2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION JAPAN

Generic High-Level Synthesis Flow from MATLAB/Simulink Model

Petri Solanti, Siemens EDA, Munich, Germany
Shusaku Yamamoto, Siemens EDA, Tokyo, Japan

SIEMENS

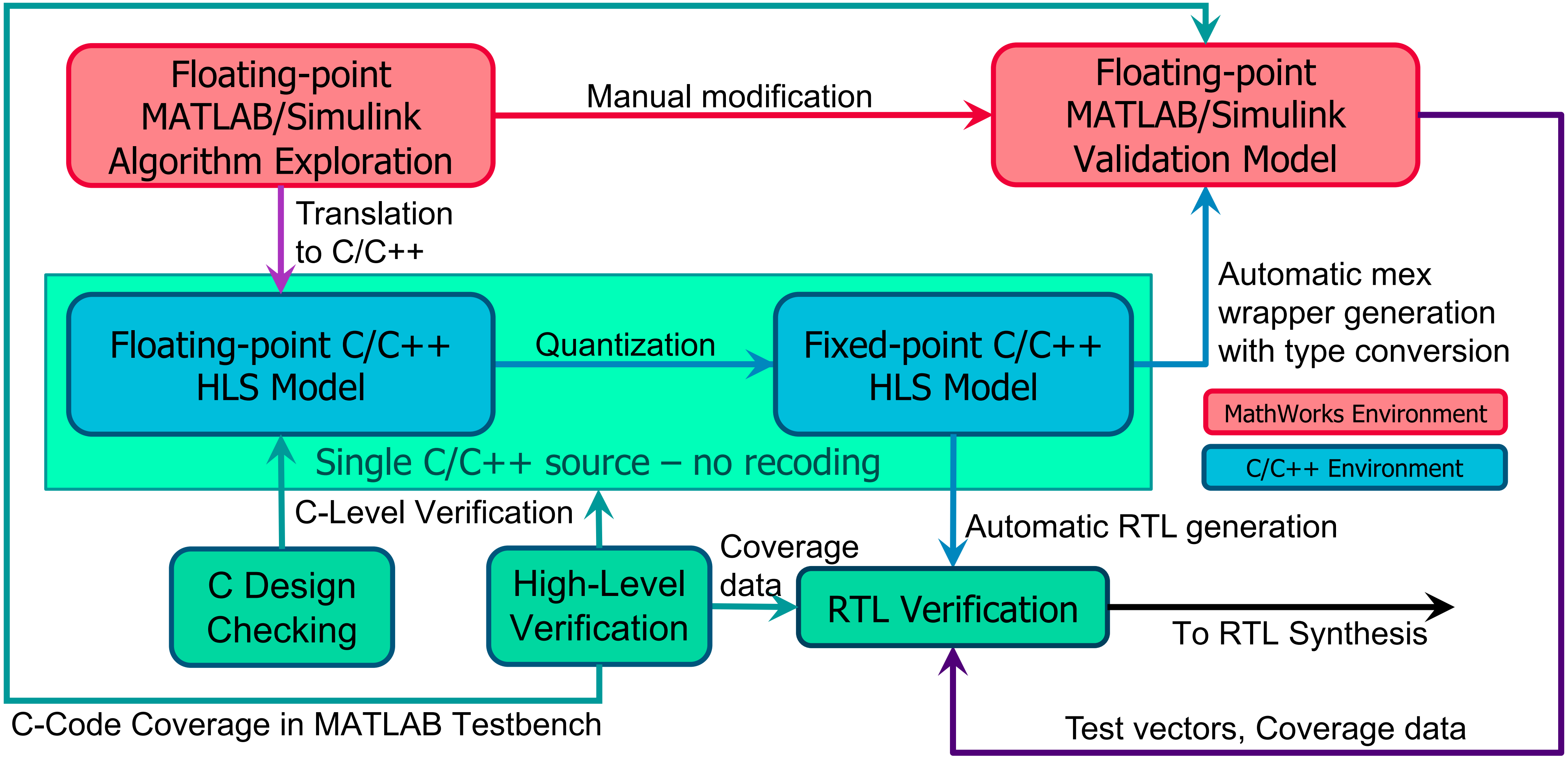
accellera SYSTEMS INITIATIVE

Outline

Introduction

- MATLAB and Simulink are the most popular Electronics-System-Level (ESL) tools for algorithm design
- Design teams are seeking for automated path from ESL to RTL
- Large abstraction level difference between MATLAB and RTL requires intermediate description to enable seamless tool flows
- C++/SystemC-based High-Level synthesis provides solid tool flow and design methodology
  - Automatic RTL code generation
  - Automated verification and validation flows

Integrated MATLAB/Simulink-to-HLS flow

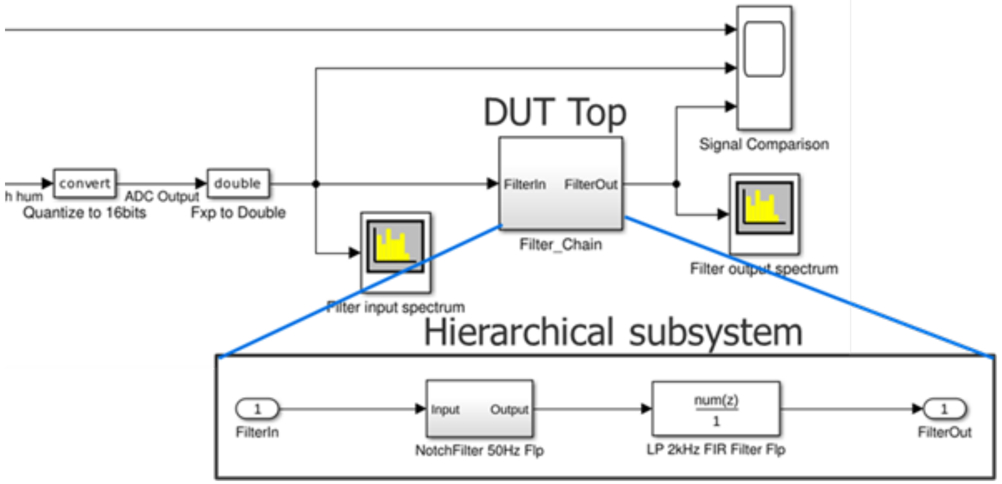


Fundamentals of model translation

- Starting point is floating-point ESL model
  - Minimum amount of code
  - No disturbing fixed-point effects
  - C++ type definitions in a separate include file
- Clear DUT communication interfaces
  - Data and control interfaces
  - Workspace variables used in the hierarchy must be converted to top-level ports
- Initial block hierarchy
  - Concurrent clocked processes
  - Data interfaces
  - Local storage memories

Analyzing block-level architecture

- Starting with MATLAB function hierarchy or Simulink block diagram
- Interface analysis
  - Data flows through the external I/O and internal connections
  - Configurable parameters (coefficients, configuration parameters, etc.)
- Independent clocked processes
  - Enables block-level concurrency
  - Alleviates block-level verification
- Re-usability of functions and blocks



Translating MATLAB scalar code to C++

- C++ class (recommended) or function implementation
- Mainly syntax conversion from MATLAB to C
  - Variable declarations and initialization
  - Array indexing
  - Loop syntax
- Replacing MATLAB toolbox functions with HLS functions
  - Lots of equivalent HLS functions, like ac_math and ac_dsp libraries
  - Manual implementation of missing functions
- Functional verification using floating-point or wide fixed-point data types

Translating MATLAB scalar code to C++

```
function out=iir_filter(inData,inGain,denGain,numGain)
   persistent SReg;

   if (isempty(SReg))
        SReg = zeros(1,2);
   end

   tmpFBAccu = 0.0;
   tmpFFAccu = 0.0;
   tmpFBDiff = 0.0;
   tmpInGainOut = inData * inGain;

   for i=2:-1:1
             tmpFBGainOut = SReg(i) * denGain(i);
             tmpFBAccu = tmpFBAccu + tmpFBGainOut;
             tmpFFGainOut = SReg(i) * numGain(i+1);
             tmpFFAccu = tmpFFAccu + tmpFFGainOut;
             tmpFBDiff = tmpInGainOut - tmpFBAccu;
   end
   SReg(2) = SReg(1);
   SReg(1) = tmpFBDiff;
   outData = tmpFBDiff + tmpFFAccu;
end
```

```
class iir_class { private: in_t Sreg[2];
public: iir_class() {for (int i=0; i<2; i++ ){Sreg[i]=0.0;}
 void iir_filter(ac_channel<in_t> &dataIn_ch, coeff_t inGain,
                 coeff_t denGain[2], coeff_t numGain[3],
                 ac_channel<out_t> &dataOut_ch  )
 {
    in_t inData, tmpInGainOut;
    out_rs_t outData;
    accu_t tmpFBAccu = 0.0;
    accu_t tmpFFAccu = 0.0;
    accu_t tmpFBDiff = 0.0;
    accu_t tmpFBGainOut, tmpFFGainOut;

    if (dataIn_ch.available(1)) {
        inData = dataIn_ch.read();
        tmpInGainOut = inData * inGain;
        IIR: for (int i=1; i>=0; i--) {
            tmpFBGainOut = SReg[i] * denGain[i];
            tmpFBAccu += tmpFBGainOut;
            tmpFFGainOut = SReg[i] * numGain[i+1];
            tmpFFAccu += tmpFFGainOut;
            tmpFBDiff = tmpInGainOut - tmpFBAccu;
            SReg[i] = (i==0) ? tmpFBDiff : SReg[i-1];  }
        outData = tmpFBDiff + tmpFFAccu;
        dataOut_ch.write(outData);    }
 }  // End void iir_filter
```

Translating MATLAB Matrix model to C++

- MATLAB allows multiple vector and matrix operations in a single statements → must be split into individual statements
  - C++ function implementation allows only one call in a statement
- Catapult matrix library contains C++ class equivalents to most MATLAB operator-based matrix and vector operations
  - Open source available soon in https://hlslibs.org/
- Intermediate data storage array sizes can be analyzed in MATLAB
  - Variable declarations in C++ required

Translating MATLAB Matrix model to C++

MATLAB

```
% PP=SS*NN*NN'*SS';
SSNN = SS * NN;
SSNN_NNtick = SSNN * NN';
PP = SSNN_NNtick * SS';
```

C++

```
private:
  vector_x_matrix_multiply_class<Tin,Tin,Taccu,Tin,MTX_ROWS,
     MTX_ROWS,MTX_COLS,MTX_COLS,false,false> Mult_10Cx10R8C;
  vector_x_matrix_multiply_class<Tin,Tin,Taccu,Tin,MTX_COLS,
     MTX_ROWS,MTX_COLS,MTX_ROWS,false,true> Mult_8Cx10R8C_T;
  vector_dot_product_class<Tin,Tin,Taccu,Tout,MTX_ROWS,false,
     true> dotProd_10x10;
…
// Implement Matlab statement PP=SS*NN*NN'*SS';
Mult_10Cx10R8C.Product(SS, NN, SSNN);
Mult_8Cx10R8C_T.Product(SSNN, NN, SSNN_NNT);
dotProd_10x10.Product(SSNN_NNT, SS, PP);
```

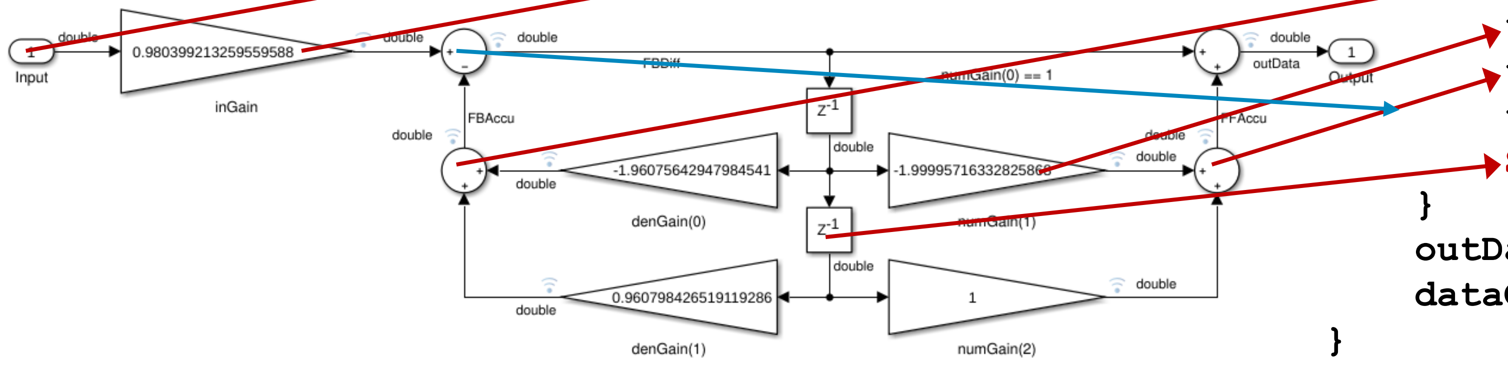
Translating Simulink design hierarchy

- Simulink block diagram consists of different modules
  - User defined subsystems
  - Library models
- User defined subsystems can be treated as hierarchical blocks
  - Define independent clocked process
  - Subsystem hierarchy defines HLS block hierarchy
- Simulink library components
  - Different complexities from simple mathematical function to complex mathematical processes
  - Simple functions can be mapped to HLS library functions
  - Complex processes may need an independent clocked process

Translating Simulink leaf-block to C++

- Simulink leaf-block contains usually primitive-level library components
  - Arithmetic operations
  - Delays (registers) or storage components (memories)
  - Simple DSP functions, e.g., filtes
  - Data flow connections between the operations
- Mapping Simulink components to operations
  - Starting from inputs moving towards output
  - Delay blocks mapped to static variables (registers)
  - Delay lines mapped from last to first

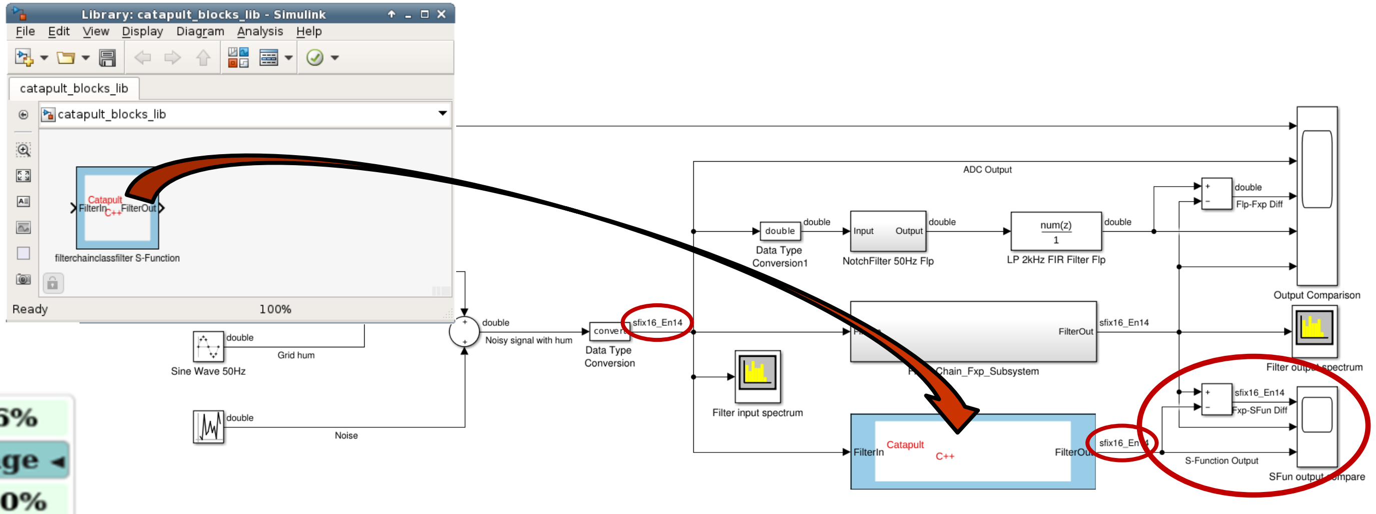
Translating Simulink leaf-block to C++




```
if (dataIn_ch.available(1)) {
   inData = dataIn_ch.read();
   tmpInGainOut = inData * inGain;
   IIR: for (int i=1; i>=0; i--) {
       tmpFBGainOut = SReg[i] * denGain[i];
       tmpFBAccu += tmpFBGainOut;
       tmpFFGainOut = SReg[i] * numGain[i+1];
       tmpFFAccu += tmpFFGainOut;
       tmpFBDiff = tmpInGainOut - tmpFBAccu;
       SReg[i] = (i==0) ? tmpFBDiff : SReg[i-1];
   }
   outData = tmpFBDiff + tmpFFAccu;
   dataOut_ch.write(outData);
}
```

Validation and Verification flow

- Automatic mex wrapper generation from HLS tool
  - Creates a mex function for MATLAB and S-Function block for Simulink
- Instantiating C++ DUT into MATLAB/Simulink testbench
- Functional verification using floating-point or wide fixed-point types
- Coverage analysis:
  - Instrumenting DUT
  - Simulate design
  - Analyze coverage data

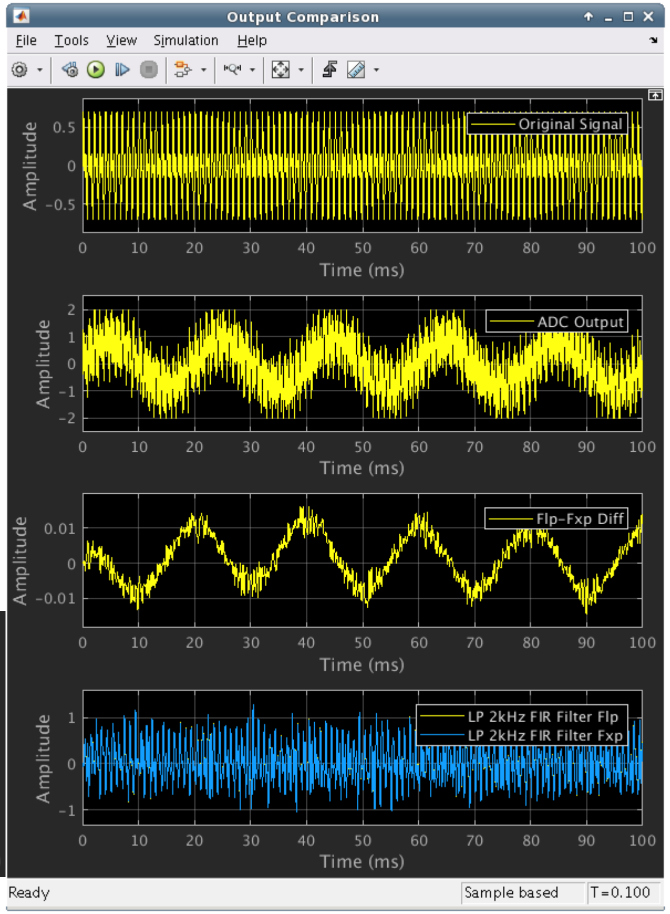


Local Instance Coverage Details:

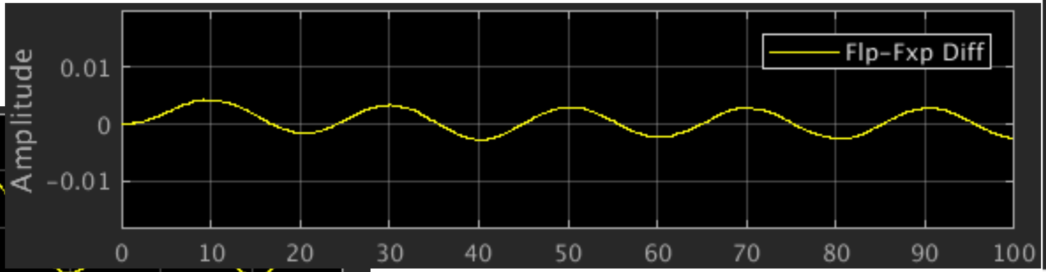
Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Total Coverage:					94.44%	91.66%
Statements	12	12	0	1	100.00%	100.00%
Branches	6	5	1	1	83.33%	83.33%

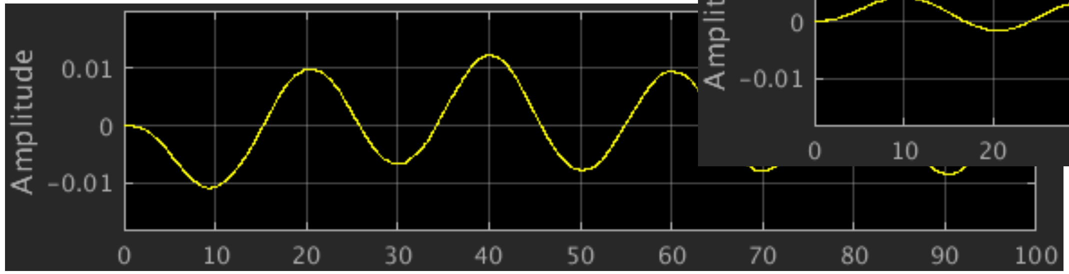
Quantization of translated model

- Analyze required bit widths using a fixed-point analysis tool
  - Value Range Analysis built in ac_fixed type class
  - MathWorks Fixed-point Designer
- Modify C++ type definitions
- Run simulation with original MATLAB testbench
- Analyze results



21-bit IIR coefficients





18-bit IIR coefficients

Conclusions

- HLS-based MATLAB-to-RTL design process is a good alternative to direct synthesis
  - C++ or SystemC as intermediate language moves HW specific design from MATLAB level to C++ → algorithm designers can use full power of MATLAB
  - HW related modifications are made to C++ model only
- Open source HLS libraries make manual translation easy
- HLS tools provide a thorough verification methodology from HLS to RTL level
- Functional and coverage analysis on C++ level with MATLAB testbench complete verification and validation flow from MATLAB to RTL

Thank you!