AN82156

UDB データパスを用いた PSoC Creator コンポーネントの設計

著者: Todd Dust および Greg Reynolds

関連プロジェクト: あり

関連製品ファミリー: CY8C3xxx, CY8C5xxx, CY8C42xx, CY8C6xxx

ソフトウェアバージョン: PSoC® Creator™ 4.2

すべての関連資料の一覧については、ここをクリックしてください。

AN82156 は、PSoC ユニバーサル デジタル ブロック (UDB) データパスを用いた PSoC Creator コンポーネントの設計方法を説明します。データパス ベースのコンポーネントは、カウンターや PWM、シフター、UART、SPI などの一般的な機能を実装することができます。これらはカスタム デジタル ペリフェラルを作成し、CPU の負担を軽減するためにデータ管理のタスクを実行するためにも使用できます。本書はまた、データパス インスタンスの作成、表示、修正用に PSoC Creator UDB エディター ツールの使用方法についても説明します。

目次

1 はじめに

PSoC® 3、PSoC 4、PSoC 5LP、および PSoC 6 (以下 PSoC という) は単なるマイクロコントローラーではありません。PSoC により、マイクロコントローラー、コンプレックスプログラマブルロジックデバイス (CPLD) および高性能アナログの機能を統合し、他にはない柔軟性を得ることができます。これにより、費用や基板面積、消費電力、開発時間を低減できます。

CPLD 設計を独立した CPLD から PSoC PLD へ移植するのは、Verilog コードのコピー & ペーストと同じくらい容易です。AN82250 は、移植のやり方の順を追った説明を提供しています。

ただし、PSoC の PLD はほとんどの CPLD または FPGA より小さく、多くのデザインが大きすぎて PSoC に移植することができません。この課題を克服するために、ユニバーサル デジタル ブロック (UDB) データパスを使用できます。データパスは複雑な計算機能を実行するよう設計されます。これにより、PLD の負荷を軽減して PLD をステートマシンやグルーロジックとして使用することができます。そのため、データパスは Verilog コードのフットプリントを大幅に削減できます。

データパスの中心は 8 ビットの算術論理演算ユニット (ALU)です。ALU は加算、減算、OR、XOR、AND、インクリメント、デクリメント、シフトなどの機能を実行します。ALU に関連しているいくつかのレジスタおよび条件付き比較ブロックがあります。データパスはチェーン接続して 1～32 ビット幅の動作を実行できます。

データパスを PLD と併用することで、複雑なカスタム デジタルペリフェラルを作成できます。これらのペリフェラルは PSoC Creator コンポーネントに統合されています。現在、PSoC Creator は UDB データパスを用いた様々なデジタル コンポーネントを備えています。しかし、標準的なコンポーネントには希望する機能がサポートされないことがあります。本アプリケーションノートは、データパスを用いたユーザー独自のコンポーネントを作成する方法を説明します。このアプリケーションノートでは、主に UDB エディターを使用してカスタムコンポーネントを作成する方法について説明します。

これは上級のアプリケーションノートであり、読者が PSoC Creator でアプリケーションを開発することに精通していることを前提します。

PSoC に慣れていない場合、下記を参照ください。

- AN54181 Getting Started with PSoC 3
- AN79953 Getting Started with PSoC 4
- AN77759 Getting Started with PSoC 5LP
- AN221774 Getting Started with PSoC 6 MCUs

PSoC Creator に慣れていない場合、下記を参照ください。

- PSoC Creator のホームページ

本アプリケーションノートはまた、デジタル設計および Verilog の基礎知識を前提にしています。これらの概念に慣れていない場合、下記を参照してください。

- AN81623 PSoC Digital Design Best Practices
- KBA86336 Just Enough Verilog for PSoC
- AN82250 Implementing Programmable Logic Designs with Verilog.

このアプリケーションノートはまた、あなたが UDB エディターとそれがどのように機能するかに精通していることを前提としています。詳細については下記を参照してください。

- Universial Digital Block (UDB) Editor Guide

関連のあるデータパスの設計リソースのリストは、関連リソースをご覧ください。

データパスに精通していても、UDB エディターとデータパス設定ツールのどちらを使用するかを判断したい場合は、データパスベースのコンポーネントに進んでください。

あなたが UDB エディターを使いたいことがわかっているならば、プロジェクト#1 – 8 ビットダウンカウンターに進んでください。

データパス設定ツールを使用したい、またはパラレルイン/アウトを使用する必要があることがわかっている場合は、付録 A – データパスコンフィギュレーションツールの使用例に進んでください。

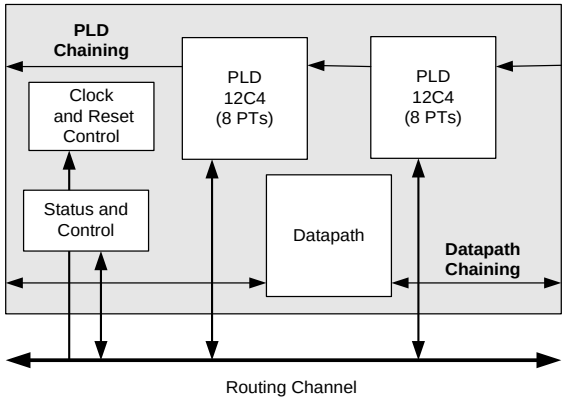
それ以外の場合は読み続けます。

2 従来の PLD と PSoC UDB の比較

PSoC は最適化されたプログラム可能なデバイスで、はるかに大きいプログラマブル ロジック製品の機能に匹敵するか、それよりも優れています。PSoC は大きな FPGA または CPLD の実装を直接統合するよう設計されていません。その代わりに、PSoC は小型、高速、低消費電力のプログラマブル デジタルブロック (UDB) のアレイを備えています。

図 1 に示すように、各 UDB は 2 個の小型 PLD、1 個のデータパス モジュールおよびステータスコントロールロジックで構成されています。

図 1. 簡略化された UDB のブロック図



データパスモジュールはインクリメント、デクリメント、加算、減算、ビット単位ロジック、シフトなどの簡単な機能を実行できます。データパスは PLD と併用することで、より複雑な機能に使用することができます。この組み合わせは、カウンターや PWM、シフター、UART、I^2C インターフェースなど頻繁に使われる機能を容易に実装できます。

この組み合わせはまた、PSoC の PLD に適合していない CPLD または FPGA 設計の機能を実装するためにも使用できます。Verilog で書かれた複雑な機能は PSoC デジタル リソースをもっと効果的に利用するためにデータパス用に最適化することができます。これらの機能には加算器、減算器、シフターなどがあります。データパスは PLD に比べてこれらをより効果的に実行できます。

3 データパス設計と PLD ベース設計の比較

UDB で実装された機能は通常、PLD の代わりにデータパスを使用して算術演算を実行する場合、必要なリソースが少なくなります。例えば、表 1 に示す PLD とデータパスで実装する 8 ビット算術演算と論理演算を考えてみましょう。

表 1. PLD 対データパスのリソース使用

機能	PLD でのみ消費するリソース		データパスでのみ消費するリソース	
	PLD	使用量(%)[1]	データパス	使用量(%)[1]
ADD8	5	10.4%	1	4.2%
SUB8	5	10.4%	1	4.2%
CMP8	3	6.3%	1	4.2%
SHIFT8	3	6.3%	1	4.2%

[1] これは PSoC 5 LP デバイスの使用に基づく

デジタル機能を最も効率的に実装するには、データパスを PLD と併用する必要があります。

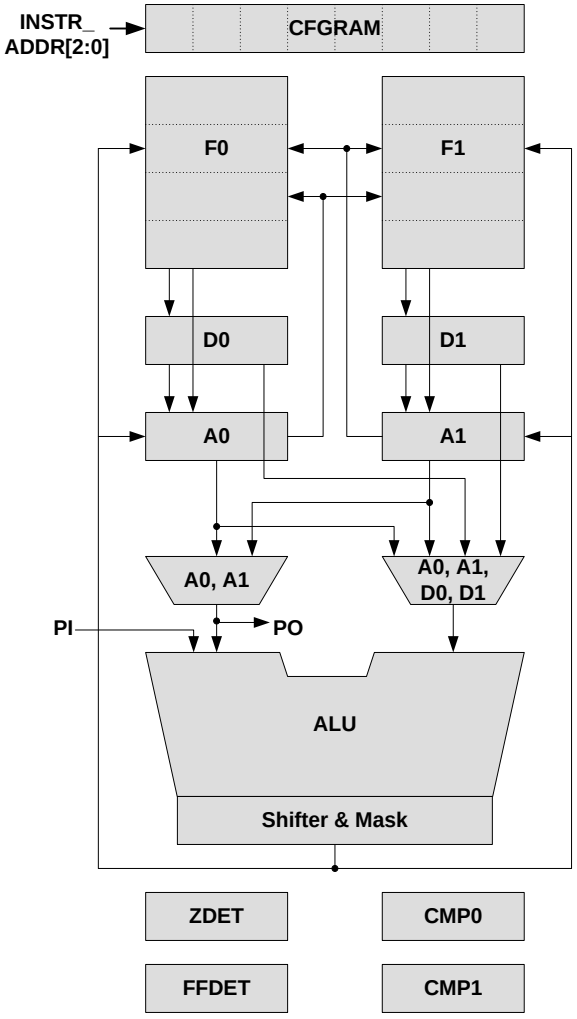
経験則として、UDB リソースを利用する最適な方法は以下の通りです。

- PLD: 組み合わせロジック、グルー ロジック、ステートマシン(PLD の実装の詳細については、AN82250 PSoC 3, PSoC 4 and PSoC 5 Implementing Programmable Logic Designs および AN81623 PSoC 3, PSoC 4 and PSoC 5 Digital Design Best Practices を参照してください)。
- データパス: CPU との FIFO インターフェース、計算、タイミング、通信、バイト幅またはワード幅の比較。

4 データパスのアーキテクチャおよび特長

データパスは図 2 に示すように、比較回路と条件生成回路を備えたコンフィギュレーション可能な 8 ビット ALU、および ALU の処理と CPU との通信用のレジスタを含んでいます。また、シフトおよびマスク用の独立ブロックもあります。

図 2. 簡略化されたデータパスブロック図



詳しいブロック図は、付録 B – データパスコンフィギュレーションツールのチートシートを参照してください。

4.1 ダイナミックコンフィギュレーション RAM (CFGRAM)

8 つの一意のデータパス命令 (またはコンフィギュレーション) をダイナミックコンフィギュレーション RAM (CFGRAM) に格納できます。命令はデータパスの機能と接続を定義します。これらには ALU 機能、ALU 入力、レジスタ書き込み、シフト動作などがあります。各命令は 1 クロック サイクル以内に実行します。

8 つの命令があるため、3 本の命令アドレス ライン(INSTR_ADDR[2:0]) があります。アドレス信号は、どの命令をデータパスで使用するかをクロックの各立ち上がりエッジで決定します。

3 つのアドレス ラインは PLD ロジックまたは外部信号によって駆動できます。一般的な使用例としては、PLD を使ってステートマシンを作成することです。ステートマシンからのロジックは、データパス命令を制御するためにアドレス入力に接続されます。

3 本のアドレス ラインは複数の名称を持っていて、混乱を招くことがあります。cs_addr[2:0]、RAD[2:0]および INSTR_ADDR[2:0]は同じものを示すことに注意してください。本アプリケーション ノートでは、INSTR_ADDR を使用します。

4.2 ALU

ALU は 8 つの汎用機能を実行できます。

- インクリメント
- デクリメント
- 加算
- 減算
- ビット単位 AND
- ビット単位 OR
- ビット単位 XOR
- パススルー

機能の選択は、CFGRAM に格納された命令によってサイクルごとに制御されます。独立したシフト (左と右) およびマスク演算は ALU の出力で可能です。

4.3 レジスタ

各データパス モジュールは 4 個の 8 ビット ワーキング レジスタと 2 個の FIFO を含んでいます。

- A0 と A1 – 通常 ALU 演算で使用されるデータを格納するアキュムレータ レジスタです。これらのレジスタを使用して Dx レジスタまたは FIFO からのデータを格納することもできます。
- D0 と D1 – 通常カウンターの開始値やリロード値などのスタティック データを格納するデータ レジスタです。
- F0 と F1 – 転送元バッファと転送先バッファとして使用できる 4 バイト FIFO であるレジスタです。これらは主に CPU または DMA をデータパスとインターフェースさせるために使用されます。

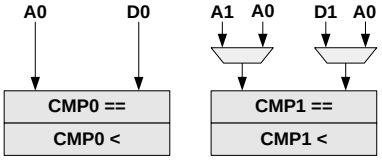
CFGRAM に格納された命令は、各命令でこれらのレジスタがどのように使用されるかを判定します。

CPU (PSoC 3 と PSoC 5LP においては DMA) を使用してデータパス レジスタへの読み書きを実行できますが、可能な限り FIFO を使用してください。アキュムレータ レジスタは互いに非同期して動作し、CPU または DMA アクセスの時などいつでも変更される可能性があります。FIFO は CPU／DMA アクセス用に同期化されます。

4.4 条件付き演算子

図 3 に示したように、各データパスは「より小さい」と「等しい」という 2 種類の比較機能があります。比較ブロックは、マスクを使用してビット単位の演算を実行することができます。

図 3. データパスの比較ブロック



A0 と A1 レジスタは 0 (ZDET) および全 1 (FFDET) 検出機能を持っています。FIFO は満杯および空状態の信号があります。

4.5 入力と出力

UDB はデジタル信号インターコネクト (DSI) というプログラム可能なデジタル配線の広範なファブリックによって囲まれています。DSI は PSoC において UDB 内、また UDB アレイと他のブロック間で信号を接続します。

データパス入力は命令、制御、データの 3 種類があります。命令入力 (INSTR_ADDR[2:0]) は CFGRAM からの現時点のデータパス命令を選択します。制御入力は FIFO からデータレジスタをロードし、アキュムレータ出力を FIFO に取り込みます。データ入力にはシフトイン (SI) とキャリーイン (CI) 信号があります。データパスは最大 6 本の入力を持っています。これらの 6 本の入力は、チップにおいて DSI に接続しているどのブロックからも来られます。

データパスは最大 6 本の出力を持っています。これらの出力は、FIFO 状態、比較状態、オーバーフロー検出、キャリーアウト、シフトアウトなど様々なデータパス状態信号に接続できます。そして、出力は DSI に接続してからその他の内蔵リソースに送信されます。

4.6 チェーン接続

各データパスは 8 ビット演算を実行することができます。複数のデータパスをチェーン接続して 1～32 ビット幅の機能を作成することができます。

シフト、キャリー、キャプチャおよびその他の条件付き信号をチェーン接続してより高精度の算術とシフト機能を形成することができます。チェーン接続された信号はデータパス入力と出力を消費しません。

UDB およびデータパスの完全な仕様については、PSoC 3 Architecture TRM を参照してください。

5 データパスベースのコンポーネント

カスタム PSoC Creator コンポーネントは、データパスを効果的に使用する最適な手段です。サイプレスは、コンポーネント作成手順の全体を説明する *Component Author Guide (CAG)* を提供しています。PSoC Creator 内でこのガイドを開くには、**Help > Documentation > Component Author Guide** を選択します。

PSoC Creator を用いてデータパス ベースのコンポーネントを実装するために 2 つの方法があります。Verilog ファイルを書き、データパス コンフィギュレーション ツールを使用してデータパスを設定することができます。Verilog で書くことまたはデータパス コンフィギュレーション ツールを使用することが難しい場合は、UDB エディターを使用できます。各方法には利点もあり欠点もあります。一般的には、Verilog の方法は UDB エディターより複雑な設定が可能ですが、UDB エディターは Verilog より使いやすいです。

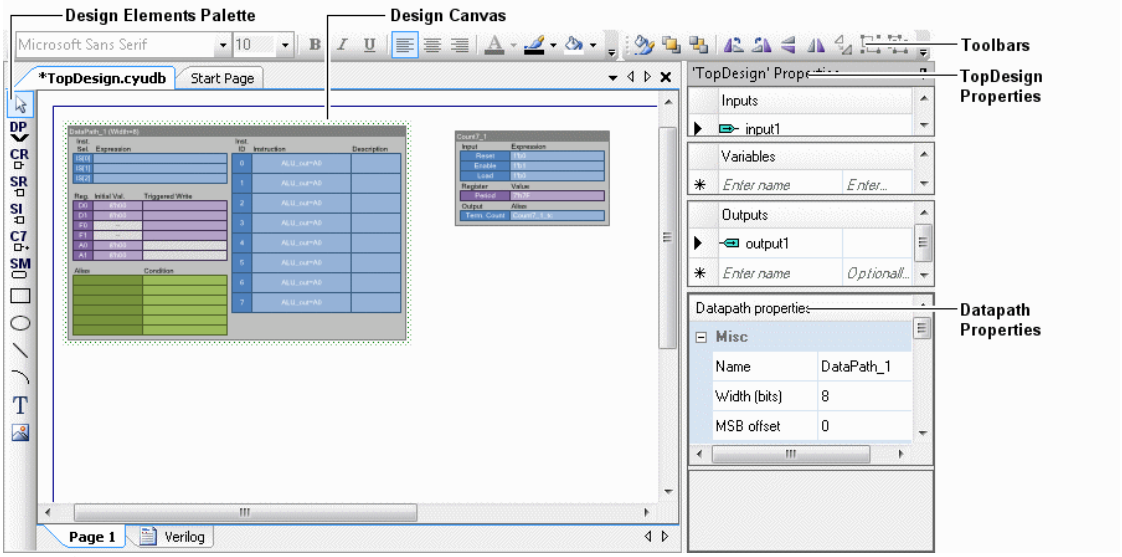
5.1 UDB エディター

UDB エディターは図 4 に示すように UDB ベース デザインを設計するために使用されるグラフィカル ツールです。Verilog コードを書くことや上級のデータパス コンフィギュレーション ツールを使用せずにコンポーネントを設計するために使用できます。UDB エディターはすべてグラフィカルで、データパスや制御レジスタ、ステータス レジスタ、ステータス割り込みレジスタ、count7 カウンター、PLD など様々な UDB 内の要素へのアクセスを可能にします。

UDB エディターにより、デジタル ロジックや Verilog についての知識がほとんどなくても UDB ベースのハードウェアを設計することができます。コードを書く必要なくドラッグ & ドロップしてハードウェアをコンフィギュレーションできるよう設計されています。これはまた、リアルタイムにデザインを Verilog に変換して、UDB ブロックがどのように Verilog に変換されるかを示します。

UDB エディターはグラフィカル ツールであるため、Verilog や UDB の込み入った細部についての知識はほとんど必要ありません。しかし、抽象概念の簡素化の結果として、柔軟性およびハードウェアに対する細かい制御が低下してしまいます。また、いくつかの高度な UDB 機能を統合せず、複雑な制限となることがあります。

図 4. UDB エディター



以下では UDB エディターの構造を説明します。

ページ – UDB エディターのドキュメントを開くと、回路図ページと同様な編集可能なページが表示されます。これは UDB 要素を配置およびコンフィギュレーションするためのキャンバスです。**Page 1** タブから UDB エディターのページをいくつでも追加できます。ページは 1 つのデザインで一緒に変換されます。

Verilog – ページのタブの隣に **Verilog** タブがあります。これは、デザインで変換されたハードウェアの読み取り専用のビューです。これは動的に更新されます。すなわち、デザインが変更されると、Verilog コードも更新されます。このコードは編集および削除が不可能であるため、望ましい結果を見るにはデザインに対して適切な変更を行う必要があります。Verilog によりデザインを編集するためにこのコードを Verilog ファイルにコピー & ペーストすることもできます。

デザイン プロパティ – デザインプロパティはデザイン キャンバスの右側にあります。これにより、設計で使用する入力、出力、変数などをコンフィギュレーションできます。設計が完了した時、入力と出力はコンポーネント端末を形成します。デザインにデータパスが含まれている場合、デザイン プロパティ ウィンドウはグローバルなデータパス コンフィギュレーションを設定するために使用されます。

デザイン要素パレット – デザイン要素パレットはデザイン キャンバスの左側にあります。これはデザインに含める UDB 要素を選択するためのメニューです。それらを下記になります。

- データパス (DP)
- 制御レジスタ (CR)
- ステータスレジスタ (SR)
- ステータス割り込みレジスタ (SI)
- count7 カウンター (C7)
- ステートマシンの状態 (SM)

UDB エディターの詳細については、Universial Digital Block (UDB) Editor Guide を参照してください。

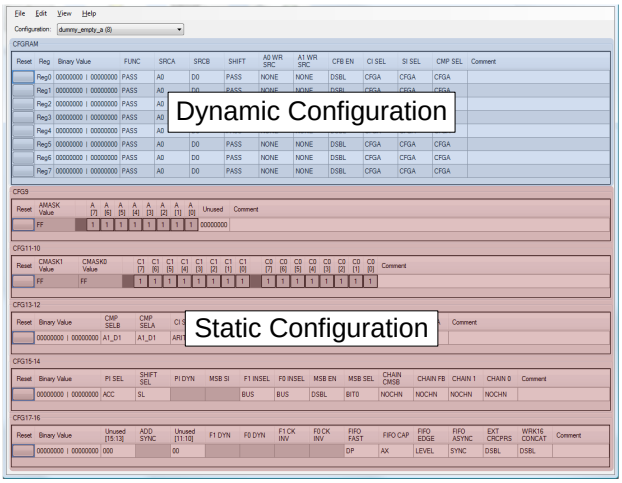
5.2 データパスコンフィギュレーションツール

データパス コンフィギュレーション ツールは、Verilog ファイル内でデータパス インスタンスの作成、表示、変更および削除を可能にするアプリケーションです。ツールは Verilog ファイルを解析し、それぞれのデータパスをエンティティとして表示します。このエンティティはツールで「コンフィギュレーション」と呼ばれます。1 つのコンフィギュレーションは 1 個の物理データパスを表します。

Verilog ファイルにデータパスがなければ、新しいデータパスを作成して追加できます。複数のデータパスを 1 つのファイルに追加し、それらをリンク (連鎖) してマルチバイト機能を作成することができます。

図 5 に示すように、各データパスはグラフィカル ユーザーインターフェース (GUI) を介して表示されます。GUI 画面は単にデータパスの CFGRAM およびスタティックコンフィギュレーションレジスタを表示します。

図 5. データパスコンフィギュレーションツールのインターフェース



付録 B – データパスコンフィギュレーションツールのチートシートで GUI の各フィールドを説明します。GUI のフィールドは PSoC デバイス内のレジスタに対応します。これらのレジスタは PSoC 3 Architecture TRM、PSoC 5LP Architecture TRM、および PSoC 4 Architecture TRM で詳しく説明されています。

データパスコンフィギュレーションツールを PSoC Creator 内からでも起動できます (**Tools** > **Datapath Config Tool…**)。

本アプリケーションノートの本文はデータパスコンフィギュレーション ツールではなく、UDB エディターに焦点を合わせます。付録 A は本アプリケーションノートで言及するすべての例をデータパスコンフィギュレーションツールを使用して実施する方法を説明します。また、付録 A – データパスコンフィギュレーションツールの使用例 にパラレル入力とパラレル出力を使用する例も示します。

5.3 適切なツールの選択

ほとんどのデザインは UDB エディターで行えます。UDB エディターはデータパスの最も一般的な機能を実装します。ただし、実装されない機能はいくつかあります。これらの機能を実装するためには、データパスコンフィギュレーションツールを使用しなければなりません。UDB エディターが対応していない機能は以下の通りです。

- ダイナミック FIFO 制御
- FIFO クロック反転
- パラレル入力とパラレル出力。付録 A – データパスコンフィギュレーションツールの使用例で、データパスコンフィギュレーションツールを使用してこれを実装する一例を説明します。
- 巡回冗長チェック (CRC)
- 疑似乱数シーケンス (PRS)
- 選択可能キャリーイン
- ダイナミックキャリーイン

これらの機能はほとんどの設計で必要ありません。そのため、まずはるかに使いやすい UDB エディターで開始してください。開発中に、望ましい機能が UDB エディターで実装されていないことに気付いた時、いつでも UDB エディターによって生成された Verilog コードをコピー & ペーストし、データパスコンフィギュレーションツールを使用して Verilog ファイルを変更できます。

注: UDB エディターは、データパス コンフィギュレーション ツールによって生成された Verilog ファイルを読むことができません。

これらの高度な機能のいずれかを必要とする場合、自分の Verilog コードを書いてデータパスコンフィギュレーションツールを使用するオプションがあります。付録 A – データパスコンフィギュレーションツールの使用例は UDB エディターの代わりにデータパスコンフィギュレーションツールを使用するすべての例を示します。

UDB エディターがふさわしいツールである場合、本アプリケーションノートは UDB エディターを用いた簡単なデータパスベースのコンポーネントを作成する順を追った説明を提供しています。

注：このアプリケーションノートに添付されているのは、いくつかの完成例です。 完成例はすべてのデバイスに存在するわけではありません。 以下の手順は、PSoC デバイス上にコンポーネントを作成するのに十分なものです。

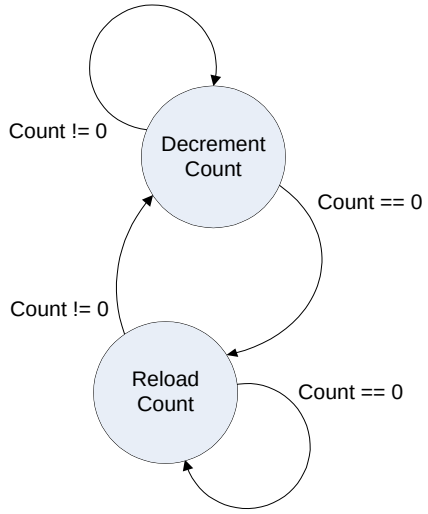
6 プロジェクト#1 – 8 ビットダウンカウンター

このプロジェクトの目的は、簡単なデータパスベースのコンポーネントを作成するために必要な手順を紹介することです。ここでは、簡単な 8 ビットダウンカウンターの作成によって示します。

6.1 8 ビットカウンターコンポーネントの詳細

図 6 に示すように、簡単なダウンカウンターは 2 つの状態を持つステートマシンで示すことができます。

図 6. 簡単なカウンターの状態図



カウンターは初期値で開始し、その値をデクリメントします。カウントが 0 に達すると、イベントがトリガーされ、周期はリロードされます。この種のカウンターは容易にデータパスに実装されます。

カウンターを実装するためには 2 つのデータパス命令のみ必要です。最初の命令は、ワーキングレジスタの 1 つに格納されたカウント値をデクリメントします。2 番目の命令は、初期値でカウントをリロードします。

この例で使用される 2 つのデータパス レジスタは以下の通りです。

- A0 – カウント値を格納し、ALU によってデクリメントされます。
- D0 – カウントが 0 に達した時に A0 にリロードされる値を格納します。

このカウンターが機能するためには、データパスがどの命令を実行しているかを判断する方法 (ロードまたはデクリメント) が必要です。図 6 は、遷移がカウントの値 (0 または 0 以外) によって制御されることを示します。

データパスは、A0 と A1 のデータの値を監視するゼロ検出ブロック (ZDET) を備えています。このブロックは z0 (A0 == 0)と z1 (A1 == 0) という 2 つの出力があり、それぞれが A0 と A1 の状態を示します。各出力は、値が 0 であれば HIGH になり、値が 0 ではなければ LOW になります。

この例では、z0 出力を使用してどの命令を実行するかを制御します。

データパスには 8 つの異なる命令を含めることができます。データパスで使用される命令は、3 ビットのアドレスラインによって選択されます。 前述のように、このカウンターに必要な命令は 2 つだけなので、ビット 0 を z0 に接続し、他のビットを LOW に保持することができます。表 2 は遷移表です。

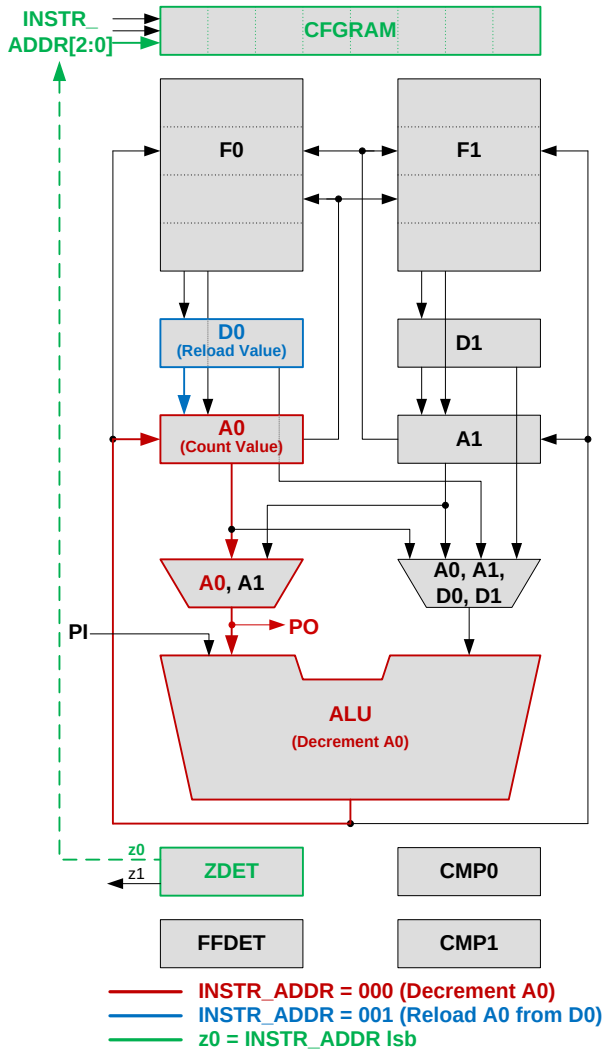
表 2. データパスの命令

CFGRAM Instruction	Instruction Address Bits (INSTR_ADDR)			Operation
	2	1	0	
0	0	0	z0 = 0	Decrement Count
1	0	0	z0 = 1	Reload Count
2-7	X	X	X	Not Used

A0 でのカウントは 0 に達すると、z0 は 1 になります。これにより、次のデータパス命令は「リロード カウント」になります。カウントが D0 から A0 にリロードされる時、z0 は 0 になり、次の命令は「デクリメント カウント」になります。

動作方法を視覚化するには、図 7 で強調表示されるデータパス ブロック図をご覧ください。

図 7. 強調表示される簡単なカウンターブロック図

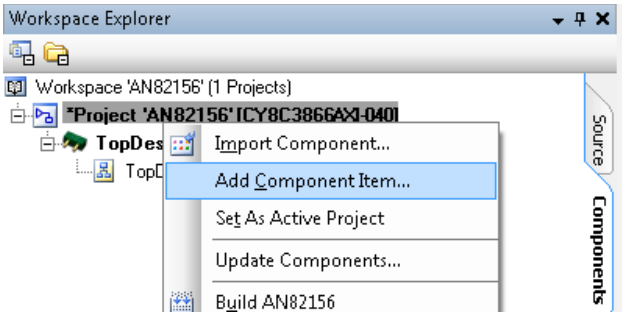


6.2 8 ビットカウンターコンポーネントの作成手順

この例では、空のプロジェクトで開始します。

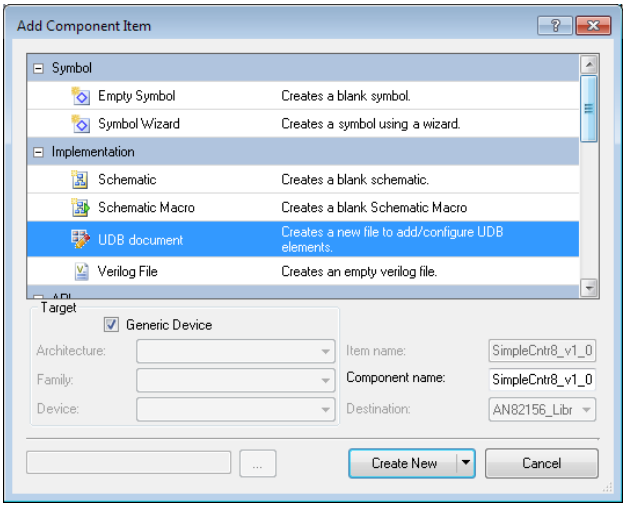
1. PSoC Creator を起動して、「AN82156」というプロジェクトを作成します。「AN82156」ワークスペースもデフォルトで作成されます。
2. **Workspace Explorer** の **Components** タブに切り替え、プロジェクト「**AN82156**」を右クリックします。図 8 に示すように、ドロップダウン メニューから **Add ComponentItem...** を選択します。

図 8. コンポーネントアイテムの追加



3. 図 9 に示すように、**UDB Document** を選択し、コンポーネントを *SimpleCntr8_v1_0* と名付けます。

図 9. UDB ドキュメントの追加

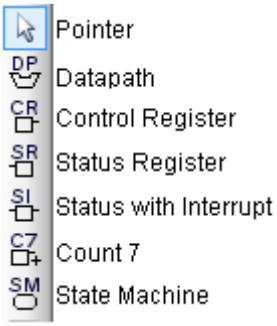


コンポーネントの名前に版数を含んだ方が良いです。コンポーネントの名前に「_vX_Y」タグ (X がメジャーバージョン、Y がマイナーバージョン) を追加します。PSoC Creator はバージョニング機能を持ち、コンポーネントの複数のバージョンを追跡して使用できます。

4. UDB ドキュメント回路図を作成するために **Create New** を作成します。

UDB ドキュメント回路図は、UDB ベースのコンポーネントを作成するためのキャンバスです。この回路図に UDB 要素をドラッグ & ドロップする方法は、普通の PSoC Creator 回路図と全く同じです。回路図にドラッグできるコンポーネントは回路図の左側のサイドバーに表示されます (図 10)。

図 10. UDB 要素

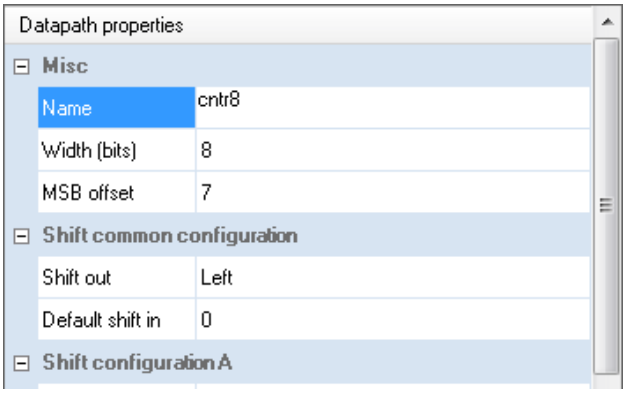


5. データパス要素を UDB ドキュメント回路図にドラッグします (図 11)。

図 11. データパス要素

6. データパスインスタンスをシングルクリックします。UDB エディターの右側にあるプロパティウィンドウの最下部には **Datapath properties** があります。図 12 に示すようにデータパス名を「*Datapath_1*」から「*cntr8*」に変更します。

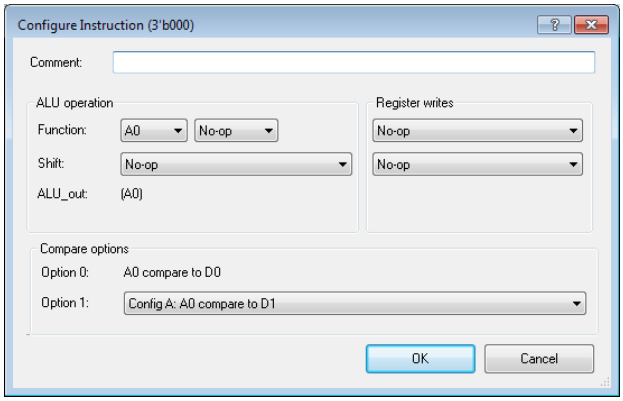
図 12. データパスプロパティ



7. データパス *cntr8* では、Inst. Addr が 3'b000 の緑色のボックス (図 13) をダブルクリックして図 14 に示すコンフィギュレーション ダイアログを開きます。

図 13. 命令 0

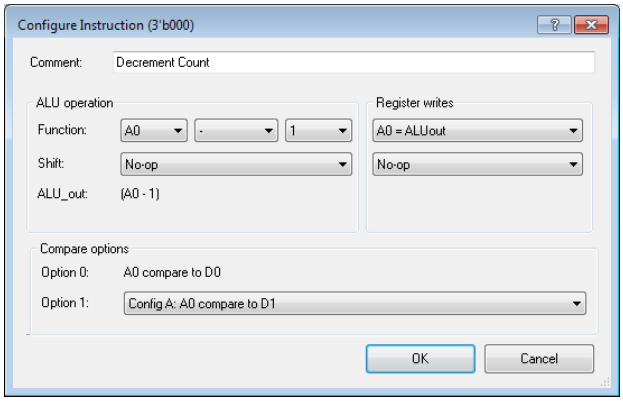
図 14. 空白命令コンフィギュレーションダイアログ



このダイアログ ボックスでは、最初のデータパス命令をコンフィギュレーションします。表 2 に示すように、命令 0 コンフィギュレーションではデータパスはカウント (A0 に格納された) をデクリメントします。

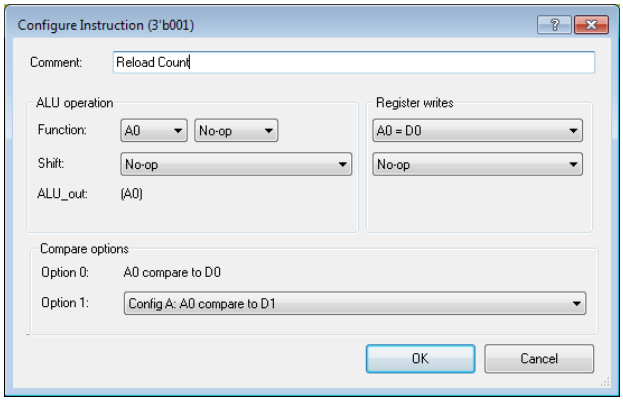
8. **ALU operation Function** を A0 – 1 (デクリメント) に設定します。**Register Writes** セクションでは *A0 = ALUout* に設定します。これにより、最初の命令が A0 レジスタをデクリメントしてその値を A0 に戻って書き込むよう設定されます。この命令をコメントとして記述することもできます。この場合は「Decrement Count」 (デクリメント カウント) です。図 15 を参照してください。

図 15. 命令 0 コンフィギュレーション



9. 次に、命令アドレス 3'b001 を設定します。そのボックスをダブルクリックします。表 2 に示すように、命令 1 は D0 に格納された値を A0 にリロードします。**Register Writes** セクションでは、*A0 = D0* に設定します。図 16 を参照してください。

図 16. 命令 1 コンフィギュレーション



次に、各クロック サイクル中にデータパスが実行する命令を選択する必要があります。この例では、データパス内蔵のゼロ検出器 (ZDET) を使用します。まず、ゼロ検出器の信号にラベルを付けます。

10. 図 17 に示す青色の出力ボックスをダブルクリックします。

図 17. データパス出力

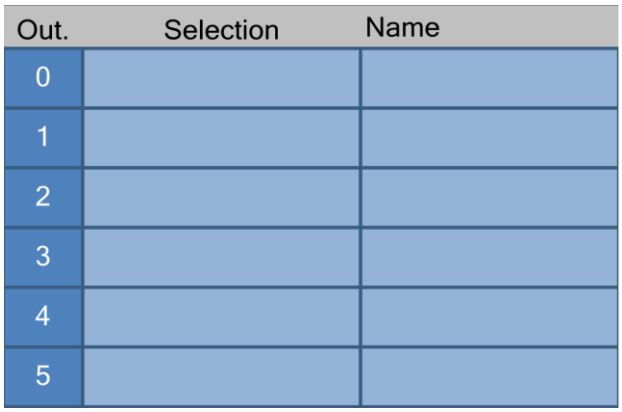
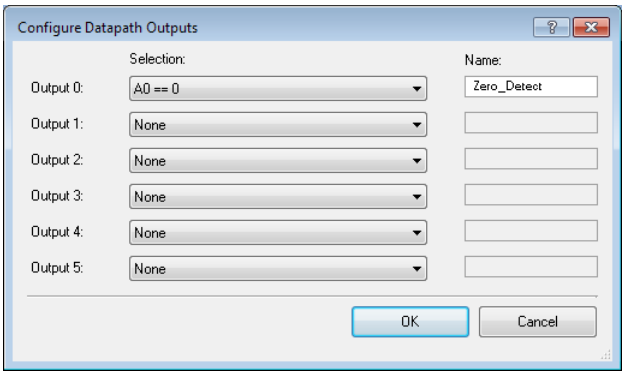


図 18 に示すように「Configure Datapath Outputs」ダイアログが表示されます。

11. Output 0 を *A0 == 0* に、Name を「*Zero_Detect*」に設定します。

図 18. データパス出力コンフィギュレーション



Zero_Detect 信号はデータパスの ZDET 出力を表します。Zero_Detect は、A0 が 0 になった時に HIGH になり、A0 が 0 以外になった時に LOW になります。

この信号をデータパスの命令アドレスに接続します。これはデータパスの入力セクションで設定されます。

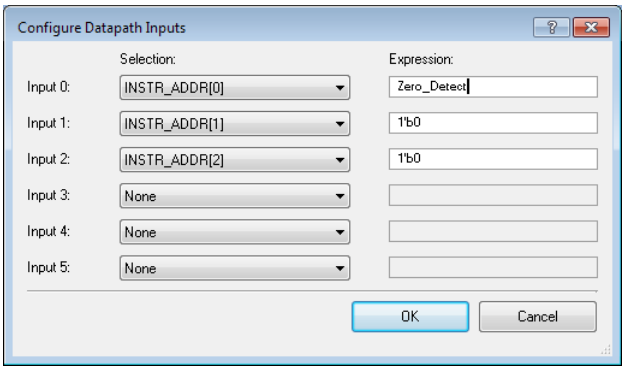
12. 図 19 に示すように青色の入力ボックスをダブルクリックします。

図 19. データパス入力

Inp.	Selection	Expression
0	INSTR_ADDR[0]	1'b0
1	INSTR_ADDR[1]	1'b0
2	INSTR_ADDR[2]	1'b0
3		
4		
5		

13. **Input 0** は図 20 に示すように、**Selection** が *INST_ADDR[0]*に、**Expression** が *Zero_Detect* に設定されたことを確認します。

図 20. データパス入力コンフィギュレーション



Zero_Detect 信号はデータパスが実行する命令を選択します。データパスは、Zero_Detect が HIGH の場合命令 1 (A0 と D0 をリロード) を実行し、LOW の場合命令 0 (A0 をデクリメント) を実行します。

次に、カウンターおよび周期レジスタの初期値を定義します。

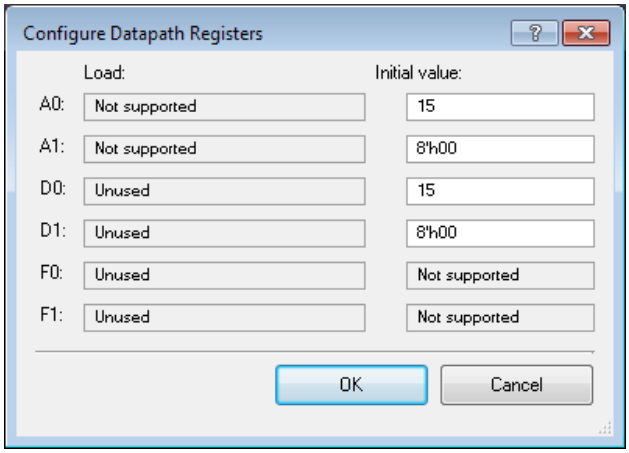
14. 図 21 に示す灰色と紫色のコンフィギュレーション ボックスをダブルクリックします。

図 21.データパスレジスタ

Reg.	Load	Initial Value
A0	Not supported	8'h00
A1	Not supported	8'h00
D0	Unused	8'h00
D1	Unused	8'h00
F0	Unused	Not supported
F1	Unused	Not supported

15. 図 22 に示すように A0 と D0 の初期値を 15 に設定します。

図 22. データパスレジスタコンフィギュレーション

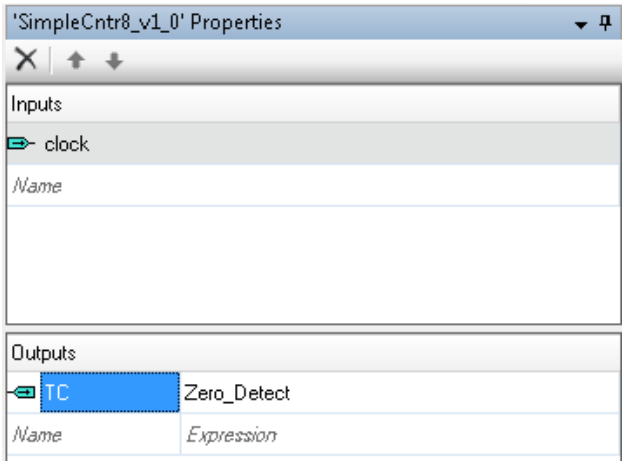


これは A0 と D0 が始まる値を設定します。 これは、コンポーネントの起動時に A0 の値が 15 になるため、0 になるとカウントダウンされ、ゼロになると D0 の値がロードされます。これを 15 に設定します。

これで、データパスのコンフィギュレーションが完了しました。次に、コンポーネント出力を定義します。この出力はターミナルカウント (TC) です。これは、カウンターが 0 に達した時に HIGH になり、その他の時に LOW になります。この出力を使用して、コンポーネントの機能をテストします。

16. 図 23 に示すように‘SimpleCntr8_v1_0’ Properties ウィンドウでは、Outputs の下で「*TC*」と名付けられた新しい出力を定義し、Expression を「*Zero_Detect*」に設定します。

図 23. コンポーネント出力



このコンポーネントのシンボルを生成します。これはトップ デザイン 回路図に表示されるシンボルです。

17. 図 24 に示すように、UDB ドキュメント (*.cyudb*) の空白領域を右クリックして **Generate Symbol** を選択します。

図 24. シンボルの生成

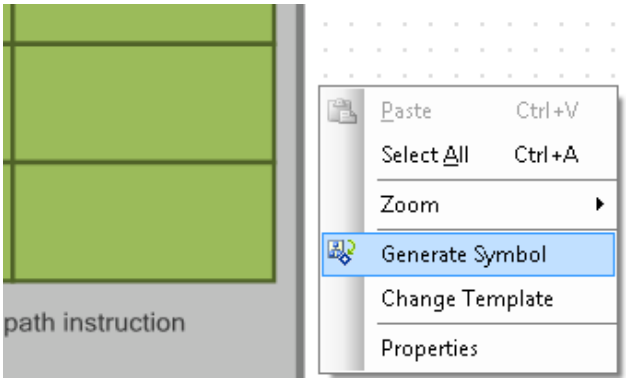
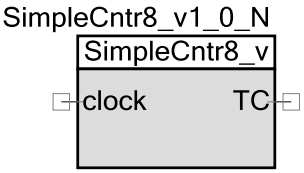


図 25 に示すように回路図シンボルが表示されます。

図 25: 回路図シンボル

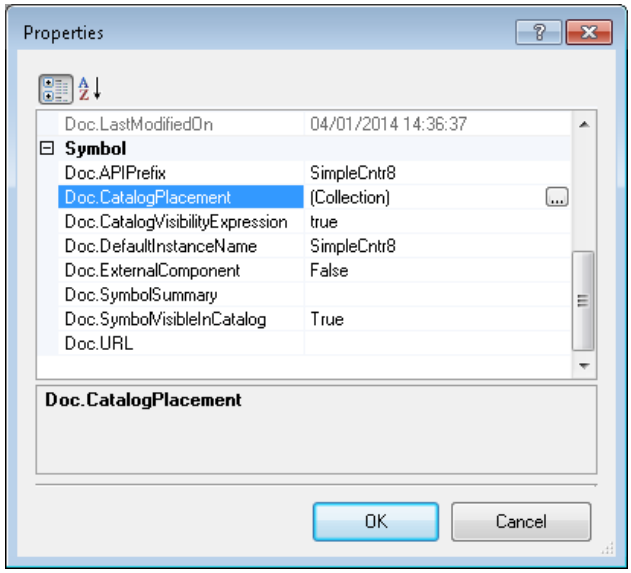


18. (シンボル自身ではなく) シンボルエディターの空白領域を右クリックして、ドロップダウンメニューから **Properties** を選択します。

19. 図 26 に示すようにプロパティフィールドの **Symbol** セクションに値を入力します。

- *Doc.APIPrefix* = SimpleCntr8
  この値は、コンポーネント用に生成されたいかなる API ファイル名の先頭にも付加されます。この例では、API を生成しませんが、コンポーネントを作成するたびにここに値を入力してください。
- *Doc.CatalogPlacement = AN82156/Digital/Cntr8*
  「…」ボタンをクリックして「Catalog Placement」ダイアログを開き、この値を入力します。PSoC Creator はこの値を使って、Component Catalog の階層を定義します。最初のアイテムは、カタログにおいてコンポーネントを表示するタブです。その後の各「/」はサブレベルを表します。階層は少なくとも 1 つのサブレベルを含む必要があります。上記の値は、コンポーネントが AN82156 タブの「Digital」サブレベル内で「Cntr8」として表示されることを示します。
- *Doc.DefaultInstanceName = SimpleCntr8*
  これは、コンポーネントが回路図に配置された時に表示されるデフォルト名です。

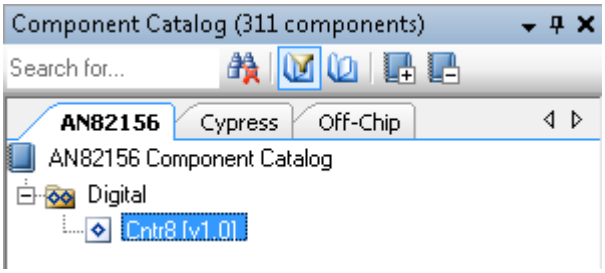
図 26. シンボルプロパティの追加



20. **File > Save All** を選択して、すべての変更がプロジェクトに適用されたことを確実にします。

コンポーネントは使用可能になりました。図 27 に示すように、新しいコンポーネントは Component Catalog の *AN82156* タブに表示されます。

図 27. Component Catalog での新しいコンポーネント



コンポーネントが Component Catalog に表示された後、それを回路図に配置して他のコンポーネントと同じように使用することができます。それをテストするには、クロックソースおよびカウンターの「TC」出力を表示する方法を追加する必要があります。

21. Cntr8 コンポーネントをプロジェクト回路図に配置します。
22. クロック コンポーネントを「clock」端末に接続します。クロックを 10kHz に設定します。どの値でも構いませんが、10kHz ならオシロスコープでの表示が容易になります。
23. オシロスコープで観察できるように、デジタル出力ピンを「clock」端末に接続します。それを P0_0_clk と名付け、他のすべての設定をデフォルト値のままにします。

    注: PSoC 4 では、クロックをピンに直接接続することはできません。クロックをピンへルーティングするために、以下の手順を行います。

    a. デジタル出力ピン コンポーネントを回路図に配置します。
    b. ピン カスタマイザーで、**Clocking** タブを選択します。
    c. **Out Clock:** を *External* に設定します。
    d. **Pins** タブに戻って、**Output** サブタブに移動します。
    e. **Output Mode:** で、*Clock* を選択します。

f. **OK** をクリックします。

g. クロック信号をピン コンポーネントの out_clk 端末に接続します。

注: For PSoC 6 MCU では、クロックをピンに直接接続することはできません。クロックをピンへルーティングするために、以下の手順を行います。

a. デジタル出力ピン コンポーネントを回路図に配置します。

b. TFF コンポーネントを回路図に配置します。

c. クロックコンポーネントの出力を TFF コンポーネントの clk 端末に接続します。

d. Logic High「1」のコンポーネントを TFF コンポーネントの t 入力に接続します。

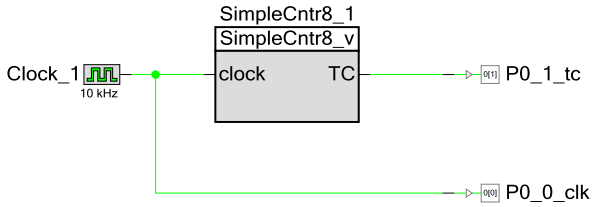
e. TFF コンポーネントの q 出力を手順 a で配置したデジタル出力ピンに接続します。

注: デバイス外部で見られるクロック周波数は、UDB コンポーネントによって使用される実際の値の半分になります。

24. デジタル出力ピン コンポーネントを「TC」端末に接続します。それを P0_1_tc と名付け、他のすべての設定をデフォルト値のままにします。カウントが 0 になると、このピンは HIGH になります。

図 28 に完全なプロジェクトの回路図を示します。

図 28. 簡単なカウンタープロジェクトの回路図



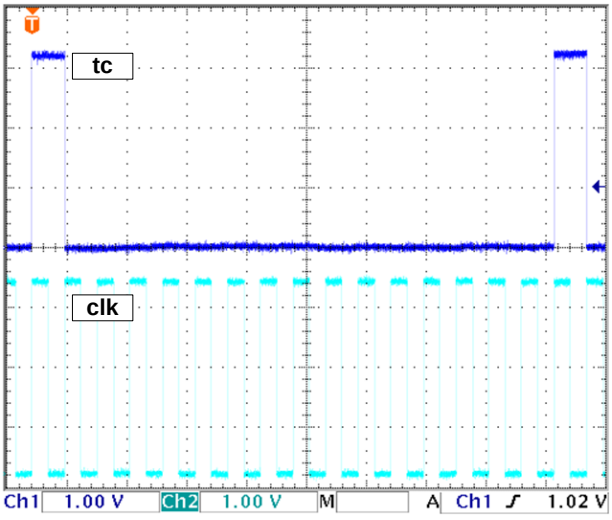
25. cydwr の *Pins* タブで、ピンの名前に応じてピンを P0[0]と P0[1]に割り当てます。

これで、プロジェクトのビルドおよび PSoC のプログラミングの準備ができました。P0[0]と P0[1]でクロックとターミナルカウントを確認することができます。

26. プロジェクトを保存してビルドし、PSoC をプログラムします。

図 29 に示すように、オシロスコープを出力ピンに接続すれば、「clock」と「tc」出力を観察できます。

図 29. 簡単なカウンターの出力



A0 に開始値 15 をロードしたため、周期が 16 (15 から 0 までカウント) クロックサイクル幅です。A0 が 0 に達する時、A0 が D0 の値でリロードされるため、「TC」ピンは 1 クロック サイクルだけの HIGH パルスを発します。A0 が 0 ではなくなる時、「TC」が LOW にセットされ、コンフィギュレーションは再び A0 のデクリメントに戻ります。

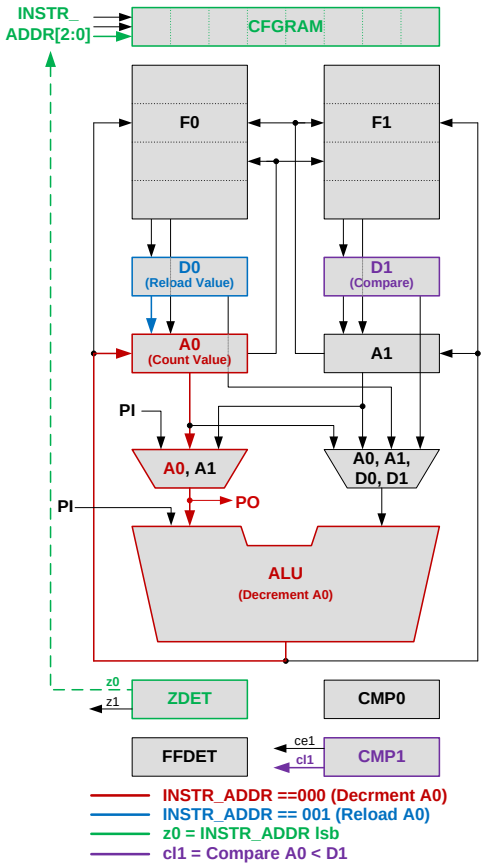
これで、ファームウェアを書くことが全く必要なく初めてのデータパスベースのコンポーネントの設計を完了しました。

6.3 カウンターから PWM への変更

PWM は単に比較機能を備えたカウンターです。PWM を作成するには、A0 の値を他の固定値と比較する方法が必要です。比較対象の固定値を D1 レジスタに格納し、A0 が D1 未満であるかどうかを確認するように比較ブロックを設定することができます。

図 30 に示すように、強調表示されたブロック図を用いてそれを視覚化することができます。

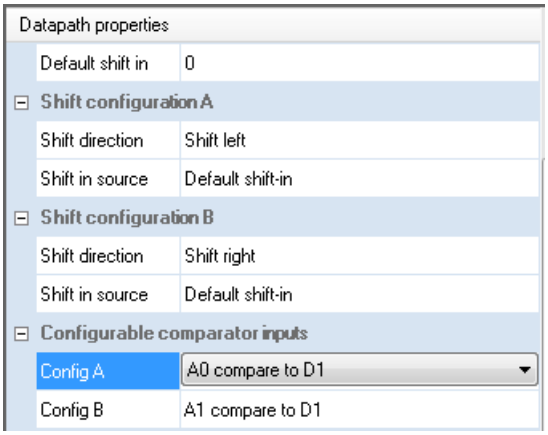
図 30. 簡単な PWM の強調表示されたブロック図



この例は、前の節で作成した 8 ビットカウンターコンポーネントを変更します。空のコンポーネントで、または以前のカウンターコンポーネントのコピーを作成することで開始することができますが、以下の手順は既存のカウンターを変更することを前提とします。

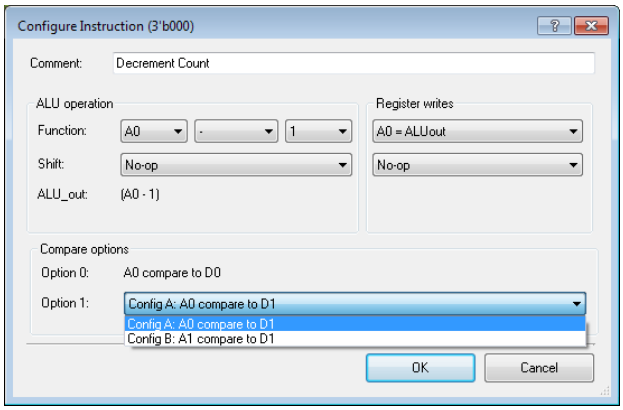
1. UDB ドキュメント (.cyudb) に戻り、データパスをクリックします。データパスプロパティウィンドウで、Configurable コンパレータ入力の Config A が D1 と比較して A0 に設定されていることを確認します。 図 31 を参照してください。

図 31. データパス比較コンフィギュレーション



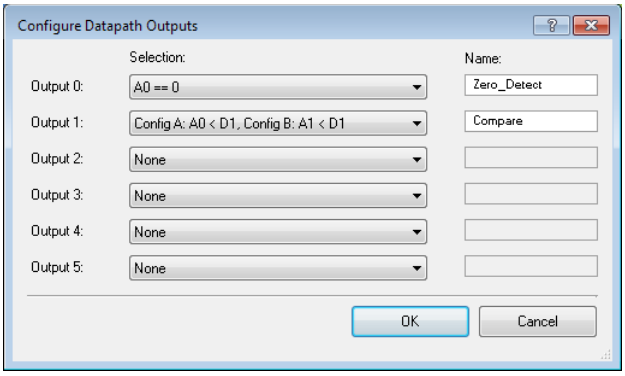
2. 図 32 に示すように命令 0 と命令 1 両方で、**Compare Option 1** が *ConfigA: A0 compare to D1* に設定されたことを確認します。このダイアログを開くために、適切な緑色の指示ボックスをダブルクリックします。

図 32. 比較オプションの設定



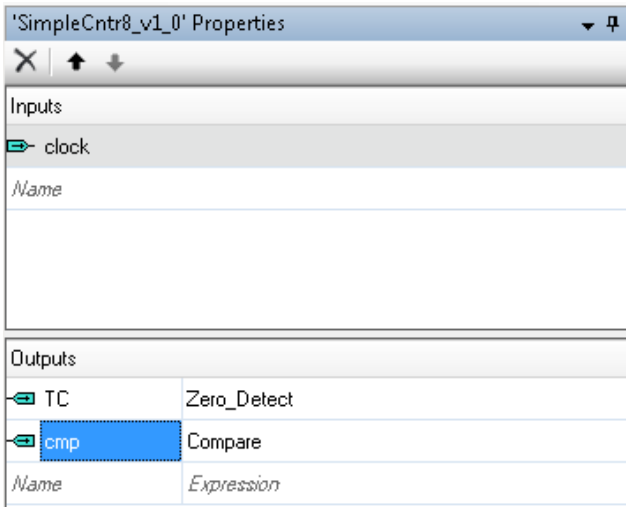
3. 図 33 に示すように、出力ボックスをダブルクリックして **Output 1:** を *Config A: A0 <D1…* に、**Name** を「*Compare*」に設定します。

図 33. 比較出力の設定



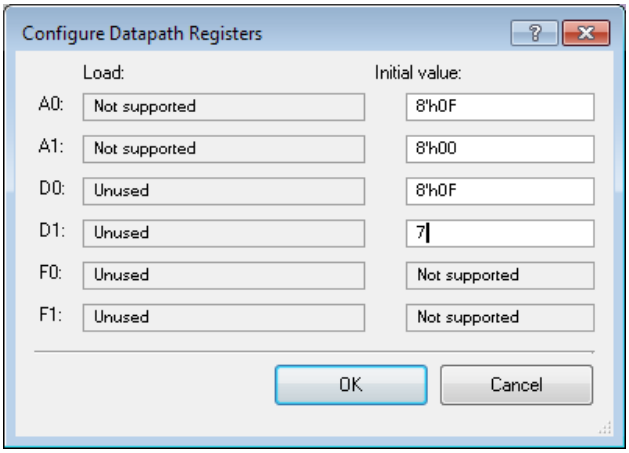
4. 図 34 に示すように、「*cmp*」と名付けられた新しいコンポーネント出力を定義し、*Expression* を Compare に設定します。

図 34. 比較出力



5. 最後に、比較レジスタ (D1) の初期値を設定します。
6. **Configure Datapath Registers** ダイアログを開きます。図 35 に示すように D1 の初期値を 7 に設定します。

図 35. D1 の初期値



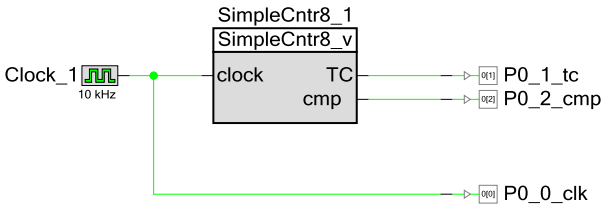
7. 回路図キャンバス上の任意の空白を右クリックして Generate Symbol を選択します。これにより、新しい出力 cmp で新しいシンボルが作成されます。
8. **File** > **Save All** を選択します。

このコンフィギュレーションでは、比較ブロックの出力は A0 が D1 より小さい時に HIGH になり、A0 が D1 より大きい時に LOW になります。

PWM コンポーネントおよびシンボルは Component Catalog の AN82156 タブにまだ表示されています。コンポーネントはプロジェクト回路図で自動的に更新されます。

9. 出力ピンを追加し、「cmp」端末に接続します。図 36 に示すように、それを *P0_2_cmp* と名付け、P0[2]ピンに割り当てます。

図 36. 簡単な PWM プロジェクトの回路図

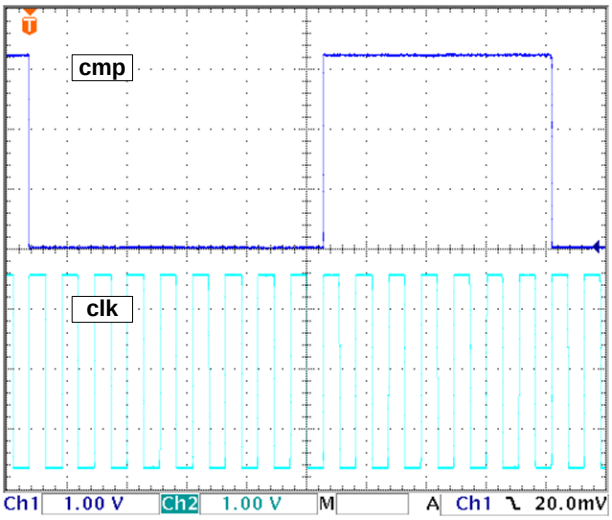


これで、プロジェクトのビルドおよび PSoC のプログラミングの準備ができました。クロックおよびターミナル カウントは P0[0]と P0[1]ピンで確認することができます。PWM 出力は P0[2]で確認できます。

10. プロジェクトを保存してビルドし、PSoC をプログラムします。

オシロスコープを出力ピンに接続すれば、「clock」、「tc」および「cmp」出力を観察することができます。図 37 は「clk」と「cmp」信号を示します。

図 37. 簡単な PWM の出力



A0 と D0 に開始値 15 をロードしたため、周期が 16 クロック サイクル幅となります。D1 を 7 に設定したため、A0 が 7 より小さくなると「cmp」ピンが HIGH になります。D1 の比較値を変更して PWM をテストすることができます。

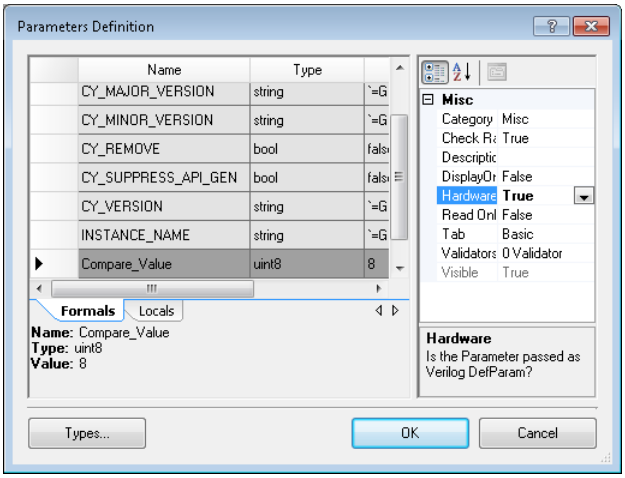
6.4 パラメーターの追加

コンポーネントのパラメーターのいずれかを変更するたびに Verilog コードを変更することは不便です。さらに、もし異なる周期および比較値を持つ 2 個の PWM を必要としたらどうなるでしょうか。ほとんどのサイプレスのコンポーネントと同様に、ユーザー設定可能なパラメーターをコンポーネントに追加することができます。

1. コンポーネントのシンボル エディター ページ (.cysym) を開き、空白領域を右クリックします。
2. ドロップダウン メニューから **Symbol Parameters** を選択します。
3. 既存のパラメーターの下の空の行に新しいパラメーターを入力します。
   - Name = Compare_Value
   - Type = uint8
   - Value = 8
4. 図 38 に示すように、ウィンドウの右側にある **Misc** 設定で「**Hardware**」フラグを「*True*」にセットします。

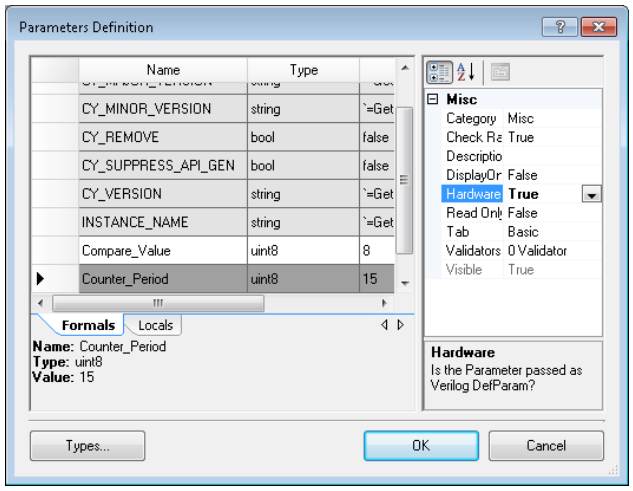
これは、UDB ハードウェアがパラメーターを使用できるようにパラメーターを Verilog コードに変換します。

図 38. コンポーネントパラメーターの追加



5. 作成した比較値の定義の下の行に他の新しいパラメーターを入力します。
   - Name = Counter_Period
   - Type = uint8
   - Value = 15
6. 図 39 に示すように、ウィンドウの右側にある **Misc** 設定で「**Hardware**」フラグを「*True*」にセットします。

図 39. 他のコンポーネントパラメーターの追加

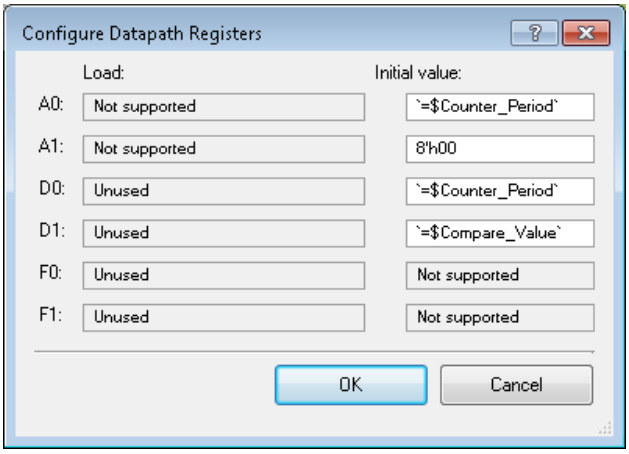


7. **OK** をクリックして **File** > **Save All** を選択することで、コンポーネントに変更を適用します。

次の手順では、パラメーターを UDB エディターにリンクする方法を説明します。

8. UDB エディター (.cyudb) に移動して **Configure Datapath Register** ダイアログを開きます。
9. **A0** と **D0** では、**Initial value:** を「`=$Counter_Period`」に設定します。単一引用符「 ‘ 」ではなく、アクセント符号「`」を使うことに注意してください。図 40 に示すように **D1** では、**Initial value:** を「`=$Compare_Value`」に設定します。

図 40. パラメーターの初期値



このコードは A0、D0 および D1 の初期値をコンポーネント パラメーターにリンクさせます。

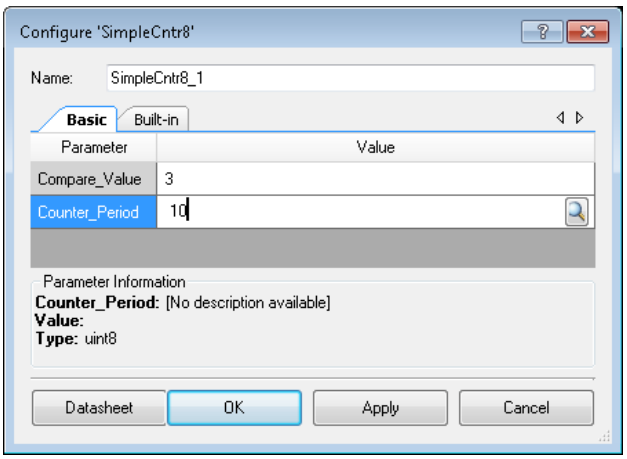
これで、周期と比較値は UDB エディターで変更せずにビルド時に設定することができます。

10. **File** > **Save All** を選択します。

プロジェクト回路図に戻って、SimpleCntr8_1 コンポーネントをダブルクリックしてプロパティダイアログを開きます。

11. 図 41 に示すように比較値を 3 に、カウンター周期を 10 に変更します。

図 41. コンポーネントパラメーターの設定

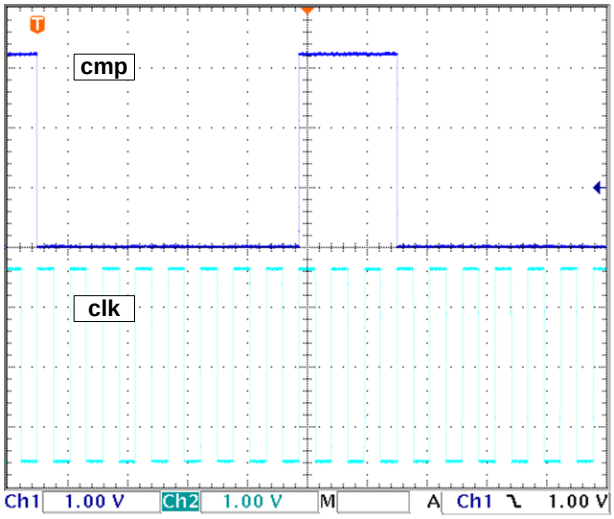


12. **OK** をクリックして変更を適用します。

13. **File** > **Save All** を選択して、プロジェクトをビルドし、PSoC をプログラムします。

図 42 に示すように、周期と比較出力が変更されます。

図 42. パラメーターの変更された PWM 出力



周期を 11 サイクルに設定し (カウンターがリロードの前に 0～10 に進むため、周期は 10+1)、比較値を 3 に設定しました。結果は、8 クロック サイクルの LOW 出力および 3 クロック サイクルの HIGH 出力となります。

パラメーターの値が uint8 である限り、ほとんど何の値にも変更することができます。また、複数のコンポーネントのインスタンスをプロジェクトに配置し、それらを異なる値に設定することもできます。

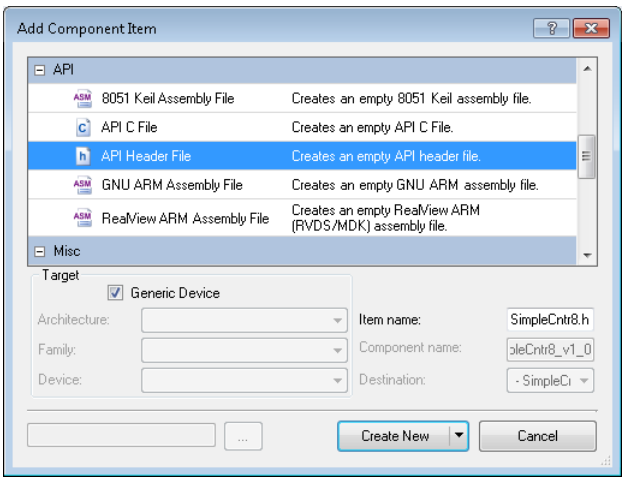
入力値制限の設定を含むコンポーネント パラメーターを追加する詳細については、*Component Author Guide* を参照してください。

6.5 ヘッダーファイルの追加

PWM の動作を設計時に変更することだけでなく、C コードにより PWM のレジスタを変更することで実行時に PWM を変更することもできます。例えば、レジスタ D0 に周期値を格納し、レジスタ D1 に比較値を格納しました。これらのレジスタに容易にアクセスできるように、使用されるレジスタを定義するヘッダーファイルを作成します。そうしてそれらを C コードで変更することができます。

1. **Components** タブで、**SimpleCntr8_v1_0** を右クリックして **Add Component Item** を選択します。
2. Add Component Item ウィンドウで、**API** セクションに移動して **API Header File** をクリックします。
3. 図 43 に示すように **Item name** を *SimpleCntr8.h* に変更します。

図 43. ヘッダーファイルの追加



4. **Create New** をクリックします。これにより、ヘッダー ファイルがコンポーネントに追加されます。

比較値 (D1) と周期値 (D0) の定義を加えます。

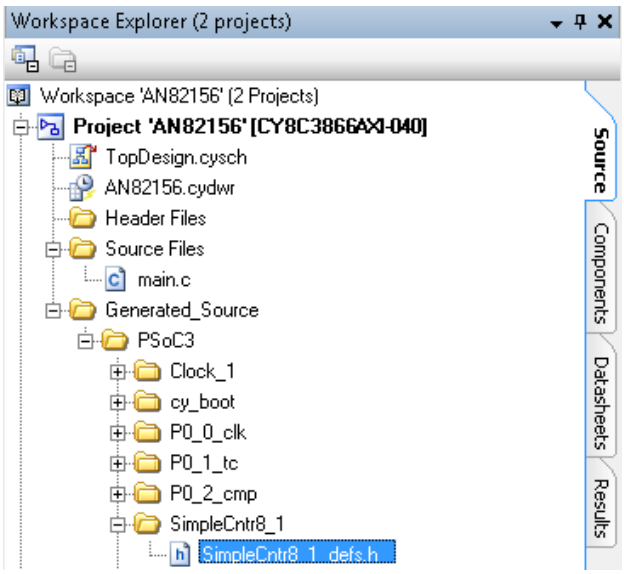
5. ヘッダー ファイルで `//[]END OF FILE` の上に以下の定義を加えます。

```
#include "`$INSTANCE_NAME`_defs.h"
#define `$INSTANCE_NAME`_Period_Reg                 `$INSTANCE_NAME`_cntr8_D0_REG
#define `$INSTANCE_NAME`_Compare_Reg                `$INSTANCE_NAME`_cntr8_D1_REG
```

これらの 2 つの定義により、ファームウェア内で D0 と D1 レジスタに直接書き込むことができます。*INSTANCE_NAME`_defs.h* ファイルはデータパス内のレジスタの定義一式を含んでいます。

INSTANCE_NAME`_defs.h は図 44 に示すように、ソース タブの生成されたソース フォルダーにあります。

図 44. *defs.h* ファイル



実行時に比較値を更新するために、作成したばかりの定義に書き込みます。コンポーネントが SimpleCntr8_1 と名付けられた場合、C コードは以下の通りです。

```
SimpleCntr8_1_Period_Reg = 0x08;

SimpleCntr8_1_Compare_Reg = 0x02;
```

これにより、周期値が 0x08 に、比較値が 0x02 に更新されます。これらの定義の使用方法については、*Component Author Guide* を参照してください。上記の C コードに示したレジスタへ直接書き込む方法は 8 ビット レジスタにのみ適用できます。16 ビット以上のレジスタに対しては、次のプロジェクトで説明する他の方法を使用しなければなりません。

6.6 16 ビットへの PWM の拡張

データパス チェーニングについて説明します。データパスは隣接のデータパスに接続される専用信号を持っています。これらの信号により、1～32 ビット幅の機能を作成することができます。この例では図 45 に示すように、最初のサンプル プロジェクトと同様な PWM を作成しますが、16 ビット幅です。

図 45. チェーン接続された UDB による 16 ビット機能

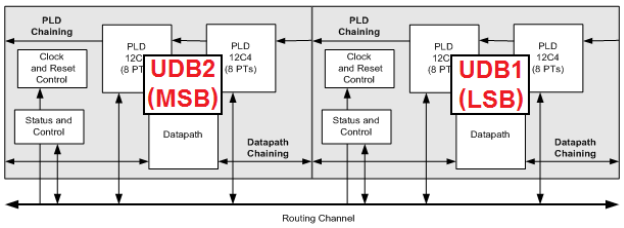
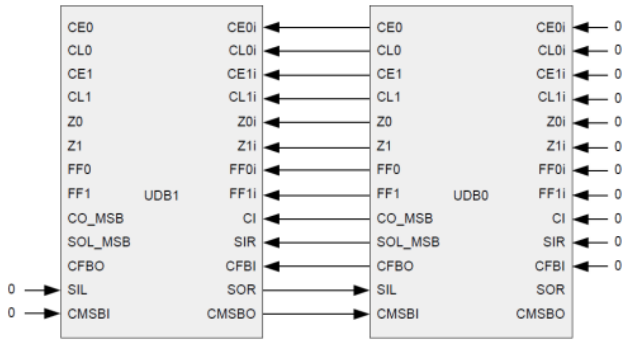


図 46 に示すように、各データパス内の ALU はキャリー、シフト データおよび条件付き信号を最も近い ALU にチェーン接続するように設計されています。すべての条件付き信号とキャプチャ信号は、最下位バイトから最上位バイトへの方向でチェーン接続されます。左シフト信号も最下位バイトから最上位バイトへの方向でチェーン接続されます。右シフト信号は最上位バイトから最下位バイトへの方向でチェーン接続されます。

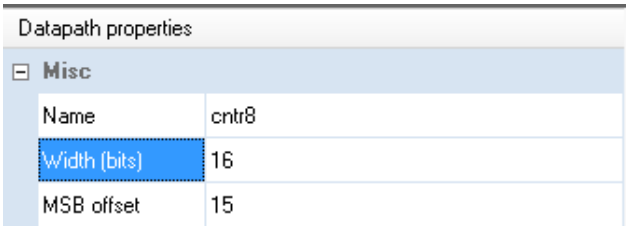
図 46. データパスのチェーン接続フロー



UDB エディターはデータパスをチェーン接続すること容易にします。作成した PWM を使用してそれを 16 ビット幅にします。

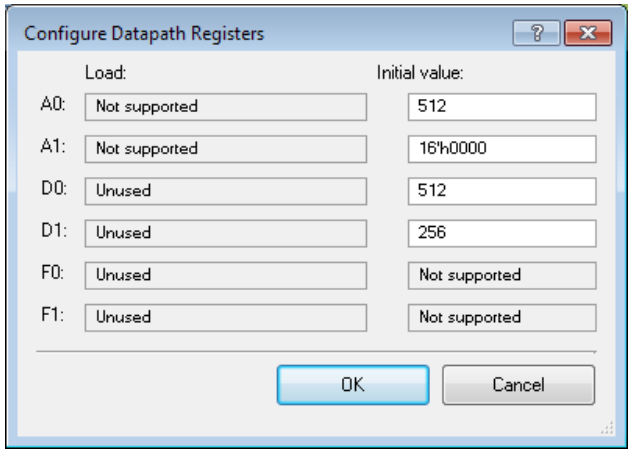
1. UDB ドキュメント (.cyudb) に移動します。
2. 図 47 に示すように Datapath properties パネルで、Width (bits) を 16 に設定します。

図 47. 16 ビットコンフィギュレーション



3. 図 48 に示すように周期値 (D0) と初期周期値 (A0) を 512 に、比較値 (D1) を 256 に設定します。

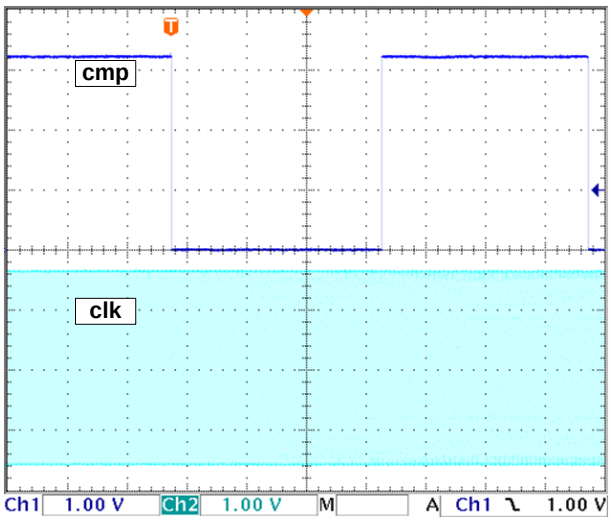
図 48. 16 ビット初期値



4. プロジェクト ウィンドウで **File** > **Save** を選択します。
5. トップ デザイン回路図 (.cysch) に戻って、クロックを 10kHz に 1MHz に変更します (オシロスコープで表示しやすくするため)。
6. **File** > **Save All** を選択して、プロジェクトをビルドし、PSoC をプログラムします。.

図 49 に示すように、周期値と比較値が作成した 8 ビット PWM よりはるかに大きいことを出力で確認できます。それらをいかなる 16 ビット値にも設定することができます。

図 49. 簡単な 16 ビット PWM の出力



チェーン接続を使用して 32 ビット幅までの機能を作成することができます。この例で説明する原理はより大きいビット幅の機能に適用できます。

6.7 PSoC 3 における 16 ビットコンポーネントのヘッダーファイル

前述したように、16 ビットレジスタへの書き込みは、8 ビットレジスタへの書き込みと違います。プロセッサとデータパス レジスタの間のエンディアンの相違を気にしなくてもかまわないため、8 ビットレジスタへ直接書き込むことができます。しかし、16 ビット以上のレジスタに書き込む際、エンディアンの相違を考慮しなければなりません。

PSoC 3 の 8051 のエンディアン性は、ペリフェラルレジスタのものとは異なります。レジスタへの書き込みを簡単にするために、サイプレスは CY_SET_REG16、CY_SET_REG24 および CY_SET_REG32 のマクロを提供しています。これらのマクロは、最初のパラメーターに、設定したいレジスタ アドレス、2 番目のパラメーターに、設定したい値を受けます。これらのマクロはエンディアンスワッピングを処理します。

そのため、データパス レジスタのアドレスを知る必要があります。これは、自動生成されたヘッダーファイル (_defs.h) 内の_PTR で終わる定義により自動的に行われます。

値を更新するために、CY_SET_REG16 およびヘッダーファイルからのポインターを使用します。本アプリケーション ノートに同梱されたサンプルプロジェクトを再度ご覧ください。

6.7.1 16 ビットパラメーター

前述した例では、シンボル パラメーターのタイプを uint8 に設定しましたが、16 ビットの PWM のパラメーターを追加することもできます。単にタイプを *uint16* に変更するだけです。

7 プロジェクト#2 – アップ／ダウンカウンター

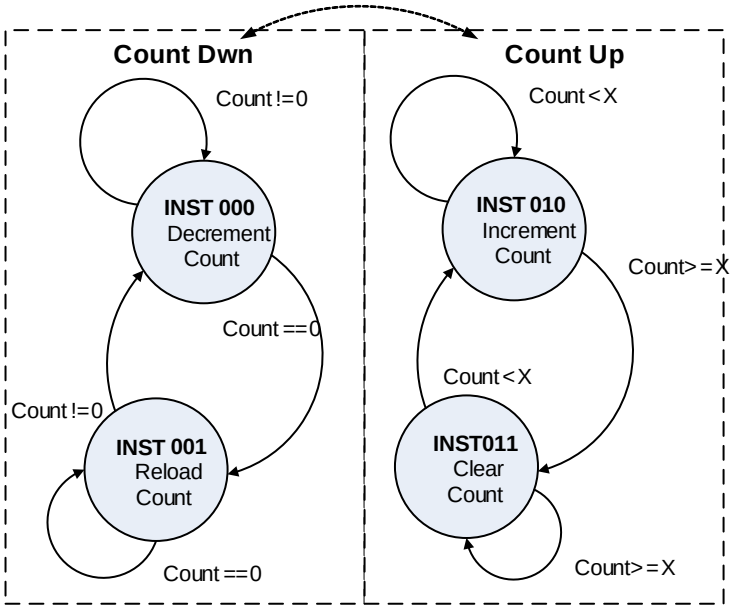
この例の目的は、高度な機能をデータパス コンポーネントに追加する手順を説明することです。同じ基本的な PWM 概念は、カウントアップ／カウントダウンの機能を追加するよう更新されます。カウント方向は、実行時に設定できるパラメーターに基づいています。

この例は、ユーザーが前述のサンプル プロジェクトで紹介された概念に精通していることを前提とします。本アプリケーションノートには、1 つの完成したアップ／ダウン PWM プロジェクトが同梱されています。

7.1 追加詳細

簡単なダウンカウント PWM は 2 つの状態を使用しました。図 50 に示すようにアップ／ダウン カウンターを実装するために、4 つの状態が必要です。

図 50. アップ／ダウンカウンターの状態図



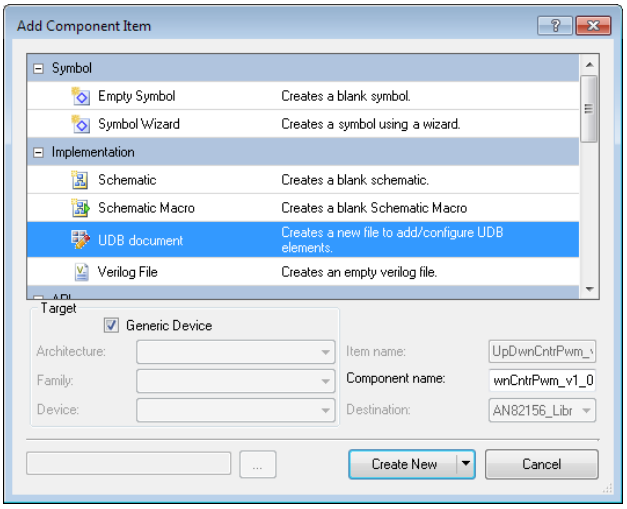
データパスは、セットしたパラメーターによって A0 をデクリメントするか、またはインクリメントします。周期値と比較値もセットできます。

7.2 サンプルプロジェクトの手順

混乱を避けるために、以前のサンプル プロジェクトのコンポーネントを変更する代わりに、新しいコンポーネントを作成します。コンポーネント作成の基本手順は同じです。

1. PSoC Creator を起動し、簡単な 8 ビット例で使用した「**AN82156**」ワークスペースを開きます。「UpDwnCntrPwm」という新しいプロジェクトをワークスペースに追加します。
2. Components タブで、Project ‘UpDwnCntrPwm’ を右クリックして Add Component Item を選択します。
3. UDB ドキュメントを選択し、**Component name:** を *UpDwnCntrPwm_v1_0* に変更します。**Create New** をクリックします。

図 51. 新しいコンポーネントの作成



4. データパス要素をデザイン キャンバスにドラッグします。
5. **Datapath properties** に移動し、**Name** を「*UpDwn*」に変更します。

図 50 に示す機能を実装するようにデータパスをコンフィギュレーションして機能を実装します。最初の 2 つの命令は簡単な 8 ビット PWM と同じで、もう 2 つの命令が追加されます。表 3 に示すように 2 つの追加の命令はアップ／カウント機能を実装します。

表 3. アップ／ダウンカウンターの命令表

INSTR_ADDR	関数	レジスタ書き込み	コメント
000	ALU = A0 - 1	A0 = ALUout	デクリメントカウント
001	No-op	A0 = D0	リロードカウント
010	ALU = A0 + 1	A0 = ALUout	インクリメントカウント
011	ALU = A0 ^ A0	A0 = ALUout	クリアカウント

XOR コンフィギュレーションはカウント (A0) レジスタをクリアするために使用されます。カウント (A0) レジスタは周期値に達した後、自身を XOR します。すると、カウントが 0 にクリアされます。

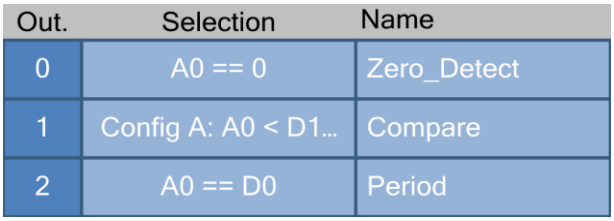
各命令は表 3 に示すように設定する必要があります。図 52 は各命令がデータパス要素にどのように表示されるかを示します。

図 52. UpDwnCntr の命令

Inst. Addr.	Instruction	Comment
3'b000	ALUout=(A0 - 1) A0=ALUout	Decrement Count
3'b001	ALUout=(A0) A0=D0	Reload Count
3'b010	ALUout=(A0 + 1) A0=ALUout	Increment Count
3'b011	ALUout=(A0 ^ A0) A0=ALUout	Clear Count

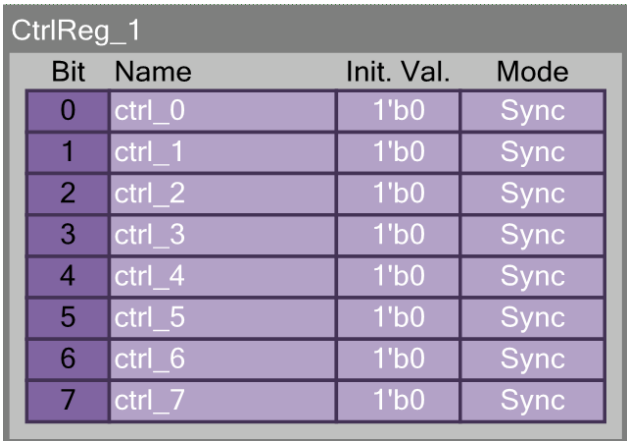
6. **Datapath properties** パネルで、**Configurable comparator inputs** の **Config A** が「*A0 compare to D1*」に設定されたことを確認します。
7. **Compare options** の各命令が「*ConfigA: A0 compares to D1*」に設定されたことを確認します。
8. 最初の例と同じように Zero_Detect と比較出力を設定し、また 3 番目の出力 (Period) を追加します。3 番目の出力はカウンターが周期レジスタ (D0) の値に達すると HIGH になります。これは、アップカウンターが周期値に達してリセットの必要があることを示します。図 53 を参照してください。

図 53. UpDwnCntr の出力



9. 図 54 に示すように制御レジスタ (CR) を回路図にドラッグします。

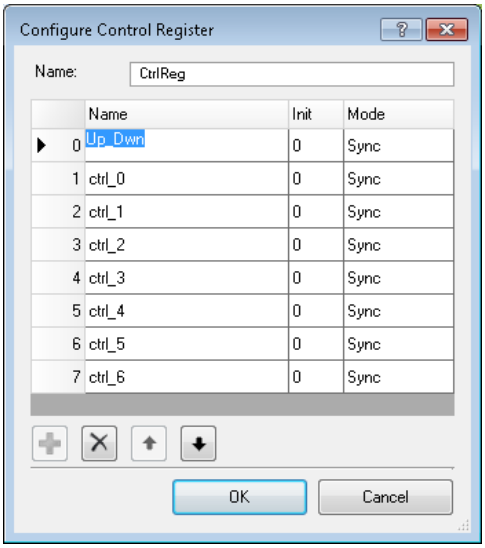
図 54. 制御レジスタ



10. 制御レジスタをダブルクリックします。ビット 0 の Name を Up_Down に設定します。図 55 に示すように制御レジスタ名を CtrlReg に変更します。

制御レジスタは CPU によって書き込まれます。そのため、CPU はこの制御レジスタのビットに書き込むことでカウント方向を制御します。値 1 はアップ カウントを示し、値 0 はダウン カウントを示します。

図 55. 制御レジスタのコンフィギュレーション



最初の例では、データパス命令が 2 つあるため、Zero_Detect 信号を使用して CFGRAM から実行される命令を容易に選択できました。この例では、命令が 4 つあります: ダウン カウント用の 2 つおよびアップ カウント用の 2 つ。

Zero_Detect をダウン カウントに依然として使用できます。アップ カウントには Period 信号を使用します。カウントアップしているかカウントダウンしているかを判定するための信号もあります (Up_Dwn)。4 つの命令があるため、2 つのアドレス ビットが必要です。しかし、表 4 に示すように Up_Dwn、Period、Zero_Dectect の 3 本の信号があります。したがって、これらの 3 本の信号を 2 本のアドレス ラインに減らすためにロジックを使用する必要があります。

表 4. UpDwnPWM 命令の復号化

Up_Dwn	Period	Zero_Detect	INSTR_ADDR	機能
Down	該当なし	0	000	デクリメント
Down	該当なし	1	001	リロード
Up	0	該当なし	010	インクリメント
Up	1	該当なし	011	クリア

表 4 に示すように、Up_Dwn 信号に常に接続するアドレス ビットが 1 つあります。その他のアドレス ビットは Period と Zero_Detect 間で多重化されます。Up_Dwn 信号が「Down」にセットされた場合、アドレス ビットに Zero_Detect の値を使用し、Up_Dwn が「Up」にセットされた場合、Period の値を使用します。

INSTR_ADDR[0] = if(up) Period else if(down) Zero_Detect

INSTR_ADDR[1] = Up_Dwn

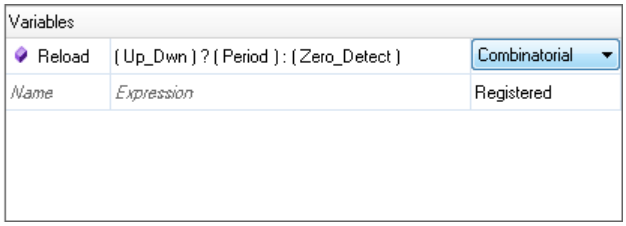
UDB エディターでは、様々なフィールドに標準 Verilog 構文を入力することができます。また、ロジックで制御できる変数を作成することもできます。この例では、INSTR_ADDR[0]に Period か Zero_Detect を使用するかを決定するための変数を作成します。

11. **Variables** の **Properties** ウィンドウで、*Reload* と名付けた変数を追加し、式を以下のように設定します。

    (Up_Dwn) ? (Period) : (Zero_Detect)

12. 図 56 に示すようにこの変数が **Combinatorial** にセットされたことを確認します。

図 56. 変数の定義



拡張 *Reload = (Up_Dwn) ? (Period) : (Zero_Detect)* 式は 3 項演算子です。 これは、簡単な if-else 文を書くよりコンパクトな方法です。

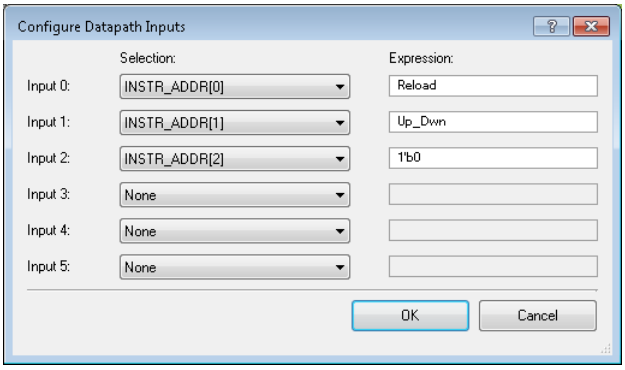
形式は以下の通りです。

A = B ? C: D。B が真 (HIGH) であれば、A = C となり、B が偽 (LOW) であれば、A = D となります。

この例では、Up_Dwn が HIGH であれば、Reload = Period となります。Up_Dwn が LOW であれば、Reload = Zero_Detect となります。

13. データパス入力に移動して、図 57 に示すように INSTR_ADDR[0]を「Reload」に、INSTR_ADDR[1]を「Up_Dwn」に設定します。

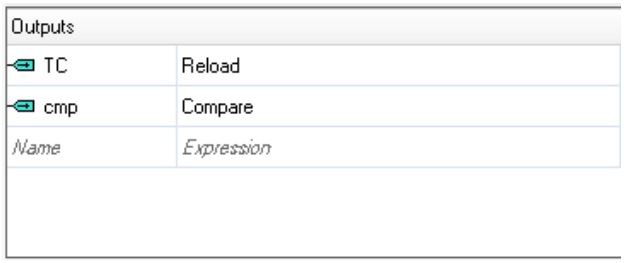
図 57. 命令のアドレス指定



14. 次に、PWM 出力を設定します。前述したように、ターミナル カウント (TC) および比較 (cmp) 出力があります。cmp は Compare 信号の出力です。TC 信号は新しい Reload 変数です。

15. 図 58 に示すように出力を設定します。

図 58. 出力



16. .cyudb ファイル内の空白領域を右クリックして **Generate Symbol** を選択します。

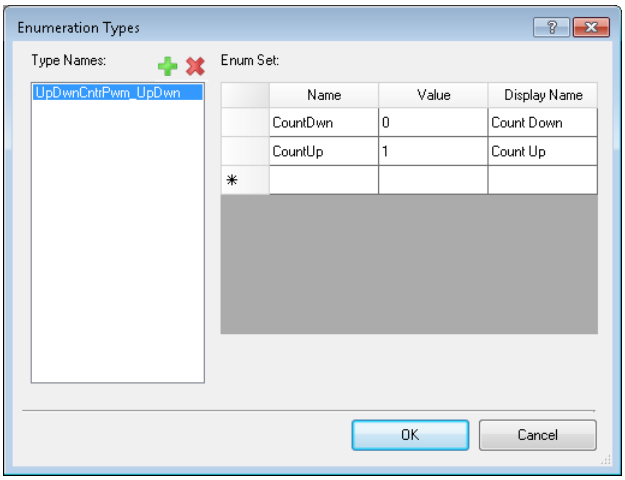
17. シンボル回路図ページ (.cysym) を右クリックして以下のようにシンボル プロパティを追加します。

- Doc.APIPrefix = *UpDwnCntrPwm*
- Doc.CatalogPlacement = AN82156/Digital/UpDwnCntrPwm
- Doc.DefaultInstanceName = *UpDwnCntrPwm*

カウンター周期、比較値、カウント モード (アップ／ダウン) などいくつかの変更可能なパラメーターを追加します。カウント モードの設定のために、新しいパラメーター タイプおよび値一式を定義します。

18. シンボル エディター ページを右クリックして Symbol Parameters ダイアログを開きます。
19. **Types** をクリックしてウィンドウを開き、新しいパラメーター タイプを作成します。
20. 緑色の「+」ボタンをクリックして新しいタイプを追加します。それを *UpDwnCntrPwm_UpDwn* と名付けます。
21. 図 59 に示すように **Enum Set** フィールドに値を入力して「CountDwn」と「CountUp」を定義します。

図 59. 新しいコンポーネントパラメータータイプの作成



22. **OK** をクリックして Symbol Parameters ダイアログに戻ります。

この新しいタイプにパラメーターを割り当て、0 (CountDwn) または 1 (CountUp) の初期値をセットすることができます。

23. 前の例と同じようにコンポーネントの 3 つの新しいパラメーターを追加します。表 5 を参照してください。

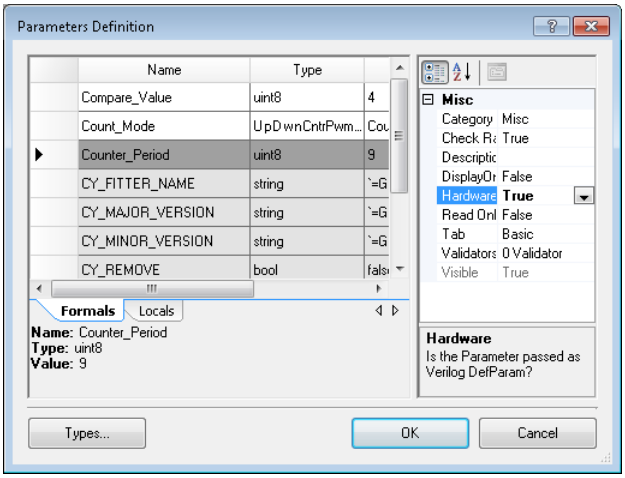
表 5. アップ／ダウンカウンターのパラメーター

パラメーター名	タイプ	値
Compare_Value	uint8	4
Count_Mode	UpDwnCntrPWM_UpDwn	Count Down
Counter_Period	uint8	9

Count_Mode パラメーターは新しいタイプと値の定義を使用します。

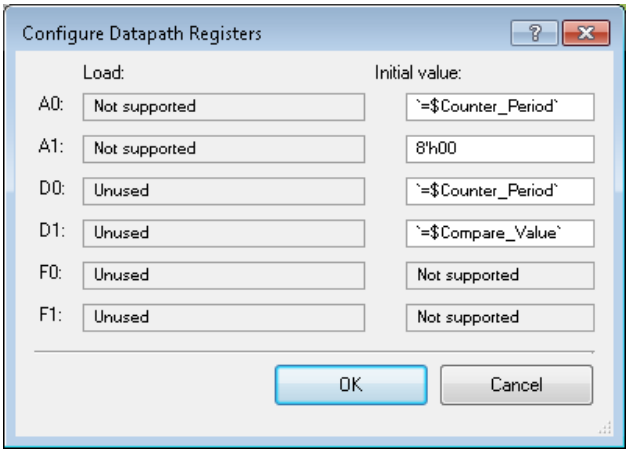
24. 図 60 に示すように、すべての 3 つの新しいパラメーターに対して **Hardware** フラグを「True」にセットします。

図 60. 新しいコンポーネントパラメーターの追加



25. **OK** をクリックしてコンポーネントへの変更を保存します。
26. .cyudb ファイルに戻って、前の例と同じようにレジスタの初期値を変更します。図 61 を参照してください。

図 61. パラメーターおよび初期値



このときハードウェアで制御レジスタの初期値をセットすることはできません。そのため、ファームウェアで設定を行います。

27. ヘッダーファイルを作成します。Components タブで、**UpDwnCntrPwm_v1_0** を右クリックして **Add Component Item** を選択します。**API Header File** を選択してそれを *UpDwnCntrPwm.h* と名付けます。**Create New** をクリックします。
28. ヘッダー ファイルで、以下のコード行を /* [] END OF FILE */: の上に加えます。

```
#define UP_DOWN      `$Count_Mode`
```

これで、UP_DOWN 定義はコンポーネント カスタマイザーでセットした Count_Mode パラメーターにリンクさせられます。

インクルードされた制御レジスタを使用してカウンターの方向を *main.c* ファイルで設定します。すべての標準 API は使用可能であることに注意してください。

29. *main.c* ファイルで、以下のコード行を加えます。

```
UpDwnCntrPwm_1_CtrlReg_Write(UP_DOWN);
```

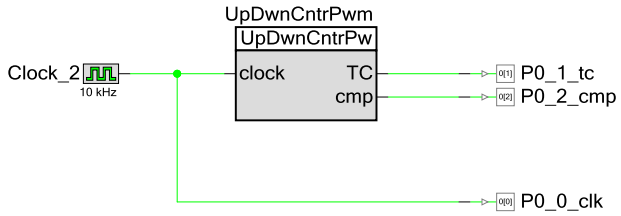
コンポーネントを「UpDwnCntrPwm_1」と名付けなかった場合、コンポーネント名で置き換える必要があります。制御レジスタを「CtrlReg」と名付けなかった場合、制御レジスタ名で置き換えてください。ステップ 28 に示すように.*cyudb* ファイルに制御レジスタを配置することで、標準制御レジスタの API へ直接アクセスことができるようになります。

これで、コンポーネントは使用可能になりました。最初のサンプル プロジェクトと同じすべてのコンポーネントを追加します。

30. UpDwnCntrPwm コンポーネントをプロジェクト回路図にドラッグします。
31. クロック コンポーネントをコンポーネントの「clock」端末に接続して 10kHz に設定します。
32. 図 62 に示すようにデジタル出力ピン コンポーネントをコンポーネントの P0_0_clk、P0_1_tc、P0_2_cmp 端末に接続します。

    注: PSoC 6 MCU と PSoC 4 では、このプロセスは異なります。 6.2 節のステップ 23 を参照してください。

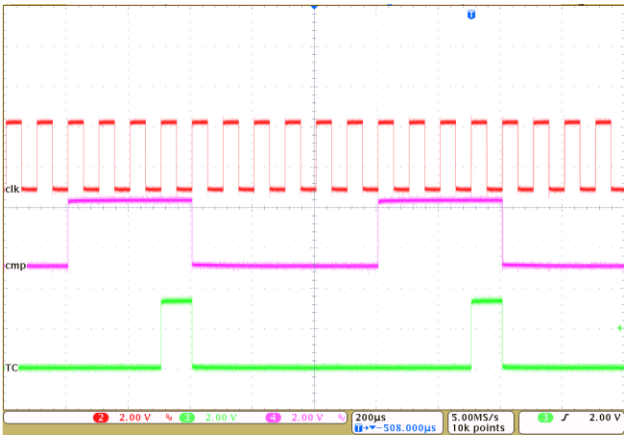
図 62. アップ／ダウンカウンターPWM プロジェクトの回路図



33. **File** > **Save All** を選択します。
34. プロジェクトをビルドし、PSoC をプログラムします。

    比較値パラメーターを 4 に、周期値を 9 に、Count_Mode を Count_down に設定します。オシロスコープで clk、cmp および TC 波形を観察します。図 63 に示すように、TC が HIGH になった後、カウントがリロードする時に cmp ラインは LOW になるため、ダウン カウントを確認できます。

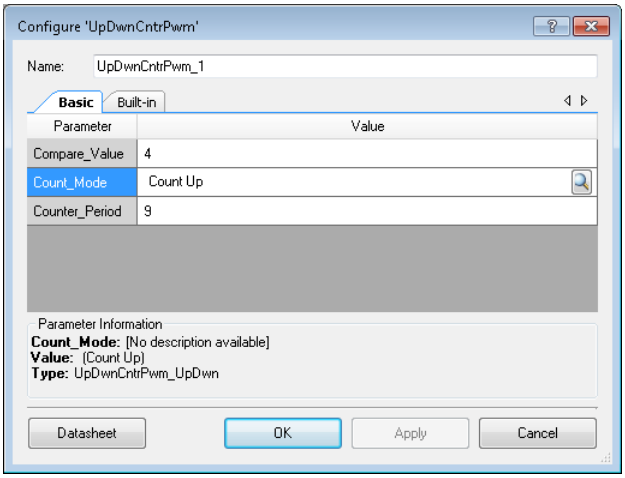
図 63. ダウンカウント PWM の波形



簡単な PWM の例で行ったように、周期値と比較値を変更することができます。PWM がカウントダウンする代わりにカウントアップするようにモード パラメーターを変更することも可能です。

35. プロジェクト回路図に戻り、UpDwnCntrPwm コンポーネントをダブルクリックしてプロパティダイアログを開きます。
36. 図 64 に示すように Count_Mode を「Count Up」に変更します。

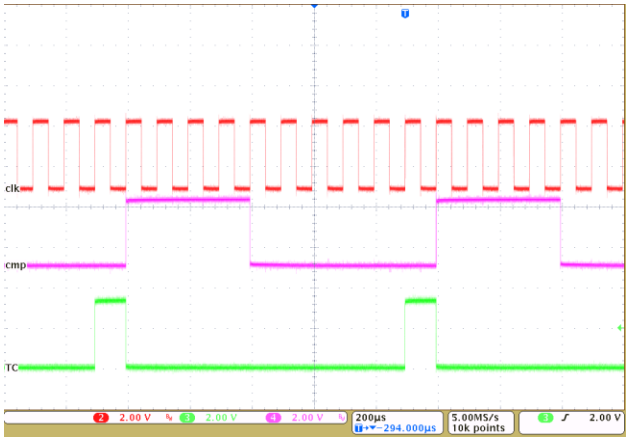
図 64. アップカウントへの PWM の設定



37. **OK** をクリックして変更を適用します。
38. **File** > **Save All** を選択して、プロジェクトをビルドし、PSoC をプログラムします。

図 65 に示すように、TC の後にカウンターが 0 でリロードされる時 cmp 値が HIGH になるため、アップ カウンターを確認できます。

図 65. アップカウント PWM 波形



TC の後に出力が HIGH になる理由は、カウントの開始値が比較値より小さく、その後インクリメントされたためであることに注意してください。Down モードの始まりに出力が LOW になる理由は、カウントの開始値が比較値より大きく、その後デクリメントされたためです。

8 プロジェクト#3 – 簡単な UART

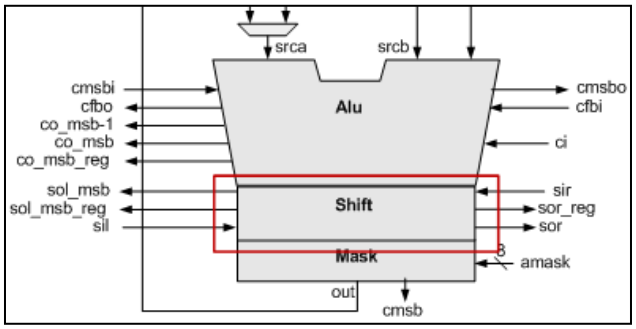
このサンプル プロジェクトは、簡単な TX UART を単一のデータパスで作成することをデモします。ここでは、コンポーネントの作成の各ステップについて説明しません。その代わりに、関連したプロジェクト内でコンポーネントと UDB エディター ドキュメントを読むことができます。完成した例の中の Simple_Tx プロジェクト内の「Simple_Tx」というコンポーネントを探します。このコンポーネントの使用方法を示す例は同じワークスペース内の「SimpleTx」プロジェクトに含まれています。

8.1 TX UART コンポーネントの詳細

このコンポーネントでのデータパスの使用方法は、値を F0 からシフトして A0 にロードする動作を使用します。

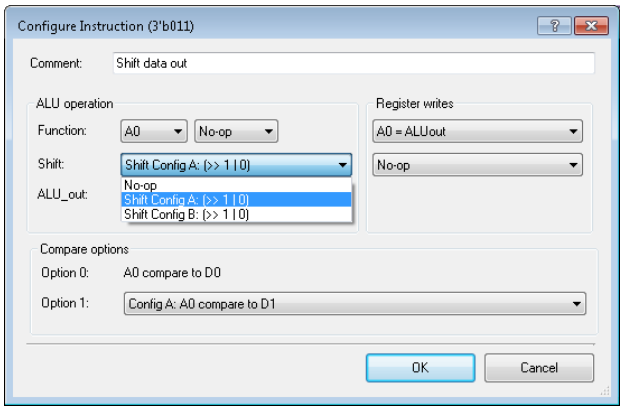
図 66 に示すように、ALU の出力に 1 個のシフターがあります。

図 66. シフターブロック



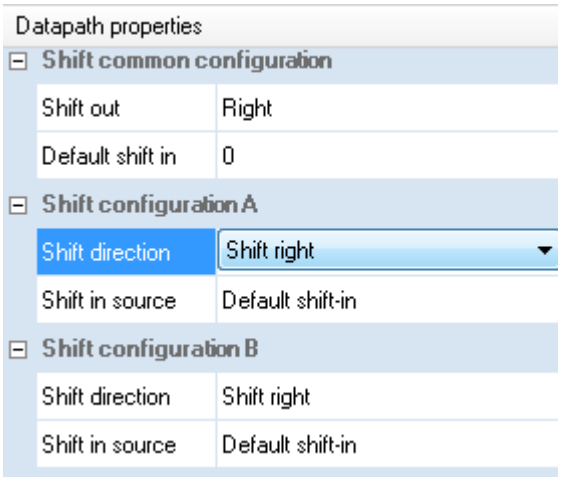
このシフターはビットを左か右にシフトできます。データパスの各命令は個別にシフターの動作を設定できます。このオプションは図 67 に示すように、各命令の **Configure Instruction** ダイアログでセットされます。

図 67. シフト動作の設定



Config A と Config B コンフィギュレーションを設定するために、**Datapath properties** に移動して、図 68 に示すように **Shift configuration A** と **Shift configuration B** を設定します。

図 68. データパスシフトコンフィギュレーション



シフトイン ソースは DSI からのシフトイン (SI) 信号であるか、または初期設定のシフトイン値 (0、1) です。

データパス出力マルチプレクサには 1 本のみのシフトアウト (SO) 出力があります。この出力は、右シフトアウトと左シフトアウトの間で共有されます。図 68 に示す **Shift common configuration** の *Shift Out* でこのマルチプレクサを設定する必要があります。

この例では、データパスの命令を制御するステートマシンを作成します。このステートマシンは UDB PLD に実装され、UART 送信のどの部分 (IDLE、Start、Data、あるいは Stop) を次に行うかを決めます。

ステートマシンは、データパスの動作を制御するために使用される状態を 4 つだけ持っています。そのため、データパスは 4 種の個別の動作を必要とします。

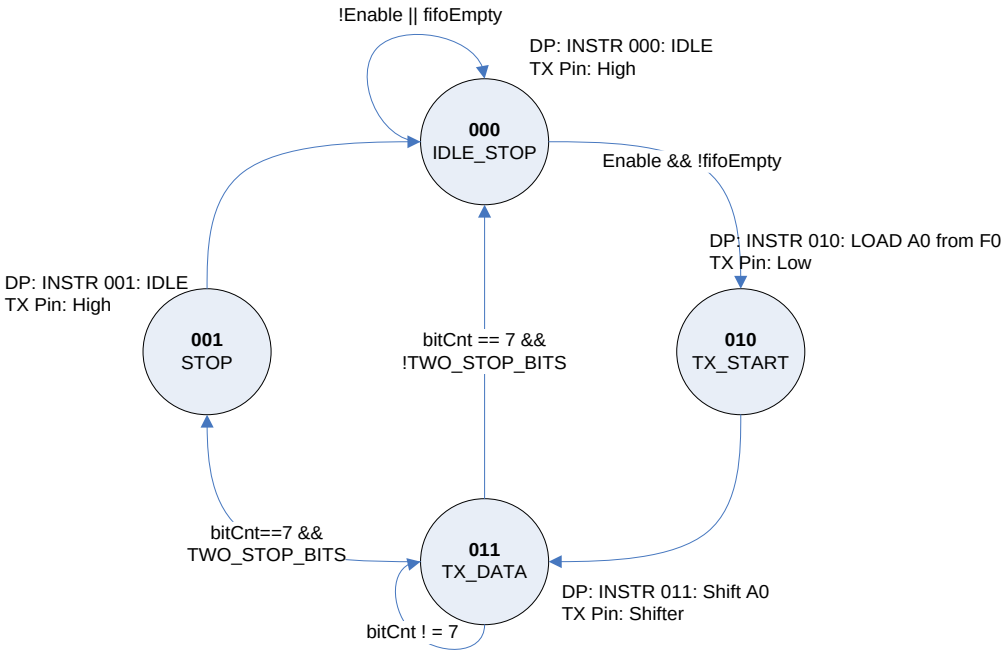
表 6. 3 つのデータパス命令の例

状態	INSTR_ADDR	機能	SHIFT	レジスタ書き込み
IDLE_STOP	000	No-op	なし	なし
STOP	001	No-op	なし	なし
TX_START	010	No-op	なし	A0 = F0
TX_DATA	011	No-op	SR	A0 = ALU

コードはステートマシンを移動するにしたがって、データパス命令を変更します。これは一般的な使用例です。

ほとんどの複雑なコンポーネントは、データパスの動作に順序付けるためにステートマシンを必要とします。図 69 は簡単な TX のステートマシンの動作方法を示します。

図 69. TX UART ステートマシンのフロー図

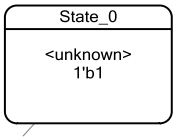


最初に、ステートマシンは fifoEmpty ステータス ビットを監視することで、新しいデータが FIFO に書き込まれるのを待ちます。FIFO にデータが書き込まれると、ステートマシンは Start 状態に進み、TX ラインを LOW にセットすることで START ビットを送信します。この状態では、データパスは FIFO (F0)内の値を A0 にロードします。

次の状態で、データパスは A0 内のデータを LSB ファースト方式 (右シフト) で TX ピンにシフトアウトします。次に、ステートマシンは 1 個か 2 個の STOP ビットを送信します。

UDB エディターでは、図 69 のようなステートマシンを作成することが可能です。このためには、図 70 に示すようにステートマシンの要素をデザイン キャンバスにドラッグ & ドロップします。

図 70. ステートマシンの要素



これらの 4 つの要素をデザイン キャンバスにドラッグして、図 69 に示すように接続します。状態を接続するために線を描くたびに、遷移が行われるタイミングを決める式を書くためのダイアログが表示されます。図 71 を参照してください。

図 71. 状態遷移の式

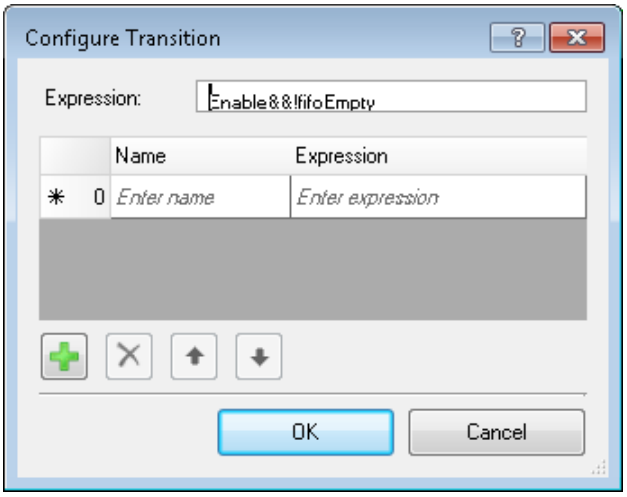
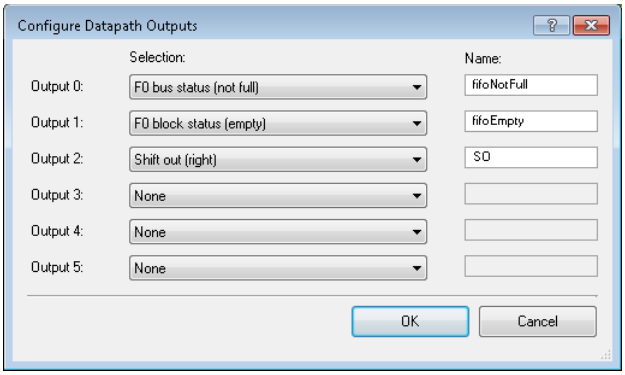


図 71 に示した遷移は、イネーブル信号が HIGH、かつ fifoEmpty 信号が HIGH でない時に行われます。これは、ステートマシンが IDLE 状態から Start 状態に進むタイミングを制御する遷移です。遷移が行われるには、制御レジスタからのイネーブル信号は HIGH、かつデータパスの FIFO が空でないことが必要です。図 72 に示すように fifoEmpty 信号はデータパスのいずれかの出力から来ます。この信号は、FIFO にデータがない時に HIGH になり、FIFO にデータがある時に LOW になります。

図 72. データパスの出力



各状態中に、変数を追加してそれらに対しロジックや算術関数を実行することができます。例えば、簡単な UART は 8 データ ビットをシフトアウトします。つまり、データパスは Data (シフト) 状態を 8 クロック サイクル維持する必要があります。

表 6 に示すように INSTR_ADDR が 011b になるとシフトが行われます。図 73 は第 3 状態 (011b) のコンフィギュレーションを示します。図 73 に示すように、Variable assignment の中で、bitCnt 変数は bitCnt +1 という式を持っていることに注意してください。つまり、状態が行われるたびに、bitCnt = bitCnt+1 となります。

ステートマシンの状態は、データパスが実行されるクロックと同じである入力クロックのポジティブ エッジで行われます。

図 73. 状態のコンフィギュレーション

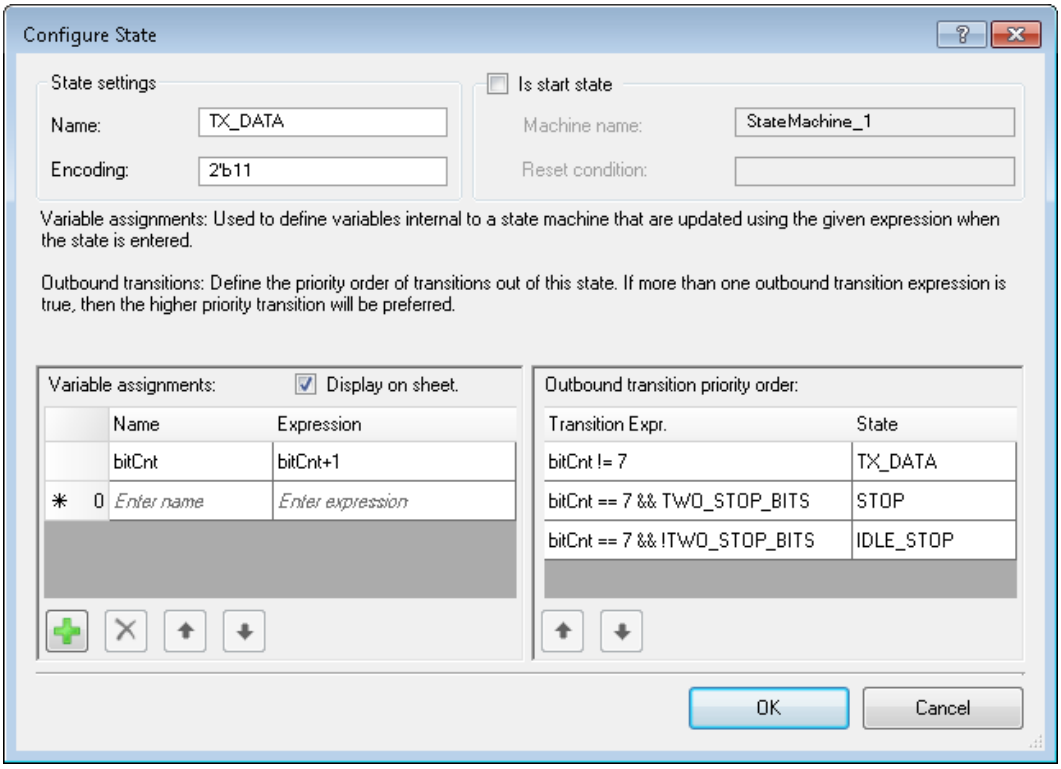
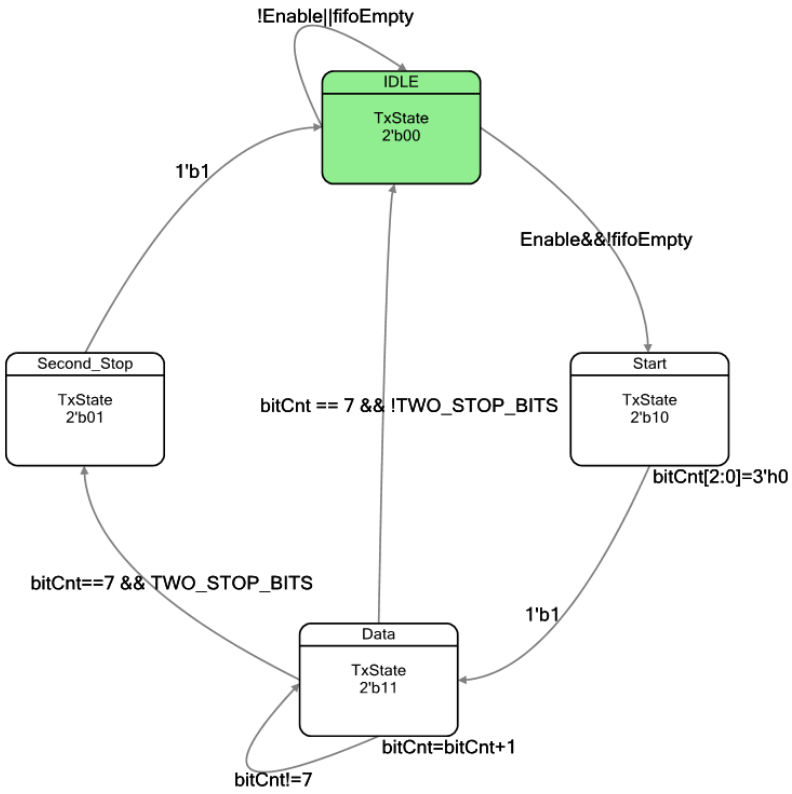


図 73 は、状態遷移を設定するための **Outbound transition priority order** ダイアログを示します。bitCnt が 7 に等しくない限り、ステートマシンは Data 状態を維持することに注意してください。前述したように、8 ビットをシフトアウトする必要がありますが、この状態を 7 サイクル維持する理由は何でしょうか。実際には、この状態を 8 サイクル維持します。遷移は bitCnt が増える前に評価されました。この式が初めて評価される時、bitCnt は 0 です。

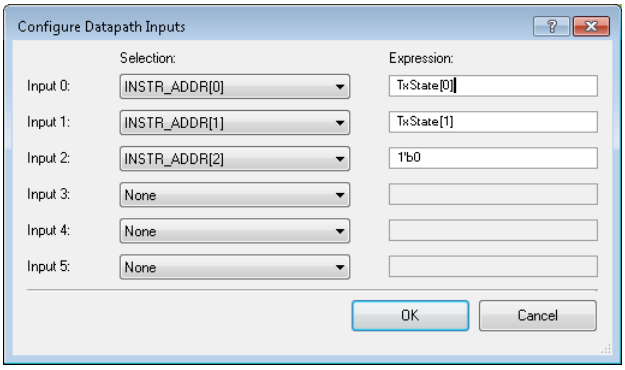
図 69 に一致するように、状態が遷移するたびに同じステップが行われます。図 74 は、UDB エディターが複製した図 69 を示します。

図 74. UDB エディターのステートマシン図



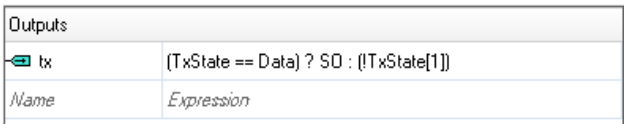
次に、ステートマシンによりデータパス命令を制御します。図 74 に示すようにステートマシンは *TXState* と名付けられています。このステートマシンには 4 つの状態があるため、UDB エディターは TxState のために 2 ビット信号を作成します。この 2 ビット信号はデータパスの INSTR_ADDR 信号に接続することができます。図 75 を参照してください。

図 75. ステートマシンで制御されるデータパス入力



次に、TX 出力ラインの制御方法を説明します。前述したように、Data 状態中にデータがシフトアウトされます。Start 状態では TX ラインを LOW にセットし、Stop および IDLE 状態では TX ラインを HIGH にセットします。これは、図 76 に示す出力の式で行います。

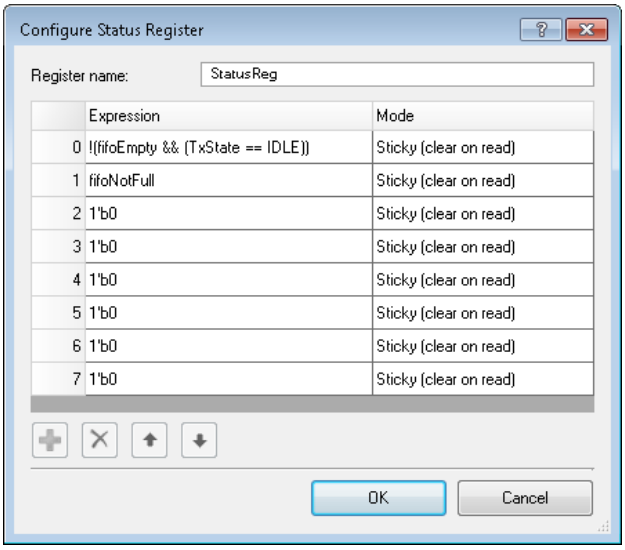
図 76. Tx 出力の式



ステートマシンは Data 状態にない場合、「TxState」の最上位ビットの反転を TX ラインに駆動します。Start 状態では、最上位ビットが 1 であるため、「0」を出力します。Stop および IDLE 状態では、最上位ビットが 0 であるため、「1」を出力します。Data 状態では、SO (シフトアウト) 信号を出力します。

このデザインに追加された唯一の要素は、ステータスレジスタです。スタータスレジスタの目的は、ステートマシンおよびデータパスの状態を CPU に通知することです。

図 77. ステータスレジスタのコンフィギュレーション



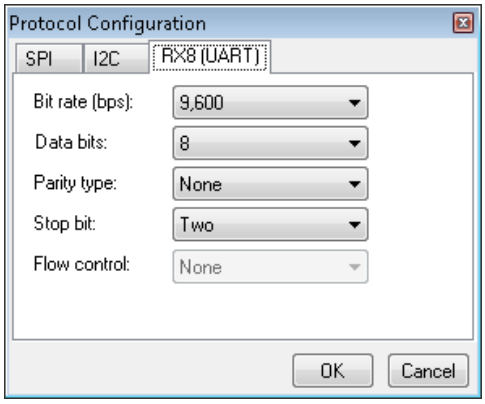
このステータスレジスタのビット 1 は UART がビジーであるタイミングを CPU に通知します。ステートマシンは、UART がデータ送信を完了するまでこの状態にあります。ビット 2 は FIFO が満杯でない状態にあるかを通知します。この信号は、FIFO が満杯になるまでデータを FIFO にロードするために CPU によって使用されます。

このコンポーネントにはヘッダーファイルが追加されています。ヘッダーファイルは、コンポーネントを有効にして 1 ストップビットモードまたは 2 ストップビットモードに設定すること、およびステータスレジスタの定義のためにいくつかの定数を定義します。ヘッダーファイルでは、定義がステータスレジスタと制御レジスタで選択されたビット位置に一致することに注意を払う必要があります。

このプロジェクトの main コードは 2 ストップビットでコンポーネントを有効にしてから、値 0〜10 を連続して送信します。これは 9600 ボーで行います。レシーバーを 9600 ボーおよび 2 ストップビットにコンフィギュレーションします。

ブリッジ コントロール パネル (BCP) というプログラムが PSoC Creator のインストールに含まれています。BCP を使用して RX キャラクタを受信することができます。BCP では、TX 出力が接続する COM ポートに接続します。**Tools > Protocol Configuration** では、図 78 のように RX8 (UART) をコンフィギュレーションします。

図 78. BCP RX8 のコンフィギュレーション



BCP のエディターでは、次のテキストを加えます: rx8 x x x x x x x x x x。これは、選択した COM ポートから 11 バイトを読み出します。引き続きデータを受信するために「Repeat」ボタンを押すことができます。

9 UDB エディターからデータパスコンフィギュレーションツールへの移植

望ましい機能を UDB エディターで実装することが不可能である場合があります。この場合、Verilog ファイルおよびデータパス コンフィギュレーション ツールに移植するほかありません。この手順はかなり簡単です。付録 A – データパスコンフィギュレーションツールの使用例に述べる Verilog ファイルの作成手順に従って、UDB エディターで生成された Verilog コードを作成したばかりの Verilog ファイルにコピー & ペーストします。これで、データパス コンフィギュレーション ツールを用いて Verilog ファイルを自由に変更することができます。コンポーネントの作成およびデータパス コンフィギュレーション ツールの使用例は、付録 A – データパスコンフィギュレーションツールの使用例を参照してください。

10 まとめ

UDB データパスを使用すると、PSoC のプログラム可能なロジックでコンポーネント作成に柔軟性が増します。UDB データパスの理解と効果的な使用により、PSoC 3、PSoC 4 および PSoC 5LP の能力を、従来のマイクロプロセッサが提供するものよりも拡張することができます。

本アプリケーション ノートで説明したサンプル プロジェクトは、カスタマイズする独自のソリューションを作成するための単なる出発点に過ぎません。コンポーネントの機能および複雑さの追加の詳細については、PSoC のアーキテクチャ TRM およびコンポーネント作成者ガイド (CAG) をご覧ください。

11 関連リソース

11.1 アプリケーションノート

- AN54181 – Getting Started with PSoC 3
- AN79953 – Getting Started with PSoC 4
- AN77759 – Getting Started with PSoC 5
- AN221774 – Getting Started with PSoC 6 MCUs
- AN81623 – PSoC 3 and PSoC 5 Digital Design Best Practices
- AN82250 – PSoC 3, PSoC 4 and PSoC 5 Implementing Programmable Logic Designs

11.2 KB 記事

- KBA86838 – Datapath Configuration Tool Cheat Sheet
- KBA86336 – Just Enough Verilog for PSoC
- KBA86338 – Creating a Verilog-based Component
- KBA81772 – Adding Component Primitives / Verilog Components to a Project
- Basics of Verilog and Datapath Configuration Tool for Component Creation

11.3 TRM

- PSoC 3 Architecture TRM
- PSoC 4 Architecture TRM
- PSoC 5LP Architecture TRM
- PSoC 6 MCU Architecture TRM

11.4 ビデオ

以下のビデオは、PSoC Creator と Verilog コンポーネントの作成プロセスについて説明します。

11.4.1 基本

- Creating a New Component Symbol
- Creating a Verilog Implementation

11.4.2 コンポーネント作成

- PSoC Creator 113: PLD Based Verilog Components
- PSoC Creator 210: Intro to Datapath Components
- PSoC Creator 211: Datapath Computation
- PSoC Creator 212: Datapath FIFOs
- PSoC Creator 213: Multi-Byte Datapath Components
- PSoC Creator 214: Datapath API Generation
- PSoC Creator Tutorial: Using the UDB Editor – Part 1
- PSoC Creator Tutorial: Using the UDB Editor – Part 2
- PSoC Creator Tutorial: Using the UDB Editor – Part 3
- PSoC Creator Tutorial: Using the UDB Editor – Part 4
- PSoC Creator Tutorial: Using the UDB Editor – Part 5
- PSoC Creator Tutorial: Using the UDB Editor – Part 6

12 著者について

氏名:	Todd Dust
肩書:	アプリケーションエンジニア
経歴:	シアトルパシフィック大学で電気工学の学士号を取得しました。それ以来、サイプレスでアプリケーション エンジニアとして働いています。
氏名:	Greg Reynolds
経歴:	10 年以上にわたってサイプレスでいくつかの職務をこなしてきました。

A 付録 A – データパスコンフィギュレーションツールの使用例

この付録では、アプリケーションノートの本文で作成したものと同じプロジェクトを作成する方法を説明します。ただし、今回はデータパス設定ツールを使用します。パラレルインおよびパラレルアウトの例も追加されています。この付録で作成されたプロジェクトのリストは次のとおりです。

- 8 ビットダウンカウンター
- 8 ビット PWM
- 16 ビット PWM
- 8 ビットアップ／ダウンカウント PWM
- TX 専用の簡単な UART
- パラレル入力およびパラレル出力

サンプル プロジェクトは PSoC 3、PSoC 4 または PSoC 5LP どのデバイスにも使用できます。完全なサンプル プロジェクトはサイプレスのウェブサイトでのこのアプリケーション ノートのランディング ページに掲載されます。

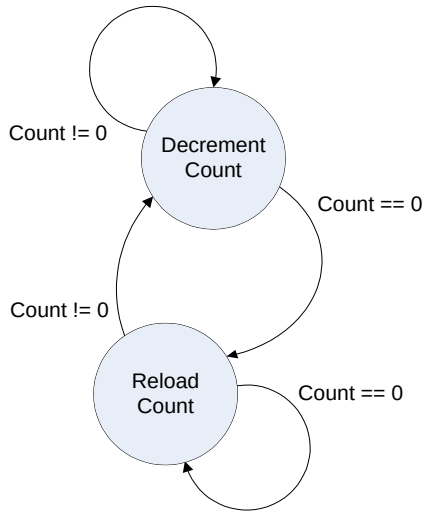
A.1 プロジェクト#1 – 8 ビットダウンカウンター

このプロジェクトの目的は、簡単なデータパスベースのコンポーネントを作成するために必要な手順を紹介することです。これは、簡単な 8 ビットダウンカウンターを作成することによって示されます。

A.1.1 8 ビットカウンターコンポーネントの詳細

図 6 に示した通り、簡単なダウンカウンターは 2 つの状態を持つステート マシンと考えられます。

図 79. 簡単なカウンターの状態図



カウンターは初期値で開始し、その値をデクリメントします。カウントが 0 になると、イベントがトリガーされ、周期の値はカウンターに再びロードされます。この種類のカウンターは容易にデータパスに実装されます。

このカウンターの実装には、2 つのデータパス命令のみが必要です。最初の命令は、カウント値をデクリメントします。2 番目はカウントを初期値でロードします。

この例で使用される 2 つのデータパスレジスタは以下の通りです。

- A0 – カウント値を格納し、ALU によってデクリメントされる。
- D0 – カウントが 0 に達した時に A0 に再びロードされる値を格納する。

このカウンターが機能するためには、データパスがどの命令を実行しているのかを判断する方法 (ロードまたはデクリメント) が必要です。 図 6 は、遷移がカウントの値、つまりカウントがゼロかどうかによって制御されることを示しています。

データパスは、A0 と A1 のデータの値を監視するゼロ検出ブロック (ZDET) があります。このブロックは z0 (A0 == 0) と z1(A1 == 0) という 2 つの出力があり、それぞれ A0 と A1 の状態を示します。各出力は、値が 0 であれば HIGH になり、値が 0 ではなければ LOW になります。

この例では、z0 出力を使用してどの命令を実行するかを制御します。

データパスは 8 つの異なる命令を持てることを忘れないでください。 データパスで使用される命令は、3 ビットのアドレスラインによって選択されます。 このカウンターについて前述したように、必要な命令は 2 つだけなので、ビット 0 を z0 に接続し、他のビットを LOW に保持できます。 表 2 は遷移表です。

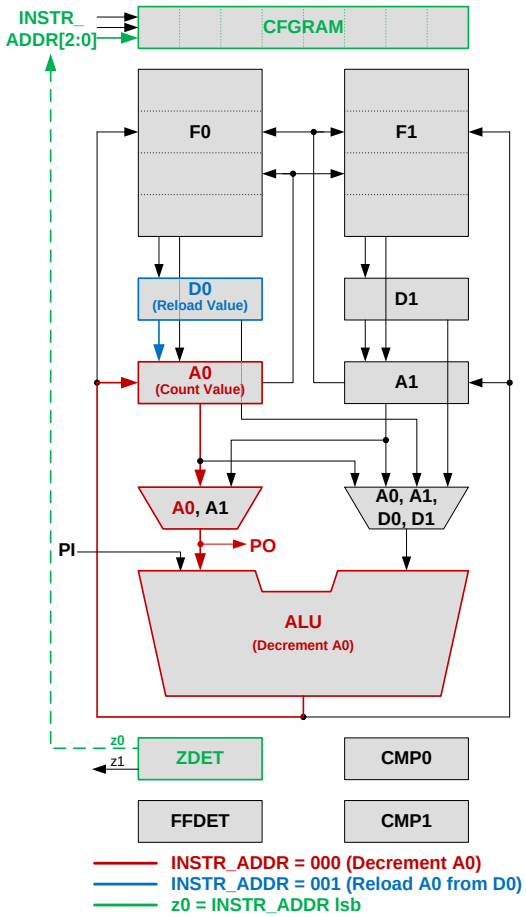
表 7. データパス命令

CFGRAM コンフィギュレーション	命令アドレスビット (INSTR_ADDR)			動作
	2	1	0	
0	0	0	z0 = 0	デクリメントカウント
1	0	0	z0 = 1	リロードカウント
2～7	X	X	X	未使用

A0 でのカウントは 0 に達すると、z0 は 1 になります。これによって、データパス命令は「リロードカウント」になります。カウントが D0 から A0 にリロードされる時、z0 は 0 になり、命令は「デクリメントカウント」になります。

この動作を視覚化するには、図 7 に示すように強調表示されるデータパス ブロック図をご覧ください。

図 80. 強調表示される簡単なカウンターブロック図



A.1.2 8 ビットカウンターコンポーネントの作成手順

1. PSoC Creator を起動して、使用しているデバイスをターゲットにした新しいデザインプロジェクトを作成します。最初に空の回路図を選択し、ワークスペース名とプロジェクト名を AN82156_Appendix に設定します。

   コンポーネントは PSoC Creator ライブラリプロジェクトに格納されます。自分のライブラリとコンポーネントがサイプレスの標準コンポーネントライブラリと混同しないように、固有の名前を付けます。

2. Workspace Explorer の **Source** タブ内のワークスペース「**AN82156_Appendix**」を右クリックし、表示するドロップダウン メニューから **Add > New Project…**を選択します。

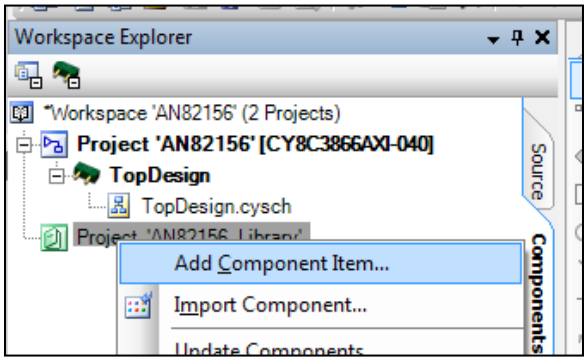
3. **Library project** を選択し、「*AN82156_Appendix_Lib*」と名前を付けて、場所をデフォルト値のままにします。これにより、*AN82156_Appendix* ワークスペースと同じ場所にライブラリが作成されます。

   新しいライブラリは Workspace Explorer に表示されます。そして、このライブラリに格納されるコンポーネントを使用できるようにライブラリをサンプル プロジェクトにリンクします。

   次に、その新しいライブラリに新しいコンポーネントを追加する必要があります。

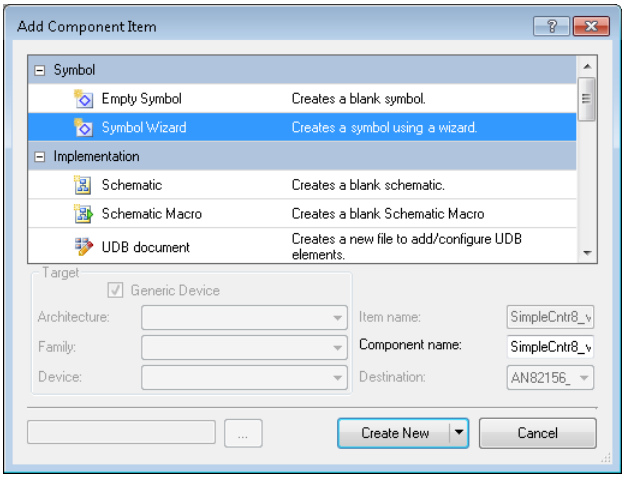
4. Workspace Explorer のタブ **Components** に切り替え、**プロジェクト AN82156_Appendix_Lib** を右クリックします。図 81 に示すようにドロップダウン メニューから **Add Component Item…**を選択します。

図 81. コンポーネントアイテムの追加



5. コンポーネント テンプレートの **Symbol Wizard** を選択して、コンポーネントを *SimpleCntr8_v1_0* の名を付けます。図 82 を参照してください。

図 82. Symbol Wizard の使用



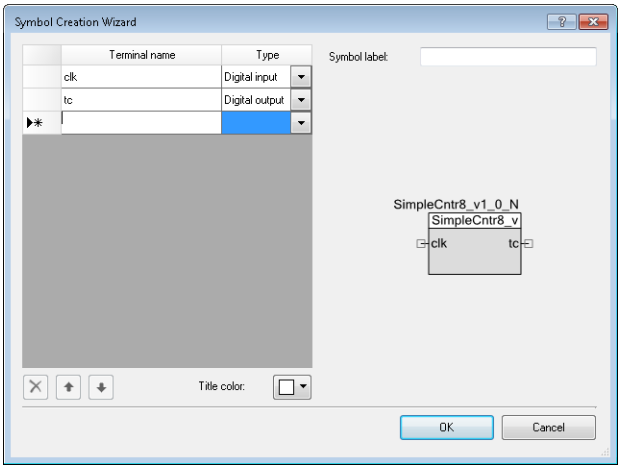
コンポーネントの名前に版数を含んだ方が良いです。コンポーネントの名前に「_vX_Y」タグ (X がメジャーバージョン、Y がマイナーバージョン) を追加します。PSoC Creator はバージョニング機能を持ち、コンポーネントの複数のバージョンを追跡して使用できます。

6. **Create New** ボタンをクリックして、コンポーネントシンボルウィザードを起動します。

   ウィザードは、ユーザーに入力と出力を定義するように求め、これらの情報を使って、コンポーネントシンボルを作成します。

7. 2 個の端末 (*clk* 入力と *tc* 出力)を **Terminal Name** フィールドに追加します。図 83 を参照してください。

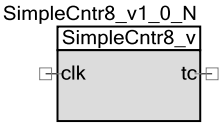
図 83. 入力と出力の追加



「clk」入力はデータパスクロックです。「tc」出力はカウント値が 0 になった時点を通知するのに使用します。

8. **OK** をクリックして、シンボルをシンボルエディターページで生成します。図 84 を参照してください。

図 84. 生成したコンポーネントシンボル

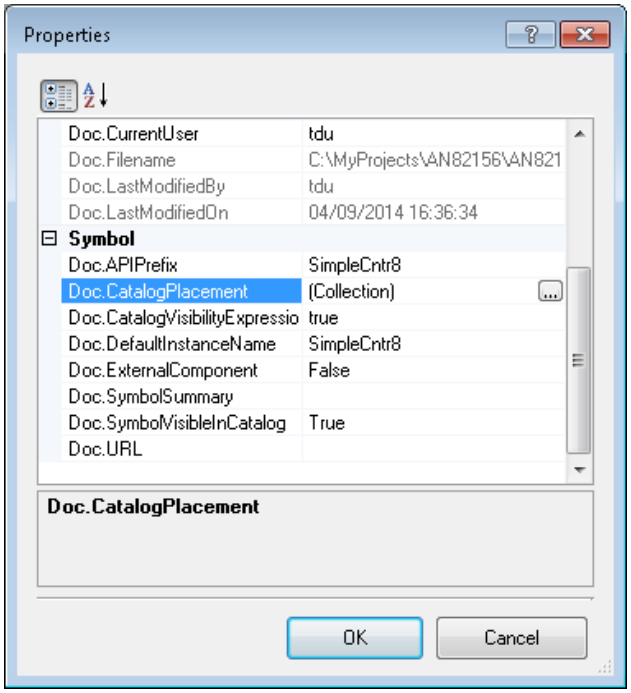


この時点では、「clk」と「tc」端末は回路図の一部だけで、何もしません。後で Verilog を使って、それらの機能を定義します。

9. (シンボル自身ではなく) シンボルエディターの空の領域を右クリックして、ドロップダウンメニューから Properties を選択します。
10. プロパティフィールドの Symbol セクションに値を入力します。図 85 を参照してください。

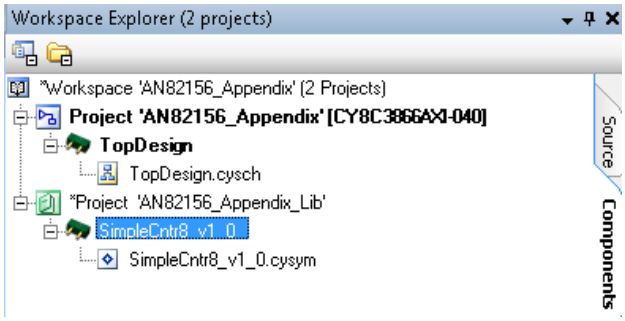
- **Doc.APIPrefix** = *SimpleCntr8*
  この値は、コンポーネント用に生成された任意の API のファイル名の先頭に付加されます。この例では、API を生成しませんが、コンポーネントを作成するたびに、ここに値を入力してください。
- **Doc.CatalogPlacement** = *AN82156_Appendix/Digital/Cntr8*
  「…」をクリックして Catalog Placement ダイアログを開き、値を入力します。PSoC Creator はこの値を使って、コンポーネント カタログの階層を定義します。最初のアイテムは、コンポーネントをカタログに表示するタブです。次の各「/」はサブレベルを表します。階層は少なくとも 1 つのサブレベルを含む必要があります。ここで表す値は、コンポーネントが AN82156_Appendix タブの「Digital」サブレベルで「Cntr8」として表示されることを示します。
- **Doc.DefaultInstanceName** = *SimpleCntr8*
  これは、コンポーネントが回路図に配置する時に表示されるデフォルト名です。コンポーネントをプロジェクト回路図に配置した後に、変更できます。

図 85. Symbol Properties (シンボルプロパティ) の追加



11. 「Save All」 (Ctrl+Shift+S) を行い、変更がすべてのプロジェクトに適用されることを確実にします。新しいシンボルは、Workspace Explorer の **Components** タブの **AN82156_Appendix_Lib** の下に表示されます。図 86 を参照してください。

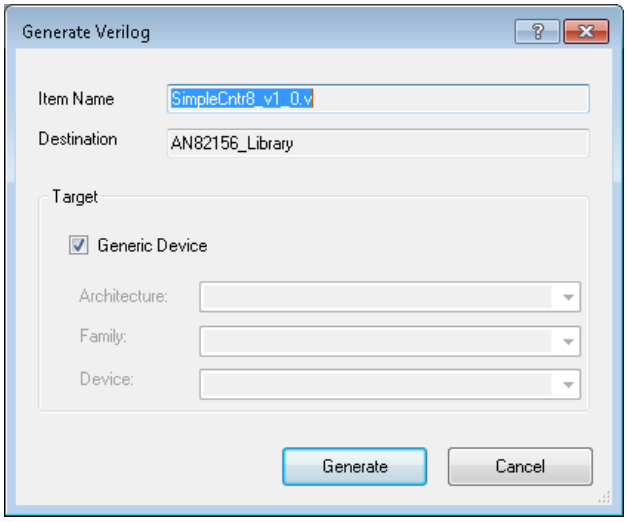
図 86. ライブラリの新しいコンポーネント



次に、回路図シンボルをデータパス実装にリンクする必要があります。

12. Symbol Editor ページの空白領域を右クリックして、ドロップダウン メニューから **Generate Verilog** を選択します。
13. Generate Verilog ダイアログ ボックスのすべての設定を初期値のままにして、**Generate** をクリックします。図 87 を参照してください。

図 87. Generate Verilog ダイアログ



14. 「Save All」を実行して、変更がプロジェクトにすべて適用されることを確実にします。新しい Verilog ファイルが生成され、コンポーネントに追加されます。

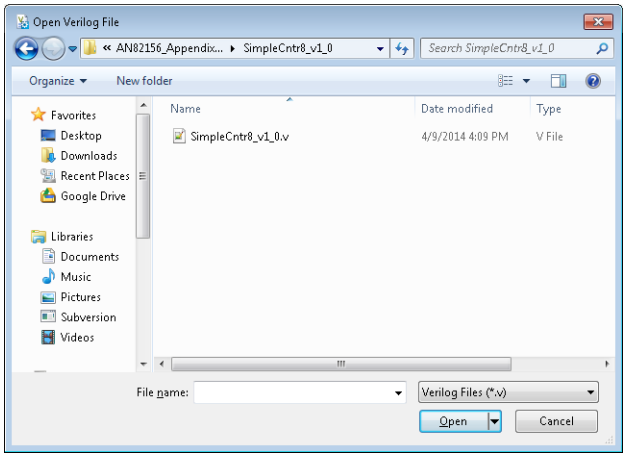
    Verilog ファイルにはデータパス実装と、データパス制御用に必要な Verilog コードが含まれます。付録 C で、新しいコンポーネント Verilog ファイルの例を記述します。データパスのインスタンスを追加および編集するには、*Datapath Configuration Tool* (データパスコンフィギュレーションツール) を使用してください。

15. データパスコンフィギュレーションツールを起動

16. **Tools** > **Datapath Config Tool**....を選択して、ツールを起動することができます。Start メニュー (**Start** > **All Programs** > **Cypress** > **PSoC Creator 3.x** > **Component Development Kit** > **DatapathConfiguration Tool**) からデータパス コンフィギュレーション ツールを起動することも可能です。

17. データパスコンフィギュレーションツールで、**File**> **Open** を選択して、PSoC Creator により生成される *SimpleCntr8_v1_0.v* ファイルを参照します。この例での手順に従えば、そのファイルは、図 88 に示すように *AN82156_Appendix* ワークスペースフォルダー内の *AN82156_Appendix_Lib* プロジェクト フォルダーに配置されます。

図 88. コンポーネント Verilog ファイルの例



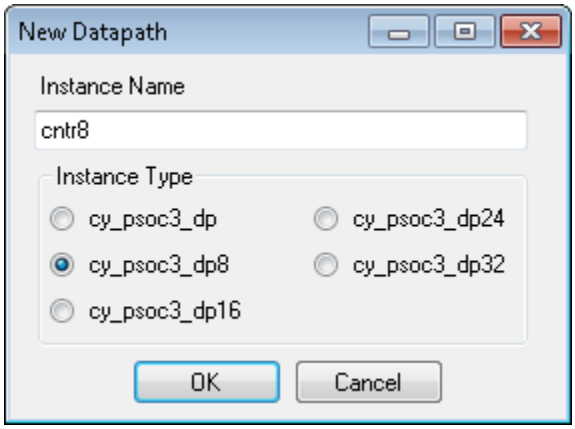
18. ファイルを選択して **Open** をクリックし、Verilog ファイルをデータパスツールに読み込みます。

    まず、新しいデータパスコンフィギュレーションを作成します。

19. メニューから **Edit** > **New Datapath** を選択して、New Datapath ダイアログウィンドウを開きます。

20. 図 89 示すように、**Instance Name** にインスタンス名 (「*cntr8*」を使用) を入力し、Instance Type には **cy_psoc3_dp8** を選択します。**OK** をクリックします。

図 89. 新しい 8 ビットのデータパスインスタンス



これは 8 ビットのカウンターです。データパスを 8 ビット (cy_psoc3_dp8) より大きく定義する必要はありません。他のサンプル プロジェクトは 8 ビットより大きいコンポーネントをデモします。

注: cy_psoc3 は総称です。これらのデータパスインスタンスはどのデバイスでも機能します。

21. **File** > **Save** をクリックして、コンフィギュレーションを Verilog ファイルに保存します。

データパスコンフィギュレーションツールは、最初にコンフィギュレーションを保存する時に、Verilog ファイル内でデータパス構造をインスタンス化します (付録 C – 強制データパス配置を参照)。コンフィギュレーションを変更してカウンター機能を実装することができます。

22. Reg0 と Reg1 フィールドの値を選択して、データパスを設定します。図 90 を参照してください。他のフィールドはそれらのデ フォルト設定のままにして良いです。

23. Reg0 はコンフィギュレーション 0 または命令 0 と同様で、Reg1 はコンフィギュレーション 1 または命令 1 と同様です。

図 90. SimpleCntr8 コンポーネントのコンフィギュレーション

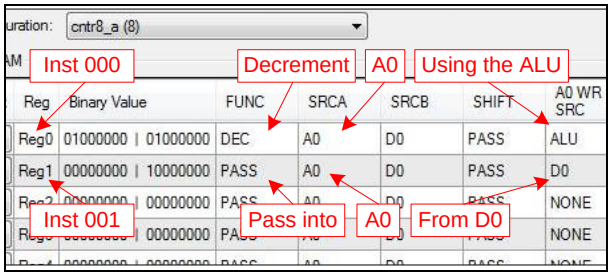


表 8. 例 1 のデータパスコンフィギュレーション

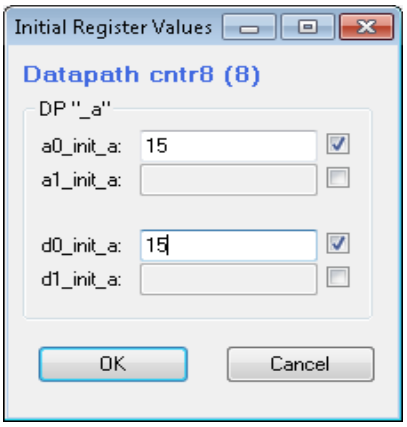
FUNC	SRCA	SRCB	SHIFT	A0 WR SRC
DEC	A0	D0	PASS	ALU
PASS	A0	D0	PASS	D0

24. Reg0 と Reg1 列は CFGRAM に保存される 8 つのデータパス コンフィギュレーションの最初の 2 つを表しています。1 つはカウンターをデクリメントするように、もう 1 つはカウンターを再ロードするようにコンフィギュレーションされます。

A0 と D0 レジスタにロードされるカウント値で開始する必要があります。Ax と Dx レジスタは CPU でアクセス可能であるため、*main.c* ファイル内のコードを使用してロードすることができます。また、Verilog ファイル内で定義することで、それらにデフォルトの初期値をロードすることもできます。データパスツールはこれらの設定を行うことができます。

25. **View** > **Initial Register Values** を選択して、レジスタ値のダイアログを開きます。
26. **a0_init_a** と **d0_init_a** のボックスにチェックを入れます。図 91 に示すように、値を *15* にセットして、開始カウントとリロード カウントを定義します。

図 91. SimpleCntr8 の Initial Register Values



27. 完了したら、**OK** をクリックします。

    この場合、開始値と同じカウント値を A0 に再ロードするように D0 と A0 値は同一にします。最初の期間が他のと異なるようにするために、別の値を選択することが可能です。この場合、カウント シーケンスが 15～0 のため、期間は 16 です。

28. メニューから **File** > **Save** を選択して、変更を Verilog ファイルに保存します。データパスコンフィギュレーションツールを閉じて、PSoC Creator の *Counter_8bit_v1_0.v* ファイルを開いて確認します。

    設定を保存すると、データパスコンフィギュレーションツールは Verilog ファイルを変更します。PSoC Creator は、戻って切り替える時、Verilog ファイルをリロードように求めることがあります。

    この時点で、Verilog ファイルに変更を行って、回路図シンボルをデータパス論理にリンクする必要があります。ここで討論するコード部分は Verilog ファイルの 77 行目からの部分です。

29. `.clk(1'b0)` を `.clk(clk)` に変更します。これによって、データパスクロックがシンボルの clk 端末にリンクされます。この端末を使用して、データパスが動作する速度をセットします。
30. `.z0()`, から `.z0(tc)` に変更します。これによって、ゼロ検索ブロック (ZDET) の z0 出力がシンボルの tc 端末にリンクされます。リンクされると、tc 端末は z0 出力の値を表します。
31. `.cs_addr(3'b0)` から `.cs_addr({2'b00,tc})` に変更します。これによって、CFGRAM アドレスの上位 2 ビットが 0 にセットされ、ビット 0 が `tc` の値にセットされます。

`tc` が常に `z0` の値を反映することに注意してください。つまり、`z0` はコンフィギュレーションを決定します。表 7 を参照してください。

32. すべての変更を Verilog ファイルに保存します。付録 C – 強制データパス配置 では、すべてが正しく入力されれば、完成した Verilog コードがどのように見えるかを示しています。

これで、コンポーネントが使用可能になりました。コンポーネントがプロジェクトのコンポーネントカタログに表示されるように、*AN82156_Library* をプロジェクトの依存関係としてセットします。

33. **Source** タブで、**プロジェクト「AN82156_Appendix」**を右クリックして **Dependencies...**を選択します。AN82156 プロジェクトのユーザー依存関係として *AN82156_Appendix_Lib* を追加します。図 92 が示すように、Components ボックスをチェックします。 これを行うには、フォルダアイコンをクリックして、作成したライブラリの.cylib ファイルに移動します。

図 92. MyLibrary をプロジェクトの依存関係として追加

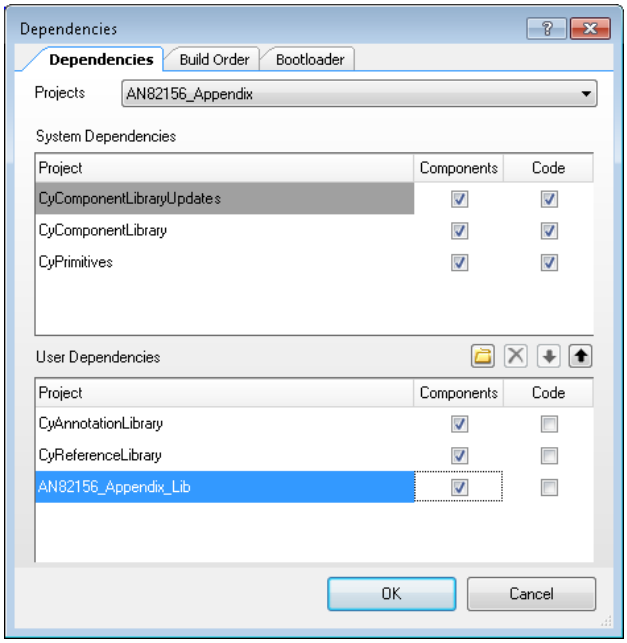


図 93 に示した通り、新しい *AN82156_Appendix_Lib* コンポーネントは *AN82156_Appendix* タブ下の Component Catalog に表示されます。

図 93. カタログ内の新しいコンポーネント

コンポーネントが Component Catalog に表示された後、それを回路図に配置して他のコンポーネントと同じように使用することができます。

それをテストするには、クロック ソースとカウンターの tc 出力を表示する方法を追加する必要があります。

34. Cntr8 コンポーネントをプロジェクト回路図に配置します。
35. クロック コンポーネントを「clk」端末に接続します。それを 10kHz にセットします。他の値でも良いですが、10kHz の場合にはオシロスコープ表示が容易になります。
36. オシロスコープで観察できるように、デジタル出力ピンを「clk」端末に接続します。それに「P0_0_clk」と名前を付け、他のすべての設定をデフォルト値のままにします。

    注: PSoC 4 では、クロック出力をピンへ直接ルーティングすることはできません。クロックをピンへルーティングするために、以下の手順に従ってください。

    a. デジタル出力ピンコンポーネントを回路図に配置します。
    b. ピン カスタマイザーで、**Clocking** タブを選択します。
    c. **Out Clock:** を *External* に設定します。

d. **Pins** タブに戻って、**Output** サブタブに移動します。

e. **Output Mode:**で、*Clock* を選択します。

f. 「OK」をクリックします。

g. クロック信号をピンコンポーネントの out_clk 端末に接続します。

注: For PSoC 6 MCU では、クロック出力をピンへ直接ルーティングすることはできません。クロックをピンへルーティングするために、以下の手順に従ってください。

a. デジタル出力ピンコンポーネントを回路図に配置します。

b. TFF コンポーネントを回路図に配置します。

c. クロックコンポーネントの出力を TFF コンポーネントの clk 端子に接続します。

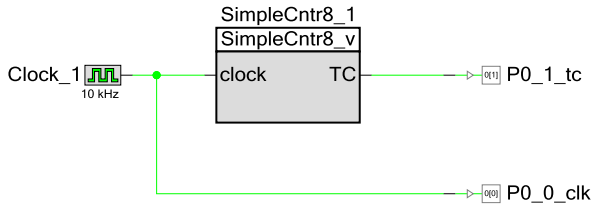
d. Logic High の「1」コンポーネントを TFF コンポーネントの t 入力に接続します。

e. TFF コンポーネントの q 出力を手順 a で配置したデジタル出力ピンに接続します。

注:デバイス外部で見られるクロック周波数は、UDB コンポーネントによって使用される実際の値の半分になります。

37. デジタル出力ピン コンポーネントを「tc」端末に接続します。それに P0_1_tc と名前を付け、他のすべての設定をデフォルト値のままにします。カウントが 0 になると、このピンは HIGH になります。図 94 に完全なプロジェクトの回路図を示します。

図 94. 簡単なカウンタープロジェクト回路図



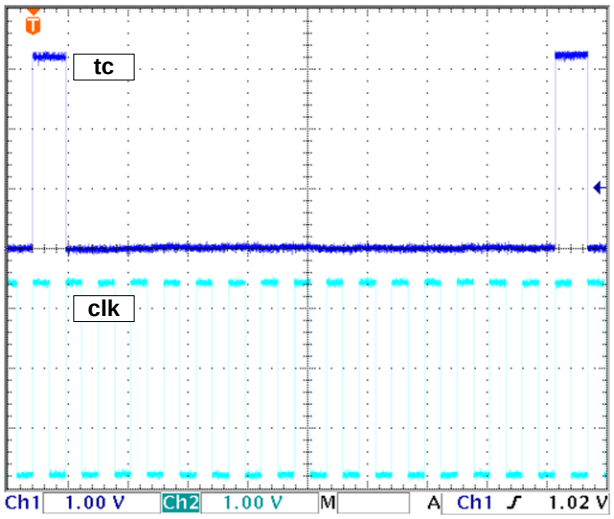
38. cydwr の *Pins* タブで、ピンの名前に応じてピンを P0[0]と P0[1]に割り当てます。

ここで、プロジェクトのビルドと PSoC のプログラミングの準備ができました。P0[0]と P0[1]上のクロックと端末カウントを観測することができます。

39. プロジェクトを保存してビルドし、PSoC をプログラムします。

図 95 に示すように、オシロスコープを出力ピンに接続すれば、「clock」と「tc」出力を観察できます。

図 95. 簡単なカウンターの出力



A0 に開始値 15 をロードしたため、周期が 16 (15 から 0 までカウントダウン) クロックサイクル幅です。tc ピンは 1 クロックサイクルだけ (A0 が 0 になる時) HIGH になります。それは、その時、A0 が D0 の値でリロードされるためです。A0 が 0 でなくなると、tc が LOW にセットされ、設定は後方に遷移して再び A0 のデクリメントを開始します。

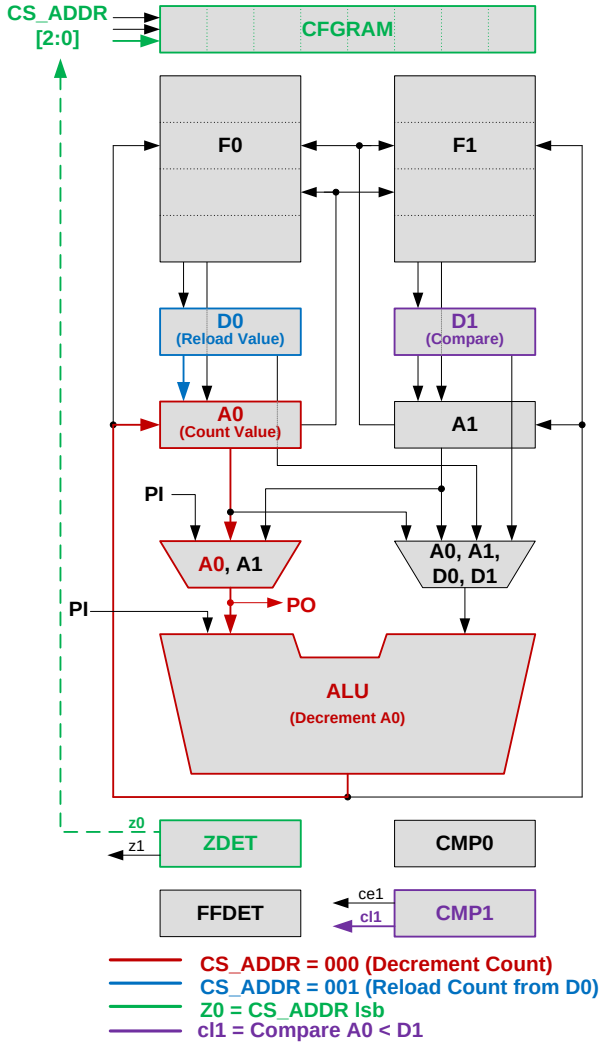
これで、ファームウェアを書くことが全く必要なしに初めてのデータパスベースのコンポーネントの設計を完了しました。

A.1.3 PWM になるようにカウンターを変更

PWM は単に比較機能のあるカウンターです。PWM を追加するには、A0 の値を他の固定の値と比較する方法が必要です。比較対象の固定値を D1 レジスタに格納し、A0 が D1 未満であるかどうかを確認するように比較ブロックを設定できます。

図 96 に示すように、強調表示されたブロック図を用いてそれを視覚化することができます。

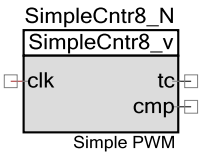
図 96. 簡単な PWM の強調表示されたブロック図



この例は、前の節で作成した 8 ビットカウンターコンポーネントを変更します。空のコンポーネントで、または以前のカウンターコンポーネントのコピーを作成することで開始することができますが、以下の手順は既存のカウンターを変更することを前提とします。

1. SimpleCntr8_v1_0 用のコンポーネントシンボル(.cysym) ファイルを開きます。
2. 「o」を押して、デジタル出力端末を追加します。それに cmp を名前付け、図 97 に示した通りに配置します。

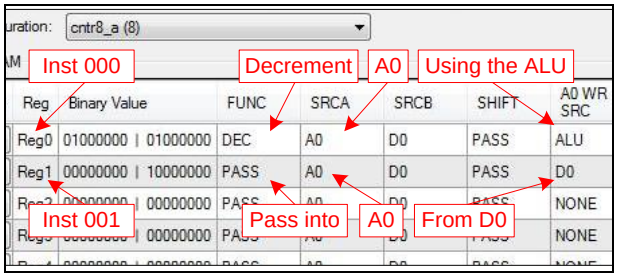
図 97. cmp 端末付きのコンポーネントシンボル



3. 「Save All」を選択して、すべての変更を保存します。
4. データパスコンフィギュレーションツールを起動して、*SimpleCntr8_v1_0.v* の Verilog ファイルを開きます。データパスコンフィギュレーションツールは自動的に Verilog ファイルを解析して、最初のコンフィギュレーションを表示します。

前述したように、PWM は単に比較機能のあるカウンターです。ここでも必要なコンフィギュレーションは 2 つのみで、A0 でのカウント値はデクリメントされて、リロードされます。つまり、図 98 に示すようにデータパスコンフィギュレーションツールでのほとんどの設定は変更されません。

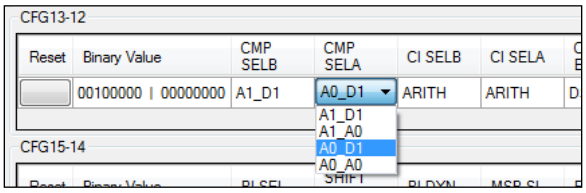
図 98. PWM 用の CFGRAM の設定



(A0 < D1) の比較を追加する必要があります。それを行うには、コンフィギュレーション可能な比較ブロックを設定します。

5. 図 99 に示すように、**CMP SELA** を *A0_D1* にセットして、比較用に A0 と D1 を使用するように比較機能を設定します。

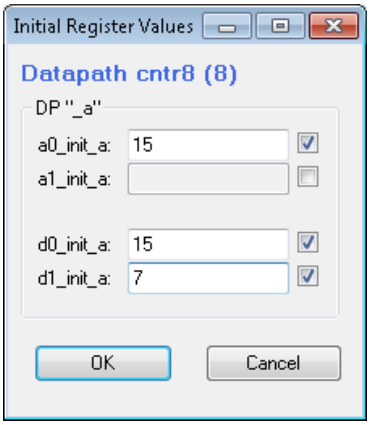
図 99. 比較ブロック 1 のマルチプレクサの設定



A0 と D0 がカウント値およびリロード値を格納することに注意してください。D1 レジスタにデフォルトの比較値を追加することが必要です。

6. メニューから **View** > **Initial Register Values** を選択して、ダイアログ ウィンドウを開きます。
7. **d1_init_a** のチェックボックスをチェックします。
8. a0 と d0 の値に 15 を入力します。これによって、期間が 16 クロックサイクルにセットされます。
9. d1 値に 7 を入力します。図 100 に示すように、これによって、比較値は 7 にセットされます。

図 100. PWM のレジスタの初期値



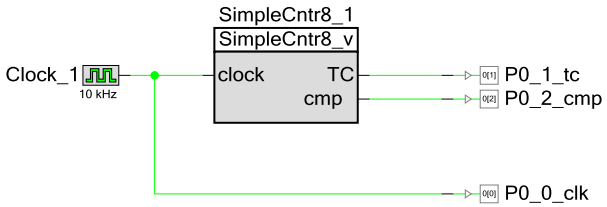
10. **OK** をクリックして、レジスタにそれらの値を適用します。
11. Verilog ファイルに変更を保存します。

A0 が D1 未満の間、比較ブロックの出力は常に HIGH です。カウンターと同じように、Verilog ファイルを使ってシンボルをデータパスハードウェアにリンクします。

12. データパスコンフィギュレーションツールを閉じて、PSoC Creator 内の Verilog ファイルを開きます。
13. `output tc` の下に、`output cmp` を追加します。これにより、コンポーネントシンボルでの cmp 端末へのリンクが作成されます。
14. `.cl1()`行を`.cl1(cmp)`に変更します。これにより、cmp 信号が Compare1 ブロックの「より小さい」出力にリンクされます。
15. すべての変更を Verilog ファイルに保存します。

    PWM コンポーネントとシンボルは Component Catalog の AN82156_Appendix タブに表示されているままです。コンポーネントはプロジェクト回路図で自動的に更新されます。
16. 出力ピンを追加し、cmp 端末に接続させます。図 101 に示すように、それに *P0_2_cmp* の名を付け、P0[2]ピンに割り当てます。

図 101. 簡単な PWM プロジェクト回路図

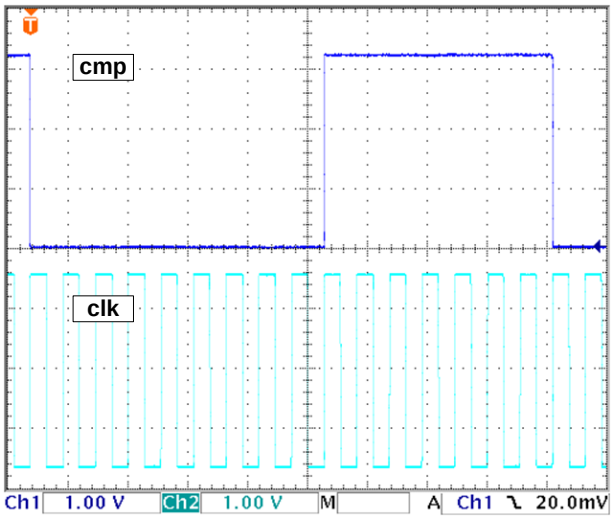


ここで、プロジェクトのビルドと PSoC のプログラミングの準備が整いました。クロックと端末カウントは P0[0]と P0[1]ピンで観察することができます。PWM 出力は P0[2]で観察されます。

11. プロジェクトを保存してビルドし、PSoC をプログラムします。

オシロスコープを出力ピンに接続すれば、clock、tc 、および cmp 出力を観察することができます。図 102 に「clk」と「cmp」信号を示します。

図 102. 簡単な PWM の出力



A0 に開始値 15 をロードしたため、周期が 16 クロック サイクル幅となります。D1 を 7 に設定したため、A0 が 7 より小さくなると「cmp」ピンが HIGH になります。D1 の比較値を変更して PWM をテストすることができます。

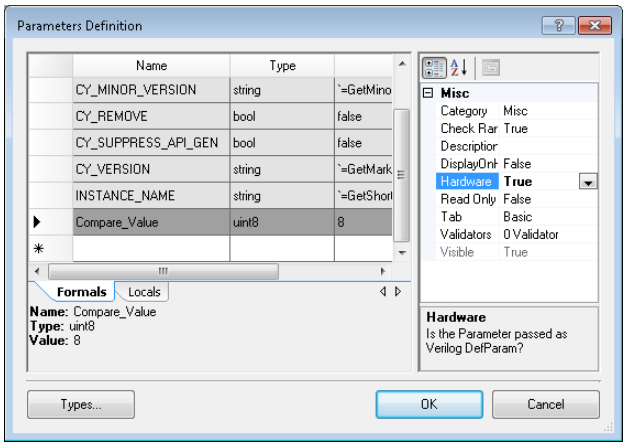
A.1.4 パラメーターの追加

コンポーネントのパラメーターのいずれかを変更するたびに Verilog コードを変更することは不便です。また、もし異なる期間および比較値を持つ 2 つの PWM を必要とする場合、どうなるでしょうか。ほとんどのサイプレスのコンポーネントと同様に、ユーザー コンフィギュレーション可能なパラメーターをコンポーネントに追加することができます。

1. コンポーネントのシンボルエディターページ (.cysym)を開き、空白領域を右クリックします。
2. ドロップダウン メニューから **Symbol Parameters** を選択します。
3. 既存のパラメーターの下の空の列に新しいパラメーターを追加します。
   - 名前 = Compare_Value
   - タイプ = uint8
   - 値 = 8
4. 図 103 に示すように、ウィンドウの右側の **Misc** 設定フィールドで、**Hardware** フラグを「*True*」に設定します。

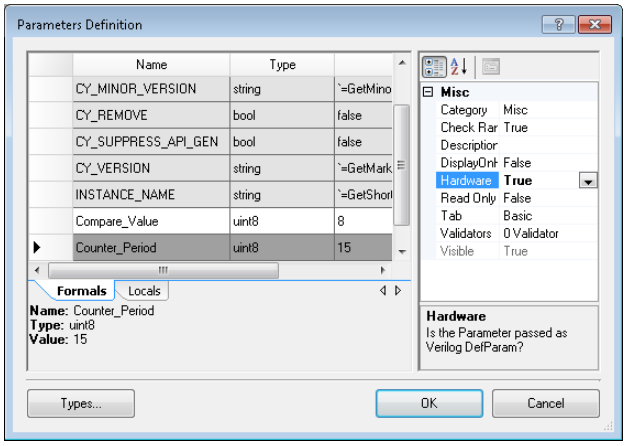
   これは、UDB ハードウェアがパラメーターを使用することができるようにパラメーターを Verilog に表示します。

図 103. 新しいコンポーネントパラメーターの追加



5. 既に作成した比較値の定義の下の列に他の新しいパラメーターを追加します。
   - 名前 = Counter_Period
   - タイプ = uint8
   - 値 = 15
6. 図 104 に示すように、ウィンドウの右側の **Misc** 設定フィールドで、**Hardware** フラグを「*True*」に設定します。

図 104. 他の新しいコンポーネントパラメーターの追加



7. **OK** をクリックし、「Save All」を選択して、コンポーネントに変更を適用します。

   新しいコンポーネントシンボルパラメーターを Verilog ロジックにリンクする必要があります。

8. コンポーネントの Verilog ファイルを開いて、25 行目またはその近くの、以下テキストを探します。

```
//      Your code goes here
```

9. そしてそのテキストを以下に置き換えます。

```
parameter [7:0]  Counter_Period = 8'd0;
parameter [7:0]  Compare_Value  = 8'd0;
```

10. レジスタの初期値を含む 29 行目またはその近くのテキストを探します。

```
cy_psoc3_dp8 #(.a0_init_a(15), .d0_init_a(15), .d1_init_a(7),
```

11. 固定値を先ほど定義したパラメーターで置き換えます。

```
cy_psoc3_dp8
#(.a0_init_a(Counter_Period), .d0_init_a(Counter_Period), .d1_init_a(Compare_Value),
```

   このコードは A0、D0、および D1 の初期値をコンポーネント パラメーターにリンクさせます。これ以降の例では、データパス コンフィギュレーション ツールを使用してこれらの手順を行う方法を説明します。

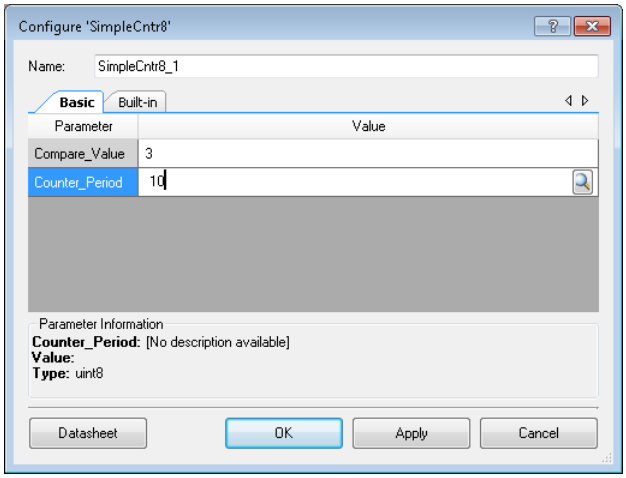
   コンポーネントを変更せず、コンパイル時にそれらを設定することができます。

12. 「Save All」を選択して、プロジェクトをビルドして PSoC をプログラムします。

   デフォルトの比較値は以前に 7 に設定されました。ここで、プロジェクト回路図内でパラメーターを変更することができます。

13. プロジェクト回路図に戻って、SimpleCntr8_1 コンポーネントをダブルクリックしてプロパティダイアログを開きます。
14. 図 105 に示すように、比較値を「3」に、カウンター周期を「10」に変更します。

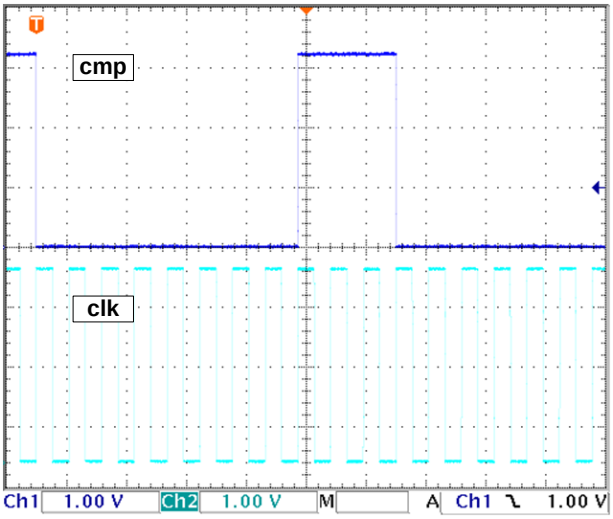
図 105. コンポーネントパラメーターの選択



15. **OK** をクリックして変更を適用します。
16. 「Save All」を選択して、プロジェクトをビルドして PSoC をプログラムします。

    図 106 に示すように、周期と比較出力が変更されます。

図 106. パラメーターの変更された PWM 出力



周期を 11 サイクルに設定し (カウンターがリロードの前に 0～10 に進むため、周期は 10+1)、比較値を 3 に設定しました。結果は、8 クロック サイクルの LOW 出力および 3 クロック サイクルの HIGH 出力となります。

パラメーターの値が uint8 である限り、ほとんど何の値にも変更することができます。また、複数のコンポーネントのインスタンスをプロジェクトに配置し、それらを異なる値に設定することもできます。

コンポーネントパラメーターの追加の詳細 (入力値制限の設定を含む) については、*Component Author Guide* を参照してください。

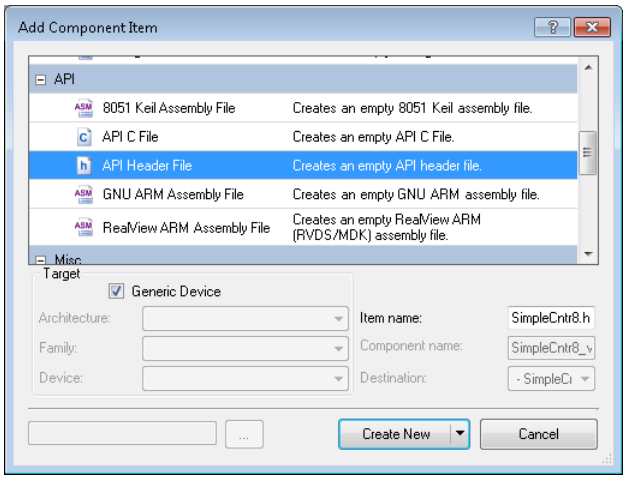
A.1.5 ヘッダーファイルの追加

PWM の動作を設計時に変更するだけでなく、C コードにより PWM のレジスタを変更することで実行時に PWM を変更することもできます。例えば、レジスタ D0 に周期値を格納し、レジスタ D1 に比較値を格納しました。これらのレジスタに容易にアクセスできるように、使用されるレジスタを定義するヘッダー ファイルを作成します。

1. Components タブで、SimpleCntr8_v1_0 を右クリックし、Add Component Item を選択します。

2. Add Component Item ウィンドウで、**API** セクションに移動して **API Header File** をクリックします。
3. 図 107 に示すように **Item name** を *SimpleCntr8.h* に変更します。

図 107. ヘッダーファイルの追加



4. **Create New** をクリックします。これにより、ヘッダー ファイルがコンポーネントに追加されます。

ここで、比較値 (D1) と周期値 (D0)の定義を追加します。

5. ヘッダー ファイルで`//[]END OF FILE` の上に以下の定義を加えます。

```
#define `$INSTANCE_NAME`_Period_Reg (*(reg8 *) `$INSTANCE_NAME`_cntr8_u0__D0_REG )

#define `$INSTANCE_NAME`_Compare_Reg (*(reg8 *) `$INSTANCE_NAME`_cntr8_u0__D1_REG )
```

これらの 2 つの定義により、ファームウェア内で D0 と D1 レジスタに直接書き込めます。*cyfitter.h* ファイルは、プロジェクトに使用されるコンポーネント内にあるレジスタの一連の定義を含みます。このアプリケーションノートに記載されている手順と異なって行った場合は、レジスタ定義を *cyfitter.h* ファイルで検索して、ここに示している定義とは異なるものを使う必要があります。

実行時に比較値を更新するために、作成したばかりの定義に書き込みます。コンポーネントに SimpleCntr8_1 の名前が付けられた場合、C コードは以下の通りです。

```
SimpleCntr8_1_Period_Reg = 0x08;

SimpleCntr8_1_Compare_Reg = 0x02;
```

これにより、周期値が0x08に、比較値が0x02に更新されます。これとその定義の使用方法については、*Component Author Guide* を参照してください。上記の C コードの場合のようにレジスタへ直接書き込む方法は 8 ビットのレジスタのみに対して実行できることに注意してください。16 ビット以上のレジスタに対しては、次のプロジェクトで説明する他の方法を使用しなければなりません。

A.2 プロジェクト#2 – 16 ビット PWM

このサンプル プロジェクトは、データパス チェーン接続の概念を説明します。データパスは隣接のデータパスに接続される専用信号を持っています。これらの信号により、最大 32 ビット幅の機能を作成することができます。この例では図 108 に示すように、最初のサンプル プロジェクトと同様な PWM を作成しますが、16 ビット幅です。

図 108. チェーン接続された UDB を使用する 16 ビット機能

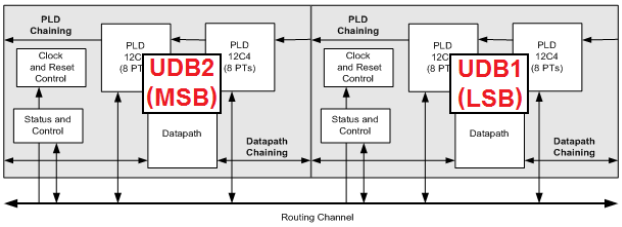
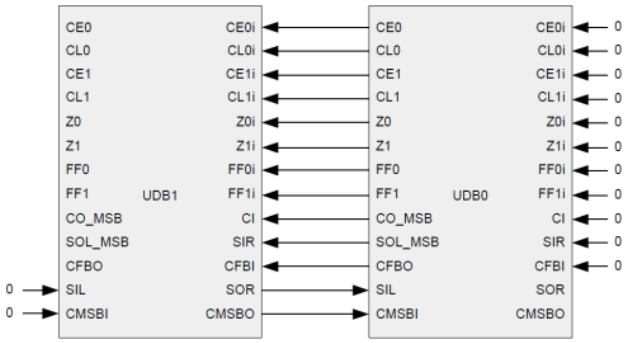


図 109 に示すように、各データパス内の ALU はキャリー、シフト データ、および条件付き信号を最も近い ALU にチェーン接続するように設計されています。すべての条件付き信号とキャプチャ信号は、最下位バイトから最上位バイトへの方向でチェーン接続されます。左シフト信号も最下位バイトから最上位バイトへの方向でチェーン接続されます。右シフト信号は最上位バイトから最下位バイトへの方向でチェーン接続されます。

図 109. データパスのチェーン接続フロー



この例は、ユーザーが前述の例に説明された概念に精通していることを前提にします。参考のために、完成された 16 ビット PWM プロジェクトが、このアプリケーションノートに含まれています。

A.2.1 16 ビット PWM コンポーネント詳細

16 ビット PWM の基本的な機能は最初のサンプルプロジェクトの 8 ビット PWM の基本的な機能と同じです。どの場合でも、カウントがデクリメントされ、カウントが比較値より少ない時に出力が HIGH になります。異なる点は、すべての 16 ビットを操作するために 2 つのデータパスを使用する必要があることです。

A.2.2 16 ビット PWM コンポーネントの作成手順

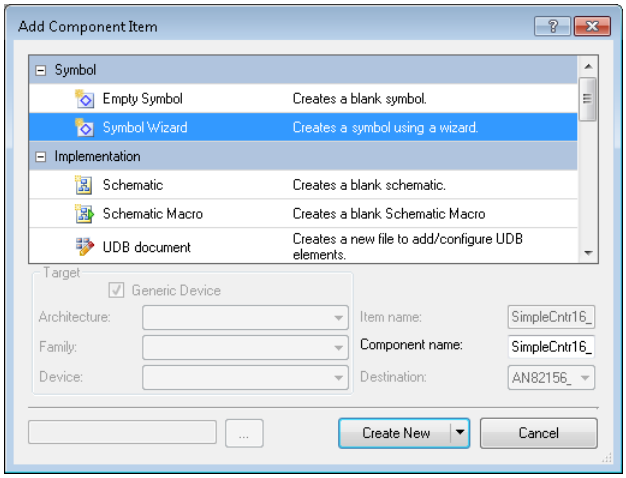
混乱を避けるために、最初のサンプルプロジェクトの 1 つのコンポーネントを修正する代わりに新しいコンポーネントを作成します。コンポーネント作成の基本手順は同じです。

1. PSoC Creator を起動し、前回の例で使用された「**AN82156_Appendix**」ワークスペースを開きます。「*16bitCounter*」という新しいプロジェクトをワークスペースに追加します。

   新しいワークスペースを起動することが可能ですが、この例では、ユーザーが以前と同じようなワークスペースを使用していることを前提にします。同じワークスペースを使用すると、ライブラリ管理と依存関係管理が簡単になります。

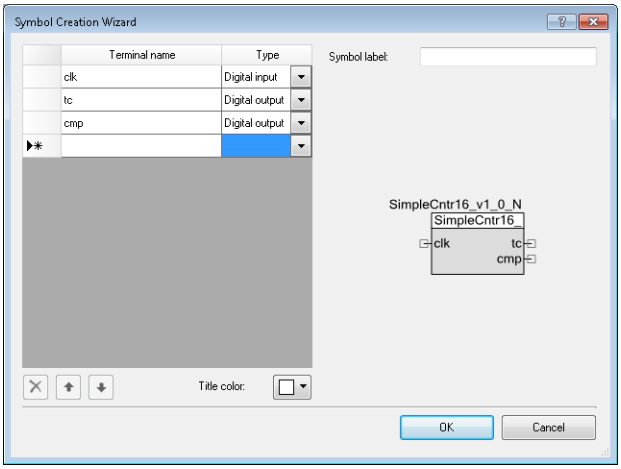
2. 図 110 に示すように、Symbol Wizard を使用して新しいコンポーネントを **AN82156_Appendix_Lib** に追加します。この例ではコンポーネントに「*SimpleCntr16_v1_0*」と名前を付けます。

図 110. 新しいシンボル作成



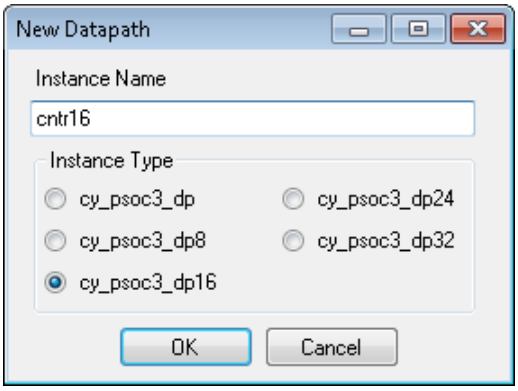
3. 図 111 に示すように、8 ビット PWM の例の通りに *clk* 入力と *tc* と *cmp* 出力を追加します。

図 111. PWM への端末追加



4. シンボル回路図のあるページを右クリックし、シンボルのプロパティを追加します。
   - Doc.APIPrefix = *SimpleCntr16*
   - Doc.CatalogPlacement = *AN82156_Appendix/Digital/Cntr16*
   - Doc.DefaultInstanceName = *SimpleCntr16*
5. 新しいコンポーネントシンボルの Verilog ファイルを生成します。すべてのアイテムをデフォルト値のままにします。
6. 「Save All」を選択して、すべての変更を適用します。
7. *データパス コンフィギュレーション ツール*を起動して、作成したばかりの Verilog ファイルを開きます。
8. 新しいデータパスコンフィギュレーションを追加します。この例では、コンフィギュレーションに「*cntr16*」と名前を付けます。図 112 に示すように、「cy_psoc3_dp16」を選択します。

図 112. 新しい PWM データパスの作成



9. **OK** をクリックしてデータパスのコンフィギュレーションを作成します。

   *cntr16_a(16)*と *cntr16_b(16)*のコンフィギュレーションがあります。2 つの個別のデータパスのコンフィギュレーションが Verilog ファイルに加えられます。「a」コンフィギュレーションは LSB に、「b」コンフィギュレーションは MSB に加えられます。

10. 図 113 と図 114 に示すように、「_a」コンフィギュレーション CFGRAM と CFG13-12 セクションを 8 ビット PWM の例の通りにセットします。

図 113. SimpleCntr16 コンポーネントのコンフィギュレーション

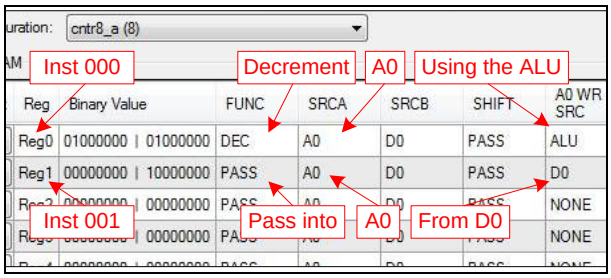
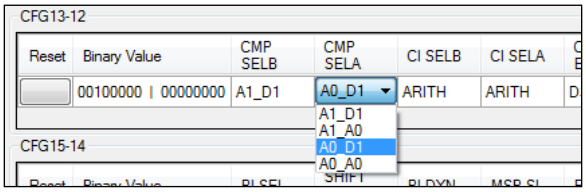


表 9. 例 2 のデータパスコンフィギュレーション

REG	FUNC	SRCA	SRCB	SHIFT	A0 WR SRC
000	DEC	A0	D0	PASS	ALU
001	PASS	A0	D0	PASS	D0

図 114. 比較ブロック 1 のマルチプレクサの設定



8 ビットの例のように ALU 関数と比較ブロックをセットすることを忘れないでください。この時、チェーン接続をまだコンフィギュレーションしていないため、設定は同様です。

11. 「Save」をクリックして Verilog ファイルの変更を保存します。

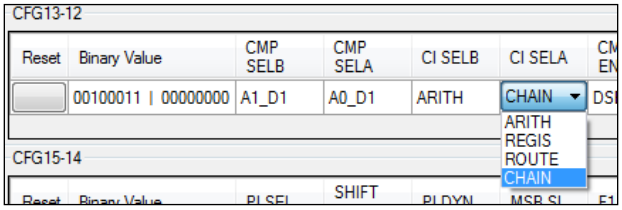
    「_b」コンフィギュレーションが PWM の上位 8 ビットであるため、同様の変更を行ってください。データパスコンフィギュレーションツールは、コンフィギュレーション間で設定をコピーすることができます。

12. **Edit** > **Copy Datapath** を選択してコンフィギュレーションをコピーします。
13. 「cntr16_b」コンフィギュレーションに切り替え、**Edit** > **Paste Datapath** を選択して、「_a」 コンフィギュレーションを「_b」コンフィギュレーションに貼り付けます。
14. 「Save」をクリックして Verilog ファイルの変更を保存します。

    チェーン接続をコンフィギュレーションします。「_b」コンフィギュレーションでのみこれらの設定を使用します。

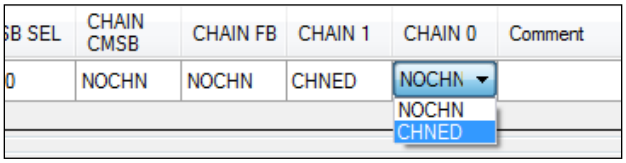
15. 図 115 に示すように、「cntr16_b」コンフィギュレーションで、CI_SELA を「CHAIN」にセットします。これにより、キャリーイン信号が前のデータパスから送信されることを設定できます。

図 115. キャリーイン信号のチェーンのコンフィギュレーション



16. 図 116 に示すように、CHAIN1 と CHAIN0 を「CHNED」にセットします。これらは ce0、ce1、cl0、cl1、z0、z1、ff0、ff1 の比較条件をチェーン接続に設定します。

図 116. データパスのチェーン接続のコンフィギュレーション



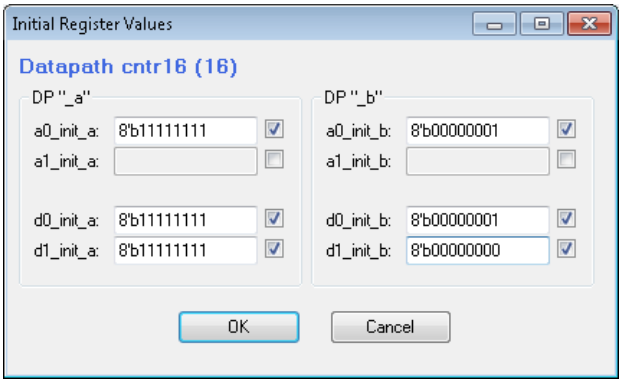
17. 「Save」をクリックして Verilog ファイルの変更を保存します。

    レジスタの初期値をセットする必要があります。以前の例では、周期値と比較値を設定するためのパラメーターの使用方法を示しました。この例はチェーン接続を中心に説明するので、話を簡単にするため、固定値を使います。

18. **View** > **Initial Register Values** を選択します。

19. 図 117 のように DP「_a」と DP「_b」の値をセットします。この例では、各レジスタが 8 ビット幅であることを明らかにするために、数値をバイナリで示します。

図 117. ビット PWM のレジスタの初期値



「_a」コンフィギュレーションは LSB で、「_b」コンフィギュレーションは MSB です。これらの値により、PWM の周期値が 511 クロックサイクルに、比較値が 255 になります。

20. Verilog ファイルの変更を保存して、データパスコンフィギュレーションツールを閉じるために「Save」を選択します。
21. *SimpleCntr16_v1_0.v* Verilog ファイルを開きます。

    Verilog コードに 2 つのデータパスコンフィギュレーションがあることに注意してください。これらの 2 つのデータパスはお互いにチェーン接続され、16 ビットの PWM を形成します。次に、Verilog 論理に使用できるためにいくつかの信号を追加する必要があります。

22. 25 行目またはその近くの「`#start body`」の後に、以下のコードを加えます。

```
// Unused Datapath Connections
wire tc_lsb, cmp_lsb;
```

    ハードウェアリンクは 8 ビット PWM のリンクと同様です。違いは出力が 1 ではなく 2 ビット幅です。Chain0 と Chain1 をデータパス コンフィギュレーションツールでチェーン接続したため、出力の上位ビットが最終結果となります。例えば、z0 出力の上位ビットは、2 つのデータパスの A0 レジスタの値がすべて 0 になる時点を示します。「tc」と「cmp」を上位ビットに割り当て、ステップ 22 で定義した 2 本のワイヤを下位ビットに使用します。これらのワイヤは使用されませんが、追加していないと、シンセサイザが警告を出します。

23. `.clk()` を `.clk(clk)` に変更します。
24. `.z0()` を `.z0({tc, tc_lsb})` に変更します。
25. `.cl1()` を `.cl1({cmp, cmp_lsb})` に変更します。
26. `.cs_addr(3'b0)` を `.cs_addr({2'b0,tc})` に変更します。
27. 「Save All」を選択して、すべての変更を適用します。

    コンポーネント準備が出来上がって、コンポーネントをプロジェクトで使用することが可能です。

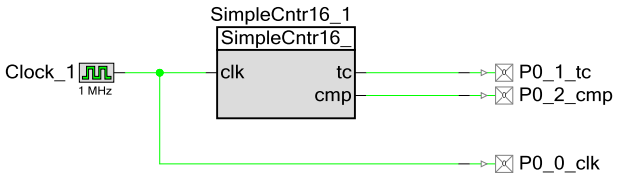
28. *AN82156_Appendix_Lib* を *16bitCounter* プロジェクトの依存関係に設定します。

    新しいコンポーネントはコンポーネントカタログに示されます。コンポーネントの準備が完了し、それをプロジェクトで使用することが可能です。

29. Cntr16 コンポーネントを回路図にドラッグします。
30. クロックを「clk」端末に接続させ、1MHz に設定します。
31. 図 118 に示すように、デジタル出力ピン コンポーネントを「clk」、「tc」、と「cmp」端末に接続します。

注: PSoC 4 および PSoC 6 MCU の場合、このプロセスは異なります。 A.1.2 項の手順 36 を参照してください。

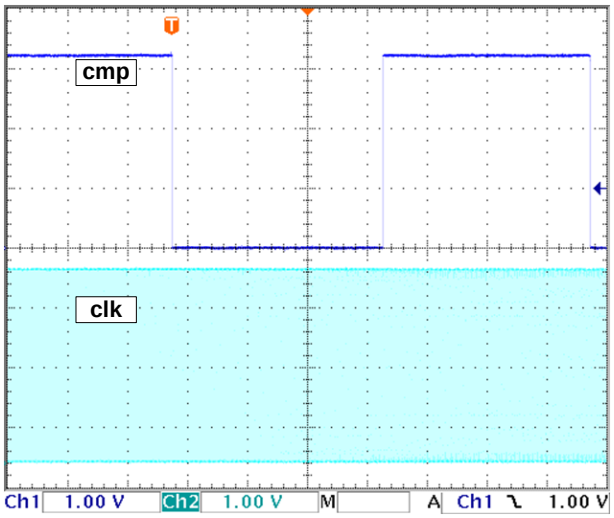
図 118. 簡単な 16 ビット PWM プロジェクト回路図



32. 上図のようにピンに名前を付けて、そしてそれらの名前に基づいて P0[0]、P0[1]、P0[2]に割り当てます。
33. 「Save All」を選択して、プロジェクトをビルドして PSoC をプログラムします。

図 119 に示すように、周期値と比較値が以前に作成した 8 ビット PWM よりはるかに大きいことを出力で確認できます。それらの値はどんな 16 ビットの値にも設定することが可能です。

図 119. 簡単な 16 ビット PWM の出力



チェーン接続を使用して関数を 32 ビット幅に上げることも可能です。この例で説明される原理をより幅広い関数に適用できます。

A.2.3 PSoC 3 における 16 ビットコンポーネントのヘッダーファイル

前述したように、16 ビット レジスタへの書き込みは、8 ビット レジスタへの書き込みと違います。プロセッサとデータパス レジスタの間のエンディアンの相違を気にしなくてもかまわないため、8 ビット レジスタへ直接書き込むことができます。しかし、16 ビット以上のレジスタに書き込む場合は、エンディアンの相違を考慮しなければなりません。

PSoC 3 の 8051 のエンディアン性は、ペリフェラルレジスタのものとは異なります。レジスタへの書き込みを単純化するために、Cypress は CY_SET_REG16 、CY_SET_REG24 、CY_SET_REG32 の幾つかのマクロを提供します。これらのマクロは、最初のパラメーターとしてセットしたいレジスタ アドレス、その後にセットしたい値を受けます。これらのマクロは必要に応じて、エンディアンスワッピングを処理します。そのため、データパスレジスタのアドレスを把握しなければなりません。これらの定義の作成手順は前と同じです。しかし、1 つの違う点があります。それは、定義で `((reg8 *)` を `(*(reg8*)` の代わりに使用します。これにより、データパスレジスタへのポインターが提供されます。_PTR を定義の名前に追加することはいい方法であり、これにより、識別することができます。

値を更新したい場合、CY_SET_REG16 とヘッダー ファイルで定義したポインターを使用してください。これでは、本アプリケーション ノートに同梱されたサンプル プロジェクトを再度ご覧ください。

A.3 プロジェクト#3 – アップ／ダウンカウンター

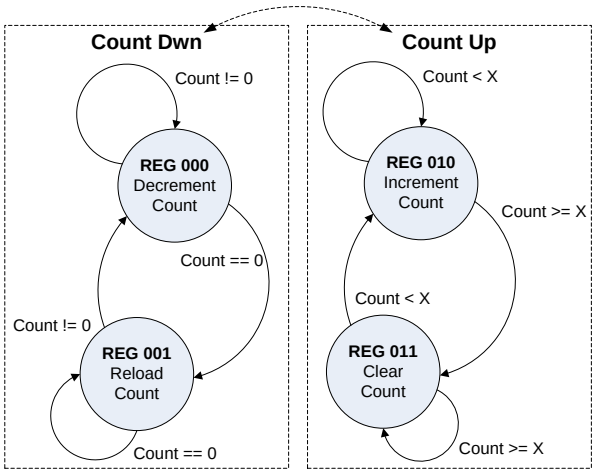
この例は、高度な機能をデータパスコンポーネントに追加する手順を説明します。同様な基本的な PWM を変更して、カウントアップ／カウントダウンの機能を追加します。カウントアップ／カウントダウンの方向は、ユーザーが設計時または実行中に設定したパラメーターに基づいて決定します。

この例は、ユーザーが前述のサンプルプロジェクトで紹介された概念に精通していることを前提とします。本アプリケーションノートには、完成したアップ／ダウン PWM プロジェクトが同梱されています。

A.3.1 追加詳細

簡単なダウンカウント PWM は 2 つの状態を使用しました。図 120 に示すように、アップ／ダウン カウンターを実装するために、4 つの状態が必要です。

図 120. アップ／ダウンカウンターの状態図



データパスは、セットしたパラメーターによって A0 をデクリメントするか、またはインクリメントします。

A.3.2 サンプルプロジェクトのステップ

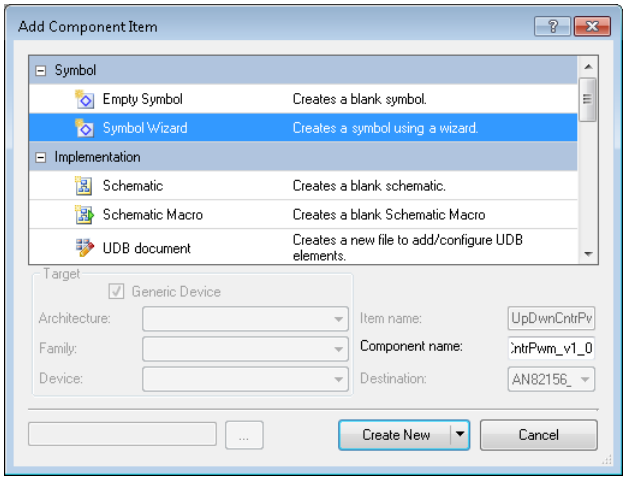
混乱を避けるために、以前のサンプルプロジェクトのコンポーネントを変更する代わりに、新しいコンポーネントを作成します。コンポーネント作成の基本手順は同じです。

1. PSoC Creator を起動し、簡単な 8 ビットの例で使用される「**AN82156_Appendix**」ワークスペースを開きます。「*UpDwnCntrPwm*」という新しいプロジェクトをワークスペースに追加します。

   新しいワークスペースを起動することが可能ですが、この例では、ユーザーが以前と同じようなワークスペースを使用していることを前提にします。これにより、ライブラリ管理と依存関係が簡単になります。

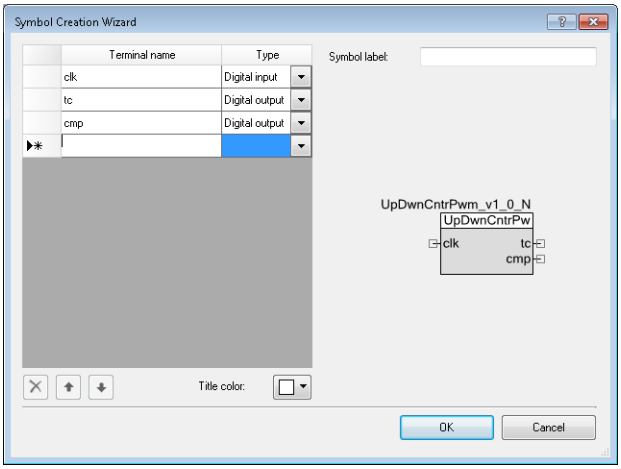
2. 図 121 に示すように、Symbol Wizard を使用して新しいコンポーネントを ***AN82156_Appendix_Lib*** に追加します。この例ではコンポーネントに「*UpDwnCntrPwm_v1_0*」と名前を付けます。

図 121. アップ／ダウンカウンター用の新しいシンボルの作成



3. 図 122 に示すように、8 ビット PWM コンポーネントのように *clk* 入力と *tc* と *cmp* 出力を追加します。

図 122. アップ／ダウンカウンターへの端末追加

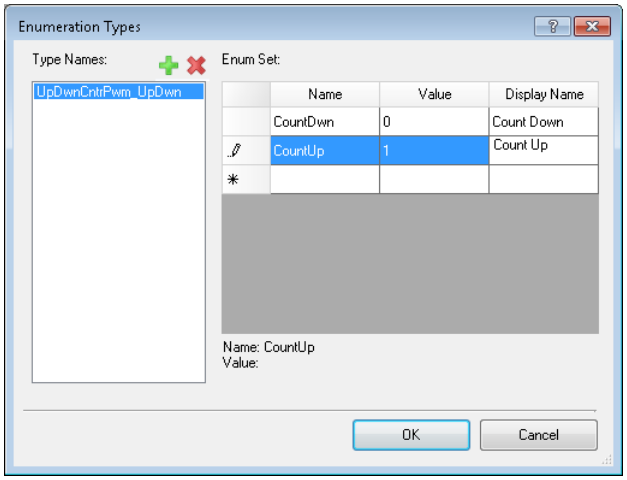


4. シンボル回路図のあるページを右クリックし、シンボルのプロパティを追加します。
   - Doc.APIPrefix = *UpDwnCntrPwm*
   - Doc.CatalogPlacement = *AN82156_Appendix/Digital/UpDwnCntrPwm*
   - Doc.DefaultInstanceName = *UpDwnCntrPwm*

     カウンター期間、比較値、およびカウントアップ／カウントダウンモードなどのユーザーが変更できるパラメーターを追加する必要があります。

     カウントモードの設定のために、新しいパラメータータイプを定義します。
5. シンボルエディターページを右クリックして Symbol Parameters ダイアログを開きます。
6. 「Types」ボタンをクリックして、ウィンドウを開いて新しいパラメータータイプを作成します。
7. 緑色の「+」ボタンをクリックして新しいタイプを追加します。その新しいタイプの名前を *UpDwnCntrPwm_UpDwn* とします。
8. 図 123 に示しているように値を ***Enum Set*** フィールドに入力して、「CountDwn」と「CountUp」を定義します。

図 123. 新しいコンポーネントパラメータータイプの作成



9. **OK** をクリックして Symbol Parameters ダイアログに戻ります。

   この新しいタイプにパラメーターを割り当て、0 (CountDwn) または 1 (CountUp) の初期値をセットすることができます。

10. 前の例と同じようにコンポーネントに 3 つの新しいパラ メーターを追加します。表 10 と図 124 を参照してください。

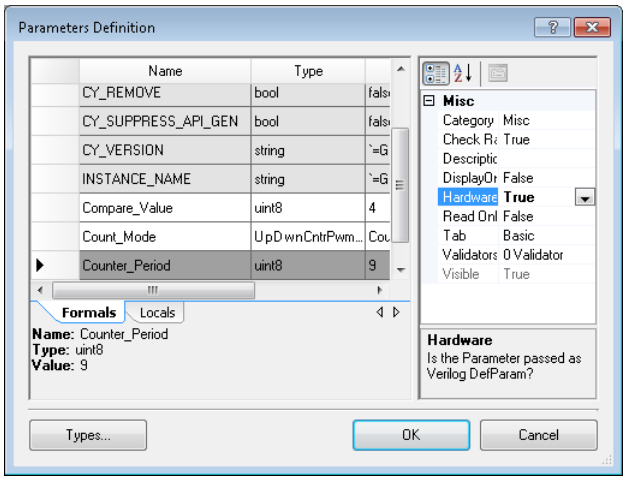
表 10. アップ／ダウンカウンターのパラメーター

Name(名称)	タイプ	値
Compare_Value	uint8	4
Count_Mode	UpDwnCntrPWM_UpDwn	カウントダウン
Counter_Period	uint8	9

Count_Mode パラメーターは新しいタイプと値の定義を使用します。

11. 図 124 に示すように、すべての 3 つの新しいパラメーターに対して **Hardware** フラグを「True」にセットします。

図 124. 新しいコンポーネントパラメーターの追加



12. **OK** をクリックしてコンポーネントへの変更を保存します。
13. シンボルエディター内の何も無い場所を右クリックして、新しいコンポーネントシンボルの Verilog ファイルを生成します (この時、すべての設定はデフォルトの値にします)。

14. 「Save All」を選択して、すべての変更を適用します。
15. データパスコンフィギュレーションツールを起動して、作成した *UpDwnCntrPwm_v1_0.v* ファイルを開きます。

    最初の 2 つのコンフィギュレーションは、簡単な 8 ビット PWM と同様ですが、アップカウント用にもう 2 つの追加コンフィギュレーションが必要です。
16. データパスのコンフィギュレーションを新規作成し「UpDwn」と名前を付けます。「cy_psoc3_dp8」タイプを使用します。
17. 表 11 のように CFGRAM セクションをコンフィギュレーションします。

表 11. 例 3 のデータパスコンフィギュレーション

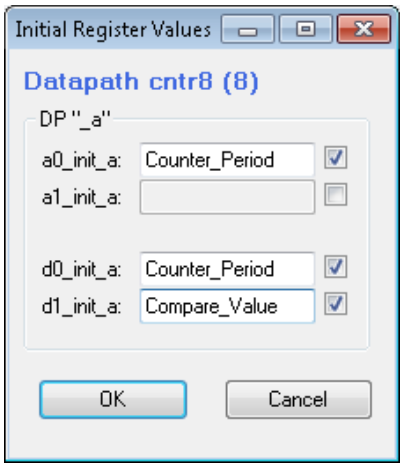
REG	FUNC	SRCA	SRCB	SHIFT	A0 WR SRC
000	DEC	A0	D0	PASS	ALU
001	PASS	A0	D0	PASS	D0
010	INC	A0	D0	PASS	ALU
011	XOR	A0	A0	PASS	ALU

18. XOR コンフィギュレーションはカウントレジスタをクリアするために使用されます。カウントレジスタは周期値に達した後、それ自身の XOR を取ります。すると、カウントが 0 にクリアされます。
19. 簡単な 8 ビット PWM のように、**CMP SELA** フィールドを「A0_D1」にセットして、D1 を比較値として設定します。

    以前、ユーザーが調整できる幾つかのパラメーターを追加しました。Verilog ファイルを手動で修正して変更する必要がないようにそれらの名前を「Initial Register Values」フィールドに入力します。

    8 ビット PWM の場合のように、この例では「Counter_Period」と「Compare_Value」をパラメーターの名前にします。
20. 図 125 に示しているように Initial Register Values ウィンドウを開いて、パラメーター名を A0、D0、および D1 に加えます。

図 125. Initial Register Values のパラメーター使用



21. **OK** をクリックして変更点を保存し Verilog ファイルに適用します。
22. 変更を保存して、データパスコンフィギュレーションツールを閉じます。

    次に、これらの機能を実装するためにいくつかの Verilog コードをコンポーネントに追加する必要があります。
23. *UpDwnCntrPwm_v1_0.v* ファイルを開いて、以下のテキストを見つけます。

```
//  Your code goes here
```

24. そしてそのテキストを以下に置き換えます。

```
// Control Register "pwmCntlReg" bits
localparam CNTL_CNT_UP_DWN = 0;
// Compare0 less than signal
wire up_reload;
//Zero Detect Signal
wire zero_detect;
// Up/down control
wire upDwn;
// Signal to control reload of counter
wire reload;
// Control register signals
wire [07:00] control;



// Up/down control
assign upDwn=control[CNTL_CNT_UP_DWN];
// Logic for the 'reload' signal
assign reload = ( upDwn ) ? ( up_reload ) : ( zero_detect );
//assign termnical count output
assign tc = reload;
       // Control register instance. This text is
       // found in the Component Author Guide.
       cy_psoc3_control #(.cy_init_value (Count_Mode), .cy_force_order(`TRUE))
       pwmCntlReg(
       /* output [07:00] */ .control(control)
       );
```

25. データパス論理に以下のリンクを作ります。

- `.clk()` を `.clk(clk)` に変更します。
- `.cs_addr(3'b0)` を `.cs_addr({1'b0, upDwn, reload})` に変更します。
- `.ce0()` を `.ce0(up_reload)` に変更します。
- `.z0()` を `.zo(tc)` に変更します。
- `.cl1()` を `.cl1(cmp)` に変更します。

ここまで、これらの変更をしましたが、何が行われているか分かるようにこれらの変更点について簡単に説明します。まず、実行時にコードによりカウント方向を変更できる制御レジスタ (cy_psoc3_control)を追加しました。制御レジスタおよび異なったオプションの詳細については、*Component Author Guide* を参照してください。

メインコードから方向を制御したい場合、制御レジスタの定義をヘッダーファイルに加えてください。例として、添付されたプロジェクトを参照してください。

次に、.cs_addr が制御 1 ビットではなく、制御 2 ビットを持つことに注意してください。前述の例では、データパスに 2 状態のみがあるため、制御 1 ビットで充分です。ここでは状態が 4 つあるため、制御 2 ビットが必要です。

さらに、データパスコンフィギュレーションを制御する時、「tc」信号のみを使用する代わりに「reload」信号を使用することに注意をしてください。こうすることが必要な理由は、カウントアップもするので、ZDET を使用するだけでは、周期の終わりを示すことができないためです。ce0 比較を使用する必要があります。そのため、カウンターはアップカウンターとして設定されると、A0 の値が D0 に格納された周期値に等しくない限り、カウントアップし続けます。カウンターが、D0 と等しくなるとレジスタのリロードをトリガーします。

「upDwn」信号はカウントアップの動作 (INC, XOR) かカウントダウンの動作 (DEC, Load) を行うかを制御します。

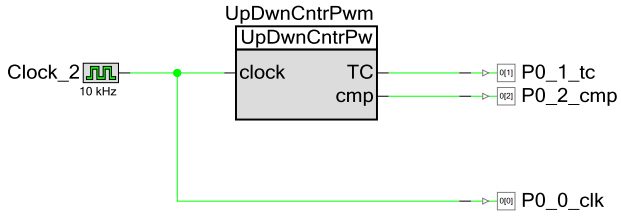
26. すべての変更を Verilog ファイルに保存します。

    Verilog ファイルを変更した後、コンポーネントの準備が完了し、コンポーネントをプロジェクトに使用することができます。最初のサンプル プロジェクトと同じすべてのコンポーネントを追加します。

27. *AN82156_Appendix_Lib* を UpDwnCntrPwm プロジェクトに依存したものとして加えます。
28. UpDwnCntrPwm コンポーネントをプロジェクト回路図にドラッグします。
29. クロック コンポーネントをコンポーネントの「clk」端末に接続して 10kHz に設定します。
34. 図 126 に示すようにデジタル出力ピンコンポーネントをコンポーネントの P0_0_clk、P0_1_tc、P0_2_cmp 端末に接続します。

注: PSoC 4 および PSoC 6 MCU の場合、このプロセスは異なります。 A.1.2 項の手順 36 を参照してください。

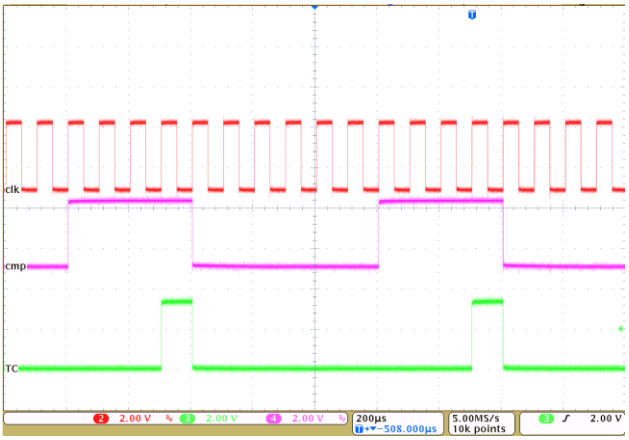
図 126. アップ／ダウンカウンターPWM プロジェクトの回路図



30. 「Save All」を選択して、プロジェクトをビルドして PSoC をプログラムします。

    デフォルトの比較値を 4 に、周期値 9 に設定します。これは、プロジェクトに追加されたピンを使用して観測することができます。

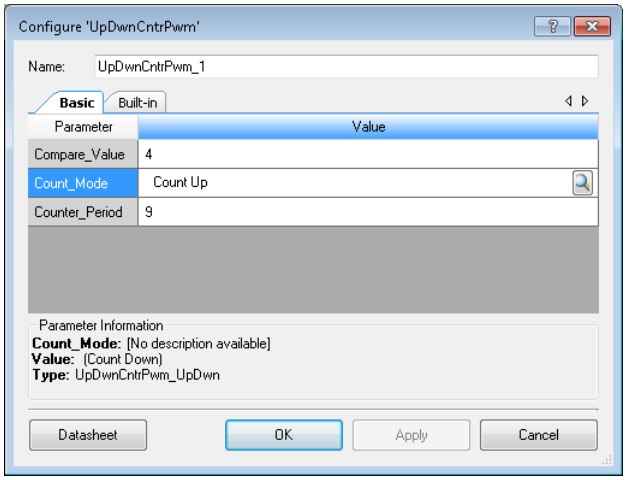
図 127. ダウンカウント PWM の波形



簡単な PWM の例で行ったように、周期値と比較値を変更することができます。PWM がカウントダウンする代わりにカウントアップするようにモードパラメーターを変更することも可能です。

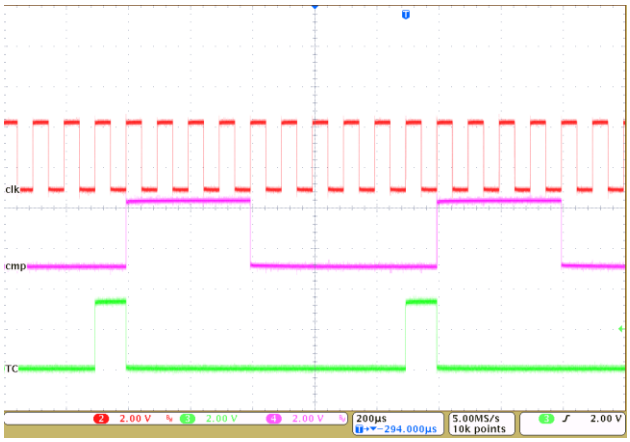
31. プロジェクト回路図に戻り、UpDwnCntrPwm コンポーネントをダブルクリックしてプロパティダイアログを開きます。
32. 図 128 に示すように **Count_Mode** を「Count Up」に変更します。

図 128. アップカウントへの PWM の設定



33. **OK** をクリックして変更を適用します。
34. 「Save All」を選択して、プロジェクトをビルドして PSoC をプログラムします。
35. 図 129 に示すように、TC の後にカウンターが 0 でリロードされる時 cmp 値が HIGH になるため、アップ カウンターを確認できます。

図 129. アップカウント PWM の波形



36. TC の後に出力が HIGH になる理由は、カウントの開始値が比較値より小さく、その後インクリメントされるためであることに注意してください。Down モードの始まりに出力が LOW になる理由は、カウントの開始値が比較値より大きく、その後デクリメントされるためです。

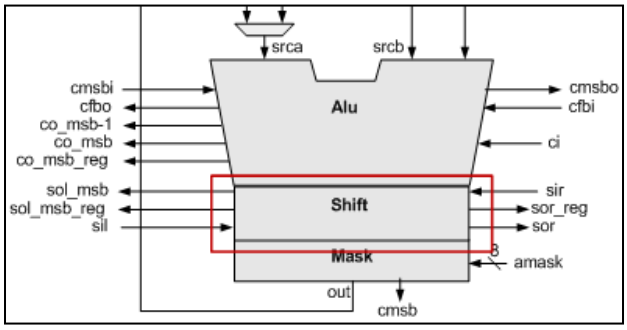
A.4 プロジェクト#4 – 簡単な UART

このサンプルプロジェクトは、簡単な TX UART を単一のデータパスで作成することをデモします。ここではコンポーネントの作成の各ステップについて説明しません。その代わりに、関連したプロジェクトにおいてコンポーネントと Verilog ファイルをご覧ください。完成した ***AN82156_Appendix_Lib*** サンプル プロジェクトで「***Simple_Tx***」というコンポーネントを検索します。このコンポーネントの使用方法を示す例は「***SimpleTx***」プロジェクト内の同じワークスペースに含まれています。

TX UART コンポーネントの詳細

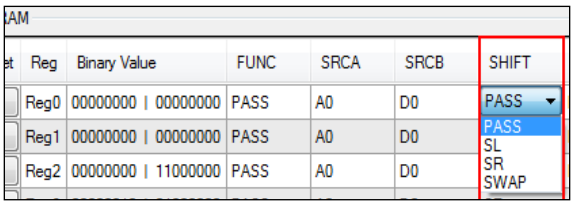
このコンポーネントのデータパスの使用方法は簡単です。データパスの動作は、F0 から 値を A0 にシフトしロードするだけです。図 130 に示すように、ALU の出力に 1 個のシフターがあります。

図 130. シフターロック



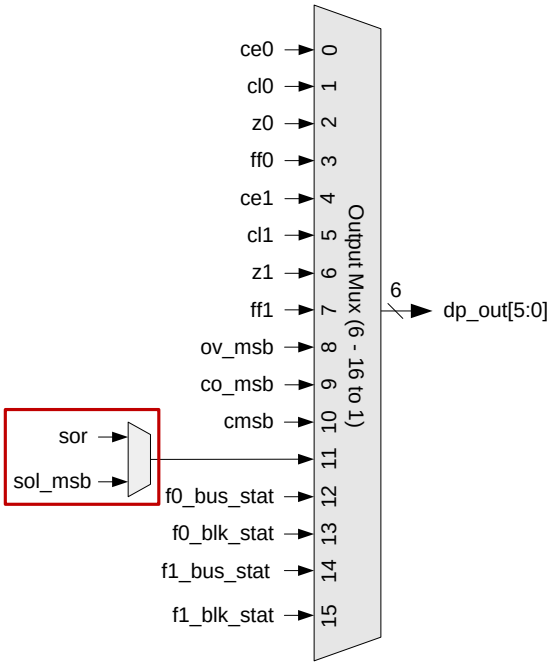
シフターはビットを左または右にシフトし、ニブル交換が可能です。データパスの各コンフィギュレーションは独立してシフターの動作を設定することができます。図 131 に示すように、このオプションは、データパスコンフィギュレーションツールの動的コンフィギュレーション領域でセットされます。

図 131. シフト動作設定



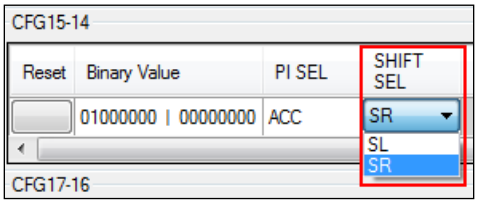
データパス出力マルチプレクサには 1 本のみのシフトアウト(SO)出力があります。図 132 に示すように、この出力は、右シフトアウトと左シフトアウトの間で共有されます。

図 132. データパス出力マルチプレクサ図



このマルチプレクサを適切に設定しなければなりません。この設定は、図 133 に示すように、CFG15-14 *SHIFT SEL* の下の静的コンフィギュレーション (Static Configuration) で行われます。

図 133. SHIFT SEL コンフィギュレーション



この例では、データパスのコンフィギュレーションを制御する Verilog ステートマシンを作成します。このステートマシンは、UDB PLD で実装され、UART (スタートビット、データ、あるいはストップ ビット) 送信のどの部分が次に送信されるかを決めます。

Verilog コードで、以下のコード行を探します。

```
reg  [1:0]  state;  // Main state machine variable and datapath control
```

データパスの入出力を見ると、*state* は CS_Addr を制御するために使用されることが分かります。

```
/*  input   [02:00]          */  .cs_addr({1'b0, state}),
```

データパスコンフィギュレーション／命令の値は Verilog のケースステートメントで実装されるステートマシンにより変更されます。

```
case (state)

       STATE_IDLE:…

       STATE_START:…

       STATE_DATA:…

       STATE_STOP:…

endcase
```

これらのケースはそれぞれ以下の通りに定義されます。

```
// Main State machine states

// Idle / Stop bit 1
localparam STATE_IDLE  = 2'b00;
// Stop bit 2
localparam STATE_STOP  = 2'b01;
// Start bit
localparam STATE_START = 2'b10;
// Data
localparam STATE_DATA  = 2'b11;
```

ステート マシンは4つの状態のみを持っています。これらの 4 状態はデータパスのコンフィギュレーションを制御するために使用されます。そのため、データパスは4種の個別のコンフィギュレーションを必要とします。

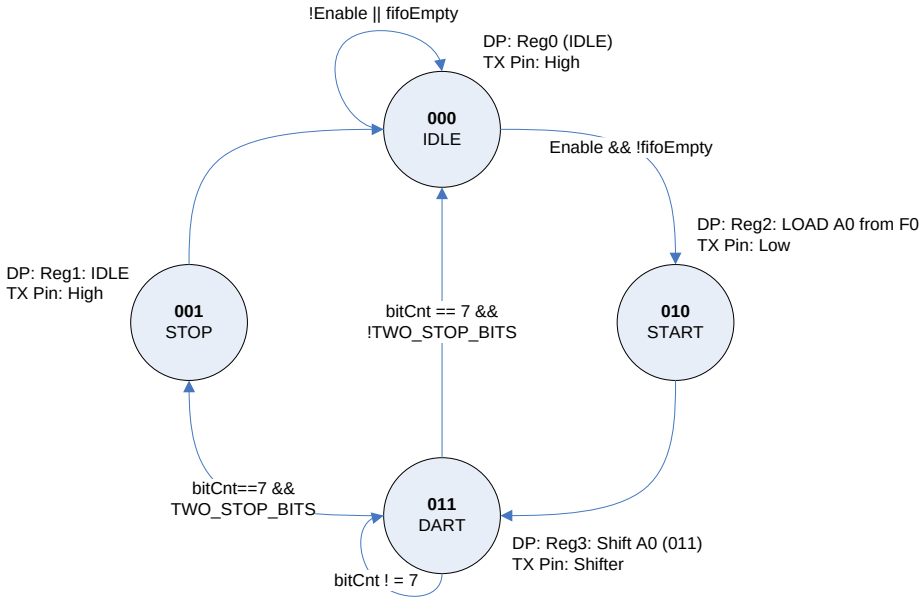
表 12. 簡単な Tx データパスコンフィギュレーション

REG	FUNC	SRCA	SHIFT	A0 WR SRC
000	Pass	A0	None	None
001	Pass	A0	None	None
010	Pass	A0	None	F0
011	Pass	A0	SR	ALU

コードが Verilog ステート マシンを通ることによって、データパスのコンフィギュレーションが変更されます。これは一般的な使用方法です。殆どの複雑なコンポーネントは、データパスのコンフィギュレーションを順序付けるために Verilog ステート マシンを必要とします。

図 134 に簡単な TX Verilog ステート マシンの動作を示します。

図 134. TX UART Verilog フローチャート



まず、ステート マシンは、fifoEmpty ステータス ビットを監視することで、新しいデータが FIFO に書き込まれるのを待ちます。新しいデータが入った後、TX ラインを LOW にすることで、START ビットを送信します。この状態では、データパスは F0 内の値を A0 にロードします。

次の状態で、データパスは A0 のデータをシフトアウトします。これが完了した後、ストップビットを送信します。1 つか 2 つのストップビットを送信することができます。

データ状態にある時は、データをシフトアウトします。Start 状態は、TX ラインを LOW に、Stop および IDLE 状態は TX ラインを HIGH にします。これは以下の Verilog コードにより実行できます。

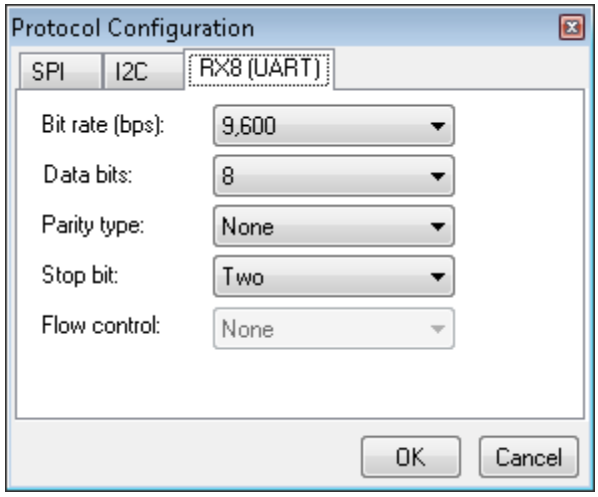
```
// This next statement determines the tx output.  if in data state, output the shift
register output "srOut", else output a low during the start state, and a high during
stop state.
assign tx = (state[1:0] == STATE_DATA ) ? srOut : ( !state[1] );
```

ステート マシンは、Data 状態にない場合、「state」の msb の反対符号のものを TX ラインに駆動します。Start 状態では、msb が 1 のため、「0」を出力します。Stop および IDLE 状態では、最上位ビットが 0 であるため、「1」を出力します。

このプロジェクトの main コードは 2 ストップ ビットでコンポーネントを有効にしてから、9600 ボーで値 0～10 を連続して送信します。レシーバーを 9600 ボーおよび 2 ストップ ビットにコンフィギュレーションします。

ブリッジ コントロール パネル (BCP) というプログラムが PSoC Creator のインストールに含まれています。BCP を使用して RX キャラクタを受信することができます。BCP でも、TX 出力が接続する COM ポートに接続します。**Tools > Protocol Configuration** では、図 135 のように RX8 (UART)をコンフィギュレーションします。

図 135. BCP RX8 のコンフィギュレーション

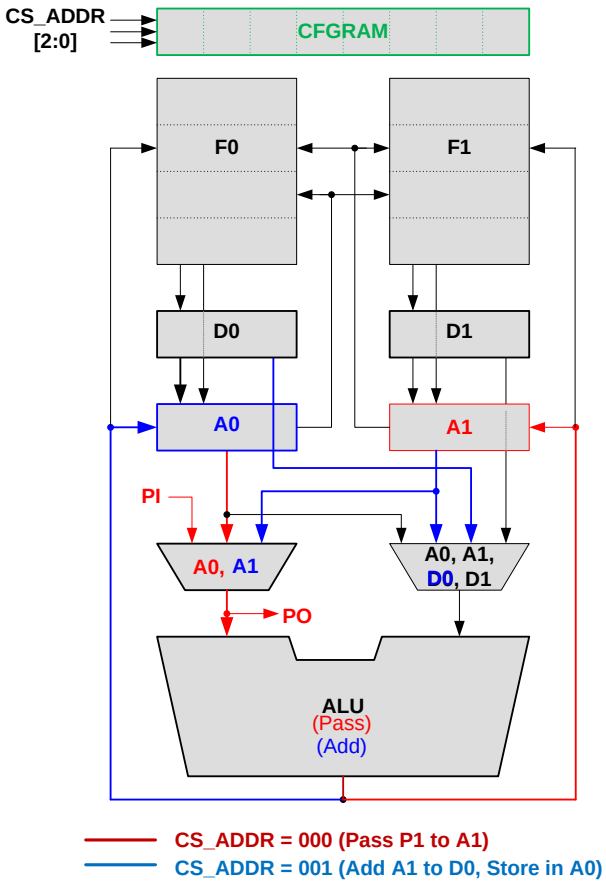


BCP のエディターでは、次のテキストを加えます: rx8 x x x x x x x x x。これは、選択した COM ポートから 11 バイトを読み出します。引き続きデータを受信するために **Repeat** ボタンを押すことができます。

A.5 プロジェクト#5 – パラレル入力とパラレル出力の例

本サンプル プロジェクトでは、データパスのパラレル入力とパラレル出力の使用方法をデモします。デモするために 8 ビット並列加算器を 1 つ使用します。加算器はパラレル入力 (PI) から 8 ビット パラレル値を読み出して、その値を A1 に格納します。次に、A1 に格納する値を取得して D0 に格納される固定値を追加します。その後、その値はパラレル出力 (PO) で出力されます。

図 136. 並列加算器の実装



この例は、ユーザーが前述のサンプルプロジェクトに説明された概念に精通していることを前提とします。1 つの完成した PI_PO_Example プロジェクトがこのアプリケーションノートに含まれています。

A.5.1 サンプルプロジェクトの手順

混乱を避けるために、以前のサンプル プロジェクトのコンポーネントを変更する代わりに、新しいコンポーネントを作成します。コンポーネント作成の基本手順は同じです。

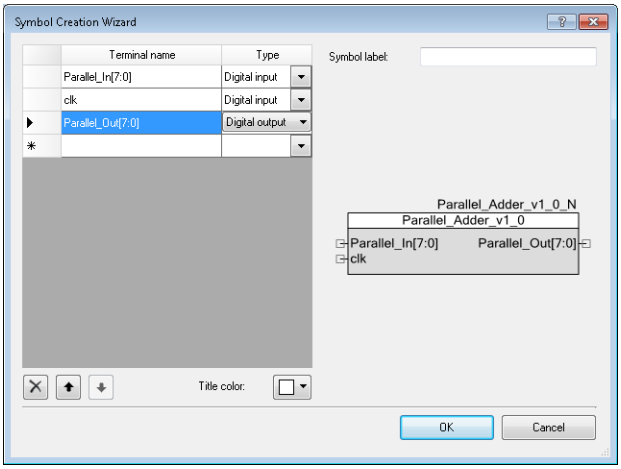
1. PSoC Creator を起動し、簡単な 8 ビットの例で使用される「**AN82156_Appendix**」ワークスペースを開きます。「*PI_PO_Example*」という新しいプロジェクトをワークスペースに追加します。

   新しいワークスペースを起動することが可能ですが、この例では、ユーザーが以前と同じようなワークスペースを使用していることを前提にします。これにより、ライブラリ管理と依存関係が簡単になります。

2. Symbol Wizard を使用して新しいコンポーネントを *AN82156_Appendix_Lib* に追加します。この例ではコンポーネントに「*Parallel_Adder_v1_0*」と名前を付けます。

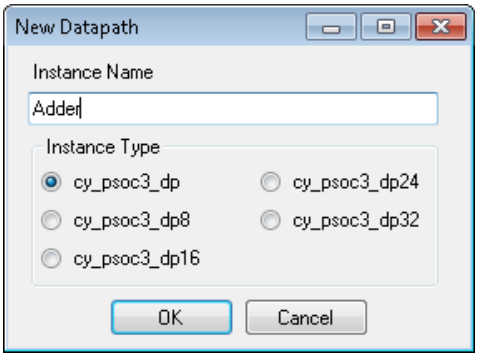
3. *clk* 入力と Parallel_In[7:0]入力 (これは 8 ビット入力を定義) を追加します。図 137 に示すように、Parallel_Out[7:0] 出力も定義します。

図 137. 並列加算器の端末を追加



4. シンボル回路図のあるページを右クリックし、シンボルのプロパティを追加します。
   - Doc.APIPrefix = *Parallel_Adder*
   - Doc.CatalogPlacement = *AN82156_Appendix/Digital/Parallel_Adder*
   - Doc.DefaultInstanceName = *Parallel_Adder*
5. *Add_Value* という新しいシンボルパラメーターを追加し、タイプを uint8 にセットし、**Hardware** を True にセットします。
6. シンボル エディター内の何も無い場所を右クリックして、新しいコンポーネントシンボルの Verilog ファイルを生成します (この時、すべての設定はデフォルトの値にします)。
7. 「Save All」を選択して、すべての変更を適用します。
8. データパス コンフィギュレーション ツールを起動して、作成したばかりの *Parallel_Adder_v1_0.v* ファイルを開きます。
9. データパスのコンフィギュレーションを新規作成し「adder」と名付けます。「cy_psoc3_dp」タイプを使用します。図 138 を参照してください。

図 138. データパス選択



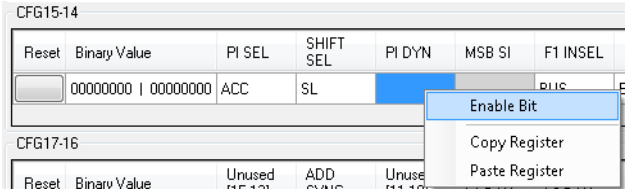
10. この選択肢ではデータパスの PI と PO 信号へアクセスすることができます。その他の 4 つのインスタンスタイプは PI と PO へのアクセスを許可しません。
11. cy_psoc3_dp は 8 ビットのみのインスタンスです。8 ビット以上必要な場合、これらのインスタンスを 2 つ以上配置し、Verilog で変更する必要があります。付録 D – 自動生成 Verilog コードにそれらをチェーン接続して、24 ビットのデータパスを構成する例を示します。
12. 表 13 のように CFGRAM セクションをコンフィギュレーションします。

表 13. 例 5 のデータパスコンフィギュレーション

REG	FUNC	SRCA	SRCB	A0 WR SRC	A1 WR SRC	CFB EN
000	PASS	A0	D0	NONE	ALU	ENBL
001	ADD	A1	D0	ALU	NONE	DSBL

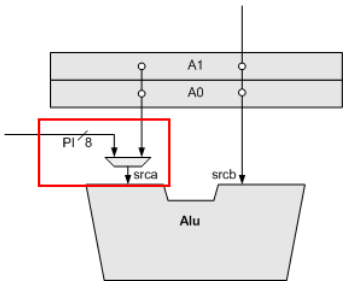
13. 最初のコンフィギュレーション (REG 000) は PI の値を取得して、それを A1 に格納します。2 番目のコンフィギュレーション (REG 001) は A1 に格納された値を取得して、それに D0 を追加し、A0 に格納します。
14. 次に、PI 制御をコンフィギュレーションします。
15. 図 139 に示すように、CFG15-14 フィールドで、PI DYN のフィールドを右クリックし、Enable Bit を選択します。

図 139. 動的な PI 制御を有効にする



16. **PI DYN** を EN にセットします。
17. これにより、SRCA 入力を PI と、A0 または A1 から動的に選択することができます。図 140 を参照してください。

図 140. scra マルチプレクサ



18. このマルチプレクサの選択は CFGRAM で、ビット CFB EN で制御されます。このビットが DSBL にセットされた場合、SRCA 入力は A0 と A1 のいずれかから取得されます。ENBL にセットされた場合、SRCA 入力は PI から取得されます。
19. 表 13 を見ると、最初のコンフィギュレーションでは、CFB EN は ENBL にセットされます。これは SRCA が PI から取得されることを意味します。よって、最初のコンフィギュレーションではデータパスは PI から A1 に値を渡します。
20. 2 番目のコンフィギュレーションでは、CFB EN は DSBL にセットされます。これは SRCA 入力が CFGRAM により制御され、この場合には A1 にセットされることを意味します。
21. D0 の初期値を Add_Value にセットします。
22. 変更を保存して、データパスコンフィギュレーションツールを閉じます。

    次に、これらの機能を実装するために幾つかの Verilog コードをコンポーネントに追加する必要があります。
23. *Parallel_Adder_v1_0.v* ファイルを開いて、以下のテキストを見つけます。

```
//  Your code goes here
```

24. そのテキストを以下に置き換えます。

```
/* Register to hold state of statemachine*/
reg state;
wire[7:0] po;
```

```
/*State loads the value from the PI into A1, latches value in A0 out to PO*/
localparam STATE_LOAD  = 2'b00;

/*State adds the value in A1 to D0 and stores in A0*/
localparam STATE_ADD  = 2'b01;

/*State machine is always run on positive edge of clock*/

always @( posedge clk )
begin
       case (state)
/* Datapath loads in value from PI to A1, the value in A0 is latched to PO*/
           STATE_LOAD:
           begin
                  state <= STATE_ADD;

/*we must latch the PO value here, because in the next state PO is not valid*/
                  Parallel_Out <= po;
           end

           STATE_ADD:
           begin
                  state <= STATE_LOAD;
           end
           endcase
```

25. データパス論理に以下のリンクを作ります。
    - `.clk()` を `.clk(clk)` に変更します。
    - `.cs_addr(3'b0)` を `.cs_addr({2'b0, state})` に変更します。
    - `.pi()` を `.pi(Parallel_In)` に変更します。
    - `.p0()` を `.p0(po)` に変更します。

    2 つのデータパスのコンフィギュレーション間をトグルする 2 ステートのステート マシンを作成しました。1 つ目のコンフィギュレーションは PI 上の値を取得して、それを A1 に格納します。2 つ目は A1 に格納される値を取得して、それを D0 に追加し、その値を A0 に格納します。

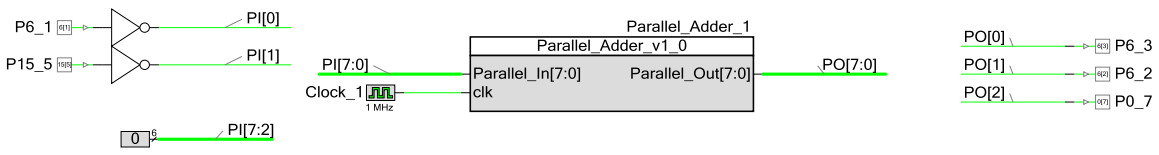
    A1 と A0 のいずれを選択しても、PO はその選択したレジスタに接続します。A0 が SRCA のために選択された時にのみ結果が有効なため、適切なタイミングに PO を登録する必要があります。下記はこのラインに必要なものです。

    ```
    Parallel_Out <= po;
    ```

    PO から値を直接取得して、Parallel_Out 信号 (コンポーネント出力) に格納します。次の状態では、SRCA が A1 として選択されたため、PO は有効ではありません。よって、PO をロード状態で登録する必要があります。

26. すべての変更を行った後、Save All (すべて保存)を行います。
27. AN82156_Appendix_Lib を PI_PO_Example プロジェクトに依存したものとして加えます。
28. 回路図に Parallel_Adder をドラッグします。
29. 図 141 のように回路図を構成します。

図 141. 並列加算器回路図



P6_1 と P15_5 はデジタル入力ピン コンポーネントであり、**抵抗プルアップ**としてコンフィギュレーションされます。

P6_3、P6_2 と P0_7 はデジタル出力ピン コンポーネントです。

この回路図は CY8CKIT-030 と CY8CKIT-050 のために特別に設計されています。 別のハードウェアプラットフォームを使用している場合は、ピン配列を次のように変更する必要があります。

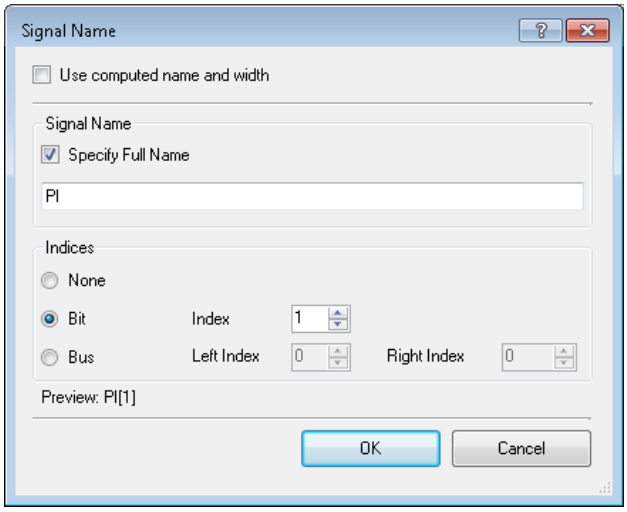
表 14. PSoC 4 キットのピンマッチング

キット	推奨入力ピン	推奨出力ピン
CY8CKIT-042	P0[7], P1[0]	P0[5], P0[6], P0[0]
CY8CKIT-043	P0[7], P1[0]	P1[1], P1[2], P1[3]
CY8CKIT-044	P0[7], P1[0]	P1[1], P1[2], P1[3]
CY8CKIT-049-42xx	P0[7], P1[0]	P0[5], P0[6], P0[0]

ワイヤに名前を付けるには (図 142 を参照):

a. Wire (ワイヤ) をダブルクリックします。
b. **Use Computed name and width** チェックボックスからチェックを外します。
c. **Specify Full Name** チェックボックスにチェックを入れます。
d. ネット (ワイヤ) の名前を入力して、**Indices** があるのかを選択します。

図 142. ワイヤ名付けダイアログ



30. コンポーネント カスタマイザー内で Add_Value を 2 にセットします。
31. 「Save All」を行い、プロジェクトをプログラムします。
32. CY8CKIT-050 または CY8CKIT-030 上でワイヤを P0_7 と LED2 の間に配置します。

ボタンと LED は簡単な加算器がどのように動作するかを示します。LED 4 は出力のビット 0 を表示し、LED 3 はビット 1 を表示し、LED 2 はビット 2 を表示します。

SW2 は入力のビット 0 を表示し、SW3 は入力のビット 1 を表示します。

表 14 に記載されているキットのいずれかを使用している場合は、外部ボタンを P1 [0]に接続し、外部 LED を表に記載されている出力ピンに接続する必要があります。

スイッチを押さずに、出力ビット 1 に接続されている LED が点灯します。 これは、D0 の値が 2 であるためです。入力ビット 0 に接続されているボタンを押すと、出力ビット 0 と出力ビット 1 に接続されている LED が点灯し、値が 3 になります。

これはデータパスの PI および PO を使用する方法をデモする非常に簡単な例です。これで、これらの信号を使用して PI と PO でより複雑な設計を作成する方法を知りました。

B 付録 B – データパスコンフィギュレーションツールのチートシート

本節では、データパス コンフィギュレーション ツールと基礎となるデータパス ハードウェアとの関係を示します。

GUI は、図 143 に示すように動的コンフィギュレーションと静的コンフィギュレーションの 2 つのセクションに分けられています。

- 動的コンフィギュレーション - データパスを異なる状態で異なる動作をするように設定することが可能
- 静的コンフィギュレーション - すべての状態で同様に動作

図 143. データパスコンフィギュレーションツールインターフェースのセクション

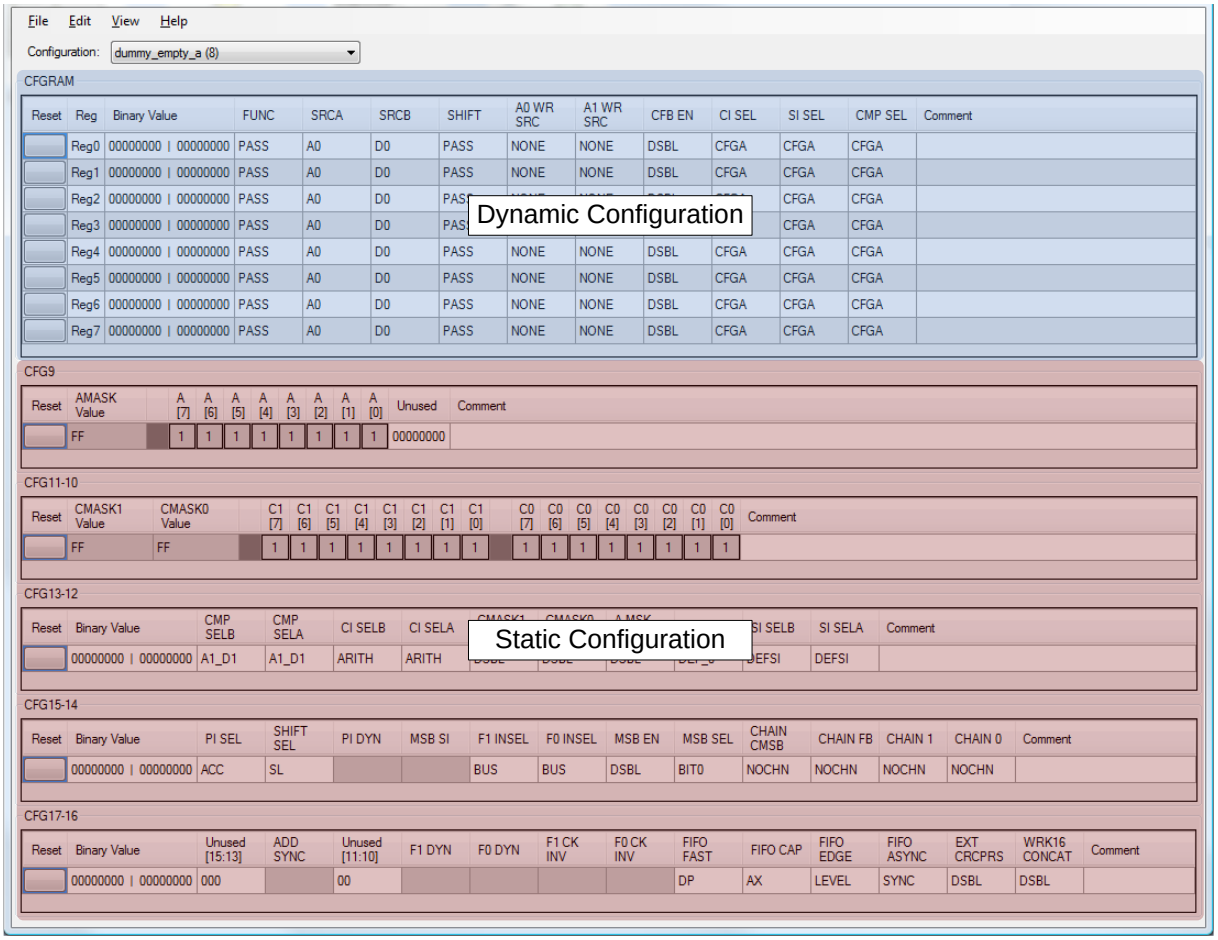
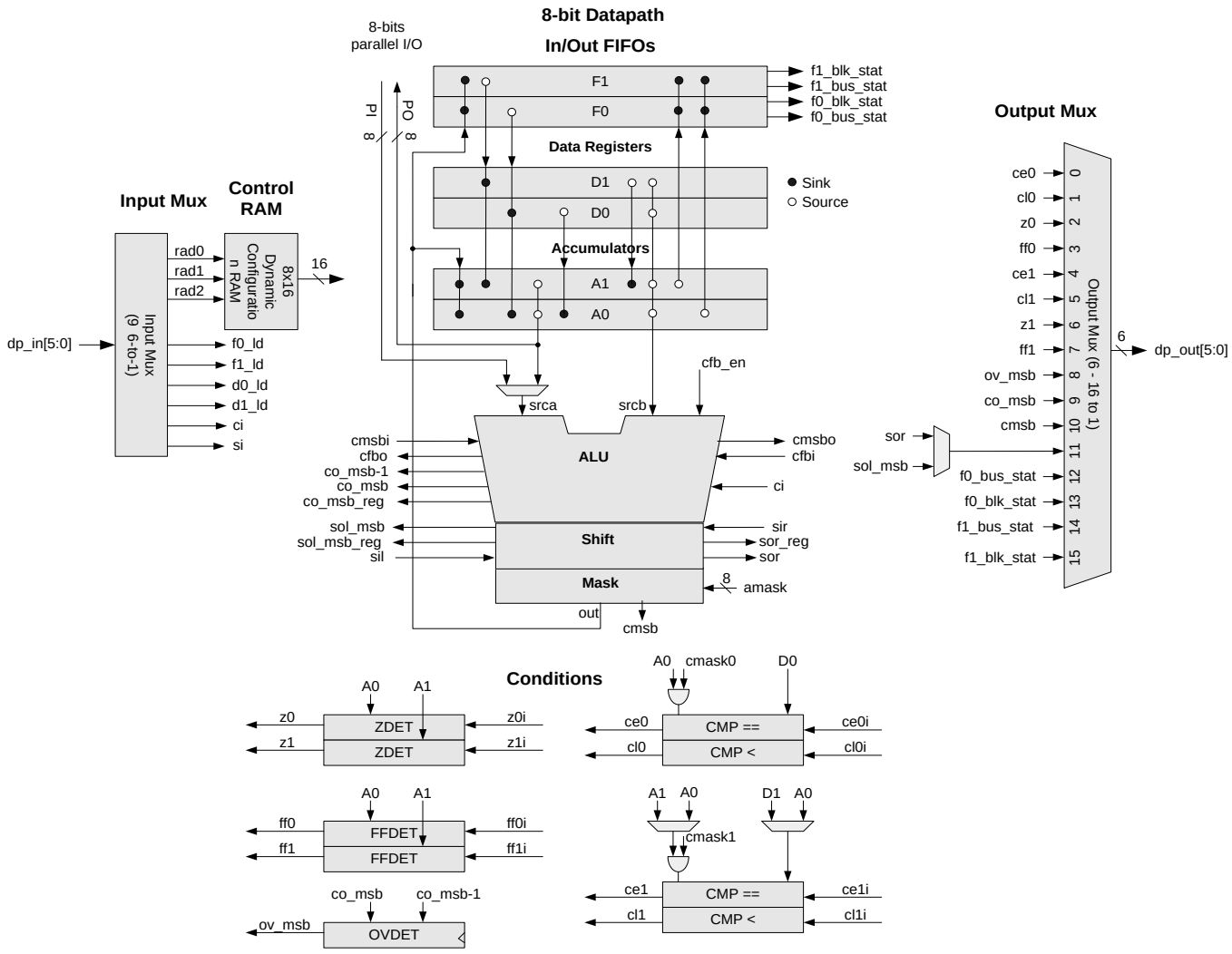


図 144. データパスブロック図



B.1 動的コンフィギュレーション RAM (CFGRAM) セクション

動的コンフィギュレーションセクションは動的コンフィギュレーション RAM を表示するものです。8 つのコンフィギュレーション／命令のデータパスの動作をコンフィギュレーションします。以下の表は GUI 内の各フィールドの機能を示します。

表 15. データパスコンフィギュレーションツールの CFGRAM セクション

動的コンフィギュレーションの RAM セクション

データパスコンフィギュレーションツール

CFGRAM

Reset	Reg	Binary Value	FUNC	SRCA	SRCB	SHIFT	A0 WR SRC	A1 WR SRC	CFB EN	CI SEL	SI SEL	CMP SEL
	Reg0	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg1	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg2	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg3	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg4	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg5	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg6	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA
	Reg7	00000000 | 00000000	PASS	A0	D0	PASS	NONE	NONE	DSBL	CFGA	CFGA	CFGA

データパスブロック図

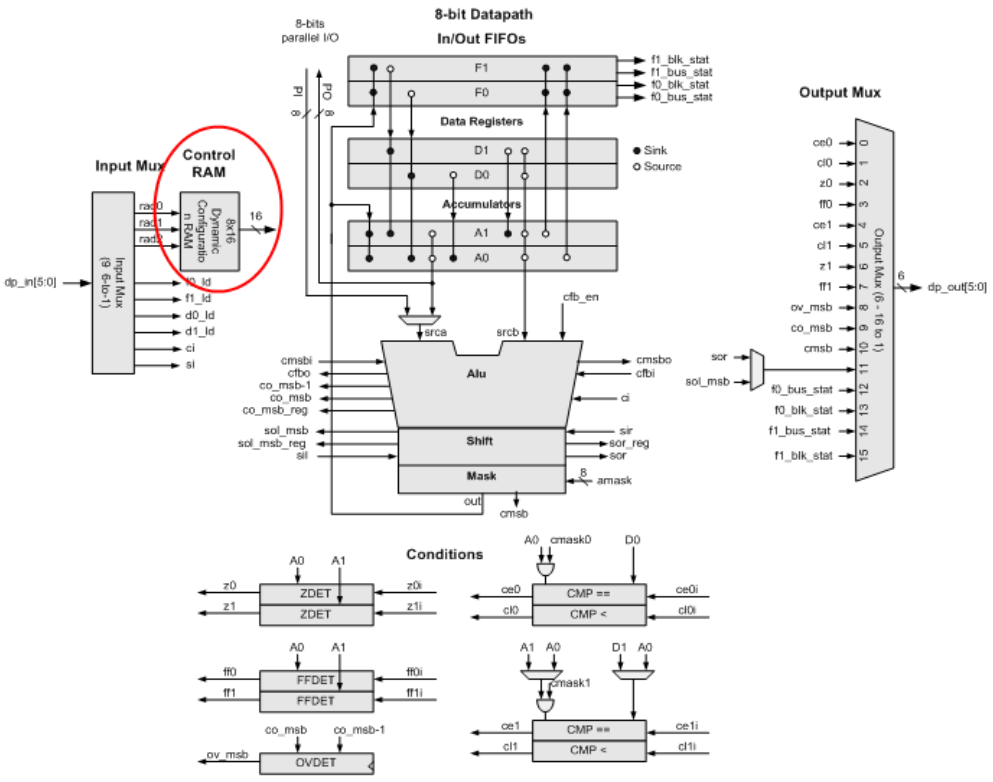


表 16. 動的なコンフィギュレーションのセクションのカラムの説明

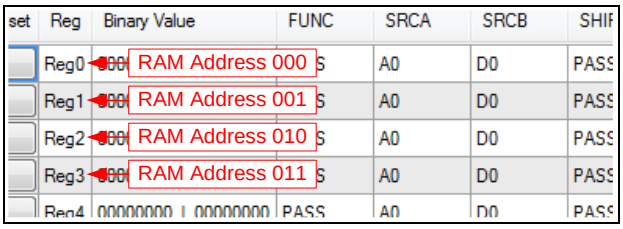
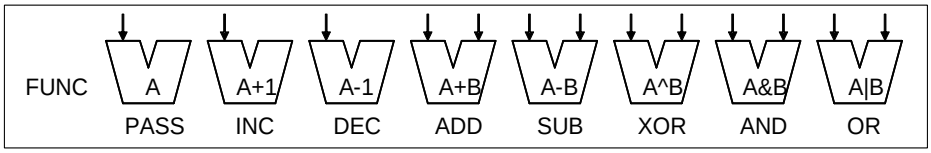
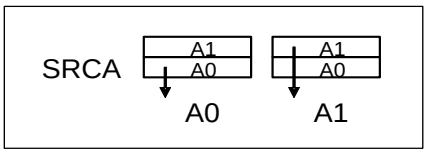
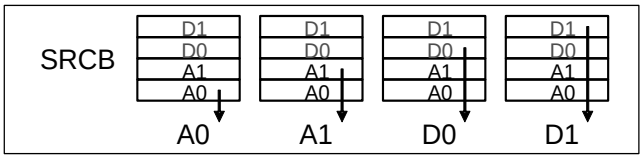
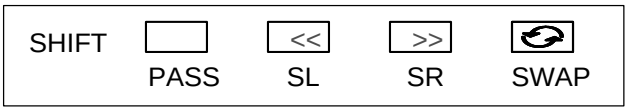
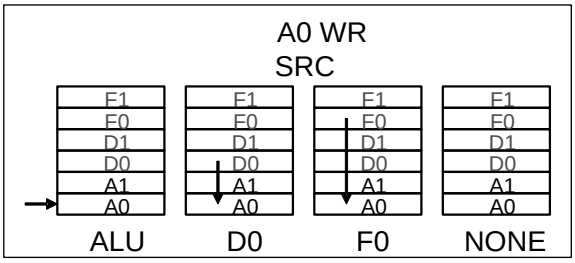
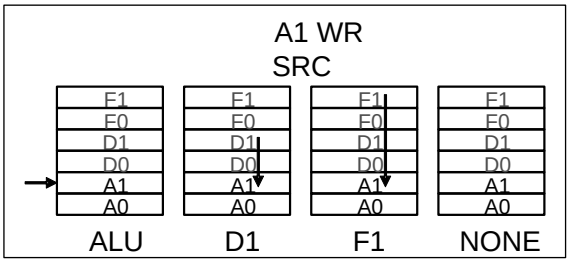
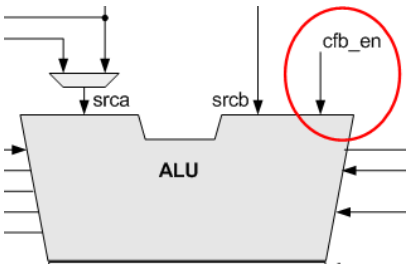
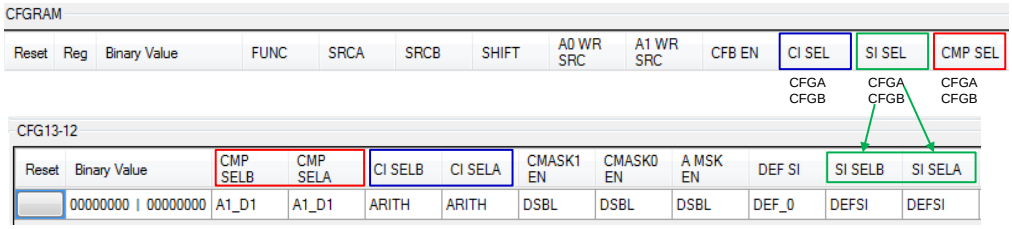
動的コンフィギュレーションのセクション	
Reg このカラムは、行が該当する「コンフィギュレーション／命令」を示す。3 CFGRAM アドレス信号の値がどのコンフィギュレーションが選択されるかを決定するため、それぞれのコンフィギュレーションに適切な行を選択	
FUNC このカラムは、8 つ ALU 機能の中でどれがそのコンフィギュレーションで実行されるかを決定	
SRCA このカラムは ALU の「SRCA」入力のソースを決定 PI も SRCA のソースになることが可能。 表 19 を参照してください。	
SRCB このカラムは ALU の srcb 入力のソースを決定	
SHIFT このカラムはシフトブロックの機能を決定	
A0 WR SRC このカラムは、ALU 演算が完了した後の A0 レジスタの値を決定	

表 16. 動的なコンフィギュレーションのセクションのカラムの説明 (続き)

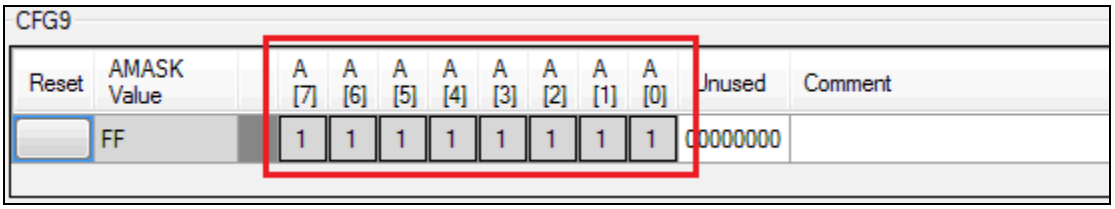
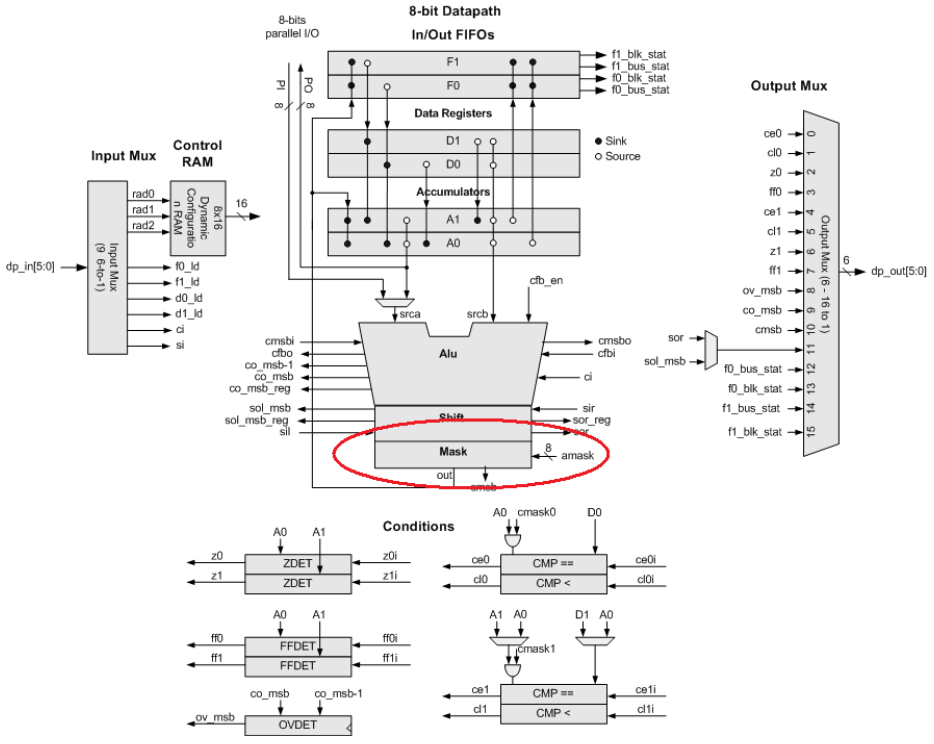
Dynamic Configuration Section	
A1 WR SRC このカラムは、ALU 演算が完了した後の A1 レジスタの値を決定	
CFB EN CRC コンフィギュレーションイネーブル – 以下の PI DYN を参照してください。 CFG15 と CFG14 レジスタ 表 19 CFG13 と CFG12 レジスタの定義	
CI SEL, SI SEL, CMP SEL キャリーイン セレクト、シフトインセレクトおよびコンペア セレクトそれぞれには A と B という 2 つのコンフィギュレーション オプションがある。各コンフィギュレーションに適切なオプションを選択することが可能 詳細は表 18 を参照してください。	

B.2 静的コンフィギュレーションのセクション

静的コンフィギュレーションのセクションは、表 17、表 18、表 19、表 20 および表 21 に示すように、CFG9～CFG17 データパス レジスタを表示します。それらのレジスタは、シフト方向、マスキング、FIFO コンフィギュレーションおよびチェーニングの静的な機能を制御します。

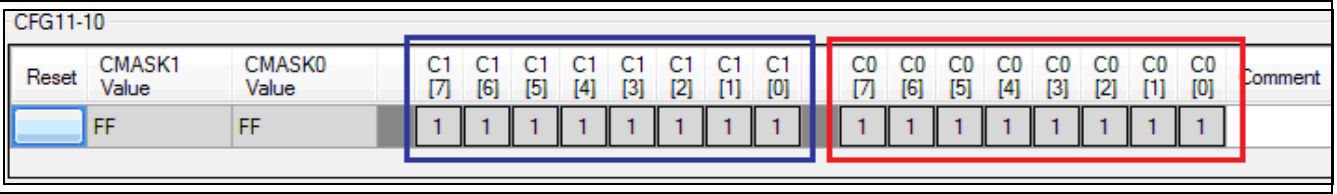
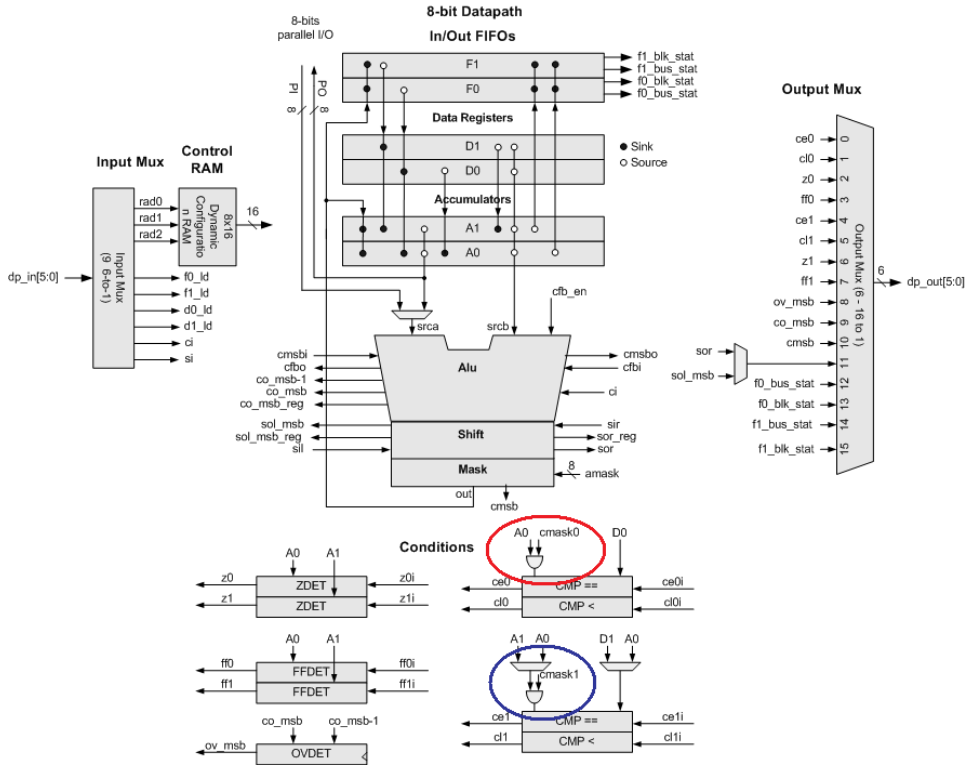
B.2.1 CFG9 レジスタ

表 17. CFG9 レジスタの定義

AMASK の値
データパスコンフィギュレーションツール

■ AMASK 値 – このフィールドはデータパス ALU ブロックに適用される 8 ビットマスク値を含む。シフトレジスタの出力はこのレジスタの値と AND される。この機能はデフォルトで無効。それを有効にするには CFG12 の AMASK EN ビットをセットする必要がある
データパスブロック図


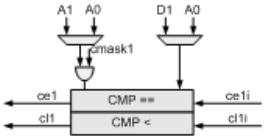
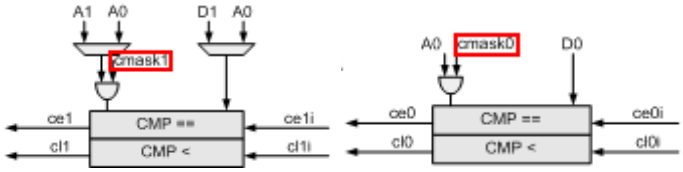
B.2.2 CFG11 と CFG10 レジスタ

表 18. CFG11 と CFG10 レジスタの定義

CMASK0 と CMASK1 の値
データパスコンフィギュレーションツール

■ CMASK0 と CMASK1 - これらのフィールドは、コンパレータブロック入力と共に使用されるマスク値をセット。比較の前に、A0 または A1 レジスタの値はこれらのレジスタの値と AND される。それらを有効にするためには、CMASK0 EN と CMASK1 EN ビットを CFG12 でセットする必要がある (表 19)
データパスブロック図


B.2.3 CFG13 と CFG12 レジスタ

表 19. CFG13 と CFG12 レジスタの定義

<table>
<tr><th colspan="2">CFG13-12 コンフィギュレーションの選択</th></tr>
<tr><td colspan="2">Datapath Configuration Tool</td></tr>
<tr><td colspan="2">
CFG13-12
<table>
<tr><th>Reset</th><th>Binary Value</th><th>CMP SELB</th><th>CMP SELA</th><th>CI SELB</th><th>CI SELA</th><th>CMASK1 EN</th><th>CMASK0 EN</th><th>A MSK EN</th><th>DEF SI</th><th>SI SELB</th><th>SI SELA</th></tr>
<tr><td></td><td>00000000 | 00000000</td><td>A1_D1</td><td>A1_D1</td><td>ARITH</td><td>ARITH</td><td>DSBL</td><td>DSBL</td><td>DSBL</td><td>DEF_0</td><td>DEFSI</td><td>DEFSI</td></tr>
</table>
</td></tr>
<tr><th colspan="2">CFG13-12 の詳細</th></tr>
<tr><td>CMP SELB と CMP SEL A
比較ブロック 1 の比較 B と比較 A オプションをコンフィギュレーション。RAM コンフィギュレーションの CMP SEL フィールドは所与のサイクルに対してオプション A と B のどちらが有効であるか決定
注: ブロック 0 を比較する場合、D0 とマスクされた A0 の値しか比較できない</td><td>A1_D1: A1 < D1, A1 == D1
A1_A0: A1 < A0, A1 == A0
A0_D1: A0 < D1, A0 == D1
A0_A0: A0 < A0 , A0 == A0
 
</td></tr>
<tr><td>CI SELB & CI SEL A
キャリーインのソースを選択</td><td>ARITH: キャリーは ALU 演算によって制御される
REGIS: キャリーインは、以前のサイクルで登録したキャリーアウト
ROUTE: キャリーインはデータパスの入力 (複数) から 1 個選ばれる
CHAIN: キャリーインはチェーン内の以前のデータパスから駆動される</td></tr>
<tr><td>CMASK0 EN, CMASK1 EN, AMASK EN
比較 1 と比較 0 ブロックのマスク (右図に示される) と ALU の出力でのマスク ブロック (示されない) を有効にする表 17 と表 18 を参照してください。</td><td> 
</td></tr>
<tr><td>DEF SI
デフォルトのシフトイン値が 0 または 1 であるかを定義。SI SELB または SI SELA が DEFSI に設定される必要があることに注意してください。</td><td>DEF_0: Zero
DEF_1: One</td></tr>
<tr><td>SI SELB & SI SEL A
A と B のシフトインコンフィギュレーション用のシフトイン ソースを選択</td><td>DEFSI: DEF SI の定義に応じて、シフトイン値は 0 または 1
REGIS: シフトインは、以前のサイクルのシフトアウト
ROUTE: シフトインはデータパス入力から選ばれる
CHAIN: シフトインはチェーン内の前のデータパスにより駆動される</td></tr>
</table>

B.2.4 CFG15 と CFG14 レジスタ

表 20. CFG15 と CFG14 レジスタの定義

CFG15-14 コンフィギュレーションの選択

データパスコンフィギュレーションツール

CFG15-14

Reset	Binary Value	PI SEL	SHIFT SEL	PI DYN	MSB SI	F1 INSEL	F0 INSEL	MSB EN	MSB SEL	CHAIN CMSB	CHAIN FB	CHAIN 1	CHAIN 0
	00000000 | 00000000	ACC	SL			BUS	BUS	DSBL	BIT0	NOCHN	NOCHN	NOCHN	NOCHN

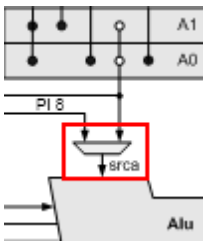
CFG15-14 Details

PI SEL

SRCA 用のマルチプレクサ入力を制御。その入力は Dynamic Config での SRCA オプションにより、または並行入力から供給される

ACC: ソース A は動的な設定領域内の選択肢により供給される

PIN: ソース A はデータパスへのパラレル入力により供給される

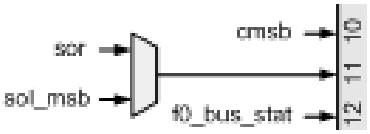


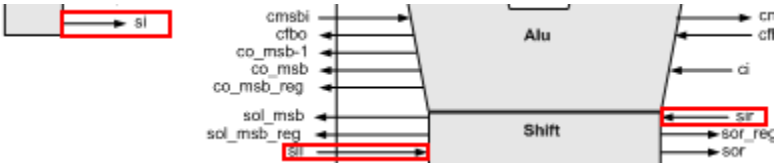
Shift SEL

データパス出力マルチプレクサでシフトアウト信号の選択を制御

SL: *sol_msb* が出力として選択される

SR: *sor* が出力として選択される





PI DYN

ALU ASRC 入力にパラレル入力 (PI) マルチプレクサ選択の動的制御を有効にする

DS: PI マルチプレクサはレジスタ内の PI SEL ビットを使用して静的に制御される

EN: PI マルチプレクサは CFB_EN ビットを使用して動的に制御される (PI SEL が「0」と前提にする)。このビットがセットされ、CFB_EN が「0」の場合、ALU ASRC 入力は A0 または A1 で、CFB_EN が「1」の場合、ALU ASRC 入力は PI からルーティングされる

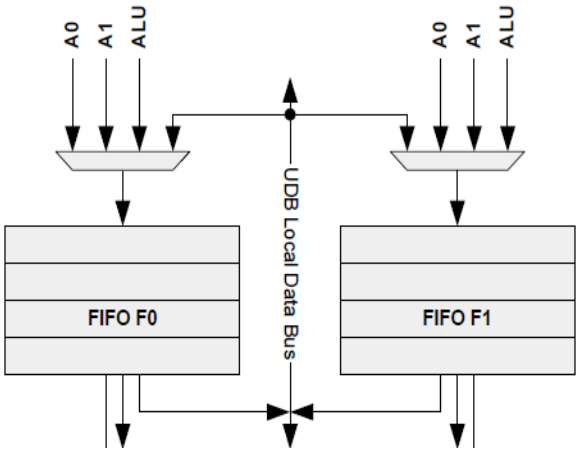
MSB SI

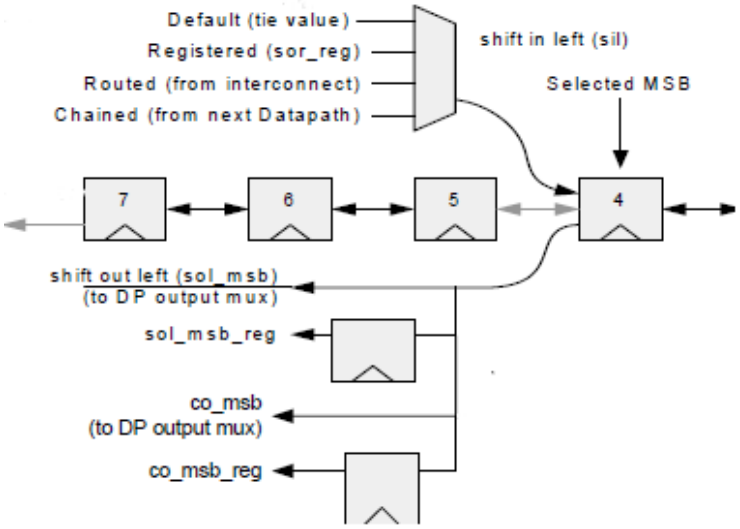
算術右シフト処理をサポート

REG: 選択のデフォルト シフトは DEF SI 値 (CFG12、表 19) により定義される

MSB: 値のデフォルト シフト (SELA[1:0]/SELB[1:0]では「00」選択肢として定義)を現在に定義される MSB (CFG14 内の MSB EN と MSB SEL フィールド) で上書きする

<table>
<tr><th colspan="2">CFG15-14 の詳細 (続き)</th></tr>
<tr><td>F1 INSEL と F0 INSEL

FIFO 1 と FIFO 0 の入力ソースを定義。</td><td>バス: FIFO 入力は CPU バスで、FIFO 出力は A0 または D0 レジスタ。
A0: FIFO 入力が A0。FIFO 出力が CPU BUS。
A1: FIFO 入力が A1。FIFO 出力が CPU BUS。
ALU: FIFO 入力が ALU。FIFO 出力が CPU BUS。
 
</td></tr>
<tr><td>MSB EN と MSB SEL

ALU 出力の MS ビットを変更することが可能。これらの 2 つのコンフィギュレーションにより、この機能を無効／有効にし、どのビットが MS ビットであるかを選択することが可能になる。</td><td>MSbit がビット 7 以外のものに変更されると、シフトイン、シフトアウトおよびキャリーアウトはすべてそれに応じて変更される。
 
</td></tr>
<tr><td>Chain CMSB

CRC MSB にチェーン内の次のデータパス ブロックからこのブロックにチェーン接続することを許可。</td><td>CRC MSB 信号の流れは MS ブロックから LS ブロックまで。このデータパスが CRC 演算の最上位ビットを含まない場合、このビットをセット。
NOCHN: CRC MSB はチェーン接続されない
CHNED: CRC MSB はチェーン内の次のデータパスブロックからチェーン接続される</td></tr>
<tr><td>Chain FB

CRC MSB にチェーン内の先行のデータパス ブロックからこのブロックにチェーン接続することを許可。</td><td>CRC FB 信号の流れは LS ブロックから MS ブロックまで。このデータパスが CRC 演算の最下位ビットを含まない場合、このビットをセット
NOCHN: CRC フィードバックはチェーン接続されない
CHNED: CRC フィードバックはチェーン内の前のデータパスブロックからチェーン接続される</td></tr>
<tr><td>Chain 1 & Chain 0

CL0、CL1、CE0、CE1、Z0、Z1、FF0 と FF1 の出力がチェーン接続されるかを定義。</td><td>チェーン接続 (CHNED) にセットされると、以前のデータパスの条件はこのデータパスにチェーン接続される。チェーン 0 は CL0、CE0、Z0 および FF0 条件に影響を与える。チェーン 1 は CL1、CE1、Z1 と FF1 条件に影響を与える</td></tr>
</table>

B.2.5 CFG17 と CFG16 レジスタ

表 21. CFG17 と CFG16 レジスタの定義

FIFO のコンフィギュレーション

データパスコンフィギュレーションツール

CFG17-16

Reset	Binary Value	Unused [15:13]	ADD SYNC	Unused [11:10]	F1 DYN	F0 DYN	F1 CK INV	F0 CK INV	FIFO FAST	FIFO CAP	FIFO EDGE	FIFO ASYNC	EXT CRCPRS	WRK16 CONCAT
	00010000 | 00000000	000	ADD	00					DP	AX	LEVEL	SYNC	DSBL	DSBL

- ADD SYNC –追加の同期フリップフロップが FIFO ブロックステータス信号に追加されるか否かを決定。これは、バス クロック分解能で読み書き間のサイクル タイミング、およびデータパス出力配置上の新しいステータスのアサートを制御。両方の FIFO のためにこれを制御する唯一のコンフィギュレーション ビットがある。次のページの「FIFO のコンフィギュレーション」節を参照してください。

- **NONE**: FIFO ブロック ステータスの出力にフリップフロップを追加しない
- **ADD**: FIFO ブロック ステータスの出力にフリップフロップを追加しない
  - F1 DYN – FIFO1 方向が動的か静的であるかを制御。静的モードでは、F1_SEL[1:0]ビットは FIFO 読み出しと書き込みアクセスを制御。このビットが静的モードのためにセットされた場合、2 つのコンフィギュレーションがある: FIFO がデータパスにより読み出されるおよび書き込まれることが可能な内部アクセス、および FIFO がシステム パスにより読み出されるおよび書き込まれることが可能な外部アクセス。このモードでは、F1_SEL[1:0]ビットは FIFO 書き込みソースを内部アクセス モードで制御
- **OFF**: 静的モードにある。FIFO 方向は静的で、F1_SEL[1:0]により制御される
- **ON**: 動的モード。FIFO 方向は動的で、DP 配線された信号 d1_load をトグルすることで内部か外部アクセスにより制御される
  - F0 DYN – F1 DYN の説明を参照してください。
  - F0 CLK INV および F1 CLK INV – FIFO クロックがデータパスのクロックを基準にして反転されるかを決定
- **NEG**: FIFO クロックは DP クロックと同じ極性
- **POS**: FIFO クロックは DP クロックに対して反転される
  - FIFO FAST – FIFO がデータパスのクロックか PSoC バス クロックいずれのクロックが供給されるかを決定。高速モードでは、FIFO はバス クロックによりクロック供給され、キャプチャ レイテンシを縮小
- このモードを使用すれば、バス クロックのマスターと象限ゲートを有効にする必要があるため、消費電力が少し上がる。このビットは、UDB 内の両方の FIFO のモードを制御するが、出力モードでコンフィギュレーションされた FIFO にのみ適用される
- **DP:** データパス クロック
- **BUS:** バス クロック
  - FIFO CAP – FIFO キャプチャ モードを有効にする。有効にされれば、A0 または A1 が読み込まれると、それぞれ F0 または F1 に書き込まれる
- **AX**: A0 または A1 を読み出すと、レジスタに値を直接返す
- **FX**: A0 (または A1) を読み出すと、F0 (または F1) へのキャプチャをトリガーする
  - FIFO EDGE – FIFO への書き込みが LOW から HIGH までの遷移の時に行われるかを決定。または、それらの書き込みがどんな時にも発生すれば、F0 または F1 ロード信号は HIGH となる
- **LEVEL**: (出力 モードでの) FIFO への書き込みはレベル センシティブ
- **EDGE**: (出力モードでの) FIFO への書き込みはエッジ センシティブ
  - FIFO ASYNC –フリップフロップが FIFO ブロック ステータス信号の出力において必要であるかを決定。次のページの「FIFO のコンフィギュレーション」節を参照してください。
- **SYNC**: FIFO ブロック ステータスの出力にフリップフロップを追加しない
- **ASYNC**: FIFO ブロック ステータスの出力にフリップフロップを追加しない。データパスクロックが MASTER_CLK に同期化された場合にのみ、ADD SYNC を *NONE* に、FIFO_ASYNC を *ASYNC* にまたは ADD SYNC を *ADD* に、FIFO ASYNC を *SYNC* に設定することが可能

FIFO のコンフィギュレーション

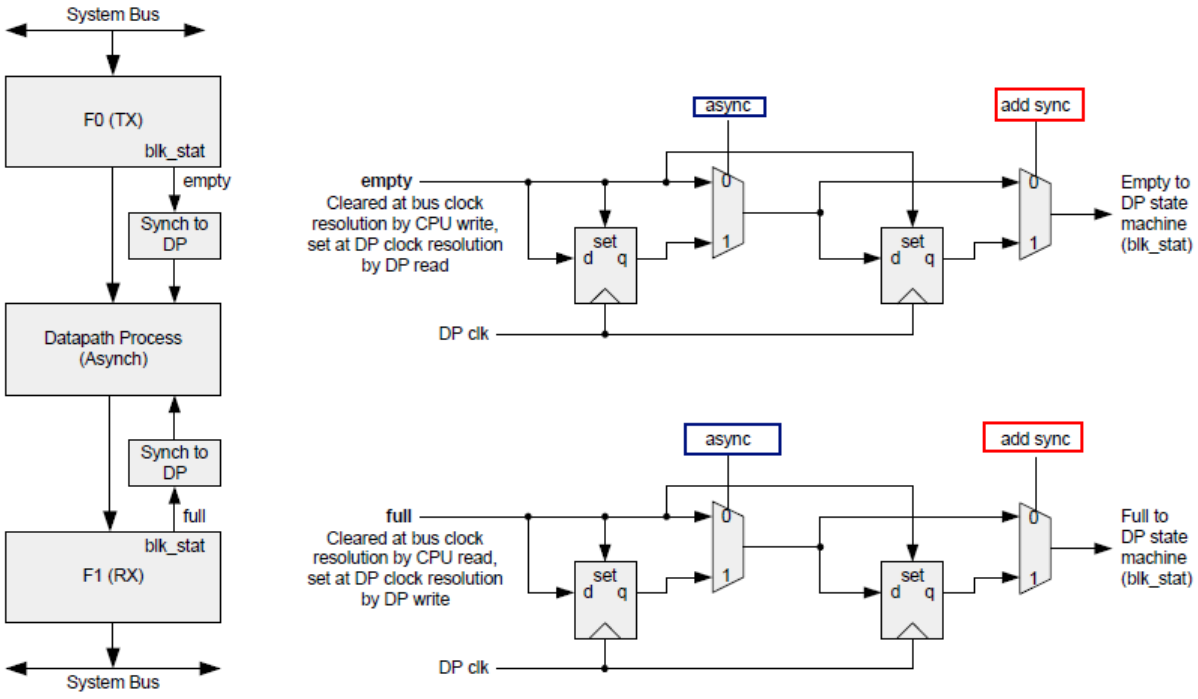
- EXT CRCPRS – CRC／PRS 計算のために内部コンフィギュレーションを上書きし、CRC／PRS 信号の外部ルーティングを許可。このビットがセットされた場合、CRC 演算のためにデータのシフトインデータとフィードバックデータを含む raw ブロック入力にアクセスが与えられ、これらの信号の計算は、外部から行われる必要がある (通常 PLD で)。

- **DSBL**: 内部 CRC／PRS ルーティング
- **ENBL**: 外部 CRC／PRS ルーティング

- WRK16 CONCAT – 16 ビット アクセス空間で動作レジスタ アクセス モードを制御。デフォルトでは、このビットが「0」である場合、アクセスがレジスタでチェーンの順序で UDB のペアごとに同じように発生。このビットが「1」の場合、16 ビット読み出しまたは書き込みは、単一の UDB 内で連結されたレジスタにアクセス。その組み合わせは{A1,A0}、{D1,D0}、{F1,F0}、{CTL,STAT}、{MSK, ACTL}、{8’b0,MC}

- **DSBL**: 16 ビットのデフォルト アクセス モード。16 ビットのアクセスは 2 つの連続の UDB 内の所与のレジスタをチェーン／アドレス順に読み出すまたは書き込む
- **ENBL**: 16 ビットの連結方式アクセスモード。16 ビット アクセスは、単一の UDB に連結されたレジスタを読み出すまたは書き込む

FIFO 出力同期化の図

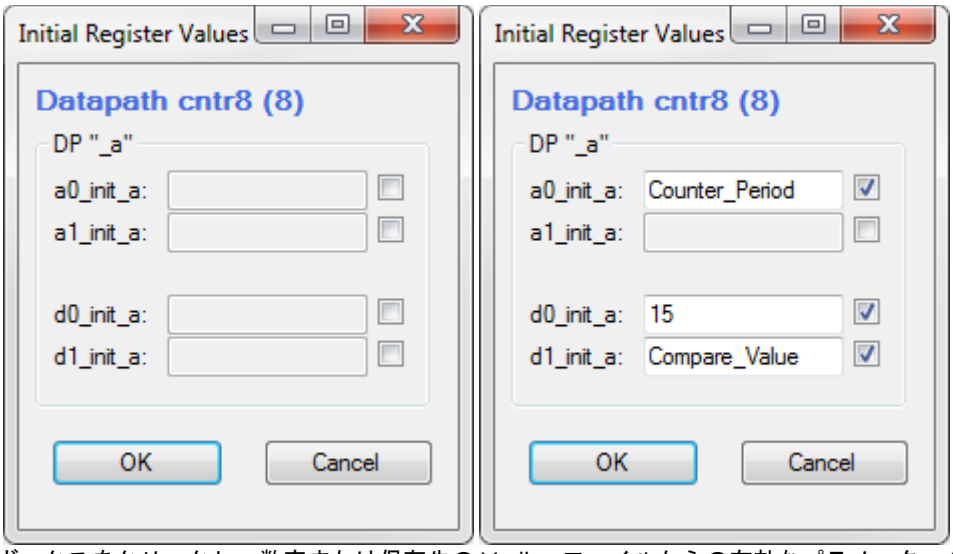


ASYNC	ADD SYNC	動作	使用モデル
0	0	バス クロックに同期	CPU 読み出し／書き込みステータスの変更は、バス クロックの分解能で発生。バス クロックのタイミングを満たすことが可能であれば、最小レイテンシのために使用することができる
0	1	バス クロックから DP クロックに再サンプルされる	これはデフォルトの同期動作モードである。CPU 読み出し／書き込みステータスの変更は現在選択されている DP クロックと同時に再サンプリングされる場合、UDB ロジックに DP クロックのセットアップ時間の完全なサイクルを与える
1	0	(冗長)	
1	1	バス クロックから DP クロックにダブルシンク	フリーランニング非同期 DP クロックが使用されている場合、この設定は DP クロックに CPU の読み書き動作をダブルシンクするために使用することが可能

B.3 初期レジスタ値の設定

A0、A1、D0、DCT で D1 の初期値を設定するには、**View** > **Initial Register Values** を開きます。例えば、データパス名が Cntr8 の場合、ウィンドウは、図 145 (左) のようになります。

図 145. Initial Register Values (レジスタの初期値)



チェックボックスをクリックし、数字または保存先の Verilog ファイルからの有効なパラメーター名のいずれかを入力することで、初期値を設定することができます (図 145 右)。

B.4 データパスチェーン接続

専用のデータパス チェーン接続信号により、チャネル ルーティング リソースを使用せず、単一サイクルの 16 ビット、24 ビット、32 ビットのビット機能の効率的な実装が可能です。

図 146 に示すように、すべての生成された条件付き信号とキャプチャ信号は、最下位から最上位への方向でチェーン接続されます。左シフト信号も最下位から最上位への方向でチェーン接続されます。右シフト信号は最上位から最下位への方向でチェーン接続されます。CFBO (フィードバック) の CRC／PRS チェーン接続信号は最下位から最上位にチェーン接続しますが、CMSBO (MSB 出力) は最上位から最下位にチェーン接続します。

図 146. データパスのチェーン信号フロー

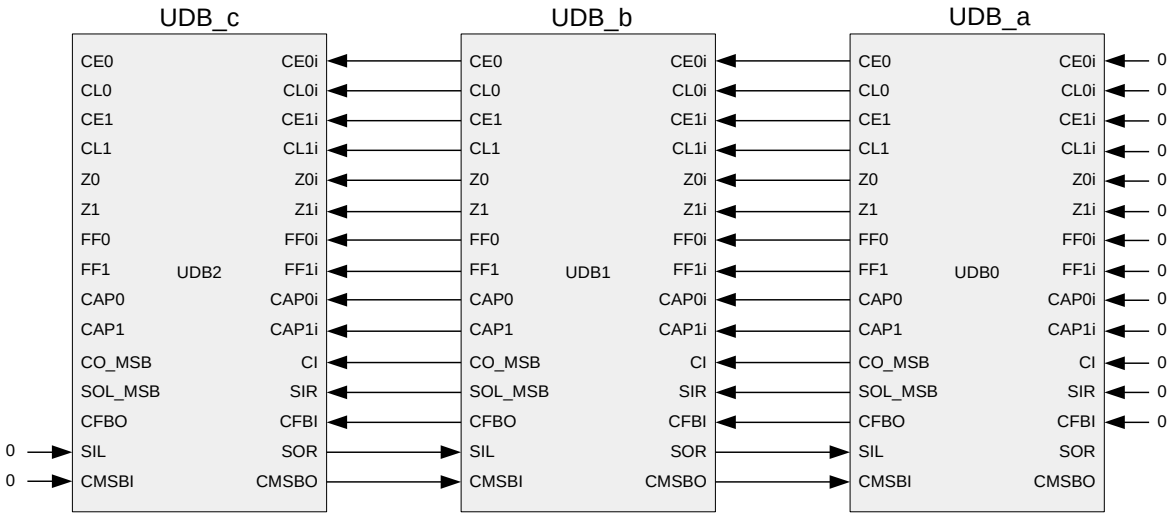
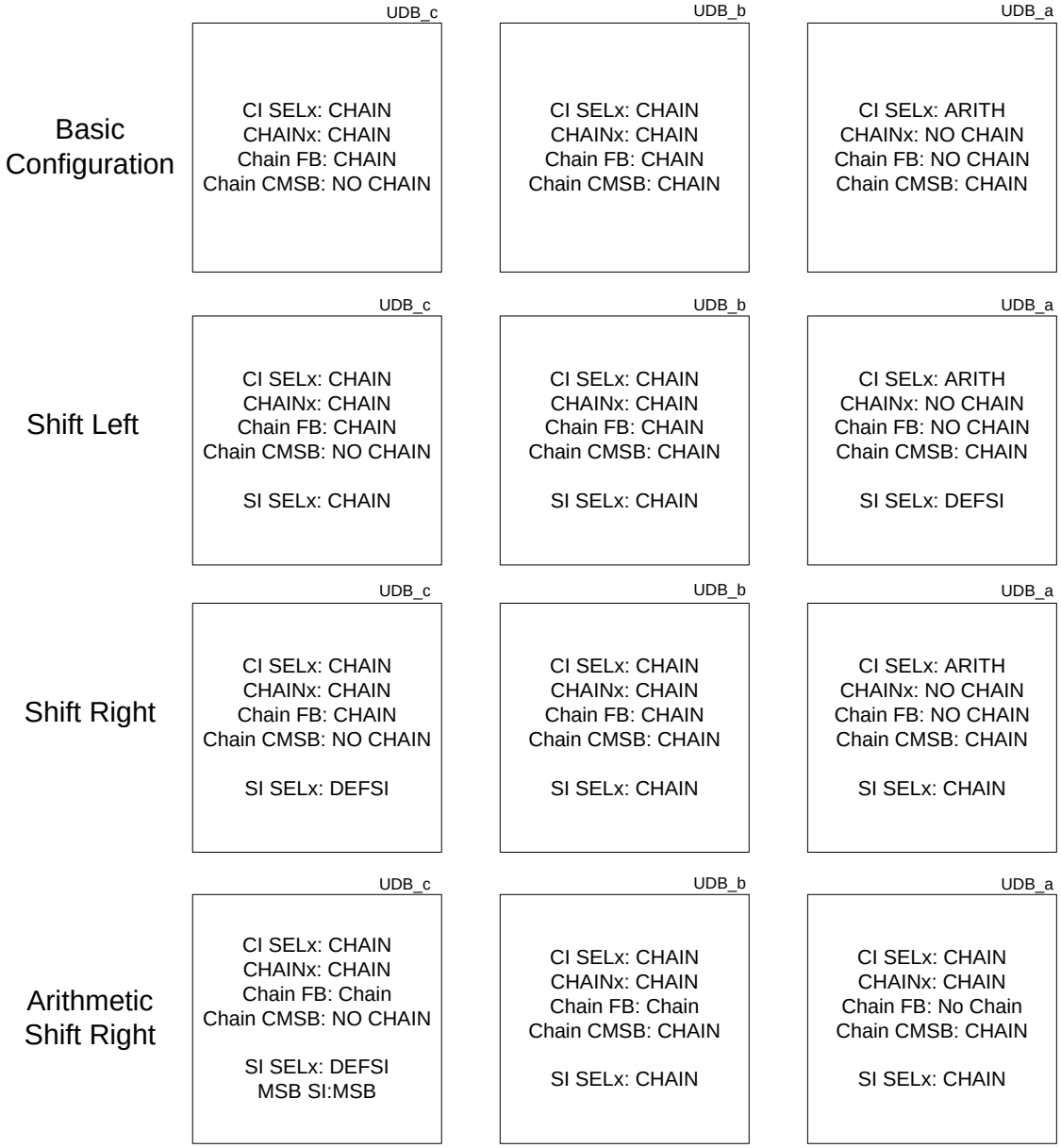


図 147 にさまざまな場合のデータパスをチェーン接続するために必要な設定を示しています。UDB_a は、最下位のブロックで、UDB_c は最上位ブロックです。図 147 に 3 UDB (最大 24 ビット) の機能を説明します；16 ビットまたは 32 ビット機能は真ん中のデータパス コンフィギュレーションを排除するまたは重複することで作成することができます。Chain FB と Chain CMSB が使用されないことがありますが、この図にはそれらのコンフィギュレーションを示します。

図 146 に示すように、データパスをチェーン接続する場合、デザインの大部分 (例えば、簡単な加算または減算) は図 147 の「Basic Configuration」行を使用して、Chain CMSB を除き LSB (UDB_a) から MSB (UDB_c) までの信号をすべてチェーン接続します。シフト処理を行う場合、シフトの方向に基づき、シフト チェーン コンフィギュレーションを変更する必要があります。図 147 内の Shift Left、Shift Right と Arithmetic Shift Right に示します。

図 147. チェーン接続のための DCT コンフィギュレーション



B.5 データパスレジスタのファームウェア制御

データパス レジスタはマクロ CY_SET_REG8 (アドレス、値) と CY_GET_REG8 (アドレス)、または場合によってこれらの機能の該当する 16、24、または 32 ビット バージョンを使用してファームウェア内でアクセスすることができます。
レジスタのアドレスは(成功したビルド後に生成される) *cyfitter.h* ファイルにあります。例えば、「cntr8」という 8 ビット データパス インスタンスが「SimpleCntr8_1」というコンポーネントでインスタンス化された場合、cyfitter.h ファイルはすべてのデータパス レジスタを一覧に表示するコードのブロックを含みます。図 148 を参照してください。

図 148. データパスレジスタのアドレス

```
/* SimpleCntr8_1 */
#define SimpleCntr8_1_cntr8_u0__16BIT_A0_REG CYREG_B1_UDB05_06_A0
#define SimpleCntr8_1_cntr8_u0__16BIT_A1_REG CYREG_B1_UDB05_06_A1
#define SimpleCntr8_1_cntr8_u0__16BIT_D0_REG CYREG_B1_UDB05_06_D0
#define SimpleCntr8_1_cntr8_u0__16BIT_D1_REG CYREG_B1_UDB05_06_D1
#define SimpleCntr8_1_cntr8_u0__16BIT_DP_AUX_CTL_REG CYREG_B1_UDB05_06_ACTL
#define SimpleCntr8_1_cntr8_u0__16BIT_F0_REG CYREG_B1_UDB05_06_F0
#define SimpleCntr8_1_cntr8_u0__16BIT_F1_REG CYREG_B1_UDB05_06_F1
#define SimpleCntr8_1_cntr8_u0__A0_A1_REG CYREG_B1_UDB05_A0_A1
#define SimpleCntr8_1_cntr8_u0__A0_REG CYREG_B1_UDB05_A0
#define SimpleCntr8_1_cntr8_u0__A1_REG CYREG_B1_UDB05_A1
#define SimpleCntr8_1_cntr8_u0__D0_D1_REG CYREG_B1_UDB05_D0_D1
#define SimpleCntr8_1_cntr8_u0__D0_REG CYREG_B1_UDB05_D0
#define SimpleCntr8_1_cntr8_u0__D1_REG CYREG_B1_UDB05_D1
#define SimpleCntr8_1_cntr8_u0__DP_AUX_CTL_REG CYREG_B1_UDB05_ACTL
#define SimpleCntr8_1_cntr8_u0__F0_F1_REG CYREG_B1_UDB05_F0_F1
#define SimpleCntr8_1_cntr8_u0__F0_REG CYREG_B1_UDB05_F0
#define SimpleCntr8_1_cntr8_u0__F1_REG CYREG_B1_UDB05_F1
```

B.6 その他

データパス コンフィギュレーション ツールの詳細は、Component Author Guide の付録 B を参照してください。この文書は **Help** > **Documentation** 下の DCT 内に、または **Help** > **Documentation** > **Component Author Guide** 内の PSoC Creator 内に用意されます。

C 付録 C – 強制データパス配置

データパスコンポーネントを特定の UDB に配置したい場合、その方法があります。ディレクティブタブの下の.cydwr ファイルで、ディレクティブを追加できます。それらのディレクティブの 1 つは ForceComponentUDB です。PSoC Creator ヘルプトピックファイルでディレクティブを検索すると、それらの使用方法に関するいくつかの指示があります。 以下はその例です。

\ `$INSTANCE_NAME`:DATAPATH0 ForceComponentUDB U(0,0)

`$INSTANCE_NAME`をあなたのコンポーネントの名前に置き換え、DATAPATH0 をあなたのコンポーネントの中のデータパスの名前に置き換える必要があります。

PSoC Creator トピックで PSoC UDB で PSoC UDB を検索する場合、U(0,0) は PSoC Creator ヘルプトピックファイルの UDB(行、列) を表し、UDB のマッピングが表示されます。行番号と列番号を参照してください。UDB を結ぶ線は、それらがどのように連鎖できるかを示しています。

チェーンに複数の UDB がある場合は、MSB データパスの配置を強制することしかできません。他のデータパスの配置を強制しようとすると、Creator がエラーを出します。

D 付録 D – 自動生成 Verilog コード

この付録は、PSoC Creator とデータパス コンフィギュレーション ツールにより生成される Verilog コードの例を提供します。以下のコードは、サンプルプロジェクト#1 のコンポーネントの作成中にコピーされたものです。このアプリケーションノートに関連する PSoC Creator プロジェクトは、フルコメントがあるサンプルプロジェクトを含みます。このコードは、PSoC Creator とデータパスコンフィギュレーションツールにより生成される Verilog コードの展開を示すためにのみ使用されます。

D.1 PSoC Creator により生成される新しい Verilog ファイル

Verilog ファイルは PSoC Creator により最初に生成された時、以下のようになります。

```
//`#start header` -- edit after this line, do not edit this line
// ========================================
//
// Copyright YOUR COMPANY, THE YEAR
// All Rights Reserved
// UNPUBLISHED, LICENSED SOFTWARE.
//
// CONFIDENTIAL AND PROPRIETARY INFORMATION
// WHICH IS THE PROPERTY OF your company.
//
// ========================================
`include "cypress.v"
//`#end` -- edit above this line, do not edit this line
// Generated on 09/17/2012 at 11:02
// Component: UpDwnCntrPwm_v1_0
module SimpleCntr8_v1_0 (
       output  tc,
       input   clk
);

//`#start body` -- edit after this line, do not edit this line

//        Your code goes here

//`#end` -- edit above this line, do not edit this line
endmodule
//`#start footer` -- edit after this line, do not edit this line
//`#end` -- edit above this line, do not edit this line
```

この場合、tc と clk ピンは、SimpleCntr8 コンポーネントシンボルにあるため、追加されます。もし自身のコンポーネントシンボルが何の端末も含まなければ、これらのコード行は省略されることになります。

D.2 新しいデータパスインスタンスのある Verilog ファイル

データパス生成ツールがデータパスコンフィギュレーション用にコードを生成する時、このコードはコンポーネントの既存の Verilog ファイルに追加されます。変更されていないコンフィギュレーションは以下のようになります。

```
//`#start header` -- edit after this line, do not edit this line
// ========================================
//
// Copyright YOUR COMPANY, THE YEAR
// All Rights Reserved
// UNPUBLISHED, LICENSED SOFTWARE.
//
// CONFIDENTIAL AND PROPRIETARY INFORMATION
// WHICH IS THE PROPERTY OF your company.
//
// ========================================
`include "cypress.v"
//`#end` -- edit above this line, do not edit this line
```

```
// Generated on 09/17/2012 at 11:02
// Component: UpDwnCntrPwm_v1_0
module SimpleCntr8_v1_0 (
       output  tc,
       input   clk
);

//`#start body` -- edit after this line, do not edit this line

//        Your code goes here

//`#end` -- edit above this line, do not edit this line
cy_psoc3_dp8 #(.cy_dpconfig_a(
{
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM0: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM1: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM2: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM3: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM4: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM5: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM6: */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM7: */
   8'hFF, 8'h00,  /*CFG9: */
   8'hFF, 8'hFF,  /*CFG11-10: */
    `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
    `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
    `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
    `SC_SI_A_DEFSI, /*CFG13-12: */
    `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'h0,
   1'h0, `SC_FIFO1_BUS, `SC_FIFO0_BUS,
    `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
    `SC_FB_NOCHN, `SC_CMP1_NOCHN,
    `SC_CMP0_NOCHN, /*CFG15-14: */
   10'h00, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
    `SC_FIFO_LEVEL,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
    `SC_WRK16CAT_DSBL /*CFG17-16: */
}
```

```
)) cntr8(
        /*  input                   */  .reset(1'b0),
        /*  input                   */  .clk(1'b0),
        /*  input   [02:00]         */  .cs_addr(3'b0),
        /*  input                   */  .route_si(1'b0),
        /*  input                   */  .route_ci(1'b0),
        /*  input                   */  .f0_load(1'b0),
        /*  input                   */  .f1_load(1'b0),
        /*  input                   */  .d0_load(1'b0),
        /*  input                   */  .d1_load(1'b0),
        /*  output                  */  .ce0(),
        /*  output                  */  .cl0(),
        /*  output                  */  .z0(),
        /*  output                  */  .ff0(),
        /*  output                  */  .ce1(),
        /*  output                  */  .cl1(),
        /*  output                  */  .z1(),
        /*  output                  */  .ff1(),
        /*  output                  */  .ov_msb(),
        /*  output                  */  .co_msb(),
        /*  output                  */  .cmsb(),
        /*  output                  */  .so(),
        /*  output                  */  .f0_bus_stat(),
        /*  output                  */  .f0_blk_stat(),
        /*  output                  */  .f1_bus_stat(),
        /*  output                  */  .f1_blk_stat()
);
endmodule
//`#start footer` -- edit after this line, do not edit this line
//`#end` -- edit above this line, do not edit this line
```

コンフィギュレーション RAM とハードウェア リンクのセクションが追加されますが、それらは実際の設定データが含まれていないことに注意してください。このデータパスコンポーネントは役に立ちません。表 22 は、データパスコンフィギュレーションでのハードウェア リンクを説明します。

表 22. Verilog ハードウェアリンクの定義

Verilog コード	定義 / マッピング
.reset(1'b0),	データパス リセット信号
.clk(1'b0),	データパス クロック入力
.cs_addr(3'b0),	コンフィギュレーション ストア アドレス信号
.route_si(1'b0),	配線されるシフトイン
.route_ci(1'b0),	配線されるキャリーイン
.f0_load(1'b0),	FIFO 0 ロード信号
.f1_load(1'b0),	FIFO 1 ロード信号
.d0_load(1'b0),	D0 ロード信号
.d1_load(1'b0),	D1 ロード信号
.ce0(),	比較ブロック 0 の「equals (等しい)」の出力
.cl0(),	比較ブロック 0 の「less than (より小)」の出力
.z0(),	ブロック 0 の 0 検出の出力
.ff0(),	ブロック 0 のすべて 1 検出の出力

Verilog コード	定義 / マッピング
.ce1(),	比較ブロック 1 の「equals (等しい)」の出力
.cl1(),	比較ブロック 1 の「less than (より小)」の出力
.z1(),	ブロック 1 の 0 検出の出力
.ff1(),	ブロック 1 のすべて 1 検出の出力
.ov_msb(),	オーバーフロー検出
.co_msb(),	キャリーアウト
.cmsb(),	この AN で記述しない
.so(),	シフトアウト
.f0_bus_stat(),	FIFO 0 バス ステータス フラグ*
.f0_blk_stat(),	FIFO 0 ブロック ステータス フラグ*
.f1_bus_stat(),	FIFO 1 バス ステータス フラグ*
.f1_blk_stat()	FIFO 1 ブロック ステータス フラグ*

*これらのステータス出力の詳細については、TRM を参照してください。

データパスが機能できるように、これらのリンクが必要となります。それらのリンクがなければ、コンポーネントシンボル、Verilog コードおよびデータパスハードウェアの間には相関関係はありません。

D.3 すべての SimpleCntr8 変更後の Verilog ファイル

以下は、簡単な 8 ビットカウンターの完了した Verilog です。

```
//`#start header` -- edit after this line, do not edit this line
// ========================================
//
// Copyright YOUR COMPANY, THE YEAR
// All Rights Reserved
// UNPUBLISHED, LICENSED SOFTWARE.
//
// CONFIDENTIAL AND PROPRIETARY INFORMATION
// WHICH IS THE PROPERTY OF your company.
//
// ========================================
`include "cypress.v"
//`#end` -- edit above this line, do not edit this line
// Generated on 09/17/2012 at 11:02
// Component: UpDwnCntrPwm_v1_0
module SimpleCntr8_v1_0 (
       output  tc,
       input   clk
);

//`#start body` -- edit after this line, do not edit this line

//        Your code goes here

//`#end` -- edit above this line, do not edit this line
cy_psoc3_dp8 #(.a0_init_a(8'b00001111), .d0_init_a(8'b00001111),

.cy_dpconfig_a(
{
    `CS_ALU_OP__DEC, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC__ALU, `CS_A1_SRC_NONE,
```

```
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM0:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC___D0, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM1:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM2:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM3:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM4:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM5:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM6:    */
    `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
    `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
    `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
    `CS_CMP_SEL_CFGA, /*CFGRAM7:    */
   8'hFF, 8'h00,  /*CFG9:     */
   8'hFF, 8'hFF,  /*CFG11-10:     */
    `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
    `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
    `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
    `SC_SI_A_DEFSI, /*CFG13-12:     */
    `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'h0,
   1'h0, `SC_FIFO1_BUS, `SC_FIFO0_BUS,
    `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
    `SC_FB_NOCHN, `SC_CMP1_NOCHN,
    `SC_CMP0_NOCHN, /*CFG15-14:     */
   10'h00, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
    `SC_FIFO_LEVEL,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
    `SC_WRK16CAT_DSBL /*CFG17-16:     */
}
)) cntr8(
        /*  input                   */  .reset(1'b0),
        /*  input                   */  .clk(clk), /* tie clk signal to datapath clock */
        /*  input   [02:00]         */  .cs_addr({2'b0,tc}), /* tc determines address lsb */
        /*  input                   */  .route_si(1'b0),
        /*  input                   */  .route_ci(1'b0),
        /*  input                   */  .f0_load(1'b0),
        /*  input                   */  .f1_load(1'b0),
        /*  input                   */  .d0_load(1'b0),
        /*  input                   */  .d1_load(1'b0),
        /*  output                  */  .ce0(),
        /*  output                  */  .cl0(),
        /*  output                  */  .z0(tc), /* tc signal comes from z0 state */
        /*  output                  */  .ff0(),
        /*  output                  */  .ce1(),
        /*  output                  */  .cl1(),
```

```
        /*  output                  */  .z1(),
        /*  output                  */  .ff1(),
        /*  output                  */  .ov_msb(),
        /*  output                  */  .co_msb(),
        /*  output                  */  .cmsb(),
        /*  output                  */  .so(),
        /*  output                  */  .f0_bus_stat(),
        /*  output                  */  .f0_blk_stat(),
        /*  output                  */  .f1_bus_stat(),
        /*  output                  */  .f1_blk_stat()
);
endmodule
//`#start footer` -- edit after this line, do not edit this line
//`#end` -- edit above this line, do not edit this line
```

A0 をデクリメントし、リロードする 2 つのコンフィギュレーションが設定され、A0 と D0 用の初期レジスタ値が定義されました。ハードウェアとシンボル端末間のリンクも設立されました。ゼロからこのファイルを作成することができますが、データパスコンフィギュレーションツールを使用すれば、はるかに簡単になります。

E 付録 E – PI と PO を持つ 24 ビットデータパスの例

```
// Requires these signals to tie the 3 datapaths together
      wire [14:0] chain0;
      wire [14:0] chain1;

// Datapath 0
cy_psoc3_dp #(.cy_dpconfig(
{
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG0 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG1 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG2 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG3 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG4 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG5 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG6 Comment: */
      `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
      `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
      `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
      `CS_CMP_SEL_CFGA, /*CS_REG7 Comment: */
        8'hFF, 8'h00,      /*SC_REG4     Comment: */
        8'hFF, 8'hFF,      /*SC_REG5     Comment: */
      `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
      `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
      `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
      `SC_SI_A_DEFSI, /*SC_REG6 Comment: */
      `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'b0,
      1'b0, `SC_FIFO1_BUS, `SC_FIFO0__A0,
      `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
      `SC_FB_NOCHN, `SC_CMP1_NOCHN,
      `SC_CMP0_NOCHN, /*SC_REG7 Comment: */
       10'h0, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
      `SC_FIFO_LEVEL,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
      `SC_WRK16CAT_DSBL /*SC_REG8 Comment: */
})) Datapath0(
      /* input */ .clk(),                 // Clock
      /* input [02:00] */ .cs_addr(),     // Control Store RAM address
      /* input */ .route_si(1'b0),        // Shift in from routing
      /* input */ .route_ci(1'b0),        // Carry in from routing
      /* input */ .f0_load(1'b0),         // Load FIFO 0
```

```
        /* input */ .f1_load(1'b0),        // Load FIFO 1
        /* input */ .d0_load(1'b0),        // Load Data Register 0
        /* input */ .d1_load(1'b0),        // Load Data Register 1
        /* output */ .ce0(),               // Accumulator 0 = Data register 0
        /* output */ .cl0(),               // Accumulator 0 < Data register 0
        /* output */ .z0(),                // Accumulator 0 = 0
        /* output */ .ff0(),               // Accumulator 0 = FF
        /* output */ .ce1(),               // Accumulator [0|1] = Data register 1
        /* output */ .cl1(),               // Accumulator [0|1] < Data register 1
        /* output */ .z1(),                // Accumulator 1 = 0
        /* output */ .ff1(),               // Accumulator 1 = FF
        /* output */ .ov_msb(),            // Operation over flow
        /* output */ .co_msb(),            // Carry out
        /* output */ .cmsb(),              // Carry out
        /* output */ .so(),                // Shift out
        /* output */ .f0_bus_stat(),       // FIFO 0 status to uP
        /* output */ .f0_blk_stat(),       // FIFO 0 status to DP
        /* output */ .f1_bus_stat(),       // FIFO 1 status to uP
        /* output */ .f1_blk_stat(),       // FIFO 1 status to DP
        /* input */ .ci(1'b0),             // Carry in from previous stage
        /* output */ .co(chain0[12]),      // Carry out to next stage
        /* input */ .sir(1'b0),            // Shift in from right side
        /* output */ .sor(),               // Shift out to right side
        /* input */ .sil(chain0[10]),      // Shift in from left side
        /* output */ .sol(chain0[11]),     // Shift out to left side
        /* input */ .msbi(chain0[9]),      // MSB chain in
        /* output */ .msbo(),              // MSB chain out
        /* input [01:00] */ .cei(2'b0),    // Compare equal in from prev stage
        /* output [01:00] */ .ceo(chain0[1:0]), // Compare equal out to next stage
        /* input [01:00] */ .cli(2'b0),    // Compare less than in from prv stage
        /* output [01:00] */ .clo(chain0[3:2]), // Compare less than out to next stage
        /* input [01:00] */ .zi(2'b0),           // Zero detect in from previous stage
        /* output [01:00] */ .zo(chain0[5:4]),   // Zero detect out to next stage
        /* input [01:00] */ .fi(2'b0),           // 0xFF detect in from previous stage
        /* output [01:00] */ .fo(chain0[7:6]),   // 0xFF detect out to next stage
        /* input [01:00] */ .capi(2'b0),         // Capture in from previous stage
        /* output [01:00] */ .capo(chain0[14:13]),     // Capture out to next stage
        /* input */ .cfbi(1'b0),                 // CRC Feedback in from previous stage
        /* output */ .cfbo(chain0[8]),           // CRC Feedback out to next stage
        /* input [07:00] */ .pi(),               // Parallel data port
        /* output [07:00] */ .po()         // Parallel data port
);

// Datapath 1
cy_psoc3_dp #(.cy_dpconfig(
{
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG0 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG1 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG2 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
```

```
        `CS_CMP_SEL_CFGA, /*CS_REG3 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG4 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG5 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG6 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG7 Comment: */
          8'hFF, 8'h00,      /*SC_REG4     Comment: */
          8'hFF, 8'hFF,      /*SC_REG5     Comment: */
        `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
        `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
        `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
        `SC_SI_A_DEFSI, /*SC_REG6 Comment: */
        `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'b0,
       1'b0, `SC_FIFO1_BUS, `SC_FIFO0__A0,
        `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
        `SC_FB_NOCHN, `SC_CMP1_NOCHN,
        `SC_CMP0_NOCHN, /*SC_REG7 Comment: */
         10'h0, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
        `SC_FIFO_LEVEL,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
        `SC_WRK16CAT_DSBL /*SC_REG8 Comment: */
}}) Datapath1(
        /* input */ .clk(),                 // Clock
        /* input [02:00] */ .cs_addr(),     // Control Store RAM address
        /* input */ .route_si(1'b0),        // Shift in from routing
        /* input */ .route_ci(1'b0),        // Carry in from routing
        /* input */ .f0_load(1'b0),         // Load FIFO 0
        /* input */ .f1_load(1'b0),         // Load FIFO 1
        /* input */ .d0_load(1'b0),         // Load Data Register 0
        /* input */ .d1_load(1'b0),         // Load Data Register 1
        /* output */ .ce0(),                // Accumulator 0 = Data register 0
        /* output */ .cl0(),                // Accumulator 0 < Data register 0
        /* output */ .z0(),                 // Accumulator 0 = 0
        /* output */ .ff0(),                // Accumulator 0 = FF
        /* output */ .ce1(),                // Accumulator [0|1] = Data register 1
        /* output */ .cl1(),                // Accumulator [0|1] < Data register 1
        /* output */ .z1(),                 // Accumulator 1 = 0
        /* output */ .ff1(),                // Accumulator 1 = FF
        /* output */ .ov_msb(),             // Operation over flow
        /* output */ .co_msb(),             // Carry out
        /* output */ .cmsb(),               // Carry out
        /* output */ .so(),                 // Shift out
        /* output */ .f0_bus_stat(),        // FIFO 0 status to uP
        /* output */ .f0_blk_stat(),        // FIFO 0 status to DP
        /* output */ .f1_bus_stat(),        // FIFO 1 status to uP
        /* output */ .f1_blk_stat(),        // FIFO 1 status to DP
        /* input */ .ci(chain0[12]),        // Carry in from previous stage
        /* output */ .co(chain1[12]),       // Carry out to next stage
        /* input */ .sir(chain0[11]),       // Shift in from right side
        /* output */ .sor(chain0[10]),      // Shift out to right side
        /* input */ .sil(chain1[10]),       // Shift in from left side
```

```
        /* output */ .sol(chain1[11]),    // Shift out to left side
        /* input */ .msbi(chain1[9]),     // MSB chain in
        /* output */ .msbo(chain0[9]),    // MSB chain out
        /* input [01:00] */ .cei(chain0[1:0]),  // Compare equal in from prev stage
        /* output [01:00] */ .ceo(chain1[1:0]), // Compare equal out to next stage
        /* input [01:00] */ .cli(chain0[3:2]),  // Compare less than in from prv stage
        /* output [01:00] */ .clo(chain1[3:2]), // Compare less than out to next stage
        /* input [01:00] */ .zi(chain0[5:4]),   // Zero detect in from previous stage
        /* output [01:00] */ .zo(chain1[5:4]),  // Zero detect out to next stage
        /* input [01:00] */ .fi(chain0[7:6]),   // 0xFF detect in from previous stage
        /* output [01:00] */ .fo(chain1[7:6]),  // 0xFF detect out to next stage
        /* input [01:00] */ .capi(chain0[14:13]),// Capture in from previous stage
        /* output [01:00] */ .capo(chain1[14:13]),     // Capture out to next stage
        /* input */ .cfbi(chain0[8]),           // CRC Feedback in from previous stage
        /* output */ .cfbo(chain1[8]),          // CRC Feedback out to next stage
        /* input [07:00] */ .pi(),              // Parallel data port
        /* output [07:00] */ .po()              // Parallel data port
);

// Datapath 2
cy_psoc3_dp #(.cy_dpconfig(
{
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG0 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG1 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG2 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG3 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG4 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG5 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG6 Comment: */
        `CS_ALU_OP_PASS, `CS_SRCA_A0, `CS_SRCB_D0,
        `CS_SHFT_OP_PASS, `CS_A0_SRC_NONE, `CS_A1_SRC_NONE,
        `CS_FEEDBACK_DSBL, `CS_CI_SEL_CFGA, `CS_SI_SEL_CFGA,
        `CS_CMP_SEL_CFGA, /*CS_REG7 Comment: */
          8'hFF, 8'h00,      /*SC_REG4     Comment: */
          8'hFF, 8'hFF,      /*SC_REG5     Comment: */
        `SC_CMPB_A1_D1, `SC_CMPA_A1_D1, `SC_CI_B_ARITH,
        `SC_CI_A_ARITH, `SC_C1_MASK_DSBL, `SC_C0_MASK_DSBL,
        `SC_A_MASK_DSBL, `SC_DEF_SI_0, `SC_SI_B_DEFSI,
        `SC_SI_A_DEFSI, /*SC_REG6 Comment: */
        `SC_A0_SRC_ACC, `SC_SHIFT_SL, 1'b0,
```

```
        1'b0, `SC_FIFO1_BUS, `SC_FIFO0__A0,
        `SC_MSB_DSBL, `SC_MSB_BIT0, `SC_MSB_NOCHN,
        `SC_FB_NOCHN, `SC_CMP1_NOCHN,
        `SC_CMP0_NOCHN, /*SC_REG7 Comment: */
         10'h0, `SC_FIFO_CLK__DP,`SC_FIFO_CAP_AX,
        `SC_FIFO_LEVEL,`SC_FIFO__SYNC,`SC_EXTCRC_DSBL,
        `SC_WRK16CAT_DSBL /*SC_REG8 Comment: */
}}) Datapath2(
        /* input */ .clk(),                    // Clock
        /* input [02:00] */ .cs_addr(),        // Control Store RAM address
        /* input */ .route_si(1'b0),           // Shift in from routing
        /* input */ .route_ci(1'b0),           // Carry in from routing
        /* input */ .f0_load(1'b0),            // Load FIFO 0
        /* input */ .f1_load(1'b0),            // Load FIFO 1
        /* input */ .d0_load(1'b0),            // Load Data Register 0
        /* input */ .d1_load(1'b0),            // Load Data Register 1
        /* output */ .ce0(),                   // Accumulator 0 = Data register 0
        /* output */ .cl0(),                   // Accumulator 0 < Data register 0
        /* output */ .z0(),                    // Accumulator 0 = 0
        /* output */ .ff0(),                   // Accumulator 0 = FF
        /* output */ .ce1(),                   // Accumulator [0|1] = Data register 1
        /* output */ .cl1(),                   // Accumulator [0|1] < Data register 1
        /* output */ .z1(),                    // Accumulator 1 = 0
        /* output */ .ff1(),                   // Accumulator 1 = FF
        /* output */ .ov_msb(),                // Operation over flow
        /* output */ .co_msb(),                // Carry out
        /* output */ .cmsb(),                  // Carry out
        /* output */ .so(),                    // Shift out
        /* output */ .f0_bus_stat(),           // FIFO 0 status to uP
        /* output */ .f0_blk_stat(),           // FIFO 0 status to DP
        /* output */ .f1_bus_stat(),           // FIFO 1 status to uP
        /* output */ .f1_blk_stat(),           // FIFO 1 status to DP
        /* input */ .ci(chain1[12]),           // Carry in from previous stage
        /* output */ .co(),                    // Carry out to next stage
        /* input */ .sir(chain1[11]),          // Shift in from right side
        /* output */ .sor(chain1[10]),         // Shift out to right side
        /* input */ .sil(1'b0),                // Shift in from left side
        /* output */ .sol(),                   // Shift out to left side
        /* input */ .msbi(1'b0),               // MSB chain in
        /* output */ .msbo(chain1[9]),         // MSB chain out
        /* input [01:00] */ .cei(chain1[1:0]), // Compare equal in from prev stage
        /* output [01:00] */ .ceo(),           // Compare equal out to next stage
        /* input [01:00] */ .cli(chain1[3:2]), // Compare less than in from prv stage
        /* output [01:00] */ .clo(),           // Compare less than out to next stage
        /* input [01:00] */ .zi(chain1[5:4]),  // Zero detect in from previous stage
        /* output [01:00] */ .zo(),            // Zero detect out to next stage
        /* input [01:00] */ .fi(chain1[7:6]),  // 0xFF detect in from previous stage
        /* output [01:00] */ .fo(),            // 0xFF detect out to next stage
        /* input [01:00] */ .capi(chain1[14:13]),// Capture in from previous stage
        /* output [01:00] */ .capo(),          // Capture out to next stage
        /* input */ .cfbi(chain1[8]),          // CRC Feedback in from previous stage
        /* output */ .cfbo(),                  // CRC Feedback out to next stage
        /* input [07:00] */ .pi(),             // Parallel data port
        /* output [07:00] */ .po()             // Parallel data port
```

改訂履歴

文書名: AN82156 – UDB データパスを用いた PSoC Creator コンポーネントの設計

文書番号: 001-89531

版	ECN	発行日	変更内容
**	4145805	10/03/2013	これは英語版 001-82156 Rev. *C を翻訳した日本語版 001-89531 Rev. **です。
*A	4722749	05/04/2015	これは英語版 001-82156 Rev. *E を翻訳した日本語版 001-89531 Rev. *A です。
*B	5473709	10/04/2016	これは英語版 001-82156 Rev. *G を翻訳した日本語版 001-89531 Rev. *B です。 新しいテンプレートに更新しました。 サンセットレビューを完了する。
*C	6657103	08/21/2019	これは英語版 001-82156 Rev. *I を翻訳した日本語版 001-89531 Rev. *C です。

ワールドワイドな販売と設計サポート

サイプレスは、事業所、ソリューション センター、メーカー代理店、および販売代理店の世界的なネットワークを保持しています。お客様の最寄りのオフィスについては、サイプレスのロケーション ページをご覧ください。

製品

Arm® Cortex® Microcontrollers	cypress.com/arm
車載用	cypress.com/automotive
クロック&バッファ	cypress.com/clocks
インターフェース	cypress.com/interface
IoT (モノのインターネット)	cypress.com/iot
メモリ	cypress.com/memory
マイクロコントローラ	cypress.com/mcu
PSoC	cypress.com/psoc
電源用 IC	cypress.com/pmic
タッチ センシング	cypress.com/touch
USB コントローラー	cypress.com/usb
ワイヤレス	cypress.com/wireless

PSoC® ソリューション

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

サイプレス開発者コミュニティ

コミュニティ | Projects | ビデオ | ブログ | トレーニング | Components

テクニカルサポート

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709