

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

Copyright © 2016–2019 Martin Schoeberl



This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. <http://creativecommons.org/licenses/by-sa/4.0/>

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.

Contents

まえがき (L274 mune10 初回校正済)	ix
序文 (L394 mune10 初回校正済)	xi
1 はじめに (L558 mune10 初回校正済)	1
1.1 ChiselとFPGA開発ツールのインストール	1
1.1.1 macOS	2
1.1.2 Linux/Ubuntu	2
1.1.3 Windows	2
1.1.4 FPGA ツール	2
1.2 Hello World	2
1.3 Chisel で Hello World	3
1.4 Chisel 用のIDE	3
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	4
1.6 参考文献	4
1.7 演習	5
2 基本コンポーネント (L1603 diningyo 初回校正済)	7
2.1 信号タイプと定数	7
2.2 組み合わせ回路	8
2.2.1 マルチプレクサ	9
2.3 レジスター	10
2.3.1 カウント	11
2.4 バンドルとVecを用いたストラクチャ	11
2.5 Chisel によるハードウェア生成	12
2.6 演習	13
3 ビルドプロセスとテスト (L2997 mune10 初回校正済)	15
3.1 sbt でプロジェクトを構築する	15
3.1.1 ソースコードの構成	15
3.1.2 sbt の実行	16
3.1.3 ツールの実行フロー (L3373)	17
3.2 Chisel をつかったテスト (L3469)	18
3.2.1 PeekPokeTester	18
3.2.2 ScalaTest の利用	20
3.2.3 波形表示	20
3.2.4 printf デバッグ	22
3.3 演習	22
3.3.1 最小のプロジェクト	22
3.3.2 テストの演習	24
4 コンポーネント (L4455 mune10 初回校正済)	25
4.1 Chisel のコンポーネントはモジュール	25
4.2 算術論理ユニット	27
4.3 バルク接続	28
4.4 関数(Function)による軽量コンポーネント	29
5 組合せ回路ブロック (L5130 mune10/diningyo 初回校正済)	31
5.1 組合せ回路	31
5.2 デコーダー	32

5.3 エンコーダー	34
5.4 演習	34
6 シーケンシャル回路ブロック (L5709 TODO)	35
6.1 レジスター (L5743 TODO)	35
6.2 カウンター (L6109 TODO)	37
6.2.1 カウントアップとダウン (L6268 TODO)	38
6.2.2 カウンタによるタイミングの生成 (L6371 TODO)	39
6.2.3 「オタク」カウンター (L6538 TODO)	40
6.2.4 タイマー (L6610 TODO)	41
6.2.5 パルス幅変調(PWM) (L6721 TODO)	41
6.3 シフトレジスタ (L6919 TODO)	43
6.3.1 パラレル出力付きシフトレジスタ (L6992 TODO)	43
6.3.2 パラレルロード付きシフトレジスタ (L7067 TODO)	43
6.4 メモリー (L7128 TODO)	44
6.5 演習 (L6986 7332)	47
7 入力処理 (L7452 TODO)	49
7.1 非同期入力 (L7502 TODO)	49
7.2 デバウンス (L7640 TODO)	49
7.3 入力信号のフィルタリング (L7820 TODO)	51
7.4 入力処理と関数の組み合わせ (L7936 TODO)	52
7.5 演習 (L7981 DONE)	52
8 有限状態機械 (FSM) (L8048 TODO)	55
8.1 有限状態機械の基本 (L8124 TODO)	55
8.2 ミーリー FSMで出力を高速化 (L78454 TODO)	58
8.3 ムーア対ミーリー (L78692 TODO)	59
8.4 演習 (L8843 TODO)	62
9 コミュニケートステートマシン (L8903 TODO)	63
9.1 ライトフラッシャーの例 (L8944 TODO)	63
9.2 データパスを持つステートマシン (L9194 TODO)	64
9.2.1 ポップカウントの例 (L9242 TODO)	67
9.3 Ready-Valid インターフェース (L9462 TODO)	69
10 ハードウェアジェネレータ (L9245 TODO)	73
10.1 パラメータを使って設定する (L8932 TODO)	73
10.1.1 シンプルなパラメータ (L9816 TODO)	73
10.1.2 型パラメータを持つ関数 (L9865 TODO)	73
10.1.3 タイプパラメータを持つモジュール (10036 TODO)	74
10.1.4 パラメータ化されたバンドル (L9225 TODO)	75
10.2 組合せ論理回路の生成 (L10203 TODO)	76
10.3 繙承を利用する (L10372 TODO)	77
10.4 関数型プログラミングによるハードウェア生成 (L19523 TODO)	78
11 デザイン例 (L10642 TODO)	81
11.1 FIFO バッファ (L10098 TODO)	81
11.2 シリアルポート (L10368 TODO)	82
11.3 FIFO設計のバリエーション (L11322 TODO)	85
11.3.1 FIFOのパラメータ化 (L1135 TODO)	85
11.3.2 バブルFIFOの再設計 (L11471 TODO)	86
11.3.3 ダブルバッファFIFO (L11554 TODO)	87
11.3.4 レジスタメモリ付きFIFO (L11646 TODO)	88
11.3.5 オンチップメモリ付きFIFO (L11872 TODO)	89
11.4 演習 (L11980 TODO)	91
11.4.1 バブルFIFOを探る (12004 TODO)	92
11.4.2 UART (L12162 TODO)	92

11.4.3 FIFO探査 (L12296 TODO)	93
12 プロセッサの設計 (L12361 TODO)	95
12.1 ALUから始める (L12464 TODO)	95
12.2 命令のデコード (L12644 TODO)	98
12.3 アセンブラー命令 (L12729 TODO)	99
12.4 演習 (L12833 TODO)	101
13 Chisel への貢献 (L12905 mune10 初回校正済)	103
13.1 開発環境の設定	103
13.2 テスト	104
13.3 プルリクエストで貢献する	104
13.4 演習	104
14 まとめ (L12556 mune10 初回校正済)	105
A Chiselを使っているプロジェクト一覧 (L12650 mune20 初回校正済)	107
B Chisel 2 (L12866 mune10 初回校正済)	109
C 略語 (L13080 mune10 初回校正済)	111
Bibliography	113
Index	115

List of Figures

2.1	(a & b) cの論理.信号は单一、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。	8
2.2	基本的な2:1マルチプレクサ。	10
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	10
3.1	Chiselプロジェクトのソースツリー(sbt利用)	15
3.2	Chiselエコシステムのツールフロー	17
4.1	A design consisting of a hierarchy of components.	25
4.2	An arithmetic logic unit, or ALU for short.	27
5.1	A chain of multiplexers.	32
5.2	A 2-bit to 4-bit decoder.	33
5.3	A 4-bit to 2-bit encoder.	34
6.1	A D flip-flop based register.	35
6.2	A D flip-flop based register with a synchronous reset.	36
6.3	A waveform diagram for a register with a reset.	36
6.4	A D flip-flop based register with an enable signal.	37
6.5	A waveform diagram for a register with an enable signal.	37
6.6	An adder and a register result in counter.	38
6.7	Counting events.	38
6.8	A waveform diagram for the generation of a slow frequency tick.	39
6.9	Using the slow frequency tick.	40
6.10	A one-shot timer.	41
6.11	Pulse-width modulation.	42
6.12	A 4 stage shift register.	43
6.13	A 4-bit shift register with parallel output.	44
6.14	A 4-bit shift register with parallel load.	44
6.15	A synchronous memory.	45
6.16	A synchronous memory with forwarding for a defined read-during-write behavior.	46
7.1	Input synchronizer.	49
7.2	Debouncing an input signal.	50
7.3	Majority voting on the sampled input signal.	51
8.1	A finite state machine (Moore type).	55
8.2	The state diagram of an alarm FSM.	57
8.3	A rising edge detector (Mealy type FSM).	58
8.4	A Mealy type finite state machine.	58
8.5	The state diagram of the rising edge detector as Mealy FSM.	59
8.6	The state diagram of the rising edge detector as Moore FSM.	59
8.7	Mealy and a Moore FSM waveform for rising edge detection.	60
9.1	The light flasher split into a Master FSM and a Timer FSM.	63
9.2	The light flasher split into a Master FSM, a Timer FSM, and a Counter FSM.	64
9.3	A state machine with a datapath.	67
9.4	State diagram for the popcorn FSM.	67
9.5	Datapath for the popcorn circuit.	68
9.6	The ready-valid flow control.	70

9.7	Data transfer with a ready-valid interface, early ready	71
9.8	Data transfer with a ready-valid interface, late ready	71
9.9	Single cycle ready/valid and back-to-back trasnfers	71
11.1	A writer, a FIFO buffer, and a reader.	81
11.2	One byte transmitted by a UART.	83

List of Tables

2.1	Chiselで定義されているハードウェアの演算子	9
2.2	vに適用できるハードウェアのメソッド	9
5.1	Truth table for a 2 to 4 decoder.	33
5.2	Truth table for a 4 to 2 encoder.	34
8.1	State table for the alarm FSM.	56
12.1	Leros instruction set.	96

Listings

1.1 A hardware Hello World in Chisel	3
6.1 A one-shot timer	41
6.2 1 KiB of synchronous memory.	45
6.3 A memory with a forwarding circuit.	47
7.1 Summarizing input processing with functions.	52
8.1 The Chisel code for the alarm FSM.	56
8.2 Rising edge detection with a Mealy FSM.	60
8.3 Rising edge detection with a Moore FSM.	61
9.1 Master FSM of the light flasher.	65
9.2 Master FSM of the double refactored light flasher.	66
9.3 The top level of the popcount circuit.	68
9.4 Datapath of the popcount circuit.	69
9.5 The FSM of the popcount circuit.	70
10.1 Reading a text file to generate a logic table.	76
10.2 Binary to binary-coded decimal conversion.	77
10.3 Tick generation with a counter.	77
10.4 A tester for different versions of the ticker.	78
10.5 Tick generation with a down counter.	79
10.6 Tick generation by counting down to -1.	79
10.7 ScalaTest specifications for the ticker tests.	79
11.1 A single stage of the bubble FIFO.	82
11.2 A FIFO is composed of an array of FIFO bubble stages.	83
11.3 A transmitter for a serial port.	84
11.4 A bubble FIFO with a ready-valid interface.	86
11.5 A FIFO with double buffer elements.	87
11.6 A FIFO with a register based memory.	88
11.7 A FIFO with a on-chip memory.	89
11.8 Combining a memory based FIFO with double-buffer stage.	91
12.1 The Leros ALU.	97
12.2 The Leros ALU function written in Scala.	98
12.3 The main part of the Leros assembler.	101

まえがき (L274 mune10 初回校正済)

デジタルデザイン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。半導体製造に関わる企業は、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、デジタルデザインの生産性向上により、このコストの削減します。設計の再利用による検証のコストの削減や、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デザイン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のようなスタンダー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。過去数十年間、ソフトウェアの世界はその設計手法の様々な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とも、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ます。

Chiselに対して、「学ぶことが困難である」と長年の批判されてきました。このような認識の多くは、自分の研究や、商業的なニーズを満足させるために、専門家によって作成された大規模で複雑な設計が普及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。むしろ、新たにChiselを使う人に向けた、様々な研修コースや、教科書、様々な学習教材が必要です。Chiselを学びたい人のための重要なリソースとして、この*Digital Design with Chisel*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、それは本書の内容からもみてとれます。インストールおよびプリミティブの解説から始めて、レンガ造り建物を、レンガを一つ一つ積み上げるように、読者の理解を深めてゆきます。付属の演習例題は、読者の理解を強固にするための接着剤です。心の中の各概念セットすることを保証し、固化が理解していることモルタルです。この本は、ハードウェアジェネレーターで頂点に達し、この屋根が残りの部分に目的を与えます。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*Digital Design with Chisel*で生産的なデジタル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・ケニッヒ
ChiselとFIRRTLメンテナ
スタッフエンジニア、SiFive社

序文 (L394 mune10 初回校正済)

この本は、ハードウェア構築言語であるChiselを使ったデジタル・デザイン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の先進のソフトウェアエンジニアリングの技術をデジタル・デザイン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、現代的な言語をASICやFPGA設計に活用することができます。 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、小さな一般的なハードウェアから中規模のハードウェアを例に、Chiselを使ったデジタル・デザインを紹介していきます。

第2版のまえがき (L452)

Chiselは、アジャイルなハードウェア設計を可能にしますし、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。 本の初版のリリース後、半年未満で、改善と拡張し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。 テストセクションを拡張しています。 シーケンシャルビルディングブロック(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、そしてノイズの多い入力信号をどのようにフィルタリングについて説明します。 デザインエグザンプルの章も拡張され、異なる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デザインのなかでどのように使うかを解説しています。

謝辞 (L505)

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人々に感謝します。 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ーで、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、先進コンピューター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。 裸から抜け出し、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 (L550)

この日本語訳は Chisel勉強会で行っています。 翻訳で用いたオリジナルのバーションは (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への登録URL <https://chisel-jp-slackin.herokuapp.com/> していただければ情報交換できると思います。

1 はじめに (L558 mune10 初回校正済)

この本は、近代的なハードウェア構成言語である Chisel [2] を使ったデジタルシステム設計を紹介します。本書では、一般的なデジタル・デザインの書籍に比べ、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および(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、Python、Java、またはTclのような他の言語も使いこなして開発をしているハードウェア設計者は、ハードウェアの生成が言語の機能の一部となっている、一つのハードウェア構築言語を使った開発に移行することができます。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。

現在一般的にハードウェアの設計は、ハードウェア記述言語 (HDL) を使って記述します。CADツールなどを使用して、ハードウェアコンポーネントをお絵かきする時代は終わりました。一部の（抽象度の）高レベルの回路構成は作りますが、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やVHDLの2つです。どちらの言語も、古く、様々な遺産を含んでいます。これらの言語を使って記述した回路は実際のハードウェアに合成可能ですが（ここ意訳、要チェック）。誤解しないでください：VHDLやVerilogは ASIC に合成可能なハードウェアブロックを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。

この本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の紹介については、他のデジタル設計の本を参照して下さい。しかしこの本はFPGAをターゲットとした設計やASICを記述するための現在の手法といった抽象レベルのデジタル設計について教えようとしています。¹ この本のための予備知識として、ブル代数や2進法のシステム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は最初のデジタルハードウェア設計になります。例題中のビルト処理は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イス (CLI、ターミナルやUnixシェルとも) 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りますし、数日で習得することができます。したがって、この本もそんなに大きな本ではありません。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があります。ChiselはScalaの「あなたを育てる (TODO いまいち)」[11]から機能を継承しています。しかしながら、Scala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、オダスカイのテキストブック [11]を参照してください。この本は、デジタル設計とChisel言語のチュートリアルです。Chisel言語リファレンスではありませんし、完全なチップ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きます。

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1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

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ます。

ンスを変更したため、[AdoptOpenJDK](#)からOpenJDKをインストールするのが簡単でしょう。

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、パケットマネージャ (TODO : パッケージマネージャか??) 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
```

[GTKWaveIntelliJ](#) (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ファイルからインストールできます。しかしこの本の執筆時点では、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。[GTKWaveIntelliJ](#) (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、Windowsインストーラでsbtをインストールする前にインストール済みの select the JDK 1.8 を選択してください (Java 11ではありません!)。関連項目:[Windowsでのsbtのインストール](#)。[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ベンダーであるIntel²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。これらの中規模の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もHello Worldと呼ばれる最小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。

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
}

```

Listing 1.1: A hardware Hello World in Chisel

\$ sbt run

Hello Worldのプログラムの期待される出力は：

```
[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？それはLEDを点滅させることがハードウェア（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。

lst:chisel:hello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を解説するので、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5000000-1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、あなたの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えば、すべてのコマンドは、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、ここでは、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）を利用してすることで、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

[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、デジタルデザイン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を含む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](#) はChiselの簡単なユーザー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、 Jupyter ノートブックとして、 ハードウェア・ジェネレーターを中心とした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）

- [Chiselを始めたい人に読んでほしい本](#) だいにんぎょーは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。

³sbt用のEclipseプラグインが必要です。

⁴日本語版は [chisel-book 日本語訳](#)

1.7 演習

各章はハンズオン演習で終わります。導入となる演習では、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。の点滅。LEDの点滅コードをコンパイルするために、次の手順を実行します。

```
$ 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ファイルをChiselのモジュールを比較するとclockとresetが含まれていないことに気づくでしょう。これらのシグナルは暗黙的に生成され、ほとんどの設計ではこれらの低レベルな信号を扱う必要がない方が便利です。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、Verilogコードをコンパイルし、得られたビットファイルでFPGAを設定します。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)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。

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は毎秒200 msの間点灯しなければなりません。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の状態が必要です。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？以上を知らせるのか？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ーが実行されます。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。

⁵FPGA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っています。ただし、この導入例では、単にコードを「コンパイル」します。

2 基本コンポーネント (L1603 diningyo 初回校正済)

ここでは、デジタル設計のための基本的な部品を紹介します。組み合わせ回路とフリップフロップです。これらの重要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。

一般的に構築された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つの方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2の補数で表現します。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)
```

定数はスカラの整数を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という概念を見つけた場合、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。この概念は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32bitの1を表す定義ではありません。その代わりに(32)は3 2bit目のビットの指定として解釈されるため、結果的に1bit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の型推論のおかげで、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に比べて完結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。次の例では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測しています。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
```

¹現在のChiselにおいてBits型は演算処理が存在しておらず、それゆえにうユーザーにとってはあまり有用ではありません。

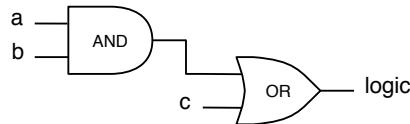


Figure 2.1: $(a \& b) | c$ の論理信号は单一、もしくは複数のビットになり得る。Chiselの表現と回路図は同じになる。

```
"b1111_1111".U // binary representation of 255
```

上記のコードは表現する定数をアンダースコアを使って桁をグループ化する方法も示しています。アンダースコアは定数値には影響しません。

Chiselでは論理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を表現できます。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ブール代数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单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、および除算とモジュロ演算では分子のビット幅になります。²

信号はあるChiselの型のWireとして定義することもできます。その後、:=update演算子を使用してWireに値を割り当てることができます。

```
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。

² 詳細はFIRRTL仕様に記載されています。

演算子	処理	データの型
* / %	乗算、除算、モジュロ	UIInt, SInt
+ -	加算、減算	UIInt, SInt
== != /=	等しい、等しくない	UIInt, SInt, returns Bool
> >= < <=	比較	UIInt, 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)	UIInt, SInt
~	ビット反転	UIInt, SInt, Bool
& ^	ビット論理積、ビット論理和、ビット排他的論理和	UIInt, SInt, Bool
!	否定	Bool
&&	論理積、論理和	Bool

Table 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	UIInt, SInt, returns Bool
v(n)	特定の1bitの選択	UIInt, SInt
v(end, start)	連続ビットの選択	UIInt, SInt
Fill(n, v)	n回ビット列を繰り返し	UIInt, SInt
Cat(a, b, ...)	ビット列の連結	UIInt, SInt

Table 2.2: vに適用できるハードウェアのメソッド

```
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

マルチプレクサは、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。

マルチプレクサはロジックから構築できます。しかし多重化は標準的な操作なので、Chiselはマルチプレクサを提供しています。

```
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

³Chisel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。

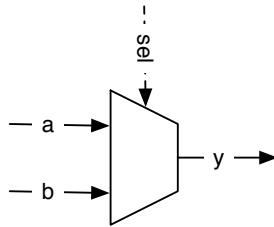


Figure 2.2: 基本的な 2:1 マルチプレクサ.

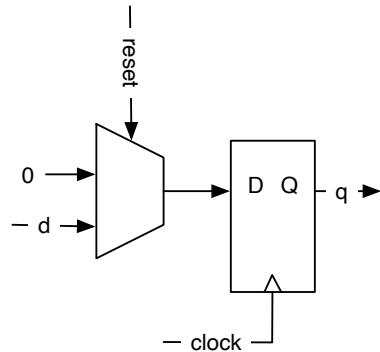


Figure 2.3: 同期リセットで0初期化されるDフリップ・フロップベースのレジスタ

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 レジスター

ChiselにはD flip-flops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黒バ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体として表現できる、任意のChiselの型として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、入力を接続する場合でも、初期値として定数を接続できます。

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

組み合わせ論理とレジスタを区別するための一般的な方法として、レジスタ名の後に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、キャメル・ケース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、値をカウントアップしていきます。しかし、コンピューター・サイエンスとデジタル設計では、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を用いたストラクチャ

Chiselは関連した信号をまとめたための構造を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は必要なら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。

⁴ ScalaにはArrayという名前のデータ型が存在しています。

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

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

Wireで包まれたVecはマルチプレクサになります。またレジスタをVecで包むと、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32bit 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のような、古典的なRISC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おこない、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))
}
```

リセットが必要なBundle型のレジスタが使う場合は、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、と思われたかもせれません。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胃しています。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。

より技術的な面においては、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コードやChiselのテスターによってテストされるハードウェアです。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が必要ない、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ーにコピーして、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、そのデザインから点滅する論理を削除し、1つのスイッチのを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つのスイッチと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。その次のステップは3つの入力からなるマルチプレクサを考えましょう。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。

ここまでで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のコンフィグ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。

3 ビルドプロセスとテスト (L2997 mune10 初回校正済)

もっと面白い Chisel コード記述を始める前に、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 のテスターは、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 で latest.release で設定することで、常に最新バージョンの Chisel を使用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、インターネット接続が検索のために必要になります。Chisel やその他の Scala ライブラリは、専用のバージョンを build.sbt で指定した方が良いでしょう。**であれば**、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

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 に別れます。Chisel は Scala を継承しており、Scala は Java からソースの packages 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

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>.

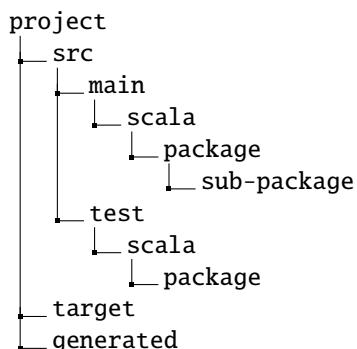


Figure 3.1: Chisel プロジェクトのソースツリー (sbt 利用)

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、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。この例ではmypacket

```
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パケット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（パケット名前空間）でAbcモジュールを使用するには、パケット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、パケット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メソッドを含むobjectを検索したり、単純にAppを拡張したりします。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"
```

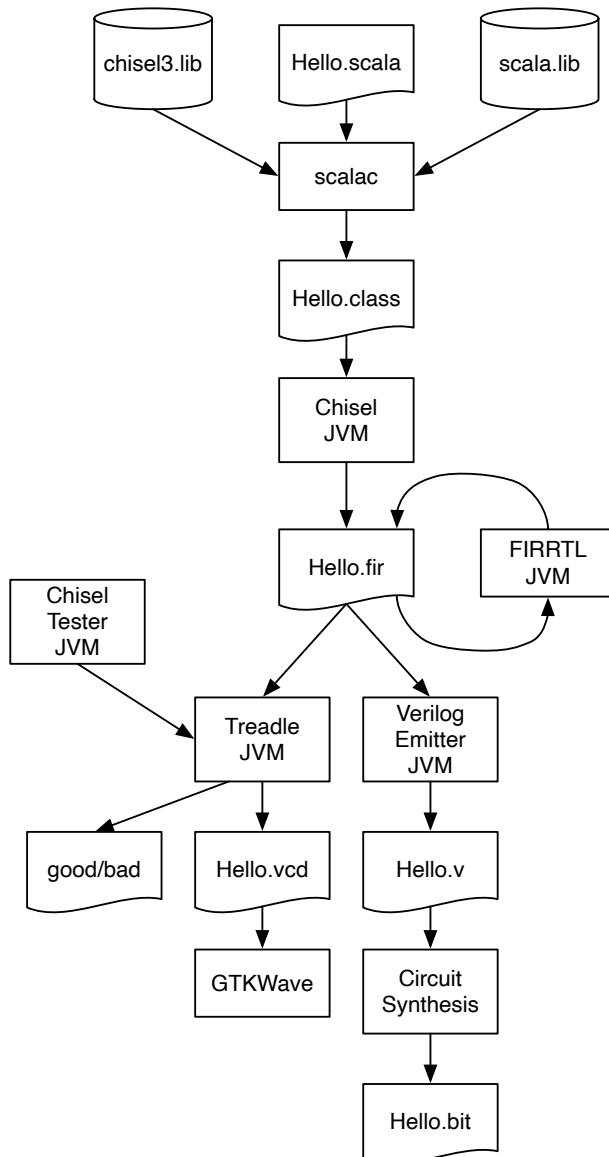


Figure 3.2: Chiselエコシステムのツールフロー

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 のテストは、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、テストツリー内の `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 (L3373)

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) 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

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ています。

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。

FIRRTLインタプリタが回路をシミュレートするエンジンです。Chiselテスターと一緒に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。このエンジンは波形ファイル (Hello.vcd) を生成することができます。生成された波形は波形ビューアで表示かのうです(無料のビューアであるGTKWaveまたはModelSimなど)³

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 (L3469)

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の強みに一つは、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、最低次の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関数を含むテストオブジェクト(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ーをDUTに持ります。

```
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)
```

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。

```

    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。テスターは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。

以下のテスターメイン(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できます。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、ハードウェアから値が表示されることはありませんが、すべての期待値が正しく、結果としてすべてのテストが合格したことになります。

```

[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る場合、テストに失敗すると、期待値と実際の値の差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、期待値の4を変更しました。



```

[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テスター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）。

```
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、（代わりに実行されている`main`の）簡単な`sbt test`ですべてのテストを実行できるようにすることです。 次のようにあなたは、`sbt`で1つだけのテストを実行できます。

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

3.2.3 波形表示

以上で紹介したテスターは、ソフトウェアの開発と同様に、小さなデザインや、unit testing、ユニットテストに対してうまく働きます。一連のユニットテストはregression testing、回帰テストにも役立ちます。

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・デザイン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 テスターは、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`の波形テスターを示します。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ーから始めます。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テスターが生成した.vcdファイルを含むフォルダーを探します。標準では、生成されたファイルは、`test_run_dir`にテスター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。

すべての可能な入力値を明示的に列挙することはスケールしません。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。以下のテスターは、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)
    }
  }
}

```

この新しいテスターのために 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)
  }
}

```

以下で実行します。

```
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 debugging”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ー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LEDGER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、唯一のハードウェアのソースファイルです。これには点滅LEDのハードウェア記述（class `Hello`）と、[Verilogコードを生成の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._
```

リストの 1.1 にあるハードウェア記述から始めます。 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。 `extends App` がアプリケーションが起動時に暗黙的に `main` 関数を生成する Scala のオブジェクトです。このアプリケーションの唯一のしごとは、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())
}
```

下記の実行で、手動でサンプル生成をします。

```
$ 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いではありません。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）はしない事です。 `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、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` と一緒に、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。メインの設定は（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、 `hello.qsf` で定義されています。ファイル見ると、どの信号がどのピンに割り当てるかがわかります。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、 FPGA を構成します。

`Hello World` が最低限な 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、 テスターが含まれます。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

最後の章の演習では、ANDゲートとマルチプレクサを構築し、FPGAでこのハードウェアを実行するために、いくつかの入力とLEDの点滅の例を高めています。私たちは今、FPGAに依存しないようにも、この例を使用してテストを自動化するためのChiselテスターで機能をテストします。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テスターを追加します。すべての可能な入力を列挙し、`except()`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。そこでは、デザインのサイズ（通常はLUTとフリップフロップ）と最大クロック周波数でのデザインのパフォーマンスを、実動作で確認できます。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 (L4455 mune10 初回校正済)

大規模なデジタル設計は、多くの場合、階層的な方法でコンポーネントのセットに構造化されています。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イスがあります。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。コンポーネント自体は、2つのサブコンポーネントから組み立てられています。BとCは、Cの入力と出力に接続されています。Aの1つの出力はBの入力に接続されています。コンポーネントDは、コンポーネントCと同じ階層レベルにあり、C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、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。信号も方向は、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：

```
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
}
```

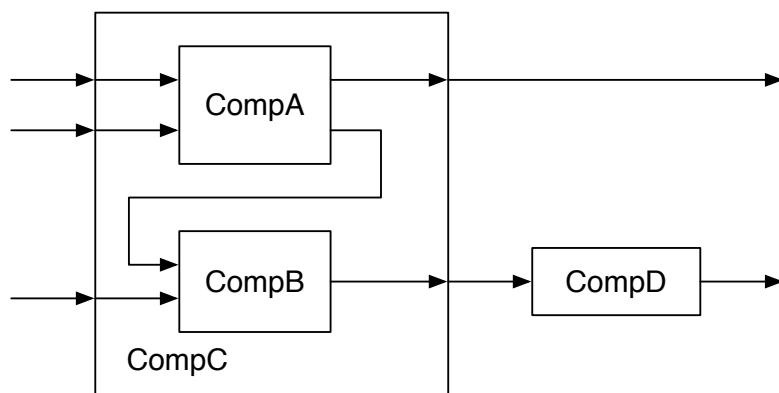


Figure 4.1: A design consisting of a hierarchy of components.

```

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
}

```

コンポーネントAは、**a**と**b**という2つの入力と、**x**と**y**2つの出力を持ちます。コンポーネントB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の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書かれます。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 valid**, や **ready** のような意味のあるポート名を使用します。

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。コンポーネントAとBを元に構成されています。ここでは、AとBがCのポートにどのように接続されているか、また、Aの出力ポートとB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**で生成され、例えば、**new CompA()**、**Module()**への呼び出しにラップされる必要があります。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。

この参照により、モジュールの**io**フィールドを間接参照 (dereferencing) することで、IOポートと IO Bundleの個々のフィールドにアクセスすることができます。

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

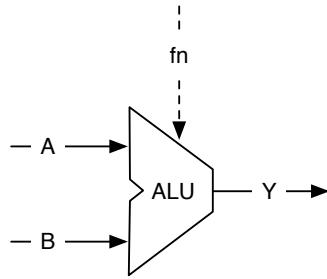


Figure 4.2: An arithmetic logic unit, or ALU for short.

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
}
```

優れたコンポーネント設計は、ソフトウェア設計における機能や手法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、デジタルデザインの本では、原理を示すために小さな部品を使っていることです。例題のサイズは、（そうした本でも、本書でも）ページに収まるように小さくしてあります。また、気が散らないように、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イスは、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イ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、ハードウェアを返す関数として、それらをより軽量に表現する方法を提供しています。

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 \(ALU\)](#) 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 Y の出力を持ちます。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のコンストラクトである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、バンドルの一部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い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(Function)による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することができます。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・ジェネレーターであることに注意してください。エラボレーション過程ででは、加算演算を実行するかわりに、2つの加算器(ハードウェアインスタンス)を作成します。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、関数には、軽量なハードウェア生成器として、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))
```

`RegNext()`既に遅延するためのレジスタを作成するという機能であるとして再度、このことは、有用であることが短すぎる例です。

関数は、`Module`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 (L5130 mune10/diningyo 初回校正済)

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方程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方程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
```

このような表現は固定と考えられています。valを使ったeへの、=による再代入はScalaではエラーになります。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
```

これは実行時に”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、例外が発生します。

Chiselでは特定の条件下で、組み合わせ回路を更新する記述もサポートされています。このような回路はWireとして宣言されています。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、conde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。私達は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。

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、デフォルト値の設定を無効にできます。

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

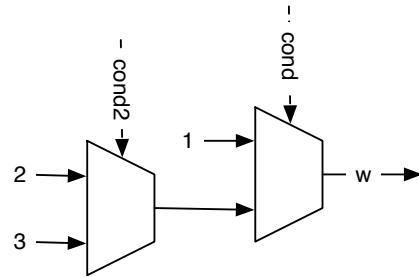


Figure 5.1: A chain of multiplexers.

Chiselは連続した条件分岐（if/elseif/else）である`.elsewhen`を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、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、条件分岐の条件が单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ーで紹介します。

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ため場合には、`WireDefault`を使うことができます。

```
val w = WireDefault(0.U)
when (cond) {
    w := 3.U
}
// ... and some more complex conditional assignments
```

質問として考えられそうな事の1つに「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？」ということあります。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 デコーダー

`decoder`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ーを紹介するためのものです。

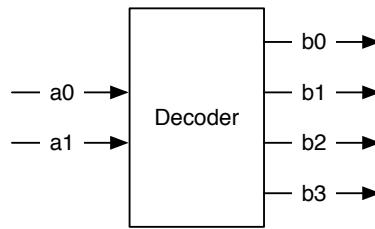


Figure 5.2: A 2-bit to 4-bit decoder.

a	b
00	0001
01	0010
10	0100
11	1000

Table 5.1: Truth table for a 2 to 4 decoder.

```

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}
}

```

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。

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

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づきます。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、ANDゲートと共にマルチプレクサへのデータ入力のため使用することで、マルチプレクサを構成するブロックの1つとして使用されます。しかし、Chiselのコアライブラリには、Muxというマルチプレクサを実装したもの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、その出力を例えれば、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。

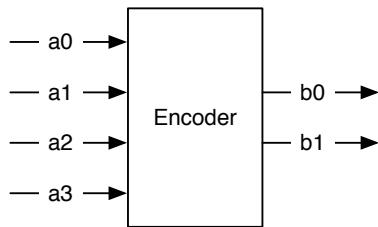


Figure 5.3: A 4-bit to 2-bit encoder.

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

Table 5.2: Truth table for a 4 to 2 encoder.

5.3 エンコーダー

エンコーダ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しており、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、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

4bitのバイナリ入力を7-segment display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、出力を7セグメント・ディスプレイに接続しましょう。

6 シーケンシャル回路ブロック (L5709 TODO)

シーケンシャル回路は、出力が入力 *and* 前の値に依存する回路です。我々は同期設計（クロックデザイン）に興味を持っているとして、我々は、順序回路について話すとき、私たちは、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、我々は状態を格納できる要素が必要です。

6.1 レジスター (L5743 TODO)

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 のコレクションです。D フリップフロップキャプチャクロックとその出力で記憶するの立ち上がりエッジで、入力の値。あるいは、言い換えれば、レジスタは、クロックの立ち上がりエッジで入力の値とその出力を更新します。

図～6.1は、レジスタの回路図シンボルを示しています。これは、入力と出力の D Q が含まれています。各レジスタはまた clock 信号の入力が含まれています。このグローバルクロック信号が同期回路内のすべてのレジスタに接続されているとして、それは通常、私たちの回路図で描かれていません。ボックスの下部にある小さな三角形は、クロック入力を象徴し、これはレジスタであることを教えてくれる。我々は、次の回路図でクロック信号を省略します。グローバルクロック信号の省略は、レジスタのクロック入力に信号の明示的な接続が必要とされない Chisel によって反射されます。

Chisel に入力 d と出力 q でレジスタを使用して定義されています。

```
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、ベクトルと束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 2 段階で定義して使用することができます。

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

```
delayReg := delayIn
```

まず、我々は、レジスタを定義し、名前を付けます。第二に、我々は、レジスタの入力に信号 delayIn 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、名前の一部としてマーカー Reg を持っているのが一般的です。また、（ノミでもそのためと）スカラ座での名前は CamelCase に通常あることに注意してください。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 信号として、Chisel で暗黙的です。私たちは、電子をリセット値を供給しています。グラム。、レジスタコンストラクタ RegInit へのパラメータとして、ゼロ。レジスタの入力は、Chisel 代入文が接続されています。

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

¹ 我々はまた、非同期ロジックやフィードバックを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。いわゆるレジスタを：

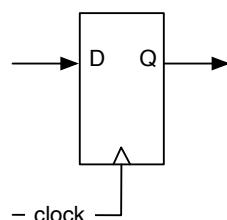


Figure 6.1: A D flip-flop based register.

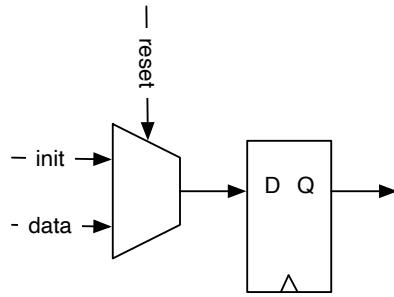


Figure 6.2: A D flip-flop based register with a synchronous reset.

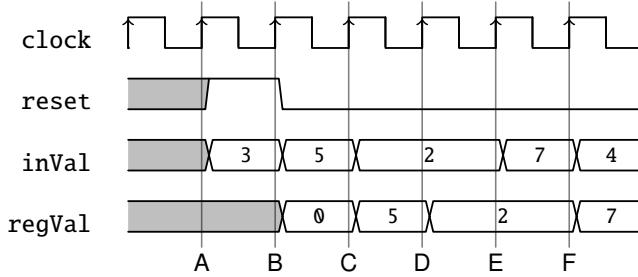


Figure 6.3: A waveform diagram for a register with a reset.

```
valReg := inVal
```

Chiselでリセットの既定の実装は同期リセットです。² 変化がDフリップフロップに必要とされない同期リセットのために、単にマルチプレクサは現在のFPGAの同期リセット入力を含むフリップフロップ追加が必要があります。そのため、追加のリソースがマルチプレクサのために必要とされていません。リセット下の初期値及びデータ値の間で選択を入力します。

図～6.2は、同期リセットトリセットドライブマルチプレクサとレジスタの回路図を示しています。しかし、同期リセットとして、現代のFPGAは、マルチプレクサのためのLUTのリソースを無駄にしないために同期リセット（セット）入力が含まれているフリップフロップかなり頻繁に使用されています。

順序回路は、時間の経過とともにその値を変更します。したがって、彼らの行動は、時間の経過と共に信号を示す図によって説明することができます。そのような図は、波形又はtiming diagramと呼ばれています。

図～6.3はリセットでレジスタのための波形を示し、いくつかの入力データは、それに適用されます。時間は左から右へと移行します。図の上部には、私たちは私たちの回路を駆動するクロックを参照してください。最初のクロックサイクルでは、リセットする前に、レジスタの内容が定義されていません。第2クロックサイクルリセットにハイにアサートされ、このクロック・サイクル（標識B）の立ち上がりエッジでレジスタが初期値0をとります。入力inValは無視されます。次のクロックサイクルでresetは0あり、inValの値は次の立ち上がりエッジ（標識C）上に捕捉されています。その時からresetにそれがあるべきよう、0を今まで、レジスタ出力は、1クロックサイクル遅れて入力信号に従います。

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、タイミングダイアグラムが便利です。また、Chiselテスターでは、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もできます。

典型的なデザインパターンは、イネーブル信号とレジスタです。true（高）、レジスタキャプチャ入力する場合にのみイネーブル信号です。それ以外の場合は、その古い値を保持します。レジスタの入力においてマルチプレクサと、同期リセットと同様、実現することができる可能にします。マルチプレクサへの一方の入力には、レジスタの出力のフィードバックです。

図～6.4が可能とレジスタの回路図を示します。これはまた、一般的なデザインパターンであるとして、現代のFPGA専用のイネーブル入力が含まれているフリップフロップ、および追加のリソースが必要ありません。

²非同期リセットのサポートは現在開発中である

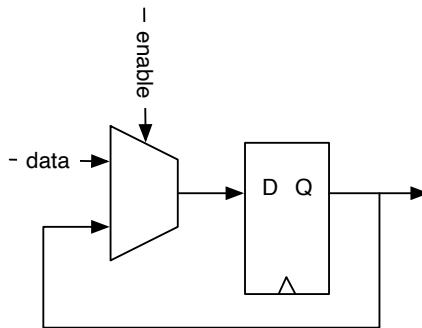


Figure 6.4: A D flip-flop based register with an enable signal.

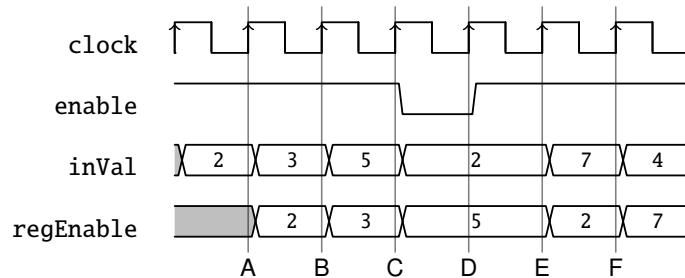


Figure 6.5: A waveform diagram for a register with an enable signal.

図～6.5有効でレジスタのための波形例を示しています。時間のほとんどは、高い (true)、それを有効にして、レジスタが1つのクロックサイクルの遅延と入力し、次。唯一の第4のクロックサイクルでenableが低く、レジスタは、立ち上がりエッジDでその値(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、信号の立ち上がりエッジを検出するために、その現在の値と最後のクロックサイクルからの値を比較します。

```
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有効に使用して、より興味深いシーケンシャル回路を構築します。

6.2 カウンター (L6109 TODO)

最も基本的な順序回路の一つはカウンターです。その最も単純な形態では、カウンタ出力は加算器に接続され、加算器の出力は、レジスタの入力に接続されるレジスタです。図～6.6は、フリーランニングカウン

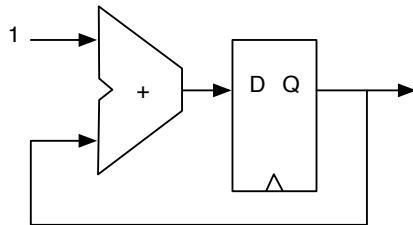


Figure 6.6: An adder and a register result in counter.

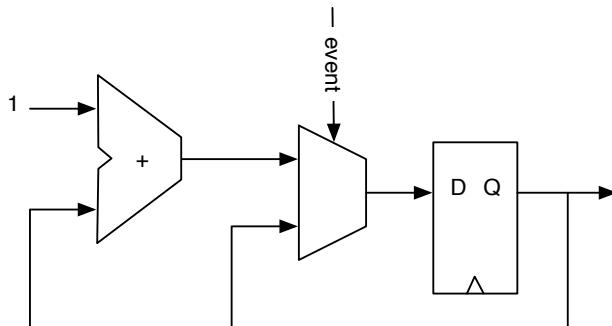


Figure 6.7: Counting events.

タを示しています。

4ビットのレジスタを持つフリーランカウンタは、0から15までカウントした後、再び0に折り返す。また、カウンタは既知の値にリセットされなければならない。

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

cntReg := cntReg + 1.U
```

私たちがイベントをカウントしたい場合には、図~6.7に、次のコードに示すように、我々は、カウンタをインクリメントする条件を使用しています。

```
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 (L6268 TODO)

値をカウントアップし、その後0で再起動するには、我々は最大の定数、Eとカウンタ値を比較する必要があります。グラム。、when条件文で。

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

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

カウンター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)
```

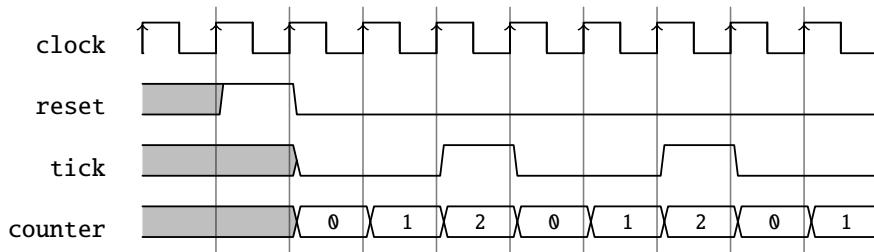


Figure 6.8: A waveform diagram for the generation of a slow frequency tick.

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
}
```

私たちはコーディングをしていて、より多くのカウンタを使っているので、私たちのためにカウンタを生成するためのパラメータを持つ関数を定義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)
```

機能genCounterの最後の文は、カウントがcntRegを登録し、この例では、関数の戻り値です。

注、すべての例では、当社のカウンターがN含む0とN、間の値を持っていたこと。私たちは10回のクロックサイクルをカウントしたい場合は、私たちは9にNを設定する必要があります。10にNを設定すると、off-by-one errorの古典的な例だろう。

6.2.2 カウンタによるタイミングの生成 (L6371 TODO)

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（壁掛け時計の時間としての時間）。同期回路は一定の周波数の時計で動作しています。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ことを、周波数の F でのシングルサイクルticksを生成することです。そのティックは、すべて N クロックサイクル、ここで $N = F/F$ を発生し、ダニが長く正確に1つのクロック・サイクルです。このダニはnot派生クロックとして使用されるが、論理的に周波数の F で動作しなければならない回路におけるレジスタのイネーブル信号として。図～6.8は、すべての3クロックサイクル発生したダニの一例を示しています。

以下の回路では、我々は0から $N - 1$ の最大値までカウントすることをカウンタについて説明します。最大値に達したとき、tickは单一サイクルのtrueであり、そしてカウンタは0にリセットされます。我々は $N - 1$ に0から数えるとき、私たちは一つの論理目盛りごとに N クロックサイクルを生成します。

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

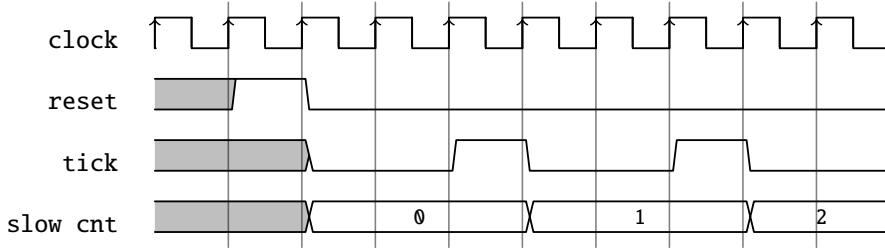


Figure 6.9: Using the slow frequency tick.

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

1のこの論理的なタイミングは、ダニのすべての N クロック・サイクルは、この遅く、論理クロックで私たちの回路の他の部分を前進させるために使用することができます。次のコードでは、私たちは別のカウンタを使用すること1インクリメントごとに N クロックサイクル。

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

図～6.9は、ダニの波形及び遅いカウンタを増分毎ティック（ N クロックサイクル）ことを示しています。

この遅いlogicalクロックの使用の例は、LEDを点滅シリアルバスのボーレートを生成し、多重化7セグメントディスプレイのための信号を生成し、ボタンおよびスイッチのデバウンスの入力値をサブサンプリングします。

幅推論サイズレジスタをべきであるが、明示的にレジスタ定義時または初期値のタイプと幅を指定することをお勧めします。 $0.U$ のリセット値は、单一ビット幅のカウンタをもたらす場合、明示的な幅の定義は、驚きを回避することができます。

6.2.3 「オタク」カウンター (L6538 TODO)

私たちの多くは、時々、nerdているように感じます。たとえば、私たちは私たちのカウンタ/ダニ世代の高度に最適化されたバージョンをデザインしたいです。標準カウンタは、次のリソースを必要とする：一つのレジスターの加算（または減算）、及びコンパレータ。私たちは、レジスタや加算器についての多くを行うことはできません。我々はカウントアップした場合、我々はビット列である数、と比較する必要があります。コンパレータは、ビット列のゼロと大型ANDゲートのためのインバータ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と、コンパレータは、ASICの定数に対するコンパレータよりも少し安くなる可能性がある、大きなNORゲートです。ロジックはルックアップテーブルから構築されているFPGAでは、0又は1と比較するとの間に差がありません。リソース要件は、アップとダウンカウンタでも同じです。

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。上か下かをカウントするには、これまでのところ、すべてのカウントビットとの比較が必要でした。N-2から-1までカウントするは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る必要があります。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
```

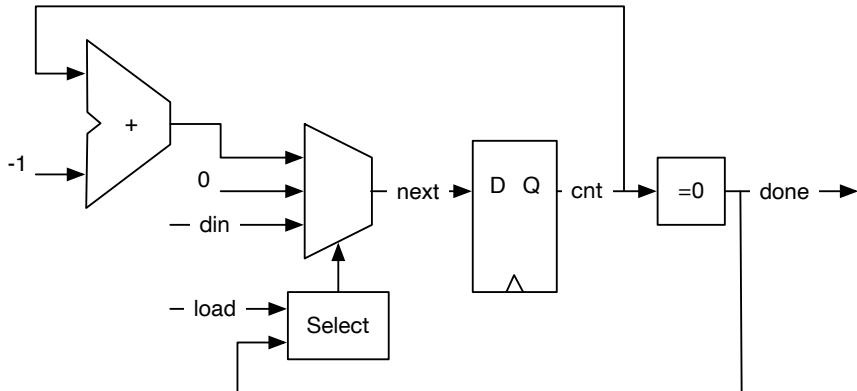


Figure 6.10: A one-shot timer.

```

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

```

Listing 6.1: A one-shot timer

```

    io.tick := true.B
}

```

6.2.4 タイマー (L6610 TODO)

私たちが作成できるタイマーの別の形態は、ワンショットタイマです。ワンショット・タイマは、キッキンタイマーのようなものです：あなたが分、プレス開始の数を設定します。指定された時間が経過すると、アラームが鳴ります。デジタルタイマーは、クロック・サイクルの時間をロードされます。そして、それがゼロに達するまでカウントダウン。ゼロでタイマーが`done`を主張します。

図～6.10は、タイマのブロック図を示します。レジスタは`load`をアサートすることによって`din`の値をロードすることができます。`load`信号である場合にはデアサートカウントダウンする（レジスタに対する入力として`cntReg - 1`を選択することによって）選択されます。カウンタが`0`に到達すると、信号`done`がアサートされ、カウンタが`0`を提供するマルチプレクサの入力を選択することによりカウントを停止します。

～6.1をリストタイマーのChiselコードを示します。私たちは、`0`にリセットされ、8ビットのレジスタ`reg`を、使用しています。ブール値`done`は`0`で`reg`を比較した結果です。入力マルチプレクサのために我々は`0`のデフォルト値を持つワイヤ`next`を紹介します。`when/elsewhen`ブロックは、選択機能を持つ他の2つの入力を導入しています。信号`load`はデクリメントの選択よりも優先されます。最後の行は、レジスタ`reg`の入力に、`next`で表されるマルチプレクサを接続します。

我々はもう少し簡潔なコードを目指す場合は、代わりに直接、中間ワイヤ`next`を使用する、レジスタ`reg`マルチプレクサ値を割り当てることができます。

6.2.5 パルス幅変調(PWM) (L6721 TODO)

Pulse-width modulation (PWM) は、一定周期の信号であり、時間の変調は、信号がその期間内に`high`あります。

図～6.11は、PWM信号を示しています。矢印は信号の周期の開始を指します。信号がハイである時間の割合は、また、デューティ・サイクルと呼ばれています。

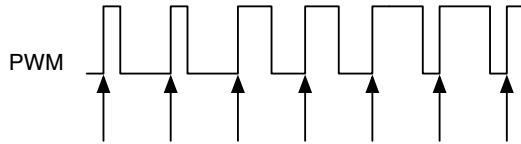


Figure 6.11: Pulse-width modulation.

簡単digital-to-analog converterにおけるPWM信号の結果にlow-pass filterを追加します。ローパスフィルタは、抵抗およびコンデンサのような単純なようであることができます。

次のコード例では、10クロックサイクルごとに3クロックサイクルのハイ（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)
```

我々は、再利用可能な、軽量コンポーネントを提供するために、PWMジェネレータの機能を使用しています。（nrCycles）クロックサイクル数にPWMを設定Scalaの整数であり、PWM出力信号のデューティサイクル（pulswidth）を与えるChisel線（din）：関数は、2つのパラメータを有しています。我々はカウンターを表現するために、この例では、マルチプレクサを使用しています。関数の最後の行は、PWM信号を返すように入力値dinとカウンタ値とを比較します。Chisel関数の最後の式は、我々の場合には、比較機能に接続されたワイヤの戻り値です。

我々は、最大の符号なし数値を表すのに必要なカウンタcntRegのビット数（を含む）nを指定する機能unsignedBitLength(n)を使用します。³ Chiselはまた、数の符号付き表現のためのビットの数を提供するために機能signedBitLengthを有しています。

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、三角関数で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)
```

我々はカウントアップまたはダウンしなければならないかどうかを判断するためのフラグとして（1）modulationRegカウントアップするとダウンし、（2）upReg：私たちは、変調のための2つのレジスタを使用します。当社は、クロック入力の周波数にカウントアップ（100MHzで私たちの例では）、0の信号で結果。0.5ヘルツ。アップ又はダウンカウントおよび方向のスイッチ長いwhen/.elsewhen/.otherwise式ハンド

³ ビット数が符号なしの数Nバイナリを表すために $\lceil \log_2 N \rceil + 1$ です。

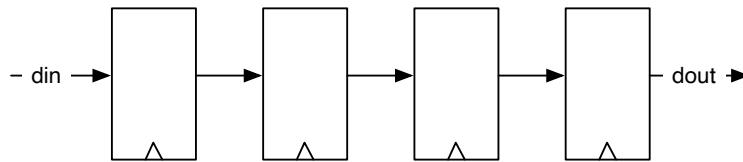


Figure 6.12: A 4 stage shift register.

ル。

までしか周波数の1000分に私達のPWMカウントは1kHzの信号を生成するように、我々は1000で変調信号を分割する必要があります。実際の分割は、ハードウェアで非常に高価であるので、我々は、単による除算を相当右、10だけシフト。我々は機能としてPWM回路を定義したように、我々は単に関数呼び出しとその回路をインスタンス化することができます。ワイヤsigはPWM変調信号を表します。

6.3 シフトレジスタ (L6919 TODO)

A **shift register**は、フリップフロップ順に接続のコレクションです。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ています。図～6.12は4段のシフトレジスタを示しています。回路shifts各クロックチックに左から右へのデータ。この単純な形態では、回路はdoutにdinから4タップ遅延を実現します。

この単純なシフトレジスタのChiselコードを行います4ビットshiftRegレジスタ作成（1）、（2）レジスタへの次の入力のための入力din有するシフトレジスタの下位3ビットを連結し、そして（3）用途出力doutとしてレジスタの最上位ビット（MSB）。

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

シフトレジスタは、多くの場合、シリアルデータからパラレルデータまたはシリアルデータをパラレルデータに変換するために使用されます。節～11.2は用途が受信および送信機能するシフトレジスタというシリアルポートを示します。

6.3.1 パラレル出力付きシフトレジスタ (L6992 TODO)

シリアルにシフトレジスタのパラレルアウト構成は、パラレルワードにシリアル入力ストリームを変換します。これは、受信機能のためにシリアルポート（UART）で使用することができます。図～6.13は、各フリップフロップの出力が1つの出力ビットに接続される4ビットのシフトレジスタを示しています。4クロックサイクル後、この回路はqで提供され、4ビットパラレルデータワードに4ビットのシリアル・データ・ワードに変換します。この例では、最初に送信されるビット0（最下位ビット）を想定し、我々は完全な言葉を読みたいときでの、最後の段階に到着します。

次Chiselコードでは、0とoutRegシフトレジスタを初期化します。その後、我々は右シフトを意味し、MSBからシフトします。パラレル結果、qは、レジスタoutRegだけの読書です。

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

図～6.13は、並列出力機能を有する4ビットのシフトレジスタを示しています。

6.3.2 パラレルロード付きシフトレジスタ (L7067 TODO)

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、並列ロード機能を有する4ビットのシフトレジスタを示しています。その関数のChiselの説明は比較的簡単です：

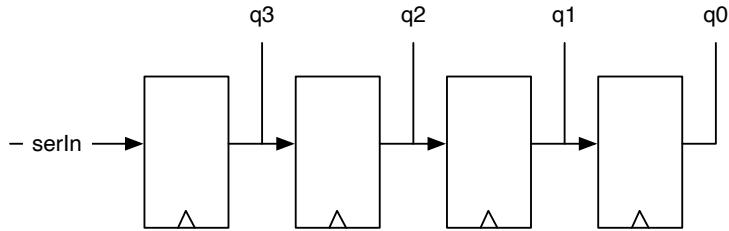


Figure 6.13: A 4-bit shift register with parallel output.

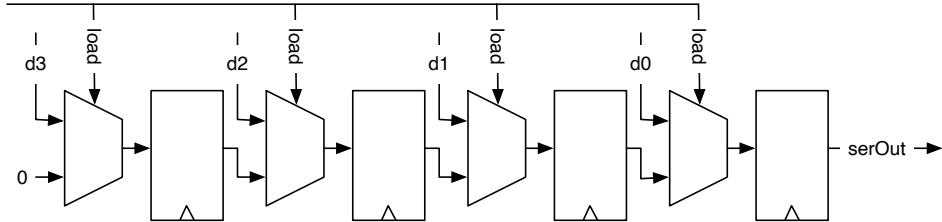


Figure 6.14: A 4-bit shift register with parallel load.

```

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

```

現在は右にシフトして、MSB のゼロを埋めていることに注意してください。

6.4 メモリー (L7128 TODO)

メモリはChiselでは、レジスタの集まりでVecのRegを構築することができます。しかし、これはハードウェアに高価であり、より大きなメモリ構造はSRAMとして構築されています。ASICの場合は、メモリコンパイラは、思い出を構築します。FPGAは、オンチップ・メモリ・ブロックとも呼ばれるブロックRAMが含まれています。これらのオンチップ・メモリ・ブロックは、より大きいメモリのために組み合わせることができます。FPGA内のメモリは、通常、1点のリードと1つの書き込みポート、または実行時に読み取りおよび書き込みの間で切り替えることができる2つのポートを有しています。

FPGA（およびASICも）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は、同期メモリの回路図を示しています。メモリは、1つの読み出しポートと1つの書き込みポートとデュアルポートです。読み出しポートは、単一の入力を有し、読み出しアドレス（rdAddr）と1つの出力、リードデータ（rdData）。アドレス（wrAddr）、（wrData）に書き込まれるデータ、および書き込みは（wrEna）を有効：ライト・ポートは3つの入力を持っています。すべての入力のために、同期挙動を示すメモリ内のレジスタがあることに留意されたいです。

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を提供します。一覧～6.2はバイト幅の入力データと出力データとの実装メモリの1～KiBの書き込みを有効にすることをコンポーネントMemoryを示しています。

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、メモリの読み書き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、読み取りデータが未定義であることを指摘します。

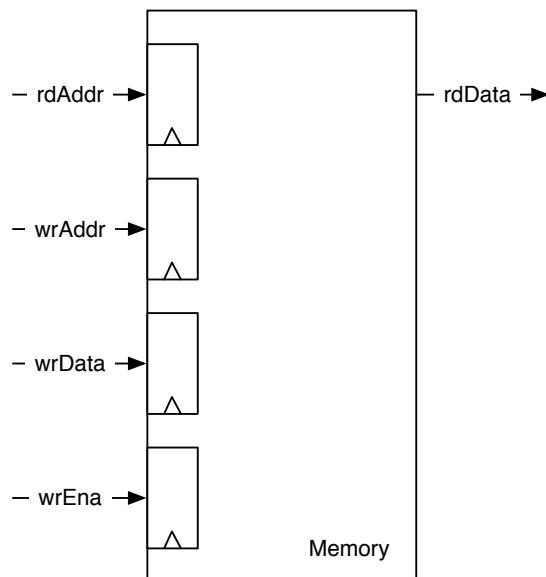


Figure 6.15: A synchronous memory.

```

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

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

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

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

```

Listing 6.2: 1 KiB of synchronous memory.

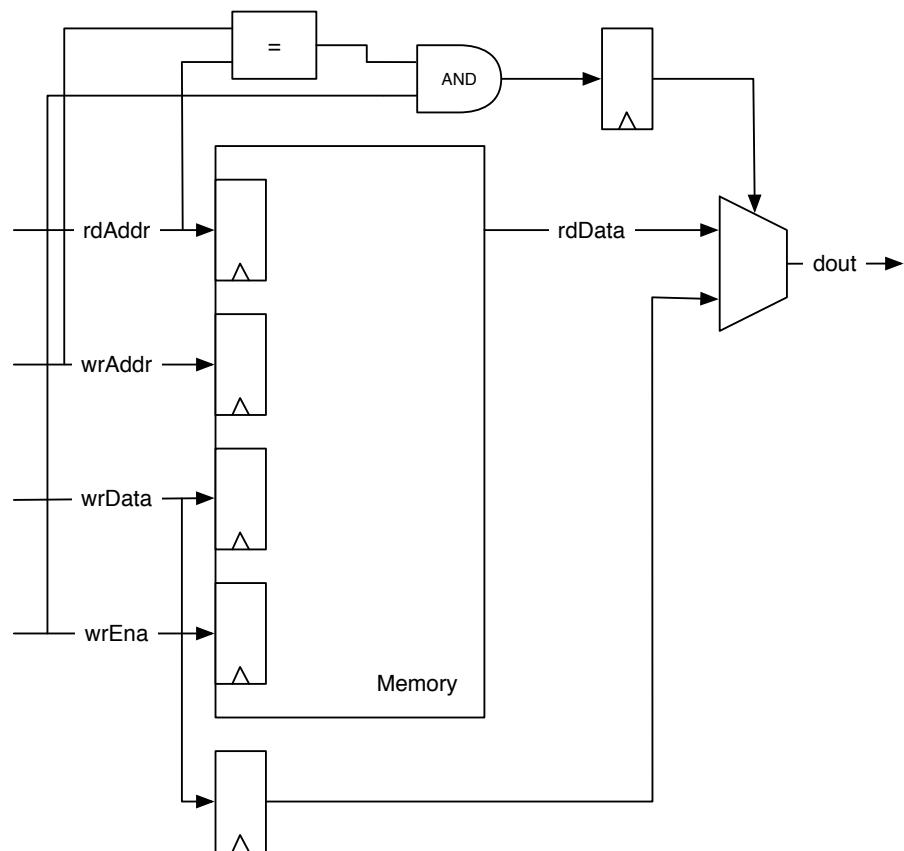


Figure 6.16: A synchronous memory with forwarding for a defined read-during-write behavior.

```

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

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

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

    val memData = mem.read(io.rdAddr)

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

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

```

Listing 6.3: A memory with a forwarding circuit.

我々は新たに書き込まれた値を読み出したい場合、我々はアドレスは、書き込みデータ等しいと*forwards*されていることを検出することを転送回路を構築することができます。図～6.16は、転送回路とメモリを示しています。読み出しおよび書き込みアドレスは、データを読み取り、書き込みデータの転送経路又はメモリとの間で選択することが可能に比べ書き込みでゲートされます。書き込みデータは、レジスタと1つのクロックサイクルによって遅延されます。

一覧～6.3は、転送回路を含む同期メモリのChiselコードを示します。我々はまた、次のクロック・サイクルでの読み出し値を提供し、同期メモリを合わせるために、次のクロックサイクルで利用できるようになるレジスタ（wrDataReg）への書き込みデータを格納する必要があります。我々は2つの入力アドレス（wrAddrとrdAddr）を比較し、wrEnaは、転送条件に当てはまるかどうかを確認します。その条件はまた、1つのクロックサイクルによって遅延されます。転送（書き込み）データ又はメモリから読み出されたデータとの間のマルチプレクサを選択します。

Chiselはまた、同期書き込みと非同期読み出しとメモリを表すMemを提供します。このメモリタイプは、FPGAに通常直接使用できないため、合成ツールは、フリップフロップのそれを構築します。したがって、我々はSyncReadMemを使用することをお勧めします。

6.5 演習 (L6986 7332)

最後運動から7セグメントエンコーダを使用しFに0から表示を切り替えるための入力として4ビットカウンタを追加します。あなたが直接FPGAボードのクロックにこのカウンタを接続した場合、あなたはすべての16個の数字は、（すべての7つのセグメントが点灯します）重なって表示されます。そのため、あなたはカウントを遅くする必要があります。单一サイクルtickは500ミリ秒ごとに信号を生成することができます別のカウンタを作成します。4ビットカウンタのインペブル信号としてその信号を使用します。

発電機能を有するPWM波形を構築し、機能（三角形または正弦関数）としきい値を設定します。三角関数は、アップとダウンカウントによって作成することができます。あなたはScalaの数行のコードを生成することができ、ルックアップテーブルを用いて、洞機能（節～10.2を参照してください）。その変調PWM機能をFPGAボード上のLEDを駆動します。あなたのPWM信号は、どのような周波数でなければなりませんか？ドライバは何周波数を実行していますか？

デジタル設計は、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 }
}
```

ここでは、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 (L7452 TODO)

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ー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。

後者の二つの問題、デバウンススイッチ、およびノイズフィルタリングは、外部、アナログコンポーネントと解決することができます。しかし、それはデジタル領域でこれらの問題に対処するために多くの(費用)効率的です。

7.1 非同期入力 (L7502 TODO)

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してもよいです。この違反は、フリップフロップのMetastabilityをもたらすことができます。準安定性は、0と1の間の出力値をもたらすことができるか、発振を生じ得ます。しかし、いくつかの時間後に、フリップフロップは、0または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が侵害されないようにになります。

図～7.1は同期回路と外部世界との間の境界線を示しています。入力同期は、2つのフリップフロップから構成されています。入力同期用Chiselのコードは、2つのレジスタをインスタンスワンライナーです。

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

すべての非同期外部からの信号が入力シンクロナイザを必要としています。¹また、外部リセット信号を同期する必要があります。リセット信号が通過しなければならない2つのフリップフロップは他のリセット信号として使用される前に回路内のフリップフロップ。クロックに同期するリセットニーズのデアセッションコンクリート。

7.2 デバウンス (L7640 TODO)

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つの状態の間で跳ね返ることができます。私たちは、さらに処理せずに、このような信号

¹入力信号は同期出力信号に依存している場合は例外であり、我々は最大伝播遅延を知っています。古典的な例は、同期回路、例えば非同期SRAMのインターフェースです。グラム、マイクロプロセッサによる。

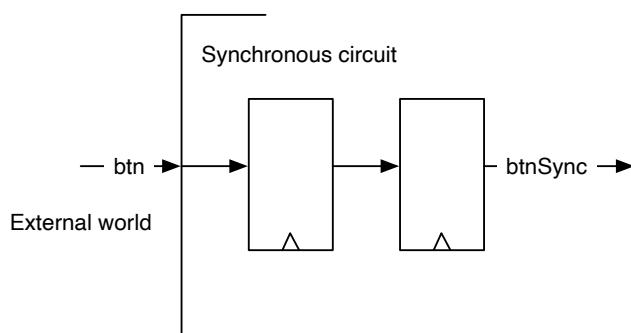


Figure 7.1: Input synchronizer.

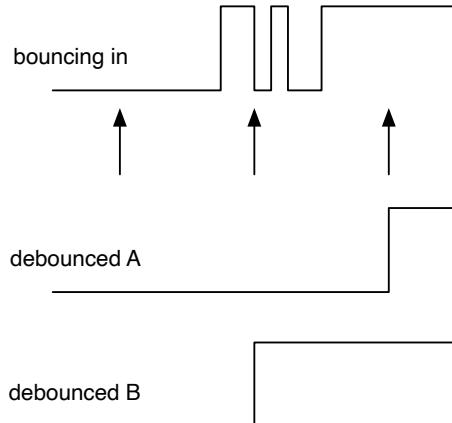


Figure 7.2: Debouncing an input signal.

を使用している場合は、私たちがしたいより多くの遷移イベントを検出することができます。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。私たちは、さらに下流のサンプリングされた信号を使用します。

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。バウンス領域に入る前のサンプルは安全に0を読み、バウンシング領域に入った後のサンプルは安全に1を読みます。バウンス領域に入ったサンプルは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はアクションでデバウンスのためのサンプリングを示しています。トップ信号が示すバウンス入力、及び下矢印はサンプリング点を示します。これらのサンプリング点間の距離が長く、最大バウンス時間よりもする必要があります。安全第一のサンプルは、サンプル0を、図サンプル1内の最後のサンプル。中央のサンプルは、バウンス時に落ちます。これは、0または1です。二つの可能な結果はdebounce Aとdebounce Bとして示されています。両方の0から1への单一の遷移を有します。これら2つの結果の間の唯一の違いは、バージョンBの遷移は、一つのサンプル期間後であることです。しかし、これは通常、非問題です。

デバウンス用Chiselコードがもう少し進化シンクロナイザのコードよりなります。私たちは、セクション～6.2.2に行ったように、单一サイクルtick信号を出力するカウンタとサンプルタイミングを生成します。

```

val FAC = 1000000000/100

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定する必要があります。上記の例は、（バウンス時間は10～ミリ秒未満であることを仮定して）100～[Hz]のサンプリング周波数で100～MHzクロックと結果を想定しています。最大カウンタ値はFAC、分割係数です。私たちは、リセット値なし、デバウンス信号用レジスタbtnDebRegを定義します。レジスタcntRegカウンタとして機能し、カウンタが最大値に達したときにtick信号が真です。その場合、when条件は、カウンタが0と (2) デバウンスレジスタは入力サンプルにリセットされtrueと (1) です。それは前のセクションで示した入力同期から出力されるようにこの例では、入力信号がbtnSync命名されます。

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

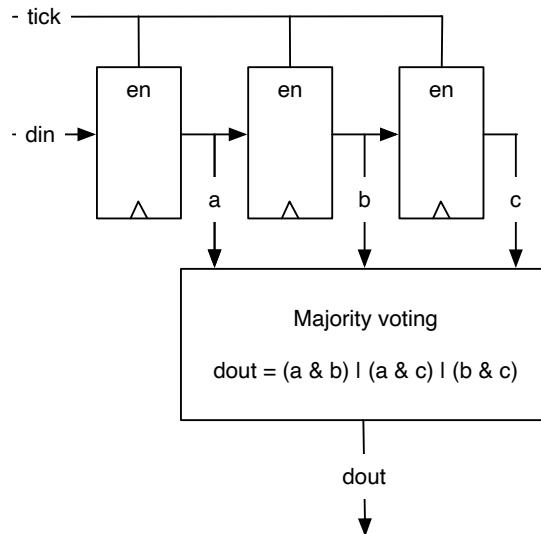


Figure 7.3: Majority voting on the sampled input signal.

メインで処理します。

7.3 入力信号のフィルタリング (L7820 TODO)

時々、私たちの入力信号は、多分私達は、入力同期とデバウンス単位で意図せずにサンプリング可能性があるスパイクを含む、うるさいかもしれません。それらの入力スパイクをフィルタリングするための1つのオプションは、多数決回路を使用することです。最も単純なケースでは、我々は三つのサンプルを取り、多数決を行います。中央値関数に関連する**majority function**は、大部分の値になります。私たちはデバウンスのためのサンプリングを使用我々のケースでは、我々は、サンプリングされた信号の多数決を行います。信号がサンプリング周期よりも長い間安定していることを多数決保証します。

図～7.3は、大多数の有権者の回路を示しています。これは、レジスタは、我々がデバウンスサンプリングに使用`tick`信号によってイネーブル3ビットのシフトで構成されています。3つのレジスタの出力は、多数決回路に供給されます。多数決関数は、サンプル期間よりも短い任意の信号変化をフィルタリングします。

以下Chiselコード示す3ビットのシフトレジスタ、信号`btnClean`その結果、`tick`信号と投票機能により可能となりました。

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

当社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`遅延素子との立ち上がりエッジを検出し、その後増加にカウンターを有効にするために`btnClean`の現在の値と、この信号を比較します。

```
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
```

```

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

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

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

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

val btnSync = sync(btn)

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

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

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

```

Listing 7.1: Summarizing input processing with functions.

}

7.4 入力処理と関数の組み合わせ (L7936 TODO)

入力処理を要約すると、我々はいくつかのより多くのChiselのコードを示しています。提示回路は小さなことが、再利用可能なビルディング・ブロックかもしれません、我々は機能でそれらをカプセル化します。節~4.4ではなく、完全なモジュールの軽量リグ機能でどのように我々はできる、抽象小さなビルディングブロックを示しました。これらのリグ機能は、電子をハードウェアのインスタンスを作成します。グラム。、機能syncは、2個の入力に、互いに接続されたフリップフロップ作成します。関数は、第2のフリップフロップの出力を返します。便利な場合は、それらの機能は、いくつかのユーティリティクラスオブジェクトに上昇させることができます。

7.5 演習 (L7981 DONE)

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構築する。(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ーとするエッジ検出回路、を備えた入力処理チェーンを構築します。

最近のボタンは必ず跳ね返る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、跳ね返り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。ボタンを数回連続で押すことで、ボタンの跳ね返りをシミュレートする。安定したプレスに落ち着く前に、回路をテストしてください。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。多数決では、カウンターの確実なインクリメントには1秒から2秒の間に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。

8 有限状態機械 (FSM) (L8048 TODO)

有限状態機械 (FSM) は、デジタル設計の基本的なビルディング・ブロックです。AN FSMは状態間statesのセットと条件 (ガード) state transitionsとして説明することができます。アンFSMは、リセット時に設定された初期状態を、持っています。FSMは、同期順序回路と呼ばれています。

(1) 現在の状態を保持するレジスタ、(2) 現在の状態と入力に依存する次の状態を計算し、組合せ論理、および計算 (3) 組み合わせ論理：FSMの実装は、3つの部分から構成さFSMの出力。

原理的には、格納状態にレジスタまたは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、電子実用的ではないかもしれません。グラム。、単一FSMとしてあなたのラップトップを記述してみてください。次の章では、FSMの通信にそれらを組み合わせることにより、より小さなFSMのうち、大規模なシステムを構築する方法について説明します。

8.1 有限状態機械の基本 (L8124 TODO)

図～8.1は、FSMの回路図を示しています。レジスタは、現在のstateが含まれています。次の状態の論理は、現在stateと入力 (in) から次の状態値 (next_state) を算出します。次のクロックチックで、stateはnext_stateになります。出力論理は、出力 (out) を計算します。出力は現在の状態のみに依存したように、このステートマシンはMoore machineと呼ばれています。

A state diagramは、視覚的なFSMの動作について説明します。状態図では、個々の状態は、状態名で標識された円として示されています。状態遷移は、状態間の矢印で示されています。この遷移が行われるガード（または状態）は矢印のラベルとして描かれています。

図～8.2は、単純な例のFSMの状態図を示します。green、orange、及びred、警報のレベルを示す：FSMは、三つの状態を有します。FSMはgreenレベルで開始します。bad eventが発生するとアラームレベルはorangeに切り替えていきます。二悪いイベントでは、警報レベルはredに切り替えていきます。その場合には、我々は、鐘を鳴らしたいです。ring bellはこのFSMの出力のみです。私たちは、red状態に出力を追加します。アラームはclear信号とリセットすることができます。

状態図は、視覚的に満足することができ、FSMの機能を迅速に把握することができるが、状態テーブルは書き留めて速くすることができます。表～8.1は、私たちのアラームFSMの状態テーブルを示しています。我々は、現在の状態、入力値の、結果として得られる次の状態、および現在の状態に対する出力値をリストします。原則として、我々はすべての可能な状態のすべての可能な入力を指定する必要があります。私たちは、clear入力がbad eventが発生したときに気にしないであることを示すことにより、テーブルを簡素化します。手段bad eventはclearに優先を持っていること。出力列には、いくつかの繰り返しを持っています。我々は、より大きなFSM及び/又は複数の出力がある場合、我々は2つの次の状態ロジック用と出力論理のための1つにテーブルを分割することができます。

最後に、私たちの警告レベルのFSMのすべての設計した後、我々は、Chiselでそれをコーディングするものとします。一覧～8.1は、アラームFSMのためのChiselコードを示します。注意、我々は入力用ChiselタイプBoolとFSMの出力を使用すること。Enumとswitch制御命令を使用するために、我々は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のための完全なチゼルコードは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きましょう。FSMは2つの入力と1つの出力信号を持っています。Chisel Bundleでキャプチャされ

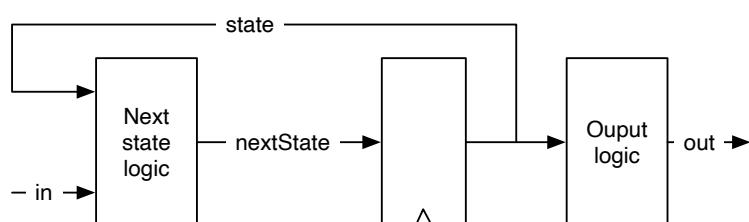


Figure 8.1: A finite state machine (Moore type).

Table 8.1: State table for the alarm FSM.

State	Input			
	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

```

import chisel3._
import chisel3.util._

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

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

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

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

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

```

Listing 8.1: The Chisel code for the alarm FSM.

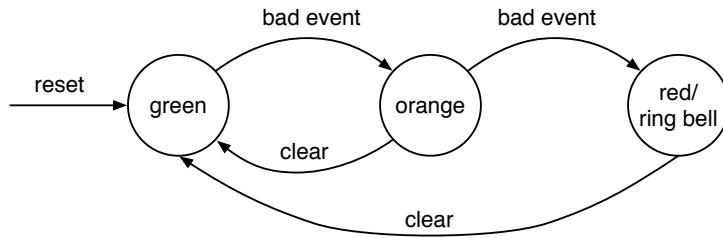


Figure 8.2: The state diagram of an alarm FSM.

ています。

```

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

```

かなりの作業が最適な状態エンコーディングに費やされました。二つの一般的なオプションは、バイナリまたはワンホットエンコーディングです。しかし、我々はのsynthesizeツールにこれらの低レベルの意思決定を残して、読みやすいコードを目指します。ChiselEnumタイプの現在のバージョンで¹したがって、我々は状態のシンボリック名を列挙型を使用します。

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

個々の状態値は、個々の要素は::演算子で連結されたリストとして記述されています。Nilは、リストの最後を表します。Enumインスタンスは、状態のリストにassignedです。状態を保持するレジスタをリセット値としてgreen状態と定義されます。

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

FSMの肉は、次の状態論理です。我々は、すべての州をカバーするために、状態レジスタにChiselスイッチを使用します。各is支店内で私達は私達の状態レジスタに新しい値を割り当てることによって、入力に依存して次の状態ロジックを、コード：

```

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
        }
    }
}

```

状態がredときに最後には、ではなく、少なくとも、私たちはringing bell出力が真であることをコーディングします。

¹バイナリエンコーディングの状態を表しています。我々は別のエンコーディングを使用する場合は、電子。グラム。、ワンホットエンコーディング、我々は状態名のChisel定数を定義することができます。

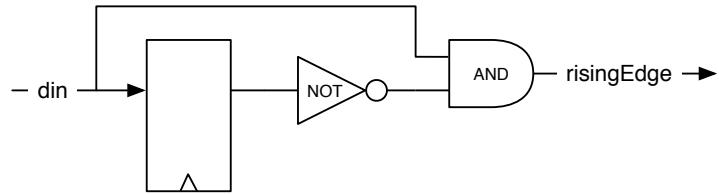


Figure 8.3: A rising edge detector (Mealy type FSM).

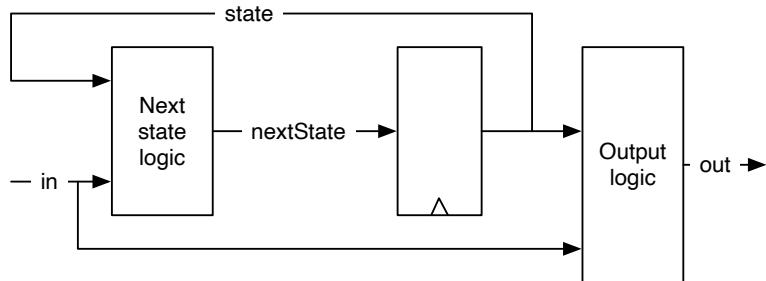


Figure 8.4: A Mealy type finite state machine.

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

それはVerilogまたはVHDLで一般的に行われているとして、我々は、レジスタの入力にnext_state信号をnot導入しなかったことに注意してください。VerilogおよびVHDLのレジスタは、特別な構文に記載されており、組み合わせブロック内で割り当てられた（及び再割り当て）することができません。したがって、組合せブロックで計算付加信号は、導入及びレジスタ入力に接続されています。Chiselのレジスタは、基本型であり、自由に組み合わせたブロック内で使用することができます。

8.2 ミーリー FSMで出力を高速化 (L78454 TODO)

ムーアFSMでは、出力は現在の状態に依存します。入力の変化は、次のクロックサイクルにおいて出力earliestの変化として見ることができます。我々はすぐに変化を観察したい場合は、我々は、入力から出力への組み合わせパスが必要です。私たちは、最小限の例では、エッジ検出回路を考えてみましょう。我々は前にこのリグワントライナーを見てきました：

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

図～8.3は、立ち上がりエッジ検出器の概略図を示します。現在の入力が1であり、最後のクロックサイクルで入力が0のとき出力は、1つのクロック・サイクルの1となります。状態レジスタは、次の状態がちょうど入力されたただ1つのDフリップフロップです。また、1つのクロックサイクルの遅延要素としてこれを考慮することができます。出力論理は、現在の状態と現在の入力をcompares。

とき出力iは、入力にも依存します。え。、これはMealy machine呼ばれるFSMの入力と出力との間の組み合わせパスがあります。

図～8.4はミーリー型FSMの概略図を示します。ムーアFSMと同様に、レジスタは現在stateを含み、次の状態の論理は、現在stateと入力(in)から次の状態値(next_state)を算出します。次のクロックチックで、stateはnext_stateになります。出力論理は、現在の状態and FSMへの入力から出力(out)を計算します。

図～8.5は、エッジ検出のためのミーリーFSMの状態図を示します。状態レジスタが1つだけDフリップフロップで構成されているように、2つだけの状態では、我々は、この例ではzeroとoneに名前を付けた、可能です。ミーリーFSMの出力が状態だけでなく、入力に依存しないように、我々は國家の円の一部として出力を記述することはできません。その代わりに、状態間の遷移は、入力値(条件)and(スラッシュの後の)出力で標識されます。私たちは、電子を自己遷移を描画することにも注意してください。グラム。、状態zeroに入力0状態zeroにおけるFSMの滞在である場合、出力は0です。立ち上がりエッジFSMは、状態zeroから状態oneへの遷移に1出力を生成します。入力が現在1であることを表す状態oneにおいて、出

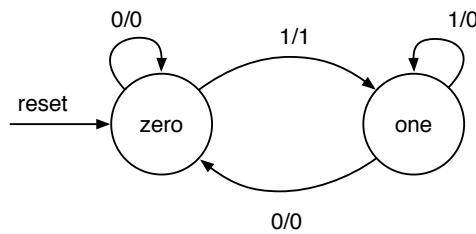


Figure 8.5: The state diagram of the rising edge detector as Mealy FSM.

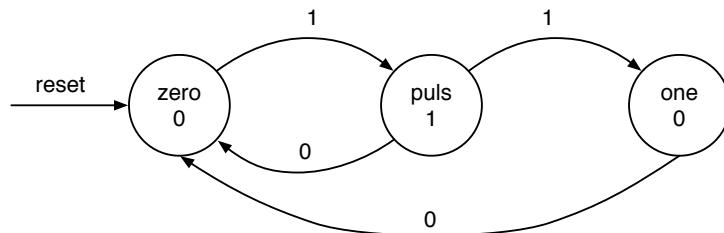


Figure 8.6: The state diagram of the rising edge detector as Moore FSM.

力はあります。私たちは、入力の各立ち上がりエッジのための単一の（サイクル）のパルスをしたいです。

～8.2をリストミーリー機械と立ち上がりエッジ検出のためのChiselコードを示します。前の例のように、我々は、単一ビットの入力及び出力のためのChisel型Boolを使用します。出力論理は現在、次の状態の論理の一部です。zeroからoneへの遷移で、出力がtrue.Bに設定されています。出力にそれ以外の場合は、デフォルトの割り当て（false.B）カウント。

特に、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格的なFSMが最適かどうかを問うことができます。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。この機能のためには、ワンライナーの方が書きやすく、読みやすいので、より重要です。したがって、ワンライナーが好ましい解決策である。

この例を用いて、可能な限り最小のMealy FSMの1つを示した。FSMは、3つ以上の状態を持つより複雑な回路に使用するものとする。

8.3 ムーア対ミーリー (L78692 TODO)

ムーアFSMとミーリー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ムーアFSMを有する立ち上がりエッジ検出のための状態図を示します。通知に最初の事はムーアFSMはミーリバージョンで二つの状態に比べて3つの状態が、必要があることです。状態pulsは、シングルサイクルPULSを生成するために必要とされます。状態pulsちょうど1クロックサイクルでFSM滞在し、入力が再び0になるのを待って、バック開始状態zeroまたはone状態のいずれかに進みます。我々は、状態遷移矢印の入力条件及び状態を表す円内のFSMの出力を示します。

一覧～8.3は、立ち上がり検出回路のムーアのバージョンを示しています。これはミーリーよりフリップフロップDの数を2倍または符号化されたバージョンを指示する使用します。得られた次の状態の論理は、従って、ミーリーまたは直接符号化されたバージョンよりも大きくなっています。

図～8.7はミーリー、立ち上がりエッジ検出FSMのムーアバージョンの波形を示しています。私たちは、ムーアの出力は、クロックティックの後に上昇しながらミーリー出力は密接に、入力の立ち上がりエッジを、次のことわかります。また、ムーア出力はミーリー出力が通常より少ないクロック・サイクルよりも1つのクロックサイクルの広い、あることがわかります。

上記の例から、一方は、それらがより少ない状態（従って論理）を必要と高速ムーアFSMよりも反応するようミーリーFSMのbetterのFSMを見つけるために誘惑されます。しかし、ミーリー機械内の組み合わせパスは、大規模なデザインでトラブルを引き起こす可能性があります。まず、FSMを（次の章を参照してください）通信のチェーンで、この組み合わせパスが長くなることができます。通信のFSMが円を構築する場合には、第2、結果は、同期設計の誤りである組合セループがあります。ムーアFSMの状態レジスタ

```

import chisel3._
import chisel3.util._

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

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

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

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

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

```

Listing 8.2: Rising edge detection with a Mealy FSM.

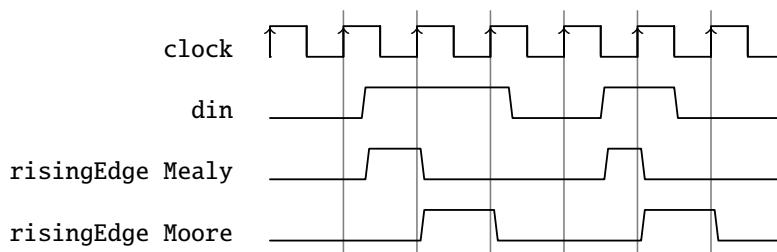


Figure 8.7: Mealy and a Moore FSM waveform for rising edge detection.

```

import chisel3._
import chisel3.util._

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

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

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

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

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

```

Listing 8.3: Rising edge detection with a Moore FSM.

との組み合わせパスでカットのためには、上記のすべての問題は、ムーアのFSMを通信するために存在していません。

要約すると、ムーアのFSMは、ステートマシンを通信するためのより良い組み合わせ。彼らはミーリのFSMより *more robust* です。使用ミーリのFSM同じサイクル内での反応が最も重要であるのみ。事実上ミーリマシンですな立ち上がりエッジ検出などの小型回路は、同様に罰金です。

8.4 演習 (L8843 TODO)

今では、いくつかの *real* FSMコードを書くための時間です。もう少し複雑な例を選び、FSMを実装し、そのためのテストベンチを書きます。

FSMのための古典的な例は、トラフィック光コントローラ (~ [3, Section 14.3]を参照します)。トラフィックライトコントローラは、赤から緑にスイッチの交差点の両方の道路が無行く光（赤、オレンジ）を有する場合との間の位相があることを確認しなければなりません。もう少し面白いこの例を作成するには、優先道路を考えます。マイナーな道路は、（交差点に両方のエントリに）2つの車の検出器を持っています。車が検出された後、優先道路のために戻って緑に切り替えているだけマイナーな道路のための緑に切り替えます。

9 コミュニケートステートマシン (L8903 TODO)

問題はしばしば、単一のFSMで記述するには複雑すぎることがあります。その場合、問題を2つ以上のより小さくて単純なFSMに分割することができます。そして、それらのFSMは信号で通信します。1つのFSMの出力は別のFSMの入力を入力し、そのFSMは他の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場合、これをファクタリングFSMと呼びます。しかし、多くの場合、通信するFSMは仕様から直接設計されています。多くの場合、単一のFSMは、実行不可能な大規模なものになります。

9.1 ライトフラッシャーの例 (L8944 TODO)

FSMの通信について議論するために、我々は～[3, Chapter 17]、光フラッシャーからの例を使用します。光フラッシャーは、1つの入力startと1つの出力lightを有しています。次のように光フラッシャーの仕様は次のとおりシーケンスが開始されます。次のスタートのために。

- when start is high for one clock cycle, the flashing sequence starts;
- the sequence is to flash three times;
- where the light goes *on* for six clock cycles, and the light goes *off* for four clock cycles between flashes;
- after the sequence, the FSM switches the light *off* and waits for the next start.

ERROR-TBD

マスターFSMを実装点滅ロジック、およびタイマーFSMの実装待ち：問題は、2つの小さなFSMにこの大規模なFSMを因数分解することで、よりエレガントに解決することができます。図～9.1は2つのFSMの組成を示します。

タイマー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。タイマーの仕様は以下の通りです。

- when timerLoad is asserted, the timer loads a value into the down counter, independent of the state;
- timerSelect selects between 5 or 3 for the load;
- timerDone is asserted when the counter completed the countdown and remains asserted;
- otherwise, the timer counts down.

以下のコードは、ライトフラッシャーのタイマーFSMを示しています。

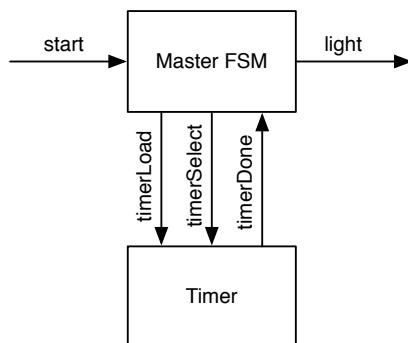


Figure 9.1: The light flasher split into a Master FSM and a Timer FSM.

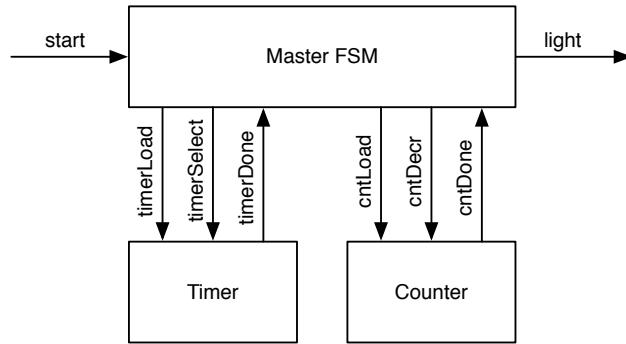


Figure 9.2: The light flasher split into a Master FSM, a Timer FSM, and a Counter FSM.

```

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
    }
}

```

一覧～9.1は、マスターFSMを示しています。

マスターFSMとタイマーとのソリューションは、まだマスターFSMのコード内の冗長性を持っています。米国flash1、flash2、およびflash3は、同じ機能を実行している、うまくしてspace1とRRR011TeVPYxsjを述べています。私たちは、第二のカウンタに残っている点滅の数を考慮することができます。off、flash、およびspace：次にマスターFSMは、次の3つの状態に還元されます。

図～9.2は、ショーマスターFSMと2つのFSMとデザインその回数：onとoffの間隔の長さのためのクロックサイクルをカウントする1つのFSM；残りの点滅をカウントするための第2のFSM。

コード・ショーダウンカウンタFSM次のとおりです。

```

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

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

```

注、カウンタはそれがremainingが点滅をカウントし、タイマーが実行されたときに、状態spaceにデクリメントされて、2～3のための点滅がロードされていること。一覧～9.2は、二重のリファクタリングフランジャーのマスターFSMを示しています。

ただ三つの状態に還元され、マスターFSMを持つだけでなく、私たちの現在のソリューションはまた、より良い設定可能です。我々はonまたはoff間隔の長さや点滅の数を変更したい場合は、[いいえ]FSMを変更する必要があります。

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。データの協調的な交換にはハンドシェイク信号を使用します。次項では、一方向データ交換のフロー制御のためのレディバリッドインター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 (L9194 TODO)

ステートマシンの通信の一つの典型的な例は、データパスと組み合わさ状態マシンです。この組み合わ

```

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

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

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

timerLoad := timerDone

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

```

Listing 9.1: Master FSM of the light flasher.

```

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

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

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

timerLoad := timerDone

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

```

Listing 9.2: Master FSM of the double refactored light flasher.

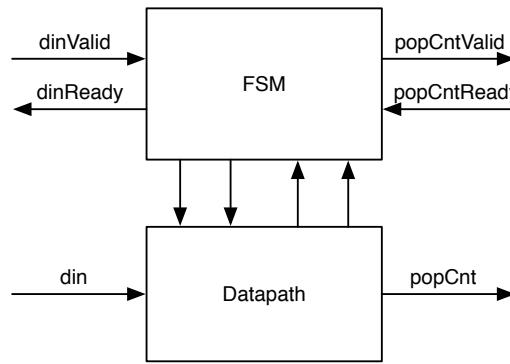


Figure 9.3: A state machine with a datapath.

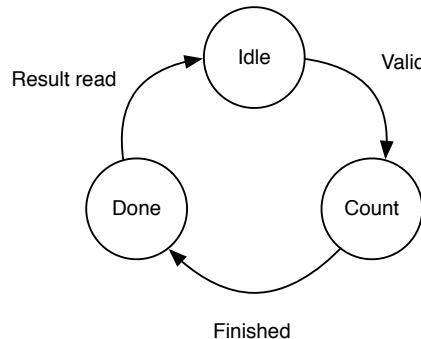


Figure 9.4: State diagram for the popcount FSM.

せは、多くの場合、データパス（FSMD）と有限状態マシンと呼ばれています。状態マシンは、データパスを制御し、データパスは、演算を行います。FSMの入力は、環境からの入力およびデータパスから入力されます。環境からのデータは、データパスに供給され、データ出力がデータパスから来ています。図～9.3は、データパスとFSMの組み合わせの一例を示しています。

9.2.1 ポップカウントの例 (L9242 TODO)

図～9.3に示すFSMDもHamming weight呼ばPOP COUNTを、計算例として機能します。ハミング重みは、ゼロシンボルとは異なるシンボルの数です。バイナリ文字列の場合、これは‘1’の数です。

POPCOUNTユニットは、データ入力`din`と結果出力`popCount`、データ経路に接続された両方を含みます。入力と出力のために我々は準備ができる、有効なハンドシェイクを使用します。データが利用可能である場合には、有効なガサートされます。受信機がデータを受け入れることができるとき、それは準備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。ハンドシェーク信号は、FSMに接続されています。FSMは、データパスに向かっておよびデータパスからのステータス信号と制御信号とのデータパスが接続されています。

次のステップとして、我々は図～9.4に示した状態図、で始まる、FSMを設計することができます。我々は、状態Idle、入力のためのFSMが待機中で開始します。データが有効な信号で合図、到着すると、FSMは、シフトレジスタをロードするために状態Loadに移行します。FSMは、次の状態Countに進み、‘1’基の存在数を順次カウントします。我々は、計算を実行するために、シフトレジスタ、加算器、アキュムレータレジスタ、及びダウンカウンタを使用します。ダウンカウンタが0に達すると、我々が終了し、状態DoneにFSMが移動しています。そこPOPCOUNT値が消費される準備ができていること、有効な信号とFSM信号を。受信機からのレディ信号に、FSMの移動は、次POPCOUNTを計算する準備ができる、Idle状態に戻り。

リスト～9.3に示す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、バルク接続とそれらを接続します。

図～9.5はPOPCOUNT回路のため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。負荷にもcntレジスタが0にリセットされます。‘1’つの数をカウントするには、shfレジスタは右にシフトし、最下位ビットは、各クロックサイクルcntに追加されます。すべてのビットが最下位ビットを介

```

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

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

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

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

```

Listing 9.3: The top level of the popcorn circuit.

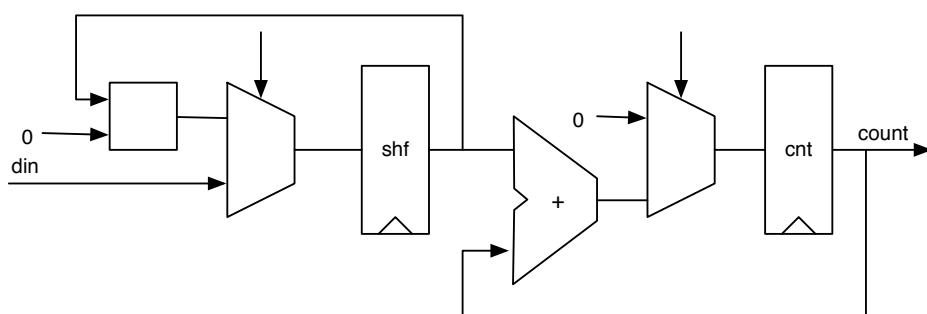


Figure 9.5: Datapath for the popcorn circuit.

```

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

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

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

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

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

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

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

```

Listing 9.4: Datapath of the popcount circuit.

してシフトされるまで、図には示されていないカウンタは、カウントダウン。カウンタが0に達すると、`POPCOUNT`が完了しました。FSMは状態`Done`及び信号に`popCntReady`をアサートすることによって結果を切り替えます。結果が読み取られると、FSWはバック`Idle`に切り替え`popCntValid`をアサートすることによって合図。

`load`信号に、`regData`レジスタが0に`regPopCount`レジスタリセット入力がロードされ、カウンタは、実行されるシフトの数に`regCount`セットを登録します。

そうでなければ、`regData`レジスタは、右`regData`の最下位ビットが0になるまで`regPopCount`レジスタに添加し、そしてカウンタをデクリメントレジスタにシフトされます。カウンタが0の場合、出力は`POPCOUNT`が含まれています。一覧～9.4は`POPCOUNT`回路のデータパスのためのChiselコードを示します。

FSMは状態`idle`で起動します。入力データの有効信号（`dinValid`）には`count`状態に切り替わり、データパスまで待機は、完成したカウントを有します。`POPCOUNT`が有効である場合、FSMは状態`done`に切り替わり、`POPCOUNT`まで待機を読み取る（`popCntReady`によって合図されます）。一覧～9.5は、FSMのコードを示します。

9.3 Ready-Valid インターフェース (L9462 TODO)

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に一般化することができます。ポップカウントの例では、有効信号とレディ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。

即時有効なインターフェイス～[3, p. 480]送信側で`data`と`valid`信号からなるインターフェースシンプルフロー制御（生産者）と受信側（消費者）に`ready`信号です。図～9.6は、すぐ有効な接続を示しています。

```

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
            }
        }
    }
}

```

Listing 9.5: The FSM of the popcount circuit.

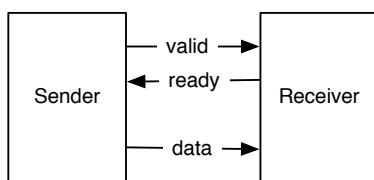


Figure 9.6: The ready-valid flow control.

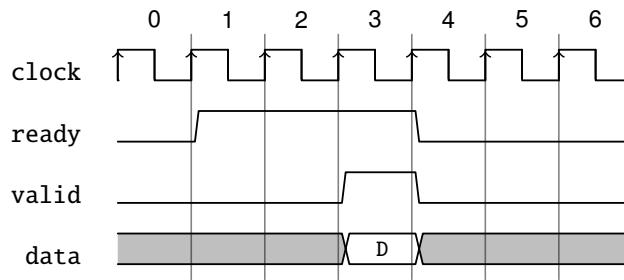


Figure 9.7: Data transfer with a ready-valid interface, early ready

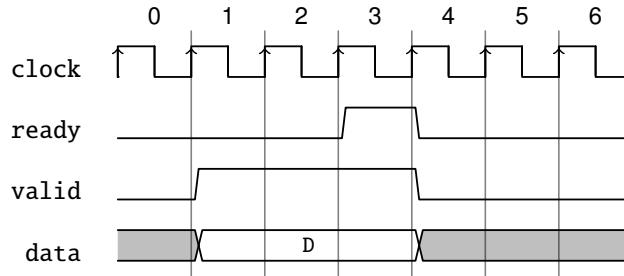


Figure 9.8: Data transfer with a ready-valid interface, late ready

dataが利用可能な場合、送信者はvalidをアサートし、1ワードのデータを受信する準備ができているとき、受信機はreadyを主張します。両方の信号、valid及びreadyが、アサートされたときにデータの送信が起こります。2つの信号のいずれかがアサートされていない場合、転送は行われません。

図～9.7は、送信者がデータを有する前に、受信機は、（上のクロックサイクル1から）readyに信号をすぐ有効な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において起こります。送信者どちらのクロックサイクル4からのデータを持っていたり、受信機は、次の転送の準備ができます。受信機は、クロック・サイクルごとにデータを受信できた場合、“常に準備ができる”インターフェースとreadyがtrueにハードコードすることが可能と呼ばれています。

図～9.8は、受信前（上のクロックサイクル1から）送信側信号validの準備ができてすぐ有効な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において起こります。送信者どちらのクロックサイクル4からのデータを持っていたり、受信機は、次の転送の準備ができます。“常に準備ができる”常に我々が想像できるインターフェイスと有効なインターフェイスに似ています。ただし、その場合にはデータは、おそらくreadyのシグナリングに変更されませんし、我々は単にハンドシェイク信号をドロップします。

図～9.8は、すぐ有効なインターフェイスの更なる変化を示しています。クロックサイクル1において、両信号（readyとvalidはちょうど单一のクロック・サイクルの間アサートとD1のデータ転送が起こるとなります。D2とD3の転送にクロックサイクル4および5に示すように、データは、バックツーバック（クロックサイクル毎に）に転送することができます

このインターフェースは、構成可能なようにするにはどちらもないreadyはvalidは、他の信号に組み合

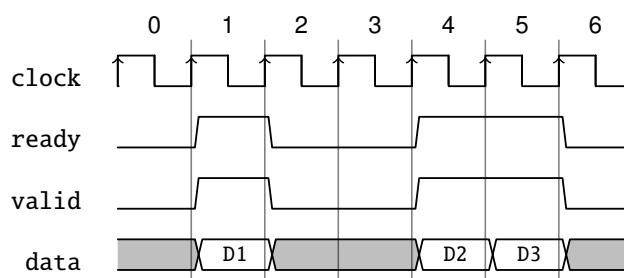


Figure 9.9: Single cycle ready/valid and back-to-back trasnfers

わせ依存するように許可されません。このインターフェースは、一般的であるように、 Chiselは、 次のようなDecoupledIOバンドルを定義します。

```
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。 Chiselによって定義されたインターフェースは、 データのフィールドbitsを使用しています。

readyまたはvalidがアクティブとnoデータ転送された後にディアサートされる場合は、 1つ疑問が残るが起こっています。 例えば、 受信機は、 いくつかの時間のために準備し、 データを受信していないかもしれません。 いくつかの他のイベントのために準備ができていないことがあります。 同じことが、 データ転送せずに一部だけクロッククロックサイクルとなってきて非有効な有効なデータを持つ、 送信者と想定することができます。 場合は、 この動作を許可するかの準備ができていません - 有効なインターフェイスの一部ではなく、 インターフェースの具体的な使用方法で定義する必要があります。

クラスDecoupledIOを使用した場合Chiselはreadyとvalidのシグナリングに何の要件を置きません。 しかし、 クラスIrrevocableIOは、 送信者に次のような制限を課します。

約束はvalidが高く、 readyが低いサイクル後bitsの値を変更しないために、 そのReadyValidIOの具象サブクラス。 validを上昇させた後さらに、 それはreadyも提起された後今まで低下することはありません。

これはクラスIrrevocableIOを使用して実施することができない規則であることに注意してください。

IrrevocableIO.

読み出しアドレスデータ、 書き込みアドレス、 書き込みデータを読み込む： AXIバスの以下の部分ごとに1つの既製有効なインターフェイスを使用しています。 AXIはreadyまたはvalidがアサートされると、 データ転送が起こったまでデアサート取得するために許可されていないこと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 (L9245 TODO)

Chiselの強さは、それが、私たちは、いわゆるハードウェアジェネレータを書くことができることです。そのようなVHDLおよびVerilogなどの古いハードウェア記述言語、で、我々は通常、別の言語、Eを使用しています。グラム。、JavaやPythonの、ハードウェアを生成します。著者は、多くの場合、VHDLテーブルを生成するために、小さなJavaプログラムを書いています。Chiselでは、スカラ（及びJavaライブラリー）のフルパワーは、ハードウェア構成で利用可能です。したがって、我々は同じ言語で、当社ハードウェアジェネレータを書いて、Chisel回路生成の一部としてそれらを実行することができます。

10.1 パラメータを使って設定する (L8932 TODO)

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 (L9816 TODO)

回路をパラメータ化するための基本的な方法は、パラメータとしてビット幅を定義することです。パラメータは、Chiselモジュールのコンストラクタの引数として渡すことができます。例に続いて、設定ビット幅の加算器を実装モジュールのおもちゃの一例です。n幅ビットがIOバンドルに使用することができるコンストラクタに渡された成分の（スカラ型Intの）パラメータです。

```
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
}
```

次のように加算器のパラメータ化されたバージョンを作成することができます。

```
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 (L9865 TODO)

設定パラメータとしてビット幅を有するハードウェア発生のためだけの出発点です。非常に柔軟な構成は、型の使用です。Chiselは、多重化のための任意のタイプを受け入れることができます。マルチプレクサ(Mux)を提供するためにその機能が可能となります。コンフィギュレーションのための型を使用する方法を示すために、我々は、任意のタイプを受け入れるマルチプレクサを構築します。次の関数は、マルチプレクサを定義します。

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

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

ChiselはChiselタイプと我々の場合には、型をパラメータ化機能を可能にします。角括弧内の式[T <: Data]定義型TセットパラメータはData又はDataのサブクラスです。DataはChisel型システムのルートです。

ブール条件、真のパスに1つのパラメータ、および偽のパスのための一つのパラメータ：当社のマルチプレクサ機能は、次の3つのパラメータがあります。両方のパスパラメータがタイプT、関数呼び出しで提供される情報です。機能自体は単純です：私たちはfPathのデフォルト値を持つ線を定義し、条件がtPathに真である場合には、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。関数の最後に、我々は、マルチプレクサのハードウェアを返します。

私たちは、このようなUIntのような単純なタイプで、当社のマルチプレクサ機能を使用することができます。

```
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)
```

我々は2つのフィールドを持つBundleとして私たちのタイプを定義します。

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

私たちは、最初のWireを作成し、サブフィールドを設定することにより、Bundle定数を定義することができます。その後、我々は、この複合型で、当社のパラメータ化マルチプレクサを使用することができます。

```
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 multiplexer with a complex type
val resB = myMux(selB, tVal, fVal)
```

機能の私たちの最初の設計では、デフォルト値を持つタイプTと線を作成するためにwireDefaultを使用しました。私たちは、デフォルト値を使用せずに、単にChisel型のワイヤーを作成する必要がある場合、我々はChiselタイプを取得するためにfPath.cloneTypeを使用することができます。機能ショー、次のマルチプレクサをコーディングする別の方法。

```
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 タイプパラメータを持つモジュール (10036 TODO)

また、Chiselタイプとモジュールのパラメータを設定することができます。私たちは別の処理コアの間でデータを移動するために、ネットワーク・オン・チップを設計したいと仮定しましょう。しかし、我々は、ルータインターフェイスでのデータ形式をハードコーディングする必要はありません。我々はparameterizeそれにしたいです。関数の型パラメータと同様に、我々は、モジュールのコンストラクタにTパラメータタイプを追加します。さらに、我々はそのタイプの1つのコンストラクタのパラメータを持

っている必要があります。また、この例では、我々はまた、ルータのポート数を設定可能にします。

```
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
    // ...
}
```

私たちのルータ、ルートに私たちが望むデータ型を定義するために、私たち最初の必要性、電子を使用します。グラム。、ChiselBundleとして：

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

ルータ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 (L9225 TODO)

アドレスのための1つおよびパラメータ化されたデータのためのいずれかのルータの例では、ルータの入力用フィールドの二つの異なるベクターを使用します。よりエレガントな解決策は、それ自体がパラメータ化されていることをBundleを持っていることであろう。何かのようなもの：

```
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タイプTのパラメータを持っています。バンドル内では、我々はパラメータにcloneTypeを呼び出すことによって、フィールドdataを定義します。私たちは、コンストラクタのパラメータを使用する場合ただし、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はBundle、Eのタイプのクローンを作成する必要がある場合。グラム。それはVecで使用されている場合、このパブリックフィールドが道です。この問題の解決策（回避策）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
}
```

その新しいBundleでは、我々は、ルータのポートを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
    // ...
}
```

パラメータとしてPayloadを取るPortでそのルータをインスタンス化します。

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

```

import chisel3._
import scala.io.Source

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

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

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

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

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

```

Listing 10.1: Reading a text file to generate a logic table.

10.2 組合せ論理回路の生成 (L10203 TODO)

Chiselでは、我々は簡単にScalaのArrayからChiselVecとの論理表を作成することにより、ロジックを生成することができます。我々は、論理テーブルのためのハードウェア生成時間の間で読むことができることを、ファイル内のデータを持っているかもしれません。一覧～10.1ショードのようにファイル“データを読み込むためのScala標準ライブラリのフォームのScala Sourceクラスを使用します。テキスト表現の定数整数含まTXT「」。

多分少し威圧的な表現上のいくつかの単語：RRR000DfcAcbmE

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

AスカラArrayは、暗黙的に写像関数mapをサポートしている配列（Seq）に変換することができます。mapは、配列の各要素に対して関数を呼び出し、関数の戻り値のシーケンスを返します。我々の機能_.U(8.W)は_としてScalaのアレイからの各Int値を表し、8ビットのサイズで、ChiselUIntのリテラルへスカラRRR011o9mPLVVq値から変換を行います。ChiselオブジェクトVecInitは、ChiselタイプのシーケンスSeqからChiselVecを作成します。

Scalaのパワーをフルに使って、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指数定数の表を生成したり、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の小さなアセンブラー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、[binary-coded decimal \(BCD\)](#) 表現に進数の変換です。BCDは、各進数字の4ビットを使用して10進形式で番号を表すために使用されます。例えば、小数13バイナリ1101であり、BCDはバイナリ1及び3としてエンコード：00010011。BCDは小数で表示数、進よりユーザーフレンドリーな数の表現を可能にします。

バイナリをBCDに変換するための表を計算するJavaプログラムを書けます。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、我々はハードウェアの世代の一部として直接この表を計算することができます。一覧

```

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)
}

```

Listing 10.2: Binary to binary-coded decimal conversion.

```

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

    val N = (n-1).U

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

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

    io.tick := cntReg === N
}

```

Listing 10.3: Tick generation with a counter.

～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 繙承を利用する (L10372 TODO)

Chiselは、オブジェクト指向言語です。ハードウェアコンポーネントは、ChiselModuleはScalaのクラスです。したがって、我々は、親クラスに要因に共通の動作を継承して使用することができます。ここでは、例と継承を使用する方法を探ります。

節～6.2では、低周波ダニの生成に使用することができるカウンターの異なる形態を模索しています。私たちは、これらの異なるバージョン、電子を探検したいと仮定しましょう。グラム。、そのリソース要件を比較します。私たちは、刻々と過ぎインターフェースを定義する抽象クラスで起動します。

```

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

```

一覧～10.3は、ダニの発生のために、カウントアップ、カウンタとその抽象クラスの最初の実装を示しています。

```

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

class TickerTester[T <: Ticker](dut: T, n: Int) extends PeekPokeTester(dut: T) {

    // -1 is the notion that we have not yet seen the first tick
    var count = -1
    for (i <- 0 to n * 3) {
        if (count > 0) {
            expect(dut.io.tick, 0)
        }
        if (count == 0) {
            expect(dut.io.tick, 1)
        }
        val t = peek(dut.io.tick)
        // On a tick we reset the tester counter to N-1,
        // otherwise we decrement the tester counter
        if (t == 1) {
            count = n-1
        } else {
            count -= 1
        }
        step(1)
    }
}

```

Listing 10.4: A tester for different versions of the ticker.

我々は、単一のテストベンチで私たちのtickerロジックのすべての異なるバージョンをテストすることができます。我々Tickerのサブタイプを受け入れるようにテストベンチを定義する必要がjust。～10.4をリストテスター用Chiselコードを示します。(1) Ticker又はTまたはそのサブタイプであるTickerから継承する任意のクラス、(2) テスト中の設計、およびクロックの(3) 数を受け入れる[T <: Ticker]パラメータタイプ: TickerTesterは、いくつかのパラメータを有しますサイクルは、我々はそれぞれの目盛りのために期待しています。tickはすべてNクロックサイクルを繰り返すことティック(開始は異なる実装のために異なるかもしれない)、次いで、チェックの最初の発生のためにテスターを待ちます。

ティッカーの最初の、簡単な実装では、我々は、おそらくいくつかのprintlnのデバッグで、テスター自身をテストすることができます。私たちは、単純なティッカーやテスターが正しいことを確信しているとき、私たちは、ティッckerの2つのバージョンを続行して探索することができます。一覧～10.5は0までカウントダウンカウンタでダニの生成を示しています。一覧～10.6は、コンパレータを回避することにより、より少ないハードウェアを使用することに-1カウント・ダウのオタクのバージョンを示しています。

私たちは、Scalatest仕様を使用してティッcker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cker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。～10.7のリストの仕様を示しています。

```
sbt "testOnly TickerSpec"
```

10.4 関数型プログラミングによるハードウェア生成 (L19523 TODO)

Scalaは関数型プログラミングをサポートしていますので、Chiselは、ありません。我々は、ハードウェアを表現する関数を使用して(「」いわゆる“高次関数を用いて”)、機能プログラミングと、これらのハードウェアコンポーネントを組み合わせることができます。私たちは、簡単な例で、ベクトルの総和を始めましょう：

```

def add(a: UInt, b: UInt) = a + b
val sum = vec.reduce(add)

```

```

class DownTicker(n: Int) extends Ticker(n) {

    val N = (n-1).U

    val cntReg = RegInit(N)

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

    io.tick := cntReg === N
}

```

Listing 10.5: Tick generation with a down counter.

```

class NerdTicker(n: Int) extends Ticker(n) {

    val N = n

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

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

```

Listing 10.6: Tick generation by counting down to -1.

```

class TickerSpec extends FlatSpec with Matchers {

    "UpTicker 5" should "pass" in {
        chisel3.iotesters.Driver(() => new UpTicker(5)) { c =>
            new TickerTester(c, 5)
        } should be (true)
    }

    "DownTicker 7" should "pass" in {
        chisel3.iotesters.Driver(() => new DownTicker(7)) { c =>
            new TickerTester(c, 7)
        } should be (true)
    }

    "NerdTicker 11" should "pass" in {
        chisel3.iotesters.Driver(() => new NerdTicker(11)) { c =>
            new TickerTester(c, 11)
        } should be (true)
    }
}

```

Listing 10.7: ScalaTest specifications for the ticker tests.

まず、機能addにおける加算器のハードウェアを定義します。ベクトルはvecに位置しています。スカラreduce()方法は、单一の値を生成する、バイナリ操作でコレクションのすべての要素を組み合わせました。reduce()方法は、左から順に減少します。これは、最初の2つの要素とを行う操作を要します。单一の結果が残されるまで結果は、その後、次の要素と組み合わされます。

要素に結合する機能は、我々の場合、加算器を返しaddに、reduceにパラメータとして提供されます。得られたハードウェアは、ベクトルvecの要素の和を計算する加算器のチェーンです。

代わりに（簡単な）add関数を定義するので、私たちは、無名関数として加算を提供し、2つのオペランドを表すために、Scalaのワイルドカード“_「」を使用することができます。

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

この1つのライナーで、私たちは、加算器のチェーンを生成しました。チェーンは、理想的な構成ではありませんSUM関数の場合は、ツリーが短い組み合わせ遅延を持つことになります。我々は、加算器チェーンを再配置するのsynthesizeツールを信用しない場合、我々は、加算器のツリー生成にChiselのreduceTreeメソッドを使用することができます。

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

11 デザイン例 (L10642 TODO)

このセクションでは、以下のようないくつかの小さなサイズのデジタルデザインを探索します。これらは、より大きな設計のためのビルディングブロックとして使用されます。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) を設計しますが、これ自体はFIFOバッファを使用します。

11.1 FIFO バッファ (L10098 TODO)

我々は、ライターとリーダーとの間のバッファによるライト (送信者) とリーダ (受信機) を切り離すことができます。EN: 共通バッファは、(HREF HTTPS先入れ先出しです。 ウィキペディア。 ERROR-TBD 図～11.1は作家、FIFO、およびリーダーを示しています。データは、アクティブwrite信号とdinの作家によってFIFOに入れられます。データは、アクティブread信号にdoutにリーダによってFIFOから読み出されます。

A FIFOはempty信号によって選抜、最初は空です。空のFIFOからの読み取りは、通常は定義されていません。データが書き込まれたときとfullなりFIFOを読んだことがありません。フルFIFOへの書き込みは、通常は無視され、データが失われます。換言すれば、信号emptyとfullはハンドシェーク信号として働きます

FIFOの幾つかの異なる実装が可能である：E. グラム。、オンチップ・メモリを使用してポインタまたは単に小さなステートマシンを有するレジスタのチェーンを読み書きします。小さなバッファ（要素の数十まで）のためのFIFOは、低リソース要件を持つ単純な実装であるバッファのチェーンに接続された個々のレジスタを整理しました。バブルFIFOのコードはchisel-examplesリポジトリに入手可能です。ERROR-TBD

We start by defining the IO signals for the writer and the reader side. The size of the data is configurable with size. The write data are din and a write is signaled by write. The signal full performs the flow control at the writer side.

私たちは、作家と読者側のためのIO信号を定義することから始めます。データのサイズがsizeで設定可能です。書き込みデータはdinあると書き込みがwriteによって通知されます。信号fullライター側にflow control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))
}
```

読者側はdoutとデータを提供し、リードreadで開始されます。empty信号は、リーダ側で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))
}
```

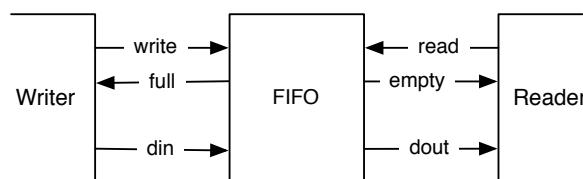


Figure 11.1: A writer, a FIFO buffer, and a reader.

```

class FifoRegister(size: Int) extends Module {
    val io = IO(new Bundle {
        val enq = new WriterIO(size)
        val deq = new ReaderIO(size)
    })

    val empty :: full :: Nil = Enum(2)
    val stateReg = RegInit(empty)
    val dataReg = RegInit(0.U(size.W))

    when(stateReg === empty) {
        when(io.enq.write) {
            stateReg := full
            dataReg := io.enq.din
        }
    }.elsewhen(stateReg === full) {
        when(io.deq.read) {
            stateReg := empty
            dataReg := 0.U // just to better see empty slots in the waveform
        }
    }.otherwise {
        // There should not be an otherwise state
    }

    io.enq.full := (stateReg === full)
    io.deq.empty := (stateReg === empty)
    io.deq.dout := dataReg
}

```

Listing 11.1: A single stage of the bubble FIFO.

～11.1リストは、単一のバッファを示しています。バッファは、エンキューポートタイプWriterIOのenq及びタイプReaderIOのデキューポートdeqを有しています。バッファの状態要素は、データを保持する一つのレジスタ（dataReg及び単純FSM（stateReg）に対して1つの状態レジスタです。FSMは、2つの状態しかあり：バッファのいずれかがemptyかfullです。バッファがemptyであれば、書き込みがfull状態への入力データと変更を登録します。バッファがfullある場合は、読み取りはempty状態へのデータおよび変更を消費します。IOポートfullとemptyライターとリーダーのためのバッファの状態を表します。

一覧～11.2は、完全なFIFOを示します。完全なFIFOは、個々の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バッファ段数のパラメータデータワードとdepthのsizeとして有しています。私たちは、depth depth FifoRegistersのうち、バブルFIFOステージを構築することができます。私たちは、クレートScalaのArrayにそれらを充填することによりステージを。Scalaの配列は何もハードウェアの意味を持っていない、それが作成したバッファへの参照を持っているコンテナを提供してくれますjust。Scalaのforループでは、個々のバッファを接続してください。最初のバッファのエンキュー側は、完全なFIFOのデキュー側に完全なFIFOのエンキューIOと最後のバッファのデキュー側に接続されています。

データがキューを介して気泡としてFIFOキューを実装するために、個々のバッファを接続の提示アイデアは、バブルFIFOと呼ばれています。これは簡単で、優れたソリューションは、データ・レートは遅いクロック・レート、電子よりもかなりあるとき。グラム。、次のセクションで提示されているシリアルポート用デカップルバッファとして。

各バッファの状態がFIFOの最大スループットは、ワード当たり2回のクロックサイクルであることを意味する、emptyとfullを切り替えなければならないように (1) : データレートはクロック周波数に近づくとしかしながら、気泡FIFOは2つの制限を有します。 (2) 完全なFIFOを介してバブルへのデータの必要性は、従って、入力から出力までの待ち時間は、少なくともバッファの数です。私は節～11.3にFIFOの他の可能な実装を紹介します。

11.2 シリアルポート (L10368 TODO)

(またUARTまたはRS-232と呼ばれる) のシリアルポートは、ノートパソコンとFPGAボードの間で通信す

```

class BubbleFifo(size: Int, depth: Int) extends Module {
    val io = IO(new Bundle {
        val enq = new WriterIO(size)
        val deq = new ReaderIO(size)
    })

    val buffers = Array.fill(depth) { Module(new FifoRegister(size)) }
    for (i <- 0 until depth - 1) {
        buffers(i + 1).io.enq.din := buffers(i).io.deq.dout
        buffers(i + 1).io.enq.write := ~buffers(i).io.deq.empty
        buffers(i).io.deq.read := ~buffers(i + 1).io.enq.full
    }
    io.enq <> buffers(0).io.enq
    io.deq <> buffers(depth - 1).io.deq
}

```

Listing 11.2: A FIFO is composed of an array of FIFO bubble stages.

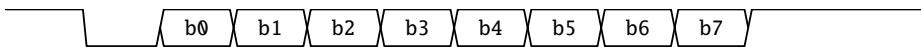


Figure 11.2: One byte transmitted by a UART.

るための最も簡単な選択肢の一つです。名前が示すように、データがシリアル伝送されます。8ビットのバイトは次のように送信される：1スタートビット (0) は、最初の8ビットのデータ、最下位ビット、および1つのまたは2つのストップビット (1)。何もデータが送信されない場合、出力は1です。図～11.2は、送信された1バイトのタイミング図を示します。

私たちは、モジュールごとに、最小限の機能を備えたモジュール式の方法で私たちのUARTを設計します。我々は、送信機 (TX)、受信機 (RX)、緩衝液、及びそれらのベースコンポーネントの使用を提示します。

まず、我々はインターフェイス、ポート定義を必要としています。UARTの設計のために、我々は送信機から見られるように方向で、準備/有効なハンドシェイク・インターフェースを使用しています。準備/有効なインターフェイスの大会は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ていることです。

一覧～11.3は、ペアボーンシリアル送信 (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、および送信機がシリアル化し送信するために文字を受信することができるChannelあります。正しいタイミングを生成するために、我々は1つのシリアルビットのためのクロック・サイクルの時間を計算することによってための定数を計算します。

我々は、3つのレジスタを使用する：(1) (シリアルライズして) (shiftReg)、(2) 正しいボーレート (cntReg) を生成するカウンタ、及びビット数依然として必要 (3) カウンタデータをシフトするレジスタシフトアウトされます。FSMの追加の状態レジスタを必要としない、すべての状態は、これらの3つのレジスタでエンコードされています。

カウンタcntRegが連続的に実行されている（開始値0に0、リセットにカウントダウン）。cntRegが0である場合、すべてのアクションにのみ行われます。私たちは、最小限の送信機を構築するように、我々は、データを格納するための唯一のシフトレジスタを持っています。cntRegが0でないビットがシフトアウトするために残っていない場合したがって、チャネルのみ準備ができます。

IOポートtxd直接シフトレジスタの最下位ビットに接続されています。

(bitsReg /= 0.U) をシフトアウトするためのより多くのビットがある場合、我々は、トップ（送信機のアイドルレベル）から1と右充填ビットをシフトします。これ以上のビットがシフトアウトする必要がない場合は、チャネルが（validポートで合図）のデータが含まれている場合、我々は確認してください。もしそうであれば、シフトアウトされるビット列は、1スタートビット (0)、8ビットのデータ、及び2つのストップビット (1) を用いて構成されています。したがって、ビット数は11に設定されています。

この非常に最小限の送信には、追加のバッファを持っていないとcntRegが0であるとき、シフトレジスタが空とクロック・サイクルである場合にのみ、新しいキャラクターを受け入れることができます。cntRegが0である場合にのみ、新しいデータを受け入れると、シフトレジスタにスペースが存在することになる準備ができたらフラグもデアサートであることを意味します。しかし、我々は、送信機に、この“複雑”を追加しますが、バッファにそれを委任する必要はありません。

```

class Tx(frequency: Int, baudRate: Int) extends Module {
    val io = IO(new Bundle {
        val txd = Output(Bits(1.W))
        val channel = new Channel()
    })

    val BIT_CNT = ((frequency + baudRate / 2) / baudRate - 1).asUInt()

    val shiftReg = RegInit(0x7ff.U)
    val cntReg = RegInit(0.U(20.W))
    val bitsReg = RegInit(0.U(4.W))

    io.channel.ready := (cntReg === 0.U) && (bitsReg === 0.U)
    io.txd := shiftReg(0)

    when(cntReg === 0.U) {

        cntReg := BIT_CNT
        when(bitsReg /= 0.U) {
            val shift = shiftReg >> 1
            shiftReg := Cat(1.U, shift(9, 0))
            bitsReg := bitsReg - 1.U
        }.otherwise {
            when(io.channel.valid) {
                // two stop bits, data, one start bit
                shiftReg := Cat(Cat(3.U, io.channel.data), 0.U)
                bitsReg := 11.U
            }.otherwise {
                shiftReg := 0x7ff.U
            }
        }
    }

    }.otherwise {
        cntReg := cntReg - 1.U
    }
}

```

Listing 11.3: A transmitter for a serial port.

一覧～??はバブルFIFOのためのFIFOレジスタに似た单一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あり、出力が反転方向とChannelインターフェースです。バッファはemptyまたはfullを示すために、最小限のステートマシンが含まれています。バッファ駆動ハンドシェーク信号(in.readyとout.valid)は、状態レジスタに依存します。

状態はemptyで、入力上のデータがvalidで、我々は状態fullにデータとスイッチを登録するとき。状態がfullであり、そして下流受信機はreadyである場合、ダウンストリームのデータ転送が発生し、そして我々は、状態emptyに戻します。

そのバッファで、私たちは私たちの裸の骨の送信機を拡張することができます。リストイング～??は前に单一のバッファと送信Txの組み合わせを示します。このバッファは、今Txは、単一のクロック・サイクルのためにreadyだったという問題を緩和します。私たちは、バッファモジュールに、この問題の解決策を委任しました。実際のFIFOへの單一ワードバッファの拡張が簡単に行われ、送信または单一バイトのバッファには変化を必要としないことができます。

～??をリストレシーバー(Rx)のコードを示します。それは、シリアルデータのタイミングを再構成する必要があるように、受信機は、少しトリッキーです。受信機は、スタートビットの立ち下がりエッジを待ちます。そのイベントから、受信機は、1を待ちます。5ビット時間は、ビット0の中央に自身を位置決めします。そして、それはビットのすべてのビット時間をシフト。あなたはSTART_CNTとBIT_CNTのように、これらの2つの待ち時間を観察することができます。両方回、同じカウンタ(cntReg)が使用されます。8ビットがシフトインされた後、valRegは、利用可能なバイトを通知します

一覧～??はフレンドリーメッセージアウトを送信することにより、シリアルポートの送信機の使い方を示しています。我々は、Scalaの列(msg)にメッセージを定義し、UIIntのChiselVecに変換します。Scalaの文字列はmapメソッドをサポートしているシーケンスです。mapメソッドは、引数として関数リテラルを取り、各要素に、この機能を適用し、関数の戻り値のシーケンスを構築します。関数リテラルは引数を1つしか持たなければならない場合は、このケースであるように、引数が_で表すことができます。私たちの関数リテラルはChiselUIIntにScalaのCharを変換するために、掘削方法.uを呼び出します。シーケンスは、その後ChiselVecを構築するためにVecInitに渡されます。バッファリング送信機に個々の文字を提供するために、カウンターとベクトルtext cntRegに私たちはインデックス。完全な文字列が送信されるまで、各ready信号によって、我々は、カウンタを増加させます。送信者は、最後の文字が送信されるまでvalidがアサートし続けます。

一覧～??は、受信機と一緒にそれらを接続することにより、送信機の使い方を示しています。この接続は、各受信された文字が送り返されるEcho回路を生成する(エコー)。

11.3 FIFO設計のバリエーション (L11322 TODO)

このセクションでは、FIFOキューのさまざまなバリエーションを実装します。節～10.3で導入され、これらの実装は、私たちが継承を使用する交換可能にします。

11.3.1 FIFOのパラメータ化 (L1135 TODO)

我々は、任意のChiselデータタイプをバッファリングすることができるようパラメータとしてChisel型とabstract FIFOクラスを定義します。抽象クラスでは、我々はまた、パラメータdepthが有用価値を持っていることをテストします。

```
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")
}
```

節～11.1では、我々は、このようなwrite、full、din、read、RRR011dqUNBk4r、およびRRR011eBUalVC9などの信号のための一般的な名前、とのインターフェースのために私たち自身のタイプを定義しました。入力などAバッファの出力は、(例えばハンドシェイクするためのデータと二つの信号から成ります。グラム。それはfullないとき、私たちは、FIFOにwrite。

しかし、我々は、いわゆる既製有効なインターフェイスにこのハンドシェイクを一般化することができます。え。G FIFOがreadyある場合、我々は、要素(FIFOへの書き込み)をエンキューすることができます。私たちは、validとライタ側でこれを合図します。この既製有効なインターフェイスは非常に一般的であるように、以下のように、ChiselはDecoupledIOで、このインターフェースの定義を提供する：ERROR-TBD

```

class BubbleFifo[T <: Data](gen: T, depth: Int) extends Fifo(gen: T, depth: Int) {

    private class Buffer() extends Module {
        val io = IO(new FifoIO(gen))

        val fullReg = RegInit(false.B)
        val dataReg = Reg(gen)

        when (fullReg) {
            when (io.deq.ready) {
                fullReg := false.B
            }
        } .otherwise {
            when (io.enq.valid) {
                fullReg := true.B
                dataReg := io.enq.bits
            }
        }

        io.enq.ready := !fullReg
        io.deq.valid := fullReg
        io.deq.bits := dataReg
    }

    private val buffers = Array.fill(depth) { Module(new Buffer()) }
    for (i <- 0 until depth - 1) {
        buffers(i + 1).io.enq <> buffers(i).io.deq
    }

    io.enq <> buffers(0).io.enq
    io.deq <> buffers(depth - 1).io.deq
}

```

Listing 11.4: A bubble FIFO with a ready-valid interface.

```

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

```

enqエンキューとFifoIOし、読み取り有効なインターフェースからなるdeqデキュー・ポート：DecoupledIOで、私たちは私たちのFIFOのためのインターフェースを定義しますインターフェイス。DecoupledIOインターフェースは、作家の（生産者）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信号方向を反転する必要があります。

```

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

```

抽象基底クラスとインターフェースで、私たちは、異なるパラメータ（速度、面積、消費電力、または単にシンプル）用に最適化された別のFIFOの実装のために特化することができます。

11.3.2 バブルFIFOの再設計 (L11471 TODO)

我々は、標準的な既製の有効なインターフェースを使用してChiselデータ型でパラメータ化され、セクション～11.1から私たちのバブルFIFOを再定義することができます。

一覧～11.4は、すぐ有効なインターフェイスを持つリファクタリングバブルFIFOを示しています。注私

たちは、プライベートクラスとしてBubbleFifoからBuffer部品の内部を置きます。このヘルパークラスは、このコンポーネントのために必要とされているので、我々はそれを隠し、名前空間を汚染しないようにします。バッファクラスも簡素化されました。代わりFSM電子用途の单一ビット、fullRegは、バッファの状態注：フルまたは空。

バブルFIFOは、単純に理解しやすく、最小限のリソースを使用しています。各バッファ段が空と満杯を切り替えなければならないしかし、このFIFOの最大帯域幅は、ワード当たり2回のクロックサイクルです。

一つはプロデューサーvalidと消費者がreadyときに新しい単語を受け入れることができますように、バッファの両方のインターフェイス側を見て検討することができます。しかし、これは、すぐ有効なプロトコルのセマンティクスに違反プロデューサーの握手に消費者の握手から組み合わせパスを導入しています。

11.3.3 ダブルバッファ FIFO (L11554 TODO)

完全な場合は、バッファが登録しても一つの解決策は、滞在readyです。消費者は、我々は第2のバッファを必要としreadyない場合には、生産者からのデータワードを受け入れることができますようにするために、我々は、シャドウ・レジスタと呼んでいます。バッファがいっぱいになると、新しいデータは、シャドウ・レジスタに格納され、readyはデアサートされます。消費者が再びreadyになるとデータレジスタにデータが消費者にシャドウレジスタからレジスタから、データが転送されます。

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

        val empty :: one :: two :: Nil = Enum(3)
        val stateReg = RegInit(empty)
        val dataReg = Reg(gen)
        val shadowReg = Reg(gen)

        switch(stateReg) {
            is (empty) {
                when (io.enq.valid) {
                    stateReg := one
                    dataReg := io.enq.bits
                }
            }
            is (one) {
                when (io.deq.ready && !io.enq.valid) {
                    stateReg := empty
                }
                when (io.deq.ready && io.enq.valid) {
                    stateReg := one
                    dataReg := io.enq.bits
                }
                when (!io.deq.ready && io.enq.valid) {
                    stateReg := two
                    shadowReg := io.enq.bits
                }
            }
            is (two) {
                when (io.deq.ready) {
                    dataReg := shadowReg
                    stateReg := one
                }
            }
        }
        io.enq.ready := (stateReg === empty || stateReg === one)
    }
}
```

```

    io.deq.valid := (stateReg === one || stateReg === two)
    io.deq.bits := dataReg
}

private val buffers = Array.fill((depth+1)/2) { Module(new DoubleBuffer(gen)) }

for (i <- 0 until (depth+1)/2 - 1) {
    buffers(i + 1).io.enq <> buffers(i).io.deq
}
io.enq <> buffers(0).io.enq
io.deq <> buffers((depth+1)/2 - 1).io.deq
}

```

Listing 11.5: A FIFO with double buffer elements.

一覧～11.5はダブルバッファを示しています。各バッファ素子として、我々はバッファ素子（ $\text{depth}/2$ ）の半分だけを必要とする2つのエントリを格納することができます。DoubleBuffer 2つのレジスタ、`dataReg`と`shadowReg`が含まれています。消費者は`shadowReg`から常に提供しています。`empty`、`one`、及び`two`、ダブルバッファの充填レベルを信号：ダブルバッファの3つの状態を有しています。バッファは、それが国家`empty`か`one`にあるときに、新しいデータを受け入れること`ready`です。それは状態`one`か`two`であるときに有効なデータを持っています。

我々はフルスピードでFIFOを実行して、消費者は常にダブルバッファの`ready`定常状態である場合`one`です。消費者の`ready`をデアサートした場合にのみ、キューがいっぱいになると、バッファは、状態`two`を入力してください。しかし、単一のバブルFIFOに比べて、キューの再起動が同じバッファ容量のためのクロック・サイクルFOのみ半分の数になります。レイテンシーによる同様の秋には、バブルFIFOの半分です。

11.3.4 レジスタメモリ付きFIFO (L11646 TODO)

あなたは、ソフトウェアエンジニアリングの背景に来るとき、あなたはすべてを並列に実行し、上流と下流の要素とハンドシェーク、我々は多くの小さな個々の小さなバッファ要素のうち、ハードウェアキューを構築していることを不思議に思っている可能性があります。小さなバッファの場合、これはおそらく最も効率的な実装です。

ソフトウェアでのキューは通常、シングルスレッドでのシーケンシャルコードで使用されます。またはキューとして生産者と消費者のスレッドを分離します。この設定の固定サイズでFIFOキューは通常`circular buffer`として実装されています。二つのポインタは、キュー用に確保したメモリ内の読み取りと書き込みの位置にポイントします。ポインタはメモリの最後に到達すると、そのメモリの開始に背を設定されています。2つのポインタの違いは、キュー内の要素の数です。二つのポインタが同じアドレスをポイントすると、キューが空またはフルのいずれかです。空とフル区別するために、我々は別のフラグが必要です。

我々としても、ハードウェアでは、このようなAメモリベースのFIFOキューを実装することができます。小さなキューの場合、我々は、(iをレジスタ・ファイルを使用することができます。え。`Reg(Vec())`)。～11.6ショーリストFIFOメモリを用いて実装し、読み取りと書き込みのポインタキュー。

```

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

    def counter(depth: Int, incr: Bool): (UIInt, UIInt) = {
        val cntReg = RegInit(0.U(log2Ceil(depth).W))
        val nextVal = Mux(cntReg === (depth-1).U, 0.U, cntReg + 1.U)
        when (incr) {
            cntReg := nextVal
        }
        (cntReg, nextVal)
    }

    // the register based memory
    val memReg = Reg(Vec(depth, gen))

    val incrRead = WireDefault(false.B)
    val incrWrite = WireDefault(false.B)
    val (readPtr, nextRead) = counter(depth, incrRead)
    val (writePtr, nextWrite) = counter(depth, incrWrite)
}

```

```

val emptyReg = RegInit(true.B)
val fullReg = RegInit(false.B)

when (io.enq.valid && !fullReg) {
    memReg(writePtr) := io.enq.bits
    emptyReg := false.B
    fullReg := nextWrite === readPtr
    incrWrite := true.B
}

when (io.deq.ready && !emptyReg) {
    fullReg := false.B
    emptyReg := nextRead === writePtr
    incrRead := true.B
}

io.deq.bits := memReg(readPtr)
io.enq.ready := !fullReg
io.deq.valid := !emptyReg
}

```

Listing 11.6: A FIFO with a register based memory.

同じように動作し、アクションにインクリメントされると、バッファの最後で折り返すことに二つのポイントタがあるように、我々は実装し、それらのラッピングカウンターという関数counterを定義します。log2Ceil(depth).Wで、私たちは、カウンタのビット長を計算します。次の値のいずれか1によってインクリメント又は0にラップアラウンドします。カウンタは入力incrがtrue.Bあるときにのみインクリメントされます。

我々はまた、可能な次の値（増分またはラップの周りに0を）必要があるとして、さらに、我々としてもcounter関数からこの値を返します。スカラ座では、単に複数の値を保持するための容器であり、いわゆるtupleを、返すことができます。そのようなtupleは単にコンマをラップさせ作成するための構文は、括弧内の値を分離しました：

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

我々は、割り当ての左側の括弧表記を使用してそのようなタプルを分解することができます。

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

メモリについては、私たち私たちのベクトルのレジスタ (Reg(Vec(depth, gen)))Chiselのデータ型gen。私たちは、読み取りと書き込みポインタをインクリメントし、機能counterと、読み取りと書き込みのポインタを作成するために、2つの信号を定義します。両方のポインタが等しい場合、バッファが空またはフルのいずれかです。私たちは、空とフルの概念のために二つのフラグを定義します。

emptyRegがディアサートされていることを確認バッファに (1) 書き込み、(2) 書き込みポインタがでリードポインタに追いつくだろう場合、(3) は、フルバッファをマーク：プロデューサーがアサートするとvalidとFIFOは、私たち一杯になつていません次のクロックサイクル（次の書き込みポインタと、現在の読み出しポインタを比較されたい）、及び (4) は増分への書き込みカウンタに信号を送ります。

消費者がreadyで、FIFOは、私たちを空にされていない場合：(1) fullRegがディアサートされていることを確認し、(2) 読み出しポインタは、次のクロック・サイクルでの書き込みポインタに追いつくかどうかバッファが空にマークし、(3) は、増分に読み出しカウンタに信号を送ります。

FIFOの出力は、読み取りポインタアドレスのメモリ素子です。準備ができて、有効なフラグは、単にフルと空の旗から派生しています。

11.3.5 オンチップメモリ付きFIFO (L11872 TODO)

FIFOの最後のバージョンは、小さなFIFOに適したソリューションであるメモリを、表現するためにレジスタ・ファイルを使用していました。大規模のFIFOにとっては、オンチップ・メモリを使用することをお勧めします。一覧～11.7は、ストレージのための同期メモリを使用してFIFOを示しています。

```

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

  def counter(depth: Int, incr: Bool): (UIInt, UIInt) = {
    val cntReg = RegInit(0.U(log2Ceil(depth).W))
    val nextVal = Mux(cntReg === (depth-1).U, 0.U, cntReg + 1.U)
    when (incr) {
      cntReg := nextVal
    }
    (cntReg, nextVal)
  }

  val mem = SyncReadMem(depth, gen)

  val incrRead = WireDefault(false.B)
  val incrWrite = WireDefault(false.B)
  val (readPtr, nextRead) = counter(depth, incrRead)
  val (writePtr, nextWrite) = counter(depth, incrWrite)

  val emptyReg = RegInit(true.B)
  val fullReg = RegInit(false.B)

  val idle :: valid :: full :: Nil = Enum(3)
  val stateReg = RegInit(idle)
  val shadowReg = Reg(gen)

  when (io.enq.valid && !fullReg) {
    mem.write(writePtr, io.enq.bits)
    emptyReg := false.B
    fullReg := nextWrite === readPtr
    incrWrite := true.B
  }

  val data = mem.read(readPtr)

  // Handling of the one cycle memory latency
  // with an additional output register
  switch(stateReg) {
    is(idle) {
      when(!emptyReg) {
        stateReg := valid
        fullReg := false.B
        emptyReg := nextRead === writePtr
        incrRead := true.B
      }
    }
    is(valid) {
      when(io.deq.ready) {
        when(!emptyReg) {
          stateReg := valid
          fullReg := false.B
          emptyReg := nextRead === writePtr
          incrRead := true.B
        } otherwise {
          stateReg := idle
        }
      } otherwise {
        shadowReg := data
        stateReg := full
      }
    }
    is(full) {
  }
}

```

```

class CombFifo[T <: Data](gen: T, depth: Int) extends Fifo(gen: T, depth: Int) {

    val memFifo = Module(new MemFifo(gen, depth))
    val bufferFIFO = Module(new DoubleBufferFifo(gen, 2))
    io.enq <> memFifo.io.enq
    memFifo.io.deq <> bufferFIFO.io.enq
    bufferFIFO.io.deq <> io.deq
}

```

Listing 11.8: Combining a memory based FIFO with double-buffer stage.

```

when(io.deq.ready) {
    when(!emptyReg) {
        stateReg := valid
        fullReg := false.B
        emptyReg := nextRead === writePtr
        incrRead := true.B
    } otherwise {
        stateReg := idle
    }
}

io.deq.bits := Mux(stateReg === valid, data, shadowReg)
io.enq.ready := !fullReg
io.deq.valid := stateReg === valid || stateReg === full
}

```

Listing 11.7: A FIFO with a on-chip memory.

読み取りと書き込みポインタの取り扱いは、レジスタメモリFIFOと同じです。しかし、同期オンチップメモリは、レジスタファイルの読み出しが同じクロックサイクルで利用可能であった次のクロックサイクルにおいて、読み出しの結果を提供します。

したがって、我々はこの遅延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を必要としています。私たちは、メモリを読み出して出力ポートへのキューの先頭の値を提供します。その値が消費されていない場合、我々はメモリから次の値を読みながら、シャドウ・レジスタshadowRegに保管する必要があります。ステートマシンが表現するために3つの状態から成る：(1) 空のFIFOは、(2) 有効なデータがメモリから読み出され、シャドウ・レジスタと、有効データ（次の要素）内のキューの（3）頭部から想い出。

FIFOベースのメモリを効率的にキューに大量のデータを保持し、待ち時間を通じて短い秋を持っていることができます。最後の設計では、FIFOの出力は、メモリの読み取りから直接来るかもしれません。このデータパスは、デザインのクリティカル・パスにある場合は、我々は2つのFIFOを組み合わせることで簡単にパイプライン私たちのデザインすることができます。一覧～11.8は、このような組み合わせを示しています。FIFOベースのメモリの出力では、我々は二重の出力からメモリ読み出し経路を分離するためにFIFOバッファを单一のステージを追加します。

11.4 演習 (L11980 TODO)

それは2つの演習が含まれているとして、この演習のセクションでは、少し長いです：(1) は、バブルFIFOを探索し、異なるFIFOの設計を実装します。及び (2) UARTを探索し、それを拡張します。両方の練習のためのソースコードは、[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 (12004 TODO)

FIFO源は、異なる読み取りおよび書き込み動作を引き起こすと [value change dump \(VCD\)](#) 形式で波形を生成するテストを含みます。 VCDファイルには、 [GTKWave](#) として、波形ビューアで表示できます。リポジトリに [FifoTester](#) を探検。リポジトリは FIFO の例は、単に入力のために、例を実行するには [Makefile](#) が含まれています RRR0009nmx3Tok この make コマンドは、 FIFO をコンパイル、テストを実行し、波形表示用に GTKWave を開始します。テストと生成された波形を探検。

```
$ 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。 テスターと生成された波形を探ります。

最初のサイクルでは、テスターは、単一の単語を書き込みます。その言葉は、したがって、 FIFO を通じて名前 *bubble FIFO* バブルなどのように我々は波形で観察することができます。この泡立ちはまた、 FIFO を介してデータワードのレイテンシは FIFO の深さに等しいことを意味します。

それがいっぱいになるまで、次のテストでは、 FIFO を埋めます。 単一の読み取りは、以下の通りです。 空の言葉は、作家側に FIFO の読者側から気泡をどのように注意してください。バブル FIFO がフルになると、それは作家側に影響を与えるために、読み取りのためのバッファの深さの待ち時間がかかります。

試験の終わりには、試行が最大速度で読み書きすることループを含んでいます。 私たちは、バブル FIFO は、単語ごとに 2 回のクロックサイクルで最大帯域幅、で動作して見ることができます。バッファ・ステージは、単一のワード転送のために、空と完全切り替えること常に有します。

バブル FIFO は簡単で、小さなバッファ用の低リソース要件があります。 N ステージバブル FIFO の主な欠点がある：一つの単語毎に 2 クロックサイクル、(2) データワードがリーダー端に NK クロックサイクルを移動しなければならないスループット (1) 最大値、及び (3) 完全な FIFO は、再起動のために N クロック・サイクルを必要とします。

これらの欠点は [circular buffer](#) と FIFO の実装によって解決することができます。 循環バッファメモリと、読み取りと書き込みポインタで実現することができます。 同じインターフェースを使用して、四つの要素を有する循環バッファとして FIFO を実装し、テスターと異なる挙動を探ります。ショートカットとして円形のバッファ使用、レジスタ (`Reg(Vec(4, UInt(size.W)))`) のベクトルの最初の実装のために。

11.4.2 UART (L12162 TODO)

UART のたとえば、（通常は USB 接続で）あなたのラップトップ用のシリアルポートを備えた FPGA ボードとシリアルポートを必要としています。あなたのラップトップ上の FPGA ボードとシリアルポート間のシリアルケーブルを接続します。電子のターミナルプログラムを起動します。グラム。 Linux 上で Windows または [gtkterm](#) 上のハイパーテーミナル : RRR000kxWLeJ0 設定あなたのポートが、これは多くの場合、 /dev/ttyUSB0 のようなものである USB の UART で、正しいデバイスを使用します。

```
$ gtkterm &
```

115200 の ポーレート と パリティ なしまだ フロー制御 (ハンドシェイク) を設定します。次のコマンドを使用すると、UART の Verilog コードを作成することができます RRR000et1CmhaB は、

```
$ make uart
```

次にデザインを合成するためにあなたの [synthesize](#) ツールを使用します。リポジトリは DE2-115 FPGA ボード用の Quartus のプロジェクトが含まれています。Quartus でデザインを合成して、FPGA を設定するには、再生ボタンを使用します。設定が完了したら、ターミナルでの挨拶のメッセージが表示されるはずです。

UART で点滅 LED の例を拡張し、LED がオフとオンのとき、シリアルラインに 0 と 1 を書き込みます。 Sender の例のように、 [BufferedTx](#) を使用してください。

文字の遅い出力 (毎秒 2) を使用すると、登録した送信 UART にデータを書き込むことができ、リード / 有効なハンドシェイクを無視することができます。繰り返し番号を書いて 0-9 として速い ポーレート が許す限りでの例を拡張します。このケースでは、送信バッファが自由であるかどうかを確認するために UART のステータスをポーリングするためにあなたのステートマシンを拡張する必要があります。

例のコードは Tx ための単一のバッファを含んでいます。あなたは送信機と受信機にバッファリングを追加するために実装されていることを FIFO を追加すること自由に感じなさい。

11.4.3 FIFO探査 (L12296 TODO)

専用レジスタで4つのバッファ要素を有する単純なFIFOに書き込みます。使用2ビットの読み取りと書き込みのカウンター、できるだけちょうどオーバーフローが。さらに簡素化したよう読み取りと書き込みポインタが空のFIFOとして等しい状況を考えます。これは、あなたが最大限に3つの要素を格納できることを意味します。この単純化は、一覧～[11.6](#)の例と同じポインタ値を持つ空またはフルの取り扱いからカウンタ機能を回避することができます。これは単独のポインタ値から導出することができるよう私たちは、空またはフルフラグは必要ありません。このデザインは、どのようにするかに簡単ですか？

異なるFIFO設計は、以下の特性に対して異なる設計のトレードオフを持って提示：レイテンシを介して（1）最大スループット、（2）秋、（3）リソース要件、及び（4）最大クロック周波数。FPGAのためにこれらを合成することによって、異なるサイズのすべてのFIFOのバリエーションを探検。ソースは[chisel-examples](#)で入手可能です。4つのワード、16ワード、および256ワードのFIFOのためのスイートスコットはどこですか？

12 プロセッサの設計 (L12361 TODO)

設計、シミュレーション、およびマイクロプロセッサのテスト：この本の最後の章の一つとして、我々は中規模のプロジェクトを提示します。このプロジェクトの扱いを保つために、我々は、単純なアキュムレータマシンを設計します。これは高度な例であり、いくつかのコンピュータアーキテクチャの知識が提示コード例に従うことが必要であることに言及したいと思います。

レロスはシンプルに設計されたが、それでもCコンパイラのための良好な標的されます。1ページの指示フィットの説明は、表～12.1を参照してください。そのテーブルのAアキュムレータを表し、PCはプログラムカウンタであり、iが(0～255)即値であり、RnレジスタRRR011tYQBO6l0(0～255)、RRR011NpcY2k6W分岐はRRR011RQtG1BaSに対してオフセット、及びRRR011jfje2hSnのアドレスレジスタメモリアクセス。

12.1 ALUから始める (L12464 TODO)

プロセッサの中心的なコンポーネントは短いためarithmetic logic unit、またはALUです。したがって、我々は、ALUとテストベンチのコーディングを始めます。まず、ALUの異なる操作を表現するためにEnumを定義します。

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

AN ALUは、通常、2つのオペランド入力（それらaとbを呼び出す）、操作op（オペコード）機能と出力yを選択する入力を有します。一覧～12.1は、ALUを示しています。

私たちは、最初の三つの入力のための短い名前を定義します。switch文はresの計算のためのロジックを定義します。したがって、0のデフォルトの割り当てを取得します。switch文は、すべての操作を列挙し、それに応じて表現を割り当てます。すべての操作は、リグ式に直接マッピングされています。最後に、我々は、ALUの出力yに結果resを割り当てます

一覧～12.2に示すように、テストのために、私たちは、平野ScalaでALU関数を記述します。

Scalaの実装によってChiselに書き込まれたハードウェアのこの重複は指定のエラーを検出しませんが、それは、少なくともいくつかの健全性チェックです。私たちは、テストベクトルとして、いくつかのコーナーケース値を使用します。

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

我々は両方の入力にこれらの値を持つすべての機能をテストします。

```
def test(values: Seq[Int]) = {
    for (fun <- add to shr) {
        for (a <- values) {
            for (b <- values) {
                poke(dut.io.op, fun)
                poke(dut.io.a, a)
                poke(dut.io.b, b)
                step(1)
                expect(dut.io.y, alu(a, b, fun.toInt))
            }
        }
    }
}
```

32ビットの引数の完全な、徹底的なテストは、我々は、入力値として、いくつかのコーナーケースを選択した理由があった、ことはできません。コーナーケースに対してテストのほかに、それはランダムな入

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	$A = A \text{ and } Rn$	And register Rn with A
andi	$A = A \text{ and } i$	And immediate value i with A
or	$A = A \text{ or } Rn$	Or register Rn with A
ori	$A = A \text{ or } i$	Or immediate value i with A
xor	$A = A \text{ xor } Rn$	Xor register Rn with A
xori	$A = A \text{ xor } i$	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	$A = \text{mem}[AR+(i \ll 2)]$	Load a word from memory into A
loadindbu	$A = \text{mem}[AR+i]_{7-0}$	Load a byte unsigned from memory into A
storeind	$\text{mem}[AR+(i \ll 2)] = A$	Store A into memory
storeindb	$\text{mem}[AR+i]_{7-0} = A$	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)

Table 12.1: Leros instruction set.

```

class Alu(size: Int) extends Module {
    val io = IO(new Bundle {
        val op = Input(UInt(3.W))
        val a = Input(SInt(size.W))
        val b = Input(SInt(size.W))
        val y = Output(SInt(size.W))
    })

    val op = io.op
    val a = io.a
    val b = io.b
    val res = WireDefault(0.S(size.W))

    switch(op) {
        is(add) {
            res := a + b
        }
        is(sub) {
            res := a - b
        }
        is(and) {
            res := a & b
        }
        is(or) {
            res := a | b
        }
        is(xor) {
            res := a ^ b
        }
        is(shr) {
            // the following does NOT result in an unsigned shift
            // res := (a.asUInt >> 1).asSInt
            // work around
            res := (a >> 1) & 0x7fffffff.S
        }
        is(ld) {
            res := b
        }
    }

    io.y := res
}

```

Listing 12.1: The Leros 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
  }
}

```

Listing 12.2: The Leros ALU function written in Scala.

力に対してもテストに便利です。

```

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

あなたがレロスのプロジェクト内でテストを実行することができます

$ sbt "test:runMain leros.AluTester"

```

とに似て成功メッセージを作成しなければなりません。

```

[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 (L12644 TODO)

ALUから、我々は後方に働くと命令デコーディングを実装しています。しかし、最初に、我々は独自のScalaのクラスの命令エンコーディングと`shared`パッケージを定義します。私たちは、レロス、レロスのためのアセンブラーのハードウェア実装、およびレロスの命令セットシミュレータ間のエンコード定数を共有したいです。

```

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
// ...
```

デコード成分のため、我々は、後でALUに部分的に供給される出力用Bundleを定義します。

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

デコードは、入力として、8ビットのオペコードを受け取り、出力として復号信号を送出します。これらの駆動信号はWireDefaultと、デフォルト値が割り当てられています。

```
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
}
```

デコーディング自体はほとんどの命令上位8ビット用のレロスにオペコードを（表し命令の一部にだけ大きなスイッチ文です。）

```
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 アセンブラー命令 (L12729 TODO)

レロスの書き込みプログラムに、私たちは、アセンブラーが必要です。しかし、非常に最初のテストのために、我々はハードコード数命令することができ、私たちは命令メモリを初期化するために使用Scalaの配列、にそれらを置きます。

```

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ための非常に非効率的なアプローチです。 Scalaのような表情豊かな言語とアセンブラーを書くことは、大きなプロジェクトではありません。したがって、我々は、コードの約100ライン内で可能であるレロスのための簡単なアセンブラーを書きます。私たちは、アセンブラーを呼び出す関数の`getProgram`を定義します。分岐先のために、我々は我々が`Map`に収集シンボルテーブルを、必要とします。2回のパスで古典アセンブラー実行：(1) シンボルテーブルの値を収集し、(2) 最初のパスに収集記号でプログラムを組み立てます。したがって、我々はそれがある渡すかを示すためにパラメータで二回`assemble`を呼び出します。

```

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`機能は、ソースファイル¹と二つの可能なオペランド解析するために2つのヘルパー関数を定義する：レジスタ番号を読み取るために（1）～（10進数または16進数表記を可能にする）整数定数及び（2）。

```

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
  }
}

```

リスト～12.3はレロスのためのアセンブラーのコアを示しています。スカラ`match`発現は、アセンブリ機能のコアを覆っています

¹この関数は、実際にソースファイルを読み取ることはできませんが、この議論のために、我々は読み取り機能としてそれを考えることができるで読んで始まります。

```

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")
    }
}

```

Listing 12.3: The main part of the Leros assembler.

12.4 演習 (L12833 TODO)

最後の章のいずれかで、この演習の割り当ては非常に自由な形です。あなたはChiselを通して、あなたの学習ツアーハイライトの最後に、あなたが面白いことを設計上の問題に取り組む準備ができます。

1つのオプションは、章を再読し、[Leros repository](#)内のすべてのソースコードと一緒に読んで、テストケースを実行し、それを破壊することにより、コードとフィドルとテストが失敗することを確認することです。

別のオプションは、レロスの実装を記述することです。リポジトリ内の実装は、パイプラインのひとつの可能な組織です。あなただけの単一のパイプラインステージでレロスのChiselシミュレーションバージョンを書き込み、またはcreasyを行くと可能な限り最高のクロック周波数のためレロスをスーパーパイプがあります。

第三の選択肢は最初からあなたのプロセッサを設計することです。たぶんレロスプロセッサを構築する方法のデモンストレーションや必要なツールは、プロセッサの設計と実装は魔法アートが、非常に楽しいことができエンジニアリングではないことを確信しています。

13 Chisel への貢献 (L12905 mune10 初回校正済)

Chiselは、絶えま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。そのため、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のライブラリの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で構成されています。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。

あなたの個人GitHubのアカウントに、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インターフェイスで 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、そのフォークしたリポジトリをクローンします。¹ この例では、chisel3を変更し、私のローカルのフォークを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
```

パブリッシュしたライブラリのバージョン文字列について、パブリッシュローカルコマンドの実行の際に気をつけてください。文字列SNAPSHOTが含まれてパブリッシュされます。もしあなたがテスターを用いる際に、パブリッシュしたバージョンが、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して、ローカルでパブリッシュしてください。

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ります。例えば、[empty Chisel project](#)をクーリング/フォークして名前を変更します。そこから.gitフォルダは削除しておきます。

ローカルにパブリッシュ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、この記事の執筆時点では、Chisel元のヘッドは、Scala の 2.12 を使用しています。しかし、Scala 2.12 は問題[anonymous bundles](#)があります。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は次のようにになります。

```
scalaVersion := "2.12.6"

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

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

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

¹Chisel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があることに注意してください

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パブリッシュバージョンを使っているか注意する。(例えば、アプリケーションコードとChiselライブラリのScalaのバージョンが異なっている場合、ローカルパブリッシュバージョンではなく、サーバからのSNAPSHOTバージョンを選択している。)

[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
```

また、あなたは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版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文書化[contribution guidelines](#)、[貢献ガイドライン](#)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辺を、それ以外の場合は右辺を返す、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、そのくらいのコード行数が必要でしょうか？²

ただし、このような感じで、些細なオペレータを追加するためにChiselプロジェクトをフォークしたりしないでください。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。

もしも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、その解決を試みるすることができます。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見てみましょう。

²てつ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 (L12556 mune10 初回校正済)

この本は、ハードウェア構成の言語である Chisel を用いたデジタル回路設計の入門書です。シンプルなものから中規模の Chiselで記述されたデジタル回路設計を見てきました。 ChiselはScalaベースのDSLですので、 Scalaの強力な抽象化を継承しています。この本は入門書として意図されているため、本書での記述は Scalaの簡単な使い方にとどめています。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のプロジェクトに適用してゆくことです。

本書のフィードバックをお寄せください、それは本書を改善し、次の版に反映されるでしょう。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。本書の修正や改善の関する 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

本書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コースとすべてのChiselのサンプルコードについてのスライ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）

本書の中でも参照している、中規模の実装例は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 (L12650 mune20 初回校正済)

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の実装コードはそんなにありません。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インターフェクト(バス)生成器を含む RISC-V [12] 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 [9] 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 [7]、ネットワーク・オン・チップ [8]、オーナーシップ対応の共有スクラッチパッドメモリ [10]、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 [13] 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。私はChiselで、ハードウェアを記述し、FPGA上でそれを実行し、Scalaでアセンブラーを書き、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) を使用したニューラルネットワークアクセラレータと統合されたRISC-Vロケットプロセッサです [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、その情報を私までメールを送ってください。その情報を、この本の将来版に含めるたいと思います。

B Chisel 2 (L12866 mune10 初回校正済)

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](#)

しかしながら、あなたがまだ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 \[9\]](#) 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の間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 パッケージを使い、この互換性レイヤを介することで、Chisel3プロジェクトで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 (L13080 mune10 初回校正済)

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

Bibliography

- [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 Groeneweld, 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 Appavoo and 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, 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.
- [9] 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.
- [10] Martin Schoeberl, Tóður Biskopstø Strøm, Oktay Baris, and Jens Sparsø. Scratchpad memories with ownership. In *2019 Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2019.
- [11] Bill Venners, Lex Spoon, and Martin Odersky. *Programming in Scala, 3rd Edition*. Artima Inc, 2016.
- [12] 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.
- [13] Michael Zimmer. *Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2015.

Index

- ALU, 27, 95
- arithmetic operations, 8
- Array, 11
- Assembler, 99
- Asynchronous Input, 49
- BCD, 76
- Binary-coded decimal, 76
- Bit
 - concatenation, 9
 - extraction, 9
 - reduction, 9
- Bitfield
 - concatenation, 9
 - extraction, 9
- Bool, 8
- Bubble FIFO, 82
- Bulk connection, 28
- Bundle, 11
- Chisel
 - Contribution, 103
 - Examples, 4, 107
- Chisel 2, 109
- Circular buffer, 88
 - read pointer, 88
 - write pointer, 88
- Clock, 35
- Collection, 11
- Combinational circuit, 31
- Communicating state machines, 63
- Component, 25
- Counter, 37
- Counting, 11
- Data forwarding, 44
- Datapath, 67
- Debouncing, 49
- Decoder, 32
- DecoupledIO, 86
- Double buffer FIFO, 87
- Edge detection, 51
- elsewhen, 31
- Encoder, 34
- FIFO, 81
- FIFO buffer, 81
- File reading, 76
- Finite-State Machine
 - Mealy, 58
 - Moore, 55
- Finite-state machine, 55
- First-in, first-out buffer, 81
- Flip-flop, 35
- FSM, 55
- FSMD, 64
- Function components, 29
- Functional programming, 78
- Hardware generators, 73
- if/elseif/else, 31
- Inheritance, 77
- Initialization, 35
- Integer
 - constant, 7
 - signed, 7
 - unsigned, 7
 - width, 7
- IO interface, 25
- Leros, 95
- Logic generation, 76
- Logic table generation, 76
- Logical clock, 40
- logical operations, 8
- Majority voting, 51
- Memory, 44
- Metastability, 49
- Module, 25
- Multiplexer, 9
- Object-oriented, 77
- Operators, 9
- otherwise, 31
- Parameters, 73
- Ports, 25
- Processor, 95
 - ALU, 95
 - instruction decode, 98
- RAM, 44
- Ready-valid interface, 69, 86
- Register, 35
 - with enable, 36

Reset, 35
sbt, 15
ScalaTest, 20
Serial port, 82
Source organization, 15
SRAM, 44
State diagram, 55
State machine with datapath, 64
Structure, 11
switch, 32
Synchronous memory, 44
Synchronous sequential circuit, 55

Testing, 18
Tick, 40
Timing diagram, 36
Timing generation, 39
tuple, 89
Type parameters, 73

UART, 82

Vector, 11

Waveform diagram, 36
when, 31

カウンター, 37
コンポーネント, 25
ポート, 25
メタステーブル, 49
メモリー, 44
モジュール, 25
リセット, 35

初期化, 35

有限オートマトン, 55
有限状態機械, 55
波形図, 36

非同期入力, 49