Two for One: SerDes Flows for AMI Model Development

Corey Mathis, Ren Sang Nah (MathWorks)
Richard Allred, Todd Westerhoff (SiSoft)

DesignCon 2016 IBIS Summit
Santa Clara, California
January 22, 2016

* Adapted from the DesignCon 2016 paper “Leveraging SerDes Flows for AMI Model Development”

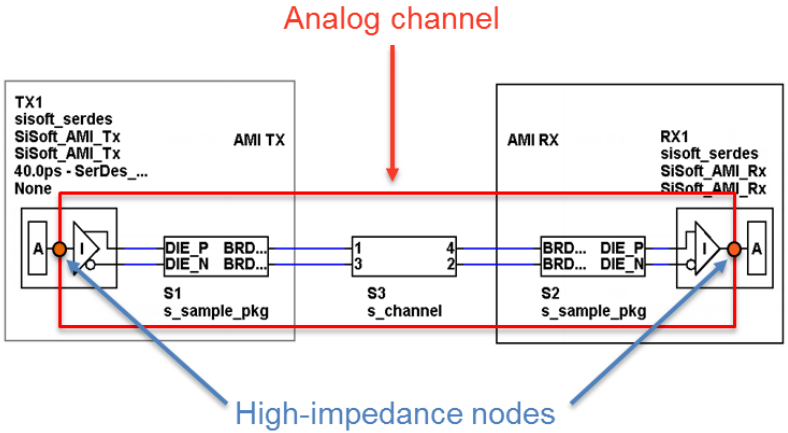
Background

- Good AMI models are hard to develop
  - Analog / algorithmic partitioning
  - IBIS-AMI requirements: samples per bit
  - Portability issues between EDA tools
- AMI development typically occurs “after the fact”
  - AMI Models are “customer collateral”
  - Created by different group
  - Limited testing before distribution

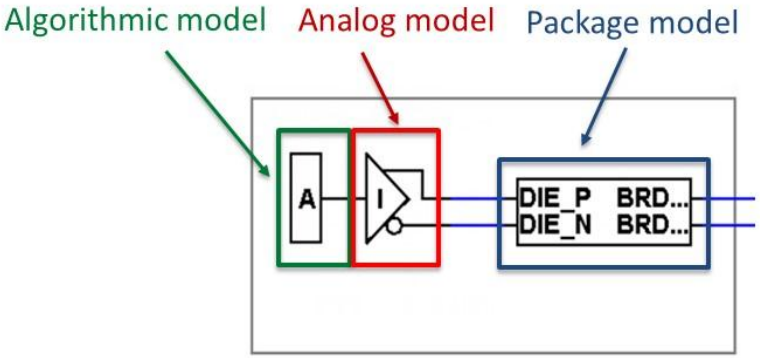
SerDes Design

- Architectural models
  - Created up-front to define architecture & budgets
  - Limited design detail, execute relatively fast
  - Good for design budgets & control loop behavior
- Implementation models
  - Detail varies from architectural to gate to circuit level
  - A “snapshot” of the design at a point in time
- This presentation is based on Architectural models

An AMI Model Primer



Fundamental Assumption



Model Components

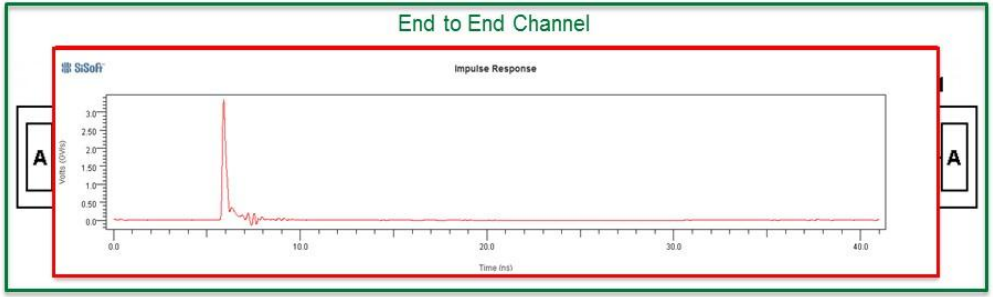
AMI Simulation Primer

AMI_Init
- Model configuration parameters
- Impulse response processing
- Linear, Time-Invariant (LTI)

AMI_Getwave
- Waveform processing
- Clock ticks
- Non-Linear, Time-Varying (NLTV)

AMI_Close
- Clean up & exit

Algorithmic Model

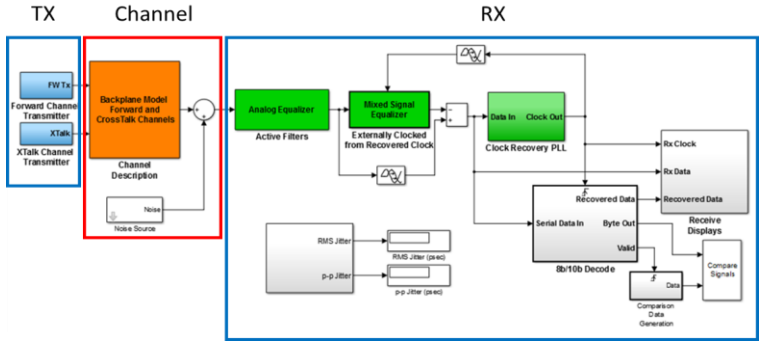


- Network Characterization (Circuit Simulation)
- Channel Simulation (Signal Processing)

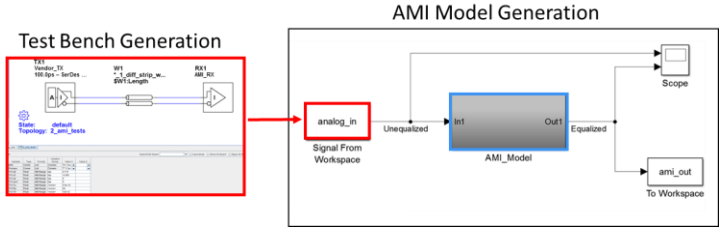
Channel Simulation

For more info: “Understanding IBIS-AMI Simulations”, DesignCon 2015

SerDes Architectural Exploration

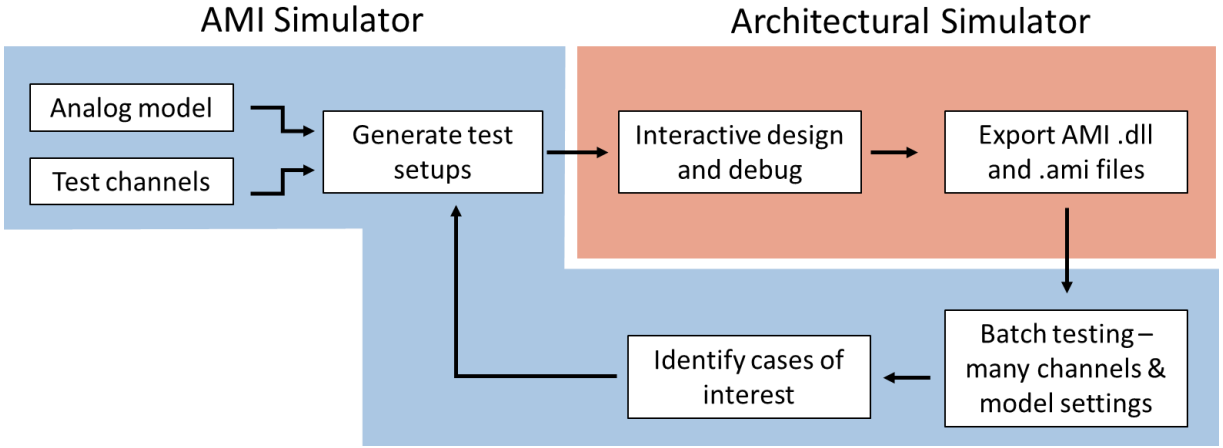


Typical flow: end to end link in Architectural simulator



Our flow: leverage strengths of Architectural and AMI simulators

IBIS-AMI Model Development Loop



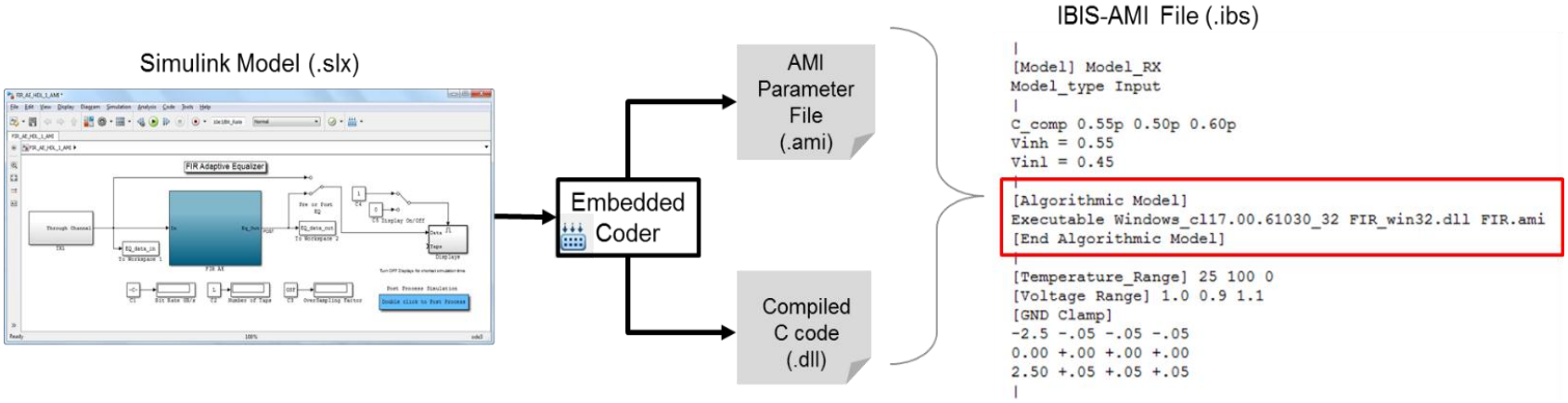
AMI Simulator

- Test case generation
- Batch & regression tests

Architectural Simulator

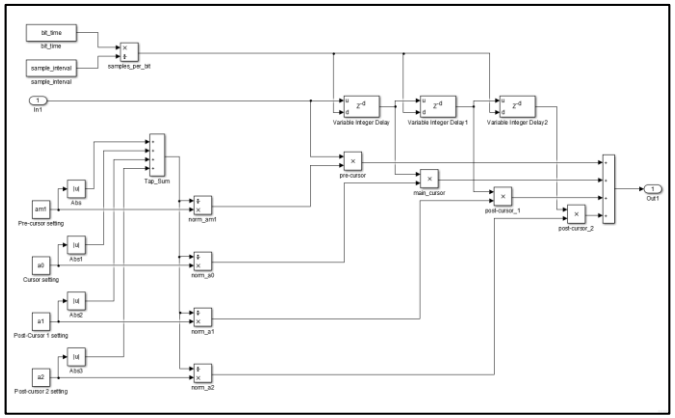
- Interactive design & analysis
- Test case debugging

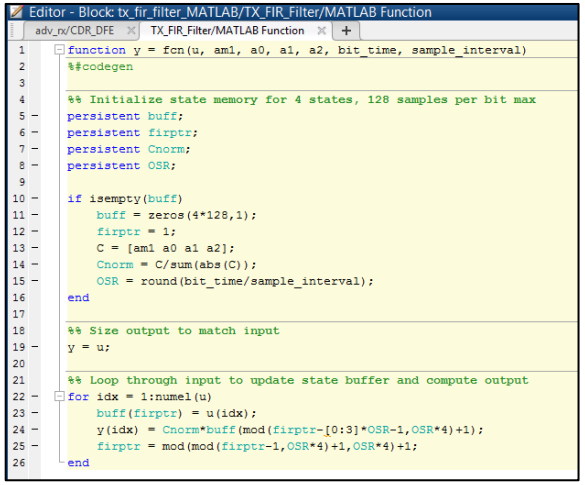
Leveraging Existing Infrastructure



Build on existing capabilities for embedded software code generation

Mixing Structure and Code

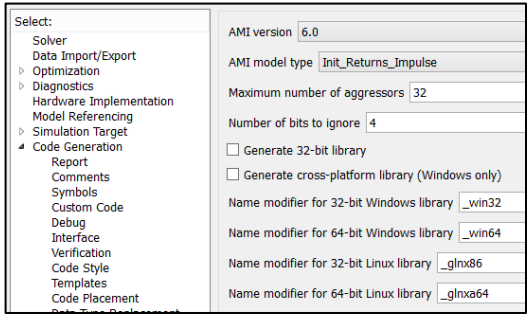




- Key parts of SerDes designs are often implemented as “code”
- Ability to mix structure and code is critical

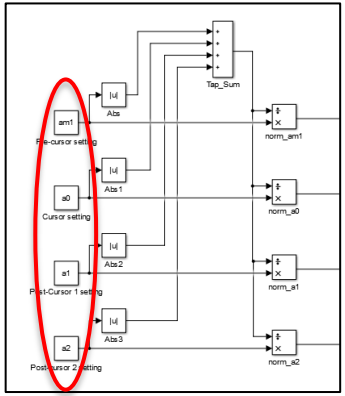
Creating Algorithmic Models

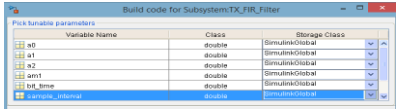
Identify model type



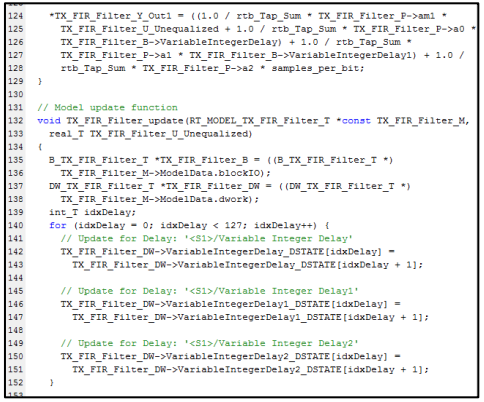
Based on design characteristics
(Init/LTI or Getwave/NLTV)

Identify AMI parameters



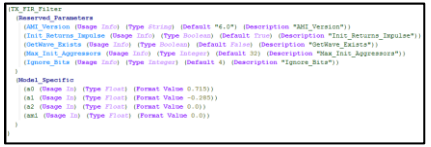


Generate Code & Compile



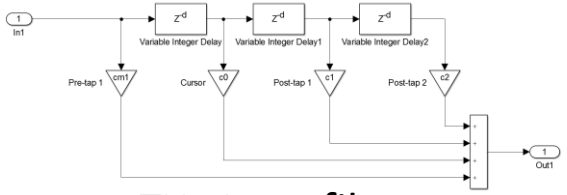
```
  *TX_FIR_Filter_Y_Out1 = ((1.0 / rtb_Tap_Sum * TX_FIR_Filter_P->am1 *
    TX_FIR_Filter_U_Unequalized + 1.0 / rtb_Tap_Sum * TX_FIR_Filter_P->a0 *
    TX_FIR_Filter_B->VariableIntegerDelay) + 1.0 / rtb_Tap_Sum *
    TX_FIR_Filter_P->a1 * TX_FIR_Filter_B->VariableIntegerDelay1) + 1.0 /
    rtb_Tap_Sum * TX_FIR_Filter_P->a2 * samples_per_bit;
}

// Model update function
void TX_FIR_Filter_update(RT_MODEL_TX_FIR_Filter_T *const TX_FIR_Filter_M,
  real_T TX_FIR_Filter_U_Unequalized)
{
  B_TX_FIR_Filter_T *TX_FIR_Filter_B = ((B_TX_FIR_Filter_T *)
    TX_FIR_Filter_M->ModelData.blockIO);
  DW_TX_FIR_Filter_T *TX_FIR_Filter_DW = ((DW_TX_FIR_Filter_T *)
    TX_FIR_Filter_M->ModelData.dwork);
  int_T idxDelay;
  for (idxDelay = 0; idxDelay < 127; idxDelay++) {
    // Update for Delay: '<S1>/Variable Integer Delay'
    TX_FIR_Filter_DW->VariableIntegerDelay_DSTATE[idxDelay] =
      TX_FIR_Filter_DW->VariableIntegerDelay_DSTATE[idxDelay + 1];

    // Update for Delay: '<S1>/Variable Integer Delay1'
    TX_FIR_Filter_DW->VariableIntegerDelay1_DSTATE[idxDelay] =
      TX_FIR_Filter_DW->VariableIntegerDelay1_DSTATE[idxDelay + 1];

    // Update for Delay: '<S1>/Variable Integer Delay2'
    TX_FIR_Filter_DW->VariableIntegerDelay2_DSTATE[idxDelay] =
      TX_FIR_Filter_DW->VariableIntegerDelay2_DSTATE[idxDelay + 1];
  }
```

C++ code



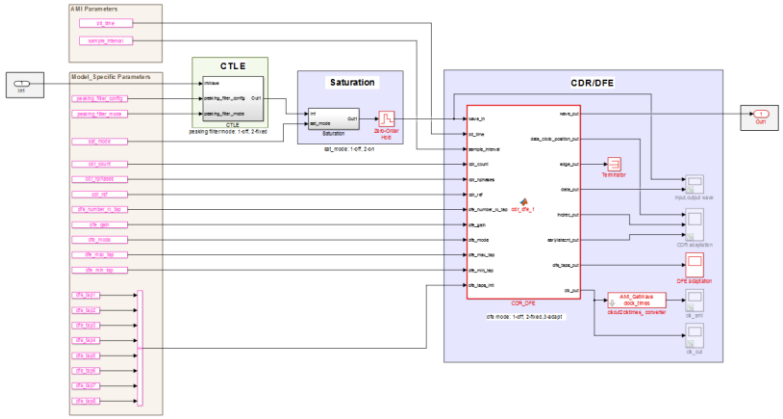
.ami file

IBIS-AMI Model Types



TX: 4 tap filter

Simple, linear, non-adaptive
➔ “Init” or LTI model



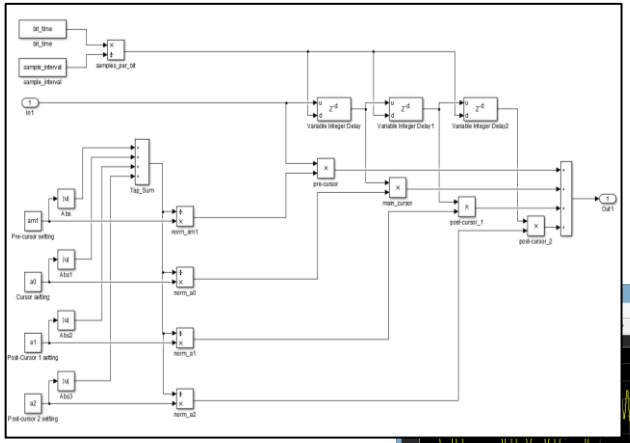
RX: CTLE, Saturation, 8 tap DFE, CDR

Complex, non-linear, adaptive
➔ “Getwave” or NLTV model

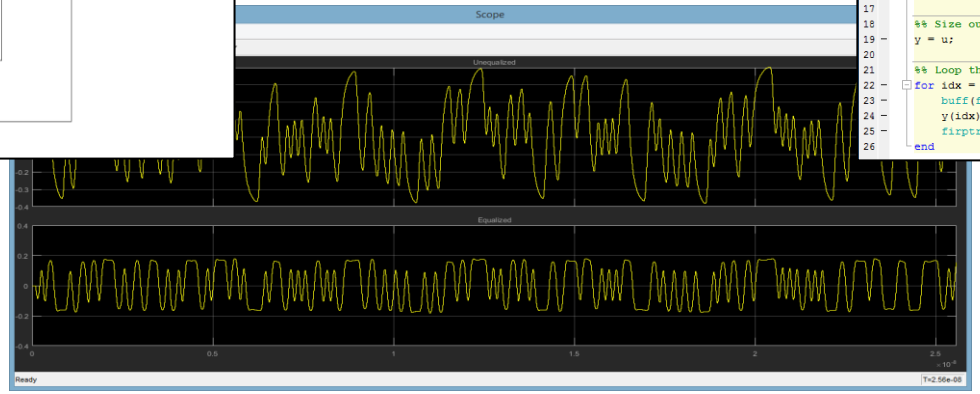
Design characteristics drive proper IBIS-AMI model type

A Simple AMI Transmitter

4 tap TX with normalization captured two ways



Structural model



Channel behavior with & without equalization

```
Editor - Block: tx_fir_filter_MATLAB/TX_FIR_Filter/MATLAB Function
simulink_csv_read.m | TX_FIR_Filter/MATLAB Function
1  function y = fcn(u, am1, a0, a1, a2, bit_time, sample_interval)
2  %#codegen
3
4  %% Initialize state memory for 4 states, 128 samples per bit max
5  persistent buff;
6  persistent firptr;
7  persistent Cnorm;
8  persistent OSR;
9
10 if isempty(buff)
11     buff = zeros(4*128,1);
12     firptr = 1;
13     C = [am1 a0 a1 a2];
14     Cnorm = C/sum(abs(C));
15     OSR = bit_time/sample_interval;
16 end
17
18 %% Size output to match input
19 y = u;
20
21 %% Loop through input to update state buffer and compute output
22 for idx = 1:numel(u)
23     buff(firptr) = u(idx);
24     y(idx) = Cnorm*buff(mod(firptr-[0:3]*OSR-1,OSR*4)+1);
25     firptr = mod(mod(firptr-1,OSR*4)+1,OSR*4)+1;
26 end
```

MATLAB code

Generated C++ Code

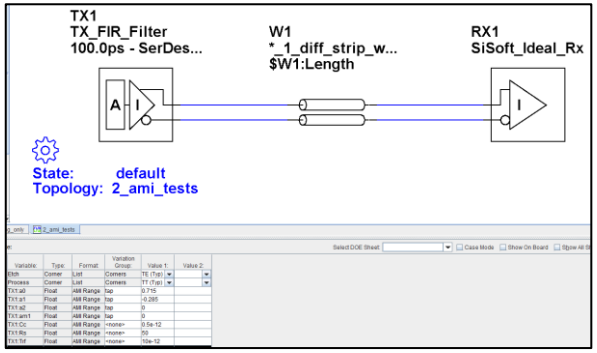
```
rtb_Tap_Sum = ((fabs(TX_FIR_Filter_P->am1) + fabs(TX_FIR_Filter_P->a0)) + fabs
               (TX_FIR_Filter_P->a1)) + fabs(TX_FIR_Filter_P->a2);

// Product: '<S1>/samples_per_bit' incorporates:
//   Constant: '<S1>/bit_time '
//   Constant: '<S1>/sample_interval '

samples_per_bit = TX_FIR_Filter_P->bit_time / TX_FIR_Filter_P->sample_interval;

// Delay: '<S1>/Variable Integer Delay' incorporates:
//   Delay: '<S1>/Variable Integer Delay1'
//   Delay: '<S1>/Variable Integer Delay2'
//   Inport: '<Root>/In1'

if ((samples_per_bit < 1.0) || rtIsNaN(samples_per_bit)) {
  TX_FIR_Filter_B->VariableIntegerDelay = TX_FIR_Filter_U_Unequalized;
  TX_FIR_Filter_B->VariableIntegerDelay1 =
    TX_FIR_Filter_B->VariableIntegerDelay;
  samples_per_bit = TX_FIR_Filter_B->VariableIntegerDelay1;
} else {
  if (samples_per_bit > 128.0) {
    samples_per_bit_0 = 128U;
  } else {
    tmp = floor(samples_per_bit);
    if (rtIsNaN(tmp) || rtIsInf(tmp)) {
      samples_per_bit_0 = 0U;
    } else {
      samples_per_bit_0 = (uint32_T)fmod(tmp, 4.294967296E+9);
    }
  }
```

Model behavior

```
//Bind model parameters
amiContainer->TX_FIR_Filter_P.a0 = param[0].p_val.dbl_val;
amiContainer->TX_FIR_Filter_P.a1 = param[1].p_val.dbl_val;
amiContainer->TX_FIR_Filter_P.a2 = param[2].p_val.dbl_val;
amiContainer->TX_FIR_Filter_P.am1 = param[3].p_val.dbl_val;
amiContainer->TX_FIR_Filter_P.bit_time = bit_time;
amiContainer->TX_FIR_Filter_P.sample_interval = sample_interval;

//Bind model data
amiContainer->TX_FIR_Filter_M->ModelData.defaultParam =
  &(amiContainer->TX_FIR_Filter_P);
amiContainer->TX_FIR_Filter_M->ModelData.blockIO =
  &(amiContainer->TX_FIR_Filter_B);
amiContainer->TX_FIR_Filter_M->ModelData.dwork =
  &(amiContainer->TX_FIR_Filter_DW);
```

AMI wrapper

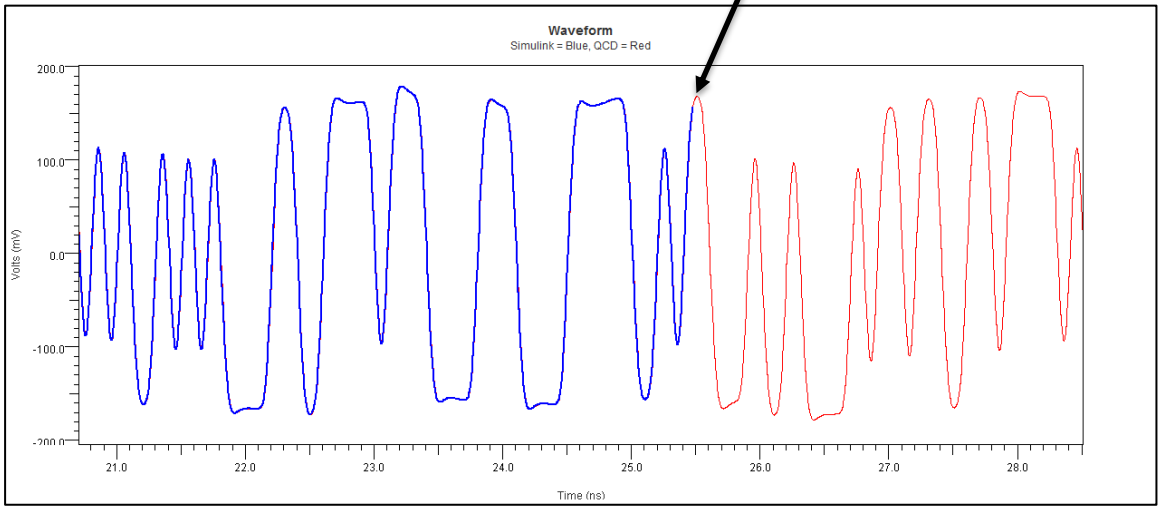
Architectural vs. AMI Results



Test bench

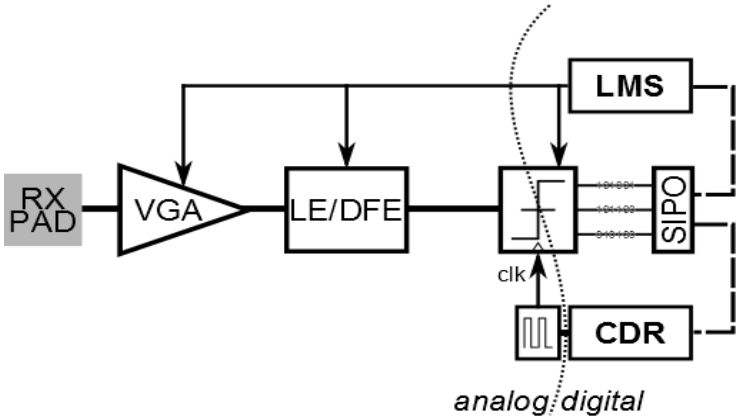
TX output resistance	50 ohms
TX output capacitance	0.5 pF
TX output voltage supply	1 volt
TX output rise/fall time	10 ps
TX EQ applied	Cursor = 0.715, 1st Postcursor = -0.285
Data rate	100ps
Channel model	Simple lossy stripline
RX input resistance	50 ohms
RX input capacitance	Zero
RX EQ applied	None

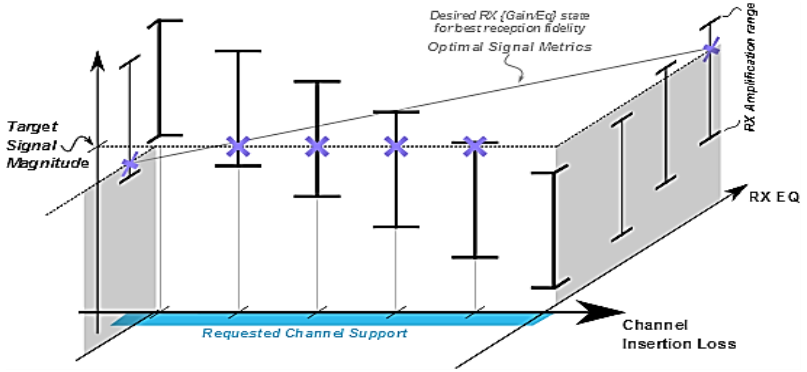
Architectural simulation ends,
AMI simulation run for more bits



Output waveform comparison

USB 3.0 Receiver Model

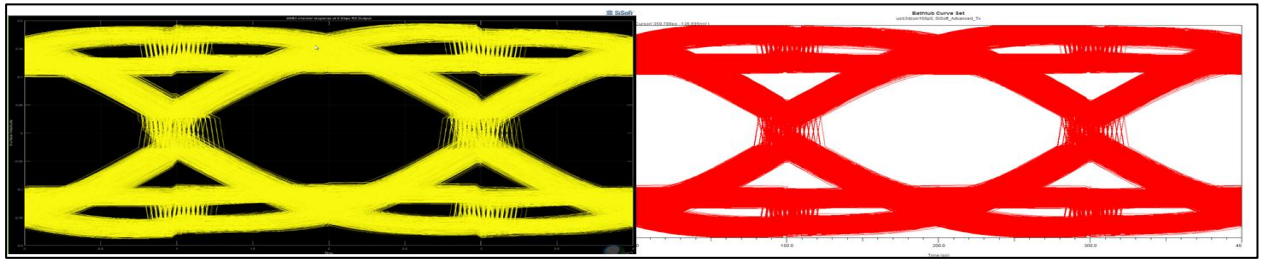




- Low power mobile receiver with multi-protocol support
- Design challenge: balance AGC, linear/non-linear EQ and CDR
- USB3.0 On-the-Go (OTG) support especially challenging

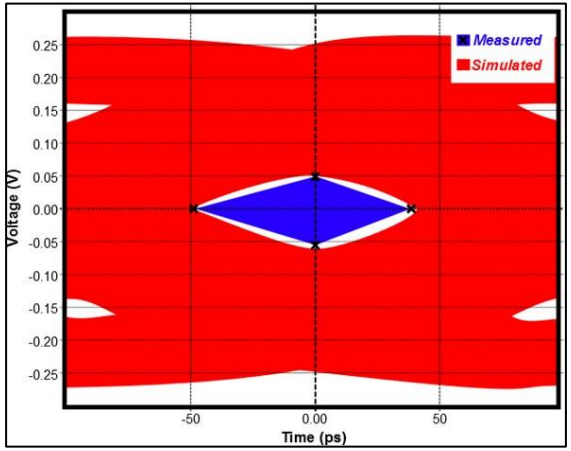
Model Correlation

Simulator to Simulator



Architectural Model | AMI Model

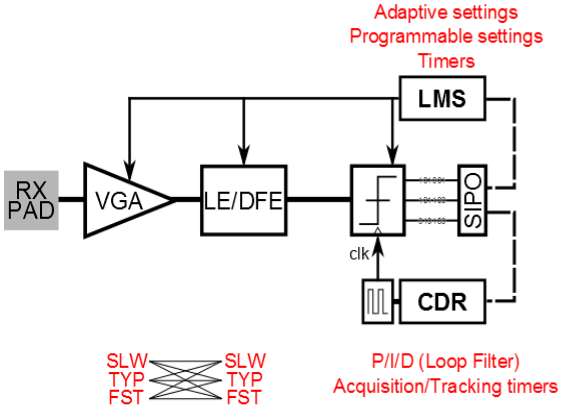
Simulator to Hardware



Receiver hardware reports eye height and width based on sampling clock

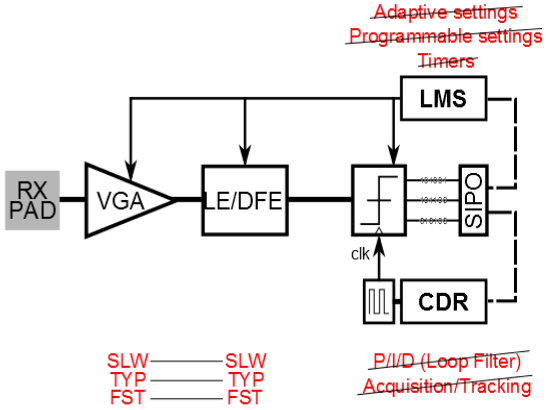
Other Findings

Internal models



- Expose “extra” controls and outputs
- Internal testing & design tuning
- Internal regression testing

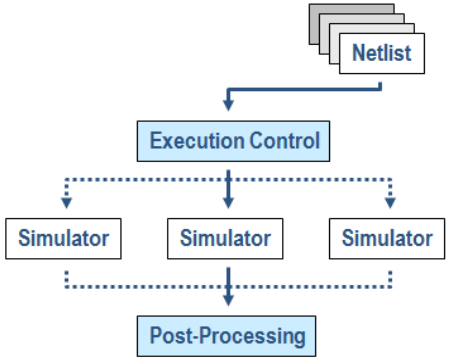
External (customer) models



- Fewer exposed controls & outputs
- Early models for key customer feedback

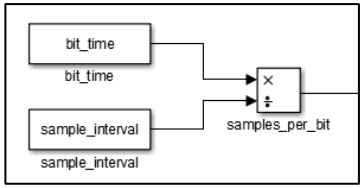
Other Findings

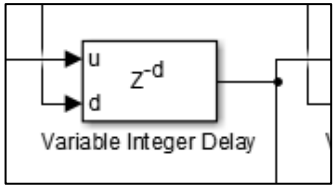
Throughput



- AMI models run 4-8x faster than their Architectural counterparts
- Running AMI simulations in parallel can provide ~150x speedup for regression testing

AMI Compliance





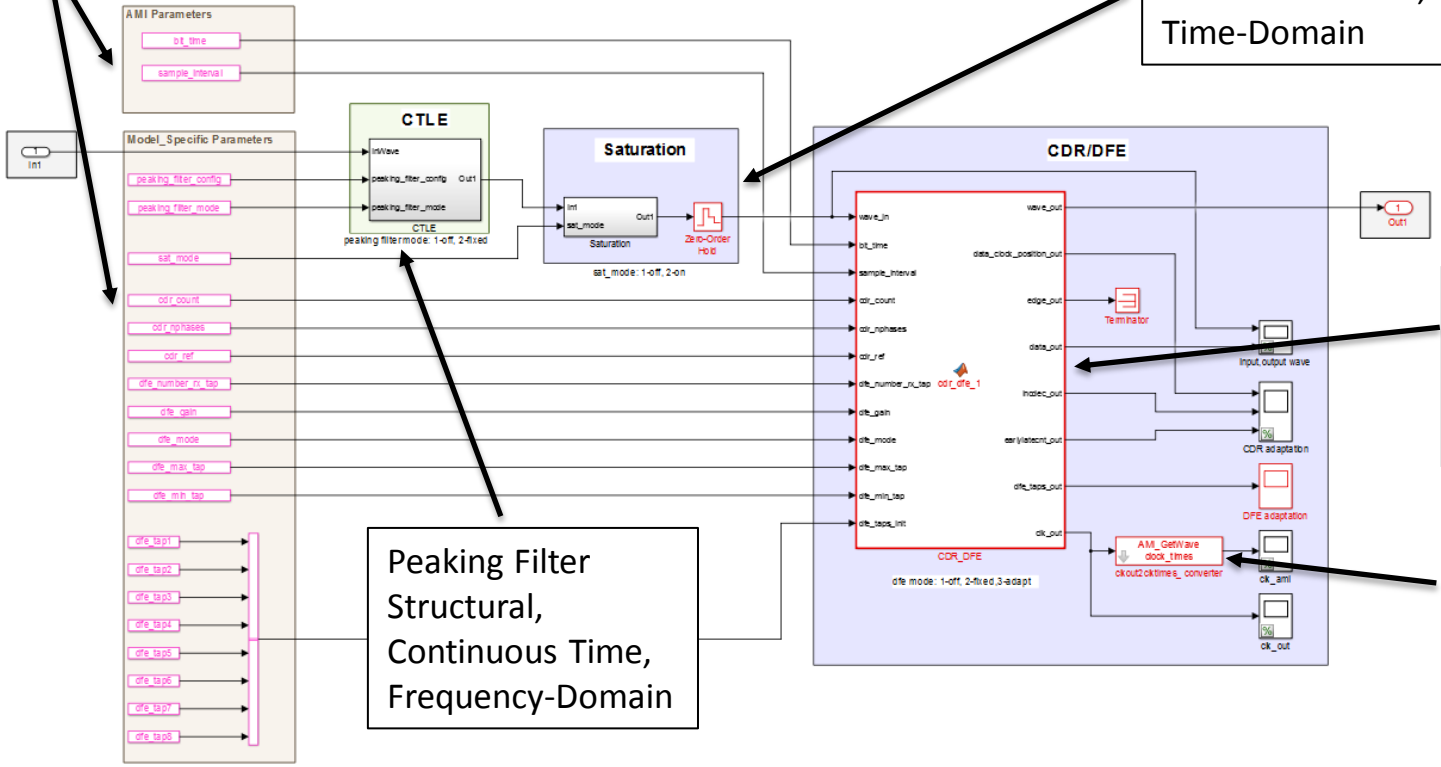

```
data_clock_position + (1+cdr_ref)*samples_per_bit-1;
```

- AMI requires models run at any setting of “samples per bit”
- Architectural models can be set up to meet this requirement

AMI RX Model

AMI Parameters

Saturation Block Code, Continuous Time, Time-Domain

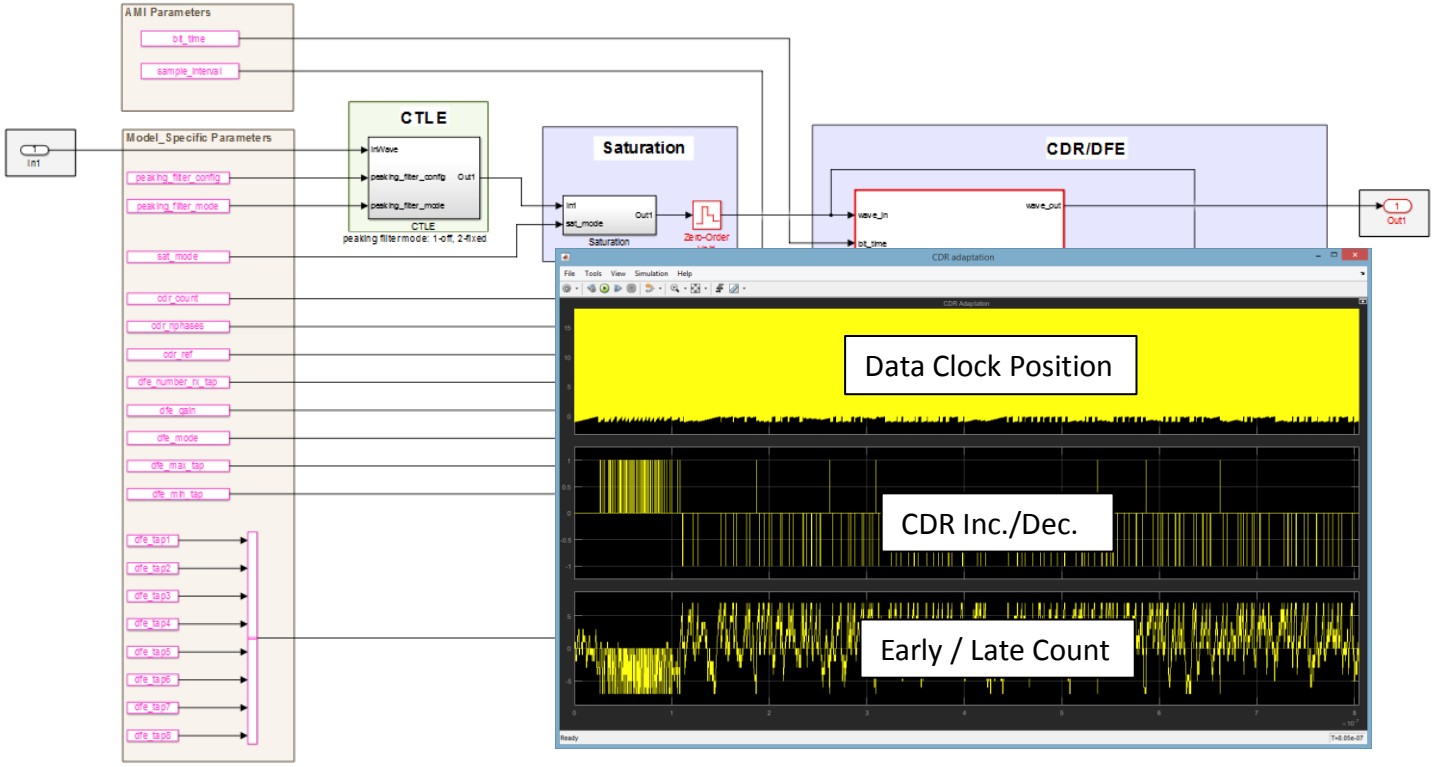


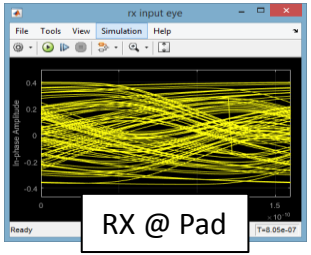
CDR/DFE Code, Sampled Time, Time-Domain

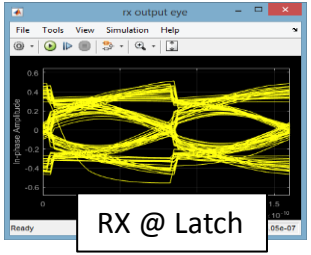
Peaking Filter Structural, Continuous Time, Frequency-Domain

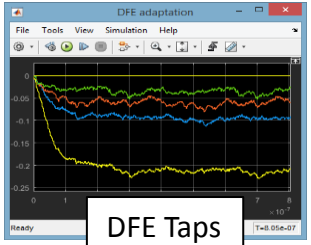
Clock Ticks output

AMI RX - Architectural Simulation

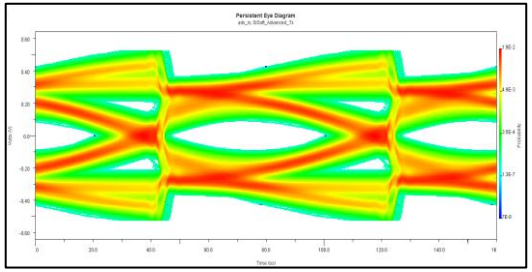




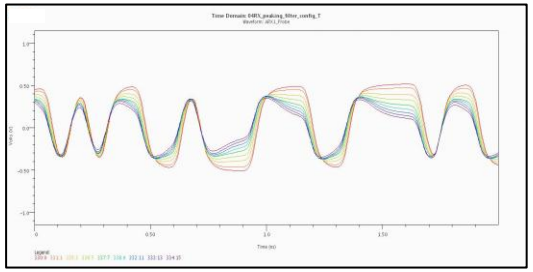




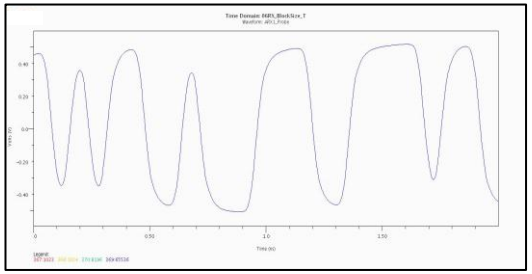
AMI RX – Compiled Model



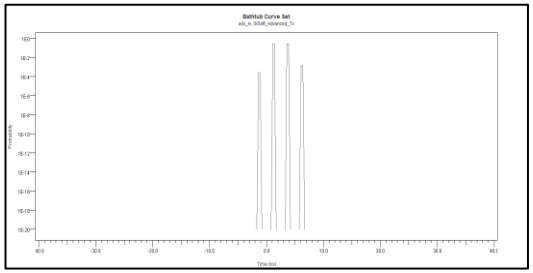
Eye Diagram Output



Control Inputs



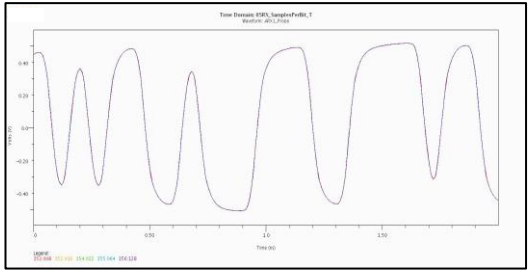
Compliance - Samples/Bit



Clock Output

Performance Test	Simulation Time	Reference Time	Relative Speed
Statistical	1 sec	1 sec	1.000x
TimeDomain_008spb	1.46 min/Mbit	1.30 min/Mbit	0.889x
TimeDomain_016spb	1.87 min/Mbit	1.46 min/Mbit	0.783x
TimeDomain_032spb	3.33 min/Mbit	2.60 min/Mbit	0.781x
TimeDomain_064spb	7.97 min/Mbit	6.02 min/Mbit	0.755x
TimeDomain_128spb	27.97 min/Mbit	22.77 min/Mbit	0.814x

Simulation Speed



Compliance - Block Size

- Comparable to hand-written models

Summary

- AMI models can be created from Architectural models normally created during the SerDes design cycle
- The parameters exposed in an AMI model can be varied depending on the application
- Models produced with this process behave just like any other well-constructed AMI model