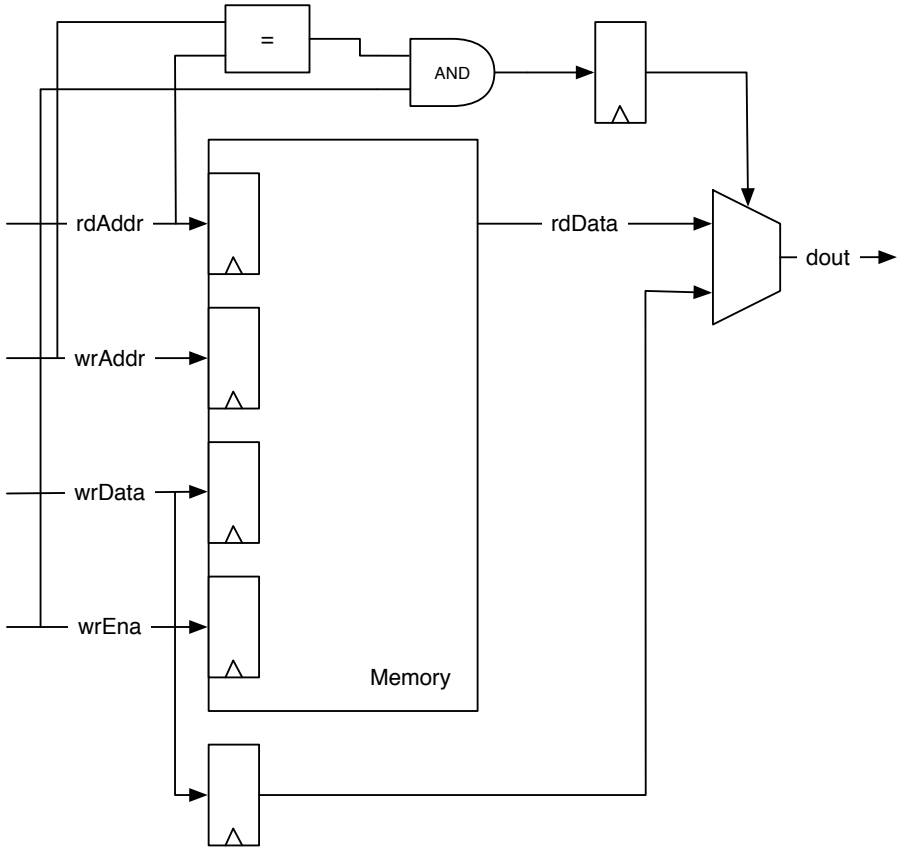
Listing 6.2: 1 KiB bộ nhớ đồng bộ.

dữ liệu được ghi mới). Khả năng nào khả dụng trong FPGA phụ thuộc vào loại FPGA và đôi khi có thể được xác định. Ở các tài liệu Chisel, dữ liệu đọc là không xác định.

Nếu muốn đọc ra giá trị được ghi mới, chúng ta có thể xây dựng một mạch điện chuyển tiếp để phát hiện rằng các địa chỉ bằng nhau và *chuyển tiếp* dữ liệu ghi. Hình 6.16 biểu diễn bộ nhớ với mạch chuyển tiếp. Địa chỉ đọc và ghi được so sánh và kiểm soát với chân cho phép ghi để chọn giữa đường chuyển tiếp của dữ liệu ghi hoặc dữ liệu đọc trong bộ nhớ. Dữ liệu ghi bị trễ một chu kỳ xung clock với một thanh ghi.

Listing 6.3 biểu diễn mã Chisel cho bộ nhớ đồng bộ bao gồm mạch điện chuyển tiếp. Chúng ta cần lưu trữ dữ liệu ghi vào một thanh ghi (`wrDataReg`) khả dụng trong chu kỳ xung clock tiếp theo để phù hợp với bộ nhớ đồng bộ, thanh ghi đó cũng sẽ cung cấp giá trị đọc trong chu kỳ xung clock tiếp theo. Chúng ta so sánh hai địa chỉ ngõ vào (`wrAddr` và `rdAddr`) và kiểm tra `wrEna` là đúng với điều kiện chuyển tiếp hay không. Điều kiện này cũng bị trễ một chu kỳ xung clock. Mạch đa hợp sẽ lựa chọn giữa dữ liệu (ghi) chuyển tiếp hoặc dữ liệu đọc từ bộ nhớ.

Chisel còn cung cấp `Mem` để biểu diễn bộ nhớ với hoạt động ghi đồng bộ và đọc bất đồng bộ. Vì loại bộ nhớ này thường không khả dụng trực tiếp trong FPGA, nên công cụ tổng hợp sẽ xây dựng nó mà không dùng các flip-flop. Do đó, lời khuyên là các bạn nên sử dụng `SyncReadMem`.



Hình 6.16: Bộ nhớ đồng bộ với chuyển tiếp cho một hành vi đọc-trong-quá-trình-ghi đã được định nghĩa.

```
class ForwardingMemory() extends Module {
  val io = IO(new Bundle {
    val rdAddr = Input(UInt(10.W))
    val rdData = Output(UInt(8.W))
    val wrEna = Input(Bool())
    val wrData = Input(UInt(8.W))
    val wrAddr = Input(UInt(10.W))
  })

  val mem = SyncReadMem(1024, UInt(8.W))

  val wrDataReg = RegNext(io.wrData)
  val doForwardReg = RegNext(io.wrAddr === io.rdAddr &&
      io.wrEna)

  val memData = mem.read(io.rdAddr)

  when(io.wrEna) {
    mem.write(io.wrAddr, io.wrData)
  }

  io.rdData := Mux(doForwardReg, wrDataReg, memData)
}
```

Listing 6.3: Bộ nhớ với mạch chuyển tiếp.

6.5 Bài tập

Sử dụng mạch mã hóa LED 7-đoạn từ bài tập trước và thêm mạch đếm 4-bit làm ngõ vào để hiển thị lần lượt từ `0` đến `F`. Khi kết nối trực tiếp mạch đếm này với tín hiệu xung clock của bo mạch FPGA, các bạn sẽ thấy tất cả 16 số chồng lên nhau (tất cả 7 đoạn sẽ sáng lên). Do đó, bạn cần phải làm chậm mạch đếm lại. Tạo một mạch đếm khác có thể tạo ra một tín hiệu *tick* chu kỳ đơn có chu kỳ mỗi 500 mili-giây. Sử dụng tín hiệu đó làm tín hiệu điều khiển cho mạch đếm 4-bit.

Xây dựng dạng sóng PWM với hàm sinh tạo và đặt giá trị ngưỡng với hàm (hàm tam giác hoặc hàm sin). Một hàm tam giác có thể được tạo ra bằng cách đếm lên và đếm xuống. Một hàm sin với việc sử dụng bảng tra mà các bạn có thể tạo ra với vài dòng mã Scala (xem Phần 10.3). Điều khiển đèn LED trên bo mạch FPGA với hàm PWM đã được điều biến đó. Tín hiệu PWM của các bạn sẽ là tần số nào? Bộ điều khiển đang chạy với tần số nào?

Các thiết kế mạch số thường được phác thảo dưới dạng một mạch điện trên giấy. Không phải tất cả các chi tiết cần phải được hiển thị. Chúng ta sử dụng sơ đồ khối, giống như trong các hình trong cuốn sách này. Đó là một kỹ năng quan trọng để có thể dịch trôi chảy giữa biểu diễn sơ đồ bản vẽ mạch điện và mô tả Chisel. Vẽ sơ đồ khối cho các mạch sau:

```
val dout = WireDefault(0.U)

switch(sel) {
  is(0.U) { dout := 0.U }
  is(1.U) { dout := 11.U }
  is(2.U) { dout := 22.U }
  is(3.U) { dout := 33.U }
  is(4.U) { dout := 44.U }
  is(5.U) { dout := 55.U }
}
```

Đây là một mạch phức tạp hơn một chút, chứa một thanh ghi:

```
val regAcc = RegInit(0.U(8.W))

switch(sel) {
  is(0.U) { regAcc := regAcc}
  is(1.U) { regAcc := 0.U}
  is(2.U) { regAcc := regAcc + din}
  is(3.U) { regAcc := regAcc - din}
}
```

7 Xử lý ngõ vào

Các tín hiệu ngõ vào từ thế giới bên ngoài vào mạch đồng bộ thường không đồng bộ với xung clock; chúng bất đồng bộ. Một tín hiệu ngõ vào có thể đến từ một nguồn không có quá trình chuyển tiếp rõ ràng từ 0 sang 1 hoặc 1 sang 0. Một ví dụ là nút bật hoặc công tắc gạt. Tín hiệu ngõ vào có thể bị nhiễu với các xung nhọn có thể kích hoạt quá trình chuyển tiếp trong mạch đồng bộ. Chương này mô tả các mạch điện xử lý các điều kiện ngõ vào như vậy.

Hai vấn đề sau, công tắc chống dội và lọc nhiễu, cũng có thể được giải quyết bởi các thành phần tương tự, bên ngoài. Tuy nhiên, sẽ hiệu quả hơn (chi phí) để xử lý những vấn đề đó trong miền kỹ thuật số.

7.1 Ngõ vào bất đồng bộ

Tín hiệu ngõ vào không đồng bộ với xung clock hệ thống được gọi là tín hiệu bất đồng bộ. Những tín hiệu đó có thể vi phạm thời gian thiết lập (setup time) và lưu giữ (hold time) của ngõ vào flip-flop. Vi phạm này có thể dẫn đến sự bất ổn định (Metastability) của flip-flop. Sự bất ổn định có thể dẫn đến giá trị ngõ ra từ 0 đến 1 hoặc nó có thể tạo ra dao động. Tuy nhiên, sau một thời gian, flip-flop sẽ ổn định ở mức 0 hoặc 1.

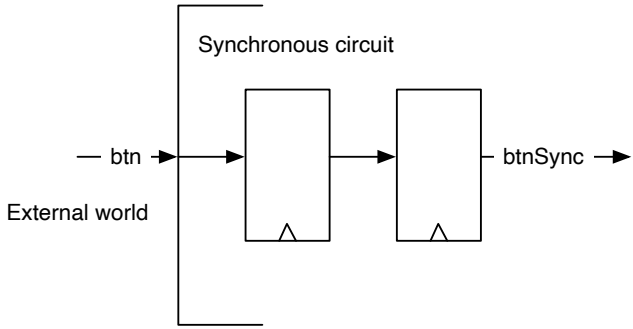
Chúng ta không thể tránh được sự bất ổn định, nhưng chúng ta có thể ngăn chặn những ảnh hưởng của nó. Một giải pháp cổ điển là sử dụng hai flip-flop ở ngõ vào. Giả định là: khi flip-flop đầu tiên trở nên bất ổn định, nó sẽ phân giải thành trạng thái ổn định trong chu kỳ xung clock để thời gian thiết lập và thời gian lưu giữ của flip-flop thứ hai sẽ không bị vi phạm.

Hình 7.1 thể hiện ranh giới giữa mạch đồng bộ và thế giới bên ngoài. Mạch đồng bộ hóa ngõ vào bao gồm hai flip-flop. Mã Chisel cho mạch đồng bộ ngõ vào là một dòng duy nhất tạo ra hai thanh ghi.

```
val btnSync = RegNext(RegNext(btn))
```

Tất cả các tín hiệu bên ngoài bất đồng bộ cần một bộ đồng bộ ngõ vào.[1] Chúng ta cũng

[1] Trường hợp ngoại lệ là khi tín hiệu ngõ vào phụ thuộc vào tín hiệu ngõ ra đồng bộ và chúng ta biết độ trễ đường truyền cực đại. Một ví dụ cổ điển là giao tiếp một SRAM bất đồng bộ với một mạch đồng bộ, ví



Hình 7.1: Mạch đồng bộ ngõ vào.

cần đồng bộ hóa một tín hiệu reset bên ngoài. Tín hiệu reset phải đi qua hai flip-flop trước khi nó được sử dụng làm tín hiệu reset cho các flip-flop khác trong mạch. Việc hủy xác nhận của tín hiệu reset cần đồng bộ với xung clock.

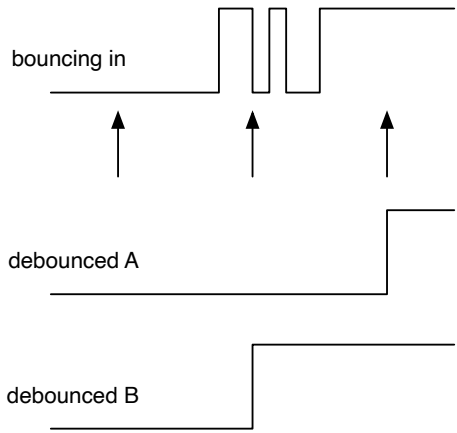
7.2 Chống dội

Các công tắc và nút nhấn có thể cần một khoảng thời gian để chuyển đổi giữa bật và tắt. Trong quá trình chuyển đổi, công tắc có thể bị dội giữa hai trạng thái đó. Nếu chúng ta sử dụng một tín hiệu như vậy mà không xử lý thêm, chúng ta có thể phát hiện nhiều sự kiện chuyển tiếp hơn chúng ta mong muốn. Một giải pháp là sử dụng thời gian để lọc bỏ sự dội này. Giả sử thời gian dội tối đa là t_{bounce}, chúng ta sẽ lấy mẫu tín hiệu đầu vào với khoảng thời gian $T > t_{bounce}$. Chúng ta sẽ chỉ sử dụng tín hiệu được lấy mẫu thêm ở luồng xuống.

Khi lấy mẫu ngõ vào với khoảng thời gian dài này, chúng ta biết rằng, khi chuyển từ 0 sang 1, chỉ một mẫu có thể rơi vào vùng dội. Mẫu trước đó sẽ đọc an toàn 0 và mẫu sau vùng dội sẽ đọc an toàn 1. Mẫu trong vùng dội sẽ là 0 hoặc 1. Tuy nhiên, điều này không quan trọng vì sau đó nó thuộc về hoặc vẫn là các mẫu 0 hoặc các mẫu 1 có sẵn. Điểm mấu chốt là chúng ta chỉ có một lần chuyển đổi từ 0 sang 1.

Hình 7.2 biểu diễn quá trình hoạt động lấy mẫu để chống dội. Tín hiệu trên cùng biểu diễn ngõ vào bị dội và các mũi tên bên dưới biểu diễn các điểm lấy mẫu. Khoảng cách giữa các điểm lấy mẫu đó cần phải dài hơn thời gian dội tối đa. Mẫu đầu tiên lấy mẫu an toàn 0, và mẫu sau trong hình lấy mẫu 1. Mẫu giữa rơi vào thời gian dội. Nó có thể là 0 hoặc 1. Hai kết quả có thể xảy ra được hiển thị là `debounce A` và `debounce B`. Cả hai

dụ: bởi bộ vi xử lý.



Hình 7.2: Chống dội tín hiệu ngõ vào.

đều có một lần chuyển đổi duy nhất từ 0 sang 1. Sự khác biệt duy nhất giữa hai kết quả này là quá trình chuyển đổi trong phiên bản B muộn hơn một chu kỳ mẫu. Tuy nhiên, đây thường không phải là vấn đề.

Mã Chisel cho việc chống dội được phát triển hơn một chút so với mã cho mạch đồng bộ. Chúng ta tạo ra thời gian mẫu bằng một mạch đếm cung cấp một tín hiệu chu kỳ đơn `tick`, như chúng ta đã thực hiện trong Phần 6.2.2.

```
val FAC = 100000000/100


val btnDebReg = Reg(Bool())

val cntReg = RegInit(0.U(32.W))
val tick = cntReg === (FAC-1).U

cntReg := cntReg + 1.U
when (tick) {
  cntReg := 0.U
  btnDebReg := btnSync
}
```

Đầu tiên, chúng ta cần quyết định tần số lấy mẫu. Ví dụ trên giả định xung clock 100 MHz và suy ra tần số lấy mẫu là 100 Hz (giả sử rằng thời gian dội là dưới 10 ms).

Giá trị mạch đếm cực đại là FAC, hệ số chia. Chúng ta định nghĩa thanh ghi btnDebReg cho tín hiệu chống dội, không có giá trị reset. Thanh ghi cntReg đóng vai trò là mạch đếm, và tín hiệu tick là đúng khi mạch đếm đạt đến giá trị lớn nhất. Trong trường hợp đó, điều kiện when là true và (1) mạch đếm được reset về 0 và (2) thanh ghi chống dội lưu trữ mẫu ngõ vào. Trong ví dụ, tín hiệu ngõ vào được đặt tên là btnSync vì nó là ngõ ra từ mạch đồng bộ ngõ vào đã được trình bày trong phần trước.

Mạch chống dội đi sau mạch đồng bộ. Đầu tiên, chúng ta cần đồng bộ hóa trong tín hiệu bất đồng bộ, sau đó chúng ta có thể xử lý thêm nó trong miền kỹ thuật số.

7.3 Lọc tín hiệu ngõ vào

Đôi khi tín hiệu ngõ vào có thể bị nhiễu, có thể nó chứa các xung nhọn mà chúng ta có thể lấy mẫu không chủ ý với mạch đồng bộ ngõ vào và bộ chống dội. Một tùy chọn để lọc các xung đột biến ngõ vào đó là sử dụng mạch biểu quyết đa số. Trong trường hợp đơn giản nhất, chúng ta lấy ba mẫu và thực hiện biểu quyết đa số. Hàm đa số, có liên quan đến hàm trung vị, cho ra giá trị của đa số. Trong trường hợp này, khi chúng ta sử dụng lấy mẫu để chống dội, chúng ta thực hiện biểu quyết đa số đối với tín hiệu được lấy mẫu. Biểu quyết đa số đảm bảo rằng tín hiệu ổn định lâu hơn chu kỳ lấy mẫu.

Hình 7.3 biểu diễn mạch biểu quyết đa số. Nó bao gồm một thanh ghi dịch 3-bit được điều khiển bởi tín hiệu tick mà chúng ta đã sử dụng để lấy mẫu chống dội. Ngõ ra của ba thanh ghi được đưa vào mạch biểu quyết đa số. Chức năng biểu quyết đa số lọc bất kỳ sự thay đổi tín hiệu nào ngắn hơn chu kỳ lấy mẫu.

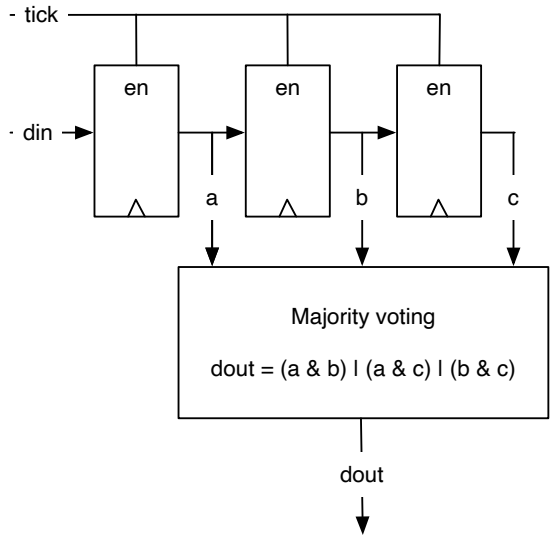
Mã Chisel sau đây trình bày thanh ghi dịch 3-bit, được điều khiển bởi tín hiệu tick và hàm biểu quyết, kết quả ở tín hiệu btnClean.

Lưu ý rằng rất ít khi cần đến biểu quyết đa số.

```
val shiftReg = RegInit(0.U(3.W))
when (tick) {
  // shift left and input in LSB
  shiftReg := Cat(shiftReg(1, 0), btnDebReg)
}
// Majority voiting
val btnClean = (shiftReg(2) & shiftReg(1)) | (shiftReg(2) &
    shiftReg(0)) | (shiftReg(1) & shiftReg(0))
```

Để sử dụng đầu ra của tín hiệu đầu vào đã được xử lý cẩn thận, trước tiên chúng ta dò cạnh lên bởi thành phần trì hoãn RegNext và sau đó so sánh tín hiệu này với giá trị hiện tại của btnClean để điều khiển mạch đếm tăng lên.

```
val risingEdge = btnClean & !RegNext(btnClean)
```



Hình 7.3: Biểu quyết đa số trên tín hiệu ngõ vào lấy mẫu.

```
// Use the rising edge of the debounced and
// filtered button to count up
val reg = RegInit(0.U(8.W))
when (risingEdge) {
  reg := reg + 1.U
}
```

7.4 Kết hợp xử lý ngõ vào với các hàm

Để tóm tắt phần xử lý đầu vào, chúng ta biểu diễn thêm một số mã Chisel. Vì các mạch điện đã được trình bày có thể là nhỏ, nhưng là các khối xây dựng có thể tái sử dụng, chúng ta gói gọn chúng trong các hàm. Phần 4.4 đã chỉ ra cách chúng ta có thể trừu tượng hóa các khối xây dựng nhỏ trong các hàm Chisel nhẹ thay vì dùng các mô-đun đầy đủ. Các hàm Chisel đó tạo ra các thực thể phần cứng, ví dụ: hàm sync tạo hai flip-flop kết nối với ngõ vào và kết nối với nhau. Hàm trả về ngõ ra của flip-flop thứ hai. Nếu hữu ích, các hàm đó có thể được nâng lên thành một số đối tượng lớp tiện ích.

```
  def sync(v: Bool) = RegNext(RegNext(v))

  def rising(v: Bool) = v & !RegNext(v)

  def tickGen(fac: Int) = {
    val reg = RegInit(0.U(log2Up(fac).W))
    val tick = reg === (fac-1).U
    reg := Mux(tick, 0.U, reg + 1.U)
    tick
  }

  def filter(v: Bool, t: Bool) = {
    val reg = RegInit(0.U(3.W))
    when (t) {
      reg := Cat(reg(1, 0), v)
    }
    (reg(2) & reg(1)) | (reg(2) & reg(0)) | (reg(1) & reg(0))
  }

  val btnSync = sync(btn)

  val tick = tickGen(fac)
  val btnDeb = Reg(Bool())
  when (tick) {
    btnDeb := btnSync
  }

  val btnClean = filter(btnDeb, tick)
  val risingEdge = rising(btnClean)

  // Use the rising edge of the debounced
  // and filtered button for the counter
  val reg = RegInit(0.U(8.W))
  when (risingEdge) {
    reg := reg + 1.U
  }
```

Listing 7.1: Tóm tắt xử lý ngõ vào với các hàm.

7.5 Bài tập

Xây dựng một mạch đếm được đếm lên bởi một nút nhấn ở ngõ vào. Hiển thị giá trị của mạch đếm ở dạng nhị phân trên các đèn LED của bo mạch FPGA. Xây dựng chuỗi xử lý đầu vào hoàn chỉnh với: (1) mạch đồng bộ hóa ngõ vào, (2) mạch chống dội, (3) mạch biểu quyết đa số để khử nhiễu, và (4) mạch dò cạnh để kích hoạt tăng mạch đếm.

Vì không có gì đảm bảo rằng nút nhấn hiện đại sẽ luôn bị dội, các bạn có thể mô phỏng độ dội và xung nhọn bằng cách nhấn nút theo cách thủ công liên tiếp nhanh và sử dụng tần số lấy mẫu thấp. Chọn, ví dụ: một giây làm tần số mẫu, tức là nếu xung clock ngõ vào hoạt động ở 100 MHz, thì hãy chia nó cho 100.000.000. Mô phỏng một nút dội bằng cách nhấn nhiều lần liên tiếp trước khi chuyển sang nhấn ổn định. Kiểm tra mạch của các bạn khi không có và khi có mạch chống dội lấy mẫu ở 1 Hz. Với biểu quyết đa số, các bạn cần nhấn từ một đến hai giây để đảm bảo mạch đếm tăng lên. Ngoài ra, việc nhả nút nhấn được tính là biểu quyết đa số. Do đó, mạch chỉ nhận ra việc nhả nút khi nó dài hơn 1-2 giây.

8 Máy trạng thái hữu hạn

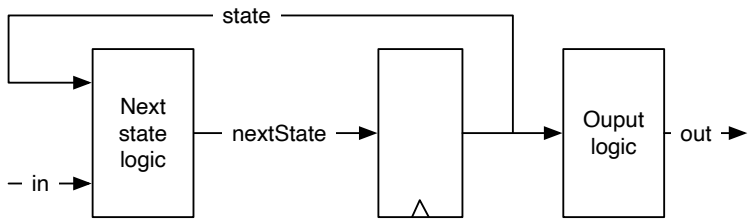
Máy trạng thái hữu hạn (Finite-State Machine - FSM) là một khối xây dựng cơ bản trong thiết kế mạch số. FSM có thể được mô tả như một tập hợp *các trạng thái* và các *chuyển tiếp trạng thái* có điều kiện (được bảo vệ) giữa các trạng thái. Một FSM có trạng thái ban đầu, trạng thái này được đặt khi reset. FSM còn được gọi là mạch tuần tự đồng bộ.

Việc triển khai FSM bao gồm ba phần: (1) thanh ghi lưu giữ trạng thái hiện tại, (2) mạch logic tổ hợp tính toán trạng thái kế tiếp phụ thuộc vào trạng thái hiện tại và ngõ vào, và (3) mạch logic tổ hợp tính toán ngõ ra của FSM.

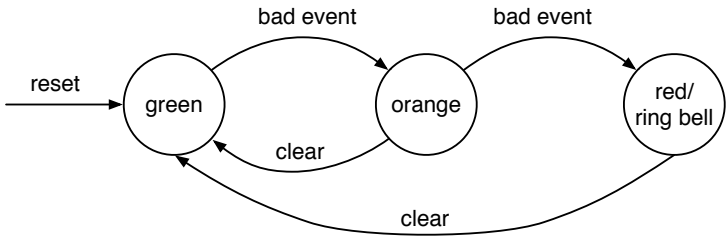
Về nguyên tắc, mọi mạch số có chứa một thanh ghi hoặc các phần tử bộ nhớ khác để lưu trữ trạng thái đều có thể được mô tả như một FSM đơn lẻ. Tuy nhiên, điều này có thể không thực tế, ví dụ: hãy thử mô tả máy tính xách tay của các bạn như một FSM đơn lẻ. Trong chương tiếp theo, chúng ta sẽ mô tả cách xây dựng các hệ thống lớn hơn từ các FSM nhỏ hơn bằng cách kết hợp chúng thành các FSM giao tiếp.

8.1 Máy trạng thái hữu hạn cơ bản

Hình 8.1 biểu diễn sơ đồ mạch điện của một FSM. Thanh ghi chứa `trạng thái` hiện tại. Mạch logic trạng thái kế tiếp sẽ tính toán giá trị của trạng thái kế tiếp (`next_state`) từ trạng thái hiện tại `state` và ngõ vào (`in`). Ở chu kỳ xung clock tiếp theo, `state` trở thành `next_state`. Mạch logic ngõ ra tính toán ngõ ra (`out`). Vì ngõ ra chỉ phụ thuộc vào trạng thái hiện tại nên máy trạng thái này được gọi là Máy trạng thái kiểu Moore.



Hình 8.1: Máy trạng thái hữu hạn (kiểu Moore).



Hình 8.2: Lưu đồ trạng thái của một FSM báo động.

Lưu đồ trạng thái mô tả hoạt động của FSM một cách trực quan. Trong lưu đồ trạng thái, các trạng thái riêng lẻ được mô tả dưới dạng các vòng tròn được gắn nhãn với tên trạng thái. Các chuyển đổi trạng thái được biểu diễn bằng các mũi tên giữa các trạng thái. Bảo vệ (hoặc điều kiện), khi chuyển đổi này được thực hiện, được vẽ như một nhãn cho mũi tên.

Hình 8.2 biểu diễn lưu đồ trạng thái của một ví dụ FSM đơn giản. FSM có ba trạng thái: *green*, *orange*, and *red*, chỉ thị mức độ báo động. FSM bắt đầu ở mức *green*. Khi có một *sự kiện xấu* xảy ra mức báo động được chuyển sang *orange*. Trong sự kiện xấu thứ hai, mức báo động được chuyển sang *red*. Trong trường hợp đó, chuông sẽ reo; *ring bell* là ngõ ra duy nhất của FSM này. Chúng ta thêm ngõ ra vào trạng thái *red*. Báo động có thể được reset bởi tín hiệu *clear*.

Mặc dù một lưu đồ trạng thái trực quan và chức năng của FSM có thể được nắm bắt nhanh chóng, nhưng một bảng trạng thái có thể được ghi xuống nhanh hơn. Bảng 8.1 hiển thị bảng trạng thái cho FSM báo động. Chúng ta liệt kê trạng thái hiện tại, các giá trị ngõ vào, kết quả trạng thái kế tiếp và giá trị ngõ ra cho trạng thái hiện tại. Về nguyên tắc, chúng ta sẽ cần xác định tất cả các ngõ vào có thể có cho tất cả các trạng thái có thể. Bảng này sẽ có $3 \times 4 = 12$ hàng. Chúng ta đơn giản hóa bảng bằng cách chỉ ra rằng ngõ vào *clear* là tín hiệu don't care (không quan tâm) khi *sự kiện xấu* xảy ra. Điều đó có nghĩa là *sự kiện xấu* được ưu tiên hơn tín hiệu *clear*. Cột ngõ ra có một số lần lặp lại. Nếu chúng ta có FSM lớn hơn và/hoặc nhiều ngõ ra hơn, thì chúng ta có thể chia bảng thành hai, một cho logic trạng thái kế tiếp theo và một cho logic ngõ ra.

Cuối cùng, sau khi tất cả thiết kế của FSM mức báo động, chúng ta sẽ lập trình nó trong Chisel. Listing 8.1 trình bày mã Chisel cho FSM báo động. Lưu ý kiểu `Bool` trong Chisel cho các ngõ vào và ngõ ra của FSM. Để dùng `Enum` và lệnh điều khiển `switch`, chúng ta cần nhập `chisel3.util._`.

Mã Chisel hoàn chỉnh cho FSM đơn giản này nằm gọn trong một trang. Chúng ta hãy lướt qua các phần riêng lẻ. FSM có hai tín hiệu ngõ vào và một tín hiệu ngõ ra, được ghi

```
import chisel3._
import chisel3.util._

class SimpleFsm extends Module {
  val io = IO(new Bundle{
    val badEvent = Input(Bool())
    val clear = Input(Bool())
    val ringBell = Output(Bool())
  })

  // The three states
  val green :: orange :: red :: Nil = Enum(3)

  // The state register
  val stateReg = RegInit(green)

  // Next state logic
  switch (stateReg) {
    is (green) {
      when(io.badEvent) {
        stateReg := orange
      }
    }
    is (orange) {
      when(io.badEvent) {
        stateReg := red
      } .elsewhen(io.clear) {
        stateReg := green
      }
    }
    is (red) {
      when (io.clear) {
        stateReg := green
      }
    }
  }

  // Output logic
  io.ringBell := stateReg === red
}
```

Listing 8.1: Mã Chisel cho FSM báo động.

Bảng 8.1: Bảng trạng thái của FSM báo động.

	Ngõ vào			
Trạng thái	Sự kiện xấu	Xóa	Trạng thái kế tiếp	Chuông reo
green	0	0	green	0
green	1	-	orange	0
orange	0	0	orange	0
orange	1	-	red	0
orange	0	1	green	0
red	0	0	red	1
red	0	1	green	1

trong một `Bundle` Chisel:

```
val io = IO(new Bundle{
  val badEvent = Input(Bool())
  val clear = Input(Bool())
  val ringBell = Output(Bool())
})
```

Khá nhiều công việc đã được dành để mã hóa trạng thái tối ưu. Hai tùy chọn phổ biến là mã hóa nhị phân hoặc mã hóa one-hot. Tuy nhiên, chúng ta để những quyết định mức-thấp đó cho công cụ tổng hợp và hướng tới mã code có thể đọc được.[1] Do đó, chúng ta sử dụng kiểu liệt kê với tên tượng trưng cho các trạng thái:

```
val green :: orange :: red :: Nil = Enum(3)
```

Các giá trị trạng thái riêng lẻ như trong danh sách, trong đó các phần tử riêng lẻ được ghép nối bởi toán tử `::`; `Nil` biểu diễn phần kết thúc của danh sách. Một thực thể `Enum` được *gán* cho danh sách các trạng thái. Thanh ghi giữ trạng thái được định nghĩa với trạng thái *green* như giá trị reset:

```
val stateReg = RegInit(green)
```

Phần quan trọng của FSM là ở logic trạng thái tiếp theo. Chúng ta sử dụng công tắc Chisel trên thanh ghi trạng thái để bao gồm tất cả các trạng thái. Trong mỗi trạng thái

[1] Trong phiên bản hiện tại của Chisel, kiểu `Enum` biểu diễn các trạng thái trong mã hóa nhị phân. Nếu chúng ta muốn một dạng mã hóa khác, ví dụ: mã hóa one-hot, chúng ta có thể định nghĩa hằng số Chisel cho tên trạng thái.

là nhánh mà chúng ta sẽ lập trình cho logic trạng thái tiếp theo, tùy thuộc vào các ngõ vào, bằng cách gán một giá trị mới cho thanh ghi trạng thái:

```
switch (stateReg) {
  is (green) {
    when(io.badEvent) {
      stateReg := orange
    }
  }
  is (orange) {
    when(io.badEvent) {
      stateReg := red
    } .elsewhen(io.clear) {
      stateReg := green
    }
  }
  is (red) {
    when (io.clear) {
      stateReg := green
    }
  }
}
```

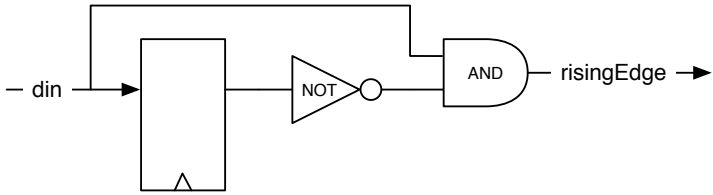
Cuối cùng, nhưng không phải sau cùng, chúng ta lập trình ngõ ra *ring bell* là giá trị đúng (true) khi trạng thái là *red*.

```
io.ringBell := stateReg === red
```

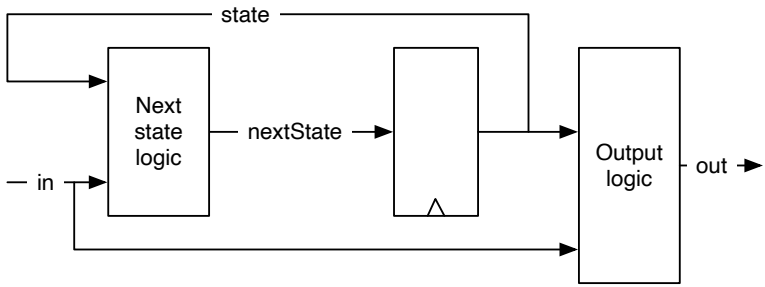
Lưu ý rằng chúng ta đã *không* đưa vào tín hiệu `next_state` cho ngõ vào thanh ghi, vì đây là bài thực hành phổ biến trong Verilog hoặc VHDL. Các thanh ghi trong Verilog và VHDL được mô tả theo một cú pháp đặc biệt và không thể được gán (và không được gán lại) trong một khối mạch tổ hợp. Do đó, tín hiệu bổ sung, được tính toán trong một khối mạch tổ hợp, được đưa vào và kết nối với ngõ vào thanh ghi. Trong Chisel, một thanh ghi là một kiểu cơ sở và có thể được sử dụng tự do trong một khối mạch tổ hợp.

8.2 Ngõ ra nhanh hơn với FSM Mealy

Với FSM Moore, ngõ ra chỉ phụ thuộc vào trạng thái hiện tại. Điều đó có nghĩa là sự thay đổi của ngõ vào có thể được xem như là thay đổi của ngõ ra *sớm nhất* trong chu kỳ xung clock tiếp theo. Nếu chúng ta muốn quan sát sự thay đổi tức thời, chúng ta cần một đường dẫn tổ hợp từ ngõ vào đến ngõ ra. Chúng ta hãy cùng xem xét một ví dụ tối thiểu,



Hình 8.3: Mạch dò cạnh lên (FSM kiểu Mealy).



Hình 8.4: Máy trạng thái hữu hạn kiểu Mealy.

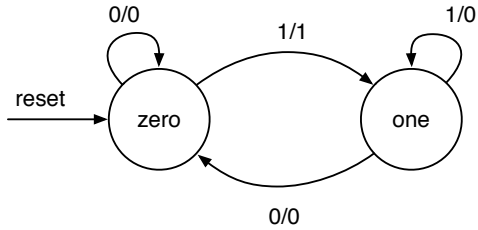
một mạch dò cạnh. Chúng ta đã thấy lệnh Chisel theo kiểu một dòng duy nhất này trước đây:

```
val risingEdge = din & !RegNext(din)
```

Hình 8.3 biểu diễn sơ đồ mạch điện của mạch dò cạnh lên. Ngõ ra trở thành 1 trong một chu kỳ xung clock khi giá trị ngõ vào hiện tại là 1 và ngõ vào trong chu kỳ xung clock cuối cùng là 0. Thanh ghi trạng thái chỉ là một flip-flop D duy nhất trong đó trạng thái tiếp theo chỉ là ngõ vào. Chúng ta cũng có thể coi đây là phần tử trễ của một chu kỳ xong clock. Logic ngõ ra *so sánh* ngõ vào hiện tại với trạng thái hiện tại.

Khi ngõ ra cũng phụ thuộc vào ngõ vào, tức là, có một đường tổ hợp giữa ngõ vào của FSM và ngõ ra, điều này được gọi là máy Mealy.

Hình 8.4 biểu diễn sơ đồ mạch của FSM kiểu Mealy. Tương tự như FSM kiểu Moore, thanh ghi chứa `trạng thái` hiện tại và logic trạng thái tiếp theo tính toán giá trị trạng thái tiếp theo (`next_state`) từ `trạng thái` hiện tại và ngõ vào (`in`). Vào chu kỳ xung clock tiếp theo, `state` trở thành `next_state`. Logic ngõ ra tính toán ngõ ra (`out`) từ trạng thái hiện tại *và* ngõ vào cho FSM.



Hình 8.5: Lưu đồ trạng thái của mạch dò cạnh lên trong FSM kiểu Mealy.

Hình 8.5 biểu diễn lưu đồ trạng thái của FSM Mealy cho mạch dò cạnh. Vì thanh ghi trạng thái chỉ bao gồm một flip-flop D duy nhất, nên chỉ có thể có hai trạng thái đặt tên là `zero` và `one` trong ví dụ này. Vì ngõ ra của FSM Mealy không chỉ phụ thuộc vào trạng thái mà còn phụ thuộc vào ngõ vào, nên chúng ta không thể mô tả ngõ ra như một phần của vòng tròn trạng thái. Thay vào đó, các chuyển đổi giữa các trạng thái được gắn nhãn với giá trị ngõ vào (điều kiện) *và* ngõ ra (sau dấu gạch chéo). Cũng lưu ý rằng, chúng ta vẽ các trạng thái tự chuyển đổi, ví dụ: ở trạng thái `zero` khi ngõ vào là `0`, FSM vẫn ở trạng thái `zero` và ngõ ra là 0. Ở cạnh lên, FSM chỉ tạo ra ngõ ra `1` khi chuyển đổi từ trạng thái `zero` sang trạng thái `one`. Trong trạng thái `one`, biểu diễn rằng ngõ vào bây giờ là `1`, ngõ ra là `0`. Chúng ta chỉ muốn một xung (chu kỳ) duy nhất cho mỗi cạnh lên của ngõ vào.

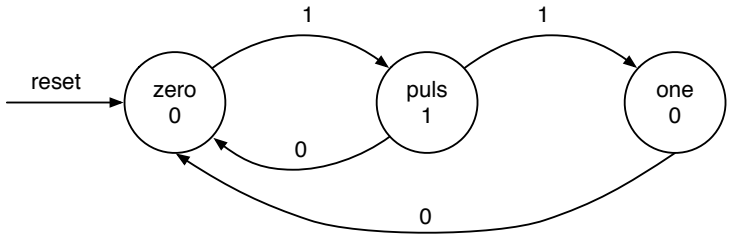
Listing 8.2 trình bày mã Chisel cho dò cạnh lên với máy Mealy. Như ở ví dụ trước, chúng ta sử dụng kiểu `Bool` trong Chisel cho ngõ vào và ngõ ra đơn bit. Logic ngõ ra bây giờ là một phần của logic trạng thái tiếp theo; trên chuyển tiếp từ `zero` sang `one`, ngõ ra được đặt là `true.B`. Nếu không, giá trị gán mặc định cho đếm ngõ ra là (`false.B`).

Người ta có thể hỏi liệu FSM toàn diện có phải là giải pháp tốt nhất cho mạch dò cạnh hay không, đặc biệt như chúng ta đã thấy cú pháp một dòng lệnh duy nhất trong Chisel cho cùng một chức năng. Việc tiêu thụ phần cứng cũng tương tự. Cả hai giải pháp đều cần một flip-flop D duy nhất cho trạng thái. Mạch logic tổ hợp cho FSM có lẽ phức tạp hơn một chút, do sự thay đổi trạng thái phụ thuộc vào trạng thái hiện tại và giá trị ngõ vào. Với chức năng này, điều quan trọng là kiểu viết một dòng lệnh duy nhất dễ viết và dễ đọc hơn. Do đó, kiểu viết một dòng lệnh duy nhất là giải pháp được ưa thích hơn.

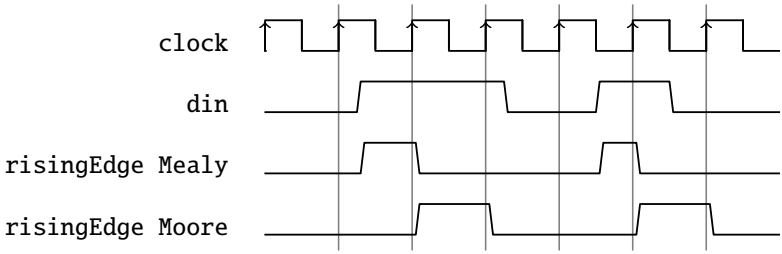
Chúng ta đã sử dụng ví dụ này để biểu diễn một trong những FSM Mealy nhỏ nhất có thể. FSM sẽ được sử dụng cho các mạch điện phức tạp hơn với ba trạng thái trở lên.

```
import chisel3._
import chisel3.util._

class RisingFsm extends Module {
  val io = IO(new Bundle{
    val din = Input(Bool())
    val risingEdge = Output(Bool())
  })

  // The two states
  val zero :: one :: Nil = Enum(2)

  // The state register
  val stateReg = RegInit(zero)

  // default value for output
  io.risingEdge := false.B

  // Next state and output logic
  switch (stateReg) {
    is(zero) {
      when(io.din) {
        stateReg := one
        io.risingEdge := true.B
      }
    }
    is(one) {
      when(!io.din) {
        stateReg := zero
      }
    }
  }
}
```

Listing 8.2: Dò cạnh lên với FSM Mealy.



Hình 8.6: Lưu đồ trạng thái mạch dò cạnh lên của FSM Moore.



Hình 8.7: Dạng sóng FSM Moore và Mealy cho dò cạnh lên.

8.3 So sánh Moore với Mealy

Để chỉ ra sự khác biệt giữa FSM Moore và Mealy, chúng ta thực hiện lại việc dò cạnh bằng FSM Moore.

Hình 8.6 biểu diễn lưu đồ trạng thái để dò cạnh lên với FSM Moore. Điều đầu tiên cần chú ý là FSM Moore cần ba trạng thái, so với hai trạng thái trong FSM Mealy. Trạng thái `puls` là cần thiết để tạo ra các xung chu kỳ đơn. FSM trong trạng thái `puls` chỉ cần một chu kỳ xung clock và sau đó tiếp tục trở lại trạng thái bắt đầu `zero` hoặc thành trạng thái `one`, chờ ngõ vào trở lại 0 lần nữa. Chúng ta biểu diễn điều kiện ngõ vào ở trên các mũi tên chuyển đổi trạng thái và ngõ ra FSM trong các vòng tròn biểu diễn trạng thái.

Listing 8.3 trình bày phiên bản Moore của mạch dò cạnh lên. Nó sử dụng gấp đôi số flip-flop D so với phiên bản Mealy hoặc được lập trình trực tiếp. Do đó, kết quả logic trạng thái tiếp theo cũng lớn hơn phiên bản Mealy hoặc được lập trình trực tiếp.

Hình 8.7 biểu diễn dạng sóng của phiên bản Mealy và Moore của FSM dò cạnh lên. Chúng ta có thể thấy rằng ngõ ra Mealy theo sát cạnh lên ngõ vào, trong khi ngõ ra Moore tăng sau tick xung clock. Chúng ta cũng có thể thấy rằng ngõ ra Moore có độ rộng một chu kỳ xung clock, trong khi đó ngõ ra Mealy thường nhỏ hơn một chu kỳ

```
import chisel3._
import chisel3.util._

class RisingMooreFsm extends Module {
  val io = IO(new Bundle{
    val din = Input(Bool())
    val risingEdge = Output(Bool())
  })

  // The three states
  val zero :: puls :: one :: Nil = Enum(3)

  // The state register
  val stateReg = RegInit(zero)

  // Next state logic
  switch (stateReg) {
    is(zero) {
      when(io.din) {
        stateReg := puls
      }
    }
    is(puls) {
      when(io.din) {
        stateReg := one
      } .otherwise {
        stateReg := zero
      }
    }
    is(one) {
      when(!io.din) {
        stateReg := zero
      }
    }
  }

  // Output logic
  io.risingEdge := stateReg === puls
}
```

Listing 8.3: Dò cạnh lên với FSM Moore.

xung clock.

Từ ví dụ trên, người ta muốn tìm FSM Mealy. FSM *tốt hơn* vì chúng cần ít trạng thái hơn (và đó là logic) và đáp ứng nhanh hơn FSM Moore. Tuy nhiên, đường tổ hợp bên trong máy Mealy có thể gây ra rắc rối trong các thiết kế lớn hơn. Đầu tiên, với một chuỗi FSM giao tiếp (xem chương tiếp theo), đường tổ hợp này có thể trở nên dài. Thứ hai, nếu các FSM giao tiếp xây dựng một vòng tròn, kết quả là một vòng lặp tổ hợp, đây là một lỗi trong thiết kế đồng bộ. Do sự đứt đoạn trong đường tổ hợp với thanh ghi trạng thái trong FSM Moore, nên tất cả các vấn đề trên không tồn tại đối với FSM Moore giao tiếp.

Tóm lại, các FSM Moore kết hợp với nhau tốt hơn để giao tiếp các máy trạng thái; chúng *mạnh mẽ hơn* so với FSM Mealy. Chỉ sử dụng các FSM Mealy khi yếu tố đáp ứng trong cùng một chu kỳ được xem là quan trọng nhất. Các mạch nhỏ như dò cạnh lên, thực tế là máy Mealy, cũng tốt.

8.4 Bài tập

Trong chương này, các bạn đã thấy nhiều ví dụ về các FSM rất nhỏ. Bây giờ đã đến lúc viết một số mã FSM *thực sự*. Chọn một ví dụ phức tạp hơn một chút, thực hiện FSM và viết testbench cho nó.

Một ví dụ cổ điển cho FSM là mạch điều khiển đèn giao thông (xem [3, Phần 14.3]). Mạch điều khiển đèn giao thông phải đảm bảo rằng quá trình chuyển đổi từ màu đỏ sang màu xanh lá cây có một pha ở giữa hai con đường trong giao lộ đều có đèn cấm đi (đỏ và cam). Để làm cho ví dụ này thú vị hơn một chút, hãy xem xét một đường ưu tiên. Đường phụ có hai mạch dò xe (trên cả hai lối vào giao lộ). Chỉ chuyển sang màu xanh cho đường phụ khi phát hiện có xe và sau đó chuyển về màu xanh cho đường ưu tiên.

9 Máy trạng thái giao tiếp

Một vấn đề thường quá phức tạp để mô tả nó bằng một FSM duy nhất. Trong trường hợp đó, vấn đề có thể được chia thành hai hoặc nhiều FSM nhỏ hơn và đơn giản hơn. Các FSM đó sau đó giao tiếp với các tín hiệu. Một ngõ ra FSM này là một ngõ vào FSM khác và FSM theo dõi ngõ ra của FSM khác. Khi chúng ta chia một FSM lớn thành những FSM đơn giản hơn, điều này được gọi là các FSM phân số. Tuy nhiên, thường các FSM giao tiếp được thiết kế trực tiếp từ đặc tả kỹ thuật, vì thường một FSM đơn lẻ sẽ không khả thi.

9.1 Ví dụ mạch chớp đèn

Để thảo luận các FSM giao tiếp, chúng ta sử dụng ví dụ từ [3, Chương 17], mạch chớp đèn. Mạch chớp đèn có một ngõ vào `start` và một ngõ ra `light`. Đặc tả của mạch chớp đèn như sau:

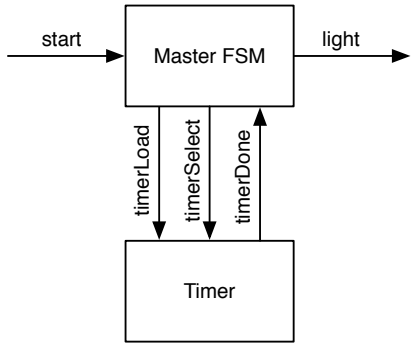
- khi tín hiệu `start` lên cao trong một chu kỳ xung clock, quy trình chớp nháy bắt đầu;
- quy trình nhấp nháy ba lần;
- tín hiệu `light` sẽ *bật* trong 6 chu kỳ xung clock, và `light` sẽ *tắt* trong 4 chu kỳ xung clock giữa những lần chớp;
- sau quy trình, FSM chuyển tín hiệu `light` *tắt* và chờ đợi lần bắt đầu kế tiếp.

Thực hiện trực tiếp FSM[1] có 27 trạng thái: trạng thái đầu tiên chờ giá trị ngõ vào, 3×6 trạng thái cho ba trạng thái *on* và 2×4 trạng thái cho các trạng thái *off*. Mã code cho việc thực hiện mạch đèn chớp sáng đơn giản này không được trình bày.

Vấn đề có thể được giải quyết một cách thanh thoát hơn bằng cách phân tách FSM lớn này thành hai FSM nhỏ hơn: FSM chủ (Master FSM) thực hiện logic nhấp nháy và FSM định thời (Timer FSM) thực hiện việc chờ. Hình 9.1 biểu diễn thành phần của hai FSM.

FSM định thời đếm xuống trong 6 hoặc 4 chu kỳ xung clock để tạo ra thời gian mong muốn. Đặc tả kỹ thuật bộ định thời như sau:

[1] Lưu đồ trạng thái được biểu diễn trong [3, p. 376].



Hình 9.1: Mạch chớp đèn tách thành Master FSM và Timer FSM.

- khi `timerLoad` được xác nhận, bộ định thời nạp giá trị vào mạch đếm xuống;
- `timerSelect` chọn giữa 5 hoặc 3 để nạp;
- `timerDone` được xác nhận khi mạch đếm hoàn thành đếm xuống và vẫn duy trì xác nhận;
- nếu không, bộ định thời đếm xuống.

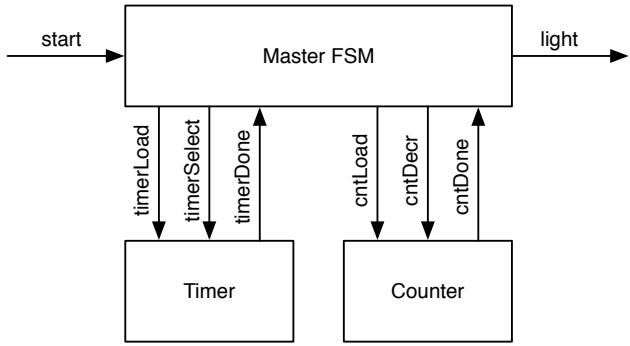
Đoạn mã sau trình bày FSM định thời của mạch chớp đèn:

```
val timerReg = RegInit(0.U)
timerDone := timerReg === 0.U

// Timer FSM (down counter)
when(!timerDone) {
  timerReg := timerReg - 1.U
}
when (timerLoad) {
  when (timerSelect) {
    timerReg := 5.U
  } .otherwise {
    timerReg := 3.U
  }
}
```

Listing 9.1 trình bày Master FSM.

```
val off :: flash1 :: space1 :: flash2 :: space2 :: flash3 ::
    Nil = Enum(6)
val stateReg = RegInit(off)

val light = WireDefault(false.B) // FSM output

// Timer connection
val timerLoad = WireDefault(false.B) // start timer with a
    load
val timerSelect = WireDefault(true.B) // select 6 or 4 cycles
val timerDone = Wire(Bool())

timerLoad := timerDone

// Master FSM
switch(stateReg) {
  is(off) {
    timerLoad := true.B
    timerSelect := true.B
    when (start) { stateReg := flash1 }
  }
  is (flash1) {
    timerSelect := false.B
    light := true.B
    when (timerDone) { stateReg := space1 }
  }
  is (space1) {
    when (timerDone) { stateReg := flash2 }
  }
  is (flash2) {
    timerSelect := false.B
    light := true.B
    when (timerDone) { stateReg := space2 }
  }
  is (space2) {
    when (timerDone) { stateReg := flash3 }
  }
  is (flash3) {
    timerSelect := false.B
    light := true.B
    when (timerDone) { stateReg := off }
  }
}
```

Listing 9.1: Master FSM mạch chớp đèn.



Hình 9.2: Mạch chớp đèn tách thành Master FSM, Timer FSM, và Counter FSM.

Giải pháp này với FSM chủ và một bộ định thời vẫn có sự dư thừa trong mã của FSM chủ. Các trạng thái `flash1`, `flash2` và `flash3` đều đang thực hiện cùng một chức năng, các trạng thái `space1` và `space2` cũng vậy. Chúng ta có thể phân số lần chớp nháy còn lại vào mạch đếm thứ hai. Sau đó, FSM chủ được giảm xuống còn ba trạng thái: `off`, `flash` và `space`.

Hình 9.2 biểu diễn thiết kế với Master FSM và hai FSM để đếm: một FSM đếm chu kỳ xung clock trong khoảng thời gian *on* và *off*; FSM thứ hai để đếm số lần chớp nháy còn lại.

Đoạn mã sau trình bày FSM mạch đếm xuống:

```
val cntReg = RegInit(0.U)
cntDone := cntReg === 0.U

// Down counter FSM
when(cntLoad) { cntReg := 2.U }
when(cntDecr) { cntReg := cntReg - 1.U }
```

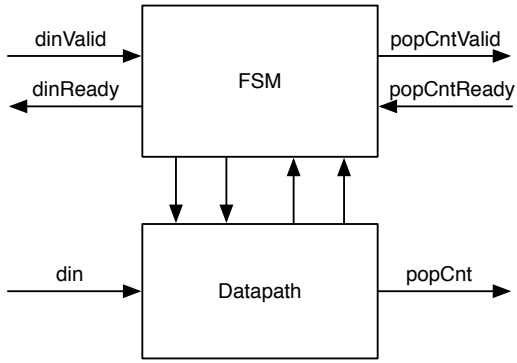
Lưu ý rằng bộ đếm được nạp với giá trị 2 cho 3 lần chớp nháy, vì nó đếm số lần nhấp nháy *còn lại* và được giảm dần trong *không gian* trạng thái khi bộ định thời được thực hiện. Listing 9.2 hiển thị FSM chủ cho mạch chớp nháy tái cấu trúc kép.

Bên cạnh việc FSM chủ được giảm xuống chỉ còn ba trạng thái, giải pháp hiện tại của chúng ta cũng có thể cấu hình tốt hơn. Không cần thay đổi FSM nếu chúng ta muốn thay đổi độ dài của khoảng thời gian *on* hoặc *off* hoặc số lần chớp nháy.

Trong phần này, chúng ta đã tìm hiểu các mạch giao tiếp, đặc biệt là FSM, chỉ trao đổi tín hiệu điều khiển. Tuy nhiên, các mạch còn có thể trao đổi dữ liệu. Để trao đổi dữ

```
val off :: flash :: space :: Nil = Enum(3)
val stateReg = RegInit(off)

val light = WireDefault(false.B) // FSM output

// Timer connection
val timerLoad = WireDefault(false.B) // start timer with a
    load
val timerSelect = WireDefault(true.B) // select 6 or 4 cycles
val timerDone = Wire(Bool())
// Counter connection
val cntLoad = WireDefault(false.B)
val cntDecr = WireDefault(false.B)
val cntDone = Wire(Bool())

timerLoad := timerDone

switch(stateReg) {
  is(off) {
    timerLoad := true.B
    timerSelect := true.B
    cntLoad := true.B
    when (start) { stateReg := flash }
  }
  is (flash) {
    timerSelect := false.B
    light := true.B
    when (timerDone & !cntDone) { stateReg := space }
    when (timerDone & cntDone) { stateReg := off }
  }
  is (space) {
    cntDecr := timerDone
    when (timerDone) { stateReg := flash }
  }
}
```

Listing 9.2: Master FSM của mạch chớp đèn tái cấu trúc kép.



Hình 9.3: Máy trạng thái với đường dữ liệu.

liệu phối hợp, chúng ta sử dụng tín hiệu bắt tay. Phần tiếp theo mô tả giao tiếp sẵn-sàng-hợp-lệ để kiểm soát luồng trao đổi dữ liệu một chiều.

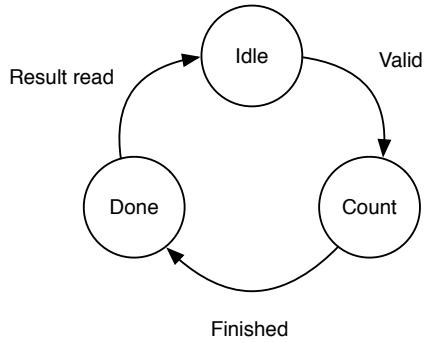
9.2 Máy trạng thái với đường dữ liệu

Một ví dụ tiêu biểu về máy trạng thái giao tiếp là máy trạng thái kết hợp với một đường dữ liệu. Sự kết hợp này thường được gọi là máy trạng thái hữu hạn với đường dữ liệu (Finite State Machine with Datapath - FSMD). Máy trạng thái kiểm soát đường dữ liệu và đường dữ liệu thực hiện tính toán. Ngõ vào FSM là ngõ vào từ môi trường và ngõ vào từ đường dữ liệu. Dữ liệu từ môi trường được đưa vào đường dữ liệu và ngõ ra dữ liệu đến từ đường dữ liệu. Hình 9.3 biểu diễn một ví dụ về sự kết hợp của FSM với đường dữ liệu.

9.2.1 Ví dụ về Popcount

FSMD được biểu diễn trong Hình 9.3 đóng vai trò là một ví dụ tính toán Popcount, còn được gọi là trọng số Hamming. Trọng số Hamming là số các ký hiệu khác với ký hiệu số 0. Đối với một chuỗi nhị phân, đây là số lần của số '1'.

Đơn vị Popcount chứa ngõ vào dữ liệu `din` và ngõ ra kết quả `popCount`, cả hai đều được kết nối với đường dữ liệu. Đối với ngõ vào và ngõ ra, chúng ta sử dụng một tín hiệu bắt tay sẵn-sàng-hợp-lệ. Khi dữ liệu có sẵn, xác nhận hợp lệ. Khi bộ thu có thể chấp nhận dữ liệu, nó xác nhận rằng nó đã sẵn sàng. Khi cả hai tín hiệu được xác nhận, quá trình chuyển dữ liệu sẽ diễn ra. Các tín hiệu bắt tay được kết nối với FSM. FSM được



Hình 9.4: Lưu đồ trạng thái cho FSM Popcount.

kết nối với đường dữ liệu với các tín hiệu điều khiển hướng tới đường dữ liệu và với các tín hiệu trạng thái từ đường dữ liệu.

Bước tiếp theo, chúng ta có thể thiết kế FSM, bắt đầu với lưu đồ trạng thái, được biểu diễn trong Hình 9.4. Chúng ta bắt đầu ở trạng thái `Idle`, nơi FSM chờ ở ngõ vào. Khi dữ liệu đến, được báo hiệu bằng tín hiệu hợp lệ, FSM sẽ chuyển sang trạng thái `Load` để nạp thanh ghi dịch. FSM chuyển sang trạng thái tiếp theo `Count`, số lượng của số '1' được đếm tuần tự. Chúng ta sử dụng một thanh ghi dịch, một mạch cộng, một thanh ghi tích lũy và một mạch đếm xuống để thực hiện tính toán. Khi mạch đếm xuống bằng 0, quá trình hoàn tất và FSM chuyển sang trạng thái `Done`. Tại đó FSM báo hiệu bằng một tín hiệu hợp lệ rằng giá trị Popcount đã sẵn sàng được sử dụng. Khi có tín hiệu sẵn sàng từ bộ thu, FSM sẽ chuyển trở lại trạng thái `Idle`, sẵn sàng tính toán số lượng Popcount tiếp theo.

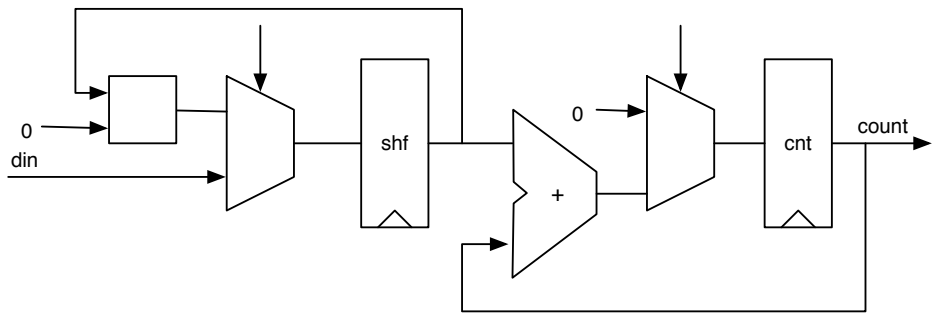
Thành phần mức cao nhất (top-level), được hiển thị trong Listing 9.3, khởi tạo FSM và các thành phần của đường dữ liệu và kết nối chúng bởi các kết nối khối.

Hình 9.5 biểu diễn đường dẫn dữ liệu cho mạch Popcount. Dữ liệu được nạp vào thanh ghi `shf`. Khi nạp, thanh ghi `cnt` cũng được reset về 0. Để đếm số lượng số '1', thanh ghi `shf` được dịch sang phải và bit LSB được thêm vào `cnt` mỗi xung clock. Mạch đếm, không được hiển thị trong hình, đếm xuống cho đến khi tất cả các bit được dịch chuyển qua bit LSB. Khi mạch đếm về 0, số lượng Popcount đã hoàn tất. FSM chuyển sang trạng thái `Done` và báo hiệu kết quả bằng tín hiệu xác nhận `popCntReady`. Khi kết quả được đọc, nó được báo hiệu bằng tín hiệu xác nhận `popCntValid`, FSM sẽ chuyển về trạng thái `Idle`.

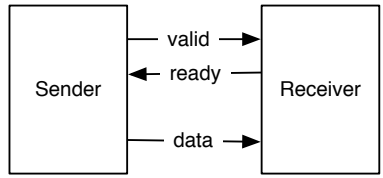
Trên tín hiệu `load`, thanh ghi `regData` được nạp với giá trị ngõ vào, thanh ghi `regPopCount` được reset về 0 và giá trị thanh ghi mạch đếm `regCount` được đặt thành số lần dịch đã

```
class PopCount extends Module {
  val io = IO(new Bundle {
    val dinValid = Input(Bool())
    val dinReady = Output(Bool())
    val din = Input(UInt(8.W))
    val popCntValid = Output(Bool())
    val popCntReady = Input(Bool())
    val popCnt = Output(UInt(4.W))
  })


  val fsm = Module(new PopCountFSM)
  val data = Module(new PopCountDataPath)

  fsm.io.dinValid := io.dinValid
  io.dinReady := fsm.io.dinReady
  io.popCntValid := fsm.io.popCntValid
  fsm.io.popCntReady := io.popCntReady

  data.io.din := io.din
  io.popCnt := data.io.popCnt
  data.io.load := fsm.io.load
  fsm.io.done := data.io.done
}
```

Listing 9.3: Mức top-level của mạch Popcount.



Hình 9.5: Đường dữ liệu mạch Popcount.



Hình 9.6: Điều khiển luồng sẵn-sàng-hợp-lệ.

được thực hiện.

Mặt khác, thanh ghi `regData` được dịch sang phải, bit LSB của thanh ghi `regData` sẽ được thêm vào thanh ghi `regPopCount` và mạch đếm giảm dần cho đến khi nó bằng 0. Khi mạch đếm bằng 0, ngõ ra chứa giá trị Popcount. Listing 9.4 trình bày mã Chisel cho đường dữ liệu của mạch Popcount.

FSM bắt đầu ở trạng thái `idle`. Với tín hiệu hợp lệ cho dữ liệu ngõ vào (`dinValid`), nó sẽ chuyển sang trạng thái `count` và đợi đến khi đường dữ liệu hoàn tất việc đếm. Khi Popcount hợp lệ, FSM sẽ chuyển sang trạng thái `done` và đợi đến khi giá trị Popcount được đọc (được báo hiệu bằng tín hiệu `popCntReady`). Listing 9.5 trình bày mã code của FSM.

9.3 Giao tiếp sẵn-sàng-hợp-lệ

Giao tiếp của các hệ thống con có thể được khái quát hóa thành chuyển động của dữ liệu và bắt tay để điều khiển luồng. Trong ví dụ về Popcount, chúng ta đã thấy một giao diện bắt tay cho dữ liệu ngõ vào và ngõ ra sử dụng các tín hiệu sẵn sàng và hợp lệ.

Giao tiếp sẵn-sàng-hợp-lệ [3, p. 480] là một giao tiếp điều khiển luồng đơn giản bao gồm tín hiệu `data` và `valid` ở phía người gửi (nhà sản xuất) và tín hiệu `ready` ở phía người nhận (người tiêu dùng). Hình 9.6 cho thấy kết nối sẵn-sàng-hợp-lệ. Người gửi xác nhận tín hiệu `valid` khi `data` khả dụng và người nhận xác nhận tín hiệu `ready` khi sẵn sàng nhận một word dữ liệu. Việc truyền dữ liệu xảy ra khi cả hai tín hiệu, `valid` và `ready`, được xác nhận. Nếu một trong hai tín hiệu không được xác nhận, sẽ không có quá trình chuyển dữ liệu nào diễn ra.

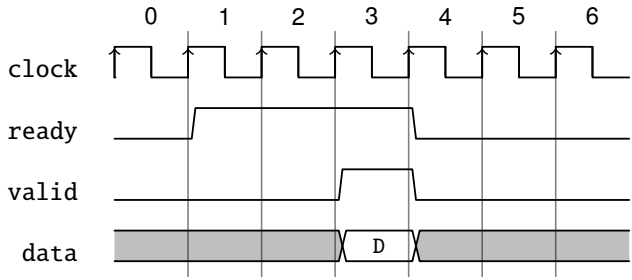
Hình 9.7 biểu diễn sơ đồ thời gian của giao dịch sẵn-sàng-hợp-lệ trong đó bên nhận báo hiệu `ready` (từ chu kỳ xung clock 1 trở đi) trước khi bên gửi có dữ liệu. Việc truyền dữ liệu xảy ra trong chu kỳ xung clock 3. Từ chu kỳ xung clock 4 trở đi, cả bên gửi không có dữ liệu và bên nhận không sẵn sàng cho lần truyền tiếp theo. Khi bên nhận có thể nhận dữ liệu trong mọi chu kỳ xung clock, nó được gọi là giao diện "luôn sẵn sàng" và tín hiệu `ready` có thể được lập trình cố định thành `true`.

```
class PopCountDataPath extends Module {
  val io = IO(new Bundle {
    val din = Input(UInt(8.W))
    val load = Input(Bool())
    val popCnt = Output(UInt(4.W))
    val done = Output(Bool())
  })

  val dataReg = RegInit(0.U(8.W))
  val popCntReg = RegInit(0.U(8.W))
  val counterReg= RegInit(0.U(4.W))

  dataReg := 0.U ## dataReg(7, 1)
  popCntReg := popCntReg + dataReg(0)

  val done = counterReg === 0.U
  when (!done) {
    counterReg := counterReg - 1.U
  }

  when(io.load) {
    dataReg := io.din
    popCntReg := 0.U
    counterReg := 8.U
  }

  // debug output
  printf("%x %d\n", dataReg, popCntReg)

  io.popCnt := popCntReg
  io.done := done
}
```

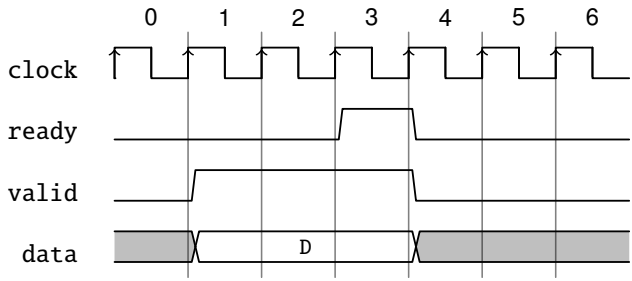
Listing 9.4: Đường dữ liệu của mạch Popcount.

```
class PopCountFSM extends Module {
  val io = IO(new Bundle {
    val dinValid = Input(Bool())
    val dinReady = Output(Bool())
    val popCntValid = Output(Bool())
    val popCntReady = Input(Bool())
    val load = Output(Bool())
    val done = Input(Bool())
  })

  val idle :: count :: done :: Nil = Enum(3)
  val stateReg = RegInit(idle)

  io.load := false.B

  io.dinReady := false.B
  io.popCntValid := false.B

  switch(stateReg) {
    is(idle) {
      io.dinReady := true.B
      when(io.dinValid) {
        io.load := true.B
        stateReg := count
      }
    }
    is(count) {
      when(io.done) {
        stateReg := done
      }
    }
    is(done) {
      io.popCntValid := true.B
      when(io.popCntReady) {
        stateReg := idle
      }
    }
  }
}
```

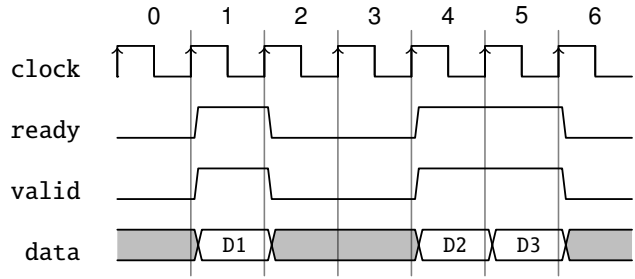
Listing 9.5: FSM của mạch Popcount.



Hình 9.7: Truyền dữ liệu với giao tiếp sẵn-sàng-hợp-lệ, sẵn sàng sớm.



Hình 9.8: Truyền dữ liệu với giao tiếp sẵn-sàng-hợp-lệ, sẵn sàng trễ.



Hình 9.9: Chu kỳ đơn sẵn sàng/hợp lệ và truyền liên tục (back-to-back).

Hình 9.8 biểu diễn sơ đồ thời gian của giao dịch sẵn sàng hợp lệ trong đó bên gửi báo hiệu `valid` (từ chu kỳ xung clock 1 trở đi) trước khi bên nhận sẵn sàng. Việc truyền dữ liệu xảy ra trong chu kỳ xung clock 3. Từ chu kỳ xung clock 4 trở đi, cả bên gửi không có dữ liệu và bên nhận không sẵn sàng cho lần truyền tiếp theo. Tương tự như giao tiếp "luôn sẵn sàng", chúng ta có thể hình dung và giao tiếp luôn hợp lệ. Tuy nhiên, trong trường hợp đó, dữ liệu có thể sẽ không thay đổi trên tín hiệu `ready` và chúng ta sẽ đơn giản bỏ các tín hiệu bắt tay.

Hình 9.9 biểu diễn các biến thể khác của giao tiếp sẵn-sàng-hợp-lệ. Trong chu kỳ xung clock 1, cả hai tín hiệu (`ready` và `valid` đều được xác nhận chỉ trong một chu kỳ xung clock duy nhất và quá trình truyền dữ liệu của `D1` xảy ra. Dữ liệu có thể được truyền liên tục (trong mọi chu kỳ đồng hồ) như được trình bày trong chu kỳ xung clock 4 và 5 với việc truyền `D2` và `D3`.

Để làm cho giao tiếp này có thể soạn thảo được, cả tín hiệu `ready` và `valid` không được phép phụ thuộc vào tổ hợp của tín hiệu khác. Vì giao tiếp này rất phổ biến, nên Chisel định nghĩa `DecoupledIO` theo Bundle, tương tự như đoạn mã sau:

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

`DecoupledIO` theo Bundle được tham số hóa với kiểu cho `data`. Giao tiếp được định nghĩa bởi Chisel sử dụng trường `bit` cho dữ liệu.

Vẫn còn một câu hỏi là nếu tín hiệu `ready` hoặc `valid` có thể bị hủy xác nhận sau khi đang hoạt động và *không* có quá trình truyền dữ liệu nào xảy ra hay không. Ví dụ, một lúc nào đó bên nhận có thể sẵn sàng và không nhận dữ liệu, nhưng do một số sự kiện

khác có thể trở nên không sẵn sàng. Điều tương tự có thể được hình dung với bên gửi, dữ liệu chỉ có giá trị hợp lệ trong một số chu kỳ xung clock và trở nên không hợp lệ mà không có truyền dữ liệu. Hành vi này có được phép hay không thì không phải là một phần của giao tiếp sẵn-sàng-hợp-lệ, mà cần được định nghĩa bằng cách sử dụng giao tiếp cụ thể.

Chisel không đặt ra yêu cầu nào về tín hiệu `ready` và `valid` khi sử dụng lớp `DecoupledIO`. Tuy nhiên, lớp `IrrevocableIO` đặt ra các hạn chế sau đối với bên gửi:

> Một lớp con cụ thể của `ReadyValidIO` hứa hẹn sẽ không thay đổi giá trị của `các bit` sau một chu kỳ, trong đó tín hiệu `valid` ở mức cao và tín hiệu `ready` ở mức thấp. Ngoài ra, một khi tín hiệu `valid` được nâng lên, nó sẽ không bao giờ bị hạ xuống cho đến khi tín hiệu `ready` cũng được nâng lên.

Lưu ý rằng, đây là quy ước không thể được thực thi bằng cách sử dụng lớp `IrrevocableIO`.

AXI sử dụng một giao tiếp sẵn-sàng-hợp-lệ cho một trong các phần sau của bus: đọc địa chỉ, đọc dữ liệu, ghi địa chỉ và ghi dữ liệu. AXI hạn chế giao tiếp khi tín hiệu `ready` hoặc `valid` được xác nhận, nó sẽ không được phép hủy xác nhận cho đến khi quá trình truyền dữ liệu diễn ra.

10 Bộ tạo phần cứng

Điểm mạnh của Chisel là nó cho phép chúng ta viết cái gọi là bộ tạo phần cứng. Với các ngôn ngữ mô tả phần cứng cũ hơn, chẳng hạn như VHDL và Verilog, chúng ta thường sử dụng một ngôn ngữ khác, ví dụ: Java hoặc Python, để tạo phần cứng. Tác giả thường viết các chương trình Java nhỏ để tạo các bảng VHDL. Trong Chisel, toàn bộ sức mạnh của Scala (và các thư viện Java) là có sẵn khi xây dựng phần cứng. Vì vậy, chúng ta có thể viết các bộ tạo phần cứng của mình ở cùng một ngôn ngữ và thực thi chúng như một phần của quá trình tạo mạch Chisel.

10.1 Một chút tản mạn về Scala

Mục này giới thiệu rất ngắn gọn về Scala. Nó sẽ đủ để viết bộ tạo phần cứng cho Chisel. Để có phần giới thiệu chuyên sâu về Scala, tôi giới thiệu sách của Odersky và cộng sự [12].

Scala có hai loại biến: `val` và `var`. Biến `val` cung cấp tên cho một biểu thức và không thể gán lại giá trị. Đoạn mã sau đây trình bày định nghĩa của một giá trị số nguyên được gọi là `zero`. Nếu chúng ta cố gắng gán lại một giá trị về `zero`, chúng ta sẽ gặp lỗi biên dịch.

```
// A value is a constant
val zero = 0
// No new assignment is possible
// The following will not compile
zero = 3
```

Trong Chisel, chúng ta chỉ sử dụng `val` để đặt tên các thành phần phần cứng. Lưu ý rằng toán tử `:=` là toán tử Chisel và không phải là toán tử Scala.

Scala cũng cung cấp phiên bản cổ điển hơn của một biến như `var`. Đoạn mã sau định nghĩa một biến số nguyên và gán lại cho nó một giá trị mới:

```
// We can change the value of a var variable
var x = 2
x = 3
```

Chúng ta sẽ cần var trong Scala để viết *các bộ tạo* phần cứng, nhưng không bao giờ cần dùng nó để đặt tên cho một *thành phần* phần cứng.

Các bạn có thể đã tự hỏi xem các biến đó có kiểu gì. Như chúng ta đã gán một hằng số là số nguyên trong ví dụ trên, kiểu của biến được *suy luận ra*; nó là kiểu Scala `Int`. Trong hầu hết các trường hợp, trình biên dịch Scala có thể suy ra kiểu. Tuy nhiên, nếu chúng ta muốn rõ ràng hơn, chúng ta có thể khai báo kiểu như sau:

```
val number: Int = 42
```

Các vòng lặp cơ bản được viết như sau:

```
// Loops from 0 to 9
// Automatically creates loop value i
for (i <- 0 until 10) {
  println(i)
}
```

Chúng ta sử dụng một vòng lặp cho bộ tạo mạch. Vòng lặp sau kết nối các bit riêng lẻ của một thanh ghi dịch.

```
val shiftReg = RegInit(0.U(8.W))

shiftReg(0) := inVal

for (i <- 1 until 8) {
  shiftReg(i) := shiftReg(i-1)
}
```

Các lệnh điều kiện được biểu diễn bởi `if` và `else`. Lưu ý rằng điều kiện này được đánh giá tại thời gian chạy Scala trong quá trình tạo mạch. Cấu trúc này *không* tạo ra mạch đa hợp.

```
for (i <- 0 until 10) {
  if (i%2 == 0) {
    println(i + " is even")
  } else {
    println(i + " is odd")
  }
}
```

10.2 Cấu hình với các tham số

Các thành phần và chức năng của Chisel có thể được cấu hình với các tham số. Các tham số có thể đơn giản như một hằng số nguyên, nhưng cũng có thể là một kiểu phần cứng Chisel.

10.2.1 Các tham số đơn giản

Cách cơ bản để tham số hóa một mạch là định nghĩa độ rộng bit như một tham số. Các tham số có thể được truyền dưới dạng đối số cho bộ tạo cấu trúc của mô-đun Chisel. Ví dụ sau là một ví dụ về một mô-đun thực hiện một mạch cộng với độ rộng bit có thể được cấu hình. Độ rộng bit `n` là một tham số (thuộc kiểu Scala `Int`) của thành phần được truyền vào bộ tạo cấu trúc, có thể được sử dụng trong gói IO.

```
class ParamAdder(n: Int) extends Module {
  val io = IO(new Bundle{
    val a = Input(UInt(n.W))
    val b = Input(UInt(n.W))
    val c = Output(UInt(n.W))
  })

  io.c := io.a + io.b
}
```

Các phiên bản được tham số hóa của mạch cộng có thể được tạo như sau:

```
  val add8 = Module(new ParamAdder(8))
  val add16 = Module(new ParamAdder(16))
```

10.2.2 Các hàm với các tham số kiểu

Độ rộng bit như một tham số cấu hình chỉ là điểm khởi đầu cho các bộ tạo phần cứng. Một cấu hình rất linh hoạt là việc sử dụng các kiểu. Tính năng đó cho phép Chisel cung cấp một mạch đa hợp (`Mux`) có thể chấp nhận bất kỳ kiểu ghép kênh/đa hợp nào. Để chỉ ra cách sử dụng các kiểu cho việc cấu hình, chúng ta xây dựng một mạch đa hợp chấp nhận các kiểu tùy ý. Hàm sau định nghĩa mạch đa hợp:

```
  def myMux[T <: Data](sel: Bool, tPath: T, fPath: T): T = {

    val ret = WireDefault(fPath)
```

```
    when (sel) {
      ret := tPath
    }
    ret
  }
```

Chisel cho phép tham số hóa các hàm với các kiểu, trong trường hợp với các kiểu Chisel. Biểu thức trong dấu ngoặc vuông `[T <: Data]` định nghĩa tập hợp `T`, tham số kiểu là `Data` hoặc một lớp con của `Data`. `Data` là gốc của hệ thống kiểu Chisel.

Hàm đa hợp của chúng ta có ba tham số: điều kiện boolean, một tham số cho đường dẫn đúng và một tham số cho đường dẫn sai. Cả hai tham số đường dẫn đều là kiểu `T`, một thông tin được cung cấp khi gọi hàm. Bản thân hàm này là thẳng tiến: chúng ta định nghĩa một dây có giá trị mặc định là `fPath` và thay đổi giá trị nếu điều kiện đúng với `tPath`. Điều kiện này là một hàm đa hợp cổ điển. Ở cuối hàm, chúng ta trả về phần cứng của mạch đa hợp.

Chúng ta có thể sử dụng hàm đa hợp của mình với các kiểu đơn giản như `UInt`:

```
  val resA = myMux(selA, 5.U, 10.U)
```

Kiểu của hai đường dẫn mạch đa hợp cần phải giống nhau. Việc sử dụng sai mạch đa hợp sau đây dẫn đến lỗi thời gian chạy:

```
  val resErr = myMux(selA, 5.U, 10.S)
```

Chúng ta định nghĩa kiểu như `Bundle` với hai trường:

```
class ComplexIO extends Bundle {
  val d = UInt(10.W)
  val b = Bool()
}
```

Chúng ta có thể định nghĩa các hằng số `Bundle` bằng cách tạo một `Wire` đầu tiên và sau đó thiết lập các trường con. Sau đó, chúng ta có thể sử dụng mạch đa hợp đã được tham số hóa với kiểu phức tạp này.

```
  val tVal = Wire(new ComplexIO)
  tVal.b := true.B
  tVal.d := 42.U
  val fVal = Wire(new ComplexIO)
  fVal.b := false.B
  fVal.d := 13.U
```

```
// The mulitplexer with a complex type
val resB = myMux(selB, tVal, fVal)
```

Trong thiết kế ban đầu của chúng ta về hàm, chúng ta đã sử dụng `WireDefault` để tạo một dây có kiểu `T` với giá trị mặc định. Nếu chúng ta cần tạo một dây chỉ thuộc kiểu Chisel mà không sử dụng giá trị mặc định, chúng ta có thể sử dụng `fPath.cloneType` để lấy kiểu Chisel. Hàm sau đây biểu diễn cách thay thế để lập trình mạch đa hợp.

```
def myMuxAlt[T <: Data](sel: Bool, tPath: T, fPath: T): T = {

  val ret = Wire(fPath.cloneType)
  ret := fPath
  when (sel) {
    ret := tPath
  }
  ret
}
```

10.2.3 Mô-đun với các tham số kiểu

Chúng ta cũng có thể tham số hóa các mô-đun với các kiểu Chisel. Giả sử chúng ta muốn thiết kế một mạng trên chip (network-on-chip) để di chuyển dữ liệu giữa các lõi xử lý khác nhau. Tuy nhiên, chúng ta không muốn lập trình cứng định dạng dữ liệu trong giao tiếp bộ định tuyến; chúng ta muốn *tham số hóa* nó. Tương tự như tham số kiểu cho một hàm, chúng ta thêm tham số kiểu `T` vào bộ tạo cấu trúc mô-đun. Hơn nữa, chúng ta cần có một tham số cho bộ tạo cấu trúc của kiểu đó. Hơn nữa, trong ví dụ này, chúng ta cũng thực hiện số lượng cổng bộ định tuyến có thể cấu hình.

```
class NocRouter[T <: Data](dt: T, n: Int) extends Module {
  val io =IO(new Bundle {
    val inPort = Input(Vec(n, dt))
    val address = Input(Vec(n, UInt(8.W)))
    val outPort = Output(Vec(n, dt))
  })

  // Route the payload according to the address
  // ...
```

Để sử dụng bộ định tuyến, đầu tiên cần định nghĩa kiểu dữ liệu mà chúng ta muốn định tuyến, ví dụ như `Bundle` trong Chisel:

```
class Payload extends Bundle {
  val data = UInt(16.W)
  val flag = Bool()
}
```

Chúng ta tạo một bộ định tuyến bằng cách chuyển một thực thể của Bundle do người dùng định nghĩa và số lượng cổng cho bộ tạo kiến trúc của bộ định tuyến:

```
  val router = Module(new NocRouter(new Payload, 2))
```

10.2.4 Các Bundle được tham số hóa

Trong ví dụ về bộ định tuyến, chúng ta đã sử dụng hai vectơ trường khác nhau cho ngõ vào của bộ định tuyến: một cho địa chỉ và một cho dữ liệu, được tham số hóa. Một giải pháp thanh thoát hơn là có một `Bundle` mà bản thân nó đã được tham số hóa. Chẳng hạn như:

```
class Port[T <: Data](dt: T) extends Bundle {
  val address = UInt(8.W)
  val data = dt.cloneType
}
```

`Bundle` có một tham số kiểu `T`, là một kiểu con của kiểu `Data` trong Chisel. Trong gói, chúng ta định nghĩa một trường dữ liệu bằng cách gọi `cloneType` trên tham số. Tuy nhiên, khi chúng ta sử dụng một tham số bộ tạo cấu trúc, tham số này sẽ trở thành một trường công khai của lớp. Khi Chisel cần sao chép kiểu của `Bundle`, ví dụ: khi nó được sử dụng trong `Vec`, trường công khai này sẽ cản trở việc sao chép. Một giải pháp (cách giải quyết) cho vấn đề này là đặt trường tham số ở chế độ riêng tư:

```
class Port[T <: Data](private val dt: T) extends Bundle {
  val address = UInt(8.W)
  val data = dt.cloneType
}
```

Với `Bundle` mới đó, chúng ta có thể định nghĩa các cổng bộ định tuyến

```
class NocRouter2[T <: Data](dt: T, n: Int) extends Module {
  val io =IO(new Bundle {
    val inPort = Input(Vec(n, dt))
    val outPort = Output(Vec(n, dt))
  })
```

```
// Route the payload according to the address
// ...
```

và khởi tạo bộ định tuyến với một `Port` và lấy `Payload` làm tham số:

```
val router = Module(new NocRouter2(new Port(new Payload), 2))
```

10.3 Tạo mạch logic tổ hợp

Trong Chisel, chúng ta có thể dễ dàng tạo mạch logic bằng cách tạo một bảng logic với `Vec` trong Chisel từ `Array` trong Scala. Chúng ta có thể có dữ liệu trong một tập tin mà chúng ta có thể đọc trong thời gian tạo phần cứng cho bảng logic. Listing 10.1 trình bày cách sử dụng lớp `Source` từ thư viện chuẩn Scala để đọc tập tin "data.txt", chứa các hằng số nguyên trong biểu diễn dạng văn bản.

Đoạn mã có thể được biểu diễn vài dòng như sau:

```
val table = VecInit(array.map(_.U(8.W)))
```

Một mảng Scala `Array` có thể được chuyển đổi hoàn toàn thành một chuỗi (`Seq`), hỗ trợ hàm ánh xạ `map`. `map` gọi một hàm trên mỗi phần tử của chuỗi và trả về một chuỗi giá trị trả về của hàm. Hàm `_.U(8.W)` biểu diễn mỗi giá trị `Int` từ mảng Scala dưới dạng `_` và thực hiện chuyển đổi từ giá trị `Int` trong Scala thành chuỗi ký tự `UInt` trong Chisel, với kích thước 8-bit. Đối tượng `VecInit` trong Chisel tạo một `Vec` trong Chisel từ một chuỗi `Seq` của các kiểu Chisel.

Chúng ta có thể sử dụng toàn bộ sức mạnh của Scala để tạo ra logic (bảng). Ví dụ: tạo một bảng các hằng số điểm cố định để biểu diễn một hàm lượng giác, tính toán các hằng số cho các mạch lọc số hoặc viết một trình hợp ngữ nhỏ trong Scala để tạo mã cho một bộ vi xử lý được viết bằng Chisel. Tất cả các hàm đó có cùng cơ sở mã code (cùng ngôn ngữ) và có thể được thực thi trong quá trình tạo phần cứng.

Một ví dụ cổ điển là chuyển đổi một số nhị phân thành biểu diễn số BCD (binary-coded decimal). BCD được sử dụng để biểu diễn một số ở định dạng thập phân sử dụng 4-bit cho mỗi chữ số thập phân. Ví dụ: số thập phân `13` ở dạng nhị phân `1101` và số BCD được mã hoá thành 1 (hàng chục) và 3 (hàng đơn vị) ở dạng nhị phân: `00010011`. BCD cho phép hiển thị số dưới dạng thập phân, một cách biểu diễn các con số thân thiện với người dùng hơn hệ thập lục phân.

Khi sử dụng ngôn ngữ mô tả phần cứng cổ điển, chẳng hạn như Verilog hoặc VHDL, chúng ta sẽ sử dụng script hoặc ngôn ngữ lập trình khác để tạo một bảng như vậy. Chúng ta có thể viết một chương trình Java tính toán bảng để chuyển đổi số nhị phân sang BCD.

```
import chisel3._
import scala.io.Source

class FileReader extends Module {
  val io = IO(new Bundle {
    val address = Input(UInt(8.W))
    val data = Output(UInt(8.W))
  })

  val array = new Array[Int](256)
  var idx = 0

  // read the data into a Scala array
  val source = Source.fromFile("data.txt")
  for (line <- source.getLines()) {
    array(idx) = line.toInt
    idx += 1
  }

  // convert the Scala integer array
  // into a vector of Chisel UInt
  val table = VecInit(array.map(_.U(8.W)))

  // use the table
  io.data := table(io.address)
}
```

Listing 10.1: Đọc một tập tin văn bản để tạo bảng logic.

```
import chisel3._

class BcdTable extends Module {
  val io = IO(new Bundle {
    val address = Input(UInt(8.W))
    val data = Output(UInt(8.W))
  })

  val table = Wire(Vec(100, UInt(8.W)))

  // Convert binary to BCD
  for (i <- 0 until 100) {
    table(i) := (((i/10)<<4) + i%10).U
  }

  io.data := table(io.address)
}
```

Listing 10.2: Chuyển đổi hệ nhị phân sang BCD.

Chương trình Java đó in ra mã VHDL có thể được đưa vào trong tập tin Project. Chương trình Java có khoảng 100 dòng mã; hầu hết là các mã tạo chuỗi VHDL. Phần quan trọng của việc chuyển đổi chỉ có hai dòng.

Với Chisel, chúng ta có thể tính bảng này trực tiếp như một phần của việc tạo phần cứng. Listing 10.2 biểu diễn cách tạo bảng để chuyển đổi số nhị phân sang BCD.

10.4 Sử dụng kế thừa

Chisel là một ngôn ngữ hướng đối tượng. Một thành phần phần cứng, mô-đun Chisel là một lớp Scala. Do đó, chúng ta có thể sử dụng tính kế thừa để đưa một hành vi chung vào một lớp cha. Chúng ta sẽ khám phá cách sử dụng kế thừa với một ví dụ.

Trong Phần 6.2, chúng ta đã khám phá các dạng mạch đếm khác nhau, có thể được sử dụng để tạo tick tần suất thấp. Giả sử chúng ta muốn khám phá các phiên bản khác nhau đó, ví dụ: để so sánh yêu cầu tài nguyên của chúng. Chúng ta bắt đầu với một lớp trừu tượng để định nghĩa giao tiếp tick:

```
abstract class Ticker(n: Int) extends Module {
  val io = IO(new Bundle{
```

```
class UpTicker(n: Int) extends Ticker(n) {

  val N = (n-1).U

  val cntReg = RegInit(0.U(8.W))

  cntReg := cntReg + 1.U
  when(cntReg === N) {
    cntReg := 0.U
  }

  io.tick := cntReg === N
}
```

Listing 10.3: Tạo tick với mạch đếm.

```
    val tick = Output(Bool())
  })
}
```

Listing 10.3 biểu diễn lần thực hiện đầu tiên của lớp trừu tượng đó với một mạch đếm, đếm lên, cho việc tạo tick.

Chúng ta có thể kiểm tra tất cả các phiên bản khác nhau của logic *ticker* bằng một testbench duy nhất. Chúng ta *chỉ* cần định nghĩa testbench để chấp nhận các kiểu con của `Ticker`. Listing 10.4 biểu diễn mã Chisel cho trình kiểm tra. `TickerTester` có nhiều tham số: (1) tham số kiểu `[T <: Ticker]` để chấp nhận một `Ticker` hoặc bất kỳ lớp nào kế thừa từ `Ticker`, (2) thiết kế đang được kiểm tra, thuộc kiểu `T` hoặc kiểu con của chúng và (3) số chu kỳ xung clock mà chúng ta mong đợi cho mỗi tick. Trình kiểm tra chờ sự xuất hiện đầu tiên của tick (thời điểm bắt đầu có thể khác đối với các triển khai khác nhau) và sau đó kiểm tra xem `tick` có lặp lại sau mỗi *n* chu kỳ xung clock không.

Với thực thi lần đầu tiên, dễ dàng của ticker, chúng ta có thể tự kiểm tra trình kiểm tra, có thể bằng cách gỡ lỗi với lệnh `println`. Khi chúng ta tự tin rằng ticker đơn giản và trình kiểm tra là chính xác, chúng ta có thể tiếp tục và khám phá thêm hai phiên bản khác của ticker. Listing 10.5 biểu diễn việc tạo tick với mạch đếm xuống đến 0. Listing 10.6 biểu diễn phiên bản Nerd đếm xuống đến -1 nhằm sử dụng ít phần cứng hơn bằng cách tránh dùng mạch so sánh.

Chúng ta có thể kiểm tra tất cả ba phiên bản của ticker bằng cách sử dụng các đặc tả kỹ thuật của ScalaTest, tạo các thực thể của các phiên bản khác nhau của ticker và

```
import chisel3.iotesters.PeekPokeTester
import org.scalatest._

class TickerTester[T <: Ticker](dut: T, n: Int) extends
    PeekPokeTester(dut: T) {

  // -1 is the notion that we have not yet seen the first tick
  var count = -1
  for (i <- 0 to n * 3) {
    if (count > 0) {
      expect(dut.io.tick, 0)
    }
    if (count == 0) {
      expect(dut.io.tick, 1)
    }
    val t = peek(dut.io.tick)
    // On a tick we reset the tester counter to N-1,
    // otherwise we decrement the tester counter
    if (t == 1) {
      count = n-1
    } else {
      count -= 1
    }

    step(1)
  }
}
```

Listing 10.4: Trình kiểm tra cho các phiên bản khác của ticker.

```
class DownTicker(n: Int) extends Ticker(n) {

  val N = (n-1).U

  val cntReg = RegInit(N)

  cntReg := cntReg - 1.U
  when(cntReg === 0.U) {
    cntReg := N
  }

  io.tick := cntReg === N
}
```

Listing 10.5: Tạo tick với mạch đếm xuống.

```
class NerdTicker(n: Int) extends Ticker(n) {

  val N = n

  val MAX = (N - 2).S(8.W)
  val cntReg = RegInit(MAX)
  io.tick := false.B

  cntReg := cntReg - 1.S
  when(cntReg(7)) {
    cntReg := MAX
    io.tick := true.B
  }
}
```

Listing 10.6: Tạo tick bằng cách đếm xuống tới -1.

```
class TickerSpec extends FlatSpec with Matchers {

  "UpTicker 5" should "pass" in {
    chisel3.iotesters.Driver(() => new UpTicker(5)) { c =>
      new TickerTester(c, 5)
    } should be (true)
  }

  "DownTicker 7" should "pass" in {
    chisel3.iotesters.Driver(() => new DownTicker(7)) { c =>
      new TickerTester(c, 7)
    } should be (true)
  }

  "NerdTicker 11" should "pass" in {
    chisel3.iotesters.Driver(() => new NerdTicker(11)) { c =>
      new TickerTester(c, 11)
    } should be (true)
  }
}
```

Listing 10.7: Đặc tả ScalaTest cho kiểm tra ticker.

chuyển chúng đến testbench chung. Listing 10.7 trình bày các đặc tả kỹ thuật. Chúng ta chỉ chạy các bài kiểm tra ticker với lệnh:

```
sbt "testOnly TickerSpec"
```

10.5 Tạo phần cứng với lập trình hàm

Scala hỗ trợ lập trình hàm, Chisel cũng vậy. Chúng ta có thể sử dụng các hàm để biểu diễn phần cứng và kết hợp các thành phần phần cứng đó với lập trình hàm (bằng cách sử dụng cái gọi là "hàm bậc cao"). Chúng ta hãy bắt đầu với một ví dụ đơn giản, tổng của một véc-tơ:

```
  def add(a: UInt, b:UInt) = a + b

  val sum = vec.reduce(add)
```

Đầu tiên, chúng ta định nghĩa phần cứng cho mạch cộng trong hàm add. Véc-tơ (kiểu Vec) trong Chisel nằm trong vec. Phương thức Scala reduce() kết hợp tất cả các phần tử của một tập hợp với một phép toán nhị phân, tạo ra một giá trị duy nhất. Phương thức reduce() giảm tuần tự bắt đầu từ bên trái. Nó lấy hai phần tử đầu tiên và thực hiện phép toán. Kết quả sau đó được kết hợp với phần tử tiếp theo, cho đến khi còn lại một kết quả duy nhất.

Hàm để kết hợp với các phần tử được cung cấp dưới dạng tham số để rút gọn, trong trường hợp này là cộng, sẽ trả về một mạch cộng. Kết quả phần cứng là một chuỗi các mạch cộng tính toán tổng các phần tử của véc-tơ vec.

Thay vì định nghĩa hàm cộng (đơn giản), chúng ta có thể đưa ra phép cộng dưới dạng hàm ẩn danh và sử dụng ký tự đại diện Scala "_" để biểu diễn hai toán hạng.

```
val sum = vec.reduce(_ + _)
```

Với câu lệnh một dòng duy nhất này, chúng ta đã tạo ra chuỗi các mạch cộng. Đối với hàm tổng, một chuỗi không phải là cấu hình lý tưởng, một cây sẽ có độ trễ tổ hợp ngắn hơn. Nếu chúng ta không tin tưởng vào công cụ tổng hợp để sắp xếp lại chuỗi mạch cộng, thì có thể sử dụng phương thức ReduceTree của Chisel để tạo một cây các mạch cộng:

```
val sum = vec.reduceTree(_ + _)
```

Trong một ví dụ chi tiết hơn, chúng ta xây dựng một mạch điện để tìm giá trị nhỏ nhất trong vec. Để biểu diễn mạch điện này, chúng ta sử dụng một hàm ẩn danh, được gọi là *function Lite* trong Scala. Cú pháp cho một ký tự hàm là các tham số trong dấu ngoặc đơn, theo sau bởi =>, và theo sau là nội dung hàm (function body):

```
(param) => function body
```

Ký tự hàm cho hàm tối thiểu sử dụng hai tham số x và y, và trả về một mạch đa hợp (Mux) để so sánh hai tham số và trả về giá trị nhỏ hơn.

```
val min = vec.reduceTree((x, y) => Mux(x < y, x, y))
```

Chúng ta hãy cùng mở rộng mạch điện này để trả về không chỉ giá trị tối thiểu từ vec, mà còn là vị trí (chỉ mục) trong vec. Để trả về hai giá trị, chúng ta định nghĩa Two kiểu Bundle để giữ giá trị và chỉ mục. Chúng ta khai báo vecTwo Vec có thể giữ các bundle này và kết nối chúng trong một vòng lặp với ngõ vào và chỉ mục gốc trong Vec.

Như trước đây, khi chúng ta sử dụng một ký tự hàm trong phương thức ReduceTree của vecTwo, so sánh trường giá trị trong bundle và trả về bundle hoàn chỉnh từ mạch đa hợp. Giá trị res trỏ đến bundle chứa vị trí và giá trị nhỏ nhất.

```
class Two extends Bundle {
  val v = UInt(w.W)
  val idx = UInt(8.W)
}

val vecTwo = Wire(Vec(n, new Two()))
for (i <- 0 until n) {
  vecTwo(i).v := vec(i)
  vecTwo(i).idx := i.U
}

val res = vecTwo.reduceTree((x, y) => Mux(x.v < y.v, x, y))
```

Với là một biến thể cuối cùng, chúng ta sử dụng nhiều đặc trưng Scala hơn để tránh tạo bundle để trả về giá trị và chỉ mục. Scala có quan niệm về bộ (dữ liệu), là chuỗi giá trị bất biến của các kiểu khác nhau. Đoạn mã code sau đây cho thấy ứng dụng của một chuỗi các hàm đối với tuần tự nguyên gốc. Các hàm chuỗi là một mẫu điển hình trong lập trình hàm. Mẫu này cũng có thể được xem như một đường ống của các phép toán.

Hàm đầu tiên (`zipWithIndex`) chuyển đổi chuỗi tuần tự nguyên gốc thành một chuỗi các bộ dữ liệu, ở đó phần tử thứ hai là giá trị chỉ mục. Nói chung, hàm `zip` hợp nhất hai chuỗi (nén chúng) thành một chuỗi duy nhất chứa hai phần tử dưới dạng bộ dữ liệu. Hàm tiếp theo ánh xạ bộ đôi của `UInt` trong Chisel và `Int` trong Scala thành hai `UInt` trong Chisel. Hàm `reduce` cung cấp việc tạo ra kết quả tìm kiếm tối thiểu. Chúng ta so sánh phần tử đầu tiên của bộ dữ liệu trong hai mạch đa hợp và trả về một bộ dữ liệu chứa vị trí và giá trị nhỏ nhất dưới dạng các kiểu `UInt` trong Chisel.

```
val resFun = vec.zipWithIndex
  .map ((x) => (x._1, x._2.U))
  .reduce((x, y) => (Mux(x._1 < y._1, x._1, y._1), Mux(x._1 <
      y._1, x._2, y._2)))
```

Lưu ý rằng toàn bộ biểu thức hàm sử dụng `Vector` trong Scala để giữ các kết quả trung gian, nhưng trả về phần cứng (các mạch đa hợp được kết nối) chỉ bao gồm các kiểu Chisel. Vì chúng ta sử dụng `Vector` trong Scala ở đây, nên chúng ta không thể sử dụng `ReduceTree`, vốn chỉ có sẵn trên `Vec` của Chisel.

```
val scalaVector = vec.zipWithIndex
  .map((x) => MixedVecInit(x._1, x._2.U(8.W)))
val resFun2 = VecInit(scalaVector)
  .reduceTree((x, y) => Mux(x(0) < y(0), x, y))
```

11 Thiết kế ví dụ

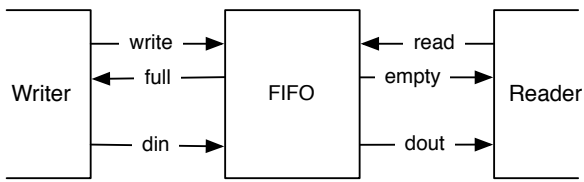
Trong phần này, chúng ta khám phá một số thiết kế mạch số kích thước nhỏ, chẳng hạn như bộ đệm FIFO được sử dụng làm khối xây dựng cho thiết kế lớn hơn. Trong ví dụ khác, chúng ta thiết kế một giao tiếp nối tiếp (còn được gọi là UART), bản thân UART này có thể sử dụng bộ đệm FIFO.

11.1 Bộ đệm FIFO

Chúng ta có thể tách bên ghi (bên gửi) và bên đọc (bên nhận) bằng một bộ đệm giữa bên ghi và bên đọc. Bộ đệm chung là bộ đệm vào trước, ra trước (FIFO). Hình 11.1 biểu diễn bên ghi, FIFO và bên đọc. Dữ liệu được đặt vào trong FIFO bởi bên ghi trên `din` đưa vào FIFO với tín hiệu `write` đang hoạt động. Dữ liệu được đọc từ FIFO bởi bên đọc trên `dout` với tín hiệu `read` đang hoạt động.

Một FIFO ban đầu thường trống được báo hiệu bởi tín hiệu `empty`. Việc đọc từ FIFO trống thường không được xác định. Khi dữ liệu được ghi và không bao giờ được đọc, FIFO sẽ trở nên `full` (đầy). Việc ghi vào FIFO đầy thường bị bỏ qua và dữ liệu bị mất. Nói cách khác, các tín hiệu `empty` và `full` đóng vai trò là các tín hiệu bắt tay.

Có thể có nhiều thực hiện khác nhau của FIFO: Ví dụ: sử dụng bộ nhớ trên chip, các con trỏ đọc và ghi, hoặc đơn giản là một chuỗi các thanh ghi với một máy trạng thái nhỏ. Với các bộ đệm nhỏ (khoảng hàng chục phần tử), một FIFO được tổ chức với các thanh ghi riêng lẻ được kết nối thành một chuỗi bộ đệm là một cách thực hiện đơn giản với yêu cầu tài nguyên thấp. Mã code của FIFO bubble có sẵn trong kho lưu trữ chisel-examples.



Hình 11.1: Bên ghi, bộ đệm FIFO, và bên đọc.

1

Chúng ta bắt đầu bằng định nghĩa IO cho bên ghi và bên đọc. Kích thước dữ liệu có thể cấu hình bởi `size`. Dữ liệu ghi là `din` và một phép ghi được báo hiệu bởi chân `write`. Tín hiệu `full` thực hiện điều khiển luồng ở bên ghi.

```
class WriterIO(size: Int) extends Bundle {
  val write = Input(Bool())
  val full = Output(Bool())
  val din = Input(UInt(size.W))
}
```

Bên đọc cung cấp dữ liệu với `dout` và quá trình đọc được bắt đầu với tín hiệu `read`. Tín hiệu `empty` chịu trách nhiệm cho việc điều khiển luồng ở bên đọc.

```
class ReaderIO(size: Int) extends Bundle {
  val read = Input(Bool())
  val empty = Output(Bool())
  val dout = Output(UInt(size.W))
}
```

Listing 11.1 biểu diễn một bộ đệm đơn. Bộ đệm có cổng xếp hàng `enq` của kiểu `WriterIO` và cổng rời hàng `deq` của kiểu `ReaderIO`. Các phần tử trạng thái của bộ đệm là một thanh ghi giữ dữ liệu (`dataReg`) và một thanh ghi trạng thái cho FSM đơn giản (`stateReg`). FSM chỉ có hai trạng thái: bộ đệm `empty` hoặc `full`. Nếu bộ đệm `empty`, phép ghi sẽ ghi lại dữ liệu ngõ vào và thay đổi thành trạng thái `full`. Nếu bộ đệm `full`, phép đọc sẽ lấy dữ liệu ra và chuyển sang trạng thái `empty`. Các cổng IO `full` và `empty` biểu diễn trạng thái bộ đệm cho bên ghi và bên đọc.

Listing 11.2 biểu diễn FIFO hoàn chỉnh. FIFO hoàn chỉnh có giao tiếp IO giống như các bộ đệm FIFO riêng lẻ. `BubbleFifo` có các tham số là `size` của từ (word) dữ liệu và `depth` cho số tầng bộ đệm. Chúng ta có thể xây dựng một FIFO bubble có `depth` tầng từ các `FifoRegister`. Chúng ta sắp xếp các tầng bằng cách điền chúng vào một kiểu `Array` trong Scala. Mảng Scala không có ý nghĩa về phần cứng, nó *chỉ* cung cấp cho chúng ta một vùng chứa để có các tham chiếu đến các bộ đệm đã tạo. Trong vòng lặp `for` của Scala, chúng ta kết nối các bộ đệm riêng lẻ. Bên xếp hàng của bộ đệm đầu tiên được kết nối với IO xếp hàng của FIFO hoàn chỉnh và bên rời hàng của bộ đệm cuối cùng nối với bên rời hàng của FIFO hoàn chỉnh.

Ý tưởng đã trình bày về việc kết nối các bộ đệm riêng lẻ để thực hiện một hàng đợi FIFO được gọi là FIFO bubble, khi dữ liệu thả nổi (bubble) qua hàng đợi. Điều này đơn giản và là một giải pháp tốt khi tốc độ dữ liệu chậm hơn đáng kể so với tốc độ xung

[1] Để hoàn thiện, kho lưu trữ sách Chisel cũng chứa bản sao của mã code FIFO.

```
class FifoRegister(size: Int) extends Module {
  val io = IO(new Bundle {
    val enq = new WriterIO(size)
    val deq = new ReaderIO(size)
  })

  val empty :: full :: Nil = Enum(2)
  val stateReg = RegInit(empty)
  val dataReg = RegInit(0.U(size.W))

  when(stateReg === empty) {
    when(io.enq.write) {
      stateReg := full
      dataReg := io.enq.din
    }
  }.elsewhen(stateReg === full) {
    when(io.deq.read) {
      stateReg := empty
      dataReg := 0.U // just to better see empty slots in the
          waveform
    }
  }.otherwise {
    // There should not be an otherwise state
  }

  io.enq.full := (stateReg === full)
  io.deq.empty := (stateReg === empty)
  io.deq.dout := dataReg
}
```

Listing 11.1: Tầng đơn của FIFO bubble.

```
class BubbleFifo(size: Int, depth: Int) extends Module {
  val io = IO(new Bundle {
    val enq = new WriterIO(size)
    val deq = new ReaderIO(size)
  })

  val buffers = Array.fill(depth) { Module(new
      FifoRegister(size)) }
  for (i <- 0 until depth - 1) {
    buffers(i + 1).io.enq.din := buffers(i).io.deq.dout
    buffers(i + 1).io.enq.write := ~buffers(i).io.deq.empty
    buffers(i).io.deq.read := ~buffers(i + 1).io.enq.full
  }
  io.enq <> buffers(0).io.enq
  io.deq <> buffers(depth - 1).io.deq
}
```

Listing 11.2: FIFO gồm một mảng các tầng FIFO bubble.

clock, ví dụ, như một bộ đệm tách rời cho cổng nối tiếp sẽ được trình bày trong phần tiếp theo.

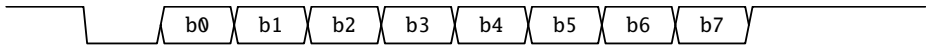
Tuy nhiên, khi tốc độ dữ liệu tiếp cận với tần số xung clock, FIFO bubble có hai hạn chế: (1) Vì mỗi trạng thái của bộ đệm phải chuyển đổi giữa *blank* và *full*, có nghĩa là thông lượng tối đa của FIFO là 2 chu kỳ xung clock mỗi từ (word). (2) Dữ liệu cần bubble thông qua FIFO hoàn chỉnh, vì vậy, độ trễ từ ngõ vào đến ngõ ra là số bộ đệm tối thiểu. Tôi sẽ trình bày các thực hiện khả dĩ khác của FIFO trong Phần 11.3.

11.2 Cổng nối tiếp

Cổng nối tiếp (còn được gọi là UART hoặc RS-232) là một trong những tùy chọn dễ dàng nhất để giao tiếp giữa máy tính và bo mạch FPGA. Như tên của nó, dữ liệu được truyền nối tiếp. Một byte 8-bit được truyền như sau: một start bit (0), dữ liệu 8-bit, bit LSB đầu tiên, và sau đó là một hoặc hai stop bit (1). Khi không có dữ liệu nào được truyền, ngõ ra là 1. Hình 11.2 biểu diễn sơ đồ thời gian của một byte được truyền.

Chúng ta thiết kế UART của mình theo cách mô-đun với chức năng tối thiểu trên mỗi mô-đun. Chúng ta trình bày một bộ phát (TX), một bộ thu (RX), một bộ đệm và sau đó là cách sử dụng các thành phần cơ sở đó.

Đầu tiên, chúng ta cần một giao tiếp, một định nghĩa cổng. Đối với thiết kế UART,



Hình 11.2: Một byte được truyền bởi UART.

chúng ta sử dụng giao tiếp bắt tay sẵn sàng/hợp lệ, với hướng (vào/ra) như được thấy từ bộ phát.

```
class Channel extends Bundle {
  val data = Input(Bits(8.W))
  val ready = Output(Bool())
  val valid = Input(Bool())
}
```

Quy ước của giao tiếp sẵn sàng/hợp lệ là dữ liệu được truyền khi cả hai tín hiệu `ready` và `valid` được xác nhận.

Listing 11.3 biểu diễn một bộ phát nối tiếp cơ bản nhất (`Tx`). Các cổng IO là cổng `txd`, nơi dữ liệu nối tiếp được gửi và một cổng `channel` nơi bộ phát có thể nhận các ký tự để chuyển thành dạng nối tiếp và gửi. Để tạo ra thời gian chính xác, chúng ta tính toán một hằng số bằng cách tính thời gian theo chu kỳ xung clock cho một bit nối tiếp.

Chúng ta sử dụng ba thanh ghi: (1) thanh ghi để dịch dữ liệu (chuyển dữ liệu thành dạng nối tiếp) (`shiftReg`), (2) một mạch đếm để tạo tốc độ baud chính xác (`cntReg`), và (3) một mạch đếm số bit vẫn cần được dịch ra ngoài. Không cần thanh ghi trạng thái FSM bổ sung, tất cả trạng thái được mã hóa trong ba thanh ghi đó.

Mạch đếm `cntReg` liên tục chạy (đếm xuống đến 0 và reset về giá trị bắt đầu khi bằng 0). Tất cả hoạt động chỉ được thực hiện khi `cntReg` bằng 0. Khi chúng ta xây dựng một bộ phát tối thiểu, chúng ta chỉ có thanh ghi dịch để lưu trữ dữ liệu. Vì vậy, kênh chỉ sẵn sàng khi `cntReg` bằng 0 và không còn bit nào để dịch ra ngoài.

Cổng IO `txd` được kết nối trực tiếp với bit LSB của thanh ghi dịch.

Khi có nhiều bit hơn để dịch ra (`bitsReg =/= 0.U`), chúng ta dịch các bit sang phải và điền bằng 1 từ mức trên cùng (mức nhàn rỗi của máy phát). Nếu không còn bit nào cần dịch ra ngoài, chúng ta kiểm tra xem kênh có chứa dữ liệu hay không (được báo hiệu bằng cổng `valid`). Nếu vậy, chuỗi bit được dịch ra ngoài được xây dựng với một start bit (0), dữ liệu 8-bit và hai stop bit (1). Do đó, số bit đếm được đặt thành 11.

Bộ phát rất tối thiểu này không có bộ đệm bổ sung và chỉ có thể chấp nhận một ký tự mới khi thanh ghi dịch trống và ở chu kỳ xung clock khi `cntReg` bằng 0. Chấp nhận dữ liệu mới chỉ khi `cntReg` bằng 0 cũng có nghĩa là cờ sẵn sàng cũng được loại bỏ xác nhận khi có chỗ trống trong thanh ghi dịch. Tuy nhiên, chúng tôi không muốn thêm "độ phức tạp" này vào bộ phát mà chỉ ủy thác nó vào bộ đệm.

```
class Tx(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(Bits(1.W))
    val channel = new Channel()
  })

  val BIT_CNT = ((frequency + baudRate / 2) / baudRate -
      1).asUInt()

  val shiftReg = RegInit(0x7ff.U)
  val cntReg = RegInit(0.U(20.W))
  val bitsReg = RegInit(0.U(4.W))

  io.channel.ready := (cntReg === 0.U) && (bitsReg === 0.U)
  io.txd := shiftReg(0)

  when(cntReg === 0.U) {

    cntReg := BIT_CNT
    when(bitsReg =/= 0.U) {
      val shift = shiftReg >> 1
      shiftReg := Cat(1.U, shift(9, 0))
      bitsReg := bitsReg - 1.U
    }.otherwise {
      when(io.channel.valid) {
        // two stop bits, data, one start bit
        shiftReg := Cat(Cat(3.U, io.channel.data), 0.U)
        bitsReg := 11.U
      }.otherwise {
        shiftReg := 0x7ff.U
      }
    }

  }.otherwise {
    cntReg := cntReg - 1.U
  }
}
```

Listing 11.3: Bộ phát cho cổng nối tiếp.

```
class Buffer extends Module {
  val io = IO(new Bundle {
    val in = new Channel()
    val out = Flipped(new Channel())
  })

  val empty :: full :: Nil = Enum(2)
  val stateReg = RegInit(empty)
  val dataReg = RegInit(0.U(8.W))

  io.in.ready := stateReg === empty
  io.out.valid := stateReg === full

  when(stateReg === empty) {
    when(io.in.valid) {
      dataReg := io.in.data
      stateReg := full
    }
  }.otherwise { // full
    when(io.out.ready) {
      stateReg := empty
    }
  }
  io.out.data := dataReg
}
```

Listing 11.4: Bộ đệm byte đơn với giao tiếp sẵn sàng/hợp lệ.

```
class BufferedTx(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(Bits(1.W))
    val channel = new Channel()
  })
  val tx = Module(new Tx(frequency, baudRate))
  val buf = Module(new Buffer())

  buf.io.in <> io.channel
  tx.io.channel <> buf.io.out
  io.txd <> tx.io.txd
}
```

Listing 11.5: Bộ phát với bộ đệm bổ sung.

Listing 11.4 biểu diễn bộ đệm byte đơn, tương tự như thanh ghi FIFO cho FIFO bubble. Cổng ngõ vào là giao tiếp `Channel` và ngõ ra là giao tiếp `Channel` với các hướng lật. Bộ đệm chứa máy trạng thái tối thiểu để cho biết FIFO là `empty` hoặc `full`. Các tín hiệu bắt tay điều khiển bộ đệm (`in.ready` và `out.valid` phụ thuộc vào thanh ghi trạng thái.

Khi trạng thái là `empty` và dữ liệu trên ngõ vào là `valid`, chúng ta nhận dữ liệu và chuyển sang trạng thái `full`. Khi trạng thái là `full` và bộ thu phía dưới là `ready`, quá trình truyền dữ liệu xuống sẽ xảy ra và chúng chuyển trở lại trạng thái `empty`.

Với bộ đệm đó, chúng ta có thể mở rộng bộ phát cơ bản của mình. Listing 11.5 biểu diễn sự kết hợp của bộ phát `Tx` với một bộ đệm đơn phía trước. Bộ đệm này hiện giải quyết vấn đề rằng `Tx` chỉ sẵn sàng cho các chu kỳ xung clock đơn lẻ. Chúng ta đã ủy quyền giải pháp của vấn đề này cho mô-đun bộ đệm. Có thể dễ dàng thực hiện việc mở rộng bộ đệm từ đơn thành FIFO thực và không cần thay đổi bộ phát hoặc bộ đệm byte đơn.

Listing 11.6 hiển thị mã code cho bộ thu (`Rx`). Bộ thu hơi phức tạp một chút vì nó cần phải tái xây dựng lại thời gian của dữ liệu nối tiếp. Bộ thu đợi cho cạnh xuống của start bit. Từ sự kiện đó, bộ thu đợi 1,5 lần bit để định vị chính nó vào giữa bit 0. Sau đó, nó dịch các bit mỗi thời gian của bit. Các bạn có thể quan sát hai lần chờ này với `START_CNT` và `BIT_CNT`. Đối với cả hai lần, cùng một mạch đếm (`cntReg`) được sử dụng. Sau khi 8-bit được dịch vào, tín hiệu `valReg` báo hiệu một byte có sẵn

Listing 11.7 biểu diễn cách sử dụng bộ phát cổng nối tiếp bằng cách gửi một thông điệp thân thiện ra ngoài. Chúng ta định nghĩa thông điệp trong một chuỗi Scala (`msg`) và chuyển đổi nó thành một `Vec` trong Chisel của `UInt`. Chuỗi Scala là một trình tự hỗ trợ

```
class Rx(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val rxd = Input(Bits(1.W))
    val channel = Flipped(new Channel())
  })

  val BIT_CNT = ((frequency + baudRate / 2) / baudRate - 1).U
  val START_CNT = ((3 * frequency / 2 + baudRate / 2) /
      baudRate - 1).U

  // Sync in the asynchronous RX data
  // Reset to 1 to not start reading after a reset
  val rxReg = RegNext(RegNext(io.rxd, 1.U), 1.U)

  val shiftReg = RegInit('A'.U(8.W))
  val cntReg = RegInit(0.U(20.W))
  val bitsReg = RegInit(0.U(4.W))
  val valReg = RegInit(false.B)

  when(cntReg =/= 0.U) {
    cntReg := cntReg - 1.U
  }.elsewhen(bitsReg =/= 0.U) {
    cntReg := BIT_CNT
    shiftReg := Cat(rxReg, shiftReg >> 1)
    bitsReg := bitsReg - 1.U
    // the last shifted in
    when(bitsReg === 1.U) {
      valReg := true.B
    }
  // wait 1.5 bits after falling edge of start
  }.elsewhen(rxReg === 0.U) {
    cntReg := START_CNT
    bitsReg := 8.U
  }

  when(valReg && io.channel.ready) {
    valReg := false.B
  }

  io.channel.data := shiftReg
  io.channel.valid := valReg
}
```

Listing 11.6: Bộ thu cho một cổng nối tiếp.

```
class Sender(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(Bits(1.W))
  })

  val tx = Module(new BufferedTx(frequency, baudRate))

  io.txd := tx.io.txd

  val msg = "Hello World!"
  val text = VecInit(msg.map(_.U))
  val len = msg.length.U

  val cntReg = RegInit(0.U(8.W))

  tx.io.channel.data := text(cntReg)
  tx.io.channel.valid := cntReg =/= len

  when(tx.io.channel.ready && cntReg =/= len) {
    cntReg := cntReg + 1.U
  }
}
```

Listing 11.7: Gửi “Hello World!" qua cổng nối tiếp.

```
class Echo(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(Bits(1.W))
    val rxd = Input(Bits(1.W))
  })

  val tx = Module(new BufferedTx(frequency, baudRate))
  val rx = Module(new Rx(frequency, baudRate))
  io.txd := tx.io.txd
  rx.io.rxd := io.rxd
  tx.io.channel <> rx.io.channel
}
```

Listing 11.8: Dữ liệu dội lại trên cổng nối tiếp.

phương pháp ánh xạ. Phương pháp ánh xạ nhận đối số là một ký tự của hàm, áp dụng hàm này cho từng phần tử và xây dựng một chuỗi các giá trị trả về của hàm. Nếu ký tự của hàm chỉ có một đối số, như trong trong trường hợp này, đối số có thể được biểu diễn bằng _. Ký tự của hàm theo nghĩa đen gọi phương pháp Chisel .U để chuyển đổi Char trong Scala thành UInt trong Chisel. Trình tự sau đó được chuyển tới VecInit để tạo một Vec trong Chisel. Chúng ta lập chỉ mục thành véc-tơ văn bản với mạch đếm cntReg để cung cấp các ký tự riêng lẻ cho bộ phát được đệm. Với mỗi tín hiệu ready, chúng ta tăng mạch đếm cho đến khi chuỗi đầy đủ được gửi đi. Bên gửi giữ tín hiệu valid được xác nhận cho đến khi ký tự cuối cùng được gửi đi.

Listing 11.8 biểu diễn cách sử dụng của bộ thu và bộ phát bằng cách kết nối chúng với nhau. Kết nối này tạo ra một mạch Echo trong đó mỗi ký tự nhận được sẽ được gửi lại (bị dội lại).

11.3 Các biến thể thiết kế FIFO

Trong phần này, chúng ta sẽ thực hiện các biến thể khác nhau của hàng đợi FIFO. Để làm cho các thực hiện này có thể hoán đổi cho nhau, chúng ta sẽ sử dụng tính năng thừa kế, như đã giới thiệu trong Phần 10.4

11.3.1 Tham số hóa các FIFO

Chúng ta định nghĩa một lớp FIFO `abstract` với kiểu Chisel làm tham số để có thể đệm bất kỳ kiểu dữ liệu Chisel nào. Trong lớp trừu tượng, chúng ta cũng kiểm tra xem tham số `depth` có giá trị hữu ích hay không.

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

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

Trong Phần 11.1, chúng ta đã định nghĩa các kiểu riêng cho giao tiếp với các tên chung cho các tín hiệu, chẳng hạn như `write`, `full`, `din`, `read`, `empty` và `dout`. Ngõ vào và ngõ ra của bộ đệm như vậy bao gồm dữ liệu và hai tín hiệu để bắt tay (ví dụ: chúng ta ghi vào FIFO qua tín hiệu `write` khi nó không bị đầy, báo hiệu bởi tín hiệu `full`.

Tuy nhiên, chúng ta có thể khái quát hóa sự bắt tay này thành giao tiếp được gọi là sẵn-sàng-hợp-lệ. Ví dụ: chúng ta có thể xếp hàng một phần tử (ghi vào FIFO) khi FIFO sẵn sàng với `ready`. Chúng ta báo hiệu điều này ở bên bộ ghi với tín hiệu `valid`. Vì giao tiếp sẵn-sàng-hợp-lệ này rất phổ biến, nên Chisel đưa ra định nghĩa về giao tiếp này trong `DecoupledIO` như sau:[2]

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

Với giao tiếp `DecoupledIO`, chúng ta định nghĩa giao tiếp cho FIFO: tín hiệu `FifoIO` với cổng xếp hàng `enq` và cổng rời hàng `deq` bao gồm các giao tiếp sẵn-sàng-hợp-lệ. Giao tiếp `DecoupledIO` được định nghĩa từ quan điểm của bên ghi (bên sản xuất). Vì vậy, cổng xếp hàng của FIFO cần lật các hướng tín hiệu.

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

[2] Đây là một sự đơn giản hóa, vì `DecoupledIO` thực sự mở rộng một lớp trừu tượng.

Với lớp cơ sở trừu tượng và một giao tiếp, chúng ta có thể chuyên biệt hóa các thực hiện FIFO khác nhau được tối ưu hóa cho các tham số khác nhau (tốc độ, diện tích, công suất hoặc chỉ là sự đơn giản).

11.3.2 Thiết kế lại FIFO Bubble

Chúng ta có thể định nghĩa lại FIFO bubble từ Phần 11.1 bằng cách sử dụng các giao tiếp sẵn-sàng-hợp-lệ tiêu chuẩn và có thể được tham số hóa với kiểu dữ liệu Chisel.

Listing 11.9 biểu diễn FIFO bubble đã được cấu trúc lại với giao tiếp sẵn-sàng-hợp-lệ. Lưu ý những gì chúng ta đặt thành phần `Buffer` bên trong từ `BubbleFifo` như là lớp riêng. Lớp trợ giúp này chỉ cần thiết cho thành phần này và do đó chúng ta ẩn nó đi và tránh làm ô nhiễm không gian tên. Lớp đệm cũng đã được đơn giản hóa. Thay vì FSM, chúng ta chỉ sử dụng một bit duy nhất, `fullReg`, để ghi nhớ trạng thái của bộ đệm: đầy hoặc trống.

FIFO bubble đơn giản, dễ hiểu và sử dụng tài nguyên tối thiểu. Tuy nhiên, vì mỗi tầng bộ đệm phải chuyển đổi giữa trống và đầy, băng thông tối đa của FIFO này là hai chu kỳ xung clock cho mỗi word.

Người ta có thể cân nhắc xem xét cả hai: bên giao tiếp trong bộ đệm để có thể chấp nhận một từ mới khi nhà sản xuất hợp lệ (`valid`) và người tiêu dùng sẵn sàng (`ready`). Tuy nhiên, điều này đưa ra một đường dẫn tổ hợp từ bắt tay của người tiêu dùng đến bắt tay của nhà sản xuất, điều này vi phạm ngữ nghĩa của giao tiếp sẵn-sàng-hợp-lệ.

11.3.3 FIFO bộ đệm kép

Một giải pháp là giữ trạng thái `ready` ngay cả khi thanh ghi bộ đệm nếu đầy. Để có thể chấp nhận một từ dữ liệu từ nhà sản xuất, khi người tiêu dùng không `ready`, chúng ta cần một bộ đệm thứ hai, gọi nó là thanh ghi bubble. Khi bộ đệm đầy, dữ liệu mới được lưu trữ trong thanh ghi bubble và tín hiệu `ready` bị hủy xác nhận. Khi người tiêu dùng trở lại `ready`, dữ liệu được truyền từ thanh ghi dữ liệu đến người tiêu dùng và từ thanh ghi bubble vào thanh ghi dữ liệu.

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

  private class DoubleBuffer[T <: Data](gen: T) extends Module {
    val io = IO(new FifoIO(gen))

    val empty :: one :: two :: Nil = Enum(3)
    val stateReg = RegInit(empty)
    val dataReg = Reg(gen)
```

```
class BubbleFifo[T <: Data](gen: T, depth: Int) extends
    Fifo(gen: T, depth: Int) {

  private class Buffer() extends Module {
    val io = IO(new FifoIO(gen))

    val fullReg = RegInit(false.B)
    val dataReg = Reg(gen)

    when (fullReg) {
      when (io.deq.ready) {
        fullReg := false.B
      }
    } .otherwise {
      when (io.enq.valid) {
        fullReg := true.B
        dataReg := io.enq.bits
      }
    }

    io.enq.ready := !fullReg
    io.deq.valid := fullReg
    io.deq.bits := dataReg
  }

  private val buffers = Array.fill(depth) { Module(new
      Buffer()) }
  for (i <- 0 until depth - 1) {
    buffers(i + 1).io.enq <> buffers(i).io.deq
  }

  io.enq <> buffers(0).io.enq
  io.deq <> buffers(depth - 1).io.deq
}
```

Listing 11.9: FIFO bubble với giao tiếp sẵn-sàng-hợp-lệ.

```
  val shadowReg = Reg(gen)

  switch(stateReg) {
    is (empty) {
      when (io.enq.valid) {
        stateReg := one
        dataReg := io.enq.bits
      }
    }
    is (one) {
      when (io.deq.ready && !io.enq.valid) {
        stateReg := empty
      }
      when (io.deq.ready && io.enq.valid) {
        stateReg := one
        dataReg := io.enq.bits
      }
      when (!io.deq.ready && io.enq.valid) {
        stateReg := two
        shadowReg := io.enq.bits
      }
    }
    is (two) {
      when (io.deq.ready) {
        dataReg := shadowReg
        stateReg := one
      }

    }
  }

  io.enq.ready := (stateReg === empty || stateReg === one)
  io.deq.valid := (stateReg === one || stateReg === two)
  io.deq.bits := dataReg
}

private val buffers = Array.fill((depth+1)/2) { Module(new
    DoubleBuffer(gen)) }

for (i <- 0 until (depth+1)/2 - 1) {
  buffers(i + 1).io.enq <> buffers(i).io.deq
}
```

```
  io.enq <> buffers(0).io.enq
  io.deq <> buffers((depth+1)/2 - 1).io.deq
}
```

Listing 11.10: FIFO với các thành phần bộ đệm kép.

Listing 11.10 biểu diễn bộ đệm kép. Mỗi thành phần bộ đệm có thể lưu trữ hai lối vào, chúng ta chỉ cần một nửa thành phần bộ đệm (`depth/2`). `DoubleBuffer` chứa hai thanh ghi, `dataReg` và `shadowReg`. Người tiêu dùng luôn được phục vụ từ `shadowReg`. Bộ đệm kép có ba trạng thái: `empty`, `one`, và `two`, báo hiệu mức lấp đầy của bộ đệm kép. Bộ đệm `ready` để chấp nhận dữ liệu mới khi nó ở trạng thái `empty` hoặc `one`. Dữ liệu hợp lệ khi nó ở trạng thái `one` hoặc `two`.

Nếu chúng ta chạy FIFO ở tốc độ tối đa và người tiêu dùng luôn `ready` thì trạng thái ổn định của bộ đệm kép là `one`. Chỉ khi người tiêu dùng hủy xác nhận `ready`, hàng đợi sẽ lấp đầy và bộ đệm nhập trạng thái `two`. Tuy nhiên, so với FIFO bubble đơn, việc khởi động lại hàng đợi chỉ mất một nửa số chu kỳ xung clock cho cùng dung lượng bộ đệm. Tương tự, độ trễ của FIFO bubble giảm xuống một nửa.

11.3.4 FIFO với bộ nhớ thanh ghi

Khi các bạn có nền tảng từ kỹ thuật phần mềm, các bạn có thể tự hỏi rằng chúng ta đã xây dựng hàng đợi phần cứng từ nhiều phần tử bộ đệm nhỏ riêng lẻ, tất cả đều thực thi song song và bắt tay với các phần tử luồng lên (upstream) và luồng xuống (downstream). Đối với các vùng đệm nhỏ, đây có lẽ là cách thực hiện hiệu quả nhất.

Hàng đợi trong phần mềm thường được sử dụng bởi một mã tuần tự trong một luồng duy nhất. Hoặc như một hàng đợi để tách nhà sản xuất và người tiêu dùng. Trong cài đặt này, hàng đợi FIFO có kích thước cố định thường được thực hiện dưới dạng bộ đệm vòng. Hai con trỏ trỏ vào vị trí đọc và ghi trong tập bộ nhớ riêng cho hàng đợi. Khi các con trỏ đến cuối bộ nhớ, con trỏ được đặt trở lại điểm bắt đầu của bộ nhớ đó. Sự khác biệt giữa hai con trỏ là số phần tử trong hàng đợi. Khi hai con trỏ trỏ đến cùng một địa chỉ, hàng đợi trống hoặc đầy. Để phân biệt giữa trống và đầy, chúng ta cần một cờ khác.

Chúng ta cũng có thể thực hiện hàng đợi FIFO dựa trên bộ nhớ trên phần cứng. Đối với các hàng đợi nhỏ, chúng ta có thể sử dụng tập tin thanh ghi (tức là `Reg(Vec())`). Listing 11.11 biểu diễn một hàng đợi FIFO được thực hiện với bộ nhớ và các con trỏ đọc và ghi.

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

  def counter(depth: Int, incr: Bool): (UInt, UInt) = {
```

```
    val cntReg = RegInit(0.U(log2Ceil(depth).W))
    val nextVal = Mux(cntReg === (depth-1).U, 0.U, cntReg + 1.U)
    when (incr) {
      cntReg := nextVal
    }
    (cntReg, nextVal)
  }

  // the register based memory
  val memReg = Reg(Vec(depth, gen))

  val incrRead = WireDefault(false.B)
  val incrWrite = WireDefault(false.B)
  val (readPtr, nextRead) = counter(depth, incrRead)
  val (writePtr, nextWrite) = counter(depth, incrWrite)

  val emptyReg = RegInit(true.B)
  val fullReg = RegInit(false.B)

  when (io.enq.valid && !fullReg) {
    memReg(writePtr) := io.enq.bits
    emptyReg := false.B
    fullReg := nextWrite === readPtr
    incrWrite := true.B
  }

  when (io.deq.ready && !emptyReg) {
    fullReg := false.B
    emptyReg := nextRead === writePtr
    incrRead := true.B
  }

  io.deq.bits := memReg(readPtr)
  io.enq.ready := !fullReg
  io.deq.valid := !emptyReg
}
```

Listing 11.11: FIFO với bộ nhớ dựa trên thanh ghi.

Vì có hai con trỏ hoạt động giống nhau, giá trị được tăng lên trên một hoạt động và được bao quanh ở cuối bộ đệm, nên chúng ta định nghĩa một hàm `counter` thực hiện các mạch đếm gói đó. Với `log2Ceil(depth).W`, chúng ta tính toán độ dài bit của mạch đếm.

Giá trị tiếp theo được tăng lên 1 hoặc bao quanh 0. Mạch đếm chỉ được tăng lên khi ngõ vào `incr` là `true.B`.

Hơn nữa, vì chúng ta cũng cần giá trị tiếp theo có thể có (tăng lên hoặc quanh 0), nên chúng ta cũng trả về giá trị này từ hàm `counter`. Trong Scala, chúng ta có thể trả về cái gọi là *bộ giá trị*, đơn giản là một vùng chứa để giữ nhiều hơn một giá trị. Cú pháp để tạo một bộ giá trị như vậy chỉ đơn giản là gói các giá trị được phân tách bằng dấu phẩy trong dấu ngoặc đơn:

```
  val t = (v1, v2)
```

Chúng ta có thể giải cấu trúc một bộ như vậy bằng cách sử dụng ký hiệu dấu ngoặc đơn ở phía bên trái của phép gán:

```
val (x1, x2) = t
```

Đối với bộ nhớ, chúng ta sử dụng một thanh ghi véc-tơ (`Reg(Vec(depth, gen))`) của kiểu dữ liệu Chisel `gen`. Chúng ta định nghĩa hai tín hiệu để tăng con trỏ đọc và ghi, và tạo ra các con trỏ đọc và ghi bằng hàm `counter`. Khi cả hai con trỏ bằng nhau, vùng đệm trống hoặc đầy. Chúng ta định nghĩa hai cờ cho khái niệm trống và đầy.

Khi nhà sản xuất xác nhận `valid` và FIFO không đầy, chúng ta: (1) ghi vào bộ đệm, (2) đảm bảo `blankReg` bị hủy xác nhận, (3) đánh dấu bộ đệm đầy nếu con trỏ ghi bắt kịp với con trỏ đọc trong chu kỳ xung clock tiếp theo (so sánh con trỏ đọc hiện tại với con trỏ ghi tiếp theo) và (4) báo hiệu mạch đếm ghi tăng lên.

Khi người tiêu dùng `ready` và FIFO không trống, chúng ta: (1) đảm bảo rằng `fullReg` bị hủy xác nhận, (2) đánh dấu bộ đệm trống nếu con trỏ đọc bắt kịp với con trỏ ghi trong chu kỳ xung clock tiếp theo và (3) báo hiệu cho mạch đếm đọc tăng lên.

Ngõ ra của FIFO là phần tử bộ nhớ tại địa chỉ con trỏ đọc. Các cờ sẵn sàng và hợp lệ chỉ đơn giản là bắt nguồn từ các cờ đầy và trống.

11.3.5 FIFO với bộ nhớ trên chip

Phiên bản cuối cùng của FIFO sử dụng các tập tin thanh ghi để biểu diễn bộ nhớ, đây là một giải pháp tốt cho một FIFO nhỏ. Với FIFO lớn hơn, tốt hơn là sử dụng bộ nhớ trên chip. Listing 11.12 biểu diễn một FIFO sử dụng bộ nhớ đồng bộ để lưu trữ.

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

  def counter(depth: Int, incr: Bool): (UInt, UInt) = {
    val cntReg = RegInit(0.U(log2Ceil(depth).W))
    val nextVal = Mux(cntReg === (depth-1).U, 0.U, cntReg + 1.U)
```

```
  when (incr) {
    cntReg := nextVal
  }
  (cntReg, nextVal)
}

val mem = SyncReadMem(depth, gen)

val incrRead = WireDefault(false.B)
val incrWrite = WireDefault(false.B)
val (readPtr, nextRead) = counter(depth, incrRead)
val (writePtr, nextWrite) = counter(depth, incrWrite)

val emptyReg = RegInit(true.B)
val fullReg = RegInit(false.B)

val idle :: valid :: full :: Nil = Enum(3)
val stateReg = RegInit(idle)
val shadowReg = Reg(gen)

when (io.enq.valid && !fullReg) {
  mem.write(writePtr, io.enq.bits)
  emptyReg := false.B
  fullReg := nextWrite === readPtr
  incrWrite := true.B
}

val data = mem.read(readPtr)

// Handling of the one cycle memory latency
// with an additional output register
switch(stateReg) {
  is(idle) {
    when(!emptyReg) {
      stateReg := valid
      fullReg := false.B
      emptyReg := nextRead === writePtr
      incrRead := true.B
    }
  }
  is(valid) {
    when(io.deq.ready) {
```

```
        when(!emptyReg) {
          stateReg := valid
          fullReg := false.B
          emptyReg := nextRead === writePtr
          incrRead := true.B
        } otherwise {
          stateReg := idle
        }
      } otherwise {
        shadowReg := data
        stateReg := full
      }

    }
    is(full) {
      when(io.deq.ready) {
        when(!emptyReg) {
          stateReg := valid
          fullReg := false.B
          emptyReg := nextRead === writePtr
          incrRead := true.B
        } otherwise {
          stateReg := idle
        }

      }
    }
  }

  io.deq.bits :=  Mux(stateReg === valid, data, shadowReg)
  io.enq.ready := !fullReg
  io.deq.valid := stateReg === valid || stateReg === full
}
```

Listing 11.12: FIFO với bộ nhớ trên chip.

Việc xử lý con trỏ đọc và ghi giống như FIFO bộ nhớ thanh ghi. Tuy nhiên, một bộ nhớ trên chip đồng bộ đưa ra kết quả của việc đọc trong chu kỳ xung clock tiếp theo, nơi việc đọc tập tin thanh ghi có sẵn trong cùng chu kỳ xung clock.

Do đó, chúng ta cần một số FSM bổ sung và một thanh ghi bubble để xử lý độ trễ này. Chúng ta đọc bộ nhớ ra và cung cấp giá trị của đầu hàng đợi cho cổng ngõ ra. Nếu giá trị đó không được sử dụng, chúng ta cần lưu trữ nó trong thanh ghi bubble `shadowReg`

```
class CombFifo[T <: Data](gen: T, depth: Int) extends Fifo(gen:
    T, depth: Int) {

  val memFifo = Module(new MemFifo(gen, depth))
  val bufferFIFO = Module(new DoubleBufferFifo(gen, 2))
  io.enq <> memFifo.io.enq
  memFifo.io.deq <> bufferFIFO.io.enq
  bufferFIFO.io.deq <> io.deq
}
```

Listing 11.13: Kết hợp bộ nhớ dựa trên FIFO với tầng bộ đệm kép.

trong khi đọc giá trị tiếp theo từ bộ nhớ. Máy trạng thái bao gồm ba trạng thái để biểu diễn: (1) FIFO trống, (2) dữ liệu hợp lệ được đọc ra từ bộ nhớ và (3) đầu hàng đợi trong thanh ghi bubble và dữ liệu hợp lệ (phần tử tiếp theo) từ bộ nhớ.

FIFO dựa trên bộ nhớ có thể giữ một cách hiệu quả lượng dữ liệu lớn hơn trong hàng đợi và có độ trễ giảm nhanh. Trong thiết kế cuối cùng, ngõ ra của FIFO có thể đến trực tiếp từ viêc đọc bộ nhớ. Nếu đường dẫn dữ liệu này nằm trong đường dẫn giới hạn của thiết kế, chúng ta có thể dễ dàng thực hiện kỹ thuật đường ống thiết kế của mình bằng cách kết hợp hai FIFO. Listing 11.13 biểu diễn một sự kết hợp như vậy. Ở ngõ ra của FIFO dựa trên bộ nhớ, chúng ta thêm một FIFO bộ đệm kép một tầng để tách đường đọc bộ nhớ khỏi ngõ ra.

11.4 Bài tập

Phần bài tập này dài hơn một chút vì nó bao gồm hai bài tập: (1) khám phá FIFO bubble và thực hiện một thiết kế FIFO khác; và (2) khám phá UART và mở rộng nó. Mã nguồn cho cả hai bài tập được chứa trong kho lưu trữ chisel-examples.

11.4.1 Khám phá FIFO Bubble

Nguồn FIFO cũng bao gồm một trình kiểm tra kích hoạt hành vi đọc và ghi khác nhau và tạo ra một dạng sóng ở dạng VCD (Value Change Dump). Tập tin VCD có thể được xem bởi một chương trình xem dạng sóng, chẳng hạn như GTKWave. Khám phá FifoTester trong kho lưu trữ FifoTester. Kho lưu trữ chứa tập tin `Makefile` để chạy các ví dụ, với ví dụ FIFO, chỉ cần gõ:

```
$ make fifo
```

Lệnh make này sẽ biên dịch FIFO, chạy kiểm tra và khởi động GTKWave để xem dạng sóng. Khám phá trình kiểm tra và dạng sóng đã được tạo ra.

Trong các chu kỳ đầu tiên, trình kiểm tra viết một từ đơn. Chúng ta có thể quan sát dưới dạng sóng làm thế nào một word được thả nổi (bubble) qua FIFO, do đó có tên là *FIFO bubble*. Việc thả nổi này cũng có nghĩa là độ trễ của một word dữ liệu qua FIFO bằng với độ sâu của FIFO.

Bài kiểm tra tiếp theo lấp dữ liệu vào FIFO đến khi nó đầy. Sau đó là tiến hành đọc đơn. Lưu ý cách các word trống bubble từ phía bên đọc của FIFO sang phía bên ghi. Khi FIFO bubble đầy, nó sẽ lấy độ trễ của độ sâu bộ đệm để đọc làm ảnh hưởng đến phía bên ghi.

Phần cuối của bài kiểm tra chứa một vòng lặp nhằm nỗ lực ghi và đọc ở tốc độ tối đa. Chúng ta có thể thấy FIFO bubble chạy ở băng thông tối đa, là hai chu kỳ đồng hồ cho mỗi word. Tầng đệm luôn đổi trạng thái giữa trống và đầy cho một lần chuyển một word đơn.

FIFO bubble đơn giản và đối với các bộ đệm nhỏ có yêu cầu tài nguyên thấp. Hạn chế chính của FIFO bubble *n* tầng là: (1) thông lượng tối đa là một word sau mỗi hai chu kỳ xung clock, (2) một word dữ liệu phải di chuyển *n* chu kỳ xung clock từ cuối bên ghi đến đầu bên đọc, và (3) FIFO đầy đủ cần *n* chu kỳ xung clock để khởi động lại.

Những hạn chế này có thể được giải quyết bằng cách thực hiện FIFO với bộ đệm vòng. Bộ đệm vòng có thể được thực hiện với một bộ nhớ và các con trỏ đọc và ghi. Thực hiện một FIFO dưới dạng bộ đệm vòng có bốn phần tử, sử dụng cùng một giao tiếp và khám phá các hành vi khác nhau với trình kiểm tra. Với thực hiện ban đầu của việc sử dụng bộ đệm vòng, như một lối tắt, dùng một véc-tơ của các thanh ghi (`Reg(Vec(4, UInt(size.W)))`).

11.4.2 UART

Với ví dụ UART, các bạn cần một bo mạch FPGA có cổng nối tiếp và cổng nối tiếp cho máy tính của bạn (thường qua kết nối USB). Kết nối cáp nối tiếp giữa bo mạch FPGA và cổng nối tiếp trên máy tính của các bạn. Chạy chương trình terminal, ví dụ: Hyperterm trên Windows hoặc `gtkterm` trên Linux:

```
$ gtkterm &
```

Cấu hình cổng để sử dụng đúng thiết bị, với UART USB, cổng này thường là `/dev/ttyUSB0`. Đặt tốc độ baud là 115200 và không có bit chẵn lẻ (parity) hoặc điều khiển luồng (bắt tay). Với lệnh sau, các bạn có thể tạo mã Verilog cho UART:

```
$ make uart
```

Sau đó, sử dụng công cụ tổng hợp mạch để tổng hợp thiết kế. Kho lưu trữ chứa một dự án Quartus cho bo mạch DE2-115 FPGA. Với Quartus, sử dụng nút Play để tổng hợp thiết kế và sau đó cấu hình FPGA. Sau khi cấu hình, các bạn sẽ thấy một thông báo chúc mừng thành công trong terminal.

Mở rộng ví dụ về đèn LED nhấp nháy bằng UART, và ghi 0 và 1 vào dòng nối tiếp khi đèn LED tắt và bật. Sử dụng `BufferedTx`, như trong ví dụ `Sender`.

Với ngõ ra chậm của các ký tự (hai ký tự mỗi giây), các bạn có thể ghi dữ liệu vào thanh ghi truyền UART và có thể bỏ qua bắt tay đọc/hợp lệ. Mở rộng ví dụ bằng cách viết các số lặp lại từ 0-9 nhanh như tốc độ baud cho phép. Trong trường hợp này, các bạn phải mở rộng máy trạng thái của mình để thăm dò trạng thái UART để kiểm tra xem bộ đệm truyền có trống hay không.

Mã code ví dụ chỉ chứa một bộ đệm đơn cho `Tx`. Vui lòng thêm FIFO mà các bạn đã thực hiện để thêm bộ đệm cho bộ phát và bộ thu.

11.4.3 Khám phá FIFO

Viết một FIFO đơn giản với 4 phần tử bộ đệm trong các thanh ghi chuyên dụng. Sử dụng các mạch đếm đọc và ghi 2-bit, có thể chỉ làm tràn. Để đơn giản hóa hơn nữa, hãy xem xét tình huống khi các con trỏ đọc và ghi bằng với FIFO trống. Điều này có nghĩa là các bạn có thể lưu trữ tối đa 3 phần tử. Việc đơn giản hóa này sẽ tránh dùng hàm mạch đếm trong ví dụ ở Listing 11.11 và xử lý trạng thái trống hoặc đầy với cùng giá trị con trỏ. Chúng ta không cần cờ báo trống hoặc cờ báo đầy, vì điều này có thể được dẫn xuất từ các giá trị con trỏ một mình. Thiết kế này đơn giản hơn bao nhiêu?

Các thiết kế FIFO khác nhau đã được trình bày có sự cân bằng thiết kế khác nhau liên quan đến các thuộc tính sau: (1) thông lượng tối đa, (2) giảm độ trễ, (3) yêu cầu tài nguyên và (4) tần số xung clock cực đại. Khám phá tất cả các biến thể FIFO ở các kích thước khác nhau bằng cách tổng hợp chúng cho một FPGA; mã nguồn có sẵn tại chisel-examples. Đâu là đặc điểm của các FIFO có 4-word, 16-word và 256-word?

12 Thiết kế bộ xử lý

Là một trong những chương cuối của cuốn sách này, chúng tôi trình bày một dự án cỡ trung bình: thiết kế, mô phỏng và kiểm tra một bộ vi xử lý. Để quản lý dự án này, chúng ta thiết kế một bộ tích lũy đơn giản. Bộ xử lý có tên là Leros [8] và có sẵn ở dạng mã nguồn mở tại https://github.com/leros-dev/leros. Chúng tôi muốn đề cập đây là một ví dụ nâng cao và cần có một số kiến thức về kiến trúc máy tính để làm theo các ví dụ mã code đã trình bày.

Leros được thiết kế đơn giản, nhưng vẫn là mục tiêu tốt cho trình biên dịch C. Mô tả của tập lệnh nằm gọn trong một trang, xem Bảng 12.1. Trong bảng đó `A` biểu diễn cho bộ tích lũy, `PC` là bộ đếm chương trình, `i` là giá trị tức thời (0 đến 255), thanh ghi `Rn` (với `n` từ 0 đến 255), `o` là độ lệch nhánh tương đối so với `PC` và thanh ghi địa chỉ `AR` để truy cập bộ nhớ.

12.1 Bắt đầu với ALU

Thành phần trung tâm của một bộ xử lý là đơn vị logic số học, hay ghi tắt là ALU (Arithmetic Logic Unit). Vì vậy, chúng ta bắt đầu với việc lập trình cho ALU và testbench. Đầu tiên, chúng ta định nghĩa `Enum` để biểu diễn các phép toán khác nhau của ALU:

```
object Types {
  val nop :: add :: sub :: and :: or :: xor :: ld :: shr :: Nil
      = Enum(8)
}
```

Một ALU luôn có hai toán hạng ngõ vào (gọi là `a` và `b`), ngõ vào `op` (hoặc opcode) để chọn chức năng (phép toán) và một ngõ ra `y`. Listing 12.1 trình bày một ALU.

Đầu tiên chúng ta định nghĩa tên ngắn hơn cho ba ngõ vào. Câu lệnh `switch` định nghĩa logic để tính toán `res`. Vì vậy, nó được gán mặc định là 0. Câu lệnh switch liệt kê tất cả các phép toán và gán biểu thức cho phù hợp. Tất cả các hoạt động ánh xạ trực tiếp đến một biểu thức Chisel. Cuối cùng, chúng ta gán kết quả `res` cho ngõ ra `y` của ALU.

Để kiểm tra, chúng ta viết hàm ALU ở dạng Scala đơn giản, như trình bày trong Listing 12.2.

Mã lệnh	Chức năng	Mô tả
add	A = A + Rn	Cộng thanh ghi Rn và A
addi	A = A + i	Cộng giá trị tức thời i và A
sub	A = A - Rn	Trừ thanh ghi Rn với A
subi	A = A - i	Trừ giá trị tức thời i với A
shr	A = A >>> 1	Dịch phải (logic) A
load	A = Rn	Nạp thanh ghi Rn vào A
loadi	A = i	Nạp giá trị tức thời i vào A
and	A = A and Rn	AND thanh ghi Rn với A
andi	A = A and i	AND giá trị tức thời i với A
or	A = A or Rn	OR thanh ghi Rn với A
ori	A = A or i	OR giá trị tức thời i với A
xor	A = A xor Rn	XOR thanh ghi Rn với A
xori	A = A xor i	XOR giá trị tức thời i với A
loadhi	A_{15-8} = i	Nạp tức thời vào byte thứ hai
loadh2i	A_{23-16} = i	Nạp tức thời vào byte thứ ba
loadh3i	A_{31-24} = i	Nạp tức thời vào byte thứ tư
store	Rn = A	Lưu A vào thanh ghi Rn
jal	PC = A, Rn = PC + 2	Nhảy tới A và lưu địa chỉ trả về trong Rn
ldaddr	AR = A	Nạp thanh ghi địa chỉ AR với A
loadind	A = mem[AR+(i << 2)]	Nạp một word từ bộ nhớ vào A
loadindbu	A = mem[AR+i]$_{7-0}$	Nạp một byte không dấu từ bộ nhớ vào A
storeind	mem[AR+(i << 2)] = A	Lưu A vào bộ nhớ
storeindb	mem[AR+i]$_{7-0}$ = A	Lưu một byte vào bộ nhớ
br	PC = PC + o	Rẽ nhánh
brz	if A == 0 PC = PC + o	Rẽ nhánh nếu A bằng zero
brnz	if A != 0 PC = PC + o	Rẽ nhánh nếu A khác zero
brp	if A >= 0 PC = PC + o	Rẽ nhánh nếu A là số dương
brn	if A < 0 PC = PC + o	Rẽ nhánh nếu A là số âm
scall	scall A	Gọi hệ thống (simulation hook)

Bảng 12.1: Tập lệnh của Leros.

```
class Alu(size: Int) extends Module {
  val io = IO(new Bundle {
    val op = Input(UInt(3.W))
    val a = Input(SInt(size.W))
    val b = Input(SInt(size.W))
    val y = Output(SInt(size.W))
  })

  val op = io.op
  val a = io.a
  val b = io.b
  val res = WireDefault(0.S(size.W))

  switch(op) {
    is(add) {
      res := a + b
    }
    is(sub) {
      res := a - b
    }
    is(and) {
      res := a & b
    }
    is(or) {
      res := a | b
    }
    is(xor) {
      res := a ^ b
    }
    is (shr) {
      // the following does NOT result in an unsigned shift
      // res := (a.asUInt >> 1).asSInt
      // work around
      res := (a >> 1) & 0x7fffffff.S
    }
    is(ld) {
      res := b
    }
  }

  io.y := res
}
```

Listing 12.1: ALU của Leros.

```
def alu(a: Int, b: Int, op: Int): Int = {

  op match {
    case 1 => a + b
    case 2 => a - b
    case 3 => a & b
    case 4 => a | b
    case 5 => a ^ b
    case 6 => b
    case 7 => a >>> 1
    case _ => -123 // This shall not happen
  }
}
```

Listing 12.2: Hàm ALU Leros được viết bằng Scala.

Dù bản sao phần cứng này, được viết bằng Chisel bởi một triển khai Scala, không phát hiện ra lỗi trong đặc tả; nhưng ít nhất một số kiểm tra đúng đắn cần thực hiện. Chúng ta sử dụng một số giá trị góc làm véc-tơ kiểm tra (test vector):

```
// Some interesting corner cases
val interesting = Array(1, 2, 4, 123, 0, -1, -2, 0x80000000,
    0x7fffffff)
```

Chúng ta kiểm tra tất cả các chức năng với các giá trị ở cả hai ngõ vào:

```
def test(values: Seq[Int]) = {
  for (fun <- add to shr) {
    for (a <- values) {
      for (b <- values) {
        poke(dut.io.op, fun)
        poke(dut.io.a, a)
        poke(dut.io.b, b)
        step(1)
        expect(dut.io.y, alu(a, b, fun.toInt))
      }
    }
  }
}
```

Việc không thể kiểm tra toàn bộ, đầy đủ đối với các đối số 32-bit là lý do chúng tôi chọn một số trường hợp giá trị góc làm giá trị ngõ vào. Bên cạnh việc kiểm tra các trường hợp góc, việc kiểm tra các ngõ vào ngẫu nhiên cũng rất hữu ích:

```
  val randArgs = Seq.fill(100)(scala.util.Random.nextInt)
  test(randArgs)
```

Các bạn có thể chạy các bài kiểm tra trong dự án Leros với lệnh:

```
$ sbt "test:runMain leros.AluTester"
```

và sẽ tạo ra một thông báo thành công tương tự như sau:

```
[info] [0.001] SEED 1544507337402
test Alu Success: 70567 tests passed in 70572 cycles taking
3.845715 seconds
[info] [3.825] RAN 70567 CYCLES PASSED
```

12.2 Giải mã lệnh

Từ ALU, chúng ta làm ngược lại và thực hiện bộ giải mã lệnh. Tuy nhiên, đầu tiên, chúng ta định nghĩa mã hóa lệnh trong lớp Scala của riêng nó và gói *chia sẻ*. Chúng ta muốn chia sẻ các hằng số mã hóa giữa việc triển khai phần cứng của Leros, một trình hợp dịch cho Leros và một trình mô phỏng tập lệnh của Leros.

```
package leros.shared {

object Constants {
  val NOP = 0x00
  val ADD = 0x08
  val ADDI = 0x09
  val SUB = 0x0c
  val SUBI = 0x0d
  val SHR = 0x10
  val LD = 0x20
  val LDI = 0x21
  val AND = 0x22
  val ANDI = 0x23
  val OR = 0x24
  val ORI = 0x25
  val XOR = 0x26
```

```
  val XORI = 0x27
  val LDHI = 0x29
  val LDH2I = 0x2a
  val LDH3I = 0x2b
  val ST = 0x30
  // ...
```

Đối với thành phần giải mã, chúng ta định nghĩa một `Bundle` cho ngõ ra, sau này được đưa một phần vào ALU.

```
class DecodeOut extends Bundle {
  val ena = Bool()
  val func = UInt()
  val exit = Bool()
}
```

Giải mã lấy ngõ vào là mã lệnh 8-bit và chuyển các tín hiệu đã giải mã dưới dạng ngõ ra. Các tín hiệu điều khiển đó được gán một giá trị mặc định với `WireDefault`.

```
class Decode() extends Module {
  val io = IO(new Bundle {
    val din = Input(UInt(8.W))
    val dout = Output(new DecodeOut)
  })

  val f = WireDefault(nop)
  val imm = WireDefault(false.B)
  val ena = WireDefault(false.B)

  io.dout.exit := false.B
```

Bản thân việc giải mã chỉ là dùng cú pháp switch trên một phần của lệnh nhằm biểu diễn mã lệnh (trong Leros, hầu hết các lệnh là 8-bit trên.)

```
  switch(io.din) {
    is(ADD.U) {
      f := add
      ena := true.B
    }
    is(ADDI.U) {
      f := add
      imm := true.B
      ena := true.B
```

```
  }
  is(SUB.U) {
    f := sub
    ena := true.B
  }
  is(SUBI.U) {
    f := sub
    imm := true.B
    ena := true.B
  }
  is(SHR.U) {
    f := shr
    ena := true.B
  }
  // ...
```

12.3 Lệnh hợp ngữ

Để viết chương trình cho Leros, chúng ta cần một trình hợp dịch. Tuy nhiên, đối với thử nghiệm đầu tiên, chúng ta có thể viết mã cứng một vài lệnh và đặt chúng vào một mảng Scala, mảng mà chúng ta sử dụng để khởi tạo bộ nhớ lệnh.

```
val prog = Array[Int](
  0x0903, // addi 0x3
  0x09ff, // -1
  0x0d02, // subi 2
  0x21ab, // ldi 0xab
  0x230f, // and 0x0f
  0x25c3, // or 0xc3
  0x0000
)

def getProgramFix() = prog
```

Tuy nhiên, đây là một cách tiếp cận rất kém hiệu quả để kiểm tra một bộ xử lý. Viết một trình hợp dịch với ngôn ngữ biểu diễn như Scala không phải là một dự án lớn. Do đó, chúng ta viết một trình hợp ngữ đơn giản cho Leros, có thể trong khoảng 100 dòng mã. Chúng ta định nghĩa một hàm `getProgram` gọi trình hợp dịch. Đối với các điểm đích rẽ nhánh, chúng ta cần một bảng ký hiệu, bảng này chúng ta thu thập trong một `Map`. Một trình hợp dịch cổ điển chạy hai lần: (1) thu thập các giá trị cho bảng ký hiệu và (2) ráp

chương trình với các ký hiệu được thu thập trong lần chạy đầu tiên. Do đó, chúng ta gọi hàm `assemble` hai lần với một tham số để cho biết đó là lần chạy nào.

```
def getProgram(prog: String) = {
  assemble(prog)
}

// collect destination addresses in first pass
val symbols = collection.mutable.Map[String, Int]()

def assemble(prog: String): Array[Int] = {
  assemble(prog, false)
  assemble(prog, true)
}
```

Hàm `assemble` bắt đầu với việc đọc trong tập tin nguồn[1] và định nghĩa hai hàm trợ giúp để phân tích cú pháp hai toán hạng có thể gồm: (1) một hằng số nguyên (cho phép ký hiệu thập phân hoặc thập lục phân) và (2) để đọc số thanh ghi.

```
def assemble(prog: String, pass2: Boolean): Array[Int] = {

  val source = Source.fromFile(prog)
  var program = List[Int]()
  var pc = 0

  def toInt(s: String): Int = {
    if (s.startsWith("0x")) {
      Integer.parseInt(s.substring(2), 16)
    } else {
      Integer.parseInt(s)
    }
  }

  def regNumber(s: String): Int = {
    assert(s.startsWith("r"), "Register numbers shall start
        with \'r\'")
    s.substring(1).toInt
  }
```

Listing 12.3 trình bày lõi của trình hợp dịch cho Leros. Biểu thức `match`trong Scala

[1] Hàm này không thực sự đọc tập tin nguồn, nhưng đối với cuộc thảo luận này, chúng ta có thể coi nó là hàm đọc.

bao hàm lõi của hàm hợp ngữ.

12.4 Bài tập

Bài tập này thuộc một trong những chương cuối cùng ở dạng rất tự do. Các bạn đã kết thúc chuyến tham quan học hỏi của mình thông qua Chisel và sẵn sàng giải quyết các vấn đề thiết kế mà bạn thấy hứng thú.

Một tùy chọn là đọc lại chương này và đọc cùng với tất cả mã nguồn trong kho lưu trữ Leros, chạy các trường hợp kiểm tra, vọc với mã code bằng cách ngừng nó và thấy rằng các bài kiểm tra không chạy.

Một tùy chọn khác là viết chương trình thực hiện Leros của các bạn. Việc thực hiện trong kho lưu trữ chỉ là một phương án tổ chức đường ống khả dĩ. Các bạn có thể viết một phiên bản mô phỏng Chisel của Leros chỉ với một tầng đường ống duy nhất, hoặc Leros siêu đường ống để có tần suất xung clock cao nhất có thể.

Tùy chọn thứ ba là thiết kế bộ xử lý của các bạn ngay từ đầu. Có thể phần trình diễn về cách xây dựng bộ xử lý Leros và các công cụ cần thiết đã thuyết phục các bạn rằng thiết kế và thực hiện bộ xử lý không phải là trò ảo thuật, mà là kỹ thuật có thể mang lại nhiều sự hứng thú.

```
    for (line <- source.getLines()) {
      if (!pass2) println(line)
      val tokens = line.trim.split(" ")
      val Pattern = "(.*:)".r
      val instr = tokens(0) match {
        case "//" => // comment
        case Pattern(l) => if (!pass2) symbols +=
            (l.substring(0, l.length - 1) -> pc)
        case "add" => (ADD << 8) + regNumber(tokens(1))
        case "sub" => (SUB << 8) + regNumber(tokens(1))
        case "and" => (AND << 8) + regNumber(tokens(1))
        case "or" => (OR << 8) + regNumber(tokens(1))
        case "xor" => (XOR << 8) + regNumber(tokens(1))
        case "load" => (LD << 8) + regNumber(tokens(1))
        case "addi" => (ADDI << 8) + toInt(tokens(1))
        case "subi" => (SUBI << 8) + toInt(tokens(1))
        case "andi" => (ANDI << 8) + toInt(tokens(1))
        case "ori" => (ORI << 8) + toInt(tokens(1))
        case "xori" => (XORI << 8) + toInt(tokens(1))
        case "shr" => (SHR << 8)
        case "loadi" => (LDI << 8) + toInt(tokens(1))
        case "loadhi" => (LDHI << 8) + toInt(tokens(1))
        case "loadh2i" => (LDH2I << 8) + toInt(tokens(1))
        case "loadh3i" => (LDH3I << 8) + toInt(tokens(1))
        case "store" => (ST << 8) + regNumber(tokens(1))
        case "ldaddr" => (LDADDR << 8)
        case "ldind" => (LDIND << 8) + toInt(tokens(1))
        case "ldindbu" => (LDINDBU << 8) + toInt(tokens(1))
        case "stind" => (STIND << 8) + toInt(tokens(1))
        case "stindb" => (STINDB << 8) + toInt(tokens(1))
        case "br" => (BR << 8) + (if (pass2) symbols(tokens(1))
            else 0)
        case "brz" => (BRZ << 8) + (if (pass2)
            symbols(tokens(1)) else 0)
        case "brnz" => (BRNZ << 8) + (if (pass2)
            symbols(tokens(1)) else 0)
        case "brp" => (BRP << 8) + (if (pass2)
            symbols(tokens(1)) else 0)
        case "brn" => (BRN << 8) + (if (pass2)
            symbols(tokens(1)) else 0)
        case "in" => (IN << 8) + toInt(tokens(1))
        case "out" => (OUT << 8) + toInt(tokens(1))
        case "scall" => (SCALL << 8) + toInt(tokens(1))
        // ...
        case "" => // println("Empty line")
        case t: String => throw new Exception("Assembler error:
            unknown instruction: " + t)
        case _ => throw new Exception("Assembler error")
```

13 Đóng góp cho Chisel

Chisel là một dự án mã nguồn mở đang được phát triển và cải tiến liên tục. Do đó, các bạn cũng có thể đóng góp cho dự án. Ở đây chúng tôi mô tả cách thiết lập môi trường để phát triển thư viện Chisel và cách đóng góp cho Chisel.

13.1 Thiết lập môi trường phát triển

Chisel bao gồm nhiều kho chứa khác nhau; tất cả được lưu trữ tại dự án chip miễn phí trên GitHub.

Phân nhánh kho lưu trữ mà các bạn muốn đóng góp vào tài khoản GitHub cá nhân của bạn. Các bạn có thể phân nhánh kho lưu trữ bằng cách nhấn nút `Fork` trên giao diện web GitHub. Sau đó, từ phân nhánh đó, hãy sao chép phân nhánh của các bạn trong kho lưu trữ[1]. Trong ví dụ, chúng ta thay đổi `chisel3`, và lệnh sao chép cho phân nhánh cục bộ như sau:

```
$ git clone git@github.com:schoeberl/chisel3.git
```

Để biên dịch Chisel 3 và xuất bản (publish) dưới dạng thực thi thư viện cục bộ:

```
$ cd chisel3
$ sbt compile
$ sbt publishLocal
```

Chú ý trong khi xuất bản, lệnh cục bộ cho chuỗi phiên bản của thư viện đã xuất bản, chứa chuỗi `SNAPSHOT`. Nếu các bạn sử dụng trình kiểm tra và phiên bản đã được xuất bản không tương thích với SNAPSHOT của Chisel, hãy phân nhánh và sao chép kho chứa chisel-tester và xuất bản nó cục bộ.

Để kiểm tra những thay đổi của mình trong Chisel, các bạn có thể cũng muốn thiết lập dự án Chisel, ví dụ: bằng cách phân nhánh/sao chép một dự án Chisel trống, đổi tên nó, và xóa thư mục `.git` khỏi nó.

Thay đổi `build.sbt` để tham chiếu đến phiên bản Chisel được xuất bản cục bộ. Hơn nữa, tại thời điểm viết bài này, người đứng đầu mã nguồn Chisel sử dụng Scala phiên

[1] Lưu ý rằng đối với một sự thay đổi trong firrtl/Chisel, các bạn cũng cần phải phân nhánh và sao chép firrtl

bản 2.12, nhưng Scala 2.12 gặp sự cố với các gói ẩn danh. Do đó, các bạn cần thêm tùy chọn Scala sau: `"- Xsource: 2.11"`. `build.sbt` sẽ trông tương tự như sau:

```
scalaVersion := "2.12.6"

scalacOptions := Seq("-Xsource:2.11")

resolvers ++= Seq(
  Resolver.sonatypeRepo("snapshots"),
  Resolver.sonatypeRepo("releases")
)

libraryDependencies +=
  "edu.berkeley.cs" %% "chisel3" % "3.2-SNAPSHOT"
libraryDependencies +=
  "edu.berkeley.cs" %% "chisel-iotesters" % "1.3-SNAPSHOT"
```

Biên dịch ứng dụng kiểm tra Chisel của các bạn và xem xét kỹ nếu nó chọn phiên bản được xuất bản cục bộ của thư viện Chisel hay không (cũng có phiên bản SNAPSHOT được xuất bản, vì vậy, nếu phiên bản Scala khác biệt giữa thư viện Chisel và mã ứng dụng của các bạn, thì nó chọn phiên bản SNAPSHOT từ máy chủ thay vì thư viện đã được xuất bản cục bộ của các bạn.)

Xem thêm một số ghi chú tại kho chứa Chisel.

13.2 Kiểm tra

Khi các bạn thay đổi thư viện Chisel, các bạn nên chạy các bài kiểm tra Chisel. Trong dự án dựa trên `sbt`, nó thường được chạy với lệnh sau:

```
$ sbt test
```

Hơn nữa, nếu các bạn thêm chức năng vào, thì các bạn cũng nên cung cấp các bài kiểm tra cho các đặc điểm mới.

13.3 Đóng góp với Pull Request

Trong dự án Chisel, không có nhà phát triển nào cam kết đóng góp trực tiếp đến kho lưu trữ chính. Đóng góp được tổ chức thông qua pull request (yêu cầu kéo về) từ một

nhánh trong phiên bản phân nhánh của thư viện. Để biết thêm thông tin, xem thêm tài liệu trên GitHub tại mục đóng góp với pull requests. Nhóm Chisel bắt đầu viết tài liệu các nguyên tắc đóng góp.

13.4 Bài tập

Phát minh một toán tử mới cho kiểu `UInt`, triển khai nó trong thư viện Chisel và viết một số mã sử dụng/kiểm tra để khám phá toán tử. Nó không cần phải là một toán tử hữu ích; chỉ cần bất cứ thứ gì sẽ tốt, ví dụ: a ? toán tử chuyển cho phía bên trái nếu nó khác 0, nếu ngược lại là bằng 0 thì cho phía bên phải. Nghe giống như một mạch đa hợp, phải không? Bạn cần thêm bao nhiêu dòng mã?[2]

Đơn giản như vậy nhưng vui lòng không mạo hiểm phân nhánh dự án Chisel và thêm các tiện ích mở rộng nhỏ của các bạn. Các thay đổi và mở rộng sẽ được phối hợp với các nhà phát triển chính. Bài tập này chỉ là một bài tập đơn giản để các bạn bắt đầu.

Nếu các bạn táo bạo hơn, các bạn có thể chọn một trong các vấn đề mở và cố gắng giải quyết nó. Sau đó, đóng góp bằng một pull request tới Chisel. Tuy nhiên, có lẽ trước tiên hãy theo dõi phong cách phát triển trong Chisel bằng cách theo dõi các kho lưu trữ GitHub, xem cách các thay đổi và các pull request được xử lý trong dự án mã nguồn mở Chisel.

[2] Việc triển khai nhanh chóng và dễ hiểu chỉ cần hai dòng mã Scala.

14 Tóm lược

Cuốn sách này trình bày nhập môn thiết kế mạch số sử dụng ngôn ngữ xây dựng phần cứng Chisel. Chúng ta đã thấy một số mạch số đơn giản đến cỡ trung bình được mô tả trong Chisel. Chisel được nhúng trong Scala và do đó kế thừa tính trừu tượng mạnh mẽ của Scala. Vì cuốn sách này nhằm mục đích giới thiệu, nên chúng tôi đã giới hạn các ví dụ với các cách sử dụng đơn giản của Scala. Bước hợp lý tiếp theo là tìm hiểu một vài điều cơ bản về Scala và áp dụng chúng vào dự án Chisel của bạn.

Tôi rất vui khi nhận được phản hồi về cuốn sách, tôi sẽ cải thiện nó hơn nữa và sẽ xuất bản các ấn bản mới. Bạn có thể liên hệ với tôi tại mailto:masca@dtu.dk hoặc yêu cầu về vấn đề trên hệ thống kho lưu trữ GitHub. Tôi cũng vui vẻ chấp nhận các yêu cầu kéo về từ kho lưu trữ sách nhằm cải tiến và sửa bất kỳ lỗi nào.

Truy cập nguồn tài nguyên

Cuốn sách này có sẵn ở dạng mã nguồn mở. Kho lưu trữ cũng chứa các trang trình bày về khóa học Chisel và tất cả các ví dụ về Chisel: https://github.com/schoeberl/chisel-book

Một tập hợp các ví dụ cỡ vừa, hầu hết được tham khảo trong cuốn sách, cũng có sẵn ở dạng mã nguồn mở. Bộ tập hợp ví dụ này cũng chứa các tập tin dạng dự án cho các bo mạch FPGA phổ biến khác nhau: https://github.com/schoeberl/chisel-examples

A Các dự án Chisel

Chisel không (chưa) được sử dụng trong nhiều dự án. Do đó, việc dùng mã nguồn mở Chisel để học ngôn ngữ và cách lập trình là rất hiếm. Ở đây, chúng tôi liệt kê một vài dự án mà chúng tôi biết đang sử dụng Chisel và có mã nguồn mở.

Rocket Chip là RISC-V [13] Risc là một bộ tạo phức hợp bộ xử lý bao gồm vi kiến trúc Rocket bộ tạo kết nối TileLink. Ban đầu được phát triển tại UC Berkeley như là dự án Chisel quy mô chip đầu tiên [1], Rocket Chip hiện tại được hỗ trợ thương mại hóa bởi SiFive.

Sodor là một tập hợp các triển khai RISC-V dành cho mục đích giáo dục. Nó chứa 1, 2, 3 và 5 tầng đường ống (pipeline). Tất cả các bộ xử lý đều sử dụng một bộ nhớ vùng tạm đơn giản được chia sẻ bằng cách nạp lệnh, truy cập dữ liệu và tải chương trình thông qua một cổng gỡ lỗi. Sodor chủ yếu được sử dụng trong mô phỏng.

Patmos thực hiện bộ xử lý được tối ưu hóa cho các hệ thống thời gian thực [10]. Kho lưu trữ Patmos bao gồm nhiều kiến trúc giao tiếp đa lõi, chẳng hạn như trình phân xử bộ nhớ có thể dự đoán thời gian [7], mạng trên chip [9], bộ nhớ vùng tạm chia sẻ có quyền sở hữu [11]. Vào thời điểm viết bài này, Patmos vẫn được mô tả trong Chisel 2.

FlexPRET là một triển khai của kiến trúc định thời chính xác [14]. FlexPRET triển khai tập lệnh RISC-V và đã được cập nhật lên Chisel 3.1.

Lipsi là một bộ xử lý nhỏ dành cho các chức năng tiện ích trên hệ thống trên chip (SoC) [6]. Vì cơ sở mã của Lipsi rất nhỏ, nó có thể là điểm khởi đầu dễ dàng cho việc thiết kế bộ xử lý trong Chisel. Lipsi cũng trưng bày về năng suất của Chisel/Scala. Tôi đã mất 14 giờ để mô tả phần cứng trong Chisel và chạy nó trên FPGA, viết trình hợp dịch (assembler) trong Scala, viết trình mô phỏng tập lệnh Lipsi trong Scala để đồng mô phỏng và viết một vài trường hợp thử nghiệm trong trình hợp dịch Lipsi.

OpenSoC Fabric là một trình tạo NoC mã nguồn mở được viết bằng Chisel [5]. Mục đích nhằm cung cấp một hệ thống trên chip để khám phá thiết kế quy mô lớn. Bản

thân NoC là một thiết kế hiện đại với định tuyến lỗ sâu, tín dụng (credits) để kiểm soát luồng và các kênh ảo. OpenSoC Fabric vẫn đang sử dụng Chisel 2.

DANA là một bộ tăng tốc mạng nơ-ron, tích hợp với bộ xử lý RISC-V Rocket dùng giao tiếp Rocket Custom Coprocessor (RoCC) [4]. DANA hỗ trợ suy luận và học hỏi.

Chiselwatt là một triển khai của POWER Open ISA. Nó bao gồm tập lệnh để chạy Micropython.

Nếu các bạn biết một dự án nguồn mở sử dụng Chisel, vui lòng gửi cho tôi một ghi chú để tôi có thể đưa nó vào ấn bản của cuốn sách trong tương lai.

B Chisel 2

Cuốn sách này bao gồm phiên bản 3 của Chisel. Hơn nữa, Chisel 3 được khuyên dùng cho các thiết kế mới. Tuy nhiên, vẫn còn mã code của Chisel 2 chưa được chuyển đổi sang Chisel 3. Có sẵn tài liệu về cách chuyển đổi tập tin dự án Chisel 2 thành Chisel 3:

- Chisel2 so với Chisel3 và
- Hướng tới Chisel 3

Tuy nhiên, các bạn có thể tham gia vào một dự án vẫn sử dụng Chisel 2, ví dụ: bộ xử lý Patmos [10]. Vì vậy, chúng tôi cung cấp ở đây một số thông tin về mã Chisel 2 cho những người đã bắt đầu với Chisel 3.

Đầu tiên, tất cả tài liệu về Chisel 2 đã bị xóa khỏi các trang web của Chisel. Chúng tôi đã lấy lại các tài liệu PDF đó và đưa chúng lên GitHub tại https://github.com/schoeberl/chisel2-doc. Các bạn có thể sử dụng hướng dẫn Chisel 2 bằng cách chuyển sang nhánh Chisel 2:

```
$ git clone https://github.com/ucb-bar/chisel-tutorial.git
$ cd chisel-tutorial
$ git checkout chisel2
```

Sự khác biệt chính có thể nhìn thấy giữa Chisel 3 và 2 là các định nghĩa về hằng số, các bundle cho IO, dây dẫn, bộ nhớ và có thể là các dạng định nghĩa thanh ghi cũ hơn.

Ở một mức độ nào đó, các cấu trúc Chisel 2 có thể được sử dụng trong dự án Chisel 3 bằng cách sử dụng lớp tương thích dùng dưới dạng gói `Chisel` thay vì `chisel3`. Tuy nhiên, việc sử dụng lớp tương thích này chỉ nên được sử dụng trong giai đoạn chuyển tiếp. Do đó, chúng tôi không trình bày ở đây.

Ở đây là hai ví dụ về các thành phần cơ bản, giống như đã được trình bày cho Chisel 3. Một mô-đun chứa logic tổ hợp:

```
import Chisel._

class Logic extends Module {
  val io = new Bundle {
    val a = UInt(INPUT, 1)
```

```
    val b = UInt(INPUT, 1)
    val c = UInt(INPUT, 1)
    val out = UInt(OUTPUT, 1)
  }

  io.out := io.a & io.b | io.c
}
```

Lưu ý rằng `Bundle` cho định nghĩa IO *không* được bọc trong lớp `IO()`. Hơn nữa, hướng của các cổng IO khác nhau được định nghĩa như là một phần của định nghĩa kiểu, trong ví dụ này là `INPUT` và `OUTPUT` như là một phần của `UInt`. Độ rộng được cho là tham số thứ hai.

Ví dụ thanh ghi 8-bit trong Chisel 2:

```
  import Chisel._

  class Register extends Module {
    val io = new Bundle {
      val in = UInt(INPUT, 8)
      val out = UInt(OUTPUT, 8)
    }

    val reg = Reg(init = UInt(0, 8))
    reg := io.in

    io.out := reg
  }
```

Ở đây, các bạn thấy định nghĩa thanh ghi tiêu biểu với giá trị reset được đưa vào dưới kiểu `UInt` cho tham số có tên `init`. Dạng này vẫn đúng trong Chisel 3, nhưng việc sử dụng `RegInit` và `RegNext` được khuyên dùng cho các thiết kế Chisel 3 mới. Cũng lưu ý ở đây là định nghĩa hằng số của 0 có độ rộng 8 bit là `UInt(0, 8)`.

Kiểm tra mã C++ và mã Verilog dựa trên Chisel được tạo ra bằng cách gọi `chiselMainTest` và `chiselMain`. Cả hai hàm "main" đều lấy một mảng `String` cho các tham số khác.

```
import Chisel._

class LogicTester(c: Logic) extends Tester(c) {

  poke(c.io.a, 1)
  poke(c.io.b, 0)
  poke(c.io.c, 1)
```

```
  step(1)
  expect(c.io.out, 1)
}

object LogicTester {
  def main(args: Array[String]): Unit = {
    chiselMainTest(Array("--genHarness", "--test",
      "--backend", "c",
      "--compile", "--targetDir", "generated"),
      () => Module(new Logic())) {
        c => new LogicTester(c)
      }
  }
}


import Chisel._

object LogicHardware {
  def main(args: Array[String]): Unit = {
    chiselMain(Array("--backend", "v"), () => Module(new
        Logic()))
  }
}
```

Bộ nhớ với các cổng đọc và ghi thanh ghi tuần tự được định nghĩa trong Chisel 2 là:

```
  val mem = Mem(UInt(width = 8), 256, seqRead = true)
  val rdData = mem(Reg(next = rdAddr))
  when(wrEna) {
    mem(wrAddr) := wrData
  }
```

C Các từ viết tắt

Các nhà thiết kế phần cứng và kỹ sư máy tính thích sử dụng các từ viết tắt. Tuy nhiên, cần thời gian để làm quen với chúng. Dưới đây là danh sách các thuật ngữ phổ biến liên quan đến thiết kế mạch số và kiến trúc máy tính.

ADC analog-to-digital converter

ALU arithmetic and logic unit

ASIC application-specific integrated circuit

CFG control flow graph

Chisel constructing hardware in a Scala embedded language

CISC complex instruction set computer

CPI clock cycles per instruction

CRC cyclic redundancy check

DAC digital-to-analog converter

DFF D flip-flop, data flip-flop

DMA direct memory access

DRAM dynamic random access memory

EMC electromagnetic compatibility

ESD electrostatic discharge

FF flip-flop

FIFO first-in, first-out

FPGA field-programmable gate array

HDL hardware description language

HLS high-level synthesis

IC instruction count

IDE integrated development environment

ILP instruction level parallelism

IO input/output

ISA instruction set architecture

JDK Java development kit

JIT just-Iin-time

JVM Java virtual machine

LC logic cell

LRU least-recently used

MMIO memory-mapped IO

MUX multiplexer

OO object oriented

OOO out-of order

OS operating system

RISC reduced instruction set computer

SDRAM synchronous DRAM

SRAM static random access memory

TOS top-of stack

UART universal asynchronous receiver/transmitter

VHDL VHSIC hardware description language

VHSIC very high speed integrated circuit

WCET Worst-Case Execution Time

Tài liệu tham khảo

[1] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a Scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference (DAC 2012)*, pages 1216–1225, San Francisco, CA, USA, June 2012. ACM.

[3] William J. Dally, R. Curtis Harting, and Tor M. Aamodt. *Digital design using VHDL: A systems approach*. Cambridge University Press, 2016.

[4] Schuyler Eldridge, Amos Waterland, Margo Seltzer, and Jonathan Appavooand Ajay Joshi. Towards general-purpose neural network computing. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 99–112, Oct 2015.

[5] Farzaf Fatollahi-Fard, David Donofrio, George Michelogiannakis, and John Shalf. Opensoc fabric: On-chip network generator. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–203, April 2016.

[6] Martin Schoeberl. Lipsi: Probably the smallest processor in the world. In *Architecture of Computing Systems – ARCS 2018*, pages 18–30. Springer International Publishing, 2018.

[7] Martin Schoeberl, David VH Chong, Wolfgang Puffitsch, and Jens Sparsø. A time-predictable memory network-on-chip. In *Proceedings of the 14th International*

Workshop on Worst-Case Execution Time Analysis (WCET 2014), pages 53–62, Madrid, Spain, July 2014.

[8] Martin Schoeberl and Morten Borup Petersen. Leros: The return of the accumulator machine. In Martin Schoeberl, Thilo Pionteck, Sascha Uhrig, Jürgen Brehm, and Christian Hochberger, editors, *Architecture of Computing Systems - ARCS 2019 - 32nd International Conference, Proceedings*, pages 115–127. Springer, 1 2019.

[9] Martin Schoeberl, Luca Pezzarossa, and Jens Sparsø. A minimal network interface for a simple network-on-chip. In Martin Schoeberl, Thilo Pionteck, Sascha Uhrig, Jürgen Brehm, and Christian Hochberger, editors, *Architecture of Computing Systems - ARCS 2019*, pages 295–307. Springer, 1 2019.

[10] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, Apr 2018.

[11] Martin Schoeberl, Tórur Biskopstø Strøm, Oktay Baris, and Jens Sparsø. Scratchpad memories with ownership. In *2019 Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2019.

[12] Bill Venners, Lex Spoon, and Martin Odersky. *Programming in Scala, 3rd Edition*. Artima Inc, 2016.

[13] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.

[14] Michael Zimmer. *Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2015.

Chỉ mục