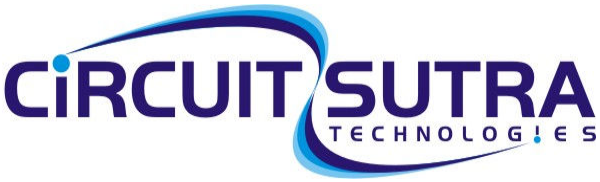
STARC Transaction Level モデリングガイド
第２版　準拠

Demo Model Set

ユーザガイド



Software for Semiconductors

Document History:

Date	Name	Comment	Version
2nd March, 2009	CircuitSutra	Created first draft in English	0.1
20th April, 2009	CircuitSutra	Translation to Japanese	0.1J

目次

1.目的

TLM2.0 および STARC TLM ガイドライン第２版に基づいて開発したデモモデル・セットの構造と機能を説明します。

2.スコープ

このドキュメントでは、STARC ガイドラインに基づいた IP のモデリング方法について説明しています。次の２つのモデルをガイドラインに従って開発しました。

・汎用タイマ （８２５４に基づく）

８２５４の仕様は下記で入手できます。

www.stanford.edu/class/cs140/projects/pintos/specs/8254.pdf

・プログラマブル割り込みコントローラ（PIC）（８２５９に基づく）

８２５９の仕様は下記で入手できます:

www.ee.hacettepe.edu.tr/~alkar/ELE414/8259.pdf

注記：タイマのコードはレビューと単体検証を実施しておりますが、PIC のコードはレビューと単体検証を実施しておりません。IP 再利用の実際をお見せするために使用していることをご了承ください。

以下に TLM2.0 および STARC ガイドラインに基づいた汎用タイマの設計手法を説明します。

3.Timer 8254

3.1 タイマのブロックダイアグラム

下図は８２５４のブロックダイアグラムです。３つのカウンタや Control Word など、８２５４タイマの基本コンポーネントを表しています。機能の詳細については８２５４の仕様書を参照ください。

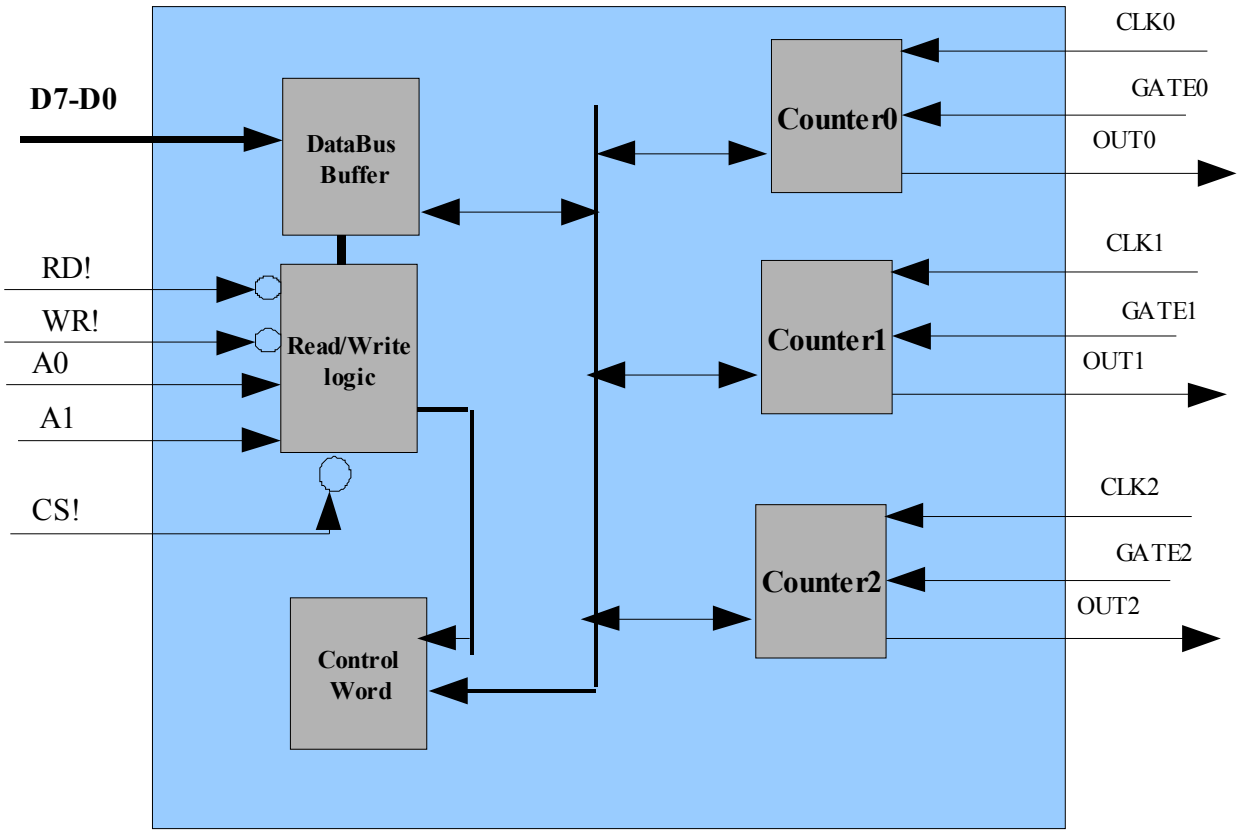


図３-１　Block Diagram of Timer 8254 (Signal Level)

3.2 モデルのアーキテクチャダイアグラム

８２５４タイマは TLM2 ライブラリでモデリングしています。STARC ガイドラインに従い IP の計算部と通信部は分けてモデリングしています。モデルはブロッキング、ノン・ブロッキング両方のトランスポートを扱うことができます。モデルの再利用性と相互運用性を考慮した構造です。以下のブロックダイアグラムは、８２５４タイマ IP の中でも重要なブロックであるタイマとイニシエータ、及びその接続の概要を表しています。

図３-２はタイマの TLM ブロックダイアグラムです。

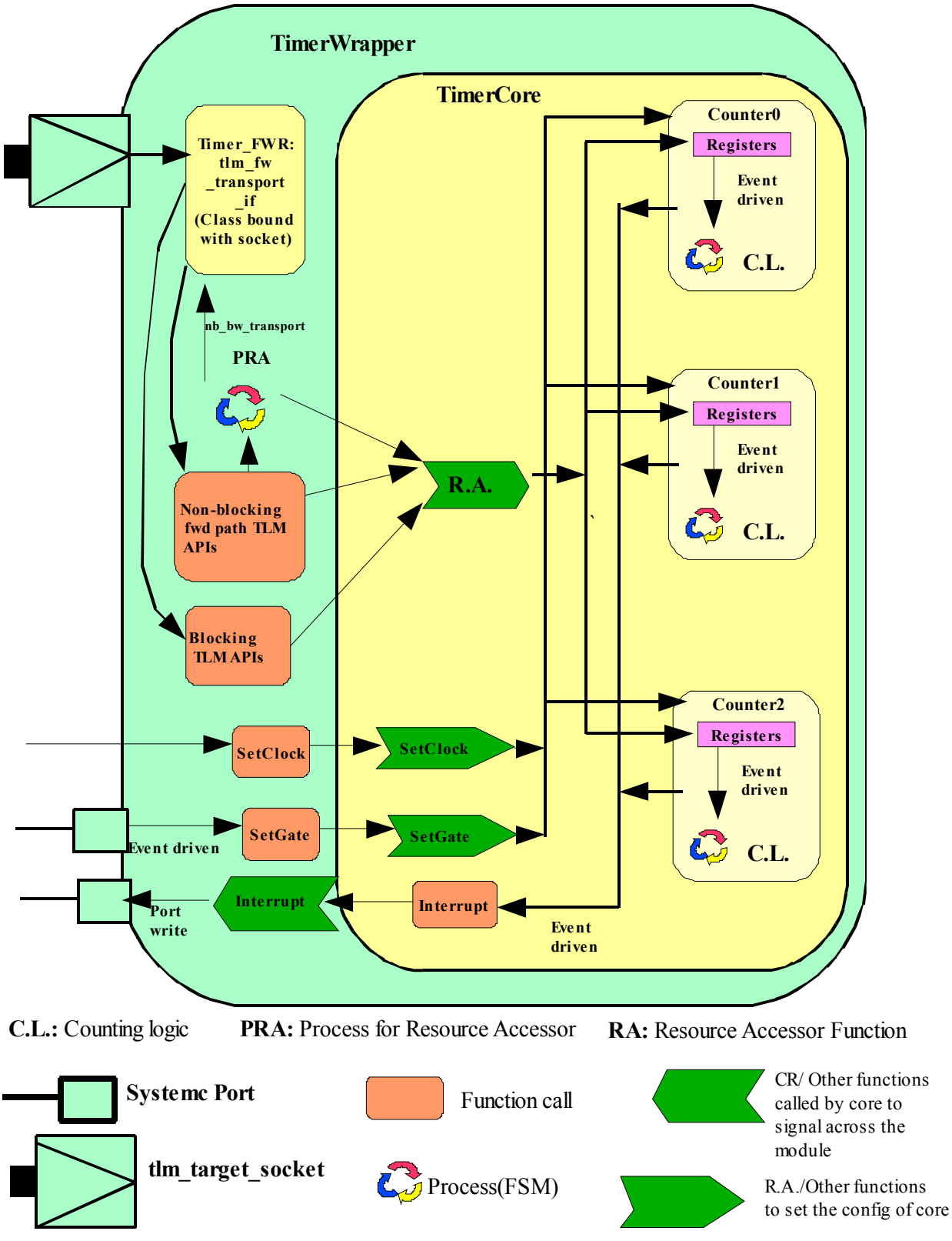


図3-2　Block Diagram of Timer 8254 (TLM Level)

図３-３は、イニシエータのブロックダイアグラムです。次の章でタイマ・モデルで使用するイニシエータ IP の重要な機能ブロックについてその概要を説明しています。イニシエータの作成に当たっても、STARC ガイドラインに適切に従っています。イニシエータの実装はこのタイマ IP を使用するユーザ毎に個別ですので、必要に応じて変更することになります。

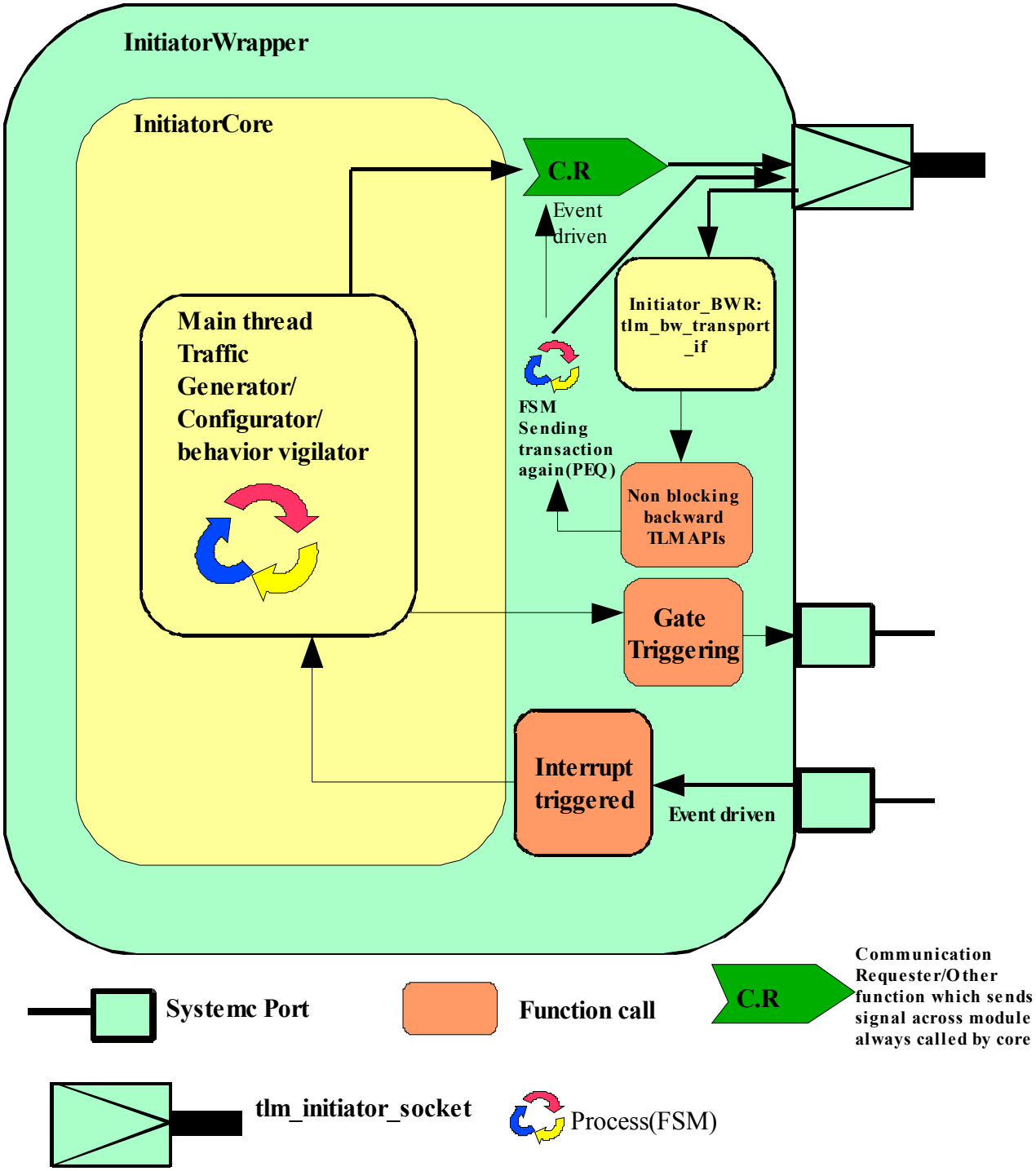


図３-３　Block Diagram of Initiator

図３-４のダイアグラムはタイマとイニシエータ IP の接続を表しています。イニシエータはタイマを各種モードに設定し、read/write 要求トラフィックの生成と、タイマ動作の観測をします。エラーコンディションに対する通知も発します。

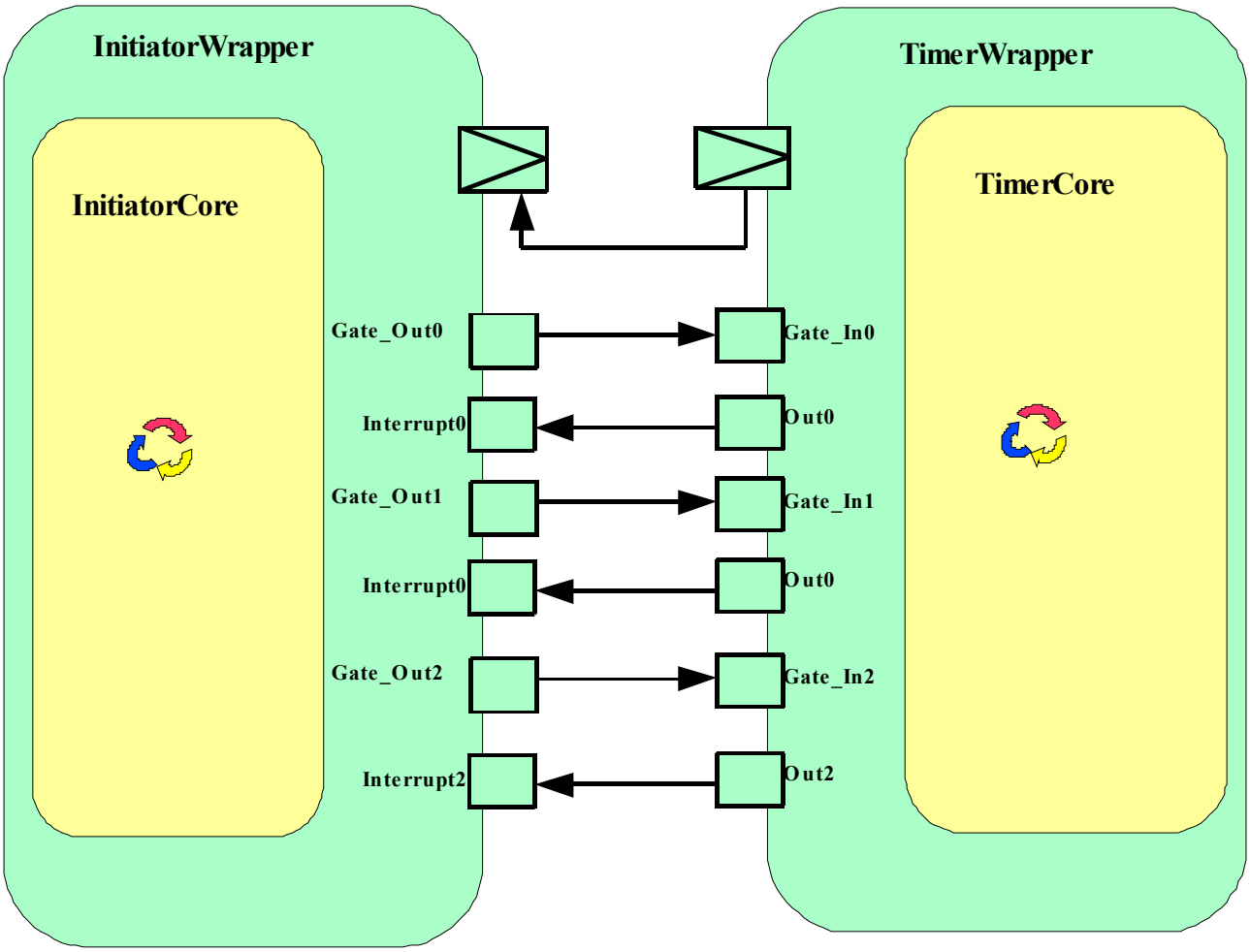


図３-４　Connection of the timer with initiator IP

4. デモデモルが対応している STARC TL ガイドラインの指針

4.1 通信機能と計算機能の分離

STARC ガイドラインは計算部の再利用性を高めるため計算部と通信部の分離を推奨しています。計算部はコアで、通信部はラッパで定義します。計算部と通信部は相互接続します。ガイドラインの５．３．２章に従っています。

4.1.1 コア

STARC ガイドライン５．３章には、コアの作り方と、どのようなブロックをコアに入れ込むべきかという指針が書いてあります。

デモモデルでは、イニシエータ・コアにはポートや TLM ソケットがありません。シンプルにラッパにくるまれており、これが SystemC ポートと TLM ソケットを持っています。メインのトラフィック生成部はイニシエータ・コアの中にあり、これがラッパで定義している CR[communication requester]関数をコールします。ラッパに定義しているポートに値を read/write する場合にも、コアはラッパのポート・アクセス関数をコールします。

ターゲット側ではタイマ・コアを作成していますが、これもタイマ・ラッパにくるまれています。タイマ・ラッパに read/write リクエストが来る毎に、タイマ・コアの RA[Resource Accessor]関数をコールします。デモモデルでは、タイマ・コアにはカウンタがあり、これらがメインの演算ロジックとなります。タイマ・コアの RA 関数は基本的にはカウンタのレジスタにアクセスし、要求されたオペレーションを実行します。カウンタはコントロールレジスタに書かれたカウント値のカウントを開始します。

4.1.2 ラッパ

タイマとイニシエータのラッパ作成にも STARC ガイドラインを参照しました。５．４章です。

イニシエータ・ラッパは tlm_initiator_socket をインスタンスしており tlm_bw_transport_if API を実装しています。また、イニシエータ・ラッパは CR 関数を備えており、トランザクション実行のためにコアが呼び出します。CR 関数では、ソケット経由の TLM API が使われています。

タイマ・ラッパは tlm_target_socket をインスタンスしており、tlm_fw_transport_if API を実装しています。タイマ・ラッパはトランスポート・コールを受け取るたびにタイマ・コアの RA 関数をコールします。

4.2 TLM API 用クラスの分離

STARC ガイドラインで推奨している通り TLM API のセットはラッパに直接実装するのではなく、別のクラスに定義するほうがよいでしょう。このようにすることにより、複数のソケットがある時には複数の TLM API のセットをもつことができます。このクラスをラッパにインスタンスするのです。

イニシエータ側には Initiator_BWR クラスを作りました。これをイニシエータ・ラッパ内にインスタンスし、 tlm_initiator_socket にバインドします。 この Initiator_BWR クラスは tlm_bw_transport_if インターフェイスを継承し、バックワード TLM API の定義を実装します。

ターゲット側には Timer_FWD クラスがあります。タイマ・ラッパ内にインスタンスし、tlm_target_socket にバインドします。Timer_FWD は tlm_fw_transport_if を継承し、フォワード TLM API の定義を実装します。

TLM プロトコルは標準のもので、IP の機能とは独立しています。ターゲット側では TLM API の実装も IP から切り離しました。TLM API の実装はターゲット・ラッパの外側で行い、同じ実装をタイマと IP の双方で使用しています。同様に汎用用途の TLM 実装をイニシエータ側でも使うことができます。

4.3 抽象レベル（データ粒度）

デモモデルは TR(transaction)と BP(Bus Phase)の２つのデータ粒度をサポートしています。イニシエータの例には、どの抽象レベル（データ粒度）を用いるかを設定する変数があります。TR の時には、イニシエータはタイマ IP と通信する際にブロッキング TLM コールを使います。BP の時にはノンブロッキング TLM コールを使います。データ粒度を設定するには、SetAbstraction() を適当な値でコールするだけで TR もしくは BP に設定できます。BP でのフェーズ数はレイテンシにより決まります。

4.4 抽象レベル（タイミング）

デモモデルは untimed と approximate timed モデルをサポートしています。基本的には UTTR, ATTR, ATBP の抽象レベルをサポートしています。タイミング抽象度はレイテンシによります。 タイマには以下に説明するように３種類のレイテンシを設定することができます。

1) Request Accept Delay: イニシエータからリクエストを受け取った後、イニシエータに対し応答を返す際にターゲット（今回のモデルの例ではタイマ）が消費する時間。またはノンブロッキング・モードの通信におけるリクエスト・フェーズの開始から終了までの時間と定義することもできます。

2)Read Response Latency: ターゲット IP が read リクエストの処理に要する時間。read タイプのリクエストにおけるリクエスト・フェーズの終わりとレスポンス・フェーズの開始との間の時間。

3) Write Response Latency: ターゲット IP が write リクエストの処理に要する時間。write タイプのリクエストにおけるリクエスト・フェーズの終わりとレスポンス・フェーズの開始との間の時間。

イニシエータ側には、１種類のレイテンシがあります。

1) Response Accept Delay: イニシエータがターゲットの応答を受け取るまでに要する時間。ターゲットは、イニシエータが前のトランザクションの応答の受け取りを完了するまで、次のトランザクションの応答を送ってはいけません。レスポンス・フェーズの開始から終了までの時間と定義することもできます。

デモモデルはこれらレイテンシの組み合わせのバリエーションによって、自動的に適当な抽象レベルに切り替わるように作ってあります。

もしデータ粒度抽象レベルが TR で全てのレイテンシが０であれば、UTTR として動作します。データ粒度抽象レベルが TR で一部もしくは全てのレイテンシが０以外であれば、ATTR として動作します。

データ粒度抽象レベルが BP である場合、レイテンシの組み合わせにより様々な TLM シーケンスが考えられます。STARC ガイド６．２章では、ATBP レベルで様々なフェーズを導入する方法を提案しています。ATBP レベルでの完全なトランザクションシーケンスは次の通りです。

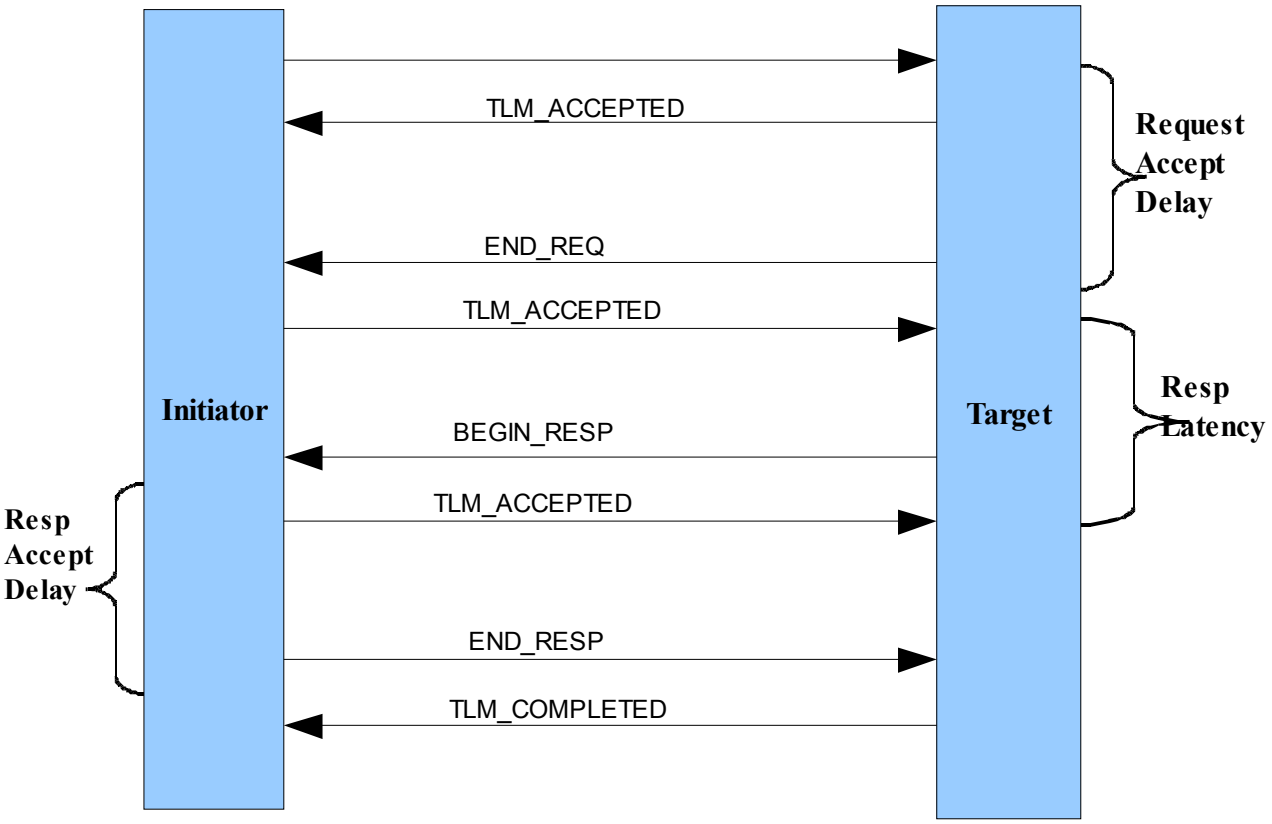


図４-１　Complete Sequence of Transaction at ATBP Level

これが、全てのレイテンシが定義されている場合のフェーズの完全な流れです。しかし

いずれかが欠けている場合にはトランザクションは STARC ガイド４．１章（4-18 ページ）に説明されているようにいずれかのショートカットに従います。

次の表はレイテンシの各コンディションによって成立するショートカットのバリエーションです。

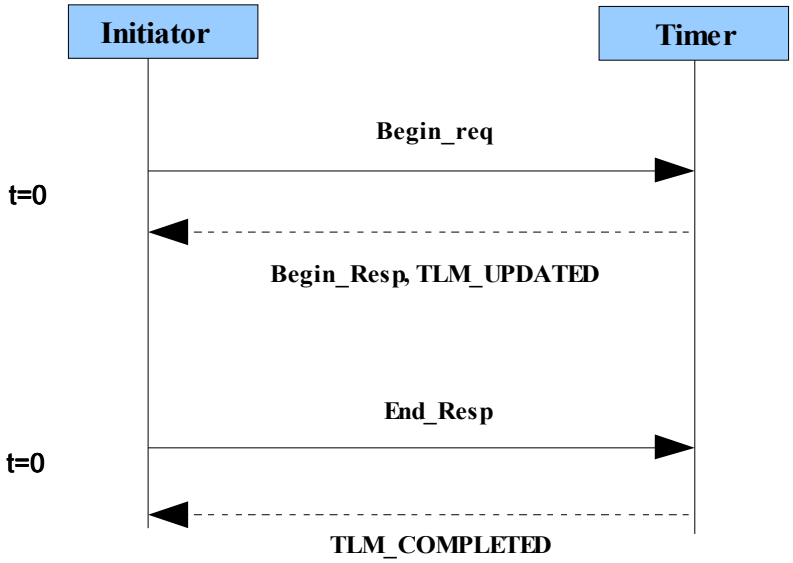
Case	Request Accept Delay	Response Latency	Response Accept Delay	ShortCut
Case1	0	0	0	Shortcut 5
Case2	0	0	1	Shortcut 5
Case3	0	1	0	Shortcut4
Case4	0	1	1	Shortcut5
Case5	1	0	0	Shortcut3
Case 6	1	0	1	Shortcut5
Case7	1	1	0	Shortcut4
Case8	1	1	1	Complete Sequence

注記: 表の中で、'１'は'０'でない値を意味します。 時間数値を表しているのではありません。

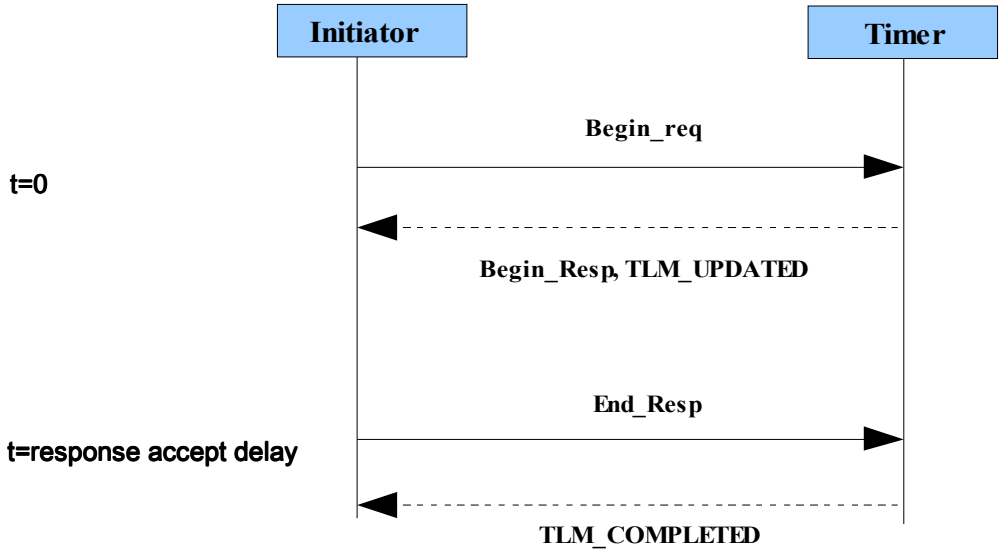
注記: この事例では、ユーザが設定したレイテンシのみを基準にしてショートカット・シーケンスを実装しましたが、モデリングした IP の機能によってはある特定のシーケンスを選択するにあたって他に様々な基準があるでしょう。

表にある各ケース毎のフォワード／バックワード・コールのシーケンスダイアグラムを示します。

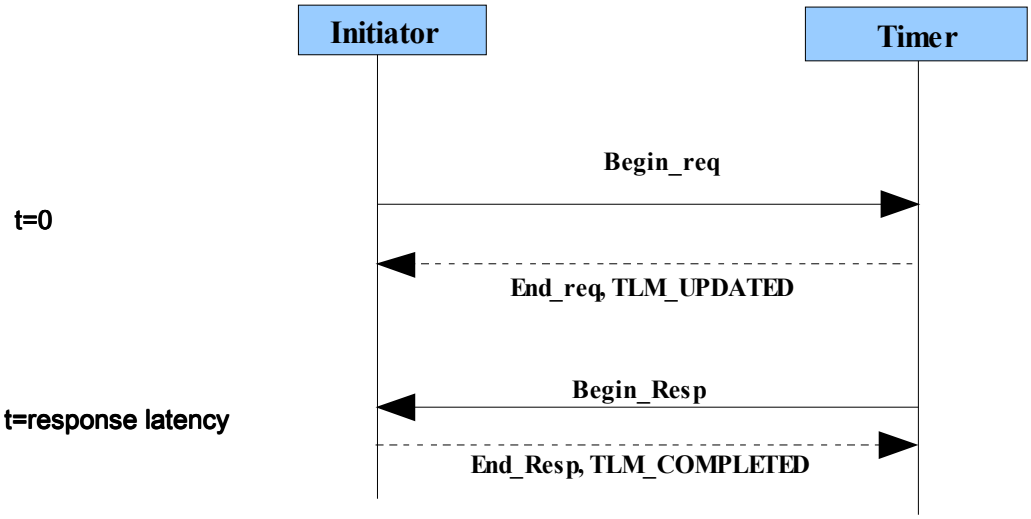
Case 1: [request accept delay=0, response latency=0, response accept delay=0]



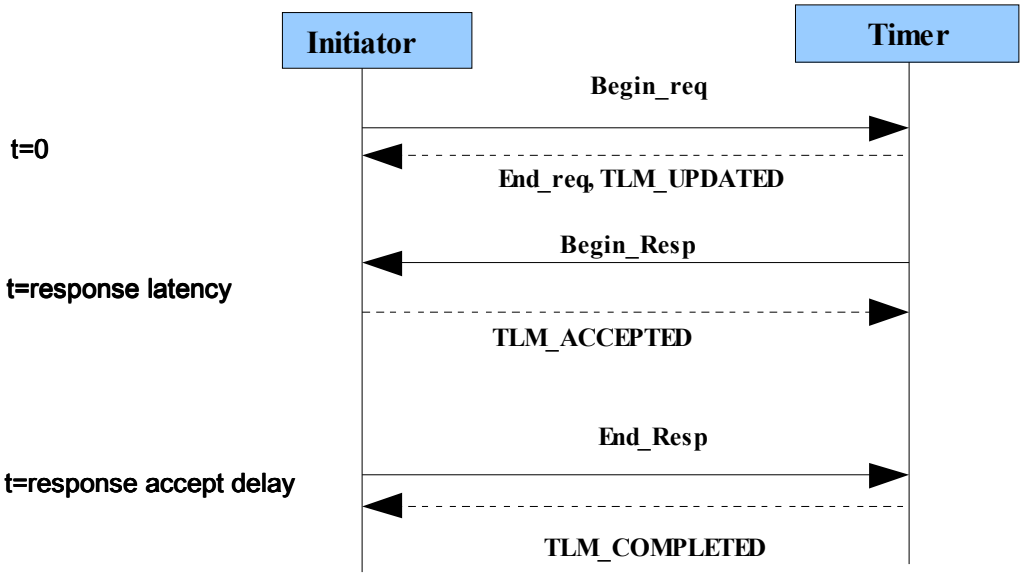
Case 2: [request accept delay=0, response latency=0, response accept delay =1]



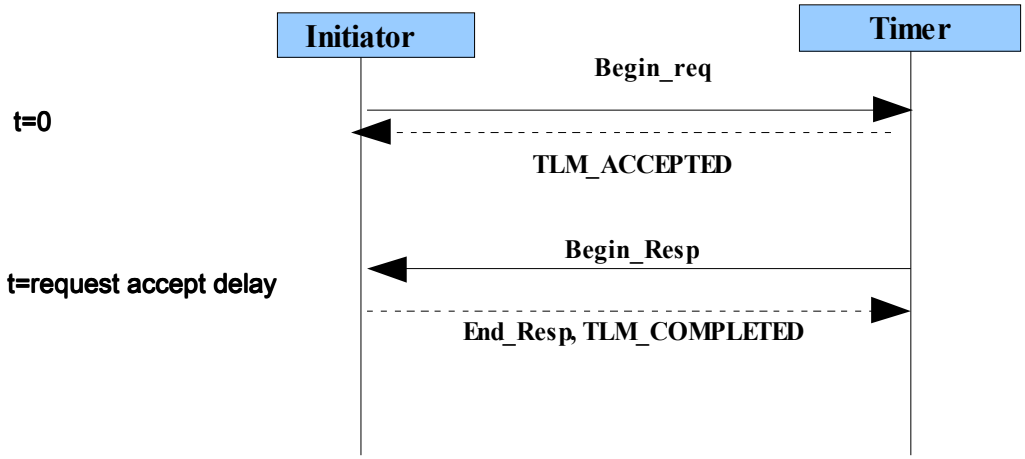
Case 3: [request accept delay=0, response latency=1, response accept delay=0]



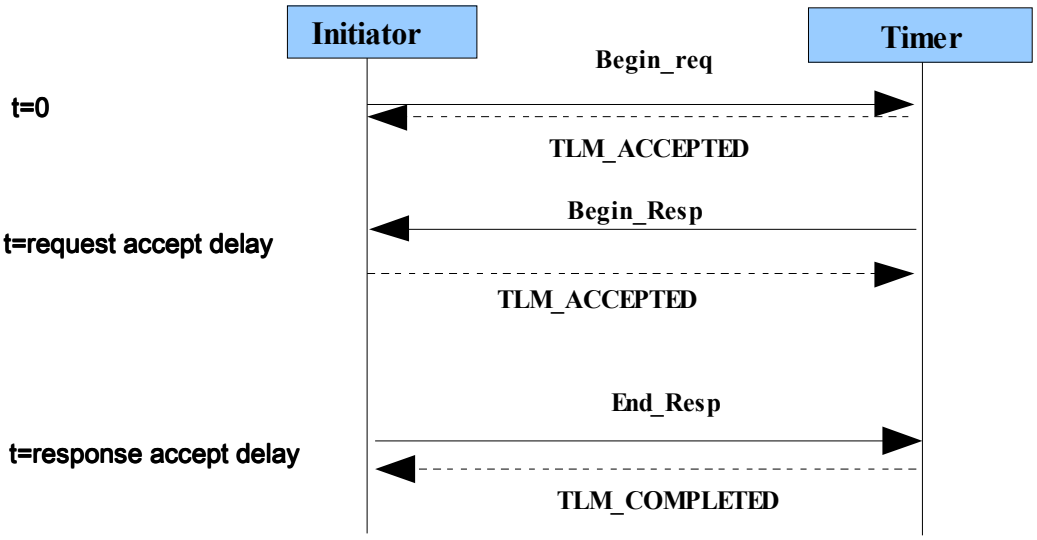
Case 4: [request accept delay=0, response latency=1, response accept delay=1]



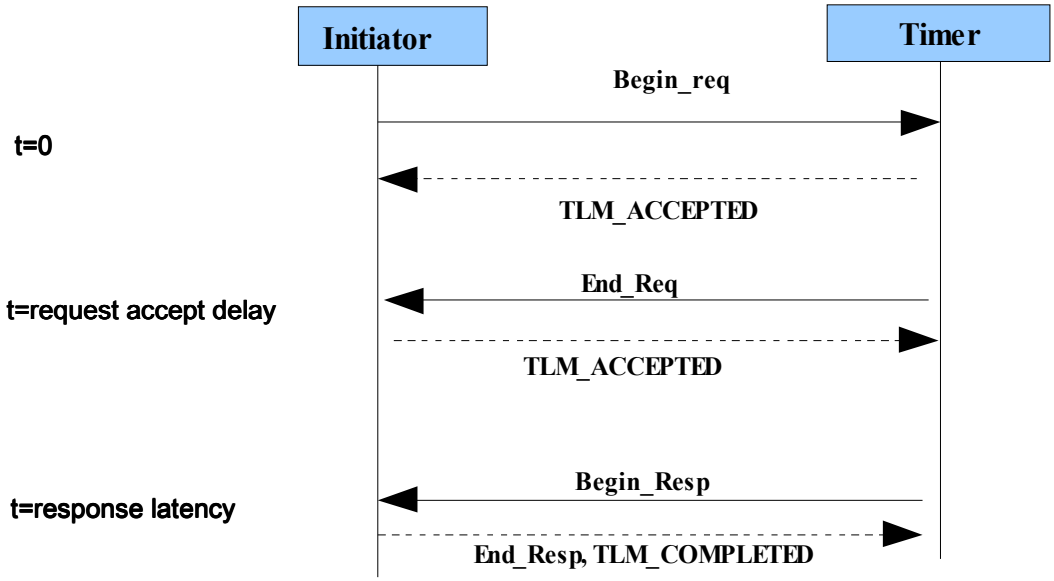
Case 5: [request accept delay=1, response latency=0, response accept delay=0]



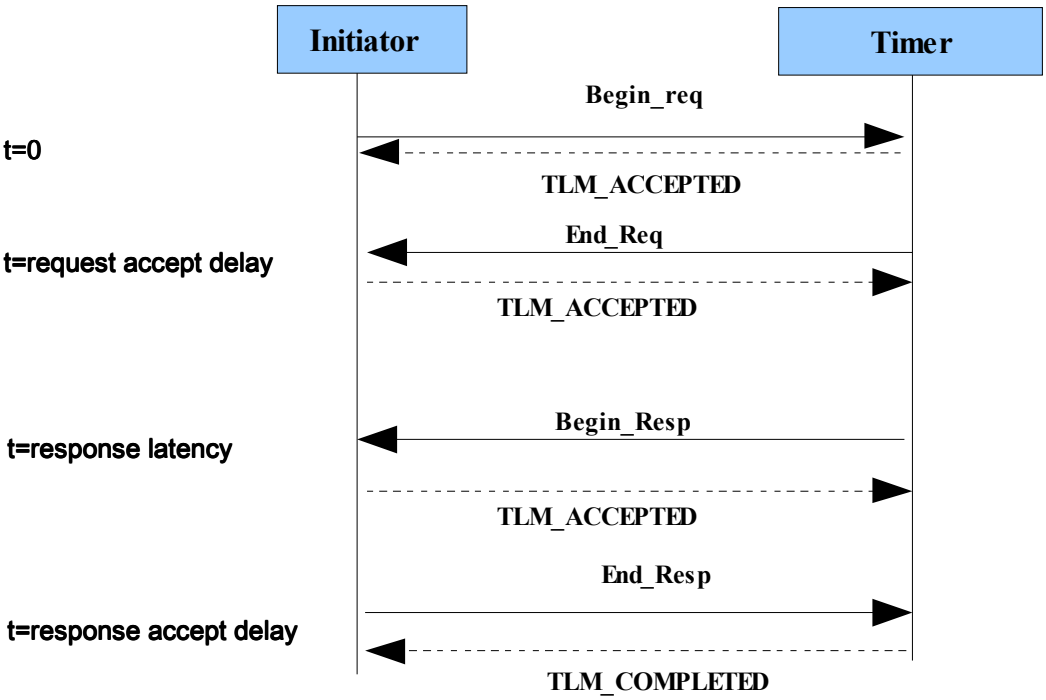
Case 6: [request accept delay=1, response latency=0, response accept delay=1]



Case 7: [request accept delay=1, response latency=1, response accept delay=0]



case 8: [request accept delay=1, response latency=1, response accept delay=1]



4.5 通信機能と計算機能の並列実行

STARC ガイド 6.2.3 章では、イニシエータ側、ターゲット側双方での通信と計算の並列実行について説明しています。 デモモデルの中でもこの並列実行を実装しました。イニシエータ側では送信リクエストと受信応答で別々のプロセスを保持することで実現しています。同様にタイマ側ではリクエストを異なるプロセスで受け取り、応答をそれぞれ別のプロセスに返します。

Initiator:

現在のデモモデルでは、イニシエータ側で ATBP モード用の基本的な TLM ハンドリングコードがリクエストと応答にそれぞれ別のプロセスを持つことにより並列実行をサポートしています。直前のトランザクションが終了していなくても次のトランザクションを開始する設定を可能にするので、様々な並列トランザクションが可能になります。その際 TLM2.0 ルールに従う必要があります。すなわち：

-イニシエータは直前のトランザクションの END_REQ/BEGIN_RESP フェーズを受け取るまでは、次のリクエストを発行してはならない。[ref TLM2 User guide sec 7.2.4 b]

ただし今回の example では並列トランザクションは起動していません。イニシエータ・ラッパからのコントロールはリクエストが終了してからコアに届きます。

Timer:

ターゲット側の並列実行は２通りで実現しています。
1.タイマのカウントロジック（演算部）は排他的に動作し通信とは独立しています。

2. ATBP の時には、リクエストと応答は別のプロセスで扱います。ペンディングリクエストとプロセスを保持するために PEQ（payload event queues)を使っていますが、これが保持したリクエストを処理してイニシエータに応答を送ります。タイマにおいても次のルールに従う必要があります：

- ターゲットは直前のトランザクションの END_RESP を受け TLM_COMPLETED を返すより前に BEGIN_RESP フェーズの応答を返してはいけない。[ref TLM2 user guide sec 7.2.4 c]

4.6. 単体検証

STARC ガイドではモデルの機能検証にあたって、全体検証環境だけではなく単体検証環境も作ることを推奨しています。８２５４タイマの主な機能は全て単体検証でカバーしています。単体検証においては、共通のトップレベルモジュール Testbench を作り、テストケースのレベルでは testcase クラスがあります。testbench クラスは全てのテストケースに共通です。

Testbench: テストベンチモジュールにはイニシエータとタイマをインスタンスし、接続しています。それ以外にテストベンチクラスには外部からコールすることができる共通関数や、イニシエータとタイマ IP の全てのレイテンシ、ベースアドレス、クロック周波数、データ粒度(TR/BP)などを設定するための派生クラスなどがあります。仮想 run()メソッドを持っており、派生クラスから特定の定義付けを与えます。

Testcase: 各テストケースクラスはテストベンチが駆動します。各テストケースは親クラスの run メソッドの個別の定義を実装します。各テストケースではレイテンシとクロック周波数の設定は異なっていてもかまいません。タイマ IP の各機能用には個別のテストケースがあります。Linux,Windows どちらでもコンパイルすることができます。

5. ディレクトリ構造

デモセットのディレクトリ構造を説明します。使い勝手を考慮して STARC_models ディレクトリにはいくつかのサブディレクトリがあります。

commonCode: 異なる IP 間で共通に使えるコードを格納

Timer_8254: ８２５４タイマに関連するソースコード、ドキュメント、テストケース、example などを格納

/IP: タイマ IP のメインコード、カウンタなど

/common: メインのトラフィックジェネレータであるイニシエータの実装コードを格納。同じイニシエータ・ラッパとコアを examples と unit_test でも使用

/unit_test: ８２５４タイマの各種機能を網羅的に検証するためのテストケース

/examples: タイマ・モデルの様々な例を格納。各例題ごとに docs/ReadMe ファイルで詳しく説明しています。異なる抽象度とデータ粒度におけるモデルの動作を示しています。

/docs: タイマ IP の機能リストと、それら機能に対応するテストケースの一覧表

PIC_8259: PIC8259 に関するソースコード、ドキュメント、テストケース、example など

注記: PIC はソースコード・レビューやユニットレベルのテストを行っていません。このリリースに含めてあるのは、様々な IP 間でのコード再利用性をお見せするためです。

6.コンパイルと実行の方法：

このデモモデル・セットでは、ソースコードの実装とモデルに関する理解を深めるため unit_test と examples サブディレクトリを設けており、ディレクトリ構造が細かく分かれています。ビルドは以下の説明に従って行ってください。

6.1 Windows:

6.1.1 Solution file を使う方法:

1) example の各ディレクトリには、サブディレクトリ/build-msvc があります。

2) MSVC++9 でソリューションファイルを開きます。

3) Project で右クリックし properties に行きます。C/C++ ->General->Additional Include directories をクリックします。

4) SystemC ヘッダー(systemc.h)をここに加えます。

5) tlm headers (tlm.h)のパスを加えます。

6) linker->general->Additional Library Directories をクリックします。

7) systemc.lib のパスをここに加えます。

8) ソリューションをコンパイルします。

9) 実行します。実行モジュールはお使いの設定により Debug/release ディレクトリに出来ます。

上記ステップ３，４，５，６，７の代わりに次のように操作することもできます。

1) tools->options->Projects と Solutions->VC++ Directories に行きます。

2) Combobox “Show Directories For”で Include files を選び systemc.h と tlm.h のパスを正しく設定します。

3) さらに同じ Combobox で“Library files” を選び“systemc.lib”ライブラリのパスを設定します。

6.1.2 Makefile を使う方法:

1）examples/config_msvc ディレクトリに Makefile.config があります。これを全ての examples で使います。
単体テスト用には、unit_test/ ディレクトリに Makefile_msvc.config があります。

2）使用している環境に合わせ、ファイル中以下の変数の設定を変更します。

a) SYSTEMC_HOME

b) TLM_HOME

c) FLAGS “Microsoft Visual Studio 9.0\VC\include”, “Microsoft Visual Studio 9.0\SDK” など Visual studio で必要な変数を加えます。

d)LDFLAGS Visual Studio SDK library (“Microsoft Platform SDK\Lib”)のパスを使用環境に合わせて設定します。

3) Programs->Visual C++ 9.0 Express Edition ->Visual Studio tools-> Visual Studio 2008 コマンドプロンプトに行きます。

4) コンパイルしたい適当なテストケースに行くか（例：unit_test/Test1/Test1.1）、または適当な examples の build-msvc ディレクトリ（例：examples/Example_nb_000/build-msvc）に行きます。

5) nmake ユーティリティを実行します。

```
>nmake
```

単体テストでは次のように実行します。

```
>nmake /F Makefile_msvc.
```

これで全てのオブジェクトファイルと実行モジュールをつくります。

6.2 Linux:

Linux の手順は簡単です。次の手順に従ってください。

1) unit_test/ または examples/config-linux/ディレクトリに行きます。

2) そこにある Makefile.config を編集し SYSTEMCDIR と TLMDIR 変数をシステム環境に応じて設定します。SYSTEMCLIB も編集します。

3) コンパイルしたい適当なテストケースに行くか（例：unit_test/Test5）、または適当な examples の build-linux ディレクトリに行き（例：examples/Example_nb_000/build-linux）、make コマンドを実行します。

```
>make
```

4) 同じディレクトリに実行モジュールが出来ます。

注記: 全てのテストケースは perl スクリプト”**runAll.pl**”で一括実行することもできます。このスクリプトは Windows と Linux どちらでも使えます。スクリプトは全ての test_case ディレクトリで makefile を探し実行して必要なオブジェクトファイルと実行モジュールを作ります。全てのテストケースを実行し、結果を **runAllStatus** に保存します。ですので **runAllStatus** を見ればいくつのテストがパスし、いくつ fail したのかを解析することができます。どのテストがパスまたは fail したのかも明らかです。

参考文献；

1. **OSCI TLM2 User Manual (version JA22)**
   [http://www.systemc.org/home]

2. **STARC TLM Guide (second edition)**
   [http://www.starc.jp/index-e.html]

3. **Specs of Programmable Interrupt Timer8254**
   [www.stanford.edu/class/cs140/projects/pintos/specs/8254.pdf]

