XILINX

ALL PROGRAMMABLE™

Introduction to High-Level Synthesis with Vivado HLS

Vivado HLS 2013.3 Version

© Copyright 2013 Xilinx

Objectives

❯ **After completing this module, you will be able to:**

- – Describe the high level synthesis flow
- – Understand the control and datapath extraction
- – Describe scheduling and binding phases of the HLS flow
- – List the priorities of directives set by Vivado HLS
- – List comprehensive language support in Vivado HLS
- – Identify steps involved in validation and verification flows

XILINX ❯ ALL PROGRAMMABLE.

Outline

➤ *Introduction to High-Level Synthesis*

➤ **High-Level Synthesis with Vivado HLS**

➤ **Language Support**

➤ **Validation Flow**

➤ **Summary**

XILINX ➤ ALL PROGRAMMABLE™

Need for High-Level Synthesis

> **Algorithmic-based approaches are getting popular due to accelerated design time and time to market (TTM)**
> – Larger designs pose challenges in design and verification of hardware at HDL level

> **Industry trend is moving towards hardware acceleration to enhance performance and productivity**
> – CPU-intensive tasks can be offloaded to hardware accelerator in FPGA
> – Hardware accelerators require a lot of time to understand and design

> **Vivado HLS tool converts algorithmic description written in C-based design flow into hardware description (RTL)**
> – Elevates the abstraction level from RTL to algorithms

> **High-level synthesis is essential for maintaining design productivity for large designs**

XILINX ➤ ALL PROGRAMMABLE.

High-Level Synthesis: HLS

High-Level Synthesis

- Creates an RTL implementation from C level source 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

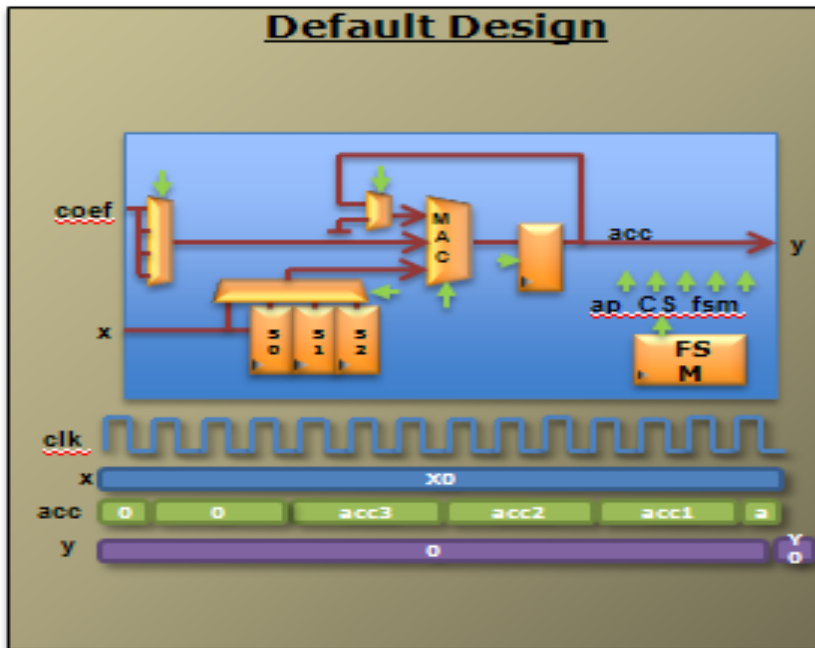- Smaller designs, faster designs, optimal designs
- Enables design exploration

XILINX ➤ ALL PROGRAMMABLE.
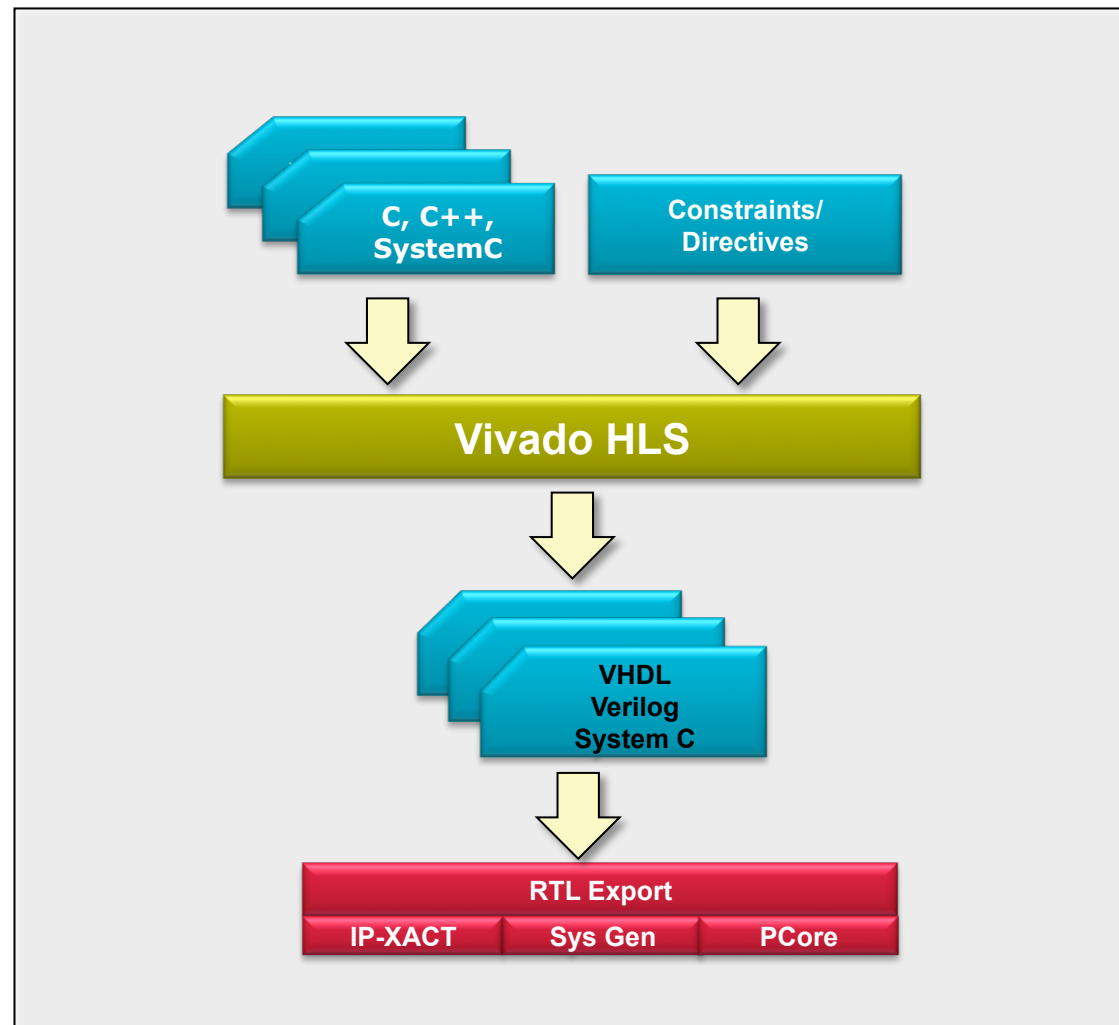
Design Exploration with Directives

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];
    }
}
….
```
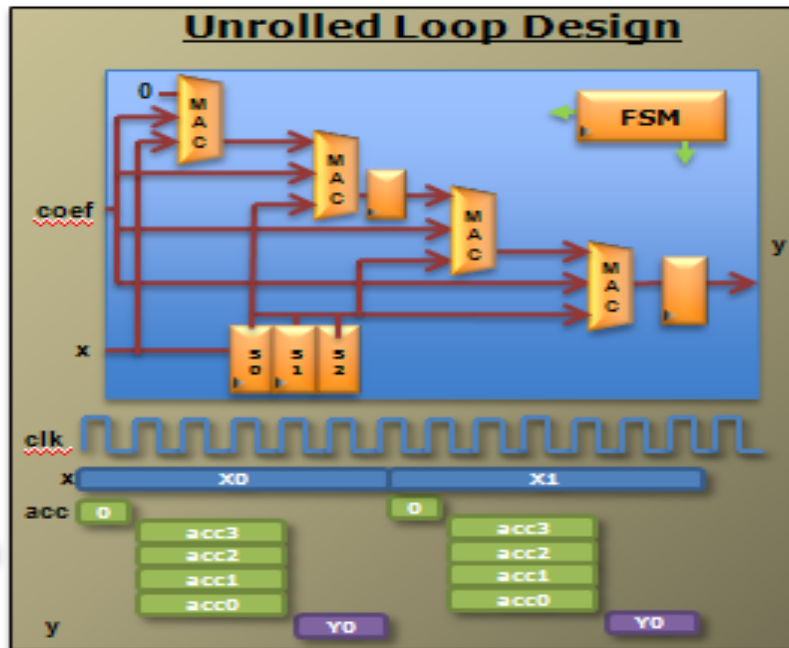
Before we get into details, let's look under the hood ….

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:
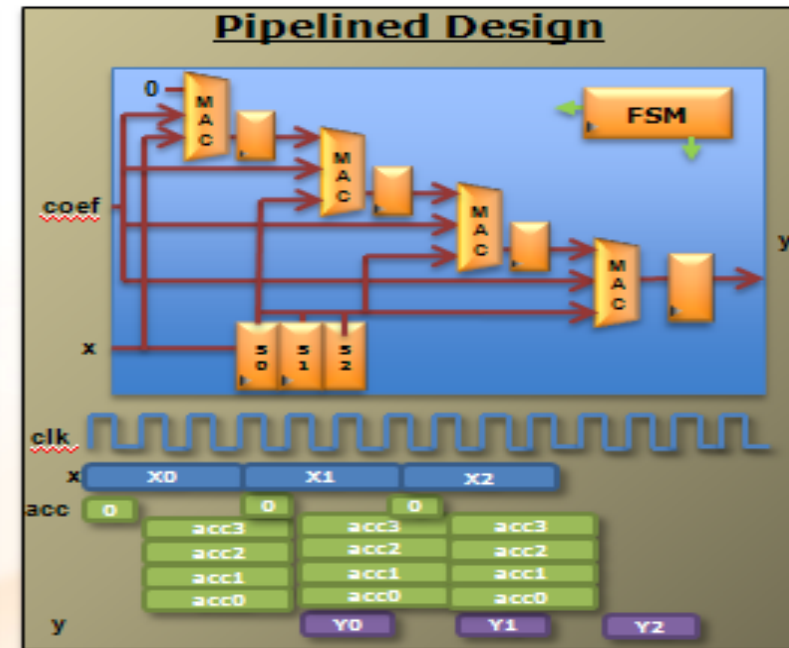- Higher area
- Short latency
- Better throughput

Different iterations are executed concurrently:
- Higher area
- Short latency
- Best throughput

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?

XILINX ➤ ALL PROGRAMMABLE™

HLS: Control Extraction

Code

Control Behavior

```
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;
}
```

Finite State Machine (FSM) states

Function Start

For-Loop Start

For-Loop End

Function End

0

1

2

From any C code example ..

The loops in the C code correlated to states of behavior

This behavior is extracted into a hardware state machine

XILINX ➤ ALL PROGRAMMABLE.

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;
}
```
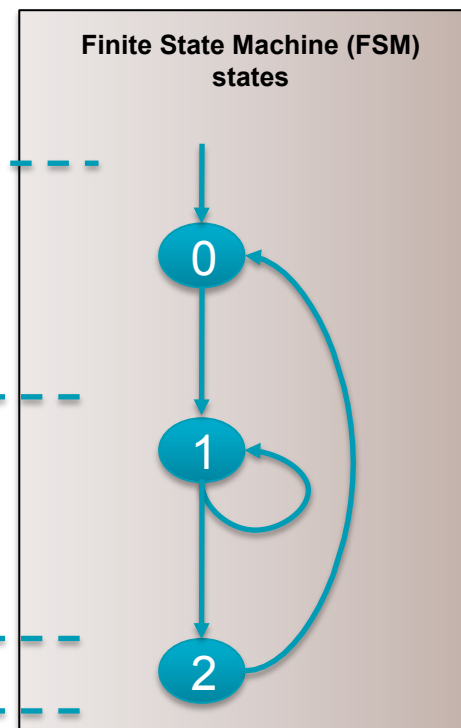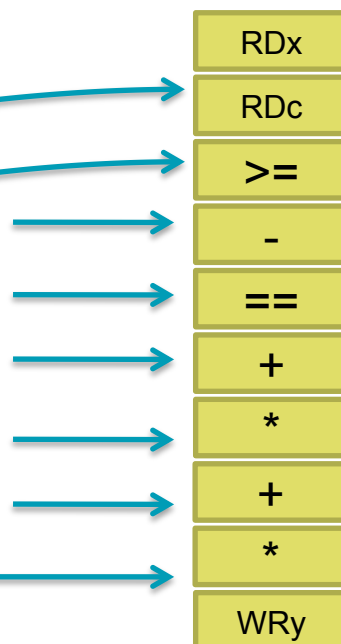
Operations

| RDx |
| RDc |
>=
==
+
*
+
*
WRy

Control Behavior

Finite State Machine (FSM) states

0
1
2

Control & Datapath Behavior

Control Dataflow

RDx	RDc
>=	-
==	-
+	*
+	*

WRy

From any C code example ..

Operations are extracted…
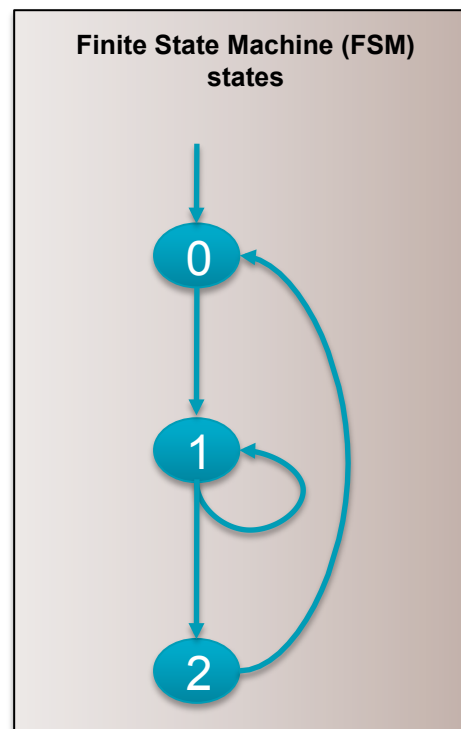
The control is known
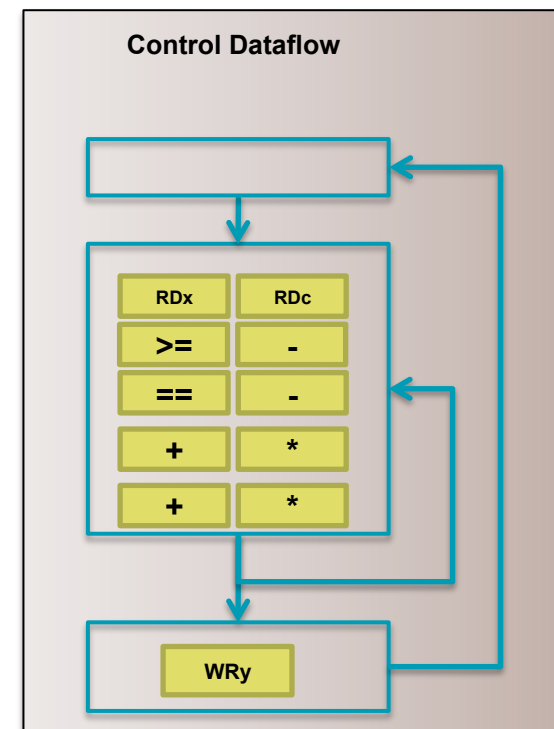
A unified control dataflow behavior is created.

XILINX ➤ ALL PROGRAMMABLE™

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
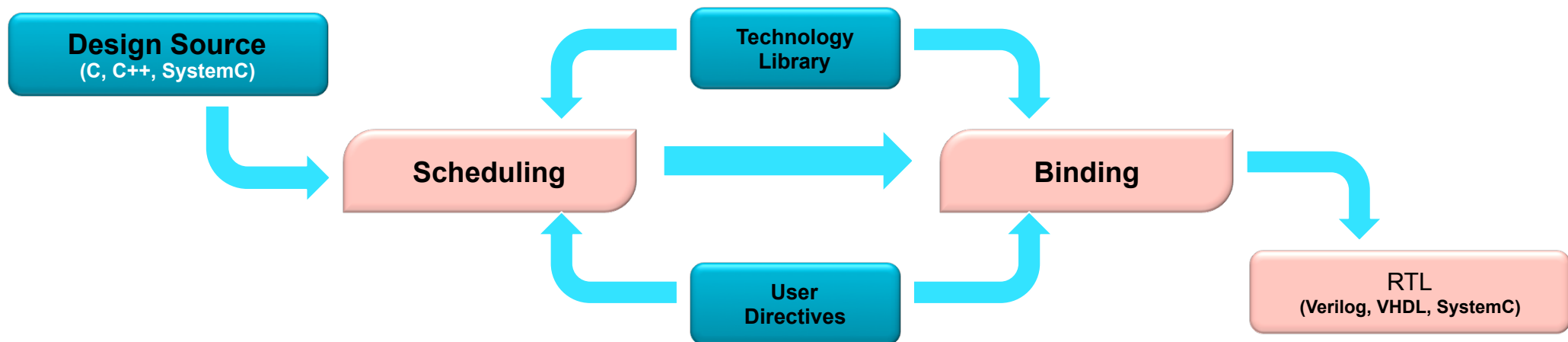
Scheduling

» **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;
}
```

a
b
c
d
e

out
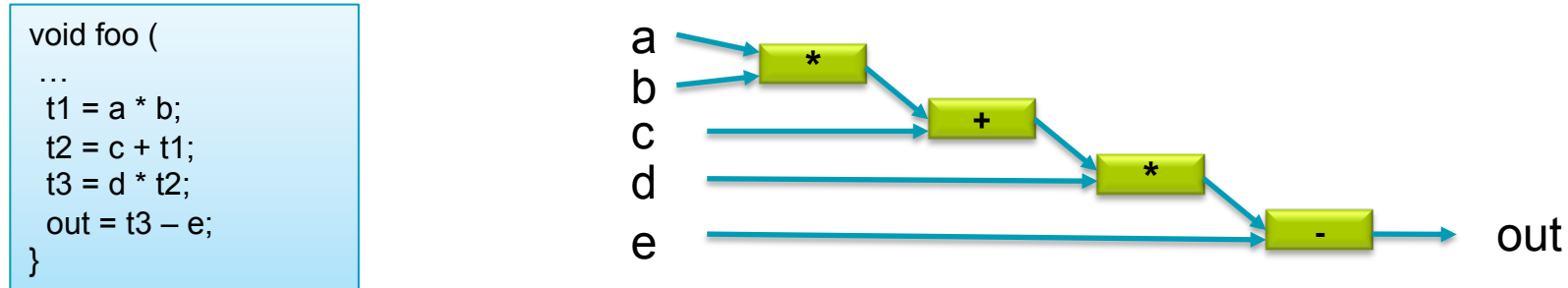
Schedule 1

» **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

Schedule 2

» **The code also impacts the schedule**
  – Code implications and data dependencies must be obeyed

XILINX ➤ ALL PROGRAMMABLE.

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 <u>and</u> subtractor <u>or</u> *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

 XILINX ➤ ALL PROGRAMMABLE.

Outline

> **Introduction to High-Level Synthesis**

> ***High-Level Synthesis with Vivado HLS***

> **Language Support**

> **Validation Flow**

> **Summary**

 XILINX ➤ ALL PROGRAMMABLE.

Vivado HLS Benefits

Productivity

- Verification
  - Functional
  - Architectural
- Abstraction
  - Datatypes
  - Interface
  - Classes
- Automation

Video Design Example			
Input	C Simulation Time	RTL Simulation Time	Improvement
10 frames 1280x720	10s	~2 days (ModelSim)	~12000x

RTL (Spec) · RTL (Sim)

C (Spec/Sim) · RTL (Sim)
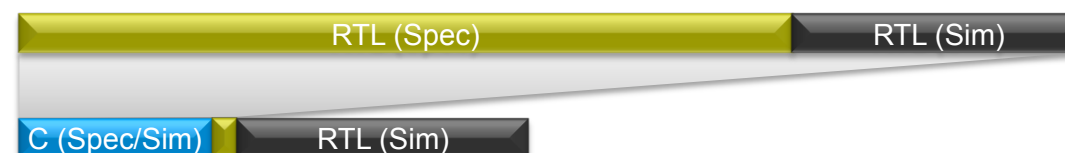
Block level specification AND verification _significantly_ reduced

XILINX ➤ ALL PROGRAMMABLE.

Vivado HLS Benefits

❯ Portability

- – Processors and FPGAs
- – Technology migration
- – Cost reduction
- – Power reduction

❯ Design and IP reuse

© Copyright 2013 Xilinx
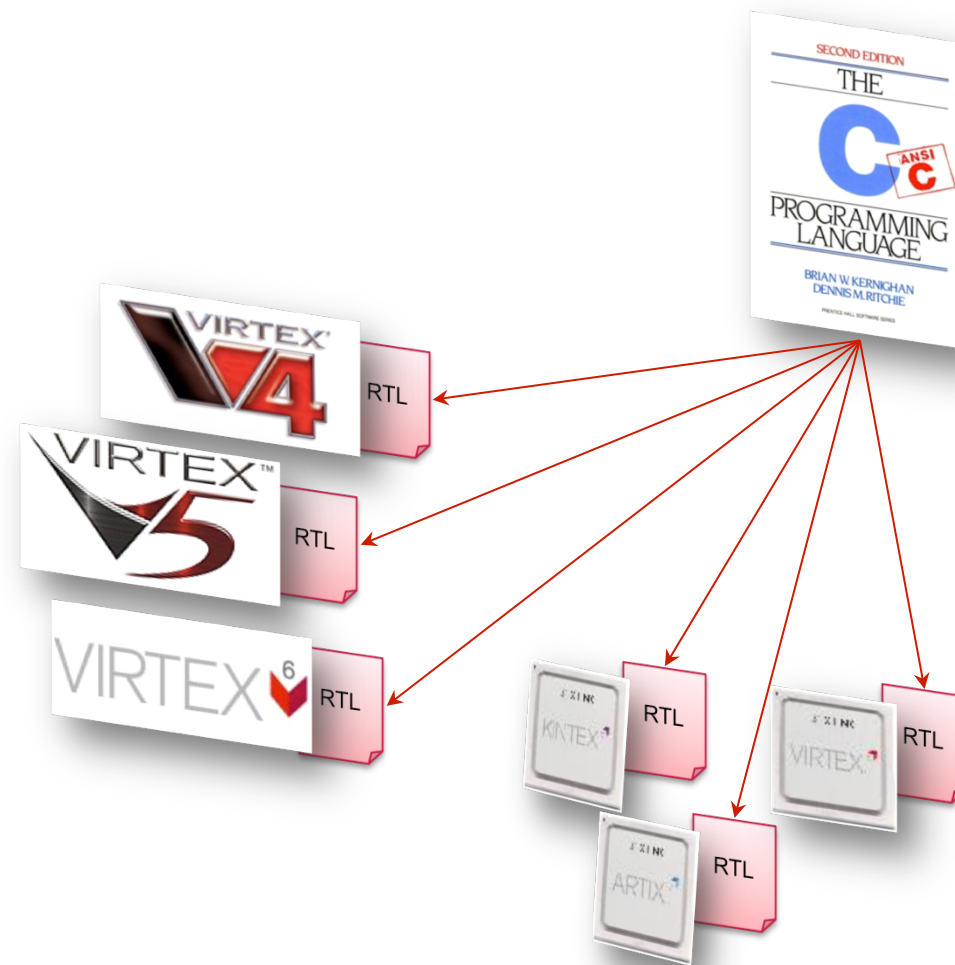
Vivado HLS Benefits

> **Permutability**

- Architecture Exploration
  - Timing
    - Parallelization
    - Pipelining
  - Resources
    - Sharing
- Better QoR

> **Rapid design exploration delivers QoR rivaling hand-coded RTL**

Understanding Vivado HLS Synthesis

➤ **HLS**

– Vivado HLS determines in which cycle operations should occur (scheduling)

– Determines which hardware units to use for each operation (binding)

– It performs HLS by :

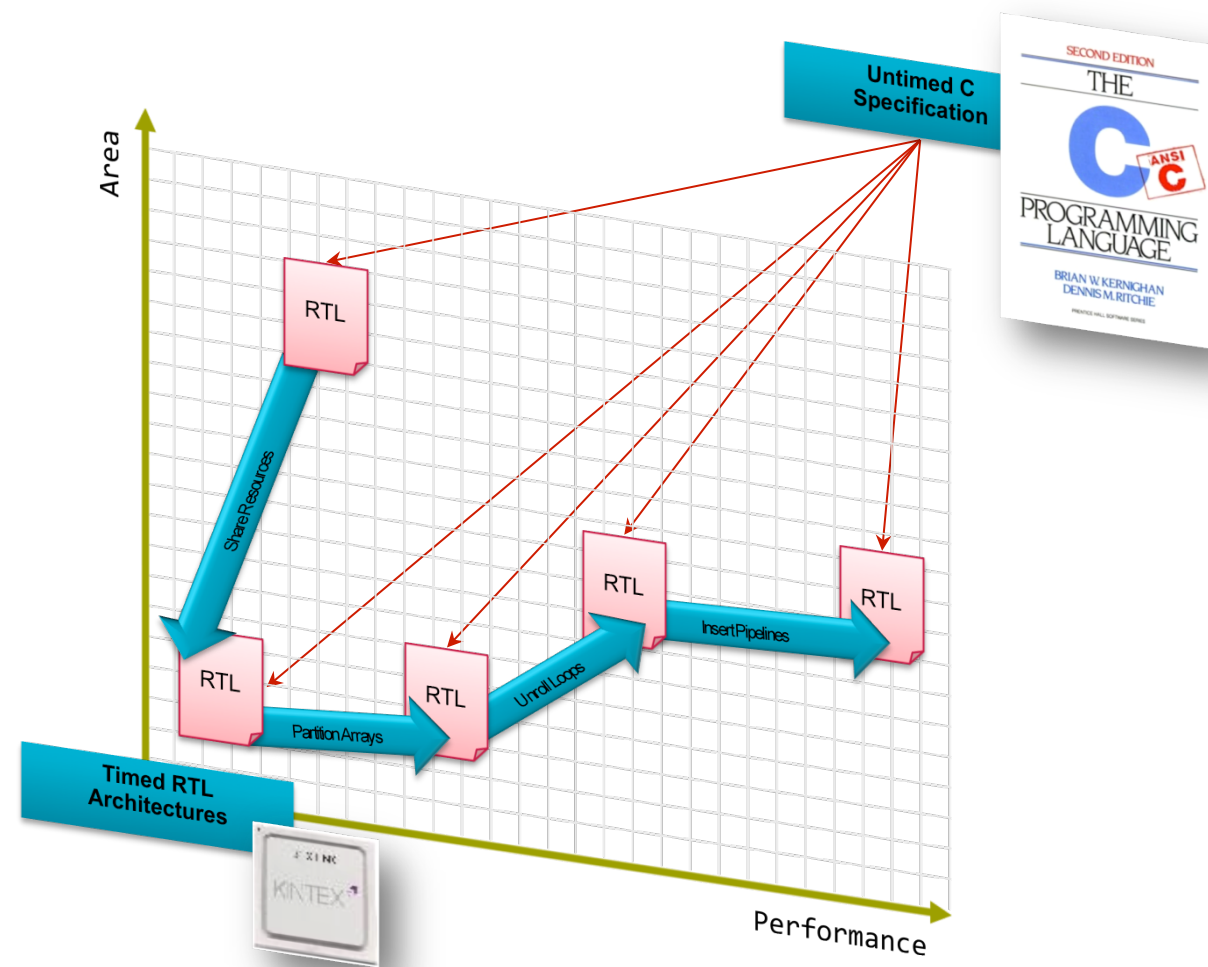 • Obeying built-in defaults

  • Obeying user directives & constraints to override defaults

  • Calculating delays and area using the specified technology/device

➤ **Understand Vivado HLS defaults**

– Key to understanding the initial design created by Vivado HLS

➤ **Understand the priority of directives**

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

XILINX ➤ ALL PROGRAMMABLE.

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 …**

XILINX ➤ ALL PROGRAMMABLE.

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

foo_top

A

C

B

D

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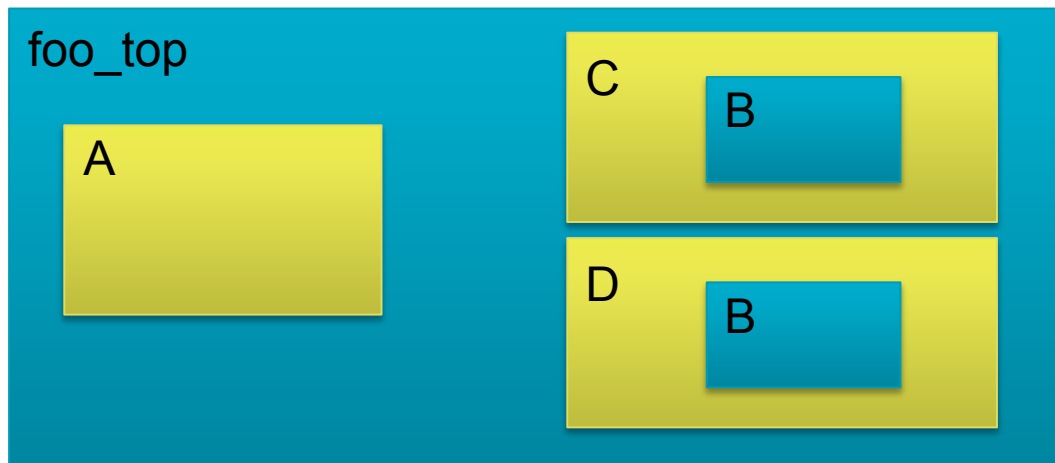
Each function/block can be shared like any other component (add, sub, etc) provided it's not in use at the same time

– By default, each function is implemented using a common instance

– Functions may be inlined to dissolve their hierarchy

• Small functions may be automatically inlined

© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE™

Types = Operator Bit-sizes

Code

Operations

Types

```
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;
}
```

RDx

RDc

>=

-

==

+

*

+

*

WRy
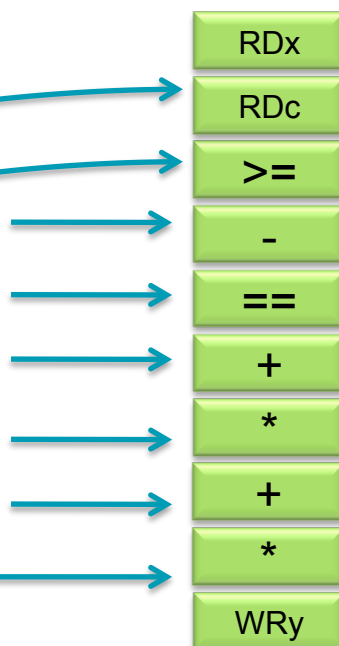
Standard C types

long long (64-bit)   short (16-bit)   unsigned types

int (32-bit)   char (8-bit)

float (32-bit)   double (64-bit)

Arbitary 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).

From any C code example ...

Operations are extracted…

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

© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.

Loops

➤ **By default, loops are rolled**

– Each C loop iteration ➜ Implemented in the same state

– Each C loop iteration ➜ Implemented with same resources

N

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

Synthesis ➜

foo_top

a[N] ➜ + ➜ ➜ b

Loops require labels if they are to be referenced by Tcl directives
(GUI will auto-add labels)
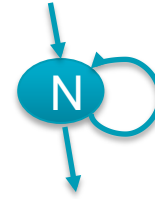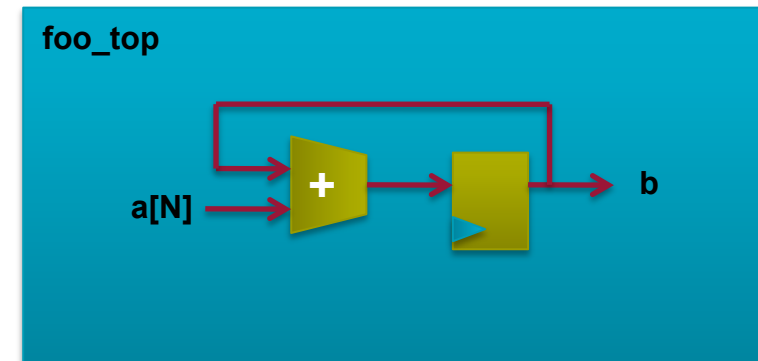
– Loops can be unrolled if their indices are statically determinable at elaboration time

• 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 ….

 XILINX ➤ ALL PROGRAMMABLE.

Data Dependencies: Good

```
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;
}
```

Default Schedule

==	*	>=	==	*	>=	==	*	>=	==	*	>=	WRy
-	-	+	-	-	+	-	-	+	-	RDx	+	
	RDc			RDc			RDc			RDc		

Iteration 1   Iteration 2   Iteration 3   Iteration 4

The read X operation has good mobility
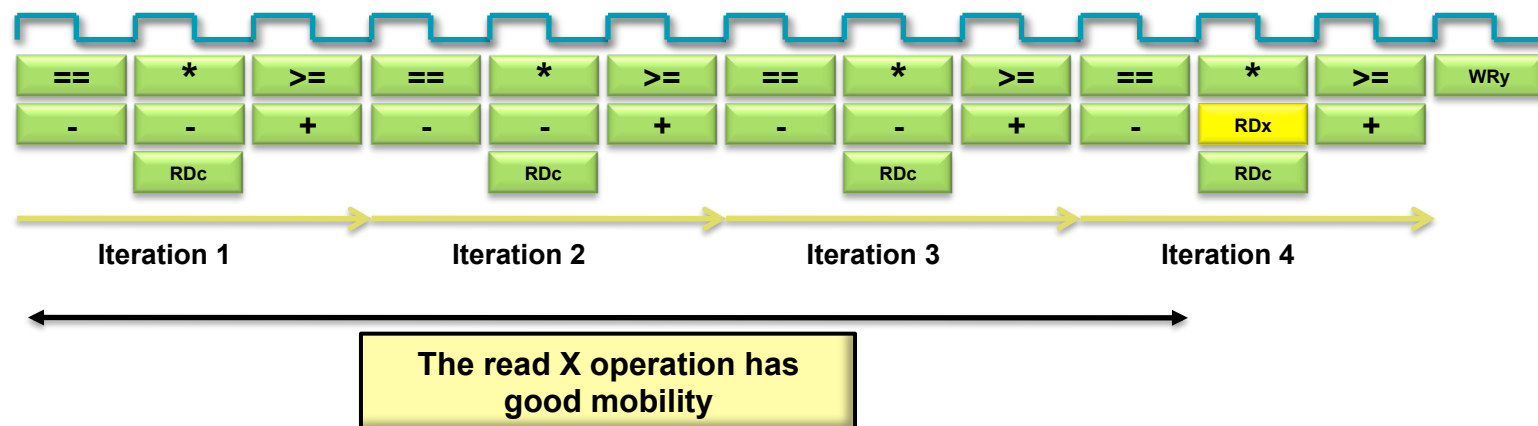
Example of good mobility

- The read on data port X can occur anywhere from the start to iteration 4
  - The only constraint on RDx is that it occur before the final multiplication
- Vivado HLS has a lot of freedom with this operation
  - It waits until the read is required, saving a register
  - There are no advantages to reading any earlier (unless you want it registered)
  - Input reads can be optionally registered
- The final multiplication is very constrained…

XILINX ➤ ALL PROGRAMMABLE.

Data Dependencies: Bad

```
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;
}
```
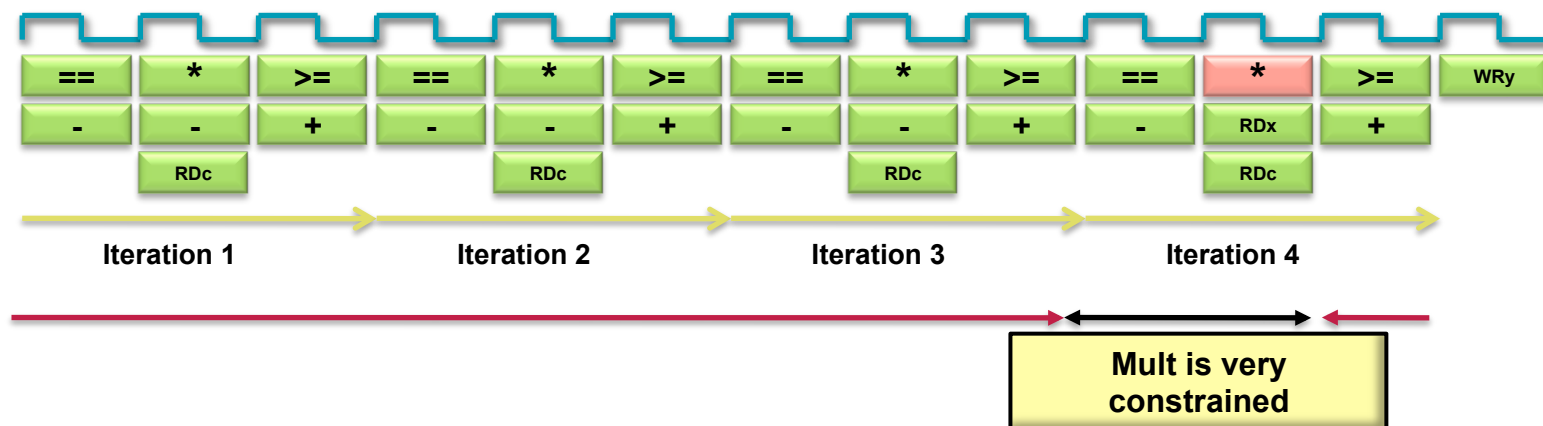
Default Schedule

Iteration 1   Iteration 2   Iteration 3   Iteration 4

Mult is very constrained

❯ Example of bad mobility

- – The final multiplication must occur before the read and final addition
  - • It could occur in the same cycle if timing allows
- – Loops are rolled by default
  - • Each iteration cannot start till the previous iteration completes
  - • The final multiplication (in iteration 4) must wait for earlier iterations to complete
- – The structure of the code is forcing a particular schedule
  - • There is little mobility for most operations
- – Optimizations allow loops to be unrolled giving greater freedom

© Copyright 2013 Xilinx

XILINX ❯ ALL PROGRAMMABLE.

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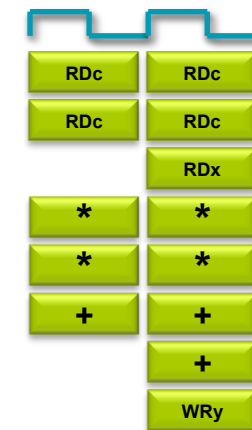
```
void fir (
…
acc=0;
 loop: for (i=3;i>=0;i--) {
   if (i==0) {
     acc+=x*c[0];
     shift_reg[0]=x;
   } else {
     shift_reg[i]=shift_reg[i-1];
     acc+=shift_reg[i]*c[i];
   }
 }
*y=acc;
}
```

RDc RDc
RDc RDc
      RDx
 *   *
 *   *
 +   +
      +
     WRy

XILINX ➤ ALL PROGRAMMABLE.

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

```
void foo_top(int x, ...)
{
  int A[N];
  L1: for (i = 0; i < N; i++)
      A[i+x] = A[i] + i;
}
```


> **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

```
void foo_top( int A[3*N] , int x)
{
  L1: for (i = 0; i < N; i++)
      A[i+x] = A[i] + i;
}
```

Synthesis →

foo_top

+

Number of ports defined by the RAM resource

DPRAMB

DIN0   DOUT0
ADDR0

CE0
WE0

DIN1   DOUT1
ADDR1

CE1
WE1
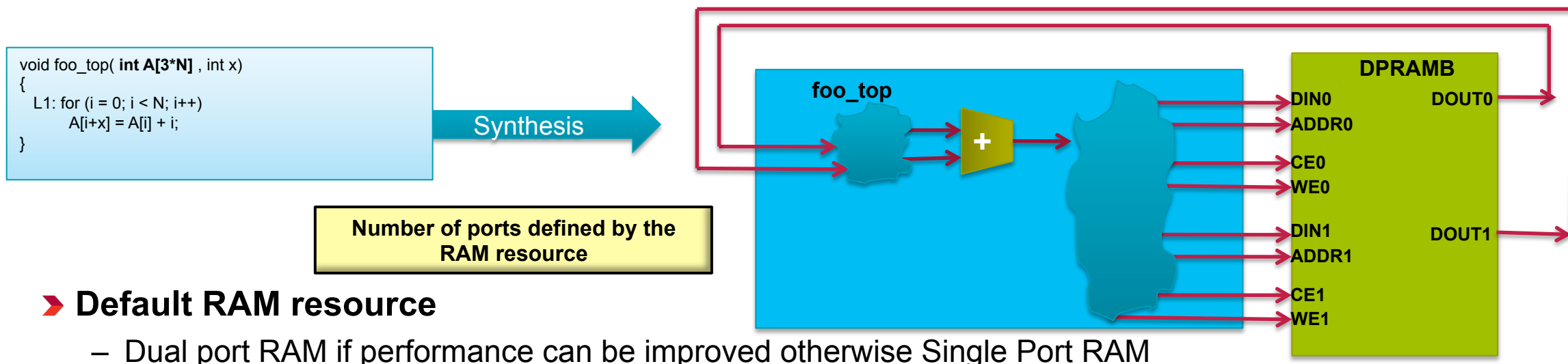
> **Default RAM resource**
  – Dual port RAM if performance can be improved otherwise Single Port RAM

XILINX ➤ ALL PROGRAMMABLE.

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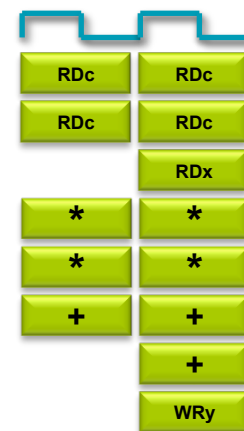
```
loop: for (i=3;i>=0;i--) {
    if (i==0) {
        acc+=x*c[0];
        shift_reg[0]=x;
    } else {
        shift_reg[i]=shift_reg[i-1];
        acc+=shift_reg[i]*c[i];
    }
}
*y=acc;
```

Operators

> **Operator sizes are defined by the type**

– The variable type defines the size of the operator

> **Vivado HLS will try to minimize the number of operators**

– By default Vivado HLS will seek to minimize area <u>after</u> constraints are satisfied

> **User can set specific limits & targets for the resources used**

– Allocation can be controlled

- An upper limit can be set on the number of operators or cores allocated for the design: This can be used to force sharing
- e.g limit the number of multipliers to 1 will force Vivado HLS to share

Use 1 mult, but take 4 cycle even if it could be done in 1 cycle using 4 mults

3   2   1   0

– Resources can be specified

- The cores used to implement each operator can be specified
- e.g. Implement each multiplier using a 2 stage pipelined core (hardware)
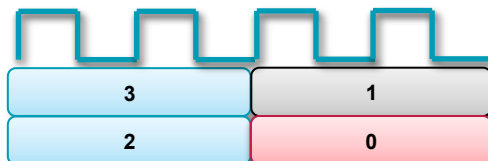
Same 4 mult operations could be done with 2 pipelined mults (with allocation limiting the mults to 2)

3   1
2   0

EX XILINX > ALL PROGRAMMABLE.

Vivado HLS Terminology for measuring in Clock Cycles

Latency	The number of cycles from input to output (final output of an array write)	14 cycles
Throughput	The number of cycle between new input samples (in this example it must wait for all operations to complete before it can read a new input)	14 cycles
Initiation Interval (II)	The number of cycles between new inputs to a pipeline (the same as throughput, but this term is used with pipelines).	Not shown in this example.
Data Rate	The 1/throughout * clock frequency	10ns clock => 7.14 Mhz, ((1/10e9)*14)
Trip count	The number of iterations in a loop	4
Loop Latency	The latency of the entire loop (divide by tripcount to get the latency for each loop iteration)	12 cycles

You may have your own terminology: this is Vivado HLS's

XILINX ➤ ALL PROGRAMMABLE.

Outline

> **Introduction to High-Level Synthesis**

> **High-Level Synthesis with Vivado HLS**

> ***Language Support***

> **Validation Flow**

> **Summary**

© Copyright 2013 Xilinx

 XILINX ➤ ALL PROGRAMMABLE.

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++ and SystemC**
>   – 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

XILINX ➤ ALL PROGRAMMABLE.

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' 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

 XILINX ➤ ALL PROGRAMMABLE.

Outline

➤ **Introduction to High-Level Synthesis**

➤ **High-Level Synthesis with Vivado HLS**

➤ **Language Support**

➤ *Validation Flow*

➤ **Summary**

© Copyright 2013 Xilinx

 XILINX ➤ ALL PROGRAMMABLE.

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

XILINX ➤ ALL PROGRAMMABLE.
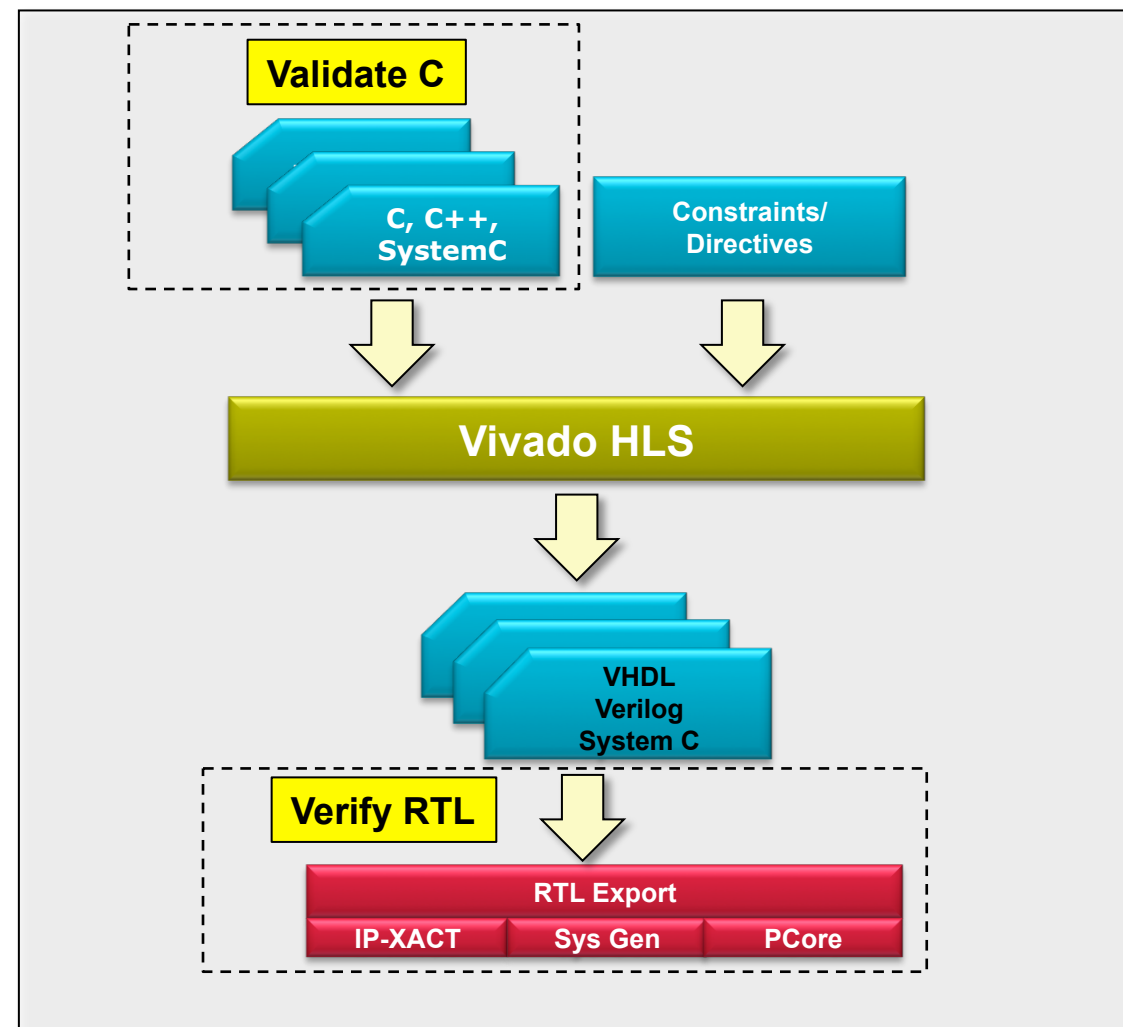
C Function Test Bench

➤ **The test bench is the level above the function**

– The main() function is <u>above</u> 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)

```c
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;
}
```

XILINX ➤ ALL PROGRAMMABLE.

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

XILINX ➤ ALL PROGRAMMABLE.

Outline

➤ **Introduction to High-Level Synthesis**

➤ **High-Level Synthesis with Vivado HLS**

➤ **Language Support**

➤ **Validation Flow**

➤ *Summary*

XILINX ➤ ALL PROGRAMMABLE™

Summary

➤ **In HLS**
- C becomes RTL
- Operations in the code map to hardware resources
- Understand how constructs such as functions, loops and arrays are synthesized

➤ **HLS design involves**
- Synthesize the initial design
- Analyze to see what limits the performance
  - User directives to change the default behaviors
  - Remove bottlenecks
- Analyze to see what limits the area
  - The types used define the size of operators
  - This can have an impact on what operations can fit in a clock cycle

XILINX ➤ ALL PROGRAMMABLE.

Summary

- **Use directives to shape the initial design to meet performance**
  - Increase parallelism to improve performance
  - Refine bit sizes and sharing to reduce area
- **Vivado HLS benefits**
  - Productivity
  - Portability
  - Permutability

 XILINX ➤ ALL PROGRAMMABLE.