

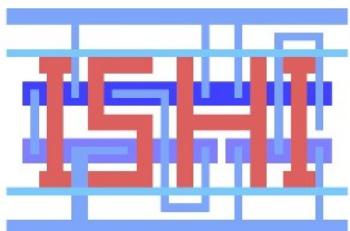
Turing Machine を題材とする 自前チップ設計試作

— Tiny Tapeout で自作CPU をシリコン化しよう —

2026年3月5日

Ver 1.0

ISHI会 : <https://ishi-kai.org>



圓山 宗智 (まるやま むねとも)

e-mail munetomo@ishi-kai.org
X @Processing_Unit

目次

第1章 CPU コアを設計して FPGA に実装して動かそう	1
1.1 CPU を作ろう！	1
1.2 コンピュータの歴史とチューリング・マシン	1
1.2.1 数学の不完全性定理の提唱	1
1.2.2 チューリング・マシンによる考察	2
1.2.3 停止性問題を解くアルゴリズムは存在しないことの証明	2
1.2.4 プログラムとデータを一体化	3
1.3 コンピュータの基礎	3
1.3.1 ノイマン型コンピュータの誕生	3
1.3.2 CPU が本質的にやっていること	3
1.3.3 CPU システムの基本構造	3
1.3.4 CPU の性能向上	5
1.4 CPU を作ろう	6
1.4.1 どういう CPU を作ろうか？	6
1.4.2 外部メモリをどうするか？	6
1.4.3 キャッシュ・メモリの活用	7
1.4.4 CPU の規模とアーキテクチャをどうするか？	7
1.5 完全チューリング・マシン bfCPU を作る	7
1.6 bfCPU の設計データ	7
1.7 bfCPU のアーキテクチャ	8
1.7.1 bfCPU の深淵さ	8
1.7.2 bfCPU の基本構成	8
1.7.3 bfCPU の命令セット	8
1.7.4 begin 命令 ([]) と end 命令 ([]) はどうやってジャンプできるのか？	11
1.7.5 bfCPU のプログラムの注意点	11
1.8 bfCPU のアセンブラー・シミュレータ bfTool	11
1.8.1 bfTool のビルト方法	12
1.8.2 bfTool の関連ファイル	12
1.8.3 プログラムのアセンブル方法	12
1.8.4 プログラムのシミュレーション方法	13
1.9 bfCPU のプログラム例	14
1.9.1 データ・メモリをゼロ・クリアするプログラム	14
1.9.2 加算プログラム	15
1.9.3 乗算プログラム	15
1.9.4 Hello World! プログラム	16
1.9.5 10進数表記の ASCII 文字列を UART で受信しバイナリ数値に変換するプログラム	17
1.9.6 バイナリ数値を 10進数表記の ASCII 文字列に変換して UART から送信するプログラム	17
1.9.7 TicTacToe ゲーム	17
1.9.8 Life ゲーム	19
1.10 bfCPU はどんな問題も解けるのか？	22
1.10.1 多バイト長加算は可能か？	22
1.10.2 AND 演算は可能か？	23

1.10.3 OR 演算は可能か？	23
1.10.4 XOR 演算は可能か？	23
1.10.5 bfCPU はコンピュータとしてのあらゆる処理が可能	23
1.11 bfCPU システムの論理設計	23
1.11.1 bfCPU システムの全体構成	23
1.11.2 bfCPU のメモリ・マップとメモリ構成	24
1.11.3 QSPI SRAM の書き込み方法	24
1.11.4 UART による入出力	24
1.12 bfCPU チップの内部バス・アクセス・タイミング	25
1.12.1 ライト・アクセス	26
1.12.2 リード・アクセス	26
1.13 bfCPU のパイプライン構成	27
1.13.1 パイプライン・ステージ	27
1.13.2 パイプラインのウェイト制御	28
1.13.3 bfCPU のパイプライン制御	29
1.13.4 (d) 命令 out(.) の動作	30
1.13.5 (e) 命令 in(,) の動作	30
1.13.6 (f) 命令 begin([]) の動作	30
1.13.7 (g) 命令 end([]) の動作	30
1.13.8 (h) 命令 reset の動作	31
1.13.9 (i) 命令 nop の動作	31
1.14 bfCPU のデータ・バス論理	32
1.15 bfCPU の制御論理	33
1.16 bfCPU の RTL 記述	34
1.17 キャッシュ・メモリ	35
1.18 bfCPU の命令キャッシュ	36
1.18.1 命令キャッシュの構造	36
1.18.2 命令キャッシュの動作	36
1.19 bfCPU のデータ・キャッシュ	38
1.19.1 データ・キャッシュの構造	38
1.19.2 データ・キャッシュの動作	38
1.20 キャッシュ・メモリの RTL 記述	41
1.21 外部メモリ QSPI SRAM とそのインターフェース	43
1.21.1 Microchip 23LC512-I/P	43
1.21.2 SPI モードから QSPI モードへの変更方法	43
1.21.3 QSPI モードから SPI モードへの変更方法	44
1.21.4 QSPI SRAM のリード・タイミング	45
1.21.5 QSPI SRAM のライト・タイミング	46
1.22 QSPI SRAM インタフェースの RTL 記述	46
1.23 入出力用 UART モジュール	47
1.23.1 UART のコア	47
1.23.2 UART のレジスタ	47
1.23.3 UART の IO バスとのインターフェース論理	47
1.24 UART の RTL 記述	48
1.25 bfCPU システム全体の構成	48
1.25.1 システム全体の階層構造	48
1.25.2 QSPI SRAM 信号の扱い	50

1.26	bfCPU システムの論理シミュレーションによる機能検証	50
1.26.1	機能検証に使う環境とリソース	50
1.26.2	テストベンチ tb.sv	50
1.26.3	論理シミュレーション実行のための準備	51
1.26.4	論理シミュレーションの実行	51
1.26.5	波形ファイルの確認	51
1.27	bfCPU システムを FPGA に実装する	52
1.27.1	使用する FPGA ボード	52
1.27.2	bfCPU の FPGA 設計データ	52
1.27.3	FPGA 開発ツール Quartus Prime	52
1.27.4	FPGA 開発ツール Quartus Prime のプロジェクト設定	53
1.27.5	タイミング制約	54
1.27.6	FPGA を合成してコンフィグ	56
1.27.7	FPGA ボード、ラズパイ、QSPI SRAM の接続	56
1.27.8	ラズパイ上に bfCPU 開発環境を構築	57
1.27.9	bfCPU システムを実装した FPGA の動作確認	60
1.27.10	Life ゲームの実行	61
1.27.11	乗算プログラムの実行	62
1.28	色々遊べる bfCPU システムが手に入った！	62
1.29	参考文献	63
第 2 章	bfCPU を Tiny Tapeout により実シリコン化しよう	64
2.1	Tiny Tapeout とは?	64
2.1.1	Tiny Tapeout の基本概念：半導体設計の民主化	64
2.1.2	独自の製造モデル：マルチ・プロジェクト・ウェハ (MPW) とタイル構造	64
2.1.3	進化する設計環境と技術インフラ	65
2.1.4	教育とコミュニティの広がり	65
2.1.5	Tiny Tapeout がもたらすインパクト	65
2.1.6	Tiny Tapeout の設計フロー “LibreLane”	66
2.1.7	Tiny Tapeout のチップ・レイアウト	66
2.1.8	Tiny Tapeout のユーザ端子	67
2.1.9	Tiny Tapeout の評価ボード	67
2.2	Tiny Tapeout のクラウド設計とローカル設計	68
2.3	LibreLane 設計フローをローカル PC 上で動かす	69
2.3.1	[準備] Windows 11 WSL2 Ubuntu の場合	69
2.3.2	[準備] ネイティブ Ubuntu の場合	70
2.3.3	サンプル設計データ factory_test で LibreLane の設計フローを動かす	71
2.4	LibreLane 設計フローをクラウド上で動かす【正式設計フロー】	76
2.4.1	Tiny Tapeout の GitHub からテンプレートを Fork する	76
2.4.2	Fork してできたリポジトリをローカルに Clone する	76
2.4.3	bfCPU の RTL 記述を準備	77
2.4.4	info.yaml を書き換える	79
2.4.5	ローカル PC で設計確認	81
2.4.6	エラーの対策	81
2.4.7	論理シミュレーション	82
2.4.8	レイアウト結果を確認	89
2.4.9	docs/info.md を編集	89
2.4.10	クラウドにデータをアップロード	90

2.4.11	レイアウト結果を 2D 表示と 3D 表示でチェック	91
2.4.12	Tiny Tapeout へ設計データを提出	91
2.5	Tiny Tapeout の活用	92
付録 A	論理設計と System Verilog 入門	94
A.1	論理設計とは？	94
A.1.1	論理設計でやること	94
A.1.2	仕様が決まった後の論理設計の大まかな流れ	94
A.2	論理設計の抽象度	95
A.2.1	抽象度とは？	95
A.2.2	論理設計における抽象度	95
A.2.3	より高い抽象度	96
A.3	ASIC と FPGA の違い	96
A.3.1	ASIC って何？	96
A.3.2	FPGA って何？	96
A.3.3	ASIC と FPGA の設計フロー	97
A.4	組み合わせ回路	99
A.4.1	論理設計の基本は組合せ回路	99
A.4.2	組合せ回路の代表 = 論理ゲート	99
A.4.3	その他の組合せ回路	99
A.4.4	組合せ回路のタイミングは遅延のみ	99
A.4.5	ハミング距離とハザード	99
A.4.6	正論理と負論理	100
A.4.7	ド・モルガンの法則	100
A.4.8	NAND があれば全ての論理回路が組める	101
A.4.9	消費電力が多い NMOS トランジスタによる論理ゲート	101
A.4.10	消費電力が少ない CMOS トランジスタによる論理ゲート	101
A.4.11	イネーブル信号やストローブ信号はなぜ負論理？	102
A.4.12	遅延時間の種類	102
A.5	順序回路と D-フリップ・フロップ	103
A.5.1	順序回路とは？	103
A.5.2	クロックと D-F/F	103
A.5.3	D-F/F のセットアップ時間とホールド時間	104
A.5.4	D-F/F のセットアップ時間とホールド時間を違反したらどうなるか	104
A.5.5	非同期信号の受け方	104
A.5.6	タイミング設計の基本	104
A.5.7	セットアップ時間の検証	105
A.5.8	ホールド時間の検証	106
A.5.9	クロック・ツリー	107
A.5.10	実機デバッグでの苦労を防ぐには	107
A.5.11	RTL レベルで設計する時の D-F/F タイミングの考え方	107
A.5.12	リセットの考え方	108
A.6	ステート・マシン	108
A.6.1	ステートマシンは順序回路の代表	108
A.6.2	状態遷移図	109
A.6.3	状態遷移の動き	109
A.6.4	ステートマシンの内部論理	109

A.6.5	ステートマシンの階層化	111
A.6.6	ステートマシンの種類	111
A.7	機能モジュールの論理設計	112
A.7.1	機能モジュールの構造	112
A.7.2	データバス部	112
A.7.3	データバス内のレジスタとその制御	112
A.7.4	データバス内の演算器やセレクタとその制御	113
A.7.5	データバスの構成	114
A.7.6	内部バスの構成方法	115
A.8	その他、ASIC 設計で検討すべき項目	115
A.9	SystemVerilog とは	116
A.9.1	Verilog HDL から SystemVerilog まで	116
A.9.2	論理シミュレーションの基本はイベント・ドリブン方式	116
A.9.3	遅延時間の扱いはタイム・マップで	116
A.10	SystemVerilog : モジュール構造と階層構造	117
A.10.1	モジュール構造と階層構造の記述	117
A.10.2	インスタンス化	117
A.11	SystemVerilog : 信号(変数)の表現	117
A.11.1	logic と wire	117
A.11.2	変数(信号)の定義方法	118
A.12	SystemVerilog : 組み合わせ回路の書き方	119
A.12.1	組合せ回路の書き方には2種類ある	119
A.12.2	継続代入文(assign)	119
A.12.3	手続き代入文(always_comb)	119
A.13	SystemVerilog : ブロックング代入とノン・ブロックング代入	119
A.14	SystemVerilog : 演算子	120
A.15	SystemVerilog : 時間と遅延	121
A.15.1	時間単位と時間精度の設定	121
A.15.2	遅延付加時の代入動作	122
A.16	SystemVerilog : 条件判断	122
A.16.1	if 文による条件判断	122
A.16.2	case 文による条件判定	122
A.16.3	ラッチ回路生成の危険	123
A.17	SystemVerilog : 順序回路の書き方	124
A.17.1	D-F/F の書き方	124
A.17.2	リセット付き D-F/F の書き方	124
A.17.3	順序回路のひな形	124
A.17.4	順序回路は必ずブロックング代入文を使う	125
A.18	論理シミュレーションのためのテストベンチ	125
A.18.1	ここまで文法で全ての論理機能は記述できる	125
A.18.2	論理シミュレーションを動かすテストベンチ	126
A.18.3	initial 文	126
A.18.4	forever 文	126
A.18.5	波形ファイル出力指示	126
A.18.6	クロック信号生成記述	127
A.18.7	リセット信号生成記述	127
A.18.8	サイクル・カウンタとシミュレーションのタイムアウト	128

A.18.9 デバッグ用メッセージ出力	128
A.18.10 機能検証用入力パターンの生成	128
A.19 システム・タスクとコンパイラ指示子	130
A.19.1 `include 文	130
A.19.2 `ifdef 文、`else 文、`elsif 文、`endif 文	130
A.20 定数パラメータの上書き機能	130
A.21 論理シミュレータと波形ビューワのインストール	131
A.22 設計例：リロード式アップ・ダウン・カウンタ “simpleCounter”	131
A.22.1 リロード式アップ・ダウン・カウンタを記述してみよう	131
A.22.2 設計ファイルの用意	131
A.22.3 論理機能モジュールを記述する	132
A.22.4 テストベンチを記述する	133
A.22.5 論理シミュレーション実行のための準備	133
A.22.6 論理シミュレーションの実行	134
A.22.7 波形ファイルの確認	135
A.23 設計例：簡易 CPU “picoCPU”	136
A.23.1 チップ設計の醍醐味！ CPU の論理設計	136
A.23.2 picoCPU の命令セット・アーキテクチャ	136
A.23.3 メモリ空間と I/O ポート	136
A.23.4 picoCPU システムのブロック図	137
A.23.5 バス・アクセスのタイミング	137
A.23.6 設計ファイルの用意	138
A.23.7 最上位階層 TOP モジュールの SystemVerilog 記述	138
A.23.8 CPU モジュールの SystemVerilog 記述	138
A.23.9 MEM モジュールの SystemVerilog 記述	139
A.23.10 論理シミュレーション実行のための準備	140
A.23.11 論理シミュレーションの実行	141
A.23.12 波形ファイルの確認	141
A.23.13 CPU 設計にチャレンジしよう	141
A.24 論理設計者の心得	142

第 1 章

CPU コアを設計して FPGA に実装して動かそう

本章ではまず、CPU コアの基本を概観し、具体的事例として、完全チューリングマシン、すなわちいかなるアルゴリズムも実現可能な CPU コアのアーキテクチャと命令セットを解説します。ちょっと面白いアーキテクチャの CPU です。この CPU の専用アセンブラーと命令シミュレータも作ります。そして、メモリや周辺機能を含めたシステム全体と、それらの具体的な論理設計内容について解説します。そしてこのシステムを FPGA に実装して色々なプログラムを動作させて楽しんでみたいと思います。

圓山 宗智 (munetomo@ishi-kai.org)

1.1 CPU を作ろう！

論理設計の楽しいところは自分で CPU(Central Processing Unit) を構築できることです。例えば RISC-V の命令セットを持つ CPU を独自開発すれば、C コンパイラ、OS、開発環境、デバッグ環境などのエコ・システムを手にすることができる、実用コアとしての活用ができるでしょう。また、AI 处理に特化した命令を追加したり、完全独自のアーキテクチャを持つ超高性能 CPU を設計するのも楽しいでしょう。本項からは、RISC-V よりはるかに小規模ですが、あらゆる処理が可能な完全チューリング・マシンを設計して、実際に FPGA や実シリコンで動作させるまでの過程を解説します。

1.2 コンピュータの歴史とチューリング・マシン

現在、主に使われているコンピュータの仕組みが、どういう過程を経て考案されたのか、その歴史を追ってみましょう。

1.2.1 数学の不完全性定理の提唱

1900 年にドイツの数学学者ダフィット・ヒルベルト (David Hilbert, 1862 年～1943 年) が“決定問題”を提唱しました。決定問題とは、数学の命題を論理式で表し、その論理式が正しいかどうか、またその論理式が定理かどうかを、数学的に論証するアルゴリズムを開発することでした。しかし 1930 年に、オーストリア・ハンガリー帝国のクラウス・ゲーデル (Kurt Gödel, 1906 年～1978 年) が、数学は不完全であり、世界の全てを記述できるなどということはあり得ない、という不完全定理を提唱したのです。[1]

1.2.2 チューリング・マシンによる考察

ゲーデルの不完全定理は、数学の論理学の側面で提唱されているため、1936 年に英国の数学者アラン・チューリング (Alan Mathison Turing, 1912 年～1954 年) が、代数学でも成立するかどうかの考察を行いました。その結果、チューリングは、1936 年に論文“計算可能数について - 決定問題への応用”を発表し、“どんな計算も可能なマシンは、本当にどんな計算もできるのか？”を考察するための、仮想的な計算機チューリング・マシンを考案します。ただし、コンピュータを作ろうと思ってこの概念を考えたわけではなく、あくまでも数学の決定問題の考察のためでした。[2] [3]

チューリング・マシンの基本的な抽象概念を図 1.1 に示します。構成は、一本のテープと、読み書きヘッドが 1 つあるだけです。

チューリング・マシンの動作としては、以下の 3 つが定義されています。有限オートマトンの状態遷移定義（プログラムに相当）とヘッドから読み取った文字に応じて動作を決めていきます。[4]

1. 文字を読み取る
2. ヘッドを右か左に 1 文字分のみ移動
3. テープに文字を書く

数学の形式体系はすべてこの仮想機械の動作に還元できるといわれています。

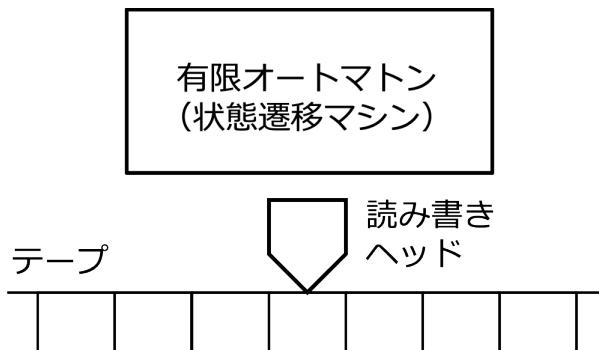


図 1.1: チューリング・マシン

1.2.3 停止性問題を解くアルゴリズムは存在しないことの証明

チューリングがこのマシンを使って示そうとしたのは、“停止性問題の決定不能性”すなわち停止性問題を解くアルゴリズムは存在しないこと、です。別にコンピュータを作ろうとしたわけではありません。

停止性問題とは、有限時間で結果が出せるか、またはその判定が可能なアルゴリズムが存在するかどうか、を問う問題のことです。この停止性問題は次のように言い換えることもできます。

プログラム A にデータ x を入力して実行することを $A(x)$ と書き、 $A(x)$ が y を出力して停止するとき $y = A(x)$ と書くことにします。停止性問題とは、“プログラム A とデータ x が与えられたとき、 $A(x)$ が停止するか、すなわち有限時間で結果を出すかどうかを決定せよ”という問題になります。

ここで、“データとしてのプログラム A”も A と書くことにします。例えばプログラム A、B があったとして “A(B)” は、“プログラム A に、データとしてのプログラム B を入力として渡す”の意味になります。このときの“停止性問題の決定不能性”は次の命題に言い換えることができます。

【命題】 全てのプログラム A と全てのデータ x に対し、 $A(x)$ が停止するかどうかを決定するプログラム H は存在しない。すなわち、以下の性質を満たすプログラム H は存在しない。

- プログラム $A(x)$ の実行が停止するなら プログラム $H(A,x)$ は YES を出力する。

- プログラム $A(x)$ の実行が停止しないなら プログラム $H(A,x)$ は NO を出力する。

【証明】 停止性問題を解くプログラム H が存在すると仮定します。 $M(A)$ を、 $H(A,A)=YES$ なら $M(A)$ 自身は停止せず、 $H(A,A)=NO$ なら 0 を出力して $M(A)$ を停止するプログラムとします。ここで、 $M(A)$ は、 $H(A,A)$ と無限ループを組み合わせれば作る事はできます。このとき、 $M(M)$ は結果を出力して停止するでしょうか？

- $M(M)$ が停止したとすると、 M の定義より $H(M,M)=NO$ であり、 H の定義より $H(M,M)=NO$ となるのは $M(M)$ が停止しないときのみなので矛盾します。
- $M(M)$ が停止しないとすると、 M の定義より $H(M,M)=YES$ であり、 H の定義より、 $H(M,M)=YES$ となるのは $M(M)$ が停止するときのみなので、矛盾します。

よって停止性問題を解くプログラム H は存在しないことが証明されました。結局、数学者自身が数学は不完全であることを証明することになったのです。

1.2.4 プログラムとデータを一体化

ここまで、長々とチューリング・マシンとその停止性問題を議論しましたが、本稿で注目したいのは、停止性問題ではありません。このチューリング・マシンにおける思考過程において、プログラムとデータを一体化して考察していたことが重要です。この考え方は斬新であり、これに衝撃を受けて、プログラム内蔵式コンピュータを考案した人物が次に紹介するフォン・ノイマンです。

1.3 コンピュータの基礎

1.3.1 ノイマン型コンピュータの誕生

ハンガリー出身の米国の数学者フォン・ノイマン (John von Neumann, 1903年～1957年) は、チューリング・マシンの思考においてプログラムとデータが区別なく一体化できることをヒントにして、1946年にプログラム内蔵方式のディジタルコンピュータを考案しました。まさに冒頭説明した現在のPCを持つCPUそのものです。これをノイマン型コンピュータといい、現在のコンピュータの基本原理になっています。こうした経緯から、計算機は数学者の思考過程の副産物にすぎない、ともいわれます。[5]

1.3.2 CPUが本質的にやっていること

CPUの仕事は実はとても単純です。図1.2に示すように、(a) ある記憶素子上のデータを別の記憶素子に移すこと(コピー)と、(b) ある記憶素子上のデータを加工して別の記憶素子に移すこと(演算)の2通りしかありません。記憶素子は、メモリや、通信などの周辺機能内のレジスタです。ただ、これらの間でひたすらデータをやりとりするだけです。このやりとりのシーケンスがプログラムである、ということです。

1.3.3 CPUシステムの基本構造

ノイマン型コンピュータは、基本的に図1.3のようになります。CPU(Central Processing Unit)が多くの記憶素子に取り囲まれてそのシステムを構成します。プログラムを格納するメモリ、データを格納するメモリ、周辺機能を構成するレジスタなどが、CPUとの間でバスで結ばれます。バスの役割は、各メモリやレジスタにアドレス(住所)をあらかじめアサインし、CPUがメモリやレジスタのアクセス要求(リードやライト)をするときにアドレスを出力するので、そのアドレスに従って正しい相手をアクセスさせることです。

CPUの内部は一般的に図1.4のようになっており、プログラム・カウンタ PC(Program Counter)を持つ

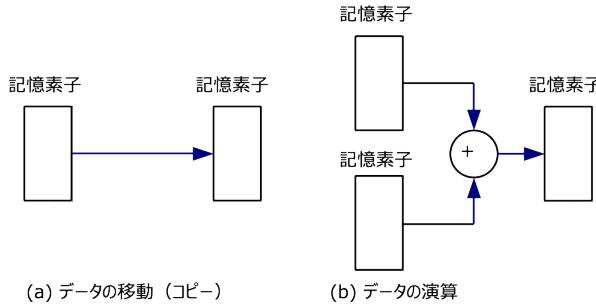


図 1.2: CPU が本質的にやっていること

ています^{*1}。PCは実行しようとするプログラム・メモリ内に格納されている命令のアドレスを示しています。CPUはPCが指す命令を取り込み(命令フェッチ)、その命令の意味を解読し(命令デコード)、その結果に応じて、CPUを取り巻く記憶素子の間でデータのやりとり(リード・ライト)を行います。メモリや周辺機能のアクセスには時間がかかることがあるので(アクセス・タイムが長い)、CPU内部に高速な一時記憶レジスタを多く置いて、できるだけその中で演算処理をして、完成した答えをメモリや周辺機能に渡すなどして性能を向上させています。

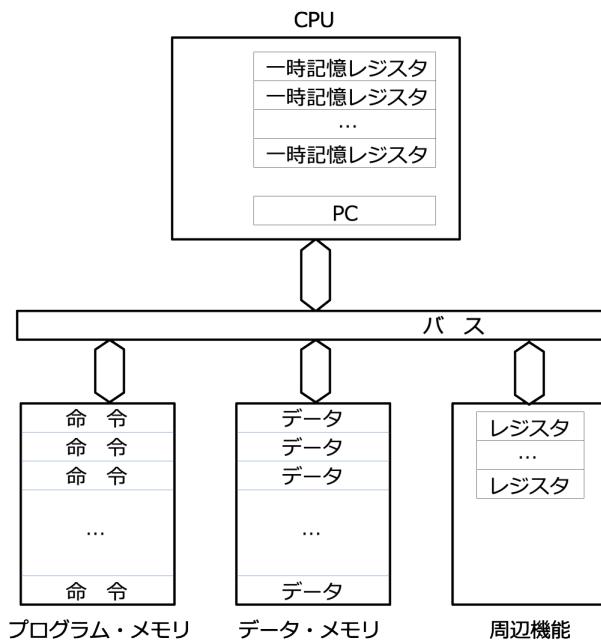


図 1.3: CPUシステムの構成

^{*1} 1980年代から1990年代にかけて、PCを持たないいわゆる非ノイマン型コンピュータの研究が盛んに行われた時期がある。その代表例がデータ・フロー・コンピュータで、データにどのデータとどういう処理をするかという情報を持たせたトーケンとしてネットワーク内に流すと、空いている演算資源に自動的に転送され、演算資源が複数あれば最適な並列処理ができるという触れ込みだった。PCや命令という考え方ではない。データ・フロー・コンピュータは一部商品化された例はあるが、本格的な普及には至らなかった。トーケンの処理内容の粒度が細かいとネットワークでネックになって並列性能を出せにくかったと思われる。ただ、この考え方には、現代ではCSP(Communicating Sequential Processes)コンピューティングとして受け継がれていて、通信パケットの粒度を大きくして並列処理性能を向上させるアーキテクチャに活用されている。

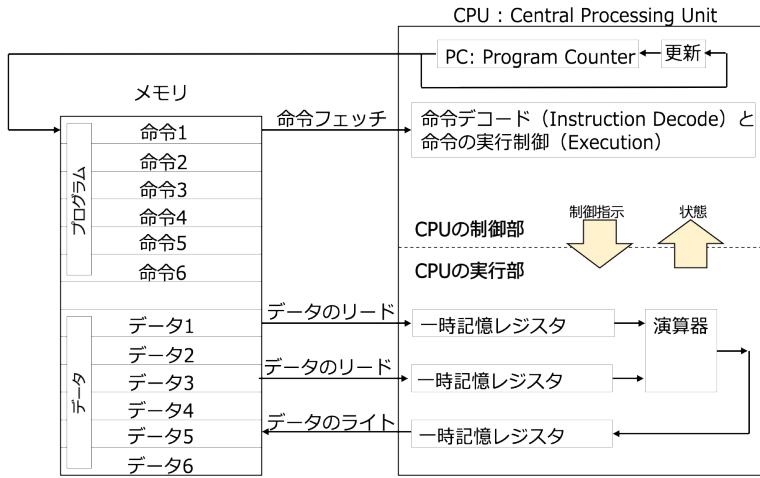


図 1.4: CPU の内部構造

1.3.4 CPU の性能向上

命令のシーケンシャル実行

黎明期のCPUアーキテクチャは、図1.5のように命令をシーケンシャルに実行していました。黎明期のCPUは命令処理内容が複雑なCISCアーキテクチャであり、命令ごとにステージ数がバラバラであり、このような方式が採用されました。この実現には、論理回路としてマイクロ・プログラム方式を使うと簡単に実現できました。

時間 →																
命令1					命令2				命令3				命令4			
T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17
F	D	E	M _r	W	F	D	E	W	F	D	m	W	F	D	E	M _w
LD R2,@(R1,R0)	ADD R3,R2,R1	MUL R6,R5,R4	ST @(R1,R0),R3													
レジスタR1とR0を加算した値をアドレスとするメモリ上に格納されている値をリードしてレジスタR2にロード	レジスタR2とR1を加算して、その結果をレジスタR3に格納	レジスタR5からR4を乗算して、その結果をレジスタR6に格納	レジスタR1とR0を加算した値をアドレスとするメモリ上にレジスタR3をライトしてストアする													

図 1.5: 命令のシーケンシャル実行

命令のパイプライン実行

1命令は複数ステージから構成されているので、その完結には複数サイクルかかりますが、命令内の各ステージが使うハードウェア・リソースはある程度限定されているので、基本的に命令間の異なるステージ同志の間では使うリソースが競合しません。そのため、命令をまたがる異なるステージ同志はオーバラップが可能という発想により、図1.6に示すパイプライン動作方式が生まれました。

命令をシーケンシャルに実行する方式に比べれば効果は絶大です。個々の命令が、1サイクル・ピッチで完結していくので、見かけ上、1命令あたり1サイクルしかかかりません。すなわち(CPI=1)を実現できます。現代のほとんど全てのCPUは、パイプライン動作方式を採用しています。

ただし、命令の前後関係に応じて制御が複雑になります。例えば、図1.6のT4では、LD命令がR2に格納

する予定のメモリをリード中であり、その次のADD命令でR2を使うので、ADD命令の実行を遅らせる(ストールさせる)制御が必要になります。

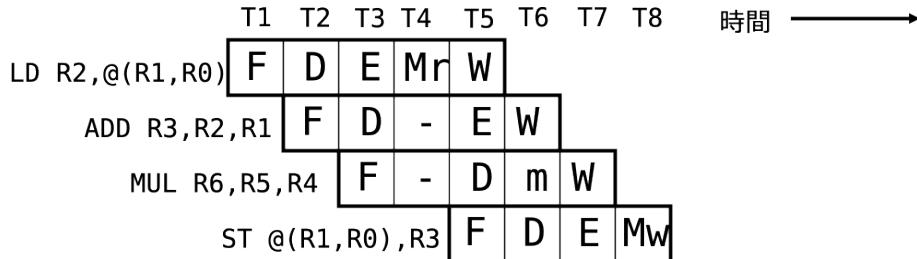


図 1.6: 命令のパイプライン実行

CPUのさらなる高性能化

CPUのさらなる高性能化のためのアーキテクチャの工夫は様々あります。本稿では詳細説明しませんが、CPU単体で高性能化するアーキテクチャとしては、分岐予測、アウト・オブ・オーダー実行、スーパスカラなどの命令レベルでのミクロな並列化があります。最近では、半導体プロセスの進化とともに、シリコン・チップの上にロジック・ゲートを大量に搭載可能になったので、命令レベルのミクロな並列化よりは、大量のコアや演算器を搭載して実現するタスク並列やデータ並列などマクロな並列化の方が有利になってきました。

1.4 CPUを作ろう

本稿の目的は、CPUコアの設計について具体的な事例を説明することです。まず、そのCPUのネタをどうするかを考えました。一旦設計できたら、FPGAに実装してリアルタイムな動作を確認し、さらにこのCPUをTiny Tapeoutでシリコンとして試作するところまで実現したいと思いました。

1.4.1 どういうCPUを作ろうか？

最近のFPGAはそこそこの論理規模を実装できますが、Tiny Tapeoutは論理規模が限られているので、プログラム・メモリやデータ・メモリを内蔵することが厳しく、CPUを実現するなら外部にメモリを置く必要があります。さらに、Tiny Tapeoutは、外部端子数に制約があります。特にユーザ用の外部端子は、入力×8本、出力×8本、入出力×8本の合計24本しかありません。外部バスを引き出そうとしても、アドレス16本、データ8本だけで終わってしまい、バス・アクセスを示すストローブ信号を出力することができません。

1.4.2 外部メモリをどうするか？

Tiny Tapeoutのような外部端子本数に制約がある場合は、外部メモリとしてシリアルROMやシリアルRAMを使うのが一般的です。最もシンプルなインターフェースは、SPI(Serial Peripheral Interface)の4線式通信です。インターフェース信号は、チップ・セレクトCS_N、シリアル・クロックSCK、シリアル出力データSO、シリアル入力データSIの4本のみです。ただし、多ビットのアドレス情報やデータ情報を1ビットのシリアル線で転送するので、性能があまり上がりません。このため市販のシリアルROMやシリアルRAMはQSPI(Quad SPI)インターフェースを備えています。インターフェース信号は、チップ・セレクトCS_N、シリアル・クロックSCK、シリアル入出力データSIO0～SIO3の6本です。アドレス情報やデータ情報を4ビットずつ転送できるので、単純なSPI方式よりも性能が向上します。

1.4.3 キャッシュ・メモリの活用

また、シリアル ROM やシリアル RAM で、アドレスをあちこちランダム・アクセスする場合は、毎回アドレス情報を送る必要があり性能が上がりません。一方、連続アドレスをアクセスするシーケンシャル・アクセスをする場合は、アドレス情報は最初の 1 回だけ送ればいいので比較的性能を出しやすくなります。一般的な CPU は、メモリをランダム・アクセスする傾向にあるため、シリアル ROM やシリアル RAM との相性がよくありません。しかし、命令やデータのアクセスがほぼ同じアドレス位置に集中する場合（ローカルティが高い場合）は、小さい容量のキャッシュ・メモリを CPU とメイン・メモリの間に挿入して、繰り返しアクセスされる命令やデータをキャッシュ・メモリに置いて外部メモリのアクセス頻度を下げることで性能低下を防げます。さらに、キャッシュ・メモリ内にアクセスしたい命令やデータがない場合は、外部メモリとキャッシュ・メモリの内容を入れ替えますが、その際は、外部メモリをシーケンシャルにアクセスできるので、シリアル ROM やシリアル RAM との相性もよくなります。このような仕組みを導入することで、外部メモリをシリアル ROM やシリアル RAM で実現することはあまり問題ではなくなります。

1.4.4 CPU の規模とアーキテクチャをどうするか？

Tiny Tapeout を使って RISC-V を実装している例は複数あります。RISC-V は、最もシンプルな命令セット RV32I でも 32 ビット汎用レジスタが 32 本必要なので、それだけで 1024 ビット分のフリップ・フロップが必要です。汎用レジスタが 16 本に縮小できる命令セット RV32E でも汎用レジスタだけで 512 ビット分のフリップ・フロップが必要です。CPU はこれ以外に、プログラム・カウンタ PC、命令デコーダ、命令実行ユニットなど多くの機能ブロックが必要であり、このため RISC-V を Tiny Tapeout で実装した事例を見ると、CPU 自体は最小構成でも、最大区画 8x2 を使っているものが多いようです。

今回設計する CPU は、できるだけ論理規模が小さいシンプルなものにする必要がありますが、それでもそのアーキテクチャとしては、いかなる問題も解ける完全チューリング・マシンにしたいと思いました。

1.5 完全チューリング・マシン bfCPU を作る

シンプルな完全チューリング・マシンとして有名なものに、1993 年にスイスの Urban Müller が提唱した Brainf*ck(*は u) というアーキテクチャがあります [6]。1 文字伏字にしたのは、少々品がない名称のためであり、本稿ではこのアーキテクチャの CPU を bfCPU と表現することにします。bfCPU はわずか 8 個の命令しか持たない 8 ビット CPU です。しかし、これだけで任意のアルゴリズムを実現できる優れものです。本稿ではこの CPU を実際に設計し、FPGA に実装して動作させ、さらに Tiny Tapeout でシリコン化して楽しんでみたいと思います。

1.6 bfCPU の設計データ

bfCPU の設計関連データは全て、<https://github.com/munetomo-maruyama/bfCPU.git> に格納しています。Ubuntu 環境なら、

```
$ git clone https://github.com/munetomo-maruyama/bfCPU.git
```

で bfCPU のリポジトリをダウンロードできます。以下、このリポジトリを参照して説明するときは、“リポジトリ内の bfCPU/...” のように場所を指定することにします。

1.7 bfCPU のアーキテクチャ

1.7.1 bfCPU の深淵さ

bfCPU のプログラムはその可読性・記述性がとても低く、難解プログラミング言語の一種に挙げられており、加算を実行するだけでもパズルのようです。

難解プログラミング言語と bfCPU について、Daniel Temkin は以下のように評しています [7]。

- 難解プログラミング言語 (Esoteric programming languages : esolangs) は、プログラミングの世界から生まれた一種の表現芸術である。それは、言語の役割を単なる“命令と制御の道具”から、“文化的な表現や既存の枠組みへの拒絶”へと作り変える試みもある。なかでも群を抜いて悪名高いのが bfCPU の言語 (命令) だ。この言語は、コンピュータ本来のロジックをむき出しのままプログラマーに突きつける一方で、人間の言葉と機械語との距離を縮めることを徹底的に拒む。そうして私たちを滑稽なまでの論理の迷宮へと引きずり込むのだ。bfCPU は究極のミニマリズムを貫いており、命令として認識できるのはわずか 8 つの記号のみしかない。マシンを動かすために“print”といった言葉を使うことは許されず、すべては記号の羅列で記述される。皮肉なことに、この“単純さ”こそが、扱う上での圧倒的な複雑さを生み出す。たとえば“32”という数字を直接表す記号すら存在しない。プログラマーはメモリの箱を一つづつ行ったり来たりしながら、値を保存する場所を探し、プラスとマイナスの記号を延々と書き連ねて、32 に到達するまでその場所の数値を増やしたり減らしたりしなければならない。こうなると、32 という数字はもはや単なる“決まった定数”ではなく、自らの知略で勝ち取るべき“リソース”へと変貌する。4 を 8 回足すループを組んで 32 を作るのか、あるいは、1 バイトの上限が 256 であることを利用して、ゼロの壁を越えて逆算し目的の値へと着地させるのかなど、なんでもアリである。32 という一つの数字をひねり出す行為そのものが、プログラマーにとっての“独自のスタイル”や“表現のこだわり”を披露する場となる。

ようするに、bfCPU のプログラムを組む時、私たちは意識せずうっかりニマニマしてしまうということを言っています。実際、筆者も bfCPU に接した時、ニマニマしていました。

1.7.2 bfCPU の基本構成

bfCPU の基本構成を図 1.7 に示します。bfCPU の内部リソースは、プログラム・カウンタ PC とデータ・メモリのポインタ PTR だけです。PC はプログラム・メモリに格納されている命令のアドレスを示します。PC が指すプログラム・メモリ上の命令をフェッチし、その命令にしたがって、データ・メモリのポインタ PTR を操作したり、PTR が指すデータ・メモリのアドレス内のデータをアクセスします。データ・メモリの一つのアドレスには 8 ビット幅 (バイト・サイズ) のデータを格納します。

また、外部とのインターフェースのための入出力機構があり、今回設計する bfCPU では、外部からの入力は UART(Universal Asynchronous Receiver/Transmitter : 調歩同期式シリアル通信インターフェース) への受信データ、外部への出力は UART からの送信データとします。bfCPU の構造はこれだけのシンプルなものです。

1.7.3 bfCPU の命令セット

bfCPU の命令セット一覧を表 1.1 に示します。bfCPU の命令としては、命令コード (2 進数) が 0000 ~ 0111 の 8 つのみです。命令数が 8 つなので命令コードは 3 ビットで足りるのですが、今回設計する bfCPU では 2 命令 (reset と nop) を追加して命令コード長は 4 ビットにしました。これらの命令コードは全て処理内容を示すオペコードのみで処理対象を指定するオペランドがないシンプルな構成になっています。

bfCPU のオリジナルの命令記述は、表 1.1 の“記号”的の一文字です (<、>、+、- など)。この文字を隙間なく、またはスペースや改行が挿入しながら並べたものがプログラムになり、これが難解プログラムと言われる所以です。後述しますが、本稿では bfCPU のプログラム開発用にアセンブラーと命令シミュレータ **bfTool** を開発しました。このアセンブラーは bfCPU のオリジナル命令 (1 文字) も受け付け、世の中に大量に公開されている

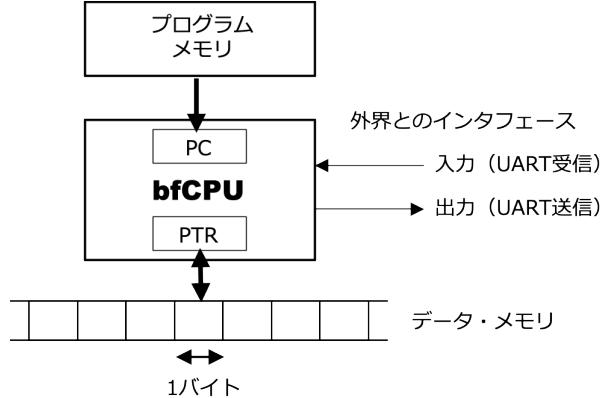


図 1.7: bfCPU の基本構成

bfCPU プログラムをそのまま使うことができるようになっていますが、表 1.1 の “ニーモニック” の列の記述も可能にし、可読性の向上を計っています。表 1.1 の “等価 C 言語” の列は、各命令の動作を C 言語で表現したものです。なお、オリジナルの bfCPU は、ASCII 文字 (<、>、+、-など) そのものをプログラムとして使いますが、本稿で設計する bfCPU は各命令を 4 ビットの命令コードに置き換えています。

表 1.1: bfCPU の命令セット

命令コード	記号	ニーモニック	等価C言語	意味
0000	>	pinc / p++	ptr++	データ・ポインタ PTR を 1 つインクリメントする (右隣のデータ・メモリを指す)
0001	<	pdec / p--	ptr--	データ・ポインタ PTR を 1 つデクリメントする (左隣のデータ・メモリを指す)
0010	+	inc	(*ptr)++	データ・ポインタ PTR が指すデータ・メモリ内のバイト・データを 1 つインクリメントする (1 を加算)
0011	-	dec	(*ptr)--	データ・ポインタ PTR が指すデータ・メモリ内のバイト・データを 1 つデクリメントする (1 を減算)
0100	.	out	putchar(*ptr)	データ・ポインタ PTR が指すデータ・メモリ内のバイト・データを出力する (UARTから送信する)
0101	,	in	*ptr=getchar()	バイト・データを入力して、データ・ポインタ PTR が指すデータ・メモリにライトする (UARTが受信するまで待つ)
0110	[begin	while(*ptr) {	データ・ポインタ PTR が指すデータ・メモリ内のバイト・データがゼロなら、次の命令に進まずに、このbegin命令 ([) の先にあるネスト深さが対応するend命令 (]) の次の命令までジャンプする。非ゼロなら、このbegin命令 ([) の次の命令に進む。
0111]	end	}	データ・ポインタ PTR が指すデータ・メモリ内のバイト・データが非ゼロなら、次の命令に進まずに、このend命令 (]) の前にあるネスト深さが対応するbegin命令 ([) の次の命令までジャンプする。ゼロなら、このend命令 (]) の次の命令に進む。
1000	なし	reset	なし	PC=0, PTR=0 に初期化し、データ・メモリ全体をゼロ・クリアして、プログラムの先頭から実行する
1001	なし	なし	なし	リザーブ
1010	なし	なし	なし	リザーブ
1011	なし	なし	なし	リザーブ
1100	なし	なし	なし	リザーブ
1101	なし	なし	なし	リザーブ
1110	なし	なし	なし	リザーブ
1111	なし	nop	なし	何も実行しない

以下、個々の命令の動作を説明します。なお、bfCPU はリセット後、データ・メモリの内容を全てゼロ・クリアし、PTR と PC をゼロ・クリアし、プログラム・メモリ上の 0 番地の命令から実行開始します。

【>】(pinc/p++)

データ・ポインタ PTR を一つインクリメントします。図 1.7 のデータ・メモリの右隣のデータ・メモリを PTR が指すようになります。この命令を実行したら PC を一つインクリメントして次命令の処理に進みます。

【<】(pdec/p--)

データ・ポインタ PTR を一つデクリメントします。図 1.7 のデータ・メモリの左隣のデータ・メモリを PTR が指すようになります。この命令を実行したら PC を一つインクリメントして次命令の処理に進みます。

【+】(inc)

データ・ポインタ PTR が指すデータ・メモリをリードして、1を加算して、同じデータ・メモリの位置にライトします。すなわち、データ・ポインタ PTR が指すデータ・メモリの内容を1つインクリメントします。この命令を実行したら PC を一つインクリメントして次命令の処理に進みます。PTR は変更しません。

【-】(dec)

データ・ポインタ PTR が指すデータ・メモリをリードして、1を減算して、同じデータ・メモリの位置にライトします。すなわち、データ・ポインタ PTR が指すデータ・メモリの内容を1つデクリメントします。この命令を実行したら PC を一つインクリメントして次命令の処理に進みます。PTR は変更しません。

【.】(out)

データ・ポインタ PTR が指すデータ・メモリをリードして、外部に1バイト分のデータを出力します。今回の bfCPU の設計では、UART から1バイト送信します。UART の送信バッファが空くまでウェイトが自動的に入ります。この命令を実行したら PC を一つインクリメントして次命令の処理に進みます。

【,】(in)

外部からデータを1バイト分入力して、データ・ポインタ PTR が指すデータ・メモリにライトします。今回の bfCPU の設計では、UART から1バイト受信します。UART の受信バッファにデータが入るまでウェイトが自動的に入ります。この命令を実行したら PC を一つインクリメントして次命令の処理に進みます。

【[]】(begin)

データ・ポインタ PTR が指すデータ・メモリ内のデータがゼロなら、次命令に進まずに (PC をインクリメントせずに)、この begin 命令 ([]) の先にあるネスト深さが対応する end 命令 ([]) の次の命令までジャンプします。データ・ポインタ PTR が指すデータ・メモリ内のデータが非ゼロなら、PC を一つインクリメントしてこの begin 命令 ([]) の次命令の処理に進みます。言い換えると、begin 命令 ([]) 実行時にデータ・ポインタ PTR が指すデータ・メモリ内のデータが非ゼロであれば、その begin 命令 ([]) とネスト深さが対応する end 命令 ([]) の間の命令処理を繰り返します。

【[]】(end)

データ・ポインタ PTR が指すデータ・メモリ内のバイト・データが非ゼロなら、次の命令に進まずに PC をインクリメントせずに)、この end 命令 ([]) の前にあるネスト深さが対応する begin 命令 ([]) の次の命令までジャンプします。データ・ポインタ PTR が指すデータ・メモリ内のデータがゼロなら、PC を一つインクリメントしてこの end 命令 ([]) の次命令の処理に進みます。

この end 命令 ([]) は、必ずしもデータ・ポインタ PTR が指すデータ・メモリ内のデータをチェックする必要はなく、常に対応する begin 命令 ([]) の次の命令までジャンプするだけでも構いません。ただその場合、もし PTR が指すメモリ内のデータがゼロだったら begin 命令 ([]) に戻ったあとまたすぐにこの end 命令 ([]) の次命令にジャンプする必要があります、余計な処理が入ってしまいます。このため、end 命令 ([]) も、データ・ポインタ PTR が指すデータ・メモリの内容をチェックしています。

(reset)

bfCPU をリセットします。データ・メモリの内容を全てゼロ・クリアし、PTR と PC をゼロ・クリアし、プログラム・メモリ上の0番地の命令から実行開始します。

(nop)

何も処理せず、PC を一つインクリメントして次命令の処理に進みます。

1.7.4 begin 命令 ([]) と end 命令 ([]) はどうやってジャンプできるのか？

一般的の CPU の命令コードには、命令が何をやるかを示すオペコードと、データ転送するメモリのアドレス、加算に使う定数値、分岐命令の分岐先アドレス情報など、操作対象やアドレス位置を示すオペランドをもちます。しかし、bfCPU の 4 ビットの命令コードにはオペコードしかありません。bfCPU は命令の操作対象として、データ・ポインタ PTR とデータ・メモリしかないのでオペコードだけでもよさそうですが、begin 命令 ([]) と end 命令 ([]) の分岐先を決めるオペランドがない点は問題になりそうです。

bfCPU の命令コードのビット長をもっと長くして、分岐命令の分岐先情報を入れることができれば、bfCPU プログラムをアセンブラーやコンパイラが機械語を生成するとき、ネストが対応しあう begin 命令 ([]) と end 命令 ([]) のアドレスはわかるので、それらを命令コードに仕込むことができるでしょう。しかし、今回の事例ではシンプル化のため、命令コードは全て 4bit に収めることにしたので、分岐命令 (begin と end) の命令コードには分岐先情報は入っていません。

では、begin 命令 ([]) と end 命令 ([]) はどうやって分岐先に辿り着けるのかですが、その答えはプログラム・メモリのサーチです。

begin 命令 ([]) が end 命令 ([]) の次命令に分岐することになったら、begin 命令 ([]) の実行内容として、内部レジスタ INDENT を 0 クリアしてからプログラム・メモリを PC を増やしながらサーチしていきます。もしまだ begin 命令 ([]) があったらネストが深くなつたと判断して INDENT をインクリメントします。INDENT が 0 より大きい時に end 命令 ([]) を見つけたらこれは当初の begin 命令 ([]) に対応するものではないので INDENT をデクリメントしてさらにサーチを続けます。最終的に INDENT が 0 の時に end 命令 ([]) を見つけたら、それがネストが対応するものと判断して、その次の命令から実行を継続します。

end 命令 ([]) が begin 命令 ([]) の次命令に分岐することになったら、end 命令 ([]) の実行内容として、内部レジスタ INDENT を 0 クリアしてからプログラム・メモリを PC を減らしながらサーチしていきます。もしまだ end 命令 ([]) があったらネストが深くなつたと判断して INDENT をインクリメントします。INDENT が 0 より大きい時に begin 命令 ([]) を見つけたらこれは当初の end 命令 ([]) に対応するものではないので INDENT をデクリメントしてさらにサーチを続けます。最終的に INDENT が 0 の時に begin 命令 ([]) を見つけたら、それがネストが対応するものと判断して、その次の命令から実行を継続します。

このように、begin 命令 ([]) と end 命令 ([]) は 1 発でジャンプできる命令ではありませんので、低速な QSPI SRAM に格納されているプログラム・メモリをサーチする方法は性能的に不利です。しかし、後述するように、命令キャッシュを備えることで、できるだけ外部の QSPI SRAM をアクセスせずに済ませて性能低下を最小限に留めています。

1.7.5 bfCPU のプログラムの注意点

begin 命令 ([]) と end 命令 ([]) はネスト深さがいくら多くてもいいですが、必ず対応させておく必要があります。また、データ・メモリのマスクのデータ幅は 8 ビット (バイト) 長ですが、値が 0xFF の時に 1 を加えると 0x00 になり、値が 0x00 の時に 1 を減ずると 0xFF になります。キャリー / ポローヤオーバフローが発生したことを示すフラグはありません。

1.8

bfCPU のアセンブラー・シミュレータ bfTool

bfCPU のプログラムを開発したり、PC 上でその命令動作をシミュレーションするための開発ツール **bfTool** を作成しました。リポジトリ内の bfCPU/bfTool の下にあります。

1.8.1 bfTool のビルド方法

ソース・コードは src の下に格納してあります。標準 C で書かれているので gcc があればコンパイルできますが、gcc 意外に flex、bison も必要です。flex と bison は下記コマンドでインストールできます。

```
$ sudo apt install flex bison
```

準備できたら **bfTool** を下記コマンドでビルドしてください。

```
$ cd bfCPU/bfTool  
$ make
```

ディレクトリ bfTool の下に実行ファイル bfTool ができます。このファイルの場所を環境変数 \$PATH にセットしておいてください。そのためには、`~/.bashrc` に

```
export PATH="$HOME/ISHI_Kai/bfCPU/bfCPU/bfTool:$PATH"
```

を追加し、

```
$ source ~/.bashrc
```

で有効化してください。

1.8.2 bfTool の関連ファイル

前述したプログラムのいくつかはサンプル・プログラムとしてディレクトリ bfTool/samples の下に格納しています。各ファイルの拡張子の意味は以下の通りです。アセンブル・ソース・ファイル (.asm) には、コメントを C 言語風にスラッシュ 2 個 (//) で記述できます。

- **.asm** : アセンブル・ソース・ファイル
- **.lis** : アセンブル・リスト・ファイル
- **.hex** : アセンブル・オブジェクト・ファイル (Intel Hex フォーマット)
- **.v** : アセンブル・オブジェクト・ファイル (Verilog のメモリ初期化ファイル)
- **.sim** : シミュレーション・ログ・ファイル

1.8.3 プログラムのアセンブル方法

ソース・プログラムをアセンブルするには、下記のようにコマンド入力してください。

```
$ cd bfTool/samples  
$ bfTool filename.asm
```

これにより、filename.lis、filename.hex、filename.v が生成されます。

filename.lis は、アセンブル結果をリスト表示しており、各命令のアドレスとコードの対応をリスト化しています。さらに 1 文字命令はインデントをつけたニーモニック命令で表示しています。

filename.hex は、アセンブルしてできたオブジェクト・コードです。Intel Hex フォーマットで、バイト単位でデータが並びますが、bfCPU の命令コードは 4 ビット幅なので、バイトの下位 4 ビットに下位アドレス側のコードが、バイトの上位 4 ビットに上位アドレス側のコードが格納されます（リトル・エンディアン）。bfTool で命令動作をシミュレーションするときは、この filename.hex を入力します。

filename.v は、アセンブルしてできたオブジェクト・コードです。こちらは、System Verilog で記述された bfCPU システムを論理シミュレーションで機能検証をするときにプログラム・メモリを初期化するために使います。

1.8.4 プログラムのシミュレーション方法

プログラムのシミュレーションをするときは、オプション “-s” をつけて、filename.hex を指定します。

```
$ cd bfTool/samples
$ bfTool -s filename.hex
```

“reset 命令”に当たると、下記のように表示され Enter キーの入力でもう一度 0 番地から実行を再開します。

```
Hit Enter to Reset
```

シミュレーション実行中に、**CTRL-C** を入力するとシミュレーションを中断し、実行直前の命令とデータ・メモリの最大消費量 (PTR の最大値) を表示します。

```
PC=0x00 ROM[0x00]=0x5 (IN ) Input 8bit Hex Number? ^C
Aborted: MAXPTR=0x0001(1)
```

このシミュレーションを実行するとき、下記のように “in 命令” と “out 命令” の動きを指定できます。

“in 命令” と “out 命令” でバイナリ数値を入出力する場合

“in 命令” と “out 命令” でバイナリ数値を入出力する場合はオプション “-s” のみでコマンドを実行してください。

```
$ bfTool -s filename.hex
```

下記のように、“in 命令” に当たったらバイト値 (0x のない 16 進数) の入力待ちになり、“out 命令” に当たったらバイト値 (0x 付きの 16 進数と括弧内に 10 進数) を表示します。

```
PC=0x00 ROM[0x00]=0x5 (IN ) Input 8bit Hex Number? 4
PC=0x02 ROM[0x02]=0x5 (IN ) Input 8bit Hex Number? 5
PC=0x0b ROM[0x0b]=0x4 (OUT ) --> PTR=0x01 RAM[0x01]=0x09 OUTPUT=0x09( 9)( )
```

“in 命令” と “out 命令” で ASCLi 文字列を入出力する場合

“in 命令” と “out 命令” で ASCLi 文字列を入出力する場合はオプション “-t” を付けてコマンドを実行してください。

```
$ bfTool -st filename.hex
```

下記のように、“in 命令” に当たったら ASCII 文字の入力待ちになり、“out 命令” に当たったら ASCII 文字を出力します。

```
Hello World!      ASCII出力
Hit Enter Key
```

```
123      ASCII入力+\n
123      ASCII出力
Hit Enter Key
```

命令実行ログ・ファイルの出力

命令シミュレーション実行時に、オプション “-g” をつけると、命令の実行ログ・ファイル filelist.sim を出力できます。オプション “-b” をつけると、実行ログを stdout に出力します。オプション “-g”、“-b”、“-t” は混在することもできます。

```
$ bfTool -sg filename.hex
$ bfTool -sb filename.hex
$ bfTool -sgb filename.hex
$ bfTool -sgbt filename.hex
```

```
00000 : PC=0x00 ROM[0x00]=0x5 (IN) --> PTR=0x00 RAM[0x00]=0x03 INPUT=0x03( 3)(^C)
00001 : PC=0x01 ROM[0x01]=0x0 (P++) --> PTR=0x01 RAM[0x01]=0x00( 0)
00002 : PC=0x02 ROM[0x02]=0x5 (IN) --> PTR=0x01 RAM[0x01]=0x04 INPUT=0x04( 4)(^D)
00003 : PC=0x03 ROM[0x03]=0x1 (P--) --> PTR=0x00 RAM[0x00]=0x03( 3)
00004 : PC=0x04 ROM[0x04]=0x6 (BEGIN) --> PTR=0x00 RAM[0x00]=0x03( 3)
00005 : PC=0x05 ROM[0x05]=0x3 (DEC) --> PTR=0x00 RAM[0x00]=0x02( 2)
00006 : PC=0x06 ROM[0x06]=0x0 (P++) --> PTR=0x01 RAM[0x01]=0x04( 4)
00007 : PC=0x07 ROM[0x07]=0x2 (INC) --> PTR=0x01 RAM[0x01]=0x05( 5)
00008 : PC=0x08 ROM[0x08]=0x1 (P--) --> PTR=0x00 RAM[0x00]=0x02( 2)
00009 : PC=0x09 ROM[0x09]=0x7 (END) --> PTR=0x00 RAM[0x00]=0x02( 2)
00010 : PC=0x05 ROM[0x05]=0x3 (DEC) --> PTR=0x00 RAM[0x00]=0x01( 1)
00011 : PC=0x06 ROM[0x06]=0x0 (P++) --> PTR=0x01 RAM[0x01]=0x05( 5)
00012 : PC=0x07 ROM[0x07]=0x2 (INC) --> PTR=0x01 RAM[0x01]=0x06( 6)
00013 : PC=0x08 ROM[0x08]=0x1 (P--) --> PTR=0x00 RAM[0x00]=0x01( 1)
00014 : PC=0x09 ROM[0x09]=0x7 (END) --> PTR=0x00 RAM[0x00]=0x01( 1)
00015 : PC=0x05 ROM[0x05]=0x3 (DEC) --> PTR=0x00 RAM[0x00]=0x00( 0)
00016 : PC=0x06 ROM[0x06]=0x0 (P++) --> PTR=0x01 RAM[0x01]=0x06( 6)
00017 : PC=0x07 ROM[0x07]=0x2 (INC) --> PTR=0x01 RAM[0x01]=0x07( 7)
00018 : PC=0x08 ROM[0x08]=0x1 (P--) --> PTR=0x00 RAM[0x00]=0x00( 0)
00019 : PC=0x09 ROM[0x09]=0x7 (END) --> PTR=0x00 RAM[0x00]=0x00( 0)
00020 : PC=0x0a ROM[0x0a]=0x0 (P++) --> PTR=0x01 RAM[0x01]=0x07( 7)
00021 : PC=0x0b ROM[0x0b]=0x4 (OUT) --> PTR=0x01 RAM[0x01]=0x07 OUTPUT=0x07( 7)(^G)
00022 : PC=0x0c ROM[0x0c]=0x8 (RESET) --> PTR=0x01 RAM[0x01]=0x07( 7)
```

1.9 bfCPU のプログラム例

1.9.1 データ・メモリをゼロ・クリアするプログラム

PTR が指すデータ・メモリ内をゼロ・クリアするプログラムの1文字命令版をリスト 1.1 に、二ーモニック版をリスト 1.2 に示します。“[”と“]”で挟まれた“+ 命令”がデータ・メモリがゼロになるまで (0xFF → 0x00 になるまで) インクリメントを繰り返します。普通の一般 CPU はメモリにゼロをライトする命令があって 1 回で処理が終わりますが、bfCPU はこのような“味のある方式”でメモリをゼロ・クリアするのです。

リスト 1.1: データ・メモリをゼロ・クリア (1) (1 文字命令版)

[+]

リスト 1.2: データ・メモリをゼロ・クリア (1) (二ーモニック版)

```
begin
  inc
end
```

同じ処理をするプログラムをリスト 1.3 とリスト 1.4 に示します。こちらは “[” と “]” で挟まれた“-命令”がデータ・メモリがゼロになるまで (0x01 → 0x00 になるまで) デクリメントを繰り返します。ゼロ・クリアしたいデータ・メモリの初期値がいくらだったかに応じて、“+ 命令”を使うプログラムと“-命令”を使うプログラムの実行時間が変わってくるという面白さがあります。

リスト 1.3: データ・メモリをゼロ・クリア (2) (1 文字命令版)

```
[ - ]
```

リスト 1.4: データ・メモリをゼロ・クリア (2) (ニーモニック版)

```
begin
  dec
end
```

1.9.2 加算プログラム

加算プログラムの例をリスト 1.5 に示します。ファイルはリポジトリ bfCPU/bfTool/samples/addition.asm です。データ・メモリの内容をアドレス PTR=0 から順に{c0, c1}とします。“in 命令”により UART から被加数と加数のバイト・データ(バイナリ値)を受け取り、それぞれ c0 と c1 に格納します。そして、begin-end ループの中で、c0 の内容を減らし、同時に c1 の内容を増やします。そして c0 の内容がゼロになれば c1 の内容が加算結果になります。最後にアドレス c1 の内容(バイナリ値)を UART から送信します。

リスト 1.5: 加算プログラム addition.asm(ニーモニック版)

```
in    // ptr=0
p++   // increment ptr
in    // ptr=1
p--   // decrement ptr
begin // Loop
  dec // ptr=0
  p++ // increment ptr
  inc // ptr=1
  p-- // decrement ptr
end
p++   // increment ptr
out   // ptr=1

reset // restart
```

1.9.3 乗算プログラム

乗算プログラムの例をリスト 1.6 に示します。ファイルはリポジトリ bfCPU/bfTool/samples/multiplication.asm です。また、データ・メモリの内容をアドレス PTR=0 から順に{c0, c1, c2, c3}とします。

“in 命令”により UART から被乗数 A と乗数 B のバイト・データ(バイナリ値)を受け取り、それ respective c0 と c1 に格納します。ここで、データ・メモリは{A, B, 0, 0}が格納されます。乗算処理の本体は [->[->+>+<<>]>>[-<<<+>>]<<<] の部分です。この中では、A 回分、B を別の場所に加算していることで乗算を実現しています。

外側のループ [-...] では A がゼロになるまで繰り返します。これは B を加算する作業を A 回繰り返す処理です。

内側のループ >[->+>+<<>] では、B を PTR=2 と PTR=3 に移動し、PTR=1 がゼロになります。この状態でデータ・メモリは{A-1, 0, B, B}になります。B の値が c2 に累積されています。

もう一つの内側のループ >>[-<<<+>>]<<<] では、c3 にある値を c1 の位置に戻します。これにより、c1 の値の復元ができる、次の外側ループの準備が整います。この状態でデータ・メモリは{A-1, B, B, 0}になります。

外側のループ [-...] が A 回繰り返されると、c2 には B が A 回足された結果(A × B)が蓄積されます。最後に、c2 の値(バイナリ値)を UART から送信します。

リスト 1.6: 乗算プログラム multiplication.asm(1 文字命令 + ニーモニック混在版)

```

in    // ptr=0
p++
in    // ptr=1
p--
//
[->[->+>+<<>]>>[-<<+>>]<<<]
//
p++
p++
out   // ptr=2
//
reset // restart

```

1.9.4 Hello World! プログラム

どんなプログラマ言語でも最初はこのプログラムですね。プログラム例をリスト 1.7 に示します。ファイルはリポジトリ bfCPU/bfTool/samples/helloworld.asm です。これは実行すると、UART から ASCII 文字列で "Hello World!\n" を送信します。最後の改行コードは 0x0A です。

リスト 1.7: Hello World! 出力プログラム helloworld.asm(1 文字命令 + ニーモニック混在版)

```

++++++[>++++[>++>++++>++++>+<<<<-]>+>+->>+[<]<-]>>.>---.++++++..+++.>>.<-.<.++.-----.->
->---.=>+.>++.
//
reset // restart

```

メモリの状態を [c0, c1, c2, c3, c4] と表すと、以下の順序で最初の ASCII 文字 'H' = 0x48 = 72 が作られます。

まず、++++++ で c0 (カウンタ) に初期値 8 を代入します。

外側のループ [>++++ ... <-] を開始します。c0 の 8 を使い、隣の c1 に 4 を掛ける準備をします (この時点で $8 \times 4 = 32$ の計算構造)。

内側のループ [>++>++++>++++>+<<<<-] を開始します。c1 の 4 をカウンタにして、さらに右側のセルに値を分配します。

- c2 に 2 を加算 (計 $8 \times 4 \times 2 = 64$)
- c3 に 3 を加算 (計 $8 \times 4 \times 3 = 96$)
- c4 に 3 を加算 (計 $8 \times 4 \times 3 = 96$)
- c5 に 1 を加算 (計 $8 \times 4 \times 1 = 32$)

その後、>+>+->>+[<]<-]>>。で数値の微調整と出力をしています。ループの過程で c2(64 が入っているセル) にさらに外側ループの回数分などの調整が加わります。すべてのループが終わった時点で、c2 の値は正確に 72 になります。最後に >>。でポインタを c2 に合わせて出力すると、ASCII コード 0x48 = 72 の "H" が表示されます。

+ を 72 回並べても "H" は出せますが、コードが非常に長くなります。このプログラムは、“8 回繰り返すループの中で、さらに 4 回繰り返す” という入れ子構造 (ネスト) を利用することで、短いコードで大きな数値を効率的に作り出しています。

"H" 出力時 のデータ・メモリは以下になります。

- c1:0 (ループカウンターとして使い切った)
- c2:72('H') ここを出力
- c3:101('e')
- c4:108('l')

生成された各セルの値を微調整しながら、続く “e l l o” も出力していく仕組みです。

1.9.5 10進数表記の ASCII 文字列を UART で受信しバイナリ数値に変換するプログラム

10進数表記の ASCII 文字列を UART で受信しバイナリ数値に変換するプログラムをリスト 1.8 に示します。ファイルはリポジトリ bfCPU/bfTool/samples/inputdec.asm です。例えば UART が ASCII 文字列で “123\n” を受信すると、このプログラムの最後の時点で PTR が指すデータ・メモリにバイナリ数値 $123=0x7b$ が格納されます。受信時の最後の改行コードは $0x0A$ です。

リスト 1.8: ASCII 文字列からバイナリ数値に変換 inputdec.asm(1 文字命令版)

```
// https://esolangs.org/wiki/brainfuck_algorithms#Input_a_decimal_number
[-]>[-]+ // Clear sum
[[[-]] // Begin loop on first temp
>[-], // Clear the input buffer to detect leave on eof and input
[
  +[ // Check for minus one on eof
    -----[ // Check for newline
      >[-]+++++[<----->-] // Subtract 38 to get the char in zero to nine
      <-<<[->>++++++<<<] // Multiply the existing value by ten
      >>[-<<+>>] // and add in the new char
    <+>
  ]
]<]
< // Current cell has the input number
```

1.9.6 バイナリ数値を 10進数表記の ASCII 文字列に変換して UART から送信するプログラム

データ・メモリの 0 番地にバイト・サイズのバイナリ数値が格納され、かつ PTR=0 に設定されているとき、そのバイト・データを 10進数表記の ASCII 文字列に変換して UART から送信するプログラムをリスト 1.9 に示します。ファイルはリポジトリ bfCPU/bfTool/samples/printdec.asm です。例えば、データ・メモリの 0 番地にバイナリ数値 $123=0x7b$ が格納されている状態で、このプログラムを実行すると UART から ASCII 文字列 “123\n” を送信します。最後の改行コードは $0x0A$ です。

リスト 1.9: バイナリ数値を 10進数表記の ASCII 文字列に変換 printdec.asm(1 文字命令版)

```
// https://stackoverflow.com/questions/12569444/printing-a-number-in-brainfuck
>
>++++++<<[->+>-[>+>>]>[+[-<>]>+>>]<<<<<>>[-]>>>++++++<[->- [>+>>]>[+[-<>]>+>>]<<<<>>[-]>>[>+++++[<->+>+>[->+>>]]<.<<+>+>[-]><[<[->-<]>+++++[<->+>+>[->+>>]]<.-[-]<<[-<>]
+++++++.<
[+] // clear zero
```

1.9.7 TicTacToe ゲーム

bfCPU でコンピュータ対戦型 3 目並べのゲームも楽しめます。そのプログラムをリスト 1.10 に示します。ファイルはリポジトリ bfCPU/bfTool/samples/tictactoe.asm です。極めて精緻なプログラムで作成者の気合を感じます。

リスト 1.10: TicTacToe ゲーム tictactoe.asm(1 文字命令 + ニーモニック混在版)

```
//[tictactoe.b -- play tic-tac-toe
//(c) 2020 Daniel B. Cristofani
//http://brainfuck.org/
//This program is licensed under a Creative Commons Attribution-ShareAlike 4.0
```

```
//International License (http://creativecommons.org/licenses/by-sa/4.0/).]

-->-->>>->->>>- ->>>>>>>>>>>>>>>>>>>>>>+++++
[<<+[
 -<+<<+<<+>>>[
 >[<->>+>>[ - ]++<<<+[<+>>+<- - ]]>+++++[>>++++++<<- ]
 >>++++. [ - ]>>+[<<<<+>>+>> - ]<<<<[>+<- ]<<
 ]+++++++. [ - ]>++
]-->>[ - > [ - ]> ]<<[
 >>-- [
 - [
 - [
 ->[>+>++++++<<+]- ->>- . - - - >, [<->-]<[ [ < ]+[->]<-]<[ <<, [ - ]]>>>>
 ]>
 ]<[
 >- [ +<+++ ]+<+++ [ +[ - - - > ]+<<<<<[>>]<[ - ]]
 >[ <+[ - - - > ]++[ < ]<[ > ]>[ [ > ]+>++++++<<- [ < ]]>[ >>> ]
 ]<[
 - [ [ >+>+<<- ]>[ <+>- ]++>+>>]<[ <<+[ - - > [ - ]]>[ [ - ]>[ <<+>> - ]> ]
 ]<[
 [ [ << ]- [ >> ]<<+<- >[ - <+ ]<<[ << ]- <[ >[ +>> ]>[ > ]>[ - ]
 >[ [ +>> ]<<- >>[ > ]+>>>
 ]<[
 - [
 - [ +<<<<-[ +>[ - ]>[ <<+>>+<- - ]<<[ >>+<<- ]]+&gt;]
 <<[ >+>+<<-]>- - [ <+>- ]++>>
 ]<[ <<<[ - ]++>[ - ]>[ <>>>+<<- - ]+>>>
 ]<[
 +[ [ < ]<<[ << ]- <->>+>[ >> ]>[ > ]<- ]+[-<+ ]<++[ [ >+<- ]++<[ <<->>+ ]<++ ]<
 <<<<<< +> > >+ > >+ [
 <<<<<< ->+>+ > >->->+
 <<<<<< +>+> >+>+
 <<<< ->->+>-
 <<<<<<< +> > >->+
 <<<< -> >+> >->+
 <<<< +>->>+>+] ] ] ] ]
 +++[ [ >+<- ]<>>>[ - [ <->- ]<>>> ]++[ [ <->- ]>> ]>[ > ]
 ]<
 ]
 ]<
]
reset // restart

// [This program plays tic-tac-toe. I've given it the first move. It needs
// interactive i/o, e.g. a command-line brainfuck interpreter or a brainfuck
// compiler that produces command-line executables. At the '>' prompt, enter
// the number of an empty space, followed by a linefeed, to play a move there.]
```

このプログラムの動作例をリスト 1.11 に示します。プログラムを起動すると UART から 1 行目～3 行目の盤面が表示されます。bfCPU が先手で石を “X” で表示し、人間は後手で石を “O” で表示します。最初の盤面では、bfCPU が “1” の位置に石を置いています。4 行目でプロンプト “>” が表示されるので人間が石を打つ場所を指定します。ここでは “5” に打ちます。ASCII 文字で “5\n” と入力します。改行コードは 0x0A です。次に 5 行目～7 行目に盤面が表示され bfCPU が “9” に打ったことがわかります。これを繰り返していく、リスト 1.11 の例では bfCPU が勝っています。プログラムは最後に “reset” 命令を置いているので、勝敗が決したらまた最初に戻ります。

リスト 1.11: TicTacToe ゲームの動作

```

1: X23
2: 456
3: 789
4: >5
5: X23
6: 406
7: 78X
8: >4
9: X23
10: 00X
11: 78X
12: >2
13: X03
14: 00X
15: 7XX
16: >7
17: X0X
18: 00X
19: 0XX

```

1.9.8 Life ゲーム

bfCPU アーキテクチャに Life ゲームを実装した強者がいます。Life ゲームとは、将棋盤のような格子（セル=細胞）があり、セルの状態が時間と共に変化する様子を楽しむものです。セルの初期状態でその後の変化が決定されます。各セルの周囲には 8 つの近傍のセルがあります。各セルには“生”と“死”の 2 つの状態があり、あるセルの次のステップ（世代）の状態は周囲の 8 つのセルの状態により決定されます。セルの生死は次のルールに従います。

- **誕生**：死んでいるセルに隣接する生きたセルがちょうど 3 つあれば、次の世代が誕生する。
- **生存**：生きているセルに隣接する生きたセルが 2 つか 3 つならば、次の世代でも生存する。
- **過疎**：生きているセルに隣接する生きたセルが 1 つ以下ならば、過疎により死滅する。
- **過密**：生きているセルに隣接する生きたセルが 4 つ以上ならば、過密により死滅する。

図 1.8 に中央のセルの生死が次のステップでどうなるかの例を示します。生きているセルは 、死んでいるセルは で表しています。

誕生	生存（維持）	死（過疎）	死（過密）

図 1.8: Life ゲームの基本ルール (<https://ja.wikipedia.org/wiki/ライフゲーム>)

このプログラムをリスト 1.12 に示します。ファイルはリポジトリ bfCPU/bfTool/samples/life.asm です。もはや芸術品です。

リスト 1.12: Life ゲーム life.asm(1 文字命令 + ニーモニック混在版)

```

// Linus Akesson presents:
// The Game Of Life implemented in Brainfuck

+>>+++++[<++++>-]<[<++++++>-]+[<[>>>+<<<<-]>>>>[<<<<+>>>>>>+<<-]<+
+++[>+++++++-]>.[ -]<+++[>++<-]>+[>>.++<-]>>[ -]<<++[<+++++>-]<.<<[>>>+
<<<<-]>>>>[<<<<+>>>>>>+<<-]<<[>>>.+<<<+++++++-+<[>>+<<-]>>[<<+>>>>>+++++++

```



```
3: b-----
4: c-----
5: d-----
6: e-----
7: f-----
8: g-----
9: h-----
10: i-----
11: j-----
12: >ed\n
13: abcdefghij
14: a-----
15: b-----
16: c-----
17: d-----
18: e---*-----
19: f-----
20: g-----
21: h-----
22: i-----
23: j-----
24: >ee\n
25: ...
26: >ef\n
27: ...
28: >df\n
29: ...
30: >ce\n
31: abcdefghij
32: a-----
33: b-----
34: c----*-----
35: d----*-----
36: e----***-----
37: f-----
38: g-----
39: h-----
40: i-----
41: j-----
42: >\n
43: abcdefghij
44: a-----
45: b-----
46: c-----
47: d----*-----
48: e----*-----
49: f----*-----
50: g-----
51: h-----
52: i-----
53: j-----
54: >\n
55: abcdefghij
56: a-----
57: b-----
58: c-----
59: d----*-----
60: e----*-----
61: f----*-----
62: g-----
63: h-----
64: i-----
```

```

65: j-----
66: >\n
67: abcdefghij
68: a-----
69: b-----
70: c-----
71: d-----*
72: e-----**
73: f-----**
74: g-----
75: h-----
76: i-----
77: j-----
78: >\n
79: abcdefghij
80: a-----
81: b-----
82: c-----
83: d-----*
84: e-----*
85: f-----**
86: g-----
87: h-----
88: i-----
89: j-----
90: >

```

1.10 bfCPU はどんな問題も解けるのか？

bfCPU はどんな計算も可能なのでしょうか？以下に一例を示します。

1.10.1 多バイト長加算は可能か？

bfCPU は 8 ビットの加算はできますが、キャリー伝搬の機能がないので、多バイト長の加算はどのようにすれば実現できるでしょうか？例えば 4 バイト（32 ビット）長の加算を行うことを考えてみましょう。

各バイトを個別のセルに格納し、下位バイトから順に加算してキャリー（繰り上がり）を処理するアルゴリズムを用意する必要があります。

メモリレイアウトの設計

このプログラムでは、以下の形式でデータを配置します。

- セル 0~3: 数値 A (A3, A2, A1, A0) A0 が最下位
- セル 4~7: 数値 B (B3, B2, B1, B0) ここに加算結果を格納
- セル 8: キャリー（繰り上がり）用の一時フラグ

基本プログラム

以下のコードは、数値 B の最下位に数値 A の最下位を加算します。

```

// 最下位バイト(A0からB0)の加算
<<<          ポインタをA0へ( 初期位置をA0と仮定 )
[
    -          A0が0になるまでループ
    A0を減らす
    >>> +
    [          B0を増やす
       もしB0がオーバーフローして0になったらキャリーを立てる(詳細略)

```

```

    - (このブロックはB0が非0の時のみ実行されるため工夫が必要)
  ]
  <<<
      A0に戻る
]

```

実用アルゴリズムへ

bfCPU にはキャリーフラグが直接存在しないため、厳密な 32 ビット加算 (255 を超えたら隣のセルに +1 する) を行うには、各ステップで以下の処理を実装する必要があります:

- 下位バイトの移動: [- > + <] を使って A の値を B に足す。
- オーバーフロー判定: 加算後の B の値をチェックし、0 を跨いだ場合に次のバイトを + する処理をループや一時セルを用いて記述する。
- 上位バイトへの伝搬: これを最上位 (4 バイト目) まで繰り返す。

必要に応じて“2 の補数”形式で多バイト表現を管理することで、多バイト長の減算や負数の加算も可能です。

1.10.2 AND 演算は可能か？

[] のループを“値が 0 でない間”という条件分岐として利用し、下記のように入れ子を作ります。

```
if (x) { if (y) { result = 1 } }
```

1.10.3 OR 演算は可能か？

以下のように順次処理します。

```
result = 0; if (x) { result = 1 }; if (y) { result = 1 }
```

1.10.4 XOR 演算は可能か？

XOR は下記の論理構造で実現できます。

```

if (x) {
  if (y) { res = 0 } else { res = 1 }
} else {
  if (y) { res = 1 } else { res = 0 }
}

```

1.10.5 bfCPU はコンピュータとしてのあらゆる処理が可能

bfCPU は完全チューリング・マシンとして認められており、コンピュータとしてのあらゆる処理が可能です。ただし、プログラムは難解になり、実行時間は長くなります。それでも、このパズルを解く工程は深淵かつ最高の娯楽になるでしょう。

1.11 bfCPU システムの論理設計

以上述べた bfCPU アーキテクチャのコンピュータ・システムを実際に設計していきましょう。

1.11.1 bfCPU システムの全体構成

今回設計する bfCPU システムの全体構成を図 1.9 に示します。チップまたは FPGA(bfCPU チップ) の内部

に、bfCPU 本体、キャッシュ・メモリ、QSPI SRAM インタフェースおよび UART を内蔵します。外部メモリとしては 512Kbits(64Kbytes) の QSPI SRAM 23LC512(Microchip) を 1 個接続します。

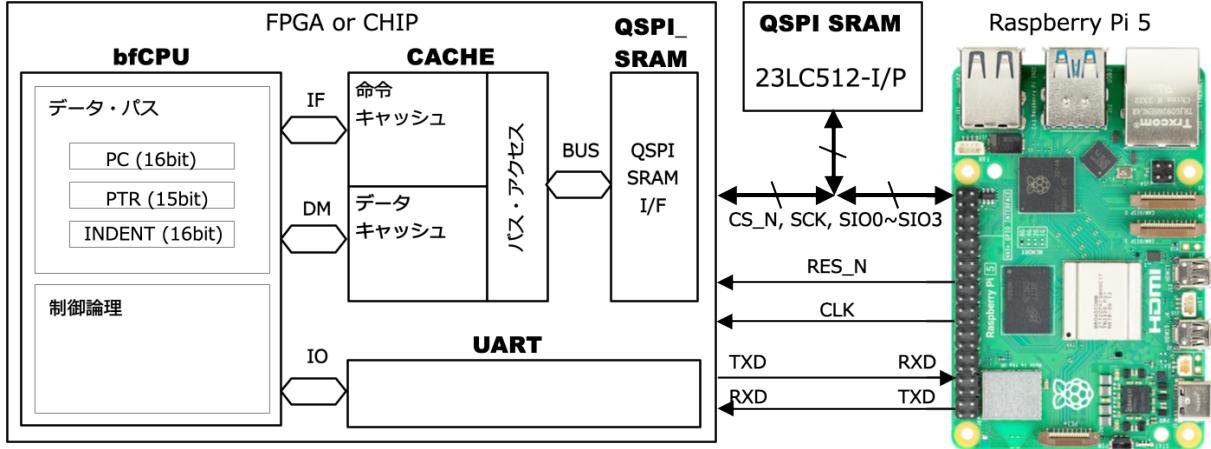


図 1.9: bfCPU システム全体のブロック図

1.11.2 bfCPU のメモリ・マップとメモリ構成

bfCPU チップが外部メモリ QSPI SRAM をどのように割り当てるか、そのメモリ・マップを図 1.10 に示します。bfCPU には 2 つのメモリ空間があります。一つは命令空間で 4 ビット幅 × 65536 ワードのサイズを持ちます。命令空間は bfCPU のプログラム・カウンタ PC がアドレス (0x0000~0xFFFF) を指定してアクセスします。もう一つはデータ空間で 8 ビット幅 × 32768 ワードのサイズを持ちます。データ空間は bfCPU のデータ・ポインタ PTR がアドレス (0x0000~0x7FFF) を指定してアクセスします。

bfCPU チップ内では、bfCPU が命令をアクセスするバスが IF バスで、これが命令キャッシュをアクセスします。また bfCPU がデータをアクセスするバスが DM バスで、これがデータ・キャッシュをアクセスします。命令キャッシュまたはデータ・キャッシュがミスすれば外部メモリ QSPI SRAM をアクセスします。命令キャッシュとデータ・キャッシュが同時に QSPI SRAM をアクセスしようとしたらデータ・キャッシュからのアクセスを優先し、命令・キャッシュによるアクセスは待たれます。もし命令キャッシュからのアクセスを優先したら bfCPU 内に実行完了できない命令が溜まってしまってハングしてしまいます。

命令キャッシュまたはデータ・キャッシュがミスしたときに BUS バスを通して QSPI SRAM をアクセスします。QSPI SRAM チップの内部は、合計 64Kbytes のうち、前半の 32Kbytes(QSPI SRAM のアドレス 0x0000~0x7FFF) を命令空間に、後半の 32Kbytes(QSPI SRAM のアドレス 0x8000~0xFFFF) をデータ空間にアサインしています。

1.11.3 QSPI SRAM の書き込み方法

QSPI SRAM の命令メモリにプログラムを書き込むために、Raspberry Pi 5 ボード (ラズパイ) を使います。ラズパイが、bfCPU チップにクロックとリセット信号を供給します。ラズパイが bfCPU チップをリセットしている間は、bfCPU チップは QSPI RAM インタフェース信号を Hi-Z 状態にし、その間、ラズパイの GPIO 操作により QSPI SRAM にプログラムを書き込みます。その後、ラズパイの QSPI SRAM インタフェース信号を Hi-Z にしてリセットを解除すると bfCPU チップが QSPI SRAM 上のプログラムの実行を開始します。

1.11.4 UART による入出力

bfCPU チップには入出力用に UART を搭載しています。UART の送受信信号は直接ラズパイに接続し、ラ

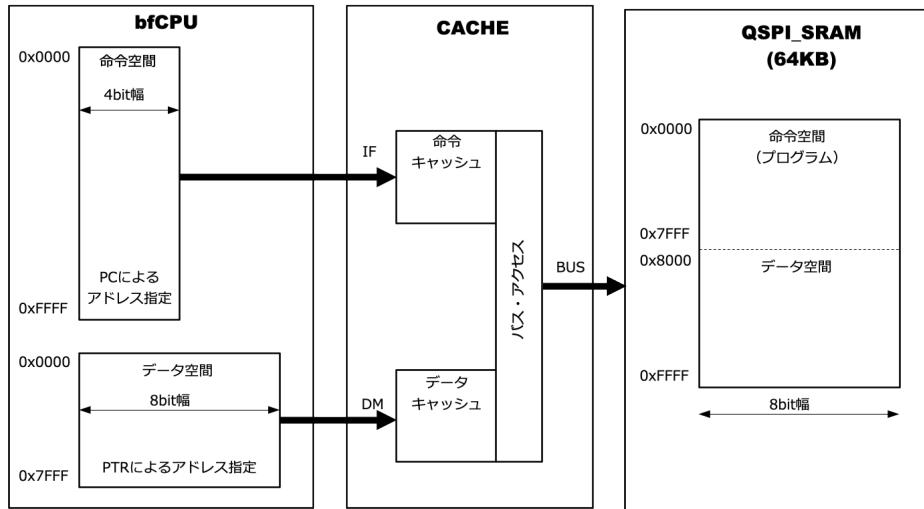


図 1.10: bfCPU のメモリ・マップとメモリ構成

ズパイ上のターミナル・ソフトで文字列やバイナリ・データを入出力します。

UART のボーレートは、bfCPU の動作周波数に依存するので、固定化はしませんでした。ラズパイが bfCPU チップをリセット中に、QSPI SRAM の最終番地に UART のボーレート設定値を書いて、bfCPU チップがリセット解除直後にその番地をリードして UART のボーレートを設定するようにします。基本的にボーレートは 115200bps になるように設定し、UART のデータ・フォーマットは 8 ビット・ノンパリティ、STOP ビット 1(8N1) で通信します。

1.12 bfCPUチップの内部バス・アクセス・タイミング

bfCPU の命令アクセス用の IF バスとデータ・アクセス用 DM バスのタイミングを図 1.11 に示します。また、bfCPU の UART アクセス用の IO バスと、キャッシングが QSPI SRAM をアクセスする BUS バスのタイミングを図 1.12 に示します。それぞれのバスの信号の意味は共通で以下の通りです。

- **xx_REQ :** バス・トランザクション要求信号
- **xx_WRITE :** ライト信号 (0 でリード、1 でライト、IF_BUS についてはリード方向のみなのでこの信号はない)
- **xx_ADDR :** アドレス
- **xx_WDATA :** ライト・データ
- **xx_RDATA :** リード・データ (スレーブ側が出力)
- **xx_RDY :** レディ信号 (スレーブ側が出力)

xxREQ、xxWRITE、xx_ADDR、xx_WDATA はマスタ側 (CPU のようにバス・アクセスを行う側) が output し、xx_RDATA、xx_RDY はスレーブ側 (メモリのようにバス・アクセスを受ける側) が output します。xx_RDY は、スレーブ側がリード・アクセスやライト・アクセスをすぐに完了できないときにネゲートしてマスタ側の動作にウェイトをかけるために使います。

バス・タイミングはいずれのバスも共通なので図 1.11 の DM バスを例に説明します。時刻 T1、T2、... はシステム・クロックの立ち上がりエッジの時刻を示しています。

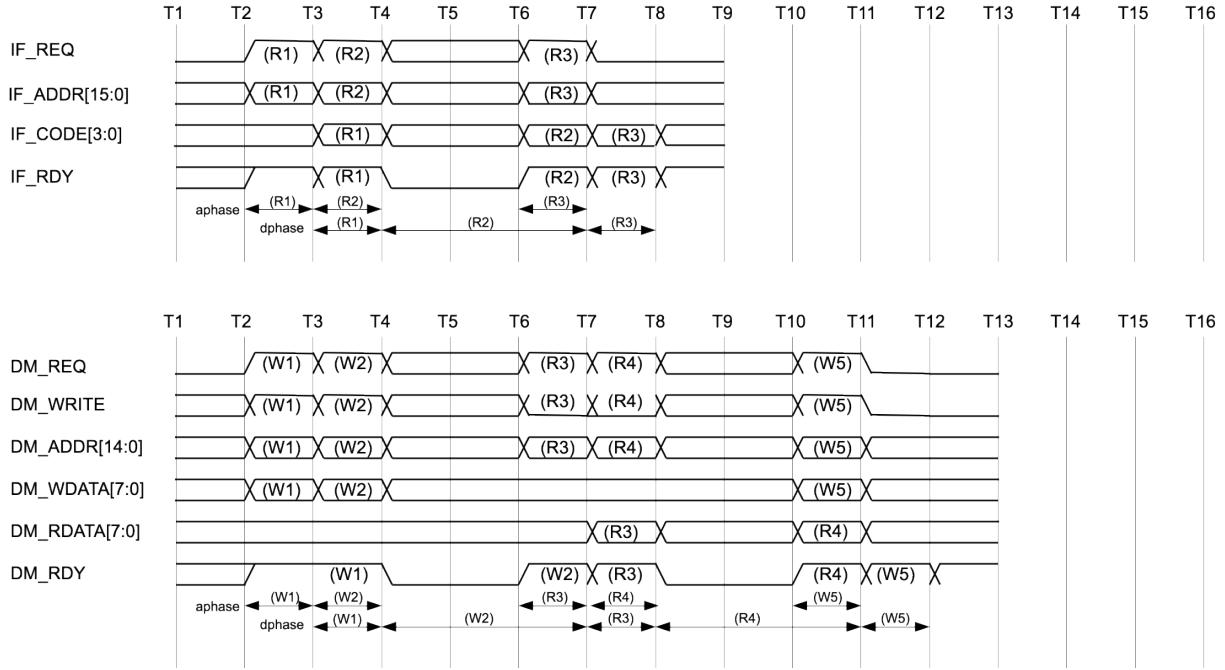


図 1.11: bfCPU チップ内部バス・タイミング (IF バス、DM バス)

1.12.1 ライト・アクセス

時刻 T2 で DM_REQ をアサートし DM_WRITE、DM_ADDR、DM_WDATA を確定させライト・アクセスを要求します。ここからアドレス・フェーズ (aphase) を開始します。時刻 T3 で DM_RDY がアサートされていればアクセス要求を受付けてアドレス・フェーズ (aphase) が終わり、データ・フェーズ (dphase) を開始します。スレーブ側はすぐにこのライト動作を完了させ、次の時刻 T3 で DM_RDY を継続アサートし、時刻 T4 でデータ・フェーズ (dphase) が終わってこのライト・アクセスが終了します。

時刻 T3 で前のアクセスのアドレス・フェーズ (aphase) が終わったので、ここから DM_REQ をアサートし DM_WRITE、DM_ADDR、DM_WDATA を確定させ次のライト・アクセスを要求できます。ここからアドレス・フェーズ (aphase) を開始します。時刻 T4 で DM_RDY がアサートされていればアクセス要求を受付けてアドレス・フェーズ (aphase) が終わり、データ・フェーズ (dphase) を開始します。スレーブ側がすぐにライト動作を完了できない場合はデータ・フェーズ (dphase) を延長するため、時刻 T4 で DM_RDY をネガートします。スレーブ側のライト動作が終われば時刻 T6 で DM_RDY をアサートし、時刻 T7 でデータ・フェーズ (dphase) が終わってこのライト・アクセスが終了します。

1.12.2 リード・アクセス

時刻 T6 で DM_REQ をアサートし DM_WRITE、DM_ADDR を確定させリード・アクセスを要求します。ここからアドレス・フェーズ (aphase) を開始します。時刻 T7 で DM_RDY がアサートされていればアクセス要求を受付けてアドレス・フェーズ (aphase) が終わり、データ・フェーズ (dphase) を開始します。スレーブ側はすぐにこのリード動作を完了させ、次の時刻 T7 で DM_RDATA を確定させ、かつ DM_RDY を継続アサートし、時刻 T8 でデータ・フェーズ (dphase) が終わってこのリード・アクセスが終了します。リード・アクセスを要求したマスタ側はこの時刻 T8 でリード・データ DM_RDATA をキャプチャします。

時刻 T7 で前のアクセスのアドレス・フェーズ (aphase) が終わったので、ここから DM_REQ をアサートし DM_WRITE、DM_ADDR を確定させ次のリード・アクセスを要求できます。ここからアドレス・フェーズ (aphase) を開始します。時刻 T8 で DM_RDY がアサートされていればアクセス要求を受付けてアドレス・フェーズ (aphase) が終わり、データ・フェーズ (dphase) を開始します。スレーブ側がすぐにリード動作を完了

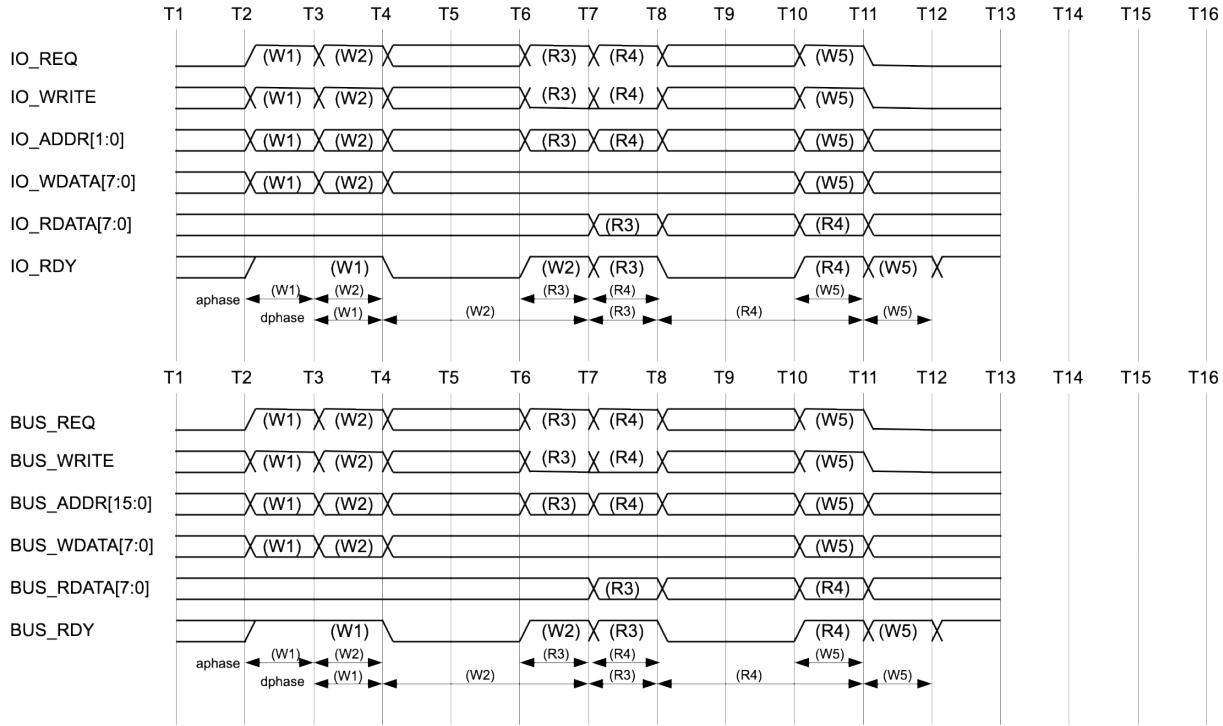


図 1.12: bfCPU チップ内部バス・タイミング (IO バス、BUS バス)

できない場合はデータ・フェース (dphase) を延長するため、時刻 T8 で DM_RDY をネゲートします。スレーブ側のリード動作が終われば時刻 T10 で DM_RDATA を確定させ、かつ DM_RDY をアサートし、時刻 T11 でデータ・フェーズ (dphase) が終わってこのリード・アクセスが終了します。リード・アクセスを要求したマスター側はこの時刻 T11 でリード・データ DM_RDATA をキャプチャします。

1.13 bfCPU のパイプライン構成

bfCPU の内部は図 1.13~図 1.18 に示すパイプライン方式で制御します。

1.13.1 パイプライン・ステージ

bfCPU のパイプライン・ステージを図 1.13 に示します。命令フェッチは F ステージ、命令デコードと実行を D ステージ、データ・メモリのライトを MW ステージ、データ・メモリのリードを MR ステージ、IO ライトを IW ステージ、IO リードを IR ステージでそれぞれ制御します。命令全体のシーケンス動作は D ステージが中心になって制御します。命令によってトータルのステージ数は異なりますが、最小で 2 ステージのパイプラインで動作します。

図 1.13(a) に命令フェッチ制御を示します。D ステージが基本的に次の命令の F ステージとして IF バスのアドレス・フェーズ (aphase) を起動します。その IF バスのデータ・フェーズ (dphase) がフェッチした命令を IF_RDATA に載せるのでそこで次の命令の D ステージを処理します。一部の命令では D ステージが次命令の F ステージを起動しない場合もあります。

図 1.13(b) にデータ・メモリのライト制御を示します。D ステージがライト・アドレスとライト・データを用意したら同時に DM バスのアドレス・フェーズ (aphase) を起動します。その DM バスのデータ・フェーズ (dphase) が DW ステージとなりライトを実行します。

図 1.13(c) にデータ・メモリのリード制御を示します。D ステージがリード・アドレスを用意したら同時に

DM バスのアドレス・フェース (aphase) を起動します。その DM バスのデータ・フェーズ (dphase) が DR ステージとなりリード・データが DM_RDATA に載ります。そのデータをそのまま次命令の D ステージが使うことができます。

図 1.13(d) に IO ライト制御、図 1.13(e) に IO リード制御を示します。これらはデータ・ライトとデータ・リードと同様な制御を行います。bfCPU システムにおいては、IO バスは UART のレジスタ・アクセスを行います。UART のレジスタは、送受信データ用レジスタが 1 つ、およびボーレート設定用のレジスタが 2 つあります。送受信データ用レジスタにライトすると送信開始しますが、送信バッファがフルの場合はそのライト・アクセスにウェイトがかかります。送受信データ用レジスタをリードすると受信データを取り込めますが、受信バッファに受信データが入るまでそのリード・アクセスにウェイトがかかります。ボーレート設定用レジスタのアクセスにはウェイトは入りません。

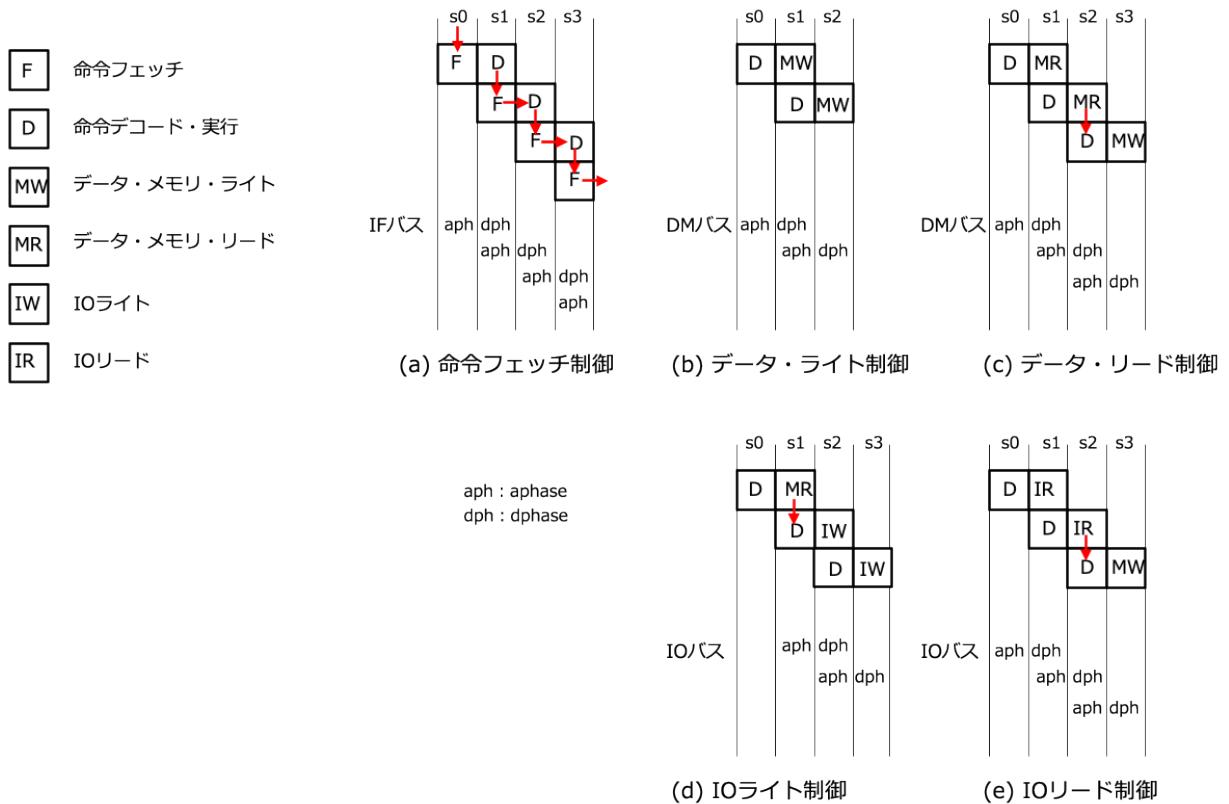


図 1.13: bfCPU パイプラインのステージ

1.13.2 パイプラインのウェイト制御

命令フェッチやデータ・アクセス (IO アクセス含む) の際、xx_RDY がネゲートされてウェイトが入ると、パイプライン制御にもウェイトを入れます。その制御方法を図 1.14 に示します。

(a)(b) に示すように、命令フェッチやデータ・アクセスの際、そのデータ・フェーズ (dphase) が延ばされると、その位置に対応するステージも延ばされます。その前後命令の同じタイミングにあるステージも一緒に延ばされます。CPU 全体のパイプライン制御のステートは、信号 slot がアサートされたタイミングで進みます。よって信号 slot は各バスの xx_RDY 信号をそのまま使えます。

(c)(d) に示すように、命令フェッチとデータ・アクセスに同時にウェイトが入る場合は、ウェイトが長い方にしたがってステージが延ばされます。リード時は、ウェイトが短い方の xx_RDATA が先に切り替わってしまう可能性があるので、それを保持する機構を bfCPU 内に用意しておく必要があります。CPU 全体のパイプラ

イン制御のステートを進める信号 slot は3つある IF バス・DM バス・IO バスの各 xx_RDY 信号を AND して生成できます。

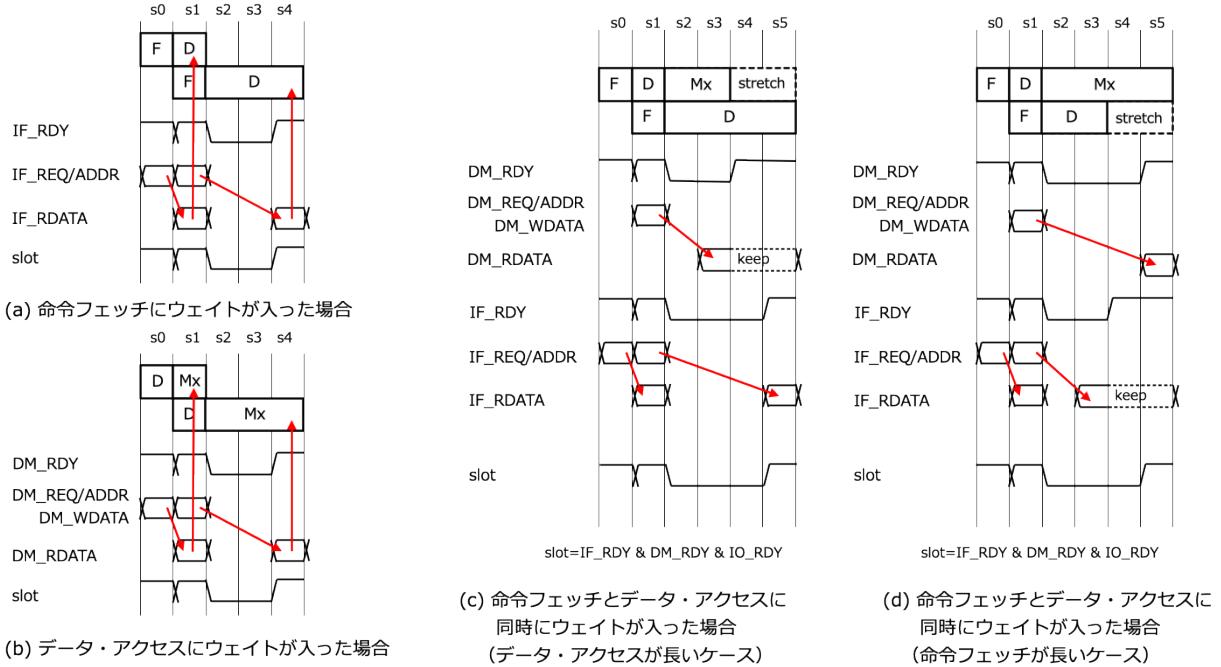


図 1.14: bfCPU パイプラインのウェイト制御

1.13.3 bfCPU のパイプライン制御

bfCPU の各命令のパイプライン動作を図 1.15~図 1.18 に示します。bfCPU の命令制御は D ステージが担い、D ステージが行う指示をステート (state) とシーケンス (seq) の2つの信号で制御します。state は処理の大きな区分を示し、seq はその state の中のシーケンス処理を示します。

(a) リセット時の動作

bfCPU がリセットされると、state が STATE_INIT になり、この中で seq を進めながら PC=16'h0000 に初期化し、データ・メモリのアドレス 0x0000~0x7FFD をゼロ・ライトします。データ・メモリのアドレス 0x7FFE と 0x7FFF に、UART のボーレート設定値が書かれているので、それらの値を IO バスにライトします。最後の D ステージが最初の命令の F ステージを PC=0 で起動し直後に PC をインクリメントし、state を STATE_DECODE に、seq を 0 に設定します。

(b) 命令 pinc/p++(<)、pdec/p (<) の動作

F ステージが起動した命令フェッチのデータ・フェーズがその命令の D ステージになります。state が STATE_DECODE の時の D ステージでは、IF_RDATA に載っている命令コードを取り込み、命令種類をデコードし最初の実行処理を行います。デコード結果が命令 pinc/p++(<) または pdec/p (<) の場合は、この D ステージ内でデータ・ポインタ PTR をインクリメントするかデクリメントします。同時のこの D ステージで次命令の F ステージを起動し直後に PC をインクリメントします。state は STATE_DECODE のまま、seq も 0 のままでです。

(c) 命令 inc(+)、dec(-) の動作

state が STATE_DECODE の時の D ステージでデコード結果が命令 inc(+)、dec(-) の場合は、この D ステージでデータ・メモリのリードのアドレス・フェーズ (aphase) をアドレス PTR で起動し、state を STATE_INC

または STATE_DEC に、seq を 1 に移行し、その D ステージがデータ・メモリのリードのデータ・フェーズ (dphase) なのでリード・データ DM_RDATA をインクリメントまたはデクリメントして、その値をライト・データにしたデータ・メモリのライトのアドレス・フェーズ (aphase) をアドレス PTR で起動します。同時にこの D ステージで次命令の F ステージを起動し直後に PC をインクリメントします。その後、state を STATE_DECODE に、seq を 0 に設定します。

1.13.4 (d) 命令 `out(.)` の動作

state が STATE_DECODE の時の D ステージでデコード結果が命令 `out(.)` の場合は、この D ステージでデータ・メモリのリードのアドレス・フェーズ (aphase) をアドレス PTR で起動し、state を STATE_OUT に、seq を 1 に移行し、その D ステージがデータ・メモリのリードのデータ・フェーズ (dphase) なのでリード・データ DM_RDATA をそのまま使って IO バスにライト・アクセスのアドレス・フェーズ (aphase) を UART の送受信レジスタに対して起動します。同時にこの D ステージで次命令の F ステージを起動し直後に PC をインクリメントします。その後、state を STATE_DECODE に、seq を 0 に設定します。

1.13.5 (e) 命令 `in(,)` の動作

state が STATE_DECODE の時の D ステージでデコード結果が命令 `in(,)` の場合は、この D ステージで IO バス・リードのアドレス・フェーズ (aphase) を UART の送受信レジスタに対して起動し、state を STATE_IN に、seq を 1 に移行し、その D ステージが IO バス・リードのデータ・フェーズ (dphase) なのでリード・データ IO_RDATA をそのまま使ってデータ・メモリのライトのアドレス・フェーズ (aphase) をアドレス PTR で起動します。同時にこの D ステージで次命令の F ステージを起動し直後に PC をインクリメントします。その後、state を STATE_DECODE に、seq を 0 に設定します。

1.13.6 (f) 命令 `begin()` の動作

state が STATE_DECODE の時の D ステージでデコード結果が命令 `begin()` の場合は、この D ステージでデータ・メモリのリードのアドレス・フェーズ (aphase) をアドレス PTR で起動し、state を STATE_BEGIN に、seq を 1 に移行し、その D ステージがデータ・メモリのリードのデータ・フェーズ (dphase) なのでリード・データ DM_RDATA をチェックします。

DM_RDATA が非ゼロならば、この D ステージで次命令の F ステージを起動し直後に PC をインクリメントします。その後、state を STATE_DECODE に、seq を 0 に設定して、次命令に進みます。

DM_RDATA がゼロならば、対応する命令 `end()` の次命令にスキップしますが、命令コードに何番地に入スキップせよという情報がないので、命令メモリを前方向に向けてフェッチし続けて、対応する `end()` 命令をサーチします。まず内部レジスタのインデント数 INDENT を 0 にして、このサーチ中に、命令 `begin()` があればインデント数 INDENT をインクリメントしてサーチを続けます。また $INDENT > 0$ の時に命令 `end()` を見つけたらこれはまだ実行中の命令 `begin()` に対応する命令 `end()` ではないので INDENT をデクリメントしてサーチを続けます。 $INDENT == 0$ の時に命令 `end()` を見つけたらこれが実行中の `begin()` に対応したものなので、サーチを終了します。state が STATE_BEGIN の間この処理を続け、state を STATE_DECODE に、seq を 0 に設定して、命令 `end()` の次命令からの処理に進みます。

1.13.7 (g) 命令 `end()` の動作

state が STATE_DECODE の時の D ステージでデコード結果が命令 `end()` の場合は、この D ステージでデータ・メモリのリードのアドレス・フェーズ (aphase) をアドレス PTR で起動し、state を STATE_END に、seq を 1 に移行し、その D ステージがデータ・メモリのリードのデータ・フェーズ (dphase) なのでリード・データ DM_RDATA をチェックします。

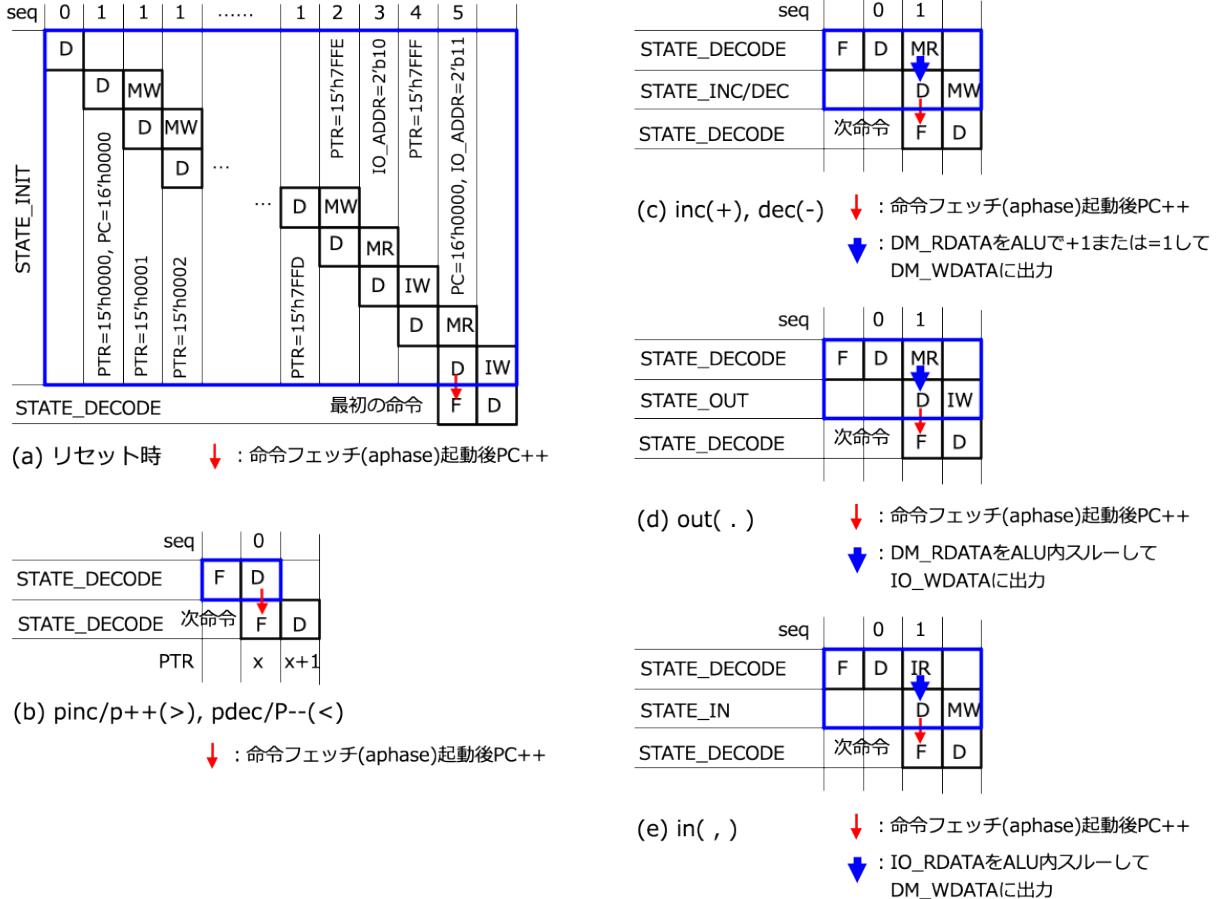


図 1.15: bfCPU 各命令のパイプライン動作 (1)

DM_RDATA がゼロならば、この D ステージで次命令の F ステージを起動し直後に PC をインクリメントします。その後、state を STATE_DECODE に、seq を 0 に設定して、次命令に進みます。

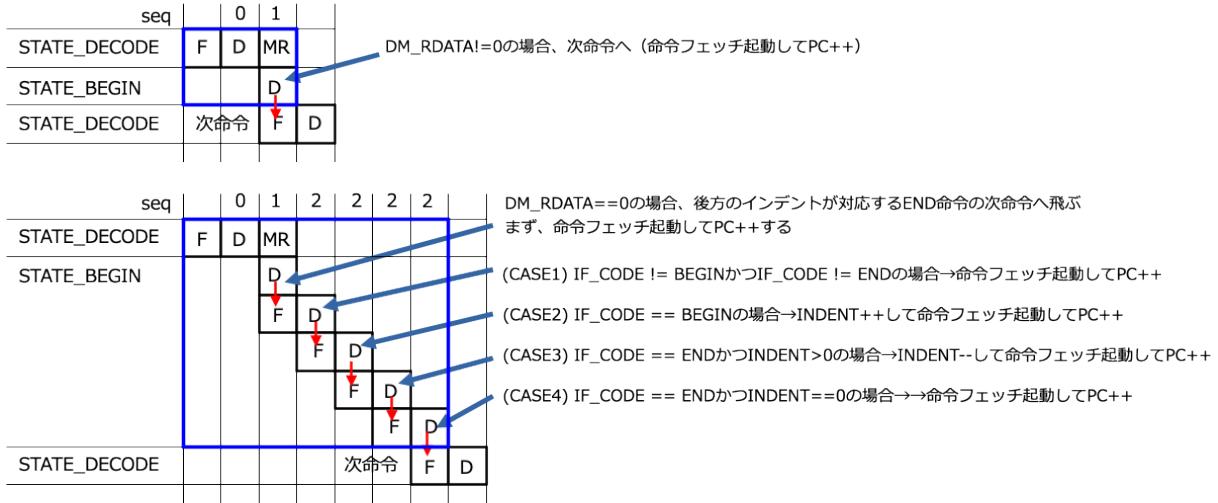
DM_RDATA が非ゼロならば、対応する命令 begin() の次命令に戻りますが、命令コードに何番地に戻れという情報がないので、命令メモリを後方向に向けてフェッチし続けて、対応する begin() 命令をサーチします。まず内部レジスタのインデント数 INDENT を 0 にして、このサーチ中に、命令 end() があればインデント数 INDENT をインクリメントしてサーチを続けます。また INDENT>0 の時に命令 begin()を見つけたらこれはまだ実行中の命令 end() に対応する命令 begin() ではないので INDENT をデクリメントしてサーチを続けます。INDENT==0 の時に命令 begin()を見つけたらこれが実行中の end() に対応したものなので、サーチを終了します。state が STATE_END の間この処理を続け、state を STATE_DECODE に、seq を 0 に設定して、命令 begin() の次命令からの処理に進みます。このサーチの間の命令アドレスと命令 begin() の次命令アドレスを正しく生成するため、PC の値に必要な調整を入れながら処理します。

1.13.8 (h) 命令 reset の動作

state が STATE_DECODE の時の D ステージでデコード結果が命令 reset の場合は、state を STATE_INIT に設定して、(a) リセット動作に移行します。

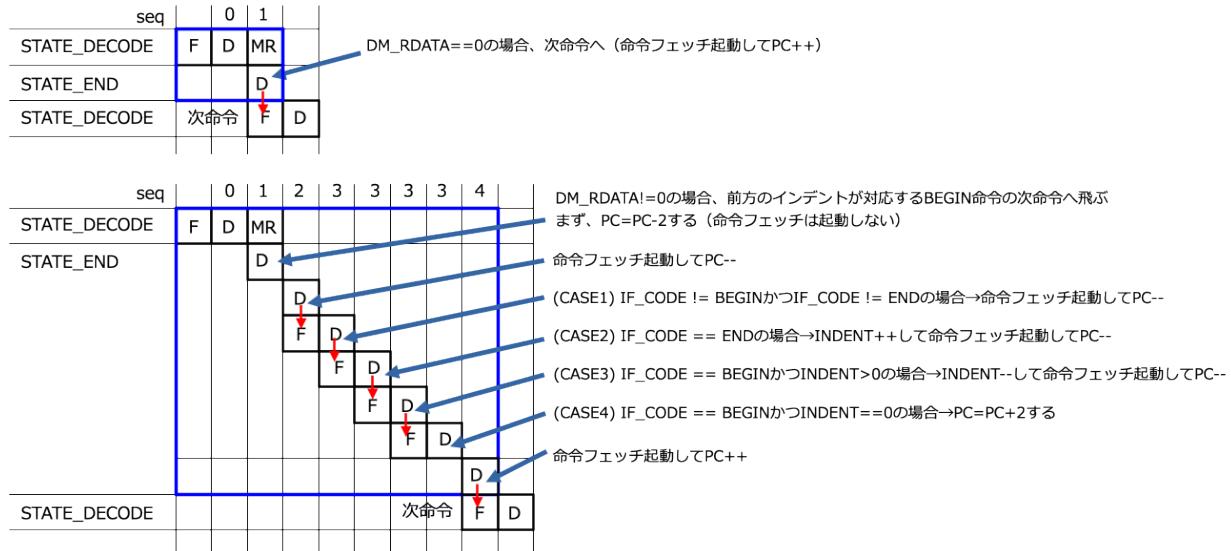
1.13.9 (i) 命令 nop の動作

state が STATE_DECODE の時の D ステージでデコード結果が命令 nop の場合は、その D ステージで次命令の F ステージを起動し直後に PC をインクリメントします。何も実行せず、次命令に処理を進めます。



(f) begin ([)

図 1.16: bfCPU 各命令のパイプライン動作 (2)



(g) end (])

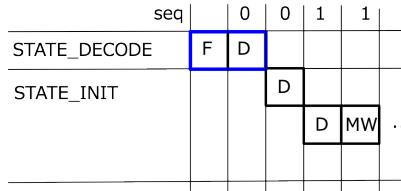
図 1.17: bfCPU 各命令のパイプライン動作 (3)

1.14 bfCPU のデータ・バス論理

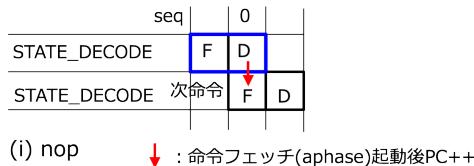
一般に CPU などのある程度規模が大きい論理回路では、データを加工しながら流していくデータ・バス論理と、そのデータ・バスを制御する信号を生成する制御論理とに分けて設計する方が見通しがよくなります。

データ・バスには、複雑なタイミングに絡む制御はいれず、できるだけデータを動かすだけの動作に徹します。今回の bfCPU もこのような構成を取りました。図 1.19 に bfCPU のデータ・バスの全体構成を示します。

パイプライン制御のスタートを進める信号 slot の生成部、IF バスのインタフェース部、DM バスのインタフェース部、IO バスのインタフェース部、プログラム・カウンタ PC、データ・ポインタ PTR、インデント・カウンタ INDENT、演算操作用の ALU(Arithmetic Logical Unit) があります。それぞれ、データを流す経路だけを持っていて、その制御信号は、次に説明する制御論理が生成します。



(h) reset



(i) nop

↓ : 命令フェッチ(apphase)起動後PC++

図1.18: bfCPU各命令のパイプライン動作(4)

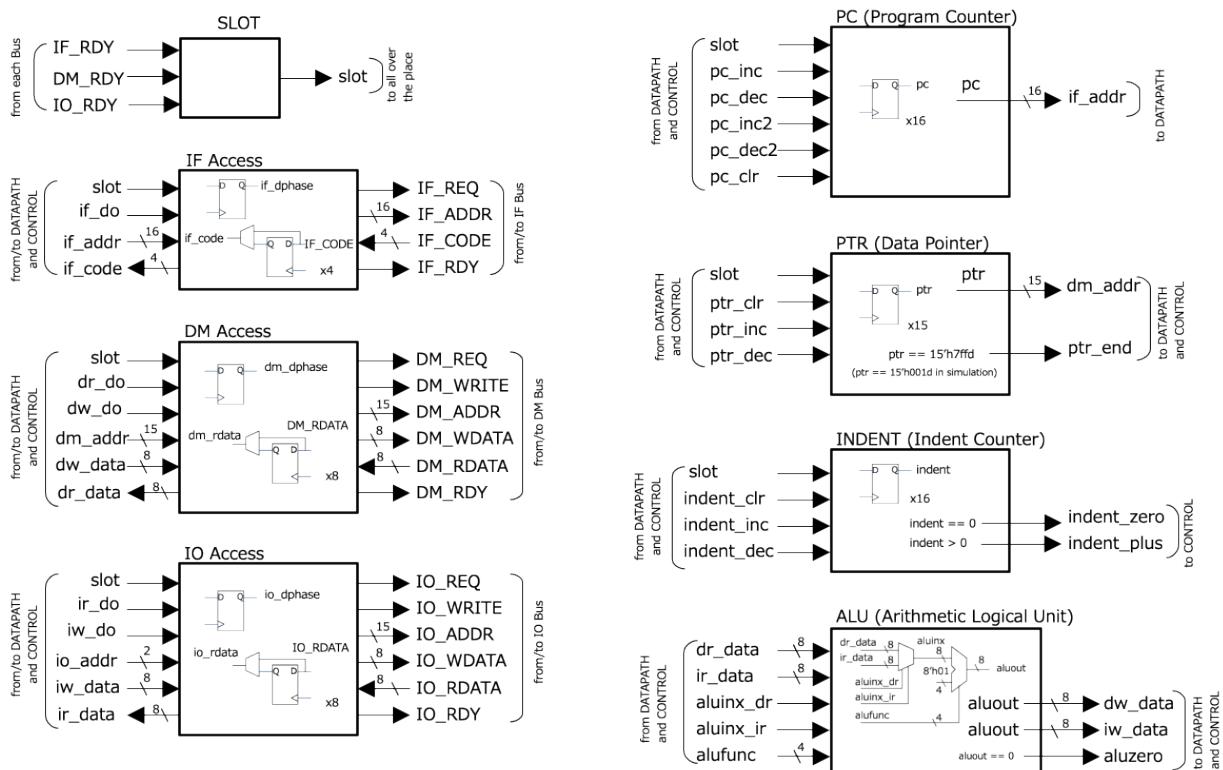


図1.19: bfCPUのデータ・パス論理

1.15 bfCPUの制御論理

bfCPUの制御論理の全体構成を図1.20に示します。パイプラインのDステージの動きをステート・マシンで制御します。信号stateとseq、命令コードid_code、データ・バスからの状態信号から大きな真理値表でデータ・バスの制御信号と次のstateとseqを作って、制御ステートを進めていきます。

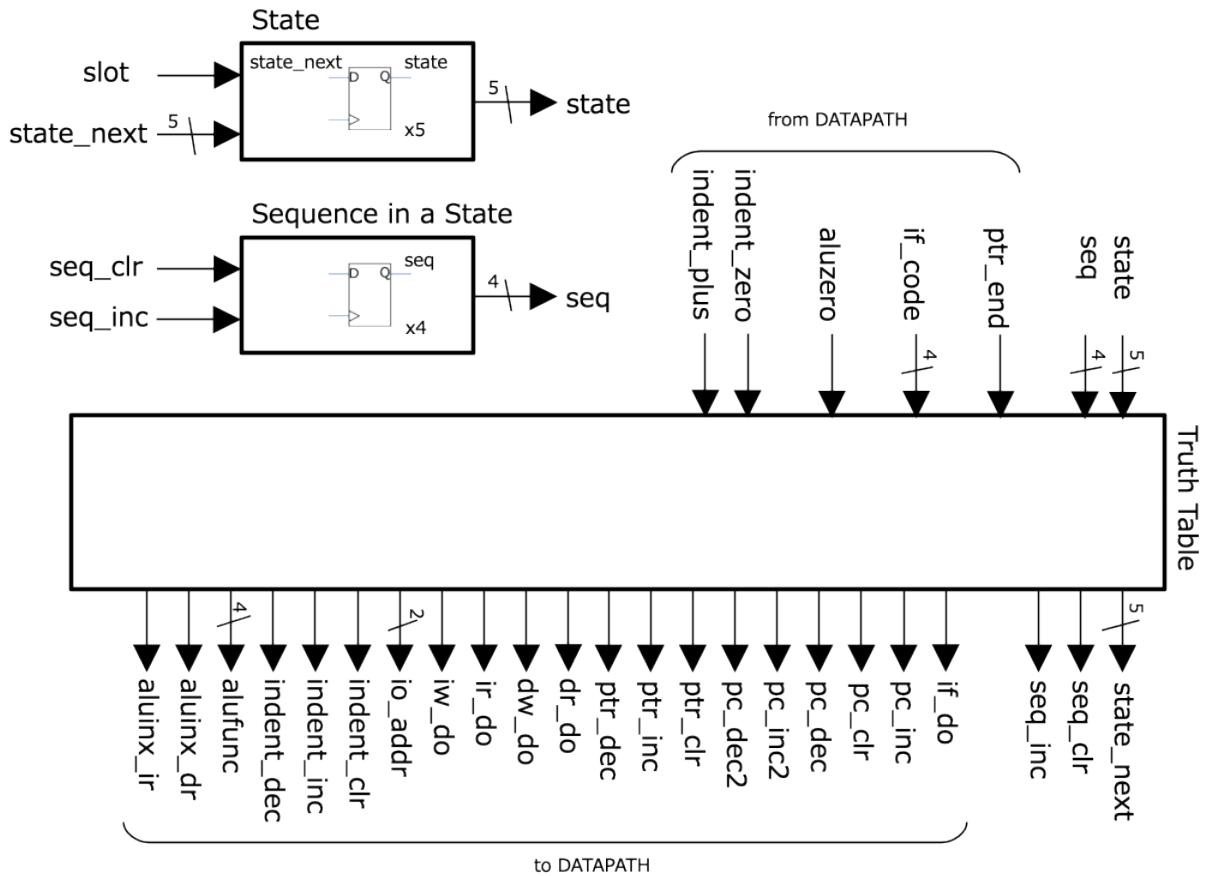


図 1.20: bfCPU の制御論理

1.16 bfCPU の RTL 記述

bfCPU の RTL 記述は、リポジトリ内の bfCPU/RTL/CPU/cpu.sv です。モジュール宣言部（入出力信号）をリスト 1.14 に示します。入出力信号は、クロック、リセット、IF パス、DM パス、IO パスです。RTL 内部は、上記で説明したデータ・パス論理と制御論理がそのまま記述されています。

リスト 1.14: bfCPU の RTL 記述 cpu.sv の モジュール宣言部

```
module CPU
(
    input  logic CLK,
    input  logic RES,
    //
    // CPU Instruction Fetch
    // PC Address (64kwords x 4bits)
    output logic      IF_REQ,
    output logic [15:0] IF_ADDR,
    input  logic [ 3:0] IF_CODE,
    input  logic      IF_RDY,
    //
    // CPU Data Memory Access
    // PTR Address (32kbytes x 8bits)
    output logic      DM_REQ,
    output logic      DM_WRITE,
    output logic [14:0] DM_ADDR,
    output logic [ 7:0] DM_WDATA,
```

```

input logic [ 7:0] DM_RDATA,
input logic      DM_RDY,
//
// I/O Access
output logic     IO_REQ,
output logic     IO_WRITE,
output logic [ 1:0] IO_ADDR,
output logic [ 7:0] IO_WDATA,
input logic [ 7:0] IO_RDATA,
input logic      IO_RDY
);

```

1.17 キャッシュ・メモリ

一般に、CPUはロジックなので動作周波数を上げやすいですが、CPUがアクセスする外部メモリは高速化しにくいデバイスです。CPUと外部メモリを直結してあるとこの速度差がCPUの性能を低下させてしまいます。これを防ぐ機構の一つがキャッシュ・メモリです。キャッシュ・メモリは、外部メモリよりも小さいサイズのSRAMで構成し、容量が小さいため、そのアクセス速度はCPUの速度と同じにできます。このキャッシュ・メモリは以下の2つの理由で性能向上を図ることができます。

- **プログラムとデータが局所的:** プログラム(命令列)にループがあればCPUはそのループ内の命令をフェッチし続けます。初回は低速な外部メモリから命令フェッチしますが、フェッチした命令をキャッシュ・メモリに入れておけば、次回はキャッシュ・メモリをアクセスすれば高速に取り込めます。データも近傍アドレスに固まった構造体を繰り返しアクセスするケースが多ければ、初回のアクセスは外部メモリから行って、それをキャッシュ・メモリに入れておけば、次回はキャッシュ・メモリをアクセスすれば高速に取り込めます。リードしかしない命令キャッシュと、リードとライトがあるデータ・キャッシュとはそれぞれ制御方法に微妙な差はありますが、一般にプログラムとデータが局所的である特性を使うことで、性能向上が可能です。
- **外部メモリは連続アクセスが高速:** キャッシュ・メモリの中にCPUがアクセスした命令やデータがあれば(ヒット)CPUはキャッシュ・メモリをアクセスするだけですが、なかった場合(ミス)は外部メモリをアクセスして新たにキャッシュ・メモリの内容を入れ替えます。外部メモリは、多くのケースはDDRメモリが使われ、本稿ではQSPI SRAMを使います。これらの外部メモリは、最初に先頭アドレス(ロウ・アドレス)を入れ、その後連続したアドレス(カラム・アドレス)のデータを順にアクセスする方法を採ります。このため、バラバラのアドレスに対するランダム・アクセスは性能が出ませんが、連続したアドレスに対するアクセスは性能が出ます。キャッシュの内容を入れ替える(リプレース)時に、外部メモリに対してアドレスが連続したまとまったデータアクセスすることができるので、性能向上を図ることができます。

bfCPUシステムでは、命令用のキャッシュ(命令キャッシュ)とデータ用のキャッシュ(データ・キャッシュ)を独立して持たせています。これはbfCPUは命令専用のIFバスとデータ専用のDMバスを独立して持っているためです。CPUアーキテクチャによっては、命令キャッシュとデータ・キャッシュを合体したユニファイド・キャッシュとして構成するケースもあります。

1.18 bfCPU の命令キャッシュ

1.18.1 命令キャッシュの構造

bfCPU の命令キャッシュの仕様は以下の通りです。

- メモリ容量: 4bit × 8word × 4 エントリ (16byte)
- ライン長: 4bit × 8word (4byte) (リプレースする単位)
- エントリ数(ライン本数): 4 エントリ (ライン × 4 本)
- ダイレクト・マップ方式: キャッシュ・メモリとメイン・メモリのアドレス位置の対応について、メイン・メモリのバイト単位のアドレス下位 4bit=16byte の塊を、本キャッシュ・メモリ全体の 16byte に直接マッピングします。キャッシュ方式によっては、メイン・メモリの一定範囲をキャッシュ・メモリ内の N 組の領域にマッピングする N-way セット・アソシエティ方式もありますが、本事例では簡易化のためダイレクト・マップ方式を採用しました。

bfCPU システムに内蔵する命令キャッシュのブロック図を図 1.21 に示します。このキャッシュにはメモリという名称を使っていますが、小容量なので中身はレジスタで構成しています。命令キャッシュは、大きく分けて 3 つの部分から構成されます。ic_vX はバリッド・ビット、ic_aX はアドレス・アレイ、ic_dX はデータ・アレイです。添字 X は 0~3 でエントリ(ライン)の番号に対応します。ic_vX の初期値は全てゼロです。データ・アレイの部分のサイズがキャッシュの容量に対応します。

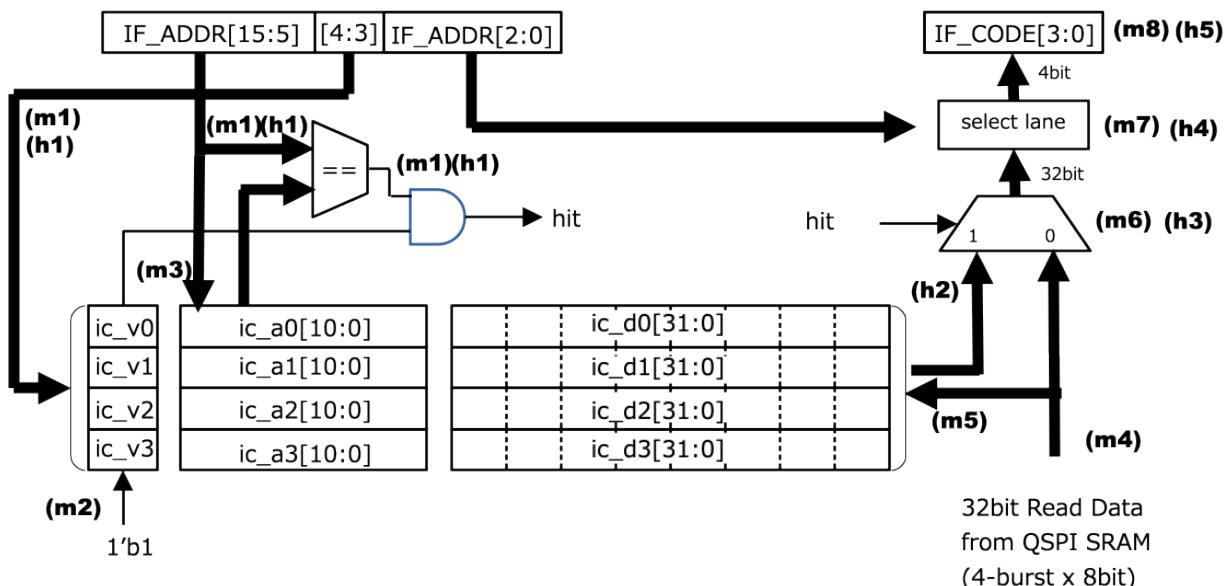


図 1.21: bfCPU の命令キャッシュのブロック図

1.18.2 命令キャッシュの動作

以下、この命令キャッシュの動作を説明します。動作タイミングを図 1.22 に示します。

(1) 命令キャッシュがミスした場合

命令キャッシュがミスした場合の動作を説明します。m1、m2、...などの記号は、図中の記号に対応しています。

- m1: bfCPU が IF バス経由で命令フェッチします。アドレス IF_ADDR[4:3] で 4 エントリの中から 1

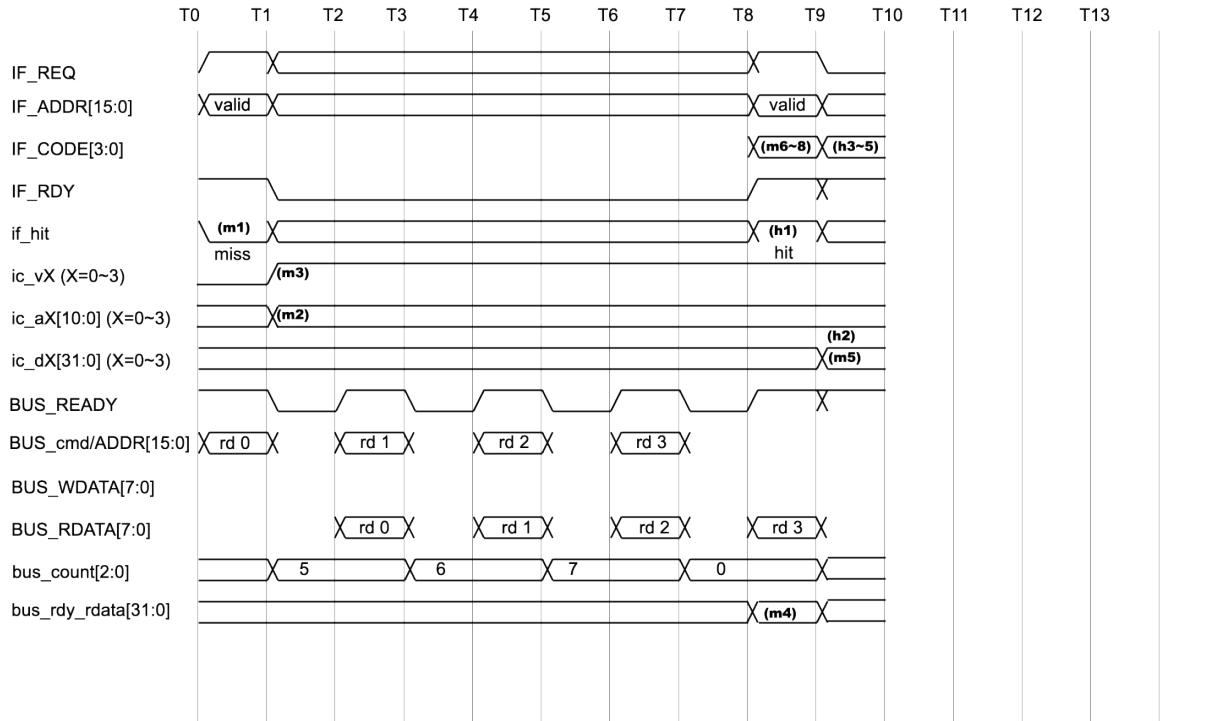


図1.22: bfCPUの命令キャッシュの動作タイミング

つ選択し、 $ic_vX == 0$ または $ic_aX != IF_ADDR[15:5]$ であればミスと判定し、信号 hit は 0 になります。

- **m2**: ミスなので選択したエントリのキャッシュのラインをリプレースすることになります。後でこのラインの内容は確定するのでパリッド・ビット ic_vX を 1 にセットしておきます。
- **m3**: 同様に、選択したエントリのアドレス・アレイ ic_aX に、 $IF_ADDR[15:5]$ の値をライトしておきます。
- **m4**: IF_ADDR に対応する外部メモリ (QSPI SRAM) をリードします。ライン長 (4 バイト) 分を連續アクセスします。
- **m5**: 選択したエントリのデータ・アレイ ic_dX に外部メモリからリードした命令コードをライト (上書き) します。
- **m6**: 信号 hit が 0 なのでマルチ・プレクサが外部メモリからリードした 1 ライン分の命令コードを選択します。
- **m7**: 選択した 1 ライン分の命令コードから $IF_ADDR[2:0]$ の値に対応する 4bit を選択します。
- **m8**: 選択した 4bit を IF_CODE に出力し、bfCPU が取り込みます。

(2) 命令キャッシュがヒットした場合

命令キャッシュがヒットした場合の動作を説明します。h1、h2、...などの記号は、図中の記号に対応しています。

- **h1**: bfCPU が IF バス経由で命令フェッチします。アドレス $IF_ADDR[4:3]$ で 4 エントリの中から 1 つ選択し、 $ic_vX == 1$ かつ $ic_aX == IF_ADDR[15:5]$ であればヒットと判定し、信号 hit は 1 になります。
- **h2**: ヒットなので、選択したエントリのデータ・アレイ ic_dX から 1 ライン分の命令コードを取り出します。
- **h3**: 信号 hit が 1 なのでマルチ・プレクサがデータ・アレイ ic_dX から取り出した 1 ライン分の命令

コードを選択します。

- **h4:** 選択した 1 ライン分の命令コードから IF_ADDR[2:0] の値に対応する 4bit を選択します。
- **h5:** 選択した 4bit を IF_CODE に出力し、bfCPU が取り込みます。

1.19 bfCPU のデータ・キャッシュ

1.19.1 データ・キャッシュの構造

bfCPU のデータ・キャッシュの仕様は以下の通りです。

- **メモリ容量:** 8bit × 4word × 2 エントリ (8byte)
- **ライン長:** 8bit × 4word (4byte) (リプレースする単位)
- **エントリ数(ライン本数):** 2 エントリ (ライン × 2 本)
- **ダイレクト・マップ方式:** キャッシュ・メモリとメイン・メモリのアドレス位置の対応について、メイン・メモリのバイト単位のアドレス下位 3bit=8byte の塊を、本キャッシュ・メモリ全体の 8byte に直接マッピングします。キャッシュ方式によっては、メイン・メモリの一定範囲をキャッシュ・メモリ内の N 組の領域にマッピングする N-way セット・アソシアティブ方式もありますが、本事例では簡易化のためダイレクト・マップ方式を採用しました。
- **ライト・バック方式:** CPU がライトした時、ライト先アドレスのデータがデータ・キャッシュのライン内にある場合(ヒット)、ライン内のデータ・アレイにライトします。キャッシュがミスしてそのラインをリプレースする時、ライン内のデータをまず外部メモリの対応するアドレスに書き出してから、外部メモリの新たなアドレスからデータをリードしてライン内のデータ・アレイを更新します。この動作をライト・バックといいます。一時的に、キャッシュ内容が先に更新され、外部メモリの更新が後回しになり、キャッシュと外部メモリの一貫性(コヒーレンシ)が保たれない時間が生じます。この方式は、同じキャッシュをアクセスする CPU コアが複数ある場合は共有データの一貫性が取れないため対策が必要になりますが、本事例は CPU が 1 個なので問題になりません。キャッシュ方式によっては CPU がライトした時、ヒットしてもライト・データを外部メモリに書き出すライト・スルーワークを採用するケースもありますが、外部メモリの速度に影響を受ける可能性もあるので採用しないことが多いようです。

bfCPU システムに内蔵するデータ・キャッシュのプロック図を図 1.23 に示します。このキャッシュにもメモリという名称を使っていますが、小容量なので中身はレジスタで構成しています。データ・キャッシュは、大きく分けて 3 つの部分から構成されます。dc_vX はバリッド・ビット、dc_aX はアドレス・アレイ、dc_dX はデータ・アレイです。添字 X は 0~1 でエントリ(ライン)の番号に対応します。dc_vX の初期値は全てゼロです。データ・アレイの部分のサイズがキャッシュの容量に対応します。

1.19.2 データ・キャッシュの動作

以下、このデータ・キャッシュの動作を説明します。データ・キャッシュの動作は命令キャッシュよりも少し複雑で、リードとライトの違いとライト・バックの有無によって動きが変わります。

- リード・アクセスでミス時のライト・バックがない場合: 図 1.24
- リード・アクセスでミス時のライト・バックがある場合: 図 1.25
- ライト・アクセスでミス時のライト・バックがない場合: 図 1.26
- ライト・アクセスでミス時のライト・バックがある場合: 図 1.27

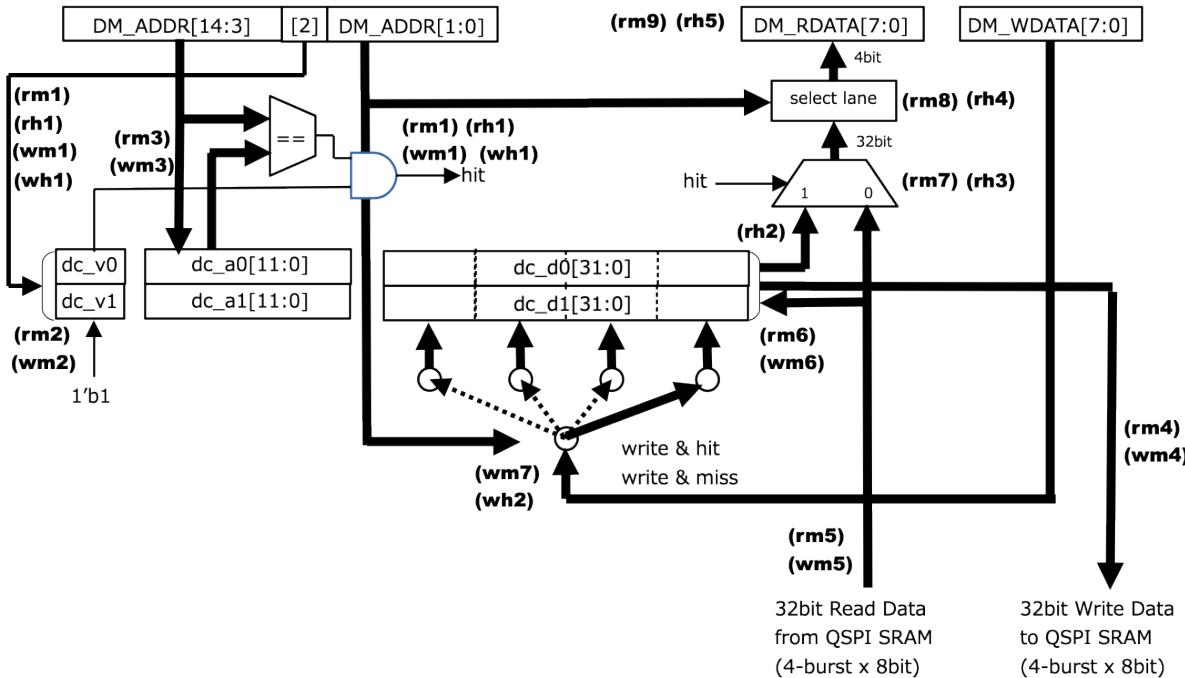


図 1.23: bfCPU のデータ・キャッシュのブロック図

(1) データ・キャッシュのリード・アクセスがミスした場合

rm1、rm2、…などの記号は、図中の記号に対応しています。

- **rm1:** bfCPU が DM バス経由でデータ・リードします。アドレス DM_ADDR[2] で 2 エントリの中から 1 つ選択し、 $dc_vX == 0$ または $dc_aX != DM_ADDR[14:3]$ であればミスと判定し、信号 hit は 0 になります。
- **rm2:** ミスなので選択したエントリのキャッシュのラインをリプレースすることになります。後でこのラインの内容は確定するのでバリッド・ビット dc_vX を 1 にセットしておきます。その前にこのエントリのバリッド・ビット dc_vX が 0 だったら、そのラインはまだ CPU がデータをライトしていないでライト・バックの必要がありません。 dc_vX が 1 だったら、そのラインは CPU がデータをライトしている可能性があるのでライト・バックの必要があります。キャッシュ方式によっては CPU がデータをライトしたかどうかを示すダーティ・ビットを用意して、CPU がライン内にライトしていなければ確実にライト・バックしないようにする方式もありますが、ここでは簡単化のため採用しませんでした。
- **rm3:** 同様に、選択したエントリのアドレス・アレイ dc_aX に、 $DM_ADDR[14:3]$ の値をライトしておきます。
- **rm4:** ライト・バックが必要な場合は、データ・アレイ dc_dX のライン内データを外部メモリ (QSPI SRAM) にライトします。ライン長 (4 バイト) 分を連続ライトします。
- **rm5:** DM_ADDR に対応する外部メモリ (QSPI SRAM) をリードします。ライン長 (4 バイト) 分を連続リードします。
- **rm6:** 選択したエントリのデータ・アレイ dc_dX に外部メモリからリードしたデータをライト (上書き) します。
- **rm7:** 信号 hit が 0 なのでマルチ・ブレクサが外部メモリからリードした 1 ライン分のデータを選択します。
- **rm8:** 選択した 1 ライン分のデータから $DM_ADDR[1:0]$ の値に対応する 8bit を選択します。
- **rm9:** 選択した 8bit を DM_RDATA に出力し、bfCPU が取り込みます。

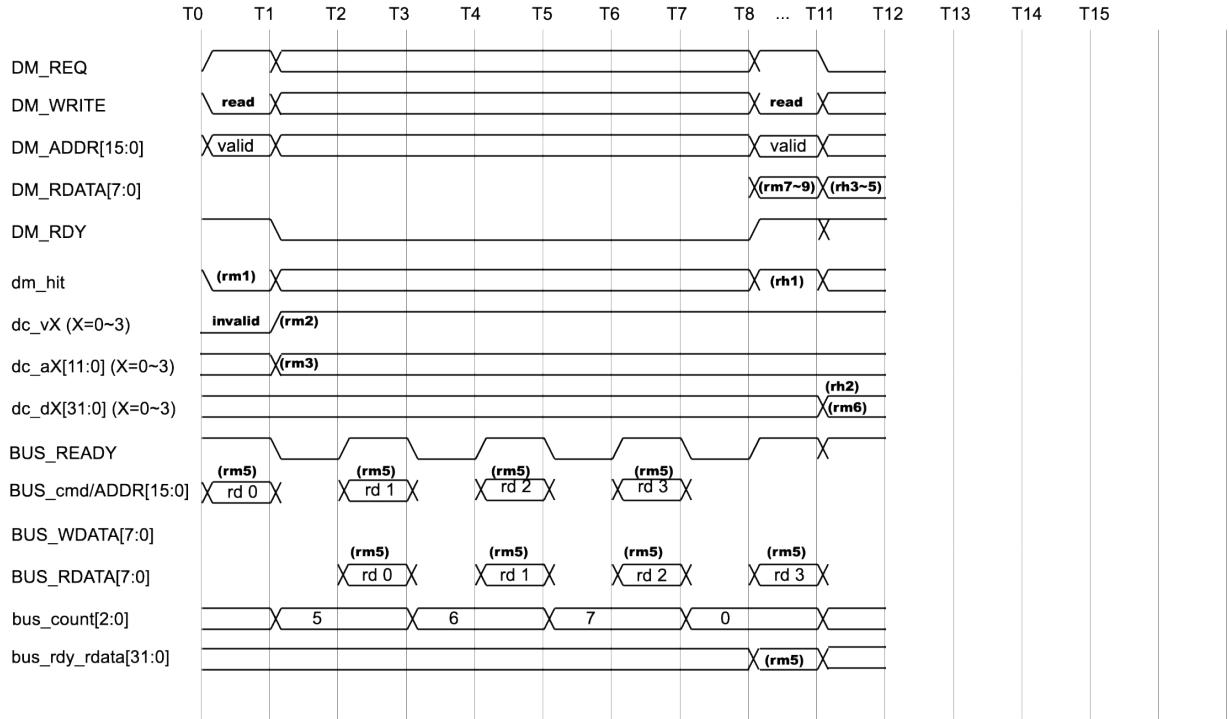


図 1.24: bfCPU のデータ・キャッシュの動作タイミング (リード・アクセスでライト・バックがない場合)

(2) データ・キャッシュのリード・アクセスがヒットした場合

rh1, rh2, ...などの記号は、図中の記号に対応しています。

- **rh1:** bfCPU が DM バス経由でデータ・リードします。アドレス DM_ADDR[2] で 2 エントリの中から 1 つ選択し、dc_vX==1 かつ dc_aX==DM_ADDR[14:3] であればヒットと判定し、信号 hit は 1 になります。
- **rh2:** ヒットなので、選択したエントリのデータ・アレイ dc_dX から 1 ライン分のデータを取り出します。
- **rh3:** 信号 hit が 1 なのでマルチ・プレクサがデータ・アレイ dc_dX から取り出した 1 ライン分のデータを選択します。
- **rh4:** 選択した 1 ライン分のデータから DM_ADDR[1:0] の値に対応する 8bit を選択します。
- **rh5:** 選択した 8bit を DM_RDATA に出力し、bfCPU が取り込みます。

(3) データ・キャッシュのライト・アクセスがミスした場合

wm1, wm2, ...などの記号は、図中の記号に対応しています。

- **wm1:** bfCPU が DM バス経由でデータ・ライトします。アドレス DM_ADDR[2] で 2 エントリの中から 1 つ選択し、dc_vX==0 または dc_aX!=DM_ADDR[14:3] であればミスと判定し、信号 hit は 0 になります。
- **wm2:** ミスなので選択したエントリのキャッシュのラインをリプレースすることになります。後でこのラインの内容は確定するのでバリッド・ビット dc_vX を 1 にセットしておきます。その前にこのエントリのバリッド・ビット dc_vX が 0 だったら、そのラインはまだ CPU がデータをライトしていないでライト・バックの必要がありません。dc_vX が 1 だったら、そのラインは CPU がデータをライトしている可能性があるのでライト・バックの必要があります。
- **wm3:** 同様に、選択したエントリのアドレス・アレイ dc_aX に、DM_ADDR[14:3] の値をライトしておきます。

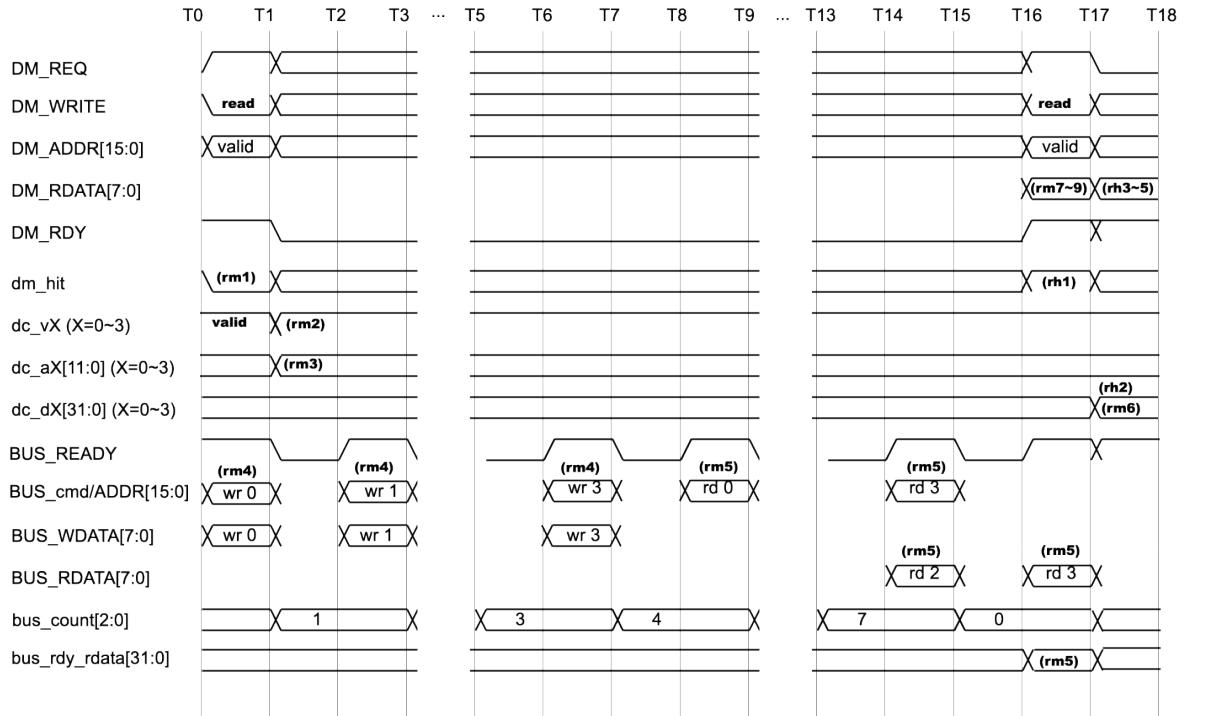


図 1.25: bfCPU のデータ・キャッシュの動作タイミング (リード・アクセスでライト・バックがある場合)

- **wm4**: ライト・バックが必要な場合は、データ・アレイ `dc_dX` のライン内データを外部メモリ (QSPI_SRAM) にライトします。ライン長 (4 バイト) 分を連続ライトします。
- **wm5**: `DM_ADDR` に対応する外部メモリ (QSPI SRAM) をリードします。ライン長 (4 バイト) 分を連続リードします。
- **wm6**: 選択したエントリのデータ・アレイ `dc_dX` に外部メモリからリードしたデータをライト (上書き) します。
- **wm7**: それと合わせて、bfCPU がライトしようとしていたデータ `DM_WDATA` を、`DM_ADDR[1:0]` が示すライン内の位置にライト (上書き) します。

(4) データ・キャッシュのライト・アクセスがヒットした場合

wh1、wh2 の記号は、図中の記号に対応しています。

- **wh1**: bfCPU が DM バス経由でデータ・ライトします。アドレス `DM_ADDR[2]` で 2 エントリの中から 1 つ選択し、`dc_vX==1`かつ `dc_aX==DM_ADDR[14:3]` であればヒットと判定し、信号 `hit` は 1 になります。
- **wh2**: ヒットなので、bfCPU がライトしようとしていたデータ `DM_WDATA` を、D アドレス `DM_ADDR[2]` が示すライン内の `DM_ADDR[1:0]` が示す位置にライト (上書き) します。

1.20 キャッシュ・メモリの RTL 記述

キャッシュ・メモリ (命令キャッシュ + データ・キャッシュ) の RTL 記述は、リポジトリ内の bfCPU/RTL/CACHE/cache.v です。モジュール宣言部 (入出力信号) をリスト 1.15 に示します。入出力信号は、クロック、リセット、IF バス、DM バス、BUS バスです。BUS バスは QSPI SRAM インタフェースをアクセスします。命令キャッシュのミスとデータ・キャッシュのミスが重なって、両方が同時に BUS バスを

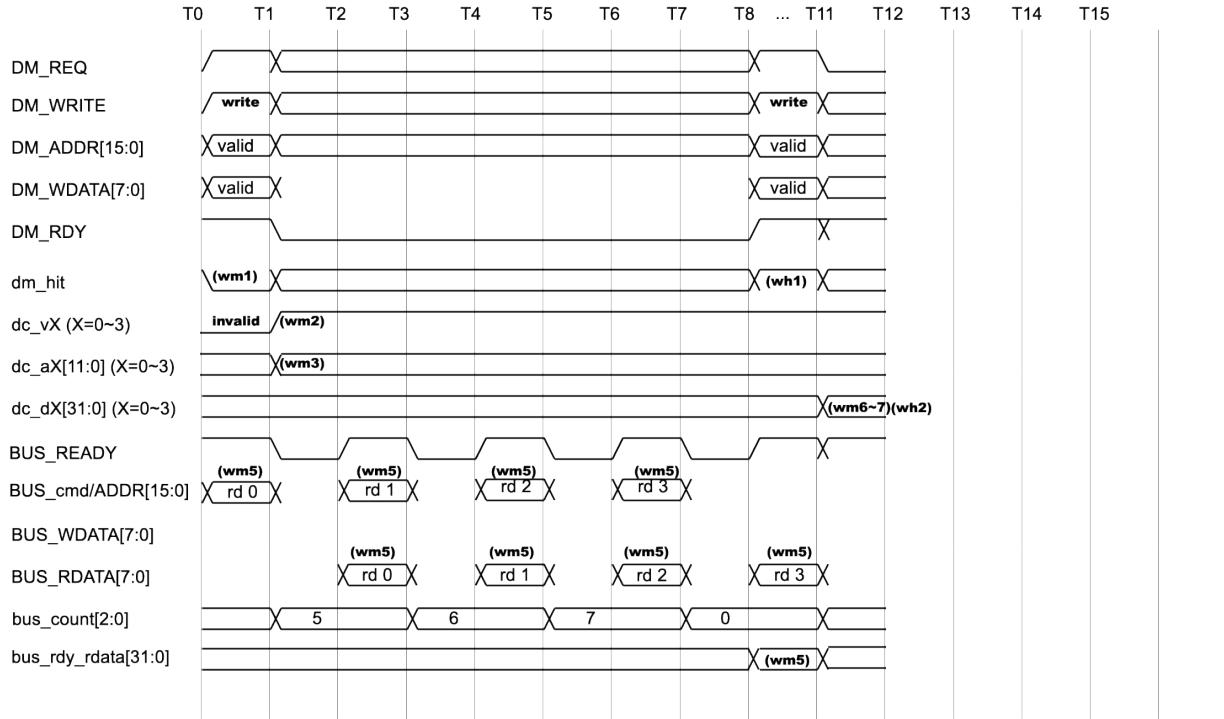


図 1.26: bfCPU のデータ・キャッシュの動作タイミング (ライト・アクセスでライト・バックがない場合)

アクセスしようとした場合は、データ・キャッシュ側のアクセスを優先します。

リスト 1.15: キャッシュ・メモリの RTL 記述 cache.sv の モジュール宣言部

```
module CACHE
(
    // System Signal
    input  logic CLK,
    input  logic RES,
    //
    // CPU Instruction Fetch
    //    PC Address (64kwords x 4bits)
    input  logic      IF_REQ,
    input  logic [15:0] IF_ADDR,
    output logic [ 3:0] IF_CODE,
    output logic      IF_RDY,
    //
    // CPU Data Memory Access
    //    PTR Address (32kbytes x 8bits)
    input  logic      DM_REQ,
    input  logic      DM_WRITE,
    input  logic [14:0] DM_ADDR,
    input  logic [ 7:0] DM_WDATA,
    output logic [ 7:0] DM_RDATA,
    output logic      DM_RDY,
    //
    // External Bus Access
    //    64kbytes x 8bits
    output logic      BUS_REQ,
    output logic      BUS_WRITE,
    output logic [15:0] BUS_ADDR,
    output logic [ 7:0] BUS_WDATA,
    input  logic [ 7:0] BUS_RDATA,
    input  logic      BUS_RDY
);
```

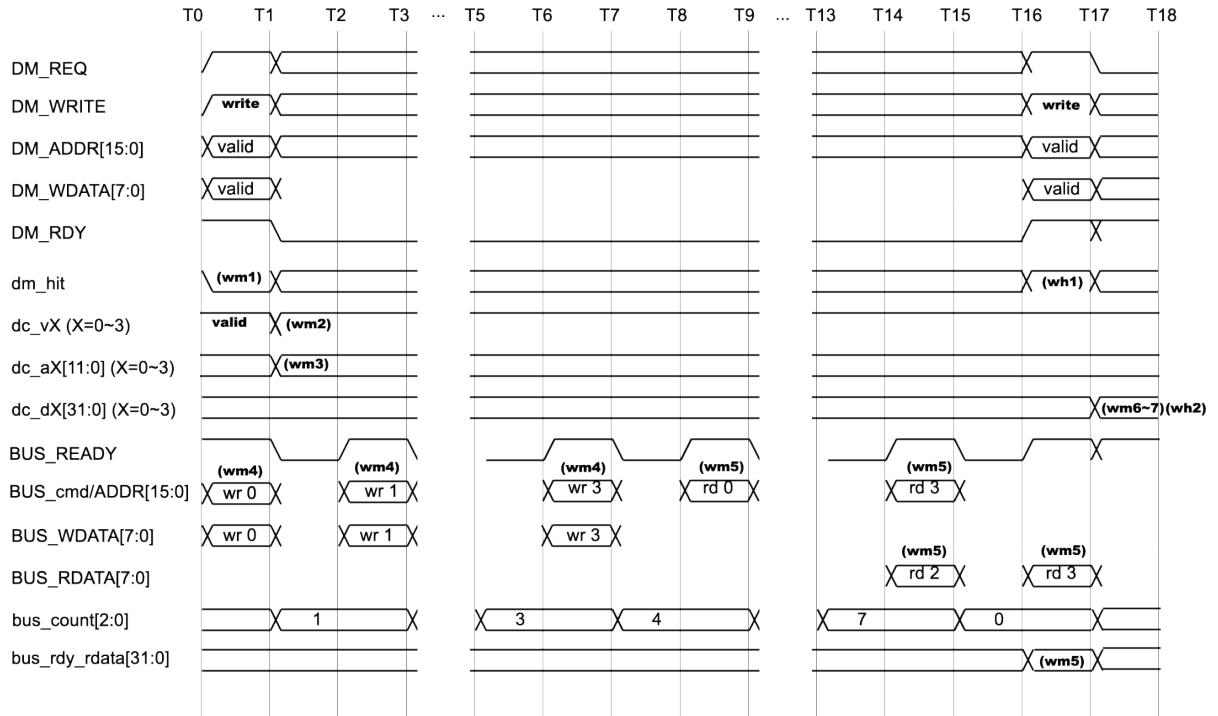


図 1.27: bfCPU のデータ・キャッシュの動作タイミング (ライト・アクセスでライト・バックがある場合)

);

なお、bfCPU の論理規模を小さくするためキャッシュ・メモリを実装しない場合は、先頭付近にある下記の行をコメント・アウトしてください。bfCPU からくる IF バスと DM バスを優先順位調停するだけで BUS バスを生成するようになります。

```
`ifndef USE_CACHE
```

1.21 外部メモリ QSPI SRAM とそのインターフェース

1.21.1 Microchip 23LC512-I/P

bfCPU システムの外部メモリとして、シリアル型 SRAM (QSPI SRAM) を使います。型名は 23LC512-I/P (Microchip) で通販や秋葉原の電子部品店などから入手できます。外形、端子配置、端子機能を図 1.28 に示します。シリアル・クロック (SCK) は最大 25MHz ですが、本事例では bfCPU を約 10MHz で動作させ、その周波数でシリアル・クロック (SCK) も動かします。シリアル・クロック (SCK) は、bfCPU 内部のシステム・クロック (CLK) を反転して出力します。QSPI SRAM は初期状態でシーケンシャル・モードになっていて、その状態のまま使います。

1.21.2 SPI モードから QSPI モードへの変更方法

この QSPI SRAM は電源立ち上がり後の初期状態は、1bit 幅のシリアル通信を行う SPI モードになります。4bit 幅の QSPI モードにするために図 1.29 に示すコマンドを送信する必要があります。bfCPU の QSPI SRAM インタフェースは、リセット直後にこのコマンドを自動的に送信するようにしました。その間、BUS パ



図 1.28: QSPI SRAM Microchip 23LC512-I/P (Microchip 社データシートから引用)

スの BUS_RDY はネゲートしてキャッシュ・メモリからバス・アクセスがきても待たせるようにしています。

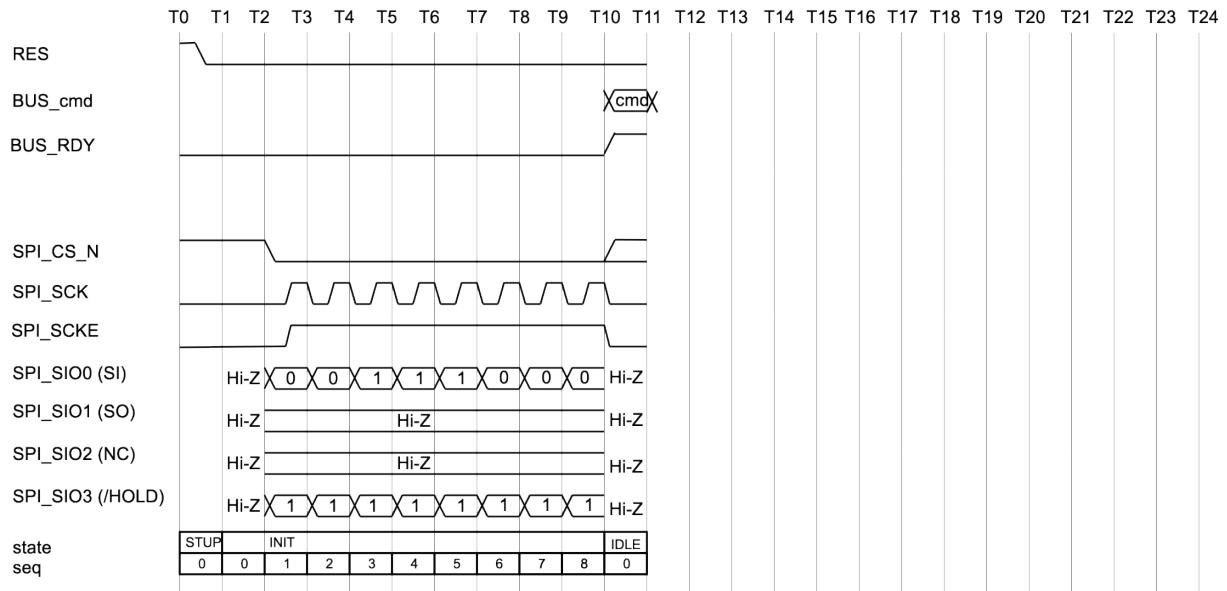


図 1.29: SPI モードから QSPI モードへの変更コマンド

1.21.3 QSPI モードから SPI モードへの変更方法

QSPI SRAM にプログラムを書き込むのは外部の Raspberry Pi 5 ボード（ラズパイ）が行います。ラズパイが bfCPU システムをリセットしている間は bfCPU システムが出力する QSPI SRAM インタフェース信号が Hi-Z になるので、ラズパイが QSPI SRAM に QSPI インタフェースでプログラムします。ラズパイも QSPI モードを使いますが、ラズパイ側は書き込み動作の前、念のために QSPI SRAM を SPI モードに戻してから再度 QSPI モードに入れる操作をします。この QSPI モードから SPI モードへの変更する方法を図 1.30 に示します。

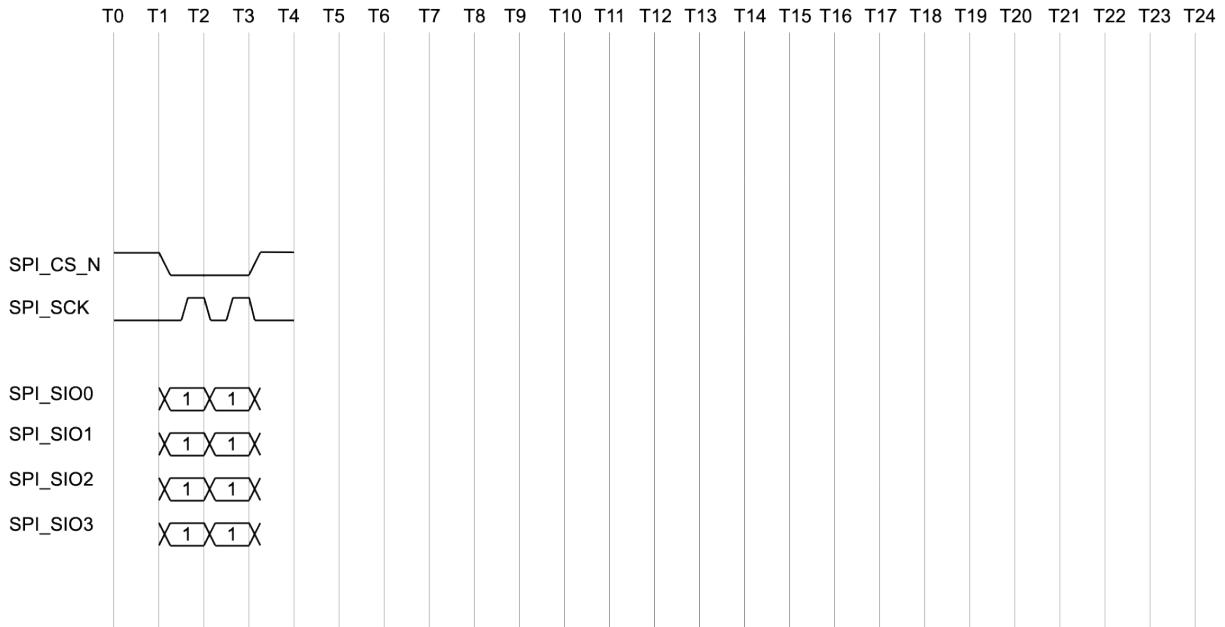


図 1.30: QSPI モードから SPI モードへの変更コマンド

1.21.4 QSPI SRAM のリード・タイミング

QSPI SRAM のリード・タイミングを図 1.31 に示します。キャッシュ・メモリから出力されている BUS バスからリード・アクセスがあると、SIO[3:0] 端子にコマンド 8bit 分とアドレス 16bit 分をそれぞれ 4bit 単位で入力し、2 サイクル (8bit 分) のダミー・サイクルのあと、SIO[3:0] 端子からリード・データが output されます。クロック SCK を印加し続ければ、隙間なくシーケンシャルにアドレスをインクリメントしながらリード・データが出力されますが、1 バイト分のリード・データを BUS_RDATA に載せて BUS_RDY をアサートしたときに確定しているアドレス BUS_ADDR で連続アクセスなのかどうかを判定する必要があります、そのためバイトとバイトの連続アクセスの間も 1 サイクル間を空けました。隙間なくリードを継続する場合は投機的なリードになるケースがあり制御が複雑になるので、このタイミングにしました。

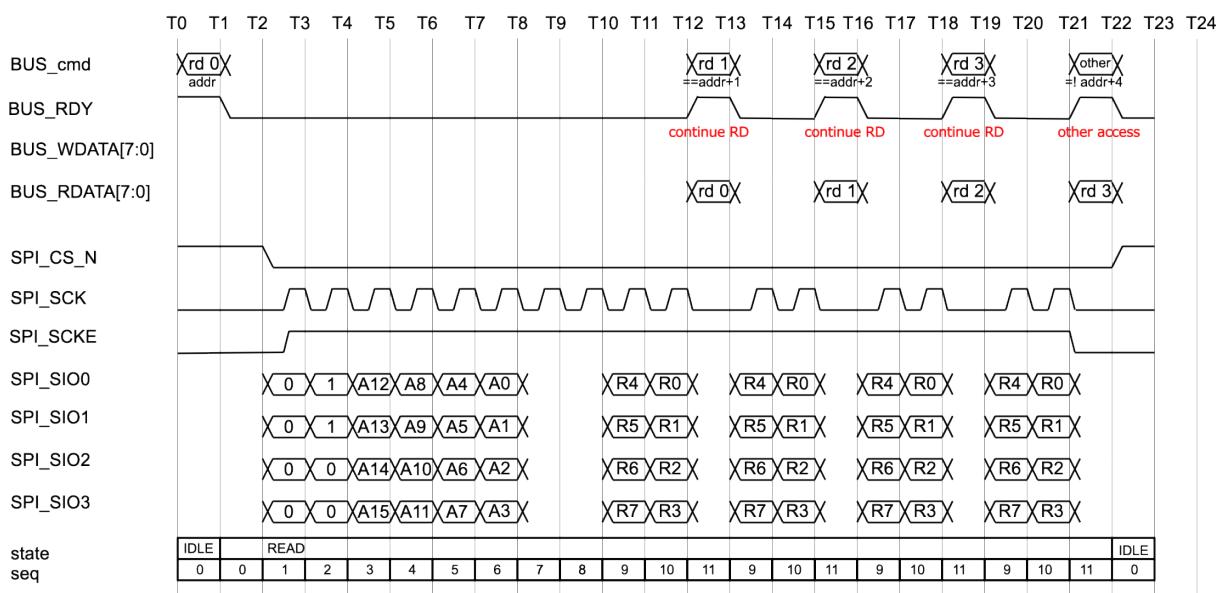


図 1.31: QSPI SRAM のリード・タイミング

1.21.5 QSPI SRAM のライト・タイミング

QSPI SRAM のライト・タイミングを図 1.32 に示します。キャッシュ・メモリから出力されている BUS バスからライト・アクセスがあると、SIO[3:0] 端子にコマンド 8bit 分とアドレス 16bit 分をそれぞれ 4bit 単位で入力し、ダミー・サイクルなくそのままライト・データを入力し続けることができます。ライト・アクセスの場合は、アドレス BUS_ADDR とライト・データ BUS_WDATA が同一タイミングであり、リード・アクセスで入れた 1 サイクルの隙間を入れる必要はなく、連続ライト可能です。

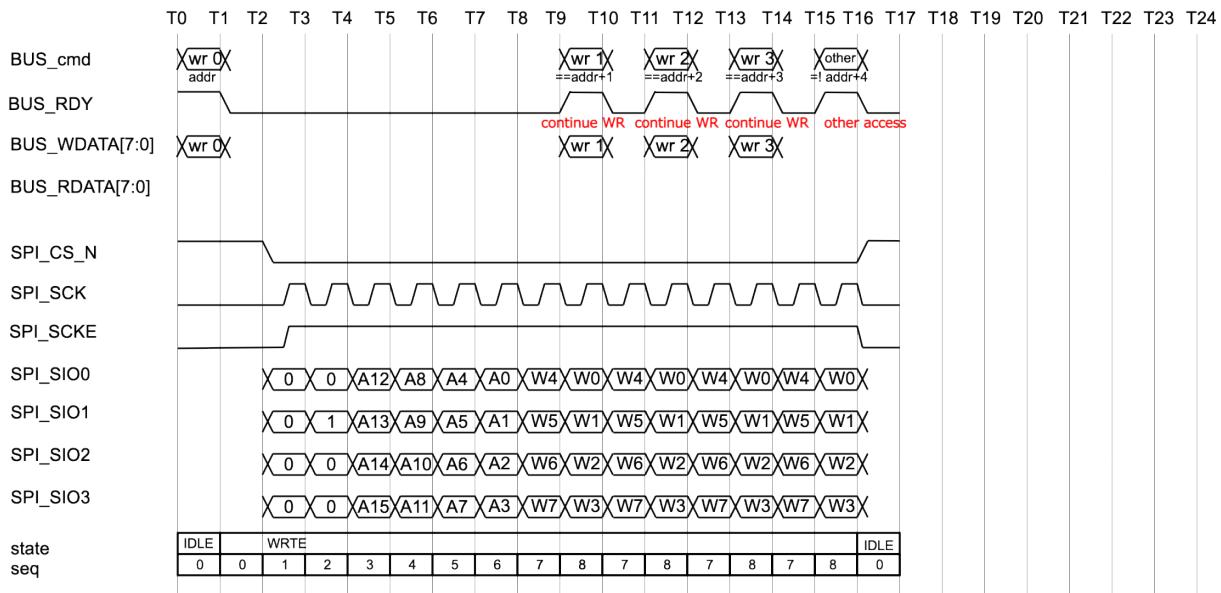


図 1.32: QSPI SRAM のライト・タイミング

1.22 QSPI SRAM インタフェースの RTL 記述

QSPI SRAM インタフェースの RTL 記述は、リポジトリ内の bfCPU/RTL/QSPI_SRAM/qspi_sram.sv です。モジュール宣言部(入出力信号)をリスト 1.16 に示します。入出力信号は、クロック、リセット、BUS バス、QSPI SRAM インタフェース信号です。QSPI_SRAM インタフェースのうち、送受信データ信号 QSPI_SIO[3:0] は双方向の入出力信号ですが、入出力バッファは最上位記述に入れるため、QSPI_SRAM モジュールでは出力 QSPI_SIO_O[3:0]、出力イネーブル QSPI_SIO_OE[3:0]、入力 QSPI_SIO_I[3:0] に分けてあります。

リスト 1.16: QSPI SRAM インタフェースの RTL 記述 qspi_sram.sv の モジュール宣言部

```
module QSPI_SRAM
(
    input  logic CLK,
    input  logic RES,
    //
    input  logic      BUS_REQ,
    input  logic      BUS_WRITE,
    input  logic [15:0] BUS_ADDR,
    input  logic [ 7:0] BUS_WDATA,
    output logic [ 7:0] BUS_RDATA,
    output logic      BUS_RDY,
    //
    output logic      QSPI_CS_N,
```

```

output logic      QSPI_SCK,
output logic [3:0] QSPI_SI0_0,
output logic [3:0] QSPI_SI0_E,
input  logic [3:0] QSPI_SI0_I
);

```

1.23 入出力用 UART モジュール

1.23.1 UART のコア

“IN 命令”と“OUT 命令”的データ入出力に使う UART モジュールは、そのコアとしてオープンソースの IP SASC(Simple Asynchronous Serial Controller: <https://opencores.org/projects/sasc>) を使いました。この IP の外側に IO バスとのインターフェースを被せてあります。

1.23.2 UART のレジスタ

IO バスのアドレス IO_ADDR は 2bit です。UART 内のレジスタのアドレス配置を表 1.2 に示します。

表 1.2: UART 内のレジスタのアドレス配置

アドレス	レジスタ	機能
2'b00	TXD/RXT	送受信データ・レジスタ
2'b10	DIV0	ボーレート設定レジスタ 0
2'b11	DIV1	ボーレート設定レジスタ 1

ボーレートはレジスタ DIV0 と DIV1 で設定し、以下の計算式で決まります。fCLK は bfCPU システムの動作クロック周波数です。

$$\text{Baud Rate} = \frac{f\text{CLK}/4}{(\text{DIV0} + 2) \times (\text{DIV1})}$$

bfCPU システムの動作クロック CLK(約 10MHz) はラズパイの GPIO から出力されるクロックを使いますが、正確に 10MHz になるとは限らないので、DIV0 と DIV1 を固定化することはできません。このため、QSPI SRAM をラズパイが初期化するとき、QSPI SRAM の最後のアドレス 0xFFFFE に DIV0 の値を、アドレス 0xFFFF に DIV1 の値をライトしておきます。bfCPU はリセット解除直後にその番地をリードして UART のボーレートを設定します。基本的にボーレートは 115200bps になるように設定し、UART のデータ・フォーマットは 8 ビット・ノンパリティ、STOP ビット 1(8N1) で通信します。

1.23.3 UART の IO バスとのインターフェース論理

UART の IO バスとのインターフェース論理は、リポジトリ内の bfCPU/RTL/UART/uart.sv です。この下位に SASC を含んでいます。IO バスのアクセス・タイミングを図 1.33 に示します。

レジスタ DIV0 と DIV1 のアクセスはノー・ウェイトで行われます。

送信データは TXD レジスタにライトすると送信開始しますが、送信バッファが満杯だとそのライト・アクセスにはウェイトが入り待たされ、送信バッファが空けばライトが完了し、送信開始します。送信バッファは 4 段の FIFO(First-In-First-Out) なので送信中の連続ライトが可能です。

受信データは RXD レジスタをリードして取り出します。受信バッファが空だとこのリード・アクセスにウェイトが入り、受信して受信バッファにデータが入るとそのリードが完了します。受信バッファも 4 段の FIFO で

構成されているので、RXD レジスタのリードがない間も連続受信が可能ですが、受信バッファが溢れるとオーバラン・エラーとなり UART の動作は不定になるので注意してください。

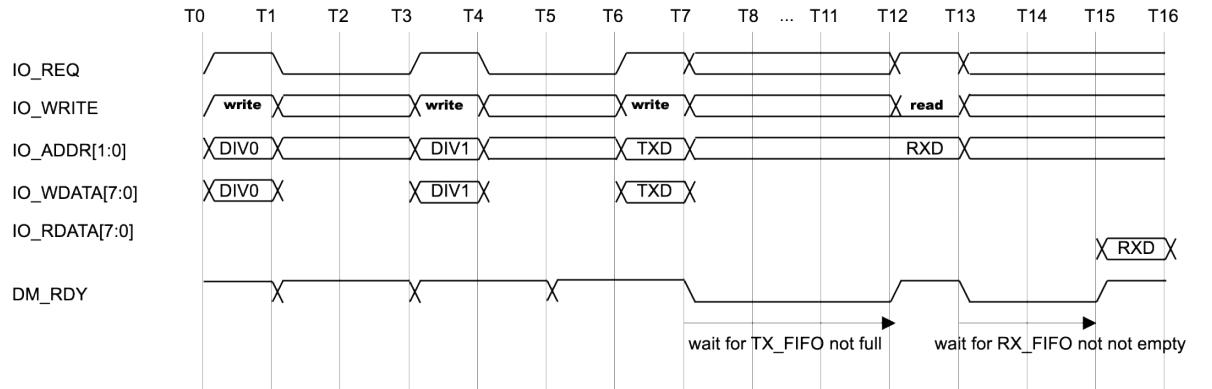


図 1.33: UART のアクセス・タイミング

1.24 UART の RTL 記述

UART の最上位 RTL 記述は、リポジトリ内の bfCPU/RTL/UART/uart.sv です。UART のモジュール宣言部（入出力信号）をリスト 1.17 に示します。

リスト 1.17: UART 最上位階層の RTL 記述 uart.sv の モジュール宣言部

```
module UART
(
    input logic CLK,
    input logic RES,
    //
    input logic      IO_REQ,
    input logic      IO_WRITE,
    input logic [1:0] IO_ADDR,
    input logic [7:0] IO_WDATA,
    output logic [7:0] IO_RDATA,
    output logic     IO_RDY,
    //
    output logic UART_TXD,
    input logic  UART_RXD
);
```

1.25 bfCPU システム全体の構成

1.25.1 システム全体の階層構造

bfCPU システムの全体階層構造を図 1.34 に示します。この図は FPGA に実装するときの階層を示します。Tiny Tapeout 向けの階層は次章で説明します。

bfCPU の最上位 RTL 記述は、リポジトリ内の bfCPU/RTL/TOP/top.sv および fpga.sv です。これまで説明した各モジュールは階層 TOP(top.sv : リスト 1.18) の下に入ります。最上位階層 FPGA(fpga.sv : リスト 1.19) は、TOP 階層と QSPI SRAM インタフェース信号の I/O バッファが入ります。

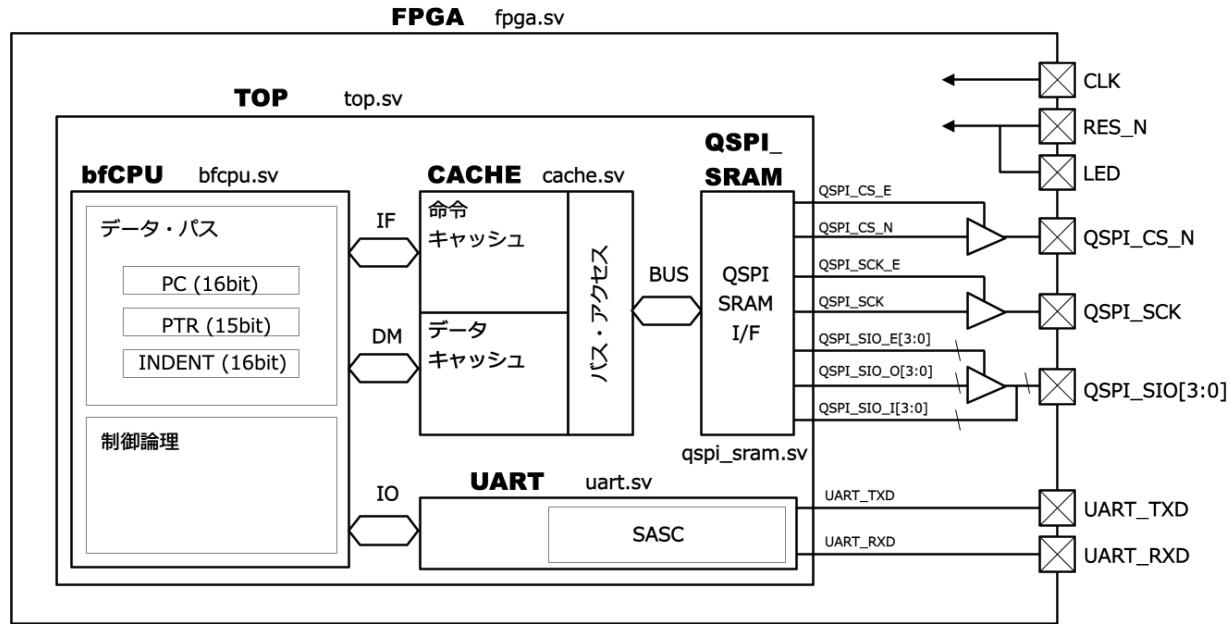


図 1.34: bfCPU の全体階層構造

リスト 1.18: TOP 階層の RTL 記述 top.sv の モジュール宣言部

```
module TOP
(
    input logic CLK,
    input logic RES_N,
    //
    output logic     QSPI_CS_N,
    output logic     QSPI_CS_E,
    output logic     QSPI_SCK,
    output logic     QSPI_SCK_E,
    output logic [3:0] QSPI_SIO_0,
    output logic [3:0] QSPI_SIO_E,
    input  logic [3:0] QSPI_SIO_I,
    //
    output logic UART_TXD,
    input  logic UART_RXD
);
```

リスト 1.19: FPGA 最上位階層の RTL 記述 fpga.sv の モジュール宣言部

```
module FPGA
(
    input logic CLK,
    input logic RES_N,
    //
    inout wire     QSPI_CS_N,
    inout wire     QSPI_SCK,
    inout wire [3:0] QSPI_SIO,
    //
    output logic UART_TXD,
    input  logic UART_RXD,
    //
    output logic LED
);
```

1.25.2 QSPI SRAM 信号の扱い

外部におかれれる QSPI SRAM は、bfCPU システム側とラズパイ側の両方からアクセスされ、bfCPU をリセット中はラズパイが駆動し、リセット解除中（動作中）は bfCPU が駆動します。基本的にはここに信号衝突は起きないですが、ラズパイ側のプログラムにバグがあって信号衝突してしまうと基板類を壊してしまうので、QSPI SRAM へ向かう双方の出力信号はオープン・ドレイン駆動とし外部にプルアップ抵抗をつけることにしました。オープン・ドレインの出力バッファの実現方法としては、バッファへの出力信号は常時 0 とし、バッファの出力イネーブルを元の出力信号が 0 の時イネーブルにし、1 の時ディセーブル（Hi-Z）にします。外部信号 LED は bfCPU のプログラム実行中を示すインジケータとして、リセット信号 RES_N をそのまま出力しており、リセット解除中（動作中）に 1（＝点灯）になるようにしました。

1.26 bfCPU システムの論理シミュレーションによる機能検証

1.26.1 機能検証を使う環境とリソース

機能検証にはオープン・ソースの環境を使います。論理シミュレータは Icarus Verilog を使い、波形ビューワには GtkWave を使います。Ubuntu 環境では、以下のコマンドでインストールできます。

```
$ sudo apt install iverilog gtkwave
```

bfCPU の機能検証関係のリソースはリポジトリ内の bfCPU/SIM/tb_TOP/以下に格納してあります。

1.26.2 テストベンチ tb.sv

検証対象は bfCPU の最上位階層 FPGA であり、その外側に機能検証用の階層すなわちテストベンチを被せます。テストベンチは、リポジトリ内の bfCPU/SIM/tb_TOP/tb.sv です。この中では以下を行っています。

- **波形ファイルの出力:** 全階層の全波形をファイル tb.vcd に出力します。
- **クロックとリセット信号の生成**
- **論理シミュレーションのタイムアウト設定:** シミュレーションを無限に実行してしまうと波形ファイルが肥大化してディスクを圧迫してしまうので、カウンタ tb_cycle_counter をクロックでカウント・アップし、一定の値になったらシミュレーションを止めます。
- **外部メモリ QSPI SRAM の内容初期化:** 機能検証したい bfCPU のプログラムを専用ツール bfTool でアセンブルしたできたバイナリ・コード (*.v) でコマンド \$readmemh() を使って外部メモリ QSPI SRAM(この後でインスタンス化) の内容をシミュレーション実行開始時に初期化します。あわせて UART のボーレートを決める値 (DIV0 と DIV1) を QSPI SRAM の特定アドレスに書き込みます。なお、シミュレーション実行時にはマクロ定義 SIMULATION を有効化します。この場合、シミュレーション時間の短縮のため bfCPU がリセット後に QSPI SRAM を初期化する範囲を 0x8000~0x801d(データ・メモリとして 0x0000~0x001d) に絞り、DIV0 の設定値を 0x801e 番地に、DIV1 の設定値を 0x801f 番地に置くようにしています。
- **検証対象の FPGA をインスタンス化:** 最上位階層モジュール FPGA のインスタンス名を U_FPGA にします。
- **SPI SRAM の動作モデルをインスタンス化:** SPI SRAM(23LC512) の動作モデル (23LC512.v、モジュール名は M23LC512) をインスタンス化します。インスタンス名は U_M23LC512 にします。QSPI SRAM インタフェース信号はオープン・ドレインで駆動されるのでプルアップしておきます。またこれら信号は双方向で複数のドライバが駆動するので wire 売りします。

- **UARTの受信データの生成:** bfCPUが“IN命令”を実行しようとしていることをモニタし、検知したらその度に順に0x02、0x03、0x04、0x05、…をRXD端子に向けて送信して、その値を\$display()文で表示します。シミュレーション開始時にRXD端子のレベルをマーク状態(1)に初期化します。
- **UARTの送信データの表示:** bfCPUのUARTがTXD端子からデータ送信したら、その値を\$display()文で表示します。

1.26.3 論理シミュレーション実行のための準備

論理シミュレーション実行のために以下のファイルを準備します。全てリポジトリ内のbfCPU/SIM/tb_TOP/以下に格納してあります。

- **flist.txt:** 使うRTL記述を全てリスト化し、マクロSIMULATIONを定義します。
- **ROM*.v:** 機能検証したいbfCPUのプログラムを専用ツールbfToolでアセンブルしたできたバイナリ・コード(*.v)を格納します。デフォルトのtb.svでは、乗算プログラムmultiplication.vでQSPI SRAMを初期化します。
- **go_sim:** シミュレーション起動用のスクリプトです。
- **tb.gtkw:** 波形ビューワGtkWaveが書き出し、または読み込む波形表示設定ファイルです。GtkWaveで波形表示した時のフォーマットをこのファイルに保存できます。

1.26.4 論理シミュレーションの実行

カレント・ディレクトリをbfCPU/SIM/tb_TOPにした状態で下記のコマンド実行論理シミュレーションを起動できます。

```
$ ./go_sim
```

Icarus Verilogがシミュレーション実行するとき、まずコマンドiverilogでRTL記述をコンパイルしてファイルtb.vvpを生成します。その後、このファイルをコマンドvvpに入力してシミュレーション本体を実行します。シミュレーション実行により波形ファイルtb.vcdが生成され、メッセージ出力がログ・ファイルlogに格納されます。

1.26.5 波形ファイルの確認

波形ファイルの確認は下記のコマンドで行います。また波形表示設定ファイルがない場合は

```
$ gtkwave tb.vcd
```

波形表示設定ファイルtb.gtkwがある場合は

```
$ gtkwave tb.vcd tb.gtkw
```

を実行します。波形の表示例を図1.35に示します。左上の論理階層をクリックすると左下にその階層の信号が表示されるので、AppendボタンやInsertボタンで右側の波形表示領域に追加できます。現在表示されている波形とその順番などはFileメニューのWrite Save File Asにより波形表示設定ファイルにセーブできます。画面上部のボタンで時間方向の拡大・縮小・移動などができます。あちこち触ってみて使い方を習得してください。この例では乗算プログラムを実行していて、乗数・被乗数が大きくなるにつれ実行時間が長くなっています。bfCPUの特徴がよく表れています。また、乗算プログラム本体を実行中はIFバスのIF_RDYやDMバスのDM_RDYがほとんどネゲートされておらずノーワイトでアクセスできており、キャッシュ・メモリの効果がよく表れていることもわかります。

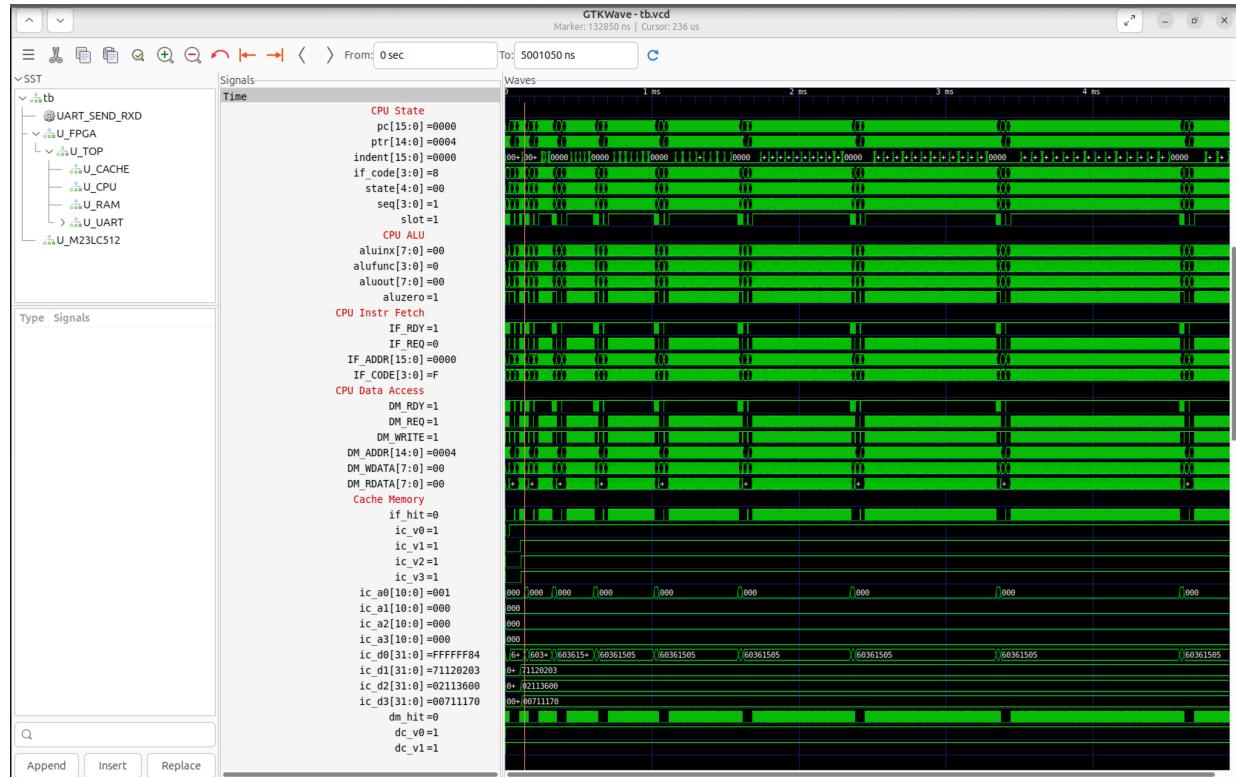


図 1.35: GtkWave で表示したシミュレーション波形

1.27 bfCPU システムを FPGA に実装する

1.27.1 使用する FPGA ボード

設計した bfCPU システムを FPGA に実装します。使う FPGA ボードは台湾 Terasic 社の **DE10-Lite** (<https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=234&No=1021#contents>) です。外観を図 1.36 に示します。電子部品の通販サイトなどで購入できます。

搭載している FPGA は Altera 社 MAX 10 シリーズの 10M50(10M50DAF484C7G) です。ロジック・エレメントを 50K 個、ブロック RAM を 1,638Kbit、乗算器 (18x18) を 144 個、PLL を 4 個、A/D 変換器 (12bit) を 2 個、それぞれ搭載しています。さらに不揮発 FLASH メモリを 5,888Kbit 内蔵しています。FLASH メモリは、FPGA のコンフィグレーション・データとユーザ・データの不揮発記憶の両方が可能です。

1.27.2 bfCPU の FPGA 設計データ

bfCPU の FPGA 設計データはリポジトリ <https://github.com/munetomo-maruyama/bfCPU.git> 内の bfCPU/FPGA の下に格納してあります。

1.27.3 FPGA 開発ツール Quartus Prime

Altera 社の FPGA MAX 10 の開発ツールとしては無償版の Quartus Prime Lite Edition を使います。<https://www.altera.com/products/development-tools/quartus> からダウンロードしてインストールしてください。画面の例を図 1.37 に示します。

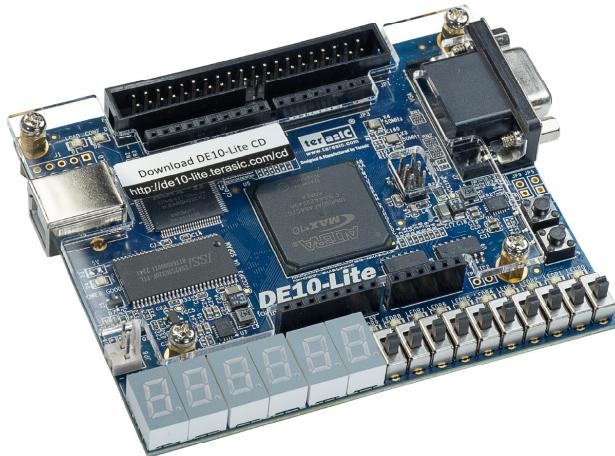


図 1.36: 使用する FPGA ボード Terasic 社 DE10-Lite

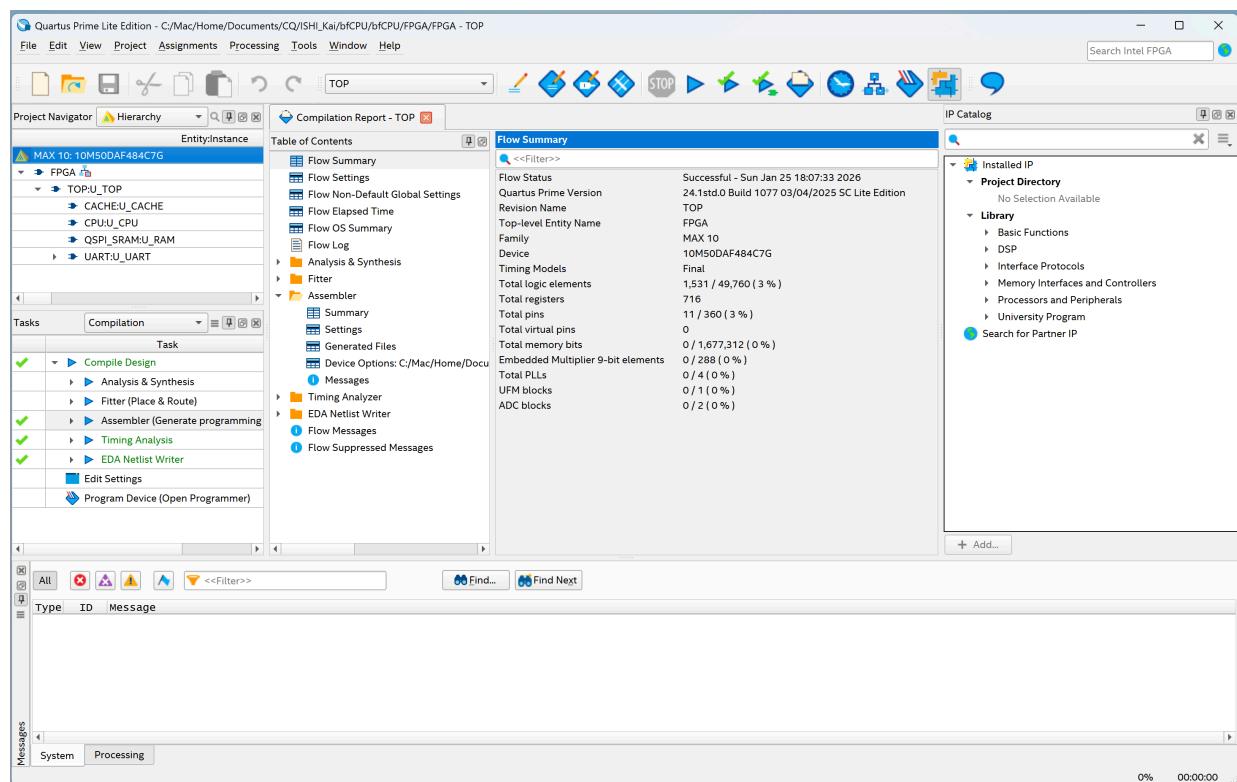


図 1.37: FPGA 開発ツール Quartus Prime Lite Edition

1.27.4 FPGA 開発ツール Quartus Prime のプロジェクト設定

bfCPU の FPGA プロジェクト・ファイルはリポジトリ bfCPU/FPGA/FPGA.qpf です。この設定内容を以下説明します。

- **デバイス選択:** メニュー **Assignments → Device** を選択して表示される画面を図 1.38 に示します。この中で **10M50DAF484C7G** を選択します。
- **プロジェクト設定:** メニュー **Assignments → Settings** を選択して表示される画面を図 1.39 に示します。
 - **最上位階層:** **Category/General** を選択し、リポジトリ bfCPU/RTL/TOP/fpga.sv に記述された

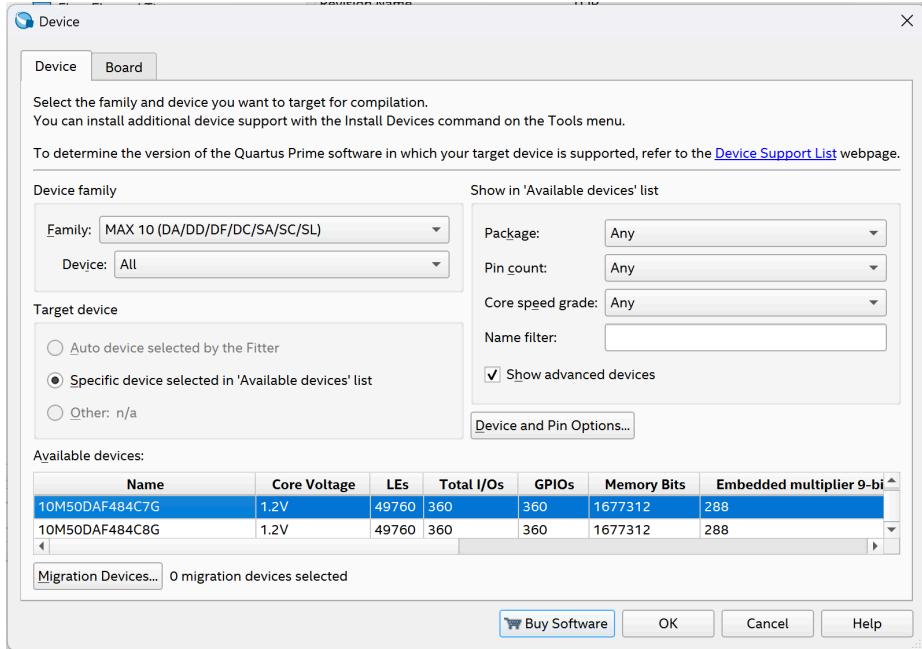


図 1.38: Quartus Prime のデバイス選択

モジュール名 **FPGA** を最上位階層名として設定します。

- **ファイル選択:** Category/Files を選択し、FPGA の合成に必要なファイルを全て指定します。RTL 記述に加えてファイル **FPGA.sdc** があります。これは後述するタイミング制約ファイルです。
- **コンパイラ設定:** Category/Compiler Settings を選択し、最適化モードを **Balanced** にします。
- **入力言語設定:** Category/Verilog HDL Input を選択し、Verilog バージョンを **System Verilog** にします。
- **FPGA の端子設定:** FPGA の端子への機能アサインを表 1.3 のように設定します。メニュー Assignments Pin Planner を選択し、図 1.40 に示すように設定します。一旦、この設定なしに FPGA を仮合成すると信号名と入出力設定がこの画面に表示されるので、FPGA の端子、内蔵 PUP(プルアップ抵抗)の有無を指定します。内蔵 PUP は入力信号と入出力信号に入れます。QSPI SRAM 用の信号はリセット中に Hi-Z にするので出力方向だけの QSPI_CS_N と QSPI_SCK もドライバ(駆動するデバイス)が複数あるため入出力信号として RTL 上で定義しています。各端子のドライバビリティ (Current Strength) はできるだけ弱く (4mA) にしてノイズ発生を抑えています。ただし信号 LED だけは直接赤色 LED を駆動するので 12mA に設定しました。

1.27.5 タイミング制約

論理回路設計にはタイミングが重要で、動作周波数や入出力信号に許容される伝搬時間を指定する必要があります。FPGA はタイミング制約を掛けなくて合成できますが、出来上がった合成結果の動作周波数や入出力信号がどうなるかわかりません。原則として、希望するタイミング制約を指定することを推奨します。

タイミング制約は、業界標準の SDC(Synopsys Design Constraints) ファイルに記述します。今回の bfCPU の合成に使った記述例をリスト 1.20 に示します。以下、設定内容の概要を説明します。

- **create_clock:** クロック CLK の周期を設定します。ここでは 100ns(周波数 10MHz) を指定しました。
- **set_input_delay:** 入力信号を受ける初段の D-F/F のクロックと同じクロックを使う仮想的な D-F/F を外部に置き、その外部の仮想 D-F/F の Q 出力から FPGA の入力端子までの信号遅延を指定します。

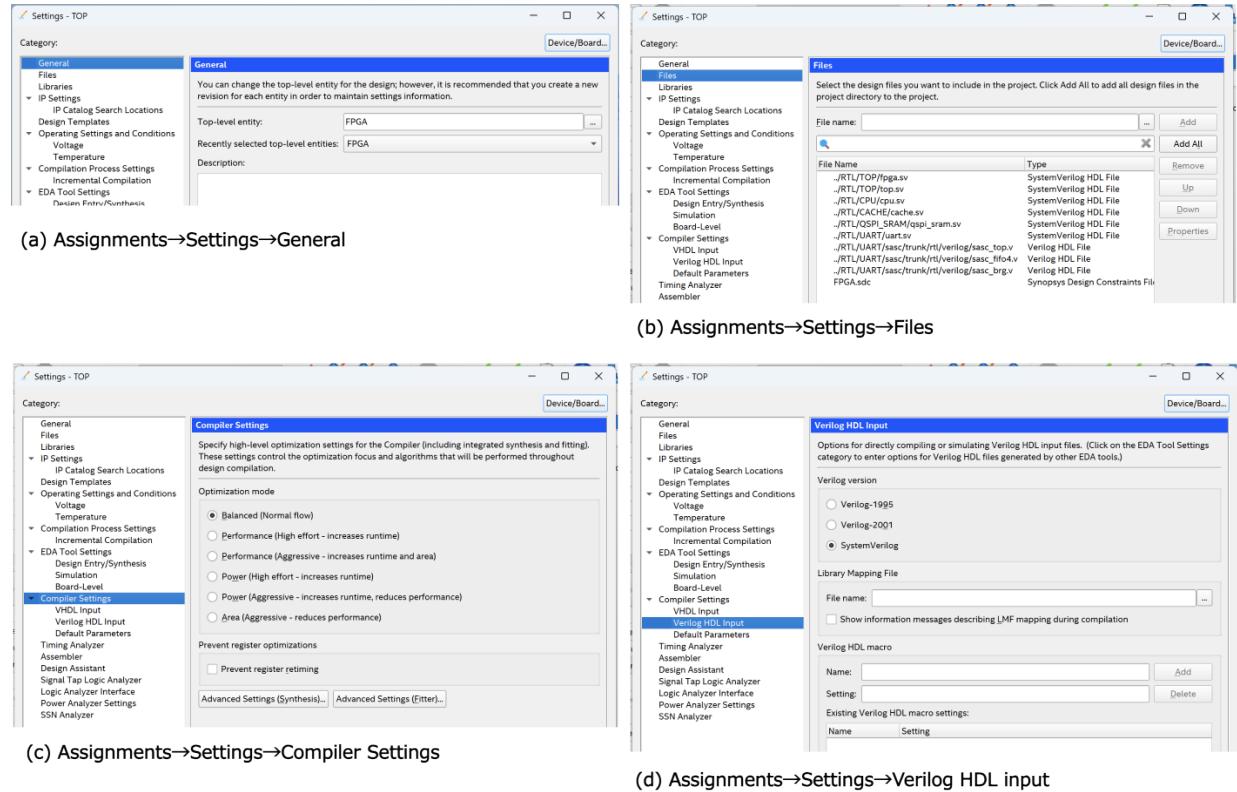


図 1.39: Quartus Prime のプロジェクト設定

表 1.3: FPGA の外部端子アサイン

端子名	入出力	端子	内蔵 PUP	機能
CLK	input	Y5	ON	クロック
RES_N	input	Y6	ON	リセット(負論理)
QSPI_CS_N	in/out	AA15	ON	QSPI SRAM チップ・セレクト(負論理) Hi-Z 制御あり
QSPI_SCK	in/out	V5	ON	QSPI SRAM シリアル・クロック Hi-Z 制御あり
QSPI_SIO[3]	in/out	W10	ON	QSPI SRAM データ入出力 3 Hi-Z 制御あり
QSPI_SIO[2]	in/out	W9	ON	QSPI SRAM データ入出力 2 Hi-Z 制御あり
QSPI_SIO[1]	in/out	W8	ON	QSPI SRAM データ入出力 1 Hi-Z 制御あり (SPI モードのシリアル入力)
QSPI_SIO[0]	in/out	W7	ON	QSPI SRAM データ入出力 0 Hi-Z 制御あり (SPI モードのシリアル出力)
UART_RXD	input	Y3	ON	UART 受信データ
UART_TXD	output	Y4	-	UART 送信データ
LED	output	A8	-	CPU Running インジケーター

ここでは、クロック 1 周期の半分 50ns を指定しました。

- **set_output_delay:** 出力信号を出す最終段の D-F/F のクロックと同じクロックを使う仮想的な D-F/F を外部に置き、FPGA の出力端子からその外部の仮想 D-F/F の D 入力までの信号遅延を指定します。
ここでは、クロック 1 周期の半分 50ns を指定しました。
- **set_false_path:** タイミング制約をかけない信号経路を指定します。SDC ファイルには、似たものとして **set_multicycle_path** もあり、こちらは複数サイクルで転送すればいい低速信号の経路に指定します。

リスト 1.20: タイミング制約ファイル FPGA.sdc

```
create_clock -name CLK -period 100 [get_ports {CLK}]
set_input_delay -clock { CLK } 50 [get_ports {UART_RXD}]
```

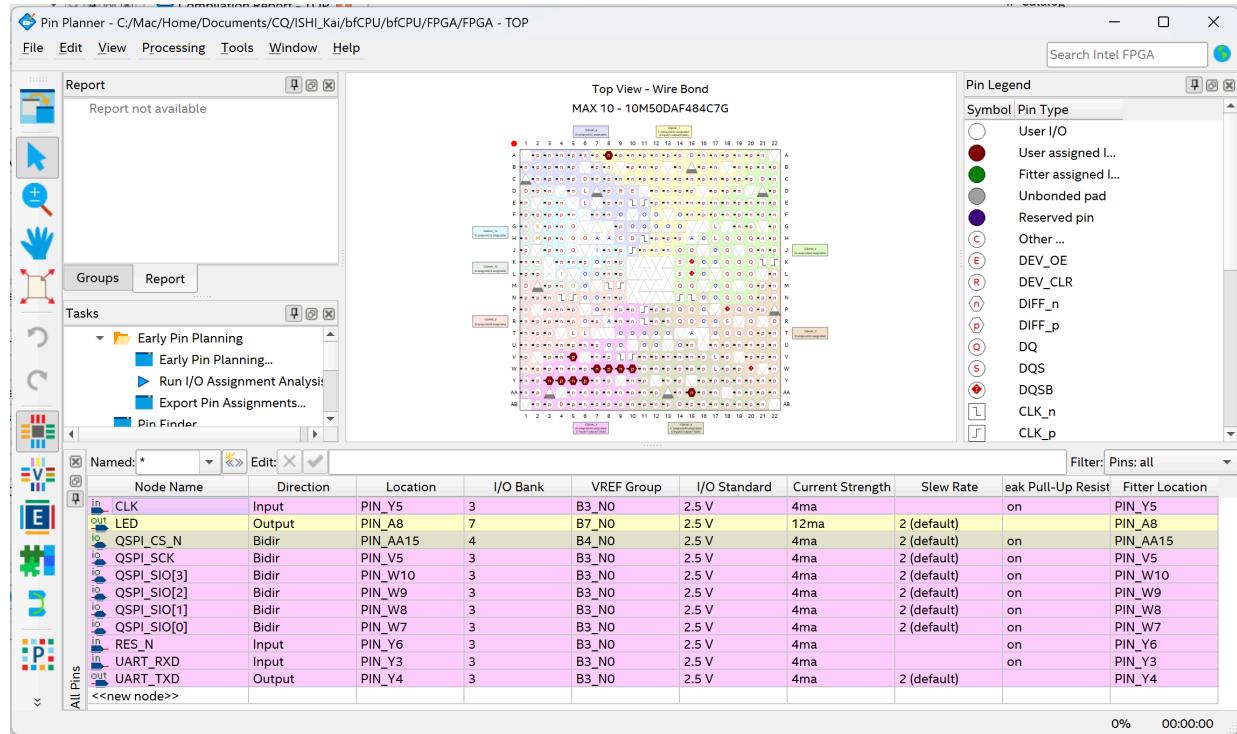


図 1.40: Quartus Prime の端子設定

```

set_input_delay -clock { CLK } -clock_fall 50 [get_ports {QSPI_SI0[0] QSPI_SI0[1] QSPI_SI0[2]
] QSPI_SI0[3]}]
set_output_delay -clock { CLK } 50 [get_ports {QSPI_SI0[0] QSPI_SI0[1] QSPI_SI0[2] QSPI_SI0[3]
} QSPI_CS_N QSPI_SCK UART_RXD}]
set_false_path -from [get_ports {RES_N}]
set_false_path -to [get_ports {QSPI_SCK}]
set_false_path -to [get_ports {LED}]

```

1.27.6 FPGA を合成してコンフィグ

ここまで各種設定したところで FPGA を合成しましょう。メニュー Processing Start Compilation を選択するか、画面上部の青い三角形の右矢印を押します。合成がエラーなく完了した時の画面が図 1.37 です。

合成が完了したら FPGA をコンフィグします。FPGA ボードと PC を USB ケーブルで接続してください。図 1.41 のメイン画面の左下 Tasks の中の **Program Device (Open Programmer)** をダブル・クリックすると **Programmer** が起動します。ボタン **Hardware Setup...** を押して **USB-Blaster** を選択してください。コンフィグ用ファイル (FLASH メモリ用)output_files/TOP.pof が表示されていることを確認して、CFM0 の Program/Configure のチェック・ボックスを ON にしてボタン **Start** を押してください。コンフィグレーション (FPGA への書き込み) が成功すれば右上の Progress バーに 100% (Successful) と表示されます。

1.27.7 FPGA ボード、ラズパイ、QSPI SRAM の接続

FPGA ボードの電源を一旦切って、FPGA ボード、ラズパイ、QSPI SRAM のそれぞれを接続します。図 1.42 に示すように結線してください。ブレッド・ボードやジャンパを使って構築した例を図 1.43 に示します。ラズパイや FPGA ボードは 3.3V 電源や 5V 電源を出力していますので、それらを間違って繋げてしまうとデバイスやボードを壊す恐れがあるので注意してください。

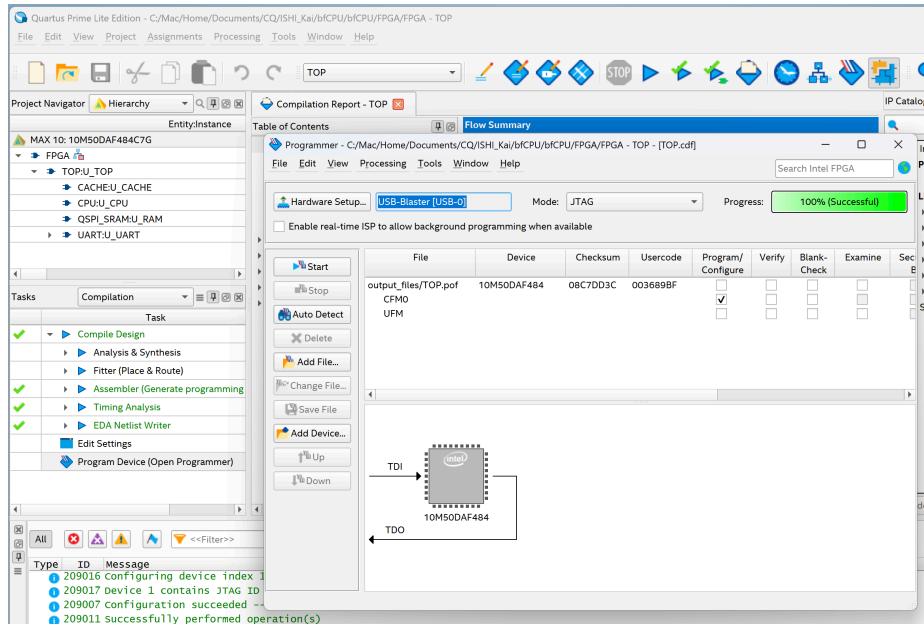


図 1.41: Programmer で FPGA をコンフィグ

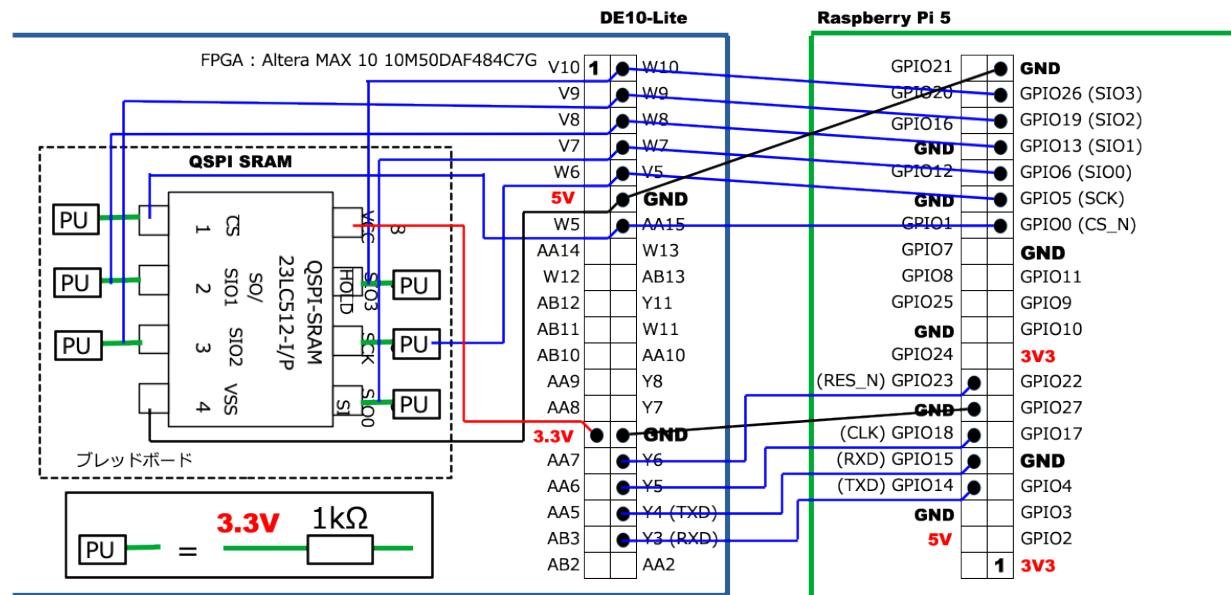


図 1.42: FPGA システムの接続

1.27.8 ラズパイ上に bfCPU 開発環境を構築

ラズパイ上に bfCPU 開発環境を構築しましょう。以下、RASPBERRY PI OS (64bit Linux) が動作している前提で説明します。

ラズパイの設定

bfCPU のシステム・クロックはラズパイの PWM 機能から出力します。また、データ入出力のために UART 機能を使います。これらを有効化するために sudo で管理者権限としてエディタを起動し、/boot/firmware/config.txt の最後に下記を追加してください。その後、ラズパイを再起動してください。

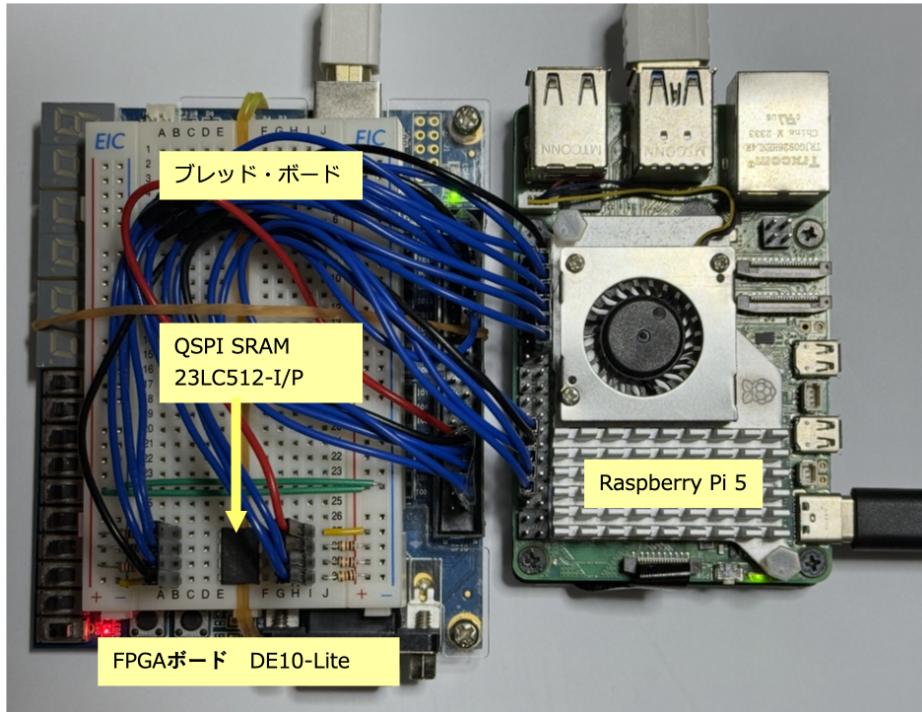


図 1.43: FPGA システムの外観

```
[all]
dtOverlay=pwm-2chan      追加
dtParam=uart0=on          追加
```

この後のビルドに必要なライブラリや関連ツールをインストールしてください。

```
$ sudo apt install -y git autoconf autoconf-archive automake libtool pkg-config python3-dev libgpiod-dev gpiod minicom cutecom
```

bfCPU プログラム開発ツール **bfTool** のビルド

前章で説明した bfCPU プログラム開発ツール **bfTool** はラズパイの上でも動作します。bfCPU の設計関連データが <https://github.com/munetomo-maruyama/bfCPU.git> に格納してあるので、

```
$ git clone https://github.com/munetomo-maruyama/bfCPU.git
```

で bfCPU のリポジトリをダウンロードしてください。ディレクトリ bfCPU/bfTool に移動し、gcc、flex、bison を導入した状態で、下記コマンドで bfTool がビルドできます。

```
$ cd bfCPU/bfTool
$ make
```

ビルドして出来上がった実行ファイル bfTool を環境変数 \$PATH に設定してどこからでもコマンドとして実行できるようにしておいてください。サンプル・プログラムが bfCPU/bfTool/samples に格納してあるので、前章で解説したように動作確認してみてください。

bfCPU プログラム書き込みツール **bfRun** のビルド

SPI SRAM のプログラムの書き込みと bfCPU の起動は Raspberry Pi 5(ラズパイ) が担います。そのためのツールが **bfRun** です。bfRun のビルドは下記コマンドで行います。

```
$ cd bfCPU/RaspI5/bfRun
$ make
```

ビルドして出来上がった実行ファイル bfRun を環境変数\$PATH に設定してどこからでもコマンドとして実行できるようにしておいてください。

bfRun ツールは下記の仕事を行います。

- bfCPU システムにシステム・クロック (約 10MHz) を供給
- bfCPU システムをリセットして、QSPI SRAM にプログラムと UART のボーレート・レジスタへの設定値 (DIV0 と DIV1) をライト
- QSPI SRAM にライトしたプログラムとデータの内容をベリファイ
- bfCPU システムのリセットを解除してプログラム実行スタート！

このシステムが使うラズパイの GPIO 端子を表 1.4 に示します。bfRun ツールが制御するのは UART 通信用の TXD、RXD 以外の端子全てです。

表 1.4: ラズパイの端子アサイン

端子機能	ピン・ヘッダ	ラズパイ GPIO 名	ラズパイ GPIO 機能
CLK	12pin	GPIO18	PWM_CLK (PWM0-CH2)
RES_N	16pin	GPIO23	オープン・ドレイン出力ポート、プルアップ抵抗 ON
QSPI_CS_N	27pin	GPIO0	オープン・ドレイン出力ポート、プルアップ抵抗 ON
QSPI_SCK	29pin	GPIO5	オープン・ドレイン入出力ポート、プルアップ抵抗 ON
QSPI_SIO0	31pin	GPIO6 (QSPI_SI)	オープン・ドレイン入出力ポート、プルアップ抵抗 ON
QSPI_SIO1	33pin	GPIO13	オープン・ドレイン入出力ポート、プルアップ抵抗 ON
QSPI_SIO2	35pin	GPIO19	オープン・ドレイン入出力ポート、プルアップ抵抗 ON
QSPI_SIO3	37pin	GPIO26	オープン・ドレイン入出力ポート、プルアップ抵抗 ON
TXD	8pin	GPIO14	UART TXD (bfCPU の RXD に接続)
RXD	10pin	GPIO15	UART RXD (bfCPU の TXD に接続)

ラズパイが bfCPU システムに供給するクロックは PWM 機能を使って生成しますが、10MHz 出力を設定したつもりでも 8.3MHz~8.4MHz 程度しか出ませんでした。このため、bfRun にはおよそのクロック周波数をパラメータとして与えて、QSPI SRAM の最終アドレスに書き込む bfCPU 内の UART ボーレートの設定値を決めています。

bfRun の起動コマンドは下記です。オプション-c でラズパイが出力する bfCPU 動作用クロックの周波数 [Hz] を指定し、最後に QSPI SRAM に書き込みたいプログラムのバイナリ (Intel Hex フォーマット; bfTool が生成するもの) を指定します。オプション-c なしの場合は、bfCPU に入力されるクロックが 10MHz だとして UART ボーレート設定値を決めます。ラズパイ個体によって出力される周波数が変わるので、念の為、ラズパイが出力するクロック周波数をオシロや Analog Discovery などで計測して-c オプションのパラメータを決めた方がいいでしょう。

```
$ cd bfCPU/RaspI5/bfRun
$ bfRun -c 8300000 xxx.hex
```

bfRun を実行すると下記メッセージが出力され、FPGA内のbfCPUシステムが起動します。終了するには Ctrl-C を押してください。

```
$ bfRun -c 8300000 helloworld.hex
Read Hex File...Done.
Decode Hex Format...Done.
```

```

Start System Clock...Done.
Set Serial SRAM in QSPI Mode...Done.
Configure GPIO of Raspberry Pi...Done.
Write Hex Data to SRAM...Done.
Read Hex Data from SRAM...Done.
Verify Hex Data in SRAM...OK.
Set Baud Rate Data in SRAM (Freq=8300000 Baud=115200 div0=7 div1=2)...OK.
Start the bfCPU System (Ctrl-C to Quit).

```

1.27.9 bfCPU システムを実装した FPGA の動作確認

bfTool のサンプル・プログラムを動かしましょう。

Hello World! プログラムの実行

Hello World! プログラムです。コンソール・ターミナルを立ち上げて下記コマンドを実行してください。

```
$ cd bfCPU/bfTool/samples
$ bfRun -c 8300000 helloworld.hex
```

別のコンソール・ターミナルを立ち上げて、シリアル通信アプリ minicom を起動します。デバイスは /dev/ttyAMA0 を指定します。

```
$ minicom -D /dev/ttyAMA0
```

デフォルトでは、bfCPU から送信されてくる LF コード (0x0A) はライン・フィードしかないので、キャリッジ・リターンもするように設定します。Ctrl-A を押してから Z を押すと下記のメニューが表示されます。

```

+-----+
|           Minicom Command Summary           |
|                                           |
| Commands can be called by CTRL-A <key>   |
|                                           |
|           Main Functions          Other Functions   |
|-----|
| Dialing directory..D  run script (Go)....G | Clear Screen.....C |
| Send files.....S  Receive files.....R | cOnfigure Minicom..O |
| comm Parameters....P Add linefeed.....A | Suspend minicom....J |
| Capture on/off.....L Hangup.....H | eXit and reset....X |
| send break.....F initialize Modem...M | Quit with no reset.Q |
| Terminal settings..T run Kermit.....K | Cursor key mode....I |
| lineWrap on/off....W local Echo on/off..E | Help screen.....Z |
| Paste file.....Y Timestamp toggle...N | scroll Back.....B |
| Add Carriage Ret...U                         |
|                                           |
|           Select function or press Enter for none. |
+-----+

```

Add Carriage Ret を選ぶため、U を押してください。これで、以下のように表示されるようになります。このプログラムは最後に “reset 命令” を実行するので繰り返し Hello World! が表示されます。

```
Hello World!
Hello World!
Hello World!
```

```
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!  
Hello World!
```

minicom を終了するには、Ctrl-A を押してから Z を押し、その後 Q を押してください。

TicTacToe ゲームの実行

コンピュータ対戦型 3 目並べゲームです。コンソール・ターミナルを立ち上げて下記コマンドを実行してください。

```
$ cd bfCPU/bfTool/samples  
$ bfRun -c 8300000 tictactoe.hex
```

minicom が起動中ならそのまま使えます。minicom が未起動なら Hello World! プログラムの時と同様に立ち上げて、Add Carriage Ret を設定しておいてください。下記のように表示されます。プロンプト > が出ている時に、石を打ちたい場所を数字 1 衍を入力します。その後 LF を入力するため Enter キーや Return キーではなく、Ctrl-J を押してください。

```
X23  
456  
789  
>5  
X23  
406  
78X  
>7  
X2X  
406  
08X  
>
```

1.27.10 Life ゲームの実行

Life ゲームは以下のコマンドで起動してください。

```
$ cd bfCPU/bfTool/samples  
$ bfRun -c 8300000 life.hex
```

これも minicom で入出力します。Life を生（イキ）にするマス目を縦・横の英文字座標で指定します。入力の最後はこれも CTRL-J です。Life を 1 世代進めるには単に CTRL-J を押します。

```
> abcdefghij  
a-----  
b-----  
c-----  
d-----  
e-----  
f-----  
g-----  
h-----  
i-----  
j-----  
>ce
```

```
abcdefghijkl
a-----
b-----
c-----*
d-----
e-----
f-----
g-----
h-----
i-----
j-----
>
```

1.27.11 乗算プログラムの実行

乗算プログラムは以下のコマンドで起動してください。

```
$ cd bfCPU/bfTool/samples
$ bfRun -c 8300000 multiplication.hex
```

このプログラムのデータ入出力は文字ではなくバイナリ数値です。シリアル通信でバイナリを 1 バイトずつ送受信できるターミナル・アプリとして cutecom があります。下記コマンドで起動します。

```
$ cutecom
```

起動すると図 1.44 が表示されます。Device に /dev/ttyAMA0 を指定し、Open ボタンを押してください。Input 入力フィールドの右側のプルダウン・リストは Hex にして、最下部の Hex Output のチェックを入れてください。その状態で Input 入力フィールドに 1 バイトの被乗数を入力し、連続して 1 バイトの乗数を入力します。ここで Enter キーや Return キーが使えます。すると乗算結果が 16 進数で表示されます。これを繰り返してください。

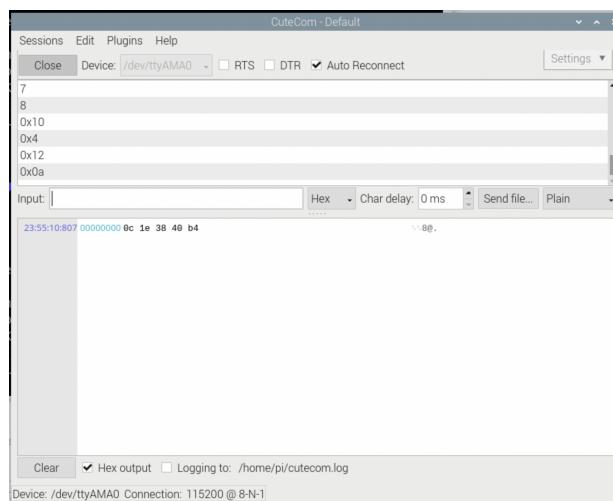


図 1.44: ラズパイの cutecom の画面

1.28 色々遊べる bfCPU システムが手に入った！

これで色々遊べる bfCPU が手に入りました。自分なりに bfCPU プログラムを書いて、bfTool でアセンブ

ル&シミュレーションし、実際にFPGA上で動作させてみましょう。完全チューリング・マシンの難解プログラムの深淵さに魅了されることでしょう！

1.29 参考文献

[1] 竹内 2024

竹内薰,『不完全性定理とはなにか 完全版 ゲーデルとチューリング 天才は何を証明したのか』,講談社(ブルーバックス),2024年

[2] 笠井 2018

笠井琢美,阿部彩芽,チューリングの考えるキカイ～人工知能の父に学ぶ,コンピュータ・サイエンスの基礎,技術評論社,2018

[3] 牧野 2022

牧野武文,チューリングマシンとは?コンピューター・ソフトウェアの生みの親アラン・チューリング,
<https://staff.persol-xtech.co.jp/corporate/security/column/>,コラム,2022.08.30

[4] Wiki Turing

Wikipedia,チューリングマシン,<https://ja.wikipedia.org/wiki/チューリングマシン>

[5] Wiki Newmann

Wikipedia,ノイマン型,<https://ja.wikipedia.org/wiki/ノイマン型>

[6] Wiki bfCPU

Wikipedia,Brainf&ck,<https://en.wikipedia.org/wiki/Brainfuck>

[7] Temkin Esolangs

Daniel Temkin,Glitch && Human/Computer Interaction,2014,<https://nooart.org/post/73353953758/temkin-glitchhumancomputerinteraction>

第 2 章

bfCPU を Tiny Tapeout により実シリコン化しよう

本章では、前章で設計した bfCPU システムを Tiny Tapeout というサービスを使って実際にシリコン化する試みを紹介します。

圓山 宗智 (munetomo@ishi-kai.org)

2.1 Tiny Tapeout とは？

Tiny Tapeout(タイニー・テープアウト)は、半導体設計の民主化を掲げ、専門知識や巨額の資金を持たない個人や学生、教育機関で、自分専用の ASIC(特定用途向け集積回路)を設計・製造できるようにした画期的な教育プロジェクトです。

現在、このプロジェクトは世界中の開発者、教育者、そして将来の半導体エンジニアにとって、実践レベルのスキルを習得するための重要なプラットフォームへと成長しています。以下に、Tiny Tapeout の仕組み、特徴、そして最新状況を解説します。

2.1.1 Tiny Tapeout の基本概念：半導体設計の民主化

従来の ASIC 設計は、数千万円から数億円に及ぶライセンス費用、物理的な製造(テープアウト)にかかる高額なコスト、そして数ヶ月から数年に及ぶ長い開発期間が必要という、極めてハードルの高い領域でした。Tiny Tapeout は、“共有チップ方式”と“オープンソースツール”を組み合わせることで、この壁を打ち破りました。

- 低コスト: 1 タイルあたり約 150 ドル～300 ドル(2026 年時点の目安)という、これまでの常識を覆す低価格で実物のチップを手に入れることができます。
- 短期間: クラウド上で動作する自動設計フローにより、数時間から数週間で設計を完了させることができます。
- アクセシビリティ: ソフトウェアのインストールは不要で、Web ブラウザ上の設計ツール(Wokwi)や GitHub を使用して設計を完了させ提出することができます。

2.1.2 独自の製造モデル：マルチ・プロジェクト・ウェハ (MPW) とタイル構造

Tiny Tapeout が低コストを実現している最大の理由は、マルチ・プロジェクト・ウェハ (MPW) という仕組みを活用したことにあります。ウェハ内に複数のプロジェクトが同居する乗合バスのような形態なので、このような試作方法をシャトルと呼ぶこともあります。

- タイル方式: 1枚の大きなシリコンチップを、多数の小さなタイル(区画)に分割します。各ユーザーは1つ以上のタイルを購入し、そこに自分の回路を配置します。多数のユーザー設計が1枚のチップに同居するため、各回路への入出力信号はチップ内部の共通コントローラを介して切り替える仕組みになっています。
- 全員に全員の設計が届く: 参加者は、自分の設計だけでなく、その回のシャトル(乗合バス)に参加した全員の設計が含まれたチップを受け取ります。ボード上でプロジェクトを切り替えることで、他人のプロジェクトを動作させて学ぶこともできます。

2.1.3 進化する設計環境と技術インフラ

Tiny Tapeout は複数の製造プロセスと設計手法に対応しています。

- アナログ・ミックスドシグナルへの対応: デジタル回路だけでなく、アナログ設計やデジタル・アナログ混在回路の製造も正式にサポートされています。デジタル回路は、NAND ゲート、NOR ゲート、フリップ・フロップなどのゲート・レベルでの設計と、ハードウェア記述言語(Verilog HDL、System Verilog など)による RTL レベルの設計のいずれも可能です。
- LibreLane の採用: 設計フローとしては、強力なオープンソース・インフラである LibreLane が採用されています。各設計ステップのツールがしっかりと連携しており、デジタル回路の場合、ワン・プッシュで RTL 記述から GDS-II(製造用レイアウト・データ)の自動生成が可能です。
- 多様な製造ファウンドリ: 下記メーカーの実績があり、今後も計画されています。
 - SkyWater(米) 130nm
 - IHP(独) 130nm
 - GlobalFoundries(米) 180nm
- Python によるテストベンチ: 機能検証プラットフォームとして Python ベースのテストベンチ(cocotb)をサポートしており、強力なライブラリ群を活用して機能検証を効率化できます。

2.1.4 教育とコミュニティの広がり

Tiny Tapeout は単なる製造サービスではなく、巨大な教育エコシステムです。

- デモボードとデバッグ: 届いたチップは、入出力端子やディスプレイ接続端子を備えた専用のデモボードに搭載されています。ボード上のマイコン(RP2040)や PC から簡単にチップを制御し、動作確認ができます。
- 活発なコミュニティ: Discord などの SNS を通じ、世界中のプロのエンジニアから初心者までがトラブルシューティングや設計のコツを共有しており、学習のリソースが極めて豊富です。

2.1.5 Tiny Tapeout がもたらすインパクト

このプロジェクトは、単に“面白いおもちゃ”を作るためのものではありません。

- スキル・ギャップの解消: 世界的な半導体人材不足に対し、学生が在学中に“自分で設計した実物チップを手にする”という、かつては博士課程レベルだった経験を、学部生や高校生レベルに提供しています。
- オープンソース・シリコンの推進: 半導体メーカー独自のクローズドな技術ではなく、オープンな PDK(Process Design Kit)や EDA ツールを活用することで、半導体産業全体の透明性とイノベーションを促進しています。
- プロトタイピングの迅速化: 小規模な研究プロジェクトやスタートアップが、数億円の投資をする前に、コンセプト・レベルの回路を極めて低コストで検証できます。

今後のシャトル(TTSKYxxx や TTIHPxxx)に向けて、世界中でワークショップが開催されています。もし、

自分のアイデアをシリコンに刻みたいと考えているなら、Tiny Tapeout は最も身近で、最も強力なプラットフォームとなっています。

2.1.6 Tiny Tapeout の設計フロー “LibreLane”

Tiny Tapeout のロジック・チップ設計は、図 2.1 に示す LibreLane を使用します。LibreLane は、以下の一連の行程をワンプッシュで一気に実行します。FPGA の設計のような簡単さです。

- System Verilog による RTL 記述の論理合成
- 仮負荷タイミング検証
- 自動配置配線
- 実負荷タイミング検証
- レイアウト・パターンの GDS-II データ生成
- DRC / LVS 検証

なお、Tiny Tapeout が使う設計フローは、かつては OpenLane でしたが、今は LibreLane に名称が変更されています。

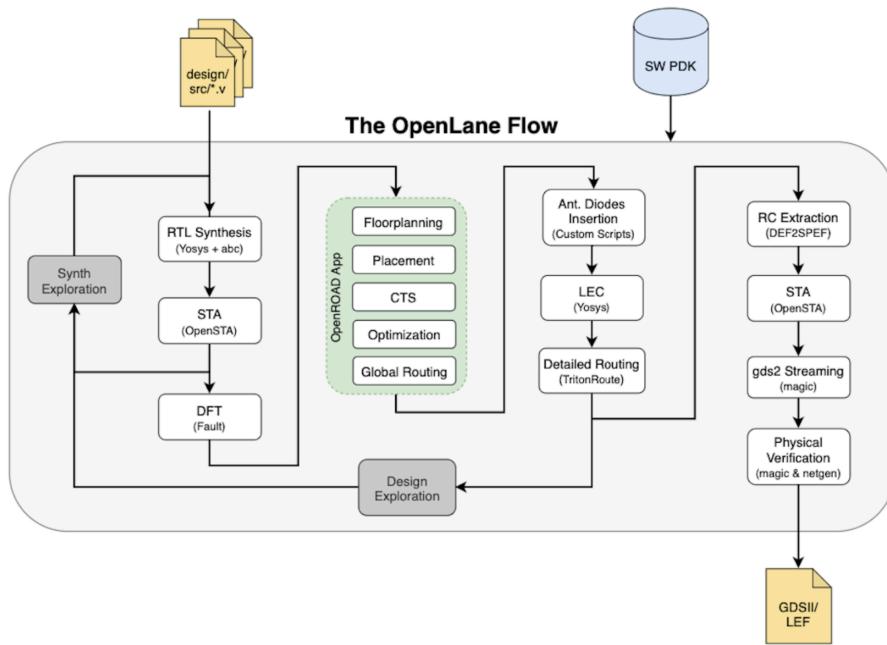


図 2.1: Tiny Tapeout の設計フロー LibreLane (<https://www.zerotoasiccourse.com/terminology/librelane/>)

Tiny Tapeout によるロジック・チップ設計は、上記の RTL 記述から合成する方法以外に、wokwi というゲート・レベルで設計するツールで行うこともできます。さらに、アナログ・チップを設計することも可能です。本稿では、LibreLane で RTL 記述 (System Verilog) からレイアウトまで一気に合成する手法を説明します。

2.1.7 Tiny Tapeout のチップ・レイアウト

Tiny Tapeout のチップ・レイアウトの一例を図 2.2 に示します。複数プロジェクトが同居する MPW (Multi Project Wafer) です。区画の最小単位は、Skaywater 130nm プロセスの場合、167um × 108um で、2000 ゲートくらいの規模です。インテル者の世界最初のプロセサ 4004 程度の論理規模なら余裕で入る程度です。図 2.2 も Tiny Tapeout で 4004 を設計した事例です。

タイルは 1 区画でも複数区画でも使えますが、一つのプロジェクトが占有できる区画は、1x1, 1x2, 2x2, 3x2, 4x2, 6x2, 8x2 のいずれかになります。もちろん区画数が増えれば費用も増えます。実際の費用は、Tiny

Tapeout のサイト (FAQ の下の What is the price?) に費用の計算ページがあるので参照してください。

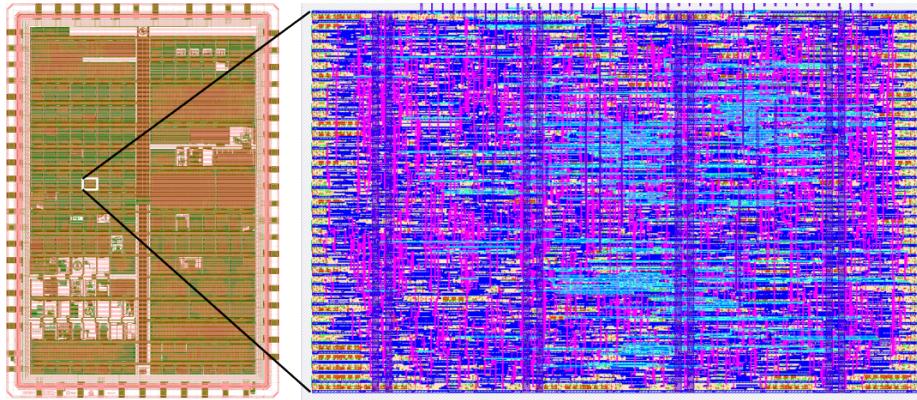


図 2.2: Tiny Tapeout のチップ・レイアウト

2.1.8 Tiny Tapeout のユーザ端子

使える外部端子の数は、表 2.1 に示すように、ユーザ機能としては合計 24 本です。これは、区画数をいくら確保しても同じです。複数プロジェクトが同居する関係でこの制約になっています。

表 2.1: Tiny Tapeout の外部端子

端子名	入出力	機能
rst_n	input	ユーザ・リセット
clk	input	ユーザ・クロック
ui[7:0]	input	ユーザ機能入力
uo[7:0]	output	ユーザ機能出力
uio[7:0]	in/out	ユーザ機能入出力
ena	input	プロジェクト選択イネーブル
sel_RST_n	input	プロジェクト選択 MUX リセット
sel_inc	input	プロジェクト選択 MUX インクリメント

なお、入手するチップには全てのプロジェクトが含まれており、自分のプロジェクトを有効化して外部端子に引き出すには、プロジェクト選択信号を操作します。図 2.3 に示すように、sel_RST_n と ena を Low にして電源投入し、sel_RST_n を High にしてから、プロジェクト提出後にアサインされるチップ内のアドレス番号の数だけのパルスを sel_inc から入力します。その後、ena を High にします。この図はアドレス番号が 12 の場合を示しています。

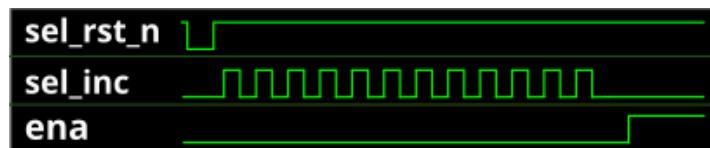


図 2.3: プロジェクトを選択するための入力信号

2.1.9 Tiny Tapeout の評価ボード

Tiny Tapeout は、チップが完成すると、パッケージに収められたチップと図 2.4 に示すような評価ボードが送られてきます。すぐにデバッグを開始できます。このボードは、Tiny Tapeout チップをマイコン RP2040 か

ら制御するなど、いろいろ活用できるよう工夫されています。詳細は、Tiny Tapeout の Github サイトに情報があります。

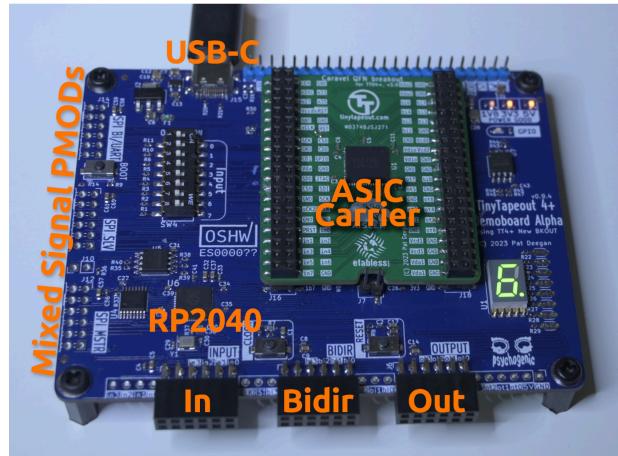


図 2.4: Tiny Tapeout の評価用ボード

2.2 Tiny Tapeout のクラウド設計とローカル設計

Tiny Tapeout を LibreLane フローで設計する時、図 2.5 に示すように大きく分けて 2 つの方法があります。

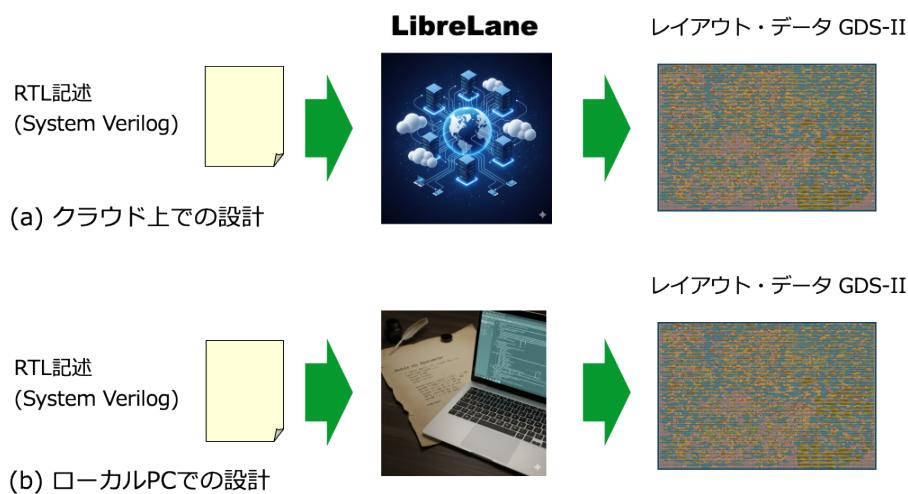


図 2.5: クラウド設計とローカル設計

- **(a) クラウド上での設計:** Tiny Tapeout の LibreLane フローの特長の一つは、設計ツールを自分の PC にインストールする必要がない、という点です。RTL 記述などの設計データをクラウドに上げれば、そこで LibreLane フローを実行して、レイアウト・データ GDS-II を生成してくれます。設計したチップを試作するため、レイアウト・データを提出する前には、必ずクラウド上での設計を流して正常に完了させる必要があります。
- **(b) ローカル PC での設計:** クラウド上での設計はツールのインストールが不要という特長はありますが、実行時間が長く、何度も設計をやり直す時は不便です。そのため、ローカル PC での設計環境を立ち上げると便利です。Docker を使うので設計ツールを個別に自分の PC にインストールする必要はありません。

ただし、ローカル PC の設計は、一部の検証（レイアウト・データのデザイン・ルール・チェック: DRC）が省略されているので、最終的にはクラウド上で設計フローを流してエラーがない状態にする必要があります。

本稿では、まず、ローカル PC での設計環境を立ち上げ、bfCPU をローカルの LibreLane フローを通してレイアウト・データを生成できることを確認します。その後、クラウド上での設計方法と、最終的に Tiny Tapeout に設計データを提出する方法を説明します。

2.3

LibreLane 設計フローをローカル PC 上で動かす

LibreLane 設計フロー環境をローカル PC 上に立ち上げましょう。ここで解説する方法は、以下の PC 環境で動作確認しています。

- Windows 11 AMD64(x64) 版: WSL2 上に Ubuntu がインストールされていること
- ネイティブ Ubuntu AMD64(x64) 版または ARM64 版

2.3.1 [準備] Windows 11 WSL2 Ubuntu の場合

WSL2 用 Ubuntu の確認とインストール

Windows 環境で Powershell を起動してください。まず、WSL Ubuntu のインストール状況を確認しましょう。

```
> wsl --list --verbose
```

WSL2 用 Ubuntu をまだインストールしていないか、または新たに追加インストールする場合は、下記コマンドでインストールしてください。ここでは、LibreLane 用の WSL 環境を新規に追加することにして、-name 以下には他とは異なるディストリビューション名を指定しておきます。終わったら、インストールした Ubuntu を起動します。

```
> wsl --install Ubuntu-24.04 --name Ubuntu-24.04-ishikai-tt
> wsl -d Ubuntu-24.04-ishikai-tt
```

インストールしたディストリビューションを削除する場合は、下記コマンドで Ubuntu をターミネート（終了）してから削除してください。このディストリビューションのファイル・システムも削除されます。

```
> wsl -t Ubuntu-24.04-ishikai-tt
> wsl unregister Ubuntu-24.04-ishikai-tt
```

Windows 版 Docker 環境のインストール

下記手順で Windows 版 Docker をインストールしてください。すでにインストール済みであればこの項はスキップして Ok です。

1. **Docker desktop for Windows** のインストーラ Docker Desktop Installer.exe を <https://www.docker.com/get-started/> からダウンロードして起動してください。
2. **Configuration 設定** の画面で、図 2.6(a) に示すようにチェックを入れて OK を押します。
3. **Docker Subscription Service Agreement** が表示されたら内容確認して Accept ボタンを押します。
4. **Welcome to Docker** が表示されたら Personal を選択してください。アカウント作成するかどうか聞かれますが、Skip しても OK です。

5. Docker Desktop X.XX.X Installation Succeeded が表示されたら終了して PC を再起動してください。
6. アプリ Docker Desktop を起動してください。図 2.6(b) に示す画面上部の歯車型のボタンを押してください。Resources 画面の中で WSL integration を選択して、LibreLane ツールを立ち上げる WSL2 環境を選択してください。その後、PC を再起動します。

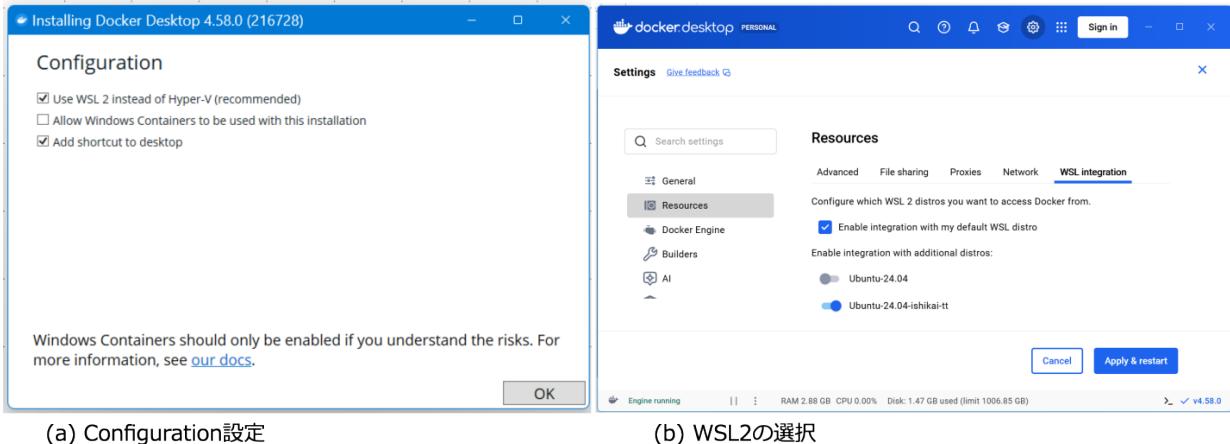


図 2.6: Docker desktop for Windows のインストール

次に WSL2 の Ubuntu の中で下記の設定をしてください。

(1) root 権限 (sudo) で /etc/wsl.conf を編集して、ファイルの最後に下記を追加してください。

```
[user]
default=ishikai
```

(2) root 権限 (sudo) で /etc/group を編集して docker グループに自分の ID を追加してください。

```
docker:x:1001:root:ishikai
```

(3) Ubuntu を再起動します。Powershell の中で下記コマンドを実行します。

```
> wsl shutdown
> wsl -d Ubuntu-24.04-ishikai-tt
```

2.3.2 [準備] ネイティブ Ubuntu の場合

下記の手順で Docker 環境をインストールしてください。すでにインストール済みであればこの項はスキップして Ok です。

```
$ sudo apt update
$ sudo apt install -y ca-certificates curl gnupg
$ sudo install -m 0755 -d /etc/apt/keyrings (公式のGPGキーを保存する場所を作成)
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/k
eyrings/docker.gpg (公式のGPGキーを取得)
$ sudo chmod a+r /etc/apt/keyrings/docker.gpg (公式のGPGキーのアクセス権限指定)
$ echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://do
wnload.docker.com/linux/ubuntu \
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null (公式のDockerリポジトリを追加)
```

```
$ sudo apt update
$ sudo apt install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-comp
>ose-plugin (Dockerをインストール)
$ sudo usermod -aG docker $USER (ユーザをdockerグループに追加)
$ su -$(USER) (変更を反映)
```

2.3.3 サンプル設計データ factory_test で LibreLane の設計フローを動かす

ここから先の作業は、Windows 11 WSL2 Ubuntu とネイティブ Ubuntu の両方で共通です。まず、ローカル PC 上の LibreLane の設計フローの動作を、サンプル設計データ <https://github.com/TinyTapeout/ttsky25b-factory-test> を使って確認しましょう。この項の内容は、<https://tinytapeout.com/guides/local-hardening/>をベースにしています。

必要なツールをインストール

下記コマンドで必要なツールをインストールしてください。

```
$ sudo apt update
$ sudo apt upgrade
$ sudo apt install python3.12-venv python3-tk make make-guile iverilog gtkwave librsvg2-bin >
>pngquant eog
```

環境セットアップ

Python のバージョンを確認してください。バージョン 3.11 以上が必要です。

```
$ python3 --version
```

作業ディレクトリを作成します。

```
$ mkdir ~/TinyTapeout
$ cd ~/TinyTapeout
```

サンプル設計データ factory_test をダウンロードしてください。

```
$ git clone https://github.com/TinyTapeout/ttsky25b-factory-test factory-test
```

tt-support-tools をセットアップ

下記コマンドを実行します。factory_test の下にディレクトリ tt ができます。

```
$ cd factory_test
$ git clone https://github.com/TinyTapeout/tt-support-tools tt
```

仮想 Python 環境の作成

仮想 Python 環境用の専用ディレクトリを作成し、初期化してください。

```
$ cd ~/TinyTapeout
$ mkdir ttsetup
$ python3 -m venv ./ttsetup/venv
$ source ./ttsetup/venv/bin/activate
```

必要な関連ツールをインストールします。

```
$ cd ~/TinyTapeout
$ pip install -r ./factory-test/tt/requirements.txt
```

環境変数の設定

ここでは、Skywater 社の 130nm プロセス向けの PDK を使うことを前提に説明します。ファイル`~/.bashrc`に下記を追加してください。

```
export PDK_ROOT=~/TinyTapeout/ttsetup/pdk
export PDK=sky130A
export LIBRELANE_TAG=2.4.2
```

下記コマンドで環境変数を有効化します。

```
$ source ~/.bashrc
```

LibreLane をインストール

下記コマンドで LibreLane をインストールしてください。

```
$ pip install librelane==$LIBRELANE_TAG
```

factory_test を LibreLane フローで合成する

ここまでが準備です。ここからサンプル設計データ `factory_test` を LibreLane フローで合成しましょう。`factory_test` の RTL 記述は `factory_test/src/tt_um_factory_test.v` です。入出力信号の宣言部は下記の通りで、内部は、8 ビットのカウンタをカウント・アップしていく、リセット信号が入力端子の状態に応じて、出力端子や入出力端子からカウンタの値または入力端子の値を出力する簡単なロジックになっています。

```
module tt_um_factory_test (
    input wire [7:0] ui_in,      // Dedicated inputs
    output wire [7:0] uo_out,    // Dedicated outputs
    input wire [7:0] uio_in,    // IOs: Input path
    output wire [7:0] uio_out,   // IOs: Output path
    output wire [7:0] uio_oe,   // IOs: Enable path (active high: 0=input, 1=output)
    input wire ena,           // always 1 when the design is powered, so you can ignore it
    input wire clk,            // clock
    input wire rst_n          // reset_n - low to reset
);
```

まず、LibreLane の設定ファイルを生成します。

```
$ cd ~/TinyTapeout/factory-test
$ ./tt/tt_tool.py --create-user-config
```

下記コマンドで LibreLane のフローを一気に流します。RTL 記述の論理合成からレイアウトまで実行します。このコマンドの実行からは Docker が動作している必要があります。初回の実行は、PDK など必要なファイルのダウンロードを実行するので時間がかかります。

```
$ ./tt/tt_tool.py --harden
```

エラーなく実行が終わると、最後の方に下記メッセージが出力されます。

```
* Antenna
Passed 煙

* LVS
Passed 煙

* DRC
Passed 煙
```

Antenna は、アンテナ・エラーの有無のことです。NMOS トランジスタや PMOS トランジスタのゲート (G) 入力に繋がるメタル配線はチップが完成した状態なら、通常は他のトランジスタのドレイン (D) 出力に繋がっており、その配線にチャージが蓄積されてもその電位はドレインと WELL 間の寄生ダイオードにより VDD(電源) ラインと VSS(GND) ラインの電位の間に収まり、トランジスタのゲート (G) 入力端子の絶縁膜を破壊することはありません。しかし、ウェハの製造途中では配線層を下から積み上げて行くので、トランジスタのゲート (G) 入力端子のメタル配線がどこにも接続されず浮いていることがあります。そのメタル配線にチャージが蓄積すると電位 (の絶対値) が際限なく大きくなる可能性がありゲート (G) 入力端子の絶縁膜を破壊する可能性があります。このため、トランジスタのゲート (G) 入力に繋がるメタル配線の長さは一定以下にするようにレイアウト・ルールで決められていて、これを違反するとエラーになります。ただし、Skywater 社 130nm を使う Tiny Tapeout ではアンテナ・エラーがあってもデータ提出と試作はできるようです。

LVS は、Layout Versus Schematic の略で、物理レイアウトとゲート・レベル・ネット (回路ネット) との一致確認です。これは必ずパスしている必要があります。

DRC は、Design Rule Check の略で、物理レイアウトがファブの寸法ルールやクリアランスなどを満たしているの確認です。これも必ずパスしている必要があります。

合成時などのワーニング・メッセージは下記コマンドで確認できます。

```
$ ./tt/tt_tool.py --print-warnings
```

LibreLane フローを再実行するには

Python の仮想環境が有効でなければ、下記コマンドを実行してください。

```
$ cd ~/TinyTapeout
$ source ./ttsetup/venv/bin/activate
```

RTL 記述やこの後説明するタイル数の設定などを変更したときは下記コマンドを入力します。

```
$ cd ~/TinyTapeout/factory-test
$ ./tt/tt_tool.py --create-user-config
```

そして、LibreLane フローを下記コマンドで再実行してください。

```
$ ./tt/tt_tool.py --harden
```

論理検証方法 (RTL ベース)

Tiny Tapeout は論理シミュレータによる機能検証をサポートしています。検証対象は System Verilog で記述されたロジックですが、テストベンチは Python で記述します。Python には大量のソフトウェア・パッケージが用意されており、柔軟かつ高機能なテストベンチを構築できるようになります。このために使う環境は **cocotb**(coroutine cosimulation testbench) です。cocotb に対応している論理シミュレータは、オープンソースの Icarus Verillog や Verilator、商用ツールの Synopsys 社の VCS、Siemens/Mentor 社の Questa や ModelSim、Cadence 社の Xcelium などたくさんあります。

基本的に cocotb テストベンチは追加の RTL コードを必要としません。検証対象の DUT(Design Under Test) は、ラッパー・コードなしでシミュレータのトップ・レベルとしてインスタンス化されます。cocotb は DUT への入力(または階層の下位)にスティミュラスを与え、Python から直接出力を監視します。なお、cocotb は HDL ブロックを個々にインスタンス化できません。DUT が一つのモジュールとして完結している必要があります。

検証動作はシンプルな Python の関数がベースです。任意の時点で、シミュレータが時間を進めているか、Python コードが実行されているかのいずれかです。Python 記述内の Await キーワードは、実行の制御をシミュレータに戻す時期を示すために使用されます。検証動作中は複数のコルーチンを生成でき、それぞれ独立して実行できます。

cocotb のしくみの全体イメージを図 2.7 に示します。論理シミュレータ側のインターフェースは、GPI(Generic Procedural Interface) という抽象化レイヤーです。GPI には、VPI、VHPI、FLI の 3 種類があります。VPI(Verilog Procedural Interface) は Verilog/SystemVerilog 標準の C 言語インターフェースで多くのシミュレータ(Icarus, Verilator, VCS 等)で利用されます。VHPI(VHDL Procedural Interface) は VHDL 標準のインターフェースで、Questa や Xcelium などでサポートが進んでいます。FLI(Foreign Language Interface) は Questa/ModelSim 独自のインターフェースで VHDL を扱う際の主流になっています。

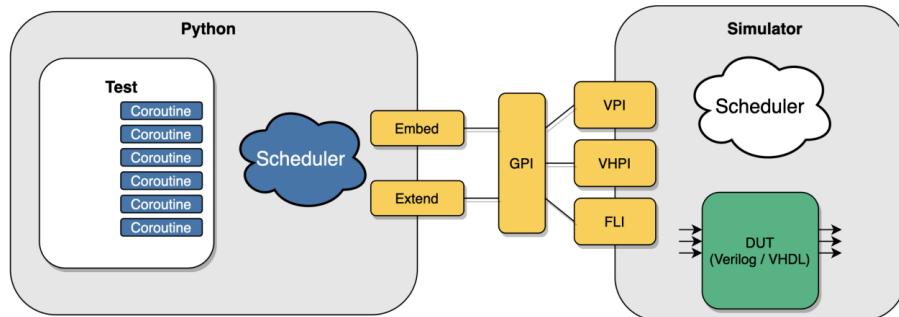


図 2.7: cocotb のしくみ (<https://docs.cocotb.org/en/stable/>)

サンプル設計データ factory_test において、機能検証用の Python で記述されたテストベンチは、factory_test/test/test.py です。この具体的な記述方法は、bfCPU を Tiny Tapeout で設計する後述する項で説明します。とりあえず、ここでは cocotb を含むツール動作を確認するまでにしておきます。

まず下記コマンドで必要な関連ツールをインストールしてください。

```
$ cd ~/TinyTapeout/factory-test/test
$ pip install -r requirements.txt
```

これ以降、RTL ベースの論理シミュレーションは下記コマンドだけで実行できます。

```
$ cd ~/TinyTapeout/factory-test/test
$ make -B
```

そして下記のような表示が出ればシミュレーションがパスしたことになります。

** test.test_loopback	PASS	5220000.00	0.02	319638332.77	**
** test.test_counter	PASS	2760000.00	0.01	379351128.59	**

論理検証方法(ゲート・レベル)

チップをテープアウトする前までには、必ずゲート・レベルの論理検証を行う必要があります。RTL 記述からゲート・レベルの論理回路を合成しそれをチップとしてレイアウト(配置・配線)しますが、チップ・レベルで物理的に正しいかどうかの確認のためゲート・レベルでの論理検証を行います。

その準備として、下記コマンドで PDK の置き場所に対してリンクを張る必要があります。このパス名のうちの長い16進数の部分(0xfe5...54af)は環境によって変わることもありますので注意してください。

```
$ cd ~/TinyTapeout/ttsetup/pdk
$ ln -s ciel/sky130/versions/0fe599b2afb6708d281543108caf8310912f54af/sky130A sky130A
$ ln -s ciel/sky130/versions/0fe599b2afb6708d281543108caf8310912f54af/sky130B sky130B
```

この準備以降は、下記コマンドでゲート・レベルのシミュレーションを実行できます。最初に、LibreLane フローで生成したゲート・レベルの Verilog HDL ネット tt_um_factory_test.pnl.v を gate_level_netlist.v にコピーしてから実行します。

```
$ cd ~/TinyTapeout/factory-test/test
$ cp ./runs/wokwi/final/pnl/tt_um_factory_test.pnl.v gate_level_netlist.v
$ make -B GATES=yes
```

こちらも下記のようなメッセージが出ればシミュレーションがパスしたことになります。

** test.test_loopback	PASS	5220000.00	0.02	284903535.29	**
** test.test_counter	PASS	2760000.00	0.01	320016559.96	**

レイアウト・データを見る

LibreLane フローで完成したレイアウト・データをレイアウト・ツールで見ることができます。Ubuntu 環境によっては、X サーバー(画面出力)への接続が許可されておらずエラーになることがあります。レイアウト・ツールを起動するコマンドでエラーが出たら下記コマンドで X サーバーへの接続を全ホストに許可します。

```
$ xhost +
```

OpenROAD GUI でレイアウト・データを見る場合は(図 2.8 左)、下記コマンドを実行します。

```
$ cd ~/TinyTapeout/factory-test
$ ./tt/tt_tool.py --open-in-openroad
```

KLayout でレイアウト・データを見る場合は(図 2.8 右)、下記コマンドを実行します。

```
$ cd ~/TinyTapeout/factory-test
$ ./tt/tt_tool.py --open-in-klayout
```

レイアウト・データを画像ファイル(PNG)にエキスポートする

レイアウト・データを画像ファイル(PNG)にエキスポートできます。下記コマンドを実行してください。

```
$ cd ~/TinyTapeout/factory-test
$ ./tt/tt_tool.py --create-png
```

PNG ファイルを見るには、下記コマンドを実行してください。

```
$ eog gds_render.png
```

ローカル環境での LibreLane フローは OK

これでローカル環境で LibreLane フローを流す環境が立ち上りました。

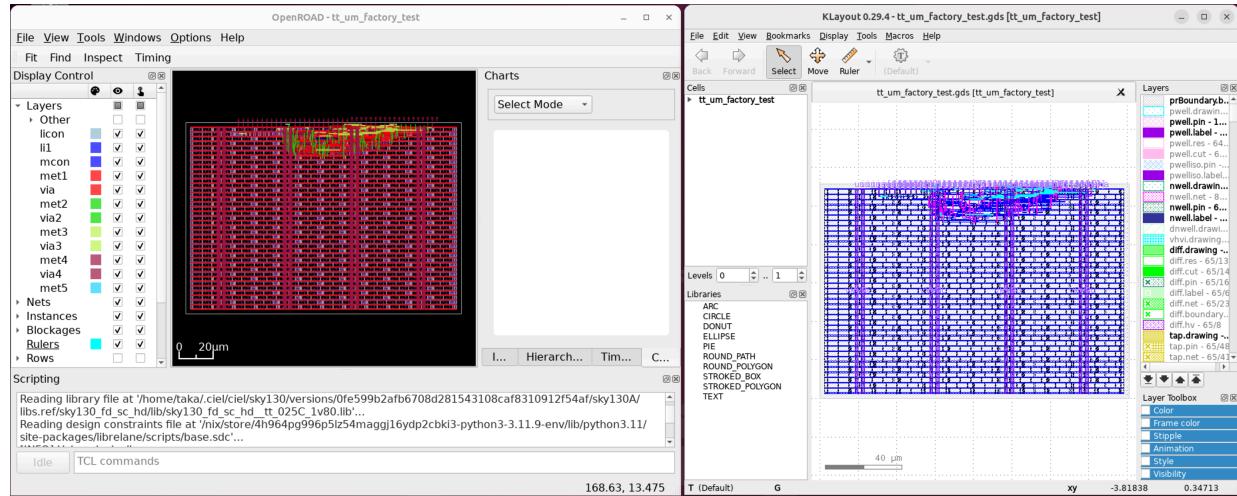


図 2.8: レイアウト・データを見る

2.4 LibreLane 設計フローをクラウド上で動かす【正式設計フロー】

Tiny Tapeout の正式設計フローはクラウド上で動かす方法です。このフローの方が設計ルールのチェック内容が多く、チップ試作のための設計データ提出は必ずクラウド上の設計フローが正常終了している必要があります。ここからは bfCPU を題材にしてローカル上とクラウド上の両方で LibreLane フローを動かしてみましょう。

2.4.1 Tiny Tapeout の GitHub からテンプレートを Fork する

ここからは、GitHub での作業がメインになります。GitHub へのユーザ登録とログインが必要になるのでこれは各自で対応してください。

まず最初に Tiny Tapeout の GitHub からテンプレートを Fork してください。ここでは、ファブ (PDK) として Skywater 社 130nm を選択し、その Verilog HDL 用のテンプレートを使います。

GitHub サイト <https://github.com/TinyTapeout/ttsky-verilog-template> を開いてください。画面図 2.9 の Fork ボタンを押して、Create New Fork を選択します。

Owner は自分の ID、Repository Name は、他の人のプロジェクトと区別がつく名称にしてください。ここでは名称を ttsky_bfCPU にした場合で説明しますが、ご自身の ID を含めた名称にすることを推奨します。View existing forks を選択すると他の人のプロジェクト名がリスト表示されるのでそれを参照して名称を決めてください。

Fork する時の注意: Tiny Tapeout のテンプレートを 1 回 Fork してそれをそのまま維持した状態で、新規に別のプロジェクトを設計しようとしてもう 1 回同じテンプレートを Fork することは Github のポリシーが同一レポジトリを複数個 Fork できないということで躓かれてしまいます。こういうケースは、前回 Fork してできたレポジトリを複製して中身を残しておいて、そのレポジトリを手動で削除して Fork をデタッチして、新規にテンプレートを Fork してください。Fork のデタッチについては <https://docs.github.com/ja/pull-requests/collaborating-with-pull-requests/working-with-forks/detaching-a-fork> を参照してください。

2.4.2 Fork してできたリポジトリをローカルに Clone する

Tiny Tapeout のテンプレートを Fork すると図 2.10 のように自分のリポジトリに登録されます。ボタン Code を押して、Clone に表示される URL をコピーして下記コマンドで Clone してください。下記に表示した

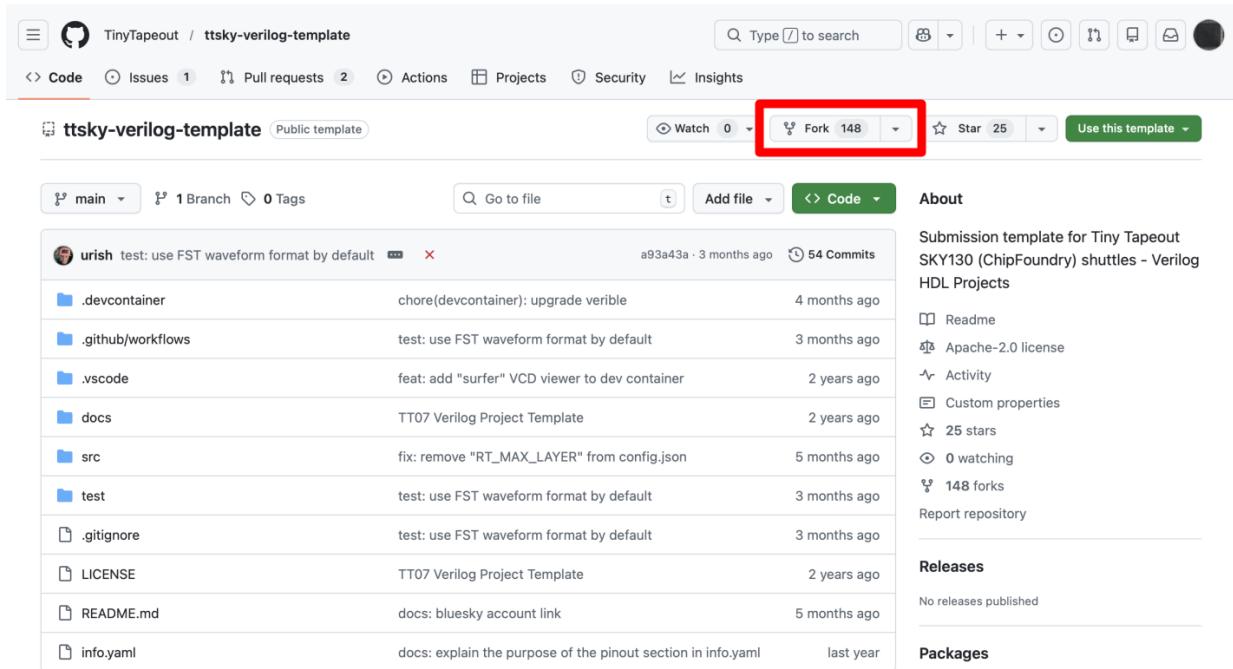


図 2.9: Tiny Tapeout の GitHub からテンプレートを Fork

URL は自分のものに置き換えてください。

```
$ cd ~/TinyTapeout
$ git clone https://github.com/munetomo-maruyama/ttsky_bfCPU.git
```

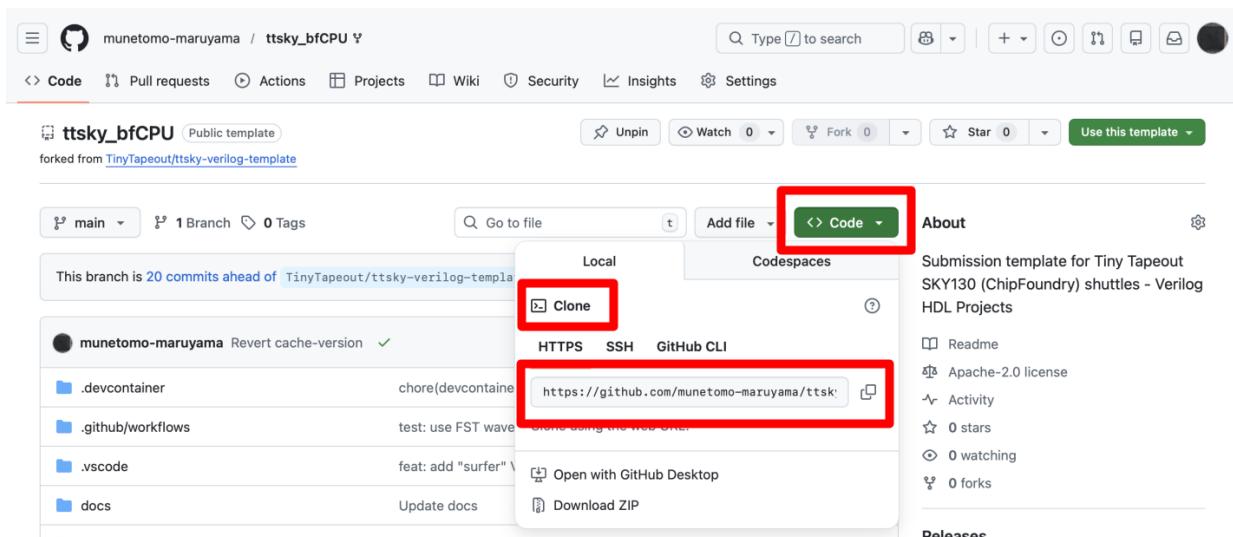


図 2.10: Fork してできたリポジトリをローカルに Clone する

Tiny Tapeout 用に完成したリポジトリをローカルに clone して設計フローを確認するだけなら、上記の git clone コマンドをそのまま実行してこのリポジトリのコピー入手しても構いません。

2.4.3 bfCPU の RTL 記述を準備

bfCPU の RTL 記述をこのリポジトリにコピーしましょう。前章で作成した bfCPU のリポジトリ bfCPU を

カレント・ディレクトリにして、下記コマンドでコピーしてください。

```
$ cd ../bfCPU (bfCPUのリポジトリ)
$ cp -r ./RTL/* ~/TinyTapeout/src
```

そして Tiny Tapeout 用の最上位記述を作成します。ファイル名は tt_um_ で始まる必要があります。ここでは tt_um_bfcpu.sv というファイル名にして ~/TinyTapeout/src の下に作成します。この最上位階層は前章の bfCPU の最上位階層 FPGA に対応しており、TOP 階層をインスタンス化し、出力端子をオープン・ドライン駆動するように記述します。記述内容を下記に示します。

```
`default_nettype none

module tt_um_bfcpu (
    input logic [7:0] ui_in,      // Dedicated inputs
    output logic [7:0] uo_out,     // Dedicated outputs
    input logic [7:0] uio_in,     // IOs: Input path
    output logic [7:0] uio_out,    // IOs: Output path
    output logic [7:0] uio_oe,    // IOs: Enable path (active high: 0=input, 1=output)
    input logic        ena,       // always 1 when the design is powered, so you can ignore i>
>t
    input logic        clk,       // clock
    input logic        rst_n     // reset_n - low to reset
);
//
logic      qspi_cs_n;
logic      qspi_sck;
logic [3:0] qspi_sio_o;
logic [3:0] qspi_sio_i;
logic      qspi_cs_e;
logic      qspi_sck_e;
logic [3:0] qspi_sio_e;
logic      uart_txd;
logic      uart_rxd;
//
TOP_U_TOP
(
    .CLK    (clk),
    .RES_N (rst_n),
    //
    //
    .QSPI_CS_N (qspi_cs_n),
    .QSPI_CS_E (qspi_cs_e),
    .QSPI_SCK  (qspi_sck),
    .QSPI_SCK_E (qspi_sck_e),
    .QSPI_SIO_O (qspi_sio_o),
    .QSPI_SIO_E (qspi_sio_e),
    .QSPI_SIO_I (qspi_sio_i),
    //
    .UART_TXD   (uart_txd),
    .UART_RXD   (uart_rxd)
);

// All output pins must be assigned. If not used, assign to 0.
assign uo_out[0] = uart_txd;
assign uo_out[1] = 1'b0;
assign uo_out[2] = 1'b0;
assign uo_out[3] = 1'b0;
assign uo_out[4] = 1'b0;
assign uo_out[5] = 1'b0;
assign uo_out[6] = 1'b0;
```

```

assign uo_out[7] = 1'b0;
//
assign ui_out[0] = 1'b0;
assign ui_out[1] = 1'b0;
assign ui_out[2] = qspi_sio_o[2];
assign ui_out[3] = qspi_sio_o[3];
assign ui_out[4] = qspi_cs_n;
assign ui_out[5] = qspi_sck;
assign ui_out[6] = qspi_sio_o[0];
assign ui_out[7] = qspi_sio_o[1];
//
assign ui_oe[0] = 1'b0;
assign ui_oe[1] = 1'b0;
assign ui_oe[2] = qspi_sio_e[2];
assign ui_oe[3] = qspi_sio_e[3];
assign ui_oe[4] = qspi_cs_e;
assign ui_oe[5] = qspi_sck_e;
assign ui_oe[6] = qspi_sio_e[0];
assign ui_oe[7] = qspi_sio_e[1];
//
assign uart_rxd = ui_in[7];
assign qspi_sio_i[0] = ui_in[6];
assign qspi_sio_i[1] = ui_in[7];
assign qspi_sio_i[2] = ui_in[2];
assign qspi_sio_i[3] = ui_in[3];

// List all unused inputs to prevent warnings
wire _unused;
assign _unused = &{1'b0, ui_in[6:0], ui_in[5:4], ui_in[1:0], ena};

endmodule

```

2.4.4 info.yaml を書き換える

自分のプロジェクト情報を info.yaml に記入してください。~/TinyTapeout/ttsky_bfCPU/info.yaml を編集します。設計結果を試作のために提出する場合は、運営からの連絡用に、Tiny Tapeout が使う SNS Discord を使えるようにして ID を取得してください。詳細は <https://discord.com> を参照してください。

- **title:** プロジェクトのタイトルを記入
- **author:** 自分の名前
- **discord:** Tiny Tapeout の SNS Discord の ID を記入 (運営からの連絡用)
- **description:** プロジェクトの概要を 1 行で説明
- **language:** SystemVerilog を指定
- **clock_hz:** 動作周波数を単位 [Hz] で記入 (bfCPU は 10MHz なので 10000000 を記入)
- **tiles:** 論理規模に応じてタイルのサイズを指定 (bfCPU のキャッシュ有版なら 2x2、キャッシュ無版なら 1x2)
- **top_module:** 最上位階層 RTL のモジュール名を指定 (tt_um_で始まるファイル名)
- **source_files:** 使用する RTL 記述ファイルを 1 行 1 ファイル で列挙 (書き方は下記参照)
- **pinout:** チップ外部の入力端子 ui[0] ~ ui[7]、出力端子 uo[0] ~ uo[7]、入出力端子 ui[0] ~ ui[7] の各端子機能を表す名称を記入

論理規模に影響を受けるタイルのサイズ (tiles) は、最初は小さいサイズを指定して LibreLane フローを流して、ロジック規模がパンクしたというエラーが出たら、少しづつ大きくしていってください。

bfCPU の場合、info.yaml は下記内容になります。

```
N# Tiny Tapeout project information
project:
    title:      "bfCPU"      # Project title
    author:     "Munetomo Maruyama"  # Your name
    discord:    "munetomo_85572"    # Your discord username, for communication and automatically assigning you a Tapeout role (optional)
    description: "A universal Turing Machine Implemented as a Superset of Brainf*ck" # One line description of what your project does
    language:   "SystemVerilog" # other examples include SystemVerilog, Amaranth, VHDL, etc
    clock_hz:   10000000       # Clock frequency in Hz (or 0 if not applicable)

    # How many tiles your design occupies? A single tile is about 167x108 uM.
    tiles:      "2x2"        # Valid values: 1x1, 1x2, 2x2, 3x2, 4x2, 6x2 or 8x2

    # Your top module name must start with "tt_um_". Make it unique by including your github username:
    top_module: "tt_um_bfcpu"

    # List your project's source files here.
    # Source files must be in ./src and you must list each source file separately, one per line.
    # Don't forget to also update `PROJECT_SOURCES` in test/Makefile.
    source_files:
        - "tt_um_bfcpu.sv"
        - "TOP/top.sv"
        - "CPU/cpu.sv"
        - "CACHE/cache.sv"
        - "QSPI_SRAM/qspi_sram.sv"
        - "UART/uart.sv"
        - "UART/sasc/trunk/rtl/verilog/sasc_top.v"
        - "UART/sasc/trunk/rtl/verilog/sasc_fifo4.v"
        - "UART/sasc/trunk/rtl/verilog/sasc_brg.v"

    # The pinout of your project. Leave unused pins blank. DO NOT delete or add any pins.
    # This section is for the datasheet/website. Use descriptive names (e.g., RX, TX, MOSI, SCL, SEG_A, etc.).
    pinout:
        # Inputs
        ui[0]: ""
        ui[1]: ""
        ui[2]: ""
        ui[3]: ""
        ui[4]: ""
        ui[5]: ""
        ui[6]: ""
        ui[7]: "UART_RXD"

        # Outputs
        uo[0]: "UART_TXD"
        uo[1]: ""
        uo[2]: ""
        uo[3]: ""
        uo[4]: ""
        uo[5]: ""
        uo[6]: ""
        uo[7]: ""

        # Bidirectional pins
        uio[0]: ""
        uio[1]: ""
        uio[2]: "QSPI_SI02"
        uio[3]: "QSPI_SI03"
```

```

uio[4]: "QSPI_CS_N"
uio[5]: "QSPI_SCK"
uio[6]: "QSPI_SIO0"
uio[7]: "QSPI_SIO1"

# Do not change!
yaml_version: 6

```

2.4.5 ローカル PC で設計確認

ここでローカル PC 上で LibreLane を使って設計内容を確認しましょう。すでに factory_test が動いているので環境自体は問題ないはずです。下記のコマンドを入力して確認します。

```

$ cd ~/TinyTapeout/ttsky_bfCPU (自分のレポジトリに移動)
$ source ../ttsetup/venv/bin/activate
$ ./tt_tt_tool.py --create-user-config (LibreLaneの設定ファイルを生成)
$ ./tt_tt_tool.py --harden (LibreLane設計フローを流す)

```

2.4.6 エラーの対策

慣れていない最初の頃はエラーが出まくると思います。RTL 記述のミス、ファイルの置き場所違い、設定ファイルのスペルミスなど、色々と悩むと思いますが、エラー・メッセージを見ながらコツコツ対策していくください。ただし、設計内容の本質によってエラーが出るものがあります。大きく分けて以下の3つです。

- **タイル・サイズと論理規模のアンマッチ:** info.yaml で指定したタイル・サイズが論理規模の実態よりも小さいとエラーになります。この場合は、タイル・サイズを大きくするか、論理規模自体を削減して、再実行してください。
- **論理セル配置密度のアンマッチ:** 論理セル配置密度は、ファイル ttsky_bfCPU/src/config.json の中のパラメータ PL_TARGET_DENSITY_PCT で指定します。0.60 に設定する場合は 60 を指定します。配置密度の設定値が高いと論理セル間の隙間がなくて全配線を通せないことがあります。逆に配置密度の設定値が低いと、面積が増加して指定したタイル・サイズに入りきらないことがあります。この設定値が問題でエラーが出た場合は、エラー・メッセージの中に推奨設定値を表示してくれることがあります。設定例をリスト 2.1 に示します。
- **アンテナ・エラー:** LibreLane の harden フローが終った時点で、LVS と DRC は必ずパスしている必要がありますが、Antenna はエラーがあっても、設計データ提出と試作は可能です。ただし、設計結果ログ・ファイル ttsky_bfCPU/runs/wokwi/45-openroad-checkantennas-1/openroad-checkantennas-1.log の中のキーワード VIOLATED を検索して、Partial area ratio が Required ratio より桁違いに大きい場合は対策した方がいいでしょう。ファイル ttsky_bfCPU/src/config.json の中にリスト 2.2 の記述を追加して再実行してみてください。

リスト 2.1: config.json の PL_TARGET_DENSITY_PCT の設定例

```
"PL_TARGET_DENSITY_PCT": 60,
```

リスト 2.2: config.json の Antenna 改善対策例

```

"DIODE_INSERTION_STRATEGY": 4,
"GRT_ANT_ITERS": 20,
"GRT_MAX_DIODE_INS_ITERS": 20,
"GRT_ANT_MARGIN": 8,
"GLB_RESIZER_ALLOW_ANTENNAS": 0,

```

2.4.7 論理シミュレーション

論理シミュレーションを実行するには、いくつかのファイルを用意します。

Makefile の編集

RTL ベースのシミュレーションとゲート・レベルのシミュレーションの両方に使う **ttsky_bfCPU/test/Makefile** を編集します。使う RTL 記述ファイルやテストベンチ記述、Icarus Verilog シミュレータのオプション指定などを記述します。記述例をリスト 2.3 に示します。

リスト 2.3: 論理シミュレーションの Makefile

```
# Makefile
# See https://docs.cocotb.org/en/stable/quickstart.html for more info

# defaults
SIM ?= icarus
FST ?= -fst # Use more efficient FST format
TOPLEVEL_LANG ?= verilog
SRC_DIR = $(PWD)/../src
PROJECT_SOURCES = \
    tt_um_bfcpu.sv \
    TOP/top.sv \
    CPU/cpu.sv \
    CACHE/cache.sv \
    QSPI_SRAM/qspi_sram.sv \
    UART/uart.sv \
    UART/sasc/trunk/rtl/verilog/sasc_top.v \
    UART/sasc/trunk/rtl/verilog/sasc_fifo4.v \
    UART/sasc/trunk/rtl/verilog/sasc_brg.v
SYS_DIR=$(PWD)/../test/QSPI
SYSTEM_SOURCES = 23LC512.v

ifeq ($(GATES),yes)

# RTL simulation:
SIM_BUILD = sim_build/rtl
VERILOG_SOURCES += $(addprefix $(SRC_DIR)/,$(PROJECT_SOURCES))
VERILOG_SOURCES += $(addprefix $(SYS_DIR)/,$(SYSTEM_SOURCES))

else

# Gate level simulation:
SIM_BUILD = sim_build/gl
COMPILE_ARGS += -DGL_TEST
COMPILE_ARGS += -DFUNCTIONAL
COMPILE_ARGS += -DUSE_POWER_PINS
COMPILE_ARGS += -DSIM
COMPILE_ARGS += -DUNIT_DELAY=\#1
COMPILE_ARGS += -DSIMULATION
COMPILE_ARGS += -DGATESIM
VERILOG_SOURCES += $(addprefix $(SYS_DIR)/,$(SYSTEM_SOURCES))
VERILOG_SOURCES += $(PDK_ROOT)/sky130A/libs.ref/sky130_fd_sc_hd/verilog/primitives.v
VERILOG_SOURCES += $(PDK_ROOT)/sky130A/libs.ref/sky130_fd_sc_hd/verilog/sky130_fd_sc_hd.v

# this gets copied in by the GDS action workflow
VERILOG_SOURCES += $(PWD)/gate_level_netlist.v

endif

# Allow sharing configuration between design and testbench via `include`:
COMPILE_ARGS += -I$(SRC_DIR)
# Add Compile Options
```

```

COMPILE_ARGS          += -g2012
COMPILE_ARGS          += -DSIMULATION

# Include the testbench sources:
VERILOG_SOURCES += $(PWD)/tb.sv
TOPLEVEL = tb

# List test modules to run, separated by commas and without the .py suffix:
COCOTB_TEST_MODULES = test

# include cocotb's make rules to take care of the simulator setup
include $(shell cocotb-config --makefiles)/Makefile.sim

```

テストベンチ cocotb と検証対象論理の間に挟まる最上位 RTL 記述 tb.sv を作成

論理シミュレーションを駆動するテストベンチは Python ベースの cocotb ですが、それと検証対象論理の間に挟まる最上位 RTL を作成します。ファイルは `ttsky_bfCPU/test/tb.sv` で内容をリスト 2.4 に示します。この記述は下記の役割を持ちます。なお、cocotb の Python 記述にここで説明する記述内容を含めることもできると思いますが、まだ筆者が cocotb の深い使い方に慣れていないので、この方式にしました。

- **波形ファイル VCD の出力指定:** RTL ベースによる波形ファイルと、ゲート・レベルによる波形ファイルを分けて出力します。
- **メモリの初期化:** QSPI SRAM 内に bfCPU プログラム（バイナリ）と UART ポーレート設定値を書き込みます。ここで使うプログラムは乗算（multiplication.v）です。
- **検証対象論理のインスタンス化:** モジュール `tt_um_bfcpu` を、`U_CHIP` という名前でインスタンス化します。
- **QSPI SRAM のインスタンス化:** bfCPU システムの外部に接続する QSPI SRAM の動作モデルをインスタンス化します。
- **UART の受信動作サポート:** bfCPU の UART が受信待ちになったら、RXD 端子にテストベンチからデータを送信して UART に受信させます。その受信データは `$display()` で表示します。
- **UART の送信動作サポート:** bfCPU の UART が TXD 端子からデータ送信を開始したら、送信データを `$display()` で表示します。

リスト 2.4: テスト用最上位 RTL 記述 tb.sv

```

`default_nettype none
`timescale 1ns / 1ps

/* This testbench just instantiates the module and makes some convenient wires
   that can be driven / tested by the cocotb test.py.
*/

//`define TB_UART_BITWIDTH 8681 //ns, 115200bps
`define TB_UART_BITWIDTH 3200 //ns, for simulation

//-----
// Testbench Top
//-----
module tb ();
integer i;

//-----
// Dump the signals to a FST file. You can view it with gtkwave or surfer.
//-----
`ifdef GATESIM
initial begin

```

```

$dumpfile("tb_gate.vcd");
$dumpvars(0, tb);
end
`else
initial begin
    $dumpfile("tb.vcd");
    $dumpvars(0, tb);
end
`endif

//-----
// Initialize RAM
//-----
initial
begin
    // Initialzie Whole Mat (to avoid get unknown data)
    for (i = 0; i < 65536; i = i + 1) U_M23LC512.MemoryBlock[i] = 8'(i);
    //
    // Program Code
    $readmemh("./QSPI/multiplication.v", U_M23LC512.MemoryBlock);
    //
    // UART Baud Rate (for Simulation; 1bit period = 3200ns)
    `ifdef GATESIM
    U_M23LC512.MemoryBlock[32768+32766] = 8'h02; // DIV0
    U_M23LC512.MemoryBlock[32768+32767] = 8'h02; // DIV1
    `else
    U_M23LC512.MemoryBlock[32768+30] = 8'h02; // DIV0
    U_M23LC512.MemoryBlock[32768+31] = 8'h02; // DIV1
    `endif
end

//-----
// Device Under Test
//-----
logic      ena;      // always 1 when the design is powered, so you can ignore it
logic      clk;      // clock
logic      rst_n;    // reset_n - low to reset
//
logic [7:0] ui_in;   // Dedicated inputs
logic [7:0] uo_out;  // Dedicated outputs
logic [7:0] uio_in;  // IOs: Input path
logic [7:0] uio_out; // IOs: Output path
logic [7:0] uio_oe;  // IOs: Enable path (active high: 0=input, 1=output)
//
wire      qspi_cs_n;
logic      qspi_cs_n_o;
logic      qspi_cs_e;
wire      qspi_sck;
logic      qspi_sck_o;
logic      qspi_sck_e;
wire [3:0] qspi_sio;
logic [3:0] qspi_sio_o;
logic [3:0] qspi_sio_e;
logic [3:0] qspi_sio_i;
logic      uart_txd;
logic      uart_rxd;
//
assign qspi_cs_n    = (qspi_cs_e)? qspi_cs_n_o : 1'bz;
assign qspi_sck     = (qspi_sck_e)? qspi_sck_o : 1'bz;
assign qspi_sio[0]  = (qspi_sio_e[0])? qspi_sio_o[0] : 1'bz;
assign qspi_sio[1]  = (qspi_sio_e[1])? qspi_sio_o[1] : 1'bz;
assign qspi_sio[2]  = (qspi_sio_e[2])? qspi_sio_o[2] : 1'bz;

```

```
assign qspi_sio[3] = (qspi_sio_e[3])? qspi_sio_o[3] : 1'bz;
assign qspi_sio_i[0] = qspi_sio[0];
assign qspi_sio_i[1] = qspi_sio[1];
assign qspi_sio_i[2] = qspi_sio[2];
assign qspi_sio_i[3] = qspi_sio[3];
//
assign qspi_cs_n_o = uio_out[4];
assign qspi_sck_o = uio_out[5];
assign qspi_sio_o[0] = uio_out[6];
assign qspi_sio_o[1] = uio_out[7];
assign qspi_sio_o[2] = uio_out[2];
assign qspi_sio_o[3] = uio_out[3];
//
assign qspi_cs_e = uio_oe[4];
assign qspi_sck_e = uio_oe[5];
assign qspi_sio_e[0] = uio_oe[6];
assign qspi_sio_e[1] = uio_oe[7];
assign qspi_sio_e[2] = uio_oe[2];
assign qspi_sio_e[3] = uio_oe[3];
//
assign uio_in[6] = qspi_sio_i[0];
assign uio_in[7] = qspi_sio_i[1];
assign uio_in[2] = qspi_sio_i[2];
assign uio_in[3] = qspi_sio_i[3];
//
assign uart_txd = uo_out[0];
assign ui_in[7] = uart_rxd;
//
`ifndef GL_TEST
    wire VPWR = 1'b1;
    wire VGND = 1'b0;
`endif
//
// Chip Top
tt_um_bfcpu U_CHIP
(
    // Include power ports for the Gate Level test:
`ifdef GL_TEST
    .VPWR(VPWR),
    .VGND(VGND),
`endif
    .ui_in (ui_in),      // Dedicated inputs
    .uo_out (uo_out),    // Dedicated outputs
    .uio_in (uio_in),    // IOs: Input path
    .uio_out(uio_out),   // IOs: Output path
    .uio_oe (uio_oe),   // IOs: Enable path (active high: 0=input, 1=output)
    .ena    (ena),       // enable - goes high when design is selected
    .clk    (clk),       // clock
    .rst_n (rst_n)      // not reset
);
//-----
// SPIRAM (23LC512)
//-----
pullup (qspi_cs_n);
pullup (qspi_sck);
pullup (qspi_sio[0]);
pullup (qspi_sio[1]);
pullup (qspi_sio[2]);
pullup (qspi_sio[3]);
//
M23LC512 U_M23LC512
```

```

(
    .SI_SI00      (qspi_sio[0]),
    .SO_SI01      (qspi_sio[1]),
    .SCK          (qspi_sck),
    .CS_N         (qspi_cs_n),
    .SI02         (qspi_sio[2]),
    .HOLD_N_SI03 (qspi_sio[3]),
    .RESET        (~rst_n)
);

//-----
// Support Functions
//-----
//-----
// UART Generate Receive Data
//-----
logic detect_in;
logic [7:0] tx_data;
//
always_ff @(posedge clk, negedge rst_n)
begin
    if (~rst_n)
        detect_in <= 1'b0;
    else
        `ifndef GATESIM
            // backslash needs one space after the name
            detect_in <= U_CHIP.\U_TOP.U_UART.rxd_dphase ;
        `else
            detect_in <= U_CHIP.U_TOP.U_UART.rxd_dphase;
        `endif
end
//
logic [7:0] uart_rxd_data[0:255];
logic [7:0] uart_rxd_seq;
logic       uart_rxd_done;
//
initial
begin
    uart_rxd      = 1'b1;
    uart_rxd_seq  = 8'h00;
    uart_rxd_done = 1'b0;
    //
    for (i = 0; i < 256; i = i + 1) uart_rxd_data[i] = 8'(i);
    //
    @(posedge clk);
    @(posedge rst_n);
    @(posedge clk);
    //
    forever
    begin
        @(posedge detect_in);
        // Message
        $display("===== UART RxD ===== (IN ) at 0x%02x", uart_rxd_data[uart_rxd_seq]);
        #(1);
        // Start RXD
        #(`TB_UART_BITWIDTHH);
        uart_rxd = 1'b0;
        uart_rxd_done = 1'b1; // Trigger
        #(`TB_UART_BITWIDTHH);
        uart_rxd = uart_rxd_data[uart_rxd_seq][0];
        uart_rxd_done = 1'b0;
        #(`TB_UART_BITWIDTHH);

```

```
uart_rxd = uart_rxd_data[uart_rxd_seq][1];
#(`TB_UART_BITWIDTH);
uart_rxd = uart_rxd_data[uart_rxd_seq][2];
#(`TB_UART_BITWIDTH);
uart_rxd = uart_rxd_data[uart_rxd_seq][3];
#(`TB_UART_BITWIDTH);
uart_rxd = uart_rxd_data[uart_rxd_seq][4];
#(`TB_UART_BITWIDTH);
uart_rxd = uart_rxd_data[uart_rxd_seq][5];
#(`TB_UART_BITWIDTH);
uart_rxd = uart_rxd_data[uart_rxd_seq][6];
#(`TB_UART_BITWIDTH);
uart_rxd = uart_rxd_data[uart_rxd_seq][7];
#(`TB_UART_BITWIDTH);
uart_rxd = 1'b1;
uart_rxd_seq = uart_rxd_seq + 8'h01;
end
end

//-----
// UART Detect Transmit Data
//-----
logic [7:0] uart_txd_data;
logic      uart_txd_done;
//
initial
begin
    @(posedge clk);
    @(posedge rst_n);
    @(posedge clk);
    //
    forever
    begin
        uart_txd_done = 1'b0;
        @(negedge uart_txd);
        // Start TXD
        uart_txd_data = 8'h00;
        #(`TB_UART_BITWIDTH / 2);
        #(`TB_UART_BITWIDTH);
        uart_txd_data[0] = uart_txd;
        #(`TB_UART_BITWIDTH);
        uart_txd_data[1] = uart_txd;
        #(`TB_UART_BITWIDTH);
        uart_txd_data[2] = uart_txd;
        #(`TB_UART_BITWIDTH);
        uart_txd_data[3] = uart_txd;
        #(`TB_UART_BITWIDTH);
        uart_txd_data[4] = uart_txd;
        #(`TB_UART_BITWIDTH);
        uart_txd_data[5] = uart_txd;
        #(`TB_UART_BITWIDTH);
        uart_txd_data[6] = uart_txd;
        #(`TB_UART_BITWIDTH);
        uart_txd_data[7] = uart_txd;
        uart_txd_done = 1'b1; // Trigger
        #(`TB_UART_BITWIDTH);
        uart_txd_done = 1'b0;
        // Message
        $display("===== UART TxD ===== (OUT) at 0x%02x", uart_txd_data);
    end
end
```

```
endmodule
```

Python 記述の cocotb テストベンチ test.py を作成

cocotb の Python 記述テストベンチ **ttsky_bfCPU/test/test.py** を作成します。内容をリスト 2.5 に示します。この記述で行っていることは以下の通りです。

- **13 行目～14 行目**: 10MHz クロックを生成しています。
- **16 行目～30 行目**: テストベンチ階層の信号レベルの初期化、リセット信号の生成、UART RXD パターンの生成を行います。
- **38 行目～39 行目**: 乗算プログラムの UART 送信が完了した時点で送信データをチェックします。1 回目は $2 \times 3 = 6$ なので、0x06 と一致しているかチェックします。
- **41 行目～42 行目**: 乗算プログラムの UART 送信が完了した時点で送信データをチェックします。2 回目は $4 \times 5 = 20$ なので、0x14 と一致しているかチェックします。
- **44 行目～45 行目**: 乗算プログラムの UART 送信が完了した時点で送信データをチェックします。3 回目は $6 \times 7 = 42$ なので、0x2a と一致しているかチェックします。

リスト 2.5: Python 記述の cocotb テストベンチ test.py

```
1: # SPDX-FileCopyrightText: © 2024 Tiny Tapeout
2: # SPDX-License-Identifier: Apache-2.0
3:
4: import cocotb
5: from cocotb.clock import Clock
6: from cocotb.triggers import ClockCycles
7:
8: @cocotb.test()
9: async def test_project(dut):
10:     dut._log.info("Start")
11:
12:     # Set the clock period to 100 ns (10 MHz)
13:     clock = Clock(dut.clk, 100, unit="ns")
14:     cocotb.start_soon(clock.start())
15:
16:     # Assert Reset
17:     dut._log.info("Reset")
18:     dut.ena.value = 1
19:     dut.ui_in.value = 0x80 # bit7=RXD
20:     dut.uio_in.value = 0
21:     dut.rst_n.value = 0
22:
23:     # Initialize UART RXD Patterns
24:     await ClockCycles(dut.clk, 10)
25:     for i in range(0, 256, 1):
26:         dut uart_rxd_data[i].value = (i + 2) % 256
27:
28:     # Negate Reset
29:     await ClockCycles(dut.clk, 10)
30:     dut.rst_n.value = 1
31:
32:     dut._log.info("Simulation of bfCPU")
33:
34:     # Wait for one clock cycle to see the output values
35:     await ClockCycles(dut.clk, 10)
36:
37:     # Wait for TXD
38:     await dut uart_txd_done.rising_edge
39:     assert dut uart_txd_data.value == 0x06 # 2*3
40:     #
```

```

41:      await dut uart_txd_done.rising_edge
42:      assert dut uart_txd_data.value == 0x14 # 4*5
43:      #
44:      await dut uart_txd_done.rising_edge
45:      assert dut uart_txd_data.value == 0x2a # 6*7
46:
47:      # Finish
48:      await ClockCycles(dut.clk, 100)

```

RTL ベースの論理シミュレーションを実行

RTL ベースの論理シミュレーションは下記コマンドで実行できます。

```
$ cd ~/TinyTapeout/ttsky_bfCPU/test
$ make -B
```

このシミュレーションでは波形ファイルを生成しているので、gtkwave で見ることができます。

```
$ gtkwave tb.vcd tb.gtkw
```

ゲート・レベルの論理シミュレーションを実行

ゲート・レベルの論理シミュレーションは下記コマンドで実行できます。

```
$ cd ~/TinyTapeout/ttsky_bfCPU/test
$ cp ..../runs/wokwi/final/pnl/tt_um_factory_test.pnl.v gate_level_netlist.v
$ make -B GATES=yes
```

このシミュレーションでは波形ファイルを生成しているので、gtkwave で見ることができます。

```
$ gtkwave tb_gate.vcd tb_gate.gtkw
```

Tiny Tapeout で設計データを提出するには、ゲート・レベルの論理シミュレーションがパスしていることが必要です。

2.4.8 レイアウト結果を確認

LibreLane フローが生成したレイアウト結果を確認するには、下記コマンドを実行します。PNG ファイルを図 2.11 に示します。

```
$ cd ~/TinyTapeout/ttsky_bfCPU
$ ./tt_tt_tool.py open-in-openroad (OpenROAD GUIで開く)
$ ./tt_tt_tool.py --open-in-klayout (KLayoutで開く)
$ ./tt_tt_tool.py create-png (PNGを生成)
$ eog gds_render.png (PNGを開く)
```

2.4.9 docs/info.md を編集

ドキュメント・ファイル **docs/info.md** をオリジナルの内容から変更しておかないと、データ提出した後のチェックでエラーになります。下記の項目を適宜編集してください。フォーマットは Markdown です。
*** ## How it works * ## How to test * ## External hardware**

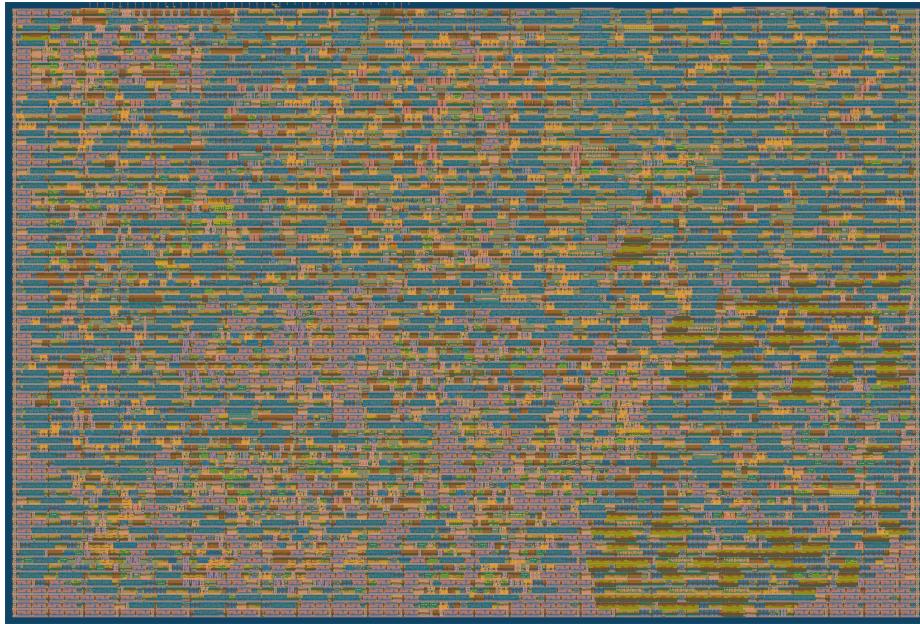


図 2.11: bfCPU のレイアウト (Skywater 0.13um プロセス)

2.4.10 クラウドにデータをアップロード

Action を有効化

Fork した自分の GitHub リモート・リポジトリで図 2.12 のように Action を有効化してください。

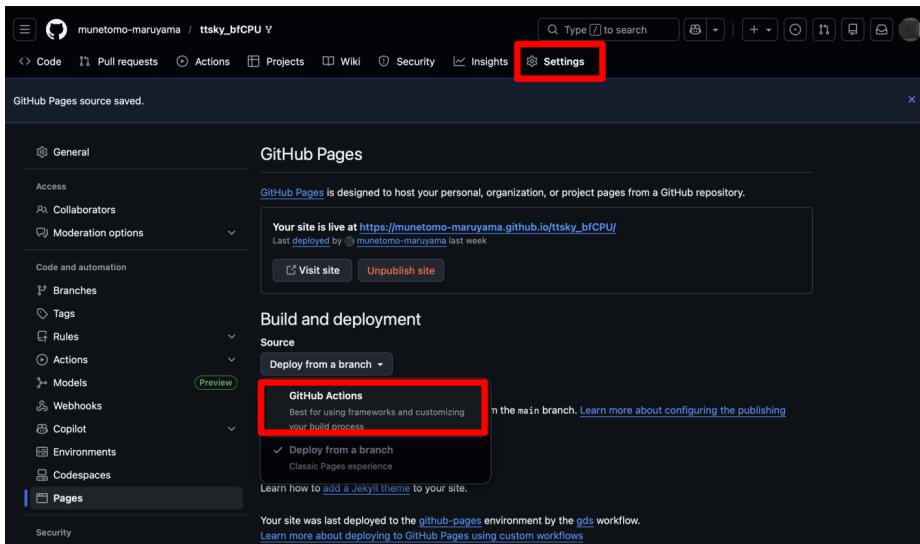


図 2.12: Fork した自分の GitHub リポジトリで Action を有効化

GitHub に追加・更新データをプッシュ

自分の GitHub リモート・リポジトリに追加・更新データをプッシュしてください。コマンドの例を以下に示します。

```
$ cd ~/TinyTapeout/ttsky_bfCPU (自分のローカル・リポジトリに移動)
$ git status (追加・修正ファイルの状況確認)
$ git add . (または追加・修正ファイルを $git add ファイル名 で指定)
```

```
$ git commit -m " 今回のコミット内容を1行で記述 "
$ git push
```

GitHub リポジトリが更新されると、自動的に LibreLane フローを実行

GitHub リモート・リポジトリが更新されると、自動的に LibreLane フローを実行します。図 2.13 のように Actions の Workflow の “docs” “gds” “test” に緑チェックが付けば OK です。“docs” は docs/info.md がオリジナルに対して更新されていれば OK になります。“test” は RTL レベルの論理シミュレーションがパスすれば OK になります。“gds” は、LibreLane フローが通ってレイアウト GDS データが正常に生成され、かつゲート・レベルの論理シミュレーションがパスすれば OK になります。詳細な結果やエラーの内容は、Workflows 内のそれぞれの項目をクリックすると見ることができます。

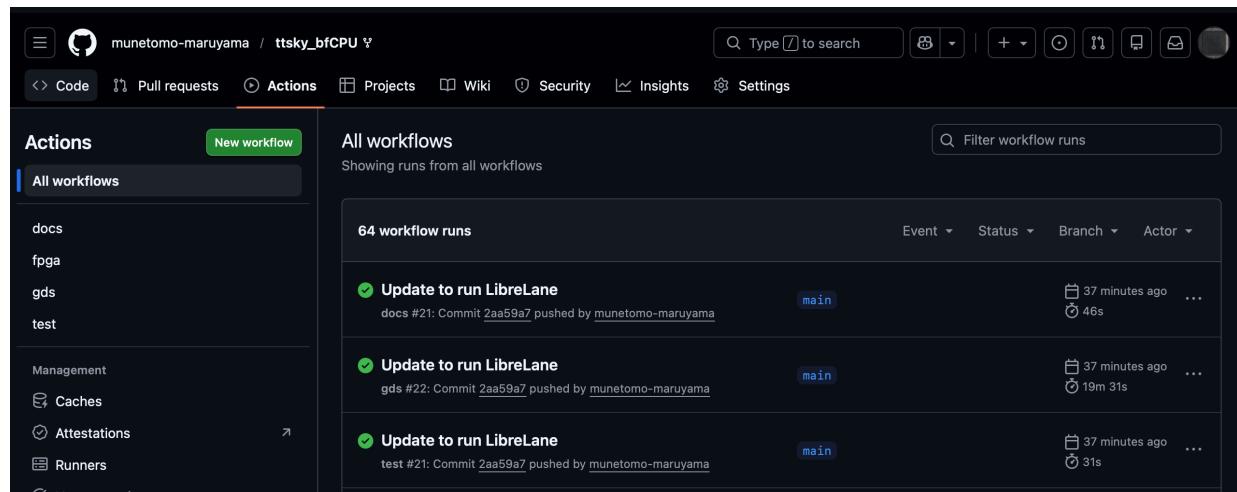


図 2.13: GitHub リポジトリが更新されると、自動的に LibreLane フローを実行

2.4.11 レイアウト結果を 2D 表示と 3D 表示でチェック

Workflows 内の項目 gds を選択し、viewer summary の欄の Load Summary を選択すると図 2.14(a) に示すようなレイアウトの 2D 表示と、図 2.14(b) に示すようなレイアウトの 3D 表示を見ることができます。3D 表示は、拡大・縮小・回転ができるので、配線層の立体関係などを観測できます。

2.4.12 Tiny Tapeout へ設計データを提出

設計が完了して Tiny Tapeout へ設計データを提出するには、GitHub リポジトリのリモート・サイトのトップ・ページ (README.md が表示されているページ) に図 2.15(a) の記述がありますので、" Submit your design to the next shuttle. " をクリックしてください。図 2.15(b) が表示されます。サイン・インしてログインしてください。受付可能なシャトルが表示されますので (この画面をキャプチャした時点では IHP のシャトルのみ受付中だった)、設計した GitHub レポジトリの URL の入力、必要タイル数や評価ボードの数量の入力、支払い方法の指定など必要な手続きを行ってください。データ提出が受け付けられたら図 2.15(c) のようにチップ内に自分のプロジェクトがアサインされます。この図は bfCPU ではありませんので占有しているタイルは 1x1 です。プロジェクトが増えるにしたがって、自分のアサイン位置が変わっていきます。シャトル試作が始まり最終的な位置が確定すれば、チップ完成後の実機評価の時にそのアドレス番号にしたがってプロジェクト選択信号を操作して自分の回路を動作させることができます。

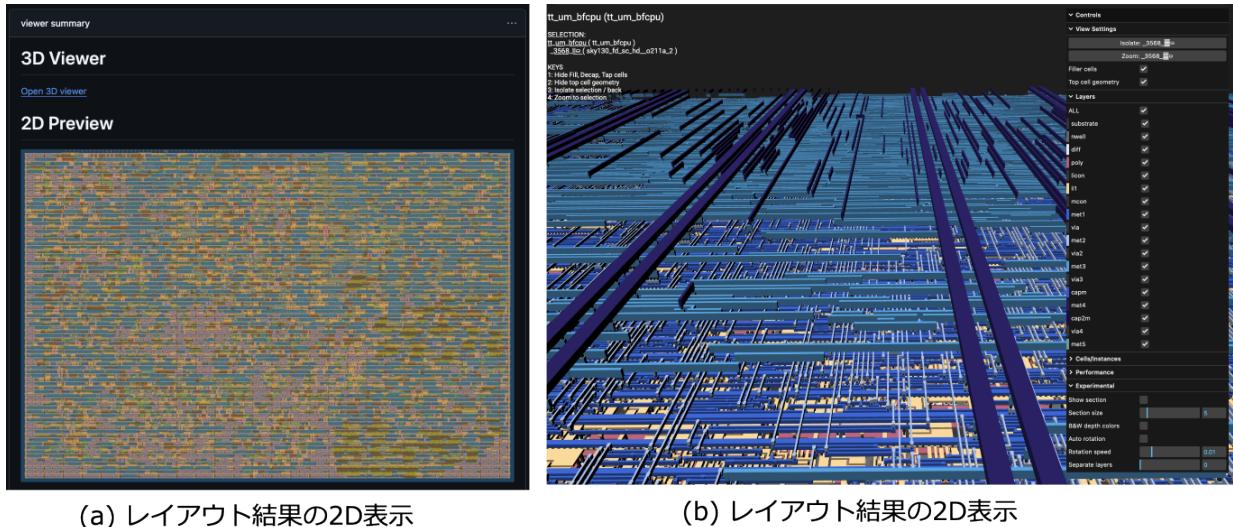


図 2.14: レイアウト結果の 2D/3D 表示

2.5 Tiny Tapeout の活用

Tiny Tapeout で設計できる回路は、面積に制約があり、また端子本数も限られているので、小規模～中規模程度のものに限られでしょう。そのため逆に、半導体設計初心者の設計入門編としてはちょうどいいプラットフォームになります。さらに、趣味や研究として、この制約の中で色々工夫したり、全く新たな回路方式をシリコン上で実証するなど、様々な活用方法があると思います。色々なアイディアで挑戦してみてはいかがでしょうか？

What next?

- Submit your design to the next shuttle.
- Edit this README and explain your design, how it works, and how to test it.
- Share your project on your social network of choice:
 - LinkedIn [#tinytapeout @TinyTapeout](#)
 - Mastodon [#tinytapeout @matthewvenn](#)
 - X (formerly Twitter) [#tinytapeout @tinytapeout](#)
 - Bluesky [@tinytapeout.com](#)

(a) 設計データを提出へ

Tiny Tapeout

Tiny Tapeout IHP 26a 46 days, 07:03:55

Your allocations
You don't have any space allocations on Tiny Tapeout IHP 26a yet.
[PREPURCHASE SPACE ON TINY TAPEOUT IHP 26A](#)

Your projects
You don't have any projects on Tiny Tapeout IHP 26a yet.
[CREATE A NEW PROJECT](#) [VIEW ALL PROJECTS](#)

Past Shuttles

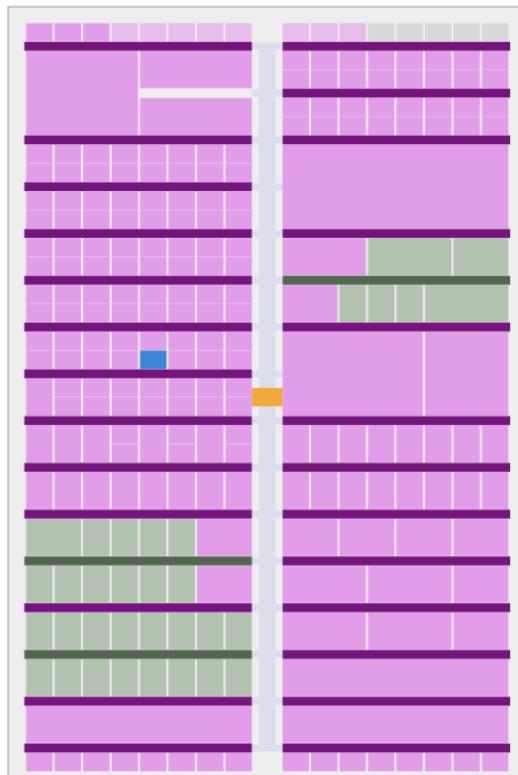
Shuttle/Project	Tiles	Status
TTSKY25a MCS-4 4004 CPU	1x1	Submitted

Copyright (C) 2023-2025, Tiny Tapeout LTD. Revision [82dcc47](#) built at 2026-02-04T13:17:24Z.

(b) 設計データ提出サイト

Project Location on ShuttleMux address: **39** *

The project is highlighted in blue on the map below:



* The project's mux address can change as long as the shuttle is open for submissions.

(b) 設計データをチップ内にアサイン

図 2.15: 設計データを提出

付録 A

論理設計と System Verilog 入門

自分の思い描くシステムを構築するには、ハードウェアとソフトウェアの知識と経験が必要です。特にデジタル機能は、汎用マイコンのソフトウェアで実現できる範囲なら簡単ですが、特殊で高性能かつ高機能な論理機能が必要になるケースだと専用ロジックの設計が不可欠です。この中には CPU コアの設計も含まれるでしょう。最近では、RISC-V コアや生成 AI 处理用の高性能コアなどを独自設計する機会も増えていると思います。そこで本章では、これら論理設計の基礎と、論理設計作業をある程度抽象化してくれるハードウェア記述言語 System Verilog の基礎を学びます。次章で、具体的な CPU の設計事例について学びます。すでに、論理設計や System Verilog の基礎をご存知の方は本章を読み飛ばしていただいて構いません。

圓山 宗智 (munetomo@ishi-kai.org)

A.1 論理設計とは？

A.1.1 論理設計でやること

まずは論理設計とはなんぞや、から説明します。図 A.1 にそのイメージを示します。また、論理機能モジュールの例を表 A.1 に示します。実現したいデジタル機能について、まず入出力インターフェースを明確に定義し、そして内部機能仕様を明確に矛盾なく定義します。矛盾のない入出力インターフェースや内部仕様を早い段階でしっかりロジカルに規定することがとても重要で、ここでの失敗による手戻りは全体の設計工数に多大な悪影響を及ぼします。

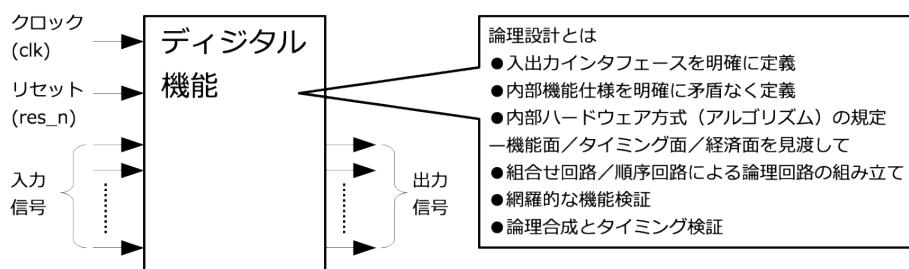


図 A.1: 論理設計とは何か？

A.1.2 仕様が決まった後の論理設計の大まかな流れ

次に、内部ハードウェアの実現方式（アルゴリズム）を規定します。ここが最も頭を使うところで、機能面、タイミング面、経済面（ハードウェア量、消費電力）を見渡しながら、最適な方式を決定します。その後、実現方

表 A.1: 論理機能モジュールの例

設計対象モジュール	入力信号	出力信号
CPU	データ・バス(リード) 割込み信号	メモリ・アクセス・ストローブ信号 アドレス・バス データ・バス(ライト)
シリアル通信 UART	受信データ(RXD) モデム制御信号(CTS, DCD, DSR)	送信データ(TXD) モデム制御信号(RTS, DTR)
タイマ	カウント・クロック インプット・キャプチャ入力 位相係数入力	アウトプット・コンペア出力 PWM出力

式に従って組合せ回路や順序回路を駆使して詳細な論理記述を(例えば後述する RTL レベルなどで)組み立て、網羅的にしっかり機能検証します。実際の論理ゲートに落とし込む論理合成を行い、タイミングに問題ないか検証します。この手順通りストレートに事が進む事はなく、実際は前後の工程を行き来しながら設計を進めています。

A.2 論理設計の抽象度

A.2.1 抽象度とは?

一般的なソフトウェア開発を例にして考えてみると、抽象度が最も低いレベルは、ROM に書込むビット列です。今時、ROM に仕込むプログラムをいきなり 16 進数で書く人はいないでしょうが、マイコンの黎明期にはデータ・バスに繋がった 2 進数トグル・スイッチを使って直接 RAM 内に CPU の機械語プログラムを書込んだりしました。十数バイトくらいの IPL(Initial Program Loader) をものすごい勢いで入力する名人もいました。まあ、この手の話は酒の席くらいにしておきましょう。8080 や 6809 の機械語で会話し始めるおばさまとおじさまがどこからともなく寄ってきますので。

機械語より抽象度を上げていく順に、アセンブラー、C 言語(手続き型高級言語)、オブジェクト指向言語、…という感じになるでしょう。ライブラリやクラスなどで抽象度を上げることで、生産性も上がります。

A.2.2 論理設計における抽象度

論理設計もソフトウェア同様にいくつかの抽象度レベルがあります。表 A.2 にその例を示します。最も抽象度が低いレベルは、回路図上で論理ゲートを組み上げていく設計です。これは LSI 内のアナログ機能に含まれる小規模な論理ブロックや、基板上のロジック回路などでは今でも使われる手法です。これでも生の MOS ドラフトを扱わないので少し抽象度は高めです。論理ゲートから 1 段階高い抽象レベルは RTL(Register Transfer Level) です。おそらく現時点で論理設計の抽象度としてはこの RTL レベルが最も普及していると思います。RTL は内部回路のレジスタだけは明確に定義して、そのレジスタ間の論理をブール式や条件式などで抽象化する記述方法であり、Verilog HDL、SystemVerilog、VHDL といったハードウェア記述言語で表現できます。論理合成ツールを使えば RTL から論理ゲートへ変換でき、現在では非常に効率のよい合成結果が得られるようになっています。RTL 記述は抽象度が高いと言っても、そこから論理ゲートで構成される回路をほぼ類推できるので、テキスト・エディタで記述できる便利な回路図のような感じです。本書での論理設計はこの RTL レベルで行います。

表 A.2: 論理設計の抽象度

名前	モデル名	内容	時間概念	動作合成	言語
システム レベル	TLM: Transaction Level Model	●データの流れだけを定義 ●インターフェースや制御信号は具体化しない ●ハード／ソフト協調検証に有効	無し(アントイムド)、 有り(タイムド) 両方とも記述可能	発展途上	System Cなど
動作レベル	BCA: Bus Cycle Accurate	●入出力インターフェースを具体的に定義 ●内部機能は時間概念なく記述可能	有り: 入出力のみ	可能	System Cなど
RTLレベル	RTL: Register Transfer Level	●入出力も内部機能も共に詳細に記述 ●具体的に内部回路のレジスタを定義 ●レジスタ間の論理は抽象的に記述	有り: 論理全体	可能	System Verilog Verilog HDL VHDL
論理ゲート レベル	—	●論理ゲートで詳細に記述	有り: 論理全体	—	System Verilog Verilog HDL VHDL EDIF

A.2.3 より高い抽象度

RTL より上位の抽象度としては動作レベル (BCA : Bus Cycle Accurate) やシステム・レベル (TLM : Transaction Level Model) があります。言語としては System C などで記述でき、ソフトウェアの C 言語との協調検証ができるため、設計工程の上流段階でシステム検証を行うことができる特長があります。しかし抽象度が上がるほど、論理機能ブロック内の時間 (クロック) 概念が薄くなるので、論理合成したときの結果には注意が必要です。例えば、動画処理などの最適なパイプライン構成や DDR4/DDR5 メモリをアクセスするときの無駄のないタイミング設計を行うなど、ぎりぎりの性能を引き出す設計では、現段階では RTL 設計がベストと筆者は感じていますが、高い抽象度からの動作合成がさらに進化して、論理設計の生産性が飛躍的に向上することを期待しています。生成 AI による自動設計も始まっています。今後、楽しみな分野です。

A.3 ASIC と FPGA の違い

A.3.1 ASIC って何？

スマホやタブレット PC に入っている ARM Cortex-Ax を内蔵したプロセッサなどは ASIC(Application Specific Integrated Circuit) あるいは SoC (System on a Chip) と呼びます。その中は図 A.2(a) のように、CPU や周辺機能などの論理回路に加え、内蔵メモリ、A/D 変換器などのアナログ・モジュール、発振器など、さまざまな機能モジュールが詰まっています。性能、消費電力、コストを最適化することができる一方で、1枚のシリコンに固定化されているので一旦作ってしまうと機能変更できません。また昨今の ASIC は開発費用が極めて高価であり、レガシー・プロセスで数千万円単位から、最先端プロセスで数百億円もしくはそれ以上もかかるのです！

A.3.2 FPGA って何？

FPGA(Field Programmable Gate Array) の構造を図 A.2(b) に示します (Altera 社 MAX 10 の例)。内部にはたくさんの LAB(Logic Array Block) が規則的に並んでいます。LAB はさらに複数の LE(Logic Element) から構成されていて、LE の内部にはルックアップ・テーブル(真理値表としてのメモリ = 組合せ回路) とフリップ・フロップが格納されていて、簡単な論理機能を実現できるようになっています。LAB の間はインターフェクト配線が縦横に走っており相互結線できるようになっています。LAB の内部機能やインターフェクト結線方法は全て外部からプログラム(コンフィグレーション) できますので、FPGA ひとつで任意の論理機能を実現できるのです。

一般的な FPGA のコンフィグレーション情報は内部の SRAM に記憶されますので、電源を落とすと消えてしまいます。このため、電源投入時に外部に接続したシリアル ROM かまたは内蔵する FLASH メモリからコンフィグレーション情報を内部の SRAM に転送する機能を持っています。

FPGA の内部には LAB だけでなく、専用の DSP(Digital Signal Processor) 機能(乗算器、積和演算器、シフタなど)やプロック RAM を持つており、LAB の消費を防ぐとともに性能も向上させています。さらに最新の FPGA の中には、Cortex-Ax プロセッサやアナログ機能をそれぞれ専用ハードとして内蔵したものも出てきており、だんだん ASIC に近づいているようです。

多くの FPGA は、無償のツールで開発できますので、初心者でも気楽に扱えるように考慮されています。

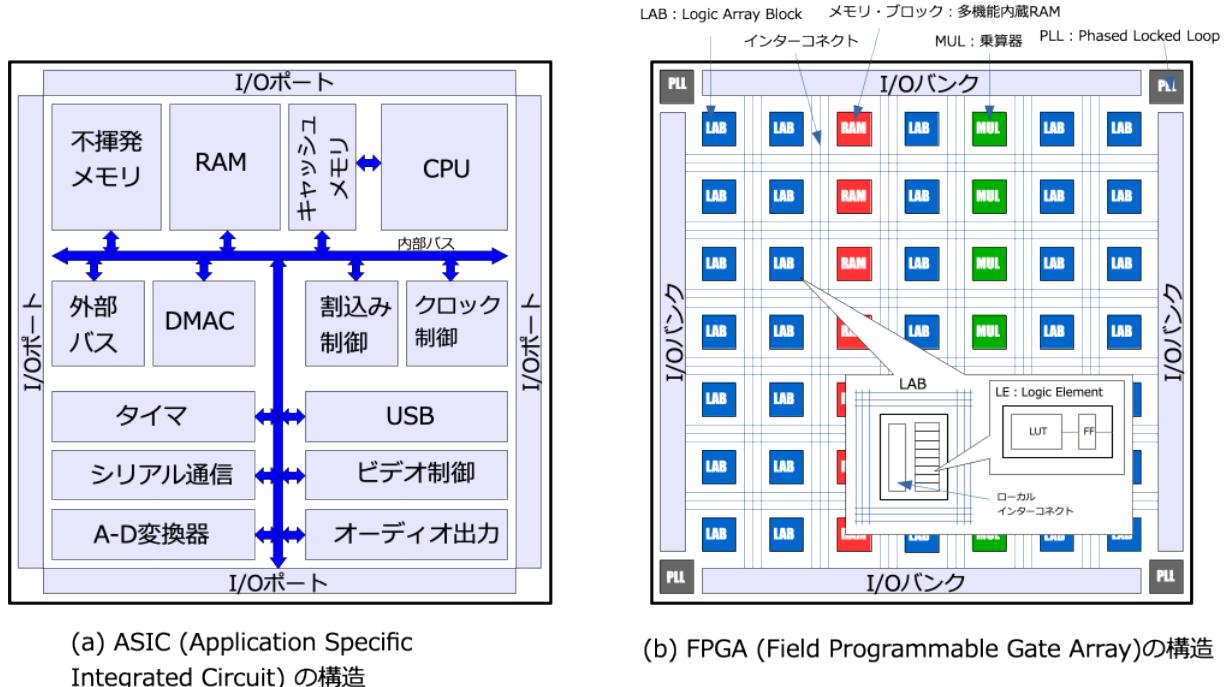


図 A.2: ASIC と FPGA の違い

A.3.3 ASIC と FPGA の設計フロー

ASIC と FPGA のそれぞれについて、論理設計を RTL レベルで進める場合の設計フローの例を図 A.3 に示します。RTL を記述するまでは両者ともほとんど同じです。

ASIC の設計フロー

ASIC を設計する場合、論理合成以降、実デバイスへのインプリメンテーションまで、多種多様な EDA(Electronic Design Automation) ツールを駆使して時間をかけた最適化設計を行いますので、設計工数はかなり多くなります。

以下は、ASIC を設計するために必要な工程の一例です。

1. 仕様設計：実現したい機能や特性を定義します。CPU コアなら、アーキテクチャ、命令セット、外部バスの信号やタイミングなどを明確に定義します。
2. 方式設計：ここが論理設計の本質です。内部のマイクロ・アーキテクチャの決定、ブロック分割、データバスの機能プロック図、制御用ステート遷移の定義、詳細なタイミング・チャートを記述して、頭とドキュメントの上で設計をクローズさせます。
3. 詳細論理設計：SystemVerilog などのハードウェア記述言語 (HDL : Hardware Description Language) を使用して論理を記述します。シリコンになる論理回路プロックは、レジスタ転送レベル (RTL) で記述することが多く、テストベンチなど一部はシステム・レベルや動作レベルで記述します。論理記述ができたら、論理シミュレーション・ツールにより想定通り動作するかどうか機能検証をしっかり行います。次

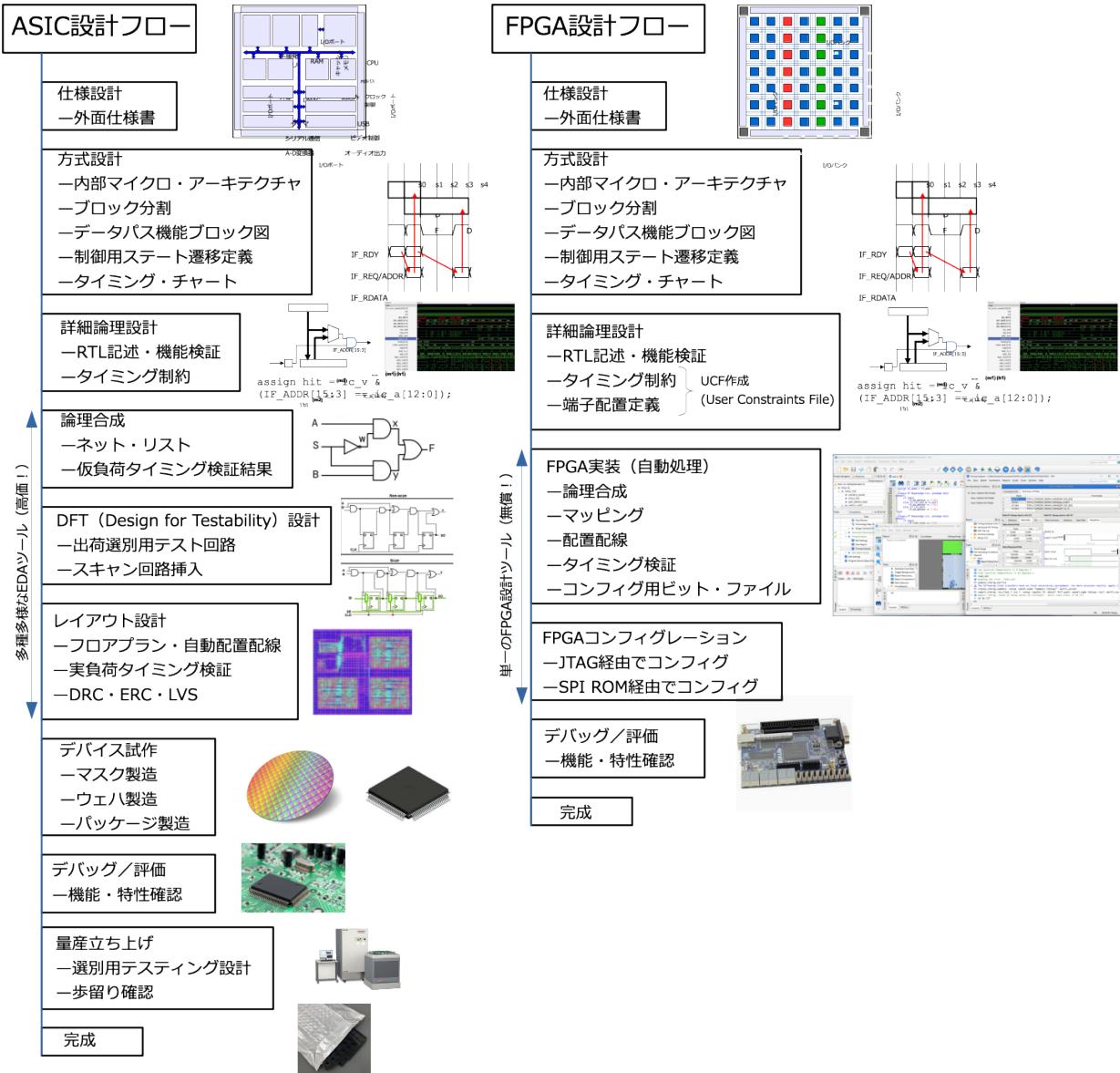


図 A.3: ASIC と FPGA の設計フロー

の論理合成に進む前に、必要な動作周波数や入出力信号の遅延時間を抑えるために、タイミング制約を作成します。

- 論理合成：論理合成ツールを使って、RTL 記述をゲート回路として合成し、ネット・リストを作成します。合成はタイミング制約に基づいて最適化しながら行い、暫定的な仮負荷ベースのタイミング検証結果を出力します。タイミングが満たせない場合は、必要に応じて詳細論理設計に戻り、論理段数の削減などの検討を行います。
- DFT(Design for Testability) 設計：詳細論理設計や論理合成の工程の中で、出荷選別用のテスト回路やスキャン回路を挿入します。ASIC は、内部に億単位のトランジスタがあるので、外部端子は、数十から数千本しかなく、その外部端子から内部のトランジスタや配線の出来栄えを検査する必要があり、検査専用の回路を追加挿入して故障検出率を向上させています。
- レイアウト設計：ここでは、回路を実際のウェハ製造に使用するフォトマスクのレイアウト・パターンを設計します。まず、各回路ブロックをどのようにチップ内に配置するか最適なフロアプランを検討します。次に自動配置配線ツールが、各論理素子を配置しそれぞれの間を配線します。この工程を P&R(Place

and Route) とも言います。物理的なレイアウトができたら、実配線から負荷(配線抵抗、配線容量)を抽出してタイミング検証を行います。最後に、パターンの図形的なルールに合致しているかどうかを DRC(Design Rule Check)で、電気的なルールに合致しているかを ERC(Electrical Rule Check)で、回路とレイアウトが一致しているかを LVS(Layout versus Schematics)で、それぞれ検証します。問題なければ、レイアウト・パターン(GDS-II ファイル)をマスク・メーカーまたはファブに払い出します。

この後、デバイスが試作され、試作品の機能・特性・信頼性を評価し、問題なければ量産開始します。

FPGA の設計フロー

一方、FPGA では、論理合成以降の作業は FPGA メーカーが提供するツール 1 本で済んでしまうことが多いです。多くの FPGA メーカーは、無償の設計ツールを提供してくれており、ほとんど全ての作業ができてしまいます。論理シミュレーションから論理合成、マッピングや配置配線、さらに FPGA のコンフィグレーション情報(ビット・ファイル)の生成まで開発工程一式をサポートしています。

FPGA の開発フローは ASIC と比べるとワン・プッシュで一連の工程を簡単に完了させることができます。

A.4 組み合わせ回路

A.4.1 論理設計の基本は組合せ回路

ここからは、具体的に論理回路を学んでいきましょう。論理設計の基本は組合せ回路です。組合せ回路とは、簡単にいえば、入力されたデジタル信号レベルを加工・変換してすみやかに出力に伝搬させるものです。

A.4.2 組合せ回路の代表=論理ゲート

組合せ回路で代表的なものは図 A.4(a) に示すような論理ゲートを組み合わせたものです。論理ゲート自体はおなじみのものばかりだと思います。ASIC の中で使われる論理ゲートはさらに複雑に機能を組み合わせた複合ゲートが使われます。ここでは記しませんが、論理合成ツールによって合成功率が上がる複合論理ゲートがたくさん考案されています(例えば、2 個の 2 入力 OR の出力を 1 個の 2 入力 NAND に接続したものなど)。

昔は論理合成ツールはなく、論理ゲートを人手で直接回路図上に描いていました。論理ゲートを綺麗に書くための定規もあり、入社したての新人に配ったこともあります。AND や OR の形から、論理ゲートを“くらげ”と呼ぶ人がいましたが、個人的にはあまり好きな言葉ではなかったですね。あれはやはり“ゲート”です。

A.4.3 その他の組合せ回路

その他の組合せ回路の種類としては、加算器、乗算器、ルックアップ・テーブル(真理値表)、ROM などがあります。基本的には、いずれも複数の入力信号を複数の出力信号に変換するものです。入力信号が N 本あれば、 2^N 通りの出力信号の組合せが生じます。

A.4.4 組合せ回路のタイミングは遅延のみ

組合せ回路のタイミングは図 A.4(b) に示すように、入力信号の変化に対して遅延時間 t_d 後に出力信号が変化します。この遅延時間 t_d は組合せ回路の複雑さ(論理ゲート段数の多さ)、使用するトランジスタ特性、配線距離、配線負荷容量などに依存します。条件によっては遅延時間がかなり短くなることもあるので、組合せ回路の遅延時間には幅(min と max)があることを意識してください。

A.4.5 ハミング距離とハザード

複数の信号が変化するとき、変化前と変化後で値が異なる信号の本数をハミング距離といいます。例えば 4 本

の信号があつて、(0000) (0001) と変化する場合のハミング距離は 1、(0001) (0111) の場合は 2、(0111) (1000) の場合は 4 となります。複数の信号が同タイミングに変化する場合、理想的な同時変化はありえなくて、微妙に時間がずれて変化していきます。この微妙にズレて変化していく段階数の最大値がハミング距離に対応します。

組み合わせ回路に入力した複数の信号が変化する時のハミング距離が 2 以上の場合は、その組み合わせ回路の出力信号変化時にハザード(ひげ)が出ることがあります。このハザードについては、遅延時間さえ間に合えば出でても問題なしとする考え方と、MOS 回路のトグル回数が増え消費電力が増大することを防止する、などのためできるだけ避けたいという考え方があります。設計対象ごとのポリシーとしてハザードの扱いを決めて設計しますが、超低消費電力化したい場合や組合せ回路の出力をクロックとして使うような特殊な場合を除き、RTL 記述を普通に論理合成する場合はハザードが発生してもかまわない、と考えることが多いです。

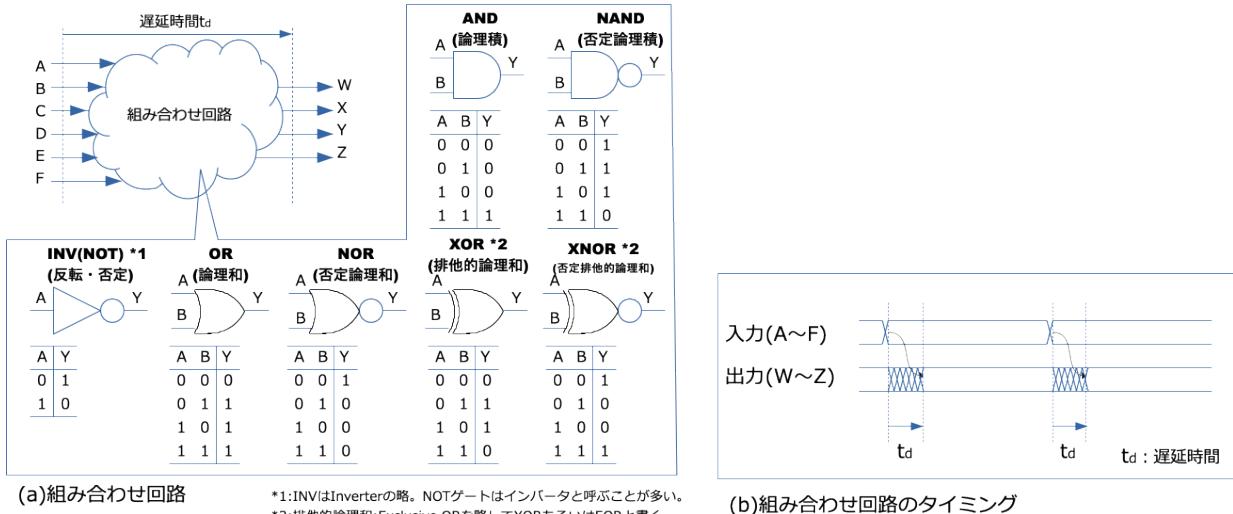


図 A.4: 論理ゲートと組合せ回路

A.4.6 正論理と負論理

デジタル回路では 0 と 1 の信号で処理が進むと言いますが、この 0 と 1 とはどういう意味でしょうか？まず物理的な信号レベルは、0 の信号は LOW レベル、1 の信号は HIGH レベルに対応させるのが通常です。そしてその信号の意味付けについては、通常、信号の 0(LOW レベル) は“偽”“ディスエーブル”“無効”などを意味し、1(HIGH レベル) は“真”“イネーブル”“有効”などを意味します。このような意味付けをした信号を正論理といいます。逆に、信号の 0(LOW レベル) に“真”“イネーブル”“有効”などを、信号の 1(HIGH レベル) に“偽”“ディスエーブル”“無効”などを意味させたものを負論理といいます。

回路図上で、ゲートや機能ブロックにおける負論理信号の入力 / 出力の部分には小さな丸印()を書いてわかりやすくします。原則として負論理で出力された信号は、負論理として入力します(受け取ります)。例えば、論理積の AND ゲートは、入力信号は正論理で出力信号も正論理です。一方、否定論理積の NAND ゲートは入力信号は正論理ですが出力信号は負論理です。入力がともに 1(真) のときにその AND(論理積) 結果としては真になりますが出力信号は負論理なので 0 を出力します。

インバータ(NOT ゲート)は正論理と負論理の変換に使います。

A.4.7 ド・モルガンの法則

数理論理学にド・モルガンの法則というものがあります。図 A.5(a)(b) にその論理式を示します。出力負論理の AND ゲート(NAND ゲート)を入力負論理の OR ゲートに変換したり、出力負論理の OR ゲート(NOR

ゲート) を入力負論理の AND ゲートに変換できるものです。

ド・モルガンの法則を使うと図 A.5(c) に示すように、複雑なゲートを単純なゲートに頭の中で変換することができます。この図では、4 入力 AND ゲートを 3 つの 2 入力 AND ゲートに変換し、途中の信号を負論理にして前段を負論理出力の AND(NAND)、後段を負論理入力の AND に置き換えて、最後にド・モルガンの法則で後段のゲートを 2 入力 NOR に変換しています。

今は、論理合成ツールの性能が向上したので、ド・モルガンの法則を使ってゲートの組合せ方を最適化することに人間が取り組む機会は減っています。論理合成手法としてカルノー図を使う方法もありましたが、これもほとんど使う必要はなくなりました。人手による組み合わせ回路の最適化に興味のある方はインターネットを検索して調べてみてください。

A.4.8 NAND があれば全ての論理回路が組める

ド・モルガンの法則を応用すれば全ての論理回路は 2 入力 NAND ゲートだけで組めます。インバータは 2 入力 NAND の入力端子を接続すればいいし、NOR ゲートは 2 入力 NAND ゲートの入力側と出力側にインバータを挿入すれば実現できます。後述するフリップ・フロップもゲート回路の組合せなので、NAND ゲートだけで作ることができます。このため、TTL IC の 74 シリーズの最初の型番 7400 には NAND ゲート × 4 個が入っているのでしょうか。同様に NOR ゲートだけでも全ての論理回路を実現することが可能です。

NAND ゲートや NOR ゲートだけで論理回路を組み上げることは現在では効率の面から現実的ではありませんが、知識ネタとしてはおもしろい話です。現実の話として、月に行ったアポロ宇宙船に搭載されていたコンピュータは、ほとんど NOR ゲートだけで構成されていました。

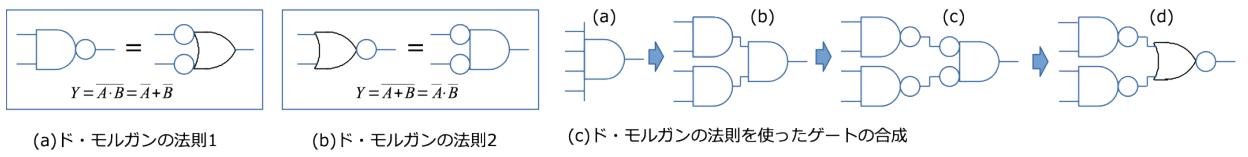


図 A.5: ドモルガンの法則

A.4.9 消費電力が多い NMOS トランジスタによる論理ゲート

論理ゲートの物理的な中身はトランジスタ (MOS FET) でできています。昔の LSI は NMOS トランジスタだけを使っていた時期があります (そのもっと大昔は PMOS だけ使っていた時期もあったが...)。N 型半導体の方が P 型半導体よりも電子の移動度が大きく、同じ面積であれば N 型半導体の方が駆動力があるトランジスタになります。このため N 型チャネルを使う NMOS が好まれました。

その NMOS によるインバタ回路を図 A.6(a) に示します。NMOS のゲート入力が HIGH だと NMOS が ON して出力レベルが LOW になります (図 A.6(b))。NMOS のゲート入力が LOW だと NMOS は OFF になり抵抗 R により後段負荷が充電され出力レベルが HIGH になります (図 A.6(c))。出力が HIGH から LOW に変わるとときは NMOS が低抵抗で出力負荷を放電するので高速ですが、出力が LOW から HIGH に変わるとときは抵抗 R が出力負荷を充電するので低速です。さらに、入力が HIGH レベルのときは電源から抵抗 R と NMOS を通って GND に DC 電流が流れ続けますので消費電力が増える問題があります。抵抗 R の値は速度と消費電力のトレードオフで決めることになりますが、一般に NMOS 型の LSI は非常に消費電力が大きくてそのパッケージ表面がチンチンに熱くなったり記憶があります。

A.4.10 消費電力が少ない CMOS トランジスタによる論理ゲート

現在のデジタル IC は CMOS(Complementary MOS) が主流です。CMOS によるインバタ回路を図 A.6(d) に示します。NMOS と PMOS を組み合わせて相補的 (Complementary) に動作することから CMOS 回路と呼

びます。図 A.6(e)(f) に示すように、LOW 出力時は NMOS がドライブし、HIGH 出力時は PMOS がドライブします。NMOS 回路と違い、常時流れる DC 電流はありませんので、NMOS 回路より消費電力は減り、また PMOS 側の電子の移動度と面積を最適化することで速度も高速化できています。しかし、HIGH → LOW 切替時は、PMOS と NMOS が同時に ON することによる貫通電流や後段負荷の充放電電流が一瞬流れますので、スイッチング頻度（信号周波数）が高くなると消費電流が増えます。

なお、最近のファイン・プロセス（ゲート長が 65nm、40nm、28nm、22nm、16nm、あるいはそれ以下）では NMOS や PMOS が OFF 状態でも電流がかなりリーキしますので、動作していないときの電力（スタティック電力）が無視できない問題になってきています。

NAND ゲートや NOR ゲートを CMOS 回路で書いた例を図 A.7 に示します。どうしても PMOS の方が速度的には不利なので NOR ゲートのように PMOS が縦積みになっているゲートは駆動力が稼ぎにくい傾向があります。OR 系のゲートで多くの負荷をドライブすることは避けたほうがベターですが、遅延時間と面積の制約条件を与えて論理合成ツールにおまかせしてしまうことが多いです。

A.4.11 イネーブル信号やストローブ信号はなぜ負論理？

RAM のチップ・イネーブル信号（/CE）やライト・ストローブ信号（/WE）は負論理なことが一般的ですが、この理由は NMOS 時代の名残です。NMOS 回路では、HIGH から LOW に変化する方が高速なことと、非選択時（論理値 1）のときは NMOS が OFF で DC 電流は流れないことから、負論理が好まれたようです。

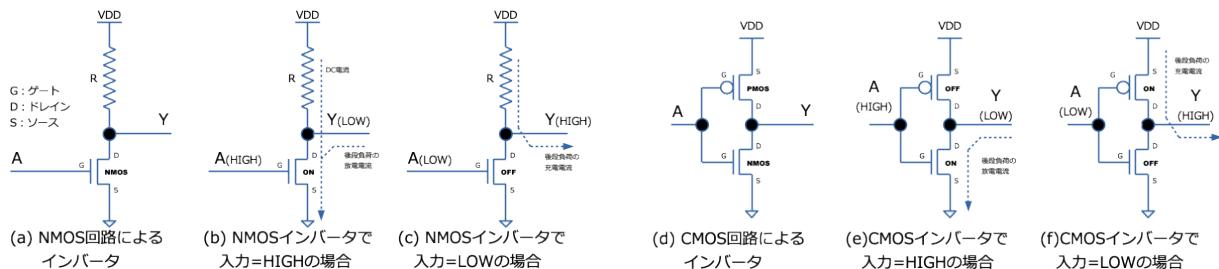


図 A.6: NMOS 回路と CMOS 回路

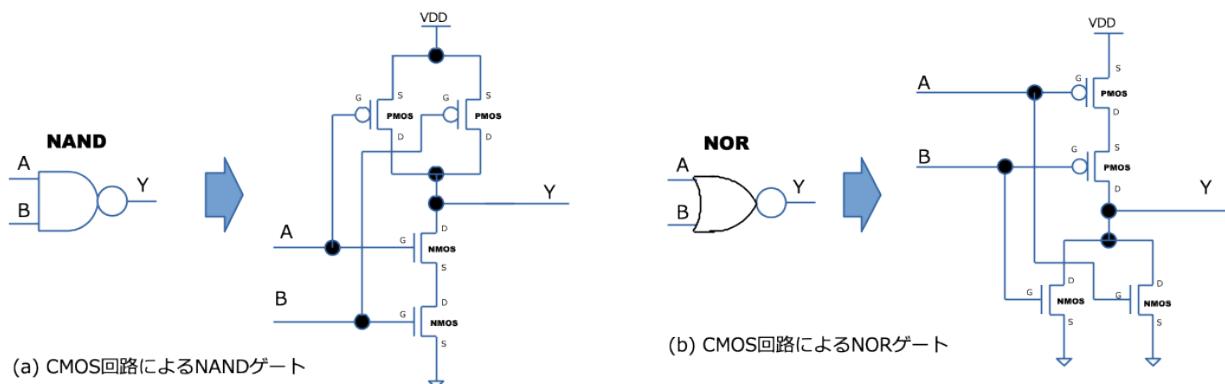


図 A.7: CMOS で書いた論理ゲート

A.4.12 遅延時間の種類

組み合せ回路（および単純な電線）を通る信号は遅延しますが、この遅延には図 A.8 に示す様に慣性遅延と伝搬遅延の 2 種類があります。

慣性遅延は、回路の遅延時間より短いパルスが伝わらない遅延です。ローパス・フィルタを通したイメージで

あり、物理的にはこの遅延動作が一般的でしょう。

伝搬遅延は、入力信号がそのままの形で出力側に伝搬する遅延特性をいいます。特殊なディレイ・ラインなどをモデル化するときに使います。

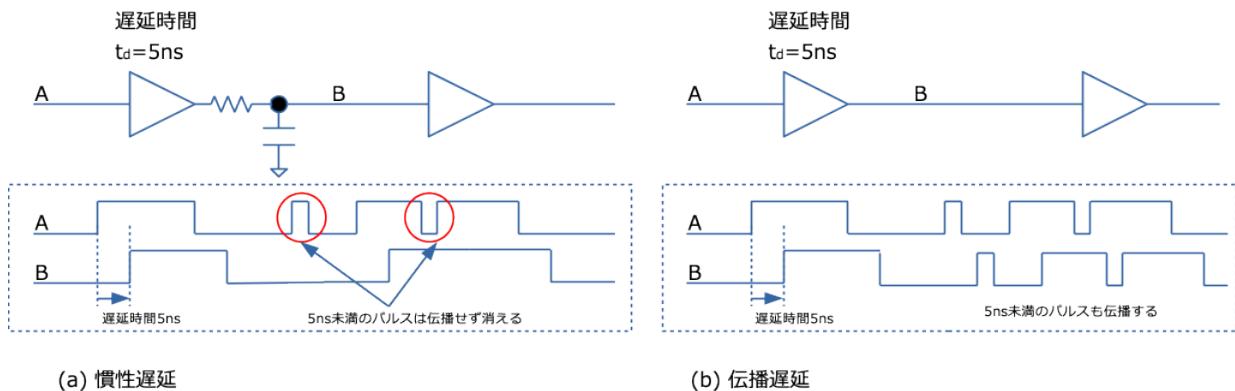


図 A.8: 慣性遅延と伝搬遅延

A.5 順序回路と D-フリップ・フロップ

A.5.1 順序回路とは？

デジタル機能モジュールは、時間経過と絡めた複雑なシーケンスなどを扱う必要があり、組合せ回路だけでは実現できません。このシーケンスを扱う回路が順序回路になります。順序回路は、クロック信号を基準にして動作するものです。ここに必要な基本要素が D フリップ・フロップ (以下 D-F/F と略す) です。

フリップ・フロップには他にも種類 (T フリップ・フロップ、JK フリップ・フロップなど) がありますが、RTL レベル設計で意識することがほとんど無くなってきたので本稿では説明を省略します。

A.5.2 クロックと D-F/F

D-F/F を図 A.9(a) に示します。D-F/F は入力信号をクロックで時間を刻んで遅延させて出力するものです。クロックの立ち上りエッジで入力信号 D を取り込んで信号 Q を出力します。Q が変化するのは必ずクロックの立ち上りエッジ (T1, T2, T3, ...) になり、入力 D が変化するだけでは Q には影響しません。D-F/F には正論理出力 (Q) に加えて負論理出力 ($/Q$) を持つものもあります。

一般的に D-F/F は、出力を初期化するためリセット機能付きが使われます。図 A.9(b) は非同期リセット付き D-F/F です。負論理リセット入力 res_n が 0 になるとただちに正論理出力 (Q) が 0 になります。

D-F/F にはリセット入力だけではなく、正論理出力 (Q) を 1 に初期化できるセット信号付きのものもあります。

さらに、リセット入力とセット入力の両方を持つ D-F/F もあります。この場合、リセット入力とセット入力の両方がアクティブになった (アサートした) 期間はリセットが優先され $Q = 0$ にするのが一般ですが、リセット入力とセット入力の両方が同時に非アクティブになった (ネガートした) ときの出力は定まりません (不定値になる) ので注意が必要です。リセット入力とセット入力は最後にネガートしたほうが Q 出力の初期値を決めます。

参考として、リセット入力とセット入力の両方を持つ D-F/F の内部回路を図 A.10 に示します。

なお、D-F/F を使って入力信号をクロックで受けて出力することを、“クロックで叩く”、“クロックで同期化する”などという言葉で表すことが多いです。

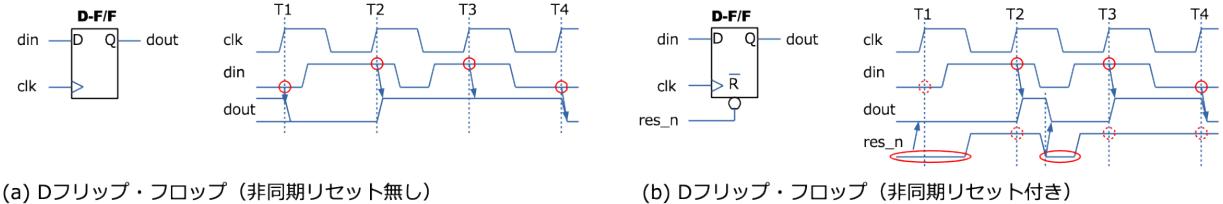


図 A.9: D フリップ・フロップ

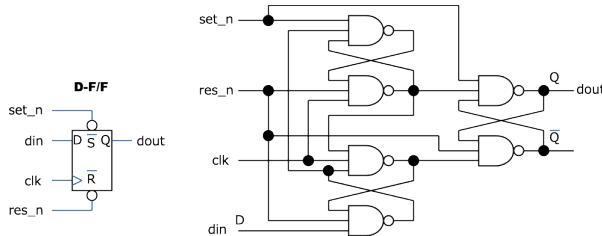


図 A.10: D フリップ・フロップの内部回路

A.5.3 D-F/F のセットアップ時間とホールド時間

組合せ回路のタイミングは遅延時間でしたが、D-F/F でも、クロックの立上がりエッジから出力レベルが定まるまでの遅延時間があります。さらに D-F/F では他にも重要なタイミング規定があります。図 A.11(a) に示すセットアップ時間とホールド時間です。

- **セットアップ時間**: クロックの立上がりに対して、入力信号 D のレベルをどれだけ前に確定させておくかを規定。
- **ホールド時間**: クロックの立上がりに対して、入力信号 D のレベルをどれだけ後ろまで保持しておくかを規定。

論理回路の誤動作を防ぐためには、この D-F/F のセットアップ時間とホールド時間を満足させることが必要です。

A.5.4 D-F/F のセットアップ時間とホールド時間違反したらどうなるか

D-F/Fにおいて、セットアップ時間とホールド時間が違反している場合、図 A.11(b) に示すように、クロックの立上がりエッジからしばらくの間、出力 Q が発振したような状態になり 0/1 が定まりません。この状態をメタ・ステーブル状態と呼び、その持続時間はデバイス特性に依存しますが確率論的であり、長い持続時間ほど発生確率は低いです。メタ・ステーブルの持続時間は採用するプロセスやデバイスごとにガイドラインが規定されていますが、イメージとしては D-F/F の遅延時間の数倍程度でしょう。

A.5.5 非同期信号の受け方

外部から自分のクロック信号と非同期な信号を D-F/F で受け取るときは、かならず初段ではメタ・ステーブル状態が生じますので、図 A.11(c) のように、少なくとも 2 段の D-F/F で受けて、メタ・ステーブル状態になった信号を内部回路に引き回さないようにしなくてはなりません。ただし、メタ・ステーブル状態の持続時間がクロック周期より十分に短いことが前提です。

A.5.6 タイミング設計の基本

論理回路のタイミング設計は、D-F/F におけるセットアップ時間とホールド時間との戦いがほとんど、と言つ

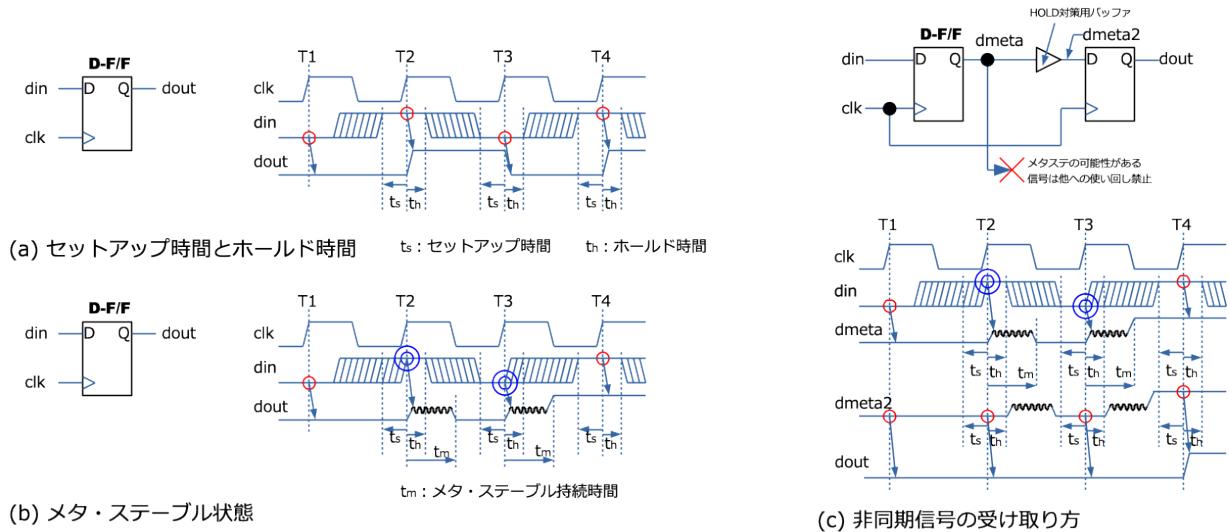


図 A.11: セットアップ時間とホールド時間

ていいでしょう。ここでは、図 A.12(a) に示す D-F/F2 段の回路を題材にしてタイミング設計の基本を説明します。

図 A.12(a) では、入力信号 din_1 を初段の D-F/F(U1) で受けて、その出力 $dout_1$ を組み合わせ回路に通しています。この組み合わせ回路の遅延時間を $td_{_comb}$ とします。この組み合わせ回路の出力 din_2 を後段の D-F/F(U2) で受けて、その出力を $dout_2$ としています。

また図 A.12(a) では、周期 $tcyc$ の入力クロック clk を、U1 のクロック clk_1 と U2 のクロック clk_2 に分配しています。現実にはそれぞれ物理的に経路が異なっているので clk_1 と clk_2 の時間のズレ(スキュー)はゼロではありません。ここでは、大元の clk から clk_1 までの遅延時間を $td_{_clk1}$ 、 clk_2 までの遅延時間を $td_{_clk2}$ とします。

D-F/F のタイミング規定としてセットアップ時間、ホールド時間、クロック立ち上がりからの出力遅延時間がありますが、U1においてはそれぞれ $ts_{_u1}$ 、 $th_{_u1}$ 、 $td_{_u1}$ とし、U2においてはそれぞれ $ts_{_u2}$ 、 $th_{_u2}$ 、 $td_{_u2}$ と定義しておきます。

ここでは、D-F/F の U2 に着目して、そのセットアップ時間とホールド時間を満足する条件を確認してみましょう。この回路の動作タイミングを図 A.12(b) に示します。

A.5.7 セットアップ時間の検証

図 A.12(b)において、初段の U1 が output $dout_1$ を変化させるタイミングは [A](ローンチ : Launch) です。これを組み合わせ回路を経由して後段の U2 で取り込むタイミングが [B](キャプチャ : Capture)になります。[B]の clk_2 の立ち上がりエッジに対して din_2 が U2 のセットアップ時間規定を満足する必要があります。

図 A.12(b) から、ラッチ・タイミング [B] における din_2 のセットアップ時間 $t_{_setup2}$ は下記で表せます。

$$t_{_setup2} = (tcyc + td_{_clk2}) - (td_{_clk1} + td_{_u1} + td_{_comb}) \quad (\text{式 } 1)$$

これが U2 自体に必要なセットアップ時間 $ts_{_u2}$ より大きければ OK です。すなわち下記の関係が必要です。

$$ts_{_u2} \leq (tcyc + td_{_clk2}) - (td_{_clk1} + td_{_u1} + td_{_comb}) \quad (\text{式 } 2)$$

$$tcyc \geq (td_{_u1}) + (td_{_comb}) + (ts_{_u2}) + (td_{_clk1} - td_{_clk2}) \quad (\text{式 } 3)$$

ワースト条件として (式 3) の右辺が大きくなる方向で考えます。一般的に表現すると下記の条件が必要です。

(クロック周期) (D-F/F 自体の最大出力遅延)

+ (D-F/F 間に挿入される組合せ回路の最大遅延)

+ (D-F/F のセットアップ時間)

- + (クロック・スキーの符号付き最大値)
- + (マージン値) (式 4)

マージン値は元のクロックが持つ周波数ジッタなどを含めた絶対値です。周波数ジッタはクロックのエッジ位置を不確実にさせることから、クロック・アンサーテンティ (uncertainty) とも呼びます。

結局、D-F/F 間に挿入される回路の遅延と D-F/F のセットアップ時間がクロックの最小周期 (最大動作周波数) を決めることになります。動作周波数を向上させるには、D-F/F 間に挿入される組合せ回路の遅延時間を短くする必要があります。

A.5.8 ホールド時間の検証

図 A.12(b)において、初段の U1 がローンチ・タイミング [A] で出力 dout1 の値を 0 から 1 に変化させます。後段の U2 の同位相のラッチ・タイミング [C] では U2 のホールド時間を満足するように、入力 din2 の値がそのまま維持されている必要があります。図 A.12(b) からラッチ・タイミング [C] における din2 のホールド時間 t_{hold2} は下記で表せます。

$$t_{hold2} = (td_{clk1} + td_{u1} + td_{comb}) - (td_{clk2}) \quad (\text{式 } 5)$$

これが U2 自体に必要なホールド時間 th_{u2} より大きければ OK です。すなわち下記の関係が必要です。

$$th_{u2} \leq (td_{clk1} + td_{u1} + td_{comb}) - (td_{clk2}) \quad (\text{式 } 6)$$

$$th_{u2} \geq (td_{u1}) + (td_{comb}) + (td_{clk1} - td_{clk2}) \quad (\text{式 } 7)$$

ワースト条件として (式 7) の右辺が小さくなる方向で考えます。一般的に表現すると下記の条件が必要です。

(D-F/F のホールド時間) \geq (D-F/F 自体の最小出力遅延)

+ (D-F/F 間に挿入される組合せ回路の最小遅延)

+ (クロック・スキーの符号付き最小値)

- (マージン値) (式 8)

マージン値にはセットアップ時間の検討と同様に元のクロックが持つ周波数ジッタ (クロック・アンサーテンティ)などを含めます。

前項のセットアップ時間は動作周波数を遅くすれば必ず確保できますが、ホールド時間はアナログ的な遅延関係だけで確保する必要があることに注意してください。クロック系統の信号とデータ系統の信号のレーシング (遅延時間の差) によっては動作周波数をどのように設定しても誤動作してしまうことがあります。このため、基本的には (式 8) 内のクロック・スキーまたは D-F/F 間に挿入される組合せ回路の最小遅延を大きくする対策が必要です。しかし、これらの項を大きくすると (式 4) により動作周波数が遅くなってしまいます。このようにセットアップ時間とホールド時間は背反する関係にありますので注意が必要です。

ちなみに、2 段の D-F/F が直結されているケースが最もホールド時間を満足させにくくなります。図 A.12(c) で D-F/F 間にバッファを挿入していたのはホールド対策のためです。

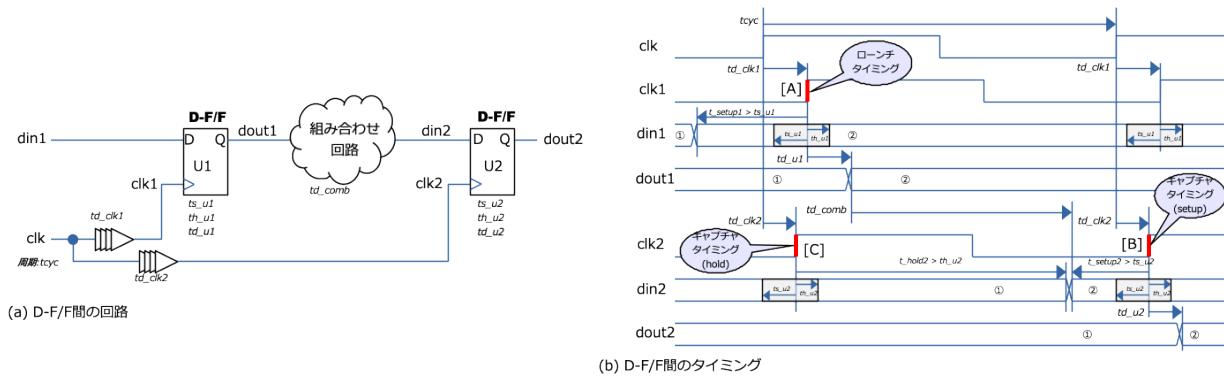
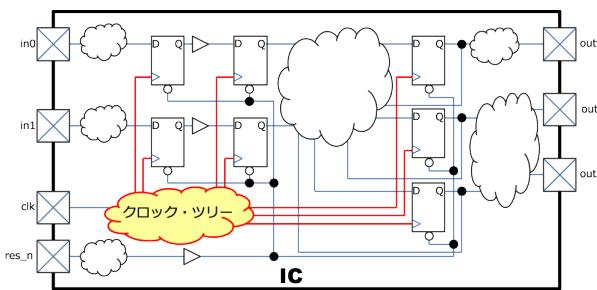


図 A.12: タイミング設計の基本

A.5.9 クロック・ツリー

セットアップ時間の(式4)とホールド時間の(式8)に共通に入っているクロック・スキューレの項は、正負の値を取り、正方向に大きくなるとセットアップ時間を満足できなくなり、負方向に大きくなるとホールド時間を満足できなくなります。このため原則としてクロック・スキューレはゼロ値が最適であり、全D-F/Fに与えるクロックの立上がりエッジの位相がぴったり合わせる必要があります。IC設計ではチップ全体に図A.13のようなクロック・ツリー設計を施します。D-F/F端点でのクロックの位相を、配線長や遅延バッファの挿入などにより合わせ込みます。遅延バッファが多く挿入されると消費電力が増大してしまいますので、物理的なレイアウト含めてコツとワザが必要な工程です。今はよいEDAツールで設計と検証ができるようになりました。賢いEDAツールになると、タイミング調整用のバッファを最小限にするため、あえて個々のD-F/Fのクロックにスキューレを残して、セットアップ時間とホールド時間を効果的に満足させるものもあります。

なお、基本的にFPGAでは、全D-F/Fに入るクロック信号の位相は既に合わせ込み済みになっています。



図A.13: クロック・ツリー

A.5.10 実機デバッグでの苦労を防ぐには

デバイスが完成して実際にボード上で実機デバッグする段になって、どうもうまく動かないときがあります。原因としてセットアップ時間に違反があった場合は、クロック周波数を下げればなんとか動作してくれますので、異常箇所を特定しやすいバグではあります。しかし、ホールド時間に違反がある場合は、クロック周波数をどう変えても動作してくれません。電圧を下げるとき遅延が増えてホールドが満たされて動くことがあります、基本的にはクロック信号とデータ信号とのレーシング(競争)による誤動作なので非常に実機デバッグしにくい性質のものです。

設計段階で、全D-F/Fに関して全動作条件(プロセス、電圧、温度)においてセットアップ時間もホールド時間もすべて満足されていることを検証することは当然なのですが、セットアップ時間は最低限、遅延ワーストになる条件で検証すればかなり安心できます。しかし、ホールド時間はレーシング動作なのでワースト条件を見いだしにくく、全動作条件での検証が必須になります。

A.5.11 RTLレベルで設計する時のD-F/Fタイミングの考え方

一般的に論理回路をRTLレベルで設計する時のD-F/F回路は、図A.14(a)に示したように、共通のクロックを根元に持つ全てのD-F/Fのクロックのスキューレは理想的にゼロ(位相が完全に一致している)と考えます。

データ系の信号の流れを考える時は、図A.14(b)に示したように必ず因果関係を意識した図を書くように心がけてください。個々のD-F/Fのセットアップ時間やホールド時間が満足されているかどうかはRTLレベル設計時にはあまり意識する必要はありません。RTL記述を論理合成して実際のゲート・レベルのネットに変化した後のタイミング検証の段階でEDAツールで網羅的に検証します。

ただし、動作周波数をむやみに落とさないように、D-F/Fの間に挿入する組み合わせ回路の遅延時間を抑える必要があり、RTLレベル段階でも、回路規模や想定ゲート段数が小さくなるような配慮が必要です。

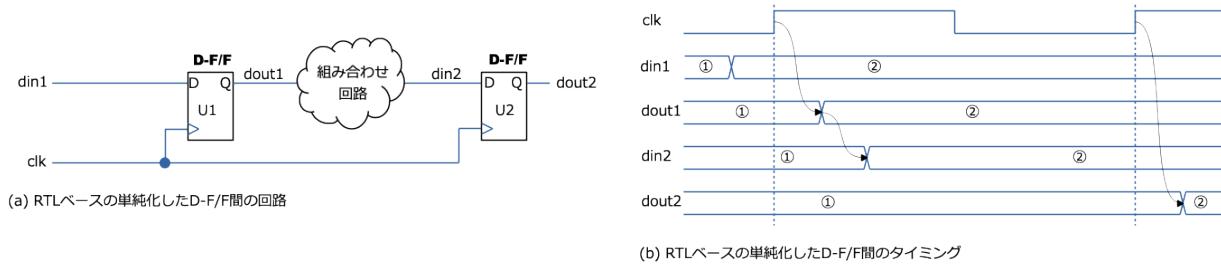


図 A.14: RTL レベルで設計する時の D-F/F 回路

A.5.12 リセットの考え方

D-F/F のリセット方式としては、図 A.15(a) に示す非同期方式があります。負論理のリセット信号 `res_n` がアサートされれば(0になれば)すぐに出力 `Q` がリセットされ `dout` は 0 になります。`res_n` がアサートされている間はクロックの立ち上がりエッジがあっても出力はリセットされたままです。

D-F/F の別のリセット方式としては、図 A.15(b) に示す同期方式があります。この方式はクロックの立ち上がりエッジがきて初めて出力がリセットされます。

ASIC 設計では一般に出荷テスト用の回路 (DFT : Design for Testability) として全ての D-F/F 間に SCAN チェーン回路を仕込みますが、そのときの D-F/F に与えるリセットは非同期式にします。よって IC 設計では、回路の共用化のため非同期リセット方式を使うことが多いです。

FPGA ではすでに D-F/F 周りの回路は仕込まれていますが、その D-F/F も非同期リセット式です。

リセット信号にノイズが乗ったときのことを考えると同期リセット方式の方がノイズ耐性が高いと主張する文献がありますが、外来ノイズはリセット入力部の回路で除去できるし、チップ内部のリセット信号へのクロストーク・ノイズは、シグナル・インテグリティ検証で除去できるものなので、筆者は非同期式リセット方式を使っています。

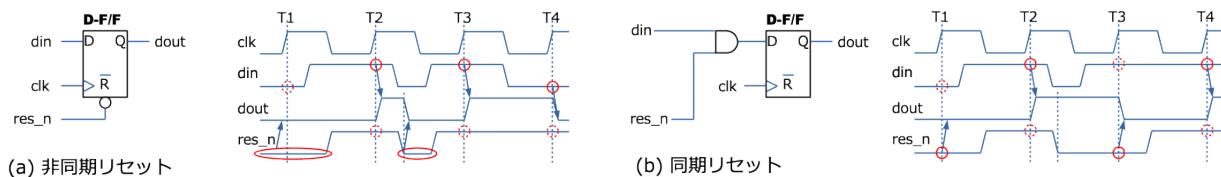


図 A.15: リセットの考え方

A.6 ステート・マシン

A.6.1 ステートマシンは順序回路の代表

ここからは D-F/F を使った具体的な順序回路の代表例としてステートマシンを取り上げます。ステートマシンは入力信号に呼応して複雑なシーケンスで出力信号を生成できる回路です。それでいて、設計手法はとてもすっきりしており、簡単で確実な設計ができる回路方式なのです。

図 A.16(a) にステートマシンの入出力信号の例を示します。動作としては入力信号に応じて、クロックの立ち上がりエッジごとにシーケンスを持った出力信号を生成します。ステートマシンの初期化のためにリセット信号が必ず必要です。

A.6.2 状態遷移図

ステートマシンを設計するとき、まず実現したいシーケンス(順番)に応じて状態遷移図(ステート・ダイアグラム)を書きます。図 A.16(b) にステートマシンの動作シーケンスを表す状態遷移図の例を示します。状態遷移図は例えば以下のようにして書きます。

1. 実現したいシーケンスが持つクロックごとの状態(ステート)を全て洗い出し、必要な数だけ状態を示す楕円を書きます。状態と出力信号が1対1対応している場合は、楕円の中に出力信号のレベルを書くとわかりやすくなります。
2. 状態と状態の間の遷移方法を書きます。入力信号の条件に応じて、同じ状態に居続けたり、他の状態に遷移します。状態間の遷移はクロックの立上がりエッジで行われます。
3. 各状態に固有コードを割り振ります。例えば状態が全部で10個あれば4ビットのコードを割当てます。このようなコード割当てでは、状態遷移時にハミング距離(状態コードの中で変化するビットの個数)が少なくなるように割り振るほうが論理規模が小さくなり、かつ消費電力が小さくなります。

状態の割当て方は他にもいろいろあります。例えば、状態が10個なら、コードは4ビットにして、状態 n に対しては n ビット目だけが1になったコードにします(ワン・ホット割当て)。

なお、この状態へのコード割当ては、論理合成ツールによっては自動的に最適化して再割当てしてくれる場合もあり、もはやその方が論理規模や動作速度が改善することが多いです。

A.6.3 状態遷移の動き

図 A.16(b) の例では、リセット直後のステートマシンの状態は state0 であり、2ビットの入力信号 IN[1:0] が 2'b01(2ビット幅の2進数で01)になればクロックの立ち上がりエッジで状態 state1 に遷移し、IN[1:0] が 2'b01 以外なら状態 state0 に留まる動作をします。出力信号 OUT[2:0] については、状態が state0 のときは 3'b000(3ビット幅の2進数で000)で、状態が state1 のときは 3'b101 になります。その他の状態や状態間の遷移も同様な書き方をします。

このステートマシンの動作タイミング例を図 A.17 に示します。状態遷移図に表現した通りに、入力信号に応じて状態と出力信号が変化していることが確認できると思います。

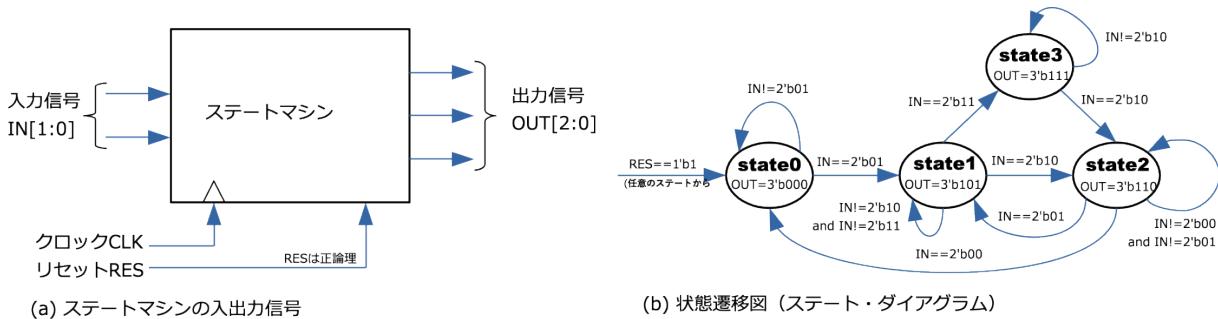


図 A.16: ステートマシンと状態遷移図

A.6.4 ステートマシンの内部論理

図 A.16 に示したステートマシンの内部論理構造を図 A.18 に示します。この例では、状態の数は全部で4個なので、ステートマシン内の状態は2ビットの state[1:0] で表現できます。

状態信号 state[1:0] のコード・アサイン例を表 A.3(a) に示します。この状態信号 state[1:0] が D-F/F の出力になります。状態信号 state[1:0] の初期値は 2'b00 なので、2つの D-F/F の出力値はリセット信号により共に0になるようにします。

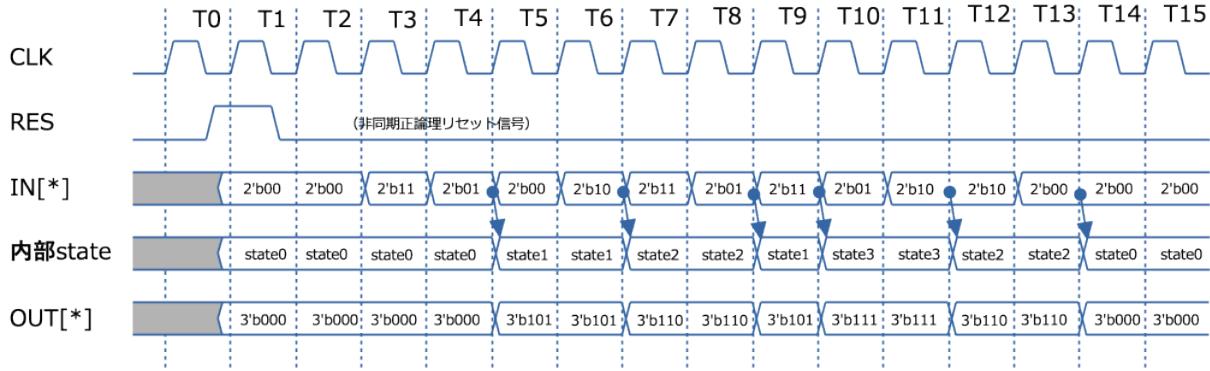


図 A.17: ステートマシンの動作タイミング

現在の状態を次の状態に遷移させるために、現状態 $state[1:0]$ と入力信号 $IN[1:0]$ を組み合わせ回路を経由して次状態 $state_next[1:0]$ を生成します。その $state_next[1:0]$ を先ほどの D-F/F に入力し、クロックの立ち上がりで $state[1:0]$ に移して状態遷移を実現しています。この組み合わせ回路の真理値表を表 A.3(b) に示します。

出力信号 [2:0] は現状態 $state[1:0]$ から別の組み合わせ回路を介して生成します。この真理値表を表 A.3(c) に示します。

基本的にステートマシンの設計は、状態遷移図ができていれば下記の作業を行うことで機械的に完成します。

1. 現状態を表す信号を複数ビットの D-F/F にアサインする。
2. 現状態と入力信号から次状態を生成する組み合わせ回路の真理値表を書く。
3. 現状態から出力信号を生成する組み合わせ回路の真理値表を書く。

これらを RTL 記述上に表現すれば論理合成ツールが最適な論理回路を生成してくれます。ステートマシンを使うと、複雑なシーケンスでも、状態遷移図の上にきちんと表現できれば、簡単に設計することができるのです。さらに、最適化が強力な論理合成ツールを使うと、設計者が考えた状態信号のビット幅やコード・アサインを勝手に変更して、よりゲート数が小さく高速な回路を合成してくれることもあります。

表 A.3: ステートマシン内の真理値表

状態(ステート)	
状態名	$state[1:0]$
state0	2'b00
state1	2'b01
state2	2'b10
state3	2'b11

(a) 状態(ステート)
のコード・アサイン

現状態		出力信号 $OUT[2:0]$
状態名	$state[1:0]$	
state0	2'b00	3'b000
state1	2'b01	3'b101
state2	2'b10	3'b110
state3	2'b11	3'b111

(c) 出力信号の真理値表

現状態		入力信号 $IN[1:0]$	次状態		備考
状態名	$state[1:0]$		状態名	$state_next[1:0]$	
state0	2'b00	2'b00	state0	2'b00	
		2'b01	state1	2'b01	遷移あり
		2'b10	state0	2'b00	
		2'b11			
state1	2'b01	2'b00	state1	2'b01	
		2'b01			
		2'b10	state2	2'b10	遷移あり
		2'b11	state3	2'b11	遷移あり
state2	2'b10	2'b00	state0	2'b00	遷移あり
		2'b01	state1	2'b01	遷移あり
		2'b10	state2	2'b10	
		2'b11			
state3	2'b11	2'b00	state3	2'b11	
		2'b01			
		2'b10	state2	2'b10	遷移あり
		2'b11	state3	2'b11	

(b) 状態遷移の真理値表

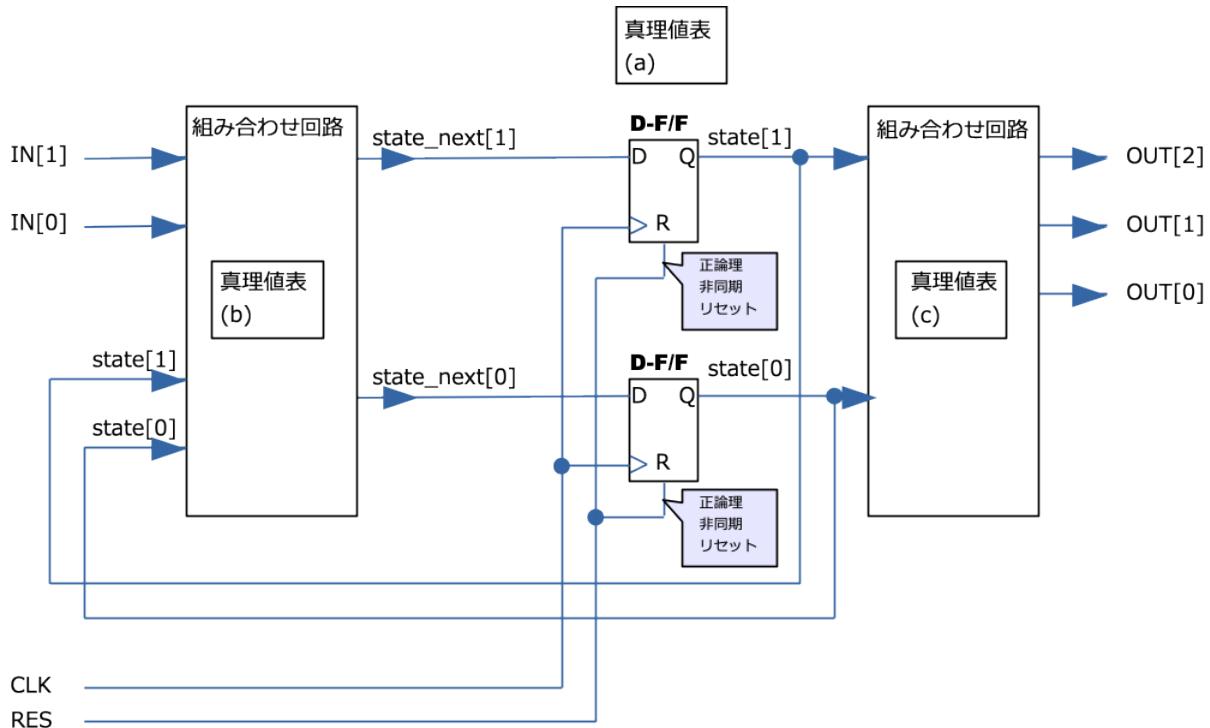


図 A.18: ステートマシンの内部論理

A.6.5 ステートマシンの階層化

ステートマシンを階層化して、別のステートマシンをサブルーチンのように使う方法もあります。あるステートに長時間留まる場合は、カウンタなどで時間を計ってから状態遷移制御させることもあります。

A.6.6 ステートマシンの種類

ステートマシンには図 A.19 に示すような種類があります。

図 A.19(a) は出力が現状態のみで決まる構造を持っておりムーア (Moore) 型ステートマシンといいます。図 A.16 で説明したステートマシンもムーア型です。

一方、図 A.19(b) に示すような、出力が現状態と入力信号から決まる構造のステートマシンをミーリー (Mealy) 型といいます。

ムーア型は出力信号を現状態の D-F/F から生成できるので高速化しやすいですが、状態の個数はミーリー型より多くなる傾向にあります。ミーリー型は次状態と出力を生成する組み合わせ回路を一体化できるので RTL 記述を書きやすい特長を持ちますが、全体の回路規模は大きくなる傾向があります。

ムーア型とミーリー型は相互に等価変換できます。まず、ムーア型からミーリー型への変換は、そもそもムーア型がミーリー型の特殊型（出力を決める組み合わせ回路に入力を入れないだけ）と見なせるので簡単です。逆にミーリー型をムーア型に変換する場合は、ミーリー型では単一の状態でも入力信号に応じて複数の出力を得られるのに対して、これをムーア型にしたときには、入力信号に対応した次状態の種類を増やすことでミーリー型と等価な動作を得られます。ただし、ミーリー型では入力信号の変化があれば直ちに出力信号が変化しますが、ムーア型では状態が遷移して初めて出力信号が変化しますので、ミーリー型をムーア型に変換した場合は細かいタイミングまで完全に同一な動作にはなりません。ステートマシンの入力信号がクロックの立ち上がりの直前で変化するという前提を（もちろん D-F/F のセットアップ時間を満足する条件で）置けば、だいたい等価なタイミングで動作します。

実際のステートマシンの設計をムーア型にするかミーリー型にするかの選択はあまり意識する必要はないで

しょう。自分が設計したいように自由に書いて、あとは論理合成ツールにお任せのパターンが多いようです。

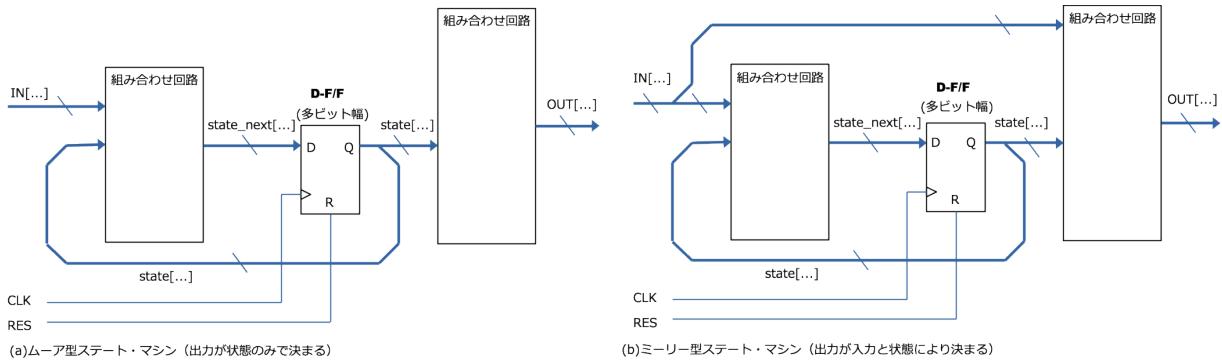


図 A.19: ムーア型ステートマシンとミーリー型ステートマシン

A.7 機能モジュールの論理設計

A.7.1 機能モジュールの構造

ここまでが論理回路の基礎知識です。ここから具体的にデジタル機能モジュールをどのように設計するのかを説明します。図 A.20 に機能モジュールの基本構造の例を示します。CPU であれ、シリアル通信モジュールであれ、どのような機能モジュールも、データバス部と制御部に分けて設計します。このような構成を探ることで設計の見通しがよくなります。

- **データバス部:** データバス部では複雑な制御は行わない。データを流すための構造だけを用意し、その流れ方を制御部からシーケンス的に指示してもらう形にする。
- **制御部:** 制御部は、入力信号やデータバスからの信号に反応して、データバスを制御する信号を必要なシーケンスにしたがって動作させる。一般的にステートマシンで設計する。制御部はデータバスに比べてゴチャゴチャするのでランダム論理とよぶこともあります。

実際の機能モジュール内では、データバスと制御部は 1 組だけではなく、それぞれ複数組あったり、階層化されてたりする場合があり、実現したい機能に応じて、いかにわかりやすく設計を分割するかが方式設計で最も重要な作業になります。

A.7.2 データバス部

データバス部は、機能モジュール内のデータの流れを司るブロックです。実現する機能内のデータの流れに着目して最適な構造を設計します。データバス内には、外部からのデータの受け取り用レジスタ、データの内部格納用レジスタ、外部へのデータ出力用レジスタなど、データを一時的に蓄えるためのレジスタがまず最低限含まれます。また、データの演算器やシフタ、データ選択用セレクタなども含まれます。必要に応じてデータを加工しながら順々にレジスタに転送していくような、パイプライン構造を採用することもあるでしょう。データ格納用のレジスタは D-F/F で構成され、その他の要素は組合せ回路で構成されるのが通常です。データ格納用レジスタへのライト・ストローブ信号、演算器の演算種類の選択信号、データ・セレクタの選択信号などは、すべて制御部から受け取ります。一部演算結果や何かの判定結果などを制御部に返すこともあります。

A.7.3 データバス内のレジスタとその制御

データバス内で一時的にデータを記憶するレジスタの書き方を図 A.21(a) に示します。クロックやリセット

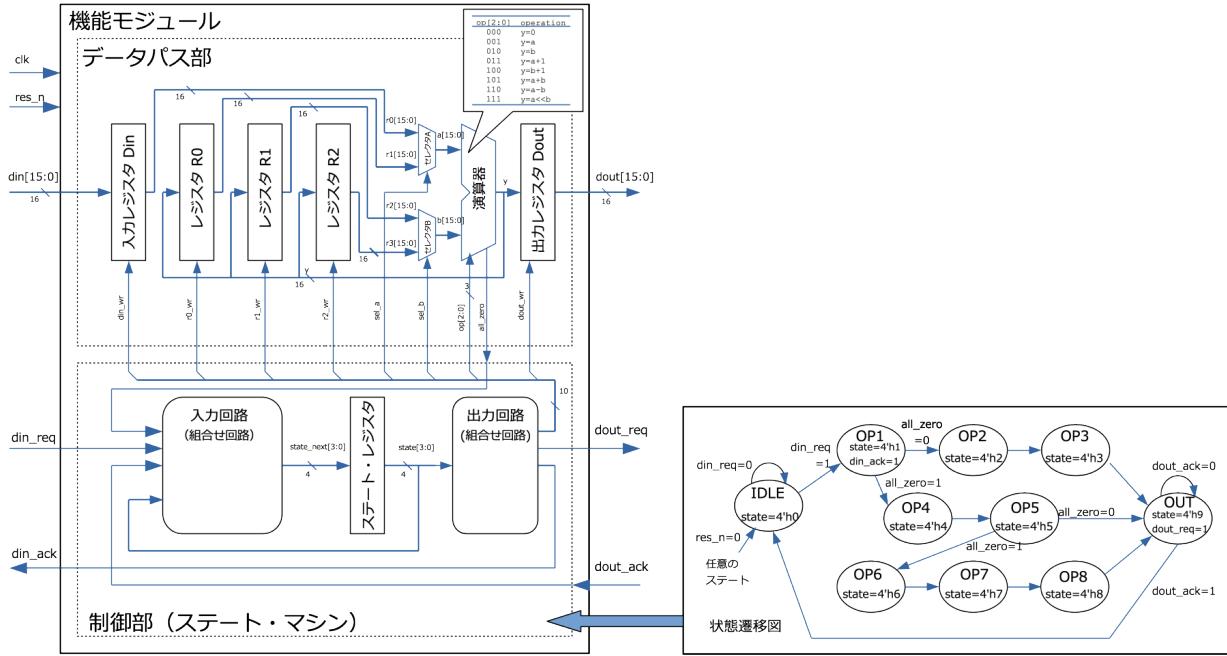


図 A.20: 機能モジュールの構造 (データパスと制御部)

信号も存在しますがここには一般的には表現しません。このレジスタではライト制御信号 wr がアサートされたときのクロックの立ち上がりエッジで入力データ din を取り込みます。出力データ $dout$ はレジスタ内の値が常時出力されています。信号 wr は制御部からもらいます。

このレジスタの回路例を図 A.21(b) に示します。データ・セレクタと D-F/F を組み合わせており、 wr 信号が 1 のときだけ din を選択して D-F/F に書き込み、 wr 信号が 0 のときは $dout$ をそのまま din に戻して同じ値を保持します。ASIC の回路では D-F/F のクロック入力が動き続けると電力消費が多くなるので、 wr 信号が 0 の間はクロックを止める回路（ゲーテッド・クロック）を論理合成ツールに自動挿入させることもできます。

動作タイミングを図 A.21(c) に示します。 wr 信号が 1 になった期間の最後のクロックの立ち上がりエッジ (T2) で $dout$ が変化する感覚を身につけてください。

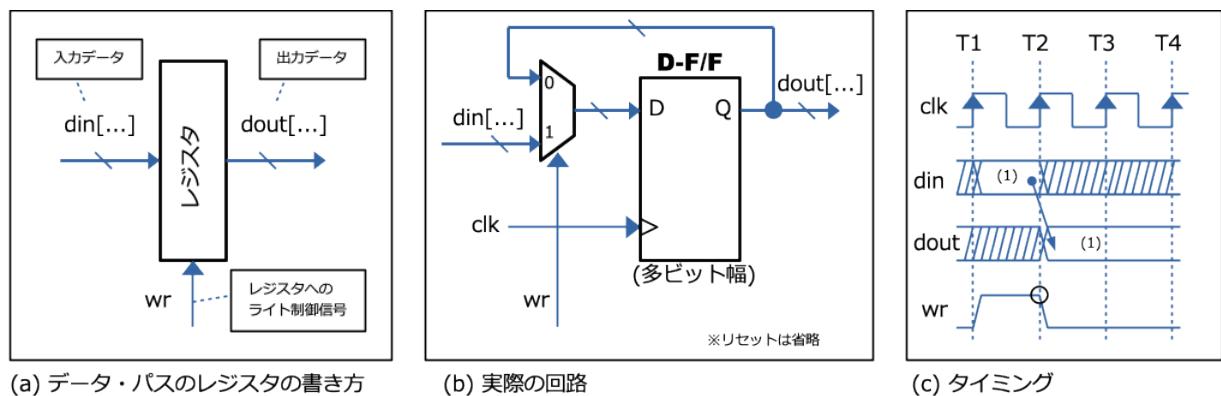


図 A.21: データパスのレジスタと制御信号

A.7.4 データパス内の演算器やセレクタとその制御

データパス内では、入力データとレジスタの間、レジスタとレジスタの間、レジスタと出力データの間には、データ値の加工を行うため演算器やセレクタを配置します。演算器やセレクタは組み合わせ回路で構成され

ます。

図 A.22(a) に、演算器やセレクタをレジスタの前に配置する例の書き方を示します。演算器には演算の種類を指示するための制御信号 op が入力されます。演算の種類が多ければその選択のため信号 op は多ビット長になります。セレクタにはデータの選択信号 $selXX$ が入力されます。信号 op や $selXX$ はともに制御部からもらいます。この例では最終段の演算器から出力されるデータをレジスタが取り込む構造になっています。図には示していませんが、演算器からオーバ・フロー信号を制御部に戻してシーケンス制御のネタにする場合もあります。

演算器やセレクタで加工されたデータをレジスタが取り込む場合の動作タイミングを図 A.22(b) に示します。ライト信号 wr が 1 になった期間の最後のクロックの立ち上がりエッジ (T2) でレジスタはデータを取り込むので、T2 の直前の期間 (T1 と T2 の間) で制御信号 $selXX$ や op を確定させる必要があります。すなわち、レジスタへのライト信号と一緒に、そのレジスタの前のデータ加工制御信号も確定させます。このタイミング感覚も身に付けてください。

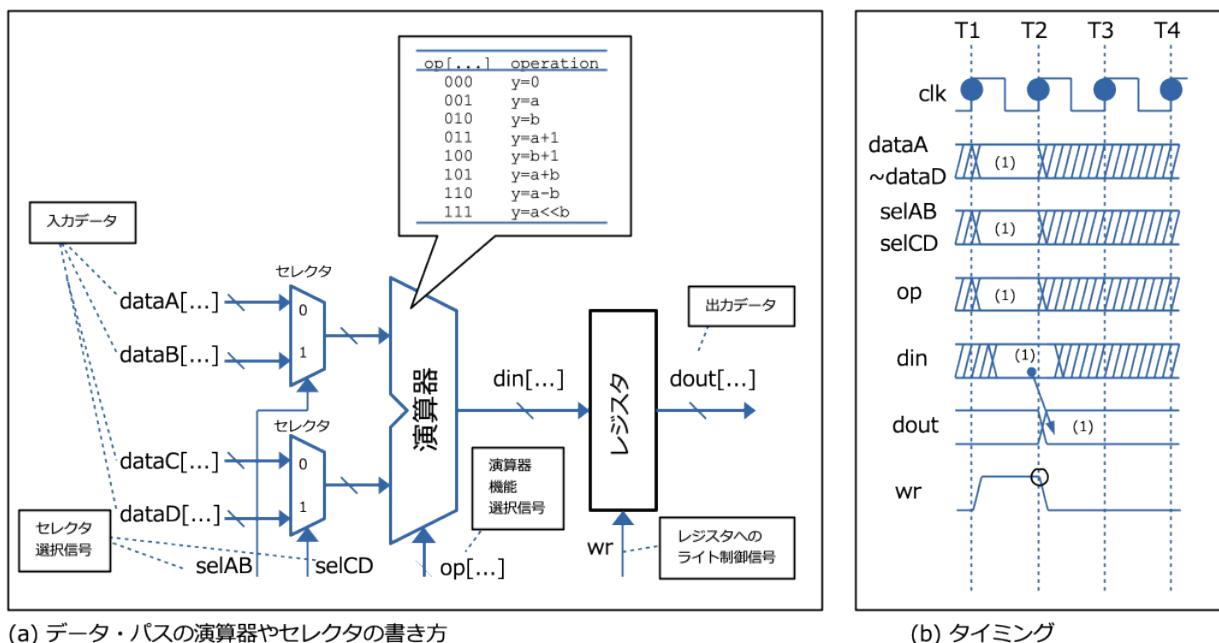


図 A.22: データパスの演算器やセレクタとその制御信号

A.7.5 データパスの構成

データパスの構成方法には、目的に応じて様々な方法があります。図 A.23 にその一例を示します。

図 A.23(a) は、バス型の構成です。レジスタと演算器を内部バス BUSX、BUXY、BUSZ で結びます。任意のレジスタから任意のレジスタに向けてデータを演算加工しながら転送できます。論理規模が小さくなる特長がありますが、演算処理を順番に一つずつしか実行できないので性能はさほど高くありません。また、制御部 (ステートマシン) は複雑になる傾向があります。一般的な CPU コアのデータパスでは、命令を順番に実行していくことが多いので、この構造を持つことが多いです。

図 A.23(b) は、パイプライン型の構成を示します。データを流す方向に向けてレジスタを順に並べて、その間に演算器などの組み合わせ回路を挿入したものです。複数の演算処理が同時に実行され、データの最終結果をクロック 1 サイクルごとに output できるので非常に性能が高くなる構造です。その分、論理規模は増えてしまします。ただし、データの加工方法がデータパスの構造で決定されているので、制御部 (ステートマシン) はシンプルになる傾向があります。

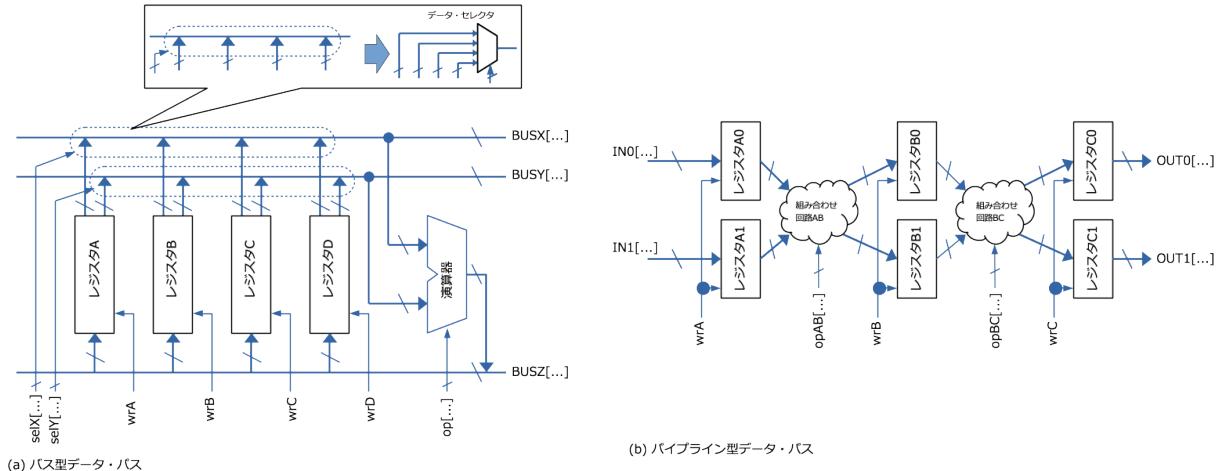


図 A.23: データパスの構成

A.7.6 内部バスの構成方法

現在の ASIC 設計や FPGA 設計では、内部に図 A.24(a) に示すような 3 ステート・バスは置いてはいけません。必ず図 A.24(b) のように、機能モジュールの出力を全てひき回して各モジュールの入力で必要なものをデータ・セレクタで選択します。

図 A.24(a) の方が配線本数が減るメリットがありますが、タイミング検証が非常にやりにくくなる問題があります。さらに、3 ステート・バスをドライブする機能モジュールは 1 個だけ (ワン・ホット) にしなくてはなりませんが、誤動作によって複数モジュールが 3 ステート・バスを一度にドライブしてしまうと、信号衝突による事故になることがあります。昔、実際にあった不良事例ですが、クロックモリセットも入らずに電源が投入されたとき、複数モジュールが 3 ステート・バスをドライブすることがあって、32 ビット幅のバズが全部衝突したため、デバイスが焼損したことがあります。怖いですよ。

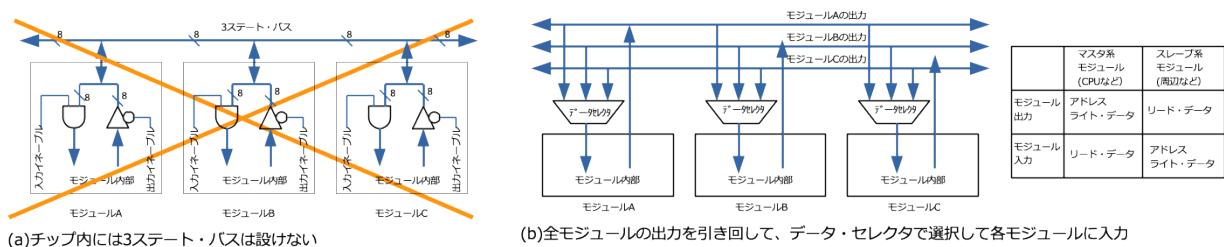


図 A.24: 内部バスの構成方法

A.8 その他、ASIC 設計で検討すべき項目

FPGA 設計に対して ASIC 設計では、特に以下の考慮が必要になります。

ひとつは低消費電力設計です。ASIC 化のメリットはとことん消費電力を最適化できることです。CMOS デバイスは信号をトグルする頻度が多いと消費電力が増え (ダイナミック電力)、また最近のファイン・プロセスのデバイスでは、信号がトグルしていないなくても生じる電源-GND 間のリーク電流 (スタティック電力) が無視できなくなっています。このためには、(1) D-F/F に入るクロックをなるべく変化させないようにするゲーテッド・クロック手法 (論理合成ツール側である程度自動化可能)、(2) 低速動作でかまわない機能モジュールのク

ロック周波数を落とすクロック・ドメイン管理手法、(3) 使用しないモジュールのクロックを停止させるクロック管理手法、(4) 使用しないモジュールの電源自体を遮断する電源遮断領域管理、などのさまざまな方法があります。もちろんこれらの設計手法は FPGA の低消費電力化にも効果的です。

もうひとつは出荷選別用のテスト回路の作り込み (DFT : Design for Testability) です。ASIC は内部に大量のトランジスタが集積されているのに、それをテストするための外部端子は桁違いに少ないです。そのため、内部テストするための専用の追加回路も設計しなくてはなりません。

最近の論理規模が大きいデバイスでは、もはや人手で十分な故障検出率を持つ出荷選別用のテスト・パターンは作成できないので、テスト・パターンを自動生成させるためスキャン回路というテスト専用回路を挿入します。この回路挿入は論理合成の工程で自動化できます。一方、A/D 変換器や FLASH メモリなどのマクロ・モジュールのテストのために各モジュールの内部信号を全て外部端子に引き出すようなテスト専用回路を作り込む必要があります。このテスト設計作業は ASIC 設計作業の中では仕様設計段階からしっかり検討を重ねて取り組む必要があるかなり重いものです。

A.9 SystemVerilog とは

A.9.1 Verilog HDL から SystemVerilog まで

論理回路を抽象的に記述するためのハードウェア記述言語としては Verilog HDL と VHDL が有名です。歴史的には、まず VHDL が米国国防総省によって開発されました。VHDL は記述量が多いのですが明確に機能仕様を定義できる特徴があります。一方、Verilog は記述量が VHDL よりも手軽にコーディングでき、現在では Verilog が大きく普及している印象です。

この Verilog は歴史的に複数のバージョンが定義されてきました。最初に標準化されたのは Verilog-1995 です。Verilog は論理シミュレータ側の視点で定義された言語であり、物理的な論理回路を確実に記述する際には多少の注意が必要でした。この不満を改善したのが Verilog-2001 で、大規模設計が容易になりました。

論理機能が高度化すると、その機能検証も複雑化してきます。これに対応するため、この後登場した SystemVerilog は、アサーション、機能カバレジ取得、ランダム検証、オブジェクト思考、など多くの機能が追加されており、機能検証の複雑化に対応しています。

本稿では、SystemVerilog による RTL 記述方法の基礎を説明します。全ての機能は説明しきれませんが、本稿の内容が理解できれば、ほとんどの論理回路は問題なく記述できると思います。

通常、SystemVerilog 記述のファイル名の拡張子には .sv を使用します。

A.9.2 論理シミュレーションの基本はイベント・ドリブン方式

ここで RTL で記述した論理機能がどのように論理シミュレーションできるかを簡単に説明します。

RTL 記述内の信号 (変数) の変化をイベントとよび、ある信号のイベントを検知するとその後段の論理機能および信号に伝搬させ、あらたなイベントを発生させ、これを順次繰り返していきます。この方法をイベント・ドリブン型論理シミュレーションとよびます。

A.9.3 遅延時間の扱いはタイム・マップで

遅延時間などの時間概念を扱うために論理シミュレータ内部にはタイム・マップというテーブルを用意します。タイム・マップには時刻ごとに処理すべきイベントを記録していきます。あるイベントを検知したら、タイム・マップに記録します。タイム・マップに現在の時刻に処理すべきイベントがあったらそれを取り出してイベントに伴った論理機能の処理と信号変化 (イベント) を生み、遅延時間に従って現在の時刻以降のタイム・マップにそのイベントを記録します。以下、これを繰り返して論理シミュレーションを実行していきます。RTL 記

述はハードウェアなので、全体を一度に並列動作させます。記述全体をなめながら全信号のイベントを確定させていくのが、論理シミュレーションの動作です。

A.10 SystemVerilog : モジュール構造と階層構造

A.10.1 モジュール構造と階層構造の記述

論理機能はモジュールという単位で記述します。図 A.25(a) のように論理機能は module 文と endmodule 文ではさみ、入出力信号と内部機能を記述します。

モジュールは一般的に階層構造を持ちます。例えば、一つの LSI の中に CPU や DMAC が内蔵されている場合、LSI という階層の下に CPU と DMAC があります。さらに CPU の下にも例えば DATAPATH と CONTROL という階層が置かれます。

A.10.2 インスタンス化

あるモジュールの下の階層として別のモジュールを配置する場合の記述方法を図 A.25(b) に示します。

ここで重要な概念があります。一つのモジュールは、あちこちで使いまわされることが一般的です。例えば、デュアル・コアを持つ LSI の場合は同じ CPU が 2 個あるでしょうし、2 入力 NAND ゲートをモジュールとして定義すれば LSI 内で同じものが大量に使いまわされます。このとき、ある階層内に置く個々の下位階層モジュールを区別するため、その階層内で固有となるインスタンス名を付けてその下位階層モジュールを置きます。これをモジュールのインスタンス化 (instantiation) といいます。インスタンス化というのは“実体化・具体化”という意味です。module 文に記述するモジュール名は定義名にすぎず、インスタンス化されて初めて物理的な回路になります。

原則として各モジュールは必ずその上位階層でインスタンス化されます。それでは LSI や FPGA の外部入出力信号が置かれる階層 (LSI や FPGA にとっての最上位階層) はどうなるのでしょうか？これもシミュレーション時にはインスタンス化されます。

後述しますが、LSI や FPGA 全体を論理シミュレーションする場合は、その外部回路にあたるテストベンチという階層を用意して、その中でインスタンス化されます (呼び出されます)。テストベンチはそれより上の階層がないのでインスタンス化されませんが、論理シミュレーション実行時にそのテストベンチが最上位階層であることを指示する必要があります。

A.11 SystemVerilog : 信号(変数)の表現

A.11.1 logic と wire

SystemVerilog 記述内のデジタル信号としては複数種の変数(信号)が定義されていますが、本稿では論理回路を記述するのに最低限必要な 2 つを解説します。

- **logic**: logic 型の信号は値が代入されるまで値を保持します。C 言語の変数と同様です。单一のドライバしか接続できず、双方向信号には使えません。値は 0, 1, x(不定), z(Hi-Z) の 4 値を取ります。
- **wire**: wire 型の信号は接続されたドライバ出力の値が反映され続けます。wire 型信号自身に値の保持機能はありません。複数のドライバを接続することができ、双方向信号や 3-ステート信号にも使えます。値は 0, 1, x(不定), z(Hi-Z) の 4 値を取ります。

もう少し正確に言うと、SystemVerilog は複数種類のデータ・オブジェクトがあり、その中に複数種類のデー

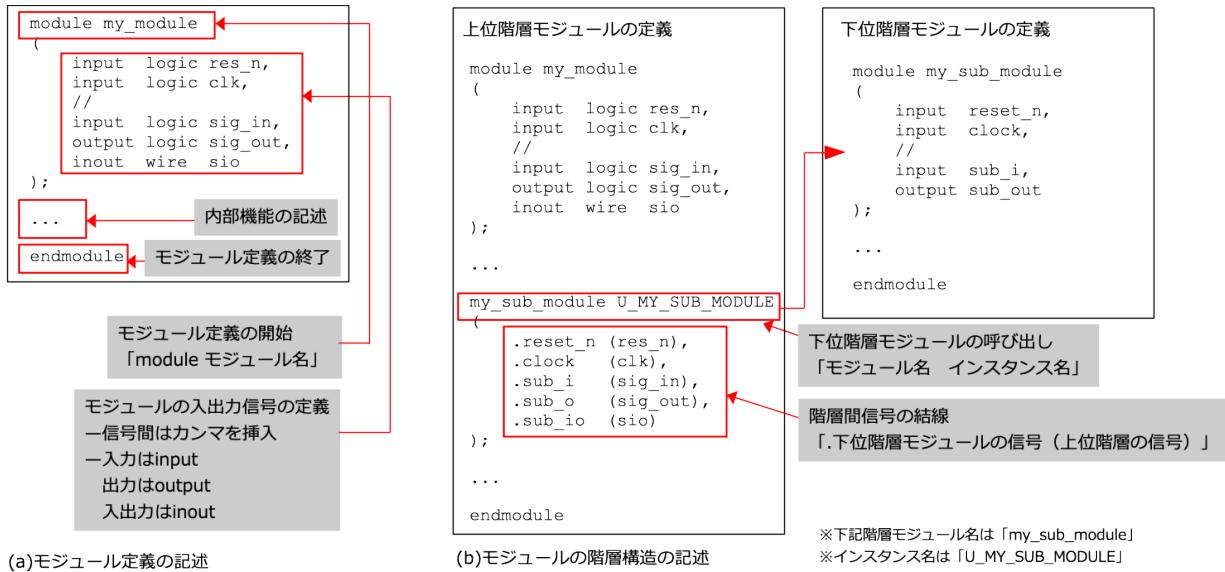


図 A.25: モジュール記述と階層構造

タ・タイプがあります。データ・オブジェクトの一つが wire と言うネット型で、データ・タイプの一つが logic と言う变数型です。

データ・タイプ logic は値として 0, 1, x(不定), z(Hi-Z) の 4 値を取ります。さらに 7 段階の信号時強度 (ドライバビリティ) を持つこともでき、信号衝突時の状態を含め全部で 120 の値を持てます。純粋な論理設計を行う範囲で信号強度を気にするのは、外部のプルアップやプルダウンとの信号競合程度でしょう。

下記のように宣言した信号 data はデータ・オブジェクトはネット型 wire で、データ・タイプは logic です。1 行目のように wire だけ宣言した場合も、暗黙的にデータ・タイプは logic となります。通常は 1 行目の宣言方式を使います。データ・オブジェクトがネット型だから、複数のドライバを接続可能です。

```
wire data;
wire logic data ;
```

下記のように宣言した信号 data はデータ・オブジェクトは变数型で、データ・タイプは logic です。logic だけ宣言した場合は、暗黙的にデータ・オブジェクトは变数型になります。通常はこの宣言方式を使います。データ・オブジェクトが变数型だから、单一のドライバしか接続できません。

```
logic data;
```

A.11.2 变数(信号)の定義方法

モジュール記述内で使う信号は、実際の記述に現れる前に定義する必要があります。モジュール内で使う信号の定義方法を図 A.26(a) に示します。また、モジュールのインターフェース記述(入出力信号)での信号定義方法を図 A.26(b) に示します。信号の定数値の表現方法を図 A.26(c) に示します。信号は必ずそのビット幅を意識した記述が必要です。

信号値の表記のうち、“x”は不定、“z”はハイ・インピーダンス状態(Hi-Z)を示します。

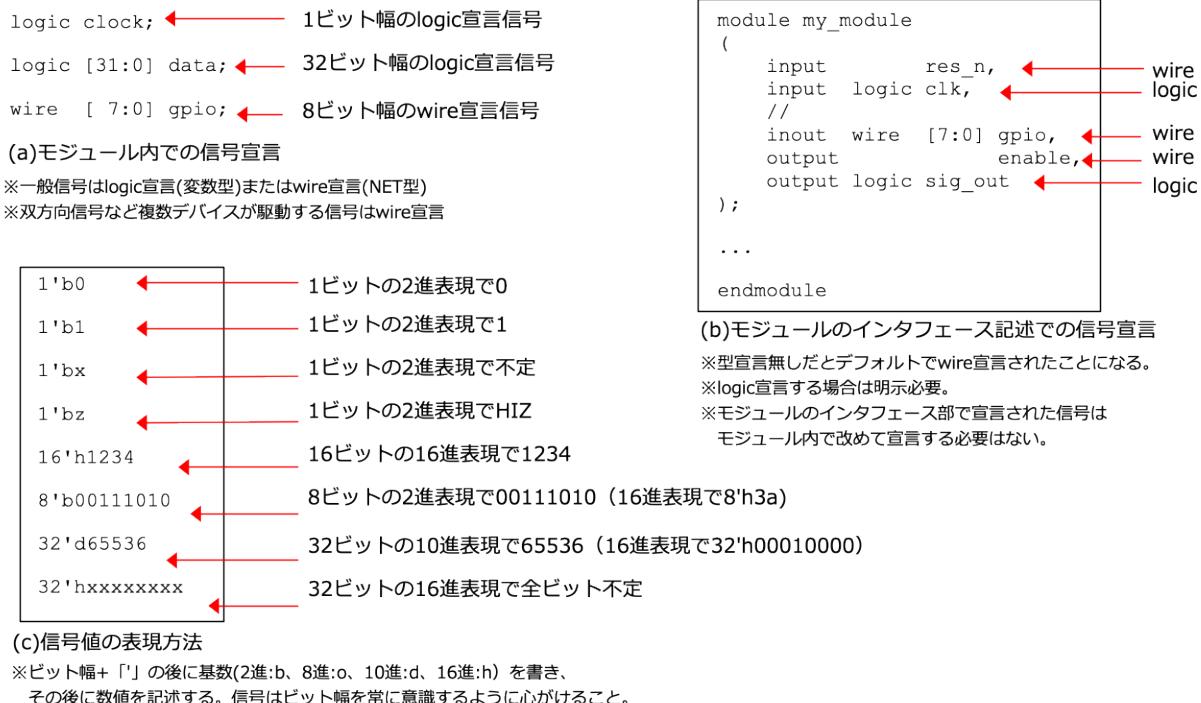


図 A.26: 信号の宣言

A.12 SystemVerilog : 組み合わせ回路の書き方

A.12.1 組合せ回路の書き方には2種類ある

組合せ回路の書き方を図 A.27 に示します。大きく分けて 2 種類の書き方があり、一つは assign 文を使う方法、もうひとつは always_comb 文を使う方法です。

A.12.2 繼続代入文 (assign)

assign 文は継続代入文とよび、図 A.27(a) のように logic 変数または wire 変数に対して代入を行うものです。代入文の左辺側の変数は常時代入され続けます。複数の assign 文が並んでいたら、上から順に実行されます。

A.12.3 手続き代入文 (always_comb)

always_comb 文は手続き代入文とよび、そのスコープ内 (always_comb 文直後のステートメント、または直後の begin-end 間のステートメント) の代入文の右辺にあるいずれかの信号が変化したらそのスコープ内の代入文が実行されます。代入文の左辺側の変数は、上記条件が成立したときだけ代入されますので、wire 宣言したものは使えません。

A.13 SystemVerilog : ブロックング代入とノン・ブロックング代入

SystemVerilog で重要な概念となる、ブロックング代入とノン・ブロックング代入について説明します。

ブロックング代入は図 A.28(a) のように、“=”で表現する代入文であり、複数のブロックング代入文があれば、記述された順番のとおりに代入が行われます。普通の C プログラムの代入文と同様な動作です。上から順に影響し合う代入操作なのでブロックングという言葉を使っています。この例では最終的に変数 d は a+b+4'h1 と

```

logic [7:0] C;
logic [7:0] D;
logic E;
wire [7:0] OUT;
//
assign C = A + B;
assign D = A & B;
assign E = |D;
assign OUT = (E == 1'b1) ? C : 8'hz;
(a)assign文による組合せ回路（継続代入文）

※assign文の代入は、logic信号またはwire信号に対して行う。
※シミュレーションの最小時間ステップ毎に右辺が左辺に継続代入。
※代入記号は=を使う(ブロッキング代入)。
```

```

logic [7:0] C;
logic [7:0] D;
logic E;
wire [7:0] OUT;
//
always_comb
begin
    C = A + B;
    D = A & B;
    E = |D;
end
//
assign OUT = (E == 1'b1) ? C : 8'hz;
(b)always文による組合せ回路（手続き代入文）

※always_comb文による代入は、logic信号に対してのみ行う。
※always_combのスコープ内（begin-end間）の式の
    入力(右辺)側のいずれかの信号が変化したとき、このスコープ内の式が
    上から順に、それぞれ右辺が評価され左辺に代入されていく。
※代入記号は=を使う(ブロッキング代入)。
```

図 A.27: 組合せ回路の書き方

なります。

ノン・ブロッキング代入は、図 A.28(b) のように、“ \leq ”で表現する代入文です。この動作は少し変わっていて、まず全てのノン・ブロッキング代入文の右辺の値を評価して一旦記憶します。この時点では左辺には代入しません。そして、全てのノン・ブロッキング代入文の右辺の評価が終ったら、記憶しておいた値を対応する左辺に代入します。上から順に影響し合わない代入操作なのでノン・ブロッキングという名前を使っています。この例では、まず変数 c には $a+b$ が代入され、変数 d には always_comb スコープ内が実行される前の変数 c の値に 4'h1 を加えたものが代入されます。最初の代入文で c が変化したので、再度この always_comb スコープが実行され、最終的に変数 d には $a+b+4'h1$ が入ります。結果は図 A.28(a) と同じですが、代入動作の流れが異なる点に注意してください。

組合せ回路を記述する場合は、assign 文でも always_comb 文でも一般的に図 A.28(a) に示したブロッキング代入文を使うことを推奨します。

```

logic [3:0] c;
logic [3:0] d;
//
always_comb
begin
    c = a + b;
    d = c + 4'h1;
end
(a) ブロッキング代入 (=)
```

※右辺の信号のいずれかに変化があれば
上の式から順に右辺が評価され
左辺に代入していく。

```

logic [3:0] c;
logic [3:0] d;
//
always_comb
begin
    c <= a + b;
    d <= c + 4'h1;
end
(b) ノン・ブロッキング代入 (<=)
```

※右辺の信号のいずれかに変化があれば
一旦、全ての式について右辺の評価値を
記憶してから左辺に代入する。

図 A.28: ブロッキング代入とノンブロッキング代入

A.14 SystemVerilog : 演算子

代入文で使える演算子を表 A.4(a) に示します。また演算子評価の優先順位を表 A.4(b) に示します。

一部注意すべき演算子について説明します。比較演算子 “== (イコールが 2 個)” は入力信号内に x または z があると比較結果は不定 (x) になります。一方 “==== (イコールが 3 個)” は入力信号内の x および z も信号値そのものとして比較判定します。結果は TRUE(1) か FALSE(0) になります。

単項演算子 & は、多ビット幅の信号の先頭に付けます。例えば $A[3:0]$ という 4 ビットの信号に対して “& A ” と書いたら、 $A[3]&A[2]&A[1]&A[0]$ のように全てのビットの論理積をとる演算操作になります。

表 A.4: SystemVerilog の演算子

種類	記号	書式例	意味
ビット単位 演算子	\sim	$\sim m$	m の各ビットを反転
	$\&$	$m \& n$	m と n の対応するビットを AND
	$ $	$m n$	m と n の対応するビットを OR
	\wedge	$m \wedge n$	m と n の対応するビットを XOR
	$\wedge\wedge$ or $\wedge\sim$	$m \wedge\wedge n$	m と n の対応するビットを XNOR
	$<<$	$m << n$	m を n ビット分左シフトし、LSB側から 0 を埋める
単項リダクション 演算子	$>>$	$m >> n$	m を n ビット分右シフトし、MSB側から 0 を埋める
	$\&$	$\&m$	m の全ビットを AND(結果は 1 ビット)
	$\sim\&$	$\sim\&m$	m の全ビットを NAND(結果は 1 ビット)
	$ $	$ m$	m の全ビットを OR(結果は 1 ビット)
	$\sim $	$\sim m$	m の全ビットを NOR(結果は 1 ビット)
	\wedge	$\wedge m$	m の全ビットを XOR(結果は 1 ビット)
論理演算子	$\wedge\wedge$ or $\wedge\sim$	$\wedge\wedge m$	m の全ビットを XNOR(結果は 1 ビット)
	!	$!m$	m は True ではないか？(結果は 1 ビットの論理値)
	$\&\&$	$m \&\& n$	m と n が共に True か？(結果は 1 ビットの論理値)
論理等号演算子 関係演算子	$ $	$m n$	m または n のいずれかが True か？(結果は 1 ビットの論理値)
	$==$	$m == n$	m と n は等しいか？(結果は 1 ビットの論理値)
(入力に x か z を含めば結果は x)	$!=$	$m != n$	m と n は等しくないか？(結果は 1 ビットの論理値)
	$<$	$m < n$	m は n より小さいか？(結果は 1 ビットの論理値)
	$>$	$m < n$	m は n より大きいか？(結果は 1 ビットの論理値)
	$<=$	$m <= n$	m は n 以下か？(結果は 1 ビットの論理値)
ケース等号演算子 (x, z 自体も比較)	$>=$	$m >= n$	m は n 以上か？(結果は 1 ビットの論理値)
	$====$	$m === n$	m と n は等しいか？(結果は 1 ビットの論理値)
条件演算子	$!==$	$m != n$	m と n は等しくないか？(結果は 1 ビットの論理値)
	$? :$	$sel ? m : n$	sel が 真なら m を返し、 sel が 真でなければ n を返す
連接演算子	$\{ \}$	$\{m, n\}$	m と n を連接して、大きいビット幅の数値を返す
	$\{\{ \}$	$\{n\{m\}\}$	m を n 個連接する
算術演算子	$+$	$m + n$	m に n を加算
	$-$	$m - n$	m から n を減算
	$-$	$-m$	m の符号反転(2 の補数)
	$*$	$m * n$	m に n を乗算
	$/$	m / n	m を n で除算
	$\%$	$m \% n$	m を n で割った剰余

(a) 演算子

優先順位	演算子	備考
高い ↑	$!$ \sim $+$ $-$	単項演算子
	$\{ \} \{ \{ \}$	
	$()$	
	$*$ $/$ $\%$	
	$+$ $-$	二項演算子
	$<<$ $>>$	
	$<$ $<=$ $>$ $>=$	
	$==$ $!=$ $====$ $!==$	
	$\&$ $\sim\&$	
	\wedge $\sim\wedge$	
	$ $ $\sim $	
	$\&\&$	
	$ $	
低い ↓	$:$	

(b) 演算子の優先順位

A.15 SystemVerilog : 時間と遅延

A.15.1 時間単位と時間精度の設定

代入文には遅延を付けることができますが、その前に論理シミュレーションにおける時間単位と時間精度の設定方法を説明します。これにはコンパイラ指示子の `timescale 文を使います。例えば、

```
`timescale 1ns/100ps
```

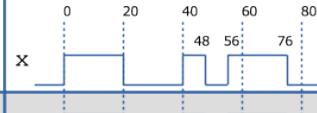
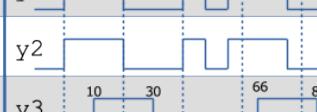
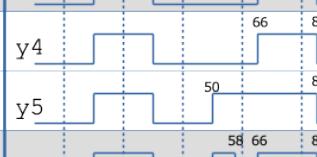
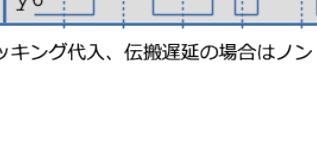
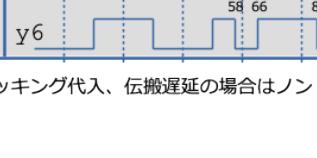
と書くと、論理シミュレーション内で記述する時間数値の単位が 1ns になります。また時間精度(論理シミュレータのタイム・マップの時間刻みの間隔、すなわち時間数値の小数点以下の桁)が 100ps になります。この `timescale 文を書いた時点でそれ以降の RTL 記述における時間パラメータが設定されます。`timescale 文は何度も書いて時間単位の設定し直しができますが、基本的には論理シミュレーションのトップ階層(テストベンチ)の最初でまず記述しておくことを推奨します。

A.15.2 遅延付加時の代入動作

代入文での信号遅延時間は、“#数値”で表現します。単位は前述の `timescale で設定したものになります。この遅延時間表記は、代入文の右辺に付ける場合と左辺に付ける場合があります。また、ブロックキング代入に付ける場合とノン・ブロックキング代入に付ける場合で動作が異なります。それらをまとめたものを表 A.5 に示します。

一般的には、ゼロ遅延または慣性遅延を使うので、ブロックキング代入を使います。どうしても伝搬遅延させたい場合だけノン・ブロックキング代入を使います。

表 A.5: ブロックキング代入とノン・ブロックキング代入の遅延付加時の動作

代入文	x	遅延	代入動作
always_comb y1 = x;		ゼロ	xの変化時点でxをy1に代入。
always_comb y2 <= x;		ゼロ	xの変化時点でxを記憶し、全ての式の評価完了後、y2に代入。
always_comb #10 y3 = x;		慣性	xの変化から10経過時点のxの値をy3に代入。その間のxの変化は無視。
always_comb #10 y4 <= x;		(慣性)	xの変化から10経過時点のxの値を記憶し全式評価後y4に代入。その間のxの変化は無視。
always_comb y5 = #10 x;		(慣性)	xの変化時点のxの値を時刻10経過後にy5に代入。その間のxの変化は無視。
always_comb y6 <= #10 x;		伝搬	xの変化時点のxの値を記憶し、全式評価後のさらに時刻10経過後にy5に代入。

推奨：ゼロ遅延と慣性遅延の場合はブロックキング代入、伝搬遅延の場合はノン・ブロックキング代入を使う。

A.16 SystemVerilog : 条件判断

条件判定処理には if 文や case 文を使います。これらの文は always 系の文の中で使いますので、if 文や case 文で判定されて代入される変数は logic 宣言します。

A.16.1 if 文による条件判断

if 文の書き方の例を図 A.29 に示します。単純な条件判定の場合は、表 A.4 に示した条件演算子 (?:) を使えます。条件演算子は assign 文でも always 型の文の中でも使えます。

if 文や条件演算子のあとに書く条件式には、重要な注意点があります。例えば、(A == B) という条件式において、A または B に不定 (x) か HIZ(z) が含まれていた場合、条件式の値そのものは不定 (x) になりますが、if 文や条件演算子では FALSE 判定側に記述したステートメントが実行されます。この仕様は、RTL ベースの論理シミュレーションと論理合成後のゲート・レベルの動作の不一致を招くことがあります。条件式の中の変数に不定 (x) か HIZ(z) が含まれる可能性がある場合は注意してください。

A.16.2 case 文による条件判定

case 文の書き方の例を図 A.30 に示します。case と endcase の間に、入力信号の条件値とその条件成立時に実行するステートメントを羅列します。default 条件は、記述した条件のいずれも一致しなかったときに選択されます。case 文は真理値表を記述するときに便利な記法です。if 文ではネスティングが深くなってしまう場合も case 文だとスッキリ記述できます。

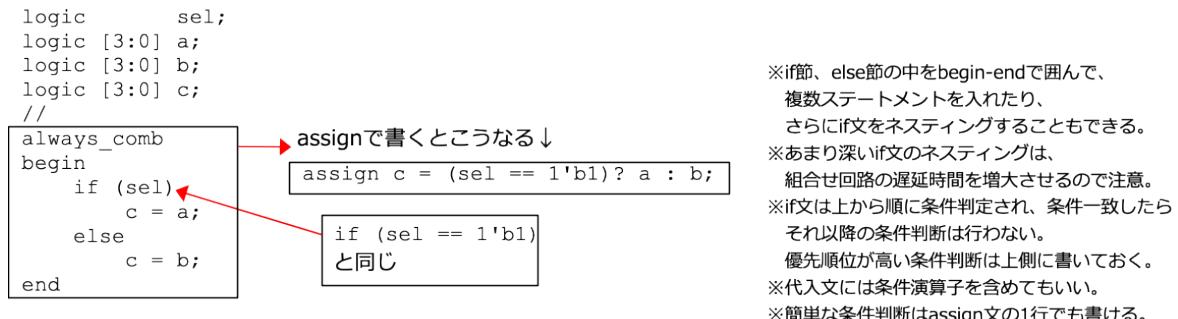


図 A.29: if 文による条件判断

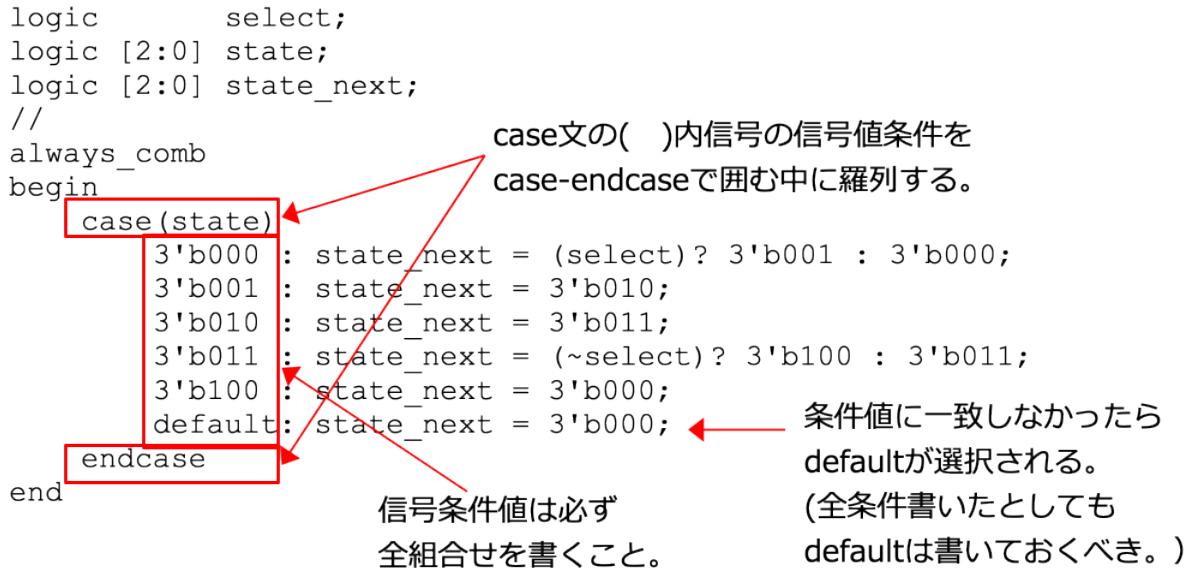


図 A.30: case 文による条件判断

case 文に似たものに casex 文と casez 文があります。その動作を図 A.31 に示します。条件値の中に“?”を書くことで don't care(0, 1, x, z の全てに一致) を表現できます。case 文、casex 文、casez 文のぞれを使うべきかは、論文 1 本分くらいの規模の議論がさまざまにされていますが、RTL 記述と合成後のネット間の論理シミュレーション動作を一致させるには casez(および don't care 指示子“?”) を使うべきとされています。

SystemVerilogにおいては、確実に入力条件に don't care を含めたい場合は case inside を使うことが推奨されていますが、論理シミュレータによってはサポートされていないことがあるので、本稿では、casez と“?”を使うことを推奨しておきます。

A.16.3 ラッチ回路生成の危険

組合せ回路記述の中で、if 文に else 項が抜けていたり、case 文で入力信号の全組合せが条件値に記述しきれていない場合、すなわち条件が成立しない場合のステートメント(代入文)が存在しない場合は、変数への代入動作が行われない(変数の値が保持される)可能性が生じます。こうした RTL 記述を論理合成すると、組合せ回路のつもりでもラッチ回路が合成されてしまいますので、注意してください。

```

logic [1:0] din;
logic [1:0] dout;
//  

always_comb  

begin  

    case(din)
        2'b00 : dout = 2'b10;
        2'b01 : dout = 2'b10;
        2'b10 : dout = 2'b01;
        2'b11 : dout = 2'b01;
        2'bxz : dout = 2'b00;
        default: dout = 2'b00;
    endcase
end

```

case文の場合はxとzはdon't care
xとzも値として比較判断される。


```

logic [1:0] din;
logic [1:0] dout;
//  

always_comb  

begin  

    casex(din)
        2'b0x : dout = 2'b10;
        2'b1z : dout = 2'b01;
        default: dout = 2'b00;
    endcase
end

```

casex文の場合はxとzはdon't care
(0,1,x,zのいずれとも一致)
として扱う。
?はxと同じと見なされる。


```

logic [1:0] din;
logic [1:0] dout;
//  

always_comb  

begin  

    casez(din)
        2'b0? : dout = 2'b10;
        2'b1? : dout = 2'b01;
        default: dout = 2'b00;
    endcase
end

```

casez文の場合はzはdon't care
(0,1,x,zのいずれとも一致)
として扱う。
xはxかzに一致するとして扱われる。
?はxと同じと見なされる。


```

logic [1:0] din;
logic [1:0] dout;
//  

always_comb  

begin  

    case (din) inside
        2'b0? : dout = 2'b10;
        2'b1? : dout = 2'b01;
        default: dout = 2'b00;
    endcase
end

```

case insideを使うと、
?をdon't careとして扱う。
最も推奨される記述だが
論理シミュレータによっては
サポートされていないことがあります。

※条件内にdon't care (wildcard) を含める場合は、本稿ではcasezを使い、don't care指示には?を使う。

図 A.31: casex 文や casez 文による条件判断

A.17 SystemVerilog：順序回路の書き方

D フリップ・フロップ(以下 D-F/F と略す)の RTL 記述は always_ff 文を使って図 A.32(a) のように書きます。D-F/F の出力信号は logic 宣言します。always_ff 文の@(...) 内のイベント・リストにクロック遷移条件を記述します。

“posedge clk(クロック信号名)”と書くと、clk の立上りと言う意味になります。clk の立ち下がりで入力データをたたく場合は“negedge clk(クロック信号名)”と記述します。always_ff 文のスコープ内に出力信号への代入文をノン・ブロッキング形式(<=)で記述します。

A.17.2 リセット付き D-F/F の書き方

非同期リセット付きの D-F/F は図 A.32(b) のように記述します。always_ff 文のイベント・リスト中にリセット信号の遷移方向を加えます。リセット信号が正論理の場合は“posedge リセット信号名”を、負論理の場合は“negedge リセット信号名”と記述します。always_ff 文のスコープ内では、リセット信号による出力信号への代入を if 文を使って記述します。

同期リセット付きの D-F/F の場合は、図 A.32(c) のように always_ff 文のイベント・リスト中にリセット信号の遷移は書きません。入力信号をリセット信号を使って加工してクロックでたたく形になります。

ここに示したような RTL 記述を論理合成ツールにかけると、自動的に D-F/F を想起して論理合成してくれます。

A.17.3 順序回路のひな形

D-F/F を使った順序回路の一般的なひな形を図 A.32(d) に示します。原則として非同期リセットを付加してください。入力条件に応じて出力信号の値をクロックに同期して変化させます。この例では、cond_A が 1'b1 のときの clk の立上りエッジ(T4) で dout が 4'ha に変化し、cond_B が 1'b1 のときの clk の立上りエッジ(T5) で dout が 4'hb に変化します。cond_A の起源となる信号もクロック clk による順序回路から生成されているとすれば、cond_A は必ず clk の立上りエッジより後ろ、すなわち T3 より後ろで(かつ HOLD を満たして)1'b0 から 1'b1 に変化しますので、この D-F/F では必ず T4 になってから cond_A を取り込むことになります。cond_B も同様です。

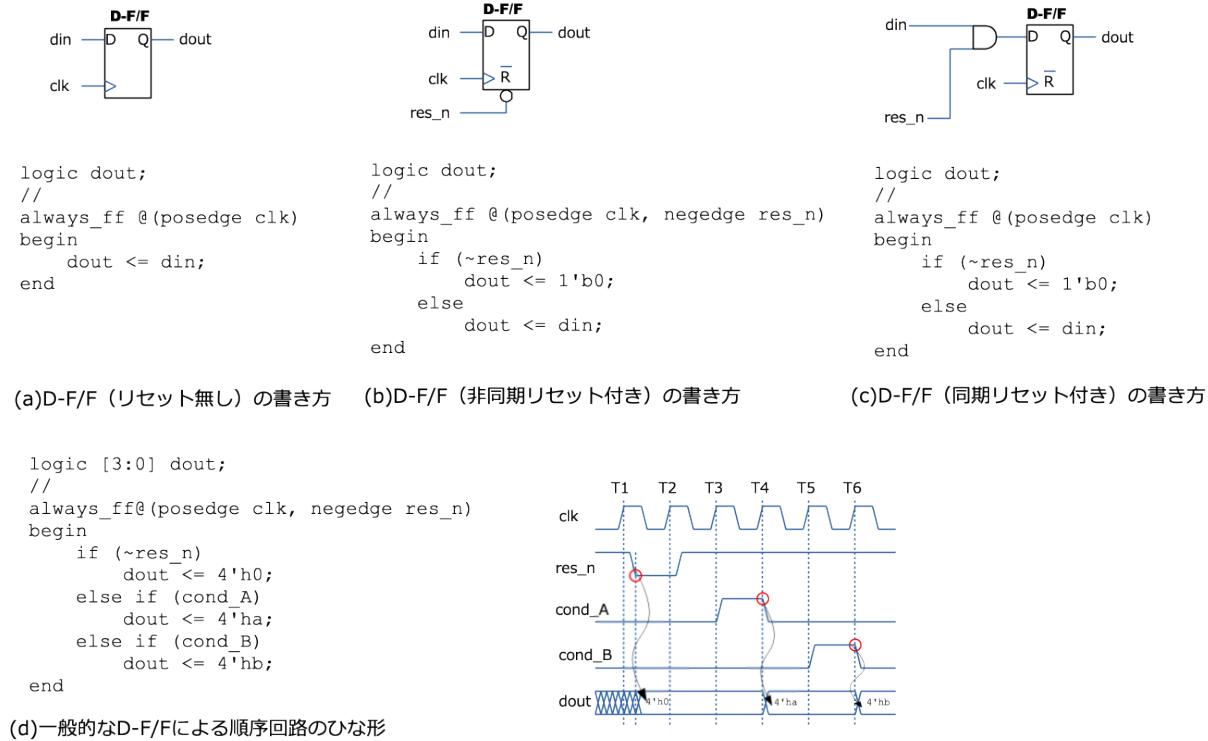


図 A.32: D フリップ・フロップの書き方

A.17.4 順序回路は必ずブロッキング代入文を使う

順序回路では、必ず図 A.33(a) のようにブロッキング代入文 (\leq) を使ってください。もし、順序回路の中身をノン・ブロッキング代入文を使ったらどうなるか、図 A.33(b)を見ればわかりますね。期待通りの動作になりません。

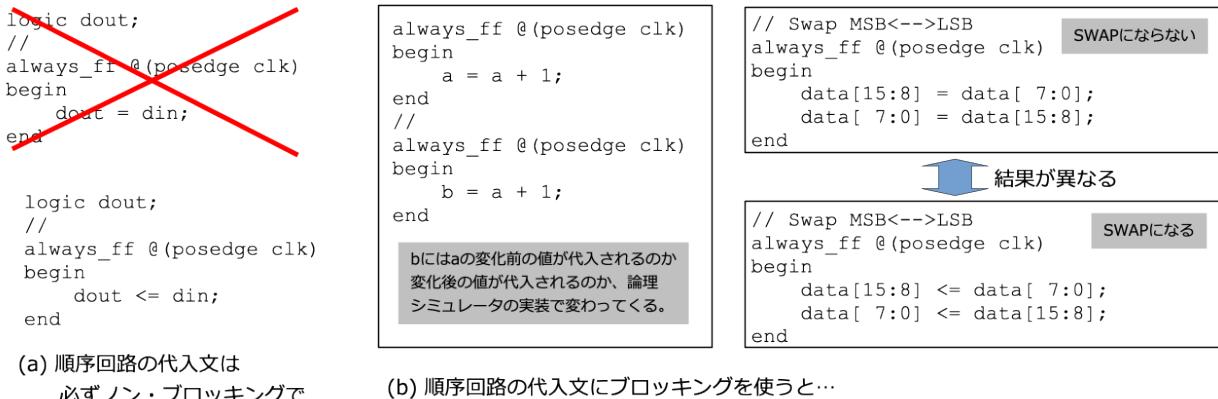


図 A.33: D フリップ・フロップの代入文

A.18 論理シミュレーションのためのテストベンチ

A.18.1 ここまで文法で全ての論理機能は記述できる

ここまで説明した SystemVerilog の文法で、基本的にあらゆる論理機能の本体は記述しきることができます。

本稿で示す事例でも、ここまで説明した記述方法だけしか使っていません。他にも多くの便利な記法がありますが、それらは各種教科書や文献を参照してください。

A.18.2 論理シミュレーションを動かすテストベンチ

設計目的とする論理機能が記述できたら、今度はそれを論理シミュレータで動作させて機能検証します。論理機能本体を動かすための、その上位階層に位置させるテストベンチを用意します。

テストベンチ記述の全体構成の一例を図 A.34 に示します。テストベンチも RTL 記述上ではひとつの階層、すなわち module です。ただしテストベンチから出入りする信号はありませんので、module 文にはモジュール名のみ(この例では“tb”のみ)を記述します。テストベンチの下層には、検証対象論理機能モジュールをインスタンス化して配置します。

論理設計において、論理機能の RTL コードの記述段階になったら、論理機能本体を書く前にまずテストベンチから記述を開始することをお勧めします。まずはリセットとクロックの生成だけでも用意するべきでしょう。最外殻のテストベンチから、順次内側に向かって各階層の論理を記述していくと、記述中の階層より外側の論理記述が自分にとってのテストベンチになってくれます。そうすれば、少しずつ論理機能を動かして基本動作を確認しながらコーディングを進めることができます。結果的に記述ミス(タイプ)によるエラーを防止しやすくなります。最近の論理シミュレータは高速化されていて、かつ RTL 記述のシミュレーション自体が軽いですから、何度もシミュレーションを繰り返すことはさほど時間がかかるものではありません。以下、テストベンチ記述内の各要素を説明します。

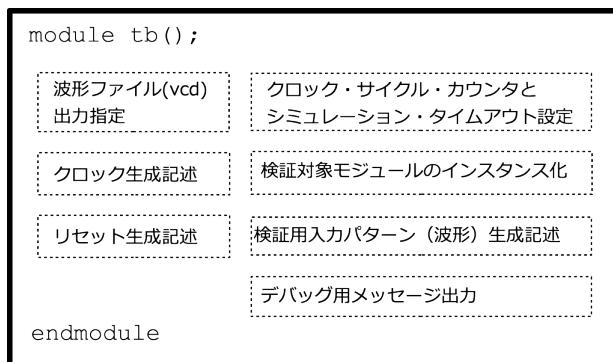


図 A.34: テストベンチの構成

A.18.3 initial 文

最外殻のテストベンチ内では initial 文を多用します。initial 文のスコープ内のステートメントは、論理シミュレーション開始時に 1 回だけ実行されます。これを使って、論理シミュレータへの動作指示や、信号の初期化、連続した信号生成などを行うことができます。initial 文内で代入する信号(変数)は logic 宣言する必要があります。

A.18.4 forever 文

同様に forever 文も使われます。このスコープ内の記述が継続的に実行されます。initial 文の中に forever 文を含めることで、例えば、特定の信号の継続的監視やクロックなど定常に動かす信号の生成などに使えます。

A.18.5 波形ファイル出力指示

論理シミュレーション結果の細かい部分は信号波形でチェックする必要があります。そのためテストベンチ内では波形ファイルを出力することを論理シミュレータに指示します。その指示方法を図 A.35 に示します。この

例ではテストベンチ階層 (tb) 以下全階層の全信号をファイル名 “tb.vcd” に出力することを指示しています。信号出力する階層位置や階層数も指定することもできます。

波形ファイルは VCD(Value Change Dump) フォーマット形式です。一般的なフォーマットで多くの EDA ツールが扱うことができるもので、中身は ASCII ファイルで普通のエディタでも読めます。この VCD 形式の波形ファイルは、波形ビューウェー GTKWave で表示できます。

```
initial ← シミュレーション開始時に1回実行
begin
    $dumpfile ("tb.vcd"); ← 出力ファイル名を指定
    $dumpvars (0, tb); ← 0 : 指定以下全階層の信号を出力
end                                         (1以上→出力階層数を指定)
                                              tb : 階層tb以下の信号を出力
```

図 A.35: 波形ファイル出力指示例

A.18.6 クロック信号生成記述

論理回路に欠かせないクロック信号は、テストベンチ上で図 A.36 のようにして生成します。クロック信号は initial 文で一旦初期化した後、always 文と遅延指定で一定周期でトグルして生成できます。この例では、コンパイラ指示子の define 文でクロック周期の値を定数として定義して使ってています。

```
`define TB_CYCLE 20 //ns ← 定数（参照時はTB_CYCLEの先頭にバック・クオートを付ける）
//
logic clock; ← シミュレーション開始時にクロックのレベルを初期化
//
initial clock = 1'b0; ← 半周期ごとに反転
always #(`TB_CYCLE / 2) clock = ~clock;
```

図 A.36: クロック信号生成記述例

A.18.7 リセット信号生成記述

RTL 記述内の全ての信号の初期値は不定 (x) です。論理シミュレーション実行開始時は、通常、テストベンチ上でリセット信号を強制的に一定期間アサートさせるところから始めます。リセット信号の生成例を図 A.37 に示します。

```
logic reset;
//
initial ← シミュレーション開始時に1回実行
begin
    reset = 1'b1; ← reset (正論理)を1にセットして
    # (`TB_CYCLE * 10) ← 10サイクル分の時間待ってから
    reset = 1'b0; ← reset (正論理)を0にクリアする
end
```

図 A.37: リセット信号生成記述例

A.18.8 サイクル・カウンタとシミュレーションのタイムアウト

論理シミュレーションを実行開始すると、何もしなければ永遠に実行を続けます。波形ファイルを出力し続けていたらディスクがパンクするまで実行し続けてしまいます。ある信号が何らかの条件になつたらシミュレーションを停止させる記述が可能ですが、その有無に関わらず、クロック・サイクル数が一定値になつたらシミュレーションを停止させるタイムアウト機能をテストベンチに入れておくことを推奨します。

その一例を図 A.38 に示します。クロックの立上りエッジごとに 32 ビット幅のレジスタ tb_cycle_counter をインクリメント (カウント動作) させ、10000 サイクルに到達したら \$display 文や \$write 文でメッセージを出力した後、システム・タスクの \$finish 文や \$stop 文で論理シミュレーションを停止させます。

\$finish 文はシミュレータ自体を終了させます。\$stop 文はシミュレーションを一時的に停止させ、シミュレータがコマンド待ちになります。

```

`define TB_FINISH_COUNT 10000 //cyc ← シミュレーションの最大サイクル数を指定
//
logic [31:0] tb_cycle_counter;
//
always_ff @(posedge clock, posedge reset)
begin
    if (reset)
        tb_cycle_counter <= 32'h0;
    else
        tb_cycle_counter <= tb_cycle_counter + 32'h1;
end
//
initial
begin
    forever
        begin
            @(posedge clock);
            if (tb_cycle_counter == `TB_FINISH_COUNT)
                begin
                    $display("*** SIMULATION TIMEOUT *** at %d", tb_cycle_counter);
                    $finish;
                end
        end
    end
end

```

1サイクルごとに
クロック・カウンタ(32ビット)
をインクリメント

クロック・カウンタが指定した
最大値になつたらシミュレーション停止

メッセージ出力

シミュレーション終了
 • \$finishはシミュレータ自体を終了させる。
 • \$stopはシミュレーションを一時的に停止させ
 シミュレータがコマンド待ちになる。

図 A.38: サイクル・カウンタとシミュレーションのタイムアウトの記述例

A.18.9 デバッグ用メッセージ出力

論理機能を検証する際、波形だけでなく、メッセージを出力したくなる場合があります。ステート遷移などはメッセージで出力した方がわかりやすいケースが多いです。その例を図 A.39 に示します。テストベンチより下の階層の信号を、“インスタンス名. インスタンス名. 信号名”的に階層ごとにドットで区切って指定して、システム・タスクの \$display 文や \$write 文により表示できます。\$display 文や \$write 文は C 言語の printf() のように使えますが、\$display 文では自動的に最後に改行が付加され、\$write 文では改行が付加されないという違いがあります。

A.18.10 機能検証用入力パターンの生成

テストベンチ内でインスタンス化した論理機能モジュールを動かすためには、リセット信号とクロック信号以外に、その論理機能への入力信号を適切に動かす必要があります。それを入力パターンまたはスティミュラス

```

always_ff @(posedge U_SUB_MODULE.clk)
begin
    if (U_SUB_MODULE.state != U_SUB_MODULE.state_next)
        begin
            $display("      state = %01x, state_next = %01x, at %d",
                    U_SUB_MODULE.state,
                    U_SUB_MODULE.state_next,
                    tb_cycle_counter);
        end
    end

```

メッセージ出力 (C言語のprintfに相当)

\$display()は最後に改行を付加する。
\$write()は最後に改行を付けない。

下位階層モジュールの信号をメッセージ出力できる。
(下の階層の信号名の指定法：
インスタンス名.インスタンス名.信号名)

図 A.39: デバッグ用メッセージ出力例

(stimulus : 刺激) とよびます。

その生成方法のひとつの例を図 A.40(a) に示します。initial 文と遅延指定を使って信号変化を順に記述する方法です。単純な入力ストリームを生成するにはシンプルで便利ですが、少し複雑な信号生成をしたくなるとたんに辛くなる方法です。

もうひとつの例を図 A.40(b) に示します。task 文を使う方法です。task 文内にはある決まった一連の入力パターンをサブルーチン風に定義することができます。task 文は引数を持つことができ、中で生成する入力パターンの一部を加工することもできます。その task 文で定義したタスクを initial 文の中から呼び出すことにより複雑な入力パターンを生成しやすくなります。

```

logic sig_in;
// initial
begin
    #(`TB_CYCLE * 0);
    sig_in <= 1'b0;

    // Wait Reset Done
    #(`TB_CYCLE * 20);
    sig_in <= 1'b1;
    #(`TB_CYCLE * 1);
    sig_in <= 1'b0;

    // Next Pulse
    #(`TB_CYCLE * 5);
    sig_in <= 1'b1;
    #(`TB_CYCLE * 1);
    sig_in <= 1'b0;

    // End of Simulation
    #(`TB_CYCLE * 20);
    $finish;
end

```

(a) 入力パターンを直接記述する場合


```

logic sig_in;
// task TASK_Pulse(input level);
begin
    #(`TB_CYCLE * 0);
    sig_in = level;
    // #(`TB_CYCLE * 1);
    sig_in = ~level;
end
endtask

```

ある決まった一連の入力パターンをサブルーチン的にtask文内に定義できる。
ここでは入力levelで指定した値で1サイクル幅のパルスを生成する。
task-endtaskで囲む


```

initial
begin
    #(`TB_CYCLE * 0);
    sig_in <= 1'b0;

    // Wait Reset Done
    #(`TB_CYCLE * 20);
    TASK_Pulse(1'b1);

    // Next Pulse
    #(`TB_CYCLE * 5);
    TASK_Pulse(1'b1);

    // End of Simulation
    #(`TB_CYCLE * 20);
    $finish;
end

```

入力パターンはinitial文内に記述するが
その中で、taskを呼び出すことができる。

(b) task文を使う場合

図 A.40: 機能検証用入力パターンの生成例

A.19 システム・タスクとコンパイラ指示子

よく使うシステム・タスクとコンパイラ指示子を表 A.6 にまとめておきます。

表 A.6: システム・タスクとコンパイラ指示子の例

種類	文	使用例	意味
システム タスク	\$display	\$display("state = %02x", u_TOP.state);	メッセージ出力(改行付き)
	\$write	\$write("state = %02x", u_TOP.state);	メッセージ出力(改行無し)
	\$finish	\$finish;	論理シミュレータを終了する。
	\$stop	\$stop;	シミュレーションを停止して 論理シミュレータのコマンド待ちになる。
コンパイラ 指示子	`timescale	`timescale 1ns/100ps	時間単位と時間精度の指定
	`define	`define TB_CYCLE 100	マクロ名の定義
	`include	`include "defines.v"	別ファイルのインクルード
	`ifdef	`ifdef IVERILOG	マクロ定義の状況に応じて コンパイル箇所をスイッチする。
	`else	\$finish;	
	`elsif	`elsif MODELSIM	
	`endif	\$stop;	
		`else	
		\$finish; // catch	
		`endif	

A.19.1 `include 文

C 言語の#include 文のように、RTL ソース記述内に別の RTL ソース・ファイルをインクルードすることができます。そのためのコンパイラ指示子として、`include 文が用意されています。

A.19.2 `ifdef 文、`else 文、`elsif 文、`endif 文

C 言語の#define 文のように、マクロ名の定義の有無に応じて有効化する行と無効化する行を選択できるコンパイラ指示子があります。それが図 A.41 に示す `ifdef 文、`else 文、`elsif 文、`endif 文です。マクロ名の定義によって、使う行と使わない行を選択できるので、一つの RTL 記述を複数の用途に使い分けることができます。マクロ名は同じ RTL 記述内で定義することもできますが、論理シミュレータの RTL 記述をコアコンパイルするときのコマンドの引数内で定義することもできます。

A.20 定数パラメータの上書き機能

定数値は

```
`define CYCLE 100
```

ように定義でき

```
if (count == `CYCLE) ...
```

のように先頭にバック・クオートを付けて使うことができます。この `define 文で指定する定数値は固定化され RTL 記述内で一定値しかとれません。

一方、論理モジュール (module 階層) をインスタンス化するときに、その中の定数パラメータを上書き変更することができる機能があります。それをサポートするのが parameter 文です。図 A.42 に示したように module の中に使う定数を parameter 文で定義しておくと、その上位階層でインスタンス化するときにその parameter

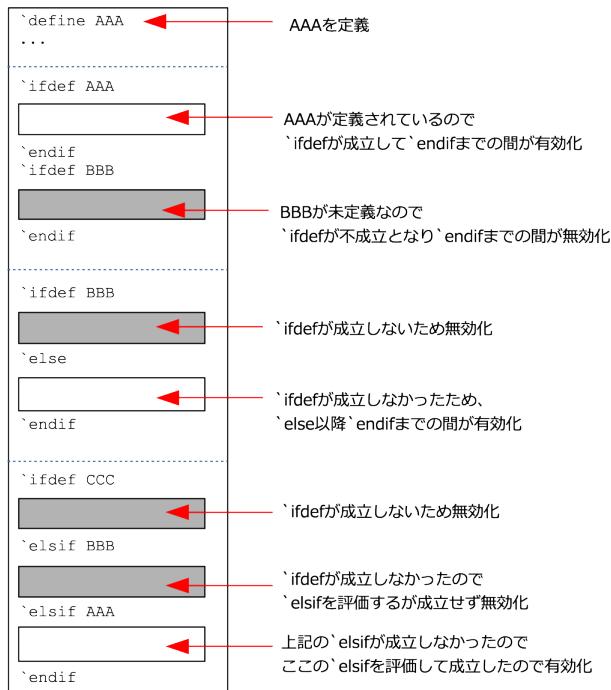


図 A.41: 記述文の選択

値を上書き変更することができます。この例ではレジスタの初期値を変更していますが、それ以外にも信号のビット幅なども parameter で指定 / 上書きできますので、module 記述の再利用性を向上させることができます。

A.21 論理シミュレータと波形ビューワのインストール

SystemVerilog による RTL 記述が期待通りに動作するかどうかは、しっかり論理シミュレーションにより機能検証します。機能検証の環境として、本稿ではオープン・ソースの環境を使います。論理シミュレータは Icarus Verilog を使い、波形ビューワには GtKWave を使います。ネイティブまたは WSL2 上の Ubuntu 環境では、以下のコマンドでインストールできます。

```
$ sudo apt install iverilog gtkwave
```

A.22 設計例：リロード式アップ・ダウン・カウンタ “simpleCounter”

A.22.1 リロード式アップ・ダウン・カウンタを記述してみよう

本事例ではリロード式アップ・ダウン・カウンタを記述してみます。また、テストベンチから入力パターンを与える方法として、タスク文を使う方法を試してみましょう。

A.22.2 設計ファイルの用意

ファイル一式は下記コマンドでダウンロードできます。

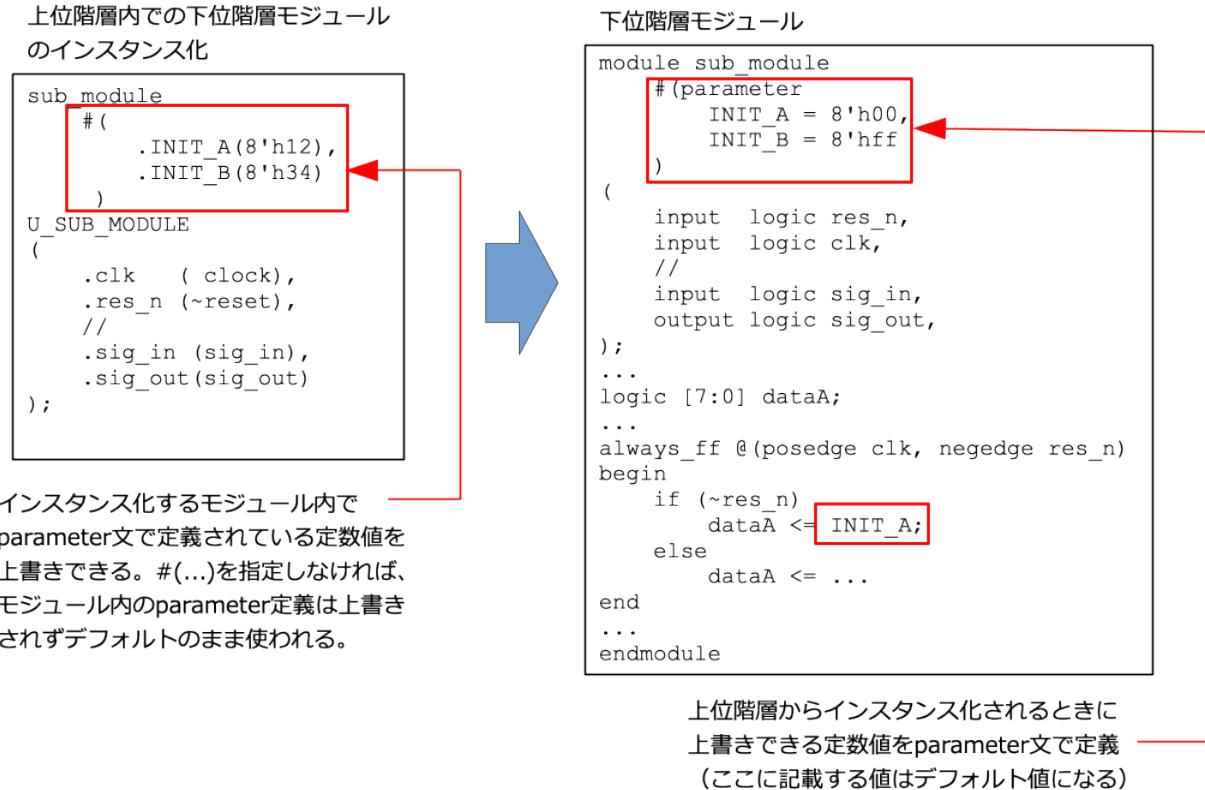


図 A.42: 定数パラメータの上書き機能

```
$ git clone https://github.com/munetomo-maruyama/simpleCounter.git
```

少なくとも以下のファイルがあることを確認してください。

simpleCounter/	
README.md	
RTL	RTL記述格納ディレクトリ
top.sv	simCounter本体のRTL記述
SIM	論理シミュレーション実行用ディレクトリ
flist.txt	ファイル・リスト
go_sim	実行スクリプト
tb.gtkw	波形ビューワの表示フォーマット
tb.sv	テストベンチ

A.22.3 論理機能モジュールを記述する

SystemVerilog の RTL で simpleCounter の論理機能モジュール TOP を記述します。ディレクトリ RTL の下に図 A.43 の内容を top.sv というファイル名で用意します。内部に 8 ビット幅のカウンタ count があり、以下の機能があります。

- **ロード機能:** 入力信号 LOAD で入力データ DIN をカウンタ count にロードする。
- **カウント・アップ:** 入力信号 UP でカウンタ count をカウント・アップする。カウンタ count が最大値になっている時に入力信号 UP でカウント・アップすると出力信号 OVF(オーバフロー) が 1 サイクル出される。
- **カウント・ダウン:** 入力信号 DN でカウンタ count をカウント・ダウンする。カウンタ count が最小値になっている時に入力信号 DN でカウント・ダウンすると出力信号 UDF(アンダーフロー) が 1 サイクル出

力される。

- **カウンタ値出力:** カウンタ count の値をそのまま出力データ DOUT から出力する。

```

//-----+
// Top Module
//-----+
module TOP
(
    input  logic CLK,
    input  logic RES,
    //-----+
    input  logic [7:0] DIN,   // data input
    input  logic LOAD, // data load
    input  logic UP,    // count up
    input  logic DN,    // count down
    //-----+
    output logic [7:0] DOUT, // counter out
    output logic OVF, // over flow
    output logic UDF // under flow
);

//-----+
// Internal Signals
//-----+
logic [7:0] counter;
logic        counter_min;
logic        counter_max;

//-----+
// Data Output
//-----+
assign DOUT = counter; // 8bit data

//-----+
// Main Counter
//-----+
always_ff @ (posedge CLK, posedge RES)
begin
    if (RES)
        counter <= 8'h00;
    else if (LOAD)
        counter <= DIN;
    else if (UP)
        counter <= counter + 8'h01;
    else if (DN)
        counter <= counter - 8'h01;
end

```

モジュールTOPと入出力信号の定義
 リセット入力信号 res_n
 クロック入力信号 clk
 カウンタへのロード信号 din[7:0]
 カウンタへのロード指示信号 load
 カウンタ・アップ指示信号 up
 カウンタ・ダウン指示信号 dn
 カウンタの出力信号 dout[7:0]
 カウンタ・オーバーフロー出力 ovf
 カウンタ・アンダーフロー出力 udf

カウンタ内部信号の定義
 カウンタ本体 counter[7:0]
 カウンタ値が8'h00になっていたらになる信号 counter_min
 カウンタ値が8'hffになっていたらになる信号 counter_max

カウンタ出力信号は
 カウンタ本体そのものの値

モジュール内部信号の定義
 カウンタ本体 counter[7:0]
 カウンタ値が8'h00になっていたらになる信号 counter_min
 カウンタ値が8'hffになっていたらになる信号 counter_max

モジュール内部信号の定義
 カウンタ本体 counter[7:0]
 カウンタ値が8'h00になっていたらになる信号 counter_min
 カウンタ値が8'hffになっていたらになる信号 counter_max

```

//-----+
// Overflow Pulse
//-----+
assign counter_max = (counter == 8'hff);
//
always_ff @ (posedge CLK, posedge RES)
begin
    if (RES)
        OVF <= 1'b0;
    else if (counter_max & UP)
        OVF <= 1'b1;
    else if (OVF)
        OVF <= 1'b0;
end

//-----+
// Underflow Pulse
//-----+
assign counter_min = (counter == 8'h00);
//
always_ff @ (posedge CLK, posedge RES)
begin
    if (RES)
        UDF <= 1'b0;
    else if (counter_min & DN)
        UDF <= 1'b1;
    else if (UDF)
        UDF <= 1'b0;
end

```

オーバーフロー信号ovfを生成
 非同期リセットで1'b0にクリア
 counter_maxが1でかつup指示があればセット
 -ovfが1にセットされいたらクリア
 -ovfのパルス幅は1サイクルになる

アンダーフロー信号udfを生成
 非同期リセットで1'b0にクリア
 counter_minが1でかつdn指示があればセット
 -udfが1にセットされいたらクリア
 -udfのパルス幅は1サイクルになる

モジュール定義の終了

図 A.43: simpleCounter の論理機能モジュール top.sv

A.22.4 テストベンチを記述する

次に simpleCounter を論理シミュレーションするためのテストベンチを記述します。ディレクトリ SIM の下に図 A.44 の内容を tb.sv というファイル名で用意します。機能検証用の入力パターンは、データ・ロードやカウンタのインクリメントなど個々の要素ごとにタスク文で用意しておいて、入力パターン全体は initial 文の中からタスク文を呼び出して生成しています。複雑な入力パターンも、タスク文を使うことすっきりと記述することができます。

A.22.5 論理シミュレーション実行のための準備

論理シミュレーション実行のために以下のファイルを準備します。全てディレクトリ SIM 以下に格納します。

- **list.txt**(リスト A.1): 使う RTL 記述のリスト。
- **go_sim**(リスト A.2): シミュレーション起動用のスクリプト。
- **tb.gtkw**: 波形ビューワ GtKWave が書き出し、または読み込む波形表示設定ファイル。GtKWave で波形表示した時のフォーマットをこのファイルに保存する。

リスト A.1: 使う RTL 記述のリスト

```
+includir+${DIR_SIM}
+includir+${DIR_RTL}
```

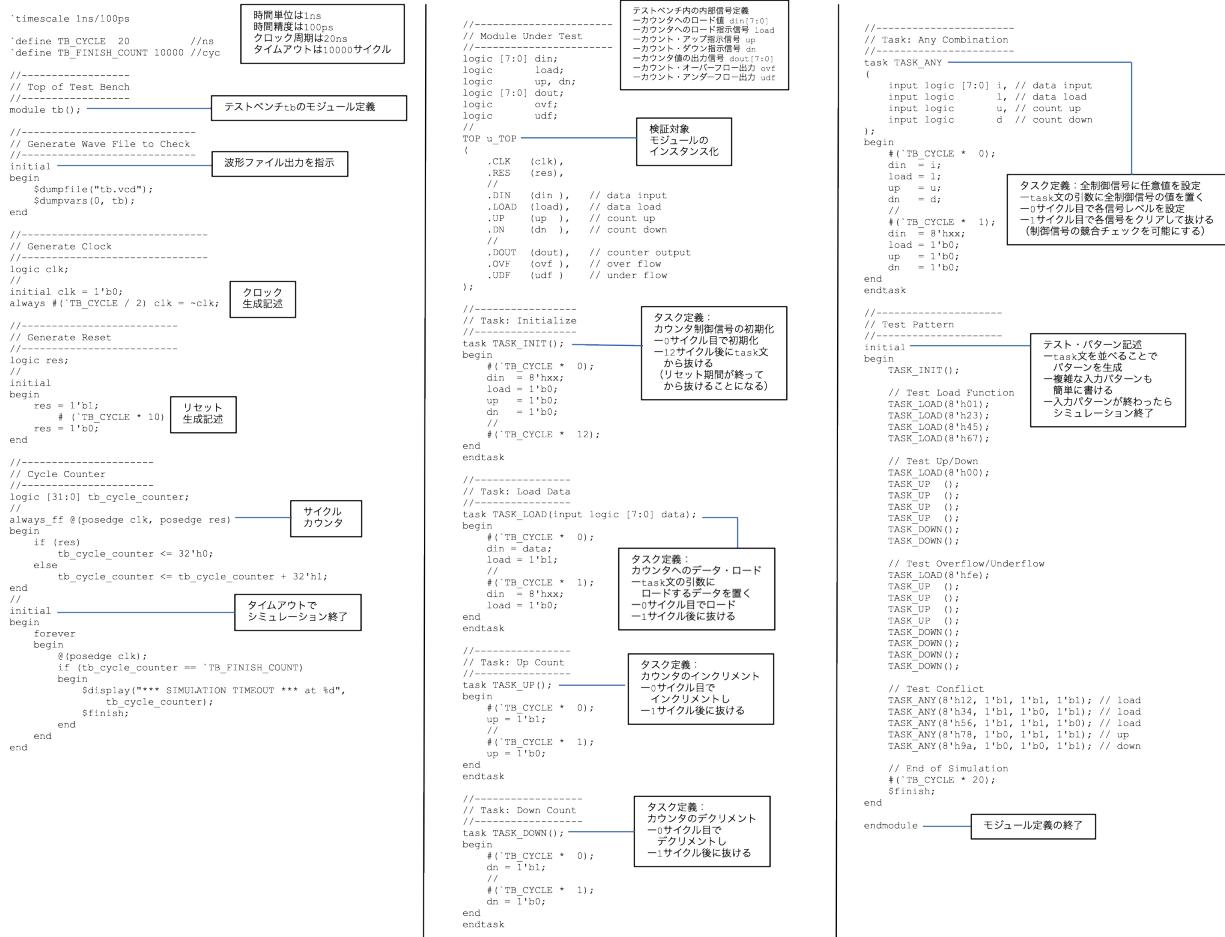


図 A.44: simpleCounter のテストベンチ tb.sv

```
`${DIR_SIM}/tb.sv
`${DIR_RTL}/top.sv
```

リスト A.2: シミュレーション起動用のスクリプト

```
#!/bin/bash

export DIR_SIM=
export DIR_RTL=../RTL

iverilog -v -g2012 -o tb.vvp \
-c ${DIR_SIM}/clist.txt -s tb

vvp tb.vvp > log
```

A.22.6 論理シミュレーションの実行

ファイル go_sim には chmod で実行権限を与えてください。ディレクトリ SIM の下で下記のコマンドを実行すると論理シミュレーションを起動できます。

```
$ ./go_sim
```

Icarus Verilog がシミュレーション実行するとき、まずコマンド iverilog で RTL 記述をコンパイルしてファ

イル tb.vvp を生成します。その後、このファイルをコマンド vvp に入力してシミュレーション本体を実行します。シミュレーション実行により波形ファイル tb.vcd が生成され、メッセージ出力がログ・ファイル log に格納されます。

A.22.7 波形ファイルの確認

波形ファイルの確認は下記のコマンドで行います。また波形表示設定ファイルがない場合は

```
$ gtkwave tb.vcd
```

波形表示設定ファイル tb.gtkw がある場合は

```
$ gtkwave tb.vcd tb.gtkw
```

を実行します。波形の表示例を図 A.45 に示します。左上の論理階層をクリックすると左下にその階層の信号が表示されるので、Append ボタンや Insert ボタンで右側の波形表示領域に追加できます。現在表示されている波形とその順番などは File メニューの Write Save File As により波形表示設定ファイルにセーブできます。画面上部のボタンで時間方向の拡大・縮小・移動などができます。あちこち触ってみて使い方を習得してください。

入力信号 LOAD、DIN、UP、DN の動きと、カウンタ count の動きを確かめてください。また、出力信号 OVF、UDF や DOUT の動きも確認しましょう。

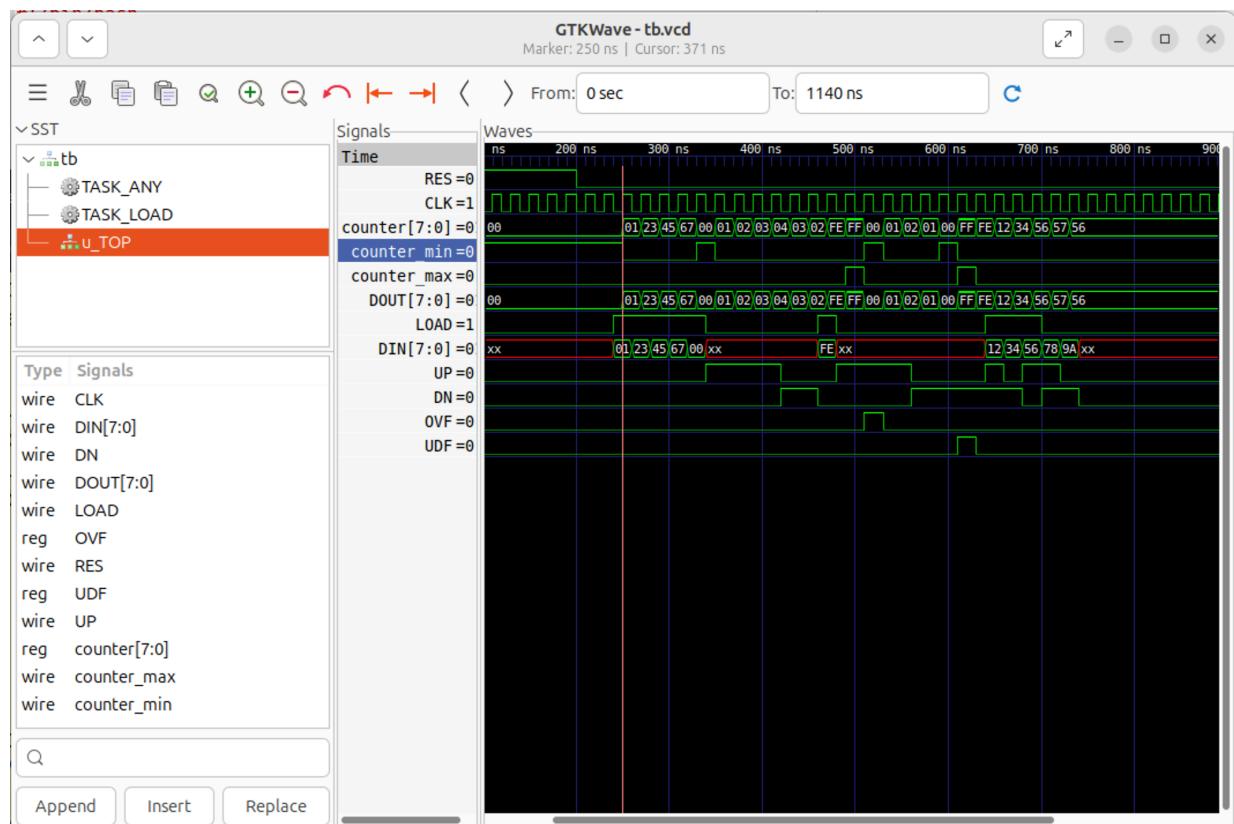


図 A.45: GtakWave で表示したシミュレーション波形

A.23 設計例：簡易 CPU “picoCPU”

A.23.1 チップ設計の醍醐味！ CPU の論理設計

論理設計の王道は自分で CPU を作ることです。ここではシンプルな命令セットの picoCPU を定義して、ハードウェア記述言語 SystemVerilog での RTL 記述方法を解説し、論理シミュレーションで動きを追ってみましょう。ここで設計する CPU は、たった 4 個の命令を持つ 8 ビット CPU で、得意技は L チカです。

A.23.2 picoCPU の命令セット・アーキテクチャ

picoCPU の命令セット・アーキテクチャを説明します。プログラマから見える CPU の内部リソースをプログラマーズ・モデルと呼び、picoCPU の場合を図 A.46 に示します。6 ビットのプログラム・カウンタ PC と 8 ビットのデータ格納用 A レジスタだけを持っています。この CPU のメモリは 6 ビット幅アドレスでアドレッシングできる 64 バイト空間であり、命令とデータは同一空間に置きます。リセット後は、PC が 0 に初期化され、0 番地の命令から処理を開始します。



図 A.46: picoCPU のプログラマーズ・モデル

命令セットを表 A.7 に示します。全て 8 ビット幅のコードで上位 2 ビットがオペコードでこの命令が何をやるかを示しています。下位 6 ビットはオペランドで処理の相手を示しています。加算命令 ADD は A レジスタに命令コードの下位 6 ビットを符号拡張して加算します。条件分岐命令 JNZ は A レジスタが 0 でなければ PC に命令コードの下位 6 ビットを転送しジャンプします。A レジスタが 0 ならジャンプせず次の命令に進みます。ロード命令 LDA は命令コードの下位 6 ビットをアドレスとするメモリからデータをリードして A レジスタに格納します。ストア命令 STA は命令コードの下位 6 ビットをアドレスとするメモリに A レジスタの内容をライトします。

表 A.7: picoCPU の命令セット

種類	オペコード	命令	動作
加算	00iiiiii	ADD #imm6	$A \leftarrow A + imm6$ (符号拡張)
条件分岐	01aaaaaaaa	JNZ @addr6	if ($A \neq 0$) $PC \leftarrow addr6$
ロード	10aaaaaaaa	LDA @addr6	$A \leftarrow MEM[addr6]$
ストア	11aaaaaaaa	STA @addr6	$MEM[addr6] \leftarrow A$

A.23.3 メモリ空間と I/O ポート

メモリ・マップを図 A.47 に示します。メモリ空間は全部で 64 バイトあり、最後の 2 バイトは I/O ポートの制御レジスタとしてアサインしました。8 ビット幅の独立した入力ポート PORTI と出力ポート PORTO です。ポート PORTI の入力レベルはアドレス 0x3e 番地からリードでき、ポート PORTO の出力レベルはアドレス 0x3f 番地へのライトで設定できます。

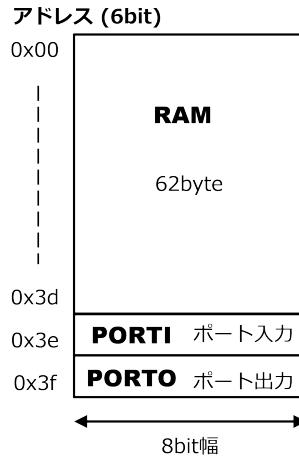


図 A.47: picoCPU のメモリ・マップ

A.23.4 picoCPU システムのブロック図

図 A.48 に picoCPU システムのブロック図を示します。最上位階層 TOP の下に CPU と MEM の各ブロックが入ります。MEM は図 A.47 のメモリ・マップを構成するメモリ本体と I/O ポートです。メモリは RAM で構成しますが、論理シミュレーション実行時はプログラム・コードで内容を初期化してから動かします。

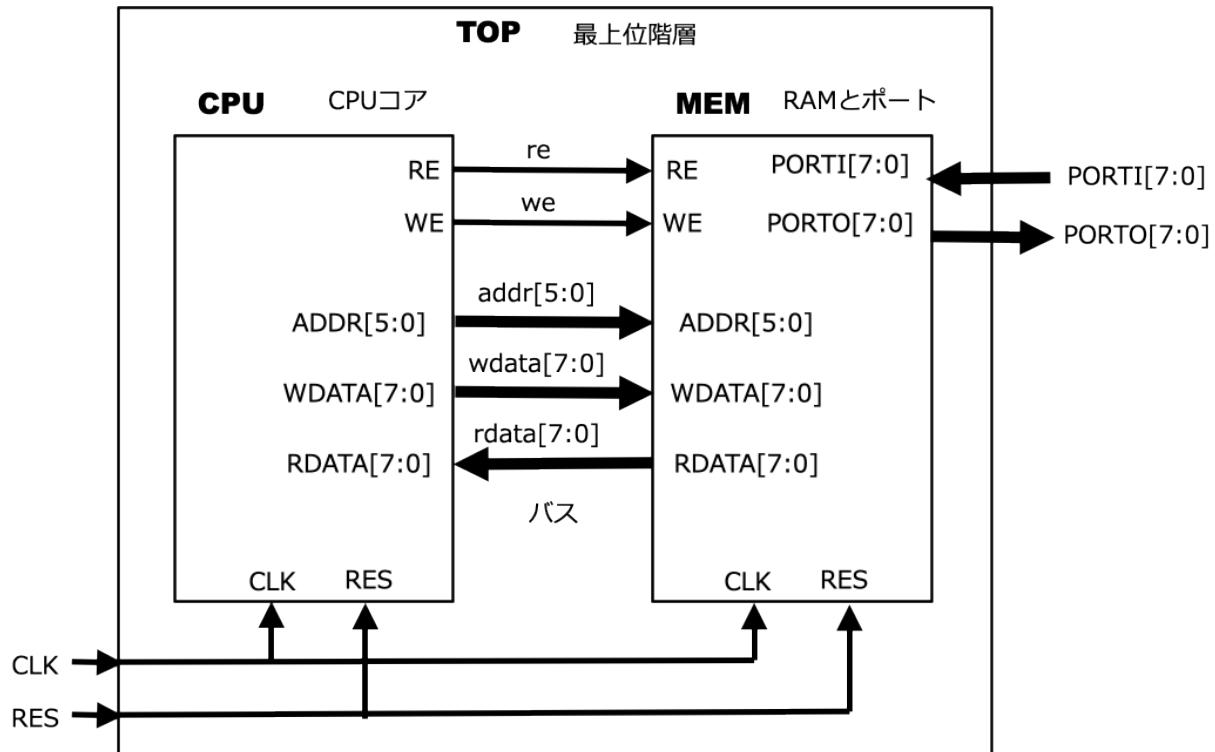


図 A.48: picoCPU システムのブロック図

A.23.5 バス・アクセスのタイミング

一般的なシステムでは、CPU に複数のメモリや周辺機能がバス経由で接続されます。本稿では CPU に接続されるのは MEM モジュールだけですが、その間はやはりバスで結びます。バスのタイミングを図 A.47 に示します。

ライト・アクセスは時刻 T2 からライト・イネーブル WE、アドレス ADDR、ライト・データ WDATA を確定させ 1 サイクルで完了します。

リード・アクセスは時刻 T4 からリード・イネーブル RE とアドレス ADDR を確定すると次の時刻 T5 からリード・データ RDATA を出力します。ウェイトする機能は無しにしました。

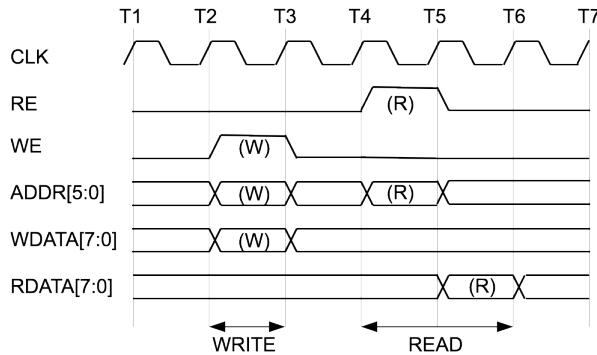


図 A.49: picoCPU システムのバス・タイミング

A.23.6 設計ファイルの用意

ファイル一式は下記コマンドでダウンロードできます。

```
$ git clone https://github.com/munetomo-maruyama/picoCPU.git
```

少なくとも以下のファイルがあることを確認してください。

picoCPU/	
RTL	RTL記述格納ディレクトリ
cpu.sv	CPUのRTL記述
mem.sv	MEMのRTL記述
top.sv	最上位TOP階層のRTL記述
SIM	論理シミュレーション実行用ディレクトリ
flist.txt	ファイル・リスト
go_sim	実行スクリプト
tb.gtkw	波形ビューワの表示フォーマット
tb.sv	テストベンチ

A.23.7 最上位階層 TOP モジュールの SystemVerilog 記述

最上位階層 TOP モジュールの RTL 記述 top.sv を図 A.50 に示します。外部との入出力信号は、クロック CLK とリセット RES(正論理) 以外は、8 ビット幅のポート入出力 PORTI と PORTO です。TOP 階層内では、モジュール CPU と MEM をインスタンス化しています。

A.23.8 CPU モジュールの SystemVerilog 記述

CPU モジュールの RTL 記述 cpu.sv を図 A.51 に示します。論理回路を設計するときは、データ関係の流れを記述するデータ・バス部と、それをコントロールする制御部に分けると見通しがよくなります。

picoCPU も同様な構成を採りました。前半のデータ・バス部は CPU 内のリソース (PC や A レジスタなど) と、それらの制御信号など、データを流す構造だけ定義し、細かい制御は制御部に任せます。

制御部は、ステート・マシンで構成しています。picoCPU のステートは 8 個 (3 ビット幅の state) で構成でき、記述の冒頭で `define 文で定義しています。state はクロック CLK の立ち上がりの度に変化させ、次の

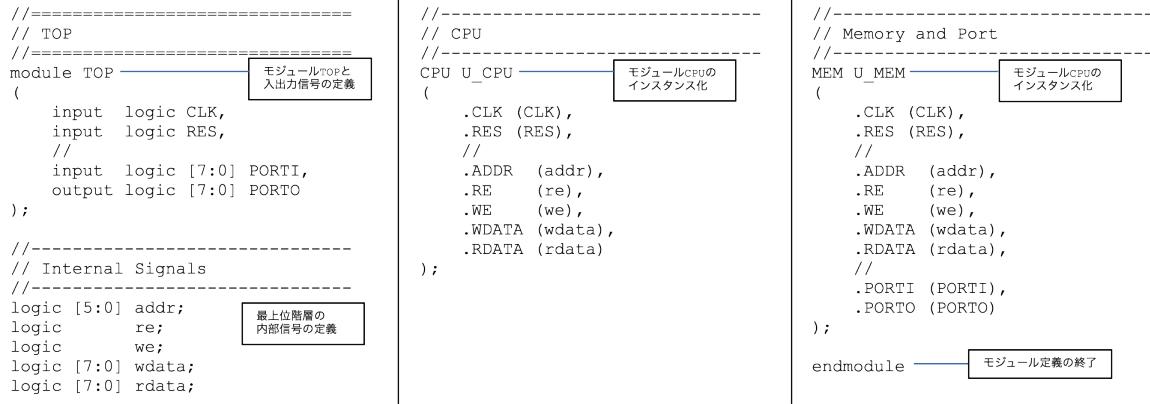


図 A.50: 最上位階層 TOP モジュールの RTL 記述 top.sv

state は state_next に用意します。state_next は、現 state と各種信号から always_comb で記述した組み合わせ回路で生成します。併せてデータ・パスを制御する信号も生成します。

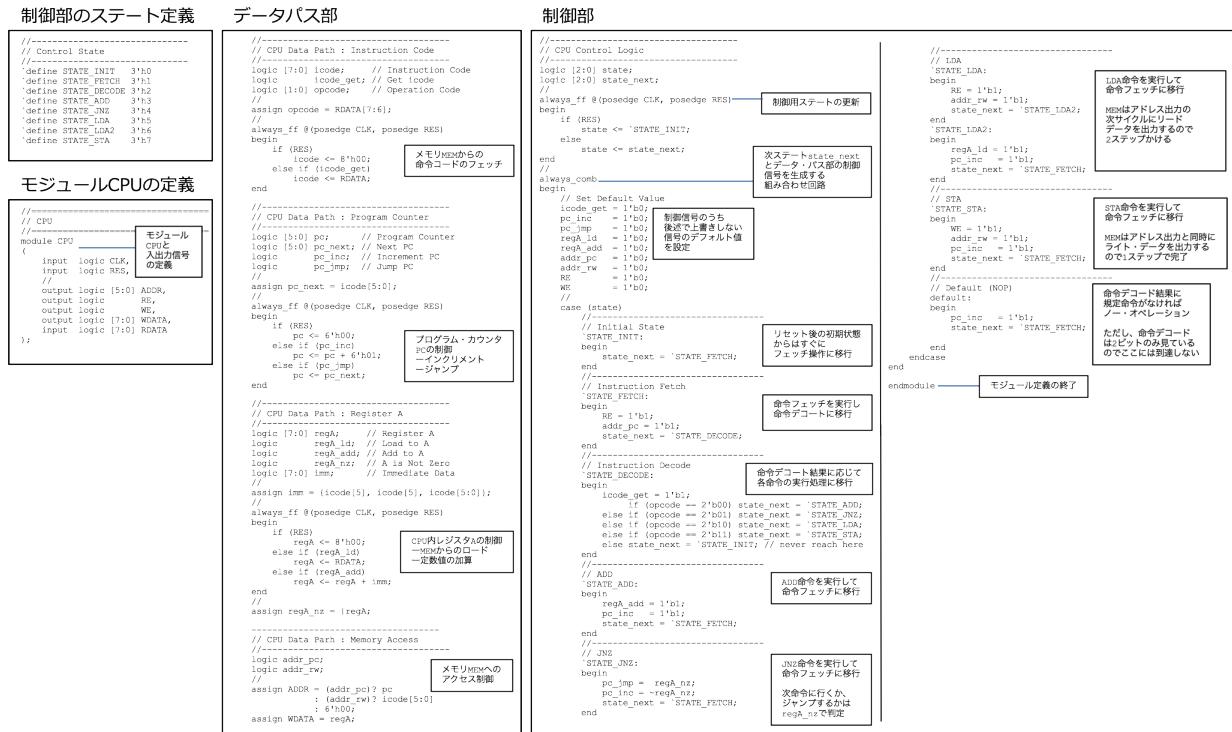


図 A.51: CPU モジュールの RTL 記述 cpu.sv

A.23.9 MEM モジュールの SystemVerilog 記述

メモリ内のプログラムは専用開発環境がないのでハンド・アセンブルで作成します。プログラム例をリスト A.3 に示します。この動作は、PORTI 入力 (アドレス 0x3e) が 0x00 なら PORTO 出力 (アドレス 0x3f) を一定間隔でインクリメントし、PORTI 入力が 0x00 以外なら PORTO 出力の変化を止めます。アドレス 0x20 と 0x21 のデータは定数で、アドレス 0x21 の方は PORTO の変化間隔を決めます。もし PORTO の出力のどれかを LED に接続すれば Lチカ (blinkLED) になります。このリストに従ってメモリ内容を initial 文で初期化します。

リスト A.3: picoCPU のプログラム例 blinkLED

```

-----  

ADDR CODE LABEL MNEMONIC  

-----  

0x00 0xbe START LDA @PORTI  

0x01 0x40 JNZ @START  

0x02 0xa1 INIT LDA @TEN  

0x03 0x3f LOOP ADD #-1  

0x04 0x43 JNZ @LOOP  

0x05 0xbf OUT LDA @POUT  

0x06 0x01 ADD #1  

0x07 0xff STA @POUT  

0x08 0xa0 RESTART LDA @ONE  

0x09 0x40 JNZ @START  

0x20 0x01 ONE CONST 0x01  

0x21 0x0a TEN CONST 0X0a  

0x3e 0x00 PORTI DATA 0x00  

0x3f 0x00 PORTO DATA 0x00  

-----  


```

MEM モジュールの RTL 記述 mem.sv を図 A.52 に示します。メモリは 64 バイトの RAM で配列変数 logic [7:0] mem[0:63] で定義しています。アドレス 0x3e(62) と 0x03f(63) はポート側のロジックで置き換えていきます。

```

//=====  

// Memory with Port I/O  

//=====  

module MEM
  //=====  

  // モジュールMEMと  
入出力信号の定義  

  //=====  

  input  logic CLK,  

  input  logic RES,  

  //  

  input  logic [5:0] ADDR,  

  input  logic      RE,  

  input  logic      WE,  

  input  logic [7:0] WDATA,  

  output logic [7:0] RDATA,  

  //  

  input  logic [7:0] PORTI,  

  output logic [7:0] PORTO  

);  

// Memory Mat  

logic [7:0] mem[0:63];  

//=====  

// メモリmemの定義  

// (8ビット幅・64ワード)  

//=====  

// Initialize Memory  

initial  

  //=====  

  // メモリmemの初期化  

begin  

  mem[ 0]=8'hbe; mem[ 1]=8'h40; mem[ 2]=8'hal; mem[ 3]=8'h3f;  

  mem[ 4]=8'h43; mem[ 5]=8'hbf; mem[ 6]=8'h01; mem[ 7]=8'hff;  

  mem[ 8]=8'h00; mem[ 9]=8'h40;  

  mem[32]=9'h01; mem[33]=8'h0a;  

end

```

```

// Read Operation  

always_ff @(posedge CLK, posedge RES)  

begin  

  if (RES)  

    RDATA <= 8'h00;  

  else if (RE & (ADDR == 6'h3e))  

    RDATA <= PORTI; // PORT Input  

  else if (RE & (ADDR == 6'h3f))  

    RDATA <= PORTO; // PORT Output  

  else if (RE)  

    RDATA <= mem[ADDR]; // Memory  

end

// Write Operation  

always_ff @(posedge CLK)  

begin  

  if (WE) mem[ADDR] <= WDATA; // Memory  

end
//  

always_ff @(posedge CLK, posedge RES)  

begin  

  if (RES)  

    PORTO <= 8'h00;  

  else if (WE & (ADDR == 6'h3f))  

    PORTO <= WDATA; // Port Output  

end

```

メモリmemとポートのリード制御
アドレスが0x3eならPORTIの値
アドレスが0x3fならPORTOの値を返す

メモリmemのライト制御
ポートのライト制御
ライトで値を変えるのはPORTOのみ

モジュール定義の終了

図 A.52: MEM モジュールの RTL 記述 mem.sv

A.23.10 論理シミュレーション実行のための準備

論理シミュレーション実行のために以下のファイルを準備します。全てディレクトリ SIM 以下に格納します。

- **flist.txt**(リスト A.4): 使う RTL 記述のリスト。
- **go_sim**(リスト A.5): シミュレーション起動用のスクリプト。
- **tb.gtkw**: 波形ビューワー GtkWave が書き出し、または読み込む波形表示設定ファイル。GtkWave で波形表示した時のフォーマットをこのファイルに保存する。

リスト A.4: 使う RTL 記述のリスト

```
+includir+${DIR_SIM}  
+includir+${DIR_RTL}
```

```
`${DIR_SIM}/tb.sv
`${DIR_RTL}/cpu.sv
`${DIR_RTL}/mem.sv
`${DIR_RTL}/top.sv
```

リスト A.5: シミュレーション起動用のスクリプト

```
#!/bin/bash

export DIR_SIM=.
export DIR_RTL=../RTL

iverilog -v -g2012 -o tb.vvp \
-c ${DIR_SIM}/clist.txt -s tb

vvp tb.vvp > log
```

A.23.11 論理シミュレーションの実行

ファイル go_sim には chmod で実行権限を与えてください。ディレクトリ SIM の下で下記のコマンドを実行すると論理シミュレーションを起動できます。

```
$ ./go_sim
```

Icarus Verilog がシミュレーション実行するとき、まずコマンド iverilog で RTL 記述をコンパイルしてファイル tb.vvp を生成します。その後、このファイルをコマンド vvp に入力してシミュレーション本体を実行します。シミュレーション実行により波形ファイル tb.vcd が生成され、メッセージ出力がログ・ファイル log に格納されます。

A.23.12 波形ファイルの確認

波形ファイルの確認は下記のコマンドで行います。また波形表示設定ファイルがない場合は

```
$ gtkwave tb.vcd
```

波形表示設定ファイル tb.gtkw がある場合は

```
$ gtkwave tb.vcd tb.gtkw
```

を実行します。波形の表示例を図 A.53 に示します。左上の論理階層をクリックすると左下にその階層の信号が表示されるので、Append ボタンや Insert ボタンで右側の波形表示領域に追加できます。現在表示されている波形とその順番などは File メニューの Write Save File As により波形表示設定ファイルにセーブできます。画面上部のボタンで時間方向の拡大・縮小・移動などができます。あちこち触ってみて使い方を習得してください。

リスト A.3 に示すプログラムの通り、PORTI 入力に対する PORTO の動作が期待通りであることがわかります。PORTI が 0x00 の期間、PORTO がインクリメントされています。

A.23.13 CPU 設計にチャレンジしよう

独自命令でも RISC-V 命令セットでも、CPU を設計すると多くのことを学べるし、動いたときの達成感は何ものにも代えがたいものがあります。ぜひ挑戦してみてください。

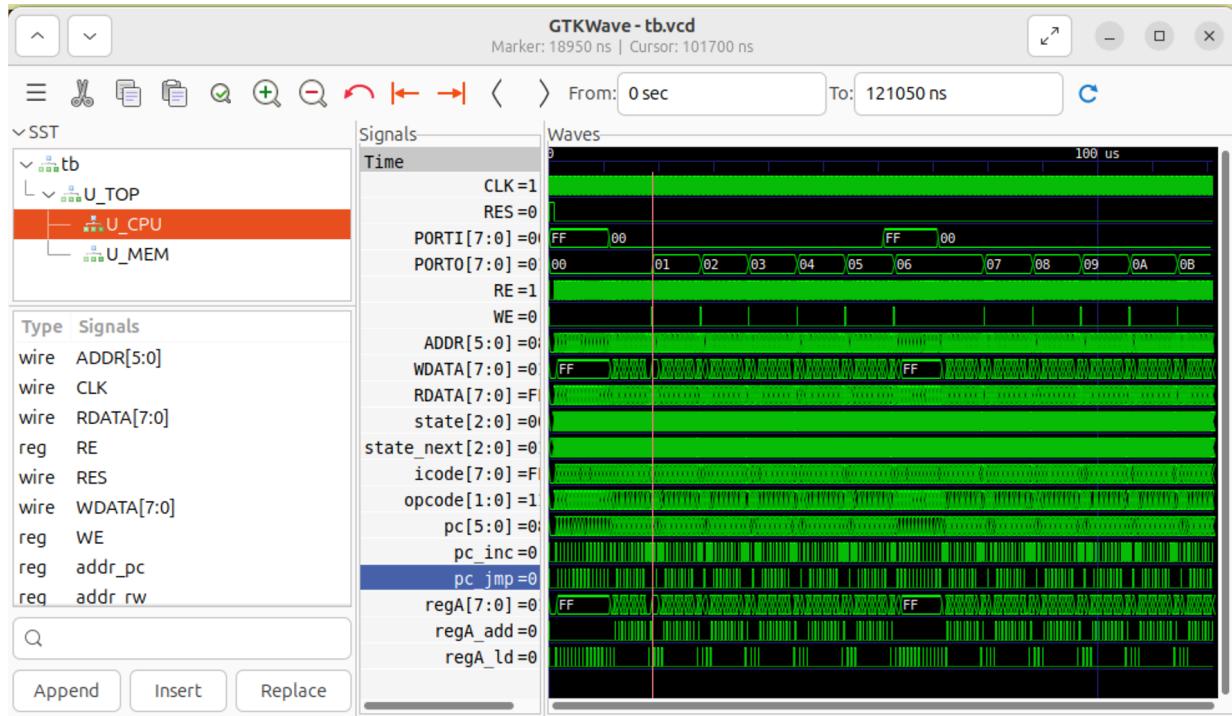


図 A.53: GtKWave で表示したシミュレーション波形

A.24 論理設計者の心得

論理設計の道具として後述するハードウェア記述言語 (HDL : Hardware Description Language) が登場してから、しばしば論理設計がソフトウェア設計化したという話が出る事がありますが、この両者は全く異なるものです。以下、かなりオッサンのつぶやきに近い感のある話を書きますが、いずれも重要なことなので、しっかり意識してください。

1. ソフトウェアのコードは書いただけ ROM を消費しますが、すでに存在している CPU 内の演算リソースを使い回すだけで論理ハードウェアが増えるわけではありません。一方、ハードウェア記述言語のコードは、書いた分だけハードウェアの物量になってしまいます。演算子一つ書くだけで演算器が生成されることになります。+ (加算) 程度なら小さいですが、* (乗算) の 1 文字をあちらこちらに書くとハード量を消費する乗算器が大量に生成されます。ハードウェア記述言語は論理回路をある程度抽象化できるので便利ですが、1 行ずつ書いたコードがどのようなハードウェアに落とし込まれるのか常に意識することが重要です。そのためコーディングの前に、ハードウェア方式設計段階で全体構造とその規模を把握するためのしっかりしたブロック図や論理機能図を書く事が大切になってきます。
2. C 言語で書いたコードは逐次的に実行されますが、ハードウェア記述言語で書いたコードは全体が同時並行に実行されます。ハードウェア設計では常に全体を俯瞰して物を考える必要があるのです。
3. 論理ハードウェアは ASIC でも FPGA でも、最後は 2 次元のシリコン内に展開されるものです。ハードウェア全体の実現方式 (ブロック分割など) を検討するときに、2 次元面内で効率的に情報や信号が流れることを意識すべきです。FPGA 設計でもその方が内部リソースの利用効率が高くなります。
4. 論理回路も最後はトランジスタから構成される電子回路 = アナログ回路になります。その物理的な挙動をしっかり理解することが重要です。回路の電源電圧、デバイスからの出力信号のレベル (High 時の出力レベル = VOH、Low 時の出力レベル = VOL)、デバイスへの入力信号レベル (High 検知の最小レベル = VIH、Low 検知時の最大レベル = VIL)、信号遅延時間、シグナル・インテグリティ (信号品質の確保 =

（リギング、反射、クロストークなどによる信号劣化の防止）などを考慮します。さらに、信頼性への配慮も必要です。例えば、立上がり時間や立下がり時間がなまった信号を CMOS デジタル回路に入力すると、電源-GND 間の貫通電流が流れる期間が長くなるため、メタル配線を劣化（エレクトロマイグレーション）させたり、ホットエレクトロン注入による MOS トランジスタの特性（スレシホールド電圧）変動を招きます。ファンイン、ファンアウト、配線長、最大負荷容量などにきちんと制約を掛けた設計が必要になります。

5. ソフトウェアのバグは避けるべきですが、FLASH メモリやハードディスクに格納されているのであれば、あとから更新することが可能です。一方、ハードウェアの場合、FPGA であればフィールドで書き換えることはできますが、ASIC の場合はもうシリコンになってしまっているので、市場に出れば更新できません。このため、バグれない！ というプレッシャーがあります。論理設計ではその機能検証が非常に重要で、いかに網羅的な検証を行うかをしっかり考える必要があります。ただ、多くの波形をもれなくチェックすることは困難です。そのために検証方法（道具立て）を良く考へる必要があります。チェック付きのモデルや、ポイントを押さえたログ出力などを用意します。あるいはアサーション・ベース検証という、自動波形チェックのような方法もあります。しかしこうした検証環境にもかなりのコード量が必要になるので、そちらにバグがあると検証漏れを引き起こす可能性もあります。だんだんと何を検証しているのかわからなくなることもあるので注意が必要です。
6. しかし、もっと重要なことがあります。それは初期の仕様設計から方式設計、コーディング、詳細論理設計までに至る、作り込み工程での設計品質です。この初期の段階からしっかりと全体を見通して不安感なく自信を持てる設計、すなわち「機能検証に頼らずとも問題ないほどの高品質設計」を行うことが大切なのです。もちろんきちんと網羅的な機能検証は必須ですが、そこで検討不足や確認不足による低レベルなバグが出てくるようでは非常にまずいと思ったほうがいいです。そうは言っても人間はミスするもの。どうするか？ まず基本は与えられた問題や課題の本質をきちんと理解して単純化して扱いやすい単位に分割することです。また、自分がどこでミスしやすいかを意識すること、そのミスを防ぐための自分なりの仕掛けを用意することも重要です。ふと心をよぎった不安感や疑問は意外に重要な問題をはらむことがあります、必ず確認と解決をすることも大切です。とにかくいかにして高品質な設計を作る込むのかの熟慮に時間を掛けてください。すごく優秀なエンジニアほど、実作業の前の事前検討に時間をかけ、かつミスを防ぐため作業中および作業後は、愚直で細かな確認やチェックを怠らないものです。
7. 論理設計者の責任は非常に重いです。論理バグがあると、特に ASIC 設計の場合、論理設計以降の工程（レイアウト設計、ウェハ試作、機能評価、特性評価、テスティング立ち上げ、量産品製造）の全てに渡って手戻りが発生してしまいます。後ろの工程になるほど人手と費用が桁違いに増えます。もちろんユーザやソフト開発者にも影響が及びます。訴訟や補償の話になることもあるほどです。設計の早い段階になるほどその責任は重くなりますので、前項で述べた設計品質の確保はとても大切なことです。

Turing Machine を題材とする自前チップ設計試作

Tiny Tapeout で自作 CPU をシリコン化しよう

2026 年 3 月 5 日 ver 1.0 (RISC-V Day Tokyo 2026 Spring)

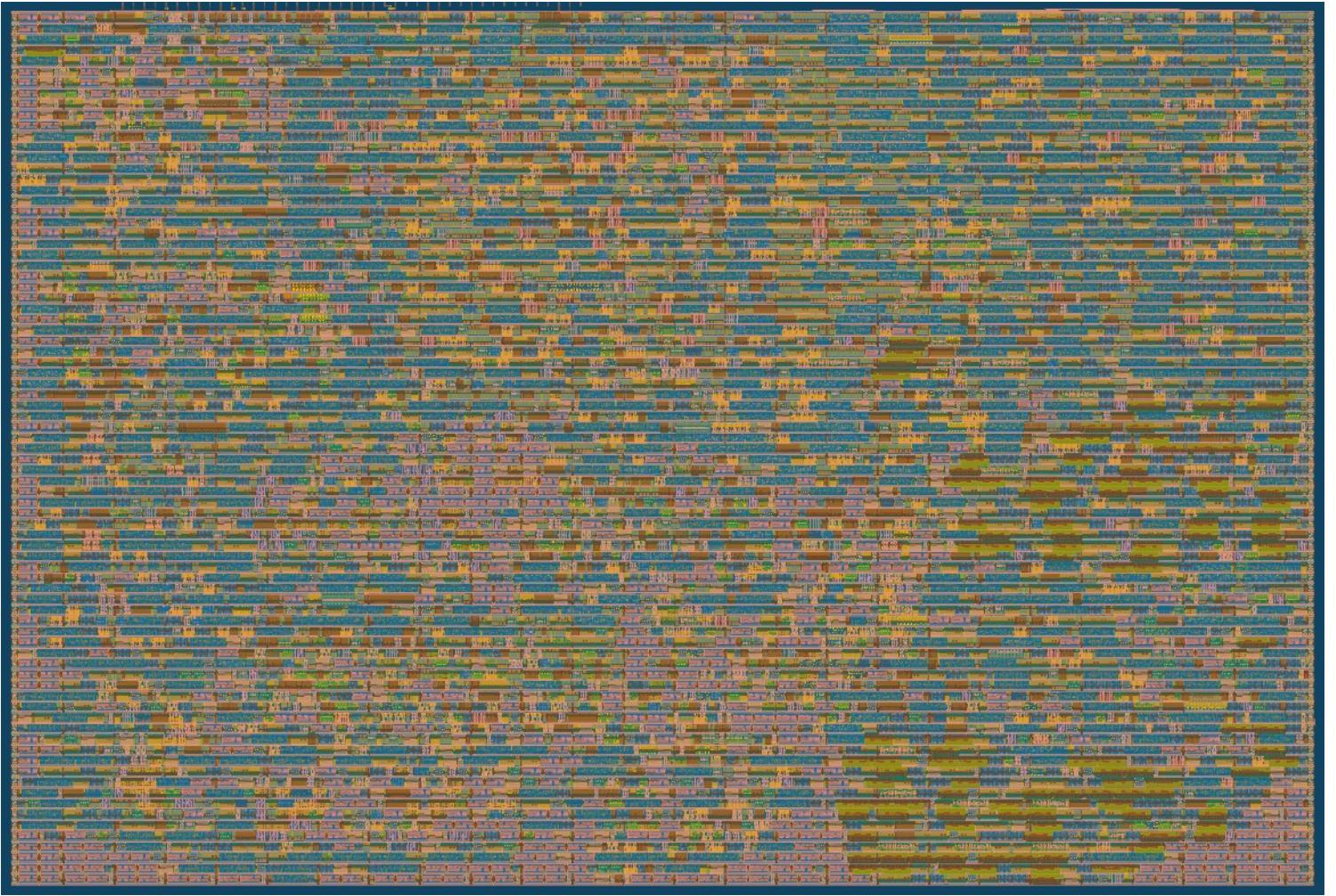
著 者 圓山宗智

発行者 圓山宗智

連絡先 munetomo@ishi-kai.org

© 2026 圓山宗智

(powered by Re:VIEW Starter)



tt_um_bfcpu (tt_um_bfcpu)

SELECTION:
tt_um_bfcpu (tt_um_bfcpu)
_3568 ↗ (sky130_fd_sc_hd_o211a_2)

KEYS
1: Hide Fill, Decap, Tap cells
2: Hide top cell geometry
3: Isolate selection / back
4: Zoom to selection

