Catapult HLS C++ Training

Yuan-Teng Chang
Application Engineer, Siemens EDA
yuan-teng.chang@siemens.com

SIEMENS

Agenda

Catapult HLS Introduction
Catapult HLS C++
Step-by-Step Lab1: FIR
Step-by-Step Lab2: EdgeDetect IP

What is HLS ?

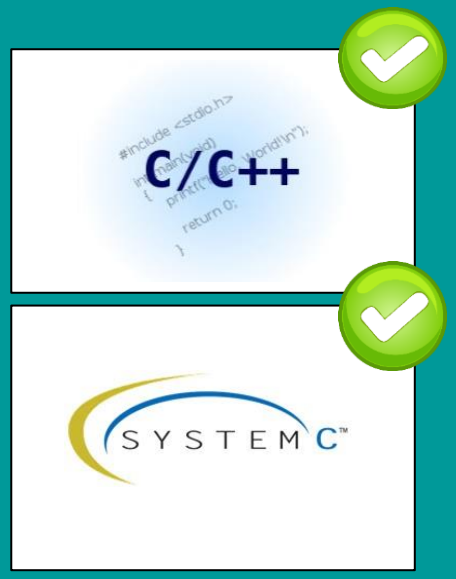
High-level synthesis (HLS) creates RTL implementations from abstract specifications described with C / C++, SystemC, etc.

HLS is still hardware design

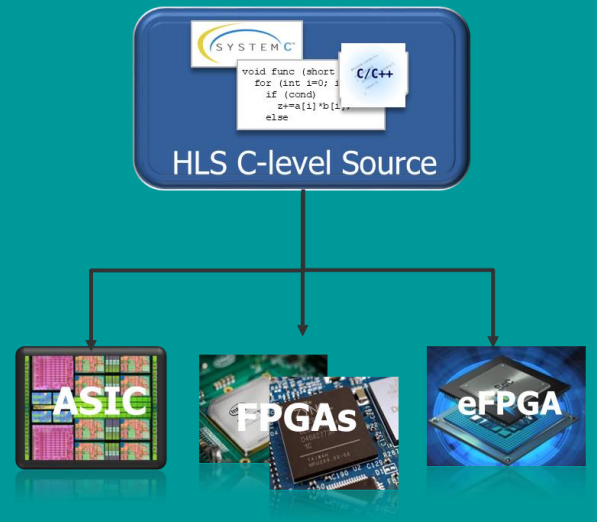
- HLS does NOT “translate” any working C++ code into a good HW
- HLS does NOT turn a SW engineer into a HW designer
- HLS does NOT replace RTL designers, it empowers them

SIEMENS

Catapult Delivers Differentiated Capabilities for Broad Spectrum of Design Needs

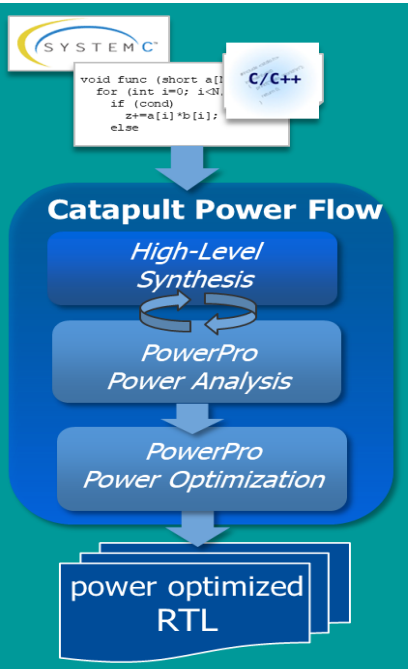


- Pure C++ well-suited for data path functions
- SystemC/MatchLib well-suited for SOC modeling
- Catapult lets the user choose



High speed FPGA and ASIC

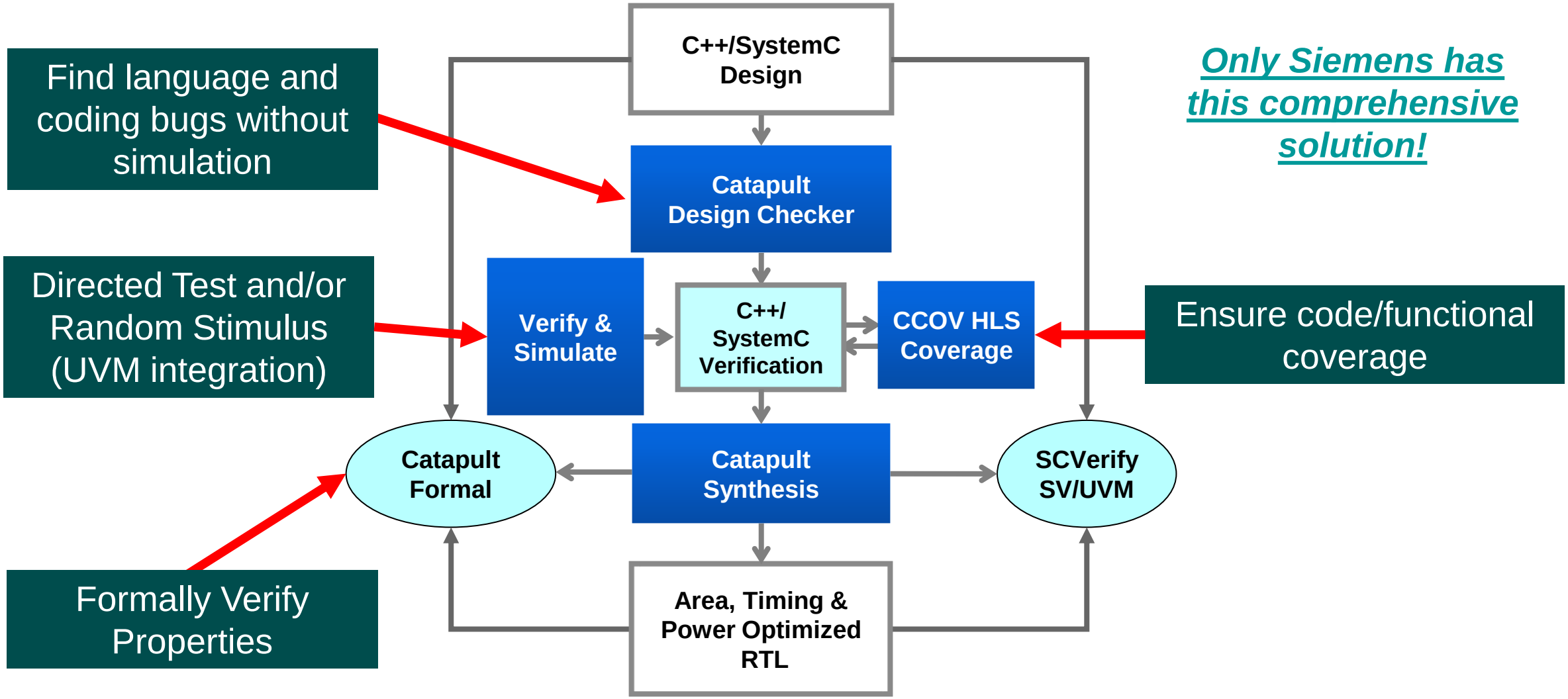
- Same HLS code can be easily retargeted to new technology
- FGPA support built-in
- ASIC libraries built with standard libraries



- PowerPro under-the-hood
- Catapult delivers both early power analysis and low power RTL

SIEMENS

Catapult HLS - More Than “C++/SystemC To RTL”



Only Siemens has this comprehensive solution!

SIEMENS

How HLS Works

Catapult HLS is the Best Solution for Rapid Algorithm to HW

Enable late functional changes without impacting schedule

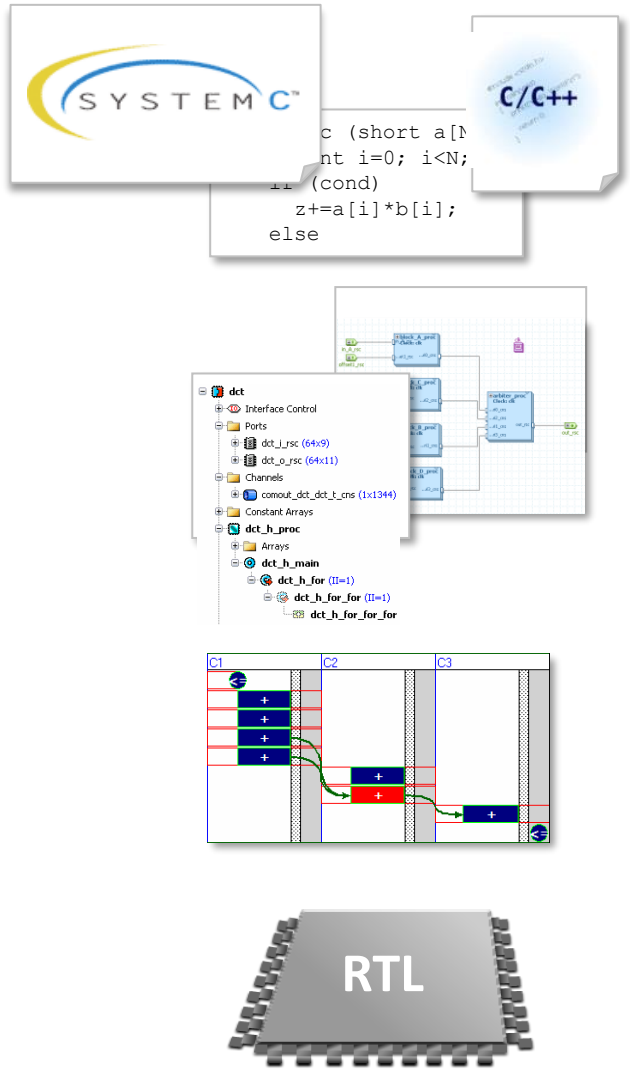
- Algorithms can be easily modified and regenerated
- New technology nodes are easy (or FPGA to ASIC)

Quickly evaluate power and performance of algorithms

- Rapidly explore multiple options for optimal Power Performance Area (PPA)

Accelerate design time with higher level of abstraction

- 1 Year reduced to a few months
- New features added in days not weeks
- 5X less code than RTL



SIEMENS

Catapult HLS builds Synchronous RTL modules from C++ Classes or Functions

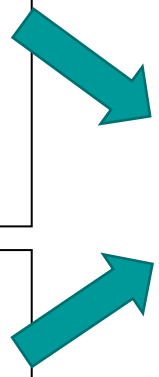
C++ Class/function synthesized as a clocked process

RTL functionality matches C++ functionality

- Make sure to fully test the C++

```
class simpleClass{
  …
  public:
  void simple_function(<function interface variables>){
    <function body>
  }
};
```

```
void simple_function(<function interface variables>){
  <function body>
}
```



```
module simple_function ( <module ports> );
  always@(posedge clk)
    begin
      <module body>
    end
endmodule
```


SIEMENS

HLS Adds Interface Protocols to Untimed C++

Adding an interface protocol to an untimed C++ design is known as “Interface Synthesis”

Source does not specify the protocol

Port size and direction is inferred from the source

Interface synthesis allows the protocol to be defined using the HLS tool

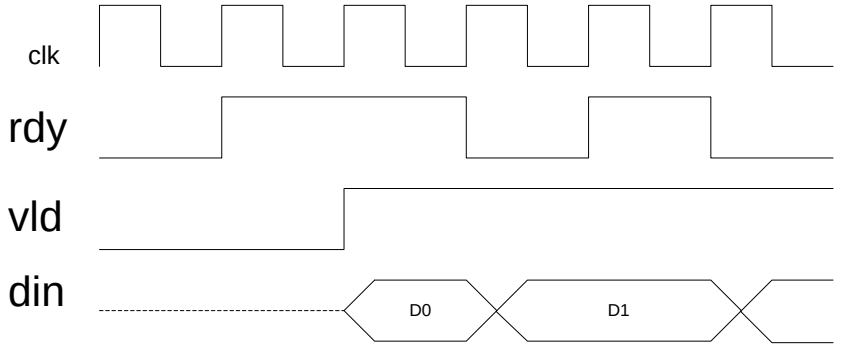
- Request/acknowledge handshake
- Data enable
- Memory interface (address, data, write enable, etc)

Interface handshaking protocol applied on “din”

```
void simple (int din, int &dout) {
  dout = din;
}
```

Variable “dout” is only written so it result in creation of an outputport.

Variable “din” is only read so it result in creation of an input port.



SIEMENS

HLS Infers Memories or Registers from C++ Arrays

Automatically mapped to ASIC or FPGA memories/registers

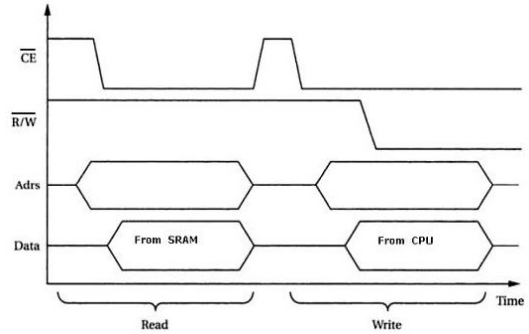
User control over memory mapping

Arrays on the design interface can be synthesized as memory interfaces

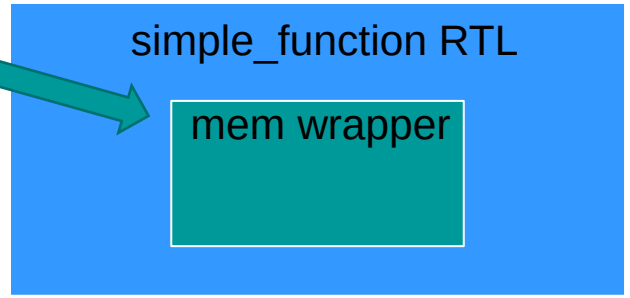
- Address, data, control

```
void simple_function(… ,int data[1024]){
  int mem[1024];
  <function body>
}
```

Memory interface protocol



Instantiated memory wrapper



SIEMENS

HLS Explores Parallelism and Concurrency

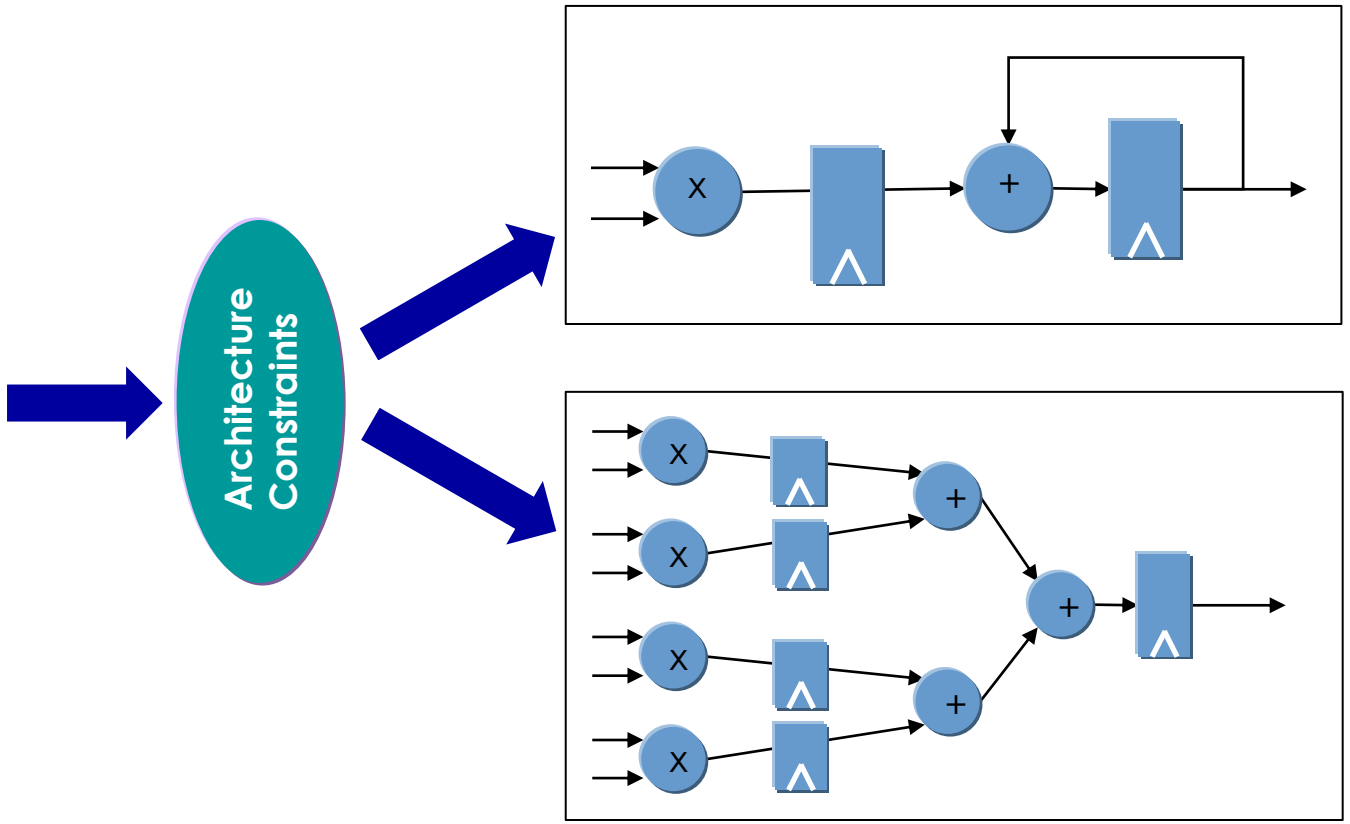
Exploration done using loop transformations

- Unrolling and pipelining

Unrolling drives intra-block parallelism

Pipelining drives intra-block concurrency

```
data_t MAC (
  data_t data_in[4],
  coef_t coef_in[4]
) {

  accu_t acc = 0 ;

  for (int i=0;i<4;i++) {
    acc += data_in[i] * coef_in[i] ;
  }
  return acc ;
}
```



SIEMENS

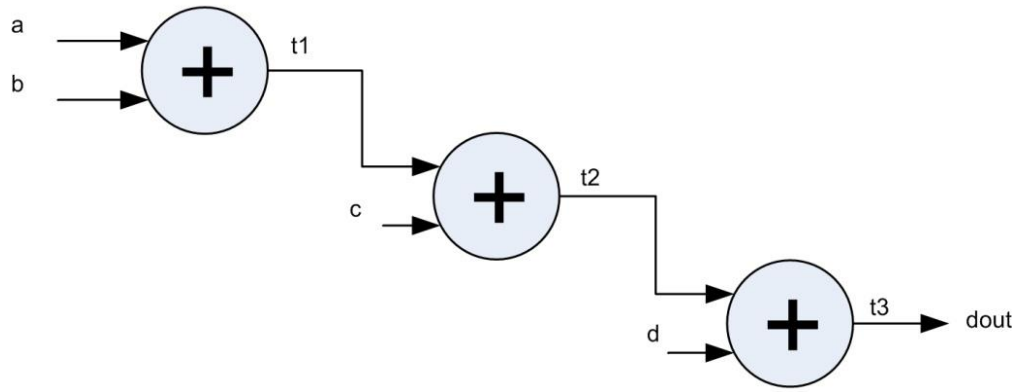
HLS Automatically Analyzes C++ Designs

High-level synthesis analyzes the data dependencies between the various steps in the algorithm

Analysis leads to a Data Flow Graph (DFG) description

- Each node of the DFG represents an operation defined in the C++ code
  - for this example all operations use the "add" operator
- Connections between nodes represents data dependencies and indicates the order of operations

```
void accumulate(int a, int b,
                int c, int d,
                int &dout){
   int t1,t2;
   t1   = a  + b;
   t2   = t1 + c;
   dout = t2 + d;
}
```



SIEMENS

HLS Performs Technology-Aware Optimizations

After DFG analysis each operation is mapped onto a hardware resource which is then used during scheduling

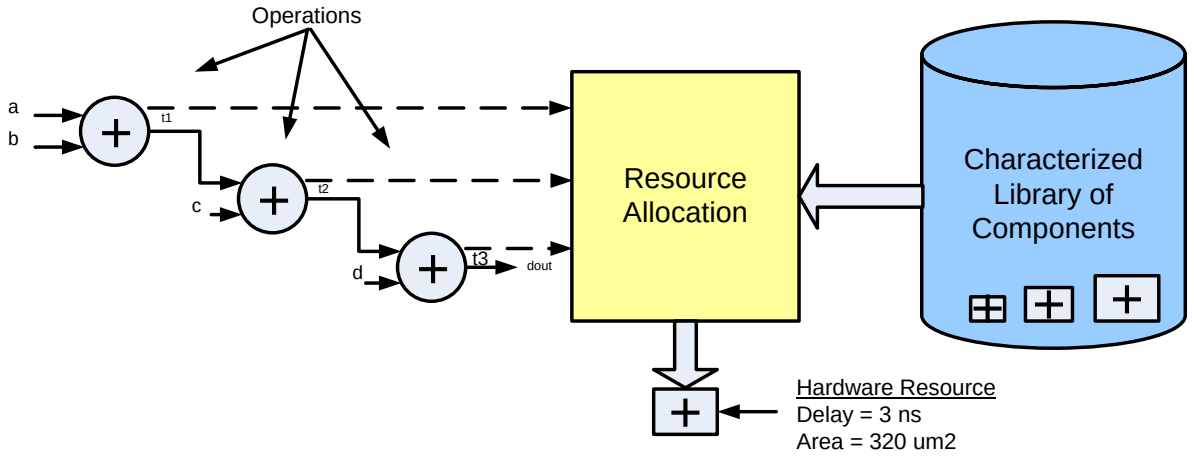
- This process is know as “Resource Allocation”

Resources are selected from a technology specific library

- E.g. TSMC 16nm, Altera Stratix10, etc

Resources correspond to a physical implementation of the operator hardware

- Implementation is annotated with both timing and area information which is used during scheduling
- Operators may have multiple hardware resource implementations that each have different area/delay/latency trade-offs



HLS Automatically Closes Timing

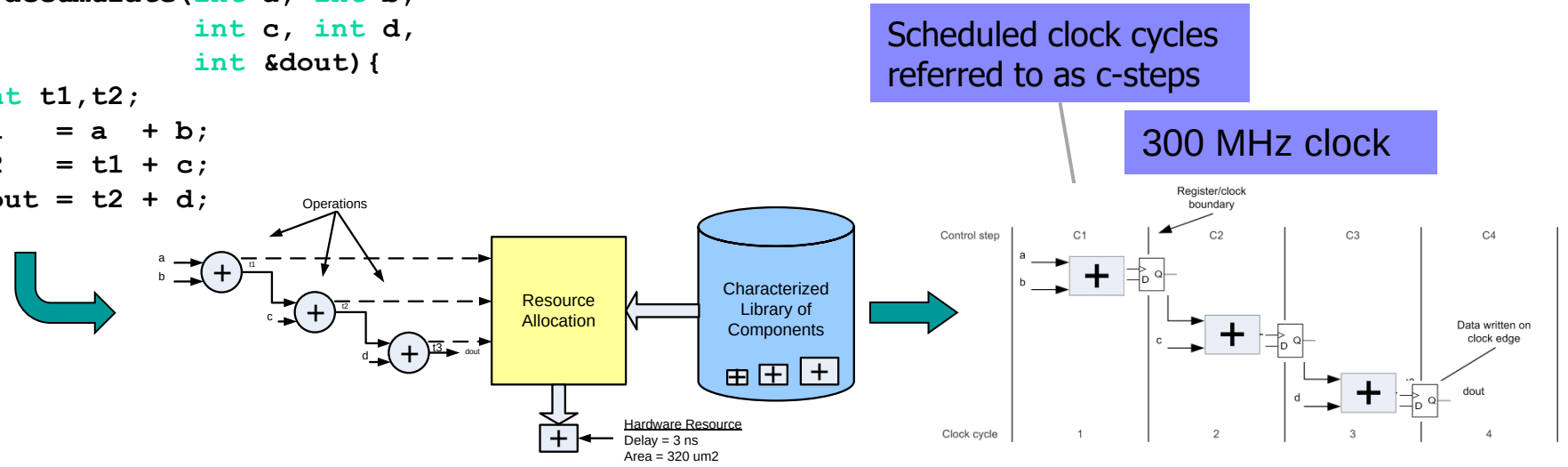
High-level synthesis adds "time" to the design during the process known as "scheduling“

Scheduling automatically shares resources

Scheduling takes the operations described in the DFG and decides when (in which clock cycle) they are performed

- Has the effect of adding registers when needed to meet timing
- Similar to what RTL designers would call pipelining, by which they mean inserting registers to reduce combinational delays

```
void accumulate(int a, int b,
                int c, int d,
                int &dout){
   int t1,t2;
   t1   = a  + b;
   t2   = t1 + c;
   dout = t2 + d;
}
```



SIEMENS

HLS Automatically Shares Resources

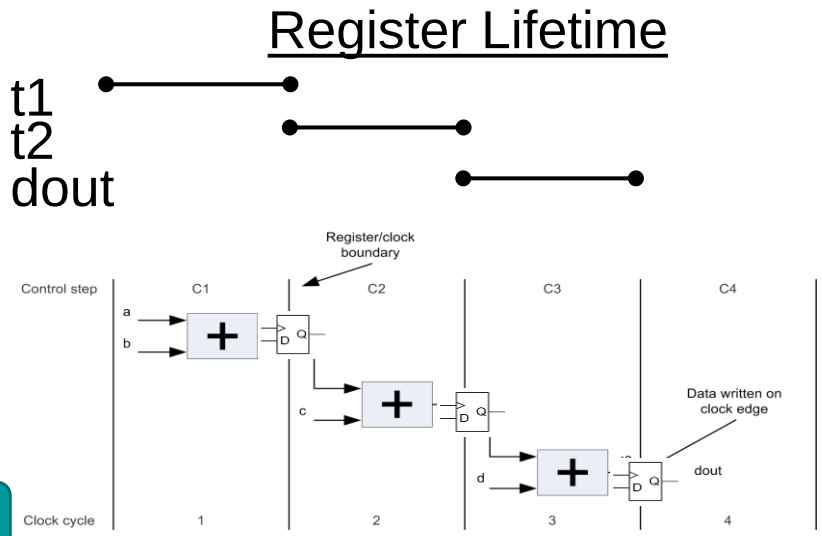
HLS shares resources

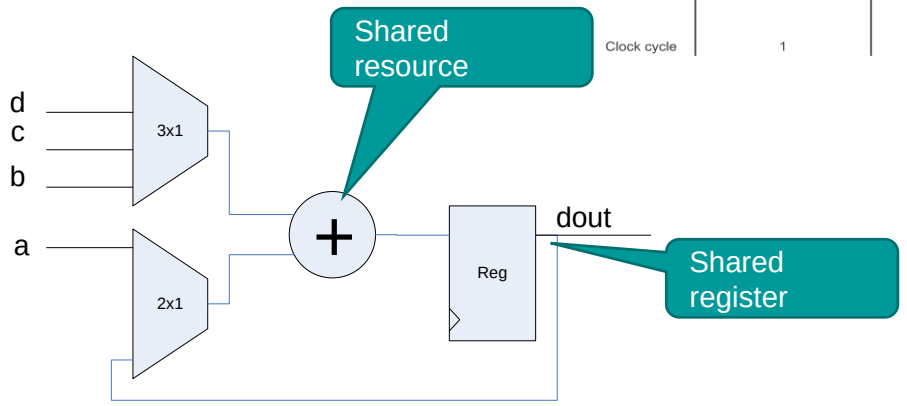
- Explicit mutual exclusivity in the code
- Via design constraints and automated analysis

HLS shares registers

- Via lifetime analysis and user constraints

```
void accumulate(int a, int b,
                int c, int d,
                int &dout){
   int t1,t2;
   t1   = a  + b;
   t2   = t1 + c;
   dout = t2 + d;
}
```





SIEMENS

HLS Builds Parallel/Concurrent Processes from Sequential C++ Classes

Single-threaded design description

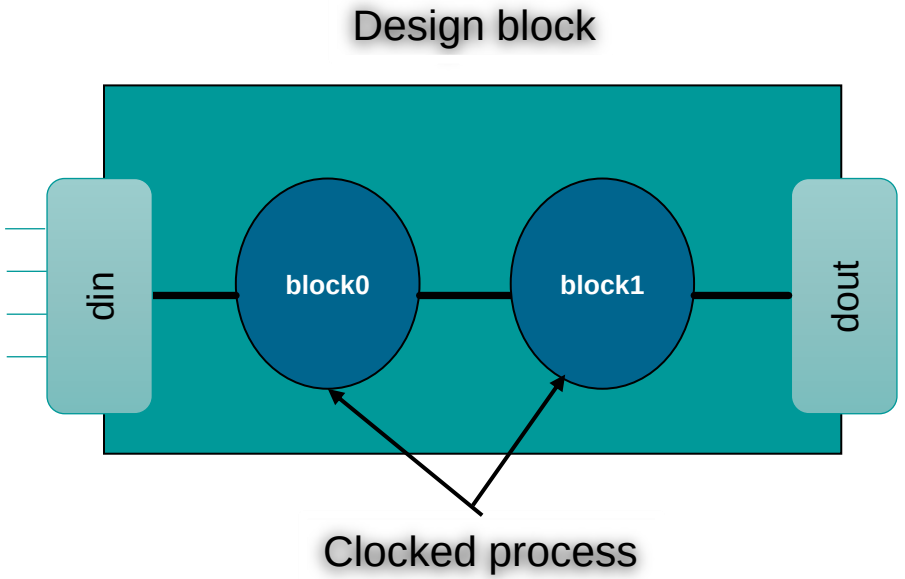
Multi-block Design

- Top-down or bottom-up

Design-blocks/processes run in parallel/concurrently

- Inter-block parallelism and concurrency

```
class simple_design{
  classA block0;
  classB block1;
  ac_channel<int> connect;
public:
  #pragma hls_design interface
  void run(ac_channel<int> &din, ac_channel<int> &dout){
    block0.run(din,connect);
    block1.run(connect,dout);
  }
};
```



SIEMENS

HLS Generates Synthesizable RTL from C++

C++ creates RTL modules with one or more sequential or combinational processes

C++ Combinational process created from design constraints

C++ HLS Input

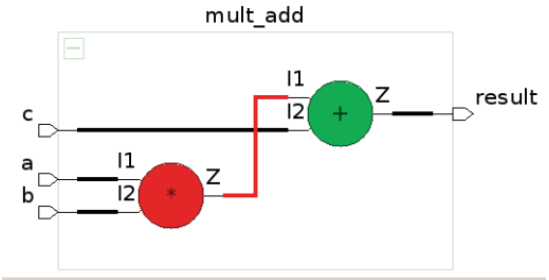
```
1 #pragma hls_design top
2 class mult_add{
3   public:
4   mult_add(){}
5 #pragma hls_design interface
6   void run(unsigned short  a,
7            unsigned short  b,
8            unsigned short  c,
9            unsigned int &result){
10    result = a*b + c;
11  }
12 };
```

HLS RTL Output

```
16 module mult_add (
17   a, b, c, result
18 );
19   input [15:0] a;
20   input [15:0] b;
21   input [15:0] c;
22   output [31:0] result;
23   wire [32:0] nl_result;
24
25
26   wire[31:0] mul_nl;
27
28   // Interconnect Declarations for Component Instantiations
29   assign mul_nl = conv_s2u_32_32(a * b);
30   assign nl_result = mul_nl + conv_u2s_16_32(c);
31   assign result = nl_result[31:0];
32
```

Combinational RTL

RTL Schematic



SIEMENS

Sequential Processes – C++

Clock and reset added by HLS

C++ HLS Input

```
1 #pragma hls_design top
2 class mult_add{
3   public:
4   mult_add(){}
5 #pragma hls_design interface
6   void run(unsigned short  a,
7            unsigned short  b,
8            unsigned short  c,
9            unsigned int &result){
10    result = a*b + c;
11  }
12 };
```

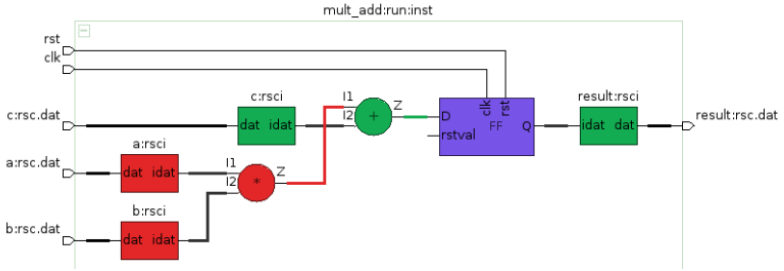
Same C++ source

HLS RTL Output

```
16 module mult_add(
17   clk, rst, a_rsc_dat, b_rsc_dat, c_rsc_dat, result_rsc_dat
18 );
19   input clk;
20   input rst;
21   input [15:0] a_rsc_dat;
22   input [15:0] b_rsc_dat;
23   input [15:0] c_rsc_dat;
24   output [31:0] result_rsc_dat;
25   reg [31:0] result_rsci_dat;
26
27   wire[31:0] mul_nl;
28
29   always @(posedge clk) begin
30     if ( rst ) begin
31       result_rsci_dat <= 32'b00000000000000000000000000000000;
32     end
33     else begin
34       result_rsci_dat <= nl_result_rsci_dat[31:0];
35     end
36   end
37   assign mul_nl = conv_s2u_32_32(a_rsci_dat * b_rsci_dat);
38   assign nl_result_rsci_dat  = mul_nl + conv_u2u_16_32(c_rsci_dat);
```

Sequential process

RTL Schematic

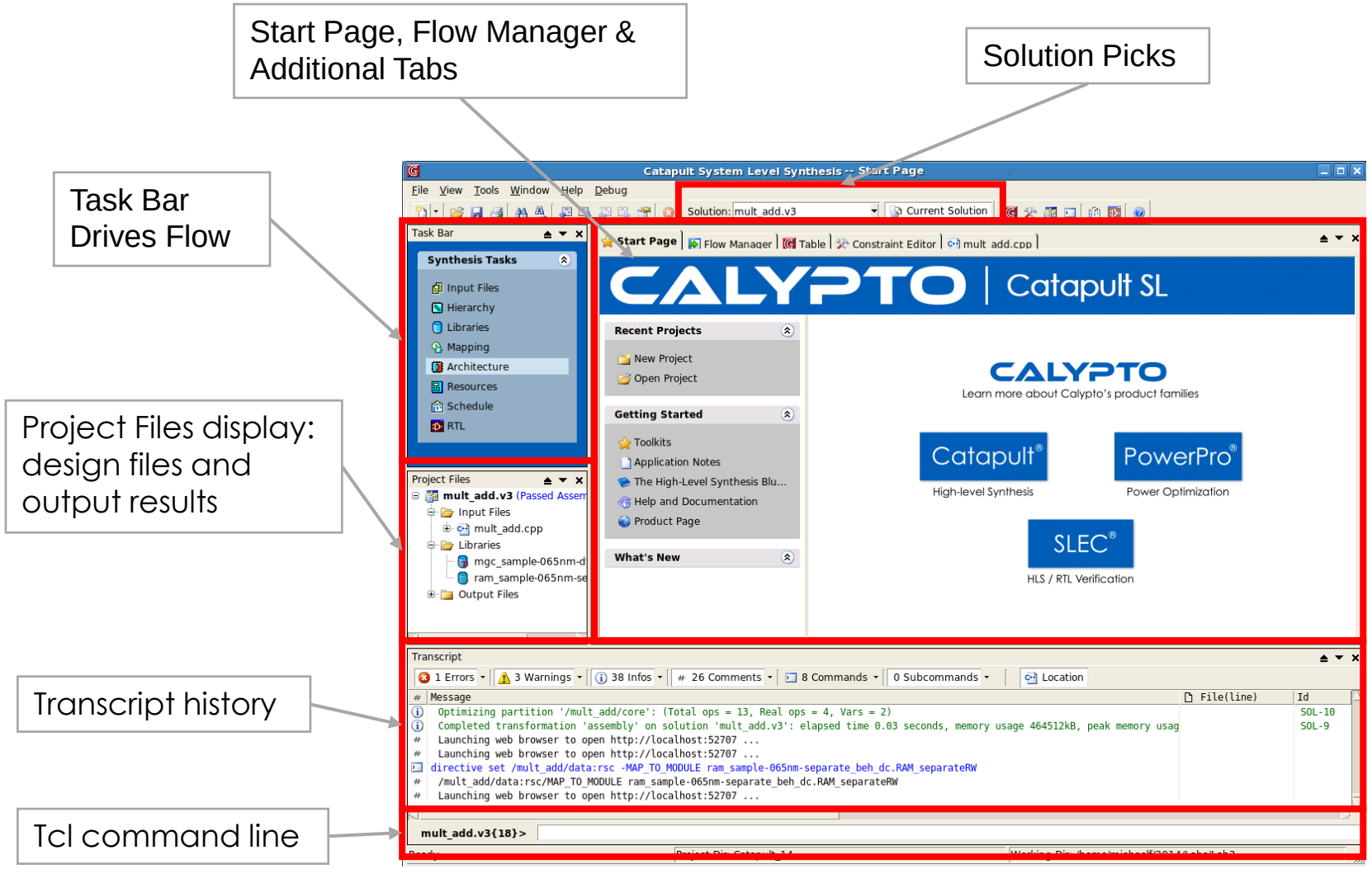


SIEMENS

Catapult C++ Flow and GUI

SIEMENS

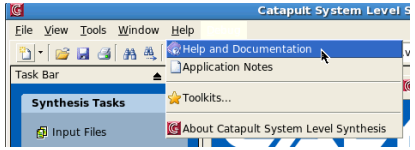
The Catapult C++ Flow

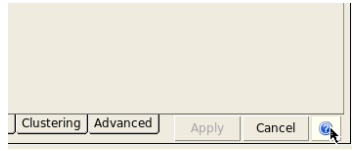


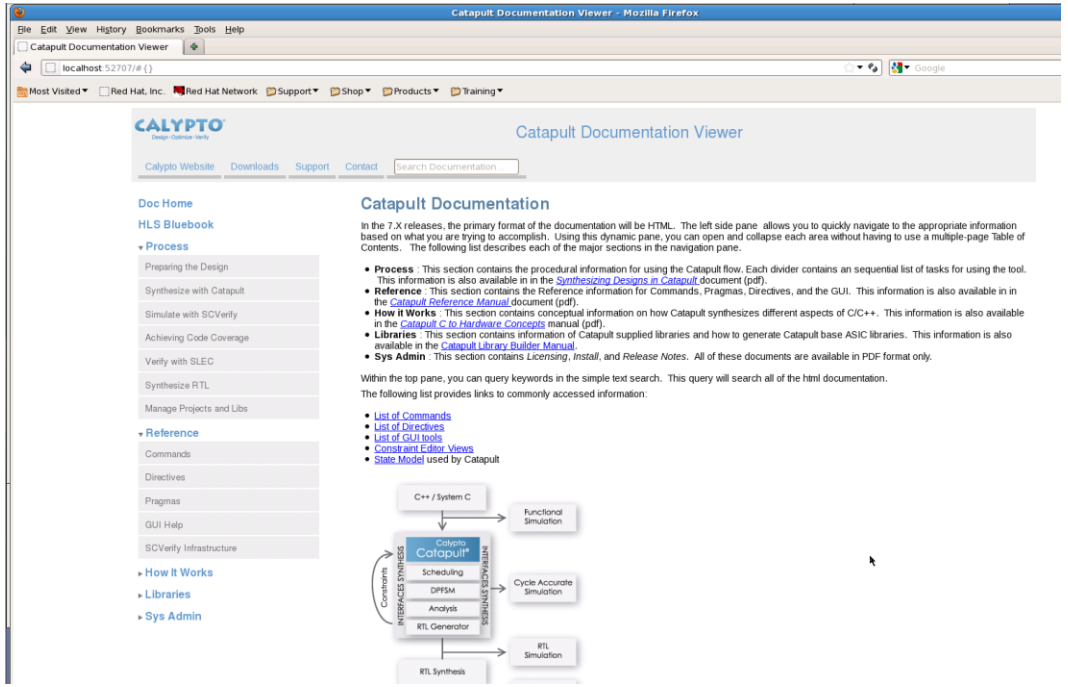
Catapult Help and Documentation

Catapult has extensive online documentation

- HTML format readable by any web browser
- Accessible via help menu or interactive help button







Docs can be found in the Catapult install tree

<Catapult install>/Mgc_home/shared/pdfdocs/pdf

Catapult C Full Task Bar Flow

Input Files

- Add Input Files

Hierarchy

- Define design modules (C++ only, explicit in SystemC)

Libraries

- Library management

Mapping

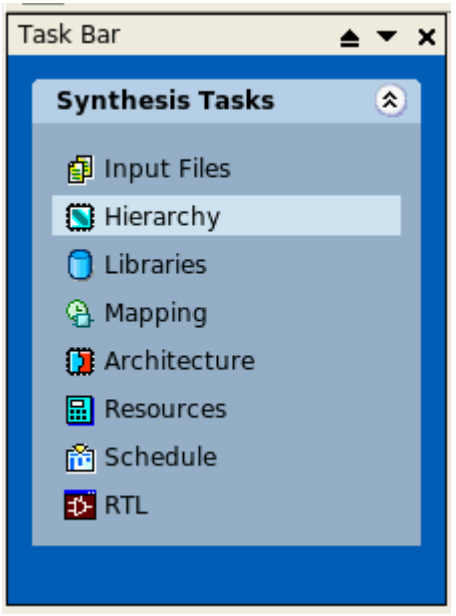
- Clock, reset and handshake settings (C++ only, explicit in SystemC)
- Bottom-up design modules

Constraints

- Architectural
- Resource

Schedule

Generate RTL



Adding Input Files

Add files needed for design

Files may be excluded if not intended for Synthesis

- Testbench files
- Helper files used as part of test infrastructure

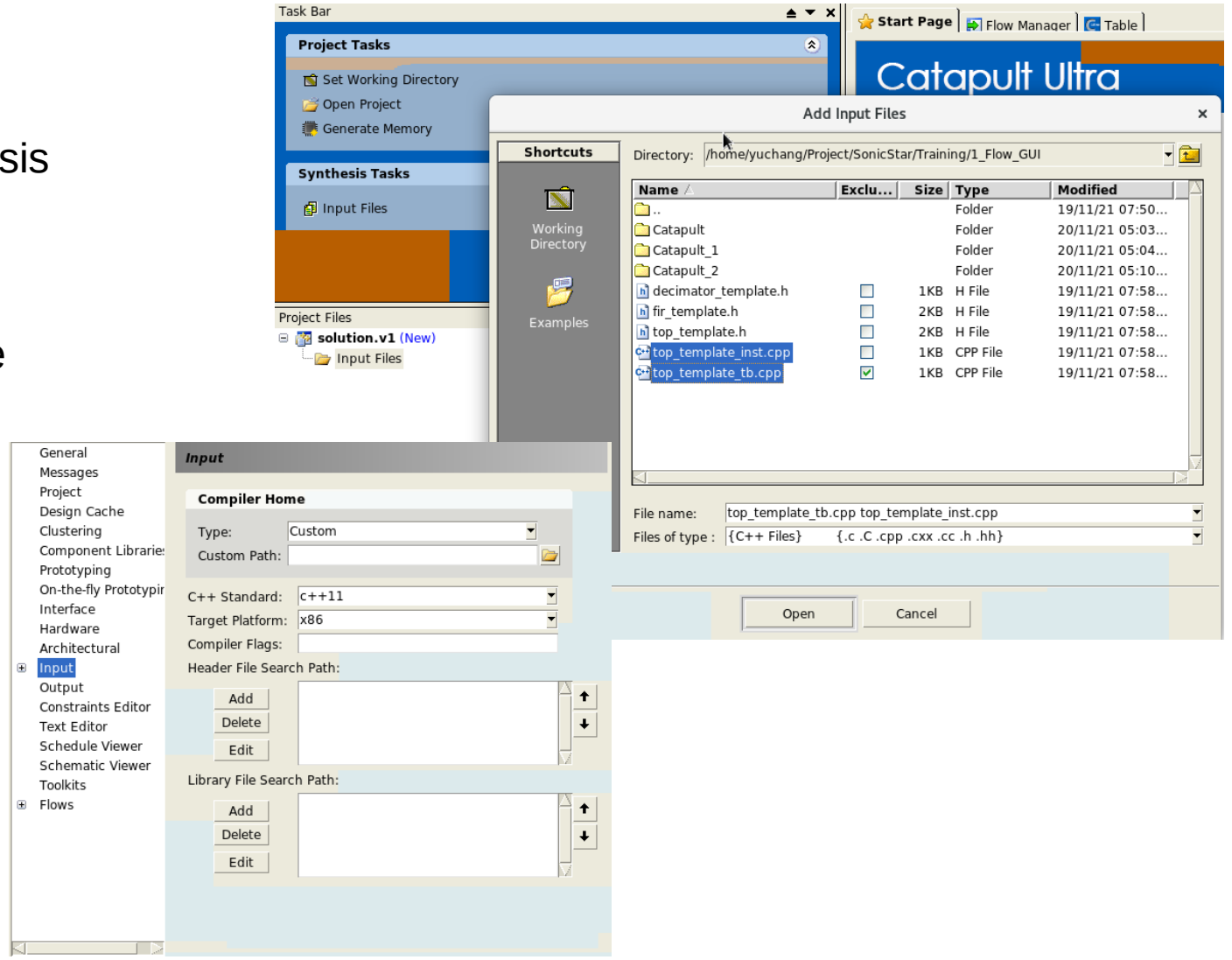
Do not add header files that are #include in code

Options may be set on files (right click)

- Exclude from build
- Options → Individual compiler flags

Menu: Tools → Set Options → Input

- Compiler flags
- Search paths
- Compiler Home directory

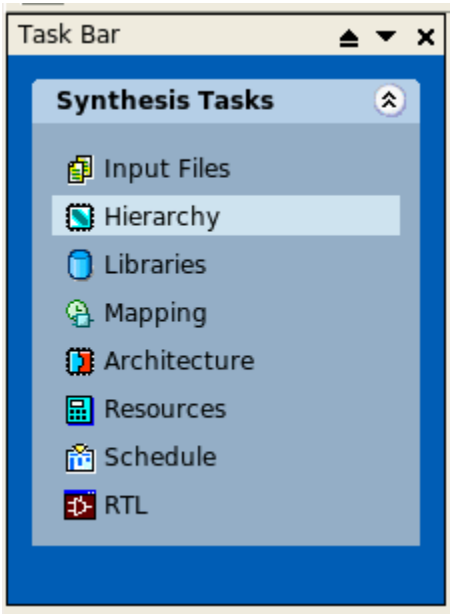


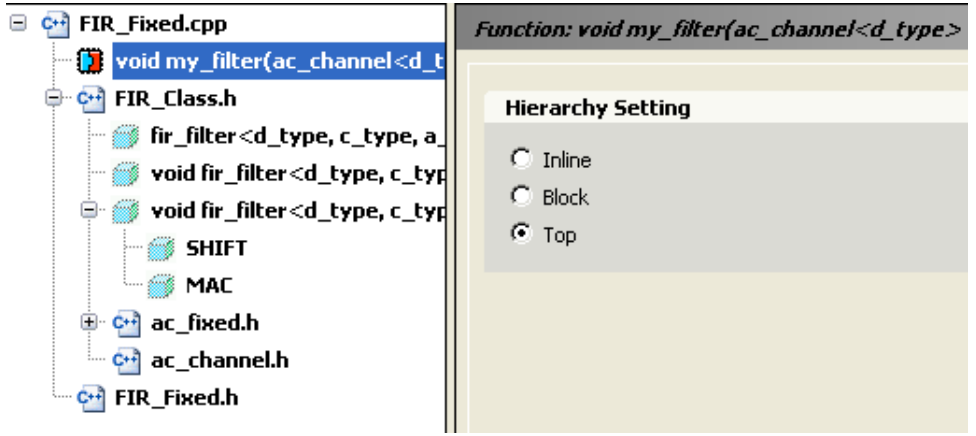
Setting the Top-level Design - Hierarchy

Select the top function of your design

- #pragma hls_design top can be added in the C++ code to define the top automatically
- Other functions will be inlined by default

Block indicates function should be created as a separate RTL module (Hierarchy)

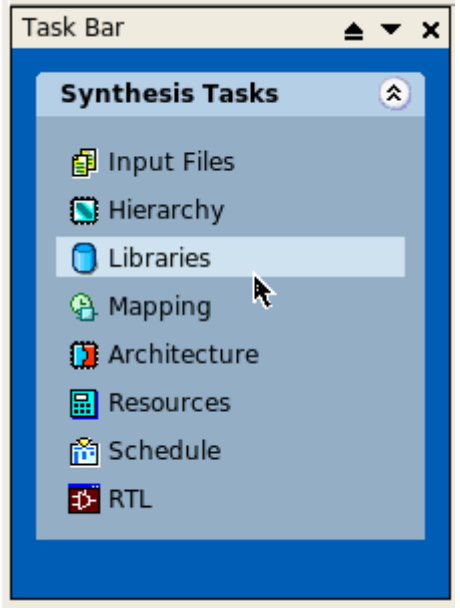


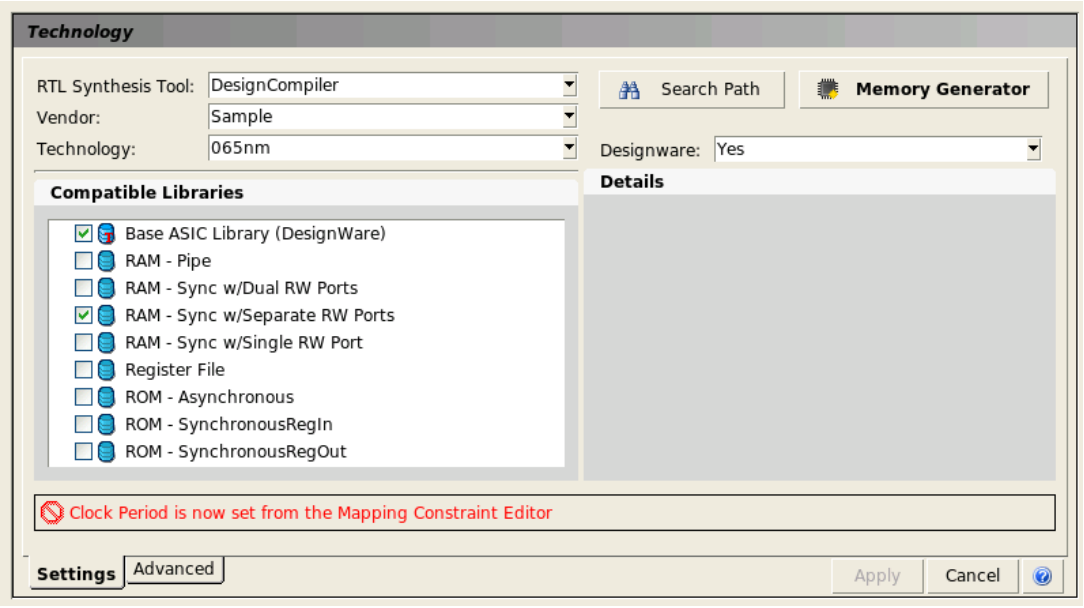


Selecting Technology - Libraries

Select Synthesis tool & Target Technology Library

- Plus Compatible libraries
  - RAM
  - ROM
  - Bottom-up blocks





Libraries

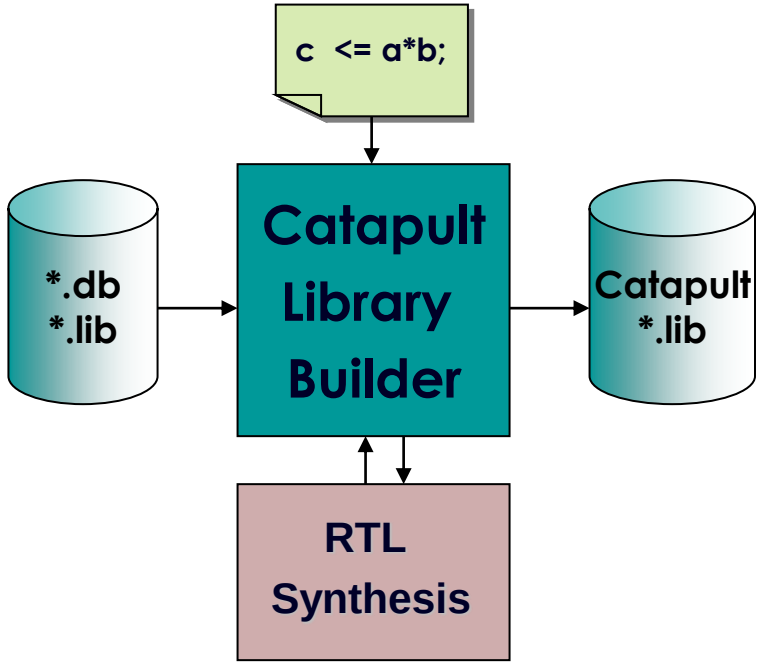
Technology defines the building blocks in your design

- Basic Blocks
  - Adders, Subtractors, Bitwise operators
- IP Blocks
  - RAMs, ROMs

Libraries are generated using Catapult C Library Builder

- Library Wizard makes this easy
- System supports DC, RC, etc
- ASIC Library is generated by user using their RTL synthesis tool
- FPGA technologies are pre-built

Best “Quality of Results”, Area, and Fmax correlation when using a correctly characterized technology library



Clocks and Resets - Mapping

Clock

- One clock created and assigned to all hierarchical blocks (default:“clk”)
- Separate hierarchical blocks can have separate clocks

Reset (per clock)

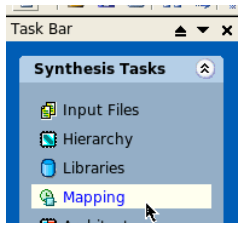
- Synchronous reset (default: rst)
- Asynchronous reset (default: arst_n)
- Supports both sync and async

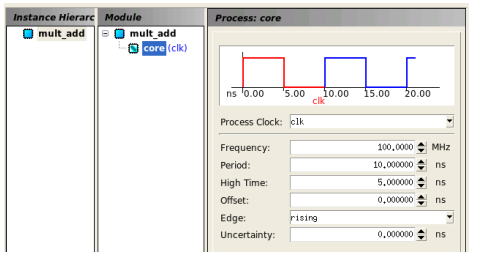
Enable (per clock)

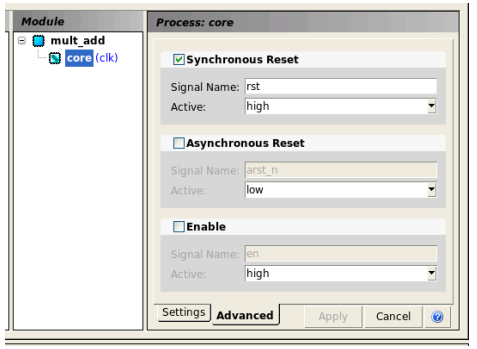
- Optional enable (default: en)

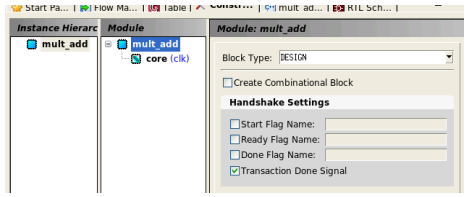
Transaction Done Signal (global)

- Used for RTL Verification flow only
- Generates a single one-bit flag for each Interface





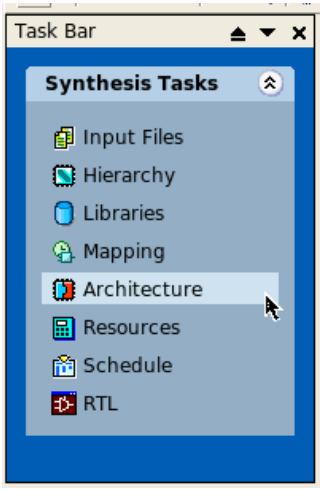




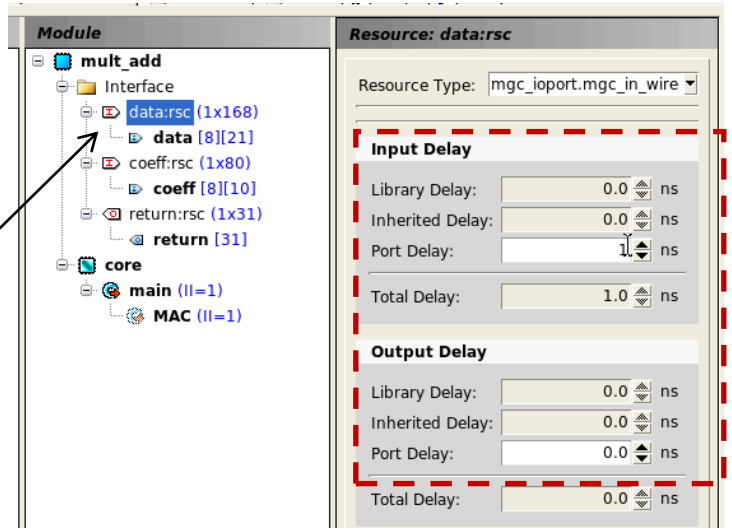
Input/Output Delay Constraints- Architecture

Architectural Constraints

- Port directions and bit-widths
- input_delay and output_delay constraints
  - Global and per port
- HLS constraints (Covered later)
  - Interface, loop, memory, etc




```
#pragma hls_design top
#include <ac_int.h>
void simple_function (ac_int<9,false> a[2],
                      ac_int<9,false> b[2],
                      ac_int<3,false> &offset,
                      ac_int<11>  result[2],
                      ac_int<3,false> *copy) {

  result[0] = a[0] + b[0] + offset;
  result[1] = a[1] + b[1] + offset;
  *copy = offset;
}
```



Port direction and width

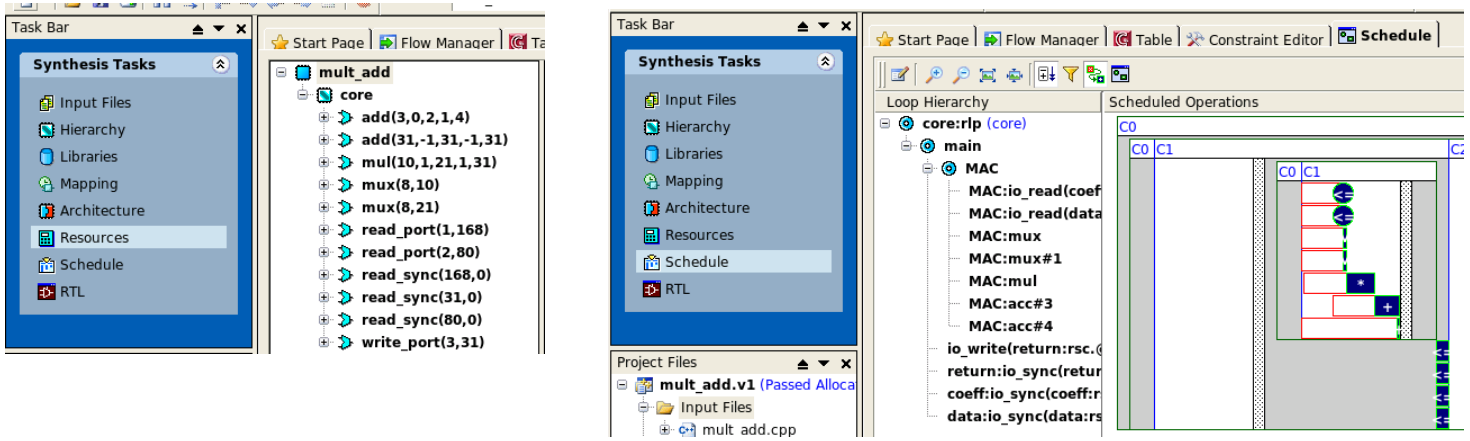
Resources and Schedule

Resource constraints provide direct control over the type and number of hardware resources

- Good for adding registers to components (pipelined multipliers)
- Not typically used for most designs

Schedule view

- Provides algorithmic/graphical view of synthesized design



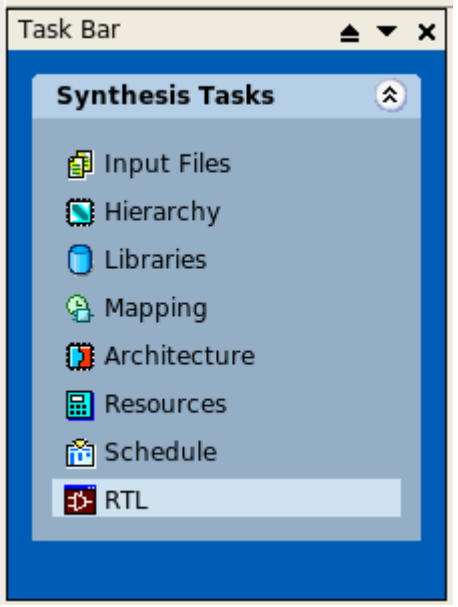
Generating RTL

Last stage of Synthesis

- Typically takes 50% of run-time, so only done when performance constraints are met

Constructs final RTL implementation

- FSM generation
- Timing analysis and sharing/replication
- Reporting – Area, Bill of Materials
- Schematics
- Downstream tool script generation

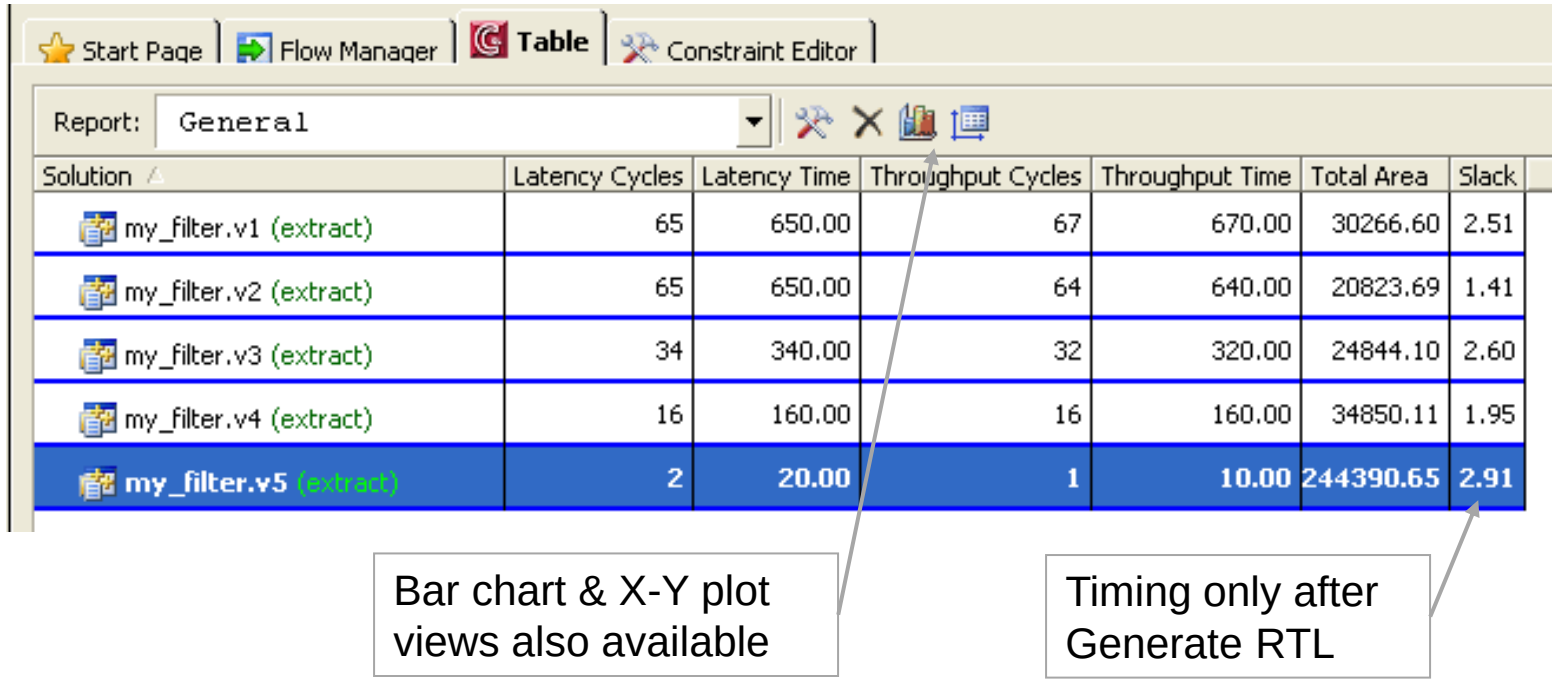


Comparing Solutions

Each different set of constraints will result in a new “solution”

Catapult places each solution in a different directory on disk

Multiple solutions are listed in the Table View



Output Files

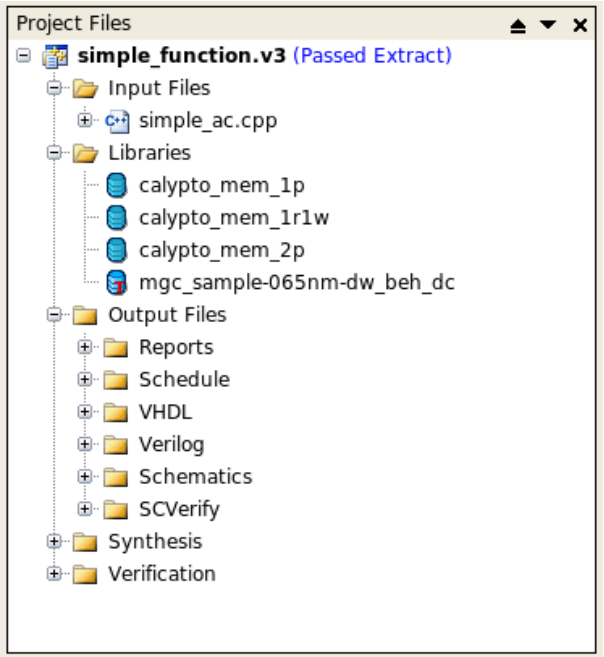
Consolidated for each solution directory

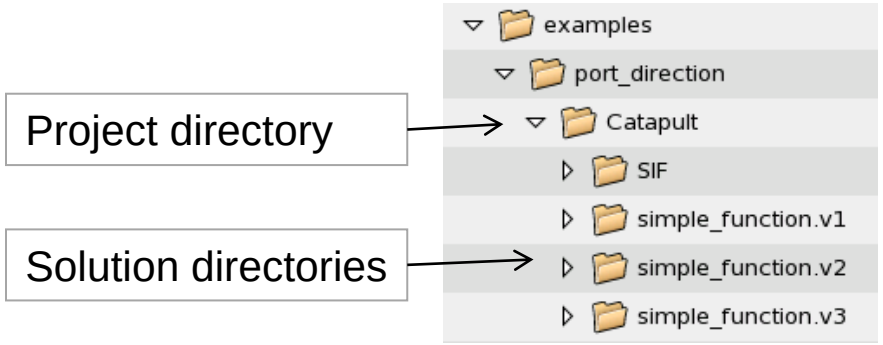
- Reports
- Schedule
- RTL
- Schematics

RTL Synthesis and downstream tool scripts

Simulation & Verification script generation

Output files written in solution directory under the project directory



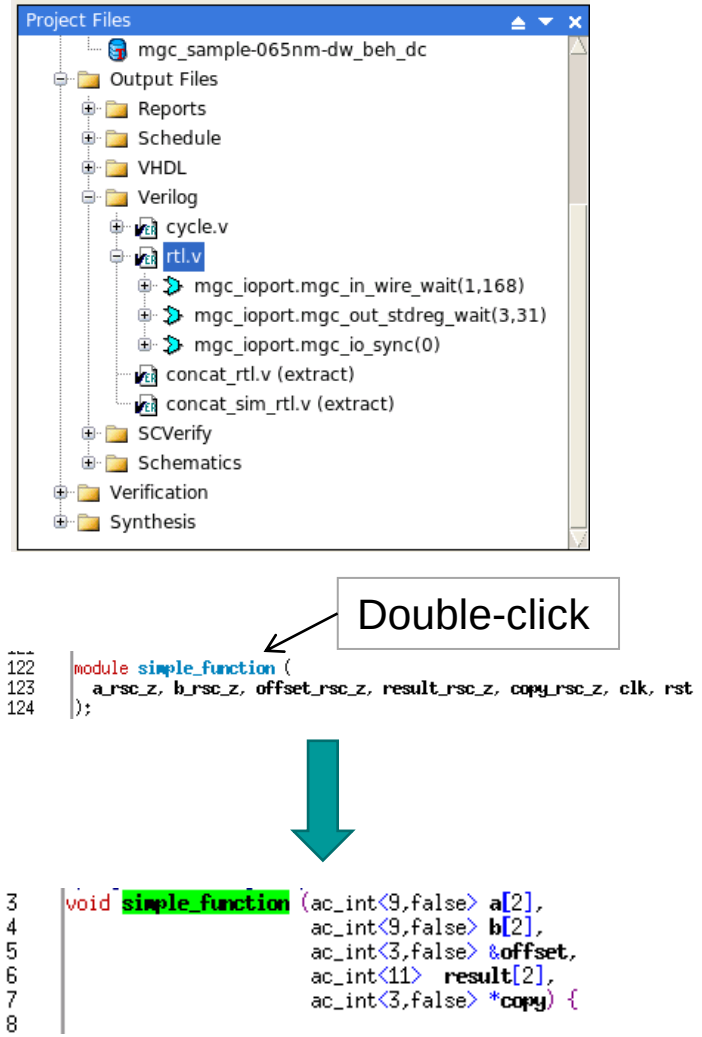


Verilog and VHDL

Catapult Generates Synthesizable Verilog and VHDL models

- Netlists are in the solution directory
- concat_rtl.v
  - Consolidated netlist in solution directory
  - Inlcudes all IO libraries

rtl.v netlist can be cross-probed back to C++



Reports

Commands

- TCL script file of commands required to reproduce the solution (“directives.tcl”)

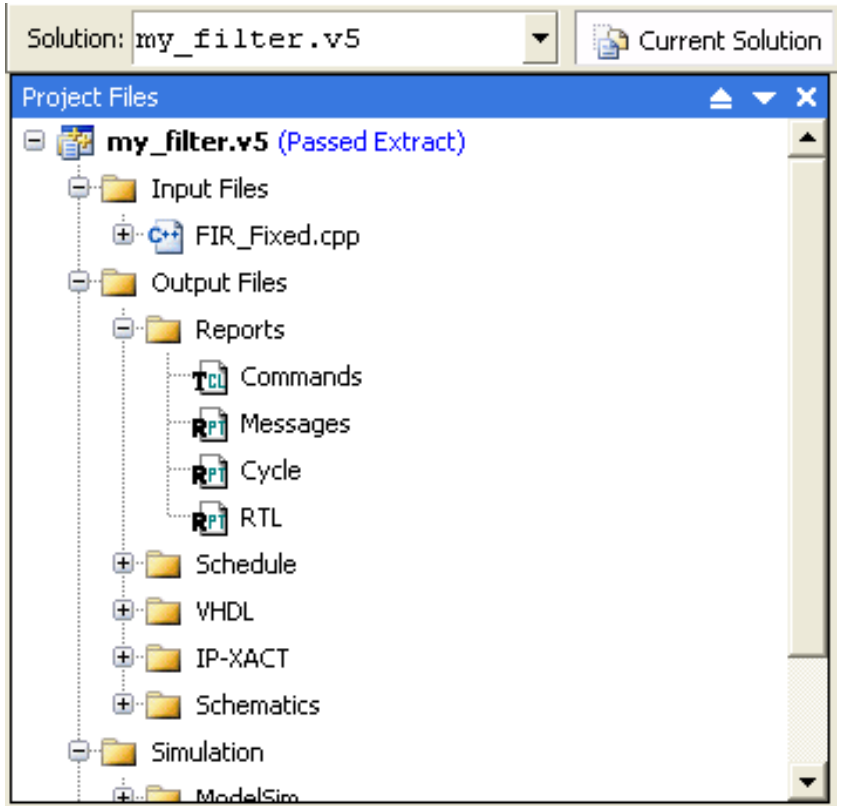
Messages from Catapult for the current solution

Cycle data report

- Clock info
- IO data ranges
- Memory resources
- Loop info

RTL report

- Most used
- Includes bill of materials & critical timing report



RTL report – Bill of Materials for ASIC

Components

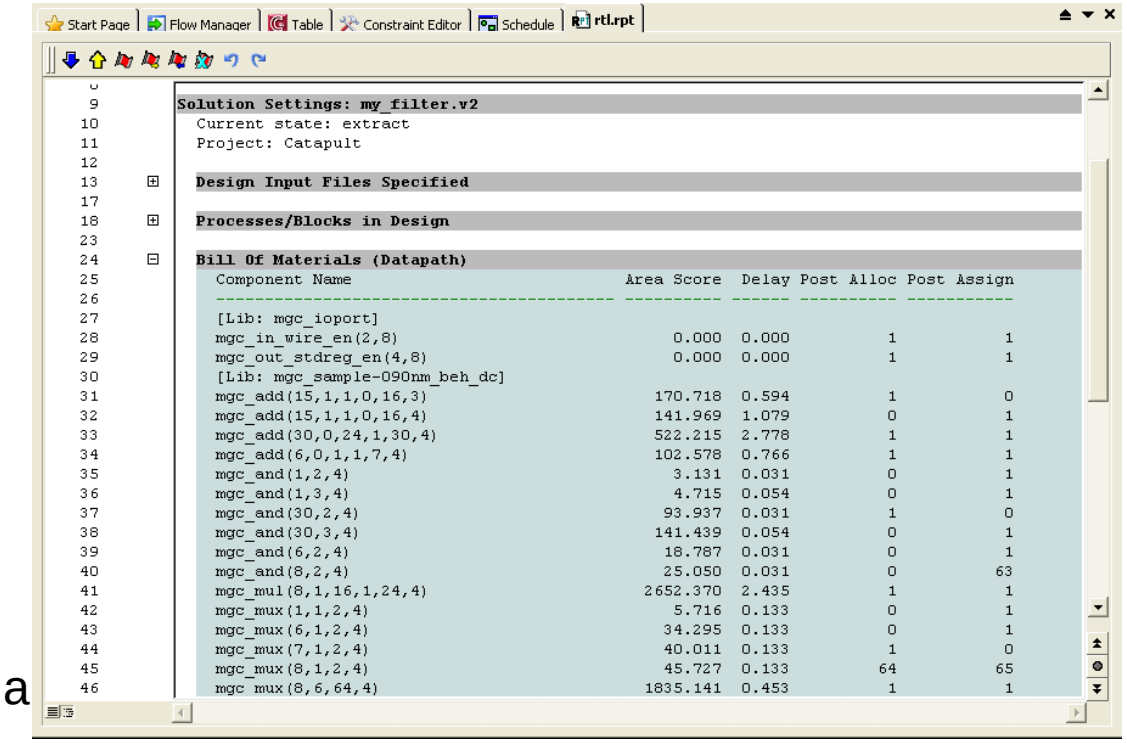
- Bit widths, Area, Delay, Quantity
- 4 component speed grades for ASIC
  - Last number in the list of parameters, e.g.
  - mgc_add(….,1) is a fast add
  - mgc_add(….,4) is a slow add

Alloc

- Number of components the scheduler thought it needed

Assign

- Number of actual hardware resources after timing analysis a resource sharing



RTL report – Bill of Materials for FPGA

Components

- Bit widths, Area, Delay, Quantity
- Catapult reports “area score” for DSP multipliers (Show in Table view)
  - Area score is used for resource sharing
- LUT area is available in BOM

Bill Of Materials (Datapath)

Component Name	Area Score	Area(DSP)	Area(LUTs)	Area(MUX_CARRYs)	Delay	Post Alloc	Post Assign
[Lib: ccs_ioport]							
ccs_in(1,16)	0.000	0.000	0.000	0.000	0.000	1	1
ccs_in(2,16)	0.000	0.000	0.000	0.000	0.000	1	1
ccs_in(3,16)	0.000	0.000	0.000	0.000	0.000	1	1
ccs_out(4,32)	0.000	0.000	0.000	0.000	0.000	1	1
[Lib: mgc_Xilinx-VIRTEX-7-1_beh]							
mgc_add(1,0,1,0,2)	1.000	0.000	1.000	0.000	0.990	0	1
mgc_add(3,0,1,0,3)	3.000	0.000	3.000	2.000	1.020	0	1
mgc_add(32,0,16,1,32)	32.000	0.000	32.000	31.000	1.455	2	2
mgc_and(1,2)	1.000	0.000	1.000	0.000	0.560	0	25
mgc_mul(16,1,16,1,32)	607.000	1.000	0.000	0.000	5.348	6	6
mgc_mux(1,1,2)	1.000	0.000	1.000	0.000	0.090	0	5
mgc_mux(32,1,2)	32.000	0.000	32.000	0.000	0.090	0	3
mgc_muxlhot(32,6)	83.149	0.000	83.149	0.000	1.520	0	1
mgc_muxlhot(32,7)	95.437	0.000	95.437	0.000	1.520	0	1
mgc_nor(1,2)	1.000	0.000	1.000	0.000	0.560	0	2
mgc_not(1)	0.000	0.000	0.000	0.000	0.000	0	26
mgc_or(1,2)	1.000	0.000	1.000	0.000	0.560	0	9
mgc_or(1,6)	1.000	0.000	1.000	0.000	0.560	0	1
mgc_reg_pos(1,0,0,1,1,0,0)	0.000	0.000	0.000	0.000	0.390	0	15
mgc_reg_pos(1,0,0,1,1,1,1)	0.000	0.000	0.000	0.000	0.390	0	1
mgc_reg_pos(16,0,0,1,1,0,0)	0.000	0.000	0.000	0.000	0.390	0	6
mgc_reg_pos(16,0,0,1,1,1,1)	0.000	0.000	0.000	0.000	0.390	0	14
mgc_reg_pos(2,0,0,1,1,0,0)	0.000	0.000	0.000	0.000	0.390	0	6
mgc_reg_pos(32,0,0,1,1,0,0)	0.000	0.000	0.000	0.000	0.390	0	3
mgc_reg_pos(32,0,0,1,1,1,1)	0.000	0.000	0.000	0.000	0.390	0	2
mgc_xor(1,2)	1.000	0.000	1.000	0.000	0.560	0	1
TOTAL AREA (After Assignment):	4027.586	6.000	386.000	64.000			

Commands Report

Commands

- Consists of collection of all the default options, default commands and default directives
- Also consists of the commands from the specified TCL file for synthesizing the design

Usefulness of this report:

- Can verify the default commands, options and directives that are affecting the given design through either optimizations, parameters or characteristic values
- History of the commands specified in the TCL file

Messages Report

```
# Info: Starting transformation 'allocate' on solution 'dct.v1' (SOL-8)
...
# Info: Completed transformation 'allocate' on solution 'dct.v1':
elapsed time 0.61 seconds, memory usage 97204kB,
peak memory usage 121676kB (SOL-9)
```

```
# Info: Optimizing partition '/dct': (Total ops = 233, Real ops = 67, Vars = 196) (SOL-10)
# Info: Optimizing partition '/dct/dct:core': (Total ops = 199, Real ops = 67, Vars = 171) (SOL-10)
```

Messages

- Consists of various run-time informational messages such as design flow and optimization messages with different types of severity level.
- ASSERT-1(SIF Asserts), can only occur if the Catapult internal database gets into an invalid state.
- You can set the severity level of a message such as Error, warning, informational, comment; but it cannot be lower than its factory default level.
- The message tag determines which part of the design flow caused and the corresponding short description gives a brief report of the problem.

General Category	Message Tag	Description
Design Flow Messages	ALOC	Allocation
	ASG	Assignment - Component binding and Sharing
	ASM	Assemble CCOREs
	CRAAS	Concurrent resource allocation and scheduling
	CIN	C SIFgen
	CRD	C reading
	FSM	FSM Extraction + Reg Sharing
	HIER	Hierarchical
	LOOP	Loop
	MEM	Memory and Interface Mapping
	NET	Netlisting
	SCHD	Scheduling
Optimization Messages	OPT	Sequential Design Analysis + Other optimizations
Other Messages	ASSERT-1	Refer to ASSERT-1 Messages
	BASIC	Low level
	CNS	Constraint Management
	LIB	Library
	LIC	License
	NL	General Netlisting
	PRJ	Project
	READ	SIF Reading
	WRITE	SIF Writing
	SEQ	Sequential Component Analysis
	SIFG	SIF Gen (VHDL and Verilog)
	SOL	Solution
	VHDL	VHDL Netlisting
	VLOG	Verilog Netlisting

Usefulness of this report:

- It keeps you informed about potential problems that it detects, latency and area information for each optimization iteration
- Elapsed time, memory usage data about transformation.

Cycle Report

Cycle

- This report consists of different processes/ blocks in the given design and their characteristics such as latency, throughput, reset length and pipeline.
- Clock information such as Edge, Time period, % of resource sharing allocation, clock uncertainty
- I/O ports with characteristics such as kind of data type, data width.
- List of memory resources used in the design with their characteristics such as memory component assigned, if it is external memory, type of packing mode, memory size, it’s offset index from the base address.
- Multi-cycle component usage.
- List of loops in the design with their characteristics such as number of C-steps, Total cycles consumed, duration in ‘ns’, iterations, unroll, pipeline.

Usefulness of this report:

- It informs about overall aspects of the design and it’s results based on default or applied constraints on variables.

RTL Report – Critical Path Timing

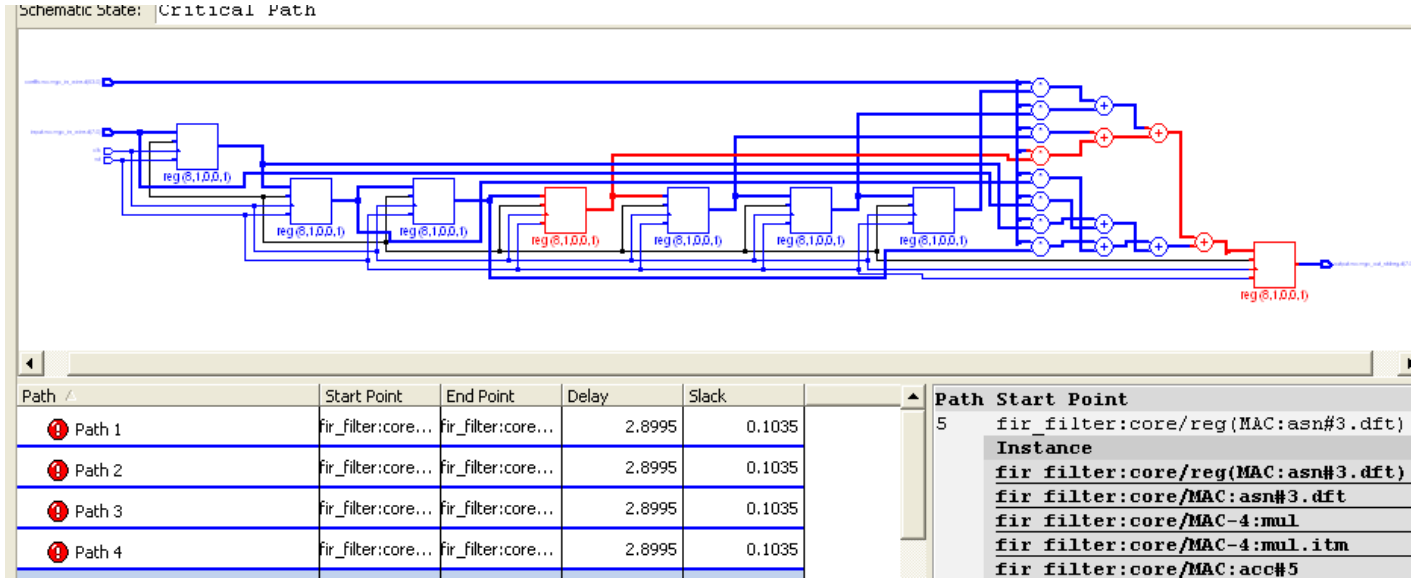
Critical path report

Can be viewed in schematic

Estimated timing

- Negative slack can usually be ignored
- Run RTL synthesis for final timing

```
Register-to-Variable Mappings

Timing Report
  Critical Path
    Max Delay:  2.89947402
    Slack:      0.1035289800000001

    Path                                                  Startpoint
    ----------------------------------------------------- ----------
    1                                                     fir_filter:core/reg(MAC

      Instance                                            Component
      --------                                            ---------
      fir_filter:core/reg(MAC:asn.dft)                    mgc_reg_pos_8_0_0_1_1_0
      fir_filter:core/MAC:asn.dft
      fir_filter:core/MAC-1:mul                           mgc_mul_8_1_8_1_15_4
      fir_filter:core/MAC-1:mul.itm
```



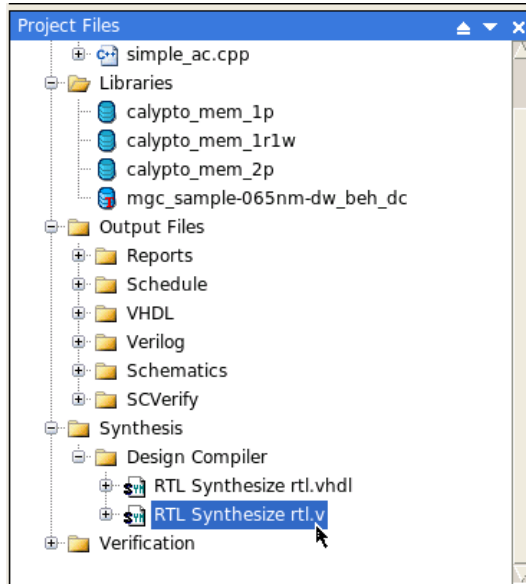
Synthesizing the RTL

Catapult allows users to launch RTL synthesis

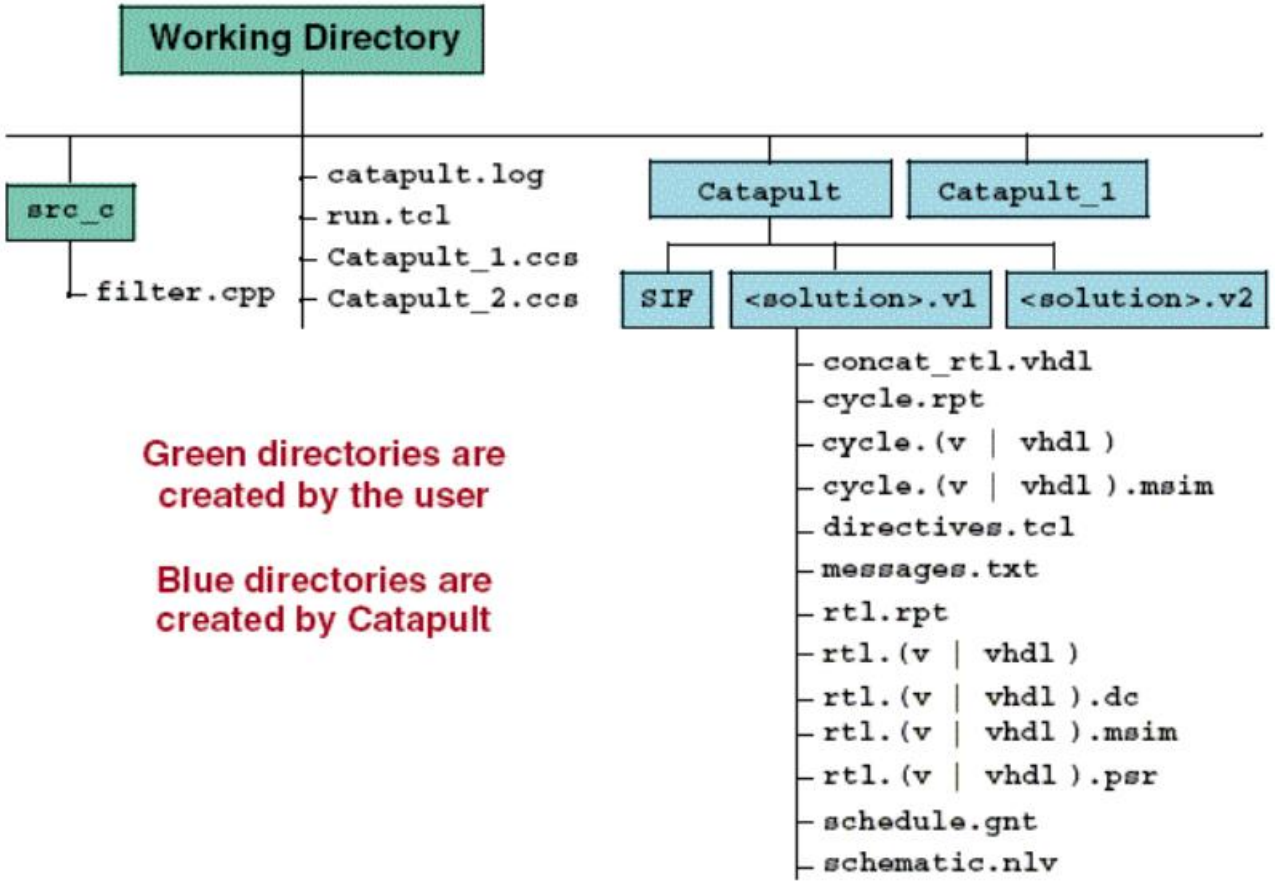
- Generates all synthesis scripts
- Back annotates area and timing into Catapult table view

Catapult needs to know where to find RTL synthesis tool executable and technology libraries

- Set under Tools > Set Options > Flows



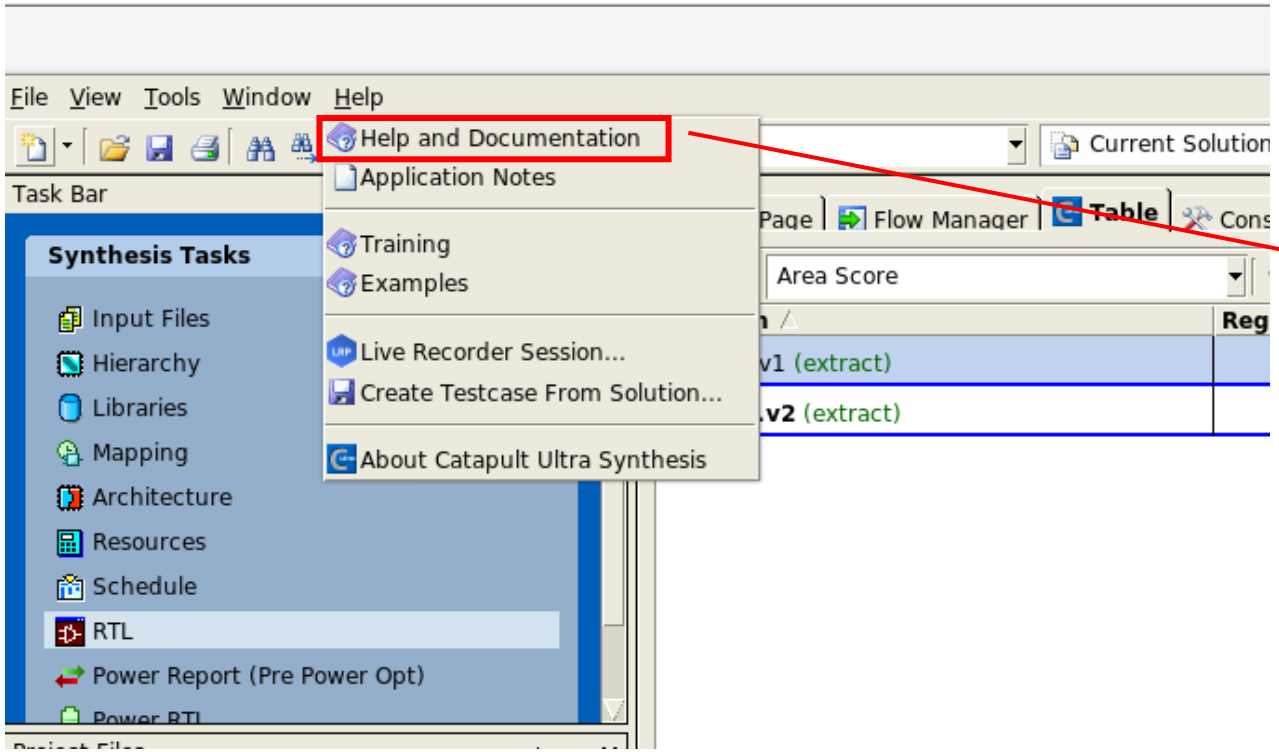
Working Directory

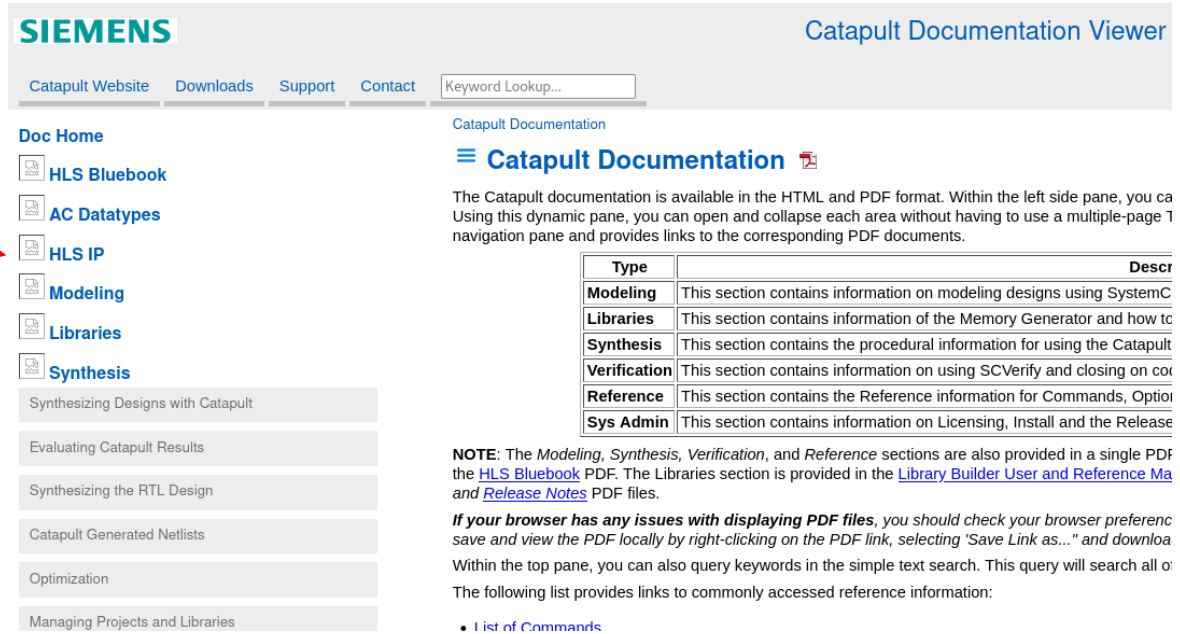


Green directories are created by the user

Blue directories are created by Catapult

Catapult Help and Documents

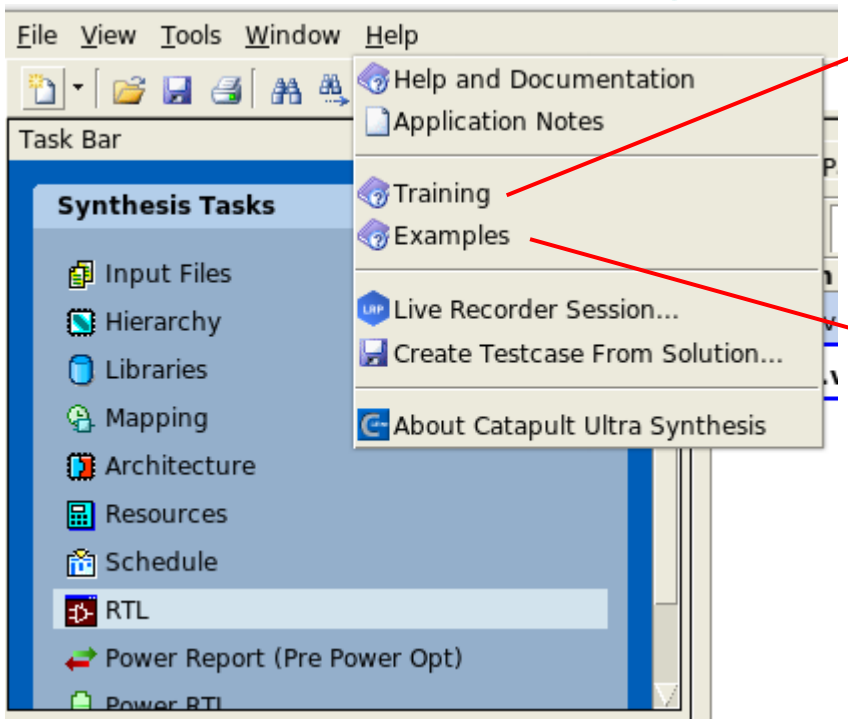


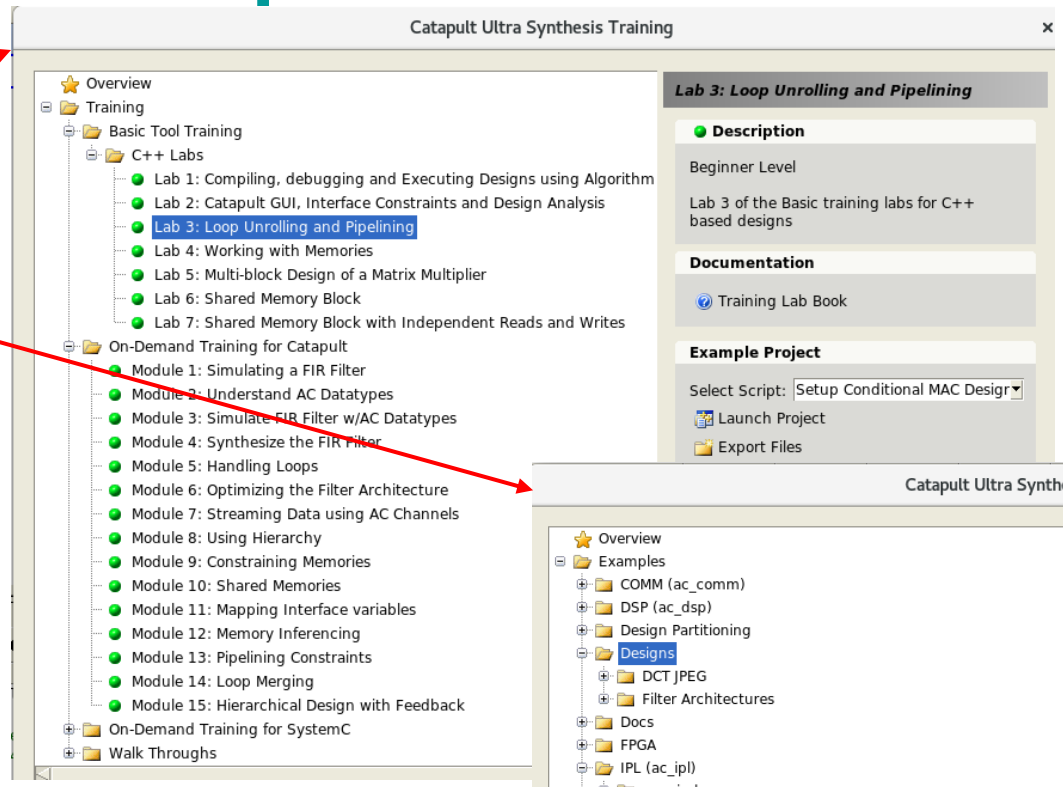


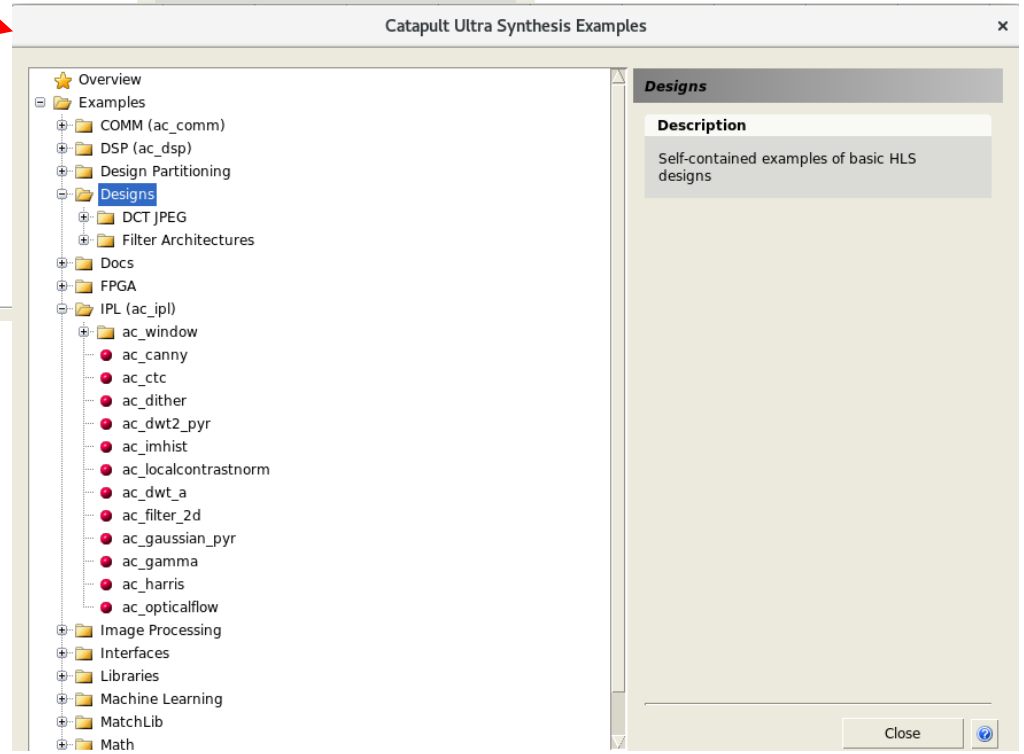

```
[mentor@RHEL74 walkthrough]$ ll $MGC_HOME/shared/pdfdocs/
ac_datatypes_ref.pdf              ac_ipl_relnotes.pdf          catapultFormal_tut.pdf       catapult_relnotes.pdf
ac_datatypes_relnotes.pdf         ac_math_ref.pdf              catapultFormal_user.pdf      catapult_useref.pdf
ac_dsp_ref.pdf                    ac_math_relnotes.pdf         catapult_install.pdf         connections-guide.pdf
ac_dsp_relnotes.pdf               ac_ml_relnotes.pdf           catapult_lb_useref.pdf       flexnet_lic_admin.pdf
ac_ipl_ref.pdf                    catapult_appnotes/           catapult_qs.pdf              hls_bluebook.pdf
```


SIEMENS

Catapult Training and Examples








```
[mentor@RHEL74 walkthrough]$ ll $MGC_HOME/shared/
etc/      examples/ include/  legal/    lib/      pdfdocs/  pkgs/     registry/ systest/  training/
```


Bit-accurate Data Types

Synthesis of Bit-Accurate Data-Types

Needed to model true hardware behaviour

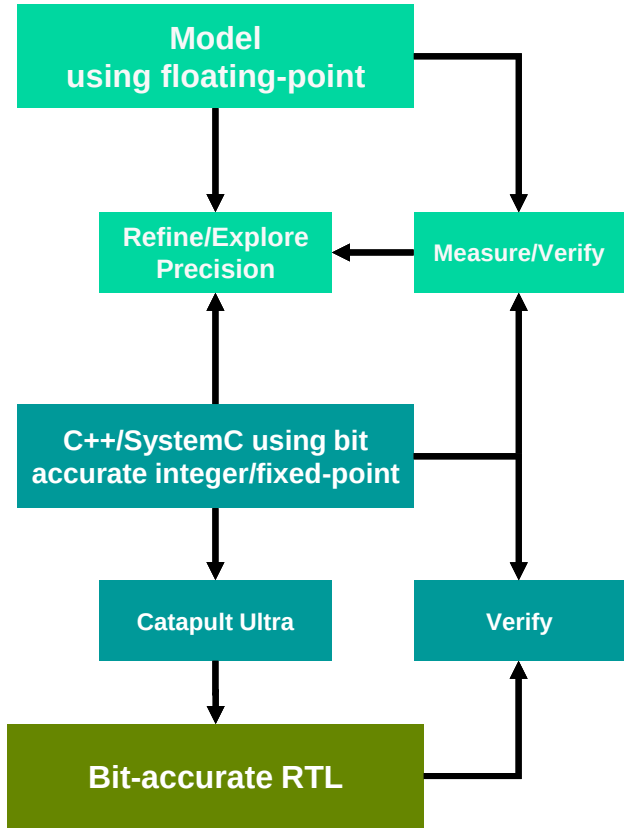
- Bit-accuracy simulated in source
- Provides a path for automated bit-for-bit comparison of C++ and RTL

HLS uses exact bit-widths to meet specification and save power/area

- bit-widths are not always pow2 (1, 8, 16, 32, 64 bits)

Algorithmic C data types

- Arbitrary precision
- Integer support (ac_int)
- Fixed-point support (ac_fixed)
- Floating-point support (ac_float)



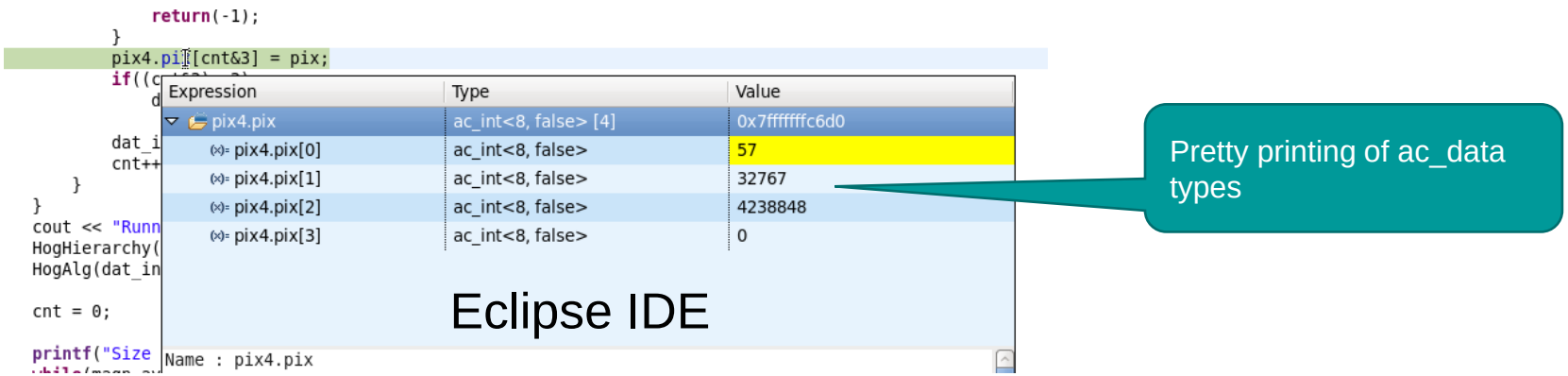
Compiling and Debugging Bit-accurate Data Types

Include Path

- <Catapult Install Tree>/Mgc_home/shared/include

Include files

- #include <ac_int.h>
- #include <ac_fixed.h>
- #include <ac_float.h>



Pretty printing of AC data types

- Support for GDB with Python
- MS Vscode and Eclipse CDT are the best open source Linux IDEs
- Need to add path in “~/.gdbinit” to pretty printing script
- Before gdb 9,2:
  - py execfile(“<Catapult Install Tree>/Mgc_home/shared/pkgs/ac_types/gdb/ac_pp.py")
- Gdb9.2 and later:
  - py exec(open(“<Catapult Install Tree>/Mgc_home/shared/pkgs/ac_types/gdb/ac_pp.py").read())

Integer Data Types

Allows designers to model a signed or unsigned bit vector with static bit precision

- Closely matches what RTL designers can do today with VHDL and Verilog 2001.
- Included using `#include <ac_int.h>`

Algorithmic C unsigned integer data types are declared as:

```
ac_int<W,S> var_name
```

where:

W = Bit width
S = Signedness (true == signed, false = unsigned)

```
ac_int<W,false> x;
```

where:

W = Bit width
0 <= x <= 2^W-1 by increments of 1

E.g.

```
//10 bits unsigned
ac_int<10,false> a;
```

```
ac_int<W,true> x;
```

where:

W = Bit width
-2^{W-1}<= x <= 2^{W-1}-1 by increments of 1

E.g.

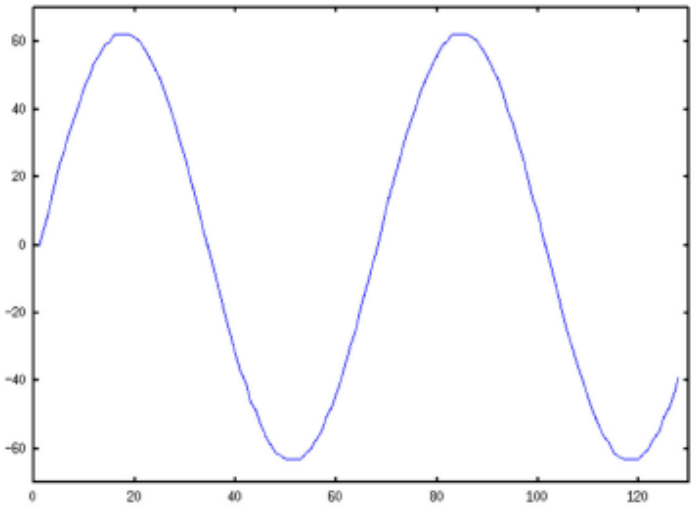
```
//10 bits signed
ac_int<10,true> a;
```

Wrapping Behavior

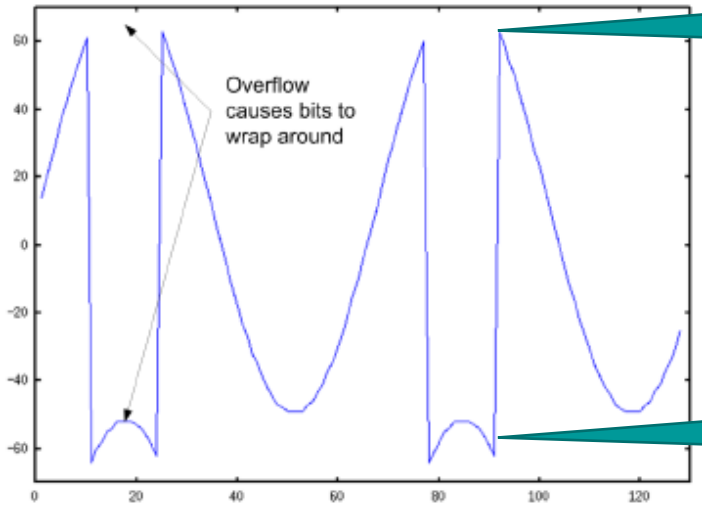
```
const double pi = 3.14;
int main(){
    fstream fptr;
    fptr.open("tmp.txt", fstream::out);
    ac_int<7,true> x[128];
    for(int i=0;i<128;i++){
        x[i] = OFFSET + 63*sin(2*pi*i/64);
        fptr << x[i] <<endl;
    }
    fptr.close();
}
```

7-bit signed: -64 < x < 63

OFFSET == 0



OFFSET == 14



Overflow

Underflow

Operator Precision

Arithmetic operators return the full bit precision data type

```
ac_int<8,false> a;
ac_int<8,false> b;
```

(a + b) returns a 9-bit type

(a * b) returns a 16-bit type

Assignment Behavior

Assignment is from LSB to MSB

```
ac_int<8,false> a = 255; // a =   “11111111”
ac_int<9,false> b = a;   // b =  “011111111”
```

- Signed data types are sign extended

```
ac_int<8,true> a = -1; // a =   “11111111”
ac_int<9,true> b = a;   // b =  “111111111”
```

Helper Functions for Static Computing of log2 for Minimum Number of Bits

Compute minimum number of bits required for array index and counters at compile time

- ac::log2_ceil computes array index bits
- ac::nbits computes counter bits

Array Index for array with 16 elements:

ac::log2_ceil<x>::val, where x is # of elements

e.g.

```
int array[16];
ac_int<ac::log2ceil<16>::val,false> arr_index;//4 bits
```

Counter that counts to 16:

ac::nbits<x>::val, where x is count value

e.g.

```
ac_int<ac::nbits<16>::val,false> count;//5 bits
```

Understanding Shift Behavior

Shifting returns the bit-width of the vector shifted

- Left shift will lose bits
- Behaves like a hardware shift register
- This is consistent irrespective of data type and length

```
ac_int<8,false> a = 255;   // a =  “11111111”
ac_int<9,false> b = a;     // b = “011111111”

ac_int<9,false> c = a << 1 ;// “11111111” << 1 gives “011111110” = 254
```

Extend the precision to keep upper bits

- Casting is ok

```
ac_int<9,false> c = b << 1 ;// “011111111” << 1 gives “111111110” = 510
ac_int<9,false> c = (ac_int<9,false>)a << 1;
```

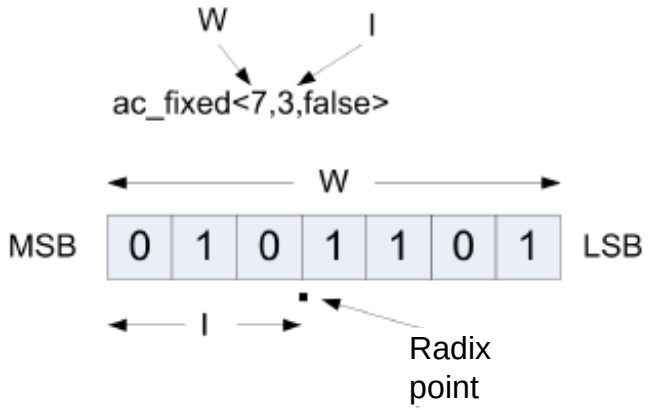
Algorithmic C Fixed-Point Data Types

Algorithmic C fixed point data types allow designers to model a signed or unsigned bit vector with static fixed point bit precision

- Allows designers to build hardware directly from a fixed point model
- ac_fixed data types are templatized, and allow designers to specify the integer & fractional width as well as the signedness of variables
- No need for shifting and masking
- Built-in support for saturation and rounding
- #include <ac_fixed.h>

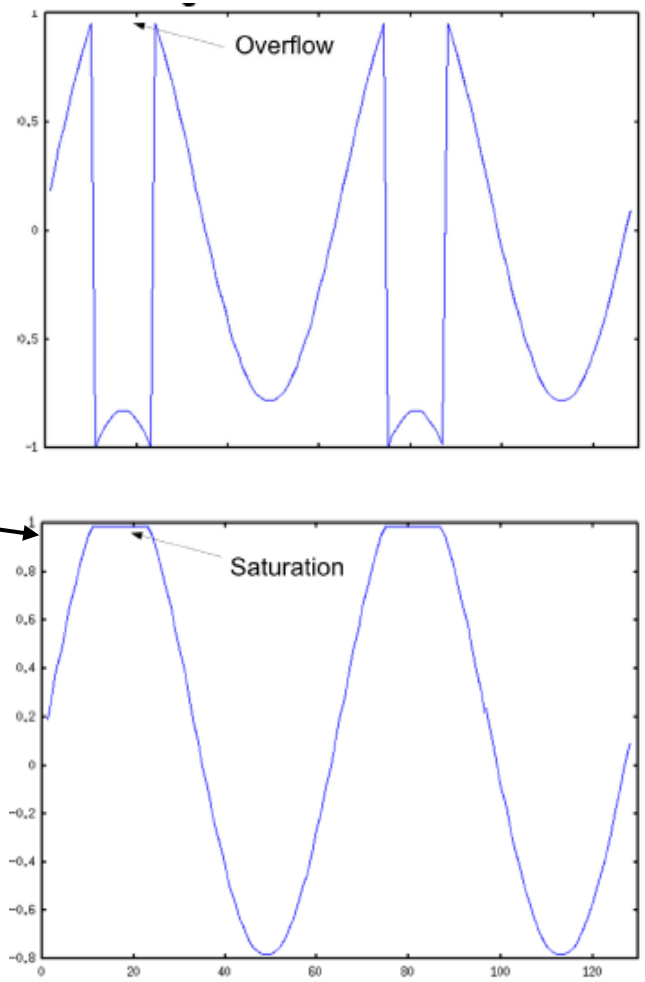
The Algorithmic C fixed point data types are declared as:

```
ac_fixed<W,I,S> x;
```



Saturation Behavior

```
#include <ac_fixed.h>
const double pi = 3.14;
const double OFFSET = 0.2;
int main(){
    fstream fptr;
    fptr.open("tmp.txt", fstream::out);
    ac_fixed<7,1,true,AC_TRN,AC_SAT> x[128];

    for(int i=0;i<128;i++){
        x[i] = OFFSET + 0.98*sin(2*pi*i/(double)64);
        fptr << x[i] <<endl;
    }
    fptr.close();
}
```



Rounding Behavior

Default behavior is truncate and to throw bits to the right of the LSB away

- Multiple rounding modes (See docs)
- Example … Total width of 7 bits, with 7 integer bits (0 fractional bits):

```
ac_fixed<7,7,true> x = 0.5;
cout << "result = " << x << endl;

> result = 0
```

- Adding “AC_RND” (round to infinity)

```
ac_fixed<7,7,true, AC_RND> x = 0.5;
cout << "result = " << x << endl;

> result = 1
```

Fixed-point Types – Important Modeling Notes

Built-in support for quantization(rounding) and overflow(saturation)

- See ac_datatypes manual
- ac_fixed<W,I,S,Q,O> x;
  - Q – rounding mode
  - O – saturation mode

Don’t globally turn on saturation and rounding

- Can give very bad QofR

Saturation and rounding happens on assignment “=“

- Move saturation outside of unrolled loops

```
ac_fixed<20,20> data[32];
…
ac_fixed<20,20,true,AC_RND,AC_SAT> acc;
acc = 0;
#pragma unroll yes
for(int i=0;i<32;i++){
  acc += data[i];//saturating here is bad
}
```

```
ac_fixed<20,20> data[32];
…
ac_fixed<25,25,true> acc;
ac_fixed<20,20,true,AC_RND,AC_SAT> sat;

acc = 0;
#pragma unroll yes
for(int i=0;i<32;i++){
  acc += data[i];
}
sat = acc;//Saturate and round on assignment
```

Bit Select Operator

Bit select is accomplished with the [ ] operator

Bit write

```
ac_int<3,false> bvec = 5; //3'b 101

bvec[1] = 1; //3'b111
```

Bit select can be used to test bool condition

Bit read

```
if(bvec[2])
  bvec[2] = 0;//3'b011
```

Read Slicing

slc method can be used on both ac_int and ac_fixed data types to read a group of bits

- Slice reads out of bounds are allowed. Out of bounds bits will be sign-extended for signed data, or zero padded for unsigned data (Note range is also now supported, see docs)

Method: slc<W>(int lsb), where W==number of bits, lsb is position of least significant bit

```
ac_int<5,false> bvec = 29; //5'b 11101

ac_int<3,false> a = bvec.slc<3>(0);//get lowest 3 bits

//a == 3'b101
```

Write Slicing

set_slc method can be used on both ac_int and ac_fixed data types to write a group of bits

- Out of bounds write slice will assert in simulation and synthesis

Method: set_slc(int lsb, const ac_int<W,S> &slc), where lsb is position of least significant bit and W is the number of bits, S is sign

```
ac_int<5,false> bvec = 29; //5'b 11101
ac_int<3,false> a = 0;//3'b000

bvec.set_slc(1,a);//set middle 3 bits

//b == 5'b10001
```

Printing Bit-accurate Data-types

cout stream operator “<<“ is implemented for ac_int, ac_fixed, ac_float

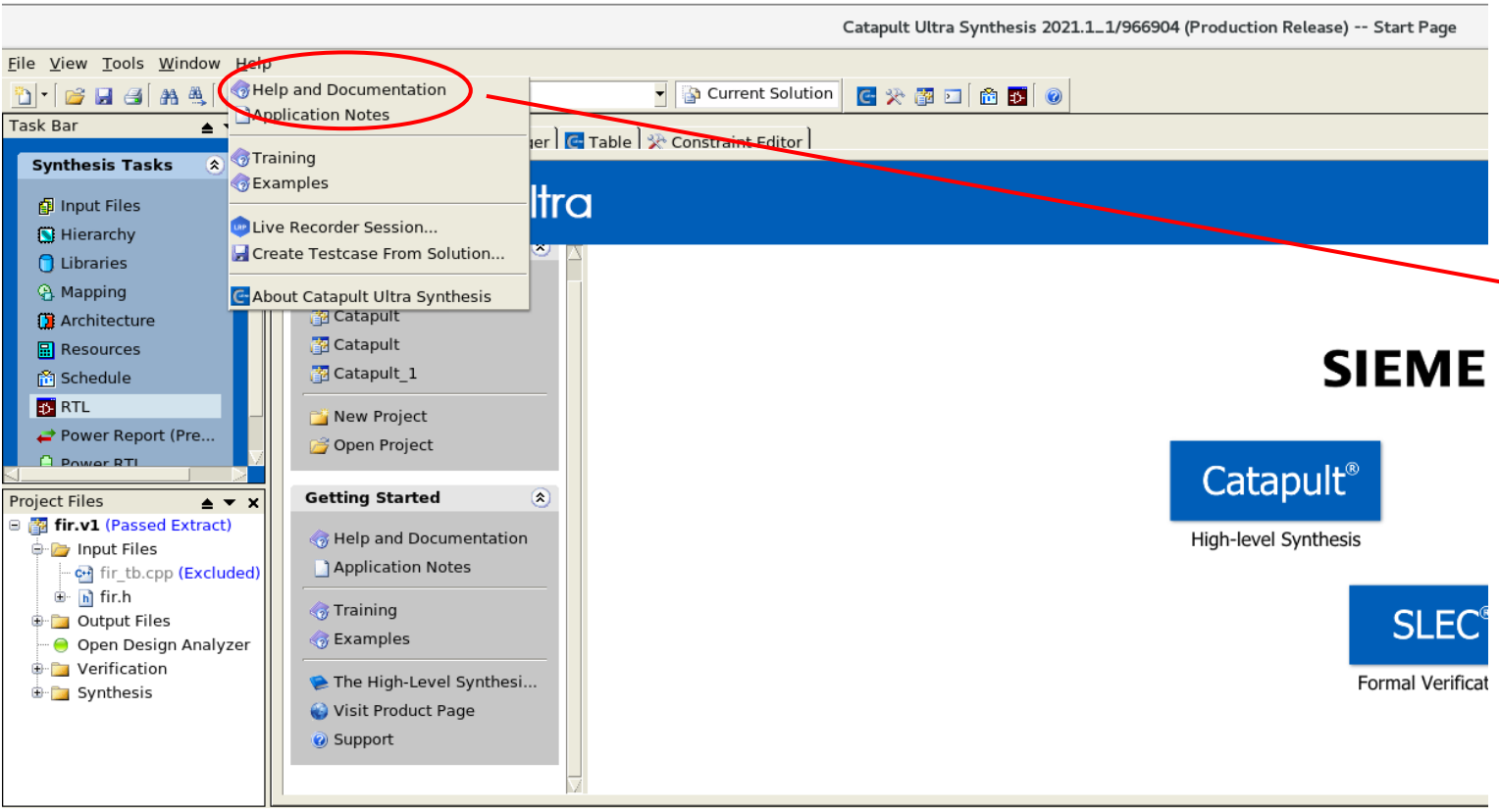
```
ac_int<10> data;
cout << data << endl;
```

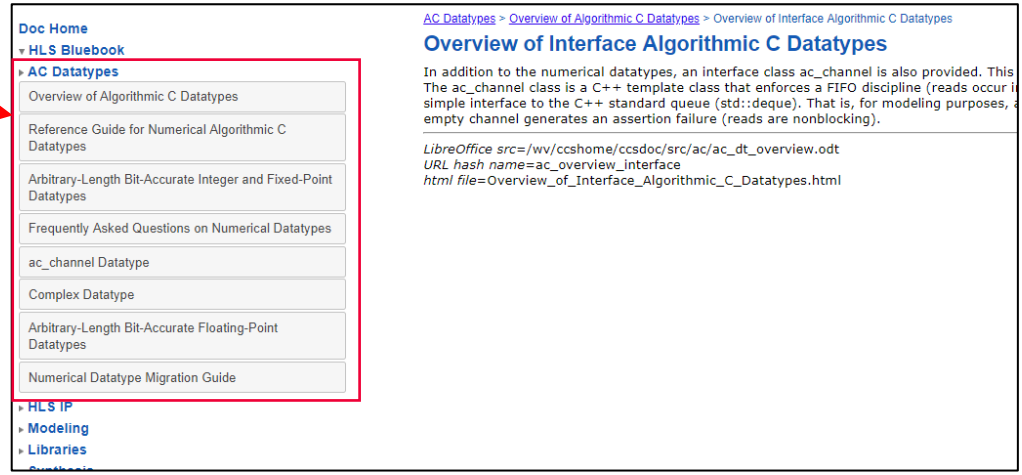
Cannot print ac_data-types directly using “printf”

- Will get a non-POD error during runtime
- Need to use built-in conversion functions for ac_data-types: to_int(), to_uint(), to_double(), etc
- Can also cast to native C data type (only works up to 32-bits)

```
printf(“%d\n”, data.to_int());
printf(“%d\n”, (int)data);
```

See data-types Documentation for Details





Modelling Datapath Interfaces

Modeling Streaming Interfaces

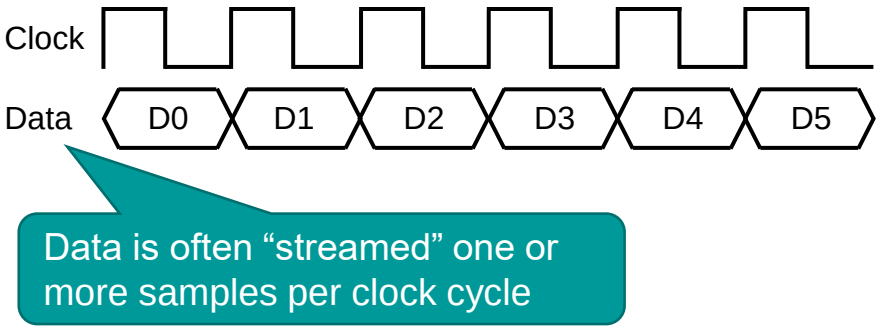
Need a way to model a streaming read/write interface
- Pointer arithmetic doesn't work well

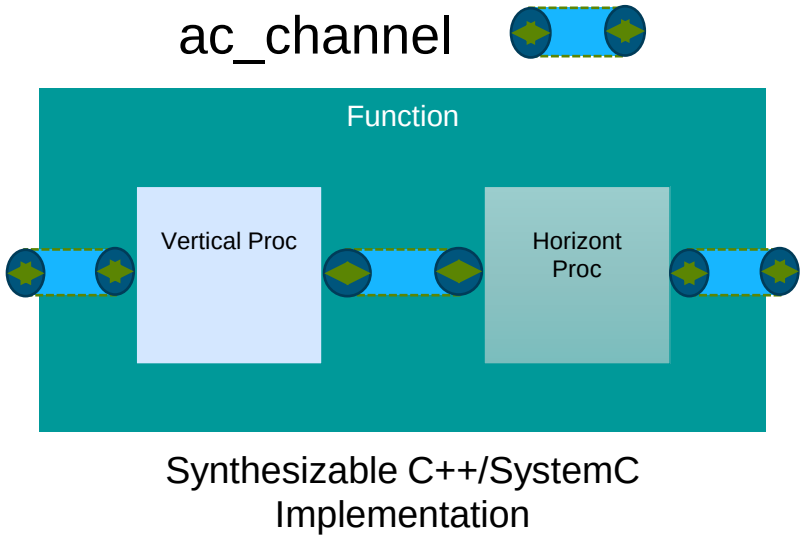
AC Channel is a modeling device for sequential C++
- Allows designers to model streaming data interfaces

Enforces a "streaming" producer/consumer

Behaves like a FIFO with infinite depth
- At the top level of hardware
- Between C++ functions

Simple use model
- Enforces streaming I/O
- Essential for multi-rate designs





AC Channel Usage

Including the header file:

```
#include <ac_channel.h>
```

ac_channel is templatized to accept any data type T

template<typename T>
class ac_channel{…

Must be declared using a reference “&” on the function interface

```
void simple_channel (ac_channel<int > &data_in,
                     ac_channel<int > &data_out);
```

AC Channel Methods for Reading and Writing

Read methods

T read(), read(T &data)

Write method

write(T data)

```
#include <ac_channel.h> //Include ac_channel class library
#include <ac_int.h>
#pragma hls_design top
void interface_modelling(ac_channel<ac_int<4,false> > &din,
                         ac_channel<ac_int<7,false> > &dout){
  ac_int<7,false> acc = 0;

  ACC:for(int i=0; i<8; i++){
    acc += din.read();
    dout.write(acc);
  }
}
```

ac_channel with 4-bit unsigned integer type

Design reads/writes a stream of 8 samples

ac_channel read

ac_channel write

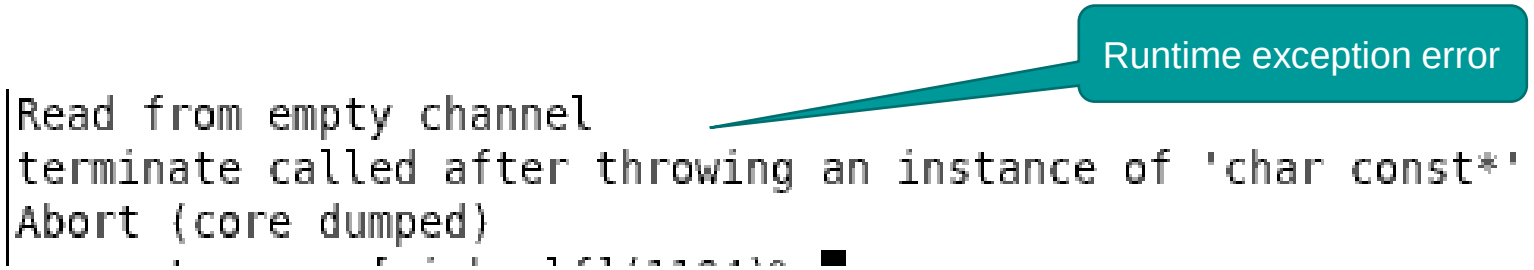
AC Channel Blocking Behavior

Reads and Writes are “blocking” interfaces in hardware and will stall the hardware pipeline if data not available

Writing a channel in software will always complete (Infinite FIFO in simulation)

Reading an empty channel in software simulation will generate a runtime exception

- Run in C++ debugger to identify channel



Preventing AC Channel Underflow

ac_channel “available” method used to prevent reading an empty channel

bool available(int n) – returns true if channel contains “n” or more data

This is a device for C++ simulation only

- Prevents runtime error from under flowing ac_channel/FIFO

“.available(n)” always synthesizes to “true”

- Data not available for the hardware interface will stall the pipeline

```
#include <ac_channel.h> //Include ac_channel class library
#include <ac_int.h>
#pragma hls_design top
void interface_modelling(ac_channel<ac_int<4,false> > &din,
                         ac_channel<ac_int<7,false> > &dout){
  ac_int<7,false> acc = 0;

  if(din.available(8)){
    ACC:for(int i=0; i<8; i++){
      acc += din.read();
      dout.write(acc);
    }
  }
}
```

Check for 8 values in channel before reading 8 values from channel

Modeling Frame/Packet Based Designs for Simulation

Frame/Packet design allows higher-level coding style using “for” loop to control the data flow

Entire dataset must be available to start processing

Often needed in testbench and C++ design

```
#include <ac_channel.h> //Include ac_channel class library
#include <ac_int.h>
#pragma hls_design top
void interface_modelling(ac_channel<ac_int<4,false> > &din,
                         ac_channel<ac_int<7,false> > &dout){
  ac_int<7,false> acc = 0;

  if(din.available(8)){
    ACC:for(int i=0; i<8; i++){
      acc += din.read();
      dout.write(acc);
    }
  }
}
```

Only execute “for” loop when the entire “frame” is available

Modeling Sample Based Designs for Simulation

Need to use sample-based coding style if amount of data to be processed is unknown

Requires that the control is explicitly coded instead of relying on loops

- Loops replaced with static counters and conditions where needed

Static counter replaces loop control

Explicitly coded control logic

```
#include <ac_channel.h> //Include ac_channel class library
#include <ac_int.h>
#pragma hls_design top
void interface_modelling(ac_channel<ac_int<4,false> > &din,
                         ac_channel<ac_int<7,false> > &dout){
  static ac_int<7,false> acc = 0;
  static ac_int<4,false> sample_cnt = 0;

  if(din.available(1)){
    acc += din.read();
    dout.write(acc);
    sample_cnt++;
    if(sample_cnt == 8){
      sample_cnt = 0;
      acc = 0;
    }
  }
}
```

Process one sample at a time

Example - Decimating Average

Reading, writing, and checking availability

```
int main () {
  ac_channel< int > data_in ;
  ac_channel< int > data_out ;

   for (int i=0;i<8;i++) {
      int z = (i+1)*10 ;
      data_in.write(z) ;
      cout << "writing " << z << endl ;
      averager(data_in, data_out) ;
   }

   cout << endl << "emptying stream" ;
   cout << endl << endl ;
   do {
      cout << data_out.read() << endl ;
   } while (data_out.available(1)) ;
}
```

```
void averager (
   ac_channel< int > &data_in,
   ac_channel< int > &data_out) {

   if (data_in.available(2)) {
      int acc = 0 ;
      for (int i=0;i<2;i++)
         acc += data_in.read() ;
      data_out.write(acc >> 1) ;
   }
}
```

```
writing 10
writing 20
writing 30
writing 40
writing 50
writing 60
writing 70
writing 80

emptying stream

15
35
55
75
```

Streaming Arrays on the Interface

Pack arrays in structs if you want to stream the entire array in a single read/write operation

Arrays of channels are possible but will create separate handshake control logic for each array element

```
#include "ac_channel.h"

struct chanStruct{
        int data[8];
};

void test(ac_channel<chanStruct> &din, ac_channel<chanStruct> &dout){
        chanStruct tmp;

        tmp = din.read();
        dout.write(tmp);
}
```

Struct allows array data to be read/written in a single assignment

Channel reads/writes moves entire array data at once

Streaming User Defined Types

Pack arrays and scalar data in a struct to stream the array and a variable in a single read/write operation

```
#include "ac_channel.h"
#include "ac_int.h"

struct chanStruct{
        ac_int<8, false> data1[4];
        ac_int<8, false> data2;
};

#pragma hls_design top
void test(ac_channel<chanStruct> &din, ac_channel<chanStruct> &dout){
        chanStruct tmp;

        tmp = din.read();
        dout.write(tmp);
}
```

A struct with variables of ac_int datatype.

Channel reads/writes the array data in parallel

Use C++ Templates to make User-types Programmable

User type can be parametrized for number of array elements, and data types (size, T1, T2)

```
#include "ac_channel.h"
#include "ac_int.h"

template <int size,typename T1,typename T2 >
struct chanStruct{
          T1 data1[size];
          T2 data2;
};
#pragma hls_design top
void test( ac_channel<chanStruct<4, ac_int<8, false>, ac_int<16, false> > > &din,
           ac_channel<chanStruct<4, ac_int<8, false>, ac_int<16, false> > > &dout){

          chanStruct<4, ac_int<8, false>, ac_int<16, false> > tmp;

          tmp = din.read();
          dout.write(tmp);
}
```

A templatized struct to specify the types of its variables and the size of the array

C++ Class Based Modeling

Catapult supports creating design blocks directly from C++ classes

Multiple instances

- Synthesize once use many times
- Some current restrictions on coding style

Flexible and elegant coding style

Should be used for multi-block design

C++ Class Based Leaf-block Coding Style Rules

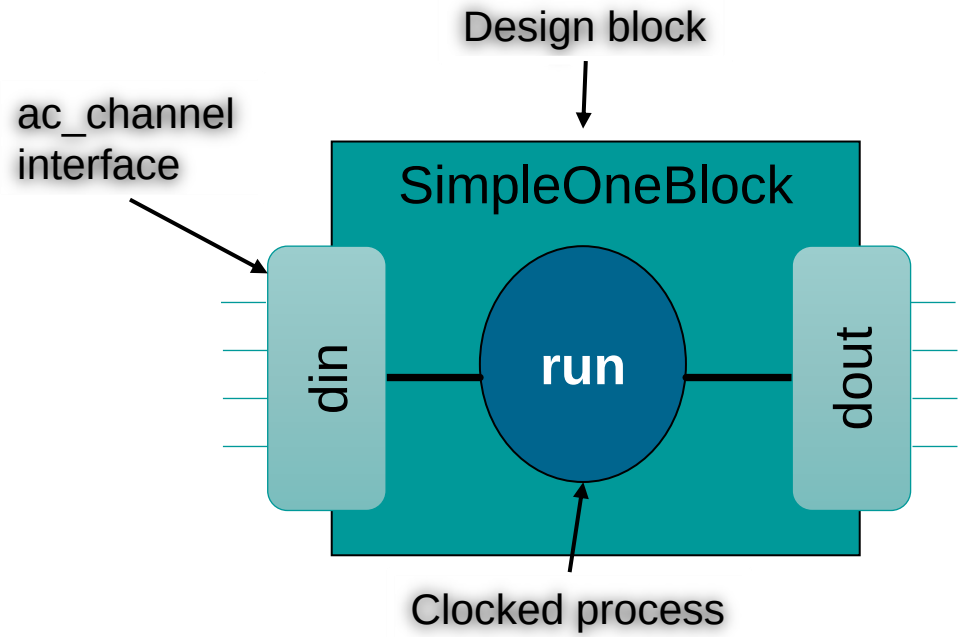
Top-level design synthesized from C++ class

- Class synthesized as a design block

Class member function defines design interfaces

- MUST be declared “public”
- **#pragma hls_design interface** defines design block interfaces
- Only allowed one public member function with #pragma hls_design interface
- All data movement into and out of the design must be done here
- Class synthesized as a clocked process
- Can instantiate other design blocks

```
class SimpleOneBlock{
  int acc;
  public:
  SimpleOneBlock(){}//Required constructor
  //only one public function with hls pragma allowed
  #pragma hls_design interface
  void CCS_BLOCK(run)(ac_channel<int> &din,
                      ac_channel<int> &dout){
    acc += din.read();
    dout.write(acc);
  }
};
```



C++ Class Based Leaf-block Coding Style Rules

Class constructor must be declared by user

- **Always required by Catapult (Error: No definition for class)**
- Persistent and member data initialized here (This is reset)
- User must take care of member data initialization
  - Catapult does not “see” class usage in testbench, doesn’t know if instance is static

```
class SimpleOneBlock{
  int acc;
  public:
  SimpleOneBlock(){//Required
    acc = 0;//Initialization of
  }

  //only one public function with hls pragma allowed
  #pragma hls_design interface
  void CCS_BLOCK(run)(ac_channel<int> &din,
                      ac_channel<int> &dout){
    acc += din.read();
    dout.write(acc);
  }
};
```

This data member is persistent. Just like static

Required class constructor

Initialization of member data

```
#  Message
#  Generating synthesis internal form...
#  Found top design routine 'SimpleOneBlock' specified by directive
   No definition for class 'SimpleOneBlock'
   Compilation aborted
   Completed transformation 'analyze' on solution 'solution.v2': elapse
#  > end dofile ./directives_no_ctor.tcl
   go compile: Failed compile
```

```
 7  class SimpleOneBlock{
 8    int acc;
 9    public:
10
11    #pragma hls_design interface
12    void CCS_BLOCK(run)(ac_channel<int> &din, ac_channel<int> &dout)
13      acc += din.read();
14      dout.write(acc);
15    }
16  };
17
18  #endif
19
```

C++ Class Based Leaf-block Coding Style Rules

Class member function definition and implementation **must** be in the same header file for verification flow

Catapult verification flow(SCVerify) does not support separate class definitions and implementations

Can use a wrapper class to work around this

#pragma hls_design interface goes on wrapper class

Class definition *.hpp

```
class SimpleOneBlock{
  int acc;
  public:
  SimpleOneBlock();
  void run(ac_channel<int> &din,
           ac_channel<int> &dout);
};
```

Class implementation *.cpp

```
SimpleOneBlock::SimpleOneBlock(){
    acc = 0;
  }
void SimpleOneBlock::run(
      ac_channel<int> &din,
      ac_channel<int> &dout){
  acc += din.read();
  dout.write(acc);
  }
};
```

Wrapper class *.hpp

```
class SimpleOneBlock_wrapper{
  SimpleOneBlock inst;
  public:
  SimpleOneBlock_wrapper(){
    acc = 0;
  }
  #pragma hls_design interface
  void CCS_BLOCK(run)(
     ac_channel<int> &din,
     ac_channel<int> &dout){
    inst.run(din,dout);
  }
};
```

Preparing for Synthesis

Writing Synthesizable Code

Some input language constructs not supported

- Dynamic memory allocation
  - Physical hardware resources have finite amount of storage
- Unions
  - Could be synthesized but could easily create bad logic
- “float” and “double” native types
  - Often not desired or efficient HW but can be modeled using ac_ieee_float, ac_std_float, ac_bfloat16, ac_float, user-defined implementation, etc.
  - Otherwise convert to ac_fixed or ac_int
- Recursion with unfixed terminal depth
  - Hardware resources are finite
  - Template recursion is OK
- Pointers to arrays mapped to memories on interfaces
- Global variables shared between design blocks

Globals for static const/ROMs are Ok

Bound Array Declarations and Formals

Catapult needs to know array bounds for internal arrays and design block interface variables

- Internal pointers are ok to move data around

typedef ac_int<8,false> uint8;

```
void HogAlg(unsigned char *dat_in,
        double *magn,
        double *angle){
    double *dx, *dy;

    dy = (double*)malloc(128*64*sizeof(double));
    dx = (double*)malloc(128*64*sizeof(double));

    VerticalGradient(dy, dat_in);
    HorizintalGradient(dx, dat_in);
    MagnitudeAngle(dx, dy, magn, angle);

}
```

Pointer to array on HW interface not synthesizable

dynamic memory allocation not synthesizable

Bound arrays on HW interface

```
void CCS_BLOCK(HogSynthesizable)(uint8 dat_in[128][64],
        uint9 magn[128][64],
        ac_fixed<8,3>  angle[128][64]){

    int9 dx[128][64], dy[128][64];

    VerticalGradient(dy, dat_in);
    HorizontalGradient(dx, dat_in);
    MagnitudeAngle(dx, dy, magn, angle);

}
```

Bound internal array declarations

Using HLS Math Libraries

math.h functions are not synthesizable

Catapult math library implements most commonly used math functions for fixed point an integer types

- Div, sqrt, atan2, sin/cos, etc
- Implemented using piecewise linear, CORDIC and recursive algorithms
- See User docs and toolkits, available on hlslibs.org

Include path

- <Catapult Install Tree>/Mgc_home/shared/include

#include <math/mgc_ac_math.h>

#include <ac_math.h>

```
for (int j = 0; j < 64; j++) {
    dx_sq = *(dx + i * 64 + j) * *(dx + i * 64 + j);
    dy_sq = *(dy + i * 64 + j) * *(dy + i * 64 + j);
    sum = dx_sq + dy_sq;
    *(magn + i * 64 + j) = sqrt(sum);
    *(angle + i * 64 + j) = atan2(dy[i * 64 + j], dx[i * 64 + j]);
}
```

Math.h not directly supported

```
MCOL:for (int j = 0; j < 64; j++) {
    dx_sq = dx[i][j] * dx[i][j];
    dy_sq = dy[i][j] * dy[i][j];
    sum = dx_sq + dy_sq;
    //Catapult's math library implementation of sqrt and atan2
    sqrt(sum, sq_rt);
    atan2((ac_fixed<9,9>) (dy), (ac_fixed<9,9>) (dx), at);
    magn[i][j] = sq_rt.to_uint();
    angle[i][j] = at;
}
```

Use Catapult's mgc_ac_math.h and ac_math.h implementations

Checking the Code for Bugs

C based languages have a certain amount of ambiguity and errors can easily create bad RTL

- Make sure that the code is “clean”

Initialize variables before their read

- Uninitialized Memory Reads (UMR) can be very unpredictable. Simulation behavior varies between compilers and platforms
- Synthesis tool may optimize away variables that are UMR

Make sure that there are no out-of-bounds array reads/writes (ABW, ABR)

- Simulation may pass/fail intermittently
- This is bad logic in hardware

Linting and Property Checking Tools

- **Catapult Design Checks (CDesignChecker)**
  - can find most user-errors like these at the push of a button
- Valgrind
- Purify

Disabling code

Debug code and status messages need to be eliminated from Synthesis Code

- High level behavioral description is often used for architecture exploration which usually have this kind of information included
- Can make the code non-synthesizable
- May result in unnecessary hardware

Unsynthesizable code can be excluded from synthesis using a compiler pre-processor definition (Catapult solution)

```
#ifndef __SYNTHESIS__
  printf (...)
  cout << ...
#endif
```

Automatic C++ to RTL Verification

SIEMENS

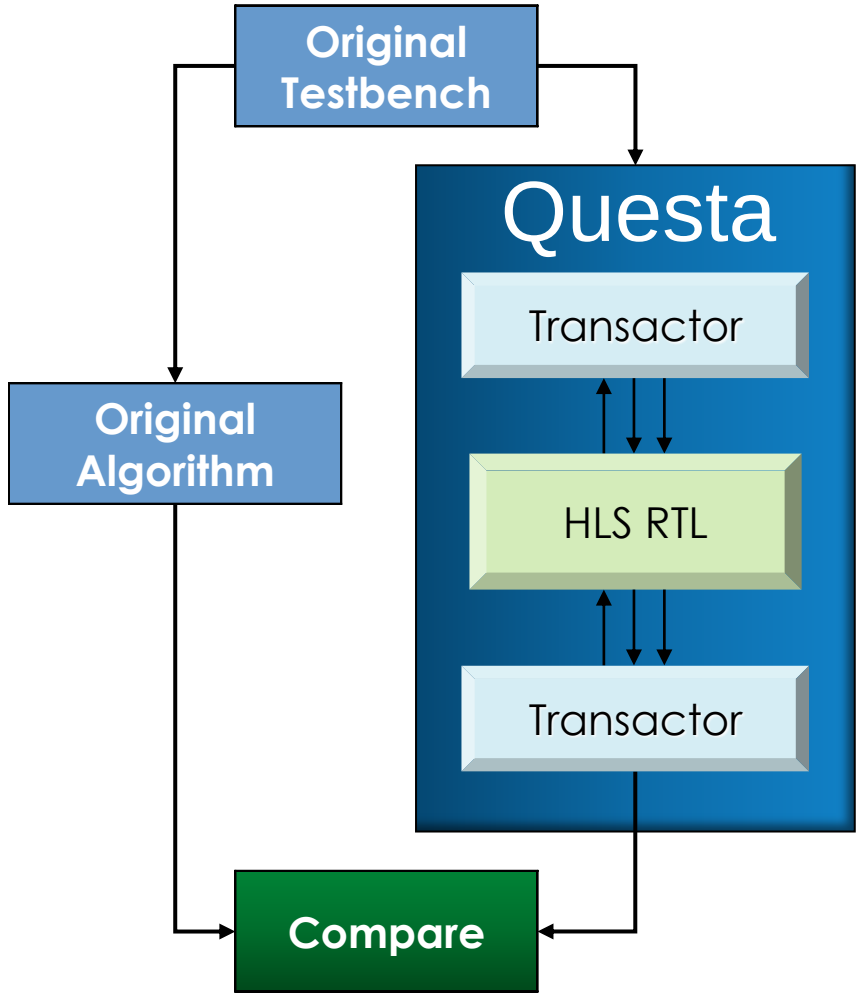
SCVerify Catapult Verification Extension

Facilitates the RTL Sanity-check of the synthesized design

The original C++ testbench can be reused (after a few changes) to verify the RTL

Transactors convert function calls to pin-level signal activity

Push button verification solution creates Makefiles and Simulation Scripts for ModelSim



Rules For Include Files

Verification flow needs to know about objects

- Use header files

Defining types in your C++ files will break the flow

- Data types common to the testbench and the design under test should be defined in a shared header

You must declare your function prototype in a header file

- Standard C/C++ practice

Rules for Includes in Design Files

Good practice:

- #include “my_types.h”
- #pragma hls_design top
- void my_func(my_type in, …

SCVerify will include this header file, put types and function prototypes in this file.

Bad practice:

- void my_func(my_type in, …);
- typedef ac_int<10> my_type;
- #pragma hls_design top
- void my_func(my_type in, …

Fine for synthesis, but SCVerify has no way to define this type or function prototype & match it with the testbench. An error may be generated.

CCS_SCVERIFY

C++ Pre-processor definition provided by the verification flow:

- “CCS_SCVERIFY”

Allows the user to include C/C++ code that is only used during the SystemC verification flow

- E.g. Using standard C/C++ I/O to print messages or write to files during simulation:

```
#ifdef CCS_SCVERIFY
cout << “Automatic verification flow” << endl;
#endif
```

Testbench Modifications

Include Catapult verification library header mc_scverify.h after other includes in testbench and DUT files

- #include <mc_scverify.h>

Use macro for “main” definition

- CCS_MAIN(int argc, char **argv) { …

Use a macro on the function call in the C++ testbench

- CCS_DESIGN(top)(…);

NEW: Use a macro on the function implementation for your hardware

- CCS_BLOCK(top)(…);
- Needed for Class based Hierarchy

Use a macro for return

- CCS_RETURN(0);

Once these changes are made – the flow is “push-button” from within Catapult

Testbench Modification Example – CCS_BLOCK

Original testbench

```
int main(int argc, char *argv){
   int a;
   int b;
   int dout;

   //run adder 10 times
   for(int i=0;i<10;i++){
      a = b = i;
      add(a,b,dout);
   }
   return 0;
}
```

Modified testbench

```
#include <mc_scverify.h>
CCS_MAIN(int argc, char *argv){
   int a;
   int b;
   int dout;
   add add_inst;
   //run adder 10 times
   for(int i=0;i<10;i++){
      a = b = i;
      add_inst.run(a,b,dout);
   }
   CCS_RETURN(0);
}
```

DUT

```
#include <mc_scverify.h>
class add{
public:
  add(){}
  #pragma hls_design interface
  void CCS_BLOCK(run)(int a, int b, int &dout){
  dout = a + b;
  }
};
```

Catapult SCVerify Verification Flow Requirements/restrictions

CCS_BLOCK verification macro must be used on class public member function to run SCVerify

DO NOT directly modify data members (Read only is ok)

- Catapult will not detect this and SCVerify will fail

```
#include <mc_scverify.h>

class SimpleOneBlock{
  int acc;
  public:
  SimpleOneBlock(){//Required constructor
    acc = 0;//Initialization of persistent data done here
  }

  //only one public function with hls pragma allowed
  #pragma hls_design interface
  void CCS_BLOCK(run)(ac_channel<int> &din, ac_channel<int> &dout){
    acc += din.read();
    dout.write(acc);
  }
};
```

CCS_BLOCK macro

```
#include "simple_one_block.h"

CCS_MAIN(int argv, char *argc){
  //Instance of top-level class
  SimpleOneBlock inst;
  ac_channel<int> din
  ac_channel<int>

  for(int i=0;i<10
    din.write(i+1);
    inst.run(din,dout);
  }

  while(dout.available(1))
    printf("acc = %d \n",dout.read());
  CCS_RETURN(0);
}
```

Testbench instantiation

SCVerify Error: Directly Modifying Public Data

Classes should only contain a single public member function that provides the design interface

Member data should always be made private

- DO NOT declare public data (Catapult will allow this)
- Modifying this directly in the testbench will generate failures in SCVerfiy
- Catapult cannot detect this

```
class SimpleOneBlock{

  public:
  SimpleOneBlock(){//Required constructor
    acc = 0;//Intialization of persistant data done here
  }
  int acc;
  #pragma hls_design interface
  void CCS_BLOCK(run)(ac_channel<int> &din, ac_channel<int> &dout){
    acc += din.read();
    dout.write(acc);
  }
```

BAD

```
#  stuck in dut fifo    = 0
#  stuck in golden fifo = 0
   output 'dout' had comparison errors
#
#
   Simulation FAILED @ 41500 ps
   (vsim-6574) SystemC simulation stopped by user.
#  End time: 08:44:03 on Jan 23,2016, Elapsed time: 0:00:01
#  Errors: 0, Warnings: 6
```

```
CCS_MAIN(int argv, char *argc){

  //Instance of top-level class
  SimpleOneBlock inst;//UPDATE
  ac_channel<int> din;
  ac_channel<int> dout;

  //This will break scverify and cannot be detected
  inst.acc = 5;
  for(int i=0;i<10;i++){
    din.write(i+1);
    inst.run(din,dout);
  }

  while(dout.available(1))
    printf("acc = %d \n",dout.read());
  CCS_RETURN(0);
}
```

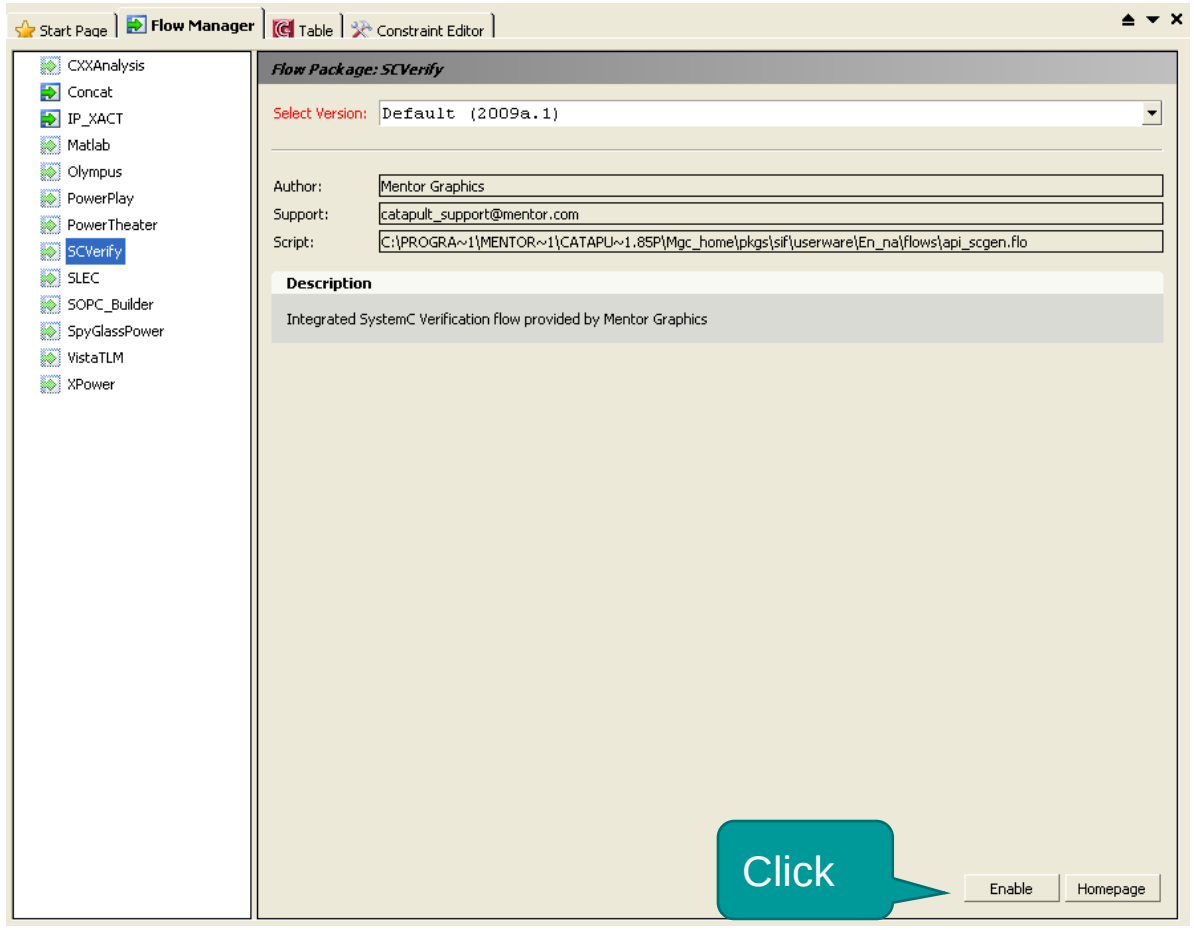
BAD

Enable the SCVerify Flow

Select Flow Manager

Select SCVerify

“Enable”



SCVerify – Flow Manager Options

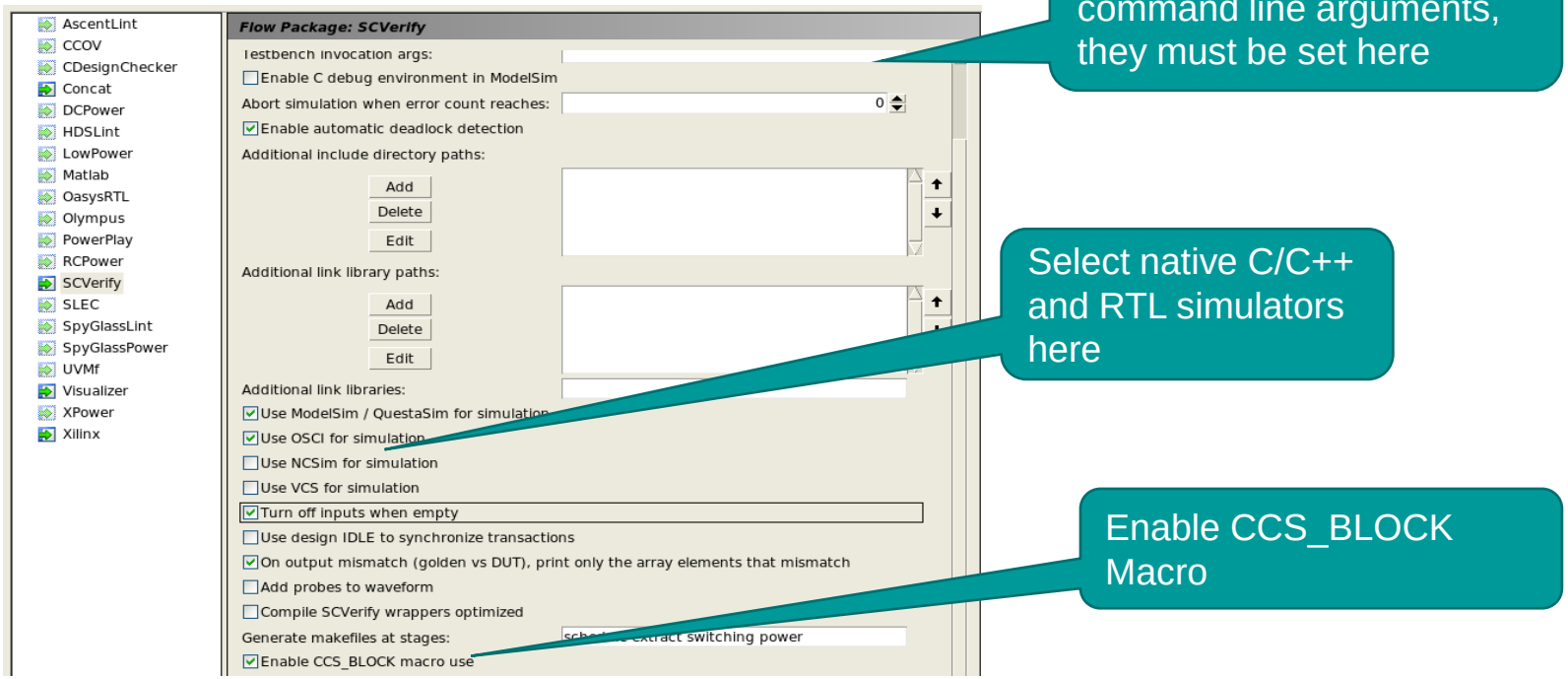
Enable flow to use CCS_BLOCK if needed

Add testbench arguments if needed

Select RTL simulation

Disable input controls

Random stall



Step-by-Step Lab1 FIR

Walkthrough

Basic Interface Protocols and Hardware Behavior

SIEMENS

Interface Synthesis

HLS infers port size and direction from the source

Source does not specify the protocol

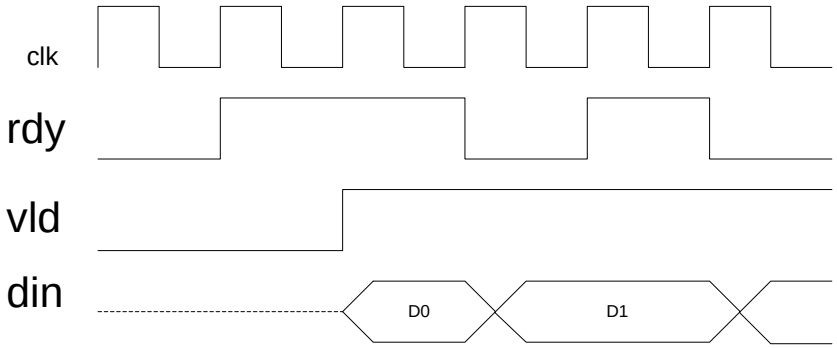
- Synchronization usually required

Interface synthesis allows the protocol to be defined using the HLS tool

- Request/acknowledge handshake
- Data enable
- Memory interface

Interface handshaking protocol applied on “din”

```
void simple (int din, int &dout) {
  dout = din;
}
```



Interface Protocols

Catapult provides a number of simple point-to-point protocols

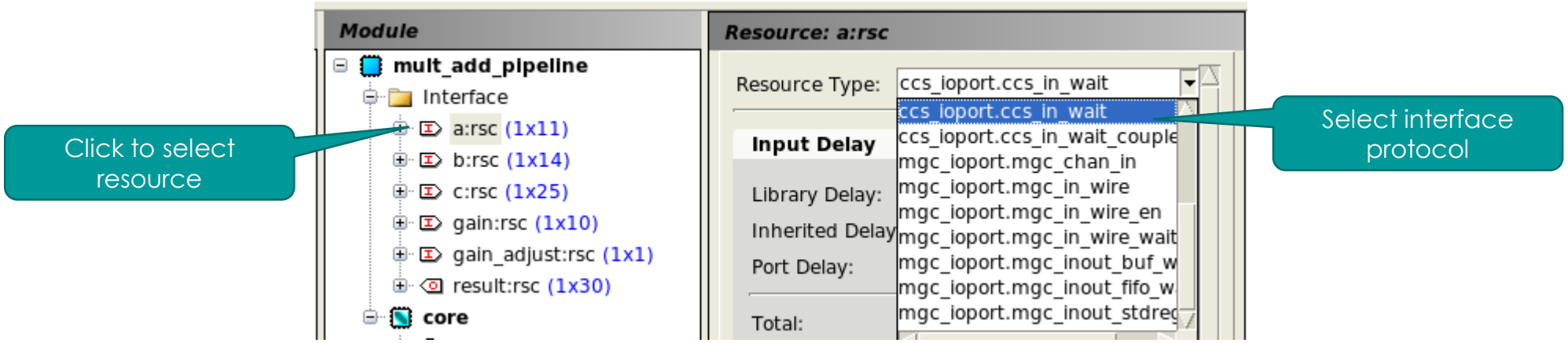
- Wire, enable, handshake, fifo, etc
- See Catapult documentation for full list

Most commonly used are the “_wait” interfaces

- Two wire handshake with stall capability (makes verification and integration easy)
- Rdy/vld/dat handshake signals
- Inputs – ccs_in_wait
- Outputs – ccs_out_wait
- Default selected for “ac_channel” interface variables
- Blocking interface

Interface Constraints

Interface protocol set by selecting a resource and applying an interface synthesis constraint



“_wait” Interfaces

“vld” always goes with the direction of the data

Interface RTL port names based on resource/variable name

- ccs_in_wait

Port Name	Port Direction	Port Type
<resource_name>_rsc_dat	Input	Read data
<resource_name>_rsc_rdy	Output	Read request
<resource_name>_rsc_vld	Input	Read data available

- ccs_out_wait

Port Name	Port Direction	Port Type
<resource_name>_rsc_dat	Output	Write data
<resource_name>_rsc_rdy	Input	Write acknowledge
<resource_name>_rsc_vld	Output	Write request

“*_vld” on read and “*_rdy” on writes sampled high indicates the data transfer can occur

- If sampled low the pipeline will stall

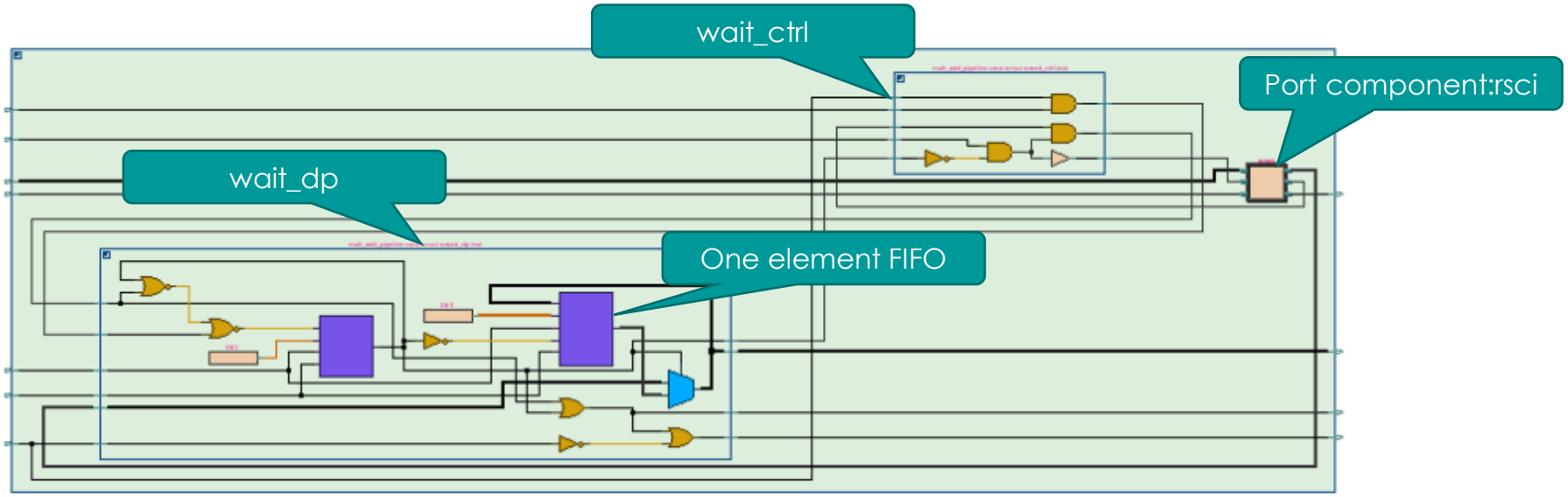
ccs_in_wait Interface Hardware

wait interface has 3 parts

- Wait controller – *wait_ctrl
- Wait Datapath with 1 Element FIFO – *wait_dp
- Port component - *rsci, just pass-through wires

2 or more wait interfaces in a design will result in a 1-element FIFO on each input wait interface

- Input FIFOs can capture data independently, even if another input is stalling the design

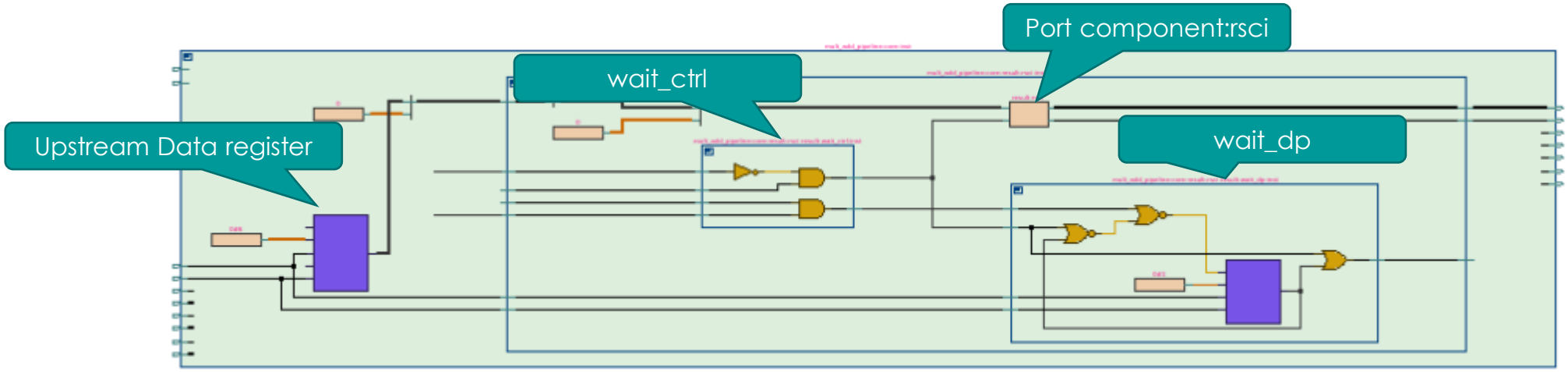


ccs_out_wait Interface Hardware

wait interface has 3 parts

- Wait controller – *wait_ctrl
- Wait Datapath - *wait_dp
- Port component - *rsci, just pass-through wires

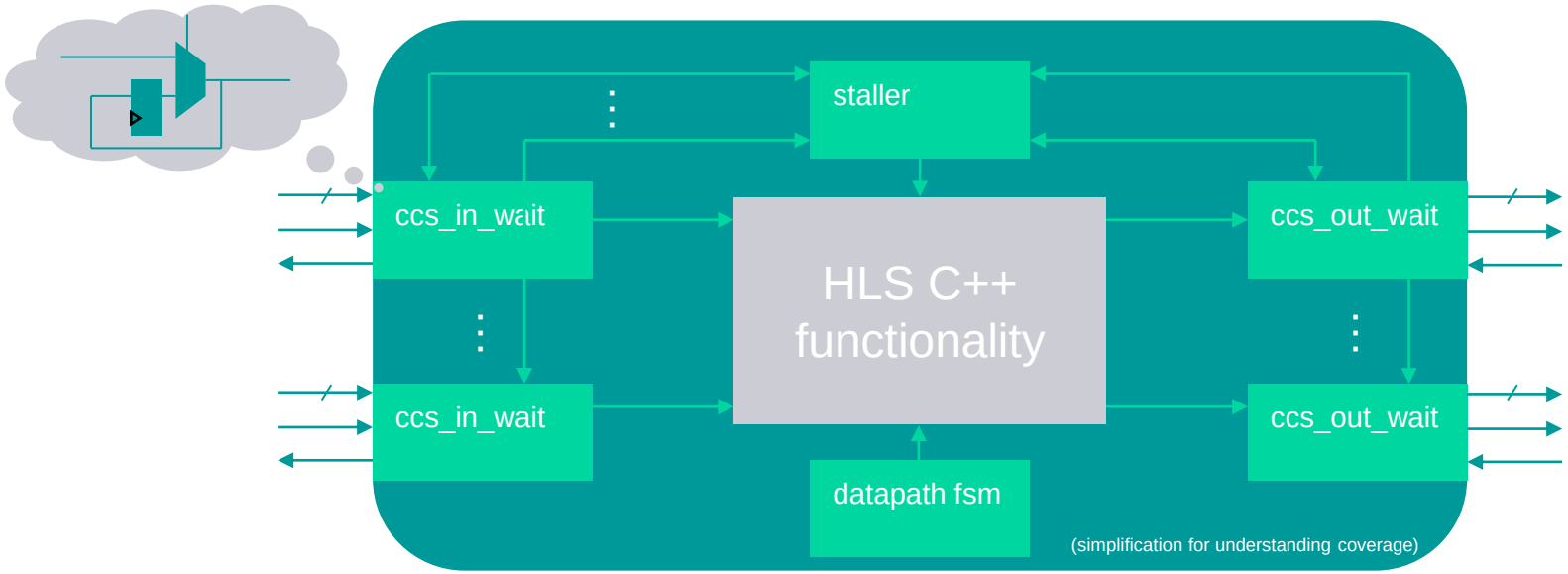
Data output is registered upstream

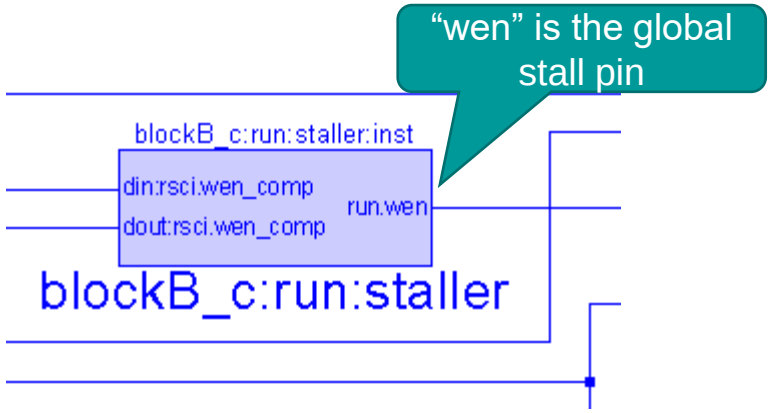


General Hardware Structure Resulting from *_wait I/F

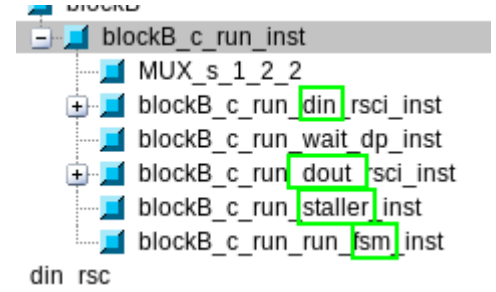
HLS adds HW not described in the source

- Control logic: interface components, staller & DPFSM
- FIFO components between blocks if so constrained





Naming in the RTL always follows basic pattern:



SIEMENS

ccs_in_wait Simulation Behavior

A stall on any *_wait input or output interface will stall the design

- Input interfaces not stalled can complete the current read

```
5  void CCS_BLOCK(mult_add_pipeline)(ac_channel<ac_int<11,false> > &a,
6                                    ac_channel<ac_int<14,false> > &b,
7                                    ac_channel<ac_int<25,false> > &c,
8                                    ac_fixed<10,2,true> gain, bool gain_adjust,
9                                    ac_channel<ac_int<30,false> > &result){
10   ac_int<25,false> product = a.read() * b.read();
11   ac_int<26,false> sum = product + c.read();
12
13   if(gain_adjust)
14     sum = ((ac_fixed<26,26,false>)(sum*gain)).to_uint();
15   result.write((ac_int<30,false> )sum<< 4);//NOTE: cast sum to ac_int<30,false
16 }
```

```
29  #ifdef CCS_SCVERIFY
30      if(i==3){
31        testbench::a_wait_ctrl.cycles = 2;
32      }
33      if(i==7){
34        testbench::result_wait_ctrl.cycles = 2;
35      }
36  #endif
37      if(i==9)
38        gain_adjust = true;
39      a_chan.write(a);
40      b_chan.write(b);
41      c_chan.write(c);
42      mult_add_pipeline_ref(a_ref,b_ref,c_ref,gain,gain_adjust,result_ref);
43      mult_add_pipeline(a_chan,b_chan,c_chan,gain,gain_adjust,result);
44      res = result.read();
```

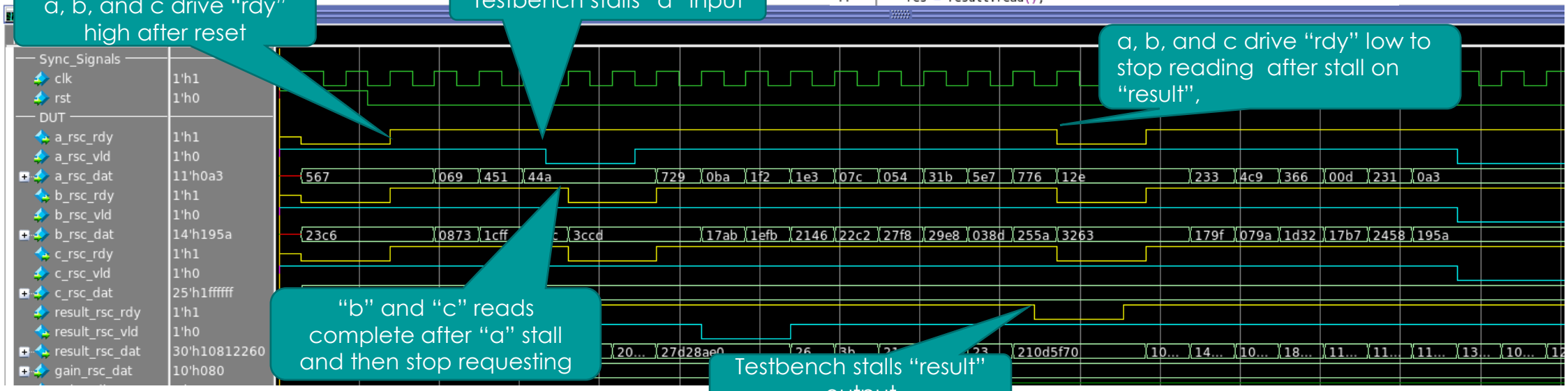
SCVerify used to test stall behavior on "a" input

SCVerify used to test stall behavior on "result" output

a, b, and c drive "rdy" high after reset

Testbench stalls "a" input

a, b, and c drive "rdy" low to stop reading after stall on "result",



"b" and "c" reads complete after "a" stall and then stop requesting

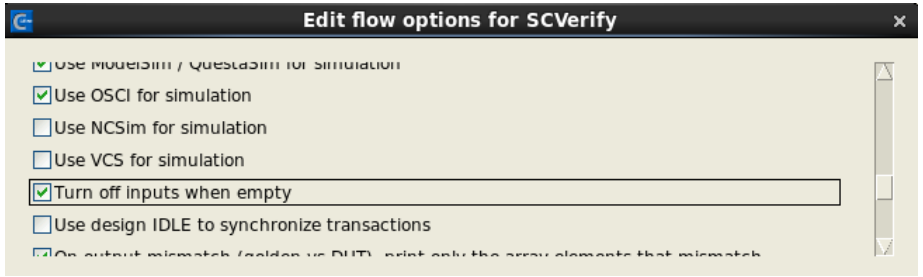
Testbench stalls "result" output

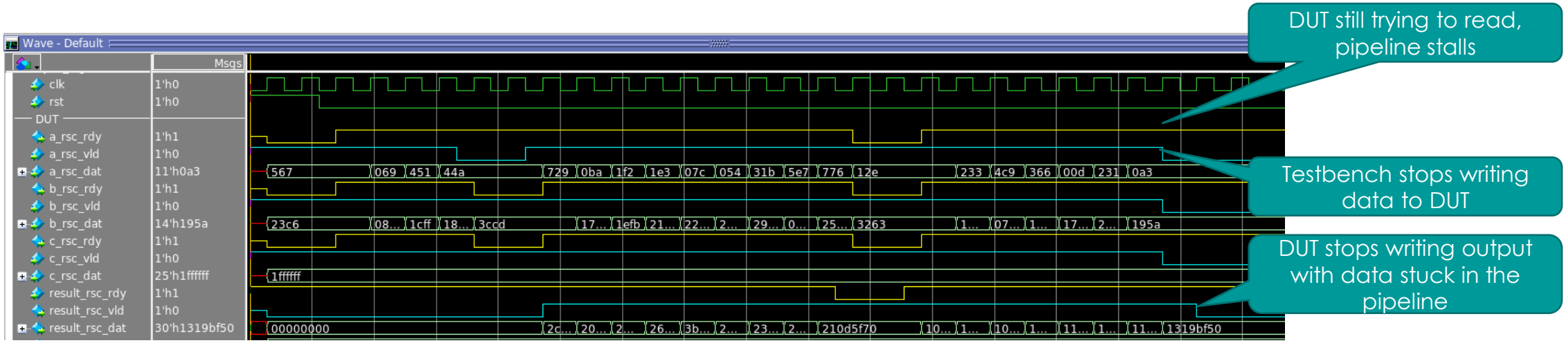
*_wait Interface Effects on Pipeline Flush Behavior

Default behavior is to stall the pipeline when an IO read stalls

SCVerify will drive “dummy” inputs to flush the pipeline

- Disable “Turn off input when empty” option to see stall behavior (This may hang the simulation

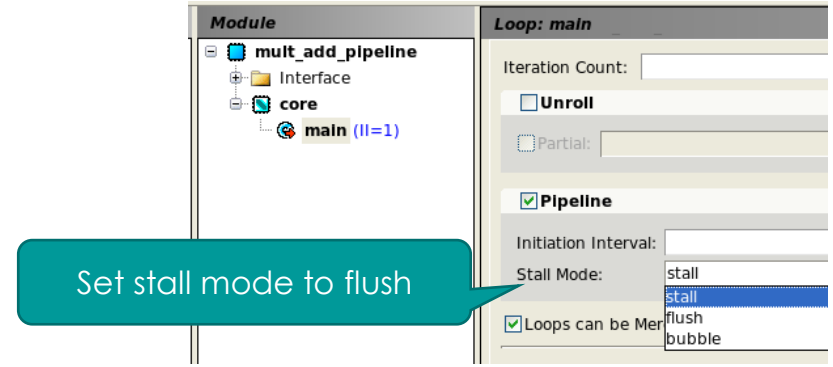


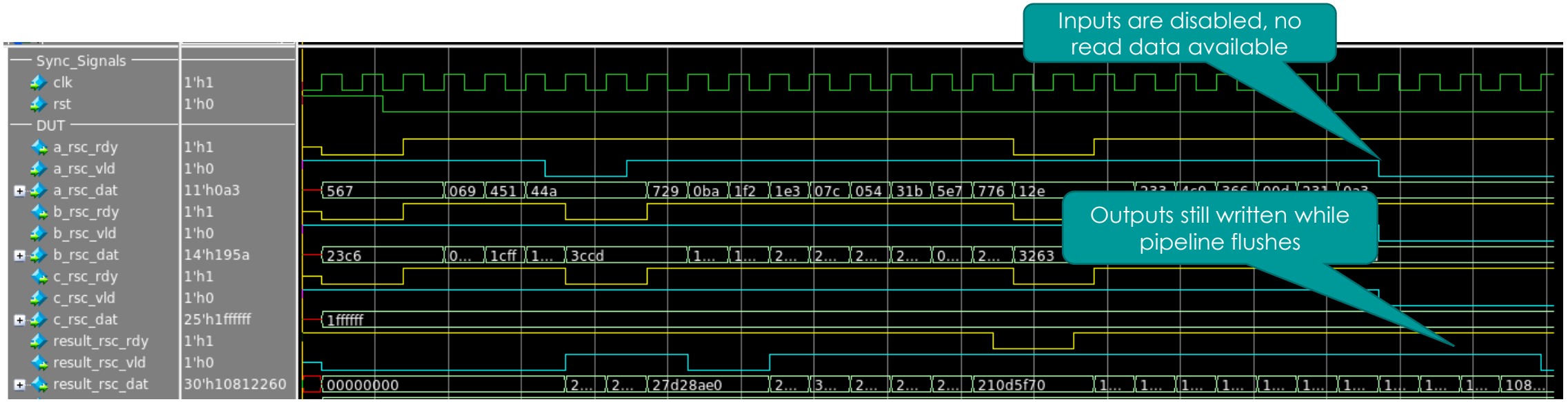


Enabling Pipeline Flushing

There are 3 different stall modes

- stall (Default) – stalls the pipeline when an input IO read stalls
- flush – flushes the pipeline when an input IO read stalls
- bubble – bubble compression





*_wait Interfaces in Summary

Easy to use

Blocking interfaces will stall the pipeline

Pipeline does not flush by default

Pipeline flushing can be enabled to flush pending data during read stalls

1-Element FIFO on every input (more area)

ccs_in_wait_coupled Interfaces

Interface RTL port names based on resource/variable name

- ccs_in_wait

Port Name	Port Direction	Port Type
<resource_name>_rsc_dat	Input	Read data
<resource_name>_rsc_rdy	Output	Read request
<resource_name>_rsc_vld	Input	Read data available

"*_vld" on read sampled high indicates the data transfer can occur

- If sampled low the pipeline will stall
- Sampled low will disable all "*_rdy" signals for all other coupled IO

Smaller area due to removal of 1-element input FIFO

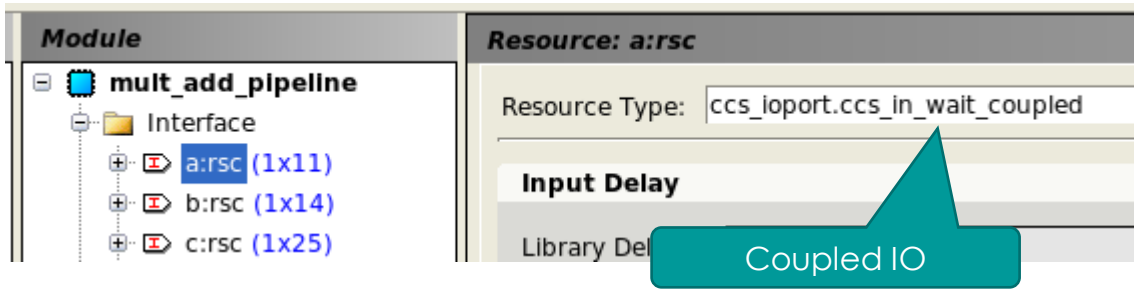
ccs_in_wait_coupled Hardware

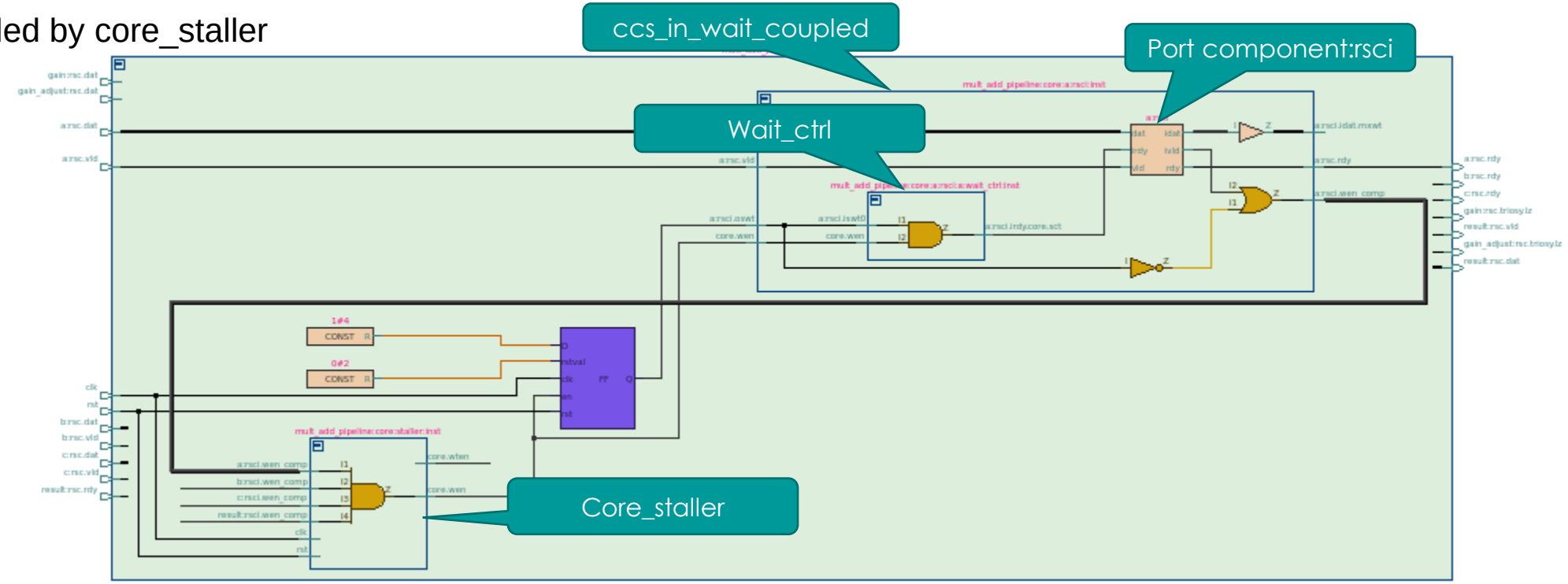
Coupled wait interface has 2 parts

- Wait controller – *wait_ctrl
- Port component - *rsci, just pass-through wires

Stalling a coupled IO will disable ready on all other coupled IO

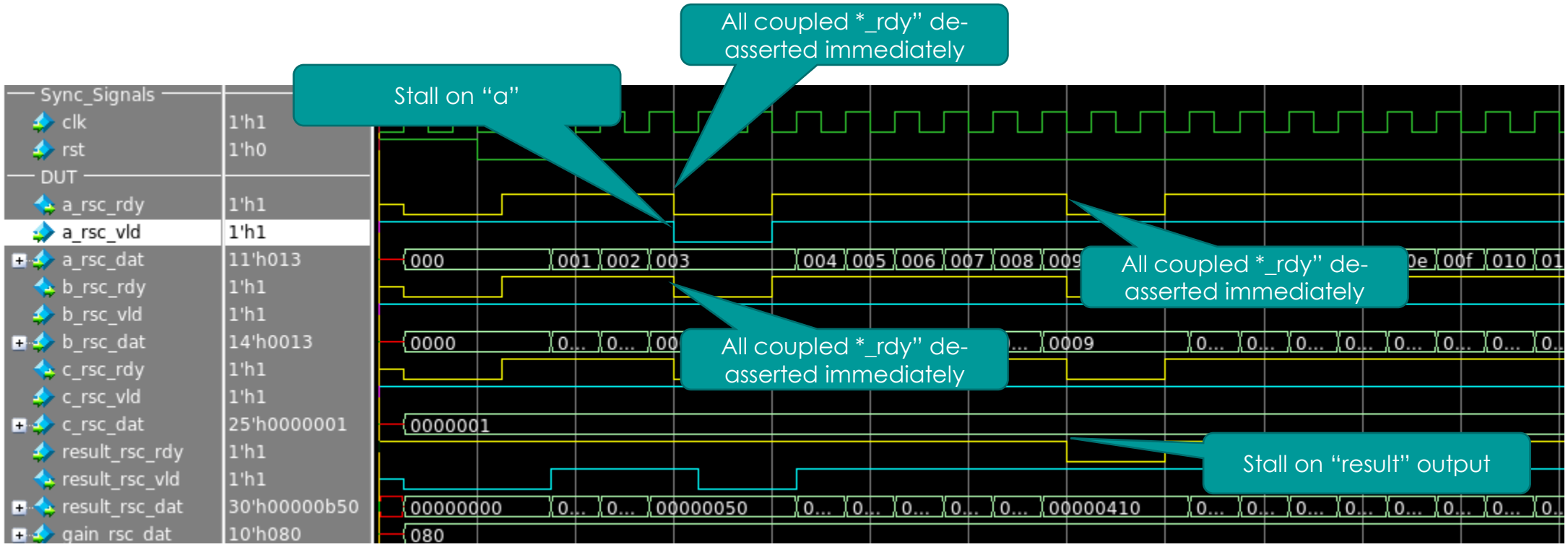
- Controlled by core_staller





ccs_in_wait_coupled Simulation Behavior

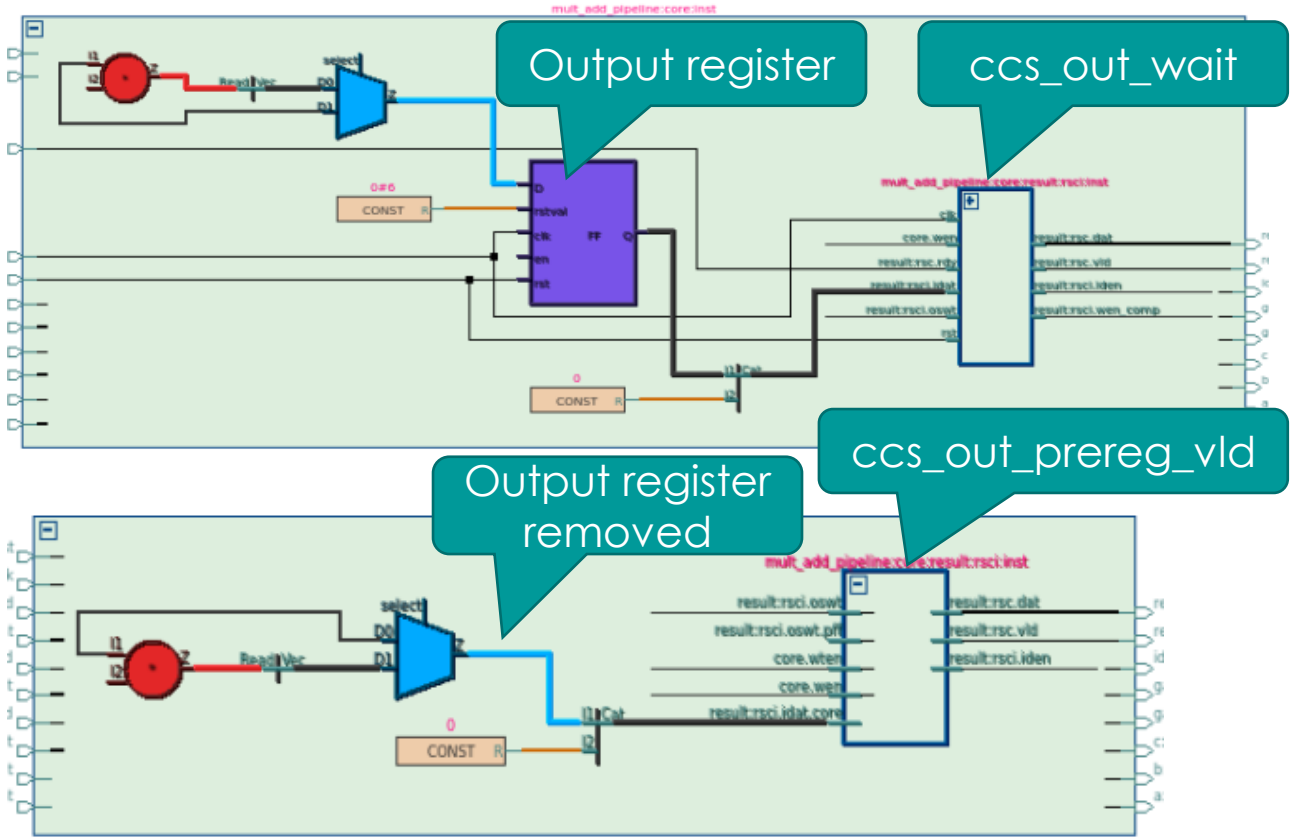
Stall on any coupled input or output immediately de-asserts all “*_rdy” signals

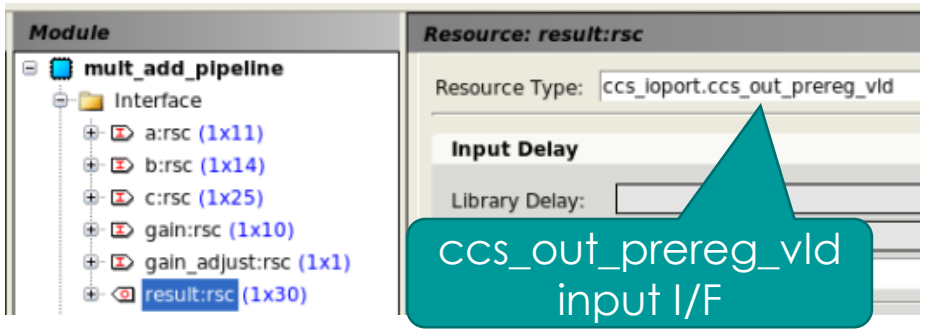


Removing Output Registers

ccs_out_prereg_vld interface will unregister the output

- Interface cannot be stalled
- Data is driven combinationally one cycle early

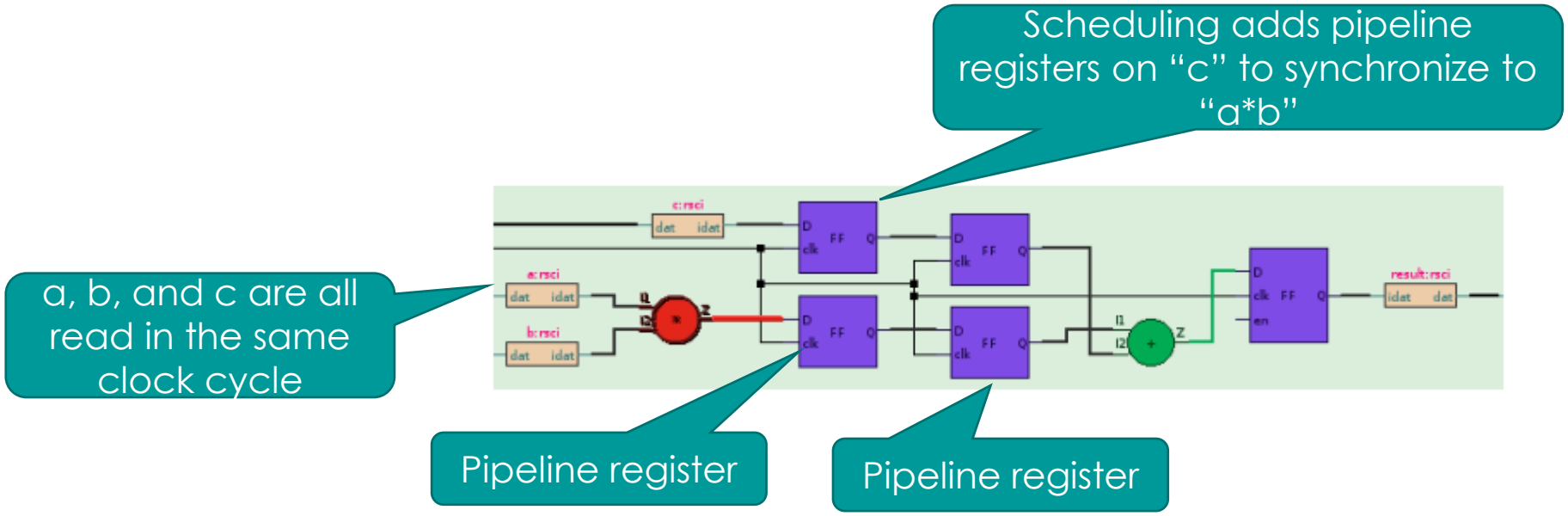




Pipeline Registers Added by Scheduling

Catapult by default will add pipeline registers on all IO to synchronize the data

```
1 #include <ac_int.h>
2 void test(uint20 a, uint20 b, uint40 c, uint41 &result){
3   result = a * b + c;
4 }
```



Removing Pipeline Registers - Direct Input

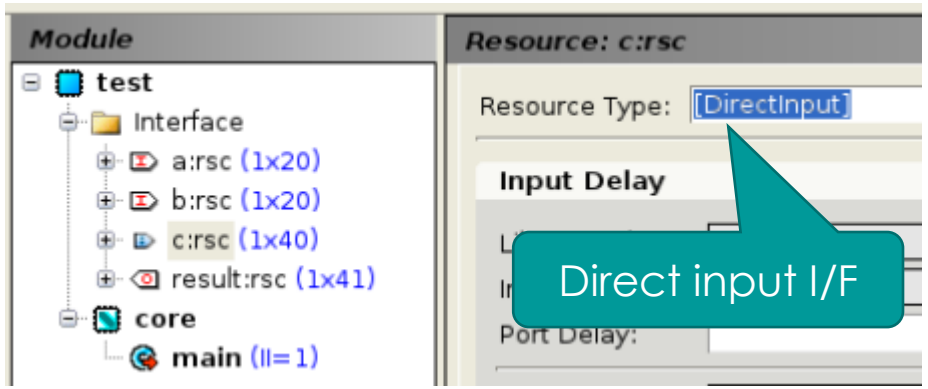
A DirectInput resource type is used to reduce design area by removing pipeline registers added by Catapult during synthesis

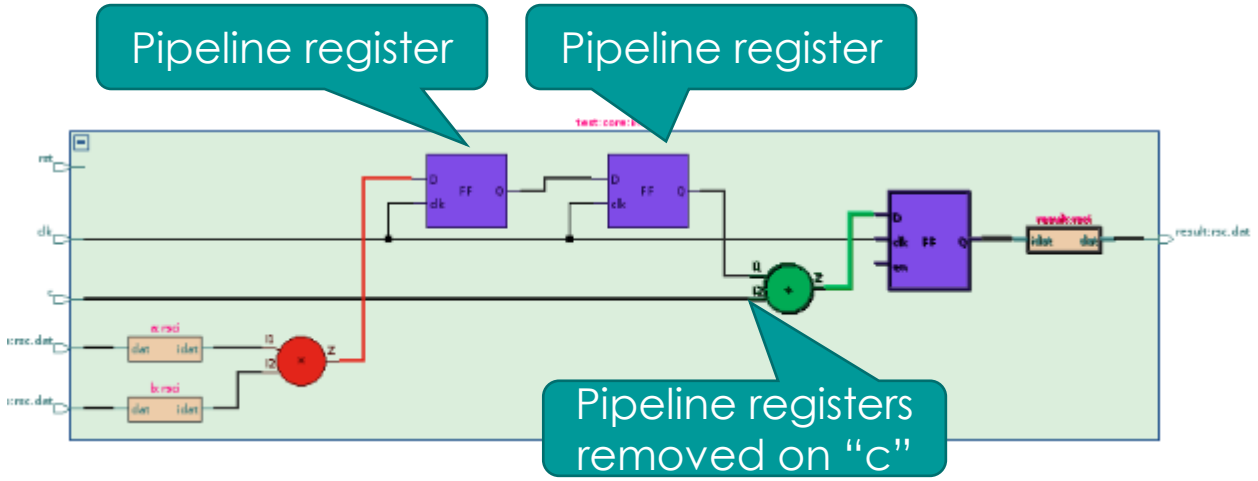
Some interfaces do not require data synchronization

- Configuration status registers

Setting an interface Resource Type to “direct_input” allows Catapult to remove pipeline registers

- Data driving port must be held stable while pipeline is active





Potential Pitfalls with Direct Input

Changing a direct input while the pipeline is active can cause verification mismatches

- Data in-flight in the pipeline may be corrupted
- Only switch Direct Inputs when the design is finished/idle

Don’t use direct input to control reading/writing of data in fully pipelined designs

- Can cause designs to deadlock
- Use ccs_in, ccs_vld, or ccs_wait

```
if(direct input variable)
  data = my_io.read()
```

Switching direct input after stall has occurred can cause deadlock

Loop Unrolling and Loop Coding Style

SIEMENS

Understanding Parallelism and Concurrency

Two Operations

- Checkout at cash register

- Bag groceries

Two shoppers

- Person A

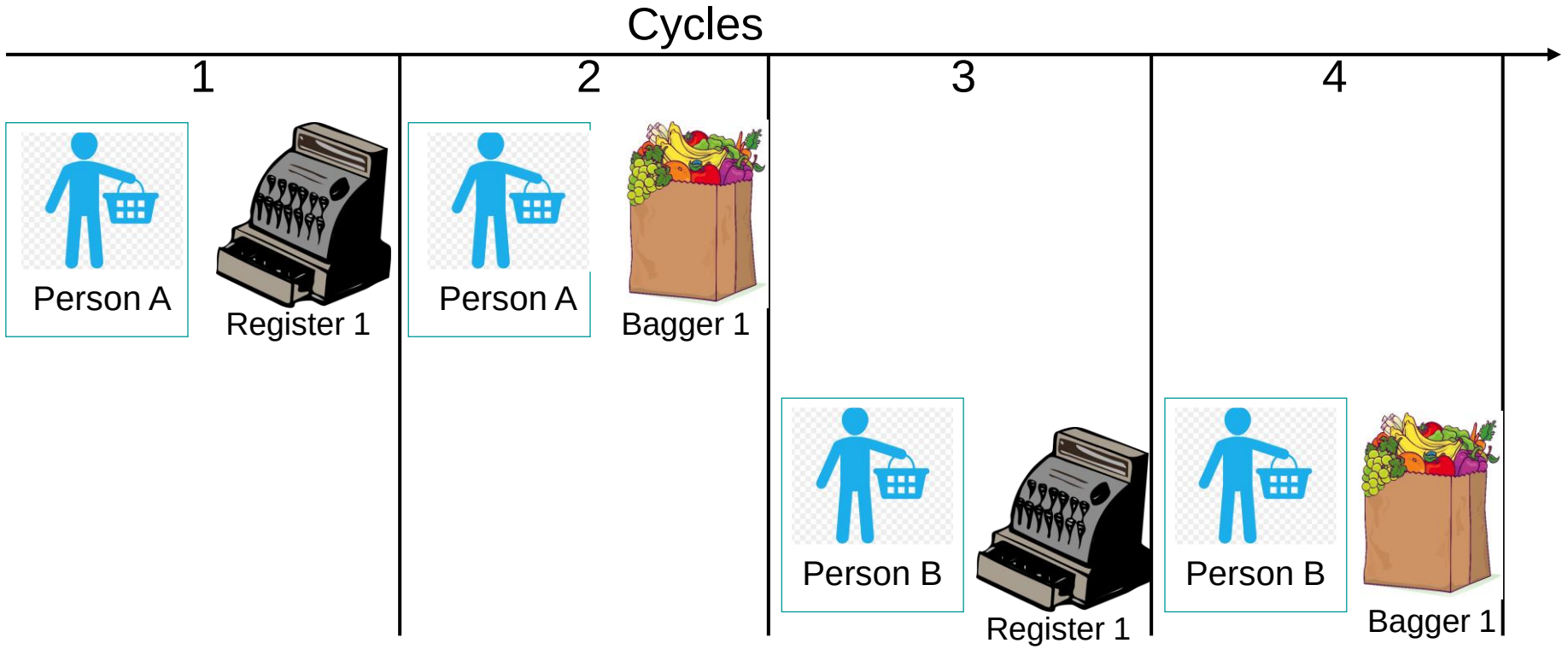
- Person B

No Parallelism or Concurrency

Person B waits until Person A groceries are bagged

Checkout Operation is Idle when Bagging Operation is active and vice versa

Latency=2, throughput = 2



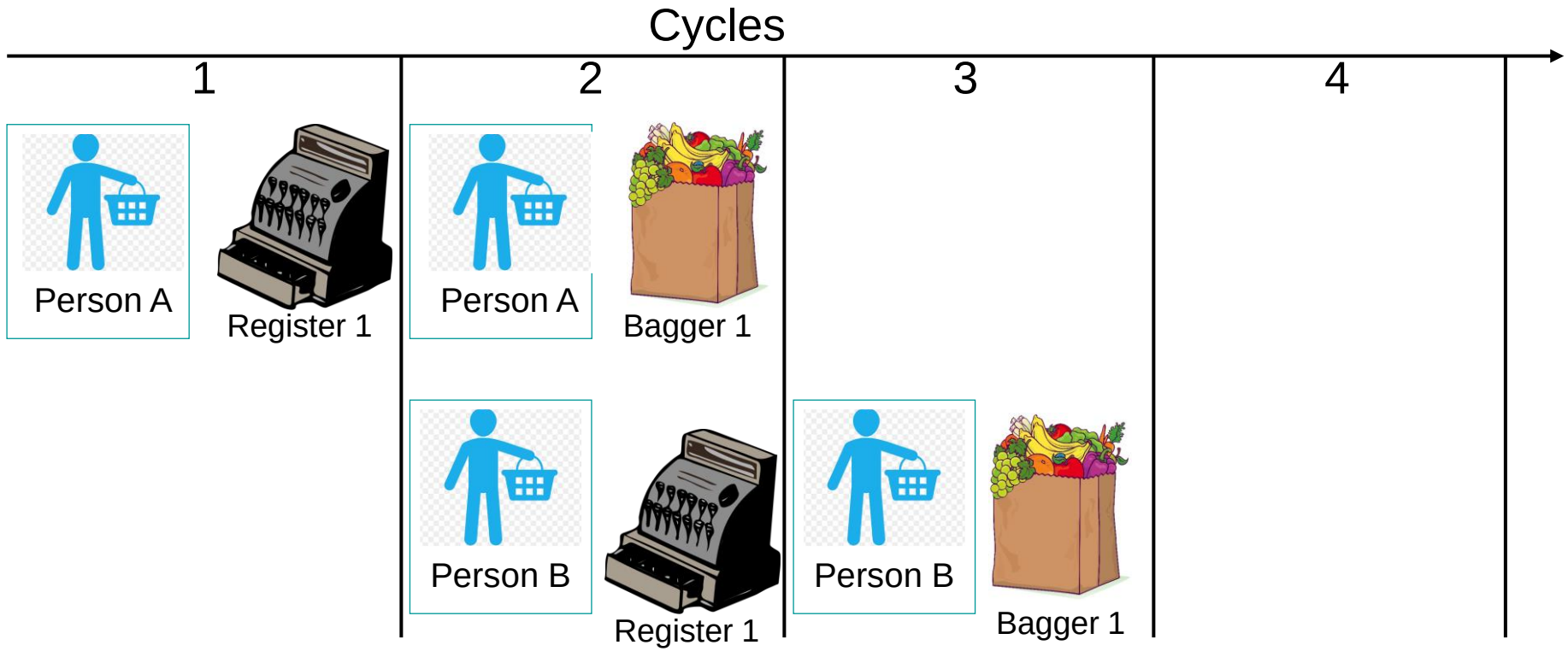
SIEMENS

Concurrency

Checkout Operation can be overlapped in time, or pipelined, with Bagging Operation

Person B can start checking out while Person A groceries are being bagged

Latency = 2, throughput = 1



SIEMENS

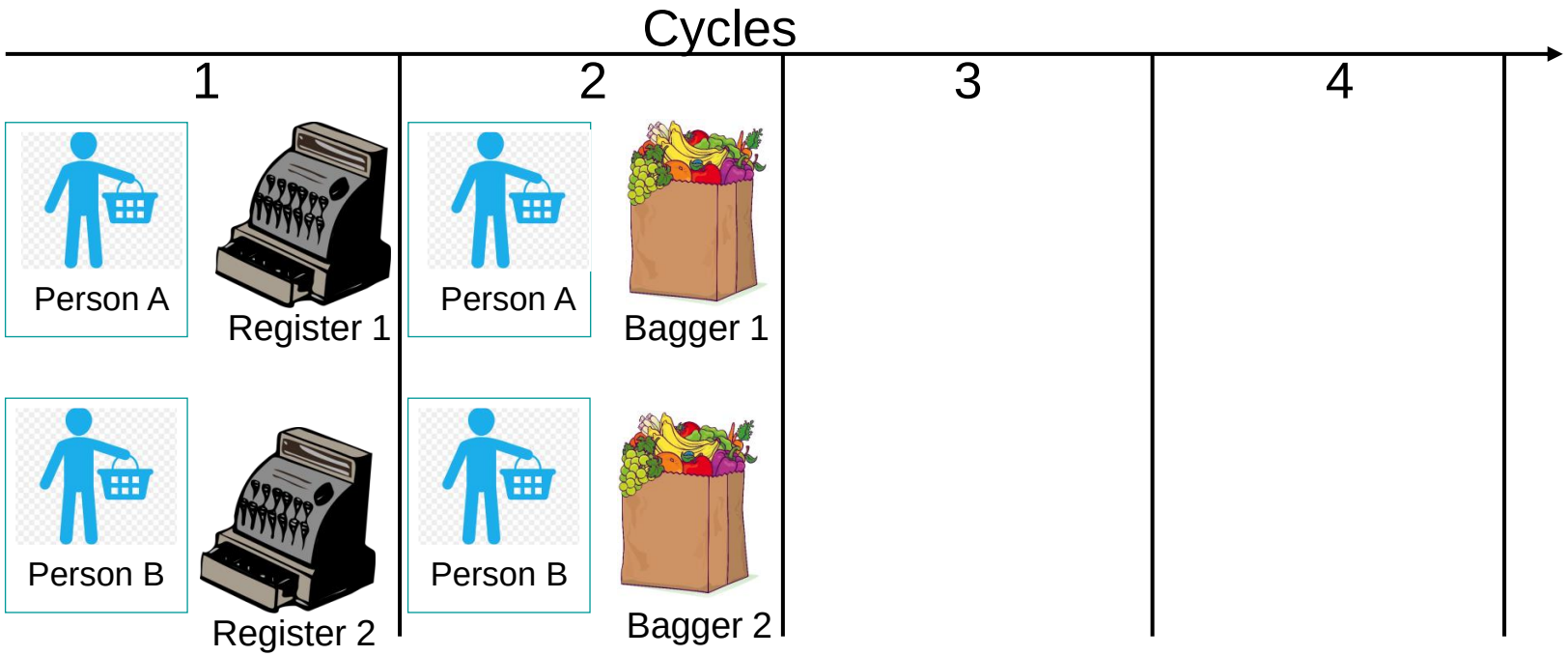
Parallelism

Person A and Person B can checkout at the same time

2 cash registers and two baggers are needed

Checkout Operations are Idle when Bagging Operations are active and vice versa

Latency=2, throughput = 2 (two bags produced every 2 cycles)



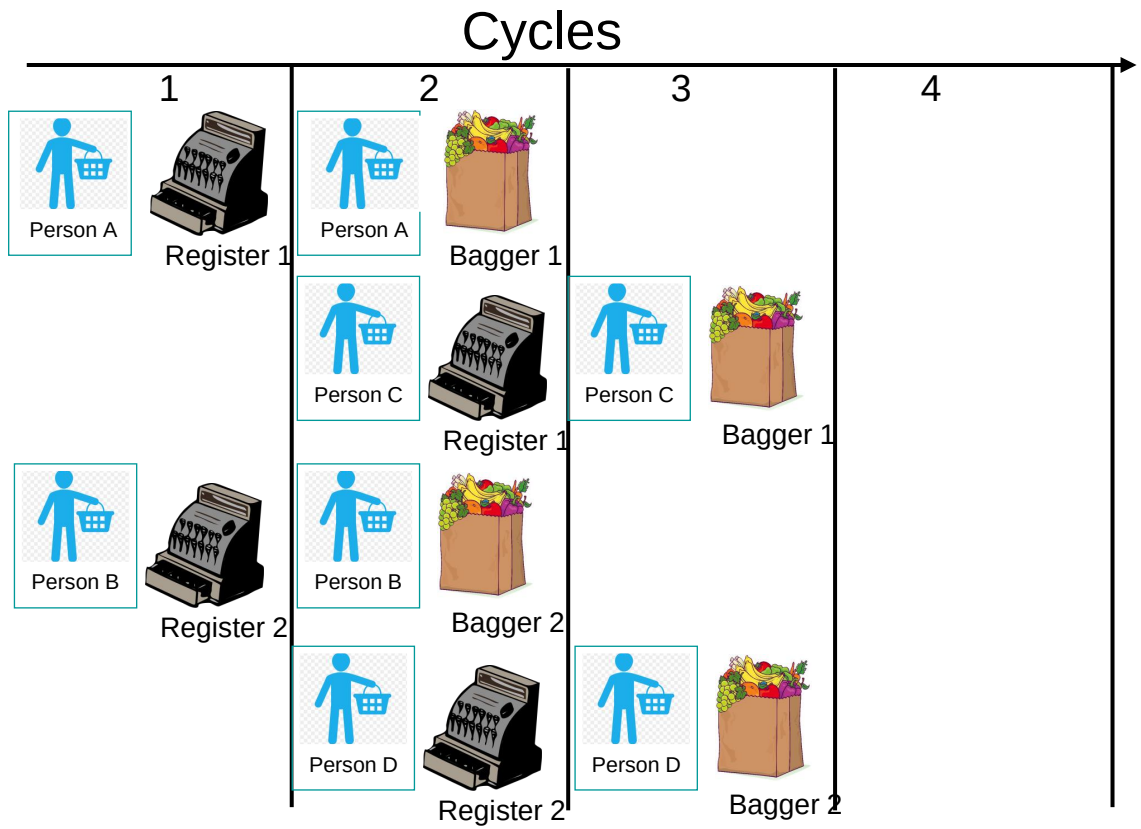
SIEMENS

Parallelism and Concurrency

2 cash registers and two baggers are needed

Check-out and bagging can happen at the same time

Latency = 2, throughput =1 (2 bags produced every 1 cycle)



SIEMENS

The Architecture

The general structure of the algorithm’s implementation

- The block diagram of the generated hardware
- The basic functionality of the computational units
- The general sequence of the computation
- How data “moves around”

Examples:

- Shift register vs. circular buffer FIR
- In place vs. systolic array FFT
- Frame based vs. window based video filtering

The architecture is also determined by the C code

- The coding style of the C++ determines the structure of the synthesized hardware

The Micro-Architecture

Implementation variant within a given algorithm/architecture pair

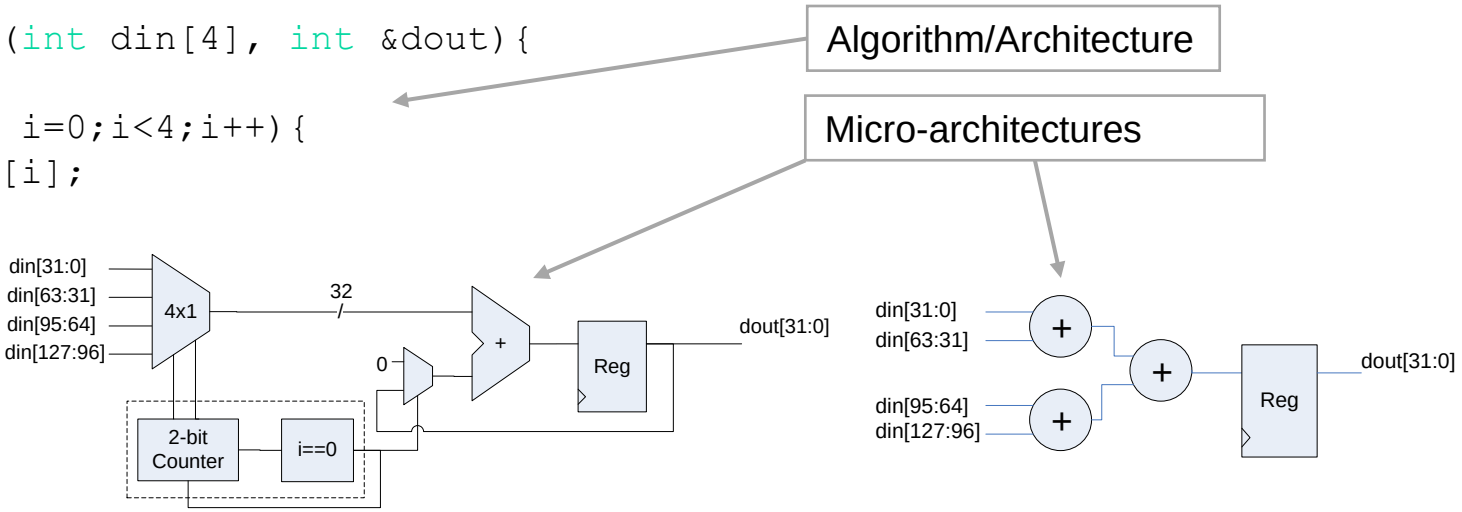
Examples:

- Implementing an array as a memory or as a register file
- Performing four multiplications using one, two or four multipliers
- Pipelining the execution of sequential operations to increase throughput

Microarchitecture determined by Catapult synthesis constraints

- Can be major rewrites in the context of manual RTL design

```
void accumulate4(int din[4], int &dout){
   int acc=0;
   ACCUM:for(int i=0;i<4;i++){
      acc += din[i];
   }
   dout = acc;
}
```



Controlling Parallelism

- Loops are the primary mechanism for applying high level synthesis constraints as well as moving data, or I/O, into and out of an algorithm
  - “for” / “while” / “do … while”
- Loop unrolling constraints provide a way to explore many possible micro-architectures for a given architecture
- A Loop synthesized by HLS is comprised of a counter/fsm controlling a scheduled data path

Loop Unrolling Theory

- Unrolling a loop copies the loop body

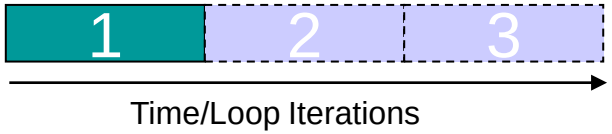
C++ loop

```
for (int i=0;i<3;i++)
{
  <loop body>
}
```

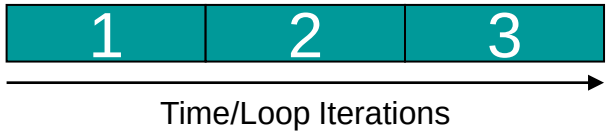
Unrolled loop

```
<i=0 loop body>
<i=1 loop body>
<i=2 loop body>
```

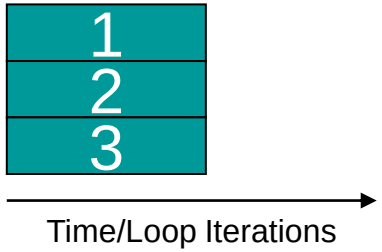
Rolled loop iterations execute sequentially



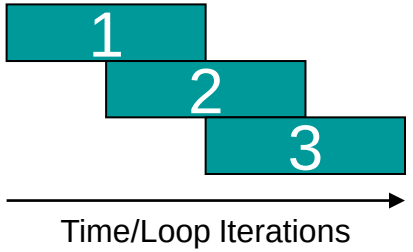
If each iteration is fully dependent on the previous one, unrolling doesn't help:



If there are no dependencies between loop iterations, all loop bodies can execute in parallel:



Often the result is somewhere in the middle:

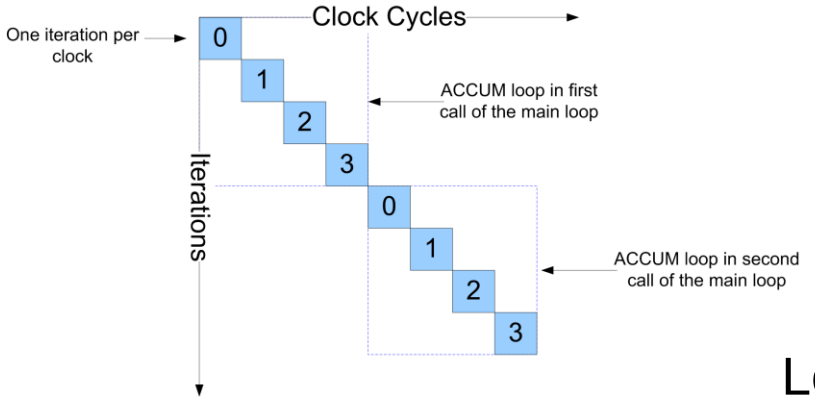


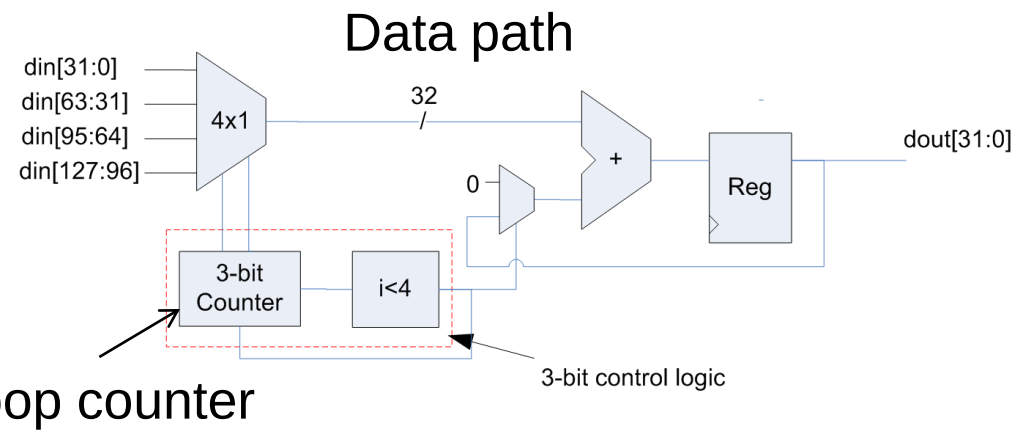
Un-optimized Loop Behavior (Rolled Loops)

If a loop is left “rolled”, each iteration of the loop takes at least one clock cycle to execute in hardware

- A “rolled” loop synthesized by HLS comprises a counter/fsm controlling a scheduled data path
- Provides a way to stream data into and out of the design
- There is an implied “wait until clock” for the rolled loop body

```
void accumulate(int din[4], int &dout){
   int acc=0;
   for(int i=0;i<4;i++){
      acc += din[i];
      <Implied wait-until-clock>
   }
   dout = acc;
}
```

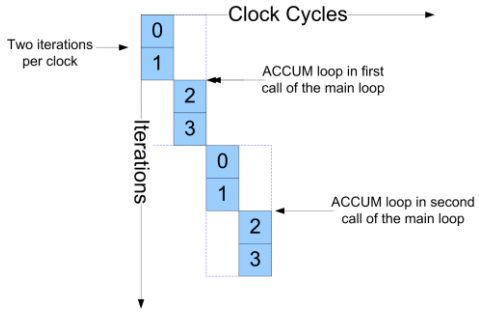


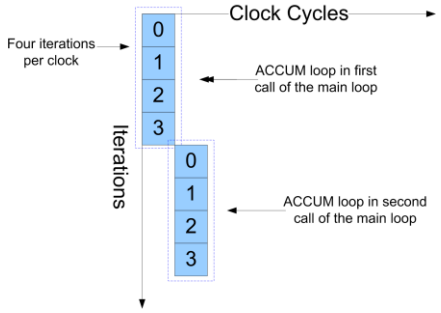


Unrolled Loops

Loops can be partially or fully unrolled

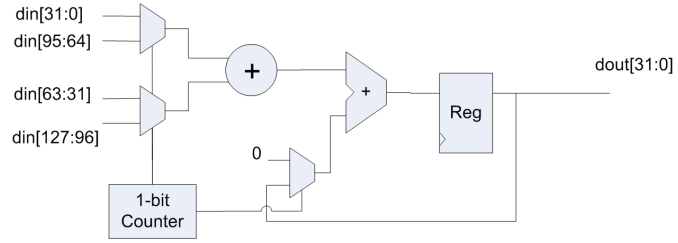
```
void accumulate(int din[4], int &dout){
    int acc=0;
    for(int i=0;i<4;i++){
        acc += din[i];
    }
    dout = acc;
}
```



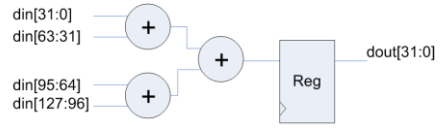


Design Constraints:
Clock freq. slow enough to fit all additions within a single clock cycle

Unroll 2x



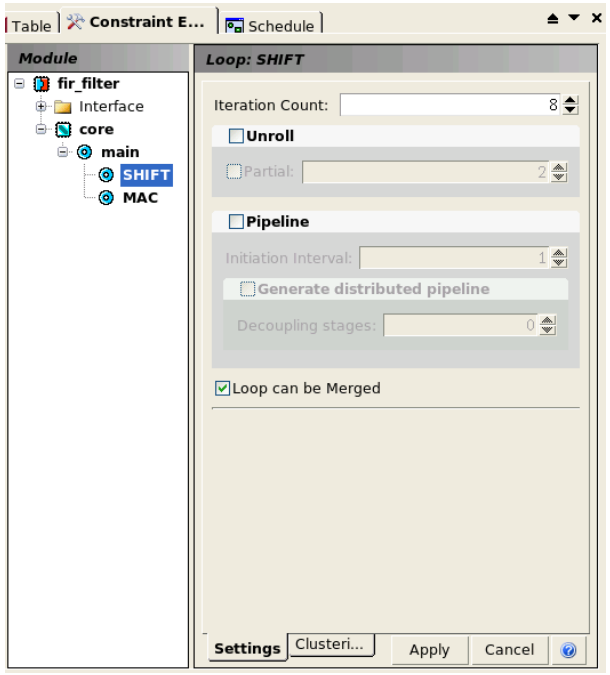
Unroll 4x



Loop Architectural Constraints

Catapult's Architectural constraints views allows you to analyze and constrain loops

- Iteration count
  - Is it correct?
  - Is it known?
- Unroll
  - Partial unroll
- Pipeline
  - Initiation Interval
- Loop Merging



Labelling loops lets you see where they came form in the source

- Double-click to cross-probe to the source

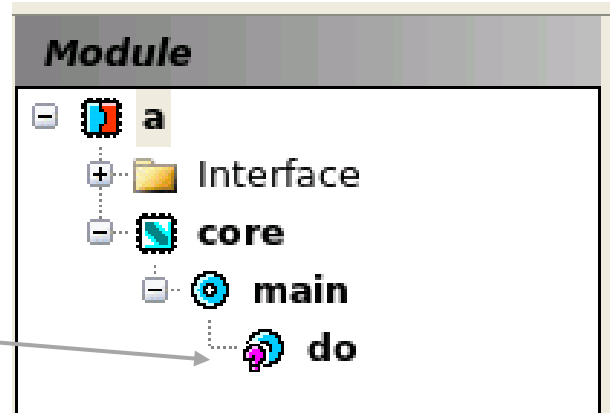
Unbounded Loops

Cannot fully unroll an unbounded loop

Loops with an unknown number of iterations can give bad quality of results

- Loop counter logic will be >32-bits wide

Catapult shows a question mark (“?”)




```
void a (int a[256],
        int &n,
        int &result) {

   int acc   = 0 ;
   int count = 0 ;

   do {
      acc+= a[count] ;
      count++ ;
   } while (count <= n-1) ;

   result = acc;
}
```

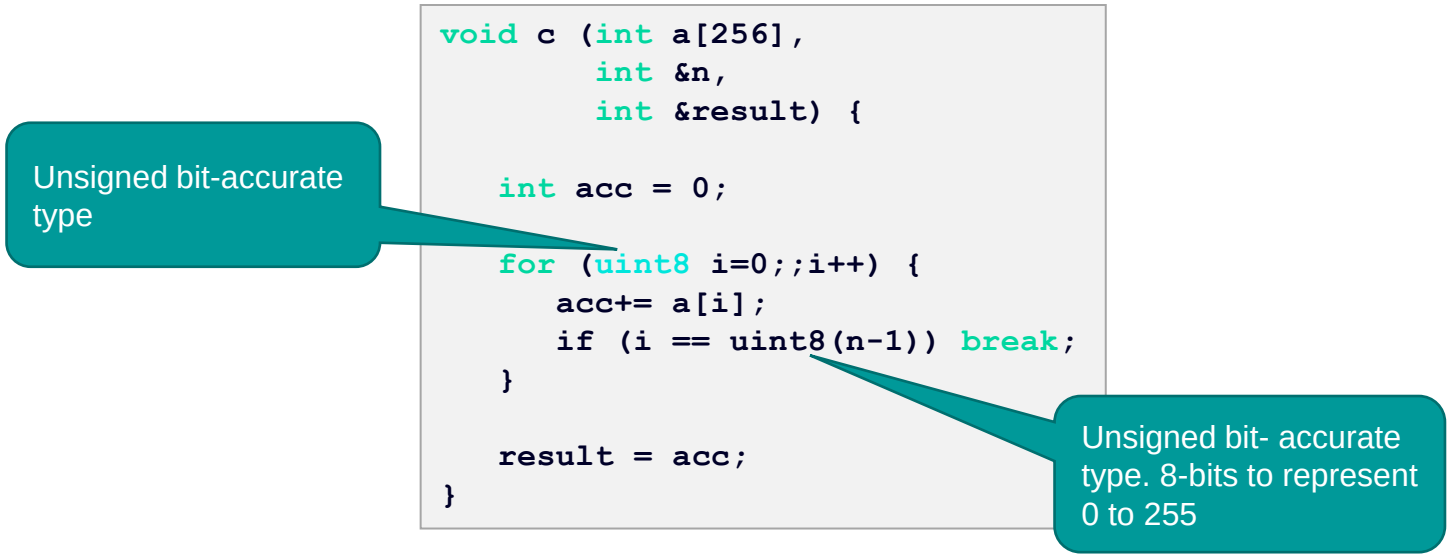
```
void b (int a[256],
        int &n,
        int &result) {

   int acc = 0 ;


   for (int i=0;i<n;i++) {
      acc+= a[i] ;
   }


   result = acc;
}
```

```
void c (int a[256],
        int &n,
        int &result) {

   int acc = 0;


   for (int i=0;;i++) {
      acc+= a[i];
      if (i == n-1)
        break;
   }
   result = acc;
}
```

Re-code Unbounded Loops for Best QofR and Code Coverage

Use unsigned bit-accurate data-type for loop iterator and exit condition

- Better for RTL code coverage



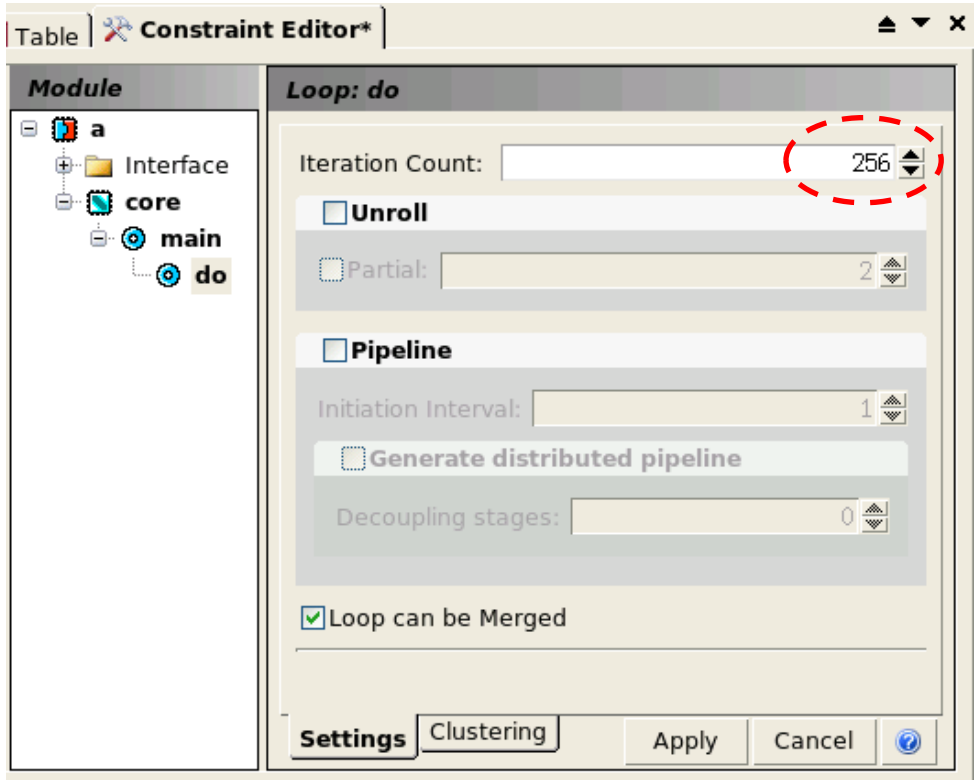
Loop Iteration Count Setting

Always check for “?” signs on your first run

- Try to solve unknown iterations before moving forward to scheduling

Worst case iteration count can be specified in loop settings dialog

- This only affects reporting



Setting Loop Unrolling Constraints

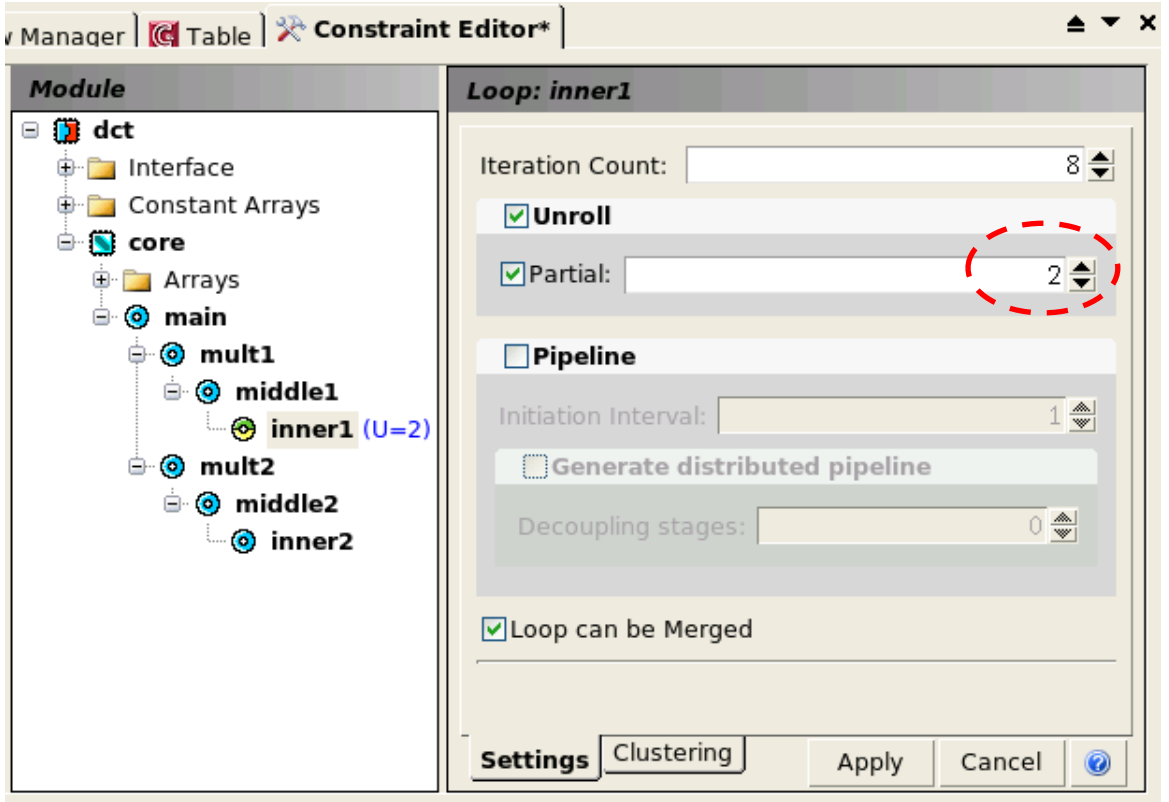
Partial unrolling

- Implement ‘n’ copies in parallel
- Powers of 2, or whole divisors typically work well

Unrolling also controlled in source via pragma on loop

- #pragma unroll <yes or #iterations to unroll>

```
void c (int a[256],
        int &n,
        int &result) {

   int acc = 0;
   #pragma unroll 2
   for (uint8 i=0;;i++) {
      acc+= a[i];
      if (i == uint8(n-1)) break;
   }

   result = acc;
}
```



Loop Unrolling – When & How to Unroll

Be methodical

- Don’t just unroll everything

Always start with the innermost loops of nested loops

- Then work your way out

Look for loops with the following properties

- Small number of iterations
- Small number of operations
- No feedback between loop iterations
- Few limited resources (i.e. RAMS)

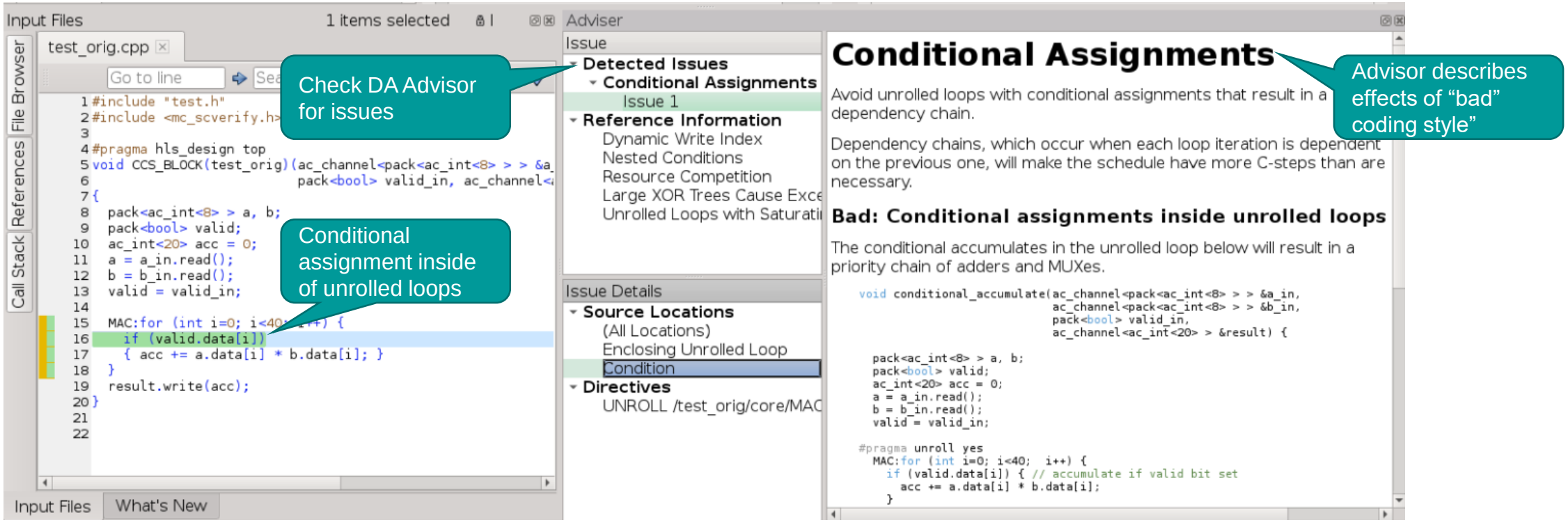
Avoid unrolling a loop with rolled sub-loops

- Creates sequential loops and complex control logic

Avoid Unrolled Loops and Conditional Breaks

Unrolling loops with complex conditions, conditional assignments, or conditional breaks may create a dependency chain

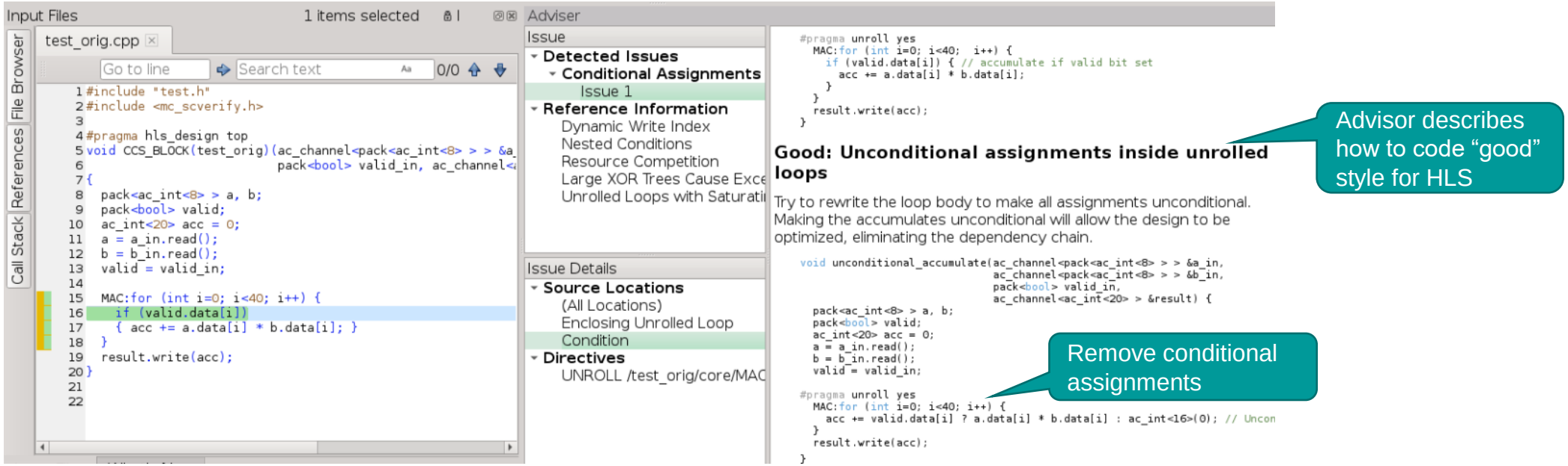
- Breaks automatic tree balancing
- Usually gives poor QofR (longer latency, larger area, longer runtime)
- Always Run **Design Analyzer Advisor** to identify these issues



Unrolled Loops and Conditional Breaks Solutions

Look at DA Advisor example of “good” coding style

- Try to remove conditional breaks when possible
- Use ? Operator to make assignments unconditional
  - This will remove the dependency chain
  - Unrolled expression can be balanced and optimized
    - Better area and performance



Avoid Unrolling Loops with Dynamic Write Indexing

Using a dynamic write index into an array mapped to registers that is inside of an unrolled loop is VERY BAD!!!

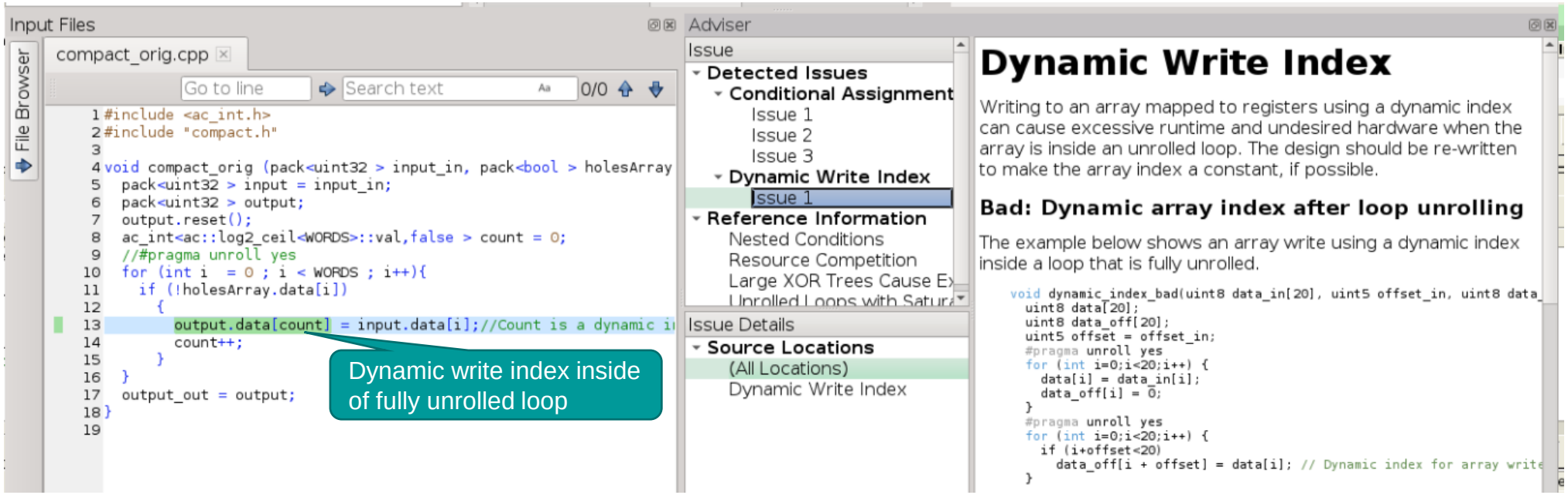
- Will result in **long runtime and poor QofR**

Be explicit, make the index constant when possible

Fully unroll loops to make the loop iterator constant when possible

Convert dynamic offsets to constant offsets

Use DA Advisor to identify and fix these issues



Recoding Dynamic Write Index Inside of Unrolled Loops

Make the write index constant when possible

- A fully unrolled loop iterator becomes constant
- Think about what the hardware should do and express that in the C++

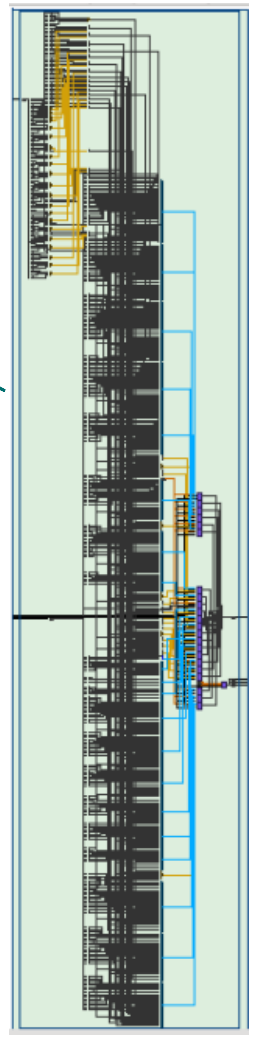
Dynamic offset

```
2   void test_orig(int din[22],  uint6 offset,
3              int dout[22]){
4     int regs[22];
5
6     //Loop is fully unrolled
7     for(int i=0;i<22;i++)
8       if(i+offset < 22)
9         regs[i + offset] = din[i];
10    //Loop is fully unrolled
11    for(int i=0;i<22;i++)
12      dout[i] = regs[i];
13
14  }
```

```
4   void test(int din[22],  uint6 offset,
5             int dout[22]){
6     static int regs[22];
7
8     #pragma unroll yes
9     for(int i=0;i<22;i++){
10      #pragma unroll yes
11      for(int j=0;j<22;j++){
12        if((j==offset) & (j+i<22)){
13          regs[j+i] = din[i];
14        }
15      }
16    }
17
18    //Loop is fully unroll
19    #pragma unroll yes
20    for(int i=0;i<22;i++
21      dout[i] = regs[i];
22
23  }
```

Hardware is simplified

Write index becomes constant after loop unrolling



Better area and performance

Report: General

Solution	Latency...	Latency...	Throug...	Throug...	Total Area	Slack
test_orig_wrapper.v1 (extract)	1	2.00	1	2.00	13450.26	1.23
test_wrapper.v1 (extract)	**1**	**2.00**	**1**	**2.00**	**11653.87**	**1.76**

Loop Merging and Sequential Loops

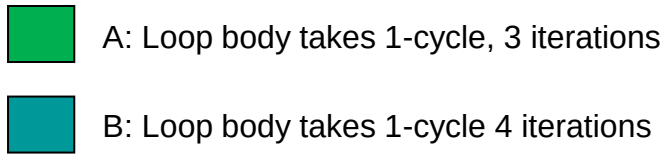
Loop Merging implements two independent sequential loops in parallel

The resulting loop has iterations equal to the maximum of it's members

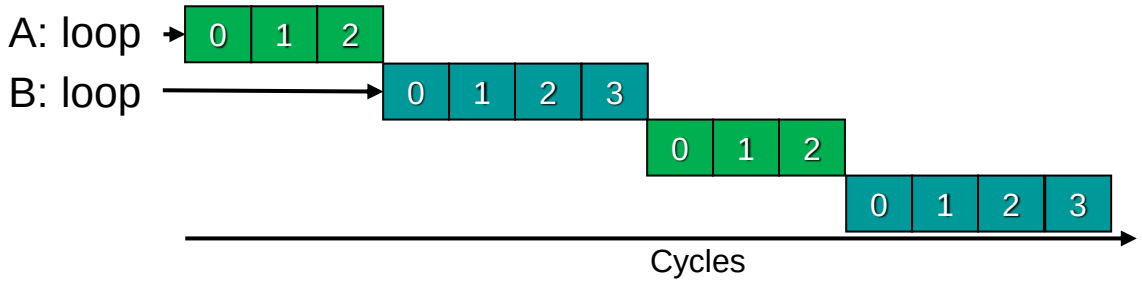
Loops that can't be merged execute sequentially in hardware

- Sometimes necessary to manual merge loops in the C++ to get them to execute in parallel to meet performance goals
- May need multi-block design

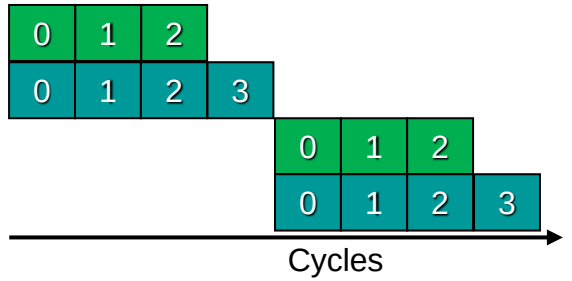
```
void loop_merging1 (unsigned char &mod,
                    char a[3], char b[3],
                    char c[4], char d[4]) {
    A:for (int i = 0; i < 3; i++)
        b[i] = (a[i] * mod) >> 8;
    B:for (int j = 0; j < 4; j++)
        d[j] = (c[j] * mod) >> 8;
}
```



Two sequential Rolled Loops that can't be merged



Two sequential Merged Rolled Loops



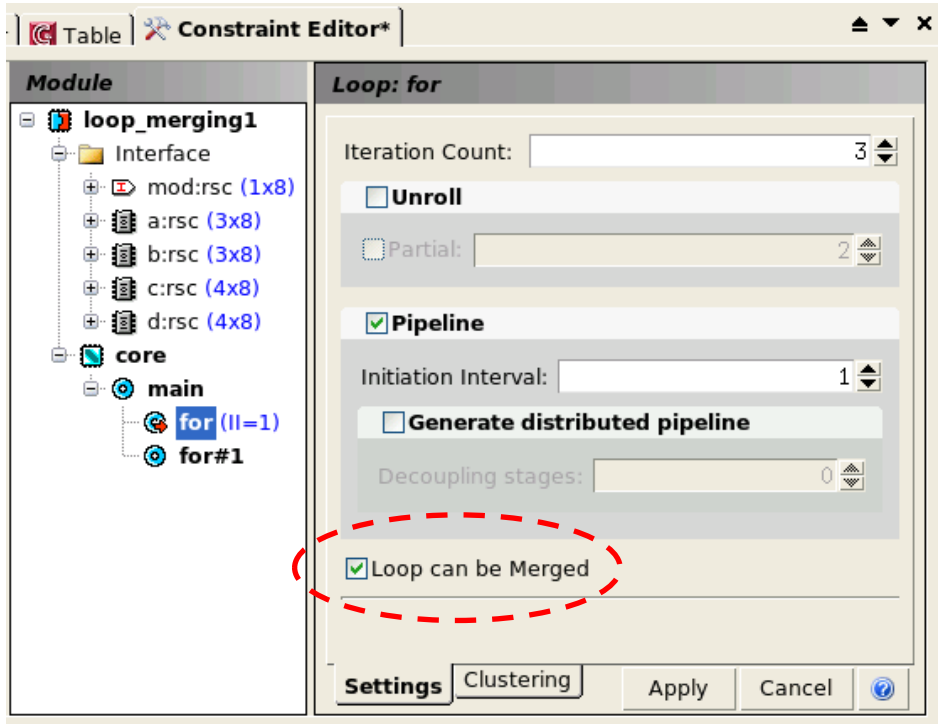
Loop Merging Setting

On by default for all loops

Pipelining settings will only be saved from the loop with the smallest II

Unknown iterations on a loop will cause resulting loop to have unknown iterations

A fully unrolled loop will not be merged



Loop Pipelining

SIEMENS

HLS Designs Always Have at Least One Loop

The top-level function call itself is treated as an infinitely running loop

- Also known as the “main” loop
- Can be pipelined

```
void simple_function ( <function interface variables> ) {
  <function body>

  <Implied wait-until-clock>
}
```

Infinite loop

```
module simple_function ( <module ports> );
  always@(posedge clk)
    begin
      <module body>
    end
endmodule
```

Hardware runs forever

Loop Pipelining

"Loop Pipelining" allows a new iteration of a loop to be started before the current iteration has finished

“main loop” can be pipelined


```
void accumulate(int a, int b, int c, int d, int &dout){
   int t1,t2;
   t1   = a  + b;
   t2   = t1 + c;
   dout = t2 + d;
}
```

"Loop pipelining" allows the execution of the loop iterations to be overlapped, increasing the design performance by running them in parallel

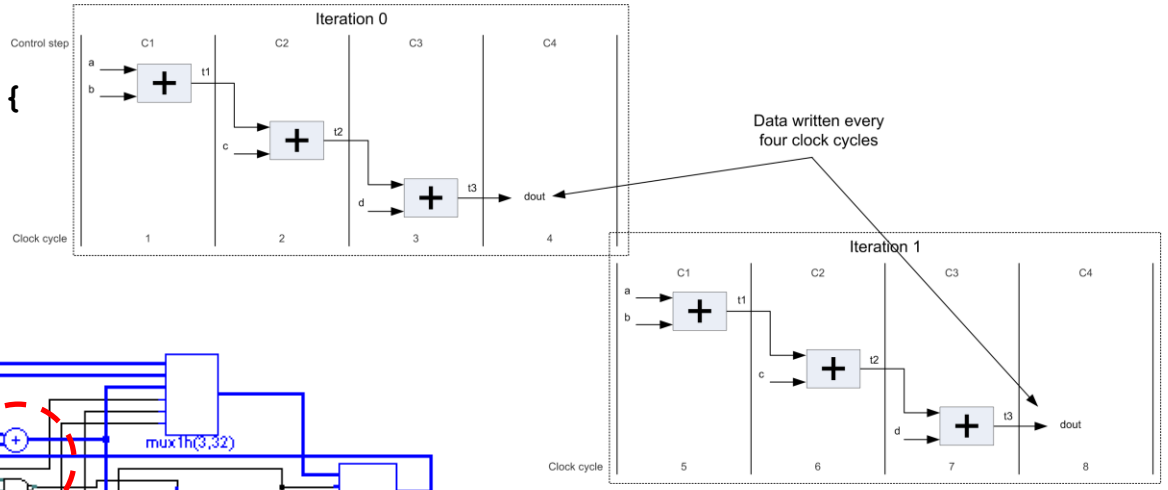
- Amount of overlap is specified by the "Initiation Interval (II)“

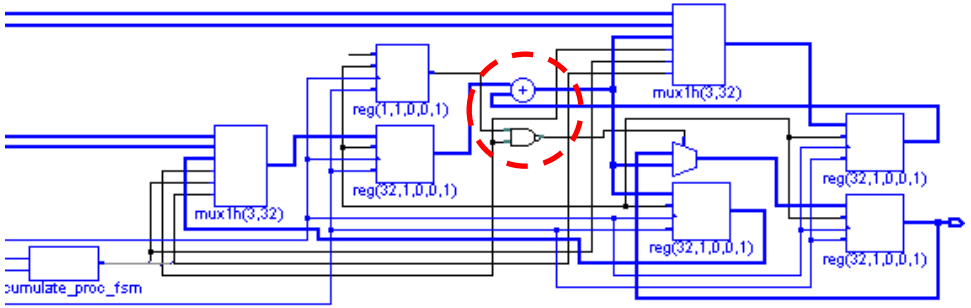
Loop Pipelining

Designs with no loop pipelining

- Single pipeline stage because there's no overlap between execution of each iteration of the main loop
- Results in data written every four clock cycles
- Because there is no overlap of any operation only a single adder is required if sharing reduces overall area
- Latency = 3, Throughput = 4

```
void accumulate(int a, int b,
        int c, int d, int &dout){
    int t1,t2;
    t1   = a  + b;
    t2   = t1 + c;
    dout = t2 + d;
}
```

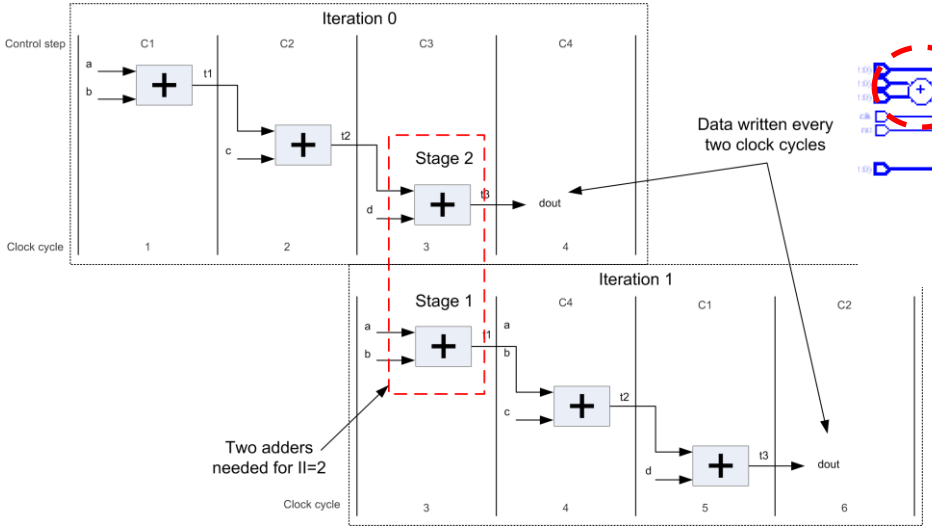


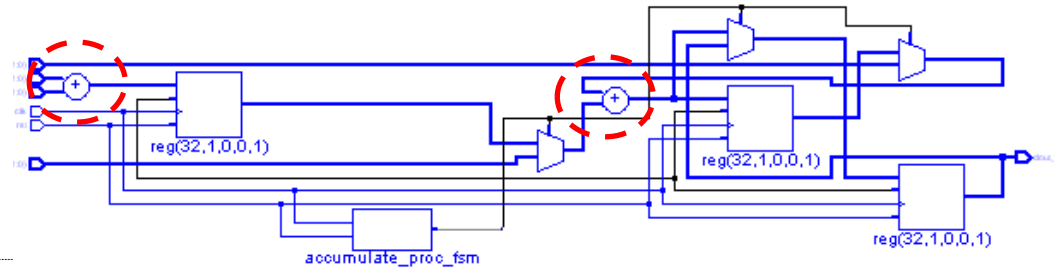


Loop Pipelining

Pipelining the main loop with an II=2 results in a new iteration started every two clock cycles

- Two pipeline stages
- Iteration one is started in C3 while iteration 0 is computing "t3 = t2 + d".
- Since iteration one is computing "t1 = a + b" it can be seen that two adders are required for the two pipeline stages

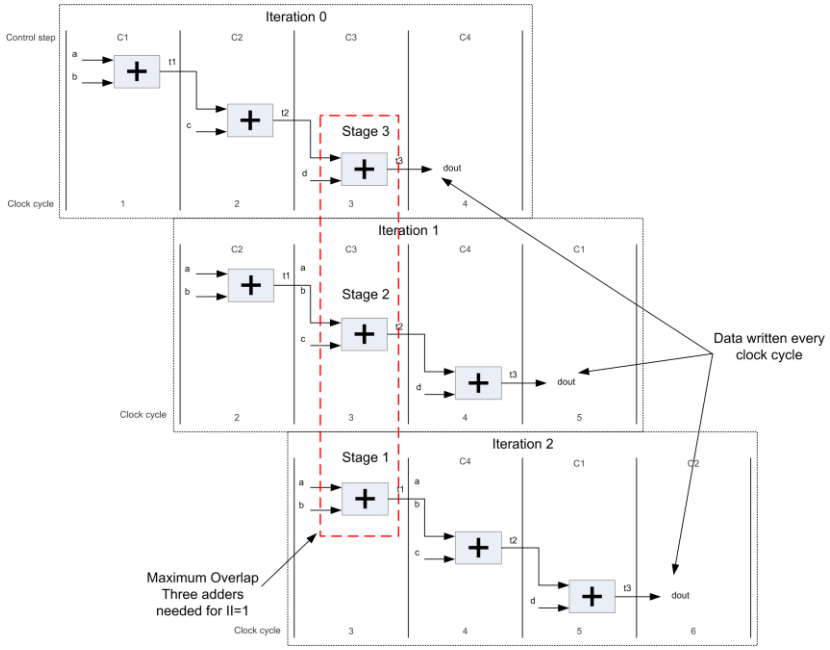


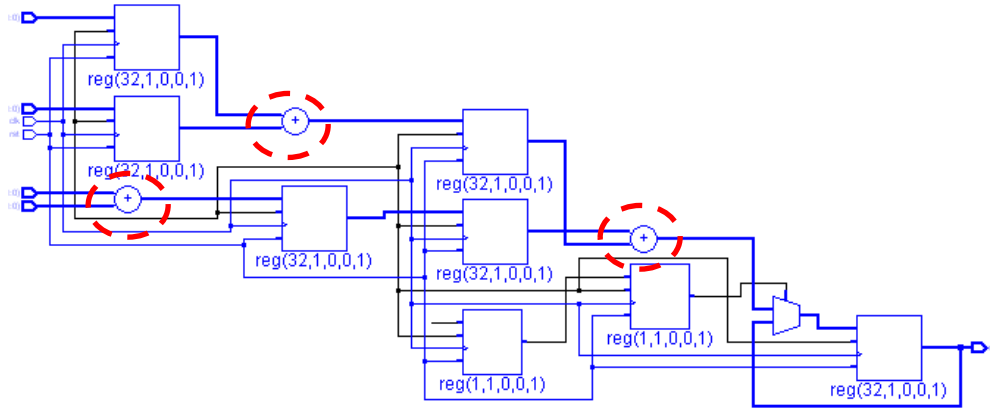


Loop Pipelining

Pipelining the main loop with an II=1 results in a new iteration started every clock cycle

- Three pipeline stages
- Iteration one is started in C2 and iteration 2 is started in C3
- Looking at C3 shows that three adders are required in hardware since all three pipeline stages are active



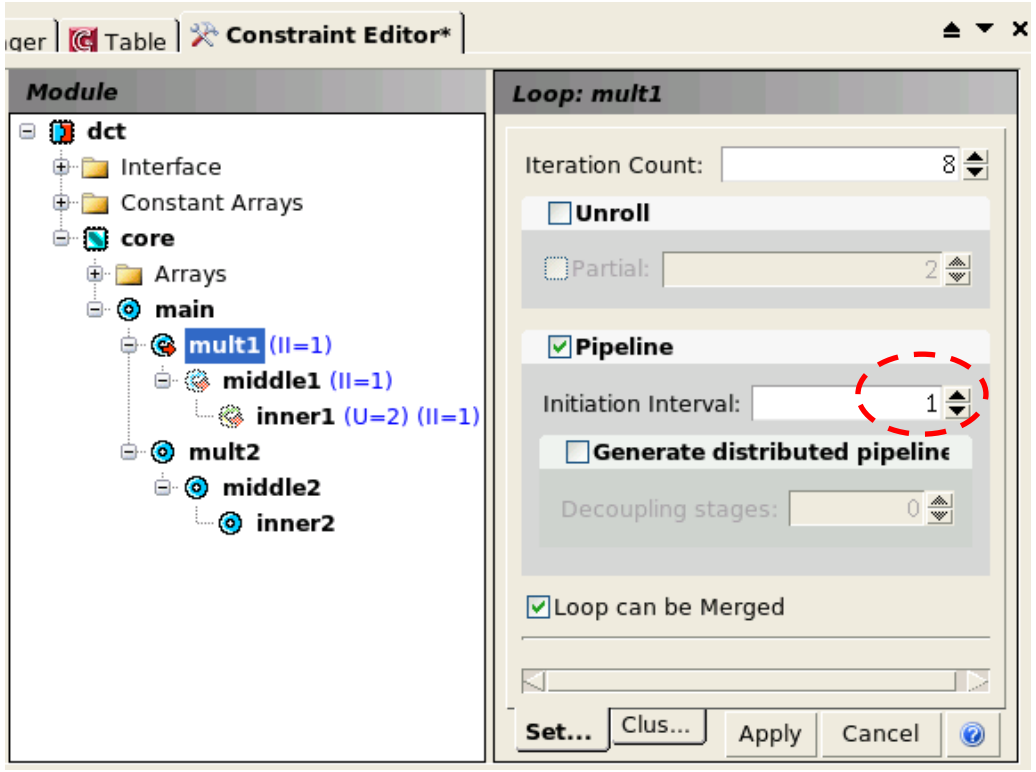


Loop Pipelining – Architectural Constraints

The red arrow on the loop indicates that it is pipelined

- Initiation Interval (II) is also shown

Sub-loops are greyed out due to nested loop pipelining

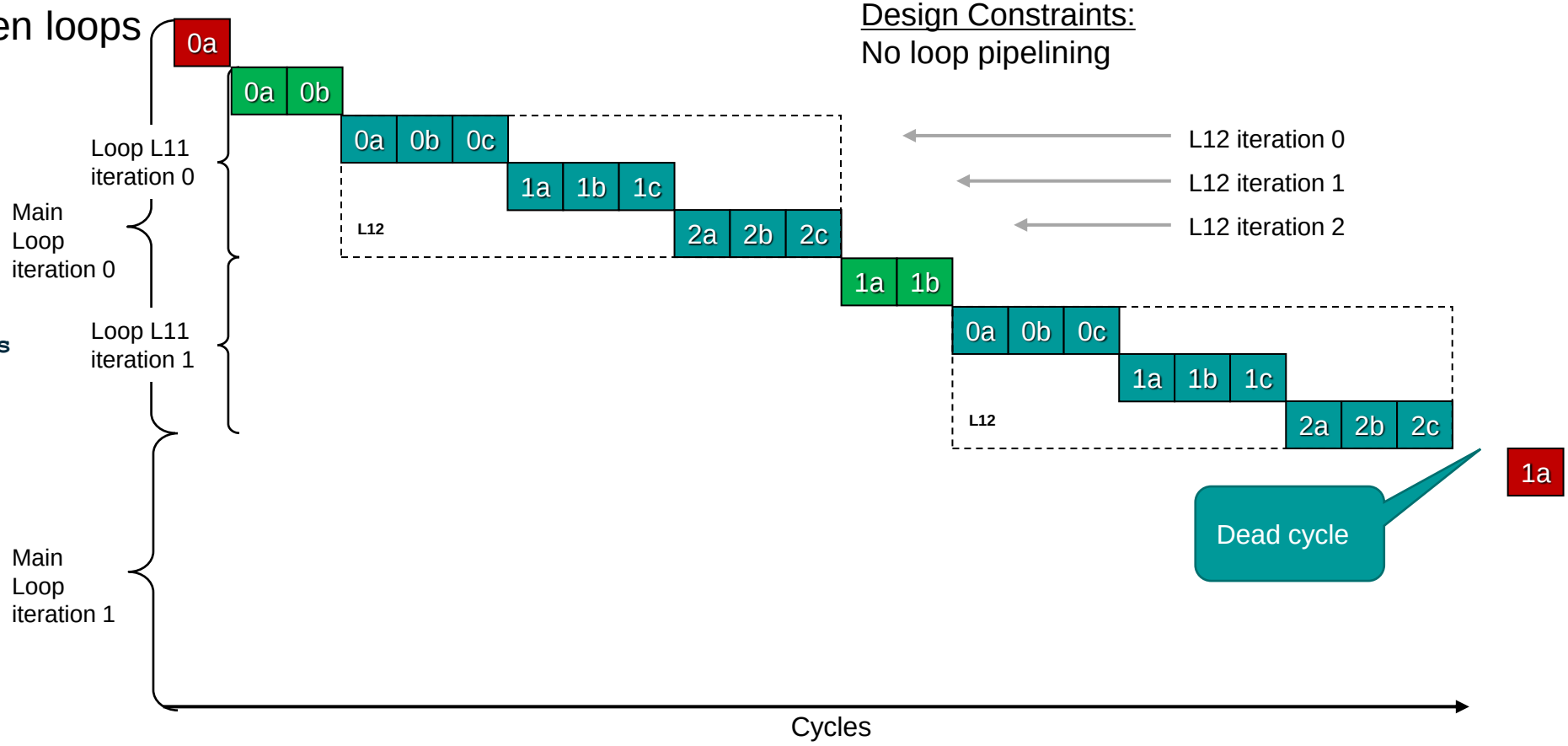


Nested Loops Without Pipelining

Un-pipelined nested loops will have low performance

- No overlap of loop bodies
- More resource sharing
- May have dead cycles between loops

```
MAIN LOOP{
     // loop body takes 1 cycles
     // (a) clock cycles
    L11: for (i=0; i<2; i++){
        // loop body takes 2 cycles
        // (a and b) clock cycles
       L12: for (j=0; j<3; j++)
          // loop body takes 3
          // (a, b & c) clock cycles
}
```

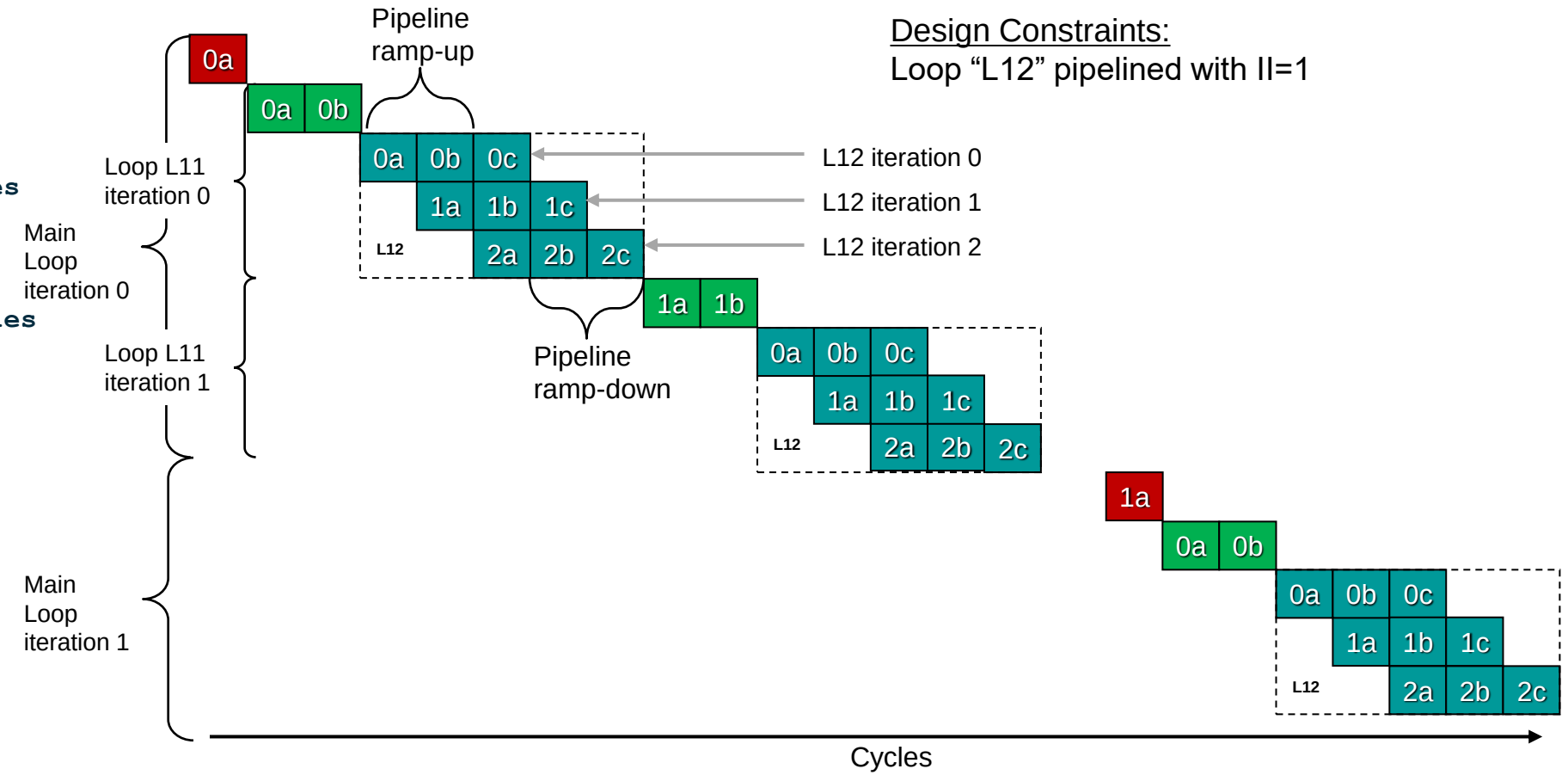


Nested Loop Pipelining

Pipelining the innermost loop will allow iterations of the innermost loop body to be overlapped

- Pipeline must ramp-up and ramp-down
- Extra cycles will be spent executing the outer loops before the inner loop can restart

```
MAIN LOOP{
     // loop body takes 1 cycles
     // (a) clock cycles
    L11: for (i=0; i<2; i++){
        // loop body takes 2 cycles
        // (a and b) clock cycles
       L12: for (j=0; j<3; j++)
             // loop body takes 3
          // (a, b & c) clock cycles
}
```

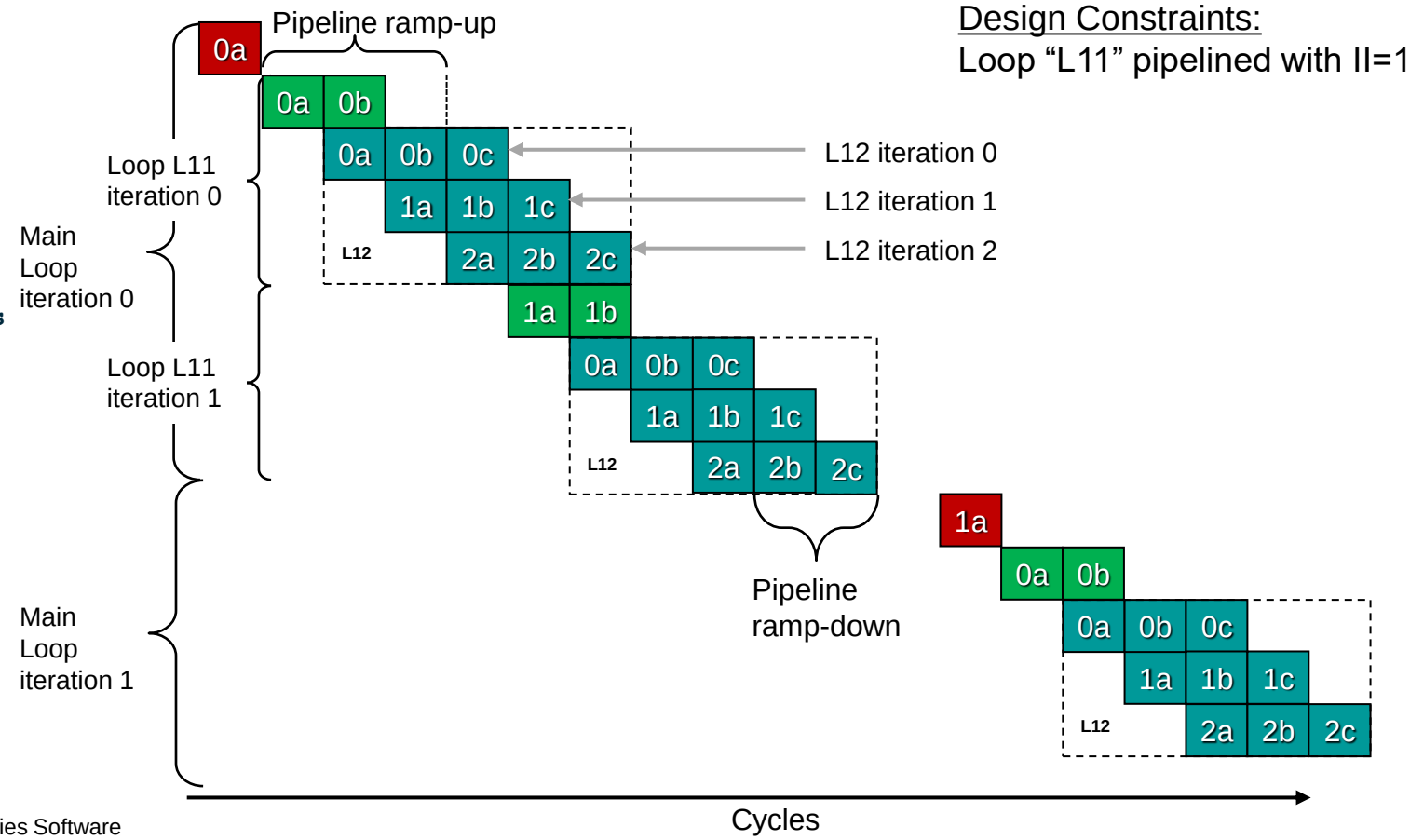


Nested Loop Pipelining

Pipelining a set of nested loops will result in them being flattened into a single loop structure

- II is essentially applied to innermost loop, out loops run conditionally when inner loop exits
- See “Loop Flattening” in chapter 4 of the HLS Blue Book

```
MAIN LOOP{
     // loop body takes 1 cycles
     // (a) clock cycles
    L11: for (i=0; i<2; i++){
        // loop body takes 2 cycles
        // (a and b) clock cycles
       L12: for (j=0; j<3; j++)
             // loop body takes 3
          // (a, b & c) clock cycles
}
```



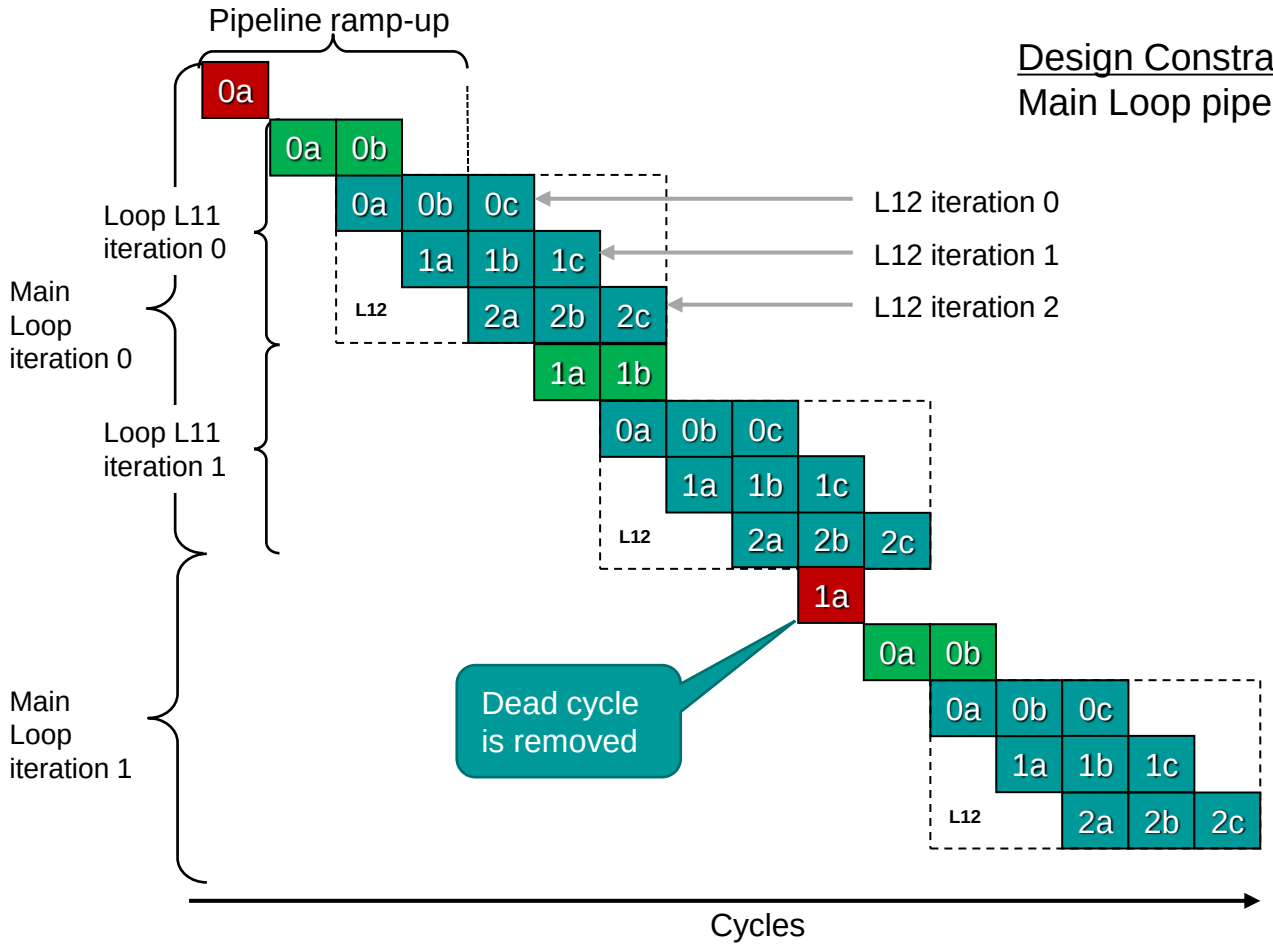
Design Constraints:
Loop “L11” pipelined with II=1

Nested Loop Pipelining

Pipelining the Main Loop will give highest performance

- Pipeline will ramp-up and run forever

```
MAIN LOOP{
      // loop body takes 1 cycles
      // (a) clock cycles
    L11: for (i=0; i<2; i++){
        // loop body takes 2 cycles
        // (a and b) clock cycles
       L12: for (j=0; j<3; j++)
             // loop body takes 3
          // (a, b & c) clock cycles
}
```



Design Constraints:
Main Loop pipelined with II=1

When and How to Pipeline

Use pipelining when you want to achieve a given throughput/data rate

Use pipelining to reduce latency for loops

Use pipelining when full unrolling is not practical

Always start with the innermost loops then work your way out

- Sometimes area may get smaller as you work your way out

Pipelining Initiation Interval can be increased to reduce area

Nested loop pipelining can be used to improve system performance (throughput and latency)

Step-by-Step Lab1 FIR

Loop unrolling & pipelining

Working With Memories and Memory Architecture

SIEMENS

Introduction

Catapult allows C++ arrays to be mapped into memories

- Automatically or under user direction via constraints

Memory libraries must be created to tell Catapult about the memory behavior

- Singleport, dualport, write mask, bist, etc

Many optimizations available

- Changing word width
- Combining multiple arrays onto the same memory
- Address mapping

Good memory architecture is essential in the C++ when using memories

- Memory accesses can be performance bottlenecks

Constant arrays mapped to ROM/LUT

Memory and Register Thresholds

Memory Mapping Threshold

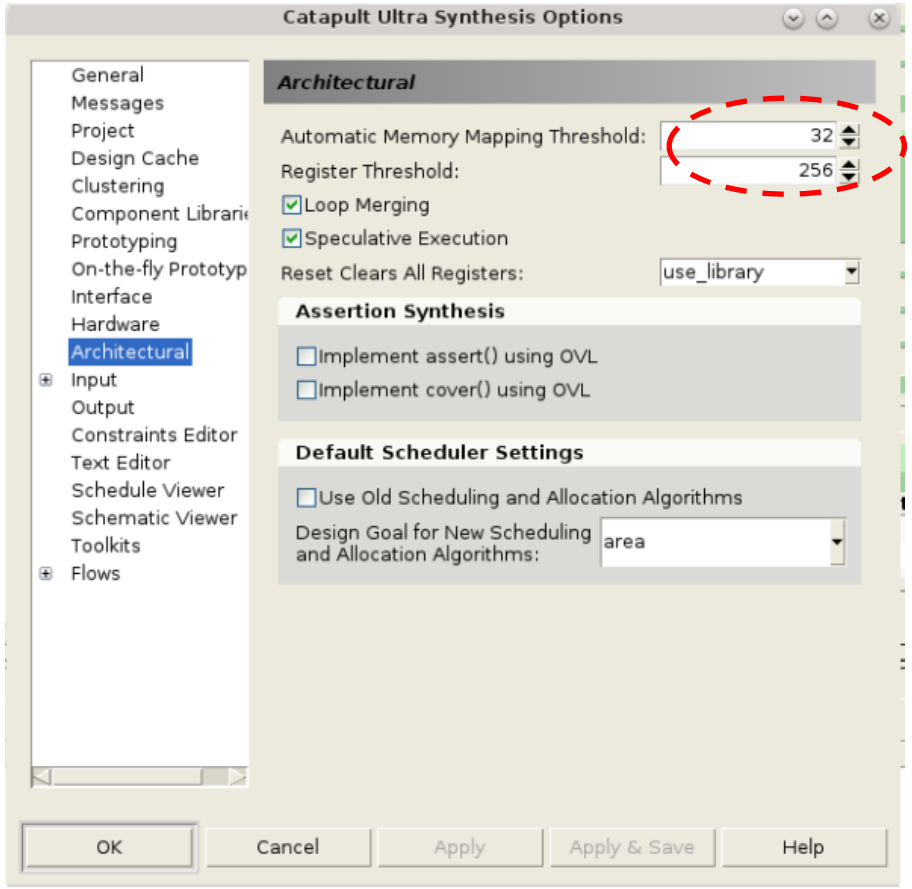
- Arrays with more than a set number of elements are automatically mapped to RAM/ROM if available

Register Threshold

- Arrays mapped to registers with more than a set number of elements will cause an error
- Prevents long runtimes

Set under

- Tools->Set Options->Project Initialization->Architectural



Memory Constraints

Memories mapped in Architectural Constraints

- Resource displayed as grey “chip” icon

Byte Enables

- Only used when memory supports masking

Stage Replication

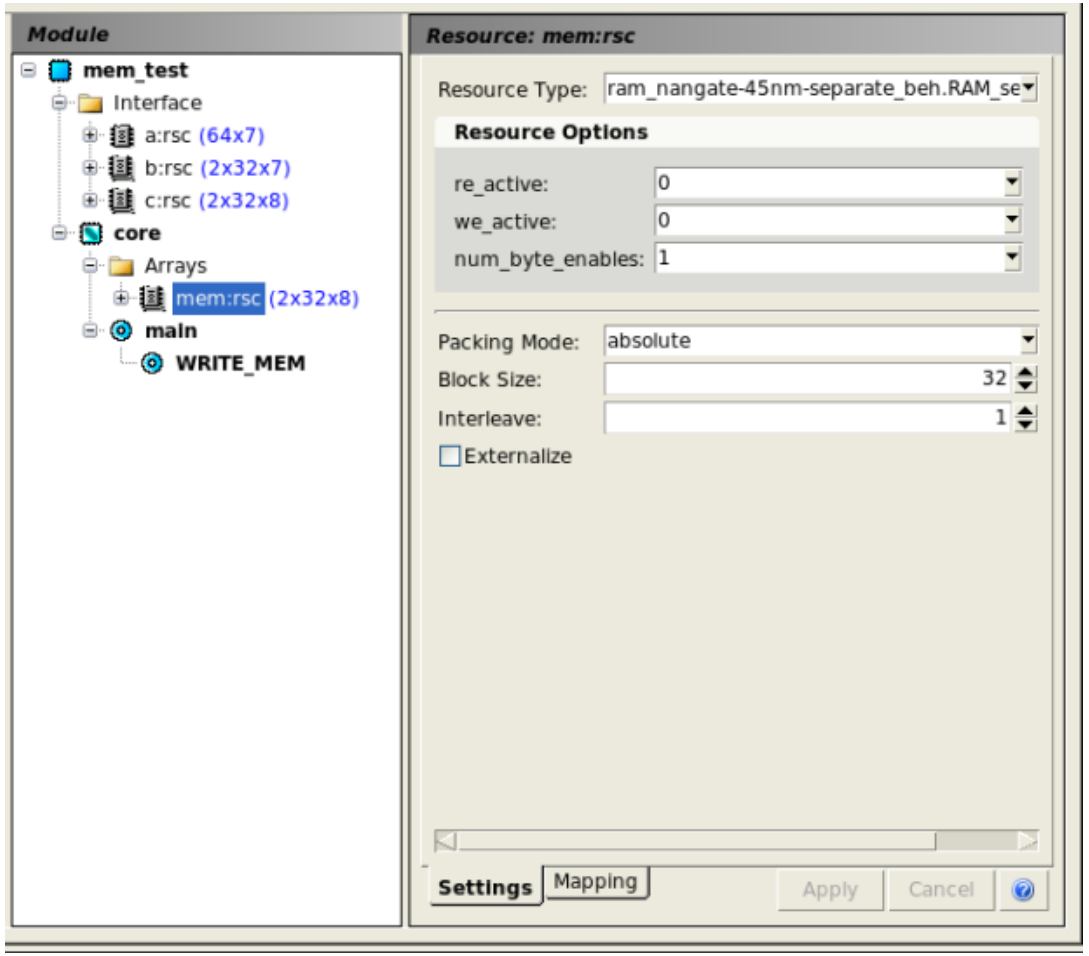
- Used for ping-pong memory architecture

Block & Interleave

Externalize

- Push internal memory to RTL interface

See Catapult DOCs for detailed information



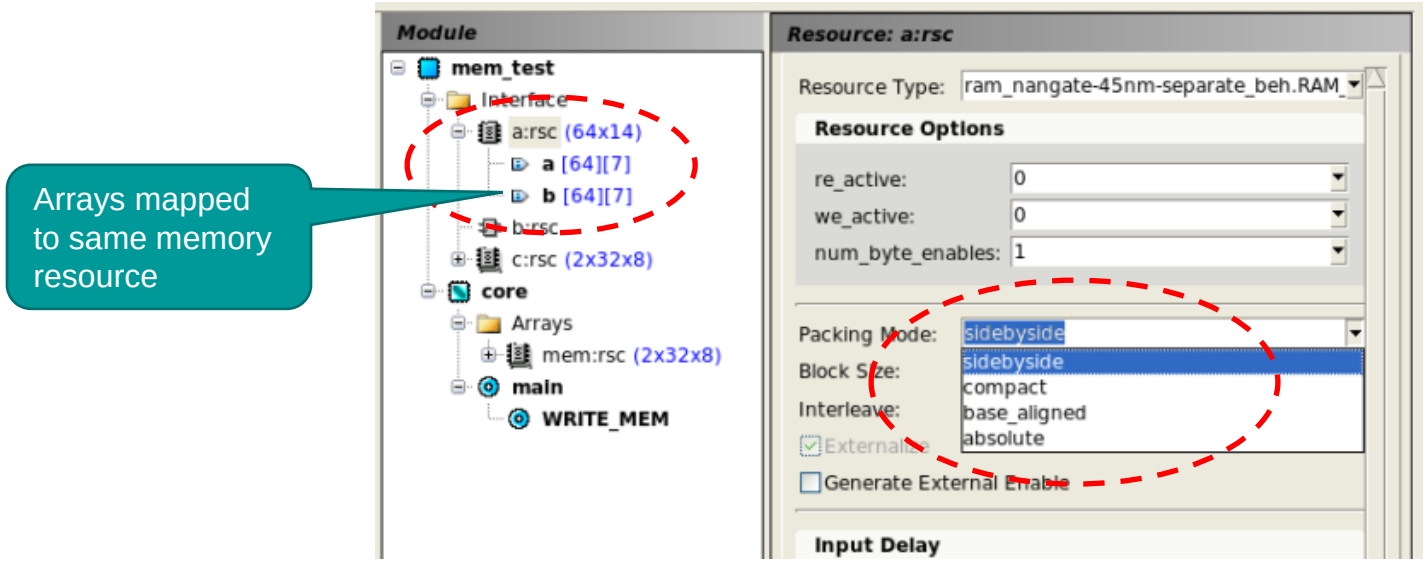
Memory Mapping

Variables mapped to the same Resource are packed using selectable packing modes:

- Sidebyside
  - Concatenates words together
- Compact
  - One after the other
- Base Aligned
  - Power of 2 alignments for efficient indexing
- Absolute
  - Attempts to preserve offsets and base bits but defaults to compact

Resource mapping can be very useful for consolidating RAM usage

- RGB, Real/Imaginary data
- Video line buffer architectures



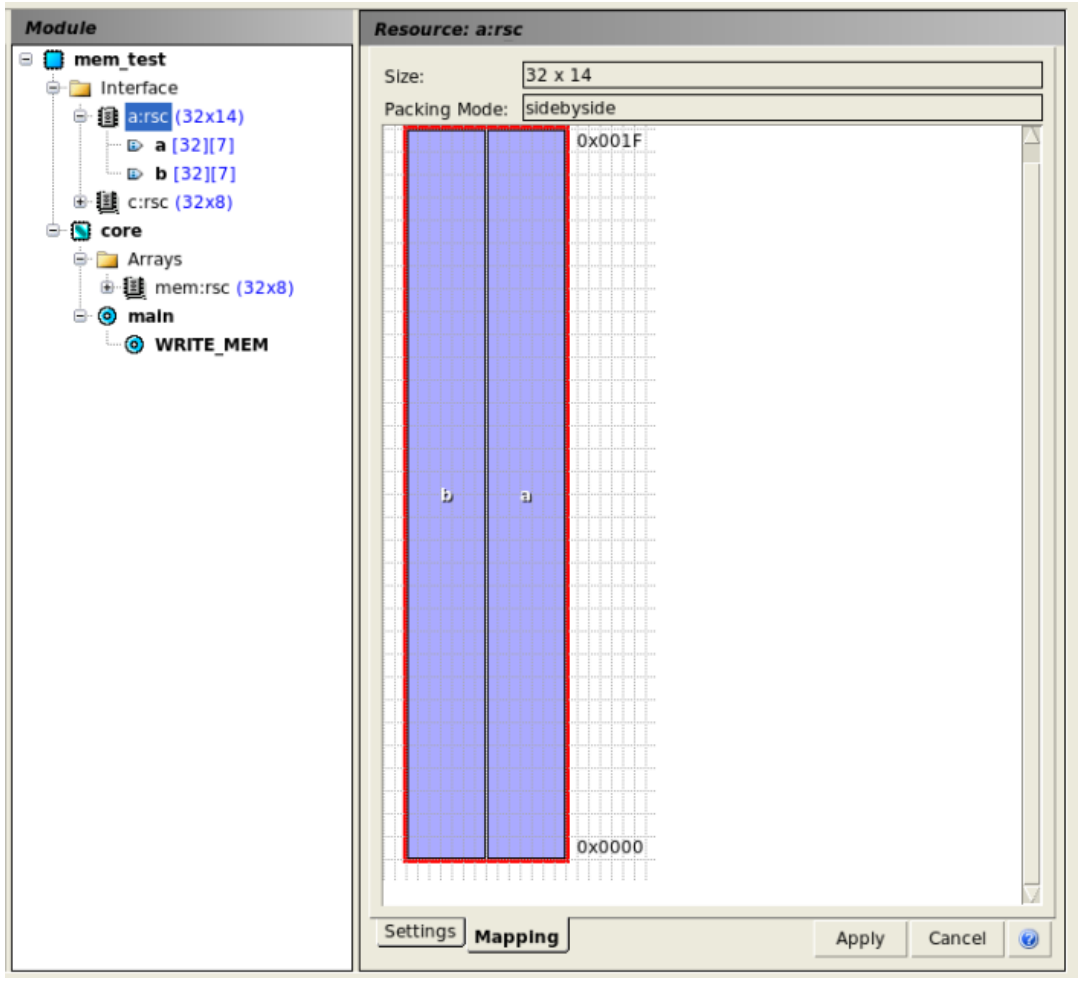
Mapping Arrays to Memories

Select the Mapping tab on a resource to see how the arrays are mapped

- View only works for small memories

Here we see “sidebyside”

- 7-bit “a”
- 7-bit “b”



Memory Mapping Constraints

Word Width

- Change width storage to match bandwidth read/write

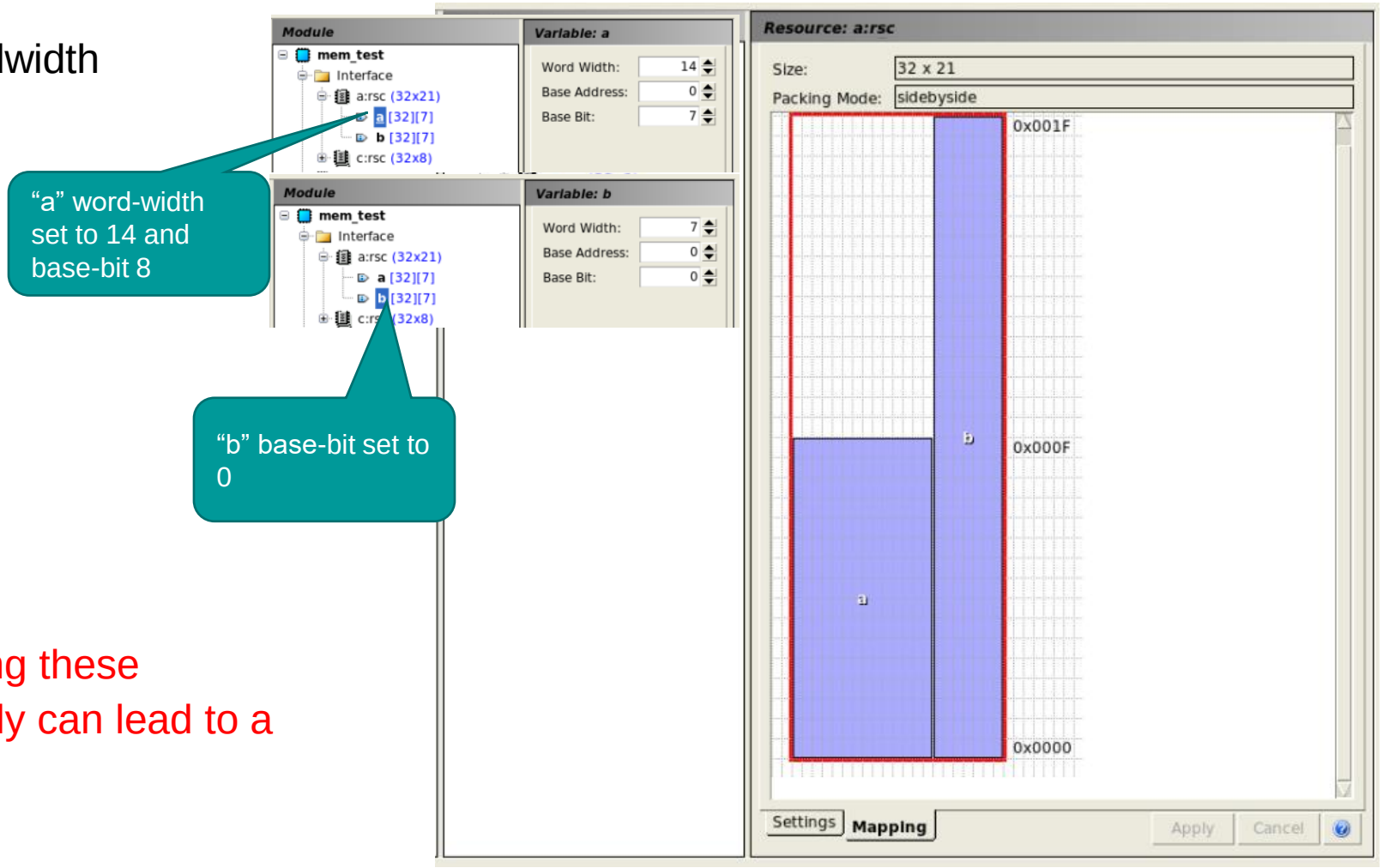
Base Address

- Move address offset in resource

Base Bit

- Specify LSB position in resource

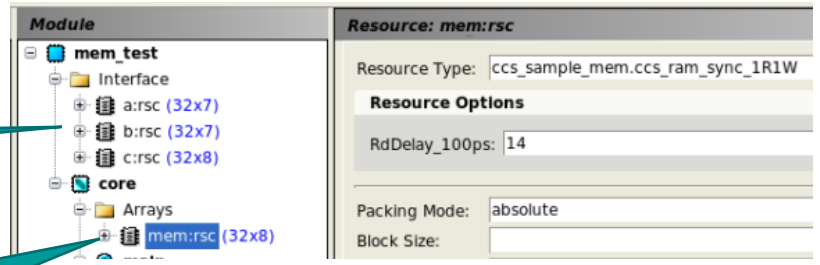
NOTE: Care must be used when setting these constraints. Setting them incorrectly can lead to a non-functioning design



Effects of Multiple Accesses to an Array Mapped to Memory

Multiple accesses to an array mapped to memory can limit design performance

"a, b, and c" mapped to singleport RAM



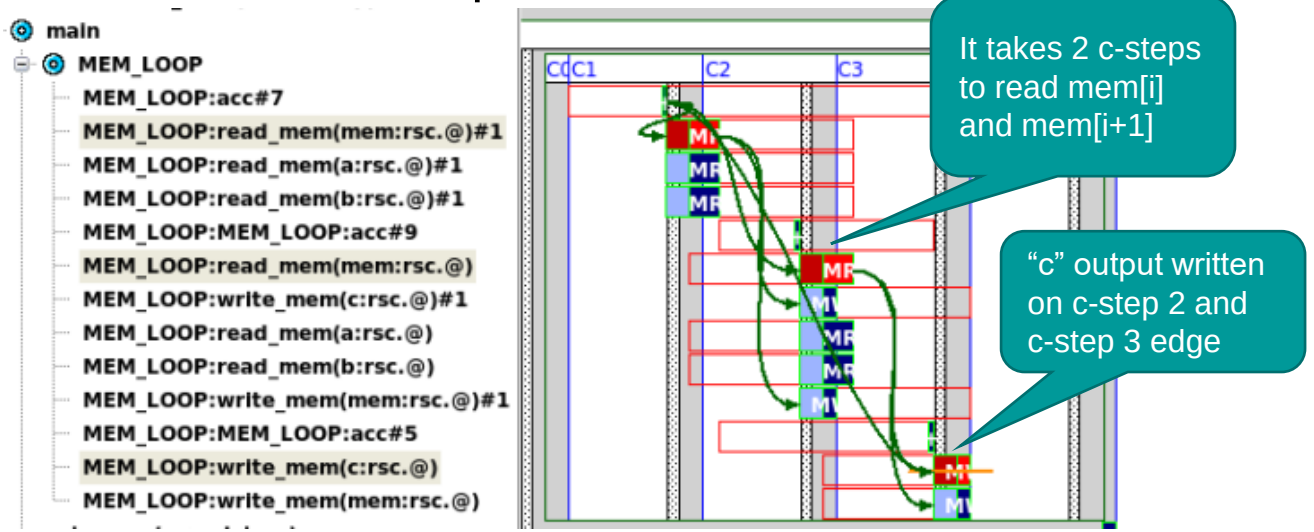
"mem" mapped to RAM with 1-read and 1-write port

```
void mem_test(ac_int<7,false> a[32], ac_int<7,false> b[32],
              ac_int<8,false> c[32]){
  static ac_int<8,false> mem[32];

  MEM_LOOP: for(int i=0;i<32;i+=2){
    c[i] = mem[i];
    c[i+1] = mem[i+1];
    mem[i] = a[i] + b[i];
    mem[i+1] = a[i+1] + b[i+1];
  }
}
```

Two reads of "mem"

Two writes to"mem"

Catapult Gantt Chart



It takes 2 c-steps to read mem[i] and mem[i+1]

"c" output written on c-step 2 and c-step 3 edge

Using Word-width Constraints to Merge Memory Accesses

Used to merge multiple array accesses to the same memory

- Usually to improve performance

Three conditions required for memory merging to happen

- Memory accesses must be sequential
- Sequential accesses must start at even word boundaries
- Memory accesses must be in same conditions

Need to consider all cases where the array is used

- Read-modify-write of a memory due to increasing the word-width can limit performance

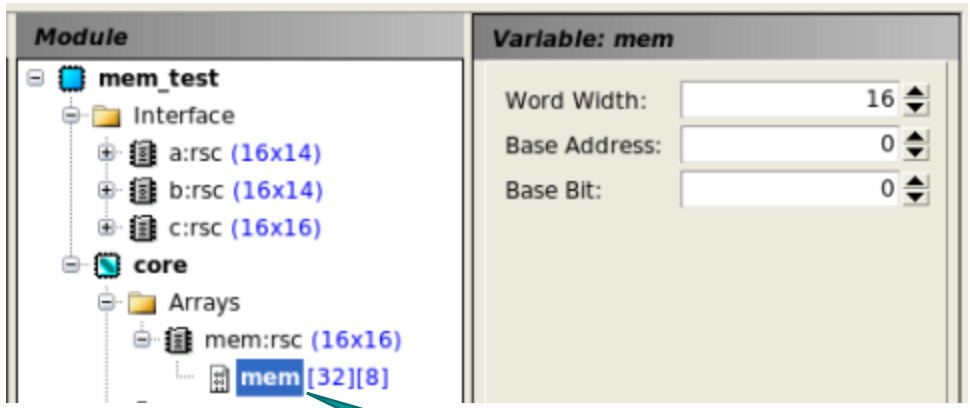
Using Word-width Constraints to Merge Memory Accesses

```
void mem_test(ac_int<7,false> a[32], ac_int<7,false> b[32],
              ac_int<8,false> c[32]){
  static ac_int<8,false> mem[32];

  MEM_LOOP: for(int i=0;i<32;i+=2){
    c[i] = mem[i];
    c[i+1] = mem[i+1];
    mem[i] = a[i] + b[i];
    mem[i+1] = a[i+1] + b[i+1];
  }
}
```

Two sequential reads of “mem”

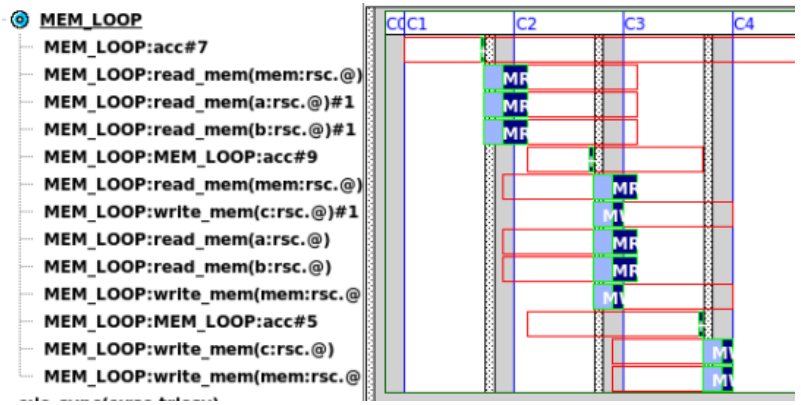
Two sequential writes to “mem”

Reads and writes of “mem” start on even word boundary and are in the same condition

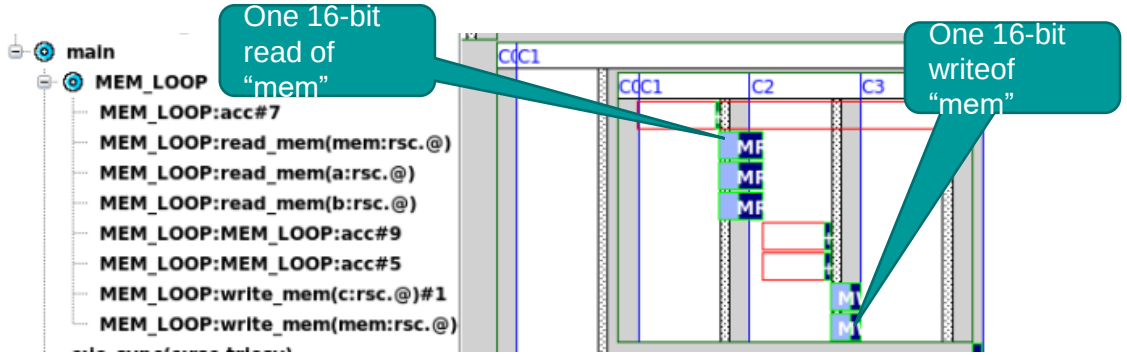


Double the word-width of “mem”, “a”, “b”, and “c”

Catapult Gantt Chart Before Word-width Constraint



Catapult Gantt Chart After Word-width Constraint



One 16-bit read of “mem”

One 16-bit writeof “mem”

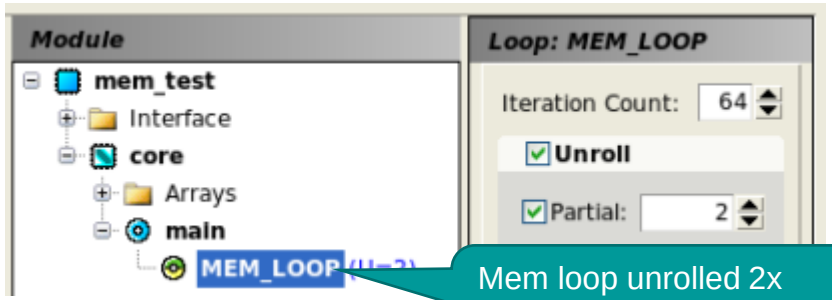
Coding Style that Breaks Automatic Memory Merging

Reads and writes to the same array mapped to a memory in different conditions can break memory merging

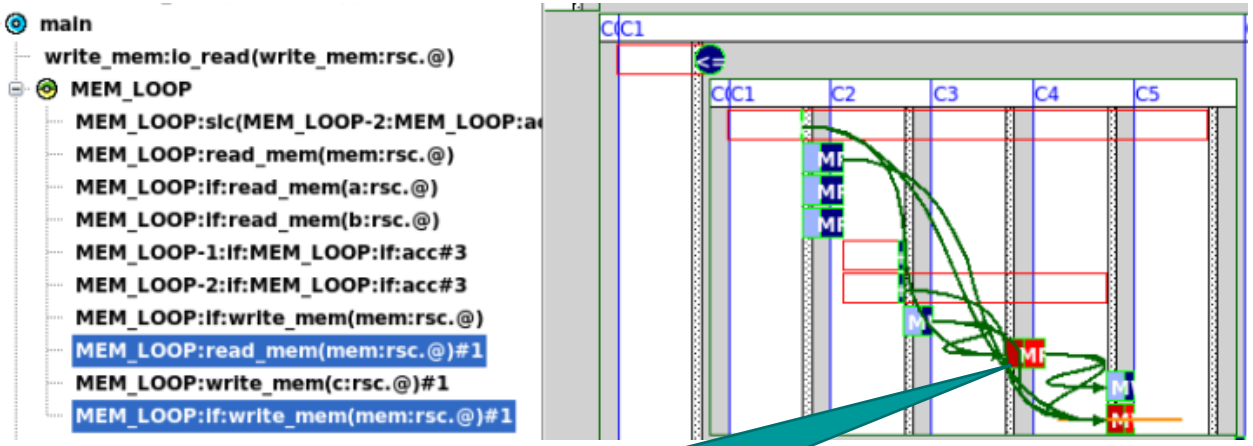
- This usually results in a read modify write, limiting performance
- May need to manually merge memory reads/writes

```
void mem_test(ac_int<7,false> a[64], ac_int<7,false> b[64],
              ac_int<8,false> c[64], bool write_mem){
  static ac_int<8,false> mem[64];

  MEM_LOOP: for(int i=0;i<64;i++){
    c[i] = mem[i];
    if(write_mem){
      mem[i] = a[i] + b[i];
    }
  }
}
```

Write to “mem” in different condition than read



Mem loop unrolled 2x creating 2 memory reads and 2 memory writes



Conditional write to “mem[i]” prevents merging resulting in a read-modify-write of “mem”

Block Size and Interleave

Catapult allows memories to be partitioned into multiple smaller memories using BLOCK_SIZE or INTERLEAVE constraints

```
ac_int<8,false> a[16];

directive set /top_func/sub_func_proc/a_rsc -INTERLEAVE 4

a_rsc_0  a_rsc_1  a_rsc_2  a_rsc_3
a[0]     a[1]     a[2]     a[3]
a[4]     a[5]     a[6]     a[7]
a[8]     a[9]     a[10]    a[11]
a[12]    a[13]    a[14]    a[15]

directive set /top_func/sub_func_proc/a_rsc -BLOCK_SIZE 4

a_rsc_0  a_rsc_1  a_rsc_2  a_rsc_3
a[0]     a[4]     a[8]     a[12]
a[1]     a[5]     a[9]     a[13]
a[2]     a[6]     a[10]    a[14]
a[3]     a[7]     a[11]    a[15]
```

"a" mapped to memory

Split "a" into 4 memories

"a" data is interleaved between the 4 memroies

Split "a" into 4 memories

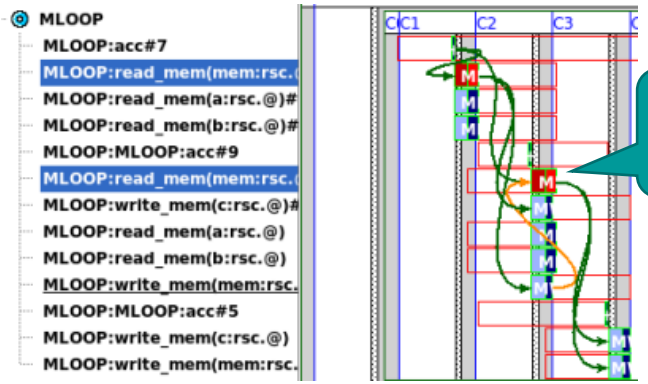
"a" data is divided up by sequential locations between the 4 memories

Block Size Example

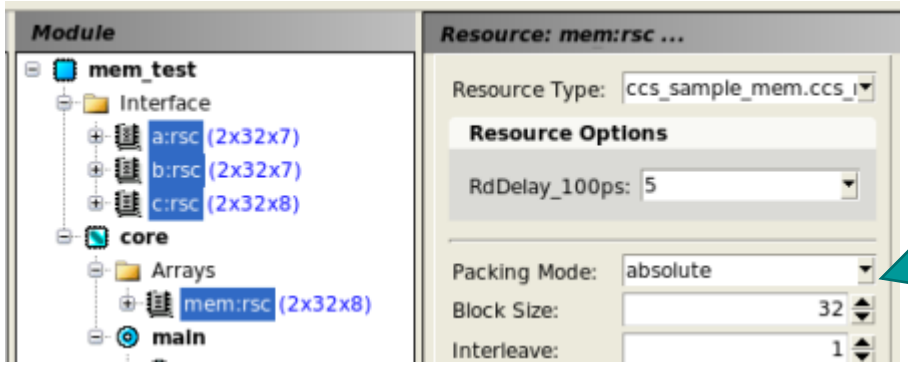
Data is read and written from locations “i” and “i+32”, all memories are singleport RAM

```
void mem_test(ac_int<7,false> a[64], ac_int<7,false> b[64],
              ac_int<8,false> c[64]){
  static ac_int<8,false> mem[64];

  MLOOP: for(int i=0;i<32;i+=
    c[i] = mem[i];
    c[i+32] = mem[i+32];
    mem[i] = a[i] + b[i];
    mem[i+32] = a[i+32] + b[i+32];
  }
}
```

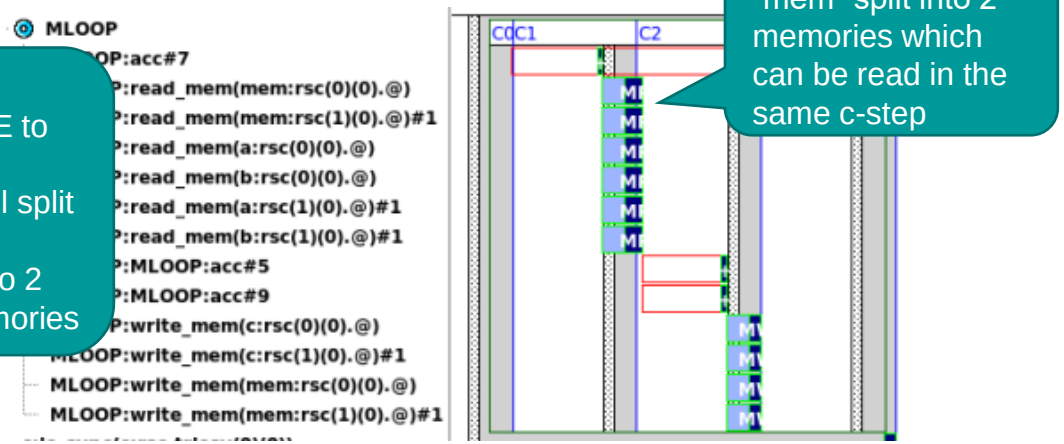
Read from “mem[i]” and “mem[i+32]



Takes 2 c-steps to read twice from “mem”



Setting BLOCK_SIZE to 32 on all memories will split the 64-word memories into 2 32-word memories



“mem” split into 2 memories which can be read in the same c-step

Step-by-Step Lab1 FIR

Memory interface

Multi-Block Design

SIEMENS

The Need for Multiple Concurrent Processes

C++ designs containing “rolled“ sequential loops that are not automatically merged will have lower performance

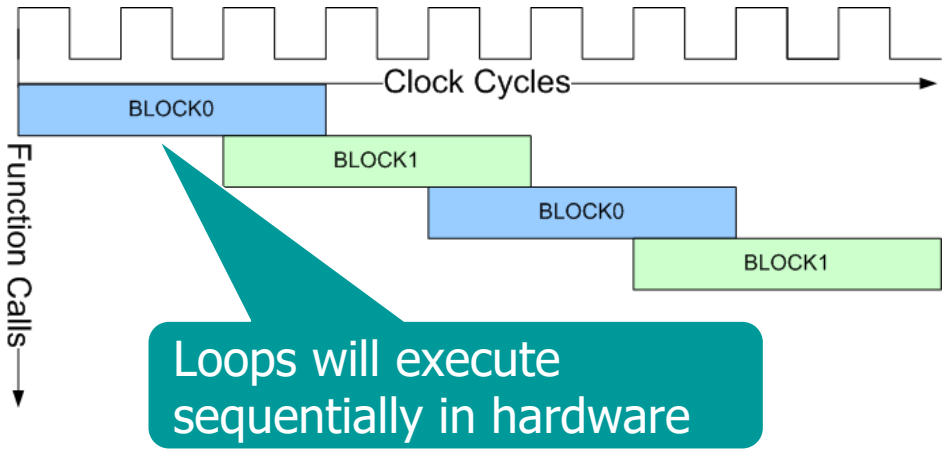
Loop merging will not happen when there is:

- Out of order array accesses between loops
- Complex control
- Non-deterministic data exchange between loops

Example: Out of order array access between sequential loops without hierarchy

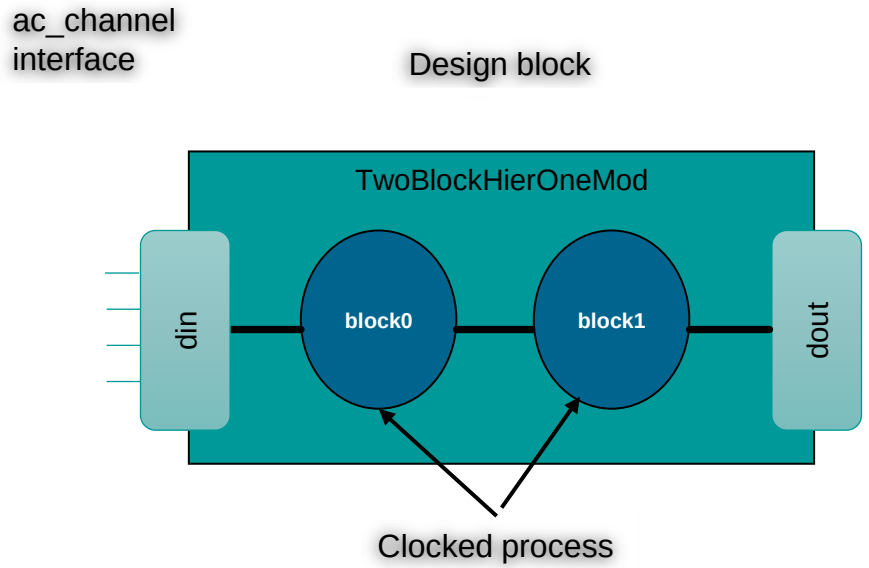
```
void BLOCK0(int din[3], int dout[3]){
 WRITE:for(int i=0;i<3;i++){
    dout[i] = din[i];
  }
}
void BLOCK1(int din[3], int dout[3]){
 READ:for(int i=2;i>=0;i--){
    dout[i] = din[i];
  }
}
void top(int din[3], int dout[3]){
  int tmp[3];
  BLOCK0(din,tmp);
  BLOCK1(tmp,dout);
}
```

Scheduled loops execute sequential in hardware giving poor performance



Modeling Multiple Concurrent Processes

- Catapult allows classes and class member functions to be mapped to separate design blocks that are concurrent processes
  - Allows sequential loops to run in parallel
  - “#pragma hls_design interface” used to make a class a design block
  - “#pragma hls_design” used to make a class member function a concurrent process
- Coding style rules must be followed



Multi-Block Coding Style Rules

ac_channel must be used to interconnect classes or class member functions mapped to design blocks

Arrays mapped to memory must be passed through ac_channel for shared memories between design blocks

- Coding style must be followed
- Arrays mapped to memories on the top-level interface are allowed

Cannot mix design blocks and glue logic

- Interconnect for classes or functions mapped to design blocks cannot be mixed with inlined C++ code
- Will get a compilation error

Multi-block Design Within a Class

Class Member Functions Mapped to Design-Block

- Can only be synthesized top-down

```
class TwoBlockHierOneMod{
  ac_channel<uint10> connect;//Interconnect channel
  uint10 acc0;
  uint20 acc1;
  #pragma hls_design
  void block0(ac_channel<uint4 > &din){
    acc0 += din.read();
    connect.write(acc0);
  }
  #pragma hls_design
  void block1(ac_channel<uint20 > &dout){
    acc1 += connect.read();
    dout.write(acc1);
  }
public:
    TwoBlockHierOneMod(){
      acc0 = 0;//Initialize in constructor
      acc1 = 0;//Initialize in constructor
    }
  #pragma hls_design interface
  void CCS_BLOCK(run)(ac_channel<uint4 > &din, ac_channel<uint20 > &dout){
    block0(din);
    block1(dout);
  }
};
```

Interconnect channels and class data variables

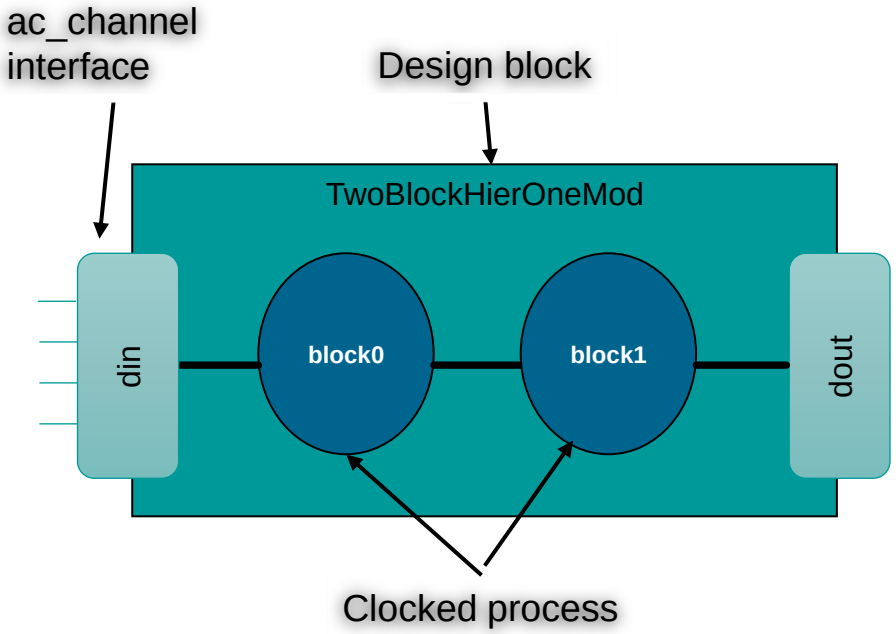
Private member function mapped to design block using #pragma hls_design

Explicit interfaces not required to access class data, including interconnect channels

Interconnect channel used to write/read data between blocks

Public function instantiates design blocks

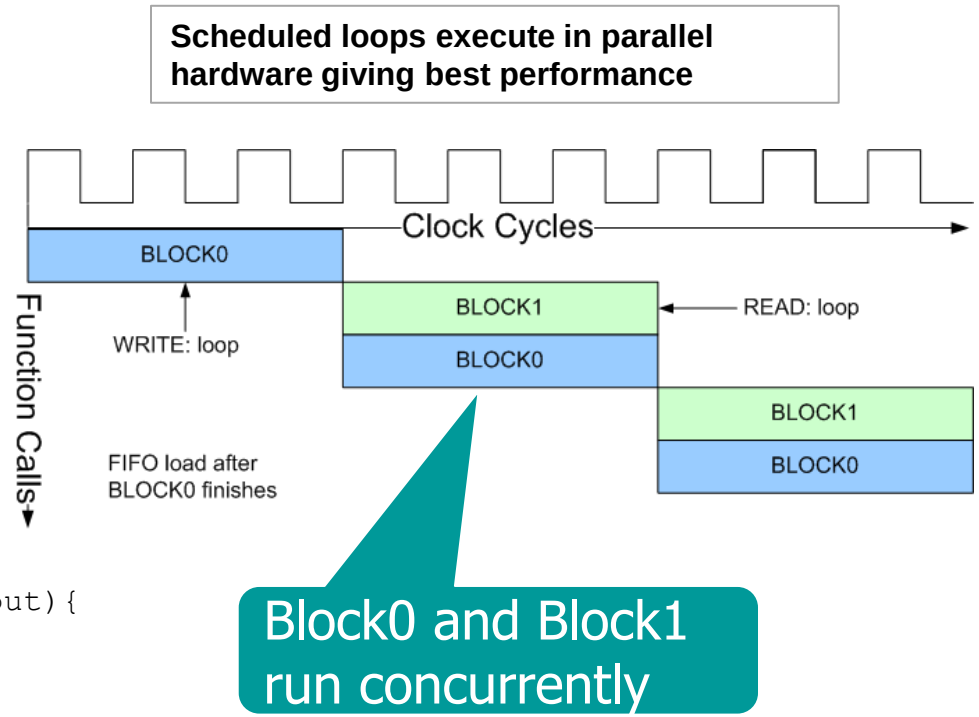
- Class member functions mapped to design blocks can directly access class data
  - Interconnect channels
  - Local variables
  - Explicit interfaces not required on member functions



Modeling Multiple Concurrent Processes

Class-member functions mapped to design blocks can run concurrently

```
class TwoBlockHierOneMod{
  ac_channel<uint10> connect;//Interconnect channel
  uint10 acc0;
  uint20 acc1;
  #pragma hls_design
  void block0(ac_channel<uint4 > &din){
    acc0 += din.read();
    connect.write(acc0);
  }
  #pragma hls_design
  void block1(ac_channel<uint6 > &dout){
    acc1 += connect.read();
    dout.write(acc1);
  }
public:
    TwoBlockHierOneMod(){
      acc0 = 0;//Initialize in constructor
      acc1 = 0;//Initialize in constructor
    }
  #pragma hls_design interface
  void CCS_BLOCK(run)(ac_channel<uint4 > &din, ac_channel<uint20 > &dout){
    block0(din);
    block1(dout);
  }
};
```



Multi-block with Multiple Instantiated Classes

Multiple classes mapped to design blocks

- Can be synthesize bottom-up or top-down

```
#include <mc_scverify.h>

class Block0{
  uint10 acc;
  public:
  Block0(){
      acc = 0;//Intialize in constructor
    }
  #pragma hls_design interface
  void CCS_BLOCK(run)(ac_channel<uint4 > &din, ac_channel<uint10 > &dout){
    acc += din.read();
    dout.write(acc);
  }
};

class Block1{
  uint20 acc;
  public:
  Block1(){
      acc = 0;//Intialize in constructor
    }
  #pragma hls_design interface
  void CCS_BLOCK(run)(ac_channel<uint10 > &din, ac_channel<uint20 > &dout){
    acc += din.read();
    dout.write(acc);
  }
};
```

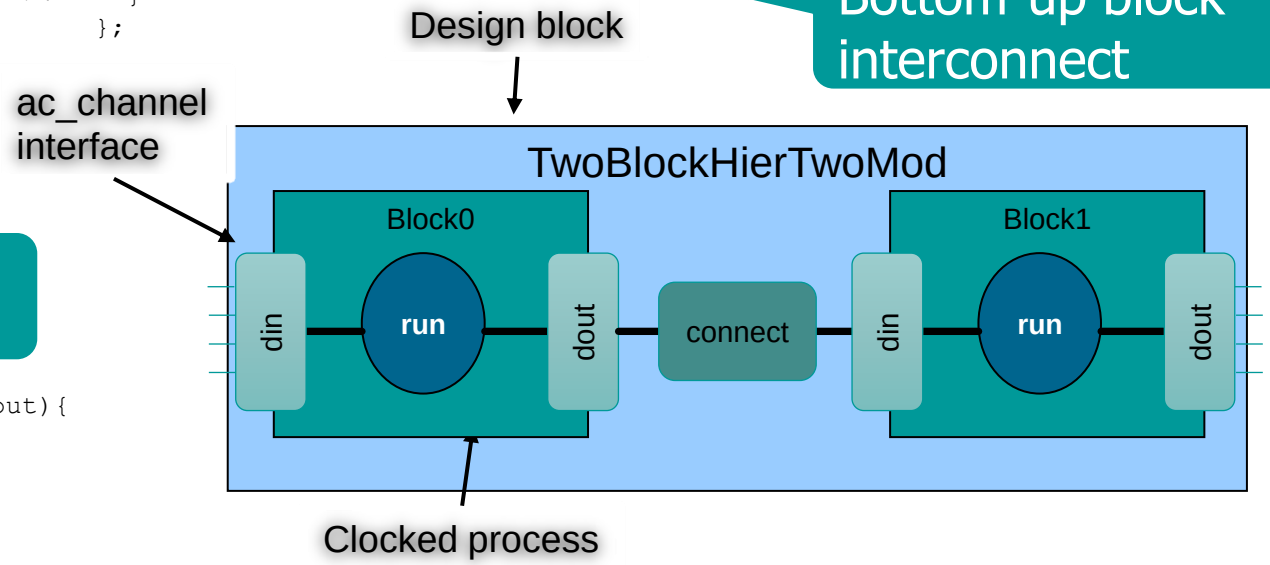
Bottom-up block/class

Bottom-up block/class

```
class TwoBlockHierTwoMod{
  ac_channel<uint6> connect;//Interconnect channel
  Block0 inst0;//Module instances
  Block1 inst1;
public:
    TwoBlockHierTwoMod(){}

  #pragma hls_design interface
  void CCS_BLOCK(run)(ac_channel<uint4 > &din, ac_channel<uint20 > &dout){
    inst0.run(din,connect);
    inst1.run(connect,dout);
  }
};
```

Bottom-up block instantiations

Bottom-up block interconnect



Multiple Instances of a Class Mapped to Design Blocks

Synthesize once, use many times

- Class instances must be identical. E.g. template classes must have same template parameters

Synthesize top-down or bottom-up

```
#include <mc_scverify.h>

template<typename T0, typename T1>
class Block{
  T1 acc;
  public:
  Block(){
      acc = 0;//Intialize in constructor
    }
  #pragma hls_design interface
  void CCS_BLOCK(run)(ac_channel<T0 > &din, ac_channel<T1 > &dout){
    acc += din.read();
    dout.write(acc);
  }
};
#pragma hls_design top
class TwoBlockHierTwoMod{
  ac_channel<uint20> connect;//Interconnect ch
  Block<uint20,uint20> inst0;//
  Block<uint20,uint20> inst1;
public:
    TwoBlockHierTwoMod(){}
  #pragma hls_design interface
  void CCS_BLOCK(run)(ac_channel<uint20 > &din, ac_channel<uint20 > &dout){
    inst0.run(din,connect);
    inst1.run(connect,dout);
  }
};
```

Templatized class for data type

Multiple instances of the same class with identical template paramters

block interconnect

Inter-Block Communication

All inter-block communication is via channels

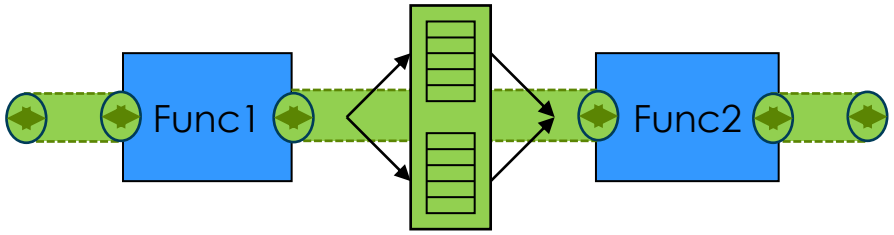
Synchronization is added automatically during synthesis

- Each variable sent between blocks is assigned a channel or pipe
- Setting constraints on channels controls data communication
  - FIFO size
  - Number of memories

Streaming

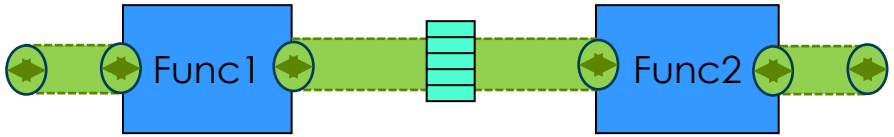
Created when using streaming interfaces

Ping-Pong



Two memories are created to allow pipelined system

Shared

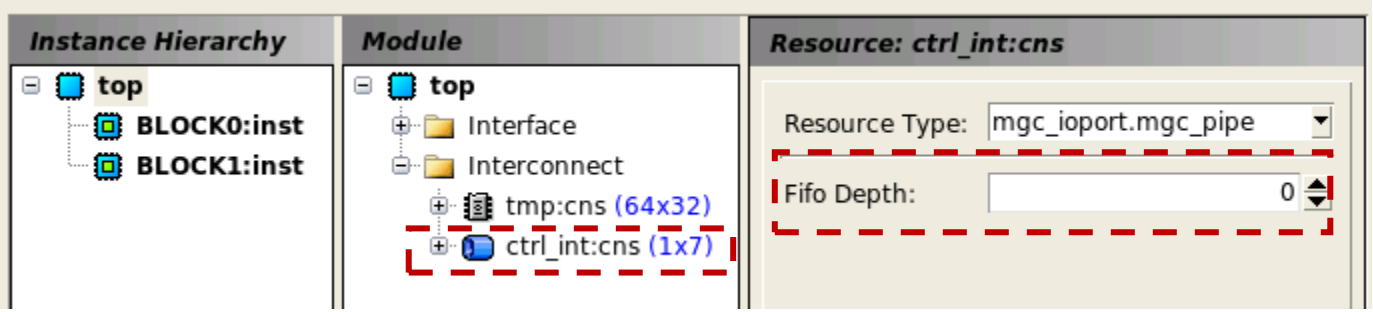


One RAM between blocks, less parallelism

FIFO Between Hierarchical Blocks

ac_channel between hierarchical functions is synthesized to hardware FIFO

- Component called mgc_pipe
- FIFO implemented with registers, memory based FIFO component could be created
- FIFO depth set as synthesis constraint, can be 0
  - Depth = 0 means wire connections
- Synchronization added automatically, user does not need to manage FIFO flags

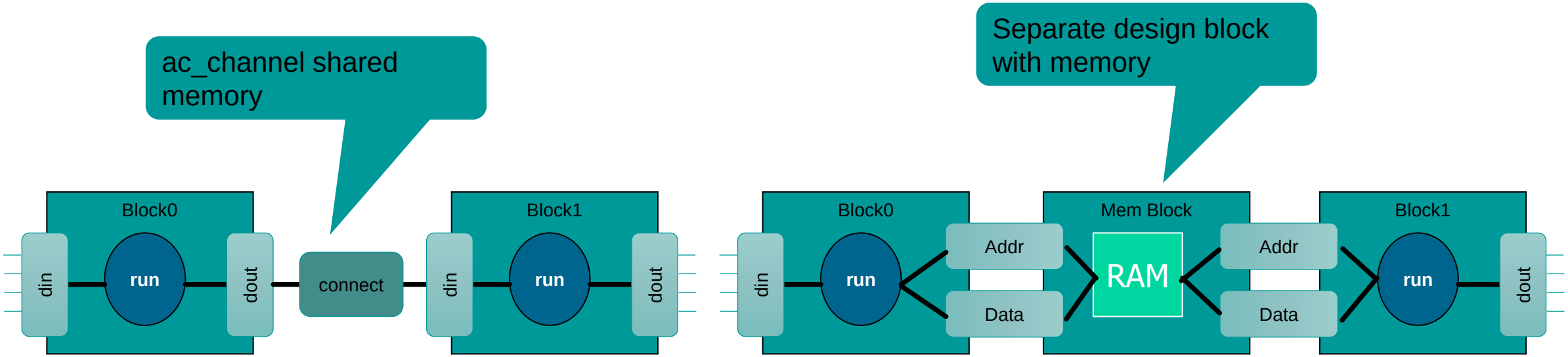


FIFO depth setting

Shared Memories Between Design Blocks

There are two ways to share a memory between classes/member functions mapped to design blocks

- Using an ac_channel and required coding style
- Explicitly coding a separate design block that contains a memory



ac_channel Shared Memory Interfaces Between Blocks

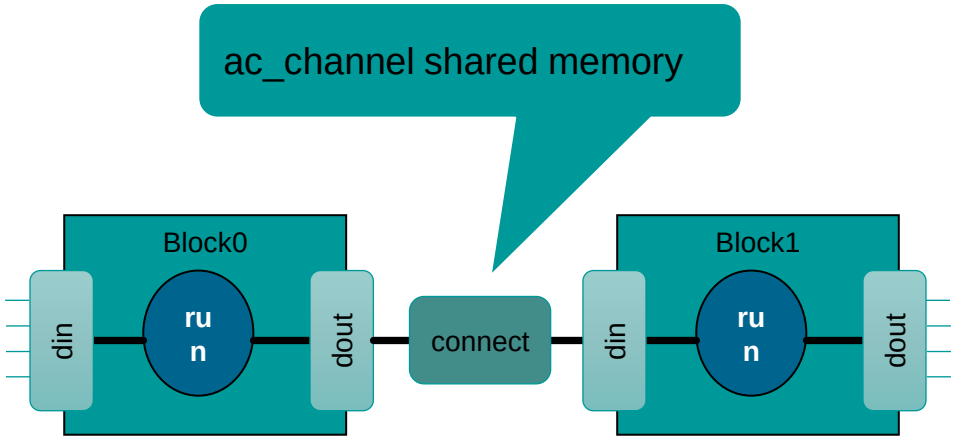
Need a way to create block-level memory interfaces

- Interface Synthesis allows arrays on the function interface to be mapped to a memory interface

Required coding style is to pack arrays inside a struct and read/write through an ac_channel

- Required style for interconnecting design blocks using shared memories
- Array operations still performed locally
- Struct/array mapped to memory using Interface Synthesis

```
struct chanStruct{
        int data[128];
};
void foo(ac_channel<chanStruct> &mem_if, …
```



Memory Write Interfaces

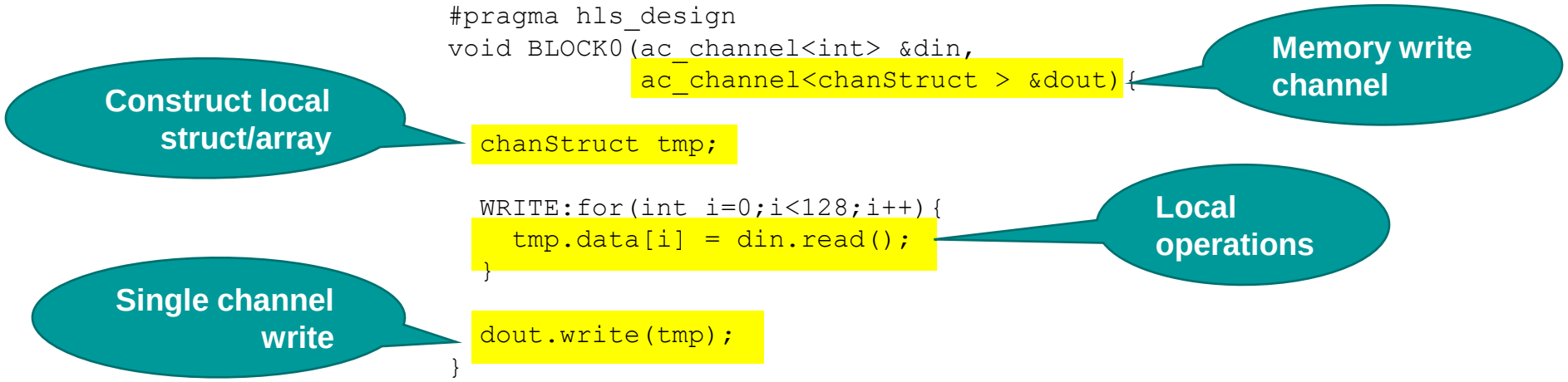
Define a “local” variable using struct type that packs the array

Operate locally on the array inside the struct

Write the struct containing the array into the channel

- Only write the channel once!

Have the HLS tool map the interface channel to a memory interface



Memory Write Interfaces

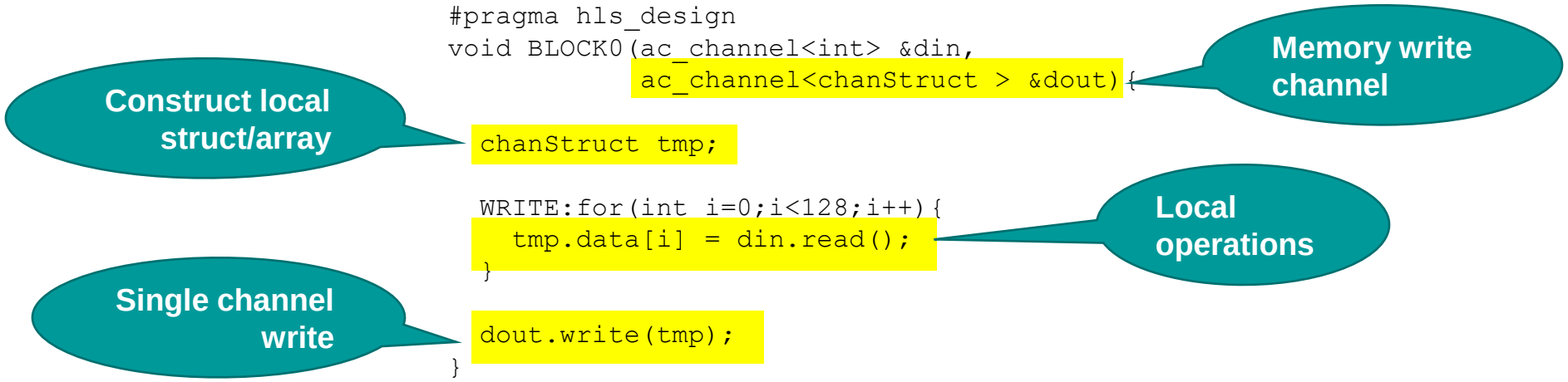
Define a “local” variable using struct type that packs the array

Operate locally on the array inside the struct

Write the struct containing the array into the channel

- Only write the channel once!

Have the HLS tool map the interface channel to a memory interface



Memory Write Interface Pitfalls

Temporary struct with array will be optimized away if coded using recommended style

- Be careful when performing memory writes
- Keep local memory operations between temporary struct declaration and memory channel write

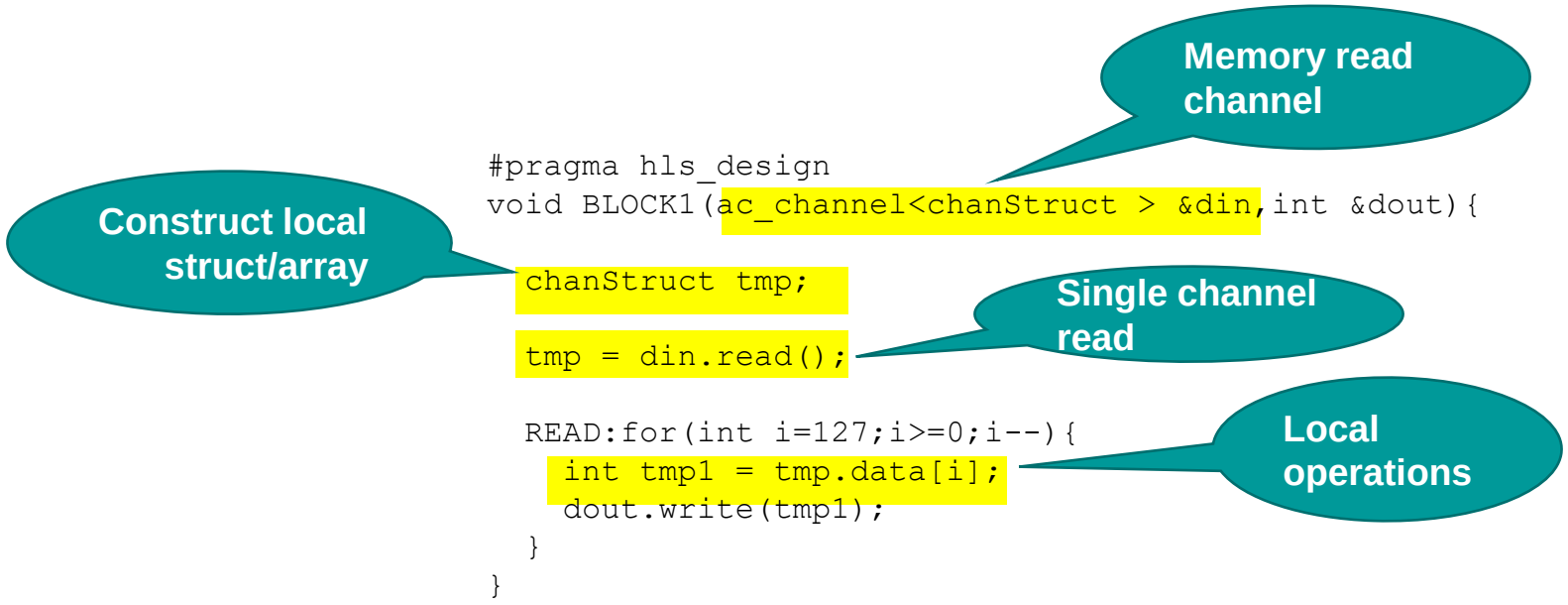
```
#pragma hls_design
void BLOCK0(ac_channel<int> &din,
            ac_channel<chanStruct > &dout){

  chanStruct tmp;

  WRITE:for(int i=0;i<128;i++){
    tmp.data[i] = din.read();
  }

  dout.write(tmp);

  …..

}
```

Memory Read Interfaces

Define a “local” variable using struct type that packs the array

Restrict operations on the local struct to be between the local struct declaration and the memory channel read

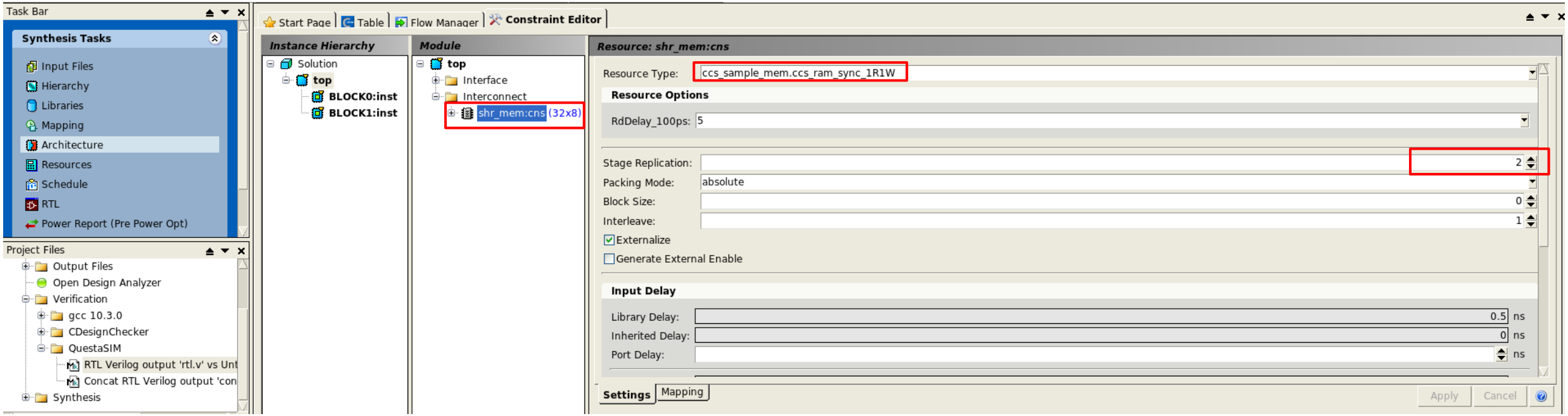
- Doing anything else will cause a local memory to be generated



Define Shared Memory Architecture

Stage Replication

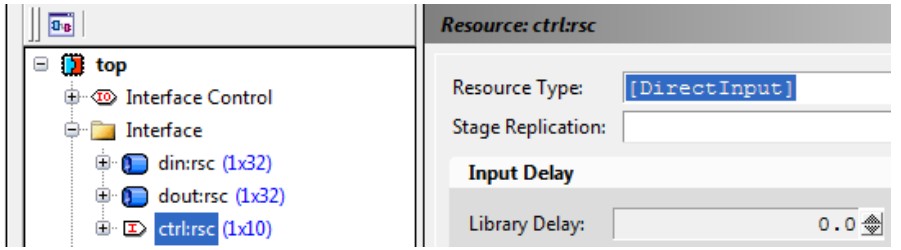
- 1: a single shared memory architecture
- 2: a ping-pong architecture



Common Input to Multiple Blocks

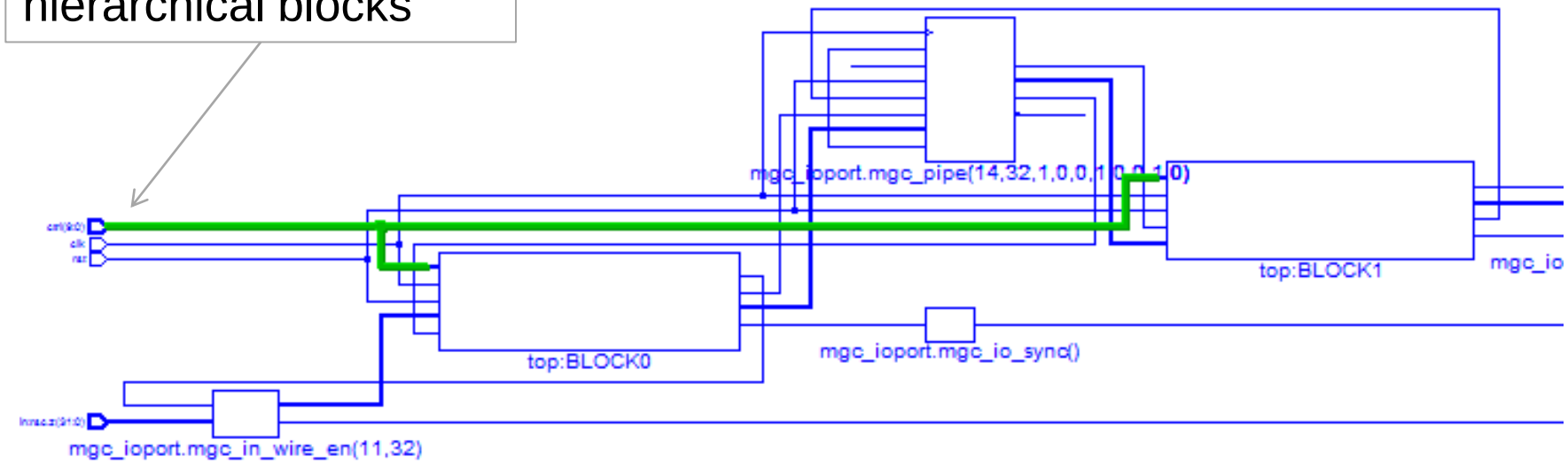
Only Direct Input interface can be used as input to multiple hierarchical blocks

```
#include "direct_input.h"
void BLOCK0(ac_channel<int> &din, ac_channel<int> &dout,
            ac_int<WIDTH,false> &ctrl){
                for(int i=0;i<1024;i++){
                    if(i==ctrl)
                        break;
                    dout.write(din.read()*13);
                }
}
void BLOCK1(ac_channel<int> &din, ac_channel<int> &dout,
            ac_int<WIDTH,false> &ctrl){
                for(int i=0;i<1024;i++){
                    if(i==ctrl)
                        break;
                    dout.write(din.read());
                }
}
void top(ac_channel<int> &din,ac_channel<int> &dout,
         ac_int<WIDTH,false> &ctrl){
             static ac_channel<int> data_int;
             BLOCK0(din,data_int,ctrl);
             BLOCK1(data_int,dout,ctrl);
}
```



Pseudo-static Inputs: DIRECT_INPUT

Unsynchronized I/O directly drives multiple hierarchical blocks



Step-by-Step Lab1 FIR

Multi-block Design

More Catapult HLS Resources

HLS resource library

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

HLS on-demand training

https://training.plm.automation.siemens.com/mytraining/viewlibrary.cfm?memTypeID=273992&memID=273992

HLS bluebook

https://resources.sw.siemens.com/en-US/e-book-high-level-synthesis-hls-blue-book

HLS C++/SystemC open-source library

https://hlslibs.org/