Improving Area and Resources

Objectives

- After completing this module, you will be able to:
  - Describe different methods for improving resource utilization
  - Control the structure of the design by using directives to improve the area
  - Explain how arbitrary precision types can result in optimal resource usage

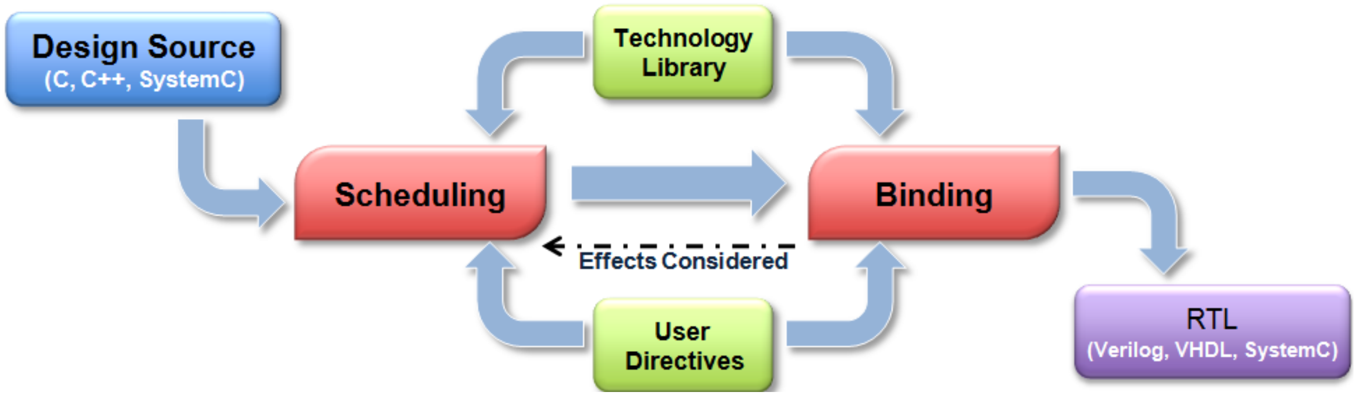
AMD XILINX

Vitis HLS Design Methodology

Baseline Performance	Simulating the Design	Validate the C function
	Synthesizing the Design	Convert the C/C++ design to RTL implementation
C/C++ Coding Style	C/C++ Coding Style	Adjust for supported and unsupported constructs
Interfaces	Defining the Interfaces	Synthesize arguments of the top-level function in HLS design into interfaces and ports that group multiple signals
Optimization Techniques	Optimizing for Throughput	Improve throughput or reduce the initiation interval by pipelining loops and functions, partitioning arrays, and managing the dependencies
	Optimizing for Latency	Use latency constraints Merge loops and flatten loops
	Optimizing for Area	Manage data types Inline functions and reshape arrays
	Optimizing for Logic	Optimizing the logic by inferring shift registers and controlling operator pipelining
	Optimizing for Burst Transfers	Control burst transfer behavior

Review: Control Scheduling & Binding

- Scheduling & Binding
  - Scheduling and Binding are the processes at the heart of HLS



- BIND_OP/ BIND_STORAGE
  - Can be used to specify a device resource for implementation
  - Can be used to assign a specific memory type in the RTL
- The allocation directive
  - Can be used to limit the number of operation in scheduling & binding stages

Directives to improve Area

Directives and Configurations	Description
ALLOCATION	Specify a limit for the number if operations, cores, or functions used. This can force the sharing of hardware resources and might increase latency
ARRAY_RESHAPE	Reshapes an array from one with many elements to one with greater word-width. Useful for improving block RAM accesses without using more block RAM
LOOP_MERGE	Merges consecutive loops to reduce overall latency, increase sharing, and improve logic optimization
OCCURRENCE	Used when pipelining functions or loops to specify that the code in a location is executed at a lesser rate than the code in the enclosing function or loop
STRAM	Specifies that a specific memory channel is to be implemented as a FIFO or RAM during dataflow optimization
Config Dataflow	This configuration specifies the default memory channel and FIFP depth in a dataflow region
BIND_STORAGE	Define a specific implementation for a storage element, or memory, in the RTL
BIND_OP	Define a specific implementation for an operation in the RTL

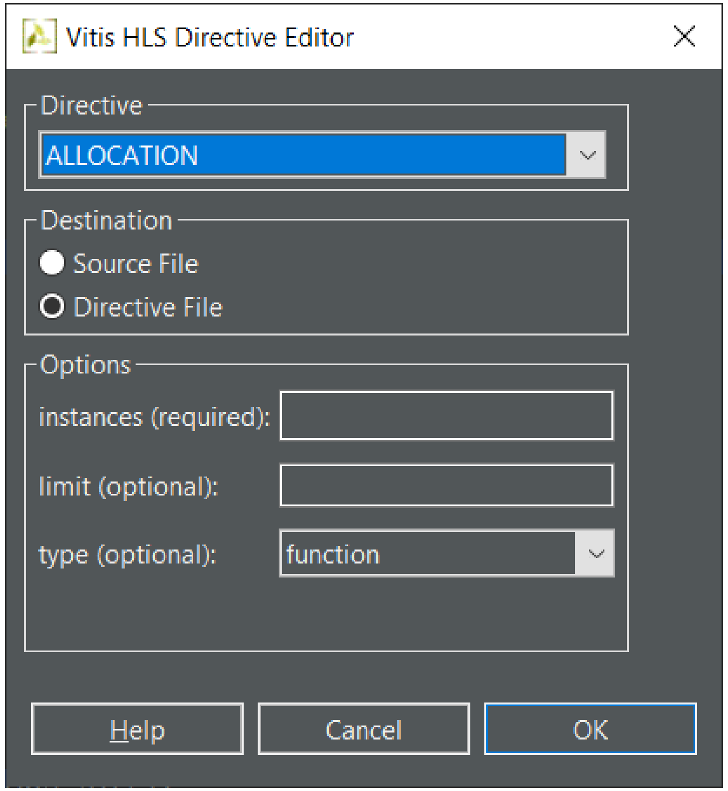
Reducing Area Usage

Improving Area/Resource Utilization

- Control the number of elements
  - Directives can be used to control scheduling and binding phases, which in turn control the number of elements used
- Control the design hierarchy
  - Like RTL synthesis, removing the hierarchy can help optimize across function and loop boundaries
    - Functions can be inlined
    - Loops can be unrolled
- Array implementation
  - VITIS HLS provides directives for combining memories
    - Allowing a single large memory to be used instead of multiple smaller memories
- Bit-width optimization
  - Bit width impact the size of the storage elements and operators
  - Arbitrary precision types ensure correct operator sizing

Allocation: Limit the Numbers

- Allocation directive limits different types
  - Type: Operation
    - The instances are the operators
    - Add, mul, urem, etc.
  - Type: Core
    - The instances are the cores
    - Adder, Addsub, PipeMult2s, etc
  - Type: Functions
    - The functions in the code
    - Discussed in more detail later

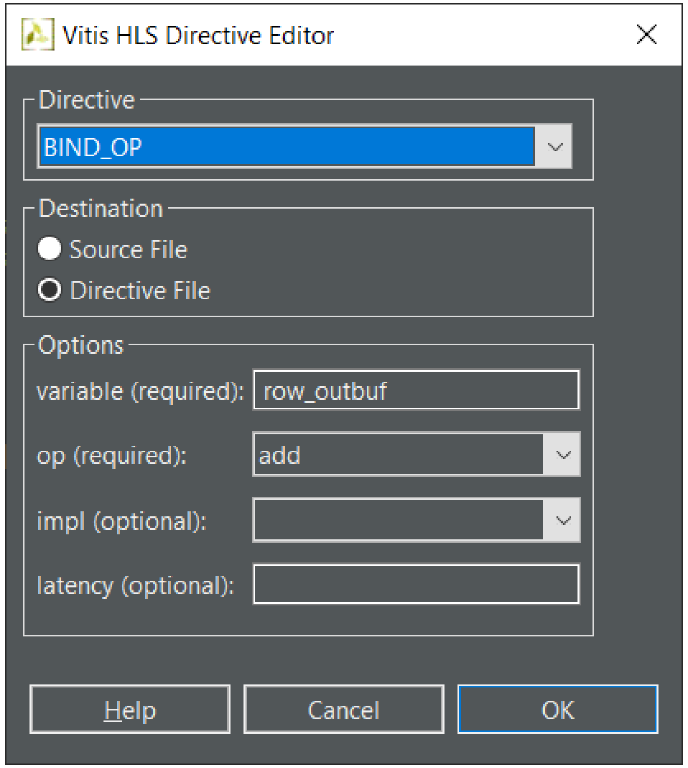


Operators and Cores are listed in the VITIS HLS Library Guide

- Allocations are defined for a scope
  - Like all directives, allocations are set for the scope they are applied in
  - If the directive is applied to a function, loop or region, it does not include objects outside that scope

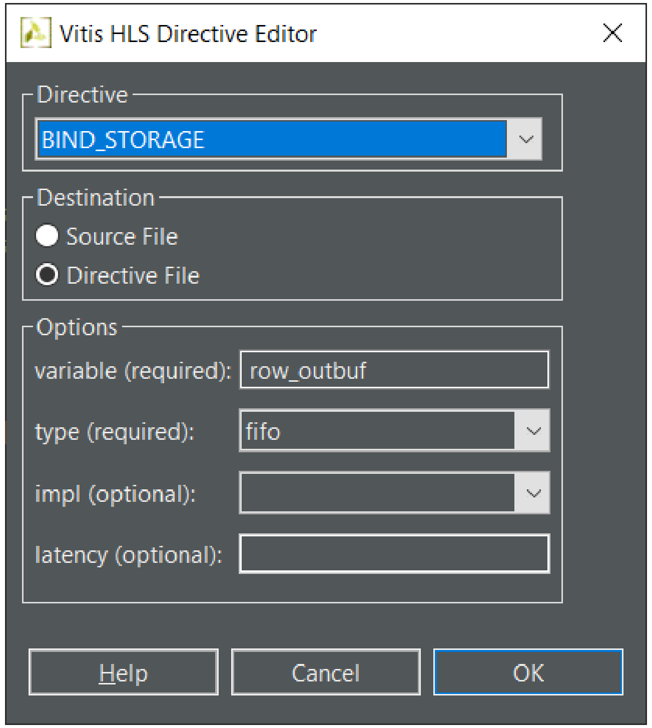
BIND_OP: Specific a device resource

- Specifies that for a given variable, an operation(mul, add,sub) should be mapped to a device resource for implementation
  - If not, Vitis HLS tool automatically determines the resource to use
- You can also specify:
  - -op: Operation to bind to a specific implementation resource
  - -impl: Implementation to use for the specified operation
  - -latency: Latency for the operation



BIND_STORAGE: Assign a specific memory

- Assigns a variable(array or function argument) to a specific memory type in the RTL
  - If not, Vitis HLS tool determines the memory type to assign
  - HLS implements the memory using specified implementations(impl) in the hardware
  - Controls the array implementation as a single/dual-port RAM
- You can also specify:
  - -type: Type of memory to bind to the specified variable
    - Support types: fifo, ram_1p, ram_1wnr, ram_2p, ram_s2p, ram_t2p, rom_1p, rom_2p, and rom_np
  - -impl: Implementation for the specified memory type
    - Support impementations: bram, bram_ecc, lutram, uram, uram_ecc and srl
  - -latency: Default latency for the binding of the storage type to the implementation



Review: Functions & RTL Hierarchy

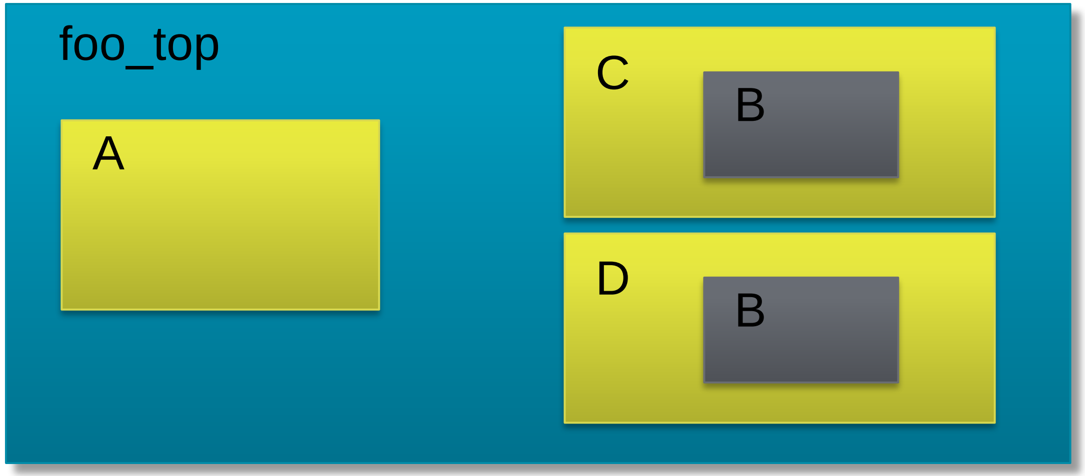
- Each function in the design gets synthesized into an RTL block, maintaining the design hierarchy

Source Code

```
void A() { ..body A..}
void B() { ..body B..}
void C() {
        B();
}
void D() {
        B();
}


void foo_top() {
        A(…);
        C(…);
        D(…)
}
```

my_code.c

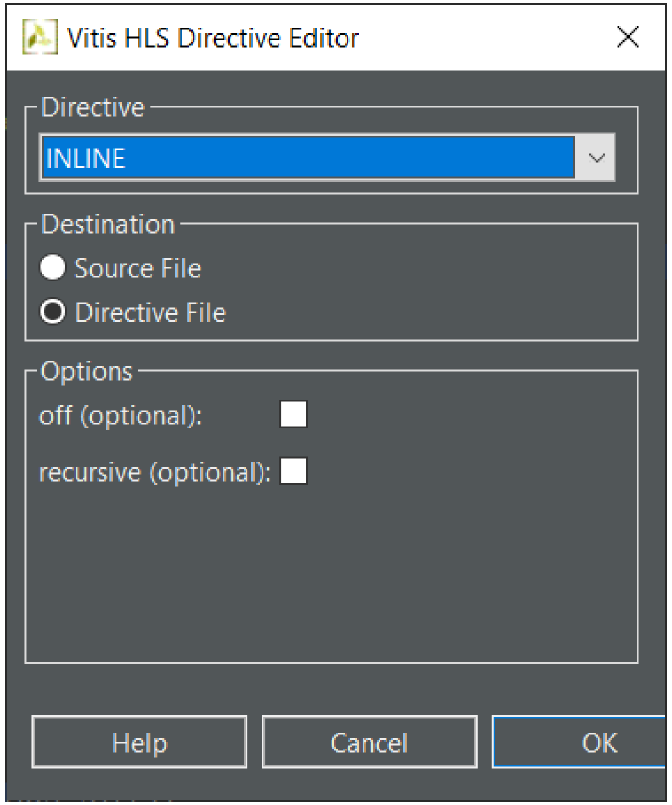
RTL hierarchy



Functions can be inlined – the hierarchy removed & the function dissolved into the surrounding function

Function Inlining

- Inlining a function may improve area by allowing the components within the function to be better shared or optimized with the logic in the calling function.
- Vitis HLS performs some inlining automatically
  - This is performed on small logic functions if Vitis HLS determines area or performance will benefit
- User Control
  - Optionally recursively down the hierarchy
  - Optionally inlining can be explicitly prevented
    - Turn inlining off
- Inlining functions allows for greater optimization
  - Like ungrouping RTL hierarchies: optimization across boundaries
  - Like ungrouping RTL hierarchies it can result in lots of operations & impact run time



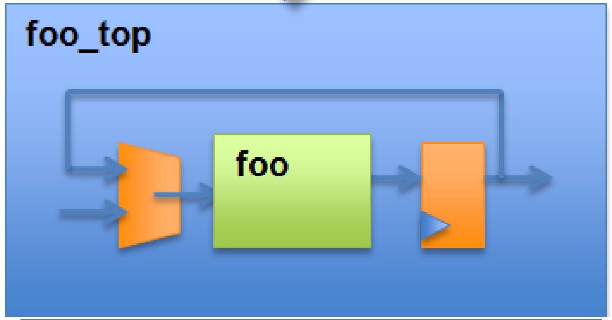
AMD XILINX

Inline and Allocation: Shape the Hierarchy

Easy to Share

```
void foo() {

}
void foo_top() {
        foo(...);
        foo(...);
}
```

```
set_directive_allocation -limit 1
    -type function foo_top foo
```



One RTL block is reused for both instances of function *foo*

Cannot be shared

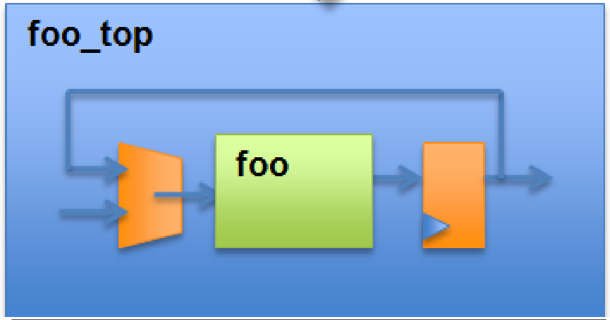
```
void dummy1() {
        foo();
}
void dummy2() {
        foo();
}
void foo_top() {
        dummy1(...);
        dummy2(...);
}
```

```
set_directive_allocation -limit 1
    -type function foo_top foo
```

Function *foo* is not within the immediate scope of *foo_top*

Controlling Sharing

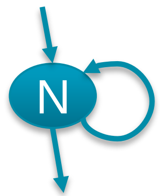
```
set_directive_allocation -limit 1
    -type function foo_top foo

set_directive_inline dummy1
set_directive_inline dummy2
```



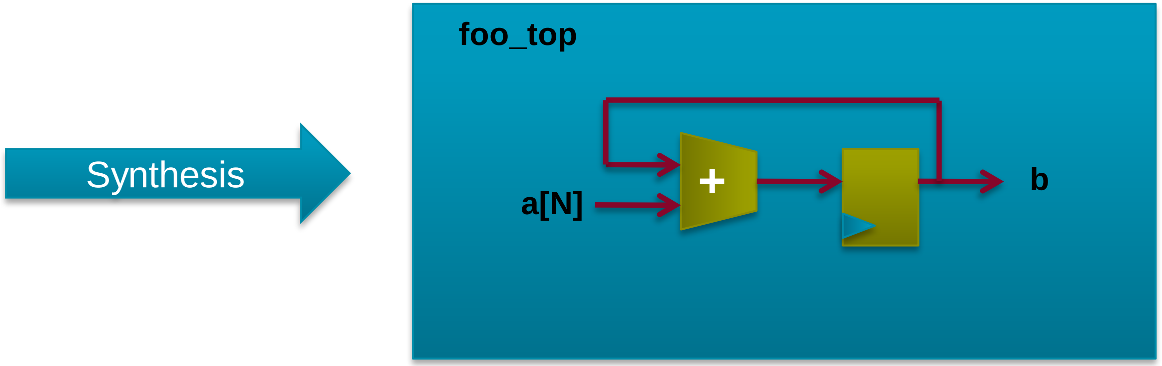
Inlining brings *foo* into function foo_top where it can be shared

Loops

- By default, loops are rolled and pipelined to improve performance
  - Each C loop iteration ➔ Implemented in the same state with same resources




```
void foo_top (…) {
  ...
  Add: for (i=3;i>=0;i--) {
           b = a[i] + b;
  ...
  }
```



- For Area optimization

Keeping loops rolled maximizes sharing across loop iterations: each iteration of the loop uses the same hardware resources

AMD XILINX

Loop Merging

- Loop merging can remove the redundant computation among multiple (related) loops
  - Improving area and performance

```
My_Region: {
#pragma HLS merge loop
for (i = 0; i < N; ++i)
      A[i] = B[i] + 1;
for (i = 0; i < N; ++i)
      C[i] = A[i] / 2;
}
```

Merge

```
for (i = 0; i < N; ++i) {
      A[i] = B[i] + 1;
      C[i] = A[i] / 2;
}
```

Effective code after compiler transformation

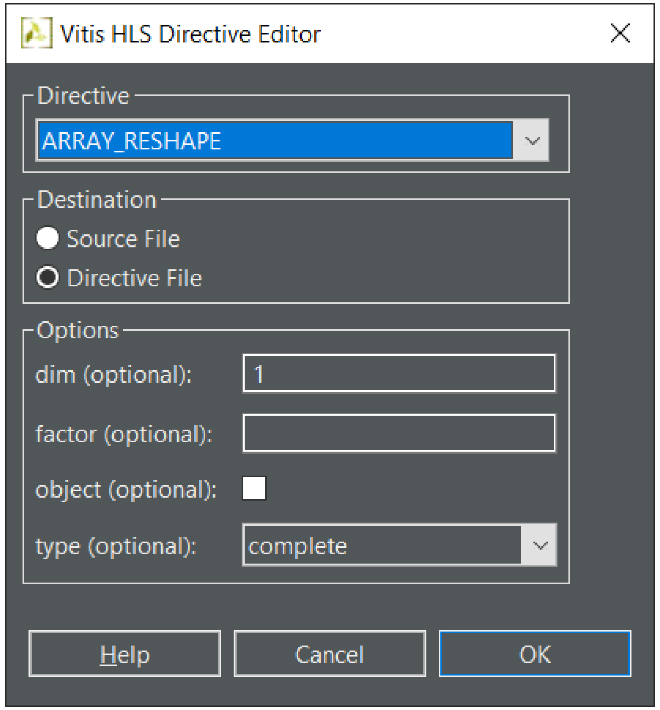
- Allows the logic within the loops to be optimized together
- Optimization cannot occur across loop boundaries
- Allows for more efficient architecture explorations

```
for (i = 0; i < N; ++i)
      C[i] = (B[i] + 1) / 2;
```

Removes A[i], any address logic and any potential memory accesses

Mapping Arrays

- The arrays in the C model may not be ideal for the available RAMs
  - The code may have many small arrays, each array is mapped into a block RAM or UltraRAM
  - Basic block RAM unit is 18K, the array may not utilize the RAMs very well
- Array Mapping
  - Mapping combines smaller arrays into larger arrays
    - Reduce the number or block RAMs required
  - Specify the array variable to be mapped
  - Give all arrays to be combined the same instance name
  - If an array is larger than 18K, they are automatically mapped into multiple 18K units.
- ARRAY_RESHAPE directive supports mapping small arrays into a larger one



Arbitrary Precision Integers

- C and C++ have standard types created on the 8-bit boundary
  - char (8-bit), short (16-bit), int (32-bit), long long (64-bit)
    - Also provides stdint.h (for C), and stdint.h and cstdint (for C++)
    - Types: int8_t, uint16_t, uint32_t, int_64_t etc.
  - Implemented hardware sometimes need different sizes of data types
  - They result in hardware which is not bit-accurate and can give sub-standard QoR
  - Sigals with arbitrary widths are also converted into fixed sized widths when native C data types are used
- Vitis HLS provides both integer and fixed-point arbitrary precision data types for C++
  - Allow any arbitrary bit-width to be specified
  - Will simulate with bit-accuracy

my_code.c

```
#include ap_cint.h

void foo_top (…) {

   int1        var1;       // 1-bit
   uint1       var1u;      // 1-bit unsigned
   int2        var2;       // 2-bit
   …           …
   int1024     var1024;    // 1024-bit
   uint1024    var1024;    // 1024-bit unsigned

   …
```

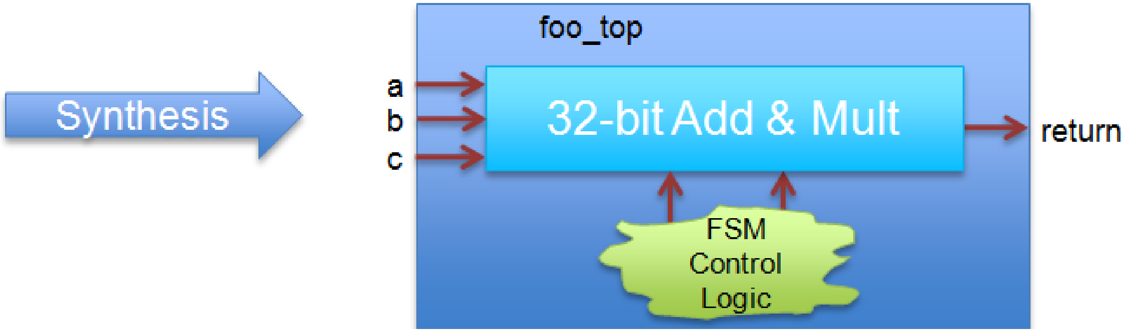
my_code.cpp

```
#include ap_int.h

void foo_top (…) {

   ap_int<1>       var1;       // 1-bit
   ap_uint<1>      var1u;      // 1-bit unsigned
   ap_int<2>       var2;       // 2-bit
   …               …
   ap_int<1024>    var1024;    // 1024-bit
   ap_int<1024>    var1024u;   // 1024-bit unsigned

   …
```

Why are Arbitrary Precision types Needed?

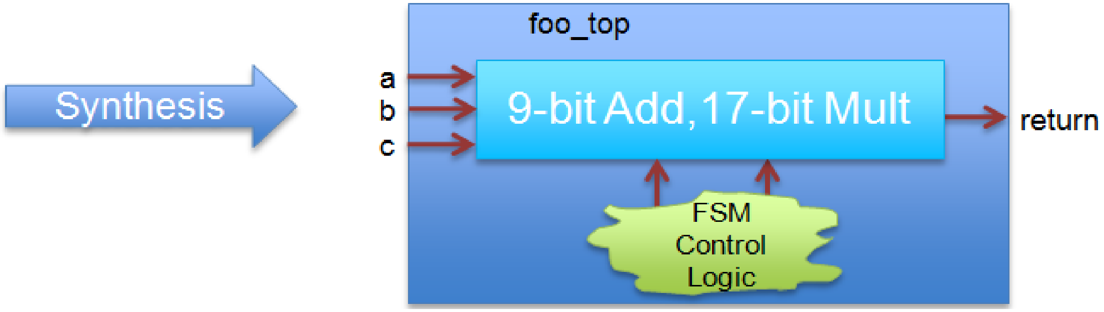
- Code using native C int type

```
int foo_top(int a, int b, int c)
{
    int sum, mult;
    sum=a+b;
    mult=sum*c;
    return mult;
}
```



- However, if the inputs will only have a max range of 8-bit
  - Arbitrary precision data-types should be used

```
int17 foo_top(int8 a, int8 b, int8 c)
{
    int9 sum;
    int17 mult
    sum=a+b;
    mult=sum*c;
    return mult;
}
```



  - It will result in smaller & faster hardware with full precision, full precision can be simulated/validated with C simulation and hardware will behave the same

AMD XILINX

DEPEDENCE Directive

Loop-Independent Dependence

Same element is accessed in a single loop iteration

```
for (i=0; i<N; i++) {
    A[i]=x;
    y=A[i];
}
```

Loop-Carried Dependence

Same element is accessed from a different loop iteration

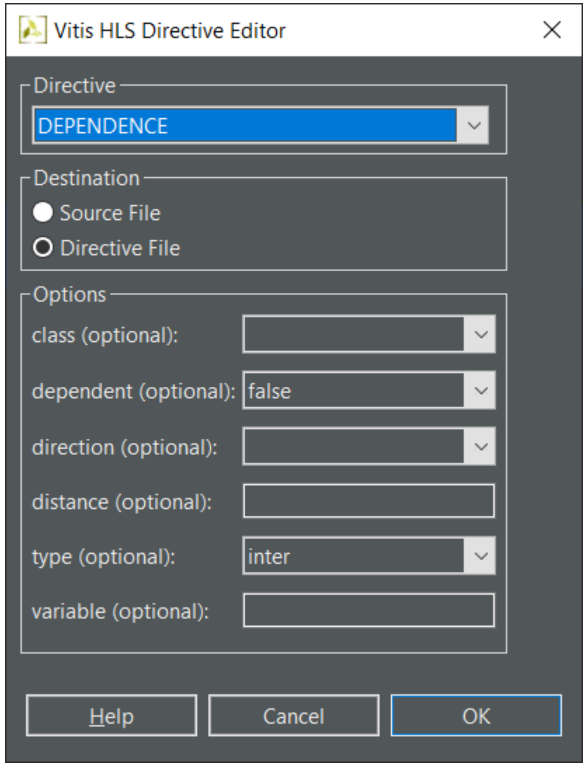
```
for (i=1; i<N; i++) {
    A[0]=0;
    A[i]=A[i-1]*2;
}
```

DEPENDENCE directive allows you to explicitly define the dependencies and eliminate a false dependence

DEPEDENCE Directive

- DEPENDENCE directive allows to explicitly define the dependencies and eliminate a false dependence
  - Provides the tool with additional information about the dependencies
  - Allows designers to inform tool of "external” conditions
  - Makes sure there are no dependencies between loop iterations

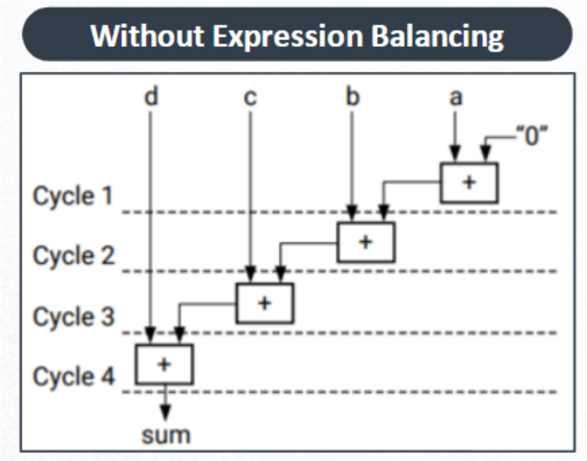
```
for (row = 0; row < rows + 1; row++) {
  L2: for (col = 0; col < cols + 1; col++) {
    #pragma HLS PIPELINE II=1
    #pragma HLS dependence variable=buff_A type=inter dependent=false
    #pragma HLS dependence variable=buff_B type=inter dependent=false
          if (col < cols) {
            buff_A[2][col] = buff_A[1][col]; // read from buff_A[1][col]
            buff_A[1][col] = buff_A[0][col]; // write to buff_A[1][col]
            buff_B[1][col] = buff_B[0][col];
            temp = buff_A[0][col];} } }
```

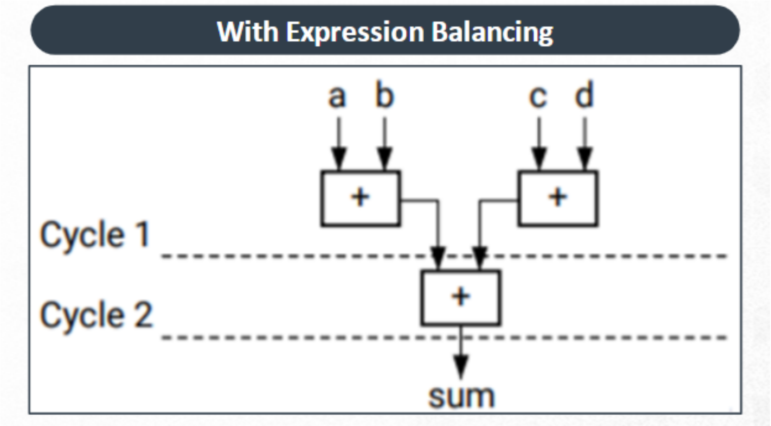


Optimizing Logic Expressions

- Expression balancing rearranges operators to construct a balanced tree and reduce latency
  - For integer operations expression balancing is on by default but may be disabled using the EXPRESSION_BALANCE pragma or directive.
  - For floating-point operations, expression balancing is off by default but may be enabled using using the config_compile -unsafe_math_optimizations command.

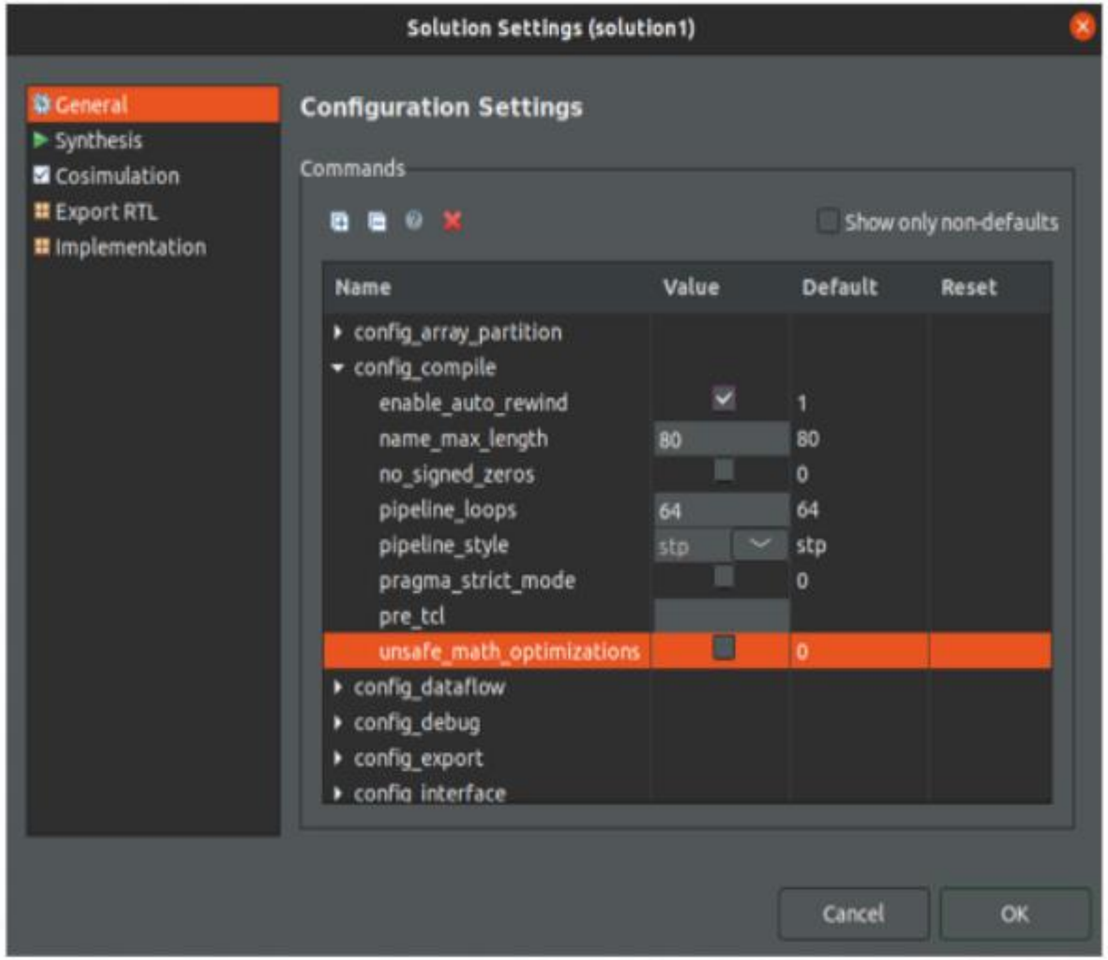
```
data_t foo_top (data_t a, data_t b, data_t c, data_t d) {
  data_t sum;
  sum = 0;
  sum += a;
  sum += b;
  sum += c;
  sum += d;
  return sum;
}
```





Optimizing Logic Expressions

- To configure the tool to enable expression balancing with float and double types
- Place the pragma in the C source within the boundaries of the required location.
  - #pragma HLS expression_balance off
  - Turns off expression balancing at this location.
  - Specifying #pragma HLS expression_balance enables expression balancing in the specified scope. Adding off disables it.
- Solution settings > General > Config_compile>unsafe_math_optimizations



[Public]

Summary

Summary

- Resource utilization can be reduced using allocation and binding controls
  - Allocation can define how many resource are used
- The design structure can be controlled by
  - Inlining functions: direct impact on RTL hierarchy & optimization possibilities
  - Loops: direct impact on reuse of resources
  - Arrays: direct impact on the RAM
- Major area optimization techniques
  - Minimize bit widths
  - Map smaller arrays into larger arrays
    - Make better use of existing RAMs
  - Control loop hierarchy
  - Control function call hierarchy
  - Control the number of operators and cores
- Arbitrary precision data types help controlling both the area and resource utilization

[Public]

Thank You

Disclaimer and Attribution

AMD
XILINX