Improving Area and Resources

XILINX

Objectives

> **After completing this module, you will be able to:**
>
> - Describe how arbitrary precision data types can reduce resource utilization
> - List various area optimization techniques
> - List means by which resource utilization can be reduced

Outline

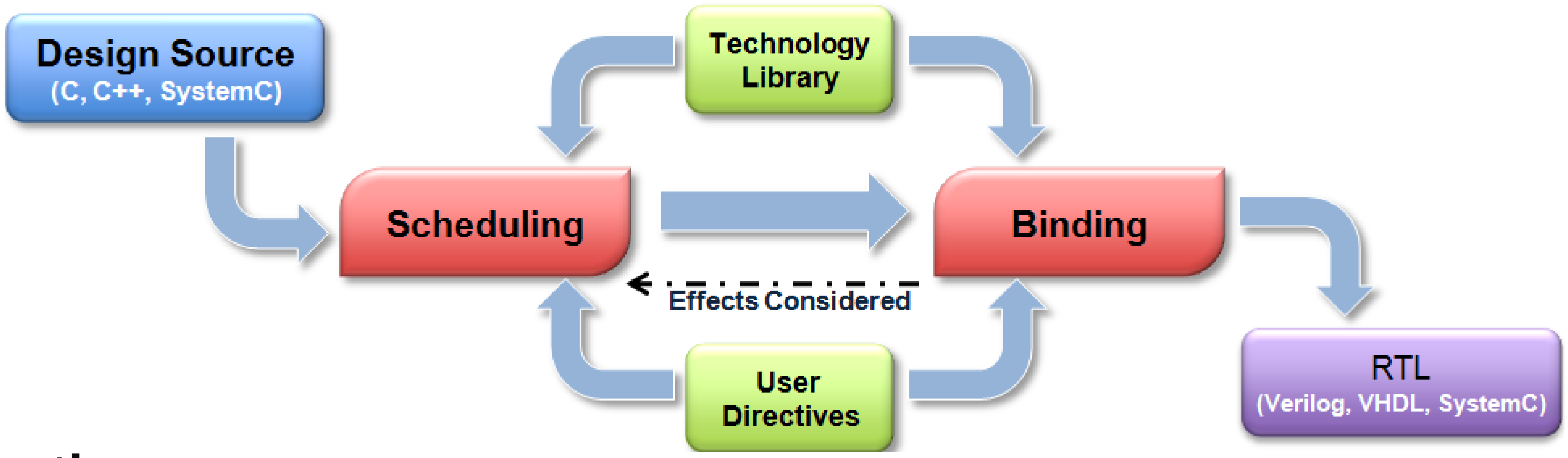
> *Optimizing Resource Utilization*

> Reducing Area Usage

> Summary

Review: Control Scheduling & Binding

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

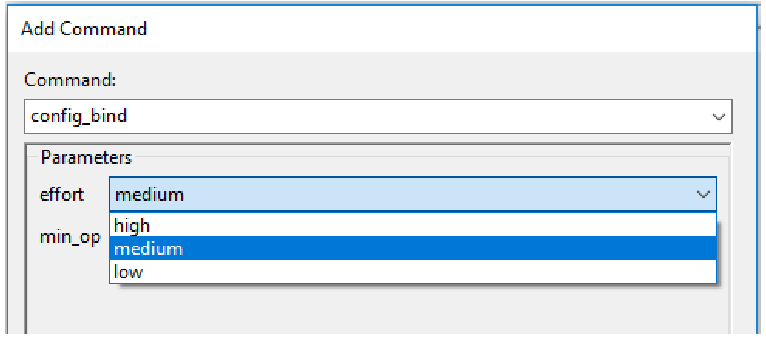


- **Binding configuration**
  - Can be used to minimize the number of operations
- **The allocation directive**
  - Can be used to limit the number of operation in scheduling & binding stages
- **The resource directive**
  - Can be used to specify which cores are to be used during binding

XILINX

Configuring Binding

- > **Binding is controlled via a configuration command**
  - >> The effort levels determine how much time is spent trying to map many operators onto fewer cores
  - >> As with all effort levels, they are worth using if you can see the design close to what is required
    - – Else the tool will spend time exploring for possibilities
    - – And simply increase run time
    - – Use efforts judiciously
- > **Binding can be configured to minimize specific operators**
  - >> Can be used to direct Vivado HLS to synthesize with the minimum number of operations
  - >> The configuration command overrides muxing costs and can be used to force sharing
    - – Works on all scopes in a design



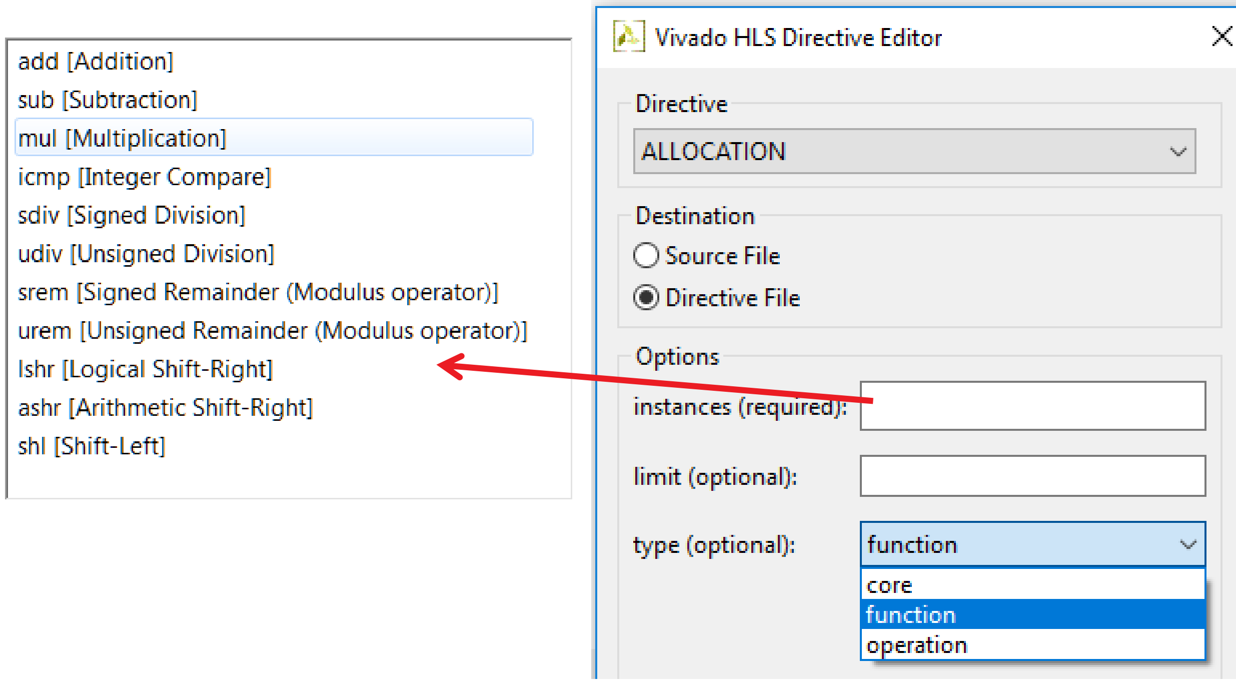
Operators which can be minimized and passed as arguments by listing in the min_op field

- add - Addition
- sub - Subtraction
- mul - Multiplication
- icmp - Integer Compare
- sdiv - Signed Division
- udiv - Unsigned Division
- srem - Signed Remainder
- urem - Unsigned Remainder
- lshr - Logical Shift-Right
- ashr - Arithmetic Shift-Right
- shl - Shift-Left

XILINX.

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



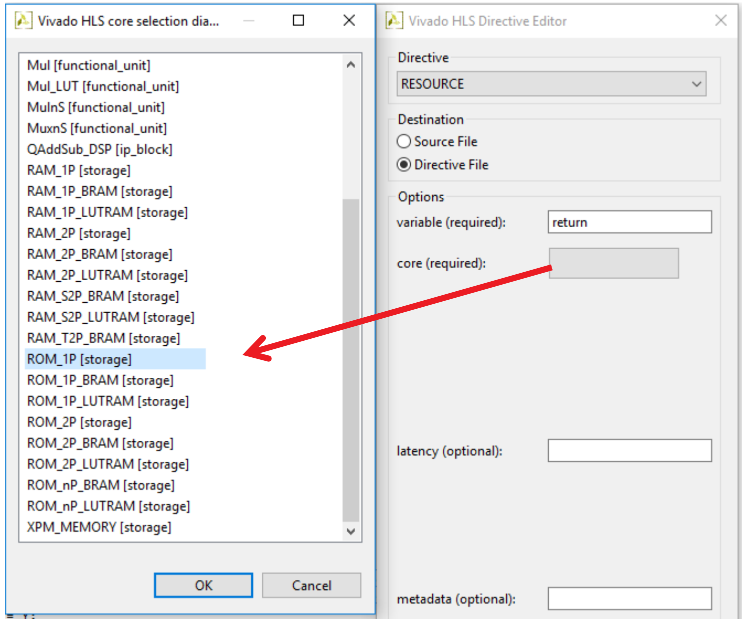
> **Allocations are defined for a scope**
>> Like all directives, allocations are set for the scope they are applied in
– If the directive is applied to a function, loop or region, it does not include objects outside that scope

Operators and Cores are listed in the Vivado HLS Library Guide

Additional Control: Specify Resources

> **User control of Resources**
  - The resource directive gives user control over the specific resource (core) used to implement operations
    - Select the scope & right-click to apply the directive
    - Select “core” for a list of resources
    - Specify the variable

In this example, “data” is implemented with a 2-stage pipelined multiplier



> **Multiple line coding caveat**
  - If multiple operations occur on a single line,
    a temporary variable is required to isolate the specific operation

a=b*c*d

data= c*d;
a = b*data;

XILINX

Reducing Area Usage

Improving Area/Resource Utilization

- **Control the number of elements**
  - Directives can be used to control scheduling and binding
- **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**
  - Vivado HLS provides directives for combining memories
    - Allowing a single large memory to be used instead of multiple smaller memories
- **Bit-width optimization**
  - Arbitrary precision types ensure correct operator sizing

Review: Functions & RTL Hierarchy

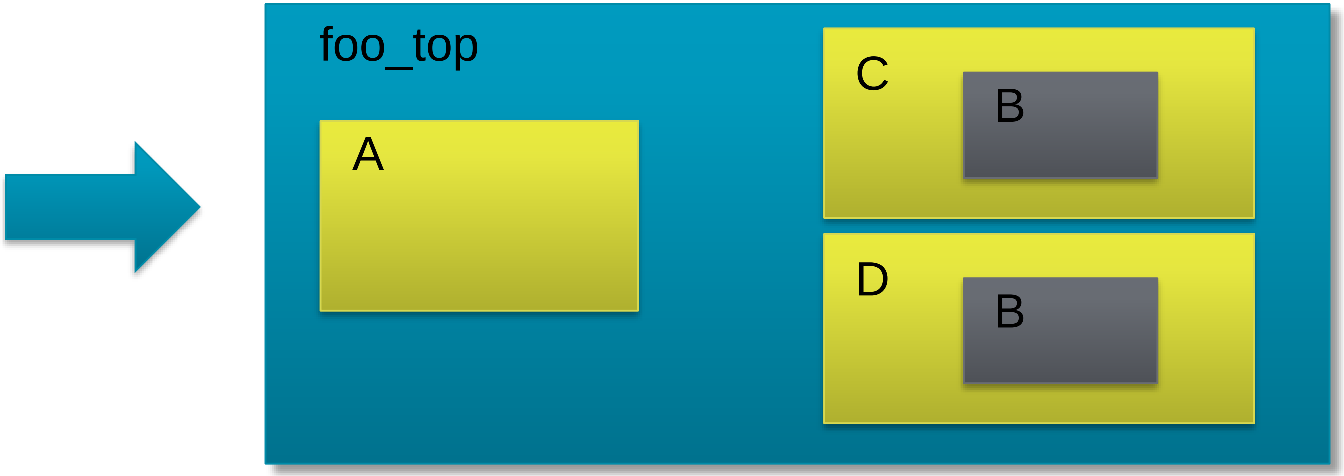
- > **Each function is translated into an RTL block**
  - >> Verilog module, VHDL entity

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

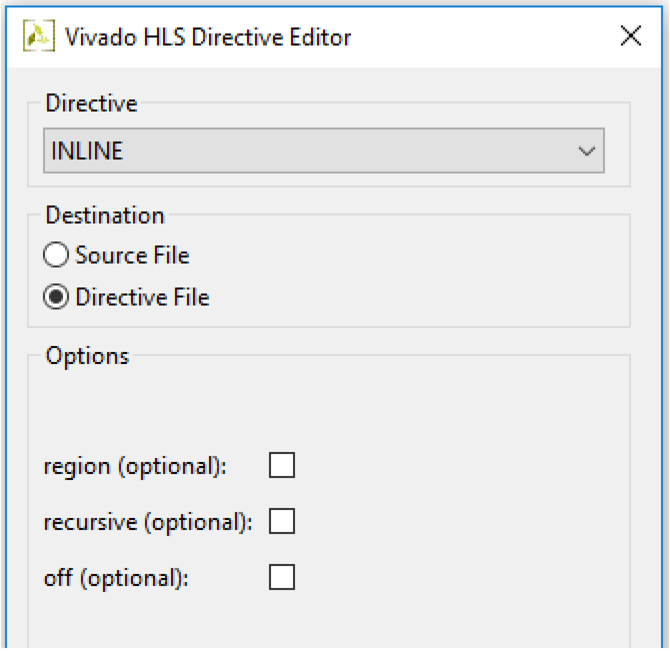


Functions can be inlined – the hierarchy removed & the function dissolved into the surrounding function

XILINX.

Controlling Inlining

- Vivado HLS performs some inlining automatically
  - This is performed on small logic functions if Vivado HLS determines area or performance will benefit
- User Control
  - Functions can be specifically inlined
    - The function itself is inlined
  - Optionally recursively down the hierarchy
  - Optionally everything within a region can be inlined
    - Everything named region or a function or a loop
  - 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

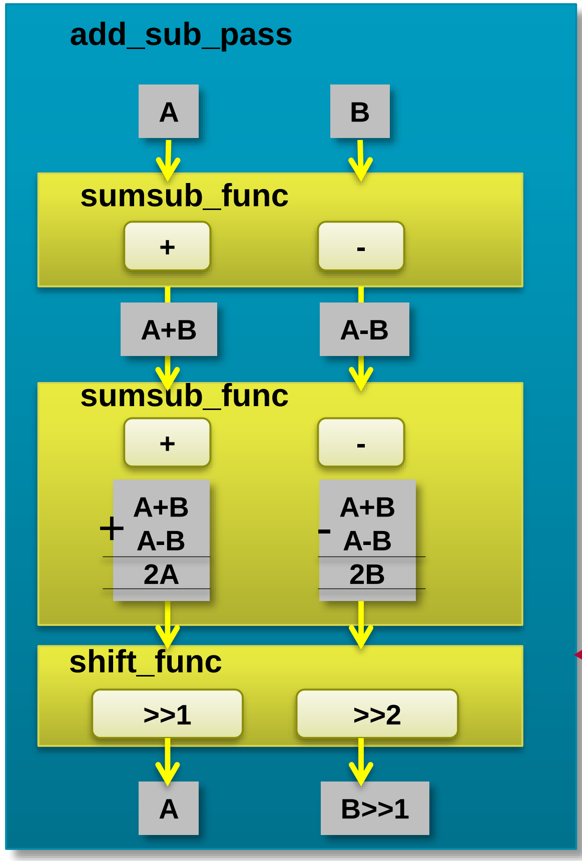


XILINX

Function Inlining

> **Inlining can be used to remove function hierarchy**

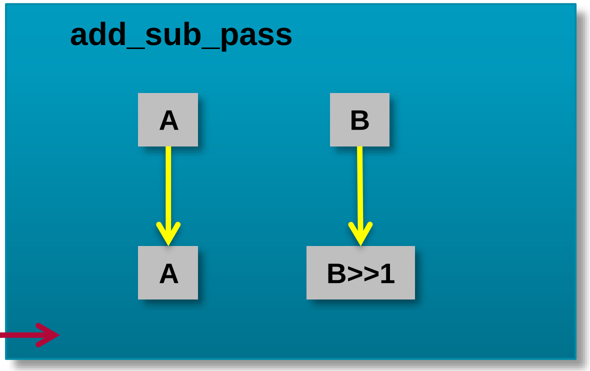
No Inlining




```
int sumsub_func (int *in1, int *in2, int *outSum, int *outSub) {
*outSum = *in1 + *in2;
*outSub = *in1 - *in2;
}

int shift_func (int *in1, int *in2, int *outA, int *outB) {
    *outA = *in1 >> 1;
    *outB = *in2 >> 2;
}

void add_sub_pass(int A, int B, int *C, int *D) {
    int apb, amb;
    int a2, b2;

    sumsub_func(&A,&B,&apb,&amb);
    sumsub_func(&apb,&amb,&a2,&b2);
    shift_func(&a2,&b2,C,D);

}
```

Zero Area

Inlining



2 Adders
2 Subtractors

Inlining allows optimization to be performed across function hierarchies

Like RTL ungrouping, too much inlining can create a lot of logic and slow runtime

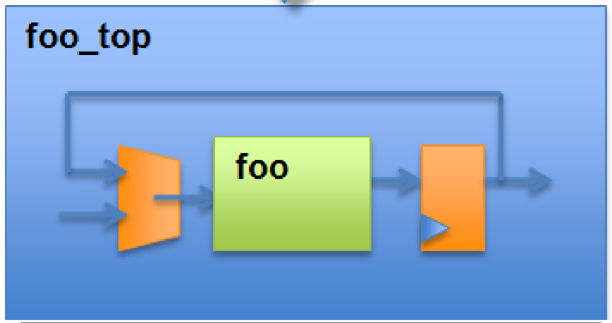
XILINX

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

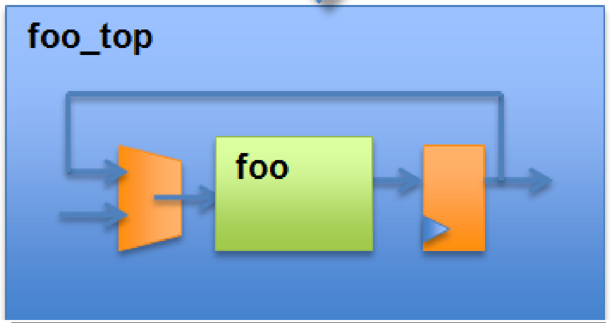
```
void dummy1() {
        foo();
}
void dummy2() {
        foo();
}
void foo_top() {
        dummy1(...);
        dummy2(...);
}
```

```
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
```

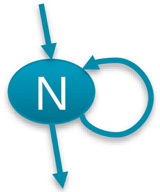


Inlining brings *foo* into function foo_top where it can be shared

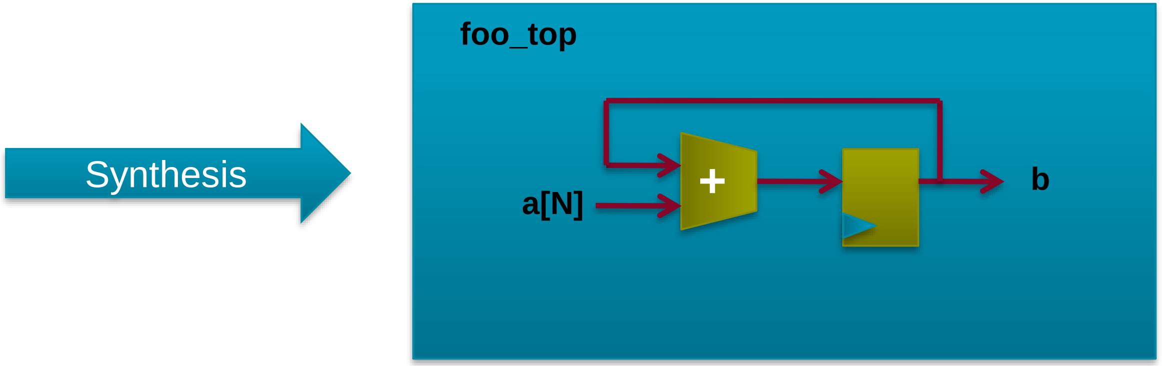
XILINX

Loops

> **By default, loops are rolled**
>> Each C loop iteration ➔ Implemented in the same state
>> Each C loop iteration ➔ Implemented 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 <u>iteration</u> of the loop uses the same hardware resources

XILINX

Loop Merging & Flattening

- **Loop merging & flattening can remove the redundant computation among multiple (related) loops**
  - Improving area (and sometimes 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

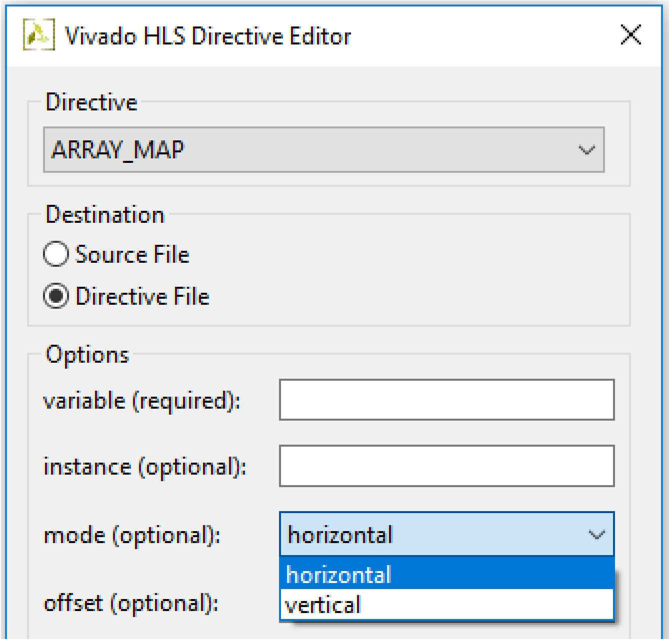
- **Allows Vivado HLS to perform optimizations**
  - Optimization cannot occur across loop boundaries

```
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

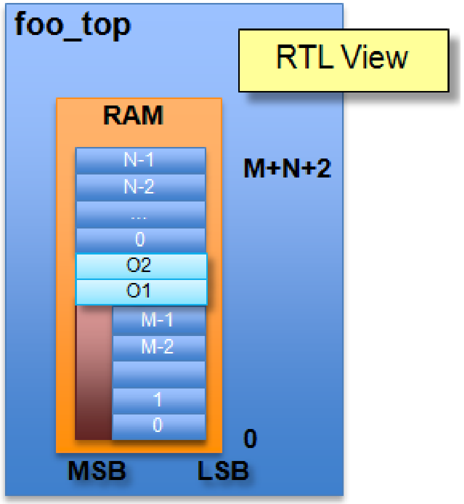
- The arrays in the C model may not be ideal for the available RAMs
  - The code may have many small arrays
  - The array may not utilize the RAMs very well
- Array Mapping
  - Mapping combines smaller arrays into larger arrays
    - Allows arrays to be reconfigured without code edits
  - Specify the array variable to be mapped
  - Give all arrays to be combined the same instance name
- Vivado HLS provides options as to the type of mapping
  - Combine the arrays without impacting performance
    - Vertical & Horizontal mapping
- Global Arrays
  - When a global array is mapped all arrays involved are promoted to global
  - When arrays are in different functions, the target becomes global
- Arrays which are function arguments
  - All must be part of the same function interface

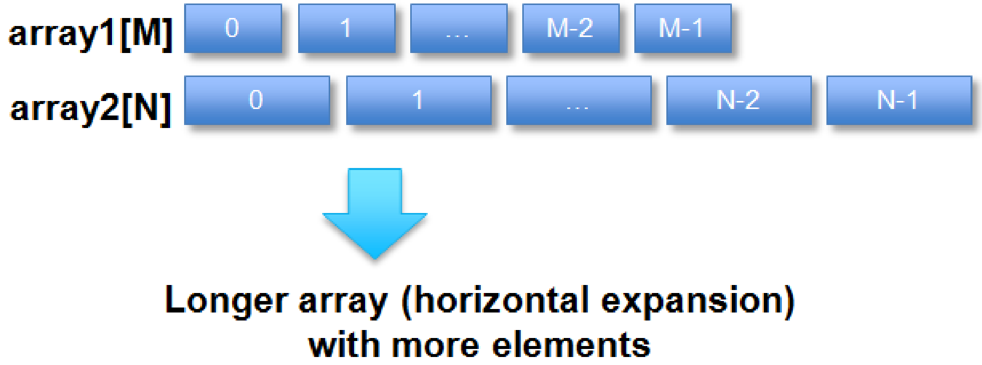


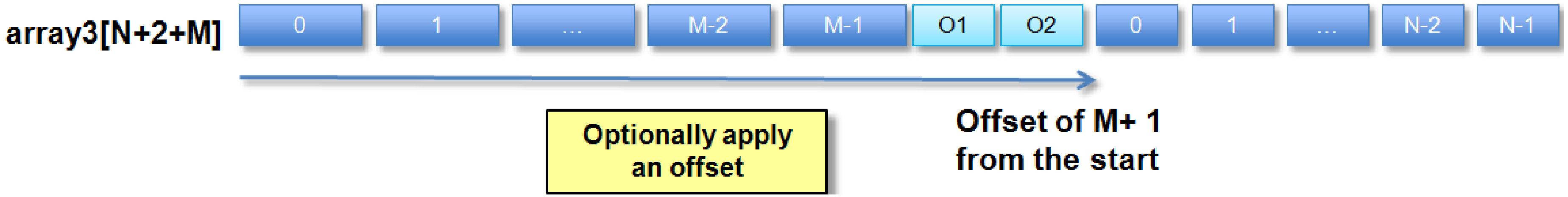
XILINX

Horizontal Mapping

- **Horizontal Mapping**
  - Combines multiple arrays into longer (horizontal) array
  - Optionally allows the arrays to be offset
    - The default is to concatenate after the last element







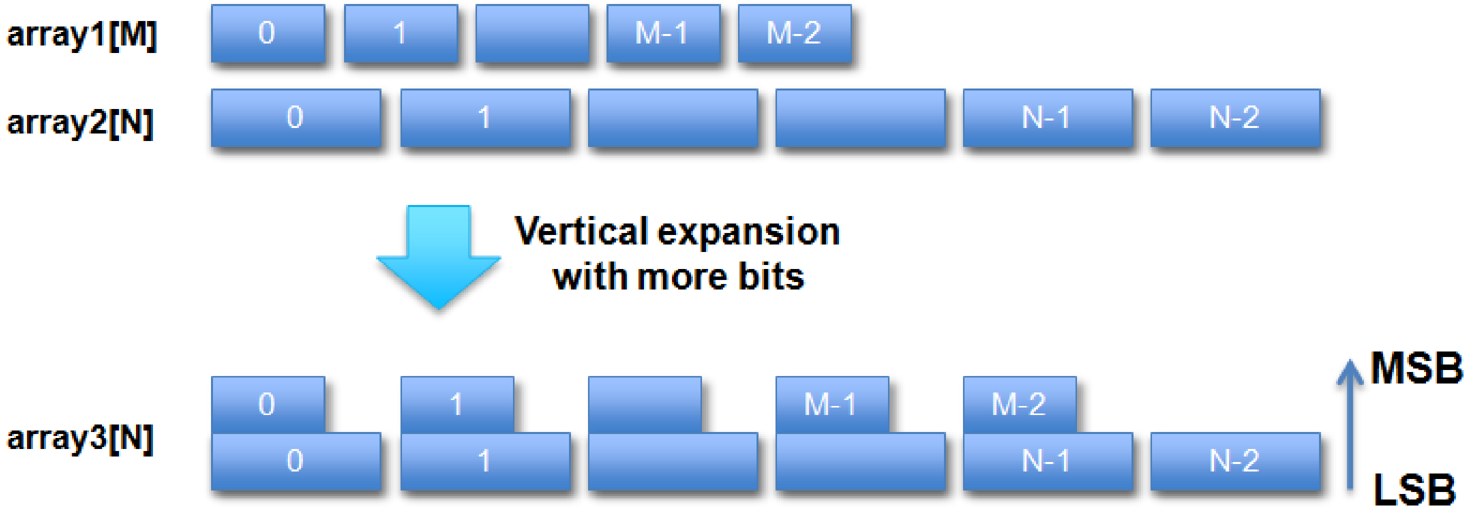
- The first array specified (in GUI or Tcl script) starts at location zero

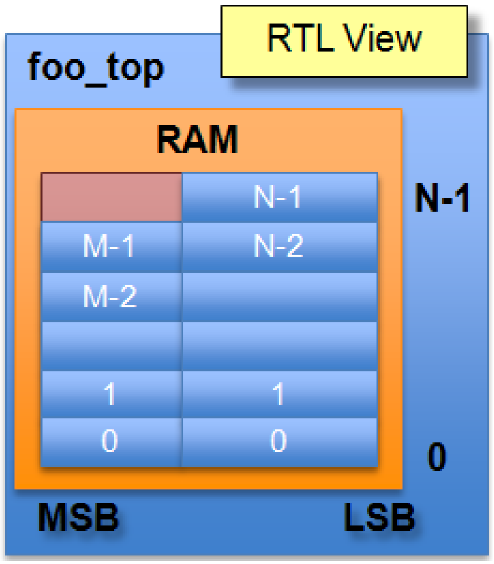
XILINX

Vertical Mapping

> Vertical Mapping

- >> Combines multiple arrays in to an array with more bits





- >> The first array specified (in Tcl or GUI) starts at the LSB

> Vertical Mapping for performance

- >> Creates RAMs with wide words ➔ Parallel accesses

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.
  - They result in hardware which is not bit-accurate and can give sub-standard QoR
- **Vivado HLS provides bit-accurate types in both C and C++**
  - Plus SystemC types can be used in 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

   …
```

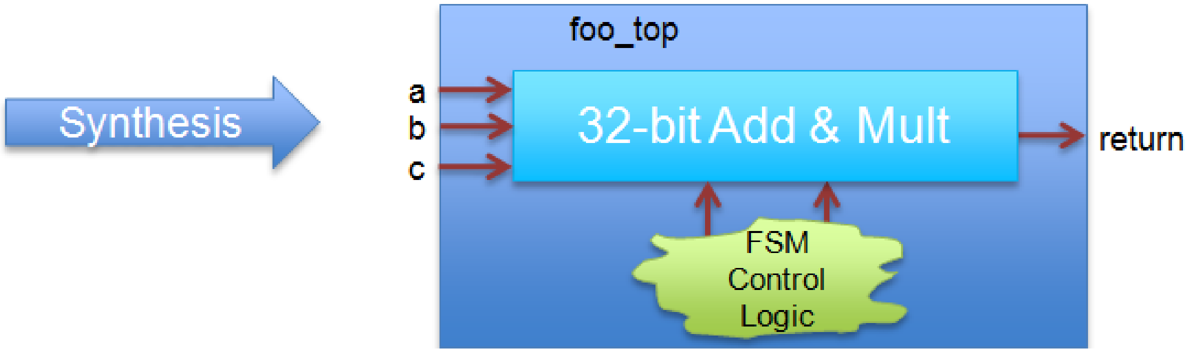
my_code.cpp

```
#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**

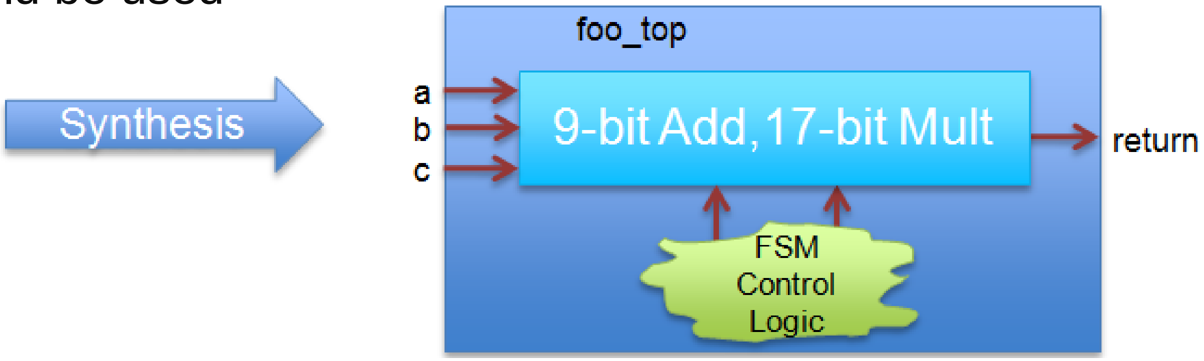
```
int foo_top(int a, int b, int c)
{
    int sum, mult;
    sum=a+b;
    mult=sum*c;
    return mult;
}
```



> **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

XILINX.

Summary

Summary

- **Resource utilization can be reduced using allocation and binding controls**
- **Arbitrary precision data types help controlling both the area and resource utilization**
- **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