IBIS-AMI with Different Languages

IBIS Summit, DesignCon, February 2008

Arpad Muranyi

© Mentor Graphics Corp., 2008. Reuse by written permission only. All rights reserved.
IBIS-AMI with Different Languages

IBIS Summit, DesignCon, February 2008

1. Review: The IBIS-AMI BIRD and Toolkits
2. The language of IBIS-AMI models
3. Comments on the Tx model in C
4. The Tx model in VHDL-AMS
5. The Tx model in Matlab
6. Benchmarks and Conclusions
The IBIS-AMI BIRD and Toolkits

- The IBIS-ATM committee spent about two years to define and write the IBIS-AMI BIRD
  - BIRD 104.1 was approved on Nov. 30, 2007
    http://www.vhdl.org/pub/ibis/birds/bird104.1.txt
  - two free toolkits have been posted
    http://www.vhdl.org/pub/ibismacromodelwip/archive/20080122/cadence/Cadence_IBIS_AMI_Evaluation_Toolkit_v2.0.zip
    http://www.vhdl.org/pub/ibismacromodelwip/archive/20080116/sisoft/SiSoft_IBIS-AMI_Eval_Toolkit_v2.03.zip

- Some IC vendors already have or are considering IBIS-AMI models for their SERDES products

- BIRD 104.1 is not part of the IBIS specification yet
  - expected to be included in the next version (v5.0)
IBIS-AMI with Different Languages

IBIS Summit, DesignCon, February 2008

1. Overview: The IBIS-AMI BIRD and Toolkits
2. The language of IBIS-AMI models
3. Comments on the Tx model in C
4. The Tx model in VHDL-AMS
5. The Tx model in Matlab
6. Benchmarks and Conclusions
The language of IBIS-AMI models

---

Keywords: [Algorithmic Model], [End Algorithmic Model]
Required: No

Description: Used to reference an external compiled model. This compiled model encapsulates signal processing functions. In the case of a receiver it may additionally include clock and data recovery functions. The compiled model can receive and modify waveforms with the analog channel, where the analog channel consists of the transmitter output stage, the transmission channel itself and the receiver input stage. This data exchange is implemented through a set of software functions. The signature of these functions is elaborated in section 10 of this document. **The function interface must comply with ANSI ‘C’ language.**

It has been stated in meetings and presentations that the IBIS-AMI model can be written in any language as long as its interface complies with ANSI C (possibly using wrappers)

The toolkit model examples are written in ANCI C
ANC1 C - a modeling language?

- **Who will write models?**
  - system or circuit designers are usually electronic engineers
  - communications experts are good with filter design, channel analysis techniques, etc…

- **Problem:**
  - no matter how brilliant the above people are, they may not have the experience in implementing FIR filters or convolution algorithms in ANSI C

- **Anybody interested in building a home-made car before driving off to a vacation with it?**
IBIS-AMI with Different Languages

IBIS Summit, DesignCon, February 2008

1. Overview: The IBIS-AMI BIRD and Toolkits
2. The language of IBIS-AMI models
3. Comments on the Tx model in C
4. The Tx model in VHDL-AMS
5. The Tx model in Matlab
6. Benchmarks and Conclusions
Comments on the Tx model in C

IBIS_AMI_API long AMI_Init(
double *impulse_matrix,
long row_size,
long aggressors,
double sample_interval,
double bit_time,
char *AMI_parameters_in,
char **AMI_parameters_out,
void **AMI_memory_handle,
char **msg ) {

AtmTxModel *self;
double samp_db1, norm, *tmp_db1;
int indx, yndx, item_count;
ParamListItem taps[4] = { { "-1", ami_double_type, 0 },
    { "0", ami_double_type, 0 },
    { "1", ami_double_type, 0 },
    { "2", ami_double_type, 0 } };
ParamListItem param[2] = { { "tap_filter", ami_list_type, 0 },
    { "tx_swing", ami_double_type, 0 } };
ParamList tap_list = { "tap_filter", 4, &taps[0] };
ParamList param_list = { "IBIS_AMI_Tx", 2, &param[0] };

//Default parameter values
taps[0].p_val.dbl_val = 0;
taps[1].p_val.dbl_val = 1;
taps[2].p_val.dbl_val = 0;
taps[3].p_val.dbl_val = 0;

Color code:
black: “normal” stuff
green: still able to handle it
red: too much for me (EE)
//Default parameter values
param[0].p_val.ptr_val = (void*)tap_list;
param[1].p_val.dbl_val = 0.8;

self = (AtmTxModel*)malloc( sizeof( AtmTxModel ) );
*AMI_memory_handle = (void*)self;

//Parse the parameter string
item_count = 0;
if( AMI_parameters_in ) { item_count = ParseTree( AMI_parameters_in, param_list );
}

self->msg = (char*)malloc( 68*sizeof( char ) );
memset( self->msg, 0, 68*sizeof( char ) );
sprintf( self->msg, "INFO: Configured %d items through parameter string.", item_count );
*msg = self->msg;

//Normalize the tap coefficients.
norm = fabs( taps[0].p_val.dbl_val ) +
    fabs( taps[1].p_val.dbl_val ) +
    fabs( taps[2].p_val.dbl_val ) +
    fabs( taps[3].p_val.dbl_val );
taps[0].p_val.dbl_val /= norm;
taps[1].p_val.dbl_val /= norm;
taps[2].p_val.dbl_val /= norm;
taps[3].p_val.dbl_val /= norm;
// Echo the parameters back out
*AMI_parameters_out = GrowTree( param_list );

self->taps[0] = taps[0].p_val.dbl_val;
self->swing = param[1].p_val.dbl_val;
self->row_size = row_size;
self->out_buf = 0;
self->buf_size = 0;
self->last_in = 0;
self->sample_interval = sample_interval;
self->bit_time = bit_time;
self->params_out = *AMI_parameters_out;

samp_db1 = bit_time/sample_interval - 0.5;
self->samples = 1;
while( self->samples < samp_db1 ) {
    self->samples++;
}

“round to the nearest” function...
Comments on the Tx model in C - cont’d

tmp_db1 = (double*)malloc( row_size*(aggressors+1)*sizeof( double ) );
for( yndx = 0; yndx < aggressors+1; yndx++ ) {
  for( indx = 0; indx < row_size; indx++ ) {
    tmp_db1[ indx+row_size*yndx ] =
      self->taps[0]*impulse_matrix[ indx+row_size*yndx ];
    if( indx >= self->samples ) {
      tmp_db1[ indx+row_size*yndx ] +=
        self->taps[1]*impulse_matrix[ indx+row_size*yndx-self->samples ];
    }
    if( indx >= 2*self->samples ) {
      tmp_db1[ indx+row_size*yndx ] +=
        self->taps[2]*impulse_matrix[ indx+row_size*yndx-2*self->samples ];
    }
    if( indx >= 3*self->samples ) {
      tmp_db1[ indx+row_size*yndx ] +=
        self->taps[3]*impulse_matrix[ indx+row_size*yndx-3*self->samples ];
    }
    tmp_db1[ indx+row_size*yndx ] *= self->swing;
  }
}
memcpy( impulse_matrix, tmp_db1, row_size*(aggressors+1)*sizeof( double ) );
free( tmp_db1 );

//Calculate the step response
self->step_response = (double*)malloc( row_size*sizeof( double ) );
self->step_response[0] = sample_interval * impulse_matrix[0];
for( indx = 1; indx < row_size; indx++ ) {
}
IBIS_AMI_API long AMI_GetWave( double *wave_in,
   long wave_size,
   double *clock_times,
   char **AMI_parameters_out,
   void *AMI_memory ) {

   int tmp_size, indx, yndx, clock_dx;
   double *tmp_dbl, step_size;
   AtmTxModel *self = (AtmTxModel*)AMI_memory;

   //Create an extended array to compute from.
   tmp_size = self->row_size + wave_size;
   if( self->buf_size < tmp_size ) {
      tmp_dbl = (double*)malloc( tmp_size*sizeof( double ) );
      memset( tmp_dbl, 0, tmp_size*sizeof( double ) );
      //Resize if necessary
      if( self->out_buf != 0 ) {
         memcpy( tmp_dbl, self->out_buf, self->buf_size*sizeof( double ) );
         free( self->out_buf );
      }
      self->out_buf = tmp_dbl;
      self->buf_size = tmp_size;
   } else {
      tmp_dbl = self->out_buf;
   }
}
// Compute the response.
clock_dx = 0;
for( indx = 0; indx < wave_size; indx++ ) {
    if( wave_in[ indx ] * self->last_in < 0 ){
        clock_times[ clock_dx++ ] = gw_time + (indx-0.5)*self->sample_interval;
    }
    if( wave_in[ indx ] != self->last_in ) { // Add step response
        step_size = wave_in[ indx ] - self->last_in;
        for( yndx = 0; yndx < self->row_size; yndx++ ) {
            tmp_dbl[ indx+yndx ] += step_size * self->step_response[ yndx ];
        }
        for( yndx = indx+self->row_size; yndx < self->buf_size; yndx++ ) {
            tmp_dbl[ yndx ] += step_size *
                self->step_response[ self->row_size-1 ];
        }
    }
    self->last_in = wave_in[ indx ];
    wave_in[ indx ] = tmp_dbl[ indx ]; // Save the output
}

// Terminate the list of clock ticks
clock_times[ clock_dx ] = -1;

using a step response as an input, this makes a convolution function
Comments on the Tx model in C - cont’d

//Terminate the list of clock ticks
clock_times[ clock_dx ] = -1;

//Save the remaining response for the next block of data
for( indx = 0; indx < self->row_size; indx++ ) {
    tmp_dbl[ indx ] = tmp_dbl[ indx+wave_size ];
}
for( ; indx < self->buf_size; indx++ ) {
    tmp_dbl[ indx ] = tmp_dbl[ self->row_size-1 ];
}

gw_time += wave_size*self->sample_interval;

return 1;
}

IBIS_AMI_API long AMI_Close( void *AMI_memory ) {
    AtmTxModel *self = (AtmTxModel*)AMI_memory;
    free( self->step_response );
    free( self->out_buf );
    free( self->params_out );
    free( self->msg );
    free( self ); //"The truth will set you free."
    return 1;
}
IBIS-AMI with Different Languages

IBIS Summit, DesignCon, February 2008

1. Overview: The IBIS-AMI BIRD and Toolkits
2. The language of IBIS-AMI models
3. Comments on the Tx model in C
4. The Tx model in VHDL-AMS
5. The Tx model in Matlab
6. Benchmarks and Conclusions
Other language candidates

- **VHDL-AMS and Verilog-AMS in IBIS**
  - multi-lingual extensions (section 6b) available starting IBIS v4.1 (since February 2004)

- **Matlab is quite popular for SERDES buffer and/or system development:**
  - StatEye (public domain) was written in Matlab
  - lots of other, home grown tools have been developed by major IC companies
### The Tx model in VHDL-AMS

```vhdl
impure function AMI_init (FileName : string := "");
    DoStepResponse : boolean := false;
    Debug : boolean := false)
return real_vector is
    constant RowSize : integer := FileRead(FileName, "row_size", 1024);
    constant Aggressors : integer := FileRead(FileName, "aggressors", 0);
    constant SampleInterval : real := FileRead(FileName, "sample_interval", 2.50E-11);
    constant BitTime : real := FileRead(FileName, "bit_time", 2.00E-10);
    constant TxSwing : real := FileRead(FileName, "tx_swing", 1.0);
    constant TapCoeff : real_vector := FileRead(FileName, "tap_filter", (0.0, 1.0, 0.0, 0.0));
    constant ImpulseMatrix : real_vector := FileRead(FileName, "Time,impulse(primary)", (1.0, 2.0, 3.0, 4.0));

    variable TapCoeffNorm : real_vector(TapCoeff'range) := TapCoeff;
    variable Norm : real := 0.0;
    variable Samples : integer := 1;
    variable Samp_dbl : real := 0.0;
    variable i : integer := 0;
    variable indx : integer := 0;
    variable yndx : integer := 0;
    variable ImpResponse : real_vector(ImpulseMatrix'range) := (others => 0.0);
    variable StepResponse : real_vector(ImpulseMatrix'range) := (others => 0.0);
```

---

The parameter tree parser was not implemented in the file I/O functions of this VHDL-AMS example.
begin

-- Normalize the tap coefficients
for i in TapCoeff'range loop
    Norm := Norm + abs(TapCoeff(i));
end loop;

-- Calculate samples per bit
Samp_dbl := BitTime / SampleInterval - 0.5;
while (real(Samples) < Samp_dbl) loop
    Samples := Samples + 1;
end loop;


The Tx model in VHDL-AMS - cont’d

-- Calculate equalized impulse response

for yndx in 0 to Aggressors loop
  for indx in 0 to RowSize-1 loop
    ImpResponse(indx+RowSize*yndx) := TapCoeffNorm(0)*ImpulseMatrix(indx+RowSize*yndx);
    if (indx >= Samples) then
      ImpResponse(indx+RowSize*yndx) := ImpResponse(indx+RowSize*yndx)
        + TapCoeffNorm(1)*ImpulseMatrix(indx+RowSize*yndx-Samples);
    end if;
    if (indx >= 2*Samples) then
      ImpResponse(indx+RowSize*yndx) := ImpResponse(indx+RowSize*yndx)
        + TapCoeffNorm(2)*ImpulseMatrix(indx+RowSize*yndx-2*Samples);
    end if;
    if (indx >= 3*Samples) then
      ImpResponse(indx+RowSize*yndx) := ImpResponse(indx+RowSize*yndx)
        + TapCoeffNorm(3)*ImpulseMatrix(indx+RowSize*yndx-3*Samples);
    end if;
    ImpResponse(indx+RowSize*yndx) := ImpResponse(indx+RowSize*yndx) * TxSwing;
  end loop;
end loop;
The Tx model in VHDL-AMS - cont’d

if (DoStepResponse = true) then

-- Calculate step response

StepResponse(0) := SampleInterval * ImpResponse(0);
for indx in 1 to RowSize-1 loop
    StepResponse(indx) := StepResponse(indx-1) + SampleInterval * ImpResponse(indx);
end loop;

-- Return step response
return StepResponse;
end if;

else

-- Return impulse response
return ImpResponse;
end if;

end function AMI_init;

Return either the Step Response or the Impulse Response
The Tx model in VHDL-AMS - cont’d

impure function AMI_getwave (IniFile : string := "";
    Debug : boolean := false) return real_vector is

  constant RowSize : integer := FileRead(IniFile, "row_size", 1024);
  constant RegisterLength : integer := FileRead(IniFile, "register_length", 7);
  constant BitTime : real := FileRead(IniFile, "bit_time", 2.00E-10);
  constant SampleInterval : real := FileRead(IniFile, "sample_interval", 2.50E-11);
  constant StopTime : real := FileRead(IniFile, "stop_time", 4.00E-08);
  constant WaveSize : integer := integer(floor(StopTime / SampleInterval));
  constant TempSize : integer := RowSize + WaveSize;

  -- constant WaveIn : real_vector := FileRead(WfmFile, "Time,wave_in", ( 1.0, 2.0, 3.0, 4.0));
  constant WaveIn : real_vector := PRBS(RegisterLength, BitTime, SampleInterval, StopTime);
  constant StepResponse : real_vector := AMI_init(IniFile, true, false);

  variable i : integer := 0;
  variable Clock_indx : integer := 0;
  variable indx : integer := 0;
  variable yndx : integer := 0;
  variable GwTime : real := 0.0;
  variable StepSize : real := 0.0;
  variable LastIn : real := 0.0;
  variable ClockTimes : real_vector(WaveIn'range) := (others => 0.0);
  variable ReturnVec : real_vector(0 to TempSize-1) := (others => 0.0);
  variable tmp_dbl : real_vector(ReturnVec'range) := (others => 0.0);

WaveIn can be read from a data file or generated by a PRBS function.
The Tx model in VHDL-AMS - cont’d

begin

-- Compute the response

for indx in WaveIn'range loop

-- Save the time of each edge of WaveIn in ClockTimes
if (WaveIn(indx) * LastIn < 0.0) then
    ClockTimes(Clock_indx) := GwTime + (real(indx)-0.5) * SampleInterval;
    Clock_indx := Clock_indx + 1;
end if;

if (WaveIn(indx) /= LastIn) then
    -- Add step response
    StepSize := WaveIn(indx) - LastIn;
    for indx in 0 to RowSize-1 loop
        ReturnVec(indx+indx) := ReturnVec(indx+indx) + StepSize * StepResponse(indx);
    end loop;
    for indx in indx+RowSize to TempSize-1 loop
        ReturnVec(indx) := ReturnVec(indx) + StepSize * StepResponse(RowSize-1);
    end loop;
end if;

    LastIn := WaveIn(indx);
end loop;

-- Terminate the list of clock ticks
ClockTimes(Clock_indx) := -1.0;

using a step response as an input, this makes a convolution function
-- Save the remaining response for the next block of data
for indx in 0 to RowSize-1 loop
    tmp_dbl(indx) := tmp_dbl(indx+WaveSize);
end loop;
for indx in RowSize-1 to tmp_dbl'right loop
    tmp_dbl(indx) := tmp_dbl(RowSize-1);
end loop;

GwTime := GwTime + real(WaveSize) * SampleInterval;
-----------------------------------------------
return ReturnVec;
---------------------------------------------------
end function AMI_getwave;
-----------------------------------------------
Impulse response from Tx AMI-init

Red: tap coefficients: 0 1 0 0
Blue: tap coefficients: -0.15 0.7 -0.125 -0.025
Step response from Tx AMI-init

Red: tap coefficients: 0 1 0 0
Blue: tap coefficients: -0.15 0.7 -0.125 -0.025
Rx pad waveform from Tx AMI-getwave

Red: tap coefficients: 0 1 0 0
Blue: tap coefficients: -0.15 0.7 -0.125 -0.025

Bit time: 200 ps, Sampling time: 25 ps
Register length: 7 (128 bits)
Simulation stop time: 40 ns (more than all bits)
**Rx pad eye diagram from Tx AMI-getwave**

<table>
<thead>
<tr>
<th>7.63n</th>
<th>7.64n</th>
<th>7.65n</th>
<th>7.66n</th>
<th>7.67n</th>
<th>7.68n</th>
<th>7.69n</th>
<th>7.70n</th>
<th>7.71n</th>
<th>7.72n</th>
<th>7.73n</th>
<th>7.74n</th>
<th>7.75n</th>
<th>7.76n</th>
<th>7.77n</th>
<th>7.78n</th>
<th>7.79n</th>
<th>7.80n</th>
<th>7.81n</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOP: WFM MN OEQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Tap coefficients: 0 1 0 0
- Bit time: 200 ps, Sampling time: 25 ps
- Register length: 15 (32,768 bits)
- Simulation stop time: 6.554 µs (all bits)
Rx pad eye diagram from Tx AMI-getwave

Tap coefficients: -0.15  0.7  -0.125  -0.025

Bit time: 200 ps, Sampling time: 25 ps
Register length: 15 (32,768 bits)
Simulation stop time: 6.554 µs (all bits)
Rx pad waveform from Tx AMI-getwave

Green: tap coefficients: 0 1 0 0

Yellow: tap coefficients: -0.15 0.7 -0.125 -0.025

Bit time: 200 ps
Register length: 22 (4,194,304 bits)

Simulation stop time: 20 µs (100,000 bits)
Rx pad eye diagram from Tx AMI-getwave

Green:
tap coefficients:
0 1 0 0

Bit time: 200 ps
Register length: 22
(4,194,304 bits)

Simulation stop time:
20 μs (100,000 bits)

Zoomed in close to the center of the eye
Rx pad eye diagram from Tx AMI-getwave

Green:
tap coefficients:
-0.15  0.7  -0.125  -0.025

Bit time:  200 ps
Register length:  22
(4,194,304 bits)

Simulation stop time:
20 μs  (100,000 bits)

Zoomed out to show the full eye diagram
IBIS-AMI with Different Languages

IBIS Summit, DesignCon, February 2008

1. Overview: The IBIS-AMI BIRD and Toolkits
2. The language of IBIS-AMI models
3. Comments on the Tx model in C
4. The Tx model in VHDL-AMS
5. The Tx model in Matlab
6. Benchmarks and Conclusions
function [ImpulseMatrix, StepResponse] = AMI_init(RowSize, ...  
    Aggressors, ...  
    SampleInterval, ...  
    BitTime, ...  
    TxSwing, ...  
    TapCoeff, ...  
    ImpulseMatrix)

%% Normalize the tap coefficients

for i = 1:1:size(TapCoeff,1)  
    Norm = Norm + abs(TapCoeff(i,2));
end
for i = 1:1:size(TapCoeff,2)  
    TapCoeffNorm(i,2) = TapCoeff(i,2) / Norm;
end
The Tx model in Matlab - cont’d

```matlab
%% Calculate samples per bit
Samp_dbl = BitTime / SampleInterval - 0.5;
while (Samples < Samp_dbl)
    Samples = Samples + 1;
end

%% Calculate equalized impulse response
for indx = 0:1:Aggressors
    for idx = 1:1:RowSize
        ImpResponse(idx+RowSize*yndx) = TapCoeffNorm(1,2)*ImpulseMatrix(idx+RowSize*yndx,2);
        if (indx > Samples)
            ImpResponse(idx+RowSize*yndx) = ImpResponse(idx+RowSize*yndx) ...
                + TapCoeffNorm(2,2)*ImpulseMatrix(idx+RowSize*yndx-Samples,2);
        end
        if (indx > 2*Samples)
            ImpResponse(idx+RowSize*yndx) = ImpResponse(idx+RowSize*yndx) ...
                + TapCoeffNorm(3,2)*ImpulseMatrix(idx+RowSize*yndx-2*Samples,2);
        end
        if (indx > 3*Samples)
            ImpResponse(idx+RowSize*yndx) = ImpResponse(idx+RowSize*yndx) ...
                + TapCoeffNorm(4,2)*ImpulseMatrix(idx+RowSize*yndx-3*Samples,2);
        end
        ImpResponse(idx+RowSize*yndx) = ImpResponse(idx+RowSize*yndx) * TxSwing;
    end
end
ImpulseMatrix(:,2) = ImpResponse(:,);
```

“round to the nearest” function...

4-tap FIR filter
The Tx model in Matlab - cont’d

%%--------------------------------------------------------------------------
%% Calculate step response
%%--------------------------------------------------------------------------
StepResponse(1) = SampleInterval * ImpResponse(1);
for indx = 2:1:RowSize
    StepResponse(indx) = StepResponse(indx-1) + SampleInterval * ImpResponse(indx);
end
%%============================================================================
%% End function AMI_init
%%============================================================================
The Tx model in Matlab - cont’d

function [ReturnVec] = AMI_getwave(RowSize, ...
    Aggressors, ...
    SampleInterval, ...
    BitTime, ...
    TxSwing, ...
    TapCoeff, ...
    ImpulseMatrix, ...
    StopTime, ...
    WaveIn)

WaveSize = floor(StopTime / SampleInterval);
TempSize = RowSize + WaveSize;

[dummy,StepResponse] = AMI_init(RowSize, ...
    Aggressors, ...
    SampleInterval, ...
    BitTime, ...
    TxSwing, ...
    TapCoeff, ...
    ImpulseMatrix);

Clock_indx = 1;
GwTime = 0.0;
StepSize = 0.0;
LastIn = 0.0;
ClockTimes = [];
ReturnVec = zeros(TempSize,1);
tmp_dbl = zeros(size(ReturnVec));
The Tx model in Matlab - cont’d

%% Compute the response

for indx = 1:1:WaveSize
    % Save the time of each edge of WaveIn in ClockTimes
    if (WaveIn(indx,2)*LastIn < 0.0)
        ClockTimes(Clock_indx) = GwTime + (indx-0.5) * SampleInterval;
        Clock_indx = Clock_indx + 1;
    end

    if (WaveIn(indx,2) ~= LastIn)
        StepSize = WaveIn(indx,2) - LastIn;
        for yndx = 0:1:RowSize-1
            ReturnVec(indx+yndx) = ReturnVec(indx+yndx) + StepSize * StepResponse(yndx+1);
        end
        for yndx = indx+RowSize:1:TempSize
            ReturnVec(yndx) = ReturnVec(yndx) + StepSize * StepResponse(RowSize);
        end
        LastIn = WaveIn(indx,2);
    end
end

%% Terminate the list of clock ticks
ClockTimes(Clock_indx) = -1.0;

%% Save the remaining response for the next block of data - not implemented
GwTime = GwTime + real(WaveSize) * SampleInterval;

% End function AMI_getwave

using a step response as an input, this makes a convolution function
Rx pad waveform from Tx AMI-getwave

Red: tap coefficients: 0 1 0 0
Blue: tap coefficients: -0.15 0.7 -0.125 -0.025

Bit time: 200 ps
Register length: 22 (4,194,304 bits)
Simulation stop time: 20 µs (100,000 bits)
function ImpulseMatrix = AMI_init(SampleInterval, ...
    BitTime, ...
    TxSwing, ...
    TapCoeff, ...
    ImpulseMatrix)

TapCoeffNorm = TapCoeff;
Norm         = 0.0;
Samples      = 1;
Aggressors   = max(0,size(ImpulseMatrix,2)-2);

% Normalize the tap coefficients
Norm = sum(abs(TapCoeff(:,2)));
TapCoeffNorm(:,2) = TapCoeff(:,2) / Norm;

% Calculate samples per bit
Samples = round(BitTime/SampleInterval);

% Calculate equalized impulse response
b = zeros(Samples,size(TapCoeffNorm,1));
b(1,:) = TapCoeffNorm(:,2)';
b = reshape(b,1,[]);
ImpulseMatrix(:,2:Aggressors+2) = TxSwing * filter(b,1,ImpulseMatrix(:,2:Aggressors+2));

% End function AMI_init

The “vector notation” can eliminate all or most FOR loops, executes faster and reads better

Built-in functions are more convenient, and execute faster

IBIS-AMI with Different Languages
The *reshape* and *filter* functions in Matlab

\[ A = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix} \]

\[ B = \text{reshape}(A,2,[]) \]

\[ B = \begin{bmatrix} 1 & 3 & 5 & 7 & 9 & 11 \\ 2 & 4 & 6 & 8 & 10 & 12 \end{bmatrix} \]

\[ Y(z) = \frac{b(1) + b(2)z^{-1} + \ldots + b(nb+1)z^{-nb}}{1 + a(2)z^{-1} + \ldots + a(na+1)z^{-na}} X(z) \]
**Tx AMI-getwave in Matlab - done properly**

```matlab
function ReturnVec = AMI_getwave(SampleInterval, ...
    BitTime, ...
    TxSwing, ...
    TapCoeff, ...
    ImpulseMatrix, ...
    StopTime, ...
    WaveIn)

WfmSize = floor(StopTime / SampleInterval);
ReturnSize = size(ImpulseMatrix,1) + WfmSize;
ReturnVec = zeros(ReturnSize,1);
ImpulseResponse = AMI_init(SampleInterval, ...
    BitTime, ...
    TxSwing, ...
    TapCoeff, ...
    ImpulseMatrix);

% Save the time of each edge of WaveIn crossing 0 volts
ClockTimes = (WaveIn(find(diff(sign(WaveIn(1:WfmSize,2)))) ,1) ... 
    + WaveIn(find(diff(sign(WaveIn(1:WfmSize,2))))+1,1) ) / 2;
% And terminate the list of clock ticks
ClockTimes(length(ClockTimes)+1) = -1.0;
% Compute the response
ReturnVec = SampleInterval * conv(WaveIn(:,2),ImpulseResponse(:,2));

%% End function AMI_getwave
```

The built-in convolution function “conv” reduced multiple FOR loops into this single line.
IBIS-AMI with Different Languages

IBIS Summit, DesignCon, February 2008

1. Overview: The IBIS-AMI BIRD and Toolkits
2. The language of IBIS-AMI models
3. Comments on the Tx model in C
4. The Tx model in VHDL-AMS
5. The Tx model in Matlab
6. Benchmarks and Conclusions
## Benchmarks

<table>
<thead>
<tr>
<th>Description</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRBS register length:</td>
<td>22</td>
</tr>
<tr>
<td>Sampling time:</td>
<td>25 ps</td>
</tr>
<tr>
<td>Bit time:</td>
<td>200 ps</td>
</tr>
<tr>
<td>Stop time:</td>
<td>20 μs</td>
</tr>
<tr>
<td>Original code in ANSI C:</td>
<td>10.0 sec</td>
</tr>
<tr>
<td>Matlab (properly coded):</td>
<td>3.5 sec</td>
</tr>
<tr>
<td>Matlab (C clone):</td>
<td>14:00.0 sec</td>
</tr>
<tr>
<td>VHDL-AMS (simulator #1):</td>
<td>23:33.0 sec</td>
</tr>
<tr>
<td>VHDL-AMS (simulator #2):</td>
<td>1:15:45.0 sec</td>
</tr>
</tbody>
</table>
Comments on the benchmarks

- The benchmarks were not obtained from rigorous coding practices
  - neither the original C code and its “clones” were optimized
  - the 2nd version of the Matlab code was developed to show the efficiency and power of the Matlab syntax

- No research has been done yet to find the reasons for the poor performance of the VHDL-AMS code
  - the code is implemented in digital functions, so the analog solver was not involved in their execution
  - much better performance was expected
  - VHDL-AMS tools compile the source code which could theoretically be as fast as compiled C code
Conclusions

- **ANSI C is the most inconvenient language, but fast**
  - nice function libraries would help a great deal
  - but most computer science “stuff” will remain in the code
    - malloc, free, pointer of pointers, etc…

- **VHDL-AMS somewhat better**
  - most computer science “stuff” is not needed
  - but could use nice function libraries
  - some language limitations exist
    - functions can return one item only
    - no control over argument passing into functions (byRef / byValue)
    - arrays can’t be resized, etc…

- **Matlab**
  - the most friendly and efficient language of these three
  - its execution speed is competitive with ANSI C
The source code will be available

The source code of the VHDL-AMS and Matlab examples will be posted on the IBIS web site shortly after the Summit

Have fun experimenting with them!