XAPP202、2000 年 2 月 28 日（バージョン 1.2）

# ATM アプリケーション CAM （Content Addressable Memory）

著者 : Marc Defossez

## 概要

CAM（Content Addressable Memory）は、その内容によってアドレスできる記憶デバイスです。CAM 記憶素子の各ビットには、比較ロジックが含まれています。CAM に入力されるデータ値は、記憶されているすべてのデータと同時に比較され、対応するアドレスが結果として出力されます。CAM は、データ並列プロセッサとして動作します。CAM は、非同期転送モード（ATM）スイッチの設計に使用できます。このアプリケーション ノートでは、ATM アプリケーションにおける CAM のインプリメントを中心に説明します。他のデザインで CAM をインプリメントするさまざまなアプローチについては、アプリケーション ノート XAPP201『Virtex デバイスにおける各種の CAM デザインの概要』を参照してください。

## はじめに

CAM は、データベース、リスト、パターンなどを高速に検索する必要があるアプリケーションで使用されるメモリ デバイスです。イメージあるいは音声システム、コンピュータ、および通信システムはすべて、CAM を使用します。CAM は、他のメモリ検索アルゴリズムよりも高速です。これは、以前に格納されたエントリのリスト全体に対して、目的の情報を同時に比較するためです。CAM は、RAM 技術が発展して生まれた技術です。

XAPP201 では、CAM ブロックと RAM ブロックの概要について説明しています。また、Virtex デバイスで CAM を設計する 3 種類のアプローチも比較しています。このアプリケーション ノート（XAPP202）では、ATM デザインのための大規模な CAM アプローチに焦点を絞ります。

### ATM の CAM

ATM スイッチは接続ベースのプロトコルなので、配線パス上のすべての点で各 ATM セル アドレスを変換する必要があります。図 1 に示すように、各 ATM セル アドレスは 2 つのフィールドの 5 バイトのヘッダに格納されます。仮想パス識別子（VPI）は、8 ～ 12 ビット幅です。通常は 12 ビット ワードで記述されます。仮想回路識別子（VCI）は、16 ビット幅です。

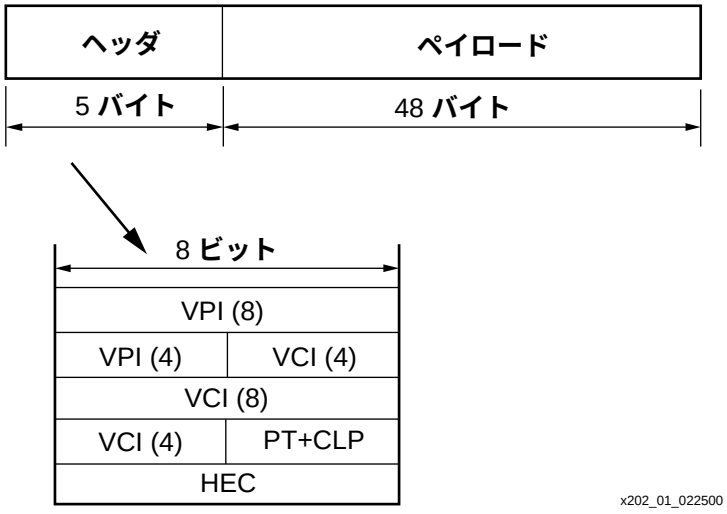


**図 1: ATM セル アドレス**

268 億までの値を設定できるアドレス可能な容量は、28 ビットの VPI/VCI によって定義されています。このすべての値を短時間でデコードするのは不可能です。しかし、スイッチ内で一度にアクティブになるのは数千種類の値だけです。スイッチは、図 2 に示すように、発信 VPI、発信 VCI、および着信セルに対応するポート値のテーブルをメモリ内に維持できます。

たとえば、スイッチ内で 4000 のアクティブな接続が必要な場合、VPI と VCI の値は 28 ビットの CAM で 4096 ワードに格納できます。生成される 12 ビットのアドレス出力は、変換された VPI と VCI が格納される RAM テーブルのインデックスとして使用できます。


**図 2: 発信 VPI、発信 VCI、出力ポートのテーブル**

## Virtex デバイスによる ATM 用 CAM

Virtex シリーズのアーキテクチャは、内部に分散 RAM およびブロック RAM という 2 種類の RAM を持っています。また、Virtex デバイスでは、Virtex SelectI/O 機能を使用して、一般的な RAM の速度で外部 RAM にアクセスできます。この CAM インプリメンテーションでは、VPI と VCI のテーブルにルックアップ テーブル（LUT）から作成した分散 RAM を使用し、RAM データ テーブルにブロック RAM または外部 RAM コンポーネントを使用します。

### CAM のビット テーブル

このデザイン例では、VPI が 12 ビット、VCI が 16 ビットを使用するので、28 ビット幅の検索テーブルが必要です。CAM の接続リストは、4096 ワード以上の長さになります。このため、RAM のサイズとして 28 × 4096 ワードが必要です。Virtex アーキテクチャは、分散 RAM およびブロック RAM という 2 種類のメモリを持っています。分散 SelectRAM+ メモリは、CLB 内にある 4 個の 4 入力 LUT を使用して作成されます。これは、1 ビット× 16 ワードの RAM としてコンフィギュレーションできます。Block SelectRAM+ メモリは、1 ビット× 4096 ワード、2

ビット× 2048 ワード、4 ビット× 1024 ワード、8 ビット× 512 ワード、または 16 ビット× 256 ワードの RAM としてコンフィギュレーションできる埋め込み型 RAM ブロックです。

このデザインに示されているアプローチでは、分散 RAM を使用して CAM を作成します。1 ビット× 4096 ワードの RAM のインプリメンテーションには、256 個の RAM が必要です。このアプリケーションでは 28 ビット× 4096 ワードが必要なので、7168 個の分散 RAM が必要になります。これらの条件を満たすのは、XCV400 デバイスです。XCV400 では 20 個の ブロック RAM を使用できるので、出力テーブルとして使用できます。

分散 RAM で比較テーブルを作成すると、CLB 内にある他のロジック（キャリー チェーン、マルチプレクサ、フリップフロップなど）を引き続き使用できます。Virtex ファミリの CLB に関する情報は、Virtex のデータシートに記載されています。このアプリケーション ノートに記されている CAM 全体を作成する際に必要な LUT の数は、1.6 × 7168 = 11469 です。

比較テーブルは、データで初期化する必要があります。これを実行するには、次の 3 つの方法があります。

- コンフィギュレーション時に、分散 RAM の INIT パラメータを使用して初期化する。
- 連続するデータのリスト（ブロック RAM）に書き込む。
- 上記 2 つの方法を組み合わせる。

リストを作成する方法に関係なく、INIT パラメータは常に使用できます。連続するリストで RAM テーブルを作成すると、リストが大きくなったときに検索の時間が長くなります。これは、4096 ワードのテーブルでもかなりの時間になります。このような場合は、分散 RAM によるアプローチの方が有効です。図 3 のテーブルは、初期化用の連続リストと比較用の 16 エントリの小さい部分から作成されています。

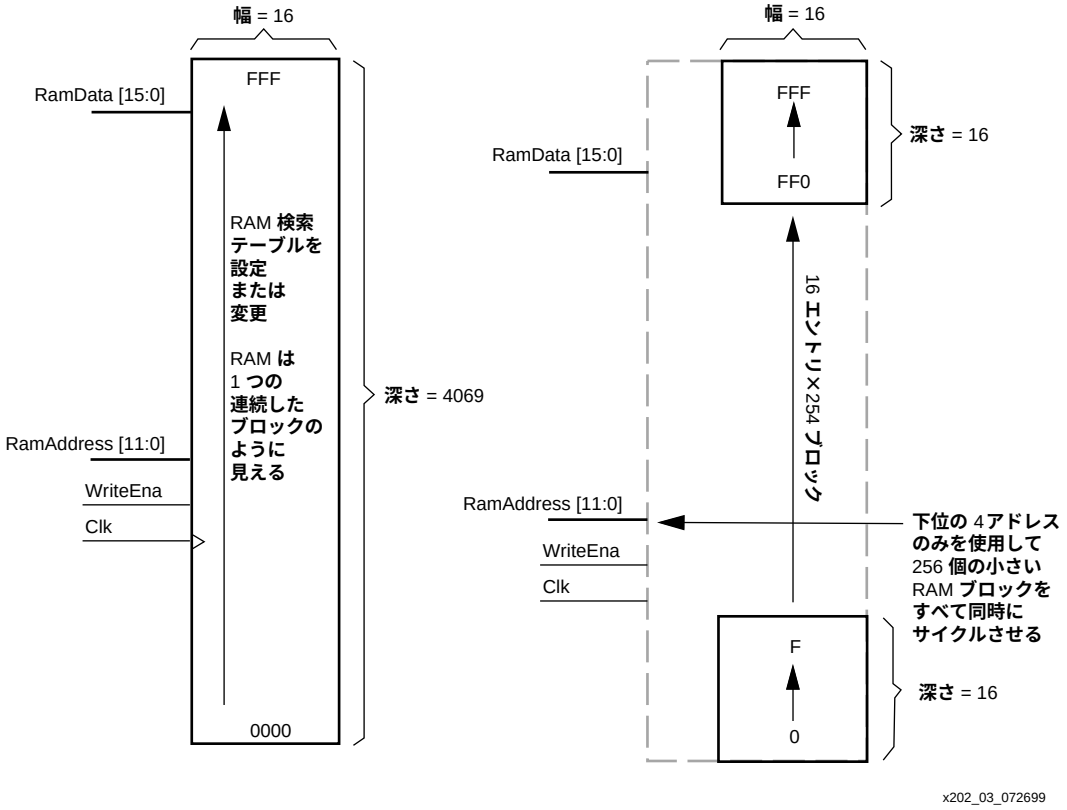


**図 3: 分散 RAM のテーブル**

### CAM の比較（ByteEngine）

ByteEngine は、CAM の基本的なブロックです。LUT によるアプローチを使用する場合には、データ幅のサイズは関係ありません。この基本的な構成要素では、16 ビットのデータ幅をそのまま使用します（12 ビットの VPI と 16 ビットの VCI）。VPI と VCI のデータ幅を組み合わせることも可能です（28 ビットの VPI/VCI）。

図 4 の ByteEngine は、必要なサイズの CAM を形成するために必要なだけ使用される小さな CAM です。これには、16 エントリの比較テーブル、比較する XNOR ゲート、有効で安定した一致（HIT）信号を生成するために必要なすべてのロジックが含まれています。

RAM テーブルの初期化には RamData バスを使用し、サイクル スルーには RamAddress バスを使用します。テーブルが初期化されると RamWe 信号が False に設定され、テーブルのサイクル読み込みができるようになります。比較する値が ToCompData バスを駆動すると、XNOR とワイド AND ゲートによって比較されます。すべての XNOR が有効な場合のみ、一致信号が生成され記憶されます（図 4）。

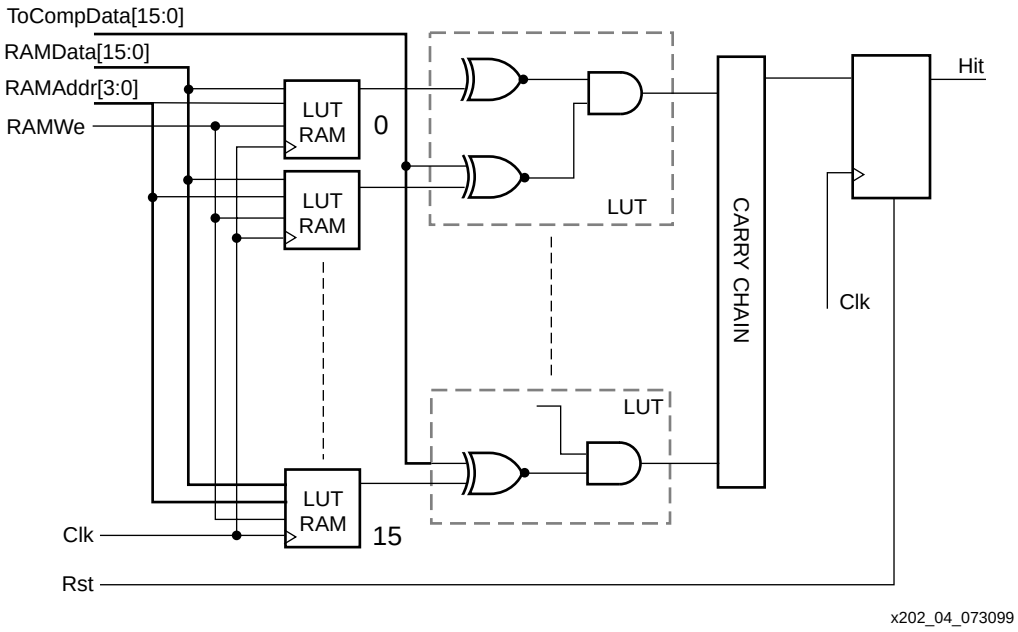


**図 4: 比較テーブル**

### CAM のサイズ

基本的な ByteEngine ブロックを使用して、任意のサイズの CAM を作成できます。図 5 に、EntriesEngine256 という名前の 256 エントリのテーブルを示します。

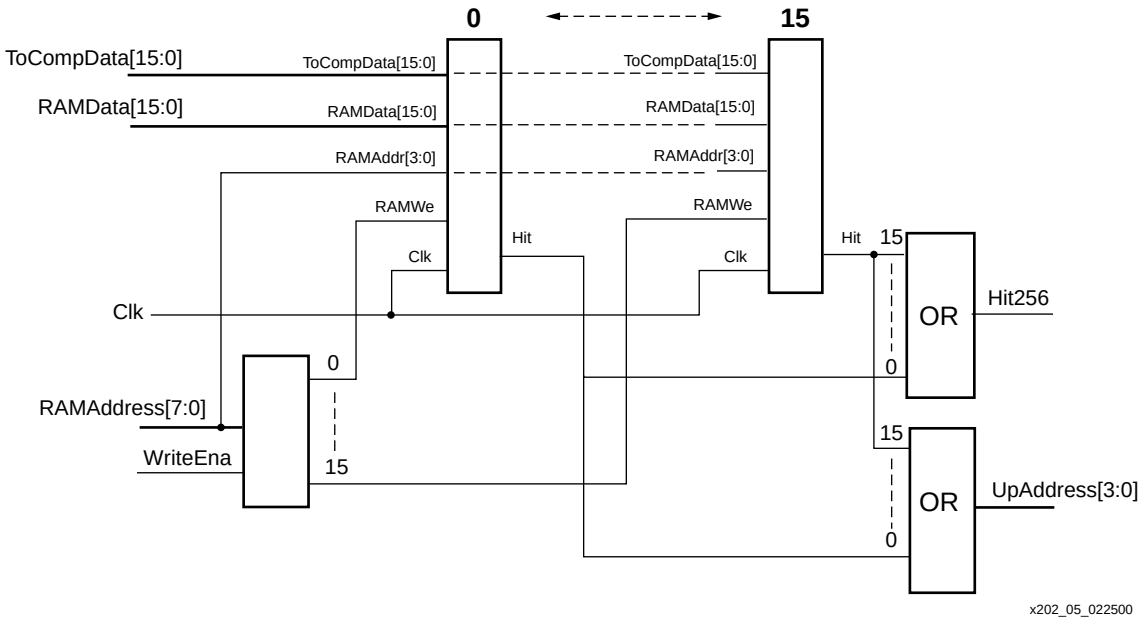


図 5: EntriesEngine256 CAM

EntriesEngine256 は、大規模な CAM を設計するための構成要素です。これには、入力で 16 個の ByteEngine ブロックをバンク選択し、出力アドレスおよび一致信号を生成するために必要なロジックのみが含まれています。16 個の ByteEngine ブロックを組み合わせて 256 入力のリストを形成します。アドレス デコーダは、リストを 1 つの長いテーブルとしてアドレス（初期化）できるようにします。出力には、一致信号を生成するためのエンコーダ（ワイド OR ゲート）があります。2 番目のエンコーダは、一致が発生したアドレスを生成するためのものです。次の例では、基本的なブロックを使用して CAM を作成します。

### CAM の例

図 6 に、最大 257 個の Virtex ファミリ スライスを使用する 16 × 256 の CAM を示します。これは、Virtex デバイス内で約 70MHz で動作します。

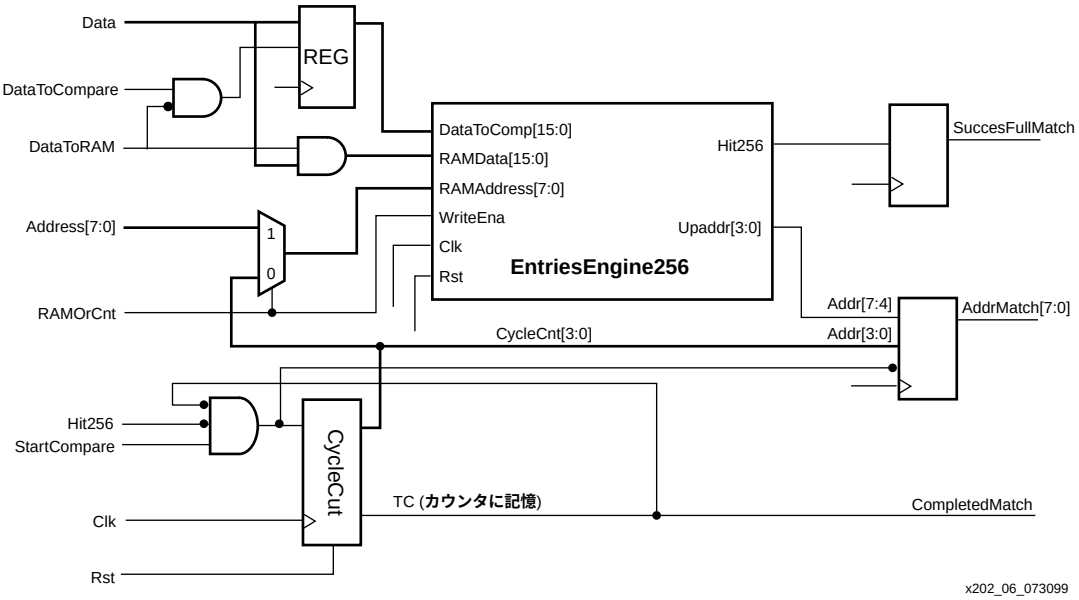


**図 6: 16 × 256 の CAM**

図 7 に、サイクル ダイアグラムを示します。CAM に対しては、次のように仮定されています。

- DataToCompare が 1、DataToRun が 0 の場合、データは DataToCompare レジスタに送られる。
- DataToRAM が 1 の場合（DataToCompare レジスタは無効）、データは RAM に渡される。
- RAMOrCnt が 0 の場合、内容を読み込むために CycleCnt が RAM に渡される。
- RAMOrCnt が 1 の場合、アドレスは RAM に渡される。
- RAM テーブルを埋めるには、DataToRAM と RAMOrCnt の両方が 1 でなければならない。
- DataToCompare を 1、DataToRAM を 0 にして RAMOrCnt を 0 に設定する。
- value_to_compare_to は、レジスタにラッチを介してレジスタに接続できる。
- 後から DataToCompare を "0" にする。
- StartCompare 信号を 1 にすると、サイクル カウンタを開始する。

これを実行すると、サイクル カウンタ（CycleCnt）は RAM のデータを順番に読み込んで DataToComp と比較します。16 バンク（深さは 256）のいずれかで一致が見つかると、そのバンクの一致レジスタが設定されます。値がデコードされ、Hit256 信号が生成されます。

Hit256 信号は、サイクル カウンタを停止させます。Hit 信号が生成されたバンクのデコードとカウンタの状態によって、与えられたデータと一致するアドレスが生成されます。

Hit256 と StartCompare は、AddressMatch レジスタを有効にして、有効なアドレスをラッチします。

Hit 信号が生成されず、サイクル カウンタが最後まで達した場合は、CompleteMatch サイクル信号が生成され、サイクル カウンタは停止します。

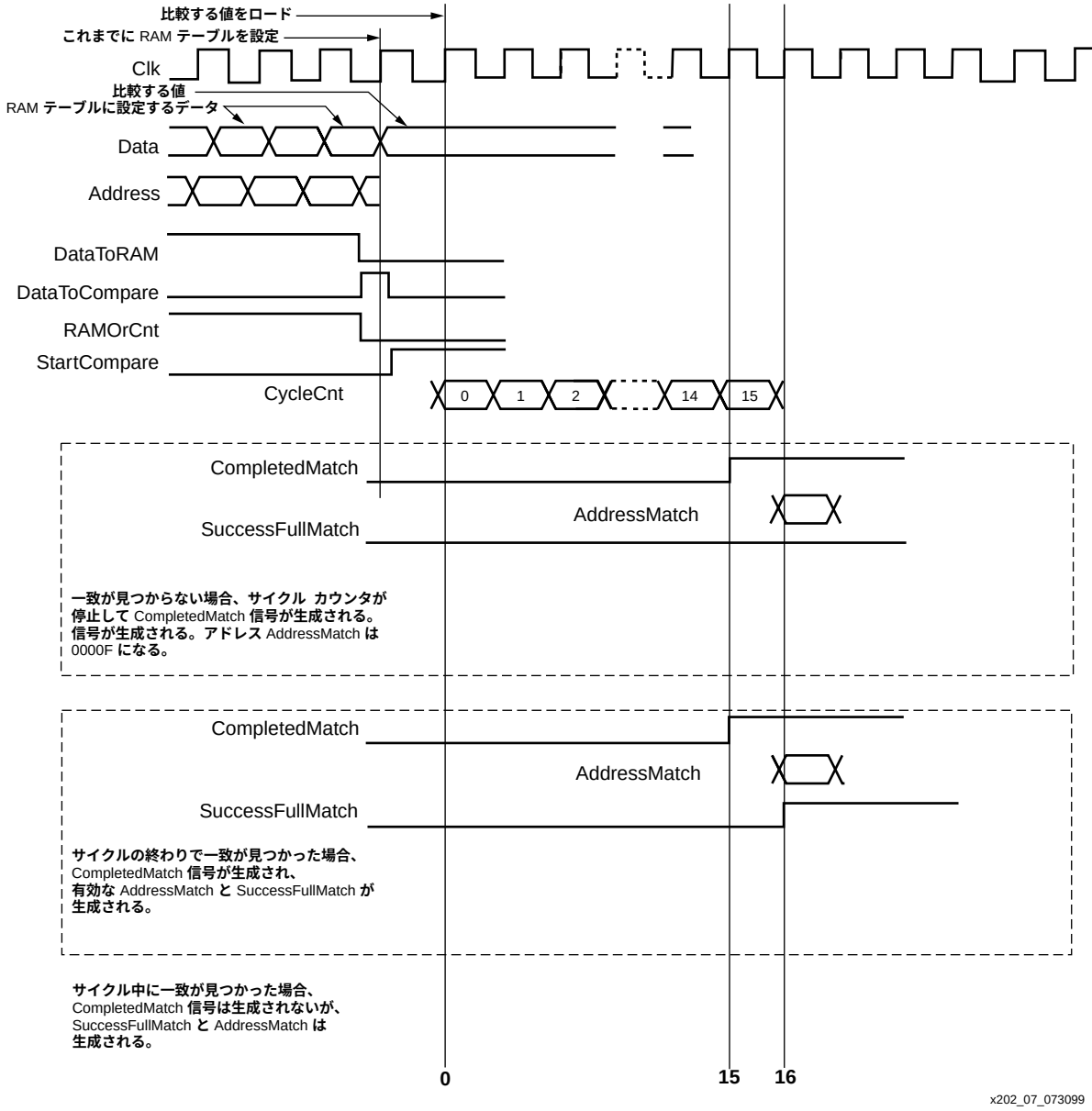


**図 7: サイクル ダイアグラム**

図 8 に、16 個の基本的な EntriesEngine256 モジュールとより多くのデコード ロジックを使用して同じ方法で作成した 4096 ワードの CAM を示します。

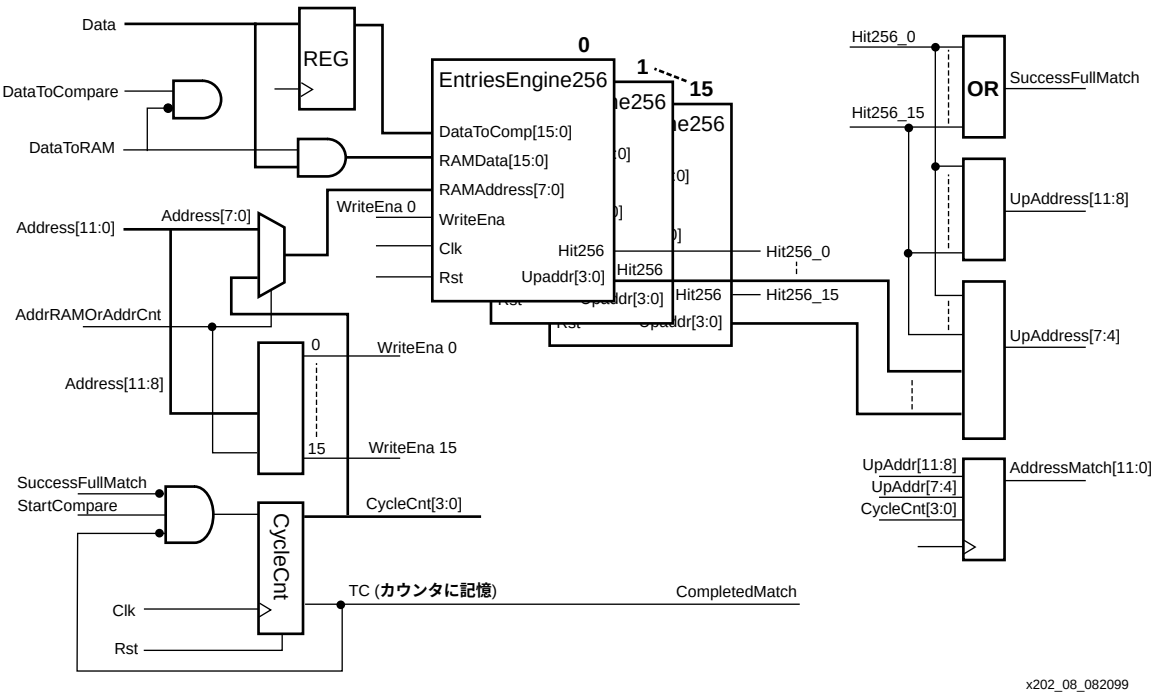


**図 8: 完全な CAM ソリューション**

図 8 に、出力データ テーブルとして Block SelectRAM+ メモリを使用した完全な CAM ソリューションの概略を示します。Block SelectRAM+ メモリの完全な Dual Read / Write Port™ 機能を使用すると、出力テーブルのデータを容易に変更できます。

## 概要

### CAM デザイン

- 4096 ワード（またはそれ以下）の編成では、1 × 16 の分散 RAM を使用します。RAM の内容は、16 クロック サイクルで照合します（同期 RAM）。
- 一致が見つかると、生成されたアドレスを使用してブロック RAM 内のデータまたは FPGA 外部にある RAM のデータを選択します。
- 2 つの比較動作の間で、RAM テーブルを簡単に検索できます。通常の連続した RAM のように表示されるため、テーブル内の特定のロケーションに書き込む必要があるのはアドレスとデータだけです。
- Block SelectRAM+ メモリの完全な Dual Read / Write Port 機能を使用すると、ブロック RAM に格納したデータに関係なく検索テーブルを変更できます。ATM の場合、ブロック RAM に格納されるデータは出力ポートになります。
- 16 クロック サイクルごとに新しくデータ照合されるので、このデザインはデータのサイズや検索テーブルのサイズに関係なく検索できます。VPI または VCI のアドレスをラッチしたり、レジスタで見つかったアドレスを出力するには、さらに数サイクル（最大 18 サイクル）が必要になります。
- 4096 入力の CAM は、XCV600 または XCV600E に収まります。この場合、使用可能な 24 個のブロック RAM を 24 ビット × 4096 ワードのデータ テーブルとして使用します。
- 256 ワード × 80 ビットなどの小さい CAM は、分散 RAM だけで作成できます。

- CAM を使用する前に、比較するデータと比較テーブルの両方を初期化する必要があります。動作中に初期化した場合は、次のようになります。
  - 分散 RAM の比較テーブルを RAM コンフィギュレーション モードに切り替える必要があります。通常の CAM 動作では、このメモリは小さいワード（16 ワード）に分割されます。
  - ブロック RAM の比較テーブルは、2 番目のポートを使用すればいつでも更新できます。ブロック RAM は、完全に独立した 2 つのポートがある真の Dual Read / Write Port RAM です。

## おわりに

この CAM デザインでは、18 サイクルごとに照合されます。この中の 16 サイクルは、小さい分散 SelectRAM+ ブロックのスクロールに必要です。1 クロック サイクルは比較するデータのロードに必要で、残りの 1 クロック サイクルは一致した値の出力に必要です。

アプリケーション ノート XAPP201 に示されているように、Virtex ファミリ デバイスの柔軟性は CAM を設計する際の主要な利点となります。このアプリケーション ノートのソリューションの他に、XAPP203 と XAPP204 では異なるアプリケーション条件に基づく別のアプローチが示されています。

ATM アプリケーションで大規模な CAM を作成する最も経済的な方法は、Virtex アーキテクチャで使用できる分散 RAM（基本的な構成は 1 × 16）とブロック RAM（基本的な構成は 1 × 4096）の両方を使用することです。分散 RAM と外部 RAM ブロックは、大規模な CAM の作成にも使用できます。CAM の比較テーブルは分散 RAM を使用して作成でき、データはブロック RAM または外部 RAM に格納できます。24 ビット × 4096 ワードの比較テーブルがある CAM は、XCV600 または XCV600E に収まります。

## 付録 A : 合成可能な参考デザインの HDL コード

付録 A では、Virtex スライスで検索エンジンまたは CAM をインプリメントする、階層的で合成可能なデザインについて説明します。

完全な HDL コードは、参考デザインとして使用できます（ファイル : xapp202.zip または xapp202.tar.z）。

各 VHDL モジュールのヘッダを次に示します。

```
Module: MatchMachine4k.vhdl


-- Entity Name:  MatchMachine4k
-- File Name:    MatchMachine4k.vhd
-- File Path:    D:\projects\Cam\vhdl\
-- Project :
--
-- Purpose:  This is a machine that can do a CAM operation
--           on 16 bits for 4096 entries in 18 clock cycles.
--           files used :
--                      ByteEngine.vhd
--                      EntriesEngine256.vhd
--
---- Authors: Marc Defossez
--
-- Tools: Synplicity 5.2.1
--
-- Revision History:       Created:       20/04/99
--                         Last opened:   Wednesday, 06 June 99
--
--
--
--   Disclaimer:  THESE DESIGNS ARE PROVIDED "AS IS" WITH NO WARRANTY
--                WHATSOEVER AND XILINX SPECIFICALLY DISCLAIMS ANY
--                IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR
--                A PARTICULAR PURPOSE, OR AGAINST INFRINGEMENT.
--
--  Copyright (c) 1999 Xilinx, Inc.  All rights reserved.
----------------------------------------------------------------------


Module: MatchMachine256.vhdl
-- Entity Name:  MatchMachine256
-- File Name:    MatchMachine256.vhd
-- File Path:    D:\projects\Cam\vhdl\
-- Project :
```

```
--
-- Purpose:  This is a machine that can do a CAM operation
--           on 16 bits for 256 entries in 18 clock cycles.
--           files used :
--
--                      ByteEngine.vhd
--                      EntriesEngine256.vhd
--
...
----------------------------------------------------------------------
Module: EntriesEngine256.vhdl
-- Entity Name:  EntriesEngine256
-- File Name:    EntriesEngine256.vhd
-- File Path:    D:\projects\Cam\vhdl\
-- Project :
--
-- Purpose:     This is the engine that compares in 16 clock
--              cycles 256 values against a given value on a
--              double byte width (16 bits).
--              This is one section of a VPI/VCI cam.
--              Makes use of
--                              Byte Engine.vhd
--                              EntireEngine256.ucf
--
…
----------------------------------------------------------------------
Module: ByteEngine.vhdl
-- Entity Name:  ByteEngine
-- File Name:    ByteEngine.vhd
-- File Path:    D:\projects\Cam\vhdl\
-- Project :    CAM
--
-- Purpose: Engine over 16 bits.
--          Compares 16 bits over 16 deeh and give a Hit
--          signal if the 16 bit value is found in to table.
--
--          Because the depth will be bigger than 16 bit's there
--          is need for working in BANKS of 16.
--          Like for 256 entries, 16 banks will be needed.
--          In the file above this, two banks are combined.
```

```
--          Reason for doing this is RLOCing.
--
--          As the ByteEngine is made now, 8 CLBs are in this way:
--          If nicely lined up, there will be a column of 8 CLBs where
--          slice S1 is used to store 2 x a RAM16X1S (16 bits).
--          and slice S0 will only contain 8 LUTs + carry chain for the
--          comparitor. Thus there is some mismatch between the RAM
--          column hight and the comparitor hight.
--
--          For UCF file test purposes, following is done
--          Combination of two of these ByteEngine.vhd files is done
--          in TwoBanks.vhd and a UCF file with RLOC's is made
--          (TwoBanks.ucf)
--          A small 256 entries engine is made, lateron this 256 engine
--          can be combined to form bigger chuncks of memory.
--
--
...
```

## 改訂履歴

次の表に、このドキュメントの改訂履歴を示します。

| 日付 | バージョン番号 | 改訂内容 |
|---|---|---|
| 1999 年 9 月 1 日 | 1.0 | 初期リリース |
| 1999 年 9 月 23 日 | 1.1 | Virtex-E の初期アップデート |
| 2000 年 2 月 28 日 | 1.2 | 新しいテンプレートに再フォーマット |