Introduction to High-Level Synthesis with Vivado HLS

Vivado HLS 2013.3 Version

Objectives

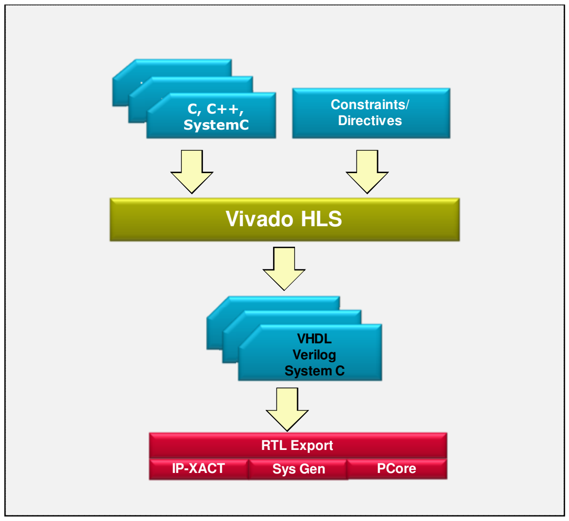
- **After completing this module, you will be able to:**
  - Describe the high level synthesis flow
  - Understand the control and datapath extraction
  - Describe scheduling and binding phases of the HLS flow
  - List the priorities of directives set by Vivado HLS
  - List comprehensive language support in Vivado HLS
  - Identify steps involved in validation and verification flows

Outline

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

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



Design Exploration with Directives

One body of code:
Many hardware outcomes

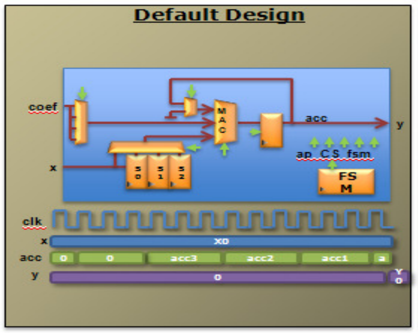
```
…
loop: for (i=3;i>=0;i--) {
   if (i==0) {
     acc+=x*c[0];
     shift_reg[0]=x;
   } else {
      shift_reg[i]=shift_reg[i-1];
      acc+=shift_reg[i]*c[i];
   }
}
….
```

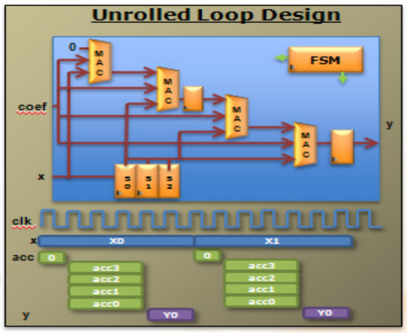
Before we get into details, let's look under the hood ….

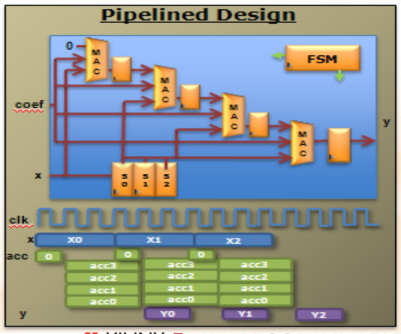
The same hardware is used for each iteration of the loop:
•Small area
•Long latency
•Low throughput

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

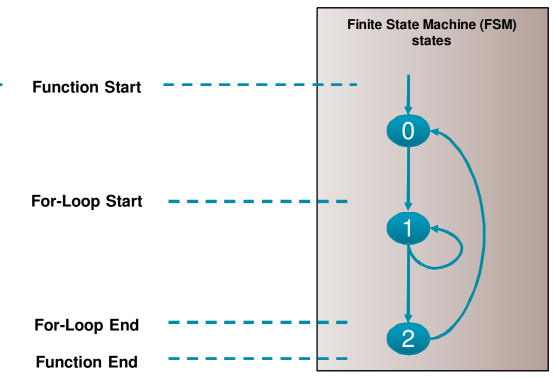
- **How is hardware extracted from C code?**
  - Control and datapath can be extracted from C code at the top level
  - The same principles used in the example can be applied to sub-functions
    - At some point in the top-level control flow, control is passed to a sub-function
    - Sub-function may 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?

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;
}
```

Control Behavior



From any C code example ..

The loops in the C code correlated to states of behavior

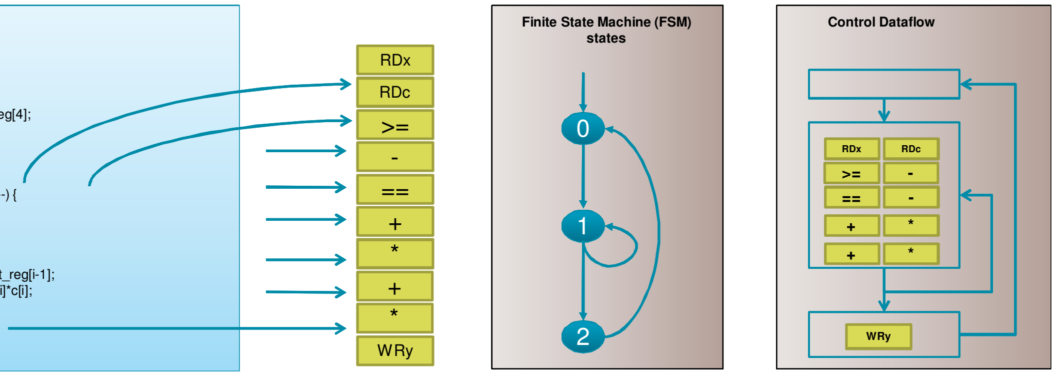
This behavior is extracted into a hardware state machine

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 **Control Behavior** **Control & Datapath Behavior**



From any C code example ..

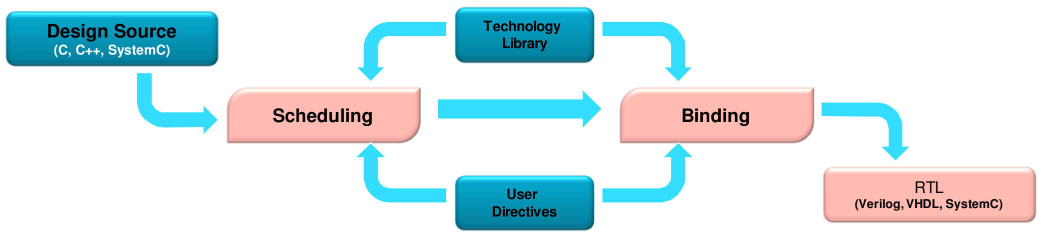
Operations are extracted…

The control is known

A unified control dataflow behavior is created.

High-Level Synthesis: Scheduling & Binding

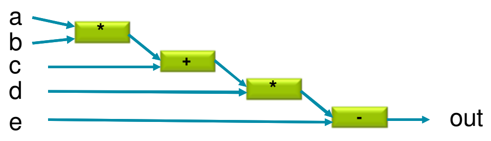
- **Scheduling & Binding**
  - Scheduling and Binding are at the heart of HLS
- **Scheduling determines in which clock cycle an operation will occur**
  - Takes into account the control, dataflow and user directives
  - The allocation of resources can be constrained
- **Binding determines which library cell is used for each operation**
  - Takes into account component delays, user directives

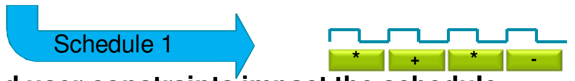


Scheduling

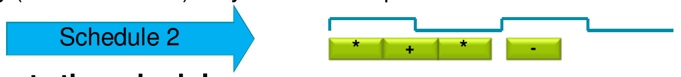
- **The operations in the control flow graph are mapped into clock cycles**

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





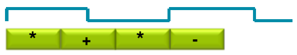
- **The technology and user constraints impact the schedule**
  - A faster technology (or slower clock) may allow more operations to occur in the same clock cycle



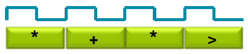
- **The code also impacts the schedule**
  - Code implications and data dependencies must be obeyed

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

Outline

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

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 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

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

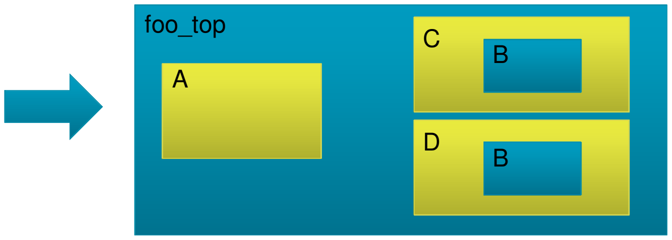
- **Each function is translated into an RTL block**
  - Verilog module, VHDL entity

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

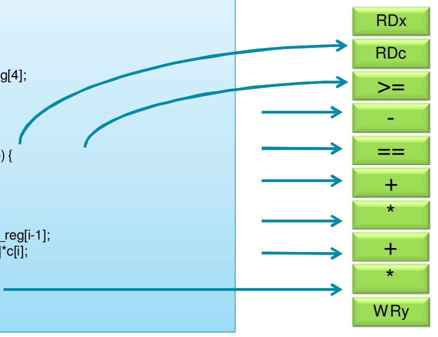
  - By default, each function is implemented using a common instance
  - Functions may be inlined to dissolve their hierarchy
    - Small functions may be automatically inlined

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;
}
```

Operations



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)	

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).

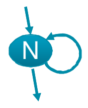
From any C code example ...

Operations are extracted…

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

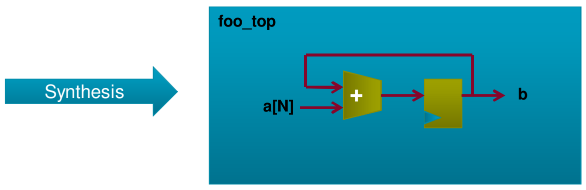
Loops

- **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;
  ...
  }
```

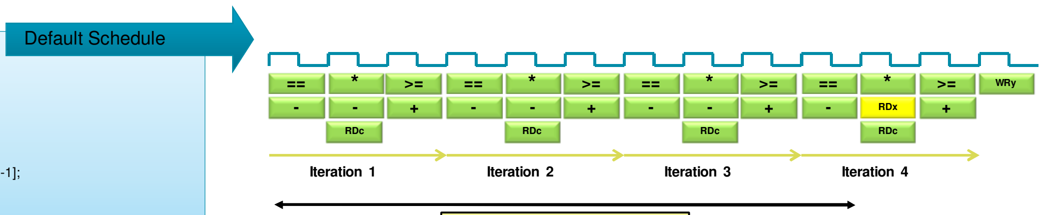
Loops require labels if they are to be referenced by Tcl directives (GUI will auto-add labels)



  - 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 ….

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;
}
```

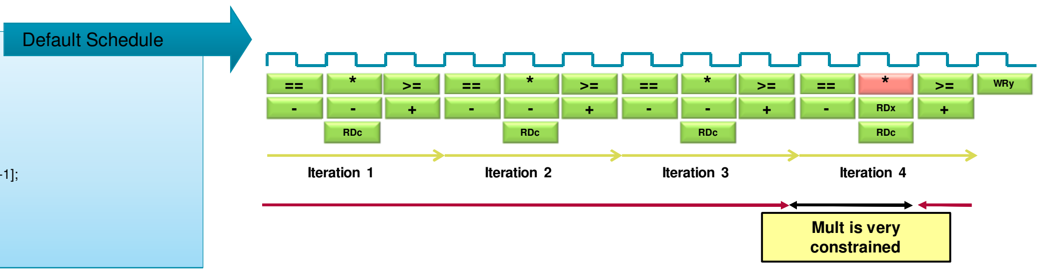


The read X operation has good mobility

- **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…

Data Dependencies: Bad

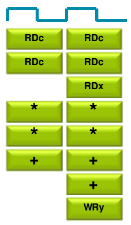
```
void fir (
 …
acc=0;
 loop: for (i=3;i>=0;i--) {
    if (i==0) {
      acc+=x*c[0];
      shift_reg[0]=x;
    } else {
       shift_reg[i]=shift_reg[i-1];
       acc+=shift_reg[i]*c[i];
    }
 }
 *y=acc;
}
```



- **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

Schedule after Loop Optimization

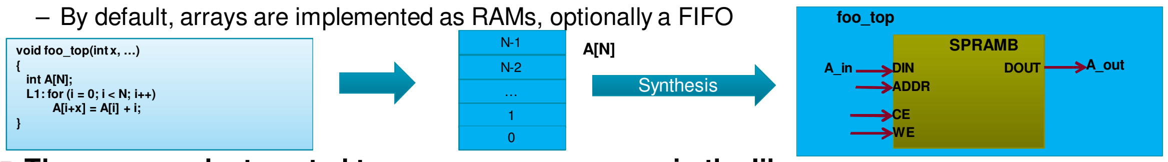
- **With the loop unrolled (completely)**
  - The dependency on loop iterations is gone
  - Operations can now occur in parallel
    - If data dependencies allow
    - If operator timing allows
  - Design finished faster but uses more operators
    - 2 multipliers & 2 Adders
- **Schedule Summary**
  - 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

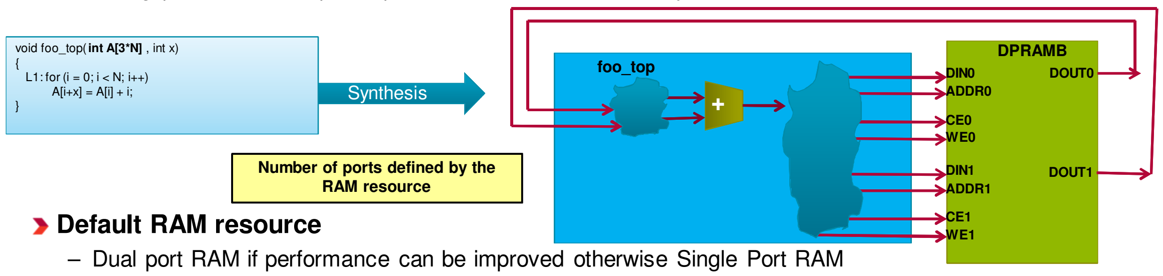


```
void foo_top(int x, …)
{
  int A[N];
  L1: for (i = 0; i < N; i++)
        A[i+x] = A[i] + i;
}
```

- **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;
}
```

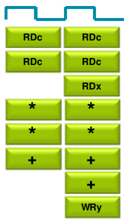
- **Default RAM resource**
  - Dual port RAM if performance can be improved otherwise Single Port RAM

Schedule after an Array Optimization

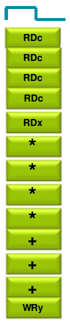
- **With the existing code & defaults**
  - Port C is a dual port RAM
  - Allows 2 reads per clock cycles
    - IO behavior impacts performance

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

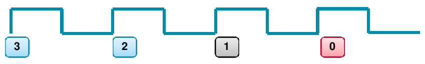


```
loop: for (i=3;i>=0;i--) {
   if (i==0) {
     acc+=x*c[0];
     shift_reg[0]=x;
   } else {
     shift_reg[i]=shift_reg[i-1];
     acc+=shift_reg[i]*c[i];
   }
}
*y=acc;
```



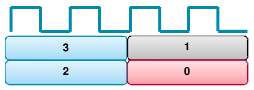
Operators

- **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

  - 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)

Outline

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

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

C, C++ and SystemC Support

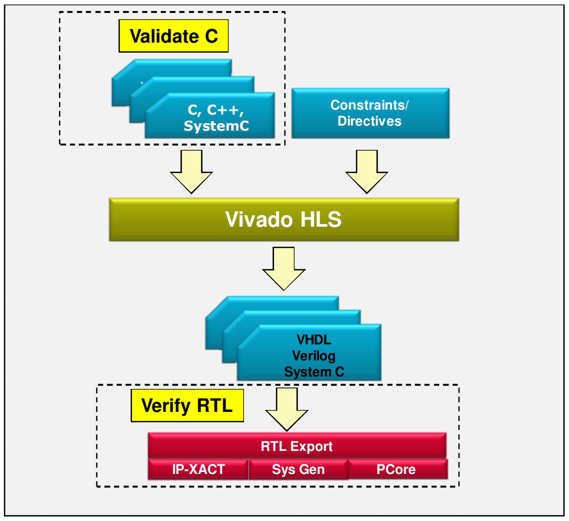
- **The vast majority of C, C++ and SystemC is supported**
  - Provided it is statically defined at **compile time**
  - If it’s not defined until **run time**, it won’ be synthesizable
- **Any of the three variants of C can be used**
  - If C is used, Vivado HLS expects the file extensions to be .c
  - For C++ and SystemC it expects file extensions .cpp

Outline

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

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

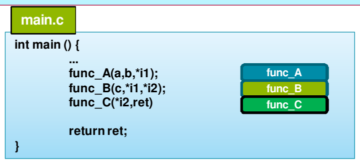
- **The test bench is the level above the function**
  - The main() function is above the function to be synthesized
- **Good Practices**
  - The test bench should compare the results with golden data
    - Automatically confirms any changes to the C are validated and verifies the RTL is correct
  - 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

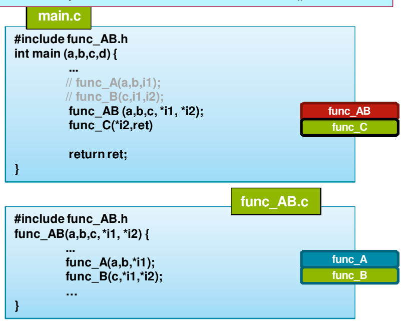
- **Determine the top-level function for synthesis**
- **If there are Multiple functions, they must be merged**
  - There can only be 1 top-level function for synthesis

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



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

        return ret;
}
```

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



```
#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;
}
```

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

Recommendation is to separate test bench and design files

Outline

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

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
- **Use directives to shape the initial design to meet performance**
  - Increase parallelism to improve performance
  - Refine bit sizes and sharing to reduce area

Using Vivado HLS

Vivado HLS 2013.3 Version

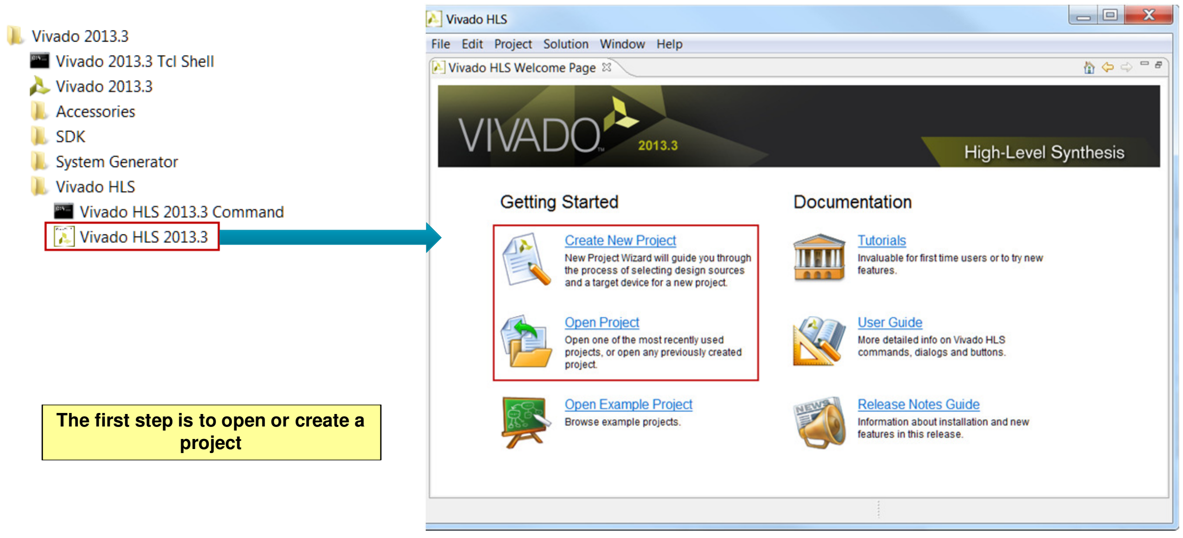
Objectives

- **After completing this module, you will be able to:**
  - List various OS under which Vivado HLS is supported
  - Describe how projects are created and maintained in Vivado HLS
  - State various steps involved in using Vivado HLS project creation wizard
  - Distinguish between the role of top-level module in testbench and design to be synthesized
  - List various verifications which can be done in Vivado HLS
  - List Vivado HLS project directory structure

Outline

- ***Invoking Vivado HLS***
- **Project Creation using Vivado HLS**
- **Synthesis to IPXACT Flow**
- **Design Analysis**
- **Other Ways to use Vivado HLS**
- **Summary**

Invoke Vivado HLS from Windows Menu



Create New Project: New Project Wizard will guide you through the process of selecting design sources and a target device for a new project.

Open Project: Open one of the most recently used projects, or open any previously created project.

Open Example Project: Browse example projects.

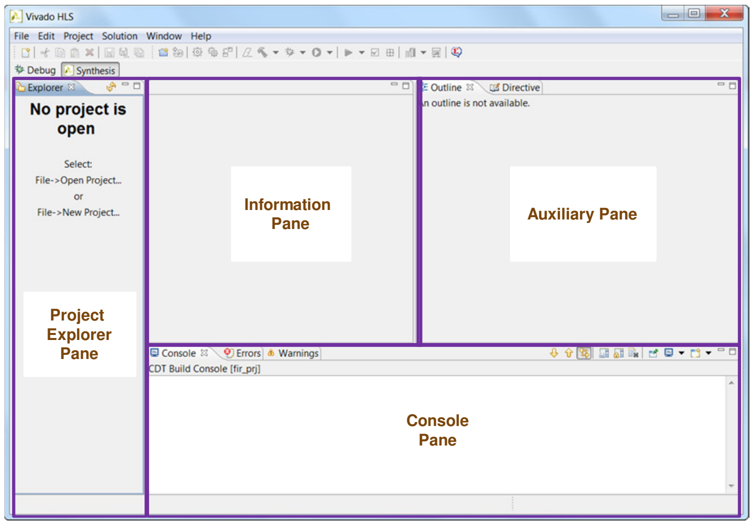
Tutorials: Invaluable for first time users or to try new features.

User Guide: More detailed info on Vivado HLS commands, dialogs and buttons.

Release Notes Guide: Information about installation and new features in this release.

The first step is to open or create a project

Vivado HLS GUI



XILINX ALL PROGRAMMABLE.

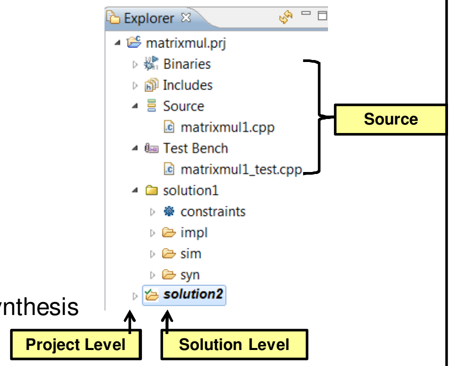
Outline

- Invoking Vivado HLS
- ***Project Creation using Vivado HLS***
- Synthesis to IPXACT Flow
- Design Analysis
- Other Ways to use Vivado HLS
- Summary

XILINX ALL PROGRAMMABLE.

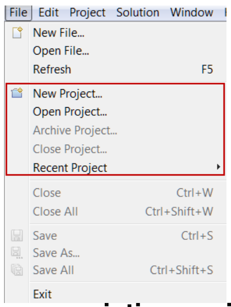
Vivado HLS Projects and Solutions

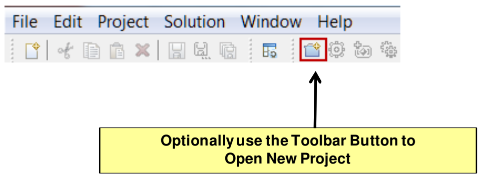
- **Vivado HLS is project based**
  - A project specifies the source code which will be synthesized
  - Each project is based on one set of source code
  - Each project has a user specified name
- **A project can contain multiple solutions**
  - Solutions are different implementations of the same code
  - Auto-named solution1, solution2, etc.
  - Supports user specified names
  - Solutions can have different clock frequencies, target technologies, synthesis directives
- **Projects and solutions are stored in a hierarchical directory structure**
  - Top-level is the project directory
  - The disk directory structure is identical to the structure shown in the GUI project explorer (except for source code location)



Vivado HLS Step 1: Create or Open a project

- **Start a new project**
  - The GUI will start the project wizard to guide you through all the steps

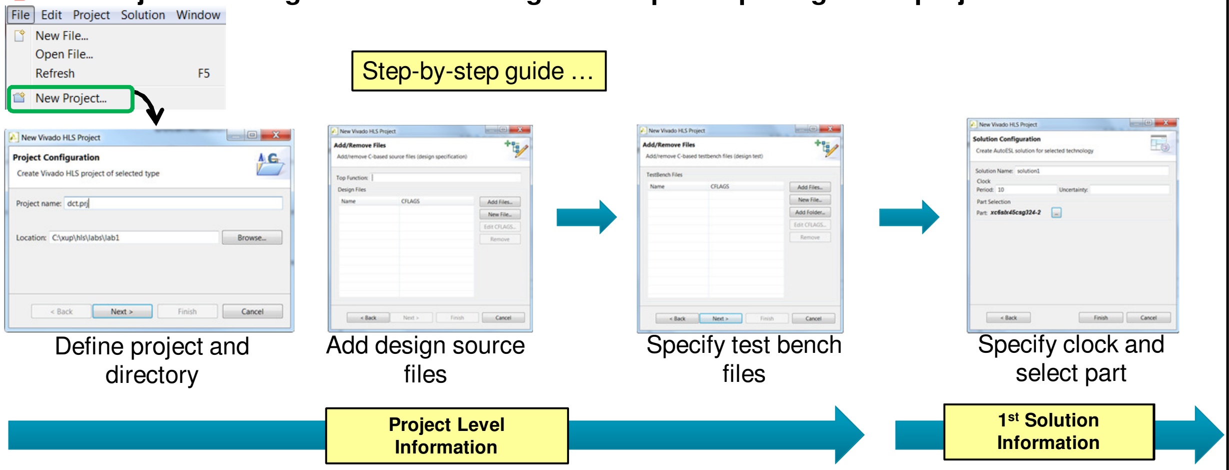



File Edit Project Solution Window Help
Optionally use the Toolbar Button to Open New Project

- **Open an existing project**
  - All results, reports and directives are automatically saved/remembered
  - Use “Recent Project” menu for quick access

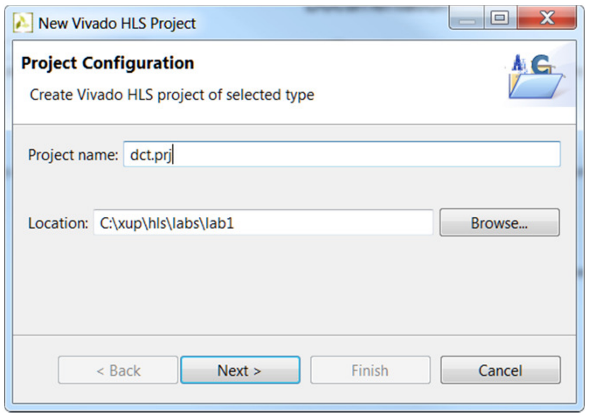
Project Wizard

- **The Project Wizard guides users through the steps of opening a new project**



Define Project & Directory

- **Define the project name**
  - Note, here the project is given the extension .prj
  - A useful way of seeing it's a project (and not just another directory) when browsing
- **Browse to the location of the project**
  - In this example, project directory "dct.prj" will be created inside directory "lab1"



New Vivado HLS Project

Project Configuration

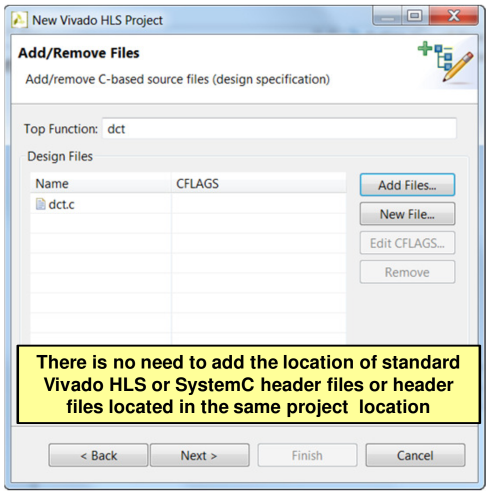
Create Vivado HLS project of selected type

Project name: dct.prj

Location: C:\xup\hls\labs\lab1

Add Design Source Files

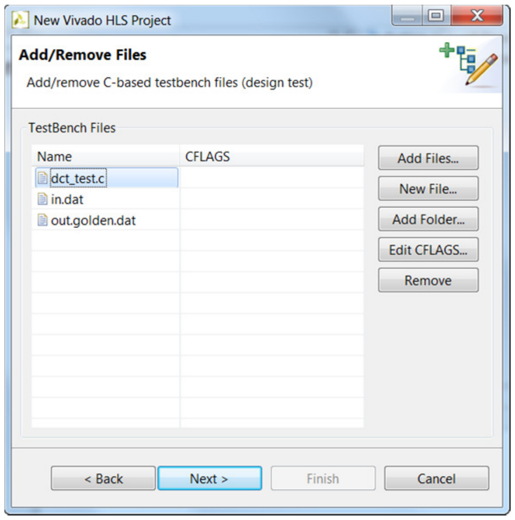
- **Add Design Source Files**
  - This allows Vivado HLS to determine the top-level design for synthesis, from the test bench & associated files
  - Not required for SystemC designs
- **Add Files…**
  - Select the source code file(s)
  - The CTRL and SHIFT keys can be used to add multiple files
  - No need to include headers (.h) if they reside in the same directory
- **Select File and Edit CFLAGS…**
  - If required, specify C compile arguments using the "Edit CFLAGS…"
    - Define macros: -DVERSION1
    - Location of any (header) files not in the same directory as the source: -I../include



There is no need to add the location of standard Vivado HLS or SystemC header files or header files located in the same project location

Specify Test Bench Files

- **Use "Add Files" to include the test bench**
  - Vivado HLS will re-use these to verify the RTL using co-simulation
- **<u>And all files</u> referenced by the test bench**
  - The RTL simulation will be executed in a different directory (Ensures the original results are not over-written)
  - Vivado HLS needs to also copy any files accessed by the test bench
  - Input data and output results (*.dat) are shown in this example
- **Add Folders**
  - If the test bench uses relative paths like "sub_directory/my_file.dat" you can add "sub_directory" as a folder/directory
- **Use "Edit CFLAGS…"**
  - To add any C compile flags required for compilation



Test benches I

- **The test bench should be in a separate file**
- **Or excluded from synthesis**
  - The Macro __SYNTHESIS__ can be used to isolate code <u>which will not be synthesized</u>
    - This macro is defined when Vivado HLS parses any code (-D__SYNTHESIS__)

```
// test.c
#include <stdio.h>
void test (int d[10]) {
  int acc = 0;
  int i;
  for (i=0;i<10;i++) {
    acc += d[i];
    d[i] = acc;
  }
}
#ifndef __SYNTHESIS__
int main () {
  int d[10], i;
  for (i=0;i<10;i++) {
    d[i]   = i;
  }
  test(d);
  for (i=0;i<10;i++) {
    printf("%d %d\n", i, d[i]);
  }
  return 0;
}
#endif
```

Design to be synthesized

<u>Test Bench</u>
Nothing in this ifndef will be read by Vivado HLS (will be read by gcc)

Test benches II

- **Ideal test bench**
  - Should be self checking
    - RTL verification will re-use the C test bench
  - If the test bench is self-checking
    - Allows RTL Verification to be run without a requirement to check the results again
  - RTL verification "passes" if the test bench return value is 0 (zero)
    - Actively return a 0 if the simulation passes

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

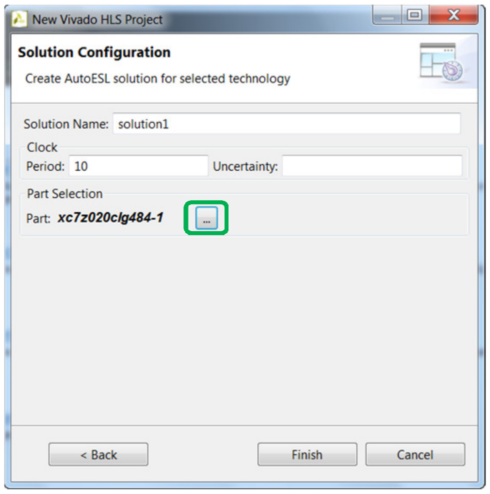
The –w option ensures the "newline" does not cause a difference between Windows and Linux files

  - Non-synthesizable constructs may be added to a synthesize function if __SYNTHESIS__ is used

```
#ifndef __SYNTHESIS__
  image_t *yuv = (image_t *)malloc(sizeof(image_t));
#else // Workaround malloc() calls w/o changing rest of code
  image_t _yuv;
#endif
```

Solution Configuration

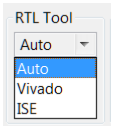
- **Provide a solution name**
  - Default is solution1, then solution2 etc.
- **Specify the clock**
  - The clock uncertainty is subtracted from the clock to provide an “effective clock period”
  - Vivado HLS uses the “effective clock period” for Synthesis
  - Provides users defined margin for downstream RTL synthesis, P&R
- **Select the part**
  - Select a device family after applying filters such as family, package and speed grade (see next slide)

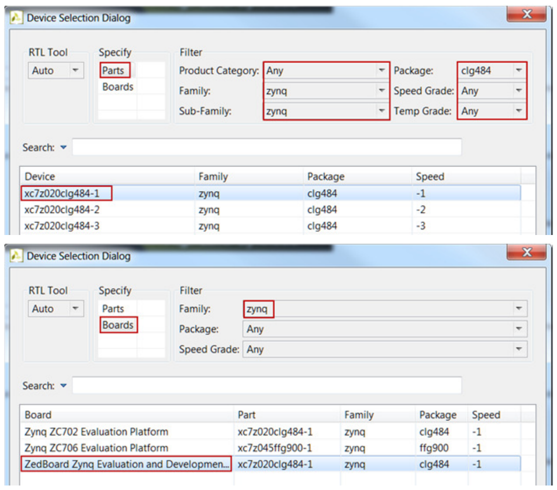


Solution Name: solution1
Clock
Period: 10 Uncertainty:
Part Selection
Part: xc7z020clg484-1

Selecting Part and Implementation Engine

- **Select the target part either through Parts or Boards specify**
- **Select RTL Tools**
  - Auto
    - Will select Vivado for 7 Series and Zynq devices
    - Will select ISE for Virtex-6 and earlier families
  - Vivado
  - ISE
    - ISE Design Suite must be installed and must be included in the PATH variable





RTL Tool: Auto
Specify: Parts
Product Category: Any
Family: zynq
Sub-Family: zynq
Package: clg484
Speed Grade: Any
Temp Grade: Any

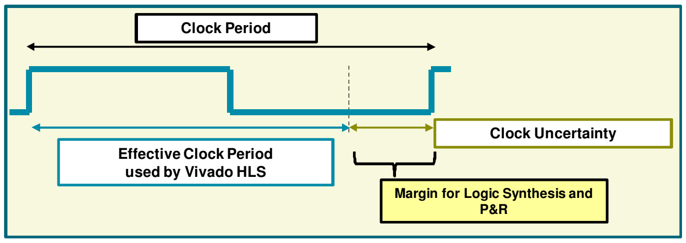
Device	Family	Package	Speed
xc7z020clg484-1	zynq	clg484	-1
xc7z020clg484-2	zynq	clg484	-2
xc7z020clg484-3	zynq	clg484	-3

RTL Tool: Auto
Specify: Boards
Family: zynq
Package: Any
Speed Grade: Any

Board	Part	Family	Package	Speed
Zynq ZC702 Evaluation Platform	xc7z020clg484-1	zynq	clg484	-1
Zynq ZC706 Evaluation Platform	xc7z045ffg900-1	zynq	ffg900	-1
ZedBoard Zynq Evaluation and Developmen...	xc7z020clg484-1	zynq	clg484	-1

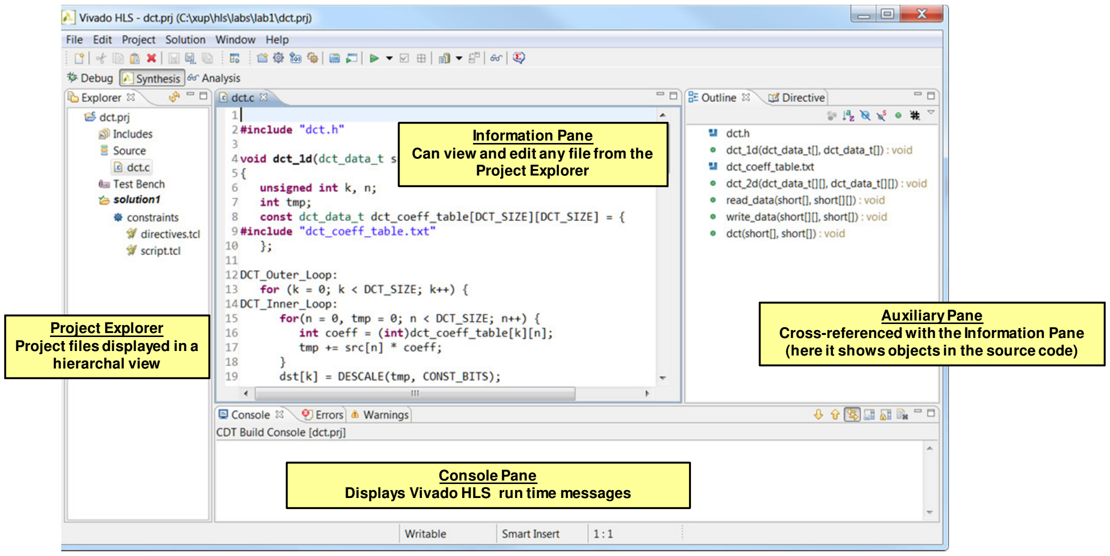
Clock Specification

- **Clock frequency must be specified**
  - Only 1 clock can be specified for C/C++ functions
  - SystemC can define multiple clocks
- **Clock uncertainty can be specified**
  - Subtracted from the clock period to give an effective clock period
  - The effective clock period is used for synthesis
    - Should not be used as a design parameter
    - Do not vary for different results: this is your safety margin
  - A user controllable margin to account for downstream RTL synthesis and P&R


Clock Period
Clock Uncertainty
Effective Clock Period used by Vivado HLS
Margin for Logic Synthesis and P&R

XILINX ALL PROGRAMMABLE.

A Vivado HLS Project


Vivado HLS - dct.prj (C:\xup\hls\labs\lab1\dct.prj)
File Edit Project Solution Window Help
Debug
Synthesis
Analysis
Explorer
dct.prj
Includes
Source
dct.c
Test Bench
solution1
constraints
directives.tcl
script.tcl
dct.c
Outline
Directive
dct.h
dct_1d(dct_data_t[], dct_data_t[]) : void
dct_coeff_table.txt
dct_2d(dct_data_t[][], dct_data_t[][]) : void
read_data(short[], short[][]) : void
write_data(short[][], short[]) : void
dct(short[], short[]) : void
Console
Errors
Warnings
CDT Build Console [dct.prj]
Writable
Smart Insert
1 : 1

```
#include "dct.h"

void dct_1d(dct_data_t s
{
   unsigned int k, n;
   int tmp;
   const dct_data_t dct_coeff_table[DCT_SIZE][DCT_SIZE] = {
#include "dct_coeff_table.txt"
   };

DCT_Outer_Loop:
   for (k = 0; k < DCT_SIZE; k++) {
DCT_Inner_Loop:
      for(n = 0, tmp = 0; n < DCT_SIZE; n++) {
         int coeff = (int)dct_coeff_table[k][n];
         tmp += src[n] * coeff;
      }
      dst[k] = DESCALE(tmp, CONST_BITS);
```

Information Pane
Can view and edit any file from the Project Explorer

Project Explorer
Project files displayed in a hierarchal view

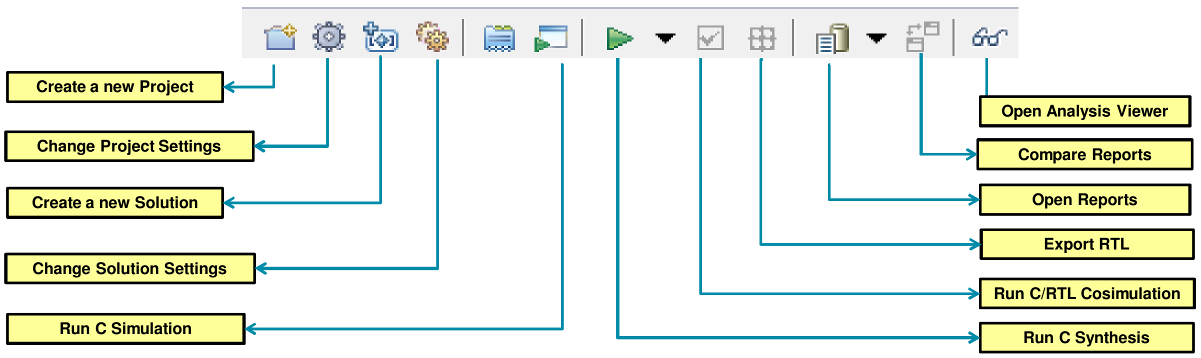
Auxiliary Pane
Cross-referenced with the Information Pane (here it shows objects in the source code)

Console Pane
Displays Vivado HLS run time messages

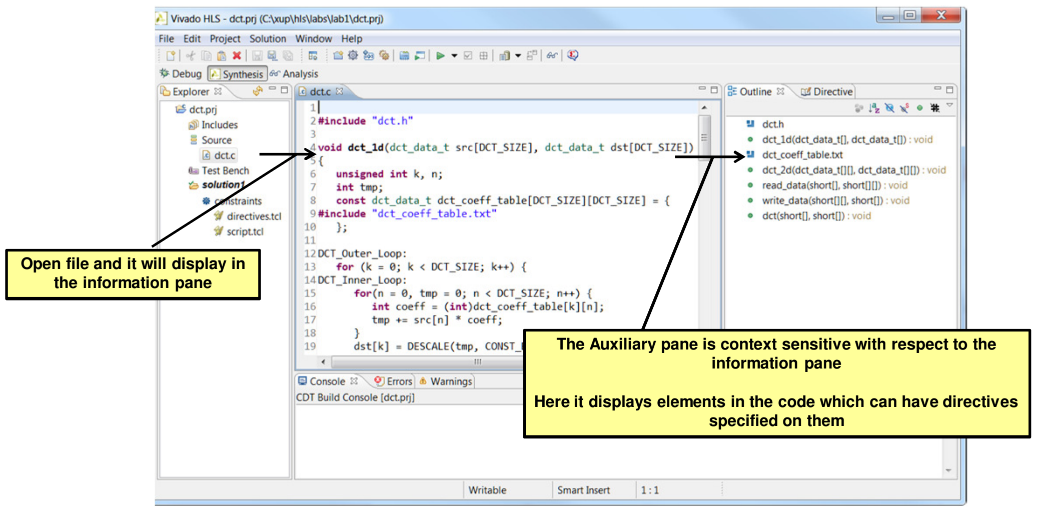
XILINX ALL PROGRAMMABLE.

Vivado HLS GUI Toolbar

- **The primary commands have toolbar buttons**
  - Easy access for standard tasks
  - Button highlights when the option is available
    - E.g. cannot perform C/RTL simulation before synthesis



Files: Views, Edits & Information



```

#include "dct.h"

void dct_1d(dct_data_t src[DCT_SIZE], dct_data_t dst[DCT_SIZE])
{
    unsigned int k, n;
    int tmp;
    const dct_data_t dct_coeff_table[DCT_SIZE][DCT_SIZE] = {
#include "dct_coeff_table.txt"
    };

DCT_Outer_Loop:
    for (k = 0; k < DCT_SIZE; k++) {
DCT_Inner_Loop:
        for(n = 0, tmp = 0; n < DCT_SIZE; n++) {
            int coeff = (int)dct_coeff_table[k][n];
            tmp += src[n] * coeff;
        }
        dst[k] = DESCALE(tmp, CONST_
```

```
dct.h
dct_1d(dct_data_t[], dct_data_t[]) : void
dct_coeff_table.txt
dct_2d(dct_data_t[][], dct_data_t[][]) : void
read_data(short[], short[][]) : void
write_data(short[][], short[]) : void
dct(short[], short[]) : void
```

```
CDT Build Console [dct.prj]
```

Open file and it will display in the information pane

The Auxiliary pane is context sensitive with respect to the information pane

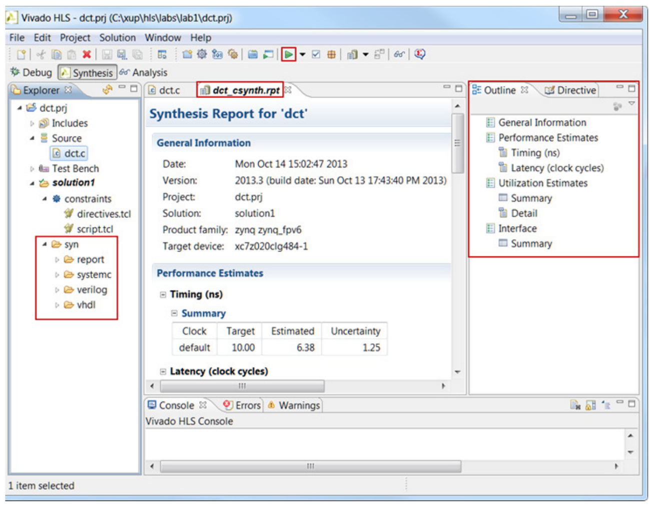
Here it displays elements in the code which can have directives specified on them

Outline

- Invoking Vivado HLS
- Project Creation using Vivado HLS
- *Synthesis to IPXACT Flow*
- Design Analysis
- Other Ways to use Vivado HLS
- Summary

Synthesis

- **Run C Synthesis**
- **Console**
  - Will show run time information
  - Examine for failed constraints
- **A "syn" directory is created**
  - Verilog, VHDL & SystemC RTL
  - Synthesis reports for all non-inlined functions
- **Report opens automatically**
  - When synthesis completes
- **Report is outlined in the Auxiliary pane**



Synthesis Report for 'dct'

General Information

Date:	Mon Oct 14 15:02:47 2013
Version:	2013.3 (build date: Sun Oct 13 17:43:40 PM 2013)
Project:	dct.prj
Solution:	solution1
Product family:	zynq zynq_fpv6
Target device:	xc7z020clg484-1

Performance Estimates

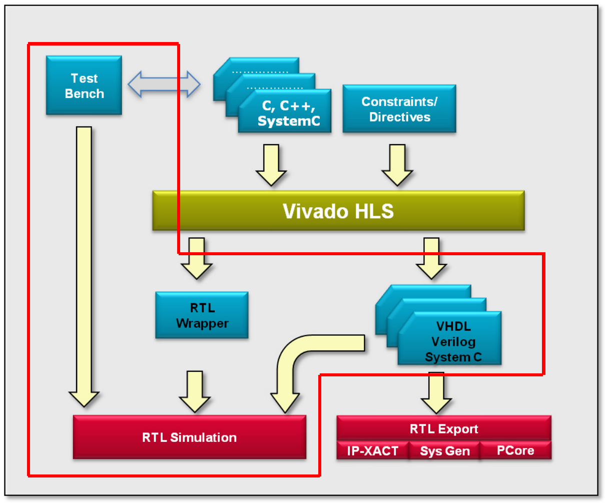
Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	6.38	1.25

Latency (clock cycles)

Vivado HLS : RTL Verification



RTL output in Verilog, VHDL and SystemC

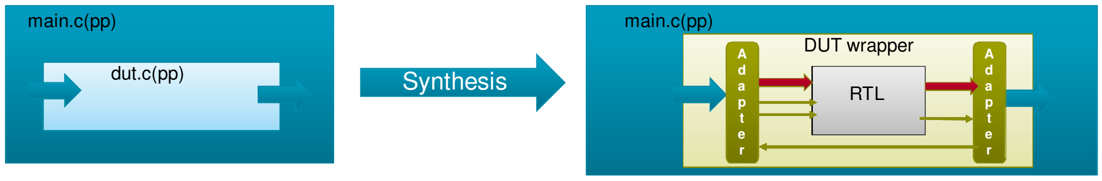
Automatic re-use of the C-level test bench

RTL verification can be executed from within Vivado HLS

Support for Xilinx simulators (XSim and ISim) and 3rd party HDL simulators in automated flow

RTL Verification: Under-the-Hood

- **RTL Co-Simulation**
  - Vivado HLS provides RTL verification
  - Creates the wrappers and adapters to re-use the C test bench



- Prior to synthesis
  - Test bench
  - Top-level C function
- After synthesis
  - Test bench
  - SystemC wrapper created by Vivado HLS
  - SystemC adapters created by Vivado HLS
  - RTL output from Vivado HLS
    - SystemC, Verilog or VHDL

There is no HDL test bench created

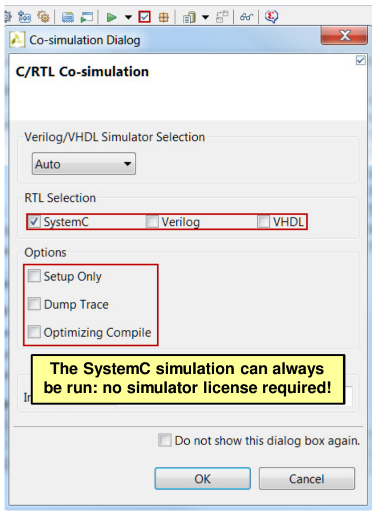
RTL Verification Support

- **Vivado HLS RTL Output**
  - Vivado HLS outputs RTL in SystemC, Verilog and VHDL
    - The SystemC output is at the RT Level
    - The input is not transformed to SystemC at the ESL
- **RTL Verification with SystemC**
  - The SystemC RTL output can be used to verify the design without the need for a HDL simulator and license
- **HDL Simulation Support**
  - Vivado HLS supports HDL simulators on both Windows & Linux
  - The 3rd party simulator executable must be in OS search path

Simulator	Linux	Windows
XSim (Vivado Simulator)	Supported	Supported
ISim (ISE Simulator)	Supported	Supported
Mentor Graphics ModelSim	Supported	Supported
Synopsys VCS	Supported	Not Available
NCSim	Supported	Not Available
Riviera	Supported	Supported

C/RTL Co-simulation

- **Start Simulation**
  - Opens the dialog box
- **Select the RTL**
  - SystemC does not require a 3rd party license
  - Verilog and VHDL require the appropriate simulator
    - Select the desired simulator
  - Run any or all
- **Options**
  - Can output trace file (VCD format)
  - Optimize the C compilation & specify test bench linker flags
  - The "setup only" option will not execute the simulation
- **OK will run the simulator**
  - Output files will be created in a "sim" directory



[✓] SystemC [ ] Verilog [ ] VHDL

Options

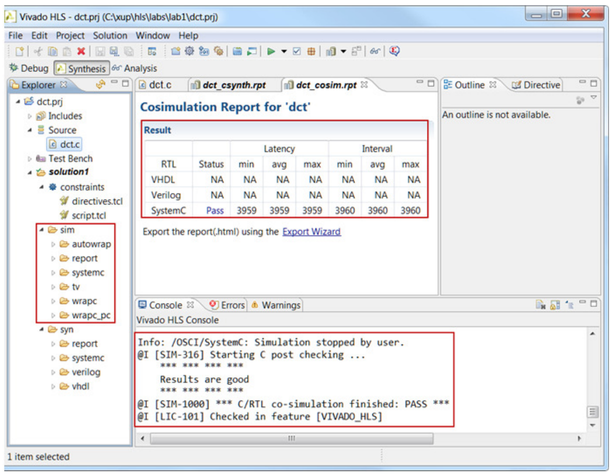
[ ] Setup Only
[ ] Dump Trace
[ ] Optimizing Compile

The SystemC simulation can always be run: no simulator license required!

[ ] Do not show this dialog box again.

Simulation Results

- **Simulation output is shown in the console**
- **Expect the same test bench response**
  - If the C test bench plots, it will with the RTL design (but slower)
- **Sim Directory**
  - Will contain a sub-directory for each RTL which is verified
- **Report**
  - A report is created and opened automatically



Cosimulation Report for 'dct'

Result

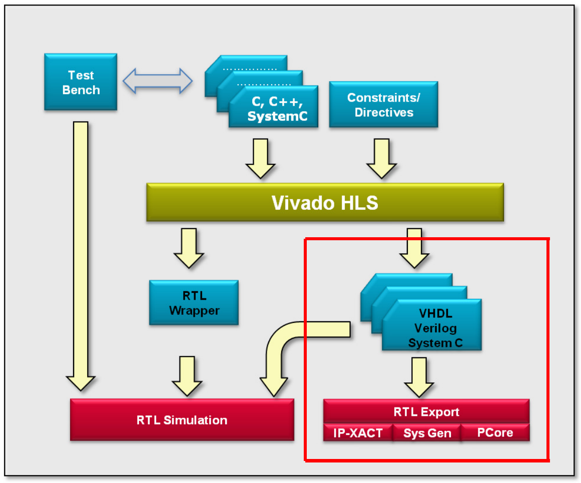
		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	NA	NA	NA	NA	NA	NA	NA
SystemC	Pass	3959	3959	3959	3960	3960	3960

Export the report(.html) using the Export Wizard

Vivado HLS Console

```
Info: /OSCI/SystemC: Simulation stopped by user.
@I [SIM-316] Starting C post checking ...
    *** *** *** ***
    Results are good
    *** *** *** ***
@I [SIM-1000] *** C/RTL co-simulation finished: PASS ***
@I [LIC-101] Checked in feature [VIVADO_HLS]
```

Vivado HLS : RTL Export



RTL output in Verilog, VHDL and SystemC

Scripts created for RTL synthesis tools

RTL Export to IP-XACT, SysGen, and Pcore formats

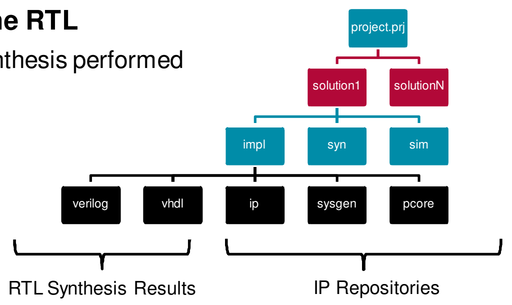
IP-XACT and SysGen => Vivado HLS for 7 Series and Zynq families
PCore => Only Vivado HLS Standalone for all families

RTL Export Support

- **RTL Export**
  - Can be exported to one of the three types
    - IP-XACT formatted IP for use with Vivado System Edition (SE)
      - 7 Series and Zynq families only
    - A System Generator IP block
      - 7 Series and Zynq families only
    - Pcore formated IP block for use with EDK
      - 7 Series, Zynq, Spartan-3, Spartan-6, Virtex-4/5/6 families
- **Generation in both Verilog and VHDL for non-bus or non-interface based designs**
- **Logic synthesis will automatically be performed**
  - HLS license will use Vivado RTL Synthesis

RTL Export: Synthesis

- **RTL Synthesis can be performed to evaluate the RTL**
  - IP-XACT and System Generator formats: Vivado synthesis performed
  - Pcore format: ISE synthesis is performed



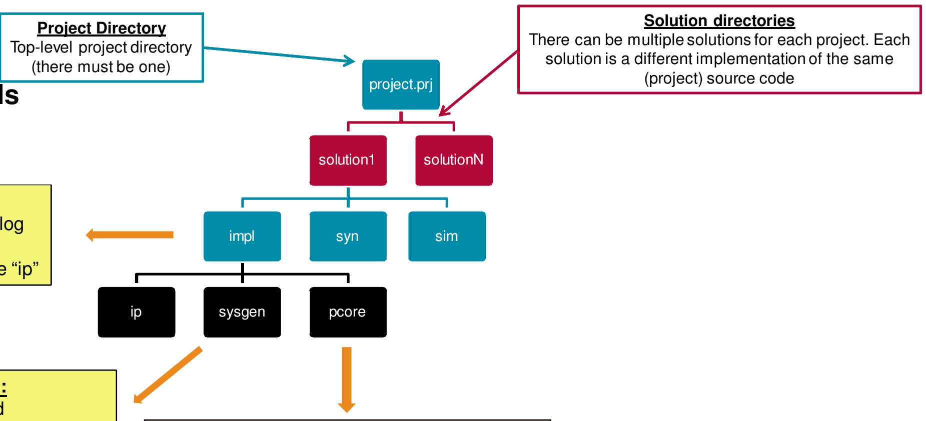
- **RTL synthesis results are not included with the IP package**
  - Evaluate step is provided to give confidence
    - Timing will be as estimate (or better)
    - Area will be as estimated (or better)
  - Final RTL IP is synthesized with the rest of the RTL design
    - RTL Synthesis results from the Vivado HLS evaluation are not used

RTL Export: IP Repositories

- IP can be imported into other Xilinx tools

Project Directory
Top-level project directory (there must be one)

Solution directories
There can be multiple solutions for each project. Each solution is a different implementation of the same (project) source code



In Vivado :
1. Project Manager > IP Catalog
2. Add IP to import this block
3. Browse to the zip file inside "ip"

In System Generator :
1. Use XilinxBlockAdd
2. Select Vivado_HLS block type
3. Browse to the solution directory

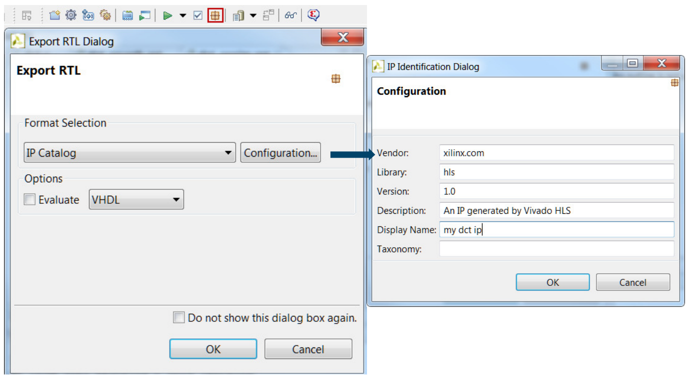
In EDK :
1. Copy the contents of the "pcore" direcory
2. Paste into the EDK project pcore direcotry
3. Project > Rescan Local Repository

RTL Export for Implementation

- Click on Export RTL
  - Export RTL Dialog opens
- Select the desired output format

IP Catalog
System Generator for DSP
System Generator for DSP (ISE)
Pcore for EDK
Synthesized Checkpoint (.dcp)

- Optionally, configure the output
- Select the desired language
- Optionally, click on Evaluate button for invoking implementation tools from within Vivado HLS
- Click OK to start the implementation

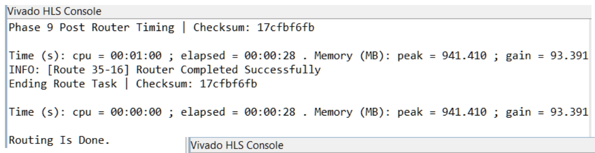


Format Selection: IP Catalog — Configuration...
Options: ☐ Evaluate — VHDL
☐ Do not show this dialog box again.

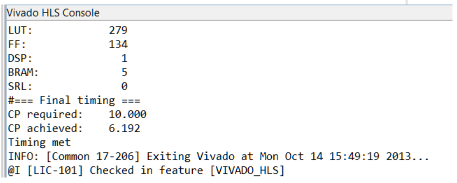
Vendor:	xilinx.com
Library:	hls
Version:	1.0
Description:	An IP generated by Vivado HLS
Display Name:	my dct ip
Taxonomy:	

RTL Export (Evaluate Option) Results

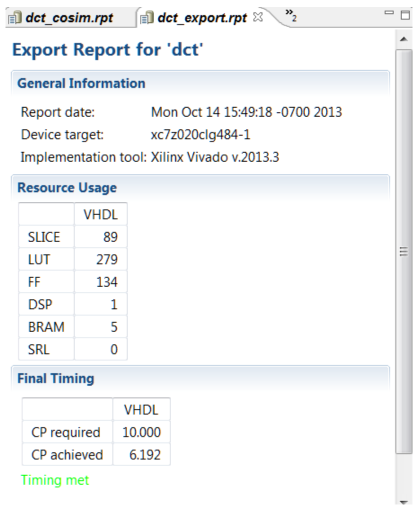
- **Impl directory created**
  - Will contain a sub-directory for each RTL which is synthesized
- **Report**
  - A report is created and opened automatically



```
Vivado HLS Console
Phase 9 Post Router Timing | Checksum: 17cfbf6fb

Time (s): cpu = 00:01:00 ; elapsed = 00:00:28 . Memory (MB): peak = 941.410 ; gain = 93.391
INFO: [Route 35-16] Router Completed Successfully
Ending Route Task | Checksum: 17cfbf6fb

Time (s): cpu = 00:00:00 ; elapsed = 00:00:28 . Memory (MB): peak = 941.410 ; gain = 93.391

Routing Is Done.
```



```
Vivado HLS Console
LUT:            279
FF:             134
DSP:              1
BRAM:             5
SRL:              0
#=== Final timing ===
CP required:    10.000
CP achieved:    6.192
Timing met
INFO: [Common 17-206] Exiting Vivado at Mon Oct 14 15:49:19 2013...
@I [LIC-101] Checked in feature [VIVADO_HLS]
```



Export Report for 'dct'

General Information

Report date: Mon Oct 14 15:49:18 -0700 2013
Device target: xc7z020clg484-1
Implementation tool: Xilinx Vivado v.2013.3

Resource Usage

	VHDL
SLICE	89
LUT	279
FF	134
DSP	1
BRAM	5
SRL	0

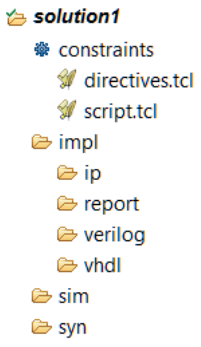
Final Timing

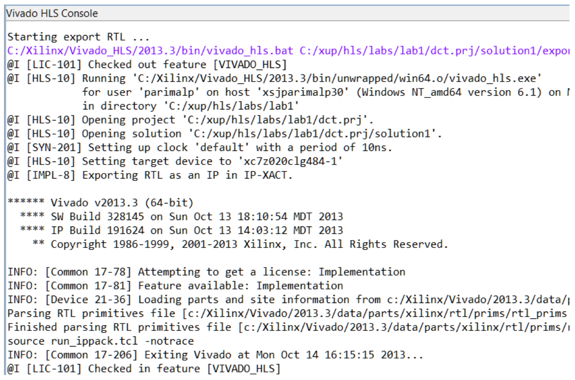
	VHDL
CP required	10.000
CP achieved	6.192

Timing met

RTL Export Results (Evaluate Option Unchecked)

- **Impl directory created**
  - Will contain a sub-directory for both VHDL and Verilog along with the ip directory
- **No report will be created**
- **Observe the console**
  - No packing, routing phases





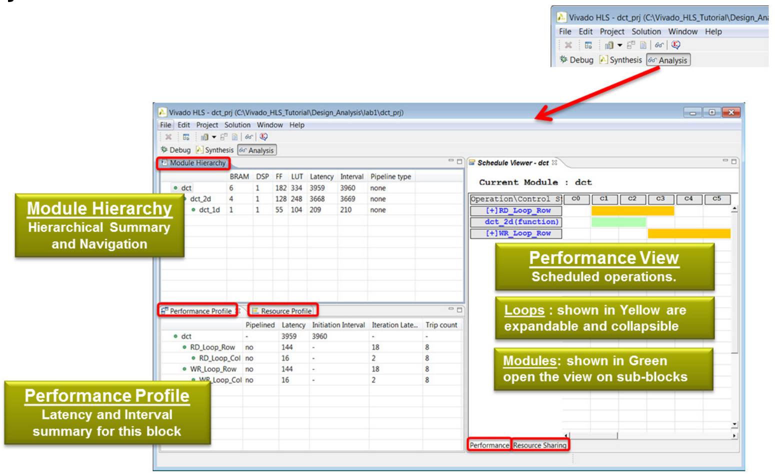
```
Vivado HLS Console

Starting export RTL ...
C:/Xilinx/Vivado_HLS/2013.3/bin/vivado_hls.bat C:/xup/hls/labs/lab1/dct.prj/solution1/expo
@I [LIC-101] Checked out feature [VIVADO_HLS]
@I [HLS-10] Running 'C:/Xilinx/Vivado_HLS/2013.3/bin/unwrapped/win64.o/vivado_hls.exe'
            for user 'parimalp' on host 'xsjparimalp30' (Windows NT_amd64 version 6.1) on
            in directory 'C:/xup/hls/labs/lab1'
@I [HLS-10] Opening project 'C:/xup/hls/labs/lab1/dct.prj'.
@I [HLS-10] Opening solution 'C:/xup/hls/labs/lab1/dct.prj/solution1'.
@I [SYN-201] Setting up clock 'default' with a period of 10ns.
@I [HLS-10] Setting target device to 'xc7z020clg484-1'
@I [IMPL-8] Exporting RTL as an IP in IP-XACT.

****** Vivado v2013.3 (64-bit)
  **** SW Build 328145 on Sun Oct 13 18:10:54 MDT 2013
  **** IP Build 191624 on Sun Oct 13 14:03:12 MDT 2013
    ** Copyright 1986-1999, 2001-2013 Xilinx, Inc. All Rights Reserved.

INFO: [Common 17-78] Attempting to get a license: Implementation
INFO: [Common 17-81] Feature available: Implementation
INFO: [Device 21-36] Loading parts and site information from c:/Xilinx/Vivado/2013.3/data/
Parsing RTL primitives file [c:/Xilinx/Vivado/2013.3/data/parts/xilinx/rtl/prims/rtl_prims
Finished parsing RTL primitives file [c:/Xilinx/Vivado/2013.3/data/parts/xilinx/rtl/prims/
source run_ippack.tcl -notrace
INFO: [Common 17-206] Exiting Vivado at Mon Oct 14 16:15:15 2013...
@I [LIC-101] Checked in feature [VIVADO_HLS]
```

Outline

- Invoking Vivado HLS
- Project Creation using Vivado HLS
- Synthesis to IPXACT Flow
- ***Design Analysis***
- Other Ways to use Vivado HLS
- Summary

Analysis Perspective

- **Perspective for design analysis**
  – Allows interactive analysis


Vivado HLS - dct_prj (C:\Vivado_HLS_Tutorial\Design_Analysis\lab1\dct_prj)
File Edit Project Solution Window Help
Debug
Synthesis
Analysis
Module Hierarchy
Performance Profile
Resource Profile
Schedule Viewer - dct
Performance
Resource Sharing
Module Hierarchy
Hierarchical Summary and Navigation
Performance Profile
Latency and Interval summary for this block
Performance View
Scheduled operations.
Loops : shown in Yellow are expandable and collapsible
Modules: shown in Green open the view on sub-blocks

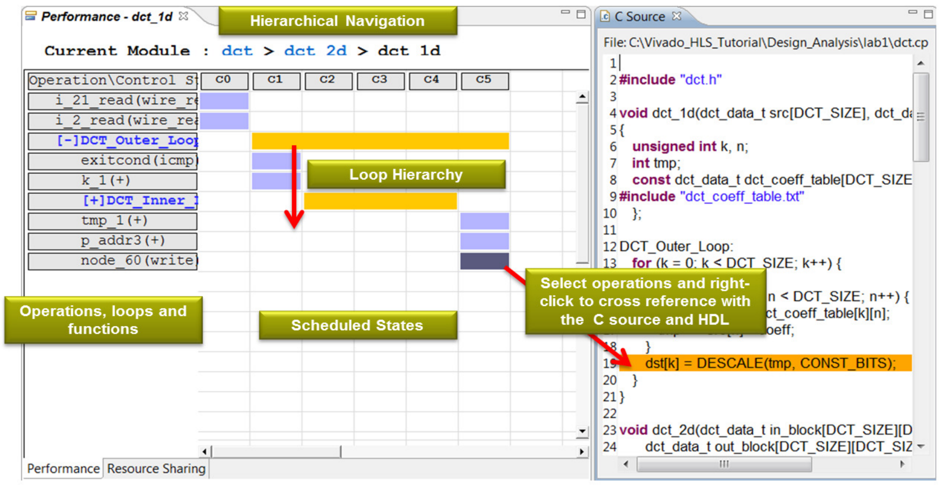
	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
dct	6	1	182	334	3959	3960	none
dct_2d	4	1	128	248	3668	3669	none
dct_1d	1	1	55	104	209	210	none

	Pipelined	Latency	Initiation Interval	Iteration Late...	Trip count
dct	-	3959	3960	-	-
RD_Loop_Row	no	144	-	18	8
RD_Loop_Col	no	16	-	2	8
WR_Loop_Row	no	144	-	18	8
WR_Loop_Col	no	16	-	2	8

Current Module : dct

Operation\Control St	C0	C1	C2	C3	C4	C5
[+]RD_Loop_Row		🟨	🟨	🟨		
dct_2d(function)		🟩	🟩			
[+]WR_Loop_Row				🟨	🟨	🟨

Performance Analysis


Performance - dct_1d
Hierarchical Navigation
Loop Hierarchy
Operations, loops and functions
Scheduled States
Select operations and right-click to cross reference with the C source and HDL
C Source
Performance
Resource Sharing

Current Module : dct > dct 2d > dct 1d

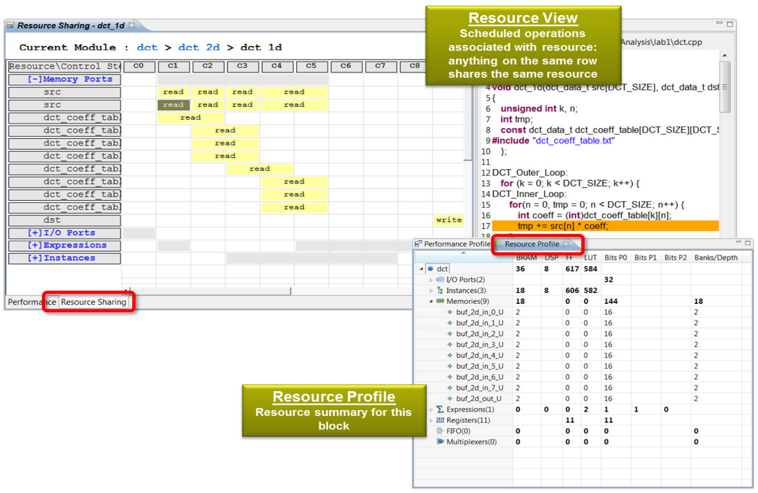
Operation\Control St	C0	C1	C2	C3	C4	C5
i_21_read(wire_re						
i_2_read(wire_re						
[-]DCT_Outer_Loop						
exitcond(icmp)						
k_1(+)						
[+]DCT_Inner_						
tmp_1(+)						
p_addr3(+)						
node_60(write)						

File: C:\Vivado_HLS_Tutorial\Design_Analysis\lab1\dct.cp

```
#include "dct.h"

void dct_1d(dct_data_t src[DCT_SIZE], dct_da
{
   unsigned int k, n;
   int tmp;
   const dct_data_t dct_coeff_table[DCT_SIZE
#include "dct_coeff_table.txt"
   };

DCT_Outer_Loop:
   for (k = 0; k < DCT_SIZE; k++) {
                       n < DCT_SIZE; n++) {
                       ct_coeff_table[k][n];
                       oeff;
      }
      dst[k] = DESCALE(tmp, CONST_BITS);
   }
}

void dct_2d(dct_data_t in_block[DCT_SIZE][D
      dct_data_t out_block[DCT_SIZE][DCT_SIZ
```

XILINX ALL PROGRAMMABLE.

Resources Analysis


Resource View
Scheduled operations associated with resource: anything on the same row shares the same resource
Resource Profile
Resource summary for this block
Resource Sharing - dct_1d
Performance
Resource Sharing
Performance Profile
Resource Profile

Current Module : dct > dct 2d > dct 1d

Resource\Control St	C0	C1	C2	C3	C4	C5	C6	C7	C8
[-]Memory Ports									
src		read	read	read	read				
src		read	read	read	read				
dct_coeff_tab		read							
dct_coeff_tab			read						
dct_coeff_tab			read						
dct_coeff_tab			read						
dct_coeff_tab				read					
dct_coeff_tab					read				
dct_coeff_tab					read				
dct_coeff_tab					read				
dst									write
[+]I/O Ports									
[+]Expressions									
[+]Instances									

Analysis\lab1\dct.cpp

```
void dct_1d(dct_data_t src[DCT_SIZE], dct_data_t dst
{
   unsigned int k, n;
   int tmp;
   const dct_data_t dct_coeff_table[DCT_SIZE][DCT_S
#include "dct_coeff_table.txt"
   };

DCT_Outer_Loop:
   for (k = 0; k < DCT_SIZE; k++) {
DCT_Inner_Loop:
      for(n = 0, tmp = 0; n < DCT_SIZE; n++) {
         int coeff = (int)dct_coeff_table[k][n];
         tmp += src[n] * coeff;
```

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth
dct	36	8	617	584				
I/O Ports(2)					32			
Instances(3)	18	8	606	582				
Memories(9)	18		0	0	144			18
buf_2d_in_0_U	2		0	0	16			2
buf_2d_in_1_U	2		0	0	16			2
buf_2d_in_2_U	2		0	0	16			2
buf_2d_in_3_U	2		0	0	16			2
buf_2d_in_4_U	2		0	0	16			2
buf_2d_in_5_U	2		0	0	16			2
buf_2d_in_6_U	2		0	0	16			2
buf_2d_in_7_U	2		0	0	16			2
buf_2d_out_U	2		0	0	16			2
Expressions(1)	0	0	0	2	1	1	0	
Registers(11)			11		11			
FIFO(0)	0		0	0	0			0
Multiplexers(0)	0		0	0	0			0

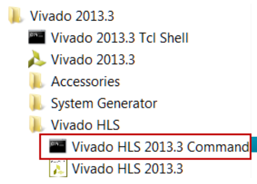
XILINX ALL PROGRAMMABLE.

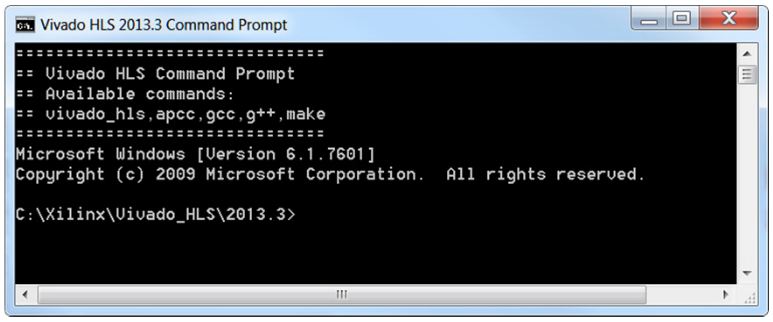
Outline

- Invoking Vivado HLS
- Project Creation using Vivado HLS
- Synthesis to IPXACT Flow
- Design Analysis
- ***Other Ways to use Vivado HLS***
- Summary

Command Line Interface: Batch Mode

- **Vivado HLS can also be run in batch mode**
  - Opening the Command Line Interface (CLI) will give a shell





```
==============================
== Vivado HLS Command Prompt
== Available commands:
== vivado_hls,apcc,gcc,g++,make
==============================
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Xilinx\Vivado_HLS\2013.3>
```

  - Supports the commands required to run Vivado HLS & pre-synthesis verification (gcc, g++, apcc, make)

Using Vivado HLS CLI

- **Invoke Vivado HLS in interactive mode**
  - Type Tcl commands one at a time

```
> vivado_hls –i
```

- **Execute Vivado HLS using a Tcl batch file**
  - Allows multiple runs to be scripted and automated

```
> vivado_hls –f run_aesl.tcl
```

- **Open an existing project in the GUI**
  - For analysis, further work or to modify it

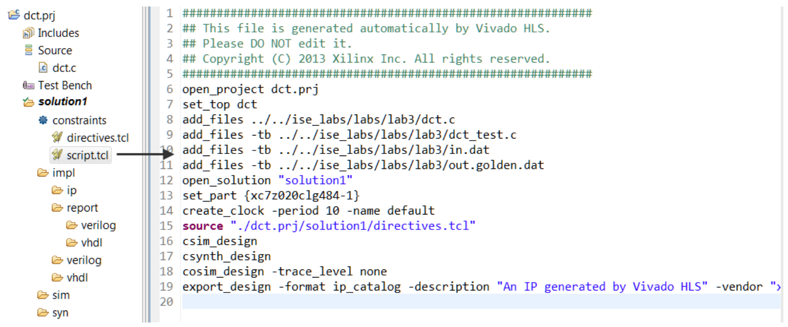
```
> vivado_hls –p my.prj
```

- **Use the shell to launch Vivado HLS GUI**

```
> vivado_hls
```

Using Tcl Commands

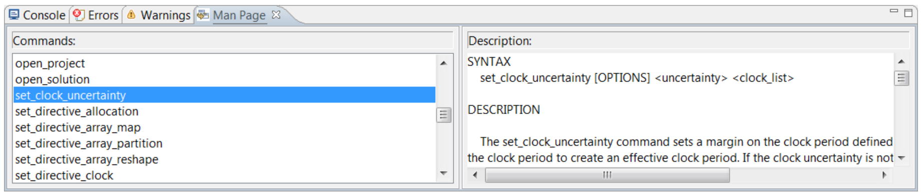
- **When the project is created**
  - All Tcl command to run the project are created in script.tcl
    - User specified directives are placed in directives.tcl
  - Use this as a template from creating Tcl scripts
    - Uncomment the commands before running the Tcl script



```
############################################################
## This file is generated automatically by Vivado HLS.
## Please DO NOT edit it.
## Copyright (C) 2013 Xilinx Inc. All rights reserved.
############################################################
open_project dct.prj
set_top dct
add_files ../../ise_labs/labs/lab3/dct.c
add_files -tb ../../ise_labs/labs/lab3/dct_test.c
add_files -tb ../../ise_labs/labs/lab3/in.dat
add_files -tb ../../ise_labs/labs/lab3/out.golden.dat
open_solution "solution1"
set_part {xc7z020clg484-1}
create_clock -period 10 -name default
source "./dct.prj/solution1/directives.tcl"
csim_design
csynth_design
cosim_design -trace_level none
export_design -format ip_catalog -description "An IP generated by Vivado HLS" -vendor "x
```

Help

- **Help is always available**
  - The Help Menu
  - Opens User Guide, Reference Guide and Man Pages



Commands:

open_project
open_solution
set_clock_uncertainty
set_directive_allocation
set_directive_array_map
set_directive_array_partition
set_directive_array_reshape
set_directive_clock

Description:

SYNTAX
set_clock_uncertainty [OPTIONS] <uncertainty> <clock_list>

DESCRIPTION

The set_clock_uncertainty command sets a margin on the clock period defined
the clock period to create an effective clock period. If the clock uncertainty is not

- **In interactive mode**
  - The help command lists the man page for all commands

```
Vivado_hls> help add_files

SYNOPSIS
   add_files  [OPTIONS]  <src_files>
Etc…
```

Auto-Complete all commands using the tab key

Outline

- Invoking Vivado HLS
- Project Creation using Vivado HLS
- Synthesis to IPXACT Flow
- Design Analysis
- Other Ways to use Vivado HLS
- ***Summary***

Summary

- Vivado HLS can be run under Windows XP, Windows 7, Red Hat Linux, and SUSE OS
- Vivado HLS can be invoked through GUI and command line in Windows OS, and command line in Linux
- Vivado HLS project creation wizard involves
  - Defining project name and location
  - Adding design files
  - Specifying testbench files
  - Selecting clock and technology
- The top-level module in testbench is main() whereas top-level module in the design is the function to be synthesized

Summary

- Vivado HLS project directory consists of
  - *.prj project file
  - Multiple solutions directories
  - Each solution directory may contain
    - impl, synth, and sim directories
    - The impl directory consists of pcores, verilog, and vhdl folders
    - The synth directory consists of reports, systemC, vhdl, and verilog folders
    - The sim directory consists of testbench and simulation files

Lab1 Intro
Vivado HLS Design Flow

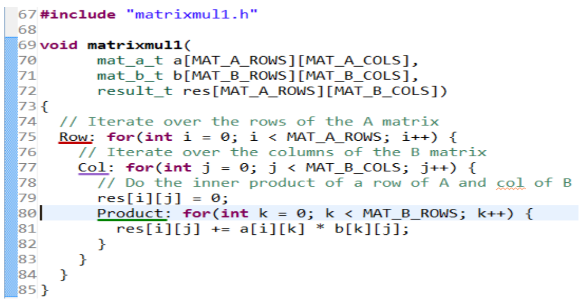
Vivado HLS 2013.3 Version
ZedBoard

Objectives

- **After completing this lab, you will be able to:**
  - Create a project in Vivado HLS
  - Run C-simulation
  - Use debugger
  - Synthesize and implement the design using the default options
  - Use design analysis perspective to see what is going on under the hood
  - Understand and analyze the generated output

The Design

- This lab uses a simple matrix multiplication example to walk you through the Vivado HLS project creation and analysis steps. The design consists of three nested loops. The Product loop is the inner most loop performing the actual elements product. The Col loop is the outer-loop which feeds next column element data with the passed row element data to the Product loop. Finally, Row is the outer-most loop. The res[i][j]=0 (line 79) resets the result every time a new row element is passed and new column element is used



```
67 #include "matrixmul1.h"
68
69 void matrixmul1(
70       mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
71       mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
72       result_t res[MAT_A_ROWS][MAT_B_COLS])
73 {
74   // Iterate over the rows of the A matrix
75   Row: for(int i = 0; i < MAT_A_ROWS; i++) {
76     // Iterate over the columns of the B matrix
77     Col: for(int j = 0; j < MAT_B_COLS; j++) {
78       // Do the inner product of a row of A and col of B
79       res[i][j] = 0;
80       Product: for(int k = 0; k < MAT_B_ROWS; k++) {
81         res[i][j] += a[i][k] * b[k][j];
82       }
83     }
84   }
85 }
```

Procedure

- Create a project after starting Vivado HLS in GUI mode
- Run C simulation
  – to understand the design behavior
- Run the debugger
  – to see how the top-level module works
- Synthesize the design
- Analyze the generated output using the Analysis perspective
- Run C/RTL cosimulation
  – to perform RTL simulation
- View simulation results in Vivado
  – to understand the IO protocol
- Export RTL in the Evaluate mode and run the implementation

Summary

- In this lab, you completed the major steps of the high-level synthesis design flow using Vivado HLS. You created a project, added source files, synthesized the design, simulated the design, and implemented the design. You also learned that how to use the Analysis perspective to understand the scheduling