第2章

FRAME #/IRDY #を読みDEVSEL #/TRDY #を操る

PCIバス・トランザクションの基礎

来須川 智久

PCIバスに準拠するためにはコンフィグレーション・レジスタが不可欠だが，PCIバスを理解することを優先し，ここではコンフィグレーション空間をもたない，アドレスを固定してデコードすることでメモリ・サイクルに応答するPCIデバイスを設計する．第1章で説明したように，基本的に各トランザクションはすべて同じタイミングで制御されている．この共通した基本ルールを明確にし，これを理解することでPCIバスをマスタできる．

（編集部）

メモリ・ライト・サイクル ― ターゲット1の設計

● トランザクションの本質を理解するために

いきなりPCIバスに完全準拠したデバイスを設計するとなると，コンフィグレーション空間や基本的なトランザクションに付随する細かなタイミングを考慮することが必要で，あれこれ説明しているうちにPCIバス・トランザクションの本質を見逃してしまうおそれがあります．

そこでまずはじめに，コンフィグレーション空間をもたずにデコード・アドレスをハードウェア的に固定（ソース・コード上で定数として定義）してしまい，しかも対応するトランザクションはメモリ・ライト・サイクルのみ（書き込み専用）という，PCIバスの規格からすれば言語道断といってよいほど仕様を削ったデバイスを設計します．

最初に設計するデバイスをターゲット1と名付け，少しずつ機能を追加していきながらターゲット2，3と設計していきます．

なお，本書で設計した設計データを，実際にデバイスに書き込んでPCIバスで使う場合は，まず**コラム1**を参照していただき，そのまま使用できるかどうかを確認してください．

● VHDLソースの雛形

図1にVHDLソースの雛形を示します．以後の解説ではこのソースの各部に，各設計で必要なソースをはめ込んでいったり，機能を追加していくスタイルを採ります．

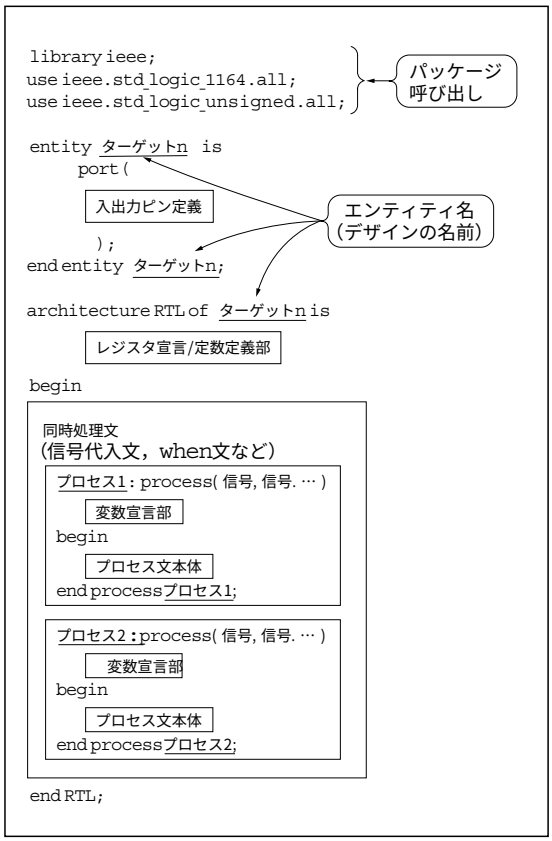


図1　PCIターゲット・デバイス設計のVHDLソースの雛形

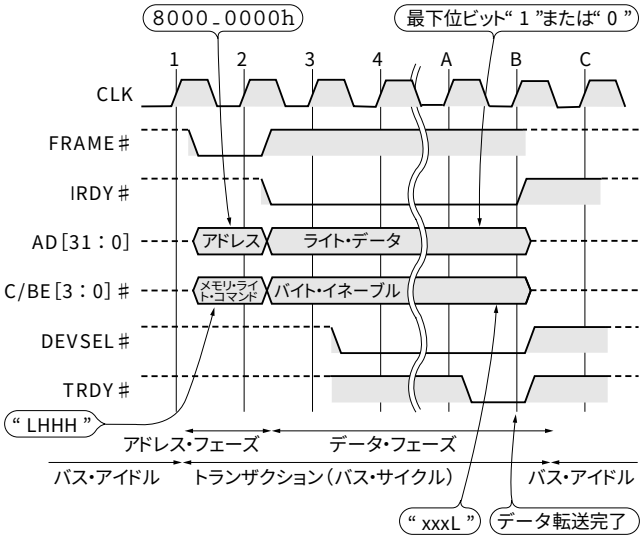


図2
メモリ・アドレス8000_0000h番地にデータを書き込む

2.1　シーケンサの骨格

● ハードウェアの仕様

まず，最初に設計するハードウェアをターゲット1と名付けました．このデバイスの仕様を次に示します．

1) ターゲット・デバイスのローカル側にLEDを1個接続
2) メモリ・ライト・トランザクションに応答するPCIターゲット・デバイス
3) 占有メモリ・アドレスは8000_0000h番地固定
4) 書き込み時のAD［0］ビットの値が“1”ならLEDを点灯/“0”なら消灯

ISAバスならあっという間にできそうな，非常に簡単なハードウェアです．このときのPCIバスの動作を予想した波形が，**図2**になります．DEVSEL#とTRDY#をこの図のようなタイミングで制御すればよいわけです．

● ステート・マシン

図2のように，クロック1で○○をして，クロック2で△△がアサートされたら□□をして…といった制御をするには，ステート・マシンと呼ばれる回路が使われます．そのためには，クロック1で必要な仕事は何か，クロック2でどの信号が変化するのを待つのか…ということを洗い出す必要があります(**図3**)．

これが，PCIバスを制御するステート・マシンの内容となります．これをPCIターゲット・シーケンサと名付け，VHDLではターゲット・シーケンサ・プロセスとしてprocess文を記述します．また，このほかに実際にアドレスをデコードする回路，そして

コラム1　テスト環境の確認

ターゲット1とターゲット2ではデコード・アドレスを8000_0000hのアドレスに固定しているため，この設計データを実際にデバイスに書き込んでPCIバスで使う場合，その環境ですでにほかのPCIデバイスによって8000_0000hのアドレス領域が使われていると正常に動作しません．

また，後述するコンフィグレーション・レジスタのベンダIDとデバイスIDも，場合によってはすでに同じIDが使われている可能性もあります．

そこで，PCIボードの動作確認に使うPC/AT互換機のPCI環境をチェックするPCICHK.EXEというプログラムを用意しました(本書付属CD-ROMに収録)．場合によっては，デコード・アドレスやベンダID/デバイスIDの変更が必要な場合があります．

ただし，このチェックでOKが出ても第10章のコラム3(p.148)のようなマザーボードでは，ターゲット1とターゲット2は動かない場合があります．詳細はAppendix2を参照してください．

LEDの点灯制御回路も必要です．

● ターゲット1のブロック図

ターゲット1の仕様ではLEDを点灯制御するだけですから，ステート・マシンなどを使わなくても十分間に合います．しかし，今後の設計で機能を追加していくためのたたき台として，最初のターゲット1の設計からステート・マシンを組み，シーケンサを実現します．

また，LEDの点灯制御だけではローカル・バスというイメージはありませんが，今後ここにSRAMなど外部デバイスを接続していくので，ローカル・バスと呼ぶことにします．VHDLでは，ローカル・バス・シーケンサ・プロセスとしてprocess文を記述しています．

また，このようにPCIバスとローカル・バスを分離して考えることにより，お互いが独立して動作するようなデバイスをPCIバスに接続するとき，設計が楽になります．

以上をまとめて，ターゲット1のブロック図を**図4**に示します．

● port宣言

リスト1にPCIバスの入出力ピンを定義するport宣言部を示します．PCIバスの信号名に“＃”が付く信号，つまり負論理の信号にはVHDLソース上では信号名に“_n”を付けて表現しています．ターゲット1で使用するPCIバス上の信号はこれだけです．

ただし，Appendix1のPCI評価ボードを使って動作確認をする場合は，これ以外にもすでに配線されている信号線があるので，**リスト1**の宣言だけでは足りません．第10章の説明や付属CD-ROMに収録してある各社試作評価用ボード向けのプロジェクト・ファイルの説明を参照してください．

● レジスタ宣言/同時処理文

リスト2にターゲット1のレジスタ宣言/定数定義部と同時処理文を示します．後述する各シーケンサで使用する信号や定数を定義し，またトライステート信号の動作を記述しておきます．

リスト1や**リスト2**は，とりあえず呪文だと思ってそのまま使ってください．肝心なのは次から説明する三つのプロセス文で，いかにしてPCIバスを制御するかです．

● 入力/出力/トライステート出力/トライステート入出力ポート

図4に各信号のポートのようすを示します．**リスト**

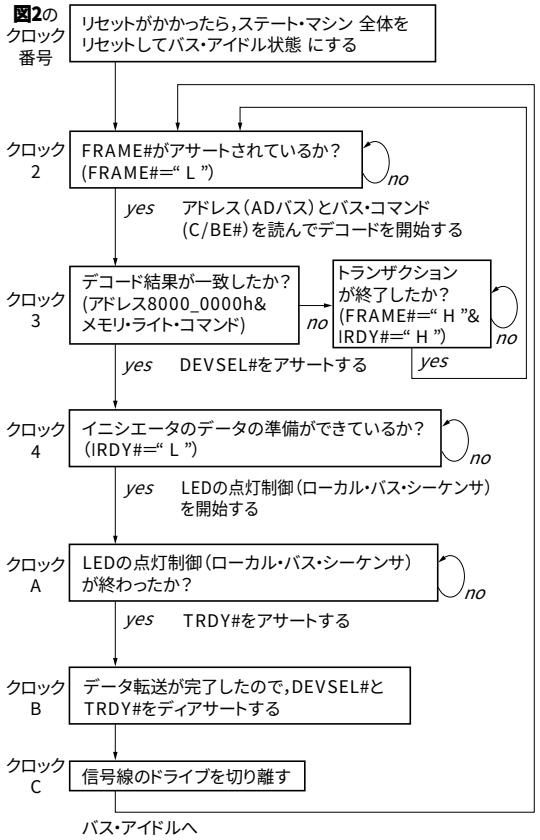


図3　PCIバスの制御手順

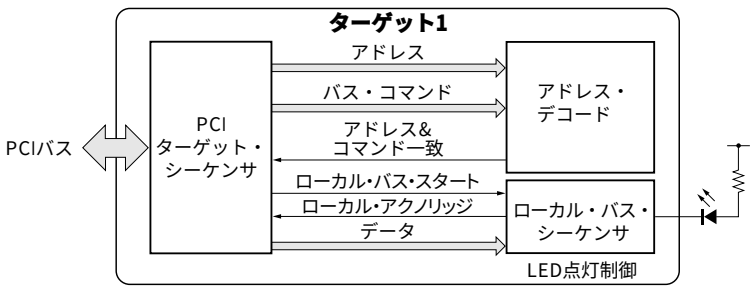


図4
ターゲット1のブロック図

リスト1
ターゲット1の port宣言部

```
entity PCI_TGT1 is
    port(
    -- PCIバス信号ピン(ターゲット1で使用する信号) --
         PCICLK   :       in    std_logic;            -- PCIバス・クロック
         RST_n    :       in    std_logic;            -- 非同期リセット
         PCIAD    :       inout std_logic_vector(31 downto 0);   -- アドレス/データ・バス
         C_BE_n   :       in    std_logic_vector(3 downto 0);    -- PCIバス・コマンド/バイト・イネーブル
         FRAME_n  :       in    std_logic;            -- フレーム
         IRDY_n   :       in    std_logic;            -- イニシエータ・レディ
         DEVSEL_n :       out   std_logic;            -- デバイス・セレクション
         TRDY_n   :       out   std_logic;            -- ターゲット・レディ
    -- LED制御ピン
         LED_OUT  :       out   std_logic
    );
end entity PCI_TGT1;
```

リスト2
ターゲット1の レジスタ宣言/ 定数定義部と 同時処理文

```
architecture RTL of PCI_TGT1 is

-- **********  レジスタ/定数 定義部分

-- PCIバス・コマンド/アドレス/IDSELホールド・レジスタ --
    signal PCI_BusCommand : std_logic_vector(3 downto 0); -- PCIバス・コマンド・レジスタ
    signal PCI_Address    : std_logic_vector(31 downto 0);-- PCIアドレス・レジスタ

-- ローカル・バス・シーケンサ スタート・フラグ
    signal LOCAL_Bus_Start : std_logic;
-- ローカル・バス・シーケンサ データ転送完了フラグ
    signal LOCAL_DTACK     : std_logic;

-- トライステート・バッファ制御用のフリップフロップ定義 --
    -- PCIバス信号線
    signal PCIAD_HiZ  : std_logic;                          -- ADポート出力ドライブ制御
    signal PCIAD_Port : std_logic_vector(31 downto 0);      -- ADポート出力レジスタ
    signal DEVSEL_HiZ, DEVSEL_Port : std_logic;             -- DEVSEL#出力ドライブ制御/出力レジスタ
    signal TRDY_HiZ, TRDY_Port     : std_logic;             -- TRDY#出力ドライブ制御/出力レジスタ

--  PCIバス・コマンド(ビット・パターン定義) --
    -- メモリ・リード・サイクル
    constant PCI_MemWriteCycle : std_logic_vector(3 downto 0) := ( "0111");

-- アドレス・デコード・フラグ
    signal Hit_Device : std_logic;  -- デバイス・ヒット

begin

-- **********  同時処理文

-- トライステート・バッファ動作
    PCIAD    <= (others => 'Z') when PCIAD_HiZ =  '1' else PCIAD_Port;
    DEVSEL_n <=  'Z' when DEVSEL_HiZ =  '1' else DEVSEL_Port;
    TRDY_n   <=  'Z' when TRDY_HiZ =  '1'  else TRDY_Port;
```

1でinとして宣言した信号は，後述するプロセス文の中でいつでも状態を参照可能，つまり代入文の右辺に使えます〔**図5**(**a**)〕.

LED出力は常時信号を出力するので，**リスト1**のようにoutと宣言するだけで，代入文で値を代入することで信号を出力します〔**図5**(**b**)〕．outで宣言したポートは出力専用なので代入文の右辺には使えません.

また，PCIターゲット・デバイスを設計する場合，DEVSEL#やTRDY#はトライステートで制御する出力信号です．port宣言ではoutとして宣言します．さらに，**リスト2**のsignal宣言で信号を定義し，同時処理文との組み合わせで，ハイ・インピーダンスの制御をします〔**図5**(**c**)〕.

たとえば，DEVSEL_HiZに“1”を代入すると，実際のDEVSEL#の出力にあたるDEVSEL_nには“Z”が代入され，ハイ・インピーダンスになるわけです．DEVSEL_HiZが“1”以外の値ならば，DEVSEL_Portの内容がDEVSEL_nに出力されます．DEVSEL_nに何を出力するかは同時処理文に記述するので，DEVSEL_HiZやDEVSEL_Portに値を代入した瞬間にDEVSEL_nの出力内容も変わります.

ADバスはデータの入出力があるのでinoutとして双方向で宣言します．これもトライステート制御なので，**リスト2**に示すようにトライステート制御用にsignal宣言で信号を定義し，同時処理文で動作を記述します〔**図5**(**d**)〕．以降は，それぞれのブロックごとに，処理内容の詳細を説明します.

2.2　PCIターゲット・シーケンサ

図3のPCIバスの制御手順をそのままシーケンサにした状態遷移図を**図6**に，PCIターゲット・シーケンサ・プロセス部分のVHDLのソースを**リスト3**に示します．また，このシーケンサを実際にPCIバスで動作させた場合のタイミングを**図7**に示します．

それでは，各ステートがどのような意味をもち，どのような処理をしているかについて詳しく解説します．

● リセット処理

PCIバスにおけるリセットは，RST#がアサートされることにより行われます．これは非同期に行われるので，クロック・イベント行より先に判定しています．

リセット時には，ステート・マシンの状態を示す変数をバス・アイドル状態にセットし，アドレスやバス・コマンドを保持するレジスタや内部変数をクリアします．

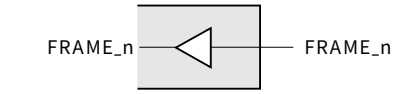
また，ターゲットがドライブするDEVSEL#とTRDY#，およびADバス(ターゲット1ではドライブしないが)をハイ・インピーダンス状態に設定します．

● BUS_IDLEステート

これは，ターゲット・シーケンサがアイドル状態のときのステートです．PCIバス上でトランザクションが開始されるまで，このBUS_IDLEステートにとどまり続けます．

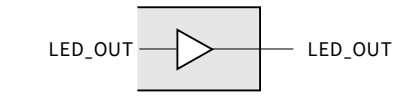
ここではトランザクションの開始は，FRAME#がアサートされIRDY#がディアサートされているという条件で判定しています．

ここで，ADバスの状態をアドレスとしてPCIアドレス・レジスタ(PCI_Address)に取り込み，C/BE#の状態をバス・コマンドとしてPCIバス・コマンド・レ



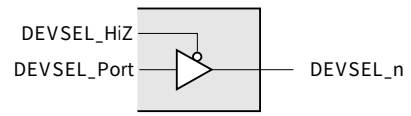

```
port宣言    FRAME_n:in std_logic;
```

(**a**) 入力ポート



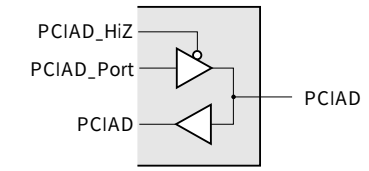

```
port宣言    LED_OUT:out std_logic;
```

(**b**) 出力ポート




```
port宣言    DEVSEL_n:out std_logic;
signal宣言  signal DEVSEL_HiZ:std_logic;
signal宣言  signal DEVSEL_Port:std_logic;
同時処理文  DEVSEL_n<='Z'when DEVSEL_HiZ='1'
                                    else DEVSEL_Port;
```

(**c**) トライステート出力ポート




```
port宣言    PCIAD:inout std_logic_vector   (31 downto 0);
signal宣言  signal PCIAD_HiZ:std_logic;
signal宣言  signal PCIAD_Port:std_logic_
                                    vector(31 downto 0);
同時処理文  PCIAD<=(others=>'Z')when
            PCIAD_HiZ='1'else PCIAD_Port;
```

(**d**) トライステート入出力ポート

図5　入力/出力/トライステート出力/トライステート入出力ポート

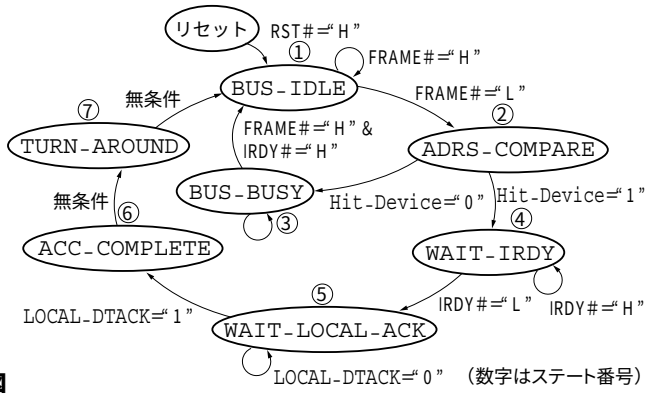


図6
PCIターゲット・シーケンサの状態遷移図

リスト3　PCIターゲット・シーケンサ・プロセス部分のVHDLのソース

```
-- **********  PCIターゲット・シーケンサ

PCI_TGT_Seq : process( PCICLK, RST_n )

-- PCIターゲット・シーケンサ・ステート・バリュー・レジスタ --
    variable PCI_CURRENT_STATE : std_logic_vector (2 downto 0);-- 現在のステート
    variable PCI_NEXT_STATE    : std_logic_vector (2 downto 0);-- 次のステート

-- PCIターゲット・シーケンサ・ステート・マシン定義
    constant BUS_IDLE          : std_logic_vector (2 downto 0) :="000";
    constant ADRS_COMPARE      : std_logic_vector (2 downto 0) :="010";
    constant WAIT_IRDY         : std_logic_vector (2 downto 0) :="011";
    constant WAIT_LOCAL_ACK    : std_logic_vector (2 downto 0) :="100";
    constant ACC_COMPLETE      : std_logic_vector (2 downto 0) :="101";
    constant BUS_BUSY          : std_logic_vector (2 downto 0) :="110";
    constant TURN_AROUND       : std_logic_vector (2 downto 0) :="111";

begin

-- ********** リセット時動作 ********** --
    if (RST_n = '0') then
            -- PCIバス・リセット時(非同期リセット)

        PCI_CURRENT_STATE:= BUS_IDLE;      -- ステート・マシン IDLE状態 リセット
        PCI_NEXT_STATE  := BUS_IDLE;       -- ステート・マシン IDLE状態 リセット

        LOCAL_Bus_Start <= '0';            -- ローカル・バス・シーケンサ スタート・フラグ クリア

        PCI_BusCommand <= (others => '0'); -- PCIバス・コマンド・レジスタ クリア
        PCI_Address <= (others => '0');    -- PCIバス・アドレス・レジスタ クリア

    -- 制御出力端子をハイ・インピーダンス
        PCIAD_HiZ <= '1';
        DEVSEL_HiZ <= '1'; DEVSEL_Port <= '1';  -- DEVSEL#="H"
        TRDY_HiZ   <= '1'; TRDY_Port   <= '1';  -- TRDY#="H"

-- ********** PCIターゲット・シーケンサ・ステート・マシン ********** --
    elsif (PCICLK'event and PCICLK = '1') then

        PCI_CURRENT_STATE := PCI_NEXT_STATE;-- ステート・マシン制御
        case PCI_CURRENT_STATE is

    -- ********** BUS_IDLE時の動作 ********** --
        when BUS_IDLE =>-- トランザクションの開始待ち

            if (FRAME_n = '0' and IRDY_n = '1') then -- トランザクション開始
                PCI_BusCommand <= C_BE_n;           -- PCIバス・コマンド取得
                PCI_Address <= PCIAD;               -- アドレス取得
                PCI_NEXT_STATE := ADRS_COMPARE;
            else -- バス・アイドル時このステートにとどまる
                PCI_NEXT_STATE := BUS_IDLE;
            end if;

    -- ********** ADRS_COMPARE時の動作 ********** --
        when ADRS_COMPARE =>  -- アドレス・デコード結果を調べる

            if (Hit_Device = '1') then                  -- 自分が選択された
                DEVSEL_Port <= '0'; DEVSEL_HiZ <= '0';-- DEVLSEL#アサート
                TRDY_HiZ <= '0';                        -- TRDY# を "H"にドライブ
                PCI_NEXT_STATE := WAIT_IRDY;            -- イニシエータ・レディを待つステートへ
            else            -- 自分が選択されていない

                PCI_NEXT_STATE := BUS_BUSY;
                            -- トランザクションの終了を待つステートへ
            end if;

    -- ********** BUS_BUSY時の動作 ********** --
        when BUS_BUSY =>-- トランザクション終了待ち

            if (FRAME_n = '1' and IRDY_n = '1') then-- トランザクション終了(アイドル)
                PCI_NEXT_STATE := BUS_IDLE;     -- トランザクション開始待ちステートへ
            else -- トランザクション中ならこのステートにとどまる
                PCI_NEXT_STATE := BUS_BUSY;
            end if;

    -- ********** WAIT_IRDY時の動作 ********** --
        when WAIT_IRDY =>     -- イニシエータ・レディ待ち

            if (IRDY_n = '0') then-- イニシエータの準備完了
                LOCAL_Bus_Start <= '1';-- ローカル・バス・シーケンサ スタート!
                PCI_NEXT_STATE := WAIT_LOCAL_ACK;-- ローカル・バス・シーケンサ終了待ちステートへ
            else -- イニシエータの準備がまだならこのステートにとどまる
                PCI_NEXT_STATE := WAIT_IRDY;
            end if;
```

リセット時の処理

クロックの立ち上がりに同期して動作するステート・マシンの記述

BUS_IDLEステート

ADR_COMPAREステート

BUS_BUSYステート

WAIT_IRDYステート

リスト3　PCIターゲット・シーケンサ・プロセス部分のVHDLのソース（つづき）

```
    -- ********** WAIT_LOCAL_ACK時の動作 ********** --
          when WAIT_LOCAL_ACK =>
          -- ローカル・バス・シーケンサ終了待ち

                LOCAL_Bus_Start <= '0';-- ローカル・バス・シーケンサ スタート・フラグ クリア
                if (LOCAL_DTACK = '1') then      -- ローカル・バス・シーケンサ データ転送完了
                     TRDY_Port <= '0';-- TRDY# アサート
                     PCI_NEXT_STATE := ACC_COMPLETE;-- アクセス完了ステートへ
                else -- ローカル・バス・シーケンサの準備がまだならこのステートにとどまる
                     PCI_NEXT_STATE := WAIT_LOCAL_ACK;
                end if;

    -- ********** ACC_COMPLETE時の動作 ********** --
          when ACC_COMPLETE =>  -- アクセス完了ステート

                DEVSEL_Port <= '1';-- DEVSEL# ディアサート
                TRDY_Port <= '1';-- TRDY# ディアサート
                PCIAD_HiZ <= '1' ;-- PCIAD[31:0]バス・ドライブ解放
                PCI_NEXT_STATE := TURN_AROUND;-- ターンアラウンド・ステートへ

    -- ********** TURN_AROUND時の動作 ********** --
          when TURN_AROUND =>   -- ターンアラウンド・ステート

                DEVSEL_HiZ <= '1';    -- DEVSEL#ドライブ解放
                TRDY_HiZ <= '1';      -- TRDY#ドライブ解放
                PCI_NEXT_STATE := BUS_IDLE;-- トランザクション開始待ちステートへ

    -- ************************************** --
          when others => null;  -- これ以外の値では何もしない場合でも必ず入れる

    end case;

    end if;

end process PCI_TGT_Seq;
```

WAIT_LOCAL_ACK ステート

ACC_COMPLETE ステート

TURN_AROUND ステート

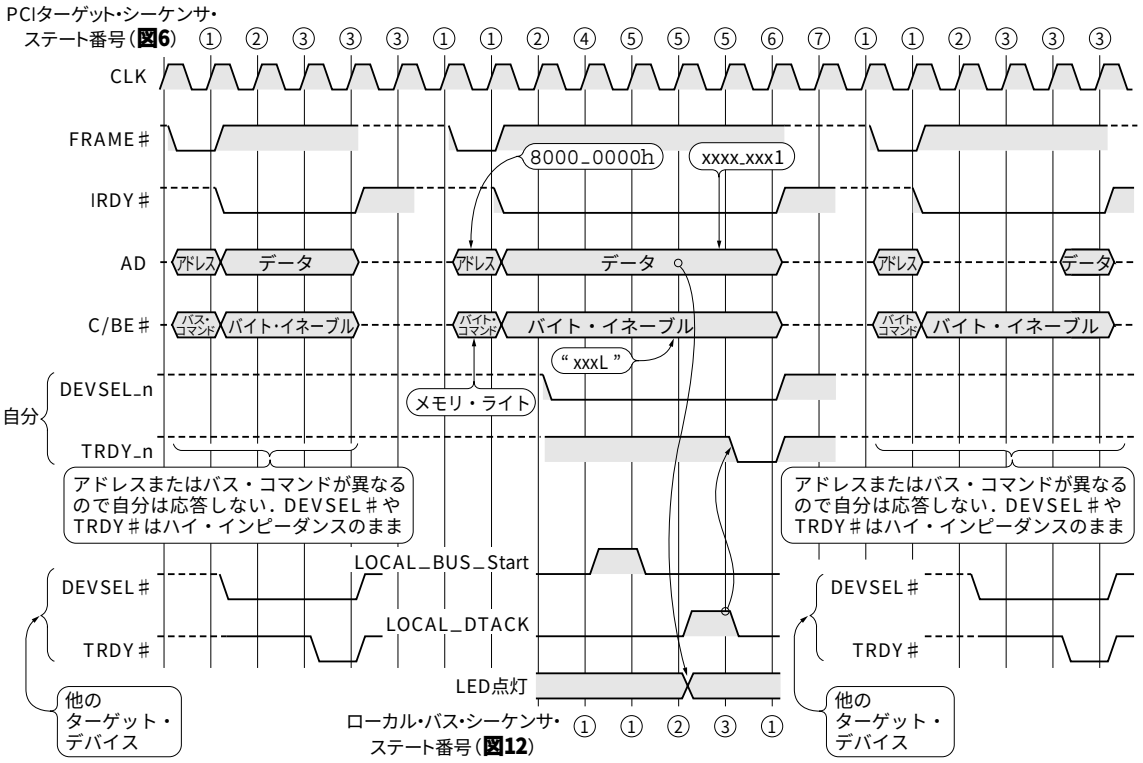


図7　ターゲット1の動作

ジスタ(PCI_BusCommand)に取り込みます．アドレス・フェーズは，バス・アイドル状態からFRAME#がアサートされた最初のPCIクロックの立ち上がりのときだけです．アドレスとバス・コマンドをレジスタに取り込んだら，ADRS_COMPAREステートへ移行します．

ここではFRAME#がアサートされたという条件以外に，さらにIRDY#がディアサートされていることもアドレス・フェーズの認識条件に追加し，より厳密にアドレス・フェーズを認識しています．FRAME#がアサートされた条件だけでは，FRAME#とIRDY#が同時にアサートされたときもトランザクションが開始されたと判定してしまいます．

FRAME#とIRDY#が同時にアサートされることは，PCIバスの規格上認められていません．もしこのようなアクセスが発生したとしたらイニシエータ・デバイスの動作不良であるとみなし，DEVSEL応答を返さないようにしています．

もっとも，最悪の条件を考えすぎてもあまり意味がないので，現状ではFRAME#がアサートされた条件だけでトランザクションの開始を判定しても，実質的には問題ないでしょう．

トランザクションの開始が判定できなければ，このステートにとどまります．ここでは，同じステートにとどまる場合は，次のステートの状態を保持するPCI_NEXT_STATEに同じステートの値を代入しています．VHDLの記述では代入してもしなくても，すでにその値が入っているわけなので，このelse文は省略できますが，条件が成立していなければ同じステートにとどまることを明示的に示す意味で記述しています．ほかのステートでも同様です．

● ADRS_COMPAREステート

アドレス・デコード結果を調べて，現在発生しているトランザクションが自分にあてられたものか，またはほかのターゲットのものかを判定します．Hit_Deviceが“1”なら，自分が選択されていることになり，DEVSEL#をアサートしてイニシエータにDEVSEL応答を返し，WAIT_IRDYステートへ移行します．

また同時に，TRDY#のドライブも開始します．TRDY#の信号レベルはリセット時に“1”を入れているので，“H”レベルとなります．

PCIバス規格ではこのDEVSEL応答を，FRAME#がアサートされたことを認識してから3クロックまでの間に行わなければなりません．速さによって高速/中速/低速応答と呼ばれますが，今回の設計では中速応答になります．

もし，Hit_Deviceが“1”ではない場合，ほかのターゲットへのアクセスであると判断して，BUS_BUSYステートへ移行します．

● BUS_BUSYステート

BUS_BUSYステートは，ほかのターゲットに対するトランザクションが発生しているとき，そのトランザクションが終了するまで待避するステートです．

たとえば，直前のADRS_COMPAREステートで，発生したトランザクションが自分宛てのものではないと判定したあと，そのままBUS_IDLEステートに戻ってもよさそうなものですが，それはできません．

PCIバス・トランザクションは一つのアドレス・フェーズと，それに続く一つもしくは複数のデータ・フェーズにより成り立っています．一つのトランザクション中に複数のデータ・フェーズが存在する転送を，バースト転送と呼んでいます．**図8**がバースト転送時のPCIバスのタイミング例です．この図からわかるように，アドレス・フェーズから最後のデータ・フェーズの直前のデータ転送まで，FRAME#がアサートされ続けます．

もう一つ注意が必要な場合があります．イニシエータが出力するFRAME#とIRDY#のアサート/ディアサート・タイミング波形のいくつかを**図9**に示します．一見してわかるように，**図9**(**b**)セットでは，バースト転送でもないのにFRAME#が数クロックの間アサートされることがあるのです．

よって，ADRS_COMPAREステートからそのままBUS_IDLEステートにもどってしまうと，バースト転送時や単一データ・フェーズのトランザクションでも，FRAME#が数クロックの間アサートされるようなイニシエータの場合，FRAME#がアサートされている途中をアドレス・フェーズだと誤って認識し，再度ADRS_COMPAREステートに移行してしまいます．

そこで，自分に対するアクセスではなかった場合は，そのトランザクションが終了するまで待つ必要があります．

PCIバスのトランザクションが終わったかどうかはバス・アイドル状態，すなわちFRAME#とIRDY#がともにディアサートになったことで判定します．

このステートでは，これを検出するまで何らかのトランザクションがほかのエージェントによって行われていると判断してこのステートにとどまり，トランザ

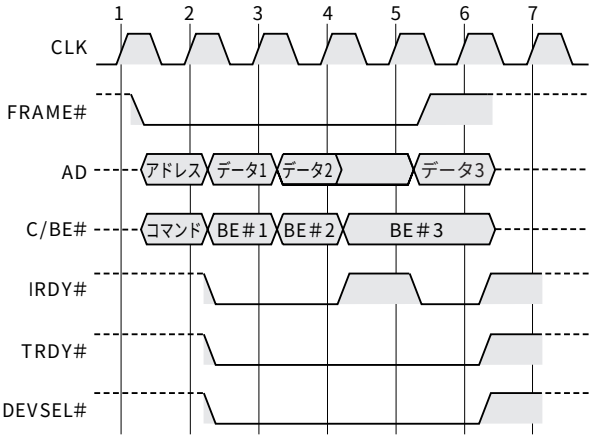
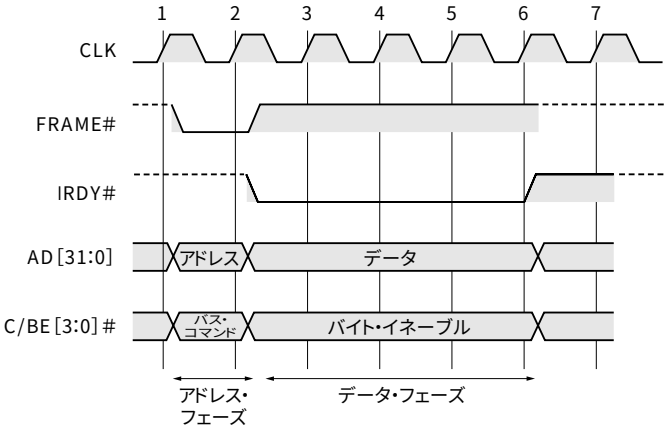


図8
バースト転送時のPCIバスの動き



（**a**）アドレス・フェーズでFRAME＃が1クロックだけ“Ｌ”

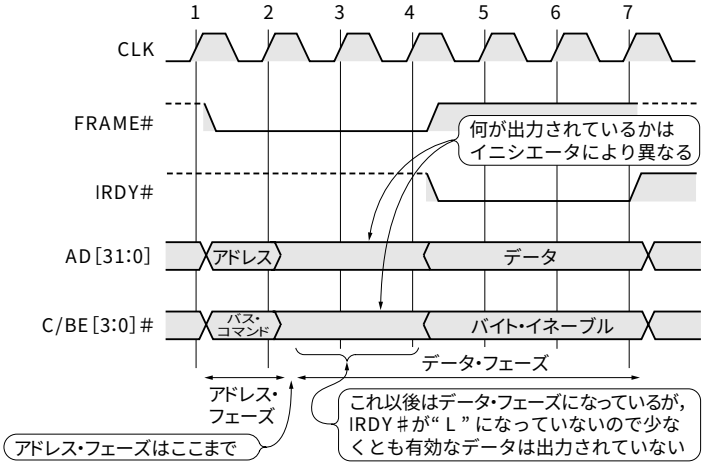


図9
トランザクション開始時の
FRAME#の動き

（**b**）アドレス・フェーズ後もFRAME＃が数クロック“Ｌ”

クションが終了したらBUS_IDLEステートに移行します．

● WAIT_IRDYステート

WAIT_IRDYステートは，イニシエータ側のデータ転送準備が整ったことを示すIRDY#信号のアサートを待ち，アサートされたらローカル・バス・シーケンサを起動させるステートです．

図7を見ると，DEVSEL応答を返す段階で，すで

にIRDY#がアサートされている(クロック4)から，あえてこのステートで明示的に調べる必要はないように思えます．しかし，**図9**(**b**)のような，アドレス・フェーズから数クロック後にイニシエータ側のデータ送受信準備が整うデバイスが存在します．

図9(**b**)の波形とここで設計したシーケンサを照らし合わせると，このWAIT_IRDYステートは，**図9**(**b**)のクロック4に位置します．つまり，**図9**(**a**)のタイミングではほとんどの場合，このステートにくる段階ではすでにIRDY#がアサートされているのに比較し，**図9**(**b**)のタイミングではまだIRDY#がアサートされていない可能性があるわけです．

よって，WAIT_IRDYステートでは，IRDY#信号がアサートされるまで，つまりイニシエータ側のデータの転送準備が整ったということが通知されるまではこのステートにとどまります．

IRDY#がアサートされたら，次はいよいよLEDの点灯制御処理が必要です．LED点灯制御回路は別のブロックで定義しているので，このターゲット・シーケンサで行うことは，その別ブロックに対して「動き出してもいいよ」という起動許可のフラグをセットすることです．

それが，LOCAL_Bus_Startという名前で宣言されたフラグ・レジスタに“1”をセットすることです．これでLED点灯制御回路，つまりローカル・バス・シーケンサが動き出します．

LOCAL_Bus_Startフラグをセットしたら，次はWAIT_LOCAL_ACKステートに移行します．

● WAIT_LOCAL_ACKステート

LED点灯制御回路の動作が完了したかどうか，つまりローカル・バス・シーケンサのデータ転送が終了したかどうかを待ちうけるステートです．

すでにローカル・バス・シーケンサはスタートしたので，直前のWAIT_IRDYステートでセットしたLOCAL_Bus_Startフラグをクリアし，ローカル・バス・シーケンサのデータ転送完了フラグLOCAL_DTACKが“1”になるのを待ちます．

このフラグがセットされるまで，何もせずにこのステートにとどまり続けているので，その間イニシエータはIRDY#をアサートし続けたまま，ターゲットからのデータ転送終了を示すTRDY#信号がアサートされるのを待っています．つまり，データ・フェーズが継続されています．

そして，LOCAL_DTACKフラグが“1”にセットされたら，データ・フェーズを正常に終了したことをイニシエータに通知するためにTRDY#信号をアサートし，ACC_COMPLETEステートに移行します．

● ACC_COMPLETEステート

直前のWAIT_LOCAL_ACKステートでTRDY#をアサートしましたが，その瞬間にイニシエータがデータ転送の完了を認識するわけではありません．

PCIバスはクロック同期ですから，実際にはこのACC_COMPLETEステートのクロックで，イニシエータはTRDY#がアサートされたことを認識します．

イニシエータが正しくTRDY#を認識しているはずなので，ターゲットもTRDY#とDEVSEL#をディアサートしてデータ転送を完了します．

ここが同期回路設計の初心者，またPCIバスの初心者が不安に思う点の一つでしょう．ほんとうにイニシエータはここでTRDY#を認識してくれるの？IRDY#がディアサートされるのを確認してからTRDY#をディアサートしたほうが安心だ……気持ちはわかりますが，それでは同期回路設計ではありません．安心してここでディアサートしてください．

さて，データ転送が終わったからといって，すぐにBUS_IDLEステートに戻ることはできません．“H”にした次のクロックで信号のドライブを切り離し，ハイ・インピーダンスにする処理，サスティンド・トライステートの処理がまだ終わっていません．トランザクションを完全に終了するために，次にTURN_AROUNDステートに移行します．

● TURN_AROUNDステート

このTURN_AROUNDステートでは，直前のACC_COMPLETEステートでディアサートしたDEVSEL#とTRDY#のドライブを切り離し，ハイ・インピーダンス状態にします．そして，BUS_IDLEステートへ移行します．ACC_COMPLETEステートでディアサート(“H”)し，このステートでハイ・インピーダンスにすることで，サスティンド・トライステートが実現できます．

ステート・マシンを知っている読者の中には，たんにディアサートした信号線をハイ・インピーダンスにするなら，ACC_COMPLETEステートから直接BUS_IDLEステートへ移行して，BUS_IDLEステートで行ってもよいのではないかと思われるでしょう．たしかに，そうやってステート数を削ることは可能です．

しかしここでは，自分の出力するDEVSEL#やTRDY#信号をハイ・インピーダンス状態にするス

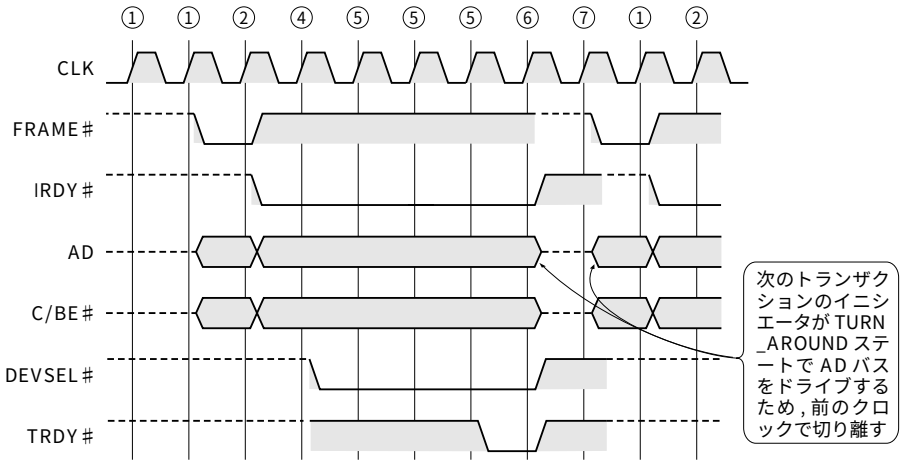


図10
トランザクションが連続して発生した場合

テートを用意することで，トランザクションの終了処理を明確に定義するとともに，BUS_IDLEステートは，あくまでアドレス・フェーズの開始を待つ処理だけをさせることで，それぞれのステートの本来の目的に専念できるわけです．はじめてPCIバスを理解しようとする場合には，これはたいへん重要なポイントだと思います．

また，TURN_AROUNDステートを設けたことにより，連続して発生するトランザクションを正しく認識できるのかどうか心配になります．**図10**に，あるトランザクションの後半と，それに続くトランザクションが連続して発生したときのタイミングを示します．

1) WAIT_LOCAL_ACKステートでローカル・バス側の処理の終了通知を受けるとTRDY#をアサートし，ACC_COMPLETEステートに移行する(クロック1)
2) クロック2でTRDY#をディアサートし，TURN_AROUNDステートに移行する
3) クロック3でDEVSEL#とTRDY#をハイ・インピーダンス状態にする．これでこのトランザクションのすべての処理を終了し，BUS_IDLEステートへ戻る．トランザクションが連続する場合はこのクロックでFRAME#がアサートされ，アドレス・フェーズが始まる
4) クロック4ですでにFRAME#がアサートされているため，トランザクションが開始されたと判断し，ADバスとC/BE#信号の値を内部レジスタに保持し，ADRS_COMPAREステートに移行する
5) 以降はこれまでの説明と同様に，アドレス・デコードの結果によりBUS_BUSYステートまたはWAIT_IRDYステートに移行する

このようにTURN_AROUNDステートを導入しても，連続するトランザクションを正しく認識することが可能です．

2.3　アドレス・デコーダ

ターゲット・シーケンサの説明では，BUS_IDLEステートにおいてアドレスとバス・コマンドを取り込み，次のADRS_COMPAREステートでデコード結果のHit_Deviceを見て判定する，と説明しました．このとき，このステートの間でひそかにアドレス・デコードを行っているのが，このアドレス・デコード・プロセスです．**リスト4**にアドレス・デコード・プロセスのVHDLソースを示します．

アドレス・デコードでは，PCI_AddressとPCI_BusCommandをデコードして，ターゲット1の仕様である，

- アドレスが8000_0000h番地か
- バス・コマンドがメモリ・ライト・コマンドか

の二つの条件が成立するかどうかをつねに調べています．このソースは，回路的には**図11**のようになります．つまり，ターゲット・シーケンサのBUS_IDLEステートでアドレスとバス・コマンドを取り込むと，その値がそのままこのアドレス・デコード回路に入力されます．回路図を見てわかるように，クロック同期でも順序回路でもない組み合わせ回路です．ゲート遅延時間後にはデコード結果であるHit_Device信号が出力されます．

VHDLのソース・リストだけを見ていると，ハードウェアというよりはソフトウェアというイメージが

リスト4
アドレス・デコード・プロセスのVHDLソース

```
-- **********  アドレス・デコード・シーケンサ
Address_Compare : process (
                    PCI_Address,     -- PCIバス・アドレス
                    PCI_BusCommand   -- バス・コマンド
                    )
    constant MEMORY_BASE_ADDRESS : std_logic_vector(31 downto 0) := X"80000000" ; -- デコード・アドレス
begin
    -- メモリ空間へのアクセス・アドレスとベース・アドレスが一致したか
    if (
            PCI_BusCommand(3 downto 0) = PCI_MemWriteCycle-- メモリ・ライト・サイクル
        ) and (
            PCI_Address = MEMORY_BASE_ADDRESS-- ベース・アドレス比較
        )
    then
        Hit_Device <= '1';          -- メモリ・サイクル・ヒット
    else
        Hit_Device <= '0';
    end if;
end process Address_Compare ;
```

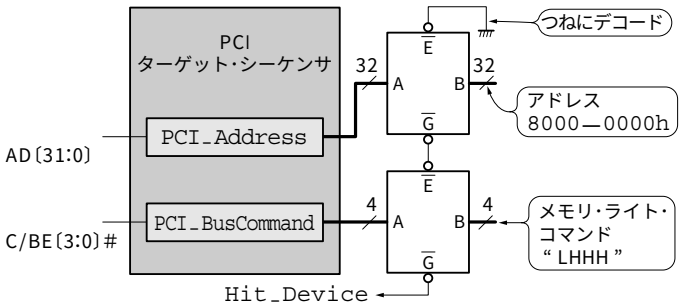


図11
アドレス・デコード・プロセスの回路

先行し，同時に動いている，つねに動いている，という部分が直感的にわかりづらいのではないかと思います．とくに，ソフトウェア技術者にはその傾向が強いのではないでしょうか．

2.4　ローカル・バス・シーケンサ

● LED点灯制御回路

ターゲット1の仕様では，本当に書き込みがうまく行っているかどうかを確認するためにLEDを制御します．このLEDはカソード側をLED_OUTピンに直結し，LEDのアノード側は1kΩ程度の抵抗を間にはさんでV_{cc}電源に接続します．こうすることにより，LED_OUTピンが“L”になればLEDが点灯するというわけです．

● シーケンサの動作

ローカル・バス・シーケンサ，つまりLEDの点灯制御を行うシーケンサです．このシーケンサもPCIバス・クロックに同期して動作させています．**図12**にローカル・バス・シーケンサの状態遷移図を，**リスト5**にローカル・バス・シーケンサのVHDLソースを示します．では，各ステートがどのような意味をもち，どのような処理をしているかを詳しく解説します．

● リセット処理

ローカル・バスにもリセット処理は必要です．リセット信号はPCIバスのリセット信号を使いました．これもクロック・イベント行より先に記述します．

リセット時には，ステート・マシンの状態を示す変

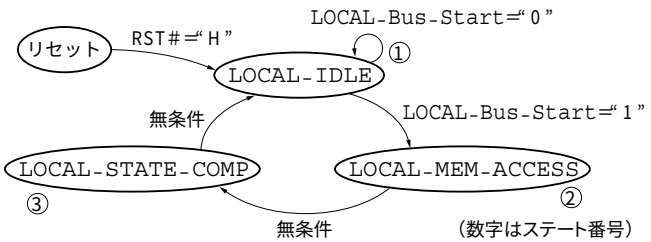


図12
ローカル・バス・シーケンサの状態遷移図

リスト5　ローカル・バス・シーケンサのVHDLソース

```
-- **********  ローカル・バス・シーケンサ
LOCAL_BUS_Seq : process( PCICLK, RST_n )

-- ローカル・バス・シーケンサ ステートバリュー・レジスタ --
    variable LOCAL_CURRENT_STATE  : std_logic_vector (1 downto 0);-- 現在のステート
    variable LOCAL_NEXT_STATE     : std_logic_vector (1 downto 0);-- 次のステート

-- ローカル・バス．シーケンサ ステート・マシン定義
    constant LOCAL_IDLE         : std_logic_vector(1 downto 0) := "00";
    constant LOCAL_MEM_ACCESS   : std_logic_vector(1 downto 0) := "01";
    constant LOCAL_STATE_COMP   : std_logic_vector(1 downto 0) := "11";

begin

-- ********** リセット時動作 ********** --
    if ( RST_n = '0' ) then-- PCIバス・リセットがアサートされたとき

          -- ステートバリュー・レジスタ クリア
          LOCAL_CURRENT_STATE := LOCAL_IDLE;      -- ローカル・バス・シーケンサ リセット
          LOCAL_NEXT_STATE := LOCAL_IDLE;         -- ローカル・バス・シーケンサ リセット

          LOCAL_DTACK <= '0' ;-- ローカル・バス・シーケンサ データ転送完了フラグ クリア

          LED_OUT <= '1'; -- LED消灯

-- ********** ローカル・バス・シーケンサ ステート・マシン ********** --
    elsif (PCICLK'event and PCICLK = '1') then

          LOCAL_CURRENT_STATE := LOCAL_NEXT_STATE ;
          case LOCAL_CURRENT_STATE is

    -- ********** LOCAL_IDLE時の動作 ********** --
          when LOCAL_IDLE =>-- ローカル・バス・シーケンサ スタート指示待ち

                if (LOCAL_Bus_Start = '1' ) then -- ローカル・バス・シーケンサ スタート!
                     LOCAL_NEXT_STATE := LOCAL_MEM_ACCESS;-- メモリ・アクセス・ステートへ
                else-- ローカル・バス・シーケンサ スタート・フラグがまだならこのステートにとどまる
                     LOCAL_NEXT_STATE := LOCAL_IDLE;
                end if;

    -- ********** LOCAL_MEM_ACCESS時の動作 ********** --
          when LOCAL_MEM_ACCESS =>          -- メモリ・サイクル

                LED_OUT <= not PCIAD(0);    -- LED点灯制御
                LOCAL_DTACK <= '1' ;-- ローカル・バス・シーケンサ データ転送完了フラグ セット
                LOCAL_NEXT_STATE := LOCAL_STATE_COMP;

    -- ********** LOCAL_STATE_COMP時の動作 ********** --
          when LOCAL_STATE_COMP => -- ローカル・バス・アクセス完了

                LOCAL_DTACK <= '0' ;-- ローカル・バス・シーケンサ データ転送完了フラグ セット
                LOCAL_NEXT_STATE := LOCAL_IDLE;

    -- ********************************************* --
          when others => null;  -- これ以外の値では何もしない場合でも必ず入れる

          end case;

    end if;

end process LOCAL_BUS_Seq ;
```

リセット時の処理

クロックの立ち上がりに同期して動作するステート・マシンの記述

LOCAL_IDLEステート

LOCAL_MEM_ACCESSステート

LOCAL_STATE_COMPステート

数をローカル・バス・アイドル状態にセットし，内部変数をクリアします．また，リセットということでLEDを消灯します．

● LOCAL_IDLEステート

ここでは，LOCAL_Bus_Startフラグに“1”がセットされるまで待ち，セットされたらLOCAL_MEM_ACCESSステートへ移行します．

● LOCAL_MEM_ACCESSステート

ここでは実際に，LEDの点灯を制御します．仕様では最下位ビットであるAD0が“1”なら点灯，“0”なら消灯なので，LEDを実際に点灯させるLED_OUTピンの論理内容とちょうど逆であることがわかります．

そこで，AD［0］の内容を反転してLED_OUT信号に出力します．そして，必要なデータ転送が終了したので，LOCAL_DTACKフラグに“1”をセットして，LOCAL_STATE_COMPステートへ移行します．

● LOCAL_STATE_COMPステート

ターゲット・シーケンサはLOCAL_DTACKフラグを

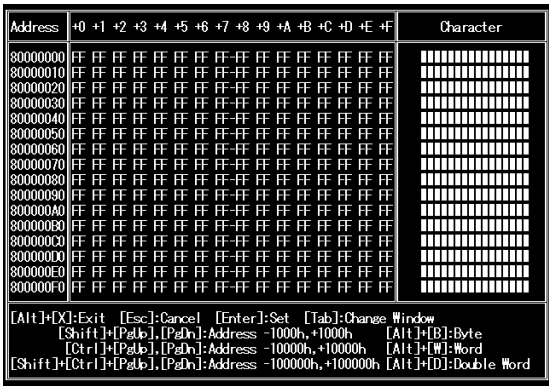


図13　物理メモリ・エディタ起動時の画面

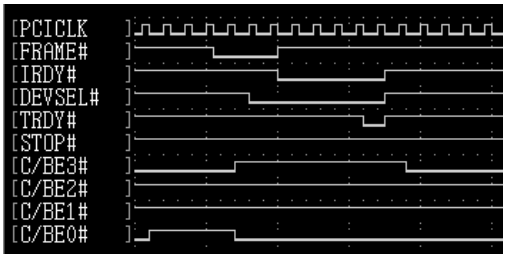


図14　ターゲット1のメモリ・ライト・サイクル時のPCIバスの動き

受け取ったはずなのでフラグをクリアし，LOCAL_IDLEステートに移行します．

● LED点灯制御の実験

それでは実際に，このボードをPCIバスを装備したPC/AT互換機に差し込み，LEDの点灯を制御してみます．まず，MS-DOSを起動して，参考文献(1)の第10章で解説されている物理メモリ・エディタMEMORY.COMを次のように起動します．

```
>MEMORY 80000000 ↵
```

コマンドライン・オプションの数値は，エディット画面を表示するアドレスです．起動すると，FFhばかりが表示される画面になると思います(**図13**)．ここで，アドレス8000_0000hのところにカーソルを移動し，“FF”と入力してみてください．LEDが点灯するはずです．カーソルを戻して“00”と入力するとLEDが消灯します．ただし，ターゲット1はメモリ・リード・サイクルには対応していないので，メモリ・エディタの表示画面はFFhのままです．

このときの実際のPCIバスの動作波形を**図14**に示します．

● メモリ・ライト・サイクルのまとめ

以上，PCIバス・トランザクションのもっとも基本的なメモリ・ライト・サイクルに対応したターゲット・シーケンサの機能について説明しました．何度も説明しているように，PCIバス・トランザクションの基本は，わずか4種類の信号と2種類のバスによって行われています．

この信号線の制御ルールが，PCIバス上におけるあらゆるトランザクションすべてに共通する基本原則なのです(一部の例外をのぞく)．

メモリ・リード/ライト・サイクル── ターゲット2の設計

● ハードウェアの仕様

ターゲット1は書き込み専用のハードウェアでしたが，次はリード/ライトに対応したターゲット2を設計します．デバイスの仕様としては，ターゲット1に読み書き可能な32ビットの内部レジスタを実装して，値の読み書きを行えるようにします．また，そのレジスタの最下位ビットの状態をLEDに表示させるようにしましょう．このときのPCIバスの動作を予想した波形が**図15**になります．

● 機能追加部分

ターゲット1に対して内部に32ビットのリード/ライト・レジスタが実装され，さらにリード機能が付きました．これにより変更が必要な箇所は以下のとおりです．

▶PCIターゲット・シーケンサ・プロセス

リード・サイクルでADバスをターゲットがドライブする

▶アドレス・デコード・プロセス

バス・コマンドに“メモリ・リード”を追加する

▶ローカル・バス・シーケンサ・プロセス

内部レジスタに対する読み書き処理を追加する

この程度の機能追加であれば，PCIターゲット・シーケンサもローカル・バス・シーケンサ，ステート・マシンそのものに変更を加えずに実現することができます．

図16に，ターゲット2のブロック図を示します．