Demonstration Project:

EXAMPLE-single\_cycle

Single Cycle Module Demonstrator Overview

Moore, Douglas John

Peimann, Scott Meyer

Shiell, Jon

Dynamic Machine Automation Technology, Inc.

This Page is Intentionally Blank

# Revision History

|  |  |  |
| --- | --- | --- |
| DATE | NAME | DESCRIPTION |
| 2024-10-25 | First uvm-EXAMPLE-single\_cycle and UVM Testing, Initial Release | The first release of a single\_cycle module including UVM testing. |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Table of Contents

Revision History i

Table of Contents ii

List of Figures iv

List of Tables iv

Abstract 1

General Goals 1

Purpose of the single\_cycle Module 2

Language Used in this Document 2

Control Signalling Protocol 2

Architecture and Design of the single\_cycle Module 4

Signals 4

Functional Methods 4

Pertinent Input and Output Signals 6

Behavior Requirements 8

Processor Pipeline Control 8

Data Signals and Processing 8

Debug Signals 9

Testing of the single\_cycle Module 10

Behavior Prediction Algorithm 10

Test of All Input Control Signal Transitions 10

Sequence of Weighted, Random Control and Data Input 11

APPENDIX A - Abbreviations, Acronyms, and Terms 12

APPENDIX B - Baseline Common Control Signal Verification 13

Basic Test 13

Evil Test 14

APPENDIX C - Saved Document Bits 15

Work in Progress 15

# List of Figures

[Figure 1: Figure Caption for Copy/Paste 15](#_Toc180741546)

# List of Tables

[Table 1 - Module Clocking 6](#_Toc180741547)

[Table 2 - Module Input Signals, Control and Data 6](#_Toc180741548)

[Table 3 - Module Output Signals, Control and Data 7](#_Toc180741549)

[Table 4 - Pipeline Control Input Signals 8](#_Toc180741550)

[Table 5 - Pipeline Control Output Signals 8](#_Toc180741551)

[Table 6 - Example Control Signal Transition 11](#_Toc180741552)

[Table 7 - Suggested Weighting 11](#_Toc180741553)

[Table 8 - Abbreviations, Acronyms, and Terms 12](#_Toc180741554)

[Table 9 - Table Caption for Copy/Paste 15](#_Toc180741555)

# Abstract

The single-cycle module demonstration is a basic demonstration of UVM applied to a prototype module implemented using SystemC.

The module was designed to demonstrate control signalling and a basic data transformation. Signalling was designed to apply to a straightforward, pipelined processor, which executes one instruction per clock cycle.

The tests comprehensively exercise the module based on all combinations of control signals, as well as the injection of an 'error' signal through the debug channel.

Keywords: SystemC, UVM, C++

# General Goals

* Implement the module using the reference implementation (without modification).
* Use of the C++ standard version at the highest level supported by the SystemC and UVM/SystemC reference implementations.
  + This was C++20 since C++23 was not compatible with SystemC and UVM/SystemC reference implementations the project started with.
* Use modern C++ principles and standards.
* Write tests using UVM methodology.

# Purpose of the single\_cycle Module

This module is intended to develop and verify basic design methodology for a single stage of a multi‑stage processor pipeline. The work will produce a prototype to be used as a basis for implementing actual processor pipeline stages. The implementation is not intended for use in an actual processor.

Following is a list of basic module goals:

1. Use of control signal conventions common to all pipeline stages.
2. Inter‑module control signalling correctness.
3. Out‑of‑band control signal response.
4. Processes valid input data in one clock cycle.
5. Compatibility with UVM testing methods.

As a consequence of the goals, the prototype module ought to be simply "daisy‑chained" in a multi‑stage pipeline demonstration.

## Language Used in this Document

Words have meaning, and are sometimes used synonomously. In particular the following:

* A stage denotes a module.
* Asserted is true and De-asserted is false.
* Set is true and clear is false.

## Control Signalling Protocol

As this module is prototype stage for a processor pipeline, the authors have implemented a basic control-flow. There are four inputs to and three outputs from each stage. Signals flowing from one stage to another use the same name, the prefix "out" or "in" denoting origin and recipient.

The Following signal conventions are used:

1. The "reset" signal places all pipeline stages into a well-known state.
2. The "is\_flush" signal places all pipeline stages into an "empty" state but does not clear any outstanding errors.
3. Stages that assert "is\_advance" will implicitly take their input.
4. Stages that assert "is\_valid" will latch their output data until it is taken.

In somewhat greater detail:

1. reset" causes all stages to clear "is\_error" and "is\_valid" (current data are discarded). Meaning that data can start flowing from the topmost stage if multiple stages.
2. "is\_flush" all stages clear "is\_valid" (current data is lost and "is\_error" is maintained). Meaning that data can start flowing from the topmost stage if multiple stages.
3. "is\_advance" is computed combinatorically, as any stage of the pipeline can stall any or all prior stages of the pipeline. (in Verilog terms, it is treated as a wire)

Notably:

* "is\_advance" flows up from "this" stage to the "prior" stage.
* "is\_valid" flows down from "this" stage to the "next" stage.

# Architecture and Design of the single\_cycle Module

Module design is straightforward, in that it demonstrates correct control signalling and trivial data processing.

The module inherits from the SystemC "sc\_module" class, although there is an intermediate namespace "SC" which encapsulates the SystemC class as "SC::module". Explicit code provides for SystemC module requirements, as opposed to use of the SystemC macros for module construction (e.g. the encapsulated type name SC\_CURRENT\_USER\_MODULE).

In general, macros are avoided, as they are considered poor C++ practice. That noted, the "SC\_METHOD" macro is used for simplicity and consistency with general SystemC conventions.

## Signals

Signals and ports are bundled into groups, and they are private to the class. Public accessor methods are provided for the various groups allowing binding of signals to ports. By intent, references are used in preferance to bare pointers.

The signal groups are:

* m\_clock - system clock.
* m\_control - control signals in and out.
* m\_data - data signals.
* m\_debug - diagnostic signals.

The input signal binding methods require explicit references, as these signals may come from disparate elements of the system.

The signal export methods return references to relevant sub-structures of the various signal groups.

Note that the signal group structures are privately held to the module, however the exported sub‑structures provide their elements publically.

## Functional Methods

Two private methods, "compute\_advance" and "cycle", give the module's run‑time functionality.

The method "compute\_advance" is responsible for computing the output control signal "is\_advance". It is triggered by change to any one of the three control signals used by the computation.

The method "cycle" is responsible for all other control and data signals. It is triggered on the leading edge of the system clock. The basic structure of the method is that it:

1. Captures all input signals.
2. Captures the current output state of the module.
3. Computes revised signal values.
4. Assigns all output signals (excepting "is\_advance").

This structure is intended to encourage robust behaviour. The structural goals provide for:

1. Straightforward code verification.
2. Minimal notational complexity.
3. Logical separation of responsibility within the method.
4. A single read and write of each port or signal, avoiding inadvertent misuse of SystemC.
5. Consistent update of the output state, as the final step writes all output signals.

Module specific data structures are declared private to the module. Single‑use data structures are also declared as private, nested types, with the intent to simplify method construction and analysis. This methodology provides for consistent naming and structure, while avoiding use of ad‑hoc naming.

# Pertinent Input and Output Signals

This section documents module signals pertinent to correct pipeline behavior. Not all signals into and out of the module are pertinent to its basic behavior, for example, most of the debug signals. Said signals are not documented herein.

A summary of the required clock signals is given in Table 1 - Module Clocking, below. Only one clock signal is required for the correct module function. Actions are performed on the positive edge of the clock cycle.

Table - Module Clocking

| SIGNAL | DESCRIPTION | NOTE |
| --- | --- | --- |
| Clock | System clock. | Module acts on the positive edge of the clock. |

A summary of the required input signals is given in Table 2 - Module Input Signals, Control and Data, below. Observe that the debug.is\_error signal is used to force the module into an error state. Unless otherwise stated, the control signals use positive logic.

Table - Module Input Signals, Control and Data

| SIGNAL | DESCRIPTION | NOTE |
| --- | --- | --- |
| Reset | Forces module into the reset state. | Negative logic.  Overrides all other control signalling. |
| in.is\_advance | Signal indicating that valid output data will be taken. | From the "next" module in the pipeline. |
| in.is\_flush | Signal to flush any available data, and enter state waiting for new input data. | A subsequent module in the pipeline "branched", rendering the current value and inputs moot. |
| in.is\_valid | Signal indicating that the input data are valid. | From the "prior" module in the pipeline. |
| in.data | Signal (bundle) with data to process. | 32-bit value.  Used to verify that processing and signal handling occurred correctly. |
| debug.is\_error | Signal to force the module into an "error" state. |  |

A summary of the required output signals is given in Table 3 - Module Output Signals, Control and Data, below. All the control signals use positive logic.

Table - Module Output Signals, Control and Data

| SIGNAL | DESCRIPTION | NOTE |
| --- | --- | --- |
| out.is\_advance | Signal indicating that the module will accept valid input data. | To "prior" module in the pipeline. |
| out.is\_error | Signal indicating that the module has detected an error. | Vestigial signal.  Forced by input debug.is\_error.  NB: all combinations of input signals are valid.  NB: no data processing errors exist in the requirements. |
| out.is\_stall | The signal indicates that the module is imposing a stall on processing. | Always false.  To "prior" module in the pipeline.  NB: this module is a single-cycle module, and thus cannot introduce a pipeline stall. |
| out.is\_valid | Signal indicating that processed data are valid. | To the "next" module in the pipeline.  NB: out.data has no meaning. |
| out.data | Signal (bundle) with processed input data. | 32-bit value.  To the "next" module in the pipeline.  Value is inverted input data.  Only valid when out.is\_valid is set to true. |

# Behavior Requirements

The module processes its input data input in one cycle unless it is stalled by subsequent processor modules, reset, or forced into error. The output data are latched until the subsequent module accepts the output, or the module is reset.

The basic intent of a module is to accept a datum, process it, and latch the output value for the next pipeline stage.

## Processor Pipeline Control

As part of a processor pipeline, the module implements generalised the control signalling protocol. This protocol is implemented using four one‑bit input signals and four one‑bit output signals. Pipeline control signals are described in Table 4 and Table 5, below. Note that there is an overloading of names, this is to facilitate module signal binding (like binds to like).

Table - Pipeline Control Input Signals

|  |  |
| --- | --- |
| SIGNAL | DESCRIPTION |
| reset | Places all pipeline stages into a well-known state. |
| is\_flush | Places all pipeline stages into an "empty" state but does not clear any outstanding errors. |
| is\_advance | Stages asserting "is\_advance" will implicitly take their input data. |
| is\_valid | Stages asserting "is\_valid" will latch their output data until taken. |

Table - Pipeline Control Output Signals

|  |  |
| --- | --- |
| SIGNAL | DESCRIPTION |
| is\_advance | Stages assert "is\_advance" indicating they will implicitly take their input data. |
| is\_error | Stages assert "is\_error" when they detect an internal or a communication error. |
| is\_stall | Stages assert "is\_stall" indicating they are performing a multi-cycle transaction. |
| is\_valid | Stages assert "is\_valid" indicating they have latched valid output data. |

## Data Signals and Processing

The module implements 32‑bit input and output value signals.

Value processing is trivial, by intent. When an input value is taken, the module simply inverts the value and writes it to the output signal.

Observe that the name "value" is used for input and output. As with the pipeline controls, the name is overloaded to facilitate binding of signals.

## Debug Signals

Of the debug signals, only one is directly pertinent to the module's behaviour. The pertinent signal is the debug input "is\_error".

The "is\_error" signal is asserted to force the module into an error state, as there are no input signal or data processing errors which can otherwise cause the module to enter an error state.

Strictly, when the input "is\_error" is asserted, the module latches "true" into the output "is\_error" signal; no other behavioral changes occur.

# Testing of the single\_cycle Module

Correctness is emphasized over performance in the initial release cycle, so that a complete, comprehensive set of unit tests can be developed. In future development cycles, performance features will be implemented.

Although we exclude performance optimizations, effort has been made to allow them in future processor modules. For example, in the initial single\_cycle Module Design is simple to prove that it and UVM Testing can be completed with SystemC.

Testing, minimally, shall include:

1. An independently developed behavior prediction algorithm.
2. All input control signal transitions.
3. A long sequence of weighted, random control and data input.

## Behavior Prediction Algorithm

Independentl of module implementation, a single behavior prediction algorithm should be constructed, which will be used to verify output from the module when testing. The prediction algorithm should be general, not specific to any one test. Tests may elect to ignore elements of the output, based on explicit requirements.

The algorithm need not predict debug signals.

Needless to say, should a discrepancy be found, assume that both the module and the algorithm are in error, until proven otherwise.

## Test of All Input Control Signal Transitions

The goal of this test is to prove that the module pipeline controls behave correctly for all possible input control signal transitions.

This test shall inject all possible single-cycle contol signal transitions into the module. Output control signal values shall be verified for each cycle of the test. This test shall verify that the input domain is completely covered (i.e. all possible transitions are injected).

Excluding the forced error signal, data and debug signals are moot for this test. The output data signals may be verified, however their verification is not required.

The input domain consists of 1024 possible control signal transitions. Table 6 illustrates an arbitrary pair of control signal inputs to inject into the module for this test. The example covers one of the possible transitions.

Table - Example Control Signal Transition

| SIGNAL | INPUT VALUE | | NOTE |
| --- | --- | --- | --- |
| **cycle n** | **cycle n+1** |
| reset | 1 | 1 | Reset uses negative logic. |
| is\_advance | 1 | 0 | From "next" pipeline stage. |
| is\_error | 1 | 0 | Forced error. |
| is\_flush | 0 | 0 | Out‑of‑band signal. |
| is\_valid | 1 | 1 | From "prior" pipeline stage. |

## Sequence of Weighted, Random Control and Data Input

The goal of this test is to prove that the module behaves correctly over a large number of cycles.

As the module is a prototype for use in a basic processor pipeline, inputs shall be weighted to emulate somewhat sane processor behavior.

Approximate field weights are given in Table 7, below. Observe that "reset" is not held high continuously, it is a part of the test. Values for the debug inputs are arbitrary, with the exception of the forced error ("is\_error").

This test shall run for a minimum of 10,000 cycles.

**NOTE**: the "reset" signal uses negative logic; it will be asserted (set to zero) approximately 1 in 250 cycles.

Table - Suggested Weighting

| SIGNAL | BITS | APPROXIMATE WEIGHT | NOTE(s) |
| --- | --- | --- | --- |
| reset | 1 | 1/250 | Negative logic.  Signal is held high for normal operation. |
| Is\_advance | 1 | 3/4 | From "next" stage. |
| Is\_error | 1 | 1/1000 | Forced error.  Very low frequency.  Error output bit latches. |
| Is\_flush | 1 | 1/50 | Out-of-band. |
| Is\_valid | 1 | 3/4 | From"prior" stage. |
| data | 32 | uniform distribution | Use random values. |

# APPENDIX A - Abbreviations, Acronyms, and Terms

Table 8 - Abbreviations, Acronyms, and Terms

|  |  |  |
| --- | --- | --- |
| TERM | DEFINITION | DESCRIPTION |
| Bit |  | Really, do we need to define this for you, just really? |
| Byte | Eight (8) bits |  |
| POR | Power on Reset |  |
| SystemC | An ANSI standard C++ event driven simulator. | The formal version 2.3.4 document at IEEE:  [Link to IEEE Document](https://ieeexplore.ieee.org/document/6134619)  Accellera, which provides a reference implementation of SystemC:  [Link to Accellera](https://accellera.org/)  Note: standard was updated to version 3.0.0 in 2023 September. |
| UVM | Universal Verification Methodology |  |
| Word | Four (4) bytes  Thirty two (32) bits | Synonym for Full-Word. |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# APPENDIX B - Baseline Common Control Signal Verification

**The tests described in this appendix document the original, purely SystemC tests, which were used to verify the single\_cycle module.**

Analogues of the tests herein are discussed in the testing sections above.

A module's control signal verification is used to show that one-cycle data movement handles its signals correctly.

The basic test performs verification of the output control signals based on a structured injection of the input control signals. The test injects every possible pair of control signals in sequence and verifies the output signals.

The evil test performs verification of output control signals based on a large number of weighted random injections of input control signals. The test injections pseudo random seeded weighted pairs of control signals and verify the output signals.

The two different verification methods are used to increase the confidence that defeats can be detected early during the development cycle of modules.

## Basic Test

Method name and signature: void do\_test\_basic(test\_t const &test)

Done to verify the control logic this performs a control signal test wherein every possible combination of two control signal sets is injected into the module under test and verifies the control signal output.

The five (5) Input Control Signals:

1. [reset] (System Reset)
2. [is\_advance] (From the following module)
3. [is\_error] (Forces an error)
4. [is\_flush] (pipeline flush; from a later module)
5. [is\_valid] (Valid information available from prior module)

The Four (4) Output Control Signals:

1. [is\_advance] (Sent to the prior module)
2. [is\_error] (An error was detected; internal or external)
3. [is\_stall] (The module is stalling)
4. [is\_valid] (The value and CC are valid)

## Evil Test

Method name and signature: void do\_test\_evil(std::uint64\_t const seed)

This test performs sending an input sequence (Evil) to the module's control signal test based on random signal values for 100,000 cycles.

The random-number generator is seeded, based on the method's input argument, allowing test repeatability, should errors be detected.

The random input sequence is generated using control bit weighting to approximate typical pipeline execution.

# APPENDIX C - Saved Document Bits

## Work in Progress

This document is a work in progress. This section contains bits, thoughts, explanatory and detail which might not be evident in the main body. Some bits in here may be patently wrong.

**Read the following bits at your own risk.**

Figure : Figure Caption for Copy/Paste

Table - Table Caption for Copy/Paste

Assumptions:

1. "reset" means reset
   1. is\_error is cleared
   2. is\_stall is cleared (module can never impose stall, anyway)
   3. is\_valid is cleared
2. "is\_flush" means branch, so values are rendered moot
   1. is\_error is updated
   2. is\_stall is cleared (module can never impose stall, anyway)
   3. is\_valid is cleared
3. "out.is\_advance" means that "this" stage will take whatever input was given from the "prior" stage, whether data is consumed and processed is dependent on "in.is\_valid". If true it does and if false it does not. (It was computed as a result of the previous cycle's actions and is an input to the "prior" stage)
4. "in.is\_advance" means that the "next" stage will take whatever input was given from the "this [prior]" stage, whether data is consumed and processed is dependent on "out.is\_valid". If true it does and if false it does not. (It was computed as a result of the previous cycle's actions and is an input to the "this" stage)
5. "in.is\_valid" means that the "prior" stage's data outputs are valid ("this" stage's data input are valid), only a "reset" or "in.is\_flush" clears "in.is\_valid".
6. "out.is\_valid" means that the "this" stage's data outputs are valid ("this" stage's data output are valid), only a "reset" or "in.is\_flush" clears "out.is\_valid".
7. "in.is\_advance" means that the "next" pipeline stage will take what I have
   1. input from the "next" stage
   2. It was computed as a result of the previous cycle's actions.
   3. When "out.is\_valid" is true, the data is consumed and processed by the "next" stage, so "this" stage can advance.
   4. When "out.is\_valid" is false, the data is not consumed or processed by the "next" stage, so "this" stage cannot advance.
   5. I will take your output data, so you can advance.
   6. Indicates that any valid (computed) value is "taken" (by "next"), and thus must be replaced by a value computed based on the current inputs ("is\_valid" and "data").
   8. Regardless of what you give me, I will take it.
8. "out.is\_advance" means that "this" stage will take whatever input is given from the "prior" stage, whether data is consumed and processed is dependent on "in.is\_valid" the same as "in.is\_advance".
9. "in.is\_valid" means that I hold a valid datum; can only be replaced when taken or by reset/flush.
10. The processor pipeline is success-oriented. As a consequence, the following signal conventions are used.
    1. Modules that calculate their internal "is\_advance" will implicitly take their input data.
    2. There is no confirmation of acceptance.
11. The following signal conventions are used:
    1. The "reset" signal places all pipeline modules into a well-known state.
    2. The "flush" signal places all pipeline modules into an "empty" state but does not clear any outstanding errors.
    3. Modules that assert "is\_advance" will implicitly take their input.
    4. Modules asserting "is\_valid" will latch their input data until it is taken.
12. xx
    1. next.is\_advance and prior.is\_valid
       1. take input and update
    2. next.is\_advance and not prior.is\_valid
       1. out.is\_valid = false; out.data = don't-care
    3. not next.is\_advance and state.is\_valid
       1. no change to output
    4. not next.is\_advance and not state.is\_valid
       1. take input and update
13. xx
    1. not state.is\_valid
       1. take input and update
    2. next.is\_advance
       1. take input and update
    3. not next.is\_advance
       1. do nothing
14. The processor pipeline is success-oriented. As a consequence, the following signal conventions are used.
    1. Modules that assert "is\_advance" will implicitly take their input data (whether the data is valid or not.
    2. There is no confirmation of acceptance between modules.
    3. The is\_valid signal will control the use of the data.
       1. true: Process data for the next module
       2. false: Drop it on the floor and wait for valid data.
15. reset causes
    1. is\_error is cleared
    2. is\_stall is cleared (module can never impose stall, anyway)
    3. is\_valid is cleared
16. flush (branch causing pipeline flush) causes
    1. is\_error is updated
    2. is\_stall is cleared
    3. is\_valid is cleared

"out.is\_advance" is computed combinatorically......it is captured at the start of the cycle for general calculation. (in Verilog terms, it is treated as a wire)

Clocked outputs are computed as follows

reset causes

1. is\_error is cleared
2. is\_stall is cleared (module can never impose stall, anyway)
3. is\_valid is cleared

flush (branch causing pipeline flush) causes

1. is\_error is updated
2. is\_stall is cleared
3. is\_valid is cleared

state.is\_advance (this module accepts and processes a new value, if available)

1. is\_error is updated
2. "out.is\_valid" and "out.value" are recomputed based on input otherwise (this module cannot advance the pipeline)
3. is\_error is updated
4. hold state

**Input Signal Conventions**

"in.reset" sets all other control signalling to false.

"in.is\_flush" leaves "in.is\_error" sets all other control signalling to false

"in.is\_valid" (from the above stage) (says please process data from above)

"in.data" (from "prior" stage)

"in.is\_error" - this is an input signalling a forced error; no error state is otherwise possible.

"in.is\_advance" (from "next" stage) (says I am accepting data and not stalling)

**Output Signal Conventions**

"out.is\_valid" (to "next" stage) (says please my process data) signals that a valid datum is available and latched.

"out.data" (to "next" stage)

"out.is\_advance" (to "prior" stage e) (says I am accepting data and not stalling)

The basic intent of the module is to accept a datum, process it, and latch the output value for the next pipeline stage.

When a valid datum is latched, the output signal "is\_valid" is asserted.

Latched datum values are held until the subsequent module is able to accept them. This is observed when the input "is\_advance" signal is asserted. A latched datum value will be held until the next stage can accept it. Implicit acceptance is indicated by the "in.is\_advance" signal being received from the "next" stage.

When a value datum is latched, no further input will be accepted until the output value is passed to the "next" stage.

Pipeline advance is implicit. There are no interlocks associated with the "is\_advance" input or output signals. When "is\_advance" is true, data values are assumed passed when they are valid.

The module shall, in order of priority:

1. Respect the "reset" signal, clearing the "error" and "is\_valid" signals.
2. Respect the debug "is\_error" signal and assert error when said signal is asserted.
3. The "out.is\_valid signal indicates that a "data" value is latched and available. Management of these signals is as follows:
   1. When "out.is\_valid is false:
      1. If "in.is\_valid" was false, "out.is\_valid" remains false.
      2. If "in.is\_valid" was asserted, the input datum will be processed and "out.is\_valid" will be asserted at end of cycle.
   3. "out.is\_valid is asserted, indicating that an output value is latched:
      1. If "in.is\_valid was de-asserted, "in.is\_advance" was de-asserted  
          out.is\_valid will remain asserted at end of cycle.  
         (holds value)
      2. If "in.is\_valid was de-asserted, "in.is\_advance" was asserted  
         "out.is\_valid" de-asserted at end of cycle  
         (next stage takes data, but no input)
      3. If "in.is\_valid was asserted, "in.is\_advance" was de-asserted  
         "out.is\_valid" asserted at end of cycle  
         (next stage cannot take value, reject input, hold value)
      4. If "in.is\_valid was asserted, "in.is\_advance" was asserted  
         "out.is\_valid" asserted at end of cycle  
         (next stage takes data, new input is processed and latched)
   5. When "is\_valid" is asserted, and the subsequent stage asserts "is\_advance" is asserted) and the prior stage (input "is\_valid" is de-asserted), "is\_valid" will be de-asserted.
   6. When the subsequent stage can advance (input "is\_advance" is asserted) and the prior stage (input "is\_valid" is de-asserted), "is\_valid" will be de-asserted.
4. "in.is\_valid" indicates that a valid datum is latched
5. Assert the "in.is\_advance" signal, module can and will accept data.
6. Assert the "in.is\_valid" signal, module has process data for the subsequent module and "out.is\_advance" input is true.
7. De-Assert the "out.is\_valid" signal, has No process data for subsequent processor.
8. De-Assert the "out.is\_advance" signal, when the subsequent modules or itself are stalled.
9. Accept the data signals, inverting them, when processing to confirm processing.
10. "out.is\_advance" is de-asserted when the module is holding a value, or the module has received an "in.is\_advance" signal from the next module
11. "out.is\_advance" is de-asserted when the module is holding a value.
12. "out.is\_advance" is asserted when the module has received the "in.is\_advance" signal from the next module