High-Level Synthesis

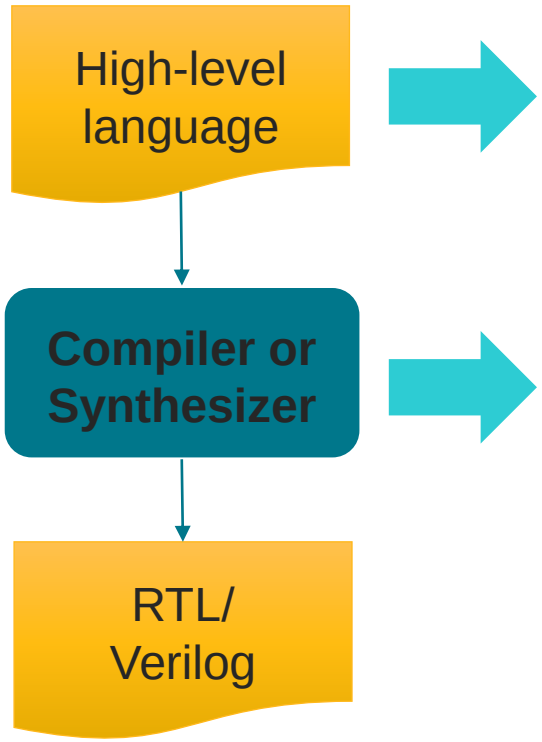
Edward Yeh (thyeh@cadence.com)
Sr. AE Manager

Dec. 2024

cadence®

What is High-Level Synthesis

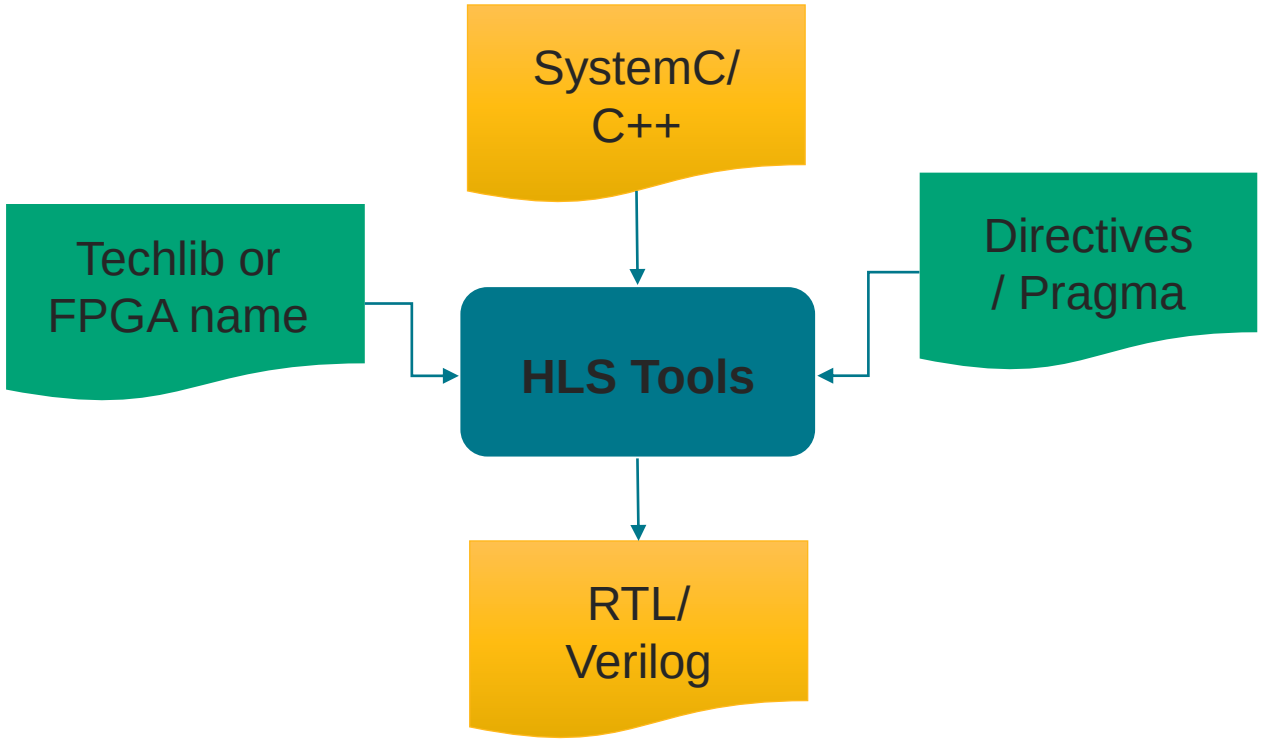
What is High-Level Synthesis (HLS)



- High level language (HLL) such as C++/SystemC/Python/Matlab
- Domain specific language
- More abstract HDL such as CHISEL, BSV (bluespec systemc verilog)

- Cadence Stratus HLS
- AMD Vitis/Vivado HLS
- Bluespec
- NEC CyberWorkBench
- MathWorks HDL Coder

Mainstream HLS Flow

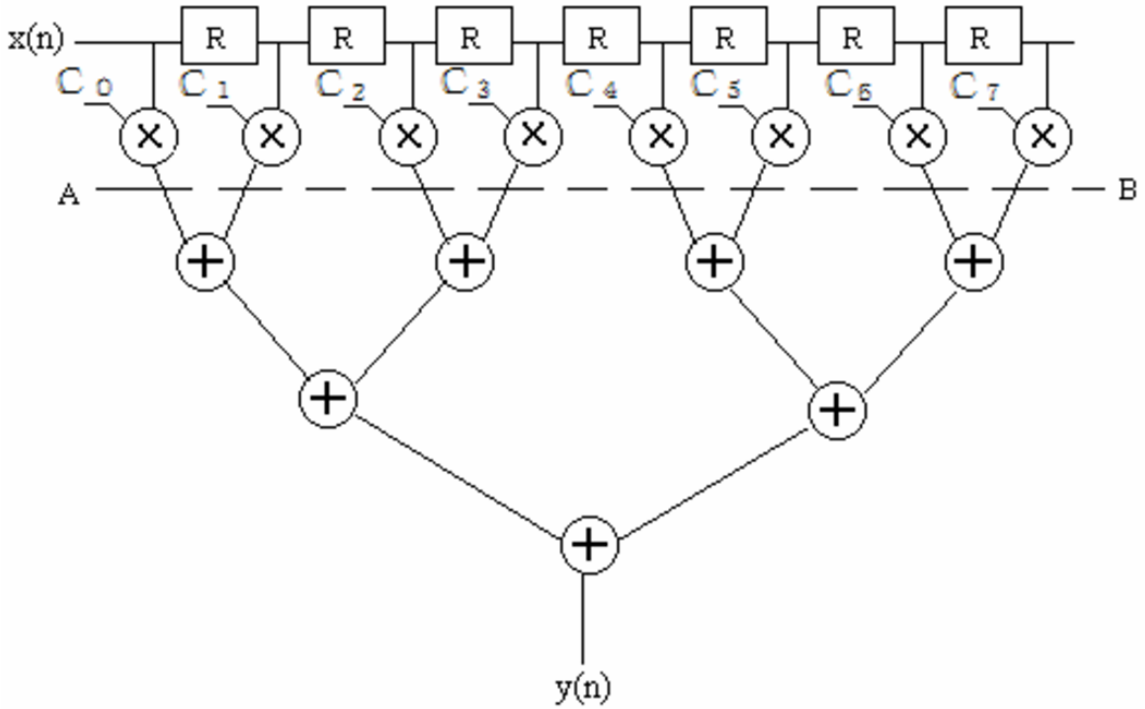


- Directives/Pragma:
  - Tell HLS tools the expected RTL micro-architectures
- Techlib:
  - Technology library
- FPGA name:
  - FPGA tools and FPGA parts

HLS Benefits

Improved Productivity

- Less Codes (implies less bugs)



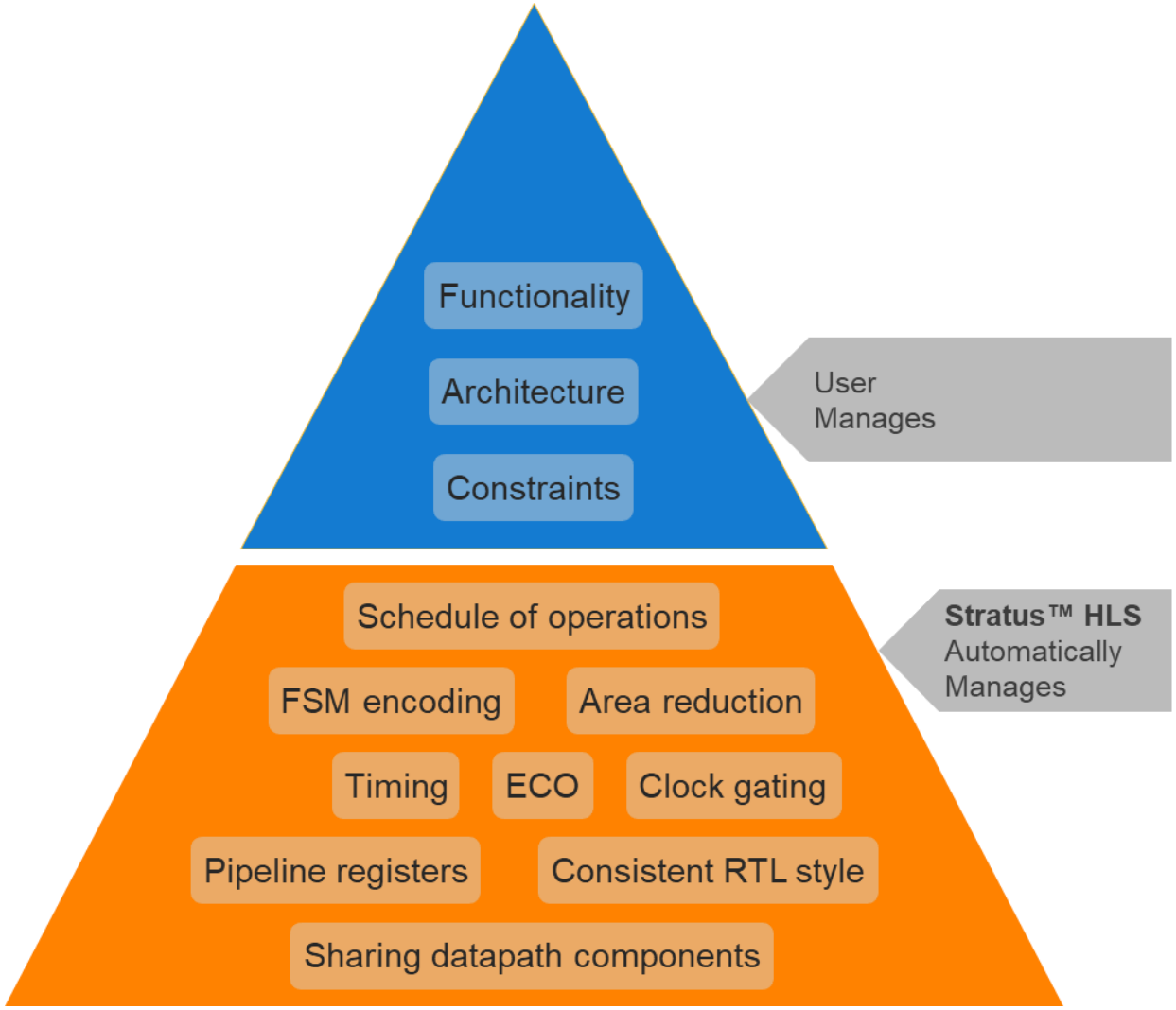
Source: https://www.researchgate.net/figure/A-direct-form-eight-tap-FIR-filter_fig1_221630163

SystemC/C++ codes

```
for (int i=7;i>0;i--) {
    shift_reg[i] = shift_reg[i-1];
}
shift_reg[0] = in;

for (int i=0;i<8;i++) {
     y = shift_reg[i] * coef_arr[i];
}
```

Improved Productivity



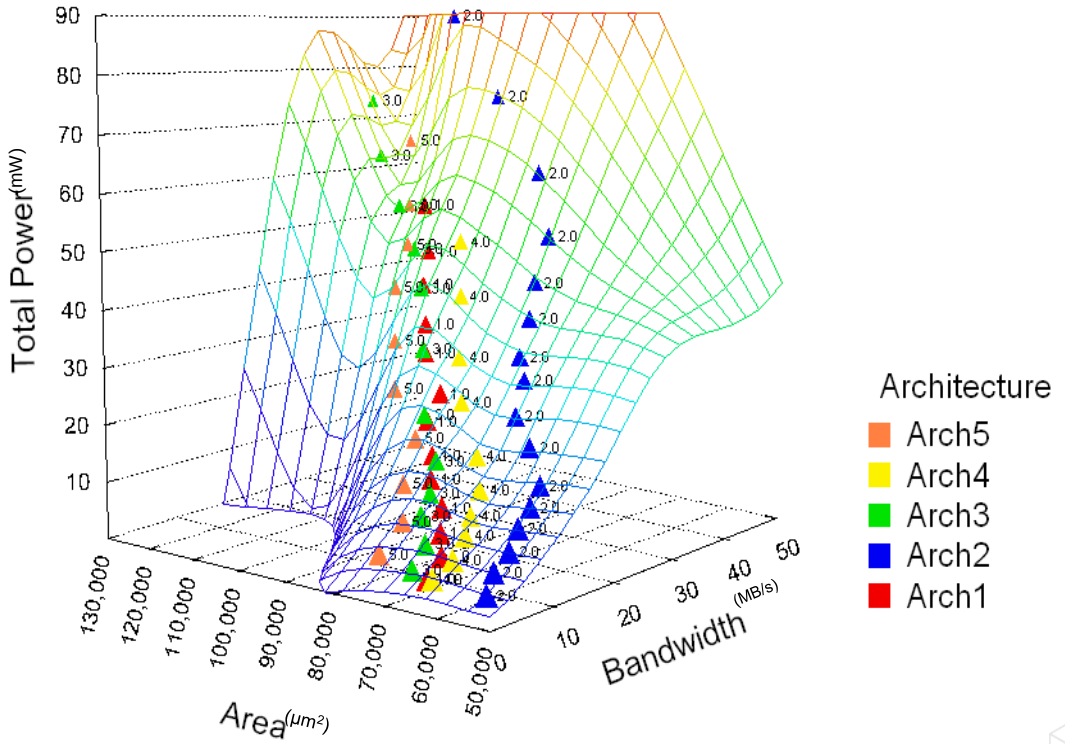
Design models focuses on “Functionality”, “Architecture”, “Constraints”

Improved Quality of Results by exploration

- Create multiple implementations from one behavioral IP description
  - Remove guesswork in picking “best” architecture
  - “Best” architecture varies for different applications
  - Explore to find optimal tradeoffs

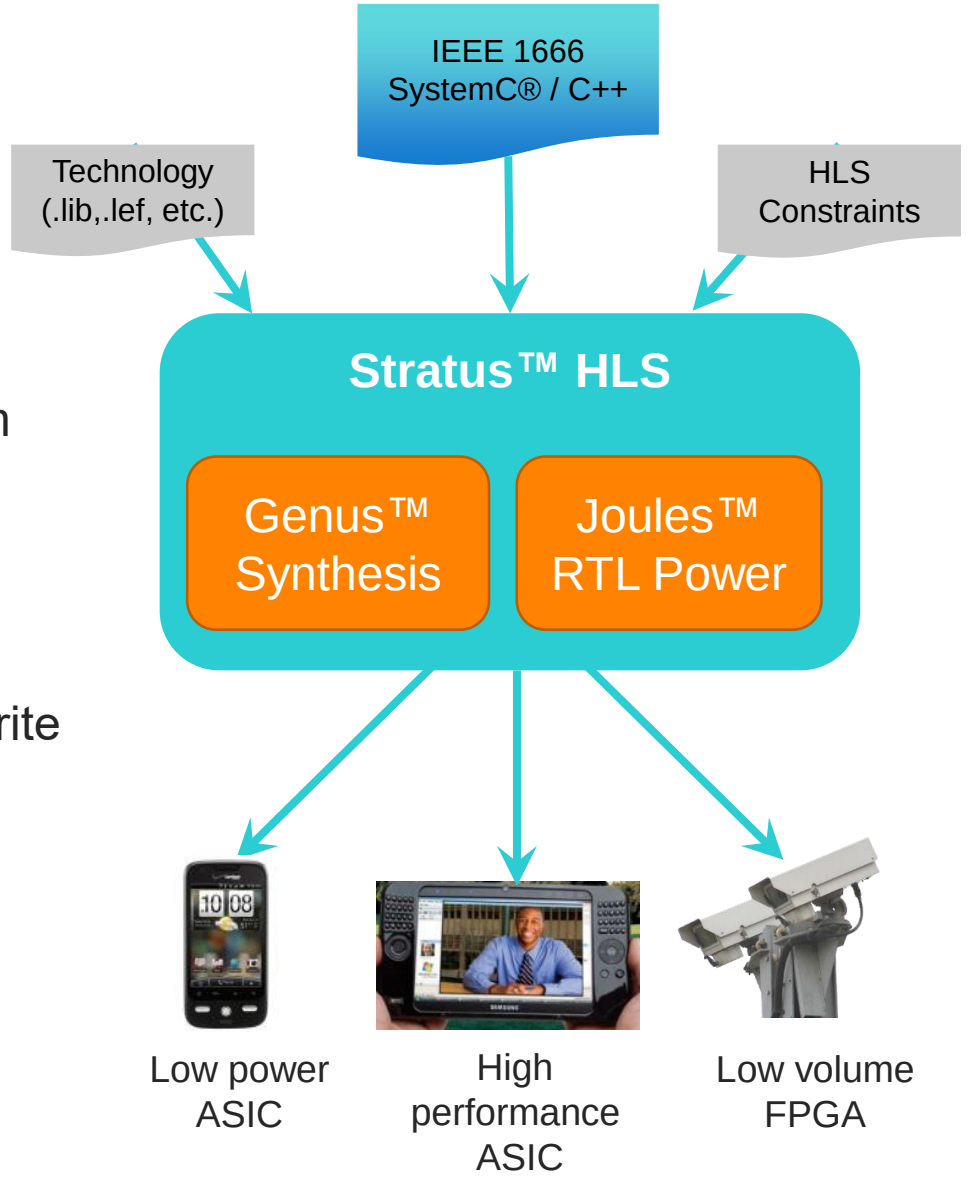
- Micro-architecture exploration
  - Automated by Stratus™ HLS
  - *Ex: Sharing vs. parallelism in the datapath*
  - *Ex: Changing perf. constraints or pipeline depth*
  - *Ex: Array storage as memory vs. register file vs. flops*

- Macro-architecture exploration
  - Assisted by Stratus HLS
  - *Ex: Changing I/O interfaces*
  - *Ex: Functional and algorithmic changes*



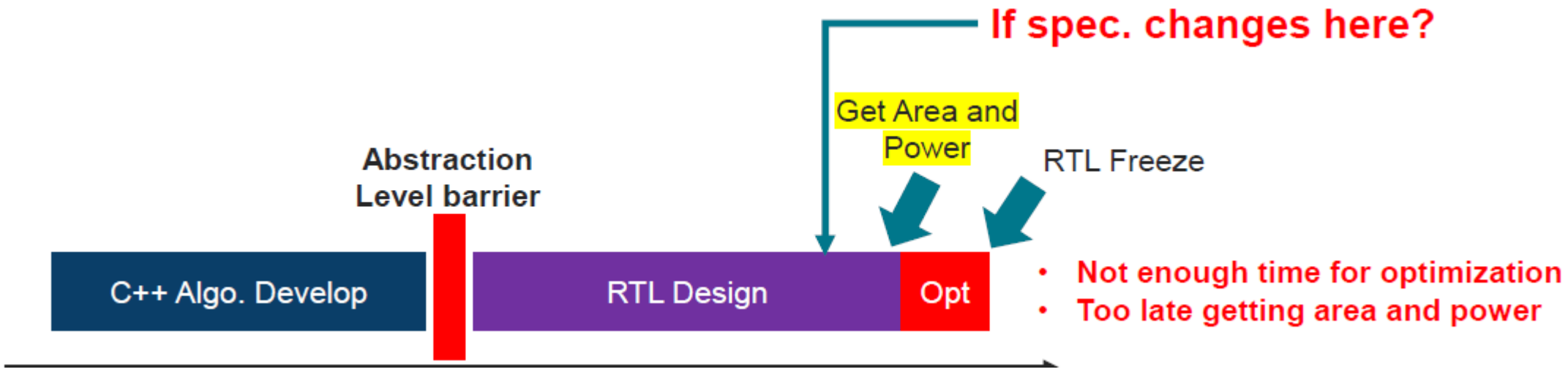
IP Reuse

- One IP model → different architectures and/or interfaces
  - C++ is golden source code
  - Directive files drive implementation for each application
- Behavioral IP is easy to change
  - “Minor” algorithmic changes require complete RTL rewrite
  - With Stratus™ HLS, reuse most of D&V environment
- Extend useful life of your IP



Stratus HLS improves ROI on your IP development

Algorithm to HW: RTL flow



- Abstraction barrier
  - RTL design starts when algorithm is complete
  - Need to understand C++ before Verilog coding
  - Long development time
- Changes in algorithm requires significant modification in RTL
- Need strong hardware expertise for good QoR
  - No time to attempt different micro-architecture implementations

Algorithm to HW: HLS flow

Before and After Stratus HLS flow

If spec. changes here?

Get Area and Power

RTL Freeze

Abstraction Level barrier

C++ Algo. Develop

RTL Design

Opt

- Not enough time for optimization
- Too late getting area and power

C++ Algo. Develop with Stratus HLS assist

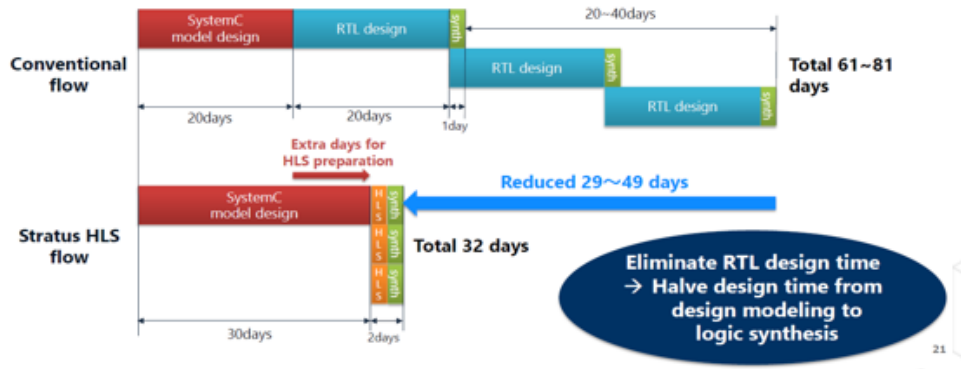
Talk in same language

RTL Freeze

Stratus HLS + Opt

Early trade-off bitwidth and performance
Re-use code in design

You can start design when algo is 80-85% complete

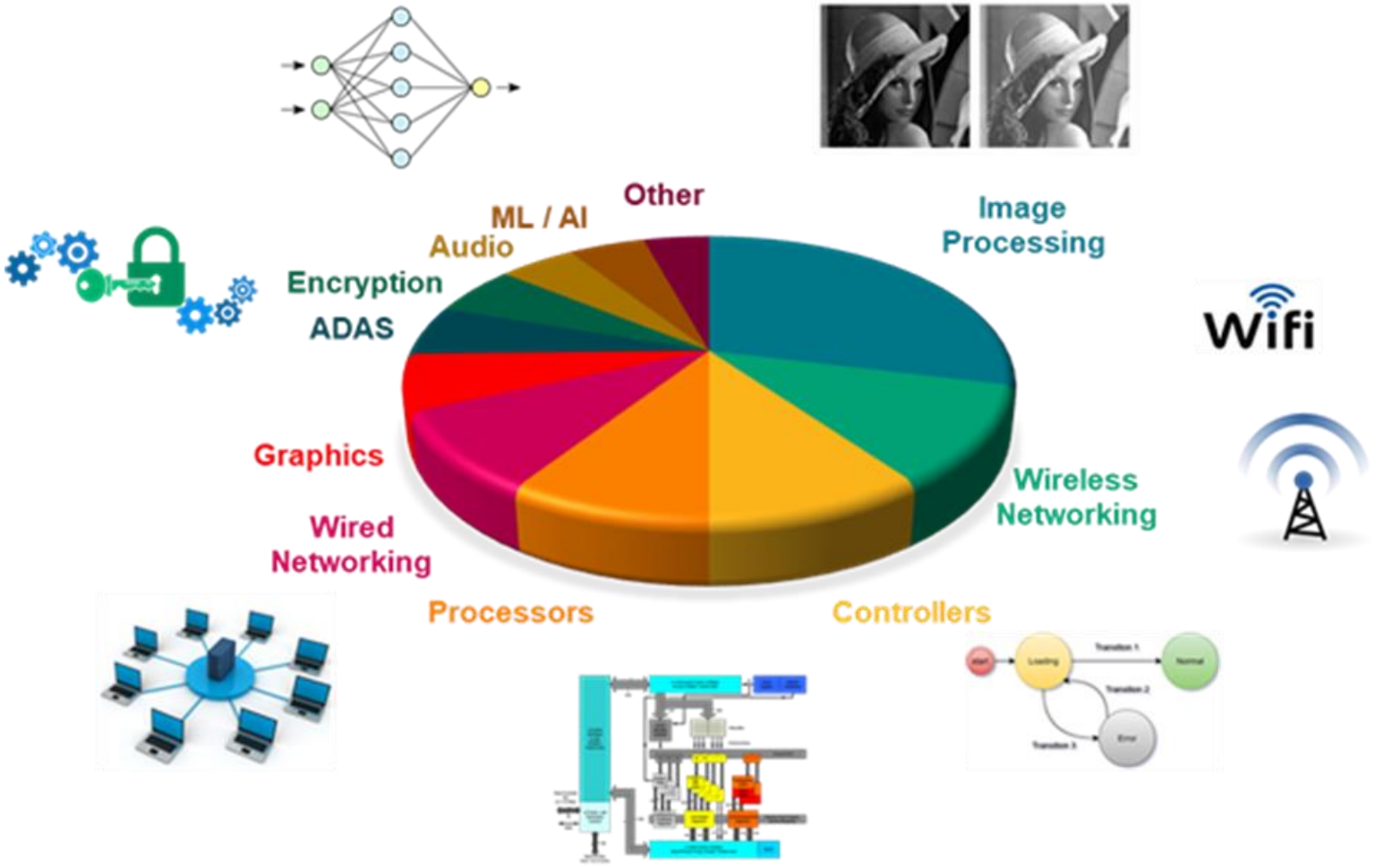


- 2X productivity
- The same patterns verifies design, faster simulation
- Get power and area during design stage

cadence

Who adopts HLS for production

Stratus HLS Production Applications



Stratus™ HLS Production Applications
Source: Cadence HLS industry survey, 2018

Industry adoption

Company
Qualcomm
Broadcom
Nvidia
AMD
MediaTek
Marvell
Realtek

Source: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/resources/

- **6 of top-7 IC design houses employ Stratus HLS**
- **Intel, Samsung, lots of customers in Japan, China**

User experiences from Industry - 1

- Design: 802.11ah baseband
  - The same baseband functionality with different performance
    - **i.e. RTL will be different**

- HLS achieves
  - Different interface by #define due to Radio is typically specified by customer
  - Single code-base with different implementations for different performance requirements
    - C++ template
  - Ability to handle new requirement due to spec. change
    - 2 months to make change
    - Hand RTL flow is infeasible to achieve this

User experiences from Industry - 2

- From Top CPU company
  - The design teams who are happiest report :

    "The HLS flow got us to meet our milestone X times faster than we estimate with our hand-written RTL approach"

    (Where the average value of X is 2)

    "The design couldn't be done using conventional methods, HLS was the only path that gets us there".

  - Higher-level language (usually C++/SystemC) is used, instead of Verilog/RTL
    - Source code is focused on describing **functionality** (architecture)
    - The source is usually written by a designer or an architect.
      - In some cases it's possible to re-use existing C++/SystemC code.
    - The compact SystemC code (average reduction of ~70%) is verified with 10X-1000X reduction in simulation time.

User experiences from Industry - 3

- From Top IC design house

- IP team's experience using HLS has been excellent
  - Small sized team able to meet **aggressive SoC deadlines** on multiple chips
  - Ability to accommodate **late enhancements** in the algorithm design and be code complete with verification ready in a matter of days.
  - No need for ECO with thorough verification in SystemC environment and RTL DV environment
  - **Design reuse** from moving target technology nodes achieved with 0 lines of code change
  - Bulk of the IP design is bug-free in RTL verification environment since the RTL is auto-generated, SystemC model is thoroughly tested in SystemC verification and 100% SystemC code coverage achieved. This results in **improved design and DV** team productivity.

Enter High-Level Synthesis World

cadence®

HLS Fundamental

Typical design steps of High-level synthesis

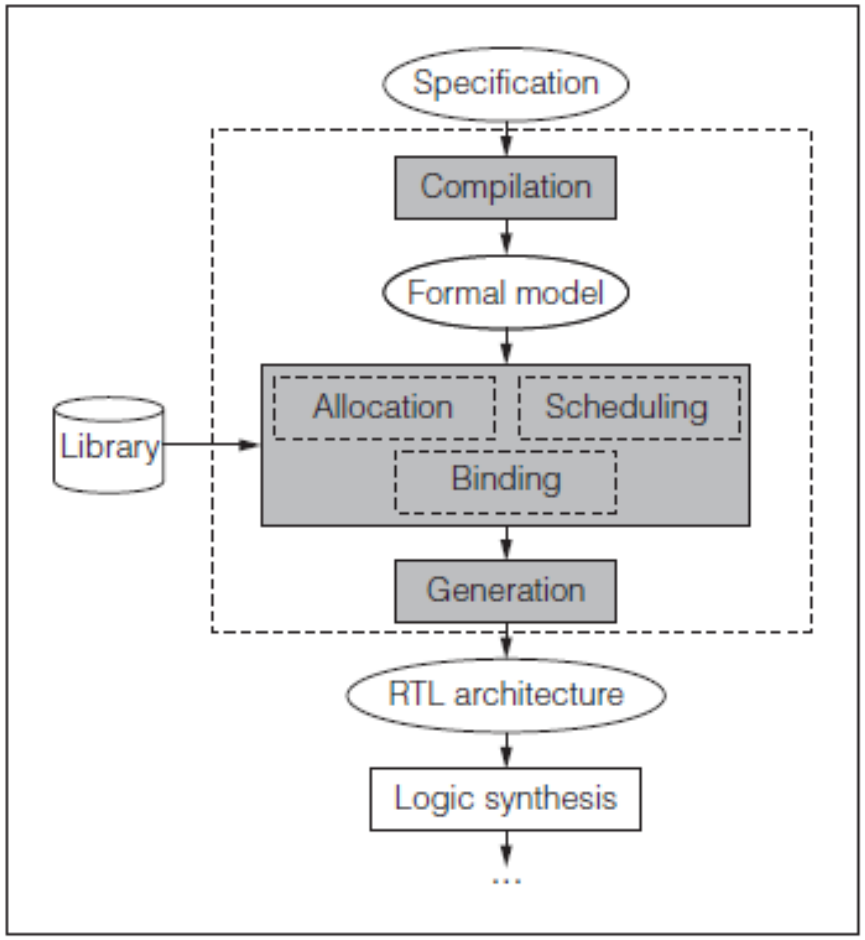


Figure 1. High-level synthesis (HLS) design steps.

Source: An Introduction to High-Level Synthesis, IEEE Design and Test of Computers, 2009.

Specification:

C++/SystemC with directives or pragma to set **micro-architectures**

Compilation:

Compiler-level optimization

Formal model:

CDFG, Control Data Flow Graph

Library:

Pre-characterization or on-the-fly characterization for functional unit (FU)/register

Allocation:

Allocation of initial FUs/registers

Post-allocation due to scheduling & binding

Scheduling:

The cycle to which operations will be assigned

Binding:

The FUs to which operations will be assigned to

The registers to which variables will be assigned to

Micro-architectures

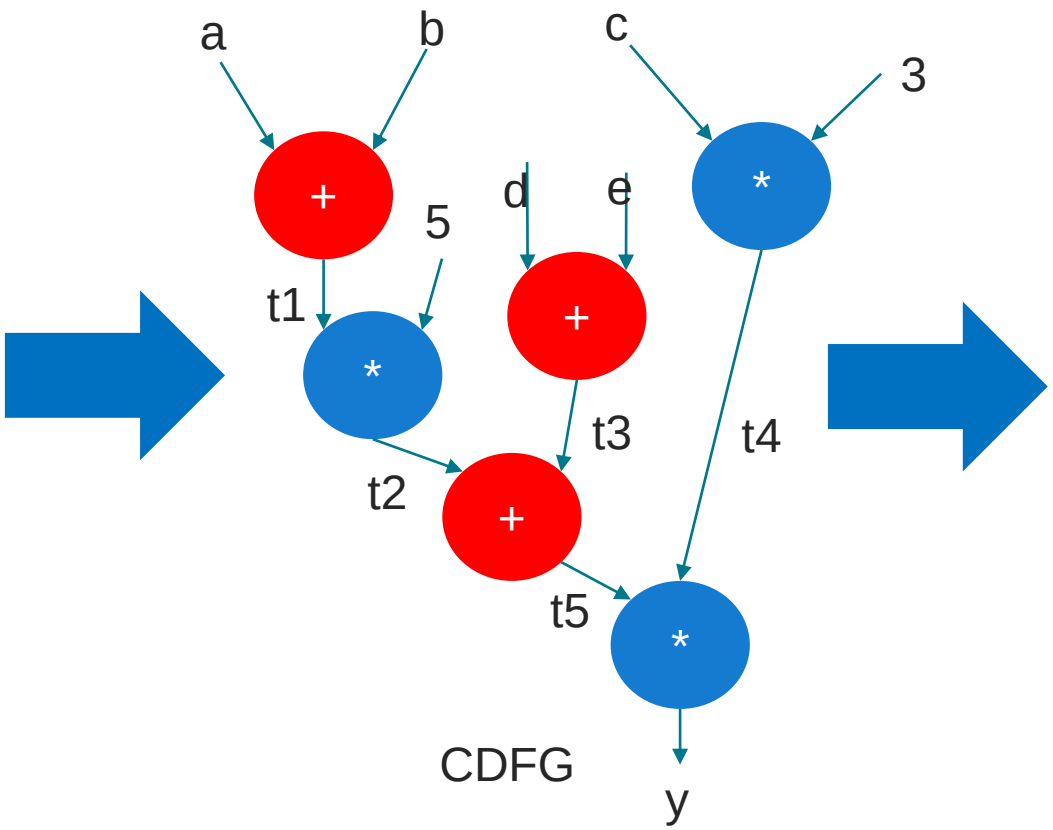
- **HLS tool does not know the micro-architectures a user want to implement.**
  - An array is a memory or a set of registers?
    - Sometimes code change is required if an array is mapped to memory
  - Pipelined design or non-pipelined design?
  - A for-loop is unrolled or not?
  - Handshaking with other modules?

- **The user should assign those by directives/pragmas to guide HLS tools**

Example

```
t1 = a + b;
t2 = t1 * 5;
t3 = d + e;
t4 = c * 3;
t5 = t2 + t3;
y = t4 * t5;
```

C++ Codes



CDFG

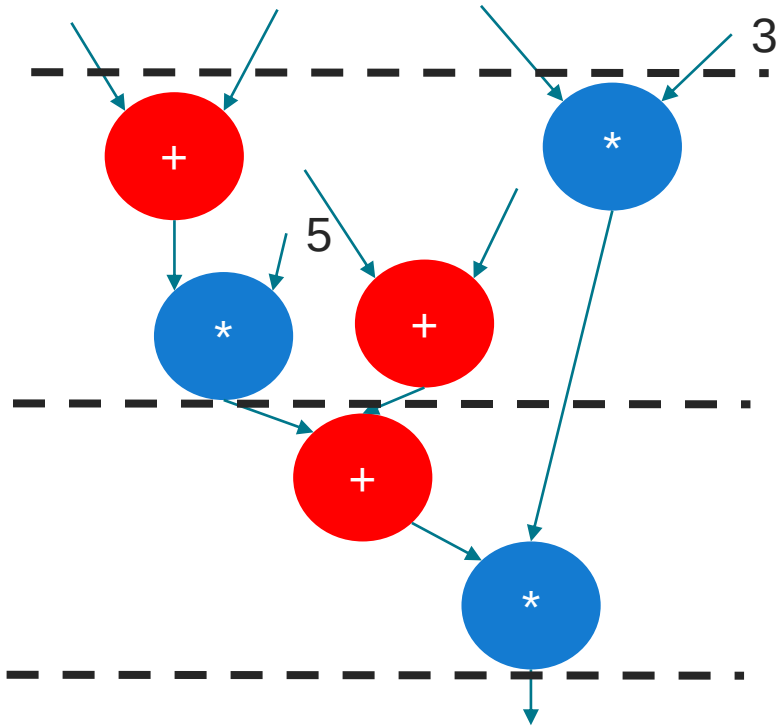
3 adders
3 multipliers

Initial Allocation

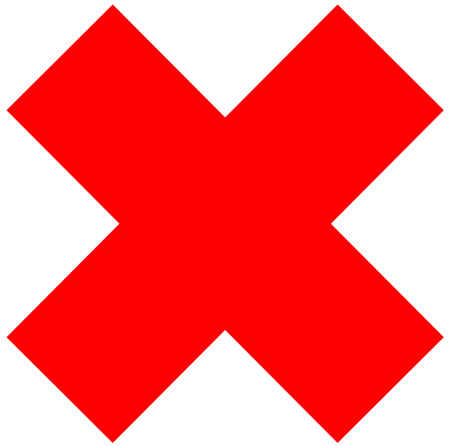
Example

- Assume the delay of multiplier is 3ns, and the delay of adder is 2ns.

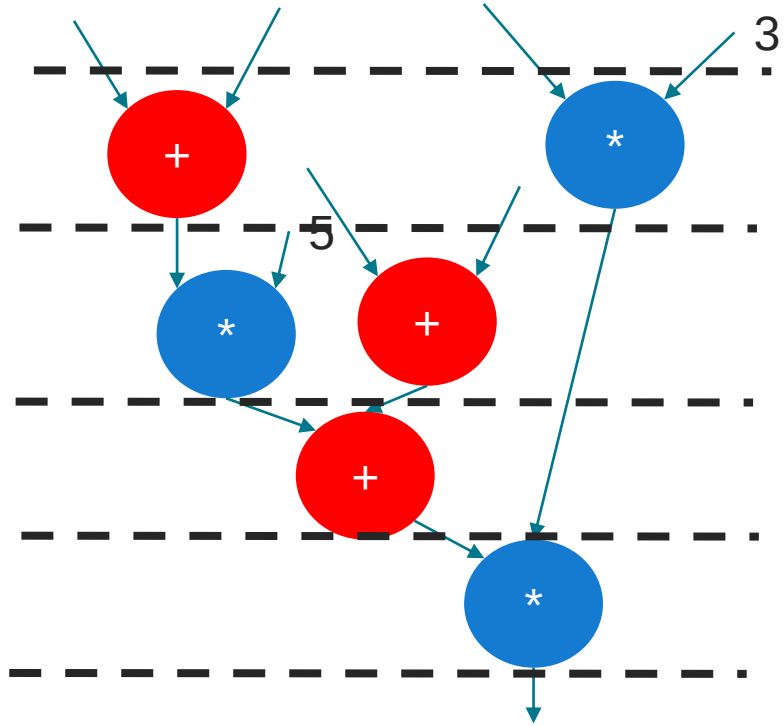
Clock period: 10ns, latency constraint:2



Clock period: 3ns latency constraint: 3

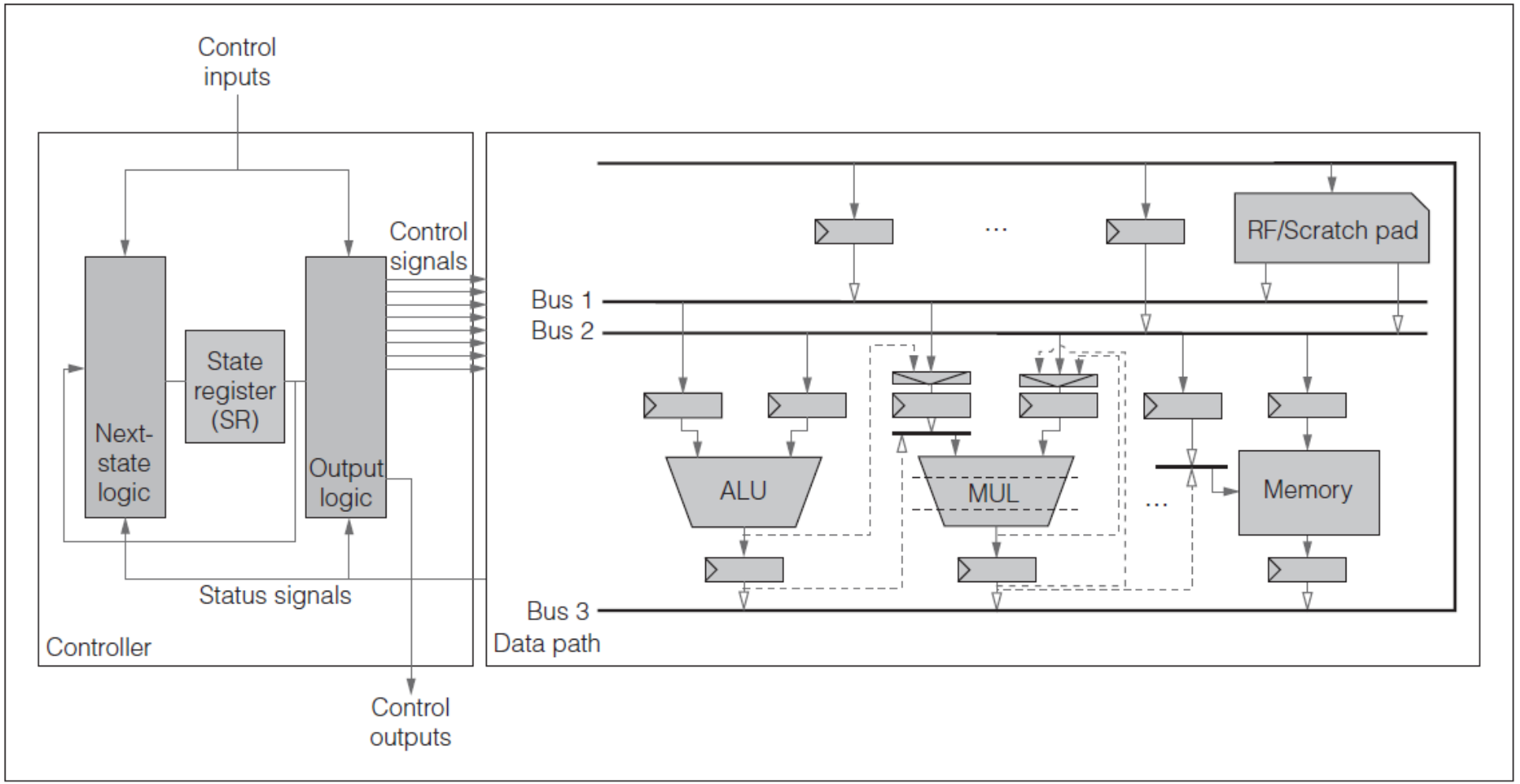


Clock period: 3ns latency constraint: 4



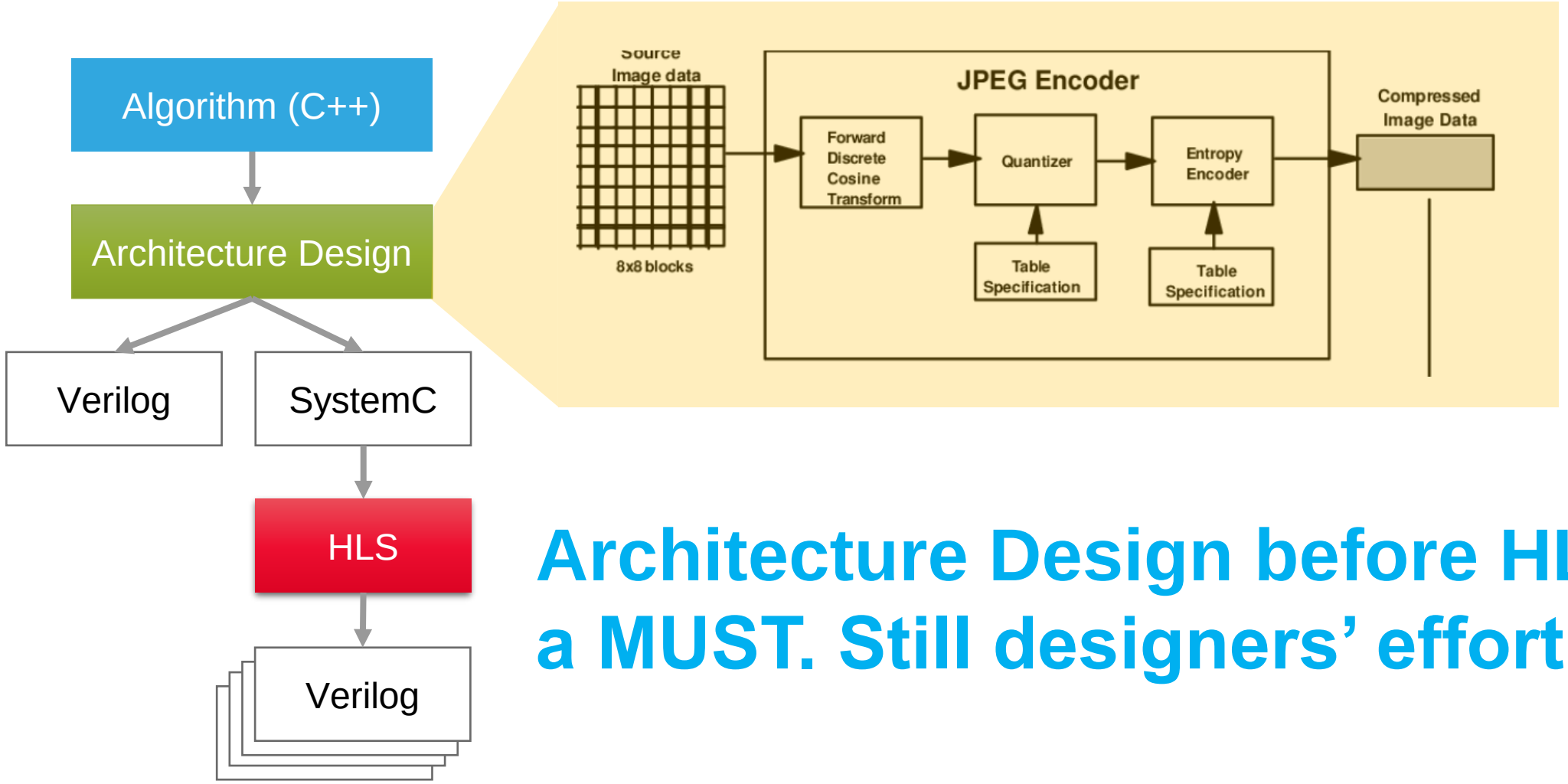
Scheduling

Typical RTL architecture



Source: An Introduction to High-Level Synthesis, IEEE Design and Test of Computers, 2009.

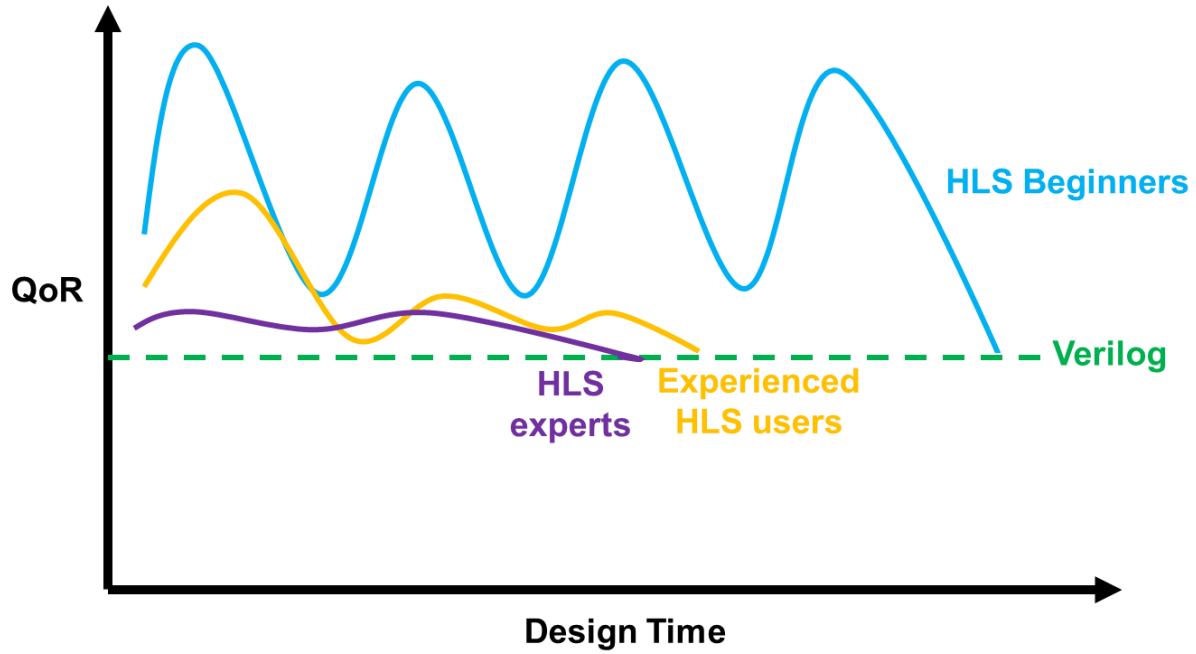
Super Important: HLS is not magic



Architecture Design before HLS is a MUST. Still designers' effort.

Super Important: HLS is not magic

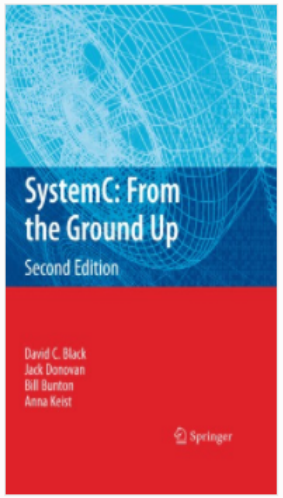
- **Current HLS technology achieves competitive QoR to hand RTL, but please be aware of**
  - **HLS tool is not as mature as logic synthesis tool**
  - **The combinations of coding and directives/pragmas affect QoR**
  - **Understand you will have growing pain due to learning curve**



Synthesizable SystemC

SystemC

- SystemC is a C++ library for HW descriptions (i.e. an HDL)
  - Virtual prototyping: System/sub-system/IP modeling
- IEEE 1666-2011
  - SystemC + TLM2.0 (Transaction-Level Modeling Standard, v2)



© 2010

SystemC: From the Ground Up

Authors (view affiliations)

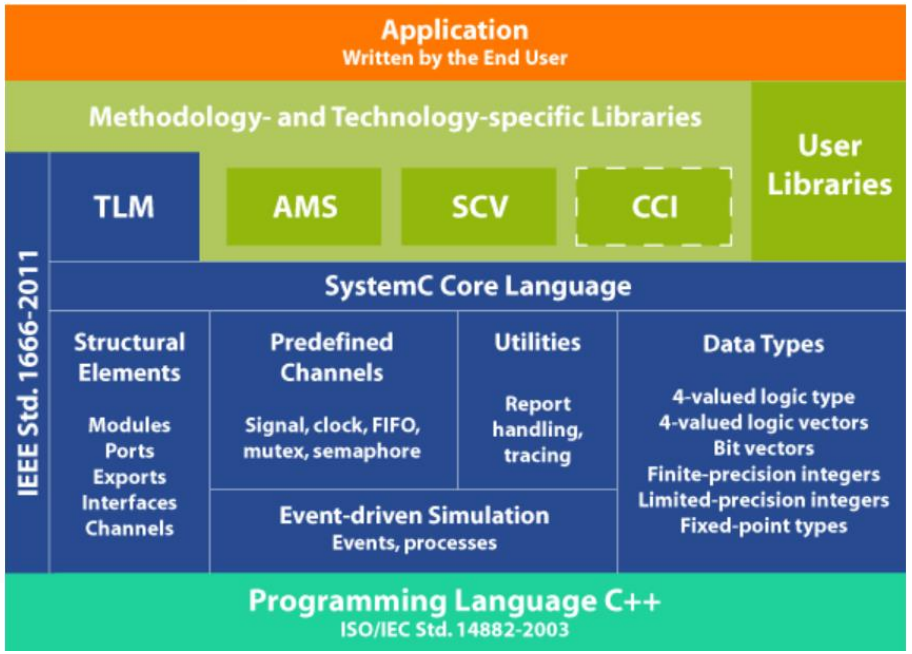
David C. Black, Jack Donovan, Bill Bunton, Anna Keist

Emphasizes TLM 2.0 and the recently adopted IEEE 1666

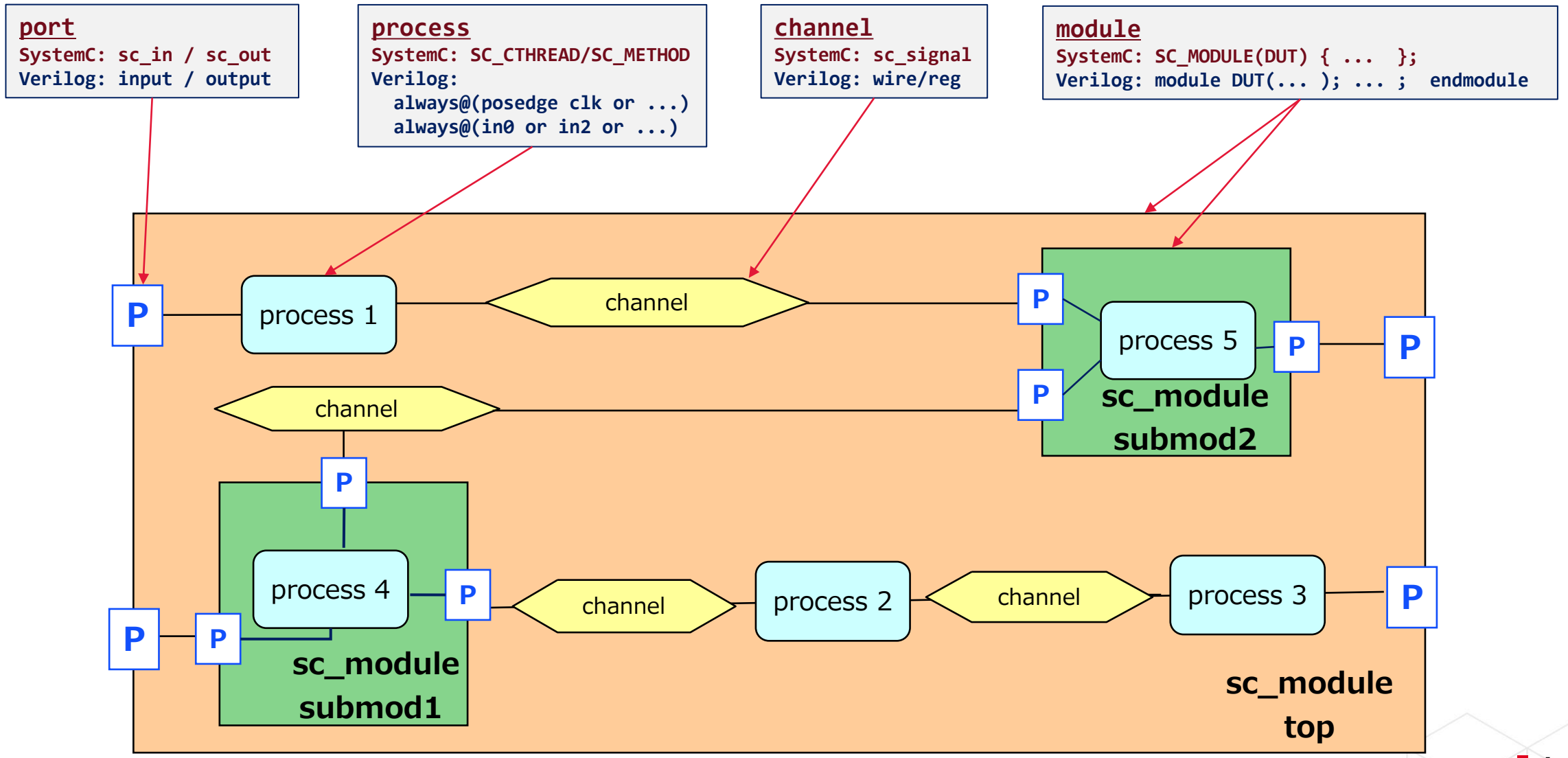
Includes SystemC Verification Library as well as an appendix that features a C++ primer for SystemC

Expands and updates material throughout the book

Includes supplementary material: sn.pub/extras



Main Components of Synthesizable SystemC

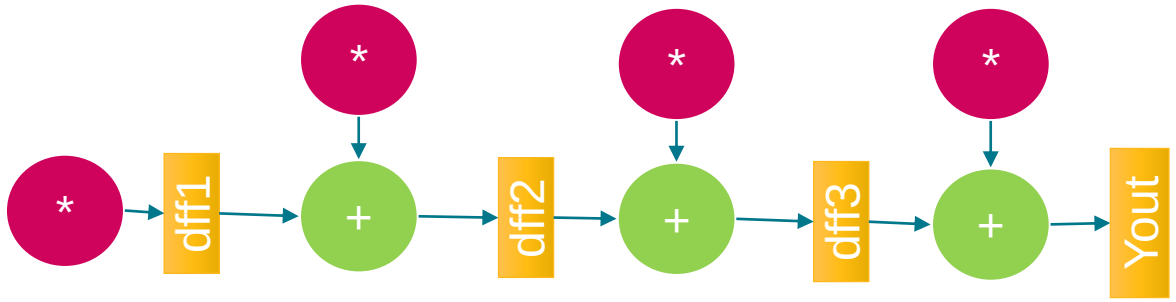


Different concepts to model HW in SystemC

- Cycle-accurate view (Verilog)

```
assign MCM3 = H3*Xin;
DFF dff1(.D(MCM3), .Q(Q1));
assign MCM2 = H2* Xin;
assign add_out1 = Q1 + MCM2;
DFF dff2(.D(add_out1), .Q(Q2));
assign MCM1 = H1*Xin;
assign add_out2 = Q2 + MCM1;
DFF dff3(.D(add_out2), .Q(Q3));
MCM0 = H0*Xin;
add_out3 = Q3 + MCM0;
always@(posedge)
    Yout <= add_out3;
```

Verilog



Different concepts to model HW in SystemC

- Iteration view (SystemC)

```
while (1) {
    MCM3 = H3*Xin;
    Q1 = MCM3;
    MCM2 = H2* Xin;
    add_out1 = Q1 + MCM2;
    Q2 = add1_out1;
    MCM1 = H1*Xin;
    add_out2 = Q2 + MCM1;
    Q3 = add_out2;
    MCM0 = H0*Xin;
    add_out3 = Q3 + MCM0;
    Yout = add_out3;
}
```

SystemC

=

```
while (1) {
    Q1 = H3*Xin;
    MCM2 = H2* Xin;
    Q2 = Q1 + MCM2;
    MCM1 = H1*Xin;
    Q3 = Q2 + MCM1;
    MCM0 = H0*Xin;
    Yout = Q3 + MCM0;
}
```

SystemC

Typical Synthesizable SystemC Module

Header file : DUT.h

```
#ifndef DUT_H
#define DUT_H

#include <systemc.h>
#include <stratus_hls.h>

SC_MODULE(DUT) {
  sc_in <bool>         CLOCK;
  sc_in <bool>         RESET_n;
  sc_in <sc_uint<8> > IN_A;
  sc_in <sc_uint<8> > IN_B;
  sc_out<sc_uint<9> > OUT;

  void proc();

  SC_CTOR(DUT)
    : CLK("CLK")
    , RSTn("RSTn")
    , IN_A("IN_A")
    , IN_B("IN_B")
    , OUT("OUT")
    {
      SC_CTHREAD(proc, CLOCK.pos());
      async_reset_signal_is(RESET_n, false);
    }
};

#endif
```

Implementation file : DUT.cpp

```
#include "DUT.h"

void DUT::proc() {
  sc_uint<8> temp1;
  sc_uint<8> temp2;
  sc_uint<9> sum;
  {HLS_DEFINE_PROTOCOL("reset");
  OUT.write(0);
  wait();
  }
  while(true) {
    {HLS_DEFINE_PROTOCOL("input");
      temp1 = IN_A.read();
      temp2 = IN_B.read();
    }

    sum = temp1 + temp2;

    {HLS_DEFINE_PROTOCOL("output");
      OUT.write(sum);
      wait();
    }
  }
}
```

Please consider HW when doing SystemC modeling

```
#ifdef CODE1
        for (int i=0;i<4;i++) {
             shift_reg[i] = din.get();
        }
        DT_2 out_val;

        out_val = f(shift_reg);

        dout.put(out_val);
#endif
```

```
#ifdef CODE2
        shift_reg[cnt] = din.get();
        cnt++;
        DT_2 out_val;
        if (cnt==4) {
           cnt=0;
           out_val = f(shift_reg);
        }
        dout.put(out_val);
#endif
```

```
#ifdef CODE3
        DT_1 in_val;
        in_val = din.get();
        for (int i=3;i>0;i--) {
            shift_reg[i] = shift_reg[i-1];
        }
        shift_reg[0] = in_val;
        cnt++;
        DT_2 out_val;
        if (cnt==4) {
           cnt=0;
           out_val = f(shift_reg);
        }
        dout.put(out_val);
#endif
```

```
DT_2 dut::f(DT_1*arr)
{
    const int coeff[4] = {3, 5, 7, 11};
    DT_2 tmp=0;
    for (int i=0;i<4;i++)
        tmp+=arr[i]*coeff[i];
    return tmp;

}
```

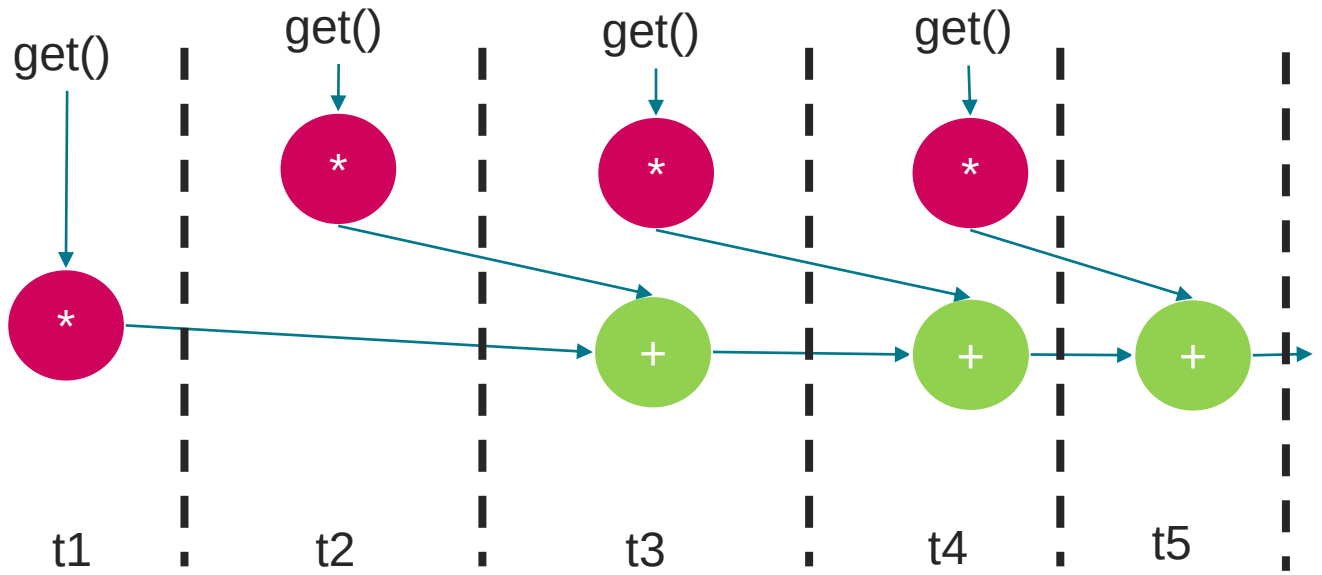
Assume micro-architectures:

1. Latency 0-10
2. Non-pipelined design
3. Unrolling all for-loops

Please consider HW when doing SystemC modeling

```
shift_reg[0] = get(); // 1t
shift_reg[1] = get(); // 1t
shift_reg[2] = get(); // 1t
shift_reg[3] = get(); // 1t
out = shift_reg[0] * coeff[0] +
shift_reg[1] * coeff[1] +
shift_reg[2] + coeff[2] +
shift_reg[3] + coeff[3];
put(out); // 1t
```

CODE 1



HLS 做的

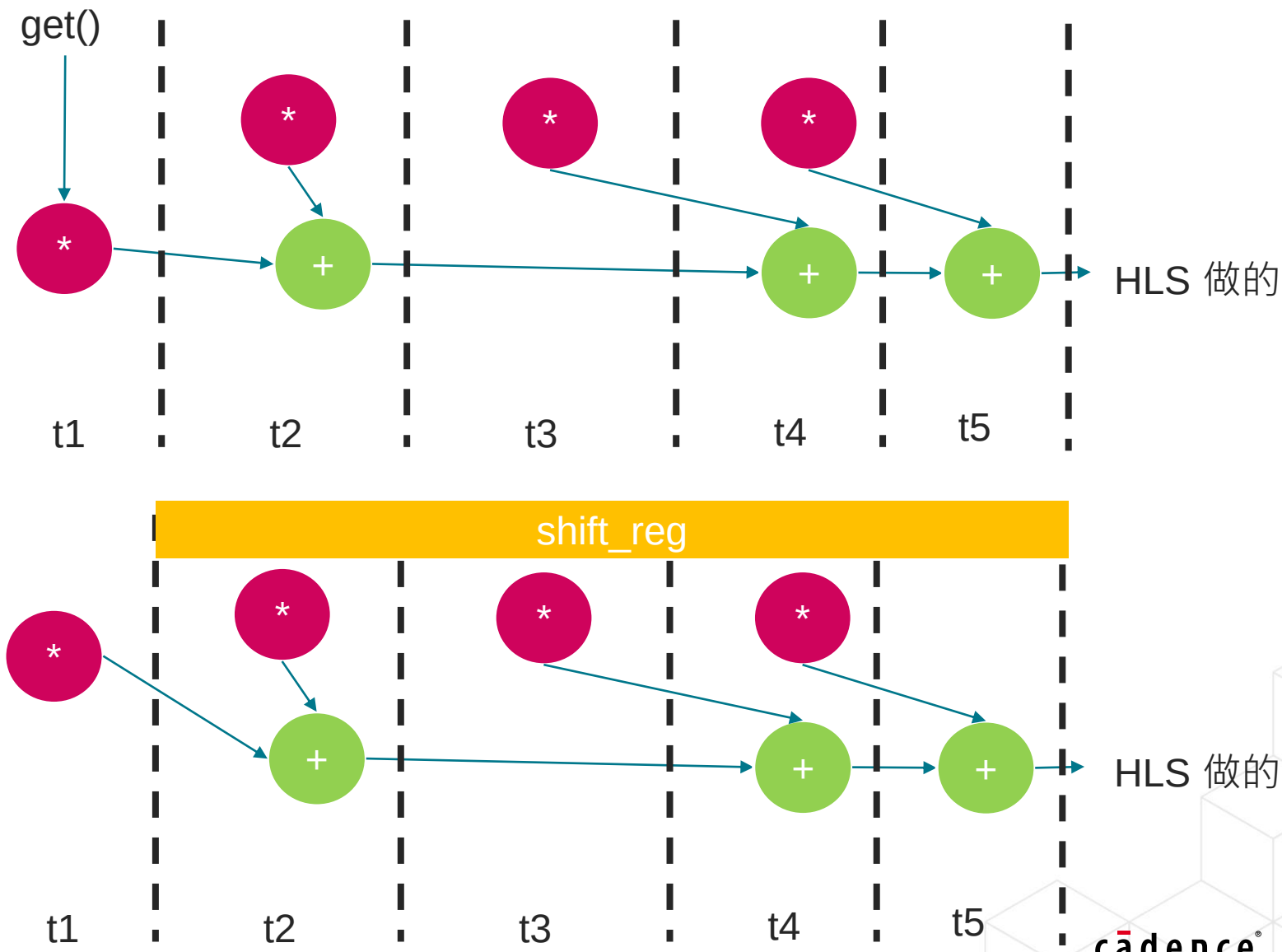
Please consider HW when doing SystemC modeling

```
#ifdef CODE2
        shift_reg[cnt] = din.get();
        cnt++;
        DT_2 out_val;
        if (cnt==4) {
           cnt=0;
           out_val = f(shift_reg);
        }
        dout.put(out_val);
#endif
```

Adder for variable cnt
MUX for shift_reg access

```
#ifdef CODE3
        DT_1 in_val;
        in_val = din.get();
        for (int i=3;i>0;i--) {
            shift_reg[i] = shift_reg[i-1];
        }
        shift_reg[0] = in_val;
        cnt++;
        DT_2 out_val;
        if (cnt==4) {
           cnt=0;
           out_val = f(shift_reg);
        }
        dout.put(out_val);
#endif
```

Adder for variable cnt



Please consider HW when doing SystemC modeling

```
#ifdef CODE1
        for (int i=0;i<4;i++) {
             shift_reg[i] = din.get();
        }
        DT_2 out_val;

        out_val = f(shift_reg);

        dout.put(out_val);
#endif
```

```
#ifdef CODE2
        shift_reg[cnt] = din.get();
        cnt++;
        DT_2 out_val;
        if (cnt==4) {
           cnt=0;
           out_val = f(shift_reg);
        }
        dout.put(out_val);
#endif
```

```
#ifdef CODE3
        DT_1 in_val;
        in_val = din.get();
        for (int i=3;i>0;i--) {
            shift_reg[i] = shift_reg[i-1];
        }
        shift_reg[0] = in_val;
        cnt++;
        DT_2 out_val;
        if (cnt==4) {
           cnt=0;
           out_val = f(shift_reg);
        }
        dout.put(out_val);
#endif
```

```
DT_2 dut::f(DT_1*arr)
{
    const int coeff[4] = {3, 5, 7, 11};
    DT_2 tmp=0;
    for (int i=0;i<4;i++)
        tmp+=arr[i]*coeff[i];
    return tmp;

}
```

Assume micro-architectures:

1. Minimum achievable latency
2. Pipelined design with throughput = 1
3. Unrolling all for-loops

Please consider HW when doing SystemC modeling

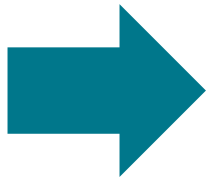
Code	Latency (#pipe stage)	Logic synthesis area
CODE1	5	891
CODE2	2	949
CODE3	2	881

Logic Synth Config	Module	Config	Rep	# Insts	Total Area	Worst Slack	Combinational Area	Sequential Area	# FFs
CODE1_LS			GATES_V		891	3328	408	468	76
CODE2_LS			GATES_V		949	3040	499	432	67
CODE3_LS			GATES_V		881	3376	453	418	68

Suggestion for SystemC Coding

- Use proper bitwidth for variables
- Function sharing manually usually results in better QoR

```
if (a)
   funcA(m, n, o);
else if (b)
   funcA(p, q, r);
else
   funcA(x, y, z);
```



```
if (a)
   t1=m; t2=n; t3=o;
else if (b)
   t1=p; t2=q; t3=r;
else
   t1=x; t2=y; t3=z;

funcA(t1, t2, t3);
```

Suggestion for SystemC Coding

- Reduce lifetime of variables (for register reduction)

```
void my_thread() {
      // declare permanent variable here
      ….
      ….
      while(1) {
          // declare intermediate variables here

          for (int i=0;i<10;i++) {
                ….
                ….
          }
      }
}
```

Suggestion for SystemC Coding

- Use switch-case with constant instead when full-case will not happen for shift operation, array access, and partial selection.

```
sc_uint<64> data;
sc_uint<6> shifter;
…
result = data << shifter;
```

Analyze algorithm

```
sc_uint<64> data;
sc_uint<6> shifter;
…
switch (shifter)
{
case 2: result = data << 2;break;
case 4: result = data << 4; break;
…
}
```

```
sc_uint<32> data;
sc_uint<5> idx
…
result.range(7,0) =
data.range(idx, idx-7);
```

Analyze algorithm

```
sc_uint<32> data;
sc_uint<5> idx;
…
switch (shifter)
{
case 7:   result.range(7, 0) = data.range(7, 0);break;
case 15: result.range(7, 0) = data.range(15,7); break;
case 23: ….
case 31: ….
}
```

Suggestion for SystemC Coding

- Explicit (constant) descriptions for shift, array accesses, partial selection (if necessary for QoR)

```
sc_uint<32> data[128];
sc_uint<7> idx;

a = data[idx];
b = data[idx+1];
```

Analyze algorithm

```
sc_uint<32> data[128];
sc_uint<7> idx;

switch (idx)
{
case 0: a = data[0]; b = data[1]; break;
case 2: a = data[2]; b = data[3]; break;
…..
case 126: a = data[126]; b = data[127];
```

Suggestion for SystemC Coding

- Schedule becomes easy

```
if (a) {
   if (b) {
      if (d) {
         if (e) {
            rdata = mem[addr];
         }
         else {
            rdata = mem[addr+2];
         }
      }
      else {
            rdata = mem[addr+8];
      }
   }
}
else {
   rdata = mem[addr+10];
}
```

```
if (a) {
   if (b) {
      if (d) {
         if (e) {
            raddr = addr;
            ren = 1
         }
         else {
            raddr = addr+2;
            ren=1;
         }
      }
      else {
            raddr = addr + 8;
            ren=1;
      }
   }
}
else {
   raddr = addr+10;
   ren=1;
}

if (ren) {
   ren=0;
   rdata = mem[raddr];
}
```

Connect to RTL implementation flow

Situations will happen

- RTL sign-off
  - Correct functionality
    - still simulation-based.
    - **Formal check between SystemC and RTL is still not mature.**
  - Acceptable area/power
    - achievable
  - Pass linting
    - Need to waive
    - Review is difficult due to huge number of lines of generated RTL
    - **Should be soft criteria for HLS-generated RTL**
  - Pass code coverage & functional coverage
    - **Impossible to achieve 100% line coverage**
    - UNR to eliminate unreachable codes can help (still cannot achieve 100%)
    - **Code coverage should be soft criteria for HLS-generated RTL**

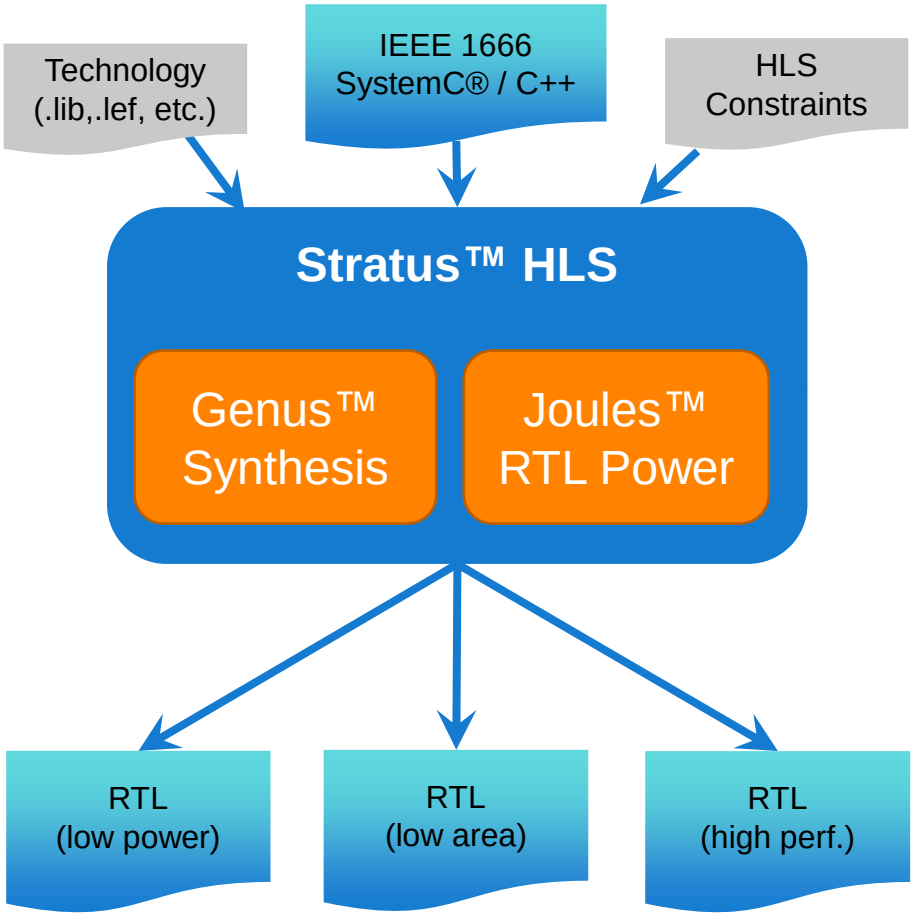
Situations will happen

- RTL Synthesis
  - Pass logic equivalent check (LEC)
    - HLS-generated RTL module is usually more complicated than hand RTL
    - Maybe longer runtime to complete LEC

- ECO if required (pre-masking ECO)
  - **ECO rarely happened in HLS designs because HLS usually is used in algorithm implementation.**
  - What's ECO scope? Reasonable?
  - Doable, but higher effort than hand-RTL

Stratus HLS Brief

Overview



Behavioral IP is
broadly reusable across
technologies and PPA targets

10x faster to verified production silicon

Writing RTL by hand

- Functionality
- Architecture
- Constraints
- Sharing
- Schedule of operations
- Multiplexers
- Pipeline registers
- Area reduction
- FSM encoding
- Timing closure
- Clock gating
- Consistent RTL style
- ECO

1-2 architectures
in 5-6 months

Creating RTL with HLS

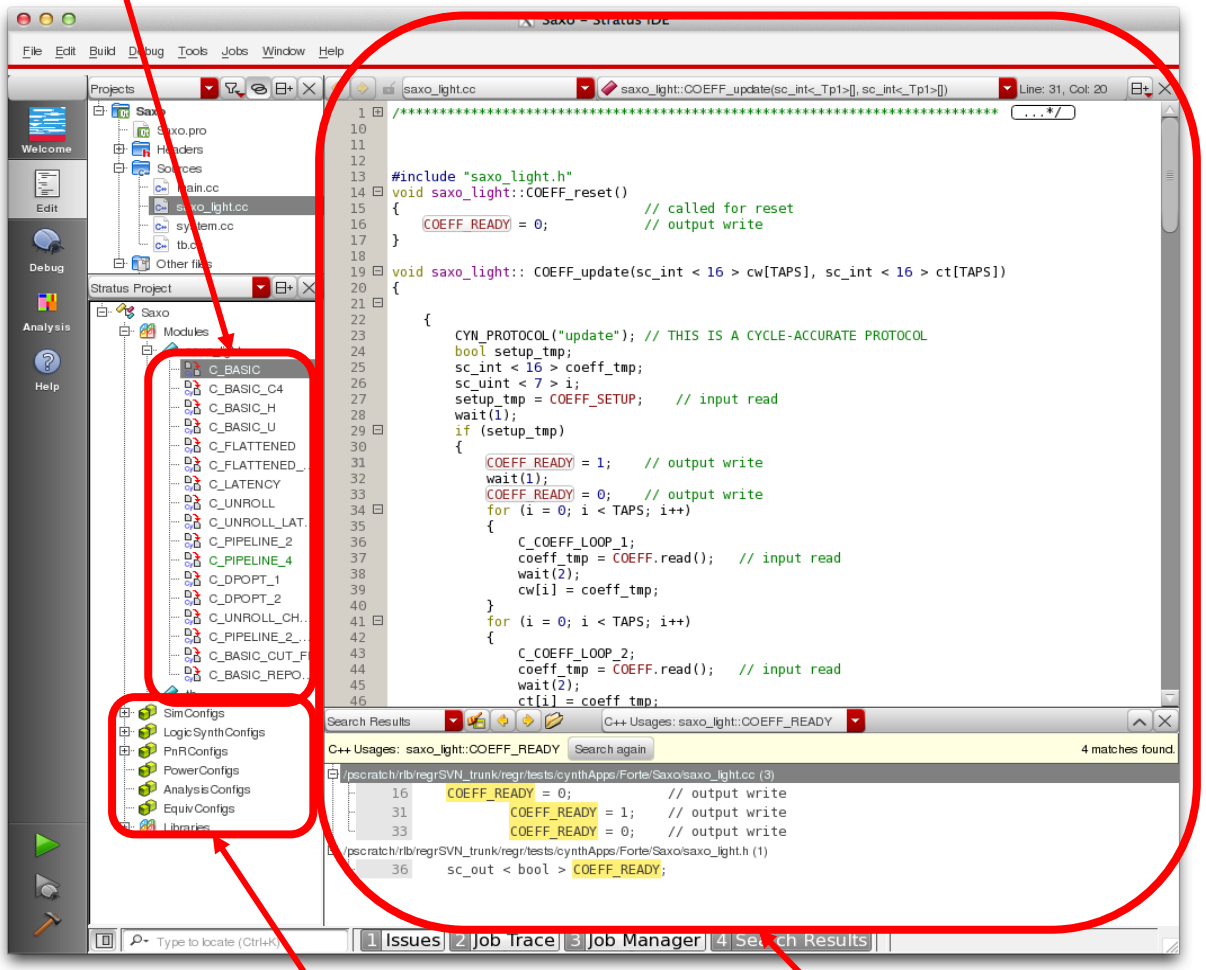
- Functionality
- Architecture
- Constraints
- **Automated by Stratus HLS**

100's of architectures
in 1-3 weeks

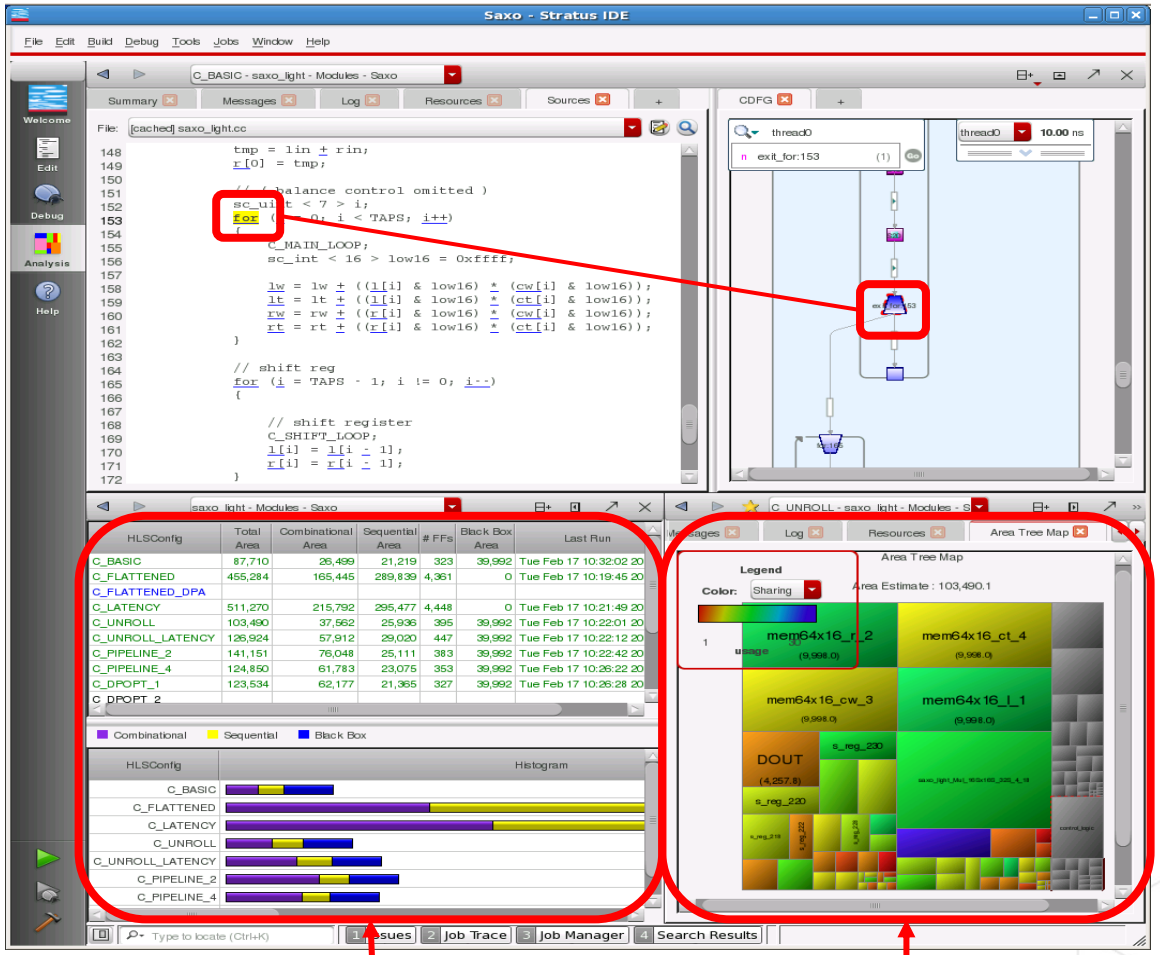
Choosing the right architecture can achieve
2x better Power, Performance & Area

Do everything in Stratus IDE

Automates multiple implementations from one source



Hot links back to IP source code



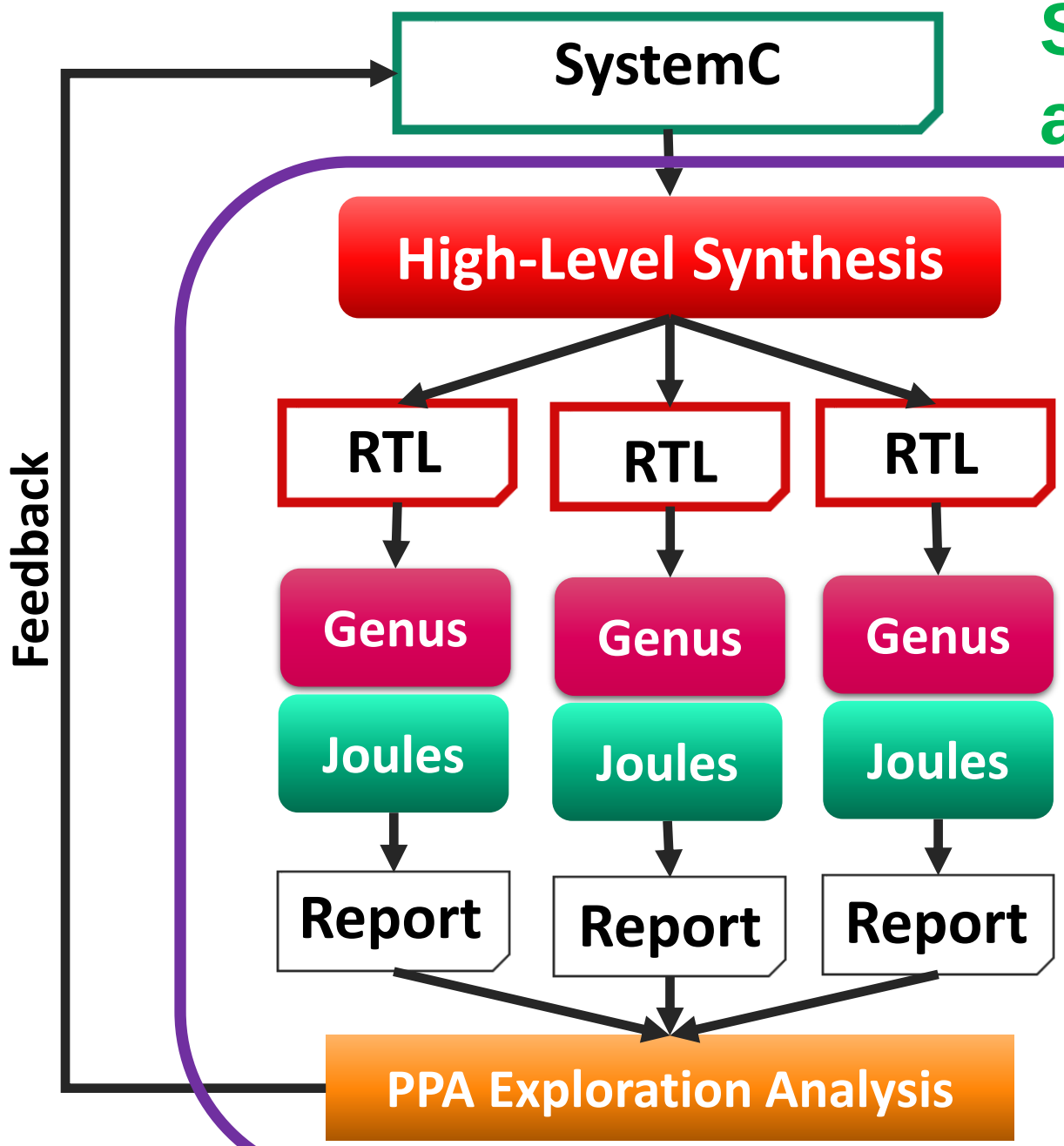
Automates integration with downstream tools

"Standard" IDE

Compare multiple configurations

Detailed analysis of each config.

Stratus HLS enables physical-aware PPA explorations



Stratus HLS
Integrated Design Environment

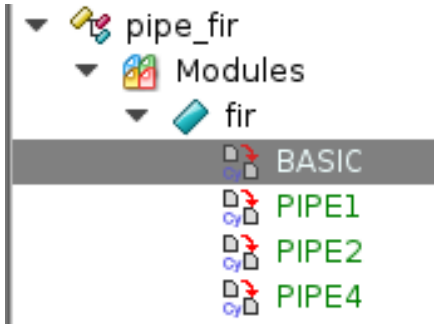
Three Genus modes are supported

Genus Basic	• Stratus license enables it
Genus PLE	• Stratus license enables it
Genus iSpatial	• Genus Synthesis (GEN100) • Genus Physical Opt (GEN40) • Innovus GigaPlace GXL Opt

Micro-architecture exploration by Stratus IDE

```
SC_MODULE(xxx)
{
   sc_in<bool> clk;
   sc_in<bool> nrst;
   cynw_p2p<my_data>::in din;
   cynw_p2p<my_data>::out dout;
   ….
   SC_CTOR() {
      ….
   }
};

….
while (1) {
#ifdef PIPE
    HLS_PIPELINE_LOOP(HARD_STALL, V_II, “pipe”);
#else
    HLS_CONSTRAINT_LATENCY(0, HLS_ACHIEVABLE, “lat”);
#endif
    idata = in.get();
    result = func(idata);
    out.put(result);
}
```

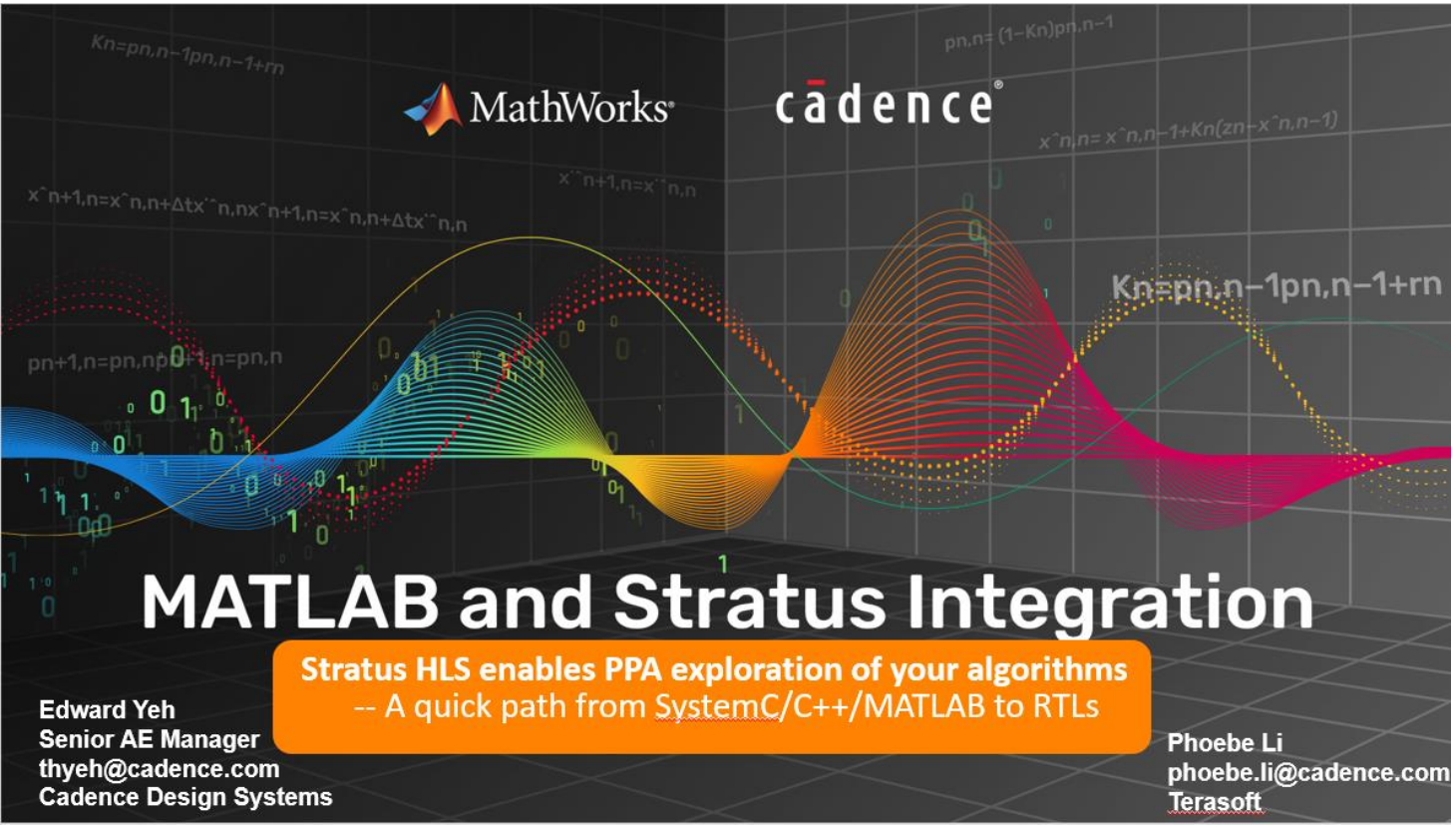



```
define_hls_config $design BASIC
define_hls_config $design PIPE1 -DPIPE -DV_II=1
define_hls_config $design PIPE2 -DPIPE -DV_II=2
define_hls_config $design PIPE4 -DPIPE -DV_II=4
```

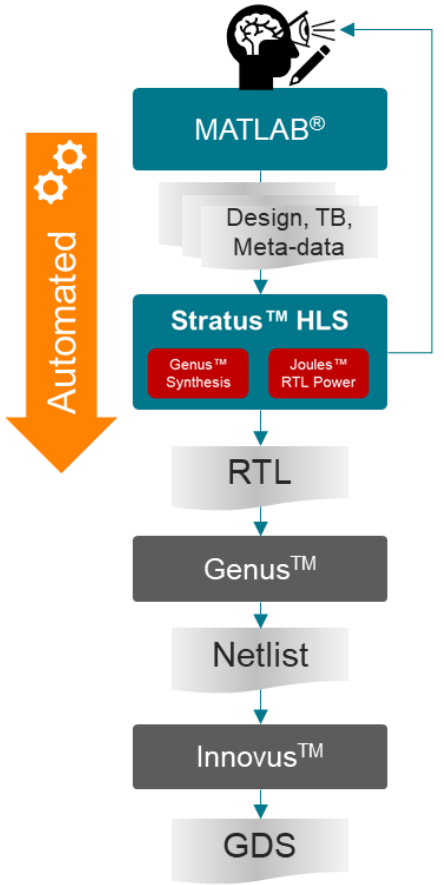
Architecture explorations

Minutes in Stratus vs. Days in RTL

Stratus HLS X MATLAB



New Flow



- Automated creation of synthesizable SystemC
- Automated import and Stratus project creation
  - Includes design and TB

Benefits

- Early PPA visibility
- Productivity force multiplier
  A single engineer produces results previously requiring a team

Stratus Synthesizable IP Functions (Highlights)

Pre-verified building blocks accelerate design and verification

Communication / Interfaces

- Simple bus
- AXI4, AXI4-Lite
- AXI3, APB
- Memories
- FIFOs
- CDCs
- Line buffer
- Stream buffer
- Circular buffer
- Point-to-point handshake
- Trig-done (with data)

Synthesizable Datatypes

- Floating point
- Fixed point
- Complex

Fixed-Point + Integer Functions

- barrelShift function
- barrelShifter class
- min / max(array)
- min_index / max_index(array)
- min / max(v1, v2)
- abs(value)
- swap(v1, v2)

Fixed-Point Functions

- Iterative fixed-point divider
- Sine / Cosine
- Normalized Sine / Cosine
- log2, exp
- atan2, tanh

Integer Functions

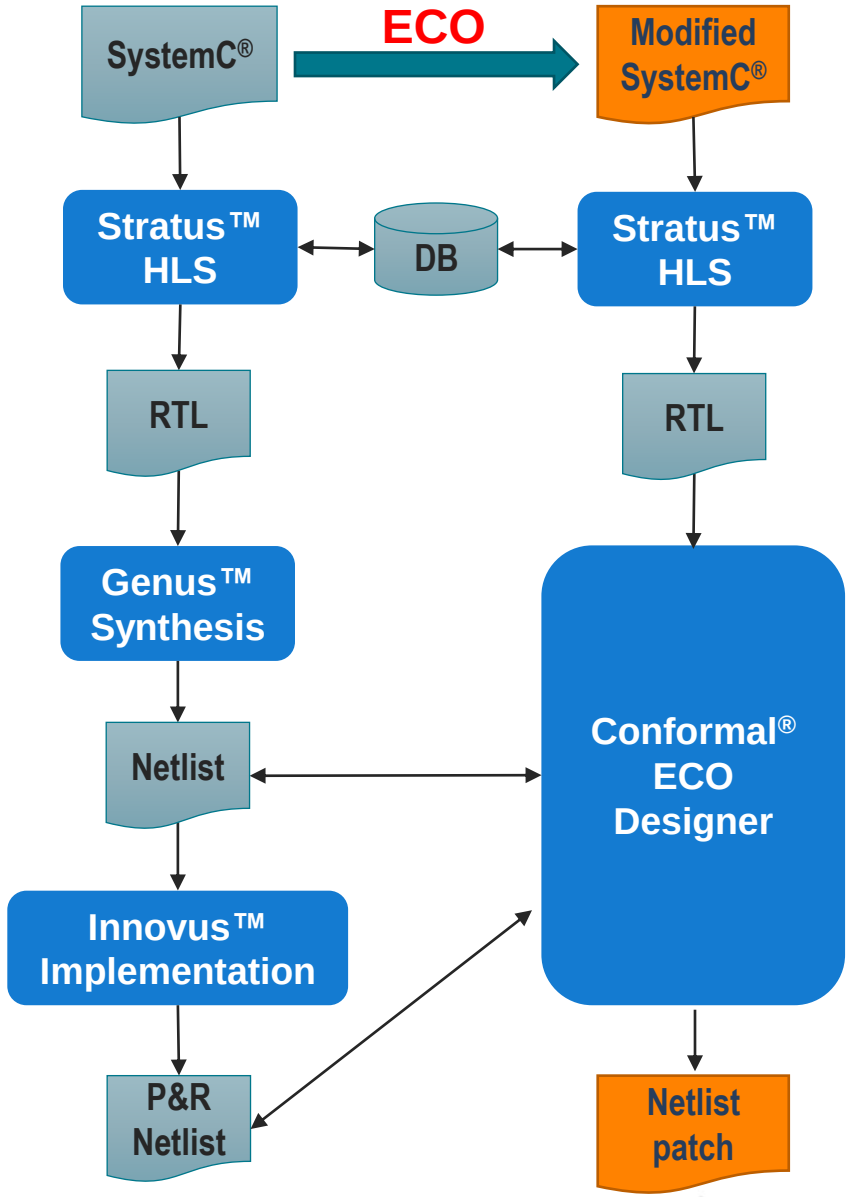
- leadingOne::fromMSB
- leadingOne::fromLSB
- iterative divider
- clip(value,min,max)
- clamp(value,min,max)

Stratus ECO Flows

- Stratus™ HLS supports two methods for ECOs

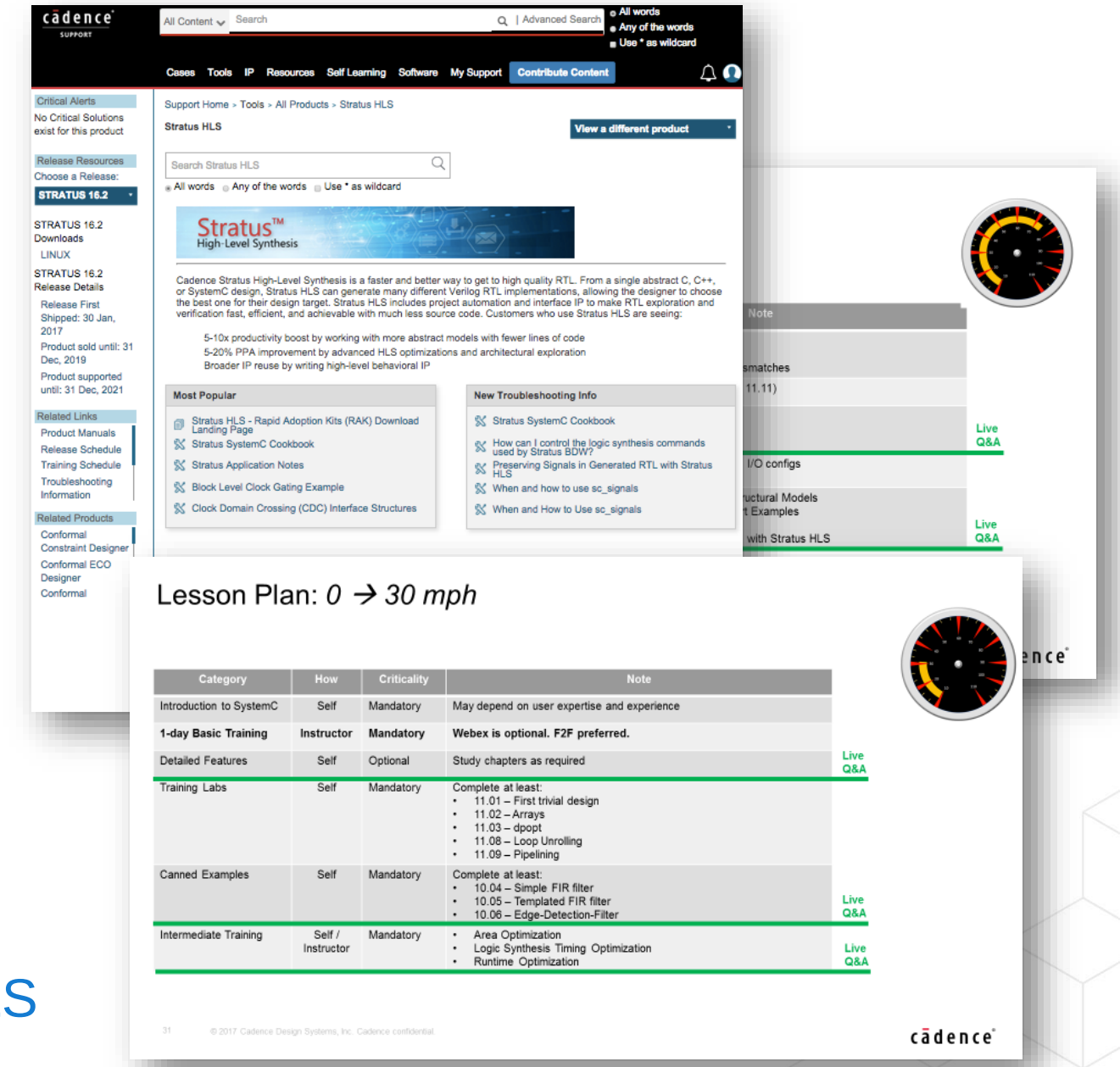
- Top-down ECO
  - Change is made in SystemC
  - Stratus ECO mode minimizes changes in RTL
  - Conformal® ECO creates netlist patch to minimize impact on implementation flow

- Bottom-up ECO
  - Change is made in netlist
  - Manual change is propagated to RTL and SystemC
  - No re-synthesis minimizes risk



Stratus HLS – Education

- Extensive web-based materials
  - Rapid Adoption Kits (RAKs)
  - Application notes
  - Tips, tricks, best practices
- Includes lesson plans
  - Based on experience and time available
- Examples and labs
  - Included in the Stratus IDE
  - Use as kick-start templates
- http://support.cadence.com/StratusHLS



Frequently-used Synthesis Directives

Synthesis constraints & micro-architecture - 1

- Tell Stratus HLS HOW to implement your design.
  - I/O protocols
  - Pipelined or non-pipelined.
  - Latency constraint
  - Loop unrolling or not
  - Flatten array or register file, or memory

- Stratus HLS assigns synthesis constraints by DIRECTIVESs in codes or TCL commands in project.tcl
  - **Different synthesis constraints result in different RTL structures.**

Synthesis constraints & micro-architecture - 2

- HLS directive's effective scopes
  - Many HLS directives are effective within the '{ }'
  - Some directives are effective entire module when it is specified in a module constructor.

- Micro-architecture constraints
  - HLS_UNROLL_LOOP
    - Specify unrolling loops of for, while, do/while
  - **HLS_PIPELINE_LOOP**
    - Specify pipelining loops of for, while, do/while
  - **HLS_CONSTRAIN_LATENCY**
    - Constrain latency in a region within '{ }'
  - HLS_FLATTEN_ARRAY
    - Specify flattening arrays
  - **HLS_MAP_TO_MEMORY**
    - Specify mapping arrays to memories

- Timing constraints
  - **HLS_DEFINE_PROTOCOL**
    - Specify cycle-accurate regions (reset, protocol specific)
  - HLS_SET_INPUT_DELAY
    - Specify input delay for input ports or sc_signals
  - **HLS_ASSUME_STABLE**
    - Specify input ports or sc_signals to be treated as stable.

I/O Protocols

- Stratus HLS will inject cycles to meet performance requirements and minimize area
  - “free scheduling”
- However, all designs communicate with their surrounding modules in a cycle accurate manner
  - “fixed scheduling”
- Example: an array must be populated by 8 input port reads in 8 consecutive cycles
  - The wait() inside the ‘for’ loop will create this timing

Argument is a label used for reporting

Enclose the protocol code in braces

```
{
  HLS_DEFINE_PROTOCOL( “read_protocol” );
  for( int i = 0; i < 8; i++ ) {
    MEM[i] = inp.read();
    wait();
  }
}
```

HLS_DEFINE_PROTOCOL() tells Stratus HLS that the schedule for this block of code is fixed.

Any wait() inside the block will always become one clock cycle

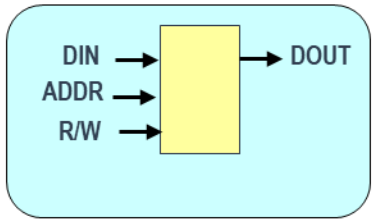
Arrays, Storage, and Memories

- Stratus HLS can implement arrays in the following hardware styles

```
SC_MODULE( dut )
{
  …
  sc_uint<8>  mem[256];

  void thread0() {
       …
       a = mem[1] + mem[2];
    …
  }
}
```

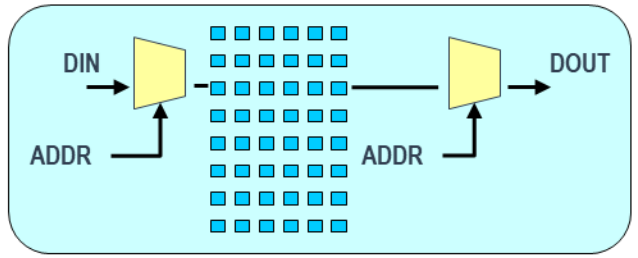
HLS_MAP_TO_MEMORY



This is the default

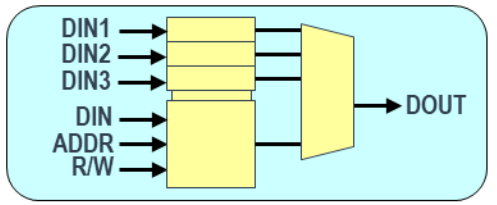
Memory

HLS_FLATTEN_ARRAY



Flattened into a set of individual variables

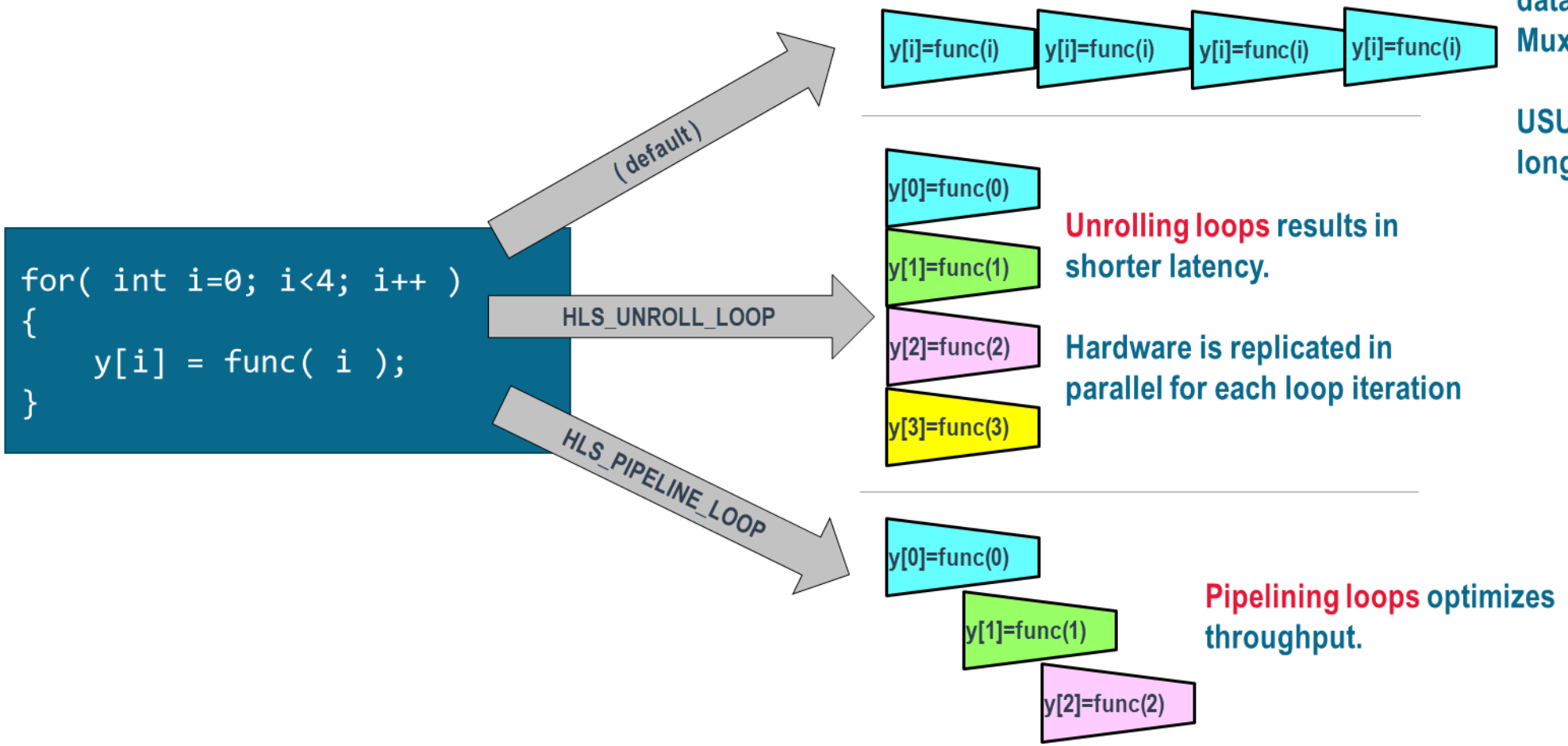
HLS_MAP_TO_REG_BANK



A register bank

Loops - 1

- Stratus HLS can implement loops with the following hardware



In rolled up loops datapath is one iteration of loop. Muxing shares hardware.

USUALLY, smallest hardware and longest latency

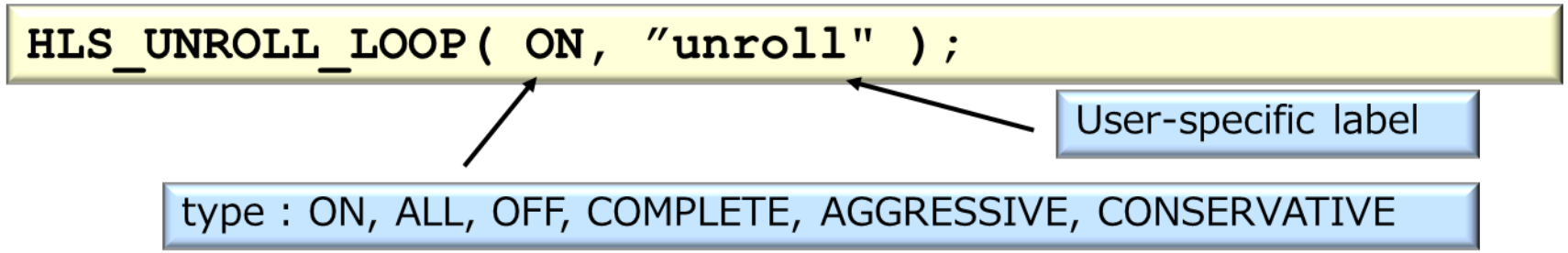
Unrolling loops results in shorter latency.

Hardware is replicated in parallel for each loop iteration

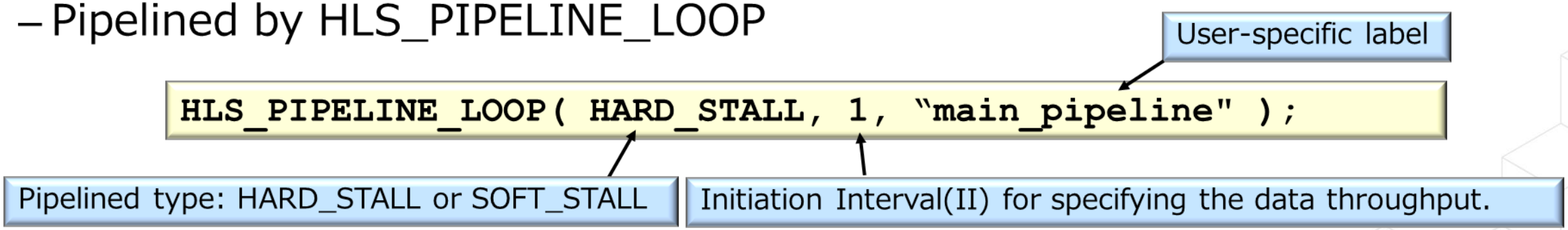
Pipelining loops optimizes throughput.

Loops - 2

- The loops can be specified with 3 types of the micro-architectures.
  - Non-unrolled and non-pipelined
  - Unrolled by HLS_UNROLL_LOOP

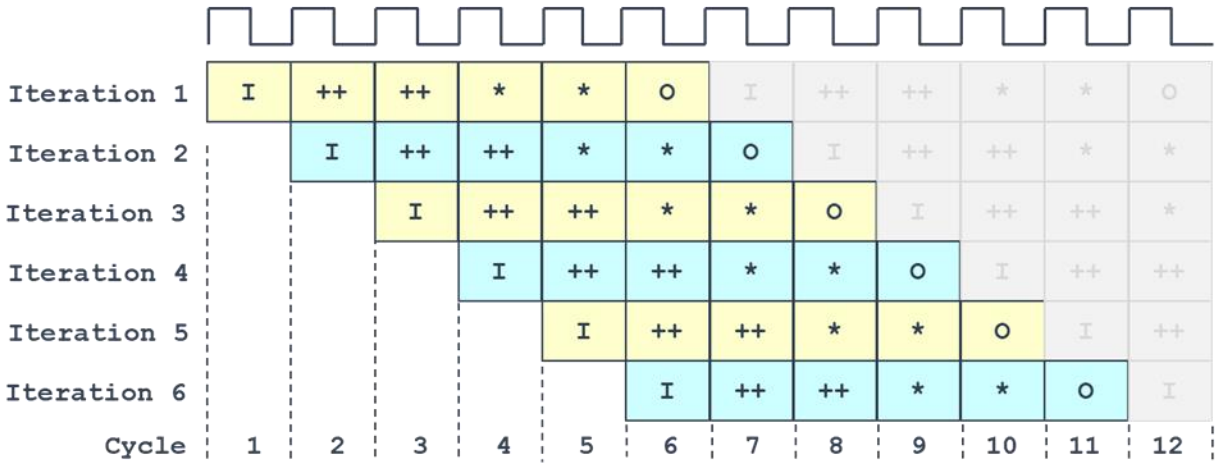


  - Pipelined by HLS_PIPELINE_LOOP



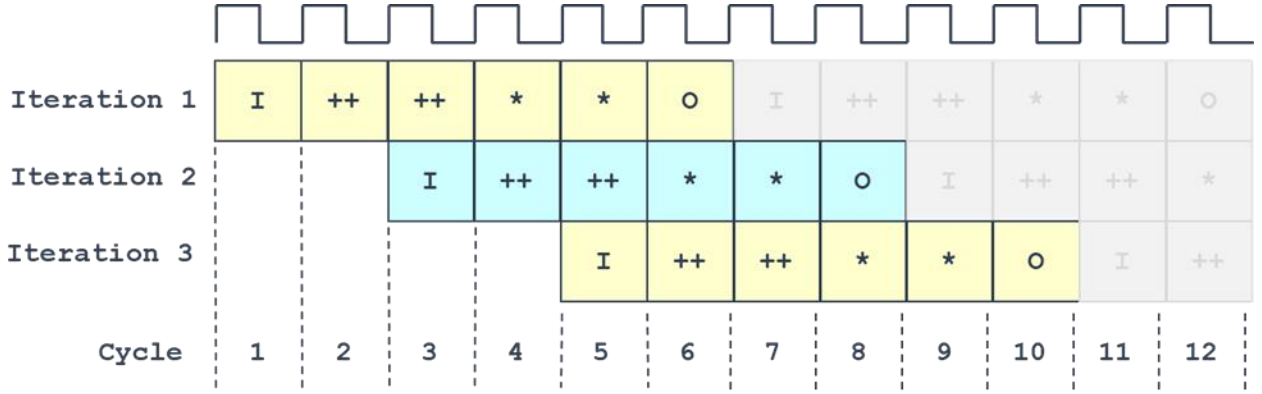
Loop Pipelining

- Different initiation interval (II) results in different pipeline architecture.



Resources	4 Adders, 2 Multipliers
Latency	6 clock cycles
Throughput	1 I/O per 1 clock cycles

Fully pipelined for a 6x improvement in throughput and only 2x increase in resources!

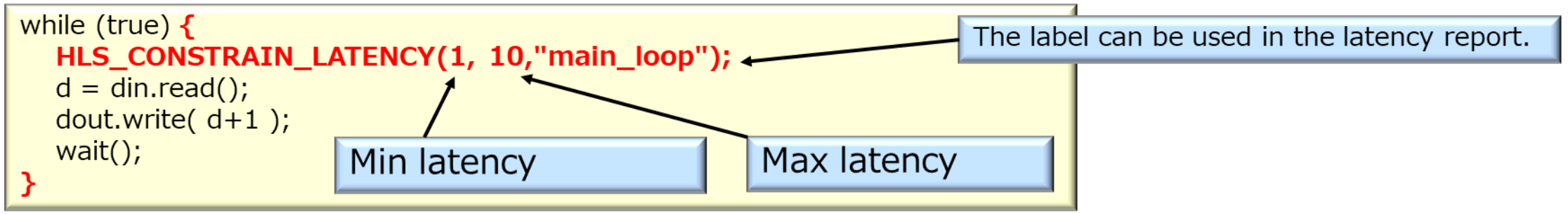


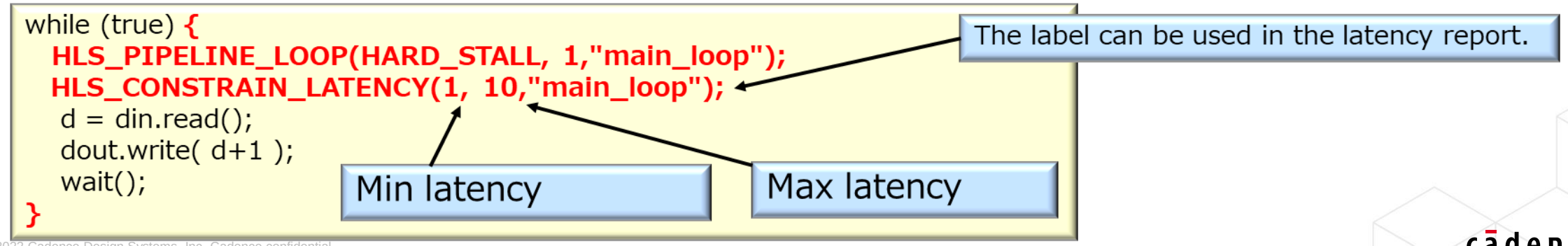
Resources	2 Adders, 1 Multiplier
Latency	6 clock cycles
Throughput	1 I/O per 2 clock cycles

Throughput tripled with no increase in resources!

Latency Constraint

- It can be used on any region bounded with {..}
- Syntax
  HLS_CONSTRAIN_LATENCY(min_lat, max_lat, “user-specific label”)
- It can also be used to specify the latency of the pipelined loop.





Typical SC_MODULE with constraints

```
#include "DUT.h"

void DUT::proc() {
  {HLS_DEFINE_PROTOCOL("reset");
    OUT.write(0);
    wait();
  }
  while(true) {
    HLS_PIPELINE_LOOP(HARD_STALL,1,"my_pipe");
    HLS_CONSTRAIN_LATENCY(0,HLS_ACHIEVABLE,"my_lat");
    HLS_ASSUME_STABLE(IN_MODE,"IN_MODE");
    sc_uint<8> temp1, temp2;
    sc_uint<32> sum;
    sc_uint<2> mode;

    {HLS_DEFINE_PROTOCOL("input");
      temp1 = IN_A.read();
      temp2 = IN_B.read();
      mode =  IN_MODE.read();
    }

    if (mode==1)
       sum = temp1 + temp2 + mode * 11;
    else if (mode==2)
       sum = temp1 - temp2 + mode * 22;

    {HLS_DEFINE_PROTOCOL("output");
      OUT.write(sum);
      wait();
    }
  }
}
```

Project.tcl

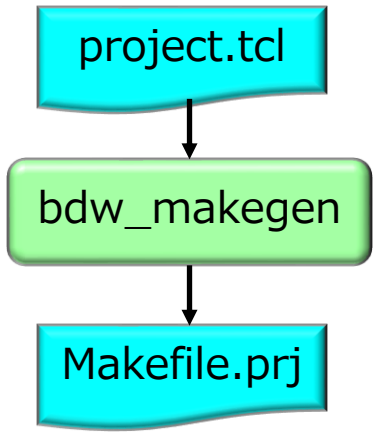
Makefile

- Makefile should include a Makefile.prj which will be generated by Stratus' bdw_makegen command based on project.tcl
- The following is the minimum settings for running Stratus.

Makefile

```
-include Makefile.prj

Makefile.prj : project.tcl
        @bdw_makegen project.tcl
```



Typical project.tcl - 1

```
#*********************************************
# Copyright 2015 Cadence Design Systems, Inc.
# All Rights Reserved.
#*********************************************

#
# Libraries
#
set LIB_PATH     "[get_install_path]/share/stratus/techlibs/ \
                  GPDK045/gsclib045_svt_v4.4/gsclib045/timing"
set LIB_NAME     "slow_vdd1v2_basicCells.lib"
use_tech_lib     "$LIB_PATH/$LIB_NAME"

#
# C++ compiler options
#
set CLOCK_PERIOD "5.0"
set_attr cc_options       " -g -DCLOCK_PERIOD=$CLOCK_PERIOD"
```

Technology library: using a dummy 45nm technology library

Compilation options

Typical project.tcl - 2

```
#
# stratus_hls options
#
set_attr clock_period                   $CLOCK_PERIOD
set_attr balance_expr                   delay
set_attr default_input_delay            0.1
set_attr default_stable_input_delay     0.0
set_attr default_protocol               false
set_attr dpopt_auto                     op (or op,expr)
set_attr flatten_arrays                 all
#set_attr ignore_cells                   "XYZ* ABC*"
set_attr message_detail                 2
set_attr output_style_reset_all         yes
set_attr path_delay_limit               120
set_attr power                          on
set_attr unroll_loops                   on
set_attr wireload                       none
```

Typical project.tcl - 3

```
#
# Simulation Options
#
use_systemc_simulator           built-in/incisive/xcelium
use_verilog_simulator           incisive/xcelium/vcs
enable_waveform_logging         -vcd/-shm/-fsdb

#
# System Module Configurations
#
define_system_module    main      main.cc
define_system_module    system    system.cc
define_system_module    tb        tb.cc

#
# Synthesis Module Configurations
#
define_hls_module       dut       ./src/DUT.cpp
define_hls_config       dut       BASIC
```

Specify simulators

Specify testbench

Specify target HLS module with configurations

Typical project.tcl - 4

```
#
# Simulation Configurations
#
define_sim_config B            {dut BEH}
define_sim_config BASIC_V     {dut RTL_V BASIC}

#
# Logic Synthesis Configurations
#
set_logic_synthesis_options     [list BDW_LS_TECHLIB $LIB_PATH/$LIB_NAME]
{BDW_LS_DO_DISSOLVE 1} {BDW_LS_NOGATES 1}
define_logic_synthesis_config   L {dut -all} -command bdw_rungenus
```

Specify simulation configurations

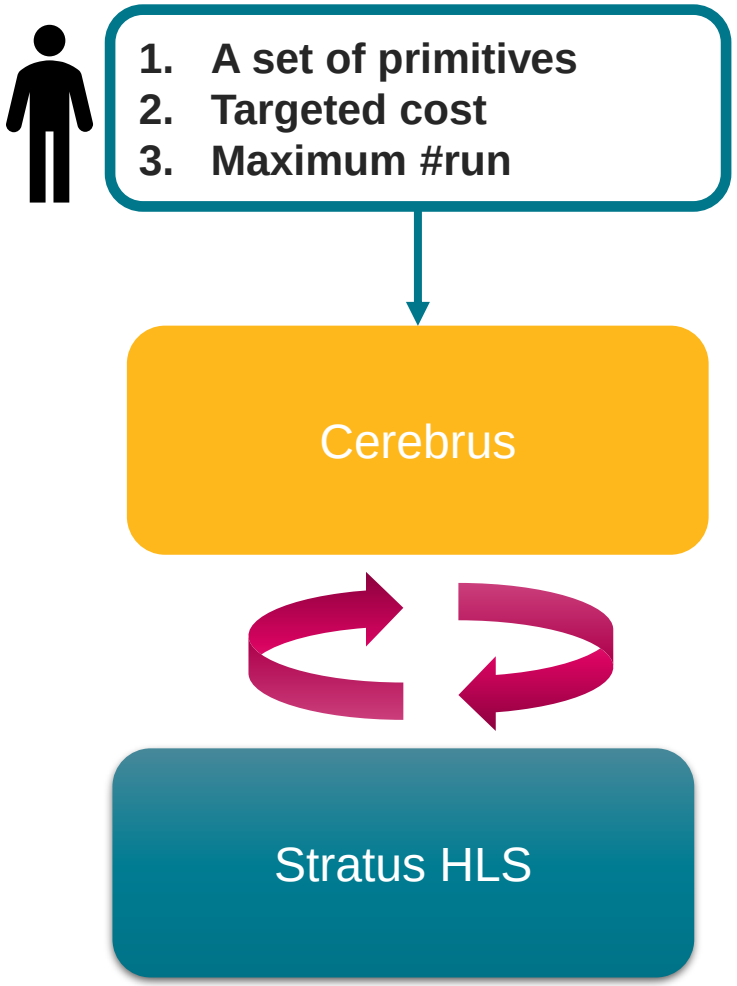
Specify logic synthesis configurations

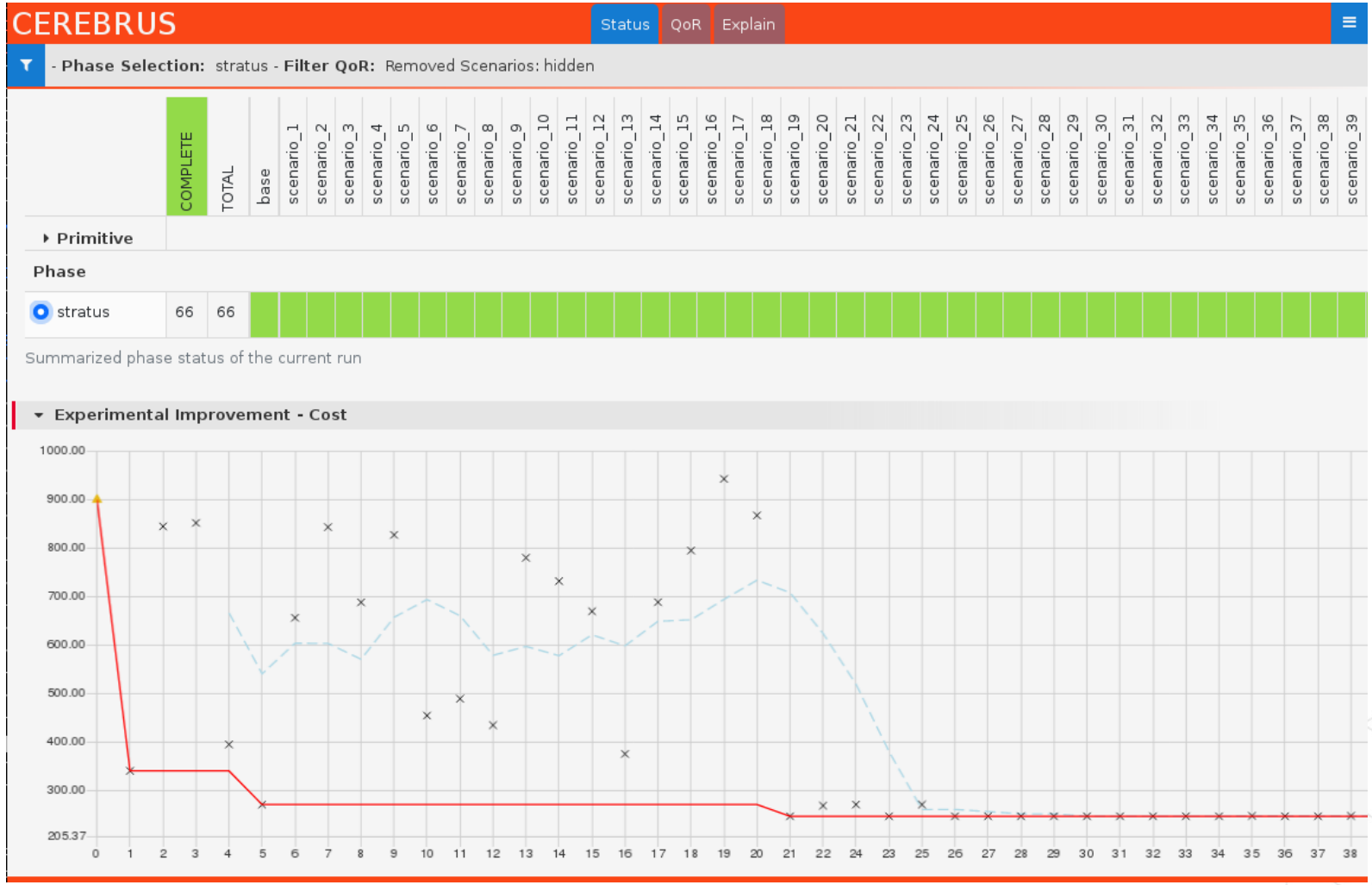
What’s new for Stratus HLS

cadence®

Machine Learning helps Design Space Exploration with HLS

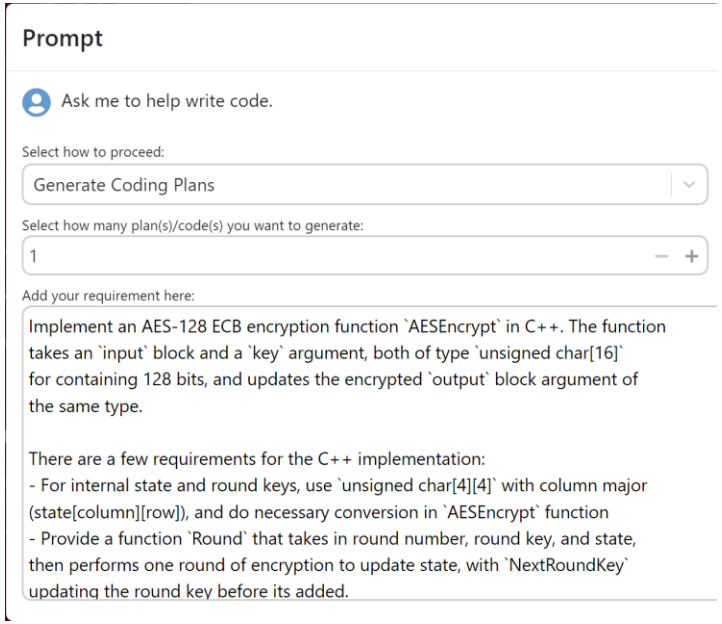
Cerebrus + Stratus

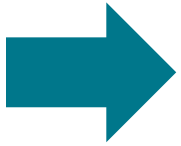




LLM for C++ Code Generation

LLM + Stratus HLS





Some well-know algorithm implementations such as AES128 ECB, RSA256 encryption, FFT1024, were attempted.

- Proof of Concept
- Need prompt engineering