Tiempo White Paper #2:
Introduction to SystemVerilog Asynchronous Modeling

Version 2.0 – March 16, 2011

www.tiempo-ic.com

Overview

This document is a first introduction on Tiempo SystemVerilog coding style that is used to write high-level synthesizable models of circuits designed in Tiempo unique asynchronous and delay-insensitive technology. Targeted audience includes chip designers and managers who are familiar with hardware description languages. Section 2 lists the key concepts in SystemVerilog that are important for the modeling and the verification of asynchronous systems and section 3 introduces their practical use in a simple design example.

Outline

1 Introduction

Tiempo offers an innovative asynchronous and delay-insensitive design technology, with a portfolio of powerful asynchronous IP cores and a fully automated synthesis tool supporting these cores and their design technology.

Chips designed in this technology and/or integrating these asynchronous cores show outstanding physical properties such as ultra-low power, ultra-low noise, ability to work at ultra-low and variable voltage levels, reactivity (sleep mode by default, immediate wake-up), robustness against process-voltage-temperature (PVT) variations and resistance to hardware attacks (e.g., power analysis, fault injections).

The ground reason for such capabilities is that Tiempo asynchronous circuits are fully clock-less and self-controlled, their behavior being governed by signal transitions rendezvous and signal levels memorization. These principles, implemented with specific signal encoding and glitch-free logic, ensure functional correctness regardless of any actual delay through gates and wires. For more information on the fundamental design techniques used in Tiempo asynchronous circuits, the reader will refer to **Tiempo White Paper #1 "Tiempo Technology Introduction"**.

Tiempo synthesis tool – ACC (Asynchronous Circuit Compiler) – takes as input descriptions written in the standard IEEE 1800 SystemVerilog language at transaction-level modeling (TLM), and generates as output standard Verilog gate-level netlists. The input models are written using a specific coding style as well as predefined SystemVerilog packages provided by Tiempo.

This document introduces Tiempo SystemVerilog asynchronous coding style through a simple design example including a typical finite state machine and a simple operative part.

2 SystemVerilog modeling principles for asynchronous design

2.1 Modeling for synthesis

Tiempo SystemVerilog coding style to model asynchronous designs uses SystemVerilog Transaction-Level Modeling (TLM). TLM is the most suitable abstraction level to model asynchronous designs, which mainly consist of concurrent processes (modeled as SystemVerilog always blocks) communicating through channels (modeled as SystemVerilog interfaces).

2.1.1 Channel definitions

Channels represent the basic medium for communication between asynchronous design entities and processes. A channel allows point to point communication between two processes, each communication through a channel involving a token exchange between the two processes. The process initiating the communication is the active process, the other being the passive process.

This point to point communication uses a handshake protocol that can be of two kinds: push or pull. In the push protocol, the active process writes a token to the channel and the passive process reads a token from the channel; in the pull protocol, the active process reads the token from the channel and the passive process writes the token to the channel.

Channels are modeled as SystemVerilog interfaces. Channel ports are represented by SystemVerilog interface **modport** port declarations. Channels are defined as push or pull channels and with a specific data type.

Tiempo provides the designer with SystemVerilog definition files that predefine an interface (i.e., channel type) for each of the predefined SystemVerilog data types (**bit**, **byte**, **logic**, **reg**, **int**, **shortint**, **longint**, **integer**), as well as macros that enable the definition of channels carrying data of any user-defined type (**enum** or array types for example). Finally, a predefined type (**event_type**) is available to model channels exchanging tokens that do not carry any data value and are used for synchronization purpose.

2.1.2 Channel operations

Channel communications are modeled as read and write operations using methods automatically predefined with each channel type (SystemVerilog interface). Channel operations include:

- **Read** channel operation: this operation enables a process to read a data from a channel; it suspends the reading process until the handshake protocol is completed; after the read operation, the data obtained may be used in the process without blocking the channel writer (this generally incurs a memorization of the read value);
- **BeginRead** and **EndRead** channel operations: these dual operations also enable a process to read a data from a channel, but unlike the single **Read** operation, they block the channel writer until the read value is processed, thus

allowing the designer to control the availability of the channel and possibly avoiding the memorization of the read value; indeed, the **BeginRead** operation acquires the data without releasing (acknowledging) the channel, the latter being released only by the **EndRead** operation;

- **Write** channel operation: this operation allows a process to write a data on a channel following a handshake protocol; the writing process is suspended until the handshake protocol completes;
- **Ready** channel status: this property is defined as a bit signal that is constantly updated so as to reflect the status of a channel; it is available on the passive port of a channel to sense if communication has been initiated over the channel by the corresponding active port: it returns bit value 1 if the active component has initiated communication over the channel, and bit value 0 otherwise; it is typically used in the expression of a SystemVerilog **wait** statement to suspend the enclosing process until one of the corresponding channels becomes ready.

Note that the implementation details of Tiempo asynchronous handshake protocols are completely hidden to the designer who can therefore focus on the high-level modeling of the design (see examples of section 3).

2.1.3 Asynchronous processes

An asynchronous process specifies a dataflow network relating a set of channels read by the process to another set of channels written by the process. The synchronization is exclusively done through channel handshake operations. Signal event statements (e.g., **posedge**, **negedge**) are not used. An asynchronous process is implemented by a SystemVerilog always process statement containing neither event control nor event expression. As they imply event control statements, the **always_ff**, **always_latch** and **always_comb** processes cannot be used to model asynchronous processes.

Because of the used point to point handshake communication protocols, asynchronous processes suspend while executing channel read-write operations. Indeed, a read operation on a channel can complete only if a corresponding write operation is executed on the same channel. Similarly, a write operation can complete only if a corresponding read operation is done on the same channel.

Any typical SystemVerilog control flow structure can be used (**if/else**, **for**, **case, unique if, unique case**) among the sequential statements of the asynchronous process (always block). In addition, **fork/join** constructs are used to express the parallelism of multiple channel communications (see examples of section 3).

2.1.4 Interfacing with the synchronous world

Tiempo provides a set of predefined modules that implement various asynchronous-to-synchronous and synchronous-to-asynchronous interface communications, allowing the designer to model mixed asynchronous-synchronous designs that can be simulated with any standard HDL simulator supporting mixed Verilog, VHDL and SystemVerilog descriptions.

2.2 Modeling for verification

Any type of SystemVerilog testbenches can be used to verify asynchronous designs or mixed synchronous-asynchronous designs. The abstraction level of SystemVerilog testbenches is perfectly suited to verify asynchronous designs modeled in Tiempo SystemVerilog asynchronous coding style.

Tiempo recommends standard SystemVerilog-based verification methodologies such as VMM (Verification Methodology Manual) or OVM (Open Verification Methodology).

3 SystemVerilog asynchronous modeling: an example

3.1 The structure of the design and its testbench

This section is an illustration of the usage of SystemVerilog to model asynchronous designs using the concepts introduced in section 2. For that purpose, a very simple design, along with its testbench, is described in this section.

This design is mainly composed of two simple asynchronous modules:

- The first module is the control unit and is based on a finite state machine (FSM). It performs a sequence of operations to control the other part of the design, also referred to as the operative unit.
- The second module is the operative unit and models a very basic arithmetic and logic unit (ALU). It can perform two types of calculation, either adding or subtracting its operands.

A third module, named Top, instantiates both FSM and ALU modules and propagates through channels the data between the FSM, the ALU and the outside environment. Finally, a test module, named Testbench, instantiates Top, stimulates its inputs with random data, collects and checks its outputs.

Figure 1 describes the architecture of this design. It shows the two asynchronous modules, the FSM and the ALU, and how they are connected within the testbench. Connections are highlighted with the red double arrows and are modeled with Tiempo channels introduced in section 2.1. These channels are used to implement the asynchronous communication protocols between processes/modules.

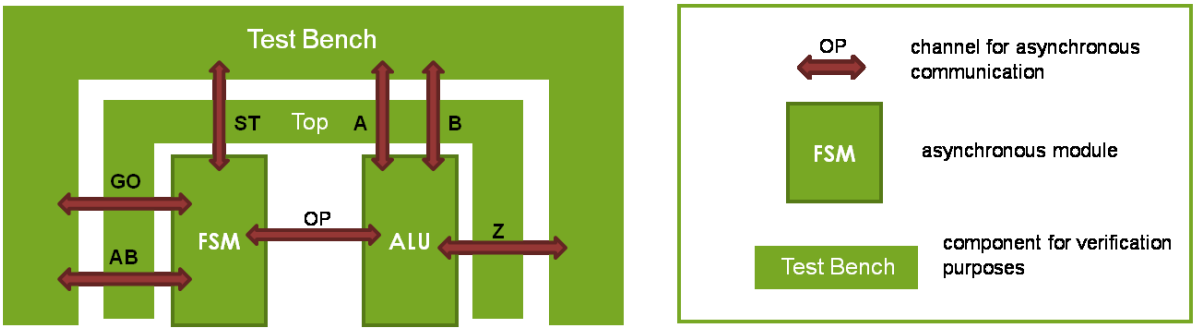

Figure 1: Architecture of the design and its testbench

The channels used to communicate with the FSM are named GO, AB, ST and OP. GO and AB are the channels used to control the FSM and are described further in the next section. ST is used to confirm that the whole FSM sequence is completed, and OP carries the type of operation to be performed (either add or subtract).

The channels used to communicate with the ALU are A, B, respectively its two operands, Z, which carries the result of the ALU operation, and finally OP, described previously.

3.2 The control part: a finite state machine

The state machine diagram is shown in *Figure 2*. It has four distinct states, S0 through S3, and can trigger the ALU with the desired operation.

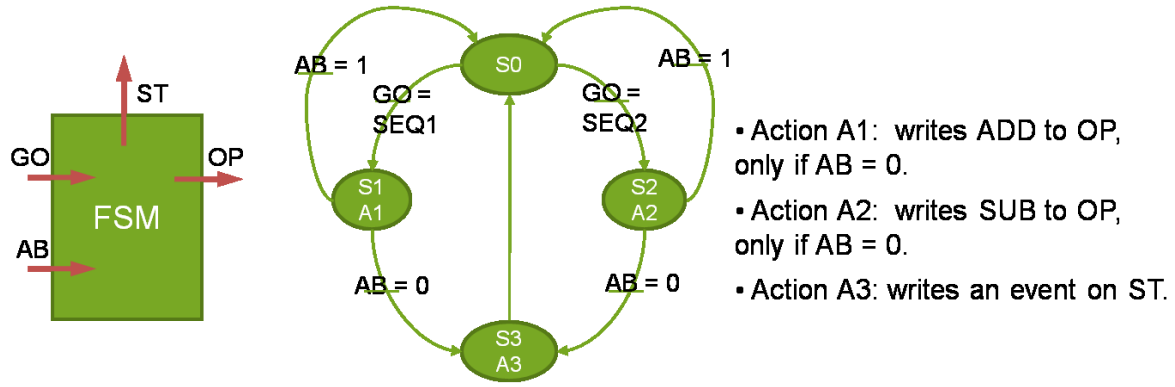

Figure 2: Channels and state diagram of the Finite State Machine

In its initial state S0, the FSM wakes up reading the GO channel. The latter can deliver two values, respectively SEQ1 and SEQ2, according to the specific sequence of actions to execute.

In the case of SEQ1, the FSM moves to state S1 and triggers the ALU to perform a signed addition (action A1). In the case of SEQ2, the FSM moves to state S2 and triggers the ALU to subtract the operands (action A2).

In both states S1 and S2, the FSM can be stopped in its sequence and forced to return to its initial state S0. This is controlled by the AB channel which aborts the FSM sequence if it carries a value equal to '1' or let it execute normally if it carries a value equal to '0'.

When not aborted (i.e., AB =0), the FSM sends the expected operation to the ALU through the OP channel and moves to the final state S3.

In state S3, the FSM generates an event on ST channel (action A3) and goes back to the initial state S0, waiting for a new start.

User-defined data types and channels

The SystemVerilog description of such an asynchronous block will typically start with several definitions of user-defined types of data and channels. These definitions are given in *Code Sample 1.*

```
`ifndef __MYDEFINES__
`define __MYDEFINES__

`include "std_async_defs.sv"

typedef enum {S0, S1, S2, S3} state_t;
typedef enum {SEQ1, SEQ2} go_t;
typedef enum {ADD, SUB} opcode_t;

`DEF_CHANNEL(go_t)
`DEF_CHANNEL(opcode_t)

`endif
```

Code Sample 1: System Verilog source code of user-defined type and channel definitions

This file, named “defines.sv”, is included in each SystemVerilog file where there is a reference to any channel or data using these user-defined types.

In this design, three types of enumerated data are defined with the **typedef** keyword:

- state_t, which defines the four FSM states,
- go_t, which defines the two possible sequences of actions,
- opcode_t, which defines the two types of ALU operations.

While state_t is used only for the definition of the internal state variable in the FSM source code, go_t and opcode_t are the types of the data handled by the GO and OP channels. Therefore, the designer has to use Tiempo predefined **`DEF_CHANNEL** macro to define the corresponding channel types (used to declare channels that transfer data of these user-defined types). The channel types generated by this macro are used in the module channel port definitions of the FSM (see *Code Sample 2* below).

Note the inclusion of the “std_async_defs.sv” file which includes Tiempo predefined asynchronous types and interfaces, enabling for example the definition of user-defined channels with the **`DEF_CHANNEL** macro or the instantiation of predefined asynchronous-to/from-synchronous interfaces.

The SystemVerilog source code of the FSM is given below in *Code Sample 2.*

```
`include "defines.sv"

module FSM (

     push_channel_opcode_t.out OP,
     push_channel_event_type.out ST,
     push_channel_go_t.in GO,
     push_channel_bit.in  AB

);


always begin : fsm_process

   go_t go;
   bit  ab;
   opcode_t op;
   static state_t state = S0;

   unique case (state)

      S0: begin
        GO.Read(go);
        unique case (go)
          SEQ1: state = S1;
          SEQ2: state = S2;
        endcase
      end

      S1, S2: begin
        AB.Read(ab);
        unique if (ab == 1'b1) state = S0;
        else begin
          unique if (state == S1) op = ADD;
          else                    op = SUB;
          OP.Write(op);
          state = S3;
        end
      end

      S3: begin
        ST.Write(SREVENT);
        state = S0;
      end

   endcase

end

endmodule
```

Code Sample 2: SystemVerilog source code of the FSM

Input and output channel ports of the asynchronous module

The FSM module first declares a list of channel input and output ports used to communicate with other units. Each one of these channel port definitions specifies

the channel kind (“push” or “pull”), the data type it carries (user-defined, SystemVerilog integer data type or predefined Tiempo type) and finally the channel port mode (“in” or “out”) to enable the module to perform either a Read or a Write operation on these channel ports.

OP and GO channels carry data of user-defined types as defined in the “define.sv” file. The AB channel carries SystemVerilog **bit** data. Finally, the ST channel is defined with Tiempo predefined type **event_type** that is used to model single event channels that are channels propagating events but not carrying any data.

All the channels are of kind “push” which means that the initiator of the communication is the process that performs a Write operation (kind “pull” means that the initiator of the communication is the process that performs a Read operation).

The mode of the GO and AB channel ports is “in“ as these channels are accessed by the FSM module for Read operations, while the mode of the OP and ST channel ports is “out” so that the module can perform Write operations on them.

The FSM process

The FSM process is modeled with an **always** block just as in any synchronous models, except that there is *no clock* and *no sensitivity list*. It instead infinitely loops with a **unique case** statement, using the FSM state and waiting for the completion of the read and write operations on its channel ports.

These read and write operations on channel ports are respectively modeled with the **Read** and **Write** operations defined with each channel type. These operations include the full handshake protocol of asynchronous communications, meaning that the process is suspended until the corresponding operation is initiated, and the channel on which such operation is performed is released as soon as the operation is completed. For the read operation, this generally results in the local memorization of the value that has been read.

For example, in state S0, the FSM process executes the GO.Read(go) operation which triggers a read access on the GO channel and stores the read value in the local variable named go. The process will wait for the completion of this reading before executing the statements which follow this operation.

In states S1 and S2, once the AB channel is read, standard equality operators (**==**) and standard controlling statements (**if/else**) are used to either abort the sequence and go back to state S0, or to continue the sequence and trigger the ALU by writing the OP channel.

3.3 The operative part: an arithmetic and logic unit

The ALU diagram in *Figure 3* is shown below and introduces the port modes of its communication channels.

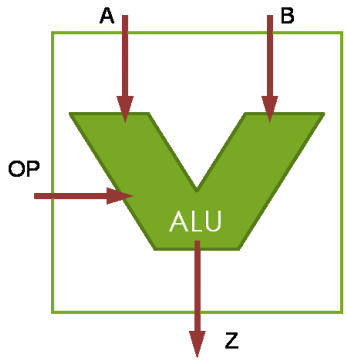

Figure 3: ALU diagram

The ALU module accesses with read operations the channel ports A, B and OP of mode “in”, where A and B respectively carry the operand values and OP carries the type of the operation to be performed, and it uses the channel port Z of mode “out” to write the result of the computation.

The *Code Sample 3* below shows the SystemVerilog source code of the ALU.

```
`include "defines.sv"

module ALU (
     push_channel_byte.out Z,
     push_channel_byte.in  A,
     push_channel_byte.in  B,
     push_channel_opcode_t.in OP
);

always begin : compute
   opcode_t op;
   byte a, b, z;

   fork
     OP.BeginRead(op);
     A.BeginRead(a);
     B.BeginRead(b);
   join

   unique case (op)
      ADD: z = a + b;
      SUB: z = a - b;
   endcase
   Z.Write(z);

   fork
     OP.EndRead();
     A.EndRead();
     B.EndRead();
   join

end

endmodule
```

Code Sample 3: SystemVerilog source code of the ALU

Here the standard SystemVerilog **byte** data type has been used to model the data carried by the operands and the result of the ALU, and the corresponding channel types, which are predefined in Tiempo SystemVerilog definition files, are used to define the corresponding channel ports (Tiempo provides SystemVerilog files including all channel type definitions for all SystemVerilog integer data types).

Modeling with fork and join statements

The designer uses in this description **fork** and **join** statements to model the fact that a set of channel communications is expected without any specific order. Otherwise, the sequential order of these events would have to be guaranteed at the cost of additional hardware resources (i.e., asynchronous sequencers).

The ALU example illustrates this parallelization with read operations on OP, A and B channels. This is possible only because there is no dependency between the processing of the related data: a, b and op local variables are all independent. On the contrary, z can only be computed when all previous variables are set: it is thus impossible to add the Z.Write(z) operation into the fork/join list of threads.

Avoiding useless memorization

Unlike in the FSM example where channel read operations are performed with a single **Read** operation, channel read operations in the ALU example are modeled using dual **BeginRead** and **EndRead** operations (also defined for each channel type).

These dual operations are used to avoid the memorization of the value that has been read. In this case, the channel on which the **BeginRead** has been executed is not released, therefore holding the value, until a corresponding **EndRead** is executed. As a consequence, the process which wrote the value being hold is also suspended until the corresponding **EndRead** operation is executed.

In this example, the OP, A and B channels carry values that are held during all other operations executed by the ALU. These channels are released only after the result has been computed and successfully written to the Z channel.

3.4 The top module and the testbench

Code Sample 4 below shows the SystemVerilog description of the Top module which instantiates the FSM and the ALU modules.

```
`include "defines.sv"

module Top (

     push_channel_event_type.out ST,
     push_channel_go_t.in GO,
     push_channel_bit.in  AB,
     push_channel_byte.out Z,
     push_channel_byte.in  A,
     push_channel_byte.in  B

  );

  push_channel_opcode_t OP ();

  FSM U0 (OP, ST, GO, AB);
  ALU U1 (Z, A, B, OP);
endmodule
```

Code Sample 4: SystemVerilog source code of the Top module

There is quite no difference in the way an asynchronous module is instantiated within another module compared to synchronous designs. The only difference is the usage of channel ports and internal channels instead of modules ports and internal signals. In this example, an internal channel, named OP, is instantiated to connect the output channel port OP of the FSM module to the input channel port OP of the ALU module. The remaining channel ports (all the others ports except OP) are connected through the hierarchy of the Top module.

Code Sample 5 below describes the Testbench module. For the sake of simplicity, only the relevant parts of the testbench source code are described here.

```
`include "defines.sv"

module Testbench;

  push_channel_go_t GO ();
  push_channel_bit  AB ();
  push_channel_event_type ST ();
  push_channel_byte A ();
  push_channel_byte B ();
  push_channel_byte Z ();

  go_t go;
  bit ab;
  byte a, b, z;

  Top U0 (ST, GO, AB, Z, A, B);

  always begin : p_bench
    for (integer i=0; i<LOOP_NB; i=i+1) begin
      // Code for generation of random stimuli on a,b, ab and go
      //
      fork
        GO.Write(go);
        AB.Write(ab);
      join

      if (ab == 1'b0) begin
        fork
          A.Write(a);
          B.Write(b);
          Z.Read(z);
          begin
            wait(ST.Ready);
            ST.EndRead;
          end
        join
       end
       // Code for data checking
       //
    end
  end
endmodule
```

Code Sample 5: SystemVerilog source code of the Testbench module

The testbench is a verification module and has no input or output channel ports.

First, internal channels are instantiated to allow communication with the Top module, then the Top module is instantiated followed by a unique **always** block with a **for loop** to generate random stimuli, apply these stimuli to the Top module input channel ports, retrieve its results and verify them against a reference.

Code Sample 5 above mainly describes how the stimuli are applied and results retrieved. One can notice that the same simple Read and Write operations are used to apply those stimuli and retrieve the results to/from the Top channel ports. It is also interesting to note that, whenever possible, these operations can be parallelized as well within fork and join statements.

4 Conclusion

Tiempo SystemVerilog asynchronous coding style offers to the designer a very simple and efficient way to write high-level models of asynchronous designs that are automatically synthesized with ACC, Tiempo Asynchronous Circuit Compiler, into a Verilog gate-level netlist.

SystemVerilog asynchronous models can be written by designers who are not expert in asynchronous design, most of the implementations details of Tiempo unique asynchronous design technology being hidden to the designer: acknowledge signals requested by the handshake communication protocols, specific signal encoding (dual-rail, multi-rail) and glitch-free logic that are mandatory to ensure the delay insensitivity of the generated designs. The designer can therefore focus on a high-level modeling style that, independently from the fact that it generates asynchronous designs, is very efficient to explore and design different architectures of complex System-On-Chips.

For further information

Tiempo
110 rue Blaise Pascal
Bâtiment Viseo – Inovallée
38330 Montbonnot Saint Martin
FRANCE
T : +33 4 76 61 10 00
F : +33 4 76 44 19 69