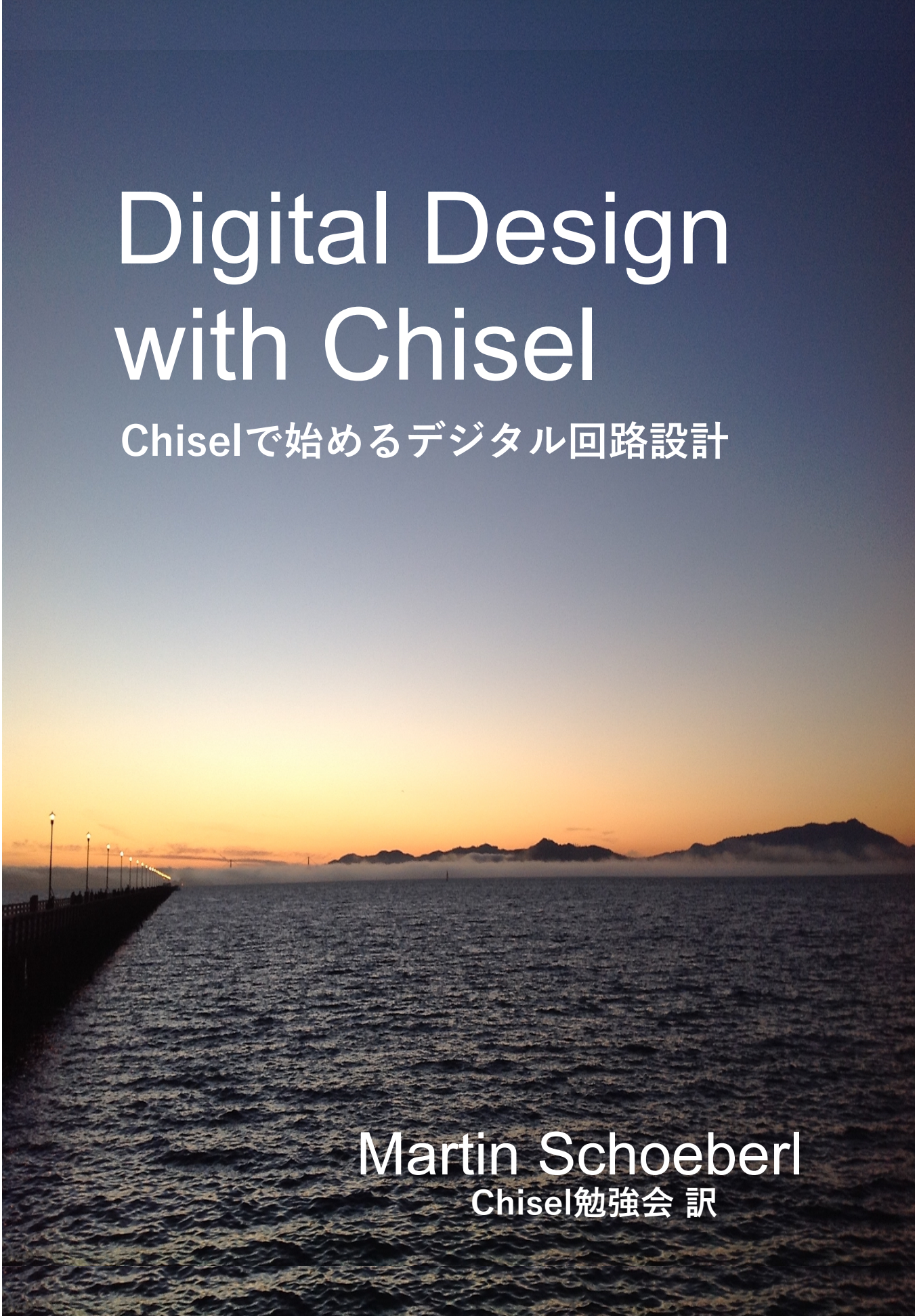
Digital Design with Chisel

Chiselで始めるデジタル回路設計



Martin Schoeberl

Chisel勉強会 訳

Chiselで始めるデジタル回路設計

第二版(日本語版)

Chiselで始める
デジタル回路設計

第二版(日本語版)

マーチン・シェーベル著
Chisel勉強会訳

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.

目次

図目次

表目次

コード目次

まえがき

デジタル回路設計の世界で作業することはとてもエキサイティングなことです。デナードスケーリングの終わりとムーアの法則の減速で、この分野での技術革新が必要となっています。半導体企業は、依然として性能向上に尽力していますが、性能改善のためのコストが大幅に上昇しています。 Chiselは、デジタル回路設計の生産性を向上させることで、このコストを削減します。設計の再利用による検証コストの削減や、開発初期投資（Non-Recurring Engineering、NRE）の削減により、設計者はより少ない時間でより多くの（論理）回路を開発することができます。また、学生や個人でも、イノベーションに取り組むことが簡単にできます。

Chiselはそれがscalaの中に埋め込まれているという点で、他のプログラミング言語とは異なります。基本的には、同期デジタル回路を表現するために必要なプリミティブを含むクラスと機能をライブラリ化したものが Chisel です。 Chiselのデザインは実際にはScalaのプログラムで、実行可能な回路を生成します。多くの人にとって、これは直感に反するかもしれません：「なぜ、Chisel をVHDLやSystemVerilogのようなスタンドアロンの言語にしないのだろうか?」。この質問に対する私の答えは次のとおりです。ソフトウェアの世界では、過去数十年間かつてないほど、様々な設計手法のイノベーションが起こりました。新しいハードウェアの言語にこれらの技術を適用しなくても、最新のプログラミング言語を 使用する だけで、これらのメリットを享受することができるのです。

Chiselは、「学ぶことが困難である」と長年批判されてきました。このような認識の多くは、自身の研究や、商業的なニーズを満足させるために、エキスパートによって作成された大規模で複雑な設計が蔓延したことによるものです。 C++のような人気のある言語を学習するとき、人々は GCCのソースコードを読んだりしません。講座やテキストなど、初習者向けに用意された豊富な学習資料から取りかかります。 Chiselを学びたい人のための重要なリソースとして、この*Chisel*で始めるデジタル回路設計をマーティンは書いてくれました。

マーティンは、経験豊富な教育者であり、それは本書の構成からもみてとれます。インストールおよび構成要素の解説から始めて、レンガを一つずつ積み上げて建物を作っていくように、読者の理解を深めてゆきます。付属の演習問題は、理解を固めるための接着剤で、それぞれの概念が読者の心に定着することを助けます。屋根が家をまとめ上げるのと同じように、ハードウェアジェネレータ の理解はこの本の1つの到達点となるでしょう。最終的には、RISCプロセッサのようなシンプルで有用なデザインを構築するための知識をこの本の読者は得ることになるでしょう。

マーティンは*Chisel*で始めるデジタル回路設計で生産的なデジタル回路設計のための強力なベースを築きました。次に何を作るかはあなた次第です。

ジャック・ケーニッヒ
ChiselとFIRRTLメンテナ
スタッフエンジニア、SiFive社

序文

この本では、ハードウェア構築言語であるChiselを使ったデジタル回路設計について解説します。 Chiselは、オブジェクト指向言語や関数型言語など、先進のソフトウェアエンジニアリング技術を、デジタル回路設計の世界にもたらします。

この本は、ハードウェア設計者とソフトウェアエンジニアの両方を対象としています。VerilogやVHDLの知識を持つハードウェア設計者は、モダンな言語をASICやFPGA設計に活用することで、その生産性を向上させることができます。オブジェクト指向と関数型プログラミングの知識を持つソフトウェアエンジニアは、例えば、クラウドで稼働するFPGAアクセラレータのようなハードウェアのプログラミング（設計）にその知識を活用することができます。

本書では、小規模もしくは中規模の一般的なハードウェアコンポーネントを例にして、Chiselを使ったデジタル回路設計を紹介していきます。

第2版のまえがき

Chiselは、アジャイルなハードウェア設計を可能にします。同様にオープンアクセスとオンデマンド印刷は、アジャイルな書籍の出版を可能にします。本書も初版のリリース後、半年未満で改善と拡張を加えた第二版をリリースすることができました。

マイナーな修正のほか、第二版での主な変更点は次の通りです。テストセクションが拡張されています。順序回路ブロック(sequential building blocks)の章では、より多くの回路例を紹介しています。入力処理（input processing）に関する新しい章が設けられ、入力の同期、デバウンス回路の設計、そしてノイズが多い入力信号のフィルタリング処理について説明します。デザイン例の章も拡張され、異なる種類のFIFOの実装方法を説明します。このFIFOのバリエーションでは、型パラメータと継承をデジタル回路設計のなかでどのように扱うかを解説しています。

謝辞

クールなハードウェア構築言語であるChiselの開発に携わったすべてのみなさまに感謝します。 Chiselは使用するのがとても楽しく、その本を書く価値があります。とてもオープンでフレンドリで、Chiselに関する質問に熱心に答えてくれる Chiselコミュニティ全体に感謝しています。

また、最後の数年間、先進コンピュータアーキテクチャコースを受講した学生たちに感謝したいと思います。ほとんどの生徒が最終プロジェクトのためにChiselを取り上げてくれました。殻から抜け出し、新しい勉強の旅に出て、最先端のハードウェア記述言語を使用していただき感謝します。 あなたたちの質問の多くは、この本を形作るのに大変役立ちました。

日本語訳について

この日本語訳は Chisel勉強会の4名（mune10, diningyo, aki, HHayashi）で行っています。翻訳で用いたオリジナルのバージョンは (2020年3月2日 b20a791)です。オリジナルの更新に合わせて翻訳も新しくしていく予定です。日本語版のソースコードはこちらhttps://github.com/chisel-jp/chisel-book で公開しています。誤訳や表現の誤りの訂正など、小さな修正も歓迎します。

Chiselに興味ある方は、勉強会に自由に参加できます。新型コロナの影響でF2Fの勉強会の開催はできておりませんが、Chisel勉強会のSlackへ登録(URL https://chisel-jp-slackin.herokuapp.com/)いただければ情報交換できると思います。

日本語版での変更点

- Wikipediaへのリンクは(英語)(日本語)の併記としました。これは日本語版Wikipediaの記述が不十分な用語があるためです。
- 原書の「デジタルデザイン」は、日本語版ではより具体的に「デジタル回路設計」としました。
- ソースコードのリストに薄いグレーの背景色を設定しました。また、コード目次に列挙されるソースコードにつきましては、挿入される位置が本文とはずれるため、枠と行番号を追加することで本文中のソースコードと区別しやすいように変更しました。

日本語版の履歴

2020-10-10 一時公開版

2021-02-01 RC 0.3 初回校正版、(目次にLatexソースの行番号追記）

2021-02-25 RC 0.4 RISCV勉強会校正版

2021-08-07 RC 0.5 行番号削除、リスト環境を変更

2021-08-22 RC 0.6 しおりの文字化け修正, A4、B5、PC版に対応

2021-08-XX RC 0.7 校正

1 はじめに

本書は、近代的なハードウェア構築言語である Chisel [2] を使ったデジタルシステム設計について解説します。本書では、一般的なデジタル回路設計の書籍に比べ、より高い抽象レベルでの設計に注目し、より複雑で、相互作用のあるデジタルシステムを短期間で開発できるようになることを目指します。

本書およびChiselは、（1）ハードウェア設計者と（2）ソフトウェアプログラマの2つの開発者のグループを対象としています。 VHDLやVerilogになれたハードウェア設計者は、Python、Java、またはTclのような言語も使いこなしながらハードウェアの開発を進めますが、今後はハードウェアの生成が言語の機能の一部となっている単一のハードウェア構築言語を使って開発を進めることができます。ハードウェア設計にも興味（例えば、インテルが性能向上にFPGAをチップに取り込むなど）があるソフトウェアプログラマにとっては、最初に覚えるハードウェア記述言語としてChiselは最適です。

Chiselは、オブジェクト指向や関数型言語といったソフトウェア工学の進歩をデジタル回路設計の世界にもちこみます。 Chiselは、レジスタ転送レベル（RTL）でのハードウェア記述をサポートするだけではなく、ハードウェアジェネレータ(生成器)を記述できます。

現在、ハードウェアの設計はハードウェア記述言語（HDL）を使うのが一般的です。CADツールなどを使用して、お絵かきでハードウェアコンポーネントを設計する時代は終わりました。回路構成図を作ることはありますが、それはシステムの概要を示すためで、システムを記述するためのものではありません。最も一般的なハードウェア記述言語は、VerilogとVHDLの2つです。どちらの言語も、歴史があり、様々な遺産を含んでいます。その言語のどのような記述がハードウェアに合成可能なのかの明確な境界がありません。勘違いしないでください：VHDLやVerilogは ASIC(英語)/ (日本語) に合成可能なハードウェアブロックを確実に記述することができます。 Chiselを使ったハードウェア設計では、Verilogはテストおよび合成のための中間言語として機能しています。

本書はハードウェア設計の一般的な紹介や基礎を扱うものではありません。 CMOSトランジスタを使ったゲートの生成といったようなデジタル回路設計の基本については、他の書籍を参照して下さい。なお、この本ではASICや FPGA(英語)/ (日本語) 向けのデザインに現在用いられている抽象レベルでのデジタル回路設計を説明しています。 [1] この本のための予備知識として、 Boolean algebra(英語)/ ブール代数(日本語) や binary number system(英語)/ 2進法のシステム(日本語) の知識を想定しています。更に何らかのプログラミング言語を使用した経験も想定しています。VerilogやVHDLの知識は必要ありません。 Chiselはデジタルハードウェアを設計するための、最初のプログラミング言語になりえます。例題中のビルド処理は`sbt`や`make`に基づいているので、コマンドラインを使ったインターフェース（CLI、ターミナルやUnixシェルとも呼ばれる）の基本的な知識が役に立つでしょう。

Chisel自体は大きな言語ではありません。基本的な文法は早見表に収まる程度なので、数日で習得することができます。したがって、本書もそんなにボリュームのある本ではありません。 Chiselは多くの資産を持つVHDLやVerilogよりは確かに小さい言語です。ChiselのパワーはChiselが表現能力の優れたScala言語に組み込まれていることにあります。ChiselはScalaの「拡張性」 [12]から機能を継承しています。しかしながら、Scala自体はこの本の主たるトピックではありません。 Scalaの一般的な紹介については、オーダスキ

[1] 著者はターゲットとする技術面ではASICよりもFPGAに詳しいため、この本で紹介されるデザインの最適化はFPGAをターゲットにしている場合があります。

ー(Scalaの開発者)のテキストブック [12]を参照してください。この本は、デジタル回路設計とChisel言語のチュートリアルです。Chiselの言語リファレンスではありませんし、完全なチップの設計に関する本でもありません。

本書に掲載されているコード例はすべてコンパイルが可能で、テスト済みの完全なプログラムから抽出されています。そのためコードにはシンタックスエラーは含まれていません。コード例は本書のGitHubリポジトリに公開されています。本書ではChiselのコードだけでなく、良いハードウェアの記述スタイルの原則や便利なデザインについても紹介しています。

この本はノートPCやタブレット（iPadなど）に向けて最適化しており、Wikipedia(英語)/(日本語) の記事を中心に補足のためのリンクを掲載しています。

1.1 ChiselとFPGA開発ツールのインストール

ChiselはScalaのライブラリの一種であり、最も簡単なChiselとScalaのインストール方法はScalaのビルドツールである sbtを使うことです。Scala自体はJava JDK 1.8に依存しています。OracleがJavaに関するライセンスを変更したため、AdoptOpenJDKからOpenJDKをインストールする方がより簡単でしょう。

1.1.1 macOS

AdoptOpenJDKからJava OpenJDK 8をインストールします。Mac OS Xでは、パッケージマネージャのHomebrewを使用することで、sbtとgitを次のようにインストールできます。

```
$ brew install sbt git
```

GTKWaveとIntelliJ（コミュニティ版）をインストールします。プロジェクトをインポートする場合、本節でインストールしたJDK 1.8を選択してください (Java 11ではありません！)。

1.1.2 Linux/Ubuntu

Ubuntuでは次のコマンドでJavaと役に立つツール類をインストールできます。

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

UbuntuはDebianが元になっているため、プログラムはたいていDebianファイル（.deb）からインストールできます。しかし本書の執筆時点では、sbtはインストール可能なパッケージが存在していませんでした。そのためインストール手順が少し複雑になります。

```
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

AdoptOpenJDKからJava OpenJDKをインストールします。 ChiselとScalaもWindowsにインストールして使用できます。 GTKWaveとIntelliJ (コミュニティ版) をインストールします。プロジェクトをインポートする際には、インストール済みの JDK 1.8を選択し てください (Java 11ではありません!) 。その前にWindowsでのsbtのインストールを参考にして、Windowsインストーラでsbtをインストールします。また、gitクライアントをインストールします。

1.1.4 FPGA ツール

FPGA向けにハードウェアをビルドするためには、論理合成ツールが必要です。 2つの有名なFPGAベンダであるIntel[2]とXilinxは小規模から中規模のFPGAをサポートしたフリーのツールを提供しています。これらの中規模なFPGAはRISCプロセッサによるマルチコアをビルドするには十分です。 IntelはQuartus Prime Liteエディションを、XilinxはVivado Design Suite, WebPACKエディションをそれぞれ提供しています。これらのツールにはWindows/Linux版はありますが、macOS向けのものはありません。

1.2 Hello World

どのプログラミング言語でも*Hello World*と呼ばれる最小の例題からスタートします。次に示すコードがその最初のアプローチです。

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

この短いプログラムをsbtを使ってコンパイルして実行します。

```
$ sbt run
```

Hello Worldのプログラムの期待される出力は次のようなものです。

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

しかしこれはChiselなのでしょうか？このハードウェアは文字列を印刷するために生成されたものでしょうか？いいえ、これはただのScalaのコードであってハードウェア設計におけるHello Worldプログラムではありません。

1.3 Chisel で Hello World

それではハードウェア設計においてHello Worldプログラムと言えるものは何なのでしょうか？簡単に見ることができて役に立つ最小のデザインとは？その答えはLEDを点滅させること（Lチカ）で、これがハードウェア（もしくは組み込みソフトウェア）におけるHello Worldです。 LEDが点滅していれば、より大きな問題を解決するための準備ができたことになります。

[2] 旧Altera

```
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
}
```

コード 1.1: Chiselで書いたハードウェア設計における Hello World

コード 1.1はLチカの処理をChiselで実装したものです。次の章でこのコードの詳細について解説するので、ここではコードの詳細について理解する必要はありません。注意しておきたいのは回路は通常50MHzといった高速なクロックで動作するため、目に見える点滅を生成するためにHzレンジのタイミングのカウンタが必要になります。上記の例では0から25000000-1までカウントし、点滅信号をトグルし（`blkReg := ~blkReg`）、カウンタを再スタートします。これによってハードウェアはLEDを1Hzで点滅させます。

1.4 Chisel 用のIDE

本書では、プログラミング環境やエディタについては何も仮定していません。基本的なことはコマンドライン上で `sbt` を使い、好きなエディタを使うだけで簡単に習得できます。他の書籍の伝統に習って、シェル/ターミナル/CLIに入力しなければならないコマンドの前には`$`がありますが、これは入力しません。例として、現在のフォルダ内のファイルを一覧表示する Unix の `ls`コマンドを示します。

```
$ ls
```

バックグラウンドでコンパイラが動いている統合開発環境（IDE）を利用することで、コーディングの高速化が可能になると言われています。 ChiselはScalaのライブラリなので、ScalaをサポートしているIDEはすべてChiselに適したIDEでもあります。例えば、 `build.sbt` で構成された sbt プロジェクトから IntelliJ や Eclipse のプロジェクトを生成することができます。

IntelliJでは、 *File - New - Project from Existing Sources...* からプロジェクトの中の `build.sbt` ファイルを選べば、既存のソースから新しいプロジェクトを作成することができます。

Eclipseでは、

```
$ sbt eclipse
```

で、そのプロジェクトをEclipseにインポートします。[3]

[3] sbt用のEclipseプラグインが必要です。

Visual Studio CodeもChisel 用 IDEとして使えます。 Scala Metals 拡張が Scala のサポートを提供します。左のバーで *Extensions*を選択、 *Metals* をサーチして、 *Scala (Metals)*をインストールします。 `sbt` ベースのプロジェクトをインポートするには、 *File - Open* でフォルダをオープンします。

1.5 本書のソースコードへのアクセスと電子書籍の機能

この本はオープンソースで、GitHub: chisel-book でホストされています。[4] この本で紹介されている Chisel のコード例はすべてリポジトリに含まれています。コードは最新バージョンの Chisel でコンパイルされており、多くの例にはテストベンチも含まれています。付属のリポジトリ chisel-examples には、より大きな Chisel の例題を集めています。本書の中に誤りやタイプミスを見つけた場合は、GitHub のプルリクエストで改善点を取り入れるのが最も便利な方法です。また、GitHub に Issue を提出することで、改善のためのフィードバックやコメントを提供することもできます。あと、昔ながらの平凡なメールも送れます。

この本は、PDFの電子ブックと古典的な印刷形式で自由に利用できます。電子ブック版には、さらなるリソースと Wikipediaの記事へのリンクがあります。本書に直接当てはまらない背景情報（例：2進数方式）については、Wikipediaの記事を利用しています。 iPadなどのタブレットで読めるように、電子書籍のフォーマットを最適化しています。

1.6 参考文献

デジタル回路設計とChiselに関する参考文献をリストします

- Digital Design: A Systems Approach, by William J. Dally and R. Curtis Harting, は、デジタル回路設計に関する最新の教科書です。ハードウェア記述言語としてVerilogとVHDLの2つのバージョンがあります。

Chiselの公式ドキュメントやその他の関連ドキュメントはオンラインで利用可能です。

- The Chisel ホームページは、Chiselをダウンロードして学ぶための公式の出発点です。
- The Chisel Tutorial はテストとソリューションおよび、小さな演習問題を含む準備されたプロジェクトを提供します。
- The Chisel Wiki にはChiselの簡単なユーザガイドと詳細情報へのリンクが含まれています。
- The Chisel Testers は、Wikiドキュメントを含む独自のリポジトリです。
- The Generator Bootcamp は、ハードウェアジェネレータを中心とした、Jupyter ノートブック形式のChisel講座です。
- A Chisel Style Guide by Christopher Celio.
- The chisel-lab には、デンマーク工科大学の「デジタルエレクトロニクス2」コースのChisel練習問題が収録されています。

日本語の情報についてもリストします（日本語版追記）

[4] 日本語版は chisel-book 日本語訳

- Chiselを始めたい人に読んでほしい本 (だいにんぎょー著) は、ChiselとそのもととなるScalaの日本語で最初の解説書です。 Amazonでも買えます。
- Chiselクイックリファレンス (だいにんぎょー著) はchisel3.utilパッケージのリファレンス集です。
- プログラマのためのFPGAによるRISC-Vマイコンの作り方 (堀江徹也著) はChiselの紹介から始まり、SiFive社のフリーのRISC-V SoC実装である Freedomを例にFPGAの開発までカバーします。
- RISC-VとChiselで学ぶはじめてのCPU自作——オープンソース命令セットによるカスタムCPU実装への第一歩 (西山悠太朗、井田健太著) はChiselを利用したCPUの作り方を解説しています。

1.7 演習

各章の終わりにはハンズオンの演習があります。最初の演習では、FPGAボードを使用してLEDを1つ点滅させます。[5] 最初のステップとして、chisel-examplesリポジトリをgithubからclone (またはfork)してください。 Hello World の例は `hello-world` フォルダにあり、最小のプロジェクトとしてセットアップされています。 `src/main/scala/Hello.scala` を見ることでLEDの点滅のChiselコードを調べることができます。LEDの点滅コードをコンパイルするために、次のコマンドを実行します。

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

最初にChiselコンポーネントのダウンロードが行われた後に、`Hello.v`という名前のVerilogファイルが生成されます。このVerilogファイルを見ていきましょう。`clock`と`reset`という2つの入力と`io_led`という出力が含まれていることがわかります。このVerilogファイルとChiselのモジュールを比較してみると、Chiselモジュールには`clock`と`reset`が含まれていないことに気づくでしょう。 Chiselではこれらの低レベルな信号は暗黙的に生成されますが、多くの場合、明示的に扱わない方が便利です。Chiselにはレジスタもコンポーネントとして含まれており、これらには必要に応じて`clock`と`reset`が接続されます。

次の手順として論理合成ツールのFPGAプロジェクトファイルを設定し、ピンをアサインし、Verilogコードをコンパイルし、得られたビットファイルでFPGAをコンフィグします（ビットファイルをFPGAやボード上のFlashROMに書き込む)。[6] これらの手順の詳細についてはここでは述べませんので、IntelのQuartusやXilinxのVivadoのマニュアルを参照してください。しかしながらexamplesリポジトリには、いくつかのポピュラーなIntelのFPGAボード（例:DE 2-115）ですぐに使えるQuartusプロジェクトが`quartus`というフォルダに含まれています。リポジトリに含まれているボードを持っている場合は、Quartusを立ち上げてプロジェクトを開き、*Play*ボタンを押してコンパイルを行い、*Programmer*ボタンを押してFPGAボードの設定を行えば、LEDが点滅します。

おめでとうございます! あなたは Chiselの最初のデザインを FPGAで動作させることに成功しました!

もしLED が点滅していない場合は、リセットの状態を確認してください。 DE2-115の設定では、リセットの入力は SW0 につながっています。

[5] FPGAボードを使用できない場合は、演習の最後にあるシミュレーション結果を参照して下さい。
[6] 実際のプロセスは、論理合成、配置配線、タイミング解析の実行、およびビットファイルの生成と、各ステップでさらに細かくなります。ただし、この導入例では、単にコードを「コンパイル」します。

次に、点滅頻度を遅い値または速い値に変更して、ビルドプロセスを再実行します。点滅周波数と点滅パターンは、異なる「状態 (emotion)」を伝えます。例えば、遅い点滅のLEDはすべてが正常であることを示し、速い点滅のLEDは異常状態を示します。どの周波数がこれらの2つの異なる「状態 (emotion)」を最もよく表現しているかを探ってみましょう。

演習のより挑戦的な拡張として、次の点滅パターンを生成します。 LED を毎秒約200msの間、点灯させます。この場合、カウンタのリセットとは切り離して、LEDの点滅を変化させます。そのためには、`blkReg` レジスタの状態を変更させる第2の定数値が必要です。このパターンは、どのような「状態 (emotion)」を生み出すのでしょうか？異常を知らせるのか？それとも活動していることを示すようなものなのでしょうか？

（まだ）FPGAボードをお持ちでない場合でも、LEDの点滅の例を実行することができます。 Chiselシミュレーションを使用します。シミュレーション時間が長くなりすぎないように、Chiselコードのクロック周波数を50000000から50000に変更してください。以下のコマンドを実行してLEDの点滅をシミュレーションします。

```
$ sbt test
```

これにより、100万クロックサイクルで動作するテスタが実行されます。点滅の頻度はシミュレーションの速度に依存しており、お使いのコンピュータの速度に依存します。そのため、想定したクロック周波数でLEDの点滅のシミュレーションが出来るか、少し実験する必要があるかもしれません。

2 基本コンポーネント

ここでは、デジタル回路設計のための基本的なコンポーネントを紹介します。組み合わせ回路とフリップフロップです。これらの基本的な要素を組み合わせることで、より大きくて面白い回路を作ることができます。

一般的なデジタルシステムでは、1 つのビットもしくは信号が 2 つの可能な値のうち 1 つしか持てないことを意味するバイナリ信号を使用します。これらの値はしばしば0と1と呼ばれます。その他、次のような用語も使用します：low/high、false/true、そしてde-asserted/asserted。これらの用語は、バイナリ信号がとりうる 2つの値を意味しています。

2.1 信号タイプと定数

Chiselでは信号や組み合わせ論理、レジスタを表現するために、`Bits`、`UInt`、`SInt`の3つのデータ型を提供しています。`UInt`と`SInt`は`Bits`から派生したもので、これら3つの型はビットの集まりを表現します。`UInt`は符号なし整数のビットの集合を、`SInt`は符号つきの整数を意味します。[1] Chiselは符号付き整数を two's complement(英語)/ 2の補数(日本語) で表現します。次に示すのは8-bitの`Bits`型、8-bitの符号なし整数、10-bitの符号つき整数の定義です。

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

ビット幅はChiselの型の1つである`Width`型によって定義されます。次の表現はScalaの整数`n`をChiselの`Width`型にキャストし、それを`Bits`型の定義に使用しています。

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

定数はScalaの整数型をChiselの型に変換することで定義できます。

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

定数はChiselの幅を表す型を使って、指定のビット幅で定義することもできます。

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

もし3.Uと4.Wという表記を見つけた場合、少しおかしく思えますが型つきの整数の変数と考えて下さい。この表記はCやJava、Scalaでlong型を表現するために3Lと表記することと同様です。

ハマりやすい落とし穴： 定数の宣言時に起こりがちなエラーとして、ビット幅指定のための`.W`を忘れることがあります。例えば`1.U(32)`のような表現は32-bitの1を表す定義ではありません。その代わりに`(32)`はビット位置32のビット抽出の指定として解釈されるため、

[1] 現在のChiselにおいて`Bits`型には演算処理が存在しません。そのため、ユーザにとってはあまり有用ではありません。

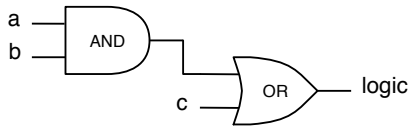


図 2.1: (a & b) | cの論理.信号は単一、もしくは複数のビットになり得る。Chiselの表現と回路図は同じになる。

結果的に1-bitの0になります。これはおそらくプログラマが本来意図したものではないはずです。

Chisel ではScalaの型推定のおかげで、多くの場合において型情報を省略できます。これはビット幅の場合においても同様です。多くの場合、Chiselは自動的に正しいビット幅を推測します。それ故に、Chiselで記述されたハードウェアはVHDLやVerilogに比べて簡潔で読みやすいものになります。

10進数以外の定数を表現するには、定数に先行する形で文字列を追加します。追加する文字列は、16進数の場合はhを、8進数の場合はoを、2進数の場合はbとなります。次の例は255という定数を異なる基数で表現したものです。この例ではビット幅の指定は省略し、Chiselが宣言する定数に収まる最小のビット幅を推定します。このケースでのビット幅は8bitになります。

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

上記のコードは、アンダースコアを使って定数値の桁をグループ化する方法も示しています。アンダースコアは無視されます。

Chiselでは論理値を表現する方法としてBool型を定義しています。 Boolは*true*か*false*の値を取ります。次に示すコードはBool型の定義と、ScalaのBoolean型の定数からの変換を用いたChiselのBool型のtrueとfalseの宣言です。

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

2.2 組み合わせ回路

Chiselは組み合わせ論理回路を記述するために、C言語やJava、 Scala、またその他のプログラミング言語と同様に Boolean algebra(英語)/ ブール代数(日本語) 演算子が使用されます。 &はAND（論理積）、 |はOR（論理和）を表現します。次の行に示すコードはaとbの信号を*and*ゲートで結合し、その結果とcを*or*ゲートに入力しています。

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

図 2.1はこの組み合わせの表現の回路図を示します。この回路のAND、ORゲートの接続信号は単一のビットのみならず、複数のビットからなるものであっても良いことに留意してください。

この例ではlogic信号のビット幅や型を定義しません。どちらも式の型やビット幅から推測されています。 Chiselでの標準的な論理演算は次のようになります。

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

算術演算には次の標準演算子を使用します。

```
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
```

演算結果のビット幅は、加算と減算は2つのうち大きい方のビット幅、乗算では2つのビット幅の合計、そして、除算とモジュロ演算では分子のビット幅になります。[2]
最初に信号をあるChiselの型の`Wire`として定義しておいて、その後、:= update演算子を使用してその`Wire`に値を割り当てることもできます。

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

w := a & b
```

特定の1ビットは、次のように抽出できます。

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

サブフィールドは終了位置から開始位置までを指定することで抽出できます。

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

ビットフィールドは`Cat`で連結できます。

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

表 2.2に、演算子の完全なリストを示します (組み込み演算子も参照)。 Chiselオペレータの優先順位は、 Scalaオペレータの優先順位に従う回路の評価順序によって決定されます。不安な場合は、括弧を使用することをお勧めします。[3]
表 2.2はChiselの型に対して定義されたさまざまな関数をまとめたものです。

2.2.1 マルチプレクサ

multiplexer(英語)/ マルチプレクサ(日本語) は、複数の選択肢からの選択を行う回路です。最も基本的な形式では、2つの選択肢のどちらかを選択します。図 2.2はそのような2:1マルチプレクサ、略してmuxを示しています。選択信号 (sel) の値に応じて信号yは信号aまたは信号bのいずれかの値になります。
マルチプレクサは論理式でも表現できます。しかしマルチプレクサは標準的な回路なので、Chiselではマルチプレクサを提供しています。

[2]厳密な詳細はFIRRTL仕様に記載されています。
[3]Chiselでの演算子の優先順位は、Scala演算子を実行してハードウェアノードのツリーを作成したときのハードウェア生成の副作用です。 Scalaの演算子の優先順位はJava/Cと似ていますが、同じではありません。 Verilogの演算子の優先順位はCと同じですが、VHDLの演算子の優先順位は異なります。 Verilogには論理演算の優先順位がありますが、VHDLではこれらの演算子は同じ優先順位を持ち、左から右に評価されます。

演算子	処理	データの型
* / %	乗算、除算、モジュロ	UInt, SInt
+ -	加算、減算	UInt, SInt
=== =/=	等しい、等しくない	UInt, SInt, returns Bool
> >= < <=	比較	UInt, SInt, returns Bool
<< >>	左シフト、右シフト (SIntの場合は符号拡張が行われる)	UInt, SInt
~	反転 (NOT)	UInt, SInt, Bool
& | ^	論理積 (AND)、論理和 (OR)、排他的論理和 (XOR)	UInt, SInt, Bool
!	論理否定	Bool
&& ||	論理積、論理和	Bool

表 2.1: Chiselで定義されているハードウェアの演算子

メソッド	処理	データ型
v.andR v.orR v.xorR	リダクションAND, OR, XOR	UInt, SInt, returns Bool
v(n)	特定の1bitの選択	UInt, SInt
v(end, start)	連続ビットの選択	UInt, SInt
Fill(n, v)	n回ビット列を繰り返し	UInt, SInt
Cat(a, b, ...)	ビット列の連結	UInt, SInt

表 2.2: vに適用できるハードウェアのメソッド

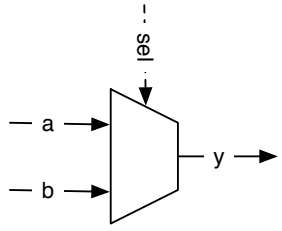


図 2.2: 基本的な 2:1 マルチプレクサ.

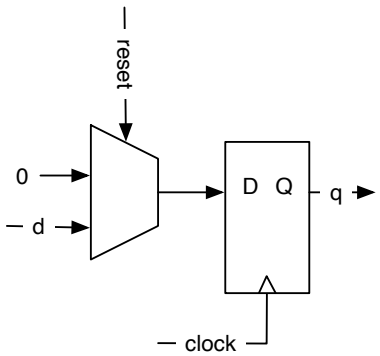


図 2.3: 同期リセットで0初期化されるDフリップフロップベースのレジスタ

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

このマルチプレクサは`sel`が`true.B`のときに`a`が選択され、そうでなければ`b`が選択されます。`sel`はChiselの`Bool`型の信号です。入力`a`および`b`は同じ型であれば、任意のChisel基本型または集約型 (`Vec`や`Bundle`) にすることができます。

論理演算、算術演算、それとマルチプレクサを使えばあらゆる組合せ回路を記述できます。しかしChiselでは、後の章で取り扱うように組み合わせ回路のより洗練された記述を目的とした、さらなるコンポーネントと制御の抽象化を提供しています。

デジタル回路を記述するために必要な第2の基本要素は、レジスタとも呼ばれる状態要素です。次節ではこれについて説明します。

2.3 レジスタ

Chiselには D flip-flops(英語)/ D型フリップフロップ(日本語) を集めた要素であるレジスタが備わっています。レジスタには暗黙のうちにグローバルクロックが接続され、そのクロックの立ち上がりエッジで更新されます。初期値はレジスタの宣言時に指定することが可能で、その値はグローバルリセットに接続された同期リセットで使用されます。レジスタは、ビットの集合を表現できる任意のChiselの型を扱うことができます。次のコードは、0で初期化された8ビットのレジスタを宣言するものです。

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

入力はレジスタにアップデート演算子 `:=` によって接続され、レジスタの出力はコード上の名前をそのまま使用できます。

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

レジスタの宣言時に、レジスタへの入力を接続することもできます。

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

図 2.3は、先ほどのレジスタの定義を回路図で示したものです。クロックと`0.U`で初期化するための同期リセット、入力`d`、出力`q`が含まれています。グローバルな信号である`clock`と`reset`は、定義されたレジスタに、暗黙のうちに接続されます。

レジスタ宣言時に、入力の接続に加えて、初期値を指定することも可能です。

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

組み合わせ論理の信号とレジスタの信号を区別するための一般的な方法として、レジスタ名の後ろにRegを付ける方法があります。他にJavaとScalaに由来した方法として、camelCase(英語)/ キャメルケース(日本語) で複数の言葉から構成される識別子を付与する方法があります。

2.3.1 カウント

カウント動作はデジタルシステムの基本的な操作です。イベントをカウントすることもありますが、より多く見られるのは時間の間隔を定義するために使用されるケースです。クロックのサイクル数をカウントして、指定の時間間隔が経過した際に、動作のトリガとします。

シンプルなアプローチでは、ある値までカウントアップしていきます。しかし、コンピュータサイエンスとデジタル回路設計では、カウントは0からスタートします。そのため、10をカウントする場合は、0から9までのカウントを行います。これを行ったのが次に示すコードで、9までカウントした後は0に戻ります。

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

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

2.4 バンドルとVecを用いた構造体

Chiselは関連した信号をまとめるための構造を2つ備えています。1つはBundleで、異なった型をグループ化して扱うもので、2つ目はVecと呼ばれ、同じ型の信号をインデックス指定可能なコレクションとして表現するものです。BundleとVecは何度でもネストすることができます。

Chiselのバンドルは複数の信号をグループ化します。バンドル全体を、単一の名称で参照することも出来ますし、個々の信号名によって各フィールドにもアクセス可能です。バンドル（信号のコレクション）を定義するためには、Bundleクラスを継承したクラスを定義し、クラスのコンストラクタ内にvalでフィールドをリストアップします。

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

バンドルを使用する場合は、そのバンドルのクラスをnewでインスタンス化し、Wireでラップします。フィールドへのアクセスは”.”（ドット）を使って行います。

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

  val b = ch.valid
```

”.”（ドット）を用いた表記はオブジェクト指向言語では一般的なものです。 x.yという表記があった場合、xはオブジェクトへの参照を示し、yはそのオブジェクトのフィールドとなります。 Chiselはオブジェクト指向言語であるため、バンドル内のフィールドへのアクセスは”.”（ドット）を使用して行います。 バンドルはC言語やSystem Verilogのstruct、VHDLではrecordに似ています。 バンドルはまた、全体としても参照することができます。

```
val channel = ch
```

ChiselのVecは同じ型（ベクトル）の信号のコレクションを表現するものです。各要素へはインデックスを用いて、アクセスできます。ChiselのVecは他のプログラミング言語においての、配列(Array)のようなデータ構造と似ているものです。 [4] Vecは2つのパラメータを持ったコンストラクタを呼び出すことで、生成できます。 2つのパラメータとは、要素数と要素の型です。組み合わせ論理のVecを使うには、Wireでラップする必要があります。

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

個々の要素は<Vecの変数>(インデックス)でアクセスします。

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

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

Wireをベクタにラップするとマルチプレクサになります。またレジスタをベクタにラップすると、レジスタの配列となります。次の例ではプロセッサのための32-bit x 32のレジスタファイルを定義しています。この例は32-bit版の RISC-V(英語)/ (日本語) のような、古典的な RISC(英語)/ (日本語) プロセッサで用いられます。

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

レジスタファイルの各要素へは、インデックスを用いてアクセスを行い、それらは通常のレジスタと同様に使用できます。

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

Bundle型とVec型は自由に混在できます。 Bundle型を要素とするVec型を作る場合、 Vecのフィールドに Bundle型のプロトタイプを渡す必要があります。先ほど、上で定義したChannelを使う場合、次のようにしてChannel型のVecを作成できます。

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

同様に、 BundleにもVecを含むことができます。

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

[4] Arrayという名前は、 Scalaによってすでに使用されています。

リセットが必要なBundle型のレジスタを使う場合は、最初にそのBundle型のWireを作成し、個々のフィールドに必要な値を設定した後、このWireの変数をRegInitに渡します。

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

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

val channelReg = RegInit(initVal)
```

Bundle型とVec型の組み合わせを使うことで、強力に抽象化された、任意のデータ構造を定義できます。

2.5 Chisel によるハードウェア生成

最初のChiselのコードを見た後に「JavaやC言語のような古典的なプログラミング言語に似ている」と思われたかもしれません。しかし、Chisel（もしくは他のハードウェア記述言語）はハードウェアコンポーネントを定義しています。ソフトウェアのプログラムでは1行のコードは他のコードの後に実行されますが、ハードウェアにおいてはすべてのコードが並列に実行されます。

Chiselのコードはハードウェアを生成するものである、ということを心に留めておく必要があります。頭で思い描いた、もしくは紙の上に書いた、個々のブロックが、Chiselの回路記述によって生成されます。すべてのコンポーネントの生成や、すべての接続の記述はANDやOR、フリップフロップ等のゲート素子を生成します。

より技術的には、ChiselのコードがScalaのプログラムとして処理されるとき、実行されたChiselのコードによって、ハードウェアコンポーネントが集約され、各々のノードに接続されます。このハードウェアノードのネットワークが、ASICやFPGAの合成用のVerilogコードとして出力され、Chiselのテスタによってテストされるハードウェアとなります。このハードウェアノードのネットワークは完全に並列に実行されます。

ソフトウェアエンジニアの方は、アプリケーション用のスレッドや通信のためのロックの取得を必要とせずに、ハードウェアによって実現される、莫大な並列性を想像してみてください。

2.6 演習

導入の演習では、ハードウェア版*Hello World*であるFPGAボードを使ったLチカを実装しました(from chisel-examples)。この実装では、唯一の内部ステートと、1つのLED出力のみで、入力はありませんでした。このプロジェクトを別のフォルダにコピーして、変数ioのBundleに`val sw = Input(UInt(2.W))`を追加してください。

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

これらのスイッチのために、FPGAボード上のピンの名前を割り振る必要があります。これらのピンのアサインはALUプロジェクト用のQuartusのプロジェクトファイルの中で、見つけられます。（例：DE2-115 FPGA board）

これらの入力とピンアサインを定義すれば、簡単なテストを始められます。そのテストとは、デザインから点滅する論理を削除し、1つのスイッチをLEDの出力に接続し、コンパイルとFPGAへの設定を行うことです。スイッチのON/OFFによってLEDを切り替えられましたか？答えが”はい”であれば、その入力は有効です。もし”いいえ”なら、FPGAの設定(コンフィギュレーション)をデバッグする必要があります。ピンの割り振りはツールのGUI画面で行うことが可能です。

2つのスイッチを用いて、ANDのような基本的な組み合わせ論理の結果をLEDに出力してみましょう。つぎに論理式を変えてみましょう。その次のステップは3つの入力を用いて、マルチプレクサを実装してみましょう。 1つの入力が選択用の信号となり、残りの2つは信号の入力となる2入力/1出力のマルチプレクサです。

ここまでは、シンプルな組み合わせ論理を実装して、その機能をFPGA上の本物のハードウェアでテストしました。次のステップでは、FPGAの設定(コンフィギュレーション)を生成するためのビルドプロセスが、どのようにして動作するのかを、少し見てみましょう。さらに、FPGAをコンフィグしたり、スイッチを切り替えることなく回路をテストできる、Chiselのシンプルなテストフレームワークを学びます。

3 ビルドプロセスとテスト

もっと面白い Chisel コードの記述を始める前に、まずChiselプログラムのコンパイル方法、FPGAで実行するためのVerilogコードの生成方法、回路が正しいことを検証するためのデバッグやテストの書き方を学ぶ必要があります。

ChiselはScalaで書かれているので、Scalaをサポートするビルドプロセスは Chiselプロジェクトで利用可能です。 Scalaの人気のあるビルドツールの一つに、Scala interactive Build Toolの頭文字をとった sbt があります。 `sbt` は、ビルドやテストプロセスを実行するだけでなく、正しいバージョンのScalaとChiselライブラリをダウンロードします。

3.1 sbt でプロジェクトを構築する

Chiselを表すScalaライブラリとChiselのテスタは、ビルド処理中にMavenリポジトリから自動的にダウンロードされます。各種ライブラリは `build.sbt` で指定します。 `build.sbt` に `latest.release` を設定することで、常に最新バージョンの Chisel を使用するようにすることができます。しかし、これは各ビルドで必要なバージョンがMavenリポジトリから検索されることを意味します。ビルドを成功させるためには、この検索のためのインターネット接続が必要になります。 Chiselやその他のScalaライブラリは、特定のバージョンを `build.sbt` で指定した方が良いでしょう。そうすれば、インターネットに接続しなくてもハードウェアコードを書いてテストすることが出来ます。例えば、飛行機の上でハードウェア設計をするのはクールですよね。

3.1.1 ソースコードの構成

`sbt` はビルド自動化ツール Maven のソース規約を継承しています。また、Mavenはオープンソースの Javaライブラリのリポジトリを取りまとめます。 [1]

[1] 最初のビルドでChiselライブラリをダウンロードした場所になります。https://mvnrepository.com/artifact/edu.berkeley.cs/chisel3.

```
project
├── src
│   ├── main
│   │   └── scala
│   │       └── package
│   │           └── sub-package
│   └── test
│       └── scala
│           └── package
├── target
└── generated
```

図 3.1: Chiselプロジェクトのソースツリー (`sbt` 利用)

図 3.1 は、典型的なChiselプロジェクトのソースツリーの構成を示します。プロジェクトのルートはプロジェクトのホームであり、build.sbtが置かれます。また、ビルドプロセスのための Makefile ファイルや、README、LICENSEファイルも配置します。 srcフォルダには、全てのソースコードが配置されています。そこから、ハードウェアのソースが含むmainと、テストコードを含むtest に分かれます。 ChiselはScalaを継承しており、ScalaはJavaからソースの packages(英語)/ パッケージ(日本語) を継承しています。 PackageはChiselのコードを名前空間に整理します。PackageにはSub-Packageを含めることもできます。 targetフォルダには、クラスファイルやその他の生成ファイルが格納されています。また、生成されたVerilogファイルを格納するフォルダは、通常はgeneratedと呼ばれます。

Chiselの名前空間機能を使用するには、次のmypacketのように、クラス/モジュールがパッケージで定義されていることを宣言する必要があります。

```
package mypack

import chisel3._

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

この例では、Chisel クラスを使用するために chisel3パッケージをインポートしていることに注意してください。

別のコンテキスト（パッケージ名前空間）でAbcモジュールを使用するには、パッケージmypacketコンポーネントがインポートされる必要があります。アンダースコア（_）はワイルドカードとして機能します。つまり、mypacketのすべてのクラスがインポートされていることを意味します。

```
import mypack._

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

  val abc = new Abc()

}
```

mypacketからすべてのタイプをインポートしないことも可能です。その場合は、完全修飾名mypack.Abcを使用して、パッケージmypack内のAbcモジュールを参照します。

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

  val abc = new mypack.Abc()

}
```

また、単一のクラスだけをインポートしてインスタンスを作成することも可能です。

```
import mypack.Abc

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

```
  val abc = new Abc()

}
```

3.1.2 sbt の実行

Chiselプロジェクトは、シンプルなsbtコマンドでコンパイルして実行することができます。

```
$ sbt run
```

このコマンドは、ソースツリー内のすべてのChiselコードをコンパイルするとともに、mainメソッドを持つobjectを含むクラス、もしくは単純にAppを拡張したクラスを検索します。もし複数のオブジェクトが存在する場合、すべてのオブジェクトがリストされ、その中から1つを選択することができます。 sbtへのパラメータとして、実行するオブジェクトを直接指定することもできます。

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

デフォルトsbtの検索対象は、ソースツリーのmain部分のみで、testは含みません。[2] しかしながら、Chiselのテスタは、ここで説明するように、mainを含みながらも、ソースツリーのtest部に配置されます。したがって、テスタツリー内のmainを実行するには下記のsbtコマンドを用います。

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

以上で、私たちはChiselプロジェクトの基本的な構造と、sbtを使ってコンパイルと実行する方法について理解しました。引き続き、簡単なテストフレームワークについて見てみましょう。

3.1.3 ツールの実行フロー

図 3.2 は、Chiselのツールフローを示しています。デジタル回路はHello.scalaとして示されるChiselのクラスで記述されています。 Scalaのコンパイラは、ChiselとScalaのライブラリと一緒にこのクラスをコンパイルし、標準の Java virtual machine (JVM)(英語)/ Java仮想マシン(日本語) で実行できるJavaクラスHello.classを生成します。 Chiselドライバでこのクラスを実行すると、 FIRRTL (flexible intermediate representation for RTL、デジタル回路の中間表現）を生成します。この例では、Hello.fir ファイルになります。 FIRRTLコンパイラがこれを回路(Verilog RTL)に変換します。

Treadleは回路をシミュレートするFIRRTLインタプリタです。 Chiselテスタと合わせて用いることで、Chiselの回路のデバッグとテストが出来ます。アサーションを用いることでテスト結果を確認できます。 Trieadleは波形ファイル（Hello.vcd）を生成することができ、生成された波形は波形ビューア(無料のビューアであるGTKWaveまたはModelSimなど)で表示できます。[3]

[2] これは、JavaやScalaではテストフォルダが単体テストしか含まず、mainをもつオブジェクトを含まない慣例からきています。

[3] 訳注：図 3.2 は若干不正確です。 VCDの生成はこのFIRRTLのエンジンではなく、生成されたVerilogを元にVerilatorが行います。

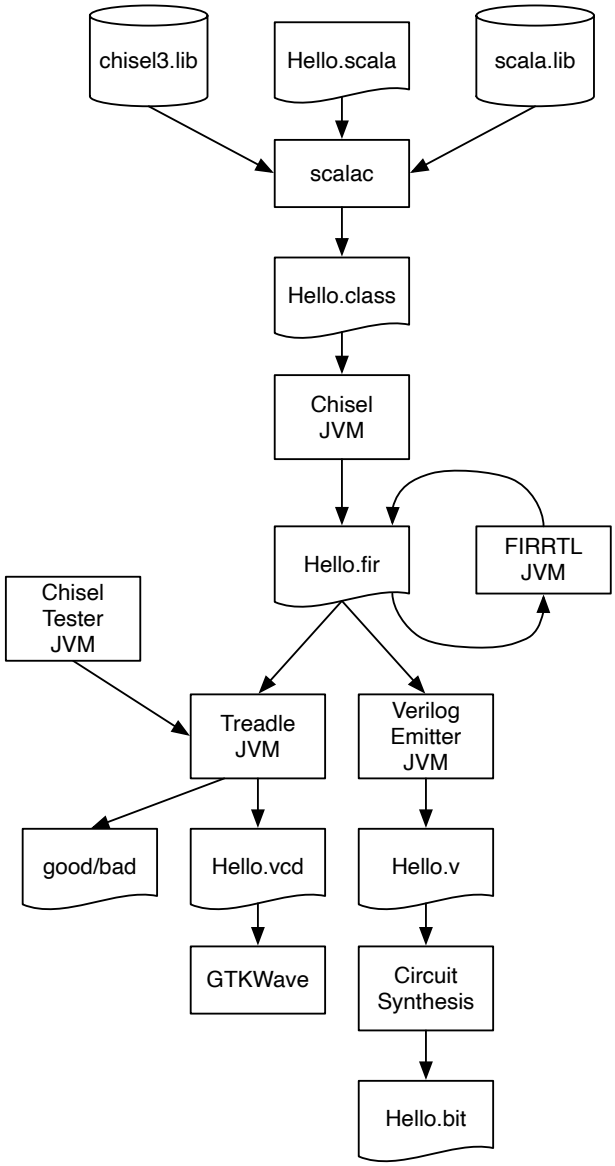


図 3.2: Chiselエコシステムのツールフロー

FIRRTLでの変換の一つは、Verilog Emitter JVMによる、論理合成用のためのVerilogコード（`Hello.v`）の生成です。論理回路の合成ツール(インテルのQuartus、ザイリンクスVivado、またはASICツール)で回路を合成します。FPGA設計フローでは、これらのツールはFPGA構成用のビットストリーム `Hello.bit` を生成します。

3.2 Chisel をつかったテスト

ハードウェア設計のテストは test benches、テストベンチと呼ばれます。テストベンチは、テスト対象となる design under test（DUT）と呼ばれる部分をインスタンス化して、入力ポートに値をセットして、出力ポートに出てくる値と期待値とを比較します。

3.2.1 PeekPokeTester

Chiselでは`PeekPokeTester`の形でテストベンチを提供しています。Chiselの強みの一つは、Scala言語のパワーをフルに活用してテストベンチを記述できることです。例えば、ハードウェアの振る舞いをシミュレートするソフトウェアを用いて、ハードウェアのシミュレーション（訳注：テスト実行のこと）と動作を比較することができます。この方法で、プロセッサの実装のテストを効率的に実施できます [6]。

`PeekPokeTester`を使用するには、以下のパッケージをインポートする必要があります。

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

回路のテストには、少なくとも次の3つのコンポーネントが含まれます：(1) テスト対象、device under test（よくDUTと略される）(2) テストベンチと呼ばれるテスト回路 (3) テストを駆動する`main`関数を含むテスタオブジェクト（訳注：これはChisel特有）

次のコードは、テスト対象となるシンプルなデザインを示しています。このコードには2つの入力ポートと 1 つの出力ポートがあり、すべて 2 ビット幅です。この回路は、ビット単位のANDを使って、出力を返します。

```
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
}
```

このDUTのためのテストベンチは、`PeekPokeTester`を拡張（Extend）し、コンストラクタに渡すパラメータとしてDUTを持ちます。

```
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はpoke()で入力値を設定し、peek()で出力値を読み出すことが出来ます。テスタはstep(1)でシミュレーションを1ステップ（= 1クロックサイクル）進めます。また、println()をつかって、出力の値を表示させることができます。

以下のテスタメイン(main)でテストを作成して実行します。

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

テストを実行すると、結果が（他の情報と共に）端末に出力されます。

```
[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
```

0 AND 1の結果は 0、3 AND 2の結果は2 となります。プリントアウトの目視確認は、最初としては良いのですが、出力ポートの値の期待値をパラメータとして与えることで、expect()を使い、テストベンチ自体で期待値をチェックさせることが出来ます。次の例でexpect()を使ったテストを示します。

```
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)
}
```

このテストを実行しても、ハードウェアからなにか値が表示されることはありませんが、すべてのテストが期待値通りの値でパスしたことは出力されます。

```
[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
```

DUT またはテストベンチのいずれかにエラーが含まれていてテストに失敗すると、期待値と実際の値の違いを記述したエラーメッセージが表示されます。以下では、テストベンチでエラーとなるように、期待値を4に変更しました。

```
[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
```

ここでは、Chiselを使った簡単なテストのための基本的なテスト機能について説明しました。しかし、Chiselでは、フルパワーのScalaでテストを記述することができます。

3.2.2 ScalaTest の利用

ScalaTestはScala（とJava）のテストツールですが、Chiselテスタの実行にも使えます。使い方は、`build.sbt`の中で下記のようにしてライブラリを追加します。

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

テストは通常 `src/test/scala` の中に置かれており、以下で実行することができます：

```
$ sbt test
```

Scalaの整数足し算をテストするためのミニマムテスト (テスト用のハローワールド) は次のようなものです。

```
import org.scalatest._

class ExampleSpec extends FlatSpec with Matchers {

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

Chiselのテストは、Scalaプログラムのユニットテストよりも重くなりますが。 Chiselテストを ScalaTestクラスでラップすることができます。前に示した`Tester`は、次のようになります：

```
class SimpleSpec extends FlatSpec with Matchers {

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

この演習の主な利点は、(`main`を実行する代わりに) 簡単な`sbt test`ですべてのテストを実行できることです。次のようにすることで、`sbt`で1つのテストのみを実行することもできます。

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

3.2.3 波形表示

以上で紹介したテスタは、ソフトウェアの開発と同様に、小さなデザインや、 unit testing(英語)/ 単体テスト(日本語) に対してうまく働きます。 一連のユニットテストは regression testing(英語)/ 回帰試験(日本語) にも役立ちます。

しかし、より複雑なデザインをデバッグする場合、複数の信号を一度に調査したい場合があります。 デジタル回路をデバッグするための古典的なアプローチは、信号を波形で表示することです。 波形では、信号は時間の経過とともに表示されます。

Chisel テスタは、すべてのレジスタとすべてのIO信号を含む波形を生成することができます。 以下の例では、前の例（2ビットのAND関数）の`DeviceUnderTest`の波形テスタを示します。 この例では、以下のクラスをインポートしています。

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

まず、入力に値を入れて、`step`でクロックを進めるだけの簡単なテスタから始めます。 出力を読み込んだり、比較したりしません。

```
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)
}
```

その代わりに、パラメータを指定して、波形ファイル(.vcdファイル)を生成するために、`Driver.execute`を呼び出します。

```
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)
  }
}
```

波形の表示には、フリーのGTKWaveや（商用の）ModelSimが使えます。 GTKWaveを起動し、*File – Open New Window*を選択して、Chiselテスタが生成した`.vcd`ファイルを含むフォルダを探します。 標準では、生成されたファイルは、`test_run_dir`にテスタの名前に番号を付加した形で保存されています。 そのフォルダに、`DeviceUnderTest.vcd`ファイルがあるはずです。 左側から信号名を選び、メインウインドウにペーストします。 設定を保存したい場合は*File – Write Save File*で保存します、読む出す際は*File – Read Save File*で読み出します。

考え得る全ての入力値を明示的に列挙するのは現実的ではありません。そのため、DUTを駆動するためにいくつかのScalaコードを使用します。以下のテスタは、2つの2ビットの入力信号に対して可能なすべての値を列挙します。

```
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)
    }
  }
}
```

この新しいテスタのために ScalaTest の仕様を追加します。

```
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)
  }
}
```

以下のように実行します。

```
$ sbt "testOnly WaveformCounterSpec"
```

3.2.4 printf デバッグ

別のデバッグの方法は、いわゆる“printfデバッグ”です。この方法は、単にプログラムの実行中に気になる変数を表示するように、Cのコードに printf文を仕込みます。同じことが、Chiselの回路のテストでも出来ます。出力はクロックの立ち上がりに実行されます。printf文は、モジュール定義の中のどこにでも仕込むことができます。 printf デバッグ版のDUTは以下のようになります。

```
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)
}
```

すべての可能な値を繰り返し処理するカウンタベースのテスタを使ってこのモジュールを

テストすると、以下のような出力が得られます。AND関数が正しいことが確認できます。

```
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
```

ChiselのprintfはC and Scala style formattingをサポートしています。

3.3 演習

この演習では、chisel-examplesのLED点滅回路を使い、Chiselのテストを試します。

3.3.1 最小のプロジェクト

まず、最小限のChiselプロジェクトについて見てみます。Hello Worldのファイルを見てみましょう。`Hello.scala`は、単一のハードウェアのソースファイルです。これにはLED点滅のハードウェア記述(`class Hello`)と、Verilogコードを生成する`App`が含まれます。
各ファイルは、Chiselと関連パッケージのインポートから始まります。

```
import chisel3._
```

そして、コード 1.1に示されるようなハードウェア記述が続きます。Verilogコードを生成するためにはアプリケーションが必要です。`App`を拡張したScalaのオブジェクトは、そこからアプリケーションが開始するmain関数を暗黙的に生成するアプリケーション (プログラム)です。このアプリケーションの唯一の仕事は、新しい`HelloWorld`オブジェクトを作成して、Chiselドライバの`execute`関数に渡すことです。最初の引数は、文字列の配列でビルドオプションがセットされます（例、出力フォルダ)。以下のコードはVerilogファイル`Hello.v`を生成します。

```
object Hello extends App {
  chisel3.Driver.execute(Array[String](), () => new Hello())
}
```

下記の実行を行って、手動でサンプルを生成します。

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

次に、生成されたHello.vファイルをエディタで見てみましょう。生成されたVerilogコードはあまり読みやすくはありませんが、いくつかのことが分かります。ファイルはChiselモジュールと同じ名前の Hello モジュールで始まります。 LEDポートはoutput io_ledとして割り当てられています。ピンの名前はChiselでの名前にプレフィックスとして io_ が付きます。モジュールには、LEDピンの他に、clockとreset の入力信号が含まれます。これら二つの信号は、Chiselによって自動的に追加されます。

さらに、2つのレジスタcntReg と blkReg、の定義を確認することができます。また、モジュールの定義の最後に、これらのレジスタのリセットとアップデートを見つけることができるかもしれません。 Chiselは、同期リセットを生成することに注意してください。

sbtが正しいScalaのコンパイラやChiselライブラリを取得するためには、build.sbt ファイルが必要です。

```
scalaVersion := "2.11.7"

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

libraryDependencies += "edu.berkeley.cs" %% "chisel3" % "3.2.2"
libraryDependencies += "edu.berkeley.cs" %% "chisel-iotesters" % "1.3.2"
```

この例で注意してもらいたいのは、具体的なChiselバージョン番号を指定していて、新バージョンのチェック（インターネットに接続されていない場合に失敗、例えば、飛行機で旅行中にハードウェア設計をしている時など）はしないことです。 build.sbtのライブラリの依存関係の設定を変更すれば、最新のChiselバージョンを使用できます。

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

その後、sbtでビルドを再実行してください。もし新しいバージョンのChiselがあれば、自動的にダウンロードされていますか？

便宜上、プロジェクトには Makefile も含まれています。中に sbtコマンドが含まれており、そのsbtコマンド名を覚えてなくても Verilog コードを生成することができます。

```
$ make
```

READMEファイル以外にも、例題プロジェクトにはいくつかのFPGA向けのプロジェクトファイルが含まれています。例えば、quartus/altde2-115 の中には、 DE2-115ボード用のQuartusプロジェクトファイルが2つ含まれています。メインの設定（ソースファイル、デバイス、ピンアサイン）は、テキストファイルである hello.qsfで定義されています。ファイルを見ると、どの信号がどのピンに割り当てられているかが分かります。もし、別のボードにプロジェクトを変更させる必要がある場合は、その部分を修正します。すでにQuartusがインストールされている場合は、そのプロジェクトファイルを開き、緑色の*Play*ボタンでコンパイルしたのち、FPGAをコンフィグします。

*Hello World*はミニマルなChiselのプロジェクトであることに注意してください。より現実的なプロジェクトではソースファイルはパッケージに整理され、テスタが含まれます。次の演習では、このようなプロジェクトについて練習します。

3.3.2 テストの演習

前章の演習では、LED点滅の例をANDゲートやマルチプレクサの入力などを構成して拡張し、このハードウェアをFPGAで実行しました。私たちは、この例を使って、Chiselテスタで機能をテストし、テストを自動化するとともに、FPGAに依存しないようにしていきます。前章からあなたのデザインを使用し、機能をテストするためにChiselテスタを追加してみましょう。すべての可能な入力を列挙し、`expect()`で出力をテストしてみてください。

Chisel内でテストを行うことで、デザインのデバッグを高速化することができます。ただし、FPGA用にデザインを合成し、FPGAでテストを実行することは常に良いアイデアです。そうすることで、デザインのサイズ（通常はLUTとフリップフロップの数）と、デザインのパフォーマンスを最大クロック周波数という指標で、確認できます。例えば、教科書的なパイプラインを構成を持つ RISC プロセッサの場合、約3000個の 4ビットLUT を使います。低コストなFPGA (Intel Cyclone またはXilinx Spartan) 上で100MHz程度で動作します。

4 コンポーネント

大規模なデジタル回路設計では、多くの場合、複数のコンポーネントから階層的に構成されます。各コンポーネントには、通常ポートと呼ばれる入力および出力ワイヤを備えたインターフェースがあります。これらは、集積回路 (IC) の入出力ピンに似ています。コンポーネントは、入力と出力を配線することで接続されます。コンポーネントは、階層を構築するためにサブコンポーネントを含むことがあります。チップ上の物理ピンに接続されている一番外側のコンポーネントをトップレベルコンポーネントと呼びます。

図 4.1 に設計例を示します。コンポーネントCは3つの入力ポートと2つの出力ポートを持っています。このコンポーネント（CompC）自体は、CompAとCompBの2つのサブコンポーネントで構成されており、それぞれの入出力端子はCompCの入力と出力に接続されています。 CompAの出力の 1 つはCompBの入力に接続されています。 CompDは、CompCと同じ階層レベルにあり、CompCに接続されています。

この章では、Chiselでコンポーネントがどのように記述されているかを説明し、標準コンポーネントのいくつかの例を示します。これらの標準コンポーネントには2つの目的があります。(1) Chiselコードの例を提供すること、(2) 設計で再利用できるコンポーネントのライブラリを提供することです。

4.1 Chisel のコンポーネントはモジュール

ハードウェアコンポーネントは、Chiselではモジュール(module)と呼ばれています。各モジュールは、`Module`クラスを継承(Extend)します。また、`io`フィールドがインターフェースのために含まれます。 `IO()`への呼び出しをラップした`Bundle`によってインターフェースは定義されます。 `Bundle`には、モジュールの入力および出力ポートを表すためのフィールドが含まれます。信号の方向はフィールドをラップする`Input()`又は`Output()`で設定します。信号の方向は、コンポーネントから見たものになります。

以下のコードでは、図 4.1からコンポーネントAとBの2つの例での定義を示します：

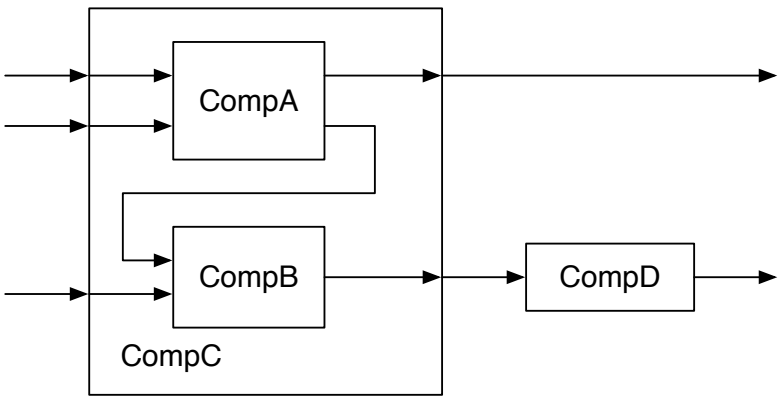


図 4.1: コンポーネントが階層構造を持つ回路デザイン

```
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
}
```

CompAは、aとbという2つの入力と、xとyの2つの出力を持ちます。 CompBのポートには、in1 in2, と out という名前を用います。すべてのポートは、8ビット幅を持つ符号なし整数（UInt）を使用します。この例のコードはコンポーネントを接続して階層を構築するものなので、コンポーネント内での実装は示していません。コンポーネントの実装は、コメントに“function of X” と書かれているところに記述します。これらの例のコンポーネントには関連する関数がないため、一般的なポート名を使用していますが、実際のデザインでは、dataやvalid、readyのような意味のあるポート名を使用します。

CompCは3つの入力ポートと2つの出力ポートを持っており、CompAとCompBから構成されています。ここでは、CompAとCompBがCompCのポートにどのように接続されているか、また、CompAの出力ポートとCompBの入力ポートの間の接続を示します。

```
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
}
```

コンポーネントは、new CompA()のようにnewを使って生成したインスタンスをModule()でラップする必要があります。そのモジュールへの参照は、ローカル変数に格納されます。この例では、val compA = Module(new CompA())です。

この参照を用いて、モジュールのioフィールドとIO Bundleの個々のフィールドを間接参照 (dereferencing) することで、IOポートへのアクセスが可能になります。

このデザインの中で最も単純なコンポーネントは、入力ポート（in）と出力ポート（out）を持っています。

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

  // function of D
}
```

この例のデザインの最後の欠けている部分は、トップレベルのコンポーネントです。コンポーネントCとDから組み立てます。

```
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
}
```

コンポーネントの優れた設計は、ソフトウェア設計における関数やメソッドの優れた設計に似ています。主な問題の一つは、コンポーネントにどれだけの機能を持たせるか、コンポーネントはどれだけ大きくすべきかということです。両極端なのは、加算器のような小さなコンポーネントと、フルマイクロプロセッサのような巨大なコンポーネントです。

ハードウェアデザインの初心者は、小さな部品から始めることが多いです。問題は、デジタル回路設計の本では、原理を示すために小さな部品を使っていることです。そのよう

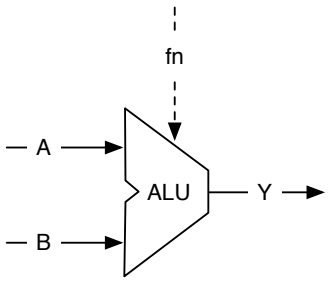


図 4.2: 算術論理演算ユニット (ALUと略される）

な例題のサイズは（そうした本でも、本書でも)、ページに収める事と気が散らないようにするために、詳細な設計を省いています。

コンポーネントのインターフェースは、少し冗長です（型、名前、方向性、IOの構築などがあります)。経験則として、私が提案するのは、コンポーネントのコアである関数は、少なくともコンポーネントのインターフェースと同じくらいの長さであるべきだということです。

カウンターのような小さなコンポーネントに対して、Chiselではそれらをより軽量に表現する方法として、ハードウェアを返す関数を提供しています。

4.2 算術論理ユニット

マイクロプロセッサなどの演算回路の中心的な構成要素の一つに算術論理演算ユニット arithmetic-logic unit(英語)/ 演算装置(日本語) （ALU）があります。図 4.2 にALUのシンボルを示します。

ALU は、図中のAとBの 2 つのデータ入力と、1 つの関数入力fnと出力の Yの持ちます。ALU は、AとBを演算し、結果を出力します。入力fnは、AとBの演算の種類を選択します。演算は通常、足し算や引き算などの演算や、and, or, xorなどの論理関数です。そのため ALU (算術論理演算ユニット)と呼ばれます。

関数入力fnは、演算を選択します。ALUは通常、状態要素(ラッチ)を持たない組合せ回路です。また、ALUは、ゼロや符号のような、追加の出力を、演算結果のプロパティとして持つ場合もあります。

次のコードは、16 ビットの入出力を持つ ALU で、足し算、引き算、OR、ANDをサポートしています。演算の種類は、2ビットのfn信号で選択します。

```
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 }
  }
}
```

この例では、新しいChiselの構文である`switch/is`を使用して、ALUの出力を選択するテーブルを記述しています。このユーティリティ関数を使用するには、以下のように別のChiselパッケージをインポートする必要があります。

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

4.3 バルク接続

複数のIOポートでコンポーネントを接続するために、Chiselはバルク接続演算子<>を提供します。この演算子は、バンドルに含まれる各要素を双方向に接続します。 Chiselはリーフフィールドの名前を使って接続します。名前がない場合は接続されません。

例として、パイプライン型のプロセッサを構築したとします。フェッチステージは以下のようなインターフェースを持っています。

```
class Fetch extends Module {
  val io = IO(new Bundle {
    val instr = Output(UInt(32.W))
    val pc = Output(UInt(32.W))
  })
  // ... Implementation od fetch
}
```

次のステージは、デコードステージです。

```
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
}
```

シンプルなプロセッサの最終段階は、実行ステージです。

```
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
```

```
}
```

3つのステージをすべて接続するために必要なのは、たった2つの<> 演算子だけです。そして、サブモジュールのポートを親モジュールに接続することにも使えます。

```
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 関数を用いた軽量コンポーネント

モジュールは、ハードウェアの記述を構造化するための一般的な方法です。しかし、モジュールを宣言するときや、インスタンス化して接続するときには、いくつかの定型的なコードがあります。関数を使用することでハードウェアを軽量に構造化できます。 Scala の関数は Chisel (および Scala) のパラメータを受け取り、生成されたハードウェアを返すことができます。簡単な例として、以下の例では加算器を生成します。

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

関数 adderを呼び出すだけで、2つの加算器を作成することができます。

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

これは、ハードウェアジェネレータ であることに注意してください。ハードウェア生成工程では、加算演算を実行しているのではなく、加算器(のハードウェアインスタンス)を２つ作成します。この例ではわざと加算器を生成しましたが、 Chiselには既に、+(that: UInt)のような加算器生成機能があります。

さらに、軽量なハードウェア生成器として関数には、ステート（レジスタを含む）を含むこともできます。以下の例では、1クロックサイクルの遅延要素（レジスタ）を生成しています。関数が1つの文だけの場合は、1行で記述して中括弧()を省略することができます。

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

関数自体をパラメータとして関数を呼び出すことで、2クロックサイクルの遅延が発生しました。

```
val delOut = delay(delay(delIn))
```

繰り返しになりますが、これはあまりにも短い例で、RegNext()がすでに遅延のためのレジスタを作成する関数なので、有用なものではありません。

関数は、モジュール の一部として宣言することができます。ただし、異なるモジュールで使用する関数は、ユーティリティ関数を集めたScalaオブジェクトの中に入れた方が良いでしょう。

5 組合せ回路ブロック

この章では、より複雑なシステムを構築するための基本的な構成要素である様々な組合せ回路を探求します。原則として、すべての組合せ回路はブール式で記述することができます。しかし、多くの場合、表の形で記述する方が効率的です。我々は、合成ツールにブール式を抽出して最小化することを任せています。表形式で記述するのに最適な基本回路は、デコーダとエンコーダの2つです。

5.1 組合せ回路

いくつかの標準的な組み合わせのビルディングブロックを説明する前に、組み合わせ回路がChiselでどのように表現できるかを探ってみましょう。最も単純な形式はブール式で、名前を割り当てることができます。

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

ブール型の式は、Scalaの値に代入することで名前(e)が与えられます。この式は他の式で再利用することができます。

```
val f = ~e
```

このような式は、再代入できなくなります。valを使ったeへの、=による再代入はScalaではコンパイラエラーになります。Chiselの演算子である:=を使って、次に示すコードを試してみてください。

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

これは”読み込み専用の変数への再代入は出来ない”という理由で、実行時例外が発生します。

Chiselでは条件付きで更新を行う組み合わせ回路記述もサポートされています。このような回路はWireとして宣言されます。この回路の論理を記述するためには、whenのような条件分岐の構文を使います。次のコードでは、wというUInt型のWireを宣言し、デフォルト値を0に設定しています。whenブロックはChiselのBool型を引数にとり、condがtrue.Bのときに、wは3になります。

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

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

この回路の論理は、2つの定数0と3を入力とし、condを選択信号とするマルチプレクサです。条件付き実行を行うソフトウェアプログラムではなく、ハードウェア回路を記述していることを心に留めておいてください。

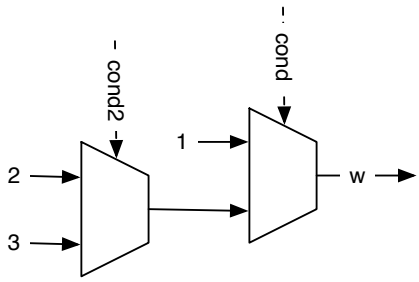


図 5.1: 2つのマルチプレクサのチェーン

Chiselの条件構文whenにもelseに相当する.otherwiseと呼ばれるものがあります。特定の条件下で値を設定することで、デフォルト値の代入式を省略できます。

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

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

Chiselは連続した条件分岐（if/elseif/else）を .elsewhen でサポートしています。

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

when (cond) {
  w := 1.U
} .elsewhen (cond2) {
  w := 2.U
} .otherwise {
  w := 3.U
}
```

このwhen、 .elsewhen、 .otherwiseのチェーンは、2つのマルチプレクサのチェーンになります。図 5.1は、このマルチプレクサを示しています。このチェーンは優先順位を持っていて、例えばcondが真になった時、その他の条件は評価されません。

Scalaで連続してメソッドを呼び出す際には、.elsewhenの '.' が必要となる点に注意してください。この.elsewhenの分岐は、必要な分だけ長く出来ます。しかしながら、条件分岐の条件が単一の信号に依存する場合には、switch文を使うほうが良いでしょう。これについて次節のデコーダで紹介します。

より複雑な組合せ回路の場合には、Wireにデフォルト値を割り当てることが実用的かもしれません。宣言時にデフォルト値を設定する場合には、WireDefaultを使うことができます。

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

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

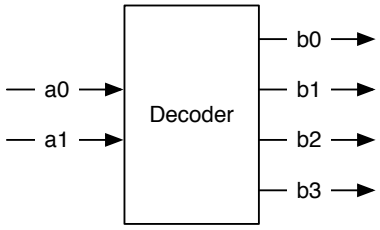


図 5.2: 2-bitから4-bitデコーダ

a	b
00	0001
01	0010
10	0100
11	1000

表 5.1: 2-bitから4-bitデコーダの真理値表

```
  // ... and some more complex conditional assignments
```

質問として考えられそうなことの1つは「Scalaにはif、 else if、 elseという構文があるのに、なぜwhen、 .elsewhen、 .otherwiseを使うのか?」ということです。これらの構文はScalaの条件分岐処理であり、Chiselのハードウェア（マルチプレクサ）を生成するものではありません。これらのScalaの条件分岐は、パラメータを用いて条件によって異なるハードウェアを生成する回路ジェネレータを作成する際に使用されます。

5.2 デコーダ

decoder(英語)/ デコーダ(日本語) は、nビットの2進数を$m \leq 2^n$であるようなmビットの信号に変換します。その出力はワンホットにエンコードされたもの（特定の1bitだけが1）になります。

図 5.2は、2ビットから4ビットのデコーダを示しています。このデコーダの機能は、表 5.2のような真理値表として表現できます。

Chiselのswitch文は、真理値表のような論理を記述します。switch文は、Chiselの言語機能の一部ではありません。そのため使用する際には、パッケージchisel3.utilをインポートする必要があります。

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

次のコードは、Chiselのswitch文で記述したデコーダを紹介するためのものです。

```
  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}
```

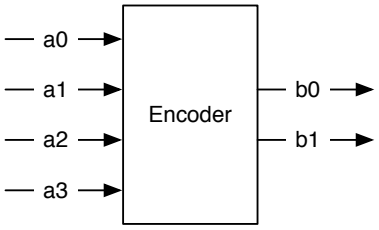


図 5.3: 4-bitから2-bitエンコーダ

```
  }
```

上記のswitch文ではselが取りうる値をすべてリストアップし、それらすべてのケースでresultにデコード後の値を割り当てています。 Chiselではたとえswitch文中で、可能性のあるすべての値を列挙したとしても、デフォルトの値を割り当てる必要がある点に注意してください。上記のコードではresultに0を割り当てている部分が該当しています。この割当ては決して有効にならないため、バックエンドの最適化において削除されます。これはVHDLやVerilogのようなハードウェア記述言語において、組み合わせ回路（Chiselではwire）での不完全な割り当てが、意図しないラッチを生成することがあることを避けるためのものです。Chiselではこのような不完全な割り当ては許容されません。

前の例では、信号に符号なし整数を使用していました。エンコード回路をより明確に表現するためには、2進数表記を使用した方が良いかもしれません。

```
  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}
  }
```

テーブルはデコーダの機能をとても読みやすく表現しますが、少し冗長でもあります。このテーブルを調べてみると、1をselだけ左にシフトした値となるという規則に気付きます。この規則を利用すると、先程のデコーダはChiselのシフト演算子<<を使って次のように表現できます。

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

デコーダはその出力をイネーブル信号として、マルチプレクサへのデータ入力とともにANDゲートの入力に使用することで、マルチプレクサを構成するブロックとして使用されます。しかし、Chiselのコアライブラリには、マルチプレクサを実装したMuxが用意されているため、自分でマルチプレクサを構築する必要はありません。デコーダはアドレスのデコードにも使用されます。その出力を、例えば、マイクロプロセッサに接続される異なる種類のIOデバイスを選択する信号として使用します。

5.3 エンコーダ

encoder(英語)/ エンコーダ(日本語) はワンホットな入力信号をバイナリの出力信号に変換します。エンコーダはデコーダの逆の処理を行います。

図 5.3は、4ビットのワンホットな入力を2ビットのバイナリに変換したものを示してお

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

表 5.2: 4-bitから2-bitエンコーダの真理値表

り、表 5.3は、そのエンコードの真理値表です。しかし、エンコーダは入力信号がワンホットな場合にのみ、期待通りに動作します。その他のすべての入力については、出力値は未定義となります。未定義の出力をもった機能を実装することは出来ないので、未定義となる入力のパターンを処理するためのデフォルト値を割り当てます。

以下のChiselコードでは、デフォルト値の00を代入してから、switch文を使用して正規の入力値を指定しています。

```
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 演習

4-bitのバイナリ入力を 7-segment display(英語)/ 7セグメントディスプレイ(日本語) のエンコードに変換する組み合わせ回路を実装してください。7セグメントディスプレイの当初の使い方であった10進数の表示を行うためのコードか、それに加えて hexadecimal(英語)/ 十六進法(日本語) に記載されている残りのパターンを含んだ、16種類の値の表示を行うコードを実装しても構いません。もし7セグメントディスプレイが搭載されているFPGAを持っているなら、実装した回路の入力を4つのスイッチ、もしくはボタンに接続し、出力を7セグメントディスプレイに接続しましょう。

6 順序回路ブロック

順序回路は、出力が入力と以前の値に依存する回路です。本書では同期設計（クロックドデザイン）を前提としており、順序回路は同期式順序回路を意味します。[1] 順序回路を構築するためには、状態を格納できる要素が必要で、それはレジスタと呼ばれます。

6.1 レジスタ

順序回路を構築するための基本的な要素は、レジスタです。レジスタは D flip-flops(英語)/D型フリップフロップ(日本語) の集まりです。 Dフリップフロップは、クロックの立ち上がりエッジで入力の値をキャプチャして保持し、出力します。別の言い方をすれば、レジスタはクロックの立ち上がりエッジの入力値で出力を更新します。

図 6.1は、レジスタの回路シンボルを表し、入力のDと出力Qが含まれています。各レジスタはclock信号の入力も持ちます。このグローバルクロック信号は通常、同期回路内のすべてのレジスタに接続されているので、回路図に描かれていません。箱の下部にある小さな三角形は、クロック入力を表しており、これはレジスタであることを示しています。本書では以降、クロック信号の表記を省略します。レジスタのクロック入力へ明示的な接続が必要ないChiselでも、グローバルクロック信号を省略します。

Chiselの入力dと出力qを持つレジスタ定義は次のようになります:

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

レジスタにクロックを接続する必要はありません。 Chiselは暗黙のうちにこれを行います。レジスタの入力と出力は、ベクトルとバンドルの組み合わせで作られた任意の複雑な型にすることができます。

また、レジスタは次の2段階で定義して使用することもできます。

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

  delayReg := delayIn
```

まず、レジスタを定義し、名前を付けます。次に、レジスタの入力にdelayIn信号を接続します。

レジスタの名前は、文字列Regが含まれていることにも注意してください。簡単に組み合わせ回路と順序回路を区別するために、名前の一部にマーカーとしてRegをつけておくのが一般的です。また、Scala (よってChiselも) の命名規則は通常 CamelCase(英語)/ キャメルケース(日本語) が使われます。変数名は小文字で始まり、クラスは大文字で始まります。

レジスタはリセット時に初期化することができます。 reset信号は、clock信号と同じく、Chiselでは暗黙的です。レジスタコンストラクタのRegInitにパラメータとしてリセット値、例えばゼロを、与えます。レジスタへの入力は、Chisel代入文で行います。

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

[1]非同期ロジックやフィードバックで順序回路を構築することができますが、これは特定のニッチな話題で、Chiselで表現することはできません。

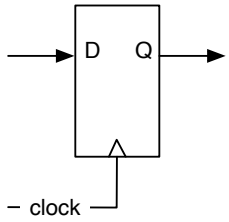


図 6.1: D フリップフロップを使ったレジスタ

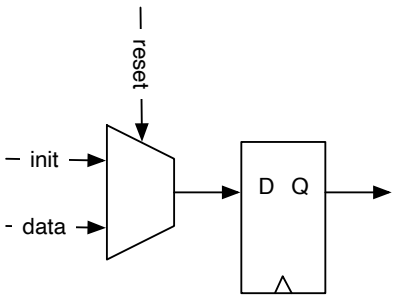


図 6.2: 同期リセットを持つ D フリップフロップを使ったレジスタ

```
valReg := inVal
```

Chiselではデフォルトのリセットは同期リセットです。[2] 同期リセットの場合は、Dフリップフロップの変更は必要なく、入力にマルチプレクサ[3]を追加してリセット時の初期化値とデータ値を選択するだけです。

図 6.2は、リセットでマルチプレクサを駆動する同期リセット付きレジスタの回路図を示しています。しかし、同期リセットはかなり頻繁に使用されるため、今時のFPGAのフリップフロップには同期リセット(およびセット)入力が含まれており、マルチプレクサのためにLUTを無駄にしないようになっています。

順序回路は、時間の経過とともにその値が変わります。そのふるまいは、時間の経過とともに変化する信号を示す図によってみることができます。このような図は、波形または timing diagram(英語)/ タイミング図(日本語) と呼ばれています。

図 6.3に、リセットと一部の入力データが適用されたレジスタの波形を示します。時間は左から右へと進みます。図の一番上には、回路を駆動するクロックが表示されています。リセット前の最初のクロックサイクルでは、レジスタの内容が定義されていません。二つ目のクロックサイクルではリセットがHighにアサートされ、このクロックサイクルの立ち上がりエッジ(ラベルB)でレジスタは初期値`0`になります。入力`inVal`は無視されています。次のクロックサイクルでは`reset`は`0`になり、`inVal`の値は次の立ち上がりエッジ(ラベルC)でキャプチャされます。それ以降は`reset`は`0`のままで、レジスタ出力は入力信号から1クロックサイクル遅れで追従します。

波形は、回路の動作をグラフィカルに指定するための優れたツールです。特に、多くの演算が並列に行われ、データが回路内をパイプラインで移動するような複雑な回路では、

[2]非同期リセットはChisel3.2.0から正式にサポートされました。
[3]現在のFPGAフリップフロップには同期リセット入力があるため、マルチプレクサに追加のリソースは必要ありません。

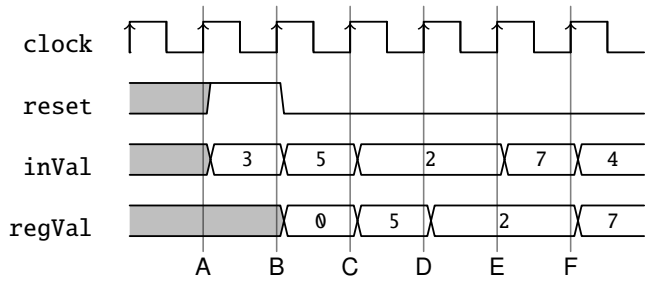


図 6.3: リセット信号を持つレジスタの波形図

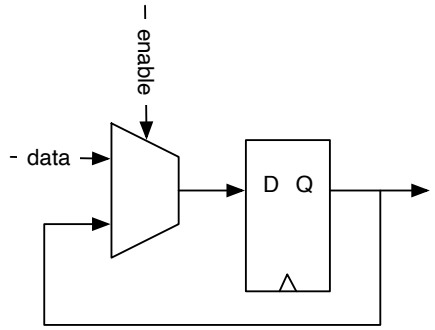


図 6.4: イネーブル信号をもつ D フリップフロップレジスタ

タイミング図が便利です。また、Chiselテスタでは、テスト中に波形を作成し、波形ビューワで表示してデバッグに利用することができます。

典型的なデザインパターンは、イネーブル信号を持つレジスタです。イネーブル信号が`true`(高)の場合のみ、レジスタは入力をキャプチャし、そうでない場合は古い値を保持します。イネーブル信号付きレジスタは、同期リセットと同様に、レジスタの入力にマルチプレクサを使用して実装することができます。マルチプレクサへの入力の1つは、レジスタの出力のフィードバックです。

図 6.4にイネーブル付きレジスタの回路図を示します。これも一般的なデザインパターンであるため、今どきのFPGAフリップフロップには専用のイネーブル入力が含まれており、追加のリソースは必要ありません。

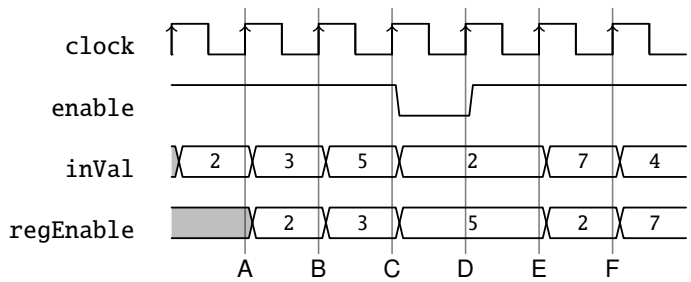


図 6.5: イネーブル信号をもつ D フリップフロップレジスタの波形図

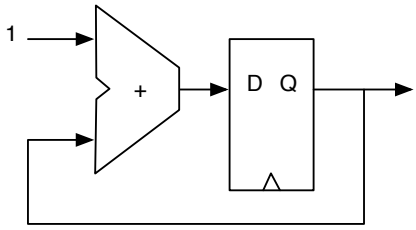


図 6.6: カウンタの中のアダーと結果保持レジスタ

図 6.5イネーブル付きレジスタの波形例を示します。ほとんどのサイクルでenableはHigh (true) で、レジスタの値は1クロックサイクルの遅延で入力に従っています。 4クロックサイクル目にのみenableがLowになり、Dの立ち上がりエッジでレジスタは値(5)を保持します。

イネーブルを持つレジスタは、条件付きアップデートを用いて数行のChiselコードで記述することができます。

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

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

また、イネーブル付きのレジスタをリセットすることもできます。

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

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

レジスタは式の一部にもなります。次の回路は、現在の信号値と直前のクロックサイクルの値とを比較することで、信号の立ち上がりエッジを検出します。

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

レジスタの基本的な使用法をすべて説明したところで、これらのレジスタを活用して、より興味深い順序回路を構築します。

6.2 カウンタ

最も基本的な順序回路の一つはカウンタです。その最も単純な形態では、カウンタ出力が加算器に接続され、加算器の出力がレジスタの入力に接続されているレジスタです。図 6.6は、このようなフリーランニングカウンタを示しています。

4ビットのレジスタからなるフリーランニングカウンタは、0から15までカウントした後、再び0に折り返します。また、カウンタは既知の値にリセットされなければなりません。

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

cntReg := cntReg + 1.U
```

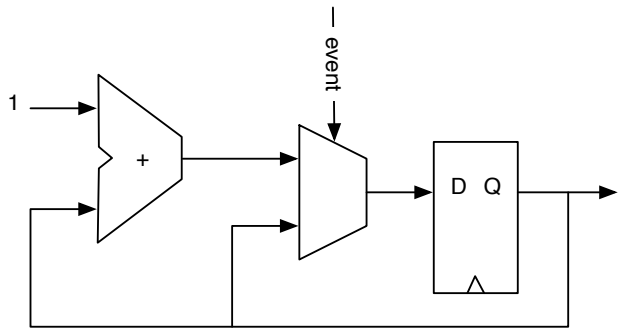


図 6.7: イベントをカウント

イベントをカウントしたいときは、図 6.7と次のコードに示すように、カウンタをインクリメントするための条件を使用します。

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

6.2.1 カウントアップとダウン

値をある値までカウントアップした後、再度0からカウントアップするには、when条件文などを用いてカウンタの値を最大値の定数と比較する必要があります。

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

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

カウンタにはマルチプレクサを使用することもできます。

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

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

カウントダウンする場合、まずカウンタレジスタを最大値でリセットし、0になったらその値にリセットします。

```
val cntReg = RegInit(N)

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

実際のコーディングでは、多くのカウンタを使用するので、カウンタを生成するパラメー

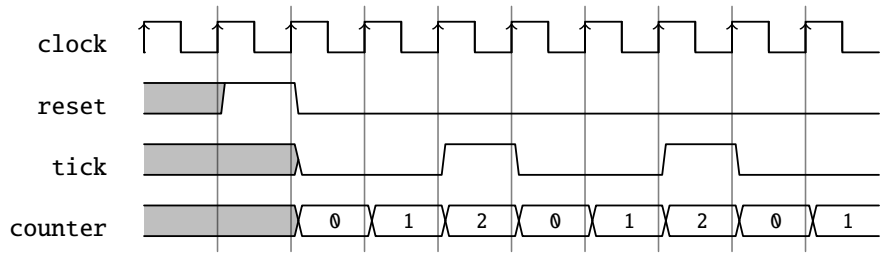


図 6.8: 低速なティック生成の波形図

タ付き関数を定義して使用することができます。

```
// 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)
```

関数(genCounter)の最後の文は、関数の戻り値であり、この例ではカウントレジスタ(cntReg)です。

なお、すべてのカウンタ例で、カウンタが0から、Nを含めた、Nの間の値を取ることに注意してください。 10クロックサイクルをカウントしたい場合は、Nを9に設定する必要があります。このように、カウントと値が1つずれるのは、 off-by-one error(英語)/ Off-by-oneエラー, 植木算(日本語) の古典的な例です。

6.2.2 カウンタによるタイミングの生成

イベントを数える以外にも、カウンタは時間の概念を生成するためによく使われます (壁掛け時計の時刻)。同期回路は一定の周波数のclockで動作しています。回路はそれらのクロックの刻みで進行します。デジタル回路では、クロックの刻みを数える以外に時間の概念はありません。クロックの周波数がわかれば、例えば Chisel の “Hello World” の例で示したようにある周波数で LED を点滅させるような時間的なイベントを発生させる回路を作ることができます。

一般的な方法は、我々の回路に必要な周波数f_{tick}でシングルサイクルのティックを生成することです。このティックはnクロックサイクルごとに発生し、$n = f_{clock}/f_{tick}$で、ティックは正確に1クロックサイクルの長さになります。このティックは派生クロックとしてではなく、論理的に周波数f_{tick}で動作する回路内のレジスタのイネーブル信号として使用されます。図 6.8に3クロック周期で発生するティックの例を示します。

以下の回路では、0から最大値であるN - 1までをカウントするカウンタを記述します。最大値に達すると、1サイクルの間、tickはtrueとなり、カウンタは0にリセットされます。0からN - 1までカウントすることで、Nクロックサイクルごとに 1つの論理ティックを生成します。

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

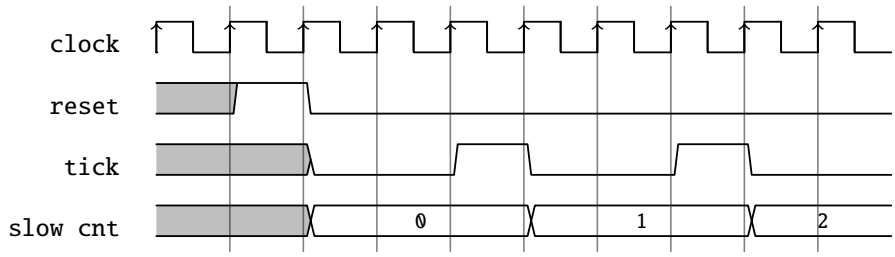


図 6.9: 低速なティックを使ったカウンタの波形図

```
val tick = tickCounterReg === (N-1).U

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

このnクロックサイクルごとの論理的な1ティックを使って、回路の他の一部分を遅く駆動することができます。次のコードでは、nクロックサイクルごとに1ずつインクリメントする別のカウンタを使用しています。

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

図 6.9は、ティックの波形とティックn(クロックサイクル)ごとにインクリメントするスローカウンタの波形を示しています。

この遅い論理クロックの使用例としては、LEDの点滅、シリアルバスのボーレートの生成、7セグメント表示のためのマルチプレクサ信号生成、ボタンやスイッチのデバウンスのための入力値のサブサンプリングなどがあります。

幅推定がレジスタのサイズを決めてくれますが、レジスタ定義時の型や初期化値で明示的に幅を指定した方が良いでしょう。明示的に幅を定義することで、リセット値`0.U`の結果として1ビットの幅を持つカウンタが生成されてしまうようなことがなくなります。

6.2.3 「オタク」カウンタ

多くの人が、たまに Nerd(英語)/ ナード(日本語)/ おたく(日本語) になった気分になることがあります。例えば、カウンタ/ティック生成の高度に最適化された設計をしたいとします。標準的なカウンタは、1つのレジスタ、1つの加算器（または減算器)、およびコンパレータというリソースを必要とします。レジスタや加算器についてはあまり手を加えることができません。カウントアップする場合は、ビット列である数値と比較する必要があります。コンパレータは、ビット列中のzero値のビットにインバータを用いた大きなANDゲートから構築することができます。ゼロまでカウントダウンする場合、コンパレータは大型のNORゲートとなるため、ASICでは組み込んだ定数と比較するものよりも必要リソースが若干少なくてすむかもしれません。ロジックがルックアップテーブルで構築されているFPGAでは、0と比較しようが1と比較しようが違いはありません。リソースの必要量はアップカウンタもダウンカウンタも同じです。

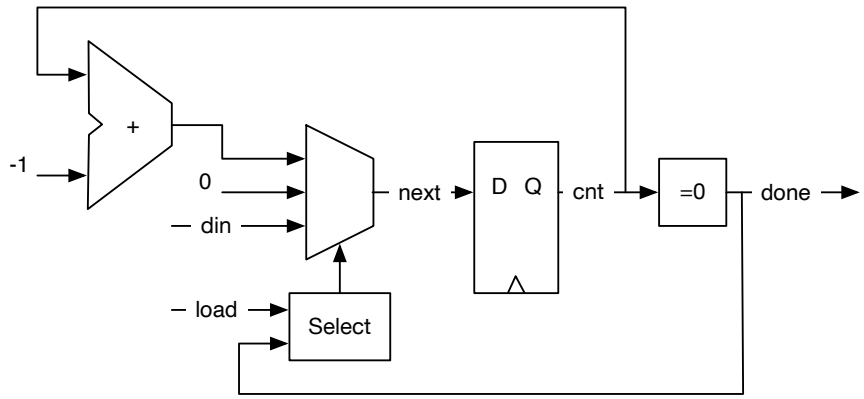


図 6.10: ワンショットタイマ

しかし、巧妙なハードウェア設計者が引き出せるトリックがもう一つあります。カウントアップもしくはカウントダウンを行っていくには、すべてのカウントビットとの比較が必要でした。しかし、N-2から-1までカウントするとどうなるでしょうか？負の数は最上位ビットが1に設定されており、正の数はこのビットが0に設定されています。このビットだけをチェックして，カウンタが-1に達したことを検出すればよいのです。これがオタクが作ったカウンタです。

```
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 タイマ

私たちがつくることのできる、もう一つのタイマは、ワンショットタイマです。ワンショットタイマはキッチンタイマのようなもので、セットした時間が経つと、アラームが鳴ります。デジタルタイマには、クロックサイクルで時間が読み込まれています。それは、ゼロに達するまでカウントダウンしてゆき、ゼロになるとタイマが完了を教えてくれます。

図 6.10に、タイマのブロック図を示します。 `load`をアサートすることで、レジスタに`din`の値をロードすることができます。 `load`信号がデアサートされると、カウントダウンが選択されます(レジスタの入力として`cntReg - 1`を選択)。カウンタが`0`になると、`0`を供給するマルチプレクサの入力が選択されて、`done`信号がアサートされ、カウンタはカウントダウンを停止します。

コード 6.1はタイマのChiselコードを示しています。リセット時の初期値が`0`となる8ビットのレジスタ`cntReg`を使用します。ブール値`done`は`cntReg`を`0`と比較している結果です。入力マルチプレクサのために、デフォルト値が`0`のワイヤ`next`を導入します。 `when/elsewhen`ブロックは、他の2つの入力を選択する機能を持ちます。 `load`信号の選択はデクリメントよりも優先されます。最後の行は、`next`で表されるマルチプレクサの出力をレジスタ`cntReg`の入

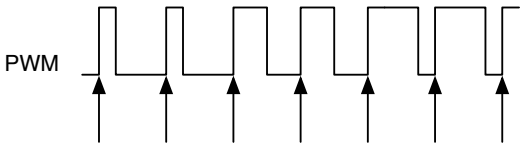


図 6.11: パルス幅変調 (PWM)

力に接続します。

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

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

コード 6.1: ワンショットタイマ

もう少し簡潔なコードにしようとするなら、中間ワイヤのnextを使用せずに、マルチプレクサの値を直接レジスタcntRegに代入することもできます。

6.2.5 パルス幅変調(PWM)

Pulse-width modulation(英語)/ パルス幅変調(日本語) （PWM）は、一定の周期を持つ信号で、その周期内で信号が*high*な時間で変調されたものです。

図 6.11はPWM信号を示しています。矢印は信号の周期の始まりを示しています。信号が高い時間の割合は、デューティサイクルと呼ばれます。最初の2周期は、デューティサイクルは25 %, つぎの2周期は50 %, そして最後の2期間は、75 %です。パルス幅は、25 %と75 %の間で変調されています。

PWM信号に low-pass filter(英語)/ ローパスフィルタ(日本語) を加えると、単純な digital-to-analog converter(英語)/ デジタル-アナログ変換回路(日本語) になります。ローパスフィルタは、抵抗とコンデンサと同じくらい単純なものです。

次のコード例では、10クロックサイクルごとに3クロックサイクルHigh (1) の波形を生成します。

```
  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)
```

PWMジェネレータを、再利用可能で軽量なコンポーネントとなるように、関数にしました。この関数には2つのパラメータがあります：PWMをクロックサイクル(nrCycles)で

設定するScala整数と，PWM出力信号のデューティサイクル(pulsewidth)を与えるChisel ワイヤ(din)です。この例では、カウンタを表現するためにマルチプレクサを使用しています。この関数の最後の行は、カウンタの値と入力値dinを比較してPWM信号を返します。Chisel関数の最後の式は戻り値であり、この例では比較関数に接続されたワイヤです。

関数unsignedBitLength(n)を使用して、最大n(を含む)までの符号なし数字を表現するために必要なカウンタcntRegのためのビット数を指定します。[4] また、Chiselには、数値の符号付き表現に必要なビット数を指定する関数signedBitLengthもあります。

もう一つのアプリケーションは、LEDを調光するためにPWMを使用することです。この場合、目はローパスフィルタとして機能します。上記の例を拡張して、三角波でPWM生成を駆動します。その結果、強度が連続的に変化するLEDが得られます。

```
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)
```

変調には2つのレジスタを使用しています: (1) カウントアップとカウントダウンのためのレジスタ modulationRegと、(2) カウントアップかカウントダウンかを判断するフラグとして利用するレジスタ upRegです。入力されたクロックの周波数 (この例では100 MHz) までカウントアップすると、0.5 Hzの信号が得られます。長い when/.elsewhen/.otherwise は、アップ/ダウンカウントと方向の切り替えを処理します。

このPWMは1 kHzの信号を生成するための周波数を1000分の1までしかカウントできないので、変調信号を1000で割る必要があります。実数の除算はハードウェア上では非常に高価なので、単に右に10ビットシフトすることで、つまり $2^{10} = 1024$による除算で代用します。PWM回路を関数として定義したので、関数呼び出しでインスタンス化することができます。ワイヤsigは変調されたPWM信号を表しています。

6.3 シフトレジスタ

shift register(英語)/ シフトレジスタ(日本語) は、順番に接続されたフリップフロップの集合体です。レジスタ(フリップフロップ)の各出力は、次のレジスタの入力に接続されます。図 6.12は、4段のシフトレジスタを示しています。この回路は、クロックティックごとにデータを左から右にシフトします。この単純なもので、回路はdinからdoutまでの4タップ遅延を実装しています。

[4] 2進数の符号なしnを表すためのビット数は$\lfloor log_2(n) \rfloor + 1$です。

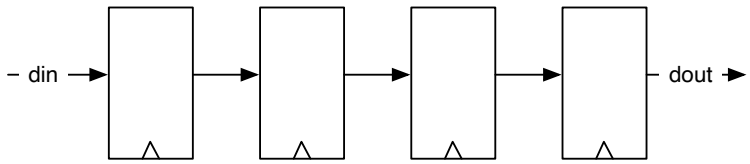


図 6.12: 4段のシフトレジスタ

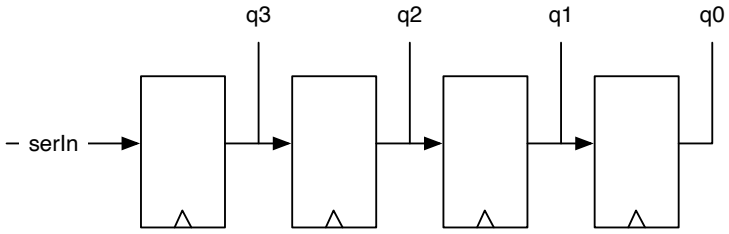


図 6.13: パラレル出力付き4ビットシフトレジスタ

この単純なシフトレジスタのためのChiselコードは以下のようになっています。(1) 4ビットのレジスタshiftRegを作成し、(2) シフトレジスタの下位3ビットをレジスタへの次の入力となるdinと連結し、(3)レジスタの最上位ビット(MSB)を出力doutとします。

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

シフトレジスタは、シリアルデータからパラレルデータへの変換やパラレルデータからシリアルデータへの変換に使用します。 11.2節では、受信機能と送信機能にシフトレジスタを使用したシリアルポートを示しています。

6.3.1 パラレル出力付きシフトレジスタ

シフトレジスタのシリアル-イン/パラレル-アウト構成は、シリアルデータ入力をパラレルワードに変換します。これは、シリアルポート (UART) で受信機能に使用することができます。図 6.13は、各フリップフロップ出力が1つの出力ビットに接続された4ビットのシフトレジスタを示しています。この回路は、4クロックサイクル後に、4ビットで1組のシリアルデータをqで使用可能な4ビットのパラレルデータに変換します。この例では、ビット0 (最下位ビット) が最初に送信され、それが最後のステージに到着すればすべてのワードを読めるようになると想定しています。

以下のChiselコードでは、シフトレジスタoutRegを0で初期化しています。次にMSBから入力していきますが、これは右シフトを意味します。結果のqはレジスタoutRegを読み込んだだけのものです。

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

図 6.13は、パラレル出力機能を持った4ビットのシフトレジスタを示しています。

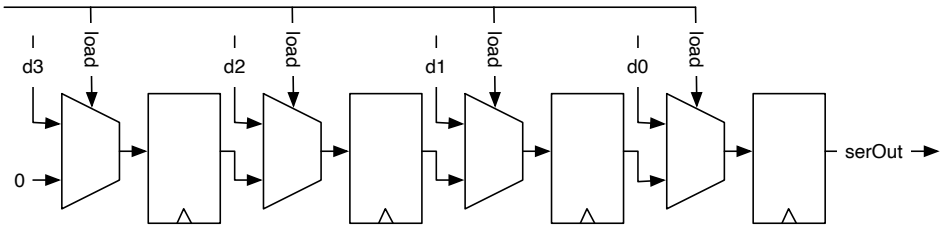


図 6.14: パラレルロード機能を持つ 4ビットシフトレジスタ

6.3.2 パラレルロード付きシフトレジスタ

シフトレジスタのパラレルインシリアルアウト構成は、ワード(バイト)のパラレル入力ストリームをシリアル出力ストリームに変換します。これは、シリアルポート(UART)で送信機能に使用することができます。

図 6.14は、パラレルロード機能を持つ4ビットのシフトレジスタを示しています。その機能をChiselで記述するのは比較的簡単です :

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

右シフトして、MSBにゼロを埋めていることに注意してください。

6.4 メモリ

Chiselの型`Vec`の`Reg`を用いて、メモリをレジスタの集まりとして構築することができます。しかし、これはハードウェアでは高価であるので、大きなメモリ構造は SRAM(英語)/ (日本語) で構築します。 ASICでは、メモリコンパイラがメモリを構築します。 FPGAは、ブロックRAMとも呼ばれるオンチップメモリブロックを内蔵しています。これらのオンチップメモリブロックを組み合わせて、大きなメモリにできます。 FPGAのメモリは通常、1つの読み取りと1つの書き込みポート、もしくは読み取りと書き込みを実行時に切り替えることができる2つのポートを持っています。

FPGA (およびASIC) は通常、同期メモリをサポートしています。同期メモリは、入力にレジスタを持っています(リードアドレスとライトアドレス、ライトデータ、ライトイネーブル)。つまり、アドレスを設定してから1クロック後に読み出しデータが利用可能になります。

図 6.15に、同期メモリの回路を示します。メモリは、1つのリードと1つのライトのデュアルポートです。読み出しポートには、読み出しアドレス(`rdAddr`)の1つの入力と、読み出しデータ(`rdData`)の1つの出力を持ちます。書き込みポートには、アドレス(`wrAddr`)、書き込みデータ(`wrData`)と、書き込みイネーブル(`wrEna`)の3つの入力があります。すべての入力について、メモリ内に同期動作を示すレジスタがあることに注意してください。

オンチップメモリをサポートするために、Chiselはメモリコンストラクタ`SyncReadMem`を提供しています。コード 6.2は、バイト単位の入出力データとライトイネーブルを持つ1 KiBのメモリを実装したコンポーネント`Memory`を示しています。

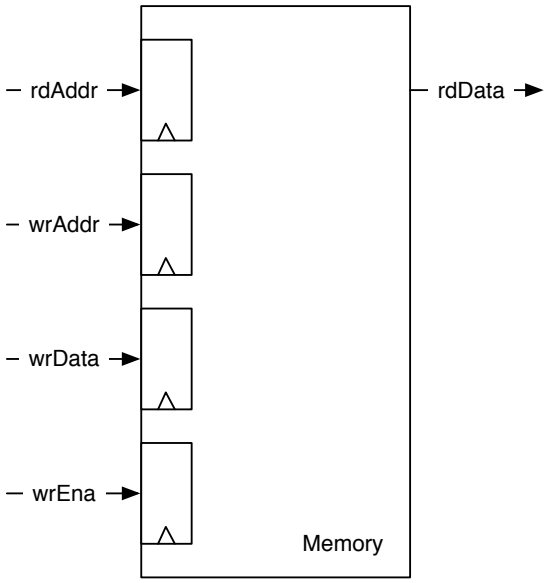


図 6.15: 同期メモリ

```
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)
  }
}
```

コード 6.2: 1 KiB 同期メモリ

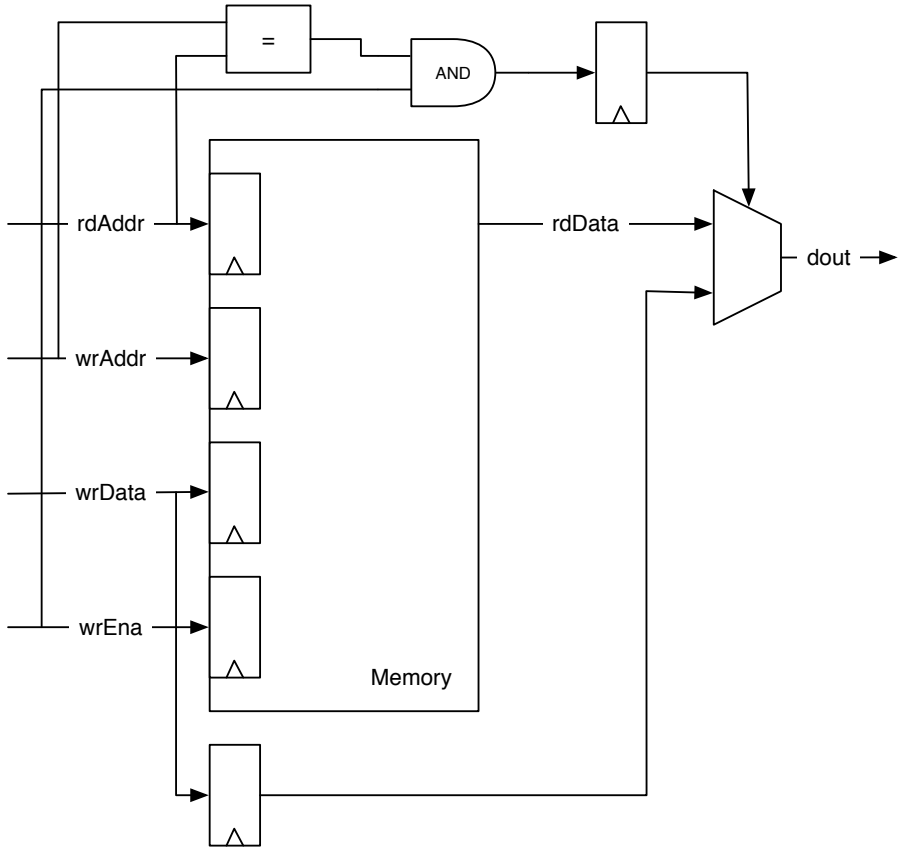


図 6.16: 書き込み中の読み出し動作に対応したフォワード(転送)回路を搭載した同期メモリ

興味深い質問は、同じクロックサイクルで新しい値が読み出されたのと同じアドレスに書き込まれた場合、どのような値が読み出されて返されるかということです。私たちは、メモリの書き込み最中の読み出し動作に興味を持っています。新たに書き込まれた値、古い値、または未定義(古い値の一部のビットと新たに書き込まれたデータの一部が混在している可能性がある)の3つの可能性があります。どの可能性がFPGAで利用可能かは、FPGAのタイプに依存し、指定できる場合もあります。 Chiselは、読み出しデータは未定義という仕様です。

新しく書き込んだ値を読み出したい場合は、アドレスが等しいことを検出して書き込みデータを送るフォワード(転送)回路を構築すればできます。図 6.16は、フォワード(転送)回路を搭載したメモリを示しています。リードアドレスとライトアドレスを比較し、ライトデータの転送パスとメモリのリードデータの転送パスをライトイネーブルでゲーティングして選択します。書き込みデータはレジスタで1クロック遅延となります。

コード 6.3に、フォワード(転送)回路を含む同期メモリのChiselコードを示します。次のクロックサイクルでも読み出し値を提供する同期メモリとするために、次のクロックサイクルで利用可能なレジスタ(`wrDataReg`)に書き込みデータを保持しておきます。 2つの入力アドレス(`wrAddr`と`rdAddr`)を比較して、フォワード(転送)条件で`wrEna`が真であるかどうかを確認します。その条件も1クロックサイクル遅れます。マルチプレクサは、フォワード(転送)データとメモリからの読み出しデータのどちらかを選択します。

```
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)
}
```

コード 6.3: フォワード(転送)回路を含む同期メモリ

Chiselはまた、同期書き込みと非同期読み出しを持つメモリの`Mem`も提供しています。このメモリタイプは通常、FPGAでは直接利用できないため、合成ツールはフリップフロップから構築します。そのため、`SyncReadMem`を使用することをお勧めします。

6.5 演習

前回の演習で使用した7セグメントエンコーダを使って、表示を`0`から`F`に切り替えるための入力として4ビットのカウンタを追加します。このカウンタをFPGAボードのクロックに直接接続すると、16個の数字がすべて重なって見えます (7セグメントすべてが点灯しているように見えます)。そのため、カウントを遅くする必要があります。500ミリ秒ごとにシングルサイクルのティック信号を生成できる別のカウンタを作成します。この信号を4ビットカウンタのイネーブル信号として使用します。

ジェネレータ関数で閾値と波形(三角波またはサイン関数)を設定しPWM波形をつくってみましょう。三角波はカウントアップとカウントダウンすることで作成できます。サイン関数は、数行のScalaコードでルックアップテーブルを生成し使えばよいでしょう(10.2節を参照してください)。 FPGAボード上のLEDを変調したPWM関数で駆動してみましょう。PWM信号の周波数はいくつにすべきでしょうか? 変調はどのくらいの周波数で動作していますか?

デジタル回路設計は、紙の上に回路としてスケッチすることが多いです。その際に、すべての詳細を示す必要はありません。本書の図のようにブロック図を使います。回路を模式的に表現したものと、Chiselの記述との間を流暢に行き来できることが重要なスキルです。それでは、以下の回路のブロック図をスケッチしてみてください。

```
  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 }
  }
```

次に、レジスタを含むもう少し複雑な回路を紹介します:

```
  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 入力処理

外部から同期回路に入力される信号は、通常、クロックに同期しているわけではなく、非同期です。入力信号は、0から1または1から0へのきれいな遷移を持たないソースから来る場合があります。例えば、ボタンやスイッチの跳ね返りなどがその例です。入力信号は、同期回路の遷移のトリガとなるスパイクを伴うノイズが多い場合があります。この章では、このような入力条件に対応する回路について説明します。

後者の2つ、デバウンススイッチとノイズフィルタリングについては、外付けのアナログ素子を使って解決することもできます。しかし、これらの問題はデジタル領域で対策を行った方が、（コスト面において）より効率的です。

7.1 非同期入力

システムクロックに同期していない入力信号は非同期信号と呼ばれています。これらの信号は、フリップフロップの入力のセットアップおよびホールド時間に違反することがあります。この違反はフリップフロップの Metastability(英語) を引き起こすことがあります。メタスタビリティは出力信号が0と1の間の値となったり、発振したりする結果を引き起こします。しかし、ある程度の時間を経ると、フリップフロップの値は0か1で安定します。

メタスタビリティを回避することはできませんが、その影響を抑えることはできます。古典的な解決策は、入力に2つのフリップフロップを使用することです。前提条件は次のとおりです。 1 番目のフリップフロップがメタスタビリティになると、クロック周期内に安定した状態に落ち着き、2 番目のフリップフロップのセットアップ時間とホールド時間制約が満たされることです。

図 7.1は同期回路と外部との間の境界線を示しています。入力信号のシンクロナイザは、2つのフリップフロップから構成されています。 Chiselではシンクロナイザのコードを、2つのレジスタをインスタンスする一行で表現できます。

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

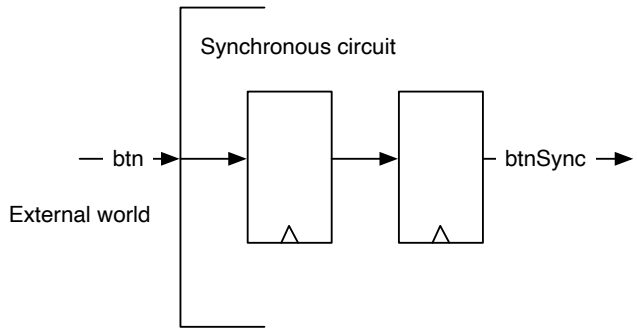


図 7.1: 入力信号のシンクロナイザ

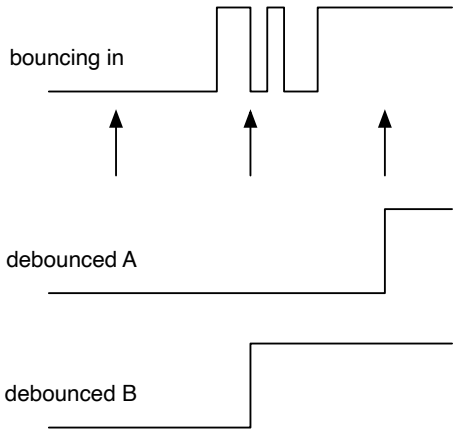


図 7.2: 入力信号のデバウンス

外部からのすべての非同期信号には入力シンクロナイザが必要です。[1] また、外部リセット信号についても同期する必要があります。リセット信号は他のフリップフロップ回路のリセット信号として使用する前に、2つのフリップフロップを通過させておきます。具体的には、リセット信号のデアサートはクロックに同期していなければなりません。

7.2 デバウンス

スイッチやボタンは、オンとオフの間の遷移にある程度の時間が必要な場合があります。遷移の間、スイッチは、これら2つの状態の間を行き来することがあります。このような信号を、追加の処理を行わずにそのまま使用した場合、意図したより多くの遷移イベントを検出することがあります。一つの解決策は、このバウンスをフィルタリングするために時間を使うことです。最大バウンス時間を t_{bounce} と仮定して、周期 $T > t_{bounce}$ で入力信号をサンプリングし、この信号のみを回路内部で使用します。

この長い周期で入力をサンプリングする場合、0 から 1 への遷移時に 1 つのサンプルだけがバウンス領域に落ちる可能性があることがわかっています。バウンス領域の前のサンプルは安全に 0 を読み、バウンシング領域の後のサンプルは安全に 1 を読みます。バウンス領域内のサンプルは 0 か 1 のどちらかになりますが、まだ 0 のサンプルか既に 1 のサンプルかのどちらかになるので、これは問題ではありません。重要なのは、0から1への遷移が1つしかないということです。

図 7.2は、動作中のデバウンスのサンプリングを示しています。上の信号はバウンシング入力を示し、下の矢印はサンプリングポイントを示しています。これらのサンプリングポイント間の距離は、最大バウンシング時間よりも長くする必要があります。最初のサンプルは安全に0をサンプリングし、図の最後のサンプルは1をサンプリングしています。真ん中のサンプルはバウンシング期間内になります。これは0か1のどちらかでしょう。 2つの可能性のある結果を、`debounce A`と`debounce B`として示します。どちらも0から1への切り替わりは1回です。これら2つの結果の唯一の違いは、バージョンBの切り替わりが1サンプル周期遅れていることです。しかし、これは通常は問題にはなりません。

デバウンスのChiselコードは、シンクロナイザ用のコードよりも少し進化します。サンプルタイミングは、 6.2.2項で行ったように、1サイクルの`tick`信号を出すカウンタで生成しま

[1]例外は入力信号が同期出力信号に依存していて、その最大伝搬遅延が分かっている場合です。古典的な例は、マイクロプロセッサなどで、非同期SRAMを同期回路とインターフェイスするような場合です。

す。

```
  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
  }
```

まず、サンプリング周波数を決めなければなりません。上の例では、100 MHzのクロックを想定しているので、サンプリング周波数は100 Hzとなります(バウンシングタイムが10 ms以下であると仮定しています)。カウンタの最大値は、除数である`FAC`です。デバウンスされた信号のために、リセット値を持たないレジスタ`btnDebReg`を定義しています。レジスタ`cntReg`がカウンタとして機能し、カウンタが最大値に達した時にティック信号が真となります。`when`条件が`true`の場合、(1)カウンタが0にリセットされ、(2)デバウンスレジスタに入力サンプルが格納されます。この例では、入力信号は前節で示した入力シンクロナイザからの出力なので、`btnSync`と名付けています。

デバウンス回路はシンクロナイザ回路の後に来ます。まず、非同期信号を同期させてから、デジタルドメインで処理します。

7.3 入力信号のフィルタリング

時々、入力信号がノイズを含んでいて、こうしたノイズを入力同期器やデバウンスユニットが意図せずにサンプリングしてしまう可能性があります。これらの入力スパイクをフィルタリングする方法の一つとして、多数決回路を使用する方法があります。最も単純なケースでは、3つのサンプルを取り、多数決を行います。中央値に関係する majority function(英語) では、多数派の値が結果として用いられます。デバウンスにサンプリングを用いる場合は、サンプリングされた信号に対して多数決を行います。多数決により、サンプリング周期よりも長い時間、信号が安定していることが保証されます。

図7.3は、多数決回路を示しています。デバウンスサンプリングに使用した`tick`信号により有効化された3ビットのシフトレジスタで構成されています。3つのレジスタの出力は多数決回路に送り込まれます。多数決機能は、サンプル期間より短い信号変化をフィルタリングします。

次のChiselコードの3ビットシフトレジスタは、`tick`信号よって駆動され、投票機能の結果として`btnClean`信号を返します。

多数決が必要なのは非常にまれであることに注意してください。

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

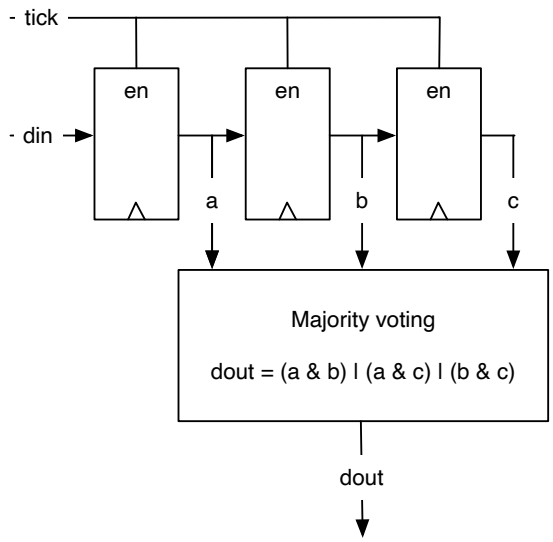


図 7.3: 入力信号のサンプリングによる多数決回路

```
  val btnClean = (shiftReg(2) & shiftReg(1)) | (shiftReg(2) & shiftReg(0))
      | (shiftReg(1) & shiftReg(0))
```

慎重に処理された入力信号の出力を使用するには、まず、RegNext 遅延要素で立ち上がりエッジを検出し、この信号をbtnCleanの現在の値と比較して、カウンタのインクリメントを行います。

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

  // 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 入力処理と関数の組み合わせ

入力処理をまとめるために、さらにいくつかのChiselのコードを示します。ここで提示された回路は、小粒ですが再利用可能なビルディングブロックなので、ここでは関数の形で表現しています。 4.4節では、完全なモジュールではなく、軽量なChisel関数の中に小さなビルディングブロックを抽象化する方法を示しました。これらの Chisel 関数はハードウェアインスタンスを生成します。例えば、関数 sync は入力と相互に接続された 2 つのフリップフロップを生成します。この関数は、2 番目のフリップフロップの出力を返します。もし有用であれば、これらの関数をいくつかのユーティリティクラスオブジェクトに昇格させることができます。

7.5 演習

入力ボタンでインクリメントされるカウンタを構築します。 FPGAボード上のLEDでカウンタの値をバイナリで表示します。 入力処理チェーン全体を: (1)入力同期回路、(2)デバウンス回路、(3)ノイズを抑制する多数決回路、(4)カウンタのインクリメントをトリガとするエッジ検出回路、から構築します。

最近のボタンが必ずバウンスを起こすという保証はありませんので、手動でボタンを高速で連続して押し、低いサンプル周波数を使用することで、バウンスやスパイクをシミュレートすることができます。サンプル周波数としては、例えば1秒を選択してください。例えば、入力クロック周波数が100 MHzであれば、100,000,000で分周します。 ボタンを押した状態にする前に、高速な押下でボタンバウンスをシミュレートしてみます。 1 Hzでサンプリングしたデバウンス回路を使用しない場合と使用した場合の回路をテストします。多数決回路を用いた場合では、カウンタの確実なインクリメントには1秒から2秒以上の間押す必要があります。また、ボタンのリリースも多数決です。そのため、回路は1〜2秒より長い場合にのみリリースを認識します。

```
1  def sync(v: Bool) = RegNext(RegNext(v))
2
3  def rising(v: Bool) = v & !RegNext(v)
4
5  def tickGen(fac: Int) = {
6    val reg = RegInit(0.U(log2Up(fac).W))
7    val tick = reg === (fac-1).U
8    reg := Mux(tick, 0.U, reg + 1.U)
9    tick
10 }
11
12 def filter(v: Bool, t: Bool) = {
13   val reg = RegInit(0.U(3.W))
14   when (t) {
15     reg := Cat(reg(1, 0), v)
16   }
17   (reg(2) & reg(1)) | (reg(2) & reg(0)) | (reg(1) & reg(0))
18 }
19
20 val btnSync = sync(btn)
21
22 val tick = tickGen(fac)
23 val btnDeb = Reg(Bool())
24 when (tick) {
25   btnDeb := btnSync
26 }
27
28 val btnClean = filter(btnDeb, tick)
29 val risingEdge = rising(btnClean)
30
31 // Use the rising edge of the debounced
32 // and filtered button for the counter
33 val reg = RegInit(0.U(8.W))
34 when (risingEdge) {
35   reg := reg + 1.U
36 }
```

コード 7.1: 関数を使った入力信号処理のまとめ

8 ステートマシン

ステートマシン (Finite-State Machine: FSM) は、デジタル回路設計の基本的な構成要素です。FSMは、状態と状態の間の遷移とその(ガードされた)条件を記述しています。 FSMはリセット時に設定される、初期状態を持ちます。 FSMは同期順序回路と呼ばれることもあります。

FSMの実装は3つの部分から構成されています : (1) 現在の状態を保持するレジスタ。 (2) 現在の状態と入力から次の状態を計算する組合せロジックと、 (3) FSMの出力を計算する組合せロジック。

原理的には、状態を保存するためのレジスタや他のメモリ要素を含むすべてのデジタル回路は、単一のFSMとして記述することができます。しかし、これは現実的ではないかもしれません、例えば、ラップトップコンピュータを単一のFSMとして記述してみることを考えてみてください。次の章では、小さなFSM同士を通信させて組み合わせることで、より大きなシステムを構築する方法を説明します。

8.1 ステートマシンの基本

図 8.1は、FSMの回路図を示しています。レジスタには現在の`state`が格納されています。「Next state logic」は、現在の`state`と入力(`in`)から次の状態値 (`nextState`) を計算します。次のクロックティックで `state` が `nextState` になります。「Output logic」は、出力(`out`)を計算します。出力は現在の状態にのみ依存するので、この状態マシンを Moore machine(英語)/ムーアマシン(日本語) と呼びます。

state diagram(英語)/ 状態遷移図(日本語) は、そのようなFSMの動作を視覚的に表現して記述したものです。状態遷移図では，個々の状態をその状態名でラベル付けした円で表現しています。状態遷移は状態の間を矢印で示し、この遷移が行われたときのガード(もしくは条件)が矢印のラベルとして描かれています。

図 8.2に、簡単なFSMの状態遷移図例を示します。このFSMにはアラームレベルを示す3つの状態: *green*(緑)、*orange*(橙)、*red*(赤)があります。 FSMは緑のレベルから始まり、 悪いできごと (`badEvent`) が発生すると、アラームレベルは橙に切り替わります。 2回目の悪いできごとが発生すると、アラームレベルは赤に切り替わります。この時にベルが鳴るようにしたいです。そこでこのFSMの唯一の出力である ベルを鳴らす (`ringBell`) を赤状態に接続します。アラームは*clear*信号でリセットできます。

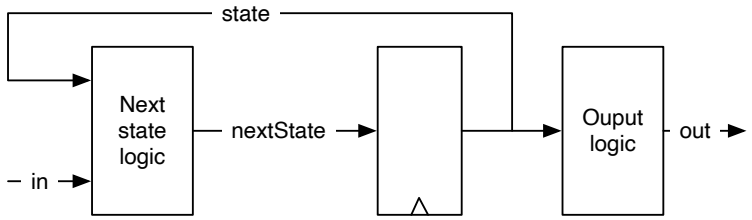


図 8.1: ステートマシン (ムーア型)

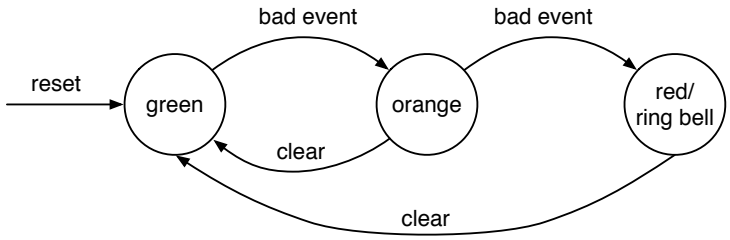


図 8.2: アラームFSMの状態遷移図

表 8.1: アラームFSMの状態表

	Input			
State	Bad event	Clear	Next state	Ring bell
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

状態遷移図は視覚的に分かりやすく、FSMの機能をすばやく把握できるものの、速く書けるのは状態表の方かもしれません。表 8.1にアラームFSMの状態表を示します。現在の状態、入力値、次の状態、現在の状態の出力値をリストアップします。原則として、取り得る全ての状態に対して、取り得る全ての入力を指定する必要があります。この表は、$3 \times 4 = 12$行で構成されるはずですが、悪いできごと (`badEvent`) が発生したときにクリア入力が無視されることを示して、表を単純化しています。つまり、悪いできごとはクリアよりも優先されます。出力の列には繰り返しがあります。もし、FSMが大きくなったり、出力が多くなったりした場合は、テーブルを次の状態のロジックの表と出力のロジックの表の2つに分割することができます。

最後に、アラームレベルFSMの設計がすべて終わったら、Chiselのコードにします。コード 8.1はアラームFSMのChiselコードを示しています。FSMの入力と出力にはChiselの`Bool`型を使用していることに注意してください。`Enum`と`switch`制御命令を使うには、`chisel3.util._`をインポートする必要があります。

このシンプルなFSMのための完全なChiselコードは1ページに収まっています。個々の部分を順を追って見ていきましょう。FSMは2つの入力と1つの出力信号を持っており、Chisel `Bundle`でキャプチャされます。

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

最適な状態のエンコーディングには、様々な研究が行われました。一般的な選択肢としては、バイナリエンコーディングとワンホットエンコーディングの2つがあります。しかし、

```
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
}
```

コード 8.1: アラームFSMのChiselコード

それらの低レベルな判断は合成ツールに任せて、読みやすいコードを目指します。[1] よって、状態名にはシンボリックな名前を持つ列挙型を使用します。

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

各状態値は、::演算子で連結したリストの要素として記述します。Nilはリストの終わりを表します。 Enumインスタンスが、状態のリストに割り当てられます。状態を保持するレジスタは、緑色の状態をリセット値として、次のように定義します。

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

FSMのミソは次の状態のロジックにあります。状態レジスタのChisel switch文で、すべての状態を記述します。それぞれの分岐で、入力に依存する次の状態ロジックを記述し、状態レジスタに新しい値を割り当てます。

```
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
    }
  }
}
```

最後に、重要なことですが、状態が赤のときにベルの出力がtrueになるように記述します。

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

レジスタ入力には、普段VerilogやVHDLで記述されるような nextState信号を導入していないことに注意してください。 VerilogやVHDLのレジスタは特別な構文で記述し、組み合わせブロック内では割り当て(および再割り当て)ができません。そのため、組み合わせブロック内で計算された追加信号を導入し、レジスタ入力に接続します。 Chiselではレジスタは基本型であり、組み合わせブロック内で自由に使用することができます。

8.2 ミーリFSMで出力を高速化

ムーアFSMでは、出力は現在の状態にのみ依存します。したがって、入力の変化による出力の変化は、最も早い場合でも、次のクロックサイクルとなります。即座に変化を観察し

[1] 現在のバージョンのChiselでは、Enum型は状態をバイナリエンコーディングで表します。もし別のエンコーディング、例えばワンホットエンコーディングをしたい場合は、状態名のためにChisel定数を定義することができます。

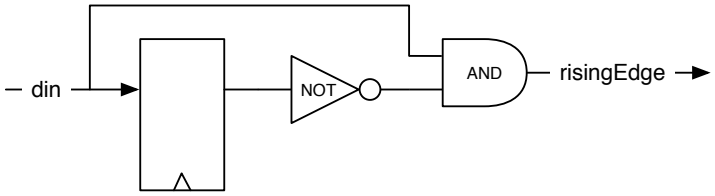


図 8.3: 立ち上がりエッジ検出器 (ミーリ型FSM)

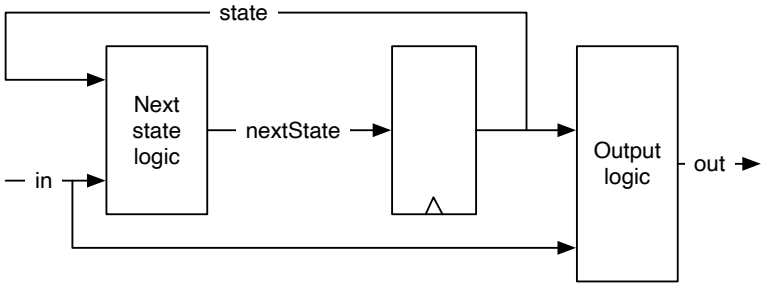


図 8.4: ミーリ型のステートマシン

たい場合は、入力から出力までの組み合わせパスが必要です。最小限の例として、エッジ検出回路を考えてみましょう。以前にもこのChiselの一行記述を見たことがあります:

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

図 8.3に立ち上がりエッジ検出器の回路図を示します。現在の入力が1で、最後のクロックサイクルの入力が0だった場合、出力は1クロックサイクルの間1になります。状態レジスタは、次の状態が入力されただけの単一のDフリップフロップです。これを1クロックサイクルの遅延素子と考えることもできます。出力ロジックは、現在の入力と現在の状態を比較します。

出力が入力にも依存している場合、すなわち、FSMの入力と出力の間に組合せ経路がある場合、これを Mealy machine(英語)/ ミーリマシン(日本語) と呼びます。

図 8.4にミーリ型FSMの回路図を示します。ムーア型FSMと同様に、レジスタには現在の`state`が格納され、次の状態ロジックは現在の`state`と入力(`in`)から次の状態値(`nextState`)を計算します。次のクロックティックで`state`は`nextState`になります。出力ロジックは、現在の状態とFSMへの入力から出力(`out`)を計算します。

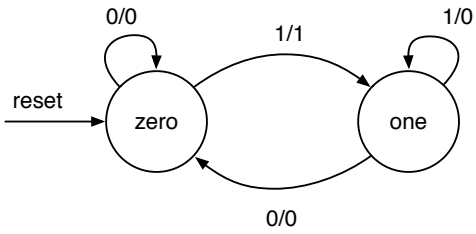


図 8.5: ミーリFSMによる立ち上がりエッジ検出回路の状態遷移図

```
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
      }
    }
  }
}
```

コード 8.2: ミーリマシンを用いた立ち上がりエッジ検出

図 8.5に、エッジ検出器のミーリFSMの状態図を示します。状態レジスタは単一のDフリップフロップで構成されているため、取り得るのは2つの状態のみであり、この例ではzeroとoneと名付けています。ミーリFSMの出力は、状態だけでなく入力にも依存するため、出力を状態円の一部として記述することはできません。かわりに、状態間の遷移は入力値(条件)と出力(スラッシュの後)でラベル付けされます。また、自己遷移を描画することにも注意してください。例えば、入力が0の場合、状態zeroではFSMは状態zeroのままで、出力は0になります。立ち上がりエッジのFSMは、状態zeroから状態oneへの遷移でのみ1の出力を生成します。入力が1になったことを表す状態oneでは、出力は0になります。入力の立ち上がりエッジごとに1つの(サイクルの)パルスが欲しいだけです。

コード 8.2は、ミーリマシンを用いた立ち上がりエッジ検出のためのChiselコードです。前の例と同様に、シングルビットの入力と出力にChisel型のBoolを使用しています。出力ロジックは次の状態ロジックの一部となり、zeroからoneへの遷移時に出力はtrue.Bとなります。それ以外の場合は、出力へのデフォルトの割り当て(false.B)がカウントされます。

私たちは、同じ機能のChisel ワンライナーを見たことがあるので、エッジ検出回路に本

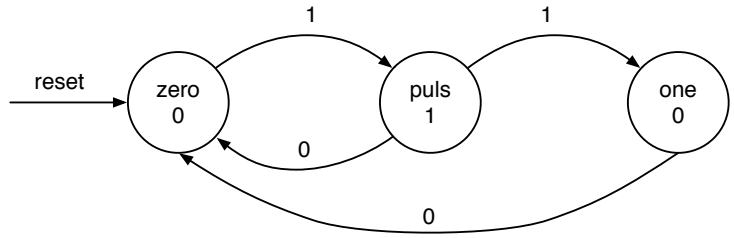


図 8.6: ムーアFSMによる立ち上がりエッジ検出回路の状態遷移図

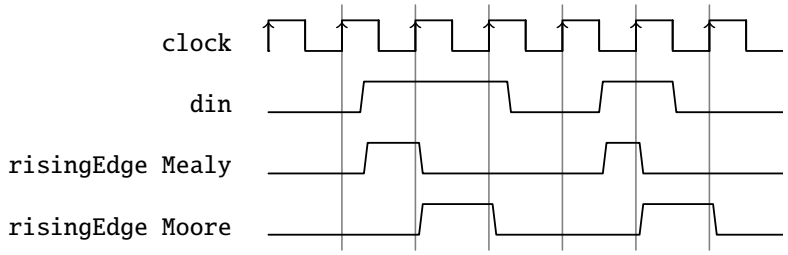


図 8.7: ミーリとムーアFSMの立ち上がりエッジ検出回路の波形図

格的なFSMが最適かどうかを比較することができます。ハードウェア消費量は似ています。どちらのソリューションも、ステート用に単一のDフリップフロップを必要とします。FSMの組み合わせロジックは、状態の変化が現在の状態と入力値に依存するため、多少複雑になります。この機能により重要なのは、ワンライナーの方が書きやすく、読みやすいことです。したがって、ワンライナーが好ましい解決策でしょう。

この例を用いて、可能な限り最小のミーリFSMの1つを示しました。 FSMは、3つ以上の状態を持つより複雑な回路に使用します。

8.3 ムーア対ミーリ

ムーアFSMとミーリFSMの違いを示すために、ムーアFSMでエッジ検出をやり直します。

図 8.6は、ムーアFSMによる立ち上がりエッジ検出の状態図です。まず注目すべきは、ムーアFSMでは3つの状態が必要なのに対し、ミーリ版では2つの状態であることです。状態`puls`は、シングルサイクルパルスを生成するために必要です。 FSMは、1クロックサイクルの間`puls`状態を維持し、入力が再び0になるのを待って、開始状態の`zero`に戻るか、`one`の状態に戻ります。状態遷移矢印に入力条件を示し、円で示した状態の中にFSMの出力を示します。

コード 8.3は立ち上がりエッジ検出回路のムーア版を示しています。それは、ミーリまたは直接実装したものに比べて2倍のDフリップフロップを使用しています。したがって、結果として得られる次の状態ロジックも、ミーリまたは直接実装された版よりも大きくなります。

図 8.7は立ち上がりエッジ検出FSMのミーリ版とムーア版の波形を示しています。ミーリ出力は入力された立ち上がりエッジに忠実に追従しているのに対し、ムーア出力はクロックティック後に立ち上がります。また、ムーア版の出力は1クロック周期の幅があるのに対し、ミーリ版の出力は通常1クロック周期以下であることがわかります。

上記の例から、ミーリFSMはムーアFSMよりも必要なステート(およびロジック)が少な

```
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
}
```

コード 8.3: 立ち上がりエッジ検出回路のムーア版

く、反応が速いため、ミーリFSMの方が優れたFSMであると考えたくなります。しかし、ミーリマシン内の組み合わせパスは、大規模な設計では問題を引き起こす可能性があります。第一に、協調型FSMのチェーン (次の章を参照) では、この組み合わせパスは長くなる可能性があります。第二に、協調型FSMが円を描く場合、その結果は組合せループとなり、同期設計のエラーとなります。ムーアFSMの状態レジスタとの組み合わせパスが切断されているため、協調型ムーアFSMには上記の問題は存在しません。

まとめると、ムーアFSMは協調型ステートマシンに適しており、ミーリFSMよりうまく動作します。ミーリFSMは、同じサイクル内での反応が最も重要な場合にのみ使用してください。実質的にミーリマシンである立ち上がりエッジ検出などの小さな回路では問題ありません。

8.4 演習)

この章では、多くの非常に小さなFSM例を見てきました。では、そろそろ実際に FSMのコードを書いてみましょう。もう少し複雑な例を選んで、FSMを実装し、テストベンチを書いてみましょう。

FSMの典型的な例は、信号機です([3, 14.3節]を参照ください)。信号機は、赤から緑への切り替えの際に、交差点内の両方の道路が進入禁止信号(赤と橙)を待っている間の段階がなくてはなりません。この例をもう少し面白くするために、優先道路を考えてみましょう。マイナーな道路には2つの車両検知器があります(交差点の両方の入口にあります)。車が検知されたときだけマイナー道路を緑に切り替えてから、優先道路を緑に戻します。

9 協調型ステートマシン

単一のFSMで記述するには (回路が) 複雑すぎることが、しばしば発生します。こうした場合、回路を複数の小さくて単純なFSMに分割します。分割されたFSMは信号を介して通信します。一方のFSMの出力は他方のFSMの入力であり、FSMは他方のFSMの出力を監視します。大きなFSMをより単純なものに分割することを、FSMのファクタリングと呼びます。しかし、単一のFSMでは実現できないほど大きなものが多いため、しばしば仕様段階から直接協調型FSMとして設計されます。

9.1 ライトフラッシャの例

協調型FSMについて議論するために、我々は [3, 17章]、ライトフラッシャの例を使用します。ライトフラッシャは、1つの入力`start`と1つの出力`light`を有しています。ライトフラッシャの仕様は以下の通りです :

- `start`が1クロックサイクルの間Highになると、点滅シーケンスを開始します。
- 1回のシーケンスでは、点滅が3回行われます。
- `light`は、6クロックサイクルのオンと、4クロックサイクルのオフを繰り返すことで点滅します。
- シーケンスの後、FSMは`light`をオフにして次のスタートを待ちます。

仕様通りに単純に実装したFSM[1]は27の状態を持っており、入力待ちの初期状態が1つ、3種類のオン状態が3×6と、オフ状態が2×4です。このライトフラッシャの単純実装のコードは、ここでは示しません。

この問題は、この大きなFSMを2つの小さなFSMにファクタリングすることで、よりエレガントに解決できます。マスターFSMは点滅ロジックを実装し、タイマFSMは待ち時間を実装します。図9.1は、2つのFSMの構成を示しています。

タイマFSMは、所望のタイミングを作り出すために6または4クロックサイクルでカウントダウンします。タイマの仕様は以下の通りです :

- `timerLoad`がアサートされると、タイマは状態に関係なくダウンカウンタに値をロードします。
- `timerSelect`はロードする値を5または3のいずれかを選択します。
- カウンタがカウントダウンを完了し、アサートされたままになると`timerDone`がアサートされます。
- それ以外の場合、タイマはカウントダウンします。

以下のコードは、ライトフラッシャのタイマFSMを示しています。

[1] 状態遷移図は [3, p. 376]に示されています。

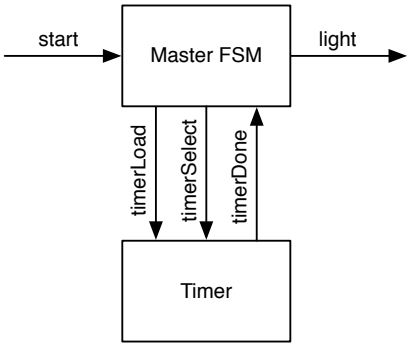


図 9.1: マスターFSMとタイマFSMに分割されたライトフラッシャ回路

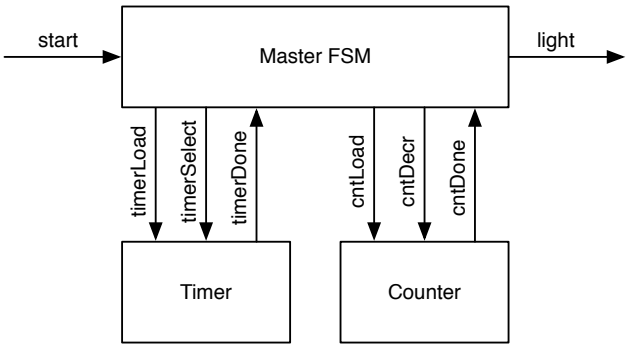


図 9.2: マスターFSMとタイマFSMとカウンタFSMに分割されたライトフラッシャ回路

```
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
  }
}
```

コード 9.1は、マスターFSMを示しています。

このマスターFSMとタイマを用いた解決法では、まだマスターFSMコードに無駄なところが残っています。ステートの`flash1`、`flash2`、および`flash3`は同じ機能を実行しており、ステートの`space1`と`space2`も同様です。残りのフラッシュの数を別のカウンタにくくりだすことができます。こうすることにより、マスターFSMは`off`、`flash`、`space`の3つの状態に減ります。

図 9.2は、マスターFSMとカウンタFSMのデザインを示しています。1つ目のFSMは、オ

```
1  val off :: flash1 :: space1 :: flash2 :: space2 :: flash3 :: Nil =
       Enum(6)
2  val stateReg = RegInit(off)
3
4  val light = WireDefault(false.B) // FSM output
5
6  // Timer connection
7  val timerLoad = WireDefault(false.B) // start timer with a load
8  val timerSelect = WireDefault(true.B) // select 6 or 4 cycles
9  val timerDone = Wire(Bool())
10
11 timerLoad := timerDone
12
13 // Master FSM
14 switch(stateReg) {
15   is(off) {
16     timerLoad := true.B
17     timerSelect := true.B
18     when (start) { stateReg := flash1 }
19   }
20   is (flash1) {
21     timerSelect := false.B
22     light := true.B
23     when (timerDone) { stateReg := space1 }
24   }
25   is (space1) {
26     when (timerDone) { stateReg := flash2 }
27   }
28   is (flash2) {
29     timerSelect := false.B
30     light := true.B
31     when (timerDone) { stateReg := space2 }
32   }
33   is (space2) {
34     when (timerDone) { stateReg := flash3 }
35   }
36   is (flash3) {
37     timerSelect := false.B
38     light := true.B
39     when (timerDone) { stateReg := off }
40   }
41 }
```

コード 9.1: ライトフラッシャのマスターFSM

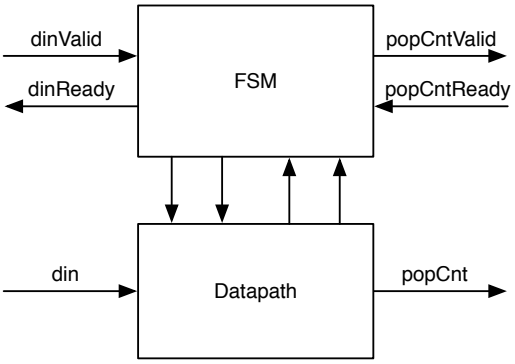


図 9.3: データパスを伴ったFSM

ンとオフの間隔の長さのクロックサイクルをカウントします。 2つ目のFSMは、残りのフラッシュをカウントします。

次のコードは、ダウンカウンタFSMを示しています :

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

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

カウンタは残りの点滅をカウントし、タイマが終了したステートのspaceでデクリメントされるので、3回点滅するために2をロードすることに注意してください。 コード 9.2は2回リファクタリングされたフラッシャのマスターFSMを示しています。

マスターFSMが3つの状態に削減されただけでなく、今回の解決法は構成の変更が容易になります。 FSMを変更することなくオン/オフの間隔の長さや点滅の数を変更できます。

ここでは、制御信号のみを交換する通信回路、特にFSMについて探ってきました。しかし、回路はデータを交換することもできます。協調的なデータ交換にはハンドシェイク信号を使用します。次節では、一方向データ交換のフロー制御のためのready-validインタフェースについて説明します。

9.2 データパスを持つステートマシン

協調型ステートマシンの典型的な例として、データパスと組み合わせたステートマシンがあります。この組み合わせは、データパス付き有限ステートマシン(FSMD)とよく呼ばれます。ステートマシンはデータパスを制御し、このデータパスが計算を実行します。 FSMの入力は、環境からの入力とデータパスからの入力です。環境からのデータをデータパスに供給し、このデータパスからデータが出力されます。図 9.3にFSMとデータパスの組み合わせの例を示します。

9.2.1 ポップカウントの例

図 9.3に示すFSMDは、また Hamming weight(英語)/ ハミング重み(日本語) と呼ばれるポップカウントを計算する例です。ハミング重みは、0ではないシンボルの個数です。バイナリビ

```
1  val off :: flash :: space :: Nil = Enum(3)
2  val stateReg = RegInit(off)
3
4  val light = WireDefault(false.B) // FSM output
5
6  // Timer connection
7  val timerLoad = WireDefault(false.B) // start timer with a load
8  val timerSelect = WireDefault(true.B) // select 6 or 4 cycles
9  val timerDone = Wire(Bool())
10 // Counter connection
11 val cntLoad = WireDefault(false.B)
12 val cntDecr = WireDefault(false.B)
13 val cntDone = Wire(Bool())
14
15 timerLoad := timerDone
16
17 switch(stateReg) {
18   is(off) {
19     timerLoad := true.B
20     timerSelect := true.B
21     cntLoad := true.B
22     when (start) { stateReg := flash }
23   }
24   is (flash) {
25     timerSelect := false.B
26     light := true.B
27     when (timerDone & !cntDone) { stateReg := space }
28     when (timerDone & cntDone) { stateReg := off }
29   }
30   is (space) {
31     cntDecr := timerDone
32     when (timerDone) { stateReg := flash }
33   }
34 }
```

コード 9.2: 2回リファクタリングしたフラッシャのマスターFSM

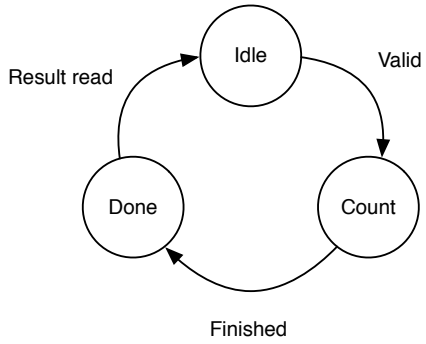


図 9.4: ポップカウント FSM の状態遷移図

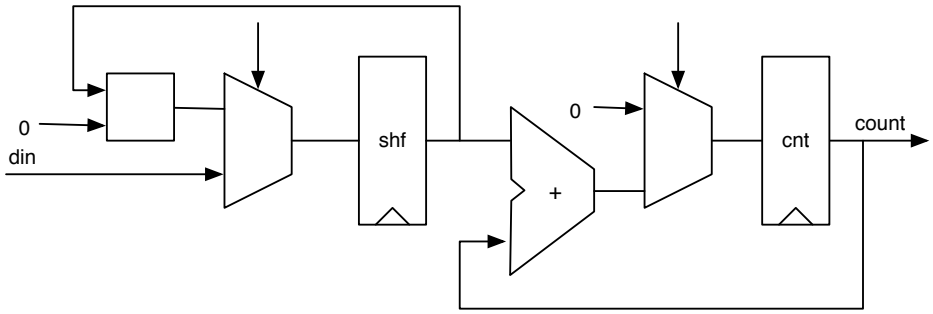


図 9.5: ポップカウント回路のデータパス

ット列の場合、これは‘1’の数となります。

ポップカウントユニットはデータ入力`din`と結果の出力`popCount`を持ち、両方ともデータパスに接続されています。入力と出力には、ready-validハンドシェイクを使用します。データが使用可能な場合は、validがアサートされます。レシーバがデータを受け取れるようになると、readyがアサートされます。両方の信号がアサートされると転送が行われます。ハンドシェイク信号はFSMに接続されます。 FSMは、データパスへの制御信号とデータパスからのステータス信号を介してデータパスと接続されています。

次のステップとして、図 9.4に示す状態図から、FSMを設計してみましょう。 FSMは入力を待つ`Idle`状態からスタートします。データが到着し、有効な信号が送られてくると、FSMはシフトレジスタをロードするために、状態`Load`に進みます。 FSMは次のステート`Count`に進み、そこで‘1’の数が順次カウントされます。シフトレジスタ、加算器、アキュムレータレジスタ、ダウンカウンタを使用して計算を行います。ダウンカウンタがゼロになると計算が終了し、FSMは`Done`の状態に移行します。ここでFSMはpopcount値が消費される準備ができたことをvalid信号で通知します。受信機からの準備完了の信号でFSMは`Idle`状態に戻り、次のpopcountを計算する準備ができています。

コード 9.3で示されるトップレベルのコンポーネントはFSMとデータパスコンポーネントをインスタンス化し、バルク接続で接続します。

図 9.5はポップカウント回路のデータパスを示しています。データは`shf`レジスタにロードされます。ロード時には`cnt`レジスタも0にリセットされます。 ‘1’の数をカウントするために、`shf`レジスタを右にシフトし、クロックサイクルごとに最下位ビットを`cnt`に加算します。図に示されていないカウンタは、全てのビットが最下位ビットを通過するまでカウン

```
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
}
```

コード 9.3: ポップカウント回路のトップレベル

```
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
}
```

コード 9.4: ポップカウント回路のデータパス

トダウンします。カウンタがゼロになると、ポップカウントは終了します。FSMはDoneの状態に切り替わり、popCntReadyをアサートすることで結果を通知します。結果が読み出された場合はpopCntValidをアサートすることで、FSMはIdle状態に戻ります。

load信号では、regDataレジスタに入力が読み込まれ、regPopCountレジスタが0にリセットされたのちに、カウンタレジスタregCountが実行するシフト数に設定されます。

そうでない場合は、regDataレジスタを右にシフトして、regDataレジスタの最下位ビットをregPopCountレジスタに追加し、カウンタが0になるまでデクリメントされます。カウンタが0の時、出力はpopcountとなります。コード 9.4はポップカウント回路のデータパスのためのChiselコードを示しています。

FSMはidle状態から開始します。入力データが有効な信号(dinValid)になると、count状態に切り替わり、データパスがカウントを終了するまで待ちます。popcountが有効な場合、FSMはdone状態に切り替わり、popcountが読み込まれるまで待ちます(popCntReadyによってシグナルが送られます)。コード 9.5はFSMのコードを示しています。

```
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
      }
    }
  }
}
```

コード 9.5: ポップカウント回路のFSM

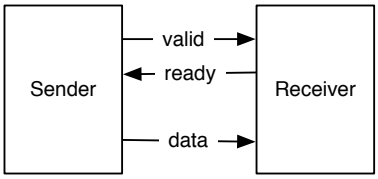


図 9.6: ready-valid のフロー制御

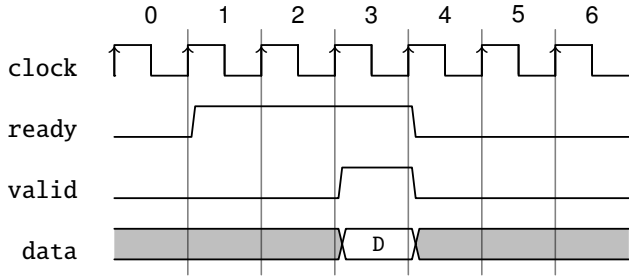


図 9.7: ready-validインタフェースのデータ転送、 ready信号が早い場合

9.3 Ready-Valid インターフェース

サブシステムの通信は、データの移動とフロー制御のためのハンドシェイクの形に一般化することができます。ポップカウントの例では、valid信号とready信号を使用した入力データと出力データのハンドシェイクインターフェースを見てきました。

ready-validインタフェース [3, p. 480]は、送信側 (プロデューサ) ではdataとvalid信号、受信側 (コンシューマ) ではready信号で構成される単純なフロー制御インタフェースです。図 9.6は、ready-valid接続を示しています。送信側はdataがあるときにvalidをアサートし、受信側は1ワードのデータを受信する準備ができているときにreadyをアサートします。データの送信は、validとreadyの両方の信号がアサートされたときに行われます。 2つの信号のいずれかがアサートされない場合、転送は行われません。

図 9.7は、送信側がデータを持つ前に(クロックサイクル1から)受信側がready信号を出すready-validトランザクションのタイミング図を示しています。データ転送はクロックサイクル3で行われます。クロックサイクル4からは、送信側がデータを持っておらず受信側も次の受信の準備はできていません。受信側がすべてのクロックサイクルでデータを受信できる場合、それは“常時レディ”インターフェースと呼ばれ、readyはtrueにハードコードすることができます。

図 9.8は、送信側が受信側の準備が整う前に(クロックサイクル1から)validな信号を送信するready-validトランザクションのタイミング図を示しています。データ転送は、クロックサイクル3で行われます。クロックサイクル4以降は、送信側も受信側も次の転送の準備ができていません。 “常時レディ”インターフェースと同様に、常に有効なインターフェースに似ています。しかし、その場合はおそらくready信号の変化でデータが変わることはなく、単純にハンドシェイク信号を落とすことになるでしょう。

図 9.9は、ready-validインタフェースの更なるバリエーションを示しています。クロックサイクル1では、両方の信号(readyおよびvalid)が1クロックサイクルの間だけアサートされ、D1のデータ転送が起こります。 D2とD3の転送で、クロックサイクル4と5に示すように、データを空きサイクルなし (クロックサイクル毎) に転送することができます。

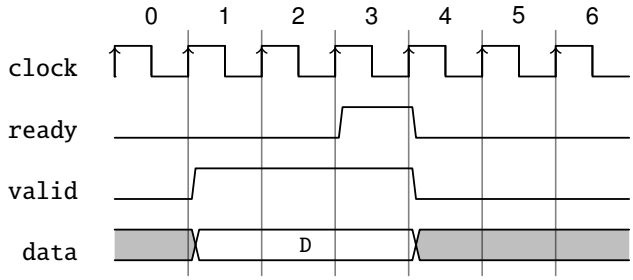


図 9.8: ready-validインタフェースのデータ転送、ready信号が遅い場合

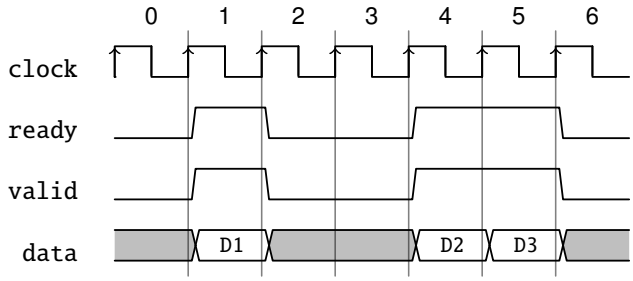


図 9.9: シングルサイクルの ready/valid 信号と back-to-back 転送

このインターフェースを構成可能にするために、readyも validも、他方の組み合わせ回路の出力に依存することは許されていません。このインタフェースは非常に一般的なものなので、Chiselでは、以下のようなDecoupledIOバンドルを定義しています：

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

DecoupledIOバンドルは、dataの型でパラメータ化されています。 Chiselで定義されたインターフェースは、データのフィールドbitsを使用します。

残る疑問の1つは、アクティブな状態でデータ転送が行われていない場合に、readyまたはvalidな状態がデアサートされる可能性があるかどうかです。例えば、受信側はデータを受信していなくても、他のイベントのために準備ができていないかもしれません。同じことが送信機にも考えられ、有効なデータがあるのは少しのクロックサイクルだけで、データ転送が起こらないことがあります。この動作が許されるかどうかは、ready-validインタフェースの一部ではなく、インタフェースの具体的な使用法によって定義される必要があります。

Chiselでは、DecoupledIOクラスを使用した場合、readyとvalidの振る舞いには何の制約もありません。ただし、IrrevocableIOクラスは送信側に次のような制限をかけています：

> ReadyValidIOのサブクラスであり、validがHighでreadyがLowのサイクルの後はbitsの値は変更されません。さらに、一度validが上がった後は、readyが上がる後までvalidが下がることはありません。

これは、クラス`IrrevocableIO`を使用した場合には保証されない制限であることに注意してください。

AXIは、次のバスの各部分: 読み出しアドレス、読み出しデータ、書き込みアドレス、および書き込みデータ、それぞれに1つのready-validインターフェースを使用します。 AXIは、一度`ready`または`valid`がアサートされると、データ転送が起こるまでアサートが解除されないようにインターフェースを制限します。

10 ハードウェアジェネレータ

Chiselの強みはいわゆるハードウェアジェネレータを書けることです。 VHDLやVerilogといった古いハードウェア記述言語では、この機能をつかうためにJavaやPythonといった他の言語を併用する必要があります。筆者はよくVHDLのテーブルを生成する小さなJavaのプログラムを書いたりします。 ChiselではScalaの全機能（とJavaのライブラリ）をハードウェアの構築に利用できます。これにより、ハードウェアジェネレータをChiselの回路生成の一部として、同じ言語で記述し、実行できます。

10.1 設定をパラメータ化する

Chiselのコンポーネントや関数は、パラメータを使って設定することができます。パラメータは整数定数のようなシンプルなものから、Chiselのハードウェアタイプのものまであります。

10.1.1 シンプルなパラメータ

回路をパラメータ化する基本的な方法は、信号のビット幅をパラメータとして定義することです。パラメータはChiselモジュールのコンストラクタに引数として渡すことができます。次に示すのは加算器のビット幅を可変にした簡単なモジュールの例です。 ScalaのInt型の変数nが、ビット幅を示すパラメータとしてコンストラクタに渡され、そのパラメータがIOバンドルで使用されています。

```
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
}
```

パラメータ化された加算器は次のようにして生成できます。

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

10.1.2 型パラメータを持つ関数

ビット幅を設定パラメータの形で持つことは、ハードウェアジェネレータとしての出発点に過ぎません。型パラメータを使用することで、さらに柔軟な設定ができます。この機能を使って実装されたのがChiselのマルチプレクサ(Mux)で、任意の型を受け入れることができ

ます。どのようにして型パラメータを使用するかを示すために、任意の型を受け入れるマルチプレクサを構築してみます。次の関数は、マルチプレクサを定義しています。

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

    val ret = WireDefault(fPath)
    when (sel) {
      ret := tPath
    }
    ret
  }
```

Chiselでは、Chiselの型を使って関数をパラメータ化できます。角括弧`[T <: Data]`を使った表現で、`Data`とそのサブクラスを型パラメータ`T`として定義します。`Data`はChiselのデータの大元となる型です。

ここで定義したマルチプレクサ関数（`myMux`）は次の3つのパラメータを持ちます。1つ目はブール値の条件、2つ目は条件が真の場合のパスで、3つ目は条件が偽の場合のパスです。真・偽の両方のパスのパラメータは`T`型で、関数呼び出し時に提供される情報となります。関数の機能自体は単純で、デフォルトの値となる`fPath`を定義し、条件が真となるとき`tPath`に値を変更します。この条件は古典的なマルチプレクサ機能です。関数の最後でマルチプレクサの機能を持ったハードウェアを返します。

`UInt`型のような単純な型で、このマルチプレクサ機能を使うと次のようになります。

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

2 つのマルチプレクサパスのタイプは同じである必要があります。以下のように、マルチプレクサの使用法を間違えるとランタイムエラーになります。

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

ここで2つのフィールドを持つ`Bundle`を定義します。

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

最初に`Wire`を作成し、その後に各要素を設定することで`Bundle`型の定数を定義できます。このようにしてパラメータ化されたマルチプレクサで複雑な型を使用できます。

```
  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)
```

この関数の最初の設計時に、デフォルト値を持った`T`型のワイヤを`WireDefault`を使って作成しました。デフォルト値が不要なワイヤを作成したい場合は`fPath.cloneType`を使うことで、Chiselの型を複製できます。次に示す関数はマルチプレクサを別の方法で実装したもの

です。

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

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

10.1.3 型パラメータを持つモジュール

Chiselの型をパラメータとしてモジュールを作成することも可能です。異なるプロセッサ間のデータを移動するNoC（Network on Chip）を作成したい場合に、ルーターのインターフェースのデータフォーマットを、コード中に固定するのは望ましくないので、パラメータ化します。関数で型パラメータを使ったときと同様に、モジュールのコンストラクタへ型パラメータTを追加します。さらにこの型をパラメータの1つとするコンストラクタが必要になります。加えてこの例では、ルーターのポート数を可変にするための変数も追加してあります。

```
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
  // ...
```

このルーターを使用するため、最初にChiselのBundleなどを用いて、ルーティングしたいデータ型を定義する必要があります。

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

ルーターのコンストラクタにユーザ定義のBundleのインスタンスとポート数を渡してルーターを作成します。

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

10.1.4 パラメータ化されたバンドル

この例では、ルーターの入力として、2つの異なる型を持つベクターを使用しました。1つはアドレス用のベクターで、もう1つはデータ用でパラメータ化されたものとなります。よりエレガントな解決策は、次の例のようにこの2つのベクター自体がパラメータ化されたBundleを持っていることです。

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

BundleはChiselのData型のサブタイプであるT型のパラメータを持っています。バンドル内では、パラメータが持つcloneTypeを呼び出すことで、フィールドdataを定義します。しかしコンストラクタのパラメータを使用する場合、このパラメータはクラスのパブリックフィールドになります。 Chiselが複製する必要がある、例えばVecに使用されるような場合には、このパブリックフィールドは邪魔になります。この問題の解決策は、パラメータフィールドをプライベートにすることです。

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

新しいBundleを使うと、ルーターのポートを次のように定義することができます。

```
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
  // ...
```

Payloadをパラメータとして受け取るPortを渡して、ルーターをインスタンス化します。

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

10.2 組合せ論理回路の生成

ChiselではScalaのArrayから派生したChiselの型であるVecを使った論理テーブルを作ることで、簡単にロジックを生成できます。ハードウェア生成時に読み込めるような論理テーブルを記載したデータファイルがあるとします。コード 10.1はScalaの標準ライブラリであるSourceを使って、整数データが含まれる‘data.txt‘を読み込む方法を示しています。

少し難解な式：

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

ScalaのArrayはmapをサポートしたシーケンス（Seq）に暗黙的に変換できます。 mapは、配列の各要素に対して関数を呼び出し、関数の戻り値からなるシーケンスを返します。上記の例で_.U(8.W)は_がScalaの Array から各要素をInt値として扱い、その値をChiselの8ビットのUIntに変換します。 Chiselのオブジェクトであ るVecInitはSeqのようなシーケンスを元にChiselのVecを生成します。

Scalaの力を最大限に利用して、ロジック（テーブル）を生成することができます。例えば、三角関数を表現するためのテーブルを生成したり、デジタルフィルタの定数を計算したり、Chiselで書かれたマイクロプロセッサ用のコードを生成するためにScalaの小さなアセ

```
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)
}
```

コード 10.1: テキストファイルを読んで論理テーブル生成

```
import chisel3._

class BcdTable 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)

  // Convert binary to BCD
  for (i <- 0 to 99) {
    array(i) = ((i/10)<<4) + i%10
  }

  val table = VecInit(array.map(_.U(8.W)))
  io.data := table(io.address)
}
```

コード 10.2: バイナリからBCDへの変換

ンブラを書いたりすることができます。これらの関数はすべて同じコードベース（同じ言語）であり、ハードウェア生成中に実行することができます。

古典的な例は、2進数を binary-coded decimal(英語)/ 二進化十進表現(日本語) (BCD)表現に変換することです。 BCDは、10進数の各桁ごとに4ビットを用いて、数値を10進数で表現するために用いられます。例えば、10進数の13は2進数では1101ですが、BCDでは1と3としてエンコードされ、結果として00010011を得ます。 BCDは数値を16進数よりもユーザフレンドリな数値表現である、10進数で表示するために用いられます。

バイナリをBCDに変換するための表を計算するJavaプログラムを書くことができます。そのJavaプログラムは、プロジェクトに含めることができるVHDLコードを出力します。Javaプログラムは約100行のコードで、コードのほとんどがVHDL文字列を生成します。変換の重要な部分はわずか2行です。

Chiselではこの論理テーブルをハードウェア生成処理の一部として扱うことができます。コード 10.2は BCD への変換バイナリ用テーブルの生成を示しています。

10.3 継承の使用

Chiselはオブジェクト指向言語であり、ハードウェアを構成する要素の1つあるModuleはScalaのクラスです。したがって、親クラスに共通の動作を実装し、継承して使用することができます。ここでは、サンプルコードとともに継承を使用する方法を探ります。

6.2節では、低周波のティック生成に利用可能な種類のカウンタを紹介しました。ここで、リソースの使用量を比較するために複数の種類の実装を実行してみたいとしましょう。最初にするのは抽象クラスでカウンタのパルス出力のインターフェースを定義することです。

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

```
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
}
```

コード 10.3: カウンタを使ったティック生成

```
  })
}
```

コード 10.3は抽象クラスを使った最初の実装で、カウントパルスを発生させるためのカウンタ、カウントアップ処理を備えています。

*ticker*を使って作成した異なるすべての論理は、単一のテストベンチを用いてテストできます。方法は簡単で、テストベンチで*Ticker*の派生クラスを受け入れるようにするだけです。コード 10.4はテスタのためのコードです。 `TickerTester`は複数のパラメータを持ちます。1つ目の型パラメータ `[T <: Ticker]` で`Ticker`または`Ticker`を継承するクラスを受け付けることができます。2つ目は`T`の型を持つDUTで、3つ目は期待するカウントパルスのクロックサイクル数です。テスタではカウントパルスの最初の出力を待ちます。これはテストするDUTの論理によって、スタートが異なる可能性があるためです。その後`tick`がnの周期で繰り返すことを確認します。

最初の簡単なティッカーといくつかの`println`デバッグを用いて、テスタ自身のテストができます。このようにして簡単なティッカーとテスタが正しいことを確認したのち、異なるバージョンのティッカーの実装へ進むことができます。コード 10.5は0までカウントダウンカウンタを使ったカウントパルスの生成を示しています。コード 10.6は-1までカウントダウンする凝った実装で、コンパレータを使用しないため、より少ないハードウェアで済みます。

ScalaTestの仕様を用いて、これら3つの異なるティッカーをテストできます。

私たちは、ScalaTest仕様を使用してティッカーの異なるバージョンのインスタンスを作成し、一般的なテストベンチに渡すことでティッカーのすべての3つのバージョンをテストすることができます。コード 10.7はテストの仕様を記載したものです。テストを実行するために必要なのは、次のコマンドを実行することのみです。

```
$ sbt "testOnly TickerSpec"
```

```
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)
  }
}
```

コード 10.4: 異なったティッカーに対するテスタコード

```
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
}
```

コード 10.5: カウントダウンカウンタを使ったティック(カウントパルス)の生成

```
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
  }
}
```

コード 10.6: -1までカウントダウンするカウンタを使ったティック(カウントパルス)の生成

```
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)
  }
}
```

コード 10.7: ティッカーテストのための ScalaTest の仕様

10.4 関数型プログラミングによるハードウェア生成

Chiselもそうですが、Scalaは関数型プログラミングをサポートしています。関数を使ってハードウェアを表現し、それらのハードウェアコンポーネントを（いわゆる「高階関数」を使う）関数型プログラミングと組み合わせることができます。ベクトルの和という簡単な例から始めましょう。

```
  def add(a: UInt, b:UInt) = a + b

  val sum = vec.reduce(add)
```

まず、関数 addで加算器のハードウェアを定義します。ベクトルは vecにあります。Scalaのreduce()メソッドは、コレクションの全要素を2項演算で結合し、1つの値を生成します。 reduce() メソッドは、左から順番に縮小(reduce)します。最初の 2 つの要素を取り、演算を実行します。その結果を次の要素と演算するという処理を、単一の結果が残るまで繰り返します。

要素を結合する関数は reduce のパラメータとして渡されます。ここでは加算器を返すaddになります。結果として得られるハードウェアは、ベクトル vecの要素の和を計算する加算器のチェーンとなります。

(単純な)add関数を定義する代わりに、匿名関数(anonymous function)を用いて加算処理を提供し、Scalaのワイルドカード “_” を使った2つのオペランドで表現することもできます。

```
  val sum = vec.reduce(_ + _)
```

このワンライナー(1行コード)で、加算器のチェーン（連鎖）を生成しました。チェーンで構成する加算関数は理想的ではありません。ツリーの方が組み合せ回路の遅延が少なくなります。論理合成ツールの加算器チェーン再配置(最適化)機能を信用しないのであれば、ChiselのreduceTreeメソッドを使ってツリー型の加算器を生成することができます。

```
  val sum = vec.reduceTree(_ + _)
```

11 デザイン例

この章では、FIFOバッファなど、より大きなデザインのビルディングブロックとして使用するいくつかの小さなデジタル回路を探索します。別の例として、シリアルインターフェース(UARTとも呼ばれます)も設計しますが、これはFIFOバッファを使用します。

11.1 FIFO バッファ

ライタ (送信側) とリーダ (受信側) の間にバッファを設けることで、ライタとリーダを切り離すことができます。よく使われるバッファは、先入れ先出し (FIFO(英語)/ (日本語)) バッファです。図 11.1は、ライタ、FIFO、リーダを示しています。 write信号がアクティブなときにdinのデータがライタによってFIFOに入れられます。 read信号がアクティブなときにFIFOがリーダによりdoutにDataが読み出されます。

FIFOは初期状態では空で、empty信号で状態が示されています。空のFIFOからの読み出しは通常未定義です。データが書き込まれても読み出されない場合、FIFOはfullになります。フルFIFOへの書き込みは通常無視され、データは失われます。言い換えると、empty信号とfull信号はハンドシェイク信号として機能します。

FIFOにはいくつかの異なる実装方法があります : 読み出し、書き込みポインタとオンチップメモリ、小さなステートマシンと単純なレジスタのチェーン、といった組み合わせです。小規模なバッファ(最大数十エレメント)の場合、個々のレジスタをバッファのチェーンに接続して構成されたFIFOは、リソース要件が小さい単純な実装となります。バブルFIFOのコードはchisel-examplesリポジトリにあります。 [1]

まずは、ライタとリーダ側のIO信号の定義から始めます。データのサイズはsizeで設定可能です。 dinが書き込みデータであり、 write信号でその書き込みが行われます。信号fullはライタ側で flow control(英語)/ フロー制御(日本語) を行います。

```
class WriterIO(size: Int) extends Bundle {
  val write = Input(Bool())
  val full = Output(Bool())
  val din = Input(UInt(size.W))
}
```

リーダ側はdoutでデータを提供し、 readでリードを開始します。 empty信号は、リーダ側のフロー制御を担当します。

```
class ReaderIO(size: Int) extends Bundle {
  val read = Input(Bool())
  val empty = Output(Bool())
  val dout = Output(UInt(size.W))
}
```

コード 11.1は1段のバッファを示しています。バッファはWriterIO型のエンキューイングポートenqとReaderIO型のデキューイングポートdeqを持っています。バッファのステート

[1] 完全性を確保するために、Chisel bookリポジトリにはFIFOコードのコピーも含まれています。

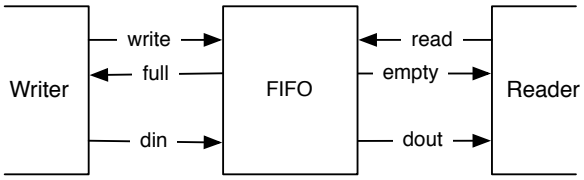


図 11.1: writer, FIFO バッファと reader 回路

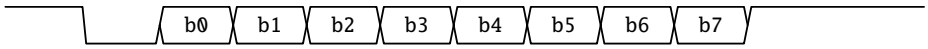


図 11.2: UARTの1バイト転送の波形図

要素は、データを保持する1つのレジスタ(dataReg)と、単純なFSM用の1つのステートレジスタ(stateReg)です。 FSMには2つの状態しかありません。バッファがemptyかfullかのどちらかです。バッファがemptyの場合、書き込みは入力データを書き込んでfull状態に変更します。バッファがfullの場合、読み出しはデータを読みだしてemptyの状態に変わります。IOポートのfullとemptyは、ライタとリーダのバッファの状態を表しています。

コード 11.2は、完全なFIFOを示しています。完全なFIFOも、それぞれのFIFOバッファと同じIOインターフェースを持っています。 BubbleFifoは、パラメータとしてデータワードのsizeとバッファステージのdepthを持っています。 depth個分のFifoRegisterから、depthステージのバブルFIFOを構成します。 Scalaの配列 (Array) にFifoRegisterを詰め込む (fill) ことで、ステージを構築します。 Scalaの配列にはハードウェア的な意味はなく、作成したバッファへの参照を持つためのコンテナを提供してくれるだけです。 Scalaのforループでは、それぞれのバッファを接続します。最初のバッファのエンキューはFIFO全体のエンキューIOに接続され、最後のバッファのデキューはFIFO全体のデキュー側に接続されています。

それぞれのバッファを接続してFIFOキューを実装するというアイデアは、データが泡(バブル)のようにキューを通るので、バブルFIFOと呼ばれています。これは、データレートがクロックレートよりもかなり遅い場合、例えば次節で提示されるシリアルポートのためのデカップリングバッファなどには、シンプルで良好な解決策です。

しかし、データレートがクロック周波数に近づくと、バブルFIFOには2つの制限があります : (1) 各バッファの状態は空と満杯を切り替えなければならないため、FIFOの最大スループットは1ワードあたり2クロックサイクルです。 (2) データの泡(バブル)はすべてのFIFOを伝搬する必要があるため、入力から出力までの間、少なくともバッファの数の分はレイテンシが発生します。 11.3節でFIFOの他の可能な実装をご紹介します。

11.2 シリアルポート

シリアルポート(UART(英語)/ (日本語) または RS-232(英語)/ (日本語) とも呼ばれる)は、ラップトップとFPGAボードの間で通信するための最も簡単なオプションの1つです。その名の通り、データはシリアルに送信されます。 8ビットのバイトデータは次のように送信されます : 1つのスタートビット(0)、8ビットのデータ、最下位ビットが最初に送信され、その後1つまたは2つのストップビット(1)が送信されます。データが送信されていないときは、出力は1となります。図 11.2に1バイト送信した場合のタイミング図を示します。

モジュールごとの機能を最小限に抑えたモジュールでUARTを設計しました。送信機(TX)、受信機(RX)、バッファ、そしてそれらの基本コンポーネントを使用していま

```
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
}
```

コード 11.1: バブルFIFOのシングルステージ

```
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
}
```

コード 11.2: FIFOバブルステージの配列で構成されたFIFO

す。

まず、インターフェースであるポートの定義が必要です。 UART設計では、送信機から見た方向で、ready/validハンドシェイクインターフェースを使用します。

```
class Channel extends Bundle {
  val data = Input(Bits(8.W))
  val ready = Output(Bool())
  val valid = Input(Bool())
}
```

ready/validインターフェースの慣習は、readyとvalidの両方がアサートされたときにデータが転送されるというものです。

```
1  class Tx(frequency: Int, baudRate: Int) extends Module {
2    val io = IO(new Bundle {
3      val txd = Output(Bits(1.W))
4      val channel = new Channel()
5    })
6
7    val BIT_CNT = ((frequency + baudRate / 2) / baudRate - 1).asUInt()
8
9    val shiftReg = RegInit(0x7ff.U)
10   val cntReg = RegInit(0.U(20.W))
11   val bitsReg = RegInit(0.U(4.W))
12
13   io.channel.ready := (cntReg === 0.U) && (bitsReg === 0.U)
14   io.txd := shiftReg(0)
15
16   when(cntReg === 0.U) {
17
18     cntReg := BIT_CNT
19     when(bitsReg =/= 0.U) {
20       val shift = shiftReg >> 1
21       shiftReg := Cat(1.U, shift(9, 0))
22       bitsReg := bitsReg - 1.U
23     }.otherwise {
24       when(io.channel.valid) {
25         // two stop bits, data, one start bit
26         shiftReg := Cat(Cat(3.U, io.channel.data), 0.U)
27         bitsReg := 11.U
28       }.otherwise {
29         shiftReg := 0x7ff.U
30       }
31     }
32
33   }.otherwise {
34     cntReg := cntReg - 1.U
35   }
36 }
```

コード 11.3: シリアルポートのトランスミッタ（送信回路）

コード 11.3はベアボーン (最小限の) シリアル送信機(Tx)を示しています。 IOポートはシ

リアルデータが送信されるtxdポートと、送信機がシリアル化して送信する文字を受け取るChannelです。正しいタイミングを生成するために、1つのシリアルビットのクロックサイクルの時間を計算して定数を算出しています。

ここでは、3つのレジスタを使用しています。(1) データをシフト(シリアライズ)するためのレジスタ(shiftReg)、(2) 正しいボーレートを生成するためのカウンタ(cntReg)と、(3) まだシフトアウトする必要があるビット数のカウンタ。FSMのための追加のステートレジスタは必要なく、すべての状態はこれら3つのレジスタでエンコードされます。

カウンタcntRegはずっと動作しています(0までカウントダウンし、0になると開始値にリセットされます)。すべての動作は、cntRegが0のときにのみ行われます。最小限の送信機を構築しているので、データを保存する場所はシフトレジスタのみです。したがって、cntRegが0で、シフトアウトするビットが残っていないときだけ、Channelポートはready状態となります。

IOポートtxdは、シフトレジスタの最下位ビットに直接接続されています。

シフトアウトするビットがさらにある場合(bitsReg =/= 0.U)、ビットを右にシフトし、先頭から1 (送信機のアイドルレベル) で埋めます。シフトアウトする必要がない場合は、チャネルにデータが含まれているかどうかをチェックします(validポートで示されています)。そうであれば、シフトアウトされるビット列は、1つのスタートビット(0)、8ビットのデータ、2つのストップビット(1)で構成されます。したがって、ビット数は11となります。

この非常に小さな送信機は、追加のバッファを持たず、シフトレジスタが空のときと、cntRegが0のときのクロックサイクルでだけ、新しい文字を受け付けることができます。cntRegが0のときにのみ新しいデータを受け入れるということは、シフトレジスタに空きがある場合には、レディフラグも解除 (デアサート) されることを意味します。しかし、この“複雑さ”を送信機に追加するのではなく、バッファにまかせることにします。

```
1  class Buffer extends Module {
2    val io = IO(new Bundle {
3      val in = new Channel()
4      val out = Flipped(new Channel())
5    })
6
7    val empty :: full :: Nil = Enum(2)
8    val stateReg = RegInit(empty)
9    val dataReg = RegInit(0.U(8.W))
10
11   io.in.ready := stateReg === empty
12   io.out.valid := stateReg === full
13
14   when(stateReg === empty) {
15     when(io.in.valid) {
16       dataReg := io.in.data
17       stateReg := full
18     }
19   }.otherwise { // full
20     when(io.out.ready) {
21       stateReg := empty
22     }
23   }
24   io.out.data := dataReg
25 }
```

コード 11.4: ready/valid インターフェースを持つ1バイトバッファ

```
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
}
```

コード 11.5: バッファを追加したトランスミッタ

コード 11.4は、バブルFIFO用のFIFOレジスタに似たシングルバイトバッファを示しています。入力ポートはChannelインターフェースで、出力は方向が反転したChannelインターフェースです。バッファには、emptyかfullかを示すためのミニマムなステートマシンが含まれています。バッファ駆動のハンドシェイク信号(in.readyとout.valid)はステートレジスタに依存します。

ステートがfullで、入力データがvalidな場合は、データを登録してfullステートに移行します。ステートがfullになり、下流側の受信機がreadyになったら、下流側のデータ転送が行われ、ステートをemptyに戻します。

このバッファを用いることで、ベアボーン送信機を拡張することができます。コード 11.5は、送信機Txとその手前の一段のバッファとの接続を示しています。このバッファは、Txがシングルクロックサイクルのみreadyであるという問題を緩和します。この問題の解決はバッファモジュールに任せました。シングルワードバッファの実FIFOへの拡張は簡単にでき、送信機やシングルバイトバッファを変更する必要はありません。

コード 11.6に受信機(Rx)のコードを示します。受信機はシリアルデータのタイミングを再構築する必要があるので、少し厄介です。受信機はスタートビットの立ち下がりを待ちます。このイベントから、受信機はビット0の中央に位置するように1.5ビット分の時間を待ちます。その後、ビット毎にシフトしていきます。この2つの待ち時間をSTART_CNTとBIT_CNTとして観測することができます。どちらも同じカウンタ(cntReg)を使用します。 8ビットシフトインした後、valRegのバイトデータが有効であることを通知します。

コード 11.7は、フレンドリなメッセージを送信するシリアルポート送信機の使い方を示しています。メッセージはScalaの文字列(msg)で定義し、それをUIntのChisel Vecに変換しています。 Scalaの文字列はmapメソッドをサポートするシーケンスです。 mapメソッドは関数リテラルを引数にとり、その関数を各要素に適用し、関数の戻り値のシーケンスを構築します。関数リテラルの引数が一つしかない場合、今回のように引数は_で表すことができます。この関数リテラルは、Chiselメソッド.Uを使ってScalaのCharをChiselのUIntに変換します。シーケンスはVecInitに渡され、ChiselのVecが構築されます。バッファリングされた送信機にベクトルtextとカウンタcntRegのインデックスを使ってそれぞれの文字を渡します。各ready信号で、文字列すべてが送信されるまでカウンタを増加させます。送信は、最後の文字が送出されるまでvalidをアサートし続けます。

コード 11.8は、受信機と送信機を接続した場合の使い方を示しています。この接続により、受信した各文字を(エコーして)送り返すEcho回路が生成されます。

```
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
}
```

コード 11.6: シリアルポートのレシーバ（受信回路）

```
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
  }
}
```

コード 11.7: シリアルポートから “Hello World!” を出力

```
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
}
```

コード 11.8: シリアルポートでのデータのエコー処理

11.3 FIFO設計のバリエーション

この節では、様々な種類のFIFOキューを実装します。これらの実装を互換性のあるものにするために、私たちは10.3節で紹介したように、継承を使用します。

11.3.1 FIFOのパラメータ化

ここでは、任意のChiselデータ型をバッファリングできるように、Chisel型をパラメータとする抽象FIFOクラスを定義しています。抽象クラスでは、パラメータのdepthが有用な値であることもテストしています。

```
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")
}
```

11.1節では、write, full, din, read, empty, と doutなどの信号に共通の名前を付けて、インターフェースのための独自の型を定義しました。このようなバッファの入出力は、データとハンドシェイクのための2つの信号で構成されます(たとえば、FIFOがfullでないときにwriteします)。

しかし、このハンドシェイクは、ready-validインターフェースと呼ばれるものに一般化することができます。例えば、FIFOがreadyのときに、要素をエンキュー(FIFOへの書き込み)することができます。この要素のエンキューをvalid信号を用いて書き込み側に通知します。このready-validインターフェースは非常に一般的なものなので、ChiselはDecoupledIOでこのインターフェースの定義を以下のように提供しています : [2]

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

DecoupledIOインタフェースを使用して、FIFOのインタフェースを定義します : enqエンキューとdeqデキューポートを備えたFifoIOであって、read-validインターフェースで構成されます。 DecoupledIOインターフェースは、プロデューサ（送信側）の視点から定義されます。したがって、FIFOのエンキューポートは信号の方向を反転させる必要があります。

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

抽象的な基底クラスとインターフェースを使用することで、異なるパラメータ(速度、面積、電力、または単純さ)に対して最適化されたFIFOの実装とすることができます。

11.3.2 バブルFIFOの再設計

標準的なready-validインターフェースを使用し、Chiselのデータ型でパラメータ化することで、11.1節のバブルFIFOを再定義することができます。

[2] DecoupledIOは、実際には抽象クラスを拡張しており、これは単純化されています。

```
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
}
```

コード 11.9: ready-valid インターフェースを持つバブルFIFO

コード 11.9は、Bubble FIFOを、ready-validインターフェースでリファクタリングしたものです。 BubbleFifoのプライベートクラスとしてBufferコンポーネントを内包していることに注意してください。このヘルパクラスは、このコンポーネントのためだけに必要なものなので、名前空間を汚さないように隠しています。また、バッファクラスも簡素化されています。バッファの状態(fullかemptyか)を記録するために、FSMの代わりに1ビットのfullRegを使用します。

バブルFIFOはシンプルでわかりやすく、最小限のリソースしか使用しません。ただし、各バッファステージはemptyとfullの間で行き来する必要があるため、このFIFOの最大帯域幅は1ワードあたり2クロックサイクルです。

プロデューサがvalidで、コンシューマ（受信側）がreadyのときに新しいワードを受け付けることができるように、バッファ内の両方のインタフェースを見るようにすることもできます。しかし、これはコンシューマのハンドシェイクからプロデューサのハンドシェイクへの組み合わせパスを作ることになり、ready-validプロトコルのセマンティクスに違反することになります。

11.3.3 ダブルバッファFIFO

一つの解決策は、バッファレジスタがfullになってもreadyな状態を保つことです。プロデューサからのデータワードを受け取れるようにするためには、コンシューマがreadyでないときのために、シャドウレジスタと呼ばれる第2のバッファが必要です。バッファがfullになると、新しいデータがシャドウレジスタに格納され、readyがデアサートされます。コンシューマが再びreadyになると、データはデータレジスタからコンシューマに転送され、シャドウレジスタからデータレジスタに転送されます。

```
1  class DoubleBufferFifo[T <: Data](gen: T, depth: Int) extends Fifo(gen: T,
       depth: Int) {
2
3    private class DoubleBuffer[T <: Data](gen: T) extends Module {
4      val io = IO(new FifoIO(gen))
5
6      val empty :: one :: two :: Nil = Enum(3)
7      val stateReg = RegInit(empty)
8      val dataReg = Reg(gen)
9      val shadowReg = Reg(gen)
10
11     switch(stateReg) {
12       is (empty) {
13         when (io.enq.valid) {
14           stateReg := one
15           dataReg := io.enq.bits
16         }
17       }
18       is (one) {
19         when (io.deq.ready && !io.enq.valid) {
20           stateReg := empty
21         }
22         when (io.deq.ready && io.enq.valid) {
23           stateReg := one
24           dataReg := io.enq.bits
25         }
26         when (!io.deq.ready && io.enq.valid) {
```

```
27            stateReg := two
28            shadowReg := io.enq.bits
29          }
30        }
31        is (two) {
32          when (io.deq.ready) {
33            dataReg := shadowReg
34            stateReg := one
35          }
36
37        }
38      }
39
40      io.enq.ready := (stateReg === empty || stateReg === one)
41      io.deq.valid := (stateReg === one || stateReg === two)
42      io.deq.bits := dataReg
43    }
44
45    private val buffers = Array.fill((depth+1)/2) { Module(new
          DoubleBuffer(gen)) }
46
47    for (i <- 0 until (depth+1)/2 - 1) {
48      buffers(i + 1).io.enq <> buffers(i).io.deq
49    }
50    io.enq <> buffers(0).io.enq
51    io.deq <> buffers((depth+1)/2 - 1).io.deq
52  }
```

コード 11.10: ダブルバッファを持つ FIFO

コード 11.10はダブルバッファを示しています。各バッファ要素が2つのエントリを格納できるので、バッファ要素は半分(depth/2)しか必要ありません。 DoubleBufferにはdataRegとshadowRegの2つのレジスタがあります。コンシューマは常にshadowRegから供給されます。ダブルバッファにはempty, one,とtwoの3つの状態があり、バッファがどれくらい一杯かを示します。バッファの状態がemptyかoneのとき、バッファは新しいデータの受け入れがreadyな状態です。バッファの状態がoneかtwoのとき、バッファはvalidなデータを持っています。

FIFOがフルスピードで実行されて、コンシューマが常にreadyであれば、ダブルバッファの定常状態はoneになります。コンシューマがreadyをデアサートした場合のみ、キューは一杯になり、バッファはtwoの状態になります。しかし、シングルバブルFIFOと比較すると、キューの再起動に要する時間は、同じバッファ容量の場合の半分のクロックサイクル数しかかかりません。

11.3.4 レジスタメモリ型FIFO

ソフトウェアエンジニアリングの経験がある方は、私たちが複数の小さなバッファ要素からハードウェアキューを構築し、全てを並列に実行しながら上流および下流の要素ともハンドシェイクも行っていることを不思議に思われるかもしれません。小さなバッファの場合は、多分これが最も効率的な実装です。

ソフトウェアのキューは、通常、単一スレッド内の順に実行されるコードによって使用されます。または、プロデューサとコンシューマのスレッドを分離するために使われます。

固定サイズのFIFOキューは通常 circular buffer(英語)/ リングバッファ, 循環バッファ(日本語)として実装されます。 2つのポインタが、キュー用に確保されたメモリ内の読み取り位置と書き込み位置を指します。 ポインタは、メモリの終端に到達すると、メモリの先端に戻されます。 2つのポインタの差分は、キュー内の要素数です。 2つのポインタが同じアドレスを指すとき、キューはemptyかfullのどちらかになります。 emptyかfullかを区別するためには、別のフラグが必要です。

このようなメモリベースのFIFOキューをハードウェアでも実装することができます。小さなキューの場合は、レジスタファイル(すなわち、`Reg(Vec())`)を使用することができます。コード 11.11は、メモリと読み書きポインタを使って実装されたFIFOキューを示しています。

```
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
}
```

コード 11.11: レジスタで構成したメモリを持つ FIFO

インクリメントされたのちバッファの終端で元に戻されるという、同じ動作をするポインタが2つあるので、そのためのカウンタを実装する関数counterを定義しました。log2Ceil(depth).Wを用いてカウンタのビット長を計算します。次に取り得る値は、1だけインクリメントされるか、0に戻るかのいずれかです。カウンタは入力incrがtrue.Bのときだけインクリメントされます。

さらに、次に取り得る値(インクリメントされた値か、0に戻された値)も必要なので、この値もcounter関数から返します。 Scalaでは、いわゆる*tuple*を返すことができます。これは、複数の値を保持するための単なるコンテナです。このようなtupleを作成する構文は、カンマで区切られた値を括弧で囲むだけです:

```
  val t = (v1, v2)
```

代入の左辺の括弧表記を使用して、tupleを分解できます:

```
val (x1, x2) = t
```

メモリにはChiselデータ型genのベクタ (Reg(Vec(depth, gen))) を使ったレジスタを使用します。読み書きポインタをインクリメントし、関数counterで読み書きポインタを作成するための2つの信号を定義しています。両方のポインタが等しくなると、バッファはemptyかfullになります。 emptyとfullを示すために、2つのフラグを定義しています。

プロデューサがvalidをアサートし、FIFOがfullでない場合は、次のように動作します: (1) バッファへ書き込み、(2) emptyRegがデアサートされていることを確認します。 (3) 次のクロックサイクルで書き込みポインタが読み取りポインタに追いつく場合はバッファがfullになるようにマークし(現在の読み取りポインタと次の書き込みポインタを比較し)、(4) 書き込みカウンタをインクリメントする信号を送ります。

コンシューマがreadyで、FIFOがemptyでない場合は、次のように動作します: (1) fullRegがデアサートされていることを確認し、(2) 読み出しポインタが次のクロックサイクルで書き込みポインタに追いついた場合はバッファをempty状態にし、(3) 読み出しカウンタをインクリメントするための信号を送ります。

FIFOの出力は、読み出しポインタアドレスのメモリ内容です。 readyフラグとvalidフラグは、単純にfullフラグとemptyフラグから派生したものです。

11.3.5 オンチップメモリ型FIFO

さきほどのFIFOでは、メモリを表現するためにレジスタファイルを使用していましたが、これは小さなFIFOの場合には良い解決策です。より大きなFIFOの場合は、オンチップメモリを使用する方が良いでしょう。コード 11.12は、ストレージに同期メモリを使用したFIFOを示しています。

```
1 class MemFifo[T <: Data](gen: T, depth: Int) extends Fifo(gen: T, depth:
      Int) {
2
3   def counter(depth: Int, incr: Bool): (UInt, UInt) = {
4     val cntReg = RegInit(0.U(log2Ceil(depth).W))
5     val nextVal = Mux(cntReg === (depth-1).U, 0.U, cntReg + 1.U)
6     when (incr) {
7       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
      }

```

```
61      }
62      is(full) {
63        when(io.deq.ready) {
64          when(!emptyReg) {
65            stateReg := valid
66            fullReg := false.B
67            emptyReg := nextRead === writePtr
68            incrRead := true.B
69          } otherwise {
70            stateReg := idle
71          }
72
73        }
74      }
75    }
76
77    io.deq.bits :=  Mux(stateReg === valid, data, shadowReg)
78    io.enq.ready := !fullReg
79    io.deq.valid := stateReg === valid || stateReg === full
80  }
```

コード 11.12: オンチップメモリで構成した FIFO

読み出しポインタと書き込みポインタの取り扱いは、レジスタメモリのFIFOと同じです。しかし、レジスタファイルの読み出し結果は同じクロックサイクルで利用可能でしたが、同期オンチップメモリでは、読み出し結果が次のクロックサイクルで利用可能となります。

したがって、このレイテンシを処理するために、いくつかの追加のFSMとシャドウレジスタが必要です。メモリを読み取り、キューの先頭の値を出力ポートに届けます。その値が消費されない場合は、メモリから次の値を読み出す間、シャドウレジスタshadowRegに格納する必要があります。ステートマシンには、3つの状態があります : (1)FIFOがempty、(2)メモリから読み出した有効なデータを保持した状態、(3)シャドウレジスタのキューの先頭とメモリからの有効なデータ(次の要素)を保持した状態、です。

メモリベースのFIFOは、効率的に大量のデータをキューに保持することができ、レイテンシの低下が少なくて済みます。最後のデザインでは、FIFOの出力はメモリの読み取りから直接届く場合があります。このデータパスがデザインのクリティカルパスにある場合、2つのFIFOを組み合わせることで、デザインを簡単にパイプライン化することができます。コード 11.13はそのような組み合わせを示しています。メモリベースのFIFOの出力には、メモリの読み取りパスを出力から切り離すために、シングルステージのダブルバッファFIFOを追加しています。

```
1   class CombFifo[T <: Data](gen: T, depth: Int) extends Fifo(gen: T, depth:
        Int) {
2
3     val memFifo = Module(new MemFifo(gen, depth))
4     val bufferFIFO = Module(new DoubleBufferFifo(gen, 2))
5     io.enq <> memFifo.io.enq
6     memFifo.io.deq <> bufferFIFO.io.enq
7     bufferFIFO.io.deq <> io.deq
8   }
```

コード 11.13: メモリベースのFIFOとダブルバッファFIFOの組み合わせ

11.4 演習

この章の演習には、2つの演習が含まれているので、少し長くなります: (1) バブルFIFOを探究してから別のFIFOデザインを実装し、(2) 同様にUARTについても、探究してからその機能を拡張します。両方の練習問題のソースコードはchisel-examplesリポジトリに含まれています。

11.4.1 バブルFIFOを探る

FIFOのソースコードには、さまざまな読み書き動作を行って、value change dump (VCD)形式の波形を生成する、テスタも含まれています。 VCDファイルは、GTKWaveなどの波形ビューアでみることができます。リポジトリ内のFifoTesterをみてください。リポジトリには、例を実行するための`Makefile`が含まれています。FIFOの例では、次のように入力します :

```
$ make fifo
```

このmakeコマンドでFIFOをコンパイルし、テストを実行し、波形を見るためにGTKWaveを起動します。テスタと生成された波形をみてみます。

最初のサイクルでは、テスタは1つのワードを書き込みます。そのワードが*bubble FIFO*と名付けられたFIFOを通って伝搬する様子を波形で観察できます。バブルFIFOという名前は、FIFOを通過するデータワードのレイテンシが FIFOの深さに等しいことも示しています。

次のテストでは、FIFOがfullになるまでFIFOを埋めます。その後、読み取りが1回行われます。空のワードがFIFOのリーダ側からライタ側に向かって伝搬する様子に注目してください。バブルFIFOがfullになると、読み出しがライタ側に伝わるまでにバッファの深さのレイテンシが必要になります。

テストの最後には、最大速度で書き込みと読み出しを試みるループが含まれています。バブルFIFOが最大帯域幅、つまり1ワードあたり2クロックサイクルで動作しているのがわかります。バッファステージは、1ワードの転送のために常にemptyとfullの間で行き来することになります。

バブルFIFOはシンプルで、小さなバッファの場合、必要なリソースが少なくて済みます。nステージのバブルFIFOの主な欠点は次の通りです : (1) 最大スループットは、2クロックサイクルごとに1ワードであること、(2) データワードは、ライタ側からリーダ側へnクロックサイクルで移動しなければならないこと、(3) fullになったFIFOは、再起動のためにnクロックサイクルが必要であることです。

これらの欠点は、 circular buffer(英語)/ リングバッファ, 循環バッファ(日本語) を用いたFIFO実装によって解決します。循環バッファは、メモリと読み書きポインタを用いて実装することができます。 FIFOを4つの要素から成る循環バッファとして、同じインターフェースを使用して実装し、異なる動作を示すことをテスタで確認してみてください。循環バッファの初期実装では、簡単に実装を行うためにレジスタのベクトル (`Reg(Vec(4, UInt(size.W)))`) を使用します。

11.4.2 UART

UARTの例では、シリアルポートとラップトップ用のシリアルポート(通常はUSB接続)を備えたFPGAボードが必要です。 FPGAボードとラップトップのシリアルポートの間にシリアルケーブルを接続します。 WindowsのHypertermやLinuxの`gtkterm`などのターミナルプログラムを起動します:

```
$ gtkterm &
```

正しいデバイスを使用するようにポートを構成します。USB UARTではたいていの場合、これは`/dev/ttyUSB0`でしょう。 ボーレートを115200に設定し、パリティやフロー制御(ハンドシェイク)をしないようにします。以下のコマンドで、UART用のVerilogコードを作成することができます:

```
$ make uart
```

次に、合成ツールを使用してデザインを合成します。リポジトリには、DE2-115 FPGAボード用のQuartusプロジェクトが含まれています。Quartusで実行ボタンを押してデザインを合成し、FPGAをコンフィグします。コンフィギュレーション後、ターミナルにグリーティングメッセージが表示されるはずです。

LEDの点滅の例をUARTで拡張して、LEDが消灯しているときと点灯しているときにシリアルラインに0と1を書き込みます。 `Sender`の例と同様に`BufferedTx`を使用します。

文字の出力が遅い(1秒間に2文字)ので、Ready/validハンドシェイクを行わずに、UART送信レジスタにデータを書き込むことができます。この例を拡張して、ボーレートが許す限りの速さで0〜9の繰り返しの数字を書き込むようにします。この場合、ステートマシンを拡張してUARTの状態をポーリングし、送信バッファが空いているかどうかをチェックする必要があります。

サンプルコードには、Tx用のバッファが1つだけです。送信機と受信機にバッファリングのためのFIFOを自由に追加してみてください。

11.4.3 FIFOの探究

専用レジスタに4つのバッファ要素を持つシンプルなFIFOを書いてください。単にオーバーフロー (wrap-around) する、2ビットの読み書きカウンタを使用します。さらに単純化して、読み出しおよび書き込みカウンタが等しい時をFIFOが (fullもしくはemptyでなく) emptyとして扱うことを考えてください。つまり、最大で3つの要素を格納することができます。この単純化により、コード 11.11の例にあるカウンタ関数と、同じポインタ値を持った場合のfullもしくはemptyの区別が不要となります。ポインタの値だけで導出できるので、emptyフラグやfullフラグは必要ありません。このデザインはとても簡単でしょ?

提示された異なるFIFO設計は、以下の特性に関連して設計上のトレードオフが異なります: (1) 最大スループット、(2) フォールスルーレイテンシ、(3) リソース要件、(4) 最大クロック周波数。ソースをchisel-examplesから入手し、FPGAに合成して、さまざまなサイズのFIFO のすべてのバリエーションを調べてみましょう。 4ワード、16ワード、256ワードのちょうどよいFIFOのサイズはどれでしょう?

12 プロセッサの設計

本書の最後の章として、中規模のプロジェクトを紹介します：マイクロプロセッサの設計、シミュレーション、テストです。このプロジェクトを管理しやすくするために、単純なアキュムレータマシンを設計します。このプロセッサはLeros [8]と呼ばれ、オープンソースでhttps://github.com/leros-dev/lerosから入手できます。これは高度な例であり、提示されるコード列を理解するためには、いくつかのコンピュータアーキテクチャの知識が必要であることを言及したいと思います。

Lerosはシンプルな設計ですが、Cコンパイラのターゲットとしては十分な機能を持っています。命令の記述は1ページに収まるようになっており、表 12.1に一覧があります。その表中でAはアキュムレータ、PCはプログラムカウンタ、iは即値(0〜255)、Rnはレジスタn(0〜255)、oはPCとの相対的な分岐オフセット、ARはメモリアクセス用のアドレスレジスタを表しています。

12.1 ALUから始める

プロセッサの中心的な構成要素は、 arithmetic logic unit(英語)/ 算術論理演算装置(日本語) 、略してALUです。そこで、まずALUのコーディングとテストベンチから始めます。最初に、ALUのさまざまな演算を表すEnumを定義します：

```
object Types {
  val nop :: add :: sub :: and :: or :: xor :: ld :: shr :: Nil = Enum(8)
}
```

ALUは通常、2つのオペランド入力(aとbと呼ぶ)と、関数を選択するための演算op(またはオペコード)入力と、出力yを持ちます。コード 12.1はALUを示しています。

最初に3つの入力信号名を短く定義します。 switch文はresの計算のロジックを定義します。 resのデフォルトの代入値は0とします。 switch文はすべての操作を列挙し、それに応じて式を代入します。全ての操作は、Chisel式に直接マッピングされます。最後に、結果のresをALU出力yに代入します。

テストのため、コード 12.2に示すように、ALU関数を平易なScalaで記述しました。このChiselとScalaでの2重の実装では、仕様中の誤りを検出することはできませんが、いくつかの簡単なチェックを行うことはできます。テストベクタとしていくつかの境界値や、まれな値を使用します:

```
  // Some interesting corner cases
  val interesting = Array(1, 2, 4, 123, 0, -1, -2, 0x80000000, 0x7fffffff)
```

両方の入力に、これらの値を使用してすべての関数をテストします：

```
  def test(values: Seq[Int]) = {
    for (fun <- add to shr) {
      for (a <- values) {
        for (b <- values) {
          poke(dut.io.op, fun)
```

Opcode	Function	Description
add	A = A + Rn	Add register Rn to A
addi	A = A + i	Add immediate value i to A
sub	A = A - Rn	Subtract register Rn from A
subi	A = A - i	Subtract immediate value i from A
shr	A = A >>> 1	Shift A logically right
load	A = Rn	Load register Rn into A
loadi	A = i	Load immediate value i into A
and	A = A and Rn	And register Rn with A
andi	A = A and i	And immediate value i with A
or	A = A or Rn	Or register Rn with A
ori	A = A or i	Or immediate value i with A
xor	A = A xor Rn	Xor register Rn with A
xori	A = A xor i	Xor immediate value i with A
loadhi	A_{15-8} = i	Load immediate into second byte
loadh2i	A_{23-16} = i	Load immediate into third byte
loadh3i	A_{31-24} = i	Load immediate into fourth byte
store	Rn = A	Store A into register Rn
jal	PC = A, Rn = PC + 2	Jump to A and store return address in Rn
ldaddr	AR = A	Load address register AR with A
loadind	A = mem[AR+(i << 2)]	Load a word from memory into A
loadindbu	A = mem[AR+i]$_{7-0}$	Load a byte unsigned from memory into A
storeind	mem[AR+(i << 2)] = A	Store A into memory
storeindb	mem[AR+i]$_{7-0}$ = A	Store a byte into memory
br	PC = PC + o	Branch
brz	if A == 0 PC = PC + o	Branch if A is zero
brnz	if A != 0 PC = PC + o	Branch if A is not zero
brp	if A >= 0 PC = PC + o	Branch if A is positive
brn	if A < 0 PC = PC + o	Branch if A is negative
scall	scall A	System call (simulation hook)

表 12.1: Leros の命令セット

```
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
}
```

コード 12.1: Leros の ALU

```
  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
    }
  }
```

コード 12.2: Scala で記述した Leros ALU 関数

```
          poke(dut.io.a, a)
          poke(dut.io.b, b)
          step(1)
          expect(dut.io.y, alu(a, b, fun.toInt))
        }
      }
    }
  }
```

32ビット引数のすべての値に対する網羅的なテストは不可能であり、これが入力値としていくつかのまれな境界値を選択した理由です。まれな境界値に対するテストの他に、ランダムな入力に対するテストも有用でしょう：

```
  val randArgs = Seq.fill(100)(scala.util.Random.nextInt)
  test(randArgs)
```

Lerosプロジェクトでテストを実行するには、次のようにします。

```
$ sbt "test:runMain leros.AluTester"
```

次のような成功メッセージが表示されるでしょう：

```
[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 命令のデコード

ALUから、逆方向に作業して命令デコーダを実装します。まず最初に、独自のScalaクラスと共有パッケージを使って命令エンコーダを定義します。 Lerosのハードウェア実装、Lerosのアセンブラ、およびLerosの命令セットシミュレータの間でエンコード定数を共有したいと思います。

```
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
  // ...
```

デコードコンポーネントのために出力のBundleを定義します。後にこのBundleの一部がALUへ供給されます。

```
class DecodeOut extends Bundle {
  val ena = Bool()
  val func = UInt()
  val exit = Bool()
}
```

デコードは8ビットのオペコードを入力として受け取り、デコードされた信号を出力とします。これらの駆動信号は、WireDefaultでデフォルト値が割り当てられます。

```
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
```

デコードは、オペコードに含まれる命令の部分に対する大きなswitch文に過ぎません(Lerosでは、ほとんどの命令は上位8ビットです)。

```
  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 アセンブラ命令

Lerosのプログラムを書くにはアセンブラが必要です。しかし、最初のテストでは、いくつかの命令をハードコーディングしてScalaの配列に入れ、それを命令メモリの初期化に使用します。

```
  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
```

しかし、これはプロセッサをテストするには大変非効率的なアプローチです。 Scalaのように表現力のある言語を用いるのであれば、アセンブラの記述は難しい作業ではありません。そこで、100行程度のコードで記述可能なLeros用の簡単なアセンブラを書きます。アセンブラを呼び出す関数`getProgram`を定義しています。分岐先には、`Map`で収集したシンボルテーブルが必要です。古典的なアセンブラは二回のパスで実行されます。 (1) シンボルテーブル値の収集を行い、(2) 1回目のパスで集めたシンボルを使ってプログラムをアセンブルします。そのため、いずれのパスであるかを示すパラメータを指定して、`assemble`を2回呼び出します。

```
  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)
}
```

assemble関数は、ソースファイル[1]を読み込むことから始まり、2つの可能なオペランドを解析するための2つのヘルパ関数を定義します。 (1) 整数定数(10進数または16進数表記が可能)と、(2) レジスタ番号を読み込みます。

```
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
  }
```

コード 12.3にLerosアセンブラのコア部分を示します。 Scalaのmatch式がアセンブリ関数のコアをカバーしています。

12.4 演習

最後の章にあるこの演習課題は、非常に自由な形で出題されています。あなたは、Chiselの学習ツアーの最後にあたり、あなたが面白いと思える設計課題に取り組む準備ができています。

1つの選択肢は、この章を読み直して、Leros リポジトリにあるすべてのソースコードを読んでテストケースを実行し、それらのテストが失敗するほどコードをいじってみることです。

別の選択肢は、あなた自身のLerosを実装してみることです。リポジトリにある実装は、可能なパイプライン構成の一つの実装に過ぎません。 Chiselシミュレーション版のLerosを1段のパイプラインステージだけで書くこともできますし、最大クロック周波数を得るためにLerosをスーパーパイプライン化することもできます。

[1] この関数は実際にソースファイルを読み込むわけではありませんが、この議論では読み込み関数として考えることができます。

```
    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 "" => // println("Empty line")
        case t: String => throw new Exception("Assembler error: unknown
            instruction: " + t)
        case _ => throw new Exception("Assembler error")
```

コード 12.3: Leros アセンブラのメインの部分

第3の選択肢は、ゼロからプロセッサを設計することです。 Lerosプロセッサならびに必要なツールを構築してみたことで、プロセッサの設計と実装は魔法の芸術ではなく、非常に楽しいエンジニアリングであることがわかっていただけたでしょう。

13 Chiselへの貢献

Chiselは、絶え間ない開発と改良のもとにあるオープンソースプロジェクトです。もちろん、あなたもプロジェクトに貢献することができます。ここでは、Chiselのライブラリ開発のための開発環境のセットアップと、Chiselへの貢献方法について説明します。

13.1 開発環境の設定

Chiselは、いくつかの異なるリポジトリから構成されています。そのすべてはGitHub上のfreechips オーガナイゼーション でホストされています。

あなた個人のGitHubアカウントに、貢献したいリポジトリをフォークします。具体的には、GitHubのWebインターフェースでForkボタンを押すことで、リポジトリをフォークすることができます。そして、そのフォークから、そのフォークしたリポジトリをクローンします。[1] 例えば、chisel3を変更するために、私のローカルフォークにクローンするコマンドは次のとおりです。

```
$ git clone git@github.com:schoeberl/chisel3.git
```

Chisel3をコンパイルし、ローカルライブラリとして公開するには、以下を実行します。

```
$ cd chisel3
$ sbt compile
$ sbt publishLocal
```

ローカルコマンド(sbt publishlocal)を使って、ライブラリを公開したときには、公開されたライブラリにSNAPSHOT文字列が含まれていることに気をつけてください。もしあなたがテスタを用いる際に、公開したバージョンが、ChiselのSNAPSHOTと互換性がない場合、chisel-testerについても同様にフォークとクローンを行い、ローカルで公開してください。

Chiselでの変更をテストするには、おそらく、Chiselプロジェクトを作成する必要があります。例えば、empty Chisel projectをクローン/フォークして名前を変更します。 .gitフォルダは削除しておきます。

ローカルに公開されたバージョンのChiselを参照するようにbuild.sbtを変更します。さらに、この本の執筆時点では、Chiselのソースコードのヘッドは、Scalaの2.12を使用しています。しかし、Scala 2.12はanonymous bundlesの問題を持っています。その回避のために、次のScalaのオプション、"-Xsource:2.11" を追加する必要があります。 build.sbtは次のようになります。

```
scalaVersion := "2.12.6"

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

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

[1] Chiselとfirrtlに変更を加える場合は、firrtlについてもフォークとクローンが必要であることに注意してください

```
)

libraryDependencies +=
  "edu.berkeley.cs" %% "chisel3" % "3.2-SNAPSHOT"
libraryDependencies +=
  "edu.berkeley.cs" %% "chisel-iotesters" % "1.3-SNAPSHOT"
```

Chiselのテストアプリケーションをコンパイルして、それがChiselライブラリのローカル公開バージョンを使っているか、よく確認してください。 (SNAPSHOTバージョンも公開されています。したがって、例えばChiselライブラリとアプリケーションコードのScalaのバージョンが異なっている場合、アプリケーションコードが、ローカルに公開されたライブラリではなく、サーバーからのSNAPSHOTバージョンを選択しているかもしれません。)

some notes at the Chisel repoも参照しておいてください。

13.2 テスト

Chiselライブラリを変更する際は、Chiselのテストも実行する必要があります。 sbtベースのプロジェクトでは、下記のように実行します。

```
$ sbt test
```

Chiselに機能を追加する場合は、その新機能のテストを提供する必要があります。

13.3 プルリクエストで貢献する

Chiselプロジェクトでは、開発者はメインリポジトリに直接コミットできません。貢献は、ライブラリのフォークバージョンの作業ブランチからpull request、プルリクエストで行います。詳細については、GitHub上のドキュメントcollaboration with pull requests、プルリクエストによるコラボレーションを参照してください。 Chiselのグループでも貢献方法についてcontribution guidelines、貢献ガイドラインで文書化を始めています。

13.4 演習

UIntタイプの新しいオペレータ(演算子)を追加します。 Chiselライブラリに組み込み、オペレータの使用例として、いくつかの使用例とテストコードを作成します。とりあえず、便利なオペレータである必要はありませんので、何でもよいです。例えばゼロでなければ左の項を、それ以外の場合は右の項を返す、マルチプレクサのような、「?」オペレータを考えます。そのためには、どれくらいのコード行数が必要でしょうか?[2]

ただし、このような些細なものでも、Chiselプロジェクトをフォークして、(ご自身のための)演習をChiselプロジェクトにプルリクエストはしないでください。 Chiselの変更や拡張は、メインの開発者(達)と相談の上すすめなくてはなりません。この演習は、そうした活動を始めるにあたっての簡単な練習です。

もしあなたが慣れてきたら、 open issues、未解決の課題 のいずれかを選んで、その解決を試みることもできます。そして、プルリクエストをChiselに送り、貢献します。まず最初は、GitHubリポジトリでの Chiselの開発スタイルを見て Chiselのオープンソースプロジェクト内で、その変更とプルリクエストがどのように扱われているかを見てみましょう。

[2]てっとりばやい実装では、2行ほどのScalaコードになります

14 まとめ

この本は、ハードウェア構築言語であるChiselを用いたデジタル回路設計の入門書です。ここまで、Chiselを使ったシンプルなものから中規模なものまでのデジタル回路設計を見てきました。 ChiselはScalaをベースとした言語で、Scalaの強力な抽象化を継承していますが、この本は入門書として意図されているため、それぞれの例はScalaの簡単な使い方の範囲にとどまるようにしました。次の論理的なステップは、Scalaの基本を学び、それをあなたのChiselプロジェクトで実践してゆくことです。

この本のフィードバックをお寄せいただければ、次の改訂版に反映します。私の連絡先は mailto:masca@dtu.dk です。 GitHub上でIssueリクエストを出していただいても構いません。この本の修正や改善に関するPullリクエストもお待ちしております。（日本語版につきましては、Chisel勉強会のSlackまで https://chisel-jp-slackin.herokuapp.com/）

Source Access

この本はオープンソースで提供されています。リポジトリには、Chisel講座のスライドと、全てのサンプルコードも含まれています：https://github.com/schoeberl/chisel-book（日本語版は：https://github.com/chisel-jp/chisel-book の japanese ブランチを参照してください）

この本の中でも参照している、中規模の実装例もオープンソースとして提供されています。 https://github.com/schoeberl/chisel-examples は、こうした様々なFPGAをターゲットとしたプロジェクトを含む実装例です。

A Chiselを使っているプロジェクト一覧

Chiselは（まだ）多くのプロジェクトで使用されているわけではありません。そのため、言語やコーディングスタイルを学ぶためのオープンソースのChiselの実装コードはそれほど多く存在しません。ここでは、私が把握している、オープンソースで公開されている Chiselを使ったプロジェクトを紹介します。

Rocket Chip はロケットマイクロアーキテクチャとTileLinkインターコネクト(バス)生成器を含む RISC-V(英語)/ (日本語) [13]プロセッサシステムの生成器です。もともと最初のチップスケール Chiselプロジェクト [1] としてカリフォルニア大学バークレー校で開発されました。現在、ロケットチップはSiFiveによって商業的にサポートされています。

Sodor は教育目的での利用を意図したRISC-V実装です。 1、2、3、及び5段のパイプラインの実装を含んでいます。すべてのプロセッサは、デバッグポートを介して、命令フェッチ、データアクセス、およびプログラムのロードがシンプルな共有スクラッチパッドメモリを使用します。 Sodorは主にシミュレーションで使用されることを意図しています。

Patmos はリアルタイムシステム [10]向けに最適化されたプロセッサです。 Patmos のリポジトリにはいくつかのマルチコア通信アーキテクチャが含まれます。時間予測可能なメモリアービター [7]、ネットワークオンチップ [9]、オーナーシップ対応の共有スクラッチパッドメモリ [11]、などを含みます。この記事の執筆時点では、Patmos はまだChisel2 で記述されています

FlexPRET は高時間精度アーキテクチャ [14]の実装です。 FlexPRETは、RISC-Vの命令セットを実装し、Chisel3.1 に更新されました。

Lipsi はシステムオンチップでのユーティリティ機能向けの小型プロセッサです [6] 。Lipsiのコードベースは非常に小さく、Chiselのプロセッサ設計の最初の出発点として利用することができます。また、LipsiはChisel/Scalaの生産性の高さの実例でもあります。私が、Chiselでハードウェアを記述し、FPGA上でそれを実行し、Scalaでアセンブラを書き、ScalaでLipsiの命令セットシミュレータを書き、いくつかのテストケースを書くのに14時間ほどかかりました。

OpenSoC Fabric はChiselで記述された、オープンソースの NoC (Network on Chip) 生成器です [5]。大規模な設計探索のためのシステムオンチップを提供することを意図しています。 NoC自体は、ワームホールルーティング、フロー制御のためのクレジット、及び仮想チャネルのための最先端の設計です。 OpenSoCファブリックはまだChisel 2を使用しています

DANA はロケットカスタムコプロセッサインターフェース（ROCC）を使用して、RISC-Vロケットプロセッサと統合したニューラルネットワークアクセラレータです [4]。DANAは推論と学習をサポートしています。

Chiselwatt は POWER Open ISA の実装です。 Micropythonを実行するための命令を含んでいます。

Chiselを使用するオープンソースプロジェクトに心当たりがありましたら、その情報を私に送っていただければ、この本の改訂版に含めたいと思います。

B Chisel 2

この本はChiselのバージョン3に対応しています。また、新規設計用にはChisel3が推奨されています。しかし、世の中には、まだChisel3に変換されていない Chisel2のコードが依然として存在しています。 Chisel2のプロジェクトをChisel3に変換する方法に関するドキュメントとして以下の2つがあります：

- Chisel2 vs. Chisel3
- Towards Chisel 3

しかし、あなたはまだChisel2を使用するプロジェクトに関わるかもしれません。例えば、Patmos [10] プロセッサはChisel2で設計されています。したがって、すでにChisel3で設計を始めている人向けに、Chisel2に関する情報を提供したいと思います。

まず、Chisel2上のすべてのドキュメントは、Chiselに属するウェブサイトから削除されています。私たちはこれらのPDF文書を救出して、GitHub https://github.com/schoeberl/chisel2-doc に保存しています。 Chisel2のブランチに切り替えることで、Chisel2のチュートリアルにアクセスできます。

```
$ git clone https://github.com/ucb-bar/chisel-tutorial.git
$ cd chisel-tutorial
$ git checkout chisel2
```

Chisel3と2の目に見える大きな違いは、定数の定義、IO、ワイヤ、メモリのバンドル、および古いレジスタ定義方法になります。

Chisel2の実装は、Chisel3の代わりに、Chisel パッケージを使うことで、この互換性レイヤを介して、Chisel3プロジェクトでもある程度利用することができます。しかし、この互換性レイヤの使用は、あくまで遷移期間に留めるべきであり、ここでは詳しくは述べません。

ここで紹介する基本的なコンポーネントの2つの例は、Chisel3 と同じものです。組み合わせロジックを含むモジュール例：

```
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
}
```

IO定義のBundleがIO()クラスにラップされて いない ことに注意してください。さらに、それぞれのIOポートはタイプ定義の一部として定義されます。この INPUTとOUTPUT例で

はUIntの一部として定義されています。また、信号の幅は2番目のパラメータで与えられています。

Chisel2 での8ビットレジスタの記述例：

```
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
}
```

ここでは、initという名前のパラメータにUIntを介して渡されたリセット値を用いる典型的なレジスタの定義方法を紹介しています。この形式は、Chisel3でもまだ有効ですが、新たなChisel3設計用には、RegInitとRegNextの使用が推奨されています。またここでは、UInt(0, 8)で8ビット幅の定数0を定義しています。

Chiselベースのテスト用C++コードとVerilogコードは、chiselMainTest と chiselMain を呼び出すことで生成されます。どちらのメイン関数もパラメータとして文字列(String) の配列を取ります。

```
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()))
  }
}
```

読み書きが一旦レジスタにラッチされるメモリは、Chisel2では以下のように記述されています。

```
  val mem = Mem(UInt(width = 8), 256, seqRead = true)
  val rdData = mem(Reg(next = rdAddr))
  when(wrEna) {
    mem(wrAddr) := wrData
  }
```

C 略語

ハードウェア設計者やコンピュータのエンジニアは略語を好みます。しかしながら、そうした略語になれるには時間がかかります。以下に、デジタル回路設計やコンピュータアーキテクチャで一般的に使われる略語を列挙します。

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-in-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

参考文献

[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.

索引