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
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

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

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
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

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

Outline

- Optimizing Resource Utilization
- Reducing Area Usage
- Summary
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**
  ```c
  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**
  ```
  foo_top
  
  Functions can be inlined – the hierarchy removed & the function dissolved into the surrounding function
  ```
**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

**Function Inlining**

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

**No Inlining**

```
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);
}
```

**Inlining**

```
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**

- 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**
**Inline and Allocation: Shape the Hierarchy**

**Easy to Share**

```c
void foo() {
    ...}
```

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

Set directive allocation limit 1 for function foo_top foo

One RTL block is reused for both instances of function foo

**Cannot be shared**

```c
void dummy1() {
    foo();
}
```

```c
void dummy2() {
    foo();
}
```

Set directive allocation limit 1 for function foo_top foo

Function foo is not within the immediate scope of foo_top

**Controlling Sharing**

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

Set directive allocation limit 1 for 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**
  - Each C loop iteration implemented in the same state
  - Each C loop iteration implemented with same resources

```c
void foo_top (...) {
    ...
    Add: for (i=N;j=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
Loop Merging & Flattening

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

```c
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;
}
```

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

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

Effective code after compiler transformation

Merge

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

- 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

Vivado HLS Directive Editor

- Type
- Directive: ARRAY_MAP
- Destination
- □ Source File □ Directive File
- Options
  - □ variable (required): col_outbuf
  - □ instance (optional):
  - □ mode (optional):
    - horizontal
    - vertical
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

```
array1[M] 0 1 M-1 M-2 M-3 M-4
array2[N] 0 1 M-1 M-2 M-3 M-4
```

```
Longer array (horizontal expansion) with more elements
array3[N+2+M] 0 1 M-1 M-2 M-3 M-4
```

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

Vertical Mapping

- Combines multiple arrays into an array with more bits

```
array1[M] 0 1 M-1 M-2
array2[N] 0 1 M-1 M-2
```

```
Vertical expansion with more bits
array3[N] 0 1 M-1 M-2
```

- 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

Why are Arbitrary Precision types Needed?

- **Code using native C int type**
  ```c
  #include ap_int.h
  void foo_top(...) {
    int i;
    uint j;
    int k;
    ...
  }
  ```

- **However, if the inputs will only have a max range of 8-bit**
  - Arbitrary precision data-types should be used
  ```c
  #include ap_int.h
  void foo_top(...) {
    ap_int<8> i, j, k;
    ...
  }
  ```
  - It will result in smaller & faster hardware with full precision
Outline

- Optimizing Resource Utilization
- Reducing Area Usage
- 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