

Introduction to High-Level Synthesis with Vivado HLS



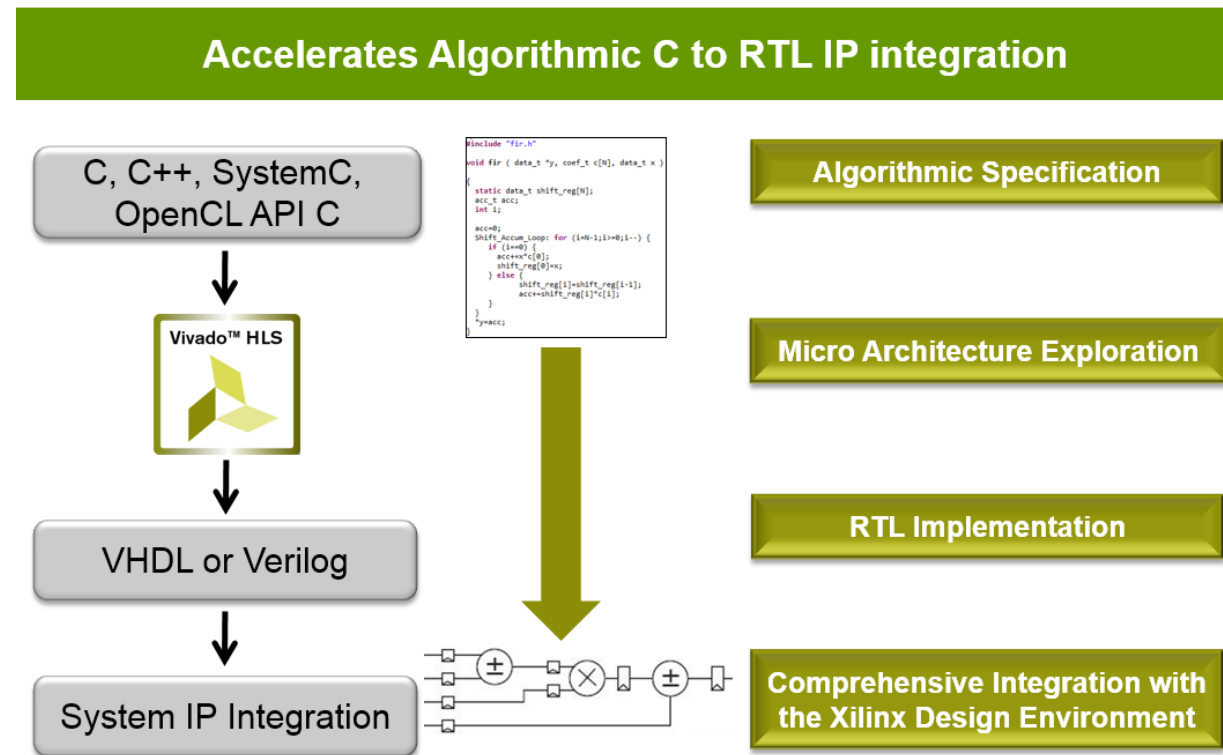
Outline

- **Introduction to High-Level Synthesis**
- **High-Level Synthesis with Vivado HLS**
- **Language Support**
- **Validation Flow**

High-Level Synthesis: HLS

➤ High-Level Synthesis

- Creates an RTL implementation from C, C++, System C, OpenCL API C kernel code
 - Extracts control and dataflow from the source code
 - Implements the design based on defaults and user applied directives
- ## ➤ Many implementation are possible from the same source description
- Smaller designs, faster designs, optimal designs
 - Enables design exploration



Design Exploration with Directives

One body of code:
Many hardware outcomes

```

...
loop: for (i=3;i>=0;i--) {
  if (i==0) {
    acc+=x*c[0];
    shift_reg[0]=x;
  } else {
    shift_reg[i]=shift_reg[i-1];
    acc+=shift_reg[i]*c[i];
  }
}
...
    
```

Before we get into details, let's look under the hood ...

The same hardware is used for each iteration of the loop:

- Small area
- Long latency
- Low throughput

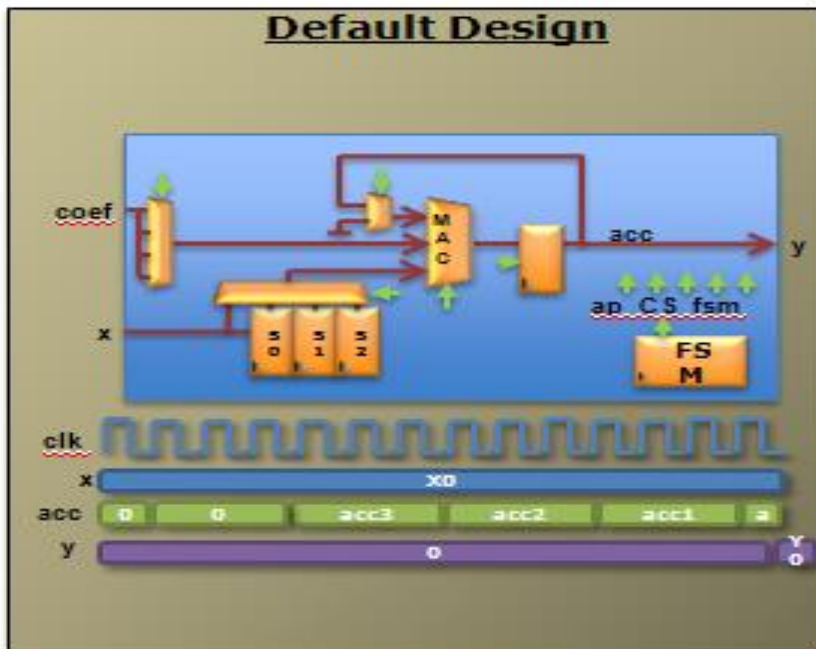
Different hardware is used for each iteration of the loop:

- Higher area
- Short latency
- Better throughput

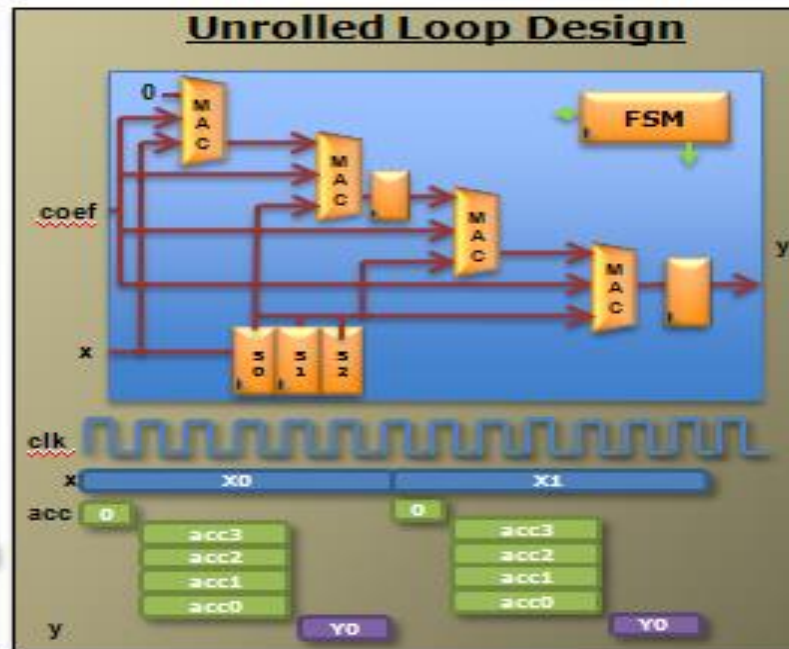
Different iterations are executed concurrently:

- Higher area
- Short latency
- Best throughput

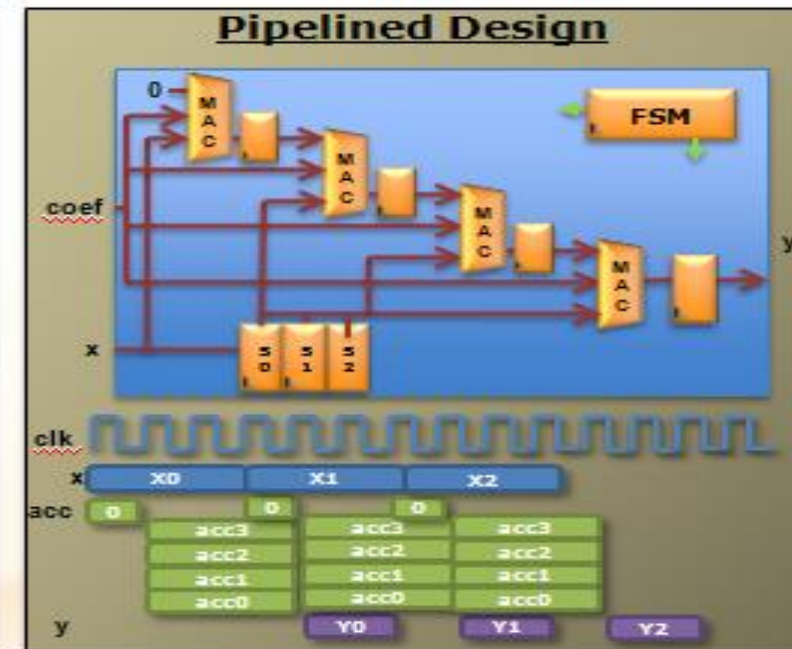
Default Design



Unrolled Loop Design



Pipelined Design



Introduction to High-Level Synthesis

➤ How is hardware extracted from C code?

- Control and datapath can be extracted from C code at the top level
- The same principles used in the example can be applied to sub-functions
 - At some point in the top-level control flow, control is passed to a sub-function
 - Sub-function may be implemented to execute concurrently with the top-level and or other sub-functions

➤ How is this control and dataflow turned into a hardware design?

- Vivado HLS maps this to hardware through scheduling and binding processes

➤ How is my design created?

- How functions, loops, arrays and IO ports are mapped?

HLS: Control Extraction

Code

```
void fir (  
  data_t *y,  
  coef_t c[4],  
  data_t x  
) {  
  
  static data_t shift_reg[4];  
  acc_t acc;  
  int i;  
  
  acc=0;  
  loop: for (i=3;i>=0;i--) {  
    if (i==0) {  
      acc+=x*c[0];  
      shift_reg[0]=x;  
    } else {  
      shift_reg[i]=shift_reg[i-1];  
      acc+=shift_reg[i]*c[i];  
    }  
  }  
  *y=acc;  
}
```

From any C code example ..

Function Start

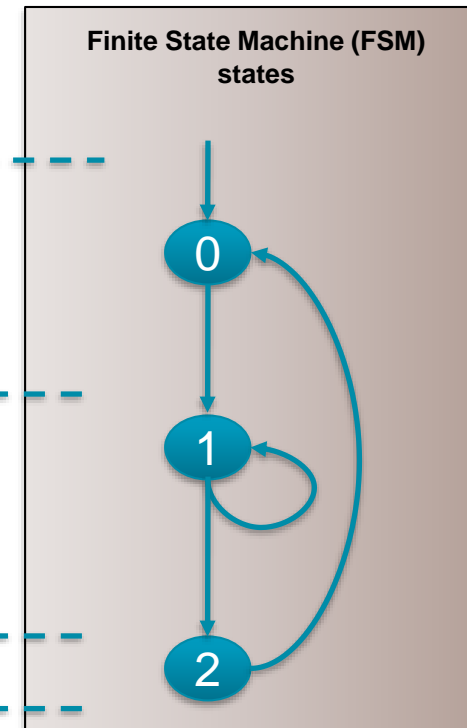
For-Loop Start

For-Loop End

Function End

The loops in the C code correlated to states of behavior

Control Behavior



This behavior is extracted into a hardware state machine

HLS: Control & Datapath Extraction

Code

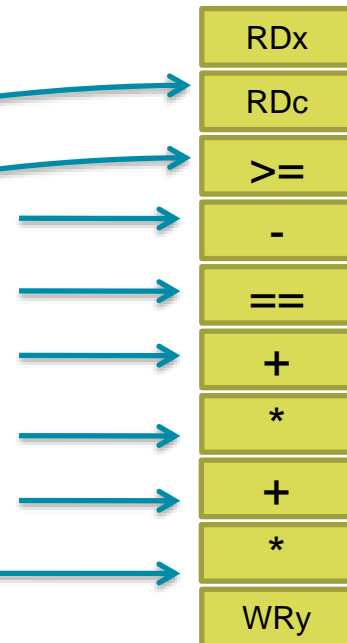
```

void fir (
  data_t *y,
  coef_t c[4],
  data_t x
) {
  static data_t shift_reg[4];
  acc_t acc;
  int i;

  acc=0;
  loop: for (i=3;i>=0;i--) {
    if (i==0) {
      acc+=x*c[0];
      shift_reg[0]=x;
    } else {
      shift_reg[i]=shift_reg[i-1];
      acc+=shift_reg[i]*c[i];
    }
  }
  *y=acc;
}
    
```

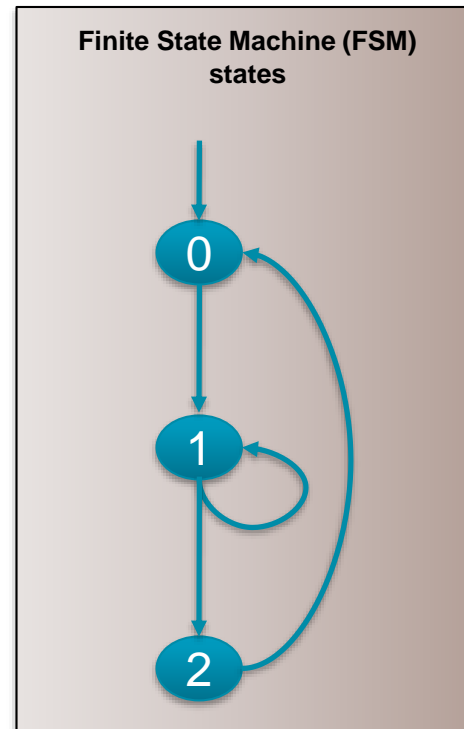
From any C code example ..

Operations



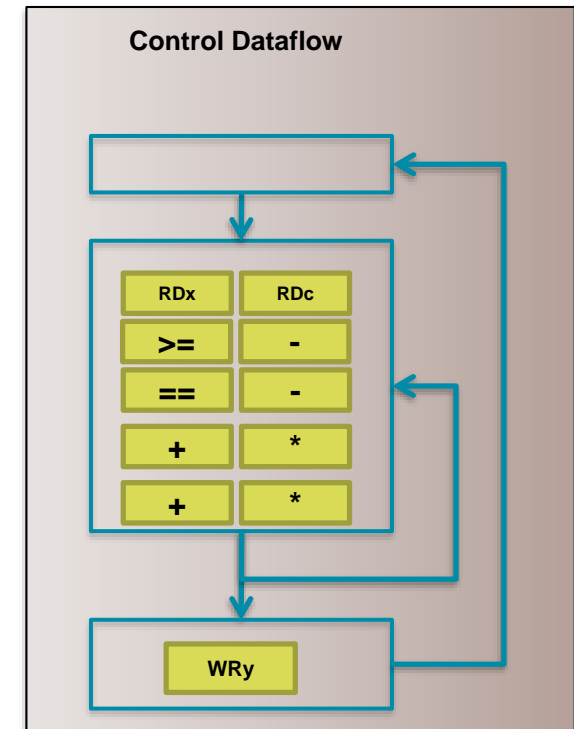
Operations are extracted...

Control Behavior



The control is known

Control & Datapath Behavior



A unified control dataflow behavior is created.

High-Level Synthesis: Scheduling & Binding

➤ Scheduling & Binding

– Scheduling and Binding are at the heart of HLS

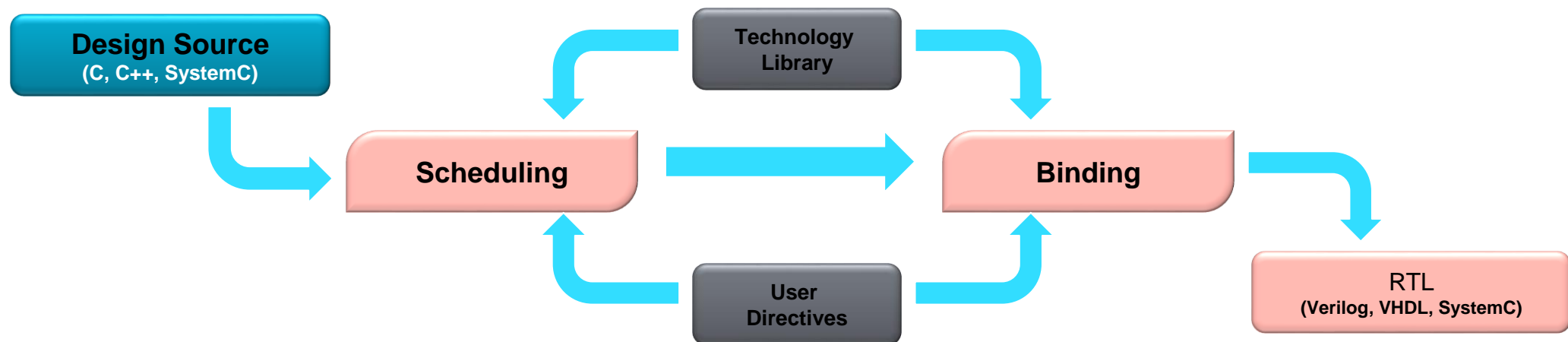
➤ Scheduling determines in which clock cycle an operation will occur

– Takes into account the control, dataflow and user directives

– The allocation of resources can be constrained

➤ Binding determines which library cell is used for each operation

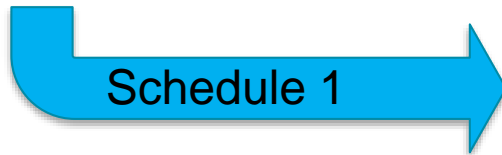
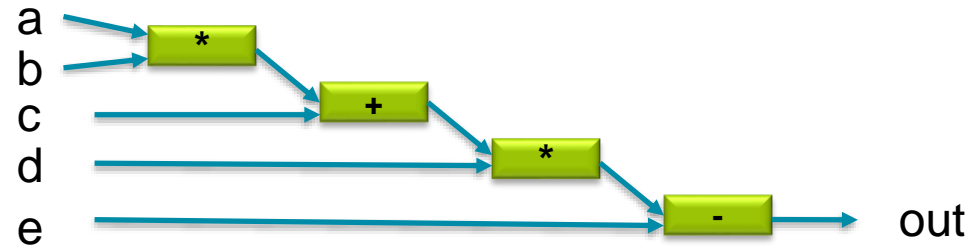
– Takes into account component delays, user directives



Scheduling

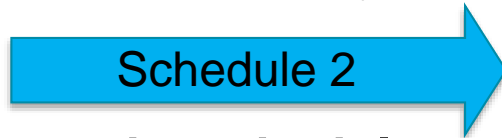
- The operations in the control flow graph are mapped into clock cycles

```
void foo (  
...  
t1 = a * b;  
t2 = c + t1;  
t3 = d * t2;  
out = t3 - e;  
}
```



- The technology and user constraints impact the schedule

- A faster technology (or slower clock) may allow more operations to occur in the same clock cycle



- The code also impacts the schedule

- Code implications and data dependencies must be obeyed

Binding

- **Binding is where operations are mapped to cores from the hardware library**
 - Operators map to cores

- **Binding Decision: to share**

- Given this schedule:



- Binding must use 2 multipliers, since both are in the same cycle
 - It can decide to use an adder and subtractor or *share* one addsub

- **Binding Decision: or not to share**

- Given this schedule:



- Binding may decide to share the multipliers (each is used in a different cycle)
 - Or it may decide the cost of sharing (muxing) would impact timing and it may decide *not to share* them
 - It may make this same decision in the first example above too

High-Level Synthesis with Vivado HLS

Understanding Vivado HLS Synthesis

➤ Vivado HLS

- Determines in which cycle operations should occur (scheduling)
- Determines which hardware units to use for each operation (binding)
- Performs high-level synthesis by :
 - Obeying built-in defaults
 - Obeying user directives & constraints to override defaults
 - Calculating delays and area using the specified technology/device

➤ Priority of directives in Vivado HLS

1. Meet Performance (clock & throughput)
 - Vivado HLS will allow a local clock path to fail if this is required to meet throughput
 - Often possible the timing can be met after logic synthesis
2. Then minimize latency
3. Then minimize area

The Key Attributes of C code

```
void fir (  
    data_t *y,  
    coef_t c[4],  
    data_t x  
) {  
  
    static data_t shift_reg[4];  
    acc_t acc;  
    int i;  
  
    acc=0;  
    loop: for (i=3;i>=0;i--) {  
        if (i==0) {  
            acc+=x*c[0];  
            shift_reg[0]=x;  
        } else {  
            shift_reg[i]=shift_reg[i-1];  
            acc+=shift_reg[i] * c[i];  
        }  
    }  
    *y=acc;  
}
```

Functions: All code is made up of functions which represent the design hierarchy: the same in hardware

Top Level IO : The arguments of the top-level function determine the hardware RTL interface ports

Types: All variables are of a defined type. The type can influence the area and performance

Loops: Functions typically contain loops. How these are handled can have a major impact on area and performance

Arrays: Arrays are used often in C code. They can influence the device IO and become performance bottlenecks

Operators: Operators in the C code may require sharing to control area or specific hardware implementations to meet performance

Let's examine the default synthesis behavior of these ...

Functions & RTL Hierarchy

➤ Each function is translated into an RTL block

– Verilog module, VHDL entity

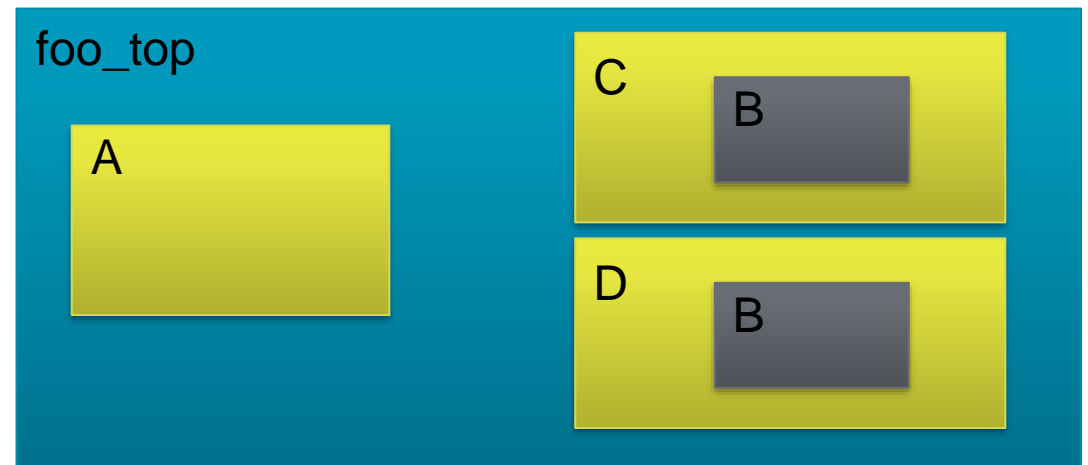
Source Code

```
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



Each function/block can be shared like any other component (add, sub, etc) provided it's not in use at the same time

- By default, each function is implemented using a common instance
- Functions may be inlined to dissolve their hierarchy
 - Small functions may be automatically inlined

Types = Operator Bit-sizes

Code

```

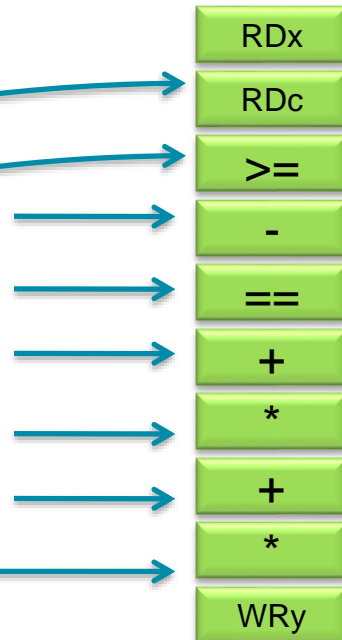
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
) {

  static data_t shift_reg[4];
  acc_t acc;
  int i;

  acc=0;
  loop: for (i=3;i>=0;i--) {
    if (i==0) {
      acc+=x*c[0];
      shift_reg[0]=x;
    } else {
      shift_reg[i]=shift_reg[i-1];
      acc+=shift_reg[i]*c[i];
    }
  }
  *y=acc;
}
    
```

From any C code example ...

Operations



Operations are extracted...

Types

Standard C types

long long (64-bit)	short (16-bit)	unsigned types
int (32-bit)	char (8-bit)	
float (32-bit)	double (64-bit)	

Arbitrary Precision types

C:	ap(u)int types (1-1024)
C++:	ap_(u)int types (1-1024) ap_fixed types
C++/SystemC:	sc_(u)int types (1-1024) sc_fixed types

Can be used to define any variable to be a specific bit-width (e.g. 17-bit, 47-bit etc).

The C types define the size of the hardware used: handled automatically

Loops

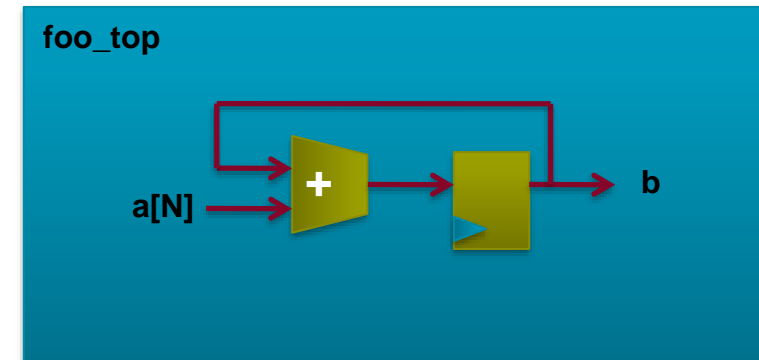
► By default, loops are rolled

- 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;  
    }  
    ...  
}
```

**Loops require labels if they are to be referenced by Tcl directives
(GUI will auto-add labels)**

Synthesis



- Loops can be unrolled if their indices are statically determinable at elaboration time
 - Not when the number of iterations is variable
- Unrolled loops result in more elements to schedule but greater operator mobility
 - Let's look at an example

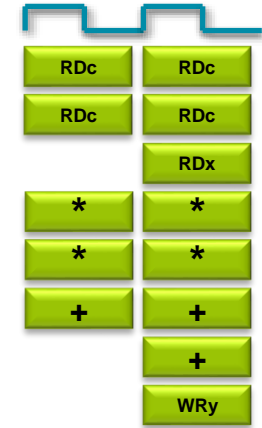
Schedule after Loop Optimization

➤ With the loop unrolled (completely)

- The dependency on loop iterations is gone
- Operations can now occur in parallel
 - If data dependencies allow
 - If operator timing allows
- Design finished faster but uses more operators
 - 2 multipliers & 2 Adders

➤ Schedule Summary

- All the logic associated with the loop counters and index checking are now gone
- Two multiplications can occur at the same time
 - All 4 could, but it's limited by the number of input reads (2) on coefficient port C
- Why 2 reads on port C?
 - The default behavior for arrays now limits the schedule...

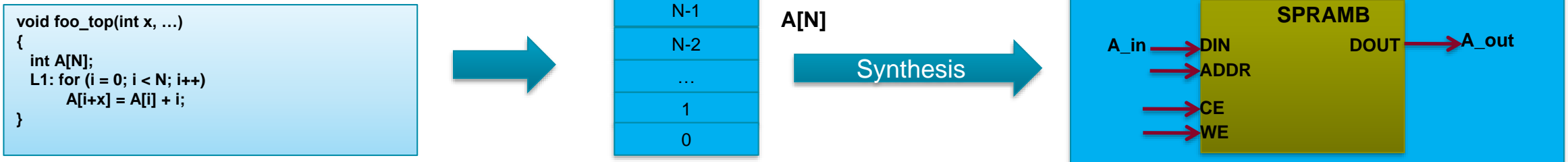


```
void fir (  
  ...  
  acc=0;  
  loop: for (i=3;i>=0;i--) {  
    if (i==0) {  
      acc+=x*c[0];  
      shift_reg[0]=x;  
    } else {  
      shift_reg[i]=shift_reg[i-1];  
      acc+=shift_reg[i]*c[i];  
    }  
  }  
  *y=acc;  
}
```

Arrays in HLS

➤ **An array in C code is implemented by a memory in the RTL**

- By default, arrays are implemented as RAMs, optionally a FIFO



➤ **The array can be targeted to any memory resource in the library**

- The ports (Address, CE active high, etc.) and sequential operation (clocks from address to data out) are defined by the library model
- All RAMs are listed in the Vivado HLS Library Guide

➤ **Arrays can be merged with other arrays and reconfigured**

- To implement them in the same memory or one of different widths & sizes

➤ **Arrays can be partitioned into individual elements**

- Implemented as smaller RAMs or registers

Top-Level IO Ports

➤ Top-level function arguments

– All top-level function arguments have a default hardware port type

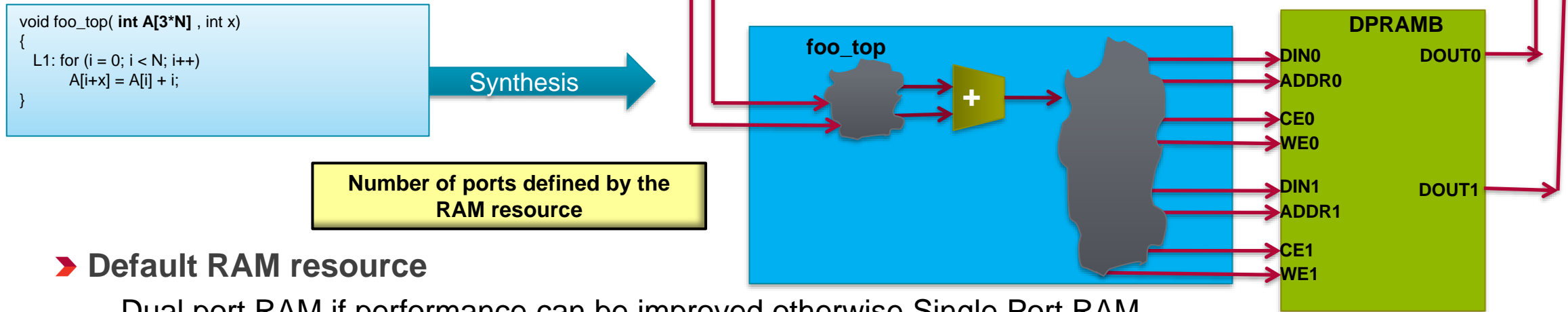
➤ When the array is an argument of the top-level function

– The array/RAM is “off-chip”

– The type of memory resource determines the top-level IO ports

– Arrays on the interface can be mapped & partitioned

- E.g. partitioned into separate ports for each element in the array



➤ Default RAM resource

– Dual port RAM if performance can be improved otherwise Single Port RAM

Schedule after an Array Optimization

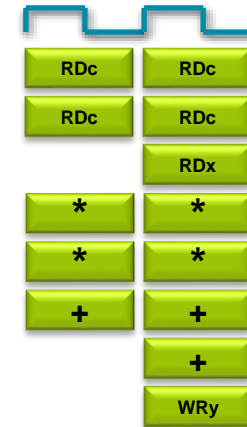
➤ With the existing code & defaults

- Port C is a dual port RAM
- Allows 2 reads per clock cycles
 - IO behavior impacts performance

Note: It could have performed 2 reads in the original rolled design but there was no advantage since the rolled loop forced a single read per cycle

➤ With the C port partitioned into (4) separate ports

- All reads and mults can occur in one cycle
- If the timing allows
 - The additions can also occur in the same cycle
 - The write can be performed in the same cycles
 - Optionally the port reads and writes could be registered



```
loop: for (i=3;i>=0;i--) {  
  if (i==0) {  
    acc+=x*c[0];  
    shift_reg[0]=x;  
  } else {  
    shift_reg[i]=shift_reg[i-1];  
    acc+=shift_reg[i]*c[i];  
  }  
}  
*y=acc;
```



Language Support

Comprehensive C Support

➤ **A Complete C Validation & Verification Environment**

- Vivado HLS supports complete bit-accurate validation of the C model
- Vivado HLS provides a productive C-RTL co-simulation verification solution

➤ **Vivado HLS supports C, C++, SystemC and OpenCL API C kernel**

- Functions can be written in any version of C
- Wide support for coding constructs in all three variants of C

➤ **Modeling with bit-accuracy**

- Supports arbitrary precision types for all input languages
- Allowing the exact bit-widths to be modeled and synthesized

➤ **Floating point support**

- Support for the use of float and double in the code

➤ **Support for OpenCV functions**

- Enable migration of OpenCV designs into Xilinx FPGA
- Libraries target real-time full HD video processing

C, C++ and SystemC Support

- **The vast majority of C, C++ and SystemC is supported**
 - Provided it is statically defined at **compile time**
 - If it's not defined until **run time**, it won't be synthesizable

- **Any of the three variants of C can be used**
 - If C is used, Vivado HLS expects the file extensions to be .c
 - For C++ and SystemC it expects file extensions .cpp

Validation Flow

C Validation and RTL Verification

➤ There are two steps to verifying the design

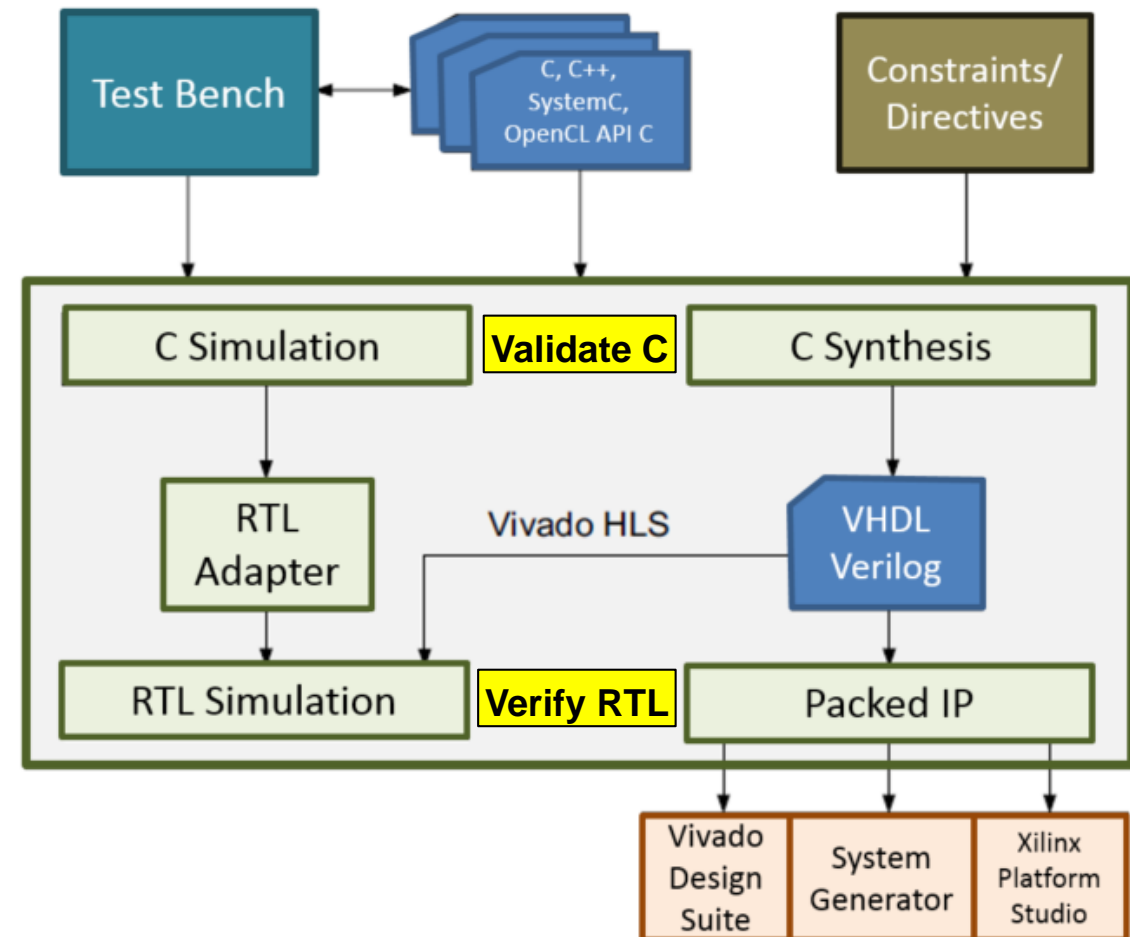
- Pre-synthesis: **C Validation**
 - Validate the algorithm is correct
- Post-synthesis: **RTL Verification**
 - Verify the RTL is correct

➤ C validation

- A **HUGE** reason users want to use HLS
 - Fast, free verification
- Validate the algorithm is correct **before** synthesis
 - Follow the test bench tips given over

➤ RTL Verification

- Vivado HLS can co-simulate the RTL with the original test bench



C Function Test Bench

➤ The test bench is the level above the function

- The main() function is above the function to be synthesized

➤ Good Practices

- The test bench should compare the results with golden data
 - Automatically confirms any changes to the C are validated and verifies the RTL is correct
- The test bench should return a 0 if the self-checking is correct
 - Anything but a 0 (zero) will cause RTL verification to issue a FAIL message
 - Function main() should expect an integer return (non-void)

```
int main () {  
    int ret=0;  
    ...  
    ret = system("diff --brief -w output.dat output.golden.dat");  
    if (ret != 0) {  
        printf("Test failed !!!\n");  
        ret=1;  
    } else {  
        printf("Test passed !\n");  
    }  
    ...  
    return ret;  
}
```

Determine or Create the Top-level Function

- Determine the top-level function for synthesis
- If there are Multiple functions, they must be merged
 - There can only be 1 top-level function for synthesis

Given a case where functions func_A and func_B are to be implemented in FPGA

main.c

```
int main () {  
    ...  
    func_A(a,b,*i1);  
    func_B(c,*i1,*i2);  
    func_C(*i2,ret)  
  
    return ret;  
}
```

func_A

func_B

func_C

Re-partition the design to create a **new** single top-level function inside main()

main.c

```
#include func_AB.h  
int main (a,b,c,d) {  
    ...  
    // func_A(a,b,i1);  
    // func_B(c,i1,i2);  
    func_AB (a,b,c, *i1, *i2);  
    func_C(*i2,ret)  
  
    return ret;  
}
```

func_AB

func_C

func_AB.c

```
#include func_AB.h  
func_AB(a,b,c, *i1, *i2) {  
    ...  
    func_A(a,b,*i1);  
    func_B(c,*i1,*i2);  
    ...  
}
```

func_A

func_B

Recommendation is to separate test bench and design files