DesignCon 2008

Demonstration of SerDes Modeling using the Algorithmic Model Interface (AMI) Standard

Michael Steinberger, Signal Integrity Software, Inc.
msteinb@sisoft.com, 715-720-4112

Todd Westerhoff, Signal Integrity Software, Inc.
twesterh@sisoft.com, 978-461-0449 x24

Christopher White, Independent
yooper4ever@gmail.com

Abstract

To date, it has been difficult to simulate a high speed serial link whose driver and receiver were supplied by different vendors because models supplied by different vendors are almost never compatible. The Algorithmic Modeling Interface is an API developed by the IBIS ATM standards committee to solve this problem.

This paper introduces the AMI standard and demonstrates its use in modeling the behavior of both the data path and the adaptive control loops. The demonstration transmission path includes a transmitter with de-emphasis and FFE, a lossy transmission line, and a receiver with peaking filter, clock recovery and DFE.

Authors' Biographies

Dr. Michael Steinberger is currently responsible for leading the development of SiSoft's serial link analysis products. He has over 29 years experience in the design and analysis of very high speed electronic circuits. Prior to joining SiSoft, Dr. Steinberger worked at Cray Inc., where he designed very high density interconnects and increased the data rate and path lengths to the state of the art. Mike holds a B.S. from the California Institute of Technology and a Ph.D. from the University of Southern California, and has been awarded 7 U.S. patents.

Todd Westerhoff, vice president of software products for SiSoft, has over 26 years experience in the modeling and analysis of electronic systems, including 10 years of signal integrity experience. Prior to joining SiSoft, Todd managed a high-speed design group that provided static timing, signal integrity and design rule consultation to various ASIC and system engineering groups within Cisco Systems, Inc. Todd holds a B.E. degree in electrical engineering from the Stevens Institute of Technology in Hoboken, New Jersey.

Christopher White has a BSEE from Michigan Technological University and a MSEE from the University of Minnesota. He has worked for IBM and JDSU in the areas of analog IC design and signal integrity analysis for optical interconnect products. He was a signal integrity engineer for Cray Inc. where he focused on high speed networks. Currently he is doing mixed-signal IC design on Read Channels for Hitachi Global Storage Technologies.

1. Problem Statement

In order to make any useful prediction of the performance of a high speed serial link, the response of any equalizers in the transmitter and/or receiver must be included in the simulation or analysis. To date, this has left the system developer with one of two options: either use a generic model of the equalizers or get a model from the SerDes IP vendor.

The SerDes IP vendor is driven by several considerations:

1. They don't want to disclose any more details about their design than they have to.
2. There is no commonly accepted software environment in which to exercise the models, so the vendor must supply their own environment.
3. Customers tend to take it as an assumed contractual requirement that the SerDes will perform as well or better than the model predicts.

This third consideration is an interesting shift in that in the past, SPICE models were considered to be the most accurate models, and IBIS models were accepted as a useful approximation which hid design details. It is impractical to run SPICE simulations for the large numbers of bit required to analyze a high speed serial link, however, and so whatever model the vendor supplies has become the de facto standard for SerDes model accuracy.

One of the consequences of this consideration is that making the models as complete and accurate as possible has been the vendors' primary concern, and other considerations such as execution time has been less important. These models typically need to be run for a million bits before the vendor will vouch for the accuracy of the results. At least two vendors have stated publicly that regardless of whether they preset the state of a receiver to a presumed optimum, it takes their models about a half million bits to reach a steady state and start producing a valid output waveform.

These considerations, and especially the lack of a commonly accepted software environment, has also meant that simulating a link in which the transmitter has been supplied by one vendor and the receiver by another has been difficult at best. Since such links are becoming more and more common, this has become a real problem for system developers.

2. The IBIS AMI Standard

To solve this problem, the IBIS Advanced Technology Modeling (ATM) committee has developed a standard for modeling high speed serial links [1]. This standard is called the Algorithmic Modeling Interface (AMI), and is based on the concept that a model should be an algorithmic one which abstracts out the details of the circuit implementation. By abstracting out non-essential details, such a model can produce useful results in a reasonable amount of time.

An algorithmic model can be delivered in executable form as a Dynamically Loadable Library (DLL) or shared object library, and can therefore be loaded into the execution environment at the time the simulation is run. The standard defines a standard Applications Programming Interface (API) between the dynamically loaded model and the execution environment it runs in.

This standard has numerous advantages:

1. Complete models of end to end serial links can be run in a reasonable time. For example, the simulations of end to end links reported in this paper typically run at a rate of a more than a half million bits a minute.
2. The model developer has a great deal of flexibility in how they choose to write their model.
3. Since they are in executable form, the models hide the details of the implementation.
4. Models from multiple vendors can be readily combined in a single simulation.
5. The same models can run as-is in multiple execution environments, including tools from multiple EDA vendors.

The API defines three functions:

AMI_Init()

- Initialize the model, including parsing input control parameters, allocating memory, and initializing data structures.
- (optionally) Compute an optimum configuration based on the channel impulse response.
- (optionally) Output the cumulative impulse response to the output of the block.
- (optionally) Output the parameter values the block will use.
- (optionally) Output an information message for the user.

AMI_GetWave()

- Process a block of time samples as part of a time domain simulation.
- Output the waveform seen at the output of the block. For a transmitter this waveform is the voltage waveform at the output of the driver. For a receiver, this waveform is the equivalent waveform at the decision point of the receiver.
- (optionally) Output a sequence of clock ticks representing the response of the clock recovery circuit.
- (optionally) Output the parameter values the block is currently using.

AMI_Close(): Clean up after execution of the model is complete.

The function signatures are

```
long AMI_Init( double *impulse_matrix,       //Victim and aggressor
                                             //   impulse responses
               long row_size,                //Impulse response length
               long aggressors,              //Number of aggressors
               double sample_interval,       //Time sample spacing
               double bit_time,              //Symbol duration
               char *AMI_dll_parameters_in,  //Input control parameters
               char **AMI_dll_parameters_out,//Output parameters
               void **AMI_dll_memory_handle, //Model heap storage
               char **msg );                 //Informative message

long AMI_GetWave( double *wave_in,               //Wave segment in/out
                  long wave_size,                //Number of samples
                  double *clock_times,           //Array of clock ticks
                  char **AMI_dll_parameters_out, //Output parameters
                  void *AMI_dll_memory );        //Model heap storage

long AMI_Close( void *AMI_dll_memory );          //Model heap storage
```

The AMI_Init() function is mandatory because it is needed to set up the parameters and memory for AMI_GetWave(). (If there were neither an AMI_Init() nor an AMI_GetWave() function, there would be no model at all, and therefore no point in supplying an AMI_Close() function.) The AMI_GetWave() and AMI_Close() functions are optional, but will usually be needed.

These functions must be supplied in a single file, with no restrictions on the name of that file other than those imposed by the operating system. The functions themselves, however, must have the names and signatures defined in the standard. Models are therefore distinguished from each other based on file name.

There is also a file format defined to tell the model execution environment the names, types, and valid values of the parameters which can be supplied to the model. This file is expected to have the file extension `.ami`. Within this file, the information is provided in an ASCII tree syntax that is at least somewhat readable. There is also a utility available to convert the file to an outline format that is more easily read.

The sequence of execution is

1. From the IBIS file, the execution environment reads the name of the shared object library file and the name of the `.ami` file.
2. The execution environment loads and parses the `.ami` file.
3. From the contents of the `.ami` file, the execution environment obtains information directly about the model (such as whether or not an AMI_GetWave() is supplied) and also executes a dialogue with the user to determine parameter values that are to be supplied to the model.
4. From the user dialogue, the execution environment assembles a parameter string conforming to the syntax specified in the IBIS AMI API.
5. The execution environment loads the shared object library file whose name was given in the IBIS file.

6. The execution environment locates the functions in the shared object library whose names were given in the IBIS AMI API.
7. The execution environment calls each function at the appropriate time, with arguments that were formed based on the assumption that the functions were compiled using the signatures that were given in the API.

Two evaluation toolkits are available in the "Work Archive" at http:///www.vhdl.org/pub/ibis/macromodel_wip/ .One of these evaluation toolkits provides a transmitter model while the other provides a receiver model. Each evaluation toolkit also provides its own execution environment in which any IBIS AMI model, including either the transmitter or receiver model, can be run. These models are intended to provide model developers with a useful starting point. In addition, the transmit model source code exercises all of the optional features of the API.

It is especially important to note that the standard defines a software interface (API), but does not mandate a programming language. While the interface is defined in the C language, it is entirely straightforward to write a C language wrapper around a model written in another language such as MatLab®, Octave, Verilog or VHDL AMS, C++, Perl Data Language (PDL), Python pyNum, or Fortran. There is a clear distinction:

- Models written in two different programming languages but conforming to the same software interface can be expected to interface with each other in the same simulation without a problem and produce reasonable results.
- Models written in the same programming language but not conforming to the same software interface cannot be expected to interface successfully with each other in the same simulation, much less produce reasonable results.

The following sections describe two distinctly different types of function arguments defined by the API. As will be seen in the example system, these two types of information make the API very versatile.

2.1 Time Domain Waveforms

As would be expected, both AMI_Init() and AMI_GetWave consume and produce time domain waveforms.

AMI_Init() takes in an impulse response for the desired channel, and can also take in the impulse responses for as many crosstalk aggressors as a given model will support. These impulse responses can be used to optimize the starting state of the model, and the model can optionally output the impulse response which represents the concatenation of the block with the path represented by the input impulse response.

The aggressor impulse responses are included in the interface partially because the level of crosstalk can affect the optimum configuration, especially for a receiver with a peaking filter at its input. If the impulse response output option is supported for the aggressor impulse responses, then these outputs can be used together with the aggressor data rate (supplied by separate means) to calculate the crosstalk amplitude distribution at the output of the block.

AMI_GetWave() takes in a block of uniformly spaced time samples of an input waveform and produces an output waveform segment with the same sample spacing and number of samples. The number of samples is supplied as a separate argument of the function. While the input waveform is determined entirely by the user and the execution environment, it is expected to at least include the desired signal waveform at the input to the block. Some of the variations that are reasonable to exercise in the input waveform are

- Data pattern, including standard test patterns or data patterns used in laboratory measurements
- Duty cycle distortion or transmit jitter
- Crosstalk aggressor signals

It worth noting that just because you can include a phenomenon in the AMI_GetWave() input waveform doesn't mean that you should. Depending on the purpose of the simulation, it may be that a separate analysis of clock phase noise or a semi-analytical bit error rate estimate may be a better choice.

2.2 Parameters

The circuits these algorithmic models represent are quite complex, and there are typically a large number of options that can be configured by the user. Furthermore, the options offered and the way they are configured will vary completely from one SerDes to the next. To support this range of variation, the AMI standard defines a hierarchical syntax for passing parameters into and out of the model. The hierarchical syntax has the advantage of giving the model developer complete flexibility in their choice of parameter names because name spaces are guaranteed not to overlap, and therefore the system configuration is guaranteed to be interpreted unambiguously.

While the original intent of the parameter interface was to configure the models, both AMI_Init() and AMI_GetWave() also have an argument for passing parameters out of the model. As will be demonstrated below, this interface can be used to monitor the state of the model, for example to observe the behavior of adaptive control loops.

3. Modes of Simulation

As illustrated in Figure 1, two modes of simulation or analysis are supported by the AMI standard: linear time invariant analysis and time domain simulation. Each of these has its advantages and disadvantages, and it is advisable to choose that mode that best suits the task to be performed. Linear time invariant analysis is quite fast, and generally results in a comprehensive summary of the full range of possible conditions. It is therefore well suited to exploring a large design space. Conversely, time domain simulation makes it possible to model nonlinear or time varying effects, and is therefore well suited to detailed analysis of very specific conditions.

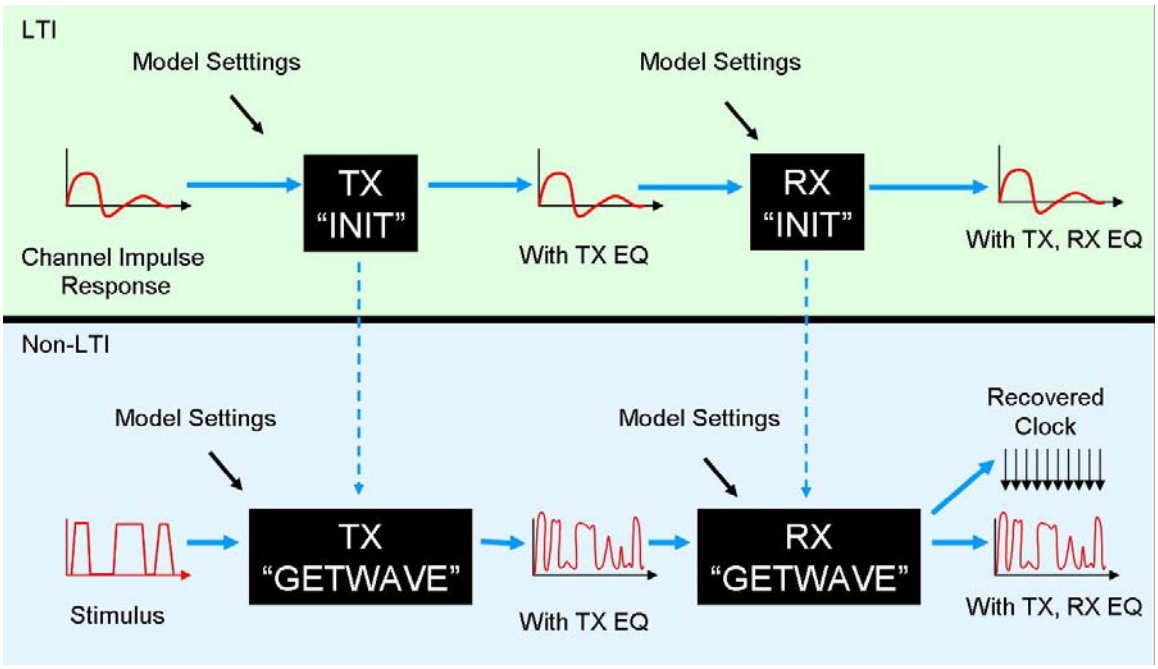


Figure 1: Linear time invariant vs. time domain simulation modes

The upper half of Figure 1 illustrates the linear time invariant flow. The channel impulse response is input into the AMI_Init() function of the transmitter model, resulting in the impulse response of the combined transmitter and channel at the input to the receiver. The receiver AMI_Init() function then transforms this impulse response into the impulse response at the decision point of the receiver.

Because it is the first function called, the AMI_Init() function is also responsible for extracting the model settings from the parameter string. These model settings may be used in either AMI_Init(), AMI_GetWave(), or both.

The lower half of Figure 1 illustrate the time domain simulation flow. A specific data pattern is input as a stimulus into the transmitter AMI_GetWave() function. In the transmitter AMI_GetWave() function, the stimulus is convolved with the transmitter output impulse response previously computed in the AMI_Init() function to produce the input waveform to the receiver AMI_GetWave() function. The receiver AMI_GetWave() function computes the waveform at the receiver decision point and, in the process, may model such complex processes as clock recovery or DFE equalization. The AMI_GetWave() function outputs the waveform at the receiver decision point, and may output a sequence of time values for the rising (or falling) edges of individual cycles of the recovered clock.

Although not shown in Figure 1, it is possible to output parameters from both AMI_Init() and AMI_GetWave(). This can be a valuable source of information, especially in the case of AMI_GetWave().

3.1 Linear Time Invariant Analyses

For linear time invariant analysis, the overall approach is to first solve for the linear response of the channel either as an impulse response, step response, pulse response, or transfer function, and then to estimate the bit error rate from that response. Convolution engine techniques [2] are generally used to generate persistent eyes, “bathtub” curves are estimated from those, and finally the bit error rate is estimated from the “bathtub” curves. It is also possible to perform a Peak Distortion Analysis [3], which is a very similar, though simpler computation for determining the worst case data pattern and corresponding eye closure.

The primary advantage of this approach is that the convolution engine technique can very efficiently calculate the average effect of all possible messages of a given length, thus providing comprehensive coverage of all possible conditions.

This comprehensiveness and efficiency is based, however, on the assumptions of linearity and time invariance. Time variations due to control loops (such as clock recovery) are not modeled, and neither are any nonlinearities in the driver.

The comprehensiveness of these techniques can also be a disadvantage, for example if one is interested in the response to a particular data pattern. The very fact that the analysis takes so many different possible messages into account masks the effect of any one message.

3.2 Time Domain Simulations

For time domain simulations, the overall approach is to perform a sampled time simulation in which any effects of interest are explicitly included in the waveforms. Traditionally, time domain simulations have been used to generate the waveforms for the desired signals, and then the effects of stochastic impairments such as noise and crosstalk have been accounted for analytically, resulting in a so-called semi-analytic estimate of the bit error rate.

This approach has the advantage that it is based on very few assumptions. Elements of the model can be nonlinear, time-varying, or both, thus making it practical to study system behavior with almost unlimited detail. It is also straightforward to generate results that can be correlated directly with measurement.

The avoidance of assumptions also means that one cannot take advantage of as many assumptions to improve computational efficiency. The result is that time domain simulations necessarily take a lot longer to run. This also means that only a small sample of the population of possible messages can be simulated, and the relatively small sample size results in a significant variation in estimated bit error rate.

4. Demonstration System

The demonstration of the AMI standard shows how functions conforming to the API can be used to model an end to end transmission path. The model consists of three major

elements: a driver, a transmission path, and a receiver. The impulse response of the transmission path is supplied to the driver, and the impulse response of the combined driver and transmission path is supplied to the receiver. Both the driver and receiver models output an impulse response and parameter values as well as a time domain waveform.

For the sake of convenience, the data rate was set at 5 Gb/s for the simulations; however, there is nothing in either the models or the execution environment that would prevent simulations from being run at other data rates.

The sample interval was 25ps, or eight samples per bit, which has proven to yield reasonable results with suitable simulator performance.

4.1 Driver

The driver is modeled as a four tap finite impulse response equalizer. The tap coefficients are normalized internally so that the sum of their absolute values is one, and a separate parameter is used to set the output swing. The default configuration consists of one pre-cursor tap for feed forward equalization (FFE), a main tap, and two post-cursor taps for de-emphasis equalization.

4.2 Transmission Path

The transmission path is a lossy transmission line. Causality effects due to both dielectric loss tangent [4] and conductor internal impedance [5] are accounted for explicitly; however, the termination impedances are assumed to be perfectly matched.

Other simulations, were run using impulse responses derived from SPICE models and input as comma separated variable (CSV) files; however, the results of those simulations are not included in this paper.

4.3 Receiver

The receiver includes a peaking filter, clock recovery, and Decision Feedback Equalization (DFE). The peaking filter is statically configured while the clock recovery and DFE are fully adaptive.

Both the clock recovery and DFE control loops have four modes:

- Off: The control loop is disabled and its output is zero.
- Fixed: The control loop is disabled and its output is set to a fixed value that has been input as a model setting.
- Auto: The output of the control loop is initialized to an expected optimum by AMI_Init() and then allowed to adapt during the time domain simulation.
- Adaptive: The output of the control loop is initialized to zero and then allowed to adapt during the time domain simulation, thus simulating the response to a “cold start”.

While most of the simulations were performed in “Auto” mode, the other modes were indispensable for creating controlled experiments for debugging or detailed investigation.

4.3.1 Peaking Filter

The peaking filter can assume any of sixteen configurations, and is implemented as a conventional digital filter using bilinear Z transform techniques [6]. That is, the Laplace variable s was approximated as

$$s \approx \frac{2}{\tau}\frac{z-1}{z+1}$$

where τ is the sample interval and z^{-1} is the unit sample delay operator.

For the 5 Gb/s data rate, the peaking filter has two poles fixed at -5 GHz and -4 GHz, and a single zero which varied in geometric steps from -380 MHz to -2.5 GHz. The gain vs. frequency is plotted in Figure 2.

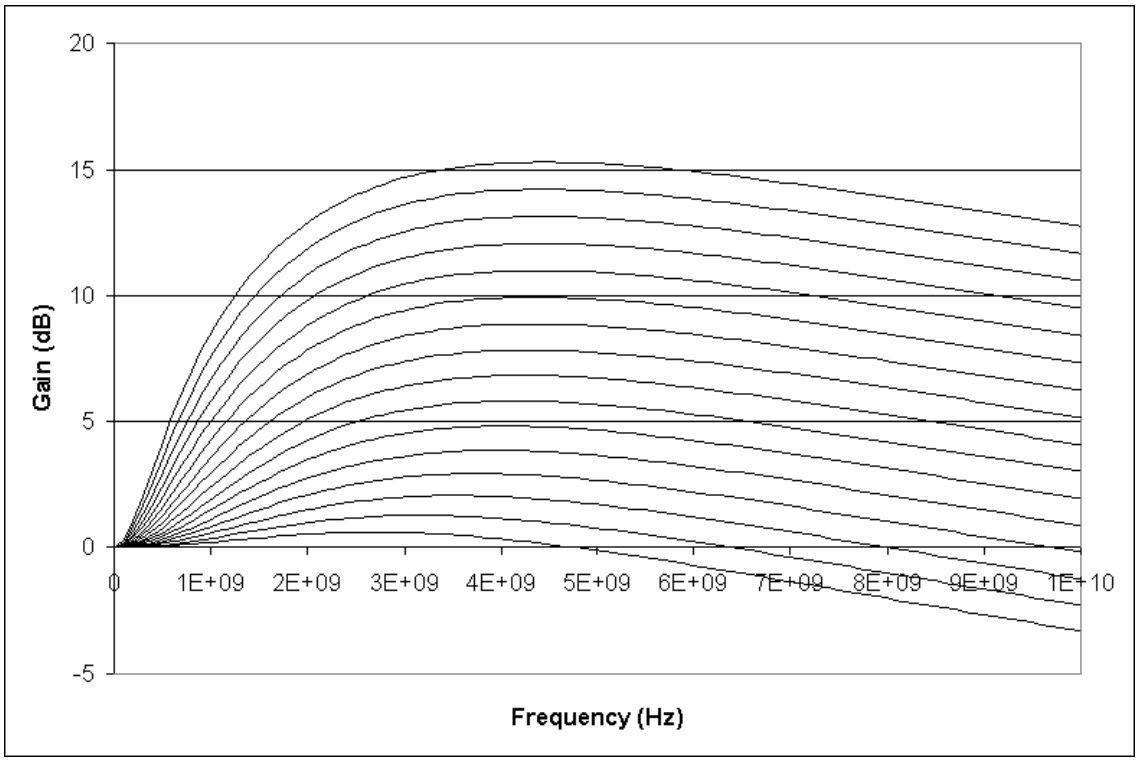


Figure 2: Receive peaking filter gain vs. frequency

As is clearly evident from Figure 2, such a peaking filter would admittedly be extremely difficult to design. The gain could be allowed to drop off faster above 5 GHz without affecting the link performance; however, the design would still be very difficult at best.

This set of poles and zeros was chosen, nonetheless, to illustrate a couple of points:

1. While most linear equalizer designs achieve their variation of gain with frequency by introducing loss at DC, this often results in receivers which fail due to insufficient receive amplitude rather than any inability to equalize the channel. Their receive eye diagrams are beautiful but there isn't enough signal left to detect the data. This set of poles and zeros was chosen to have unity gain at DC and to introduce gain at the higher frequencies.
2. Using a single zero and as few poles as possible results in a very effective equalizer. This is true regardless of the overall gain. The point is that more poles and zeros don't improve performance- they detract from it.

4.3.2 Clock Recovery

The clock recovery is performed by a first order early/late gate clock recovery loop with a conventional design. It operates as follows:

1. Data latched a half symbol before/after the output data is compared to the output data. When the bit latched before the output data matches the output bit but the bit latched after the output data does not, that is interpreted as an "early" event. The opposite set of match/mismatch conditions is interpreted as a "late" event.
2. An up/down counter counts the imbalance between "early" and "late" events.
3. When the up/down counter overflows with a count of eight, the recovered clock phase is incremented or decremented by 1/125 of a bit period.

Both the clock recovery overflow count and the phase step are input parameters to the model, so one could experiment with other values for these parameters.

Reference clock offset is modeled by calculating in double precision floating point the time at which the receive reference clock has a rising or falling edge. The reference clock offset as a fraction of the clock frequency is input as a parameter into the receiver model. A similar mechanism is also used to model duty cycle distortion in the receive reference clock.

4.3.3 Decision Process

The data decision process in the receiver is modeled by linearly interpolating the waveform at the receiver decision point to obtain and estimate of the receive signal exactly at the (double precision) sample time from the clock recovery loop. This allows the simulations to run with fewer samples per bit and still accurately model the decision process.

The receiver decision process also explicitly models the phenomenon of minimum latch overdrive or "metastability". While every decision circuit is guaranteed to eventually output a one or a zero, finite gain and parasitic capacitance in the circuit will delay the time at which that decision is rendered. When the input amplitude is low enough, that decision delay will be long enough that the decision circuit output will fail to meet the setup time of the next latch in the data path.

This phenomenon is modeled in decision circuit model by adding the minimum latch overdrive to the receive threshold when the previous output bit was a zero, and

subtracting the minimum latch overdrive from the receive threshold when the previous bit was a one.

In addition, it was observed that this type of model could be used to model error propagation in DFE equalizers. (We are indebted to Brian Kirk for this observation.) We therefore added a feature to inject a decision error (ostensibly due to unspecified causes) at the output of the decision circuit. Since the error is injected at a specific time and in a reproducible way, we can therefore observe the difference in output waveform at that time with and without the injected error.

4.3.4 DFE

The DFE has four taps, and uses the clock timing from the clock recovery loop. The DFE control loop operates on the principle that its goal is to minimize the correlation between the detected data and the signal at the receiver decision point.

In the AMI_Init() function, the optimum DFE tap settings are estimated by computing the pulse response at that receiver decision point and then assuming that the recovered clock will converge to the time at which the pulse amplitude is largest. The DFE tap settings are then estimated as the negative of the integral of the pulse response in adjacent bit periods, as illustrated in Figure 3.

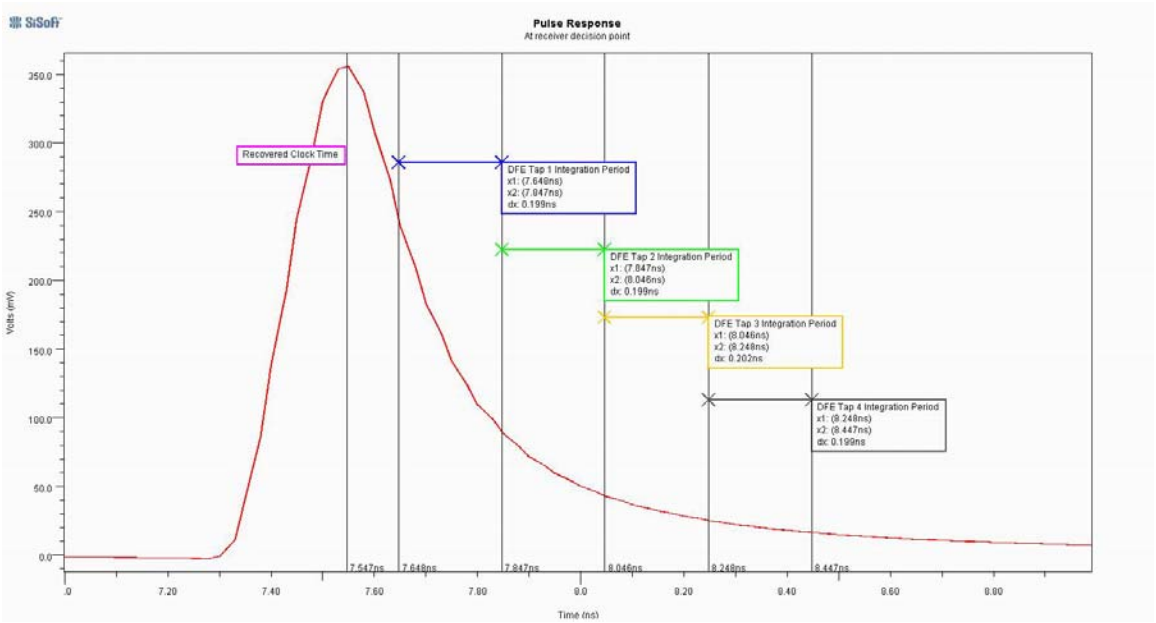


Figure 3: DFE tap estimation integration periods

Given pulse response $p(t)$ and bit period T, the equation for the estimated tap weight for the N-th DFE tap is

$$W_N \approx -\int_{(N-\frac{1}{2})T}^{(N+\frac{1}{2})T} p(t)dt$$

The contribution from the DFE taps are included in the impulse response output of the AMI_Init() function.

In the AMI_GetWave() function, the equation for the *N*-th tap weight is

$$W_N = -g\int v(t)D(t-NT)dt$$

where $v(t)$ is the voltage at the decision point (including the contribution from the DFE taps) and $D(t)$ is the output of the decision circuit. For the results reported here, the DFE control loop gain g is $3\text{x}10^{-6}$.

4.4 Execution Environment

One of the disadvantages of a million bit simulation is that it can consume and produce a great deal of data. Several features were added to the execution environment to cope with all the data, as illustrated in Figure 4.

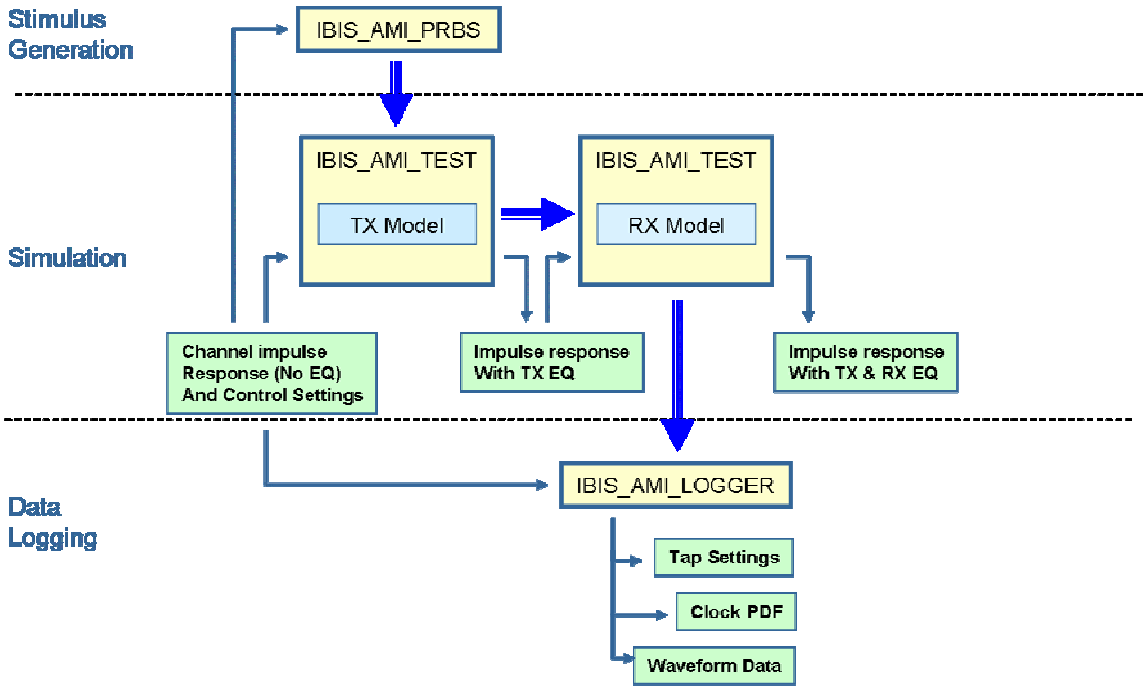


Figure 4: Execution environment flow diagram

The data is produced by a Linear Feedback Shift Register (LFSR) data generator with a register length of 7, 15, or 22 bits. Most simulations were run with the 22 bit length. The register length is specified in the simulation control file which also contains the channel impulse response and the control parameters for the transmitter model.

The output of the data generator is piped to a copy of the IBIS AMI model testing program loaded with the transmitter model. The output of this program is piped to a second copy of the model testing program loaded with the receiver model.

For a simulation of this type, it is usually only the waveform near the end of the simulation that is of interest. The execution environment was therefore designed to record output data at a time specified by the user. This function is performed by a separate program. The waveform recording start and stop times are also contained in the simulation control file.

While only the waveform at the end of the run is typically of interest, the behavior of the control loops throughout the run are often of primary interest. Each call to AMI_GetWave() returns a sample of the output parameters, and these samples are accumulated separately from the output waveform. Note that the sampling rate for the control loop monitoring is therefore determined by the block size in the AMI_GetWave() calls. This feature therefore provides a very flexible development environment for the model developer.

Finally, the data logger accumulates a probability density function (PDF) of the recovered clock from the clock times output by the AMI_GetWave() function. At the end of the simulation, it also calculates a Gaussian function to fit this PDF.

The shell script to run this simulation is

```
./clean_test
IBIS_AMI_test -f ./libIBIS_AMI_Tx.so -i tx_config.csv -c
cp tx_config_out.csv rx_config.csv
IBIS_AMI_prbs -f tx_config.csv | IBIS_AMI_test -f ./libIBIS_AMI_Tx.so -
i tx_config.csv -g -c | IBIS_AMI_test -f ./libIBIS_AMI_Rx.so -i
rx_config.csv -g -c | IBIS_AMI_logger -f rx_log.csv -c tx_config.csv -w
rx_test_out.csv -n
```

5. Results

Simulations of up to a billion bits were run using the AMI models; however, most runs were limited to a million bits. Execution time for this model is consistently faster than a half million bits a minute.

5.1 Comparison of Equalization Options

There are three equalization techniques in the model, and therefore eight possible combinations of equalization solution. All eight combinations were run, providing an interesting comparison between the relative merits of each solution. Figure 5 shows an example of several impulse responses output from AMI_Init(). Note that the contribution from the DFE taps is clearly evident along with the effects of the transmit de-emphasis and the peaking filter.

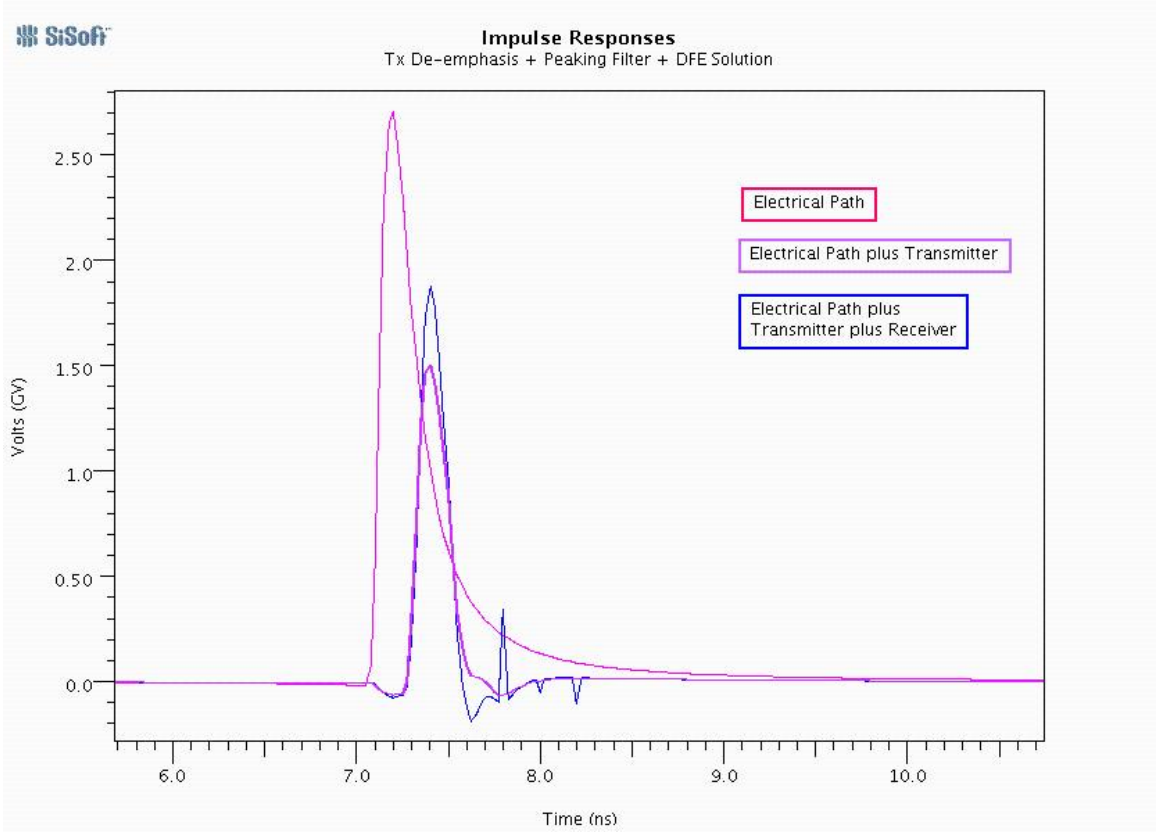
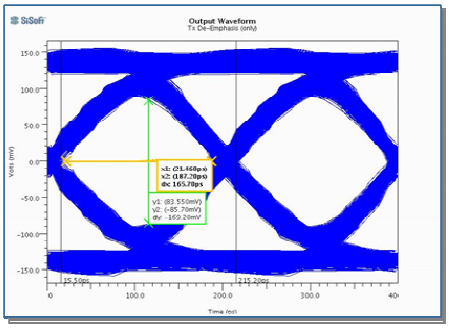


Figure 5: AMI_Init() output impulse responses

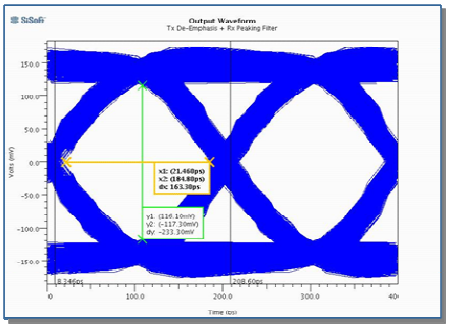
Figure 6 shows the eye diagrams obtained after a million bits for each combination of equalization techniques.

With TX EQ

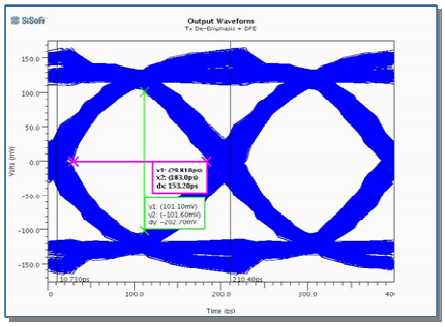
TX EQ only: 166ps, 169mV



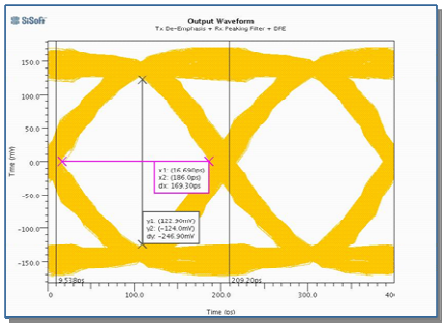
TX EQ & RX Peaking: 163ps, 233mV



TX EQ & RX DFE: 153ps, 203mV

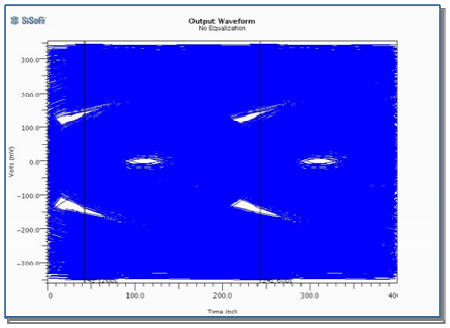


Full TX & RX EQ: 169ps, 247mV

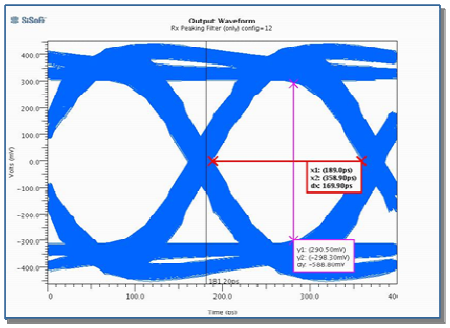


Without TX EQ

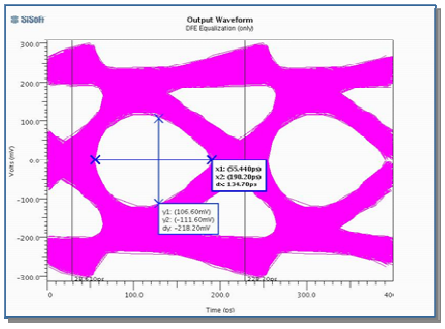
No EQ: no measurable eye



RX peaking: 170ps, 589mV



RX DFE: 135ps, 218mV



RX peaking & DFE: 176ps, 656mV

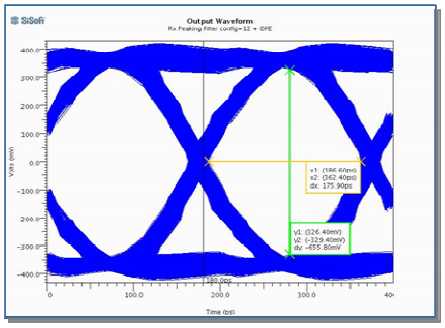


Figure 6: Eye diagrams with all combinations of equalization techniques

In Figure 6, the optimal peaking filter configuration when used with the transmit de-emphasis is configuration 3, with a maximum gain of 2.9dB; whereas when the peaking filter is used without the transmit de-emphasis, the optimal configuration is configuration 11, with a maximum gain of 11.0dB. The difference in gain is reflected in the maximum eye heights for the cases.

Note also from Figure 6 that while the eye diagrams look pretty good for all cases that include transmit de-emphasis, the cleanest eye is obtained with a combination of receive peaking filter and DFE, but no transmit de-emphasis.

Figure 7 shows the DFE taps as a function of time for a million bit run.

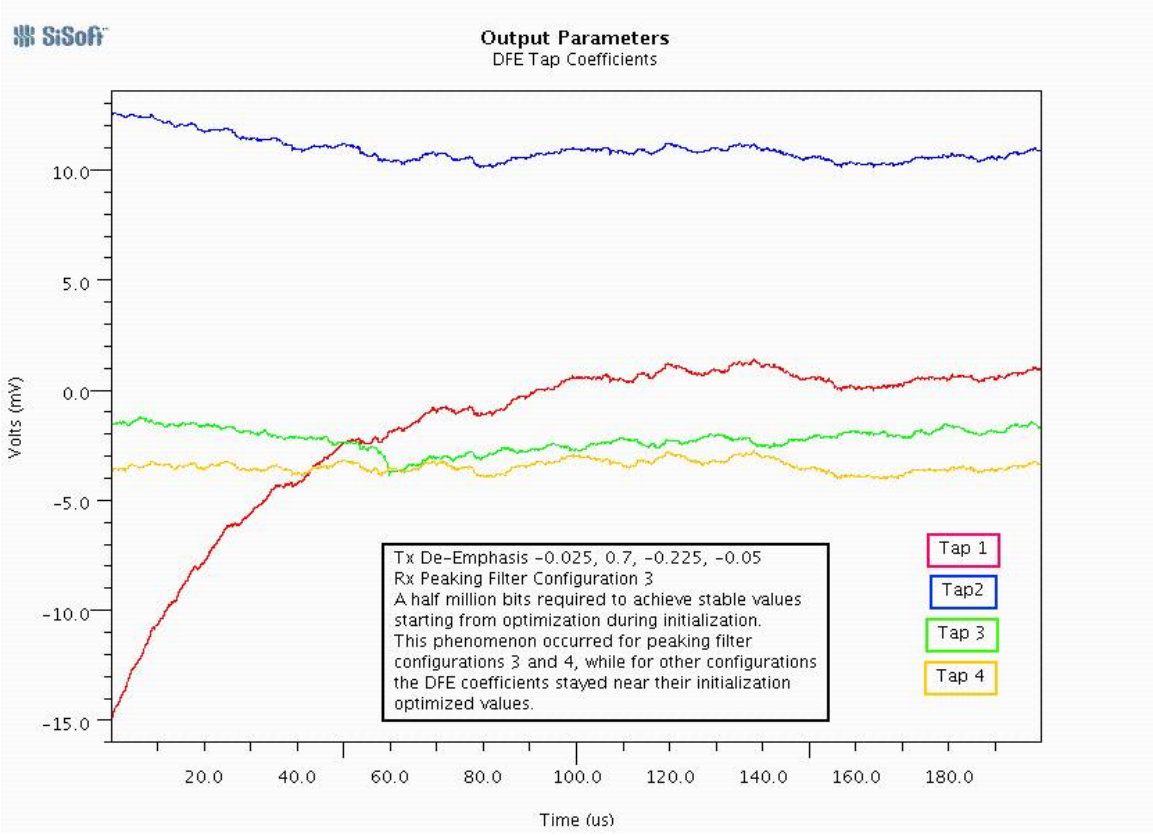


Figure 7: DFE tap weights vs. time

Note that, similar to the experience stated by some IP vendors, it took about a half million bits for the taps to reach steady state. Further investigation determined that, at least for this model, the value of especially the first DFE tap was a very sensitive function of recovered clock timing, and therefore AMI_Init() needed a more accurate estimate of the recovered clock time in order to predict the optimum tap setting accurately.

Figure 8 is an example of the recovered clock phase, in this case with different receiver sensitivities.

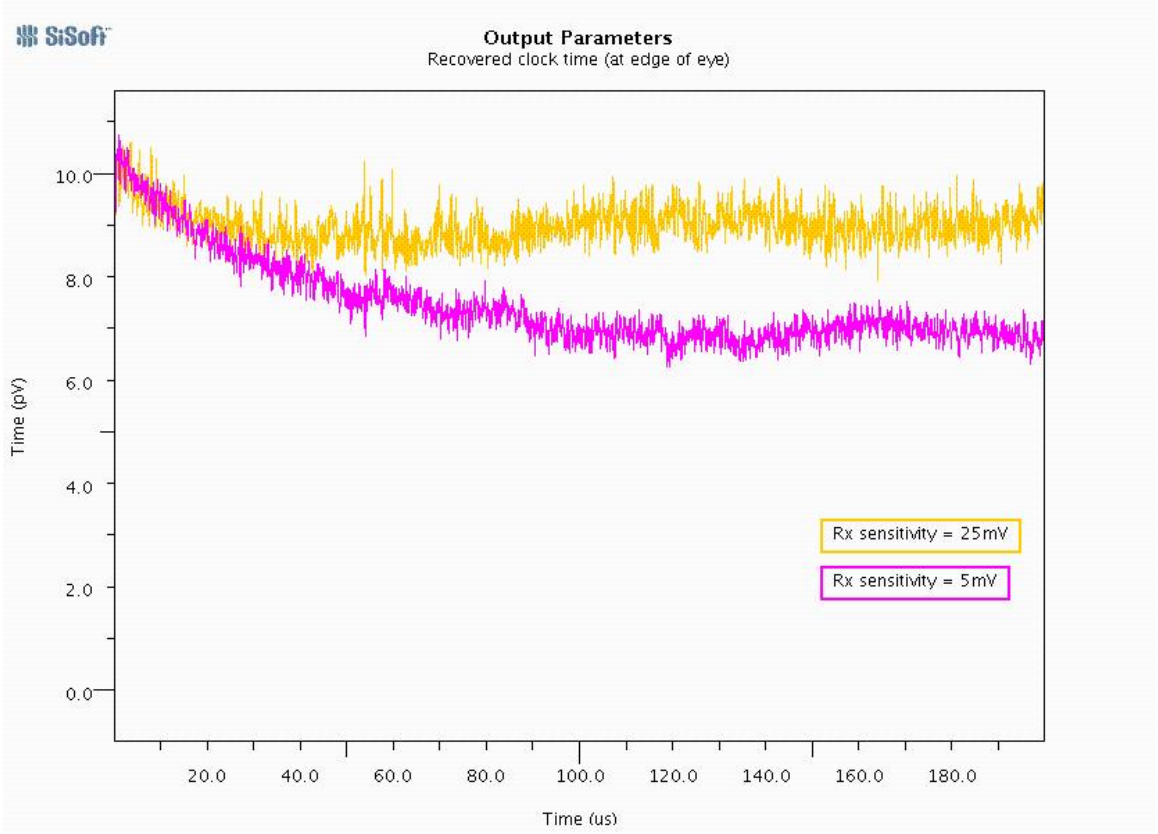


Figure 8: Example of recovered clock phase

There are two observations that can be made. First of all, note that the recovered clock phase variations mirror the variation in DFE tap 1 in Figure 7. These variations are very small, but they were observed consistently when DFE was enabled, and otherwise not. This is therefore some evidence supporting the theory that there is an interaction between clock recovery and DFE control loops.

Note also in Figure 8 that the receiver sensitivity appears to affect the recovered clock phase at least by a few picoseconds.

Figure 9 shows an example of a recovered clock PDF.

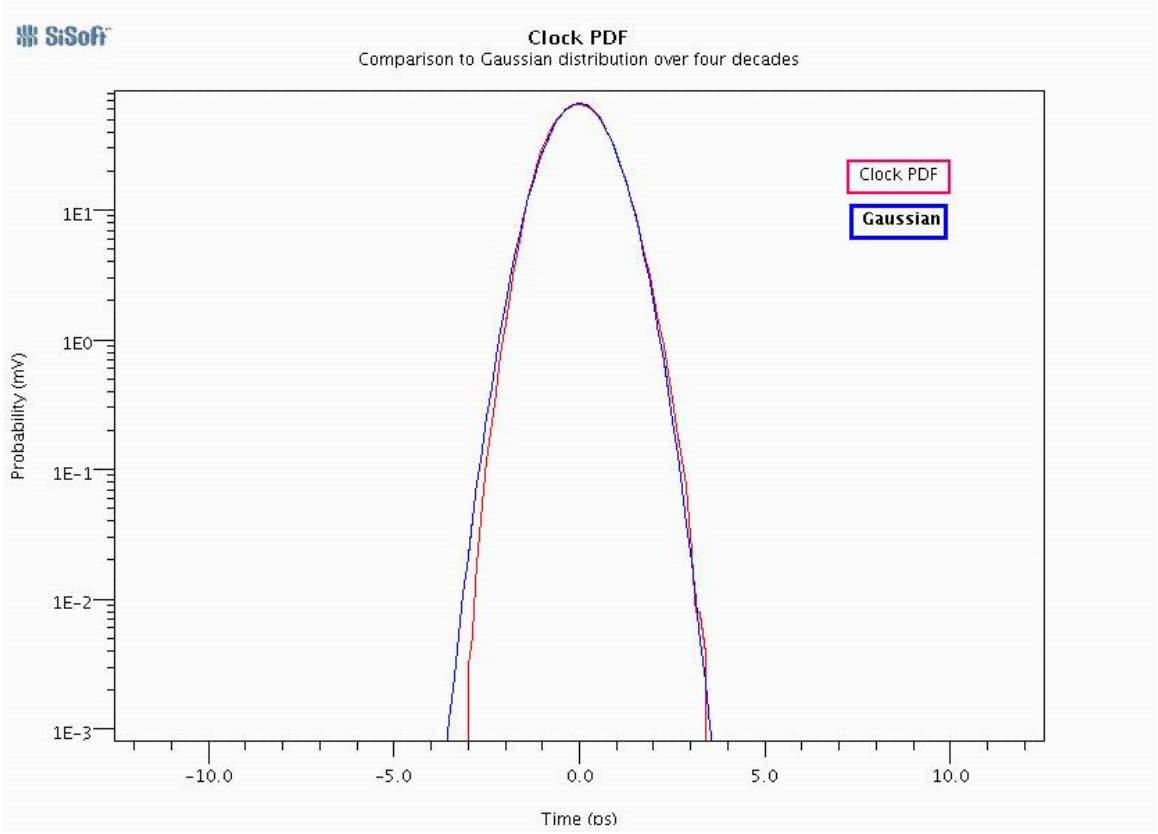


Figure 9: Recovered clock PDF

Note in Figure 9 that for the first four decades of probability density, the clock PDF closely resembles a Gaussian. When these results are plotted with more decades, however, the recovered clock PDF drops off more rapidly than the Gaussian because of the limited sample size that can be obtained in a time domain simulation. These results therefore cannot determine whether or not the recovered clock PDF is accurately approximated by a Gaussian for all probabilities of interest.

5.3 DFE Error Propagation

Using the error injection feature of the receiver model, we investigated the extent of error propagation due to a DFE equalizer. Figure 10 shows a typical result in which the output waveform with error injection can be compared to an error-free waveform.

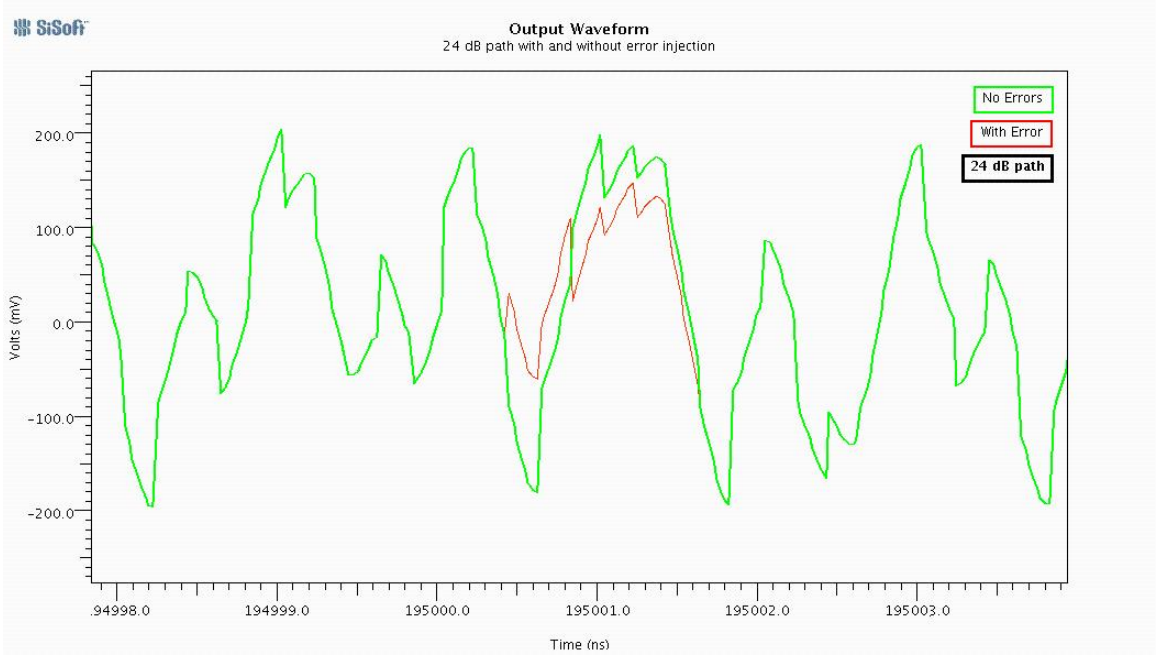


Figure 10: DFE error propagation induced by error injection

Figure 10 shows an error burst of perhaps four bits. It is interesting to note that one strong data edge is enough to stop any further error propagation, and thus any significant impact on the DFE control loop. The impact on downstream error detection/correction algorithms is another matter entirely.

6. Conclusion

The algorithmic model API defined by the IBIS AMI standard has proven to offer an efficient and versatile simulation environment for both system developers and SerDes model developers. The combination of visibility into the control loops and clock path as well as the data path makes it convenient to study tradeoffs in equalization, clock recovery, and adaptive control.

References

[1] Kumar, Shah, Varma, Dodd, Hawes, Angulo, Muranyi, Katz, Steinberger and Westerhoff, “Algorithmic Modeling API (AMI) Support in IBIS”, IBIS BIRD 104, submitted 10 October 2007.

[2] Anthony Sanders, Mike Resso, John D’Ambrosia, “Channel Compliance Testing Utilizing Novel Statistical Eye Methodology”, DesignCon 2004 Proceedings.

[3] Bryan Casper, Matthew Haycock and Randy Mooney, “An accurate and efficient analysis method for multi-Gb/s chip to chip signaling schemes”, VLSI Digest of Technical Papers, June 13, 2002, pages 54-7.

[4] Djordjević, Biljić, Likar-Smiljanić, and Sarkar, “Wideband Frequency-Domain Characterization of FR-4 and Time-Domain Causality”, IEEE Transactions on Electromagnetic Compatibility, Vol. 43, No. 4, November 2001.

[5] Ramo, Whinnery and Van Duzer, *Fields and Waves in Communication Electronics*, Section 4.5, pages 182-6, John Wiley and Sons, Inc., copyright 1994.

[6] Gold and Rader, *Digital Processing of Signals*, Section 3.10, pages 70-8, McGraw-Hill, Inc., copyright 1969.