Advanced Computer Architecture

—

Part II: Embedded Computing
High Level Synthesis

EPFL – I&C – LAP

Adapted from Xilinx UG871 & UG902

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Vivado High Level Synthesis (HLS)

- Software tool for synthesis and analysis of HDL design
- Compiles C programs to generate RTL
- Targets FPGA platforms

- Our goal is to understand how to analyze & optimize HDL design using HLS, in particular Vivado HLS

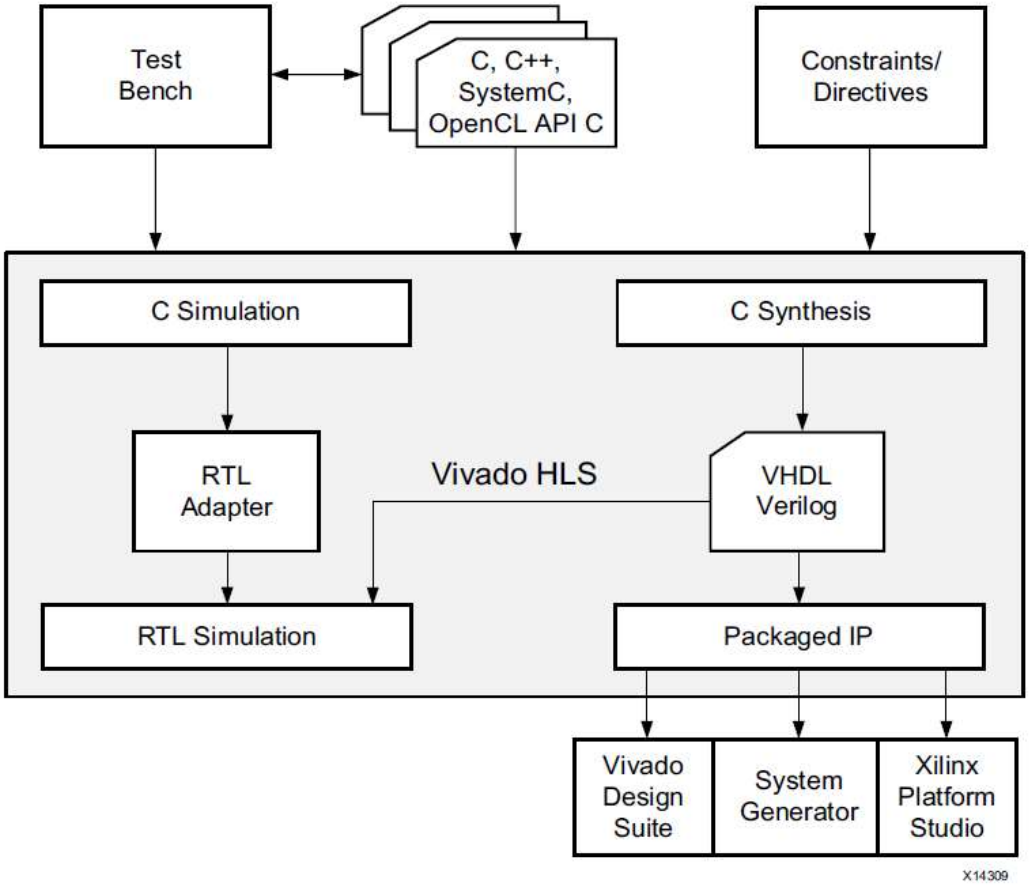
Vivado HLS Workflow

- Create a C file describing the HDL design
- Optimize performance using HLS directives
- Validate its functionality using C simulation
- Analyze the results using HLS analysis tools

Not covered in this lab

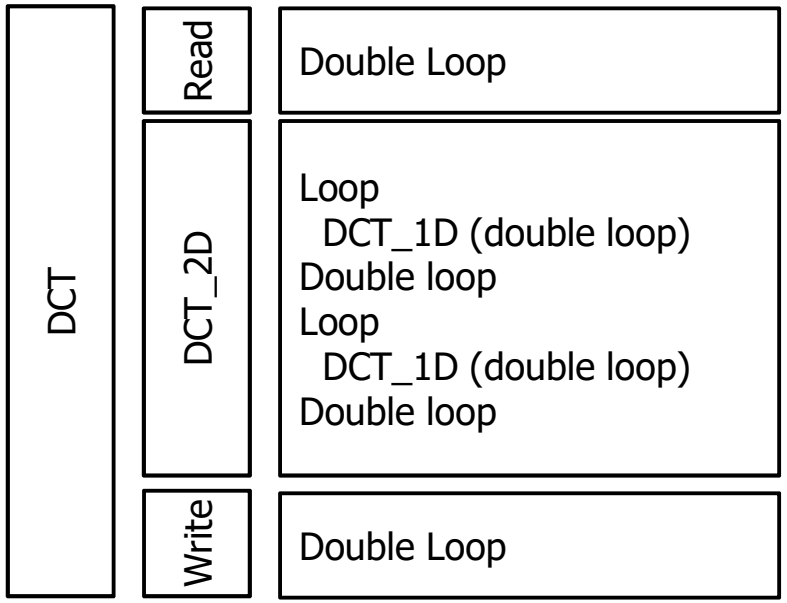
- Validate RTL
- Synthesize design and integrate it on an FPGA

Vivado Workflow



Design Under Test

- Implementation of "Discrete Cosine Transform"
- C code describes the function
- The code is given in the following hierarchy



Creating a Project

- Use tcl scripts to automatically create a project
  - Add C files for design
  - Add C files and data files for testbench
  - Create a new solution
    - One project can have multiple solutions
    - Each solution has different set of HLS optimization directives
    - One solution is active at a time
  - Set target FPGA and target clock period

Creating a Project

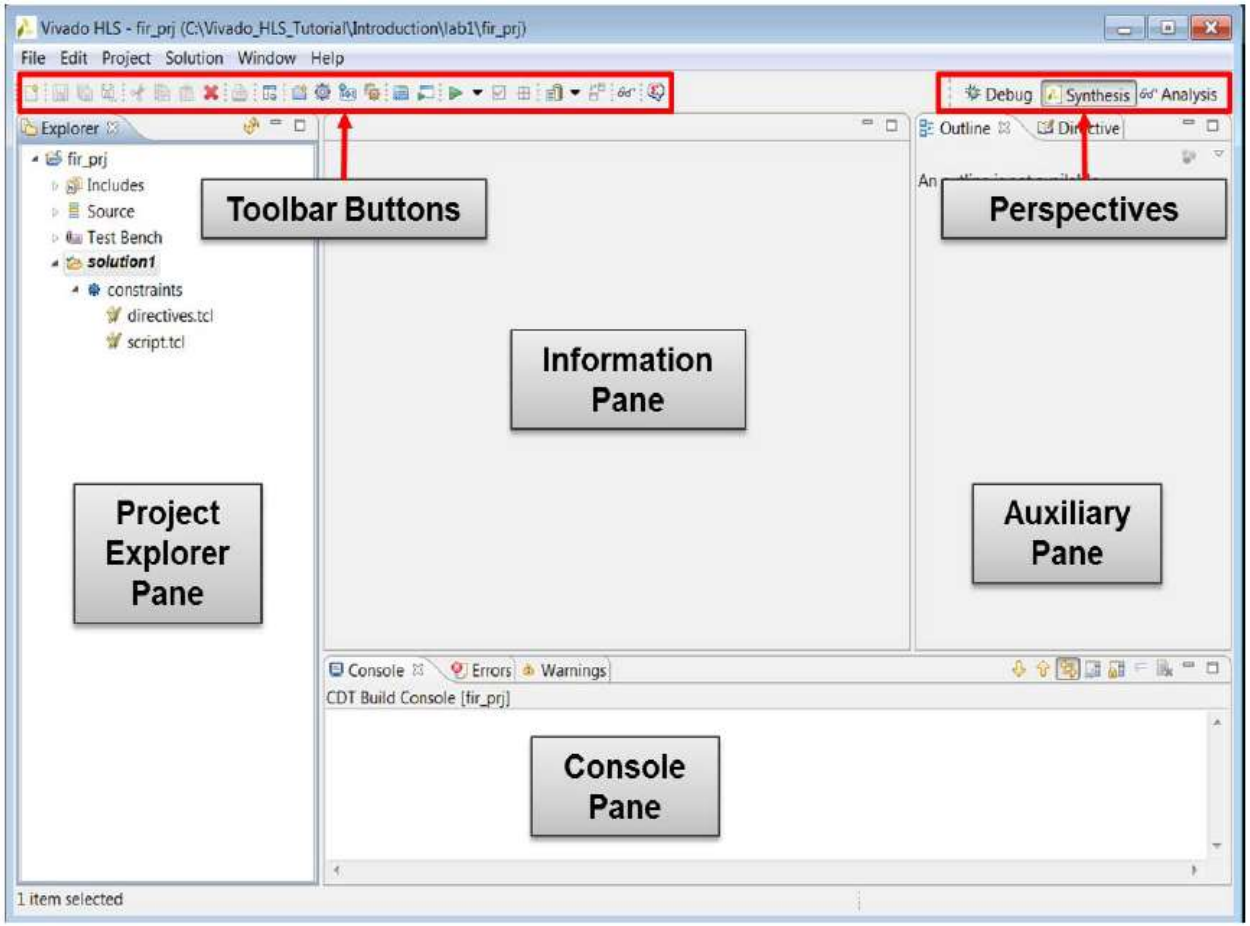
- Download the project files from Moodle page
  - Don't use any spaces in the file path
- Create a project from a tcl script
  - Windows: Using Vivado HLS Command Prompt
  - Linux: Using the terminal

```
vivado_hls -f run_hls.tcl
```

  - A new folder (dct_prj) should be created
- Open the project in Vivado GUI

```
vivado_hls -p dct_prj
```

Vivado GUI


Vivado HLS - fir_prj (C:\Vivado_HLS_Tutorial\Introduction\lab1\fir_prj)
File Edit Project Solution Window Help
Debug
Synthesis
Analysis
Explorer
fir_prj
Includes
Source
Test Bench
solution1
constraints
directives.tcl
script.tcl
Outline
Toolbar Buttons
Perspectives
Information Pane
Project Explorer Pane
Auxiliary Pane
Console
Errors
Warnings
CDT Build Console [fir_prj]
Console Pane
1 item selected

Navigating the Project

- The project starts in synthesis perspective
- Explorer pane shows different files & solutions in project
  - Open the source file (dct.cpp) from the explorer menu
  - The source file contains the main design
- Auxiliary pane shows outline of design & interface to add optimization directives

- Debug perspective: useful for C simulation
- Analysis perspective: useful for design performance analysis

Design Validation

- Validate design using write C file testbenches
  - Testbench already added to this project
- Allows to add design as a C function
- Run simulation
  - Project → Run C Simulation
  - You can use toolbar button as well
- Prints output on console
- Debugger (Run C Simulation + Launch debugger) gives advanced debug capabilities

Synthesis

- Synthesize the design (one solution at a time)
  - Solution → Run C Synthesis → Active Solution
  - You can use toolbar button as well
- Generates synthesis reports
  - Access reports in Explorer menu
  - Solution# → syn → project name
  - Each function in the design is an instance
  - Each instance has a report
- Keep an eye on synthesis report on the console

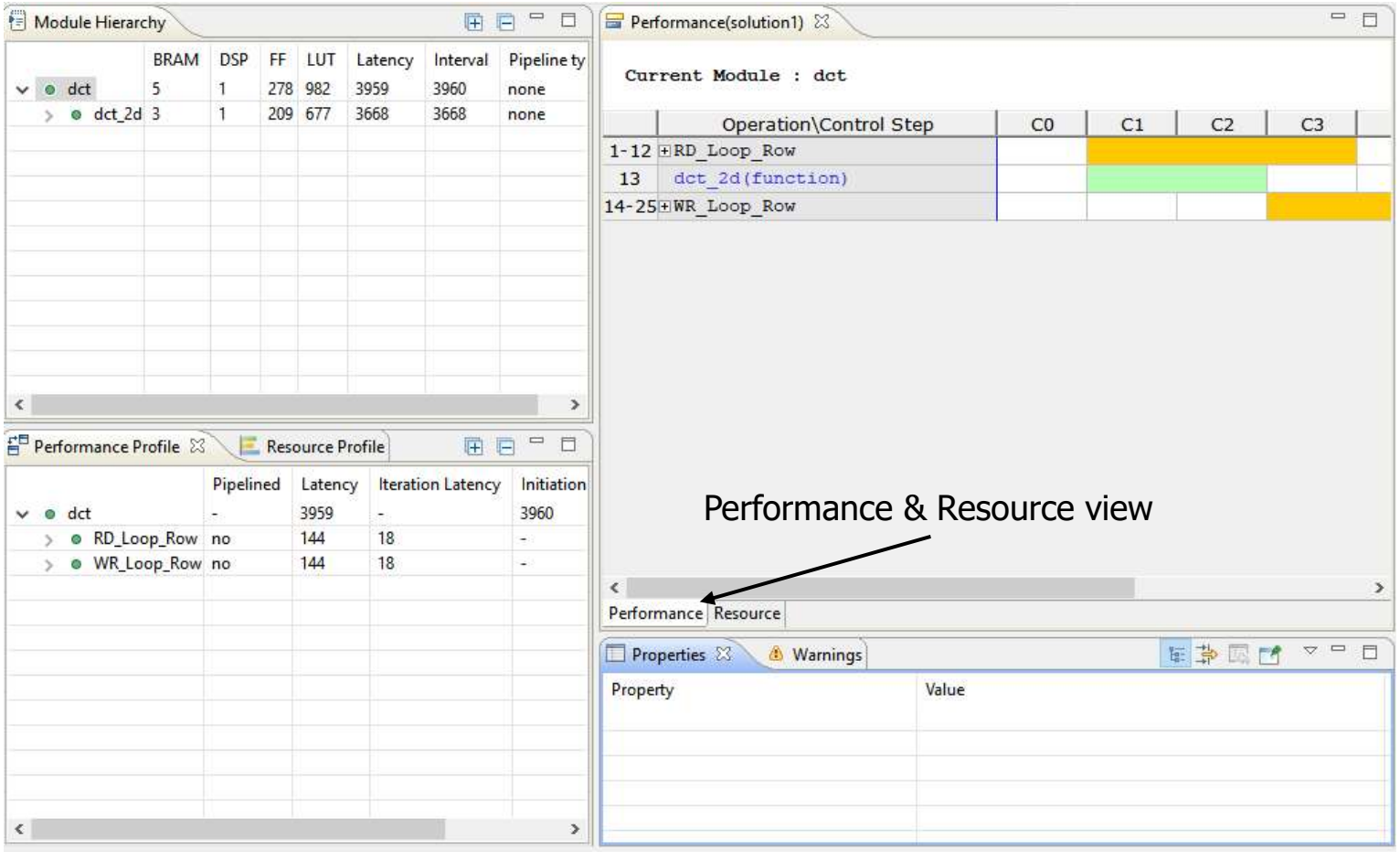
Synthesis Report

- Performance estimates
  - If design meets timing constraints
  - Reports latency and initiation interval (II)
  - Breaks down performance per instance and loop in the design
- Utilization estimates
  - Summarizes the resources needed for the design
- Interface
  - Reports the ports used by the design

Analysis Perspective

- Switch to Analysis perspective
- Performance view shows how operations in particular block are scheduled into clock cycles/control states
- The Resource view shows how the resources in the design are used in different control states
- Module Hierarchy breaks down performance per instance
  - Switch modules using this menu
- Performance Profile breaks down performance per loop

Analysis Perspective

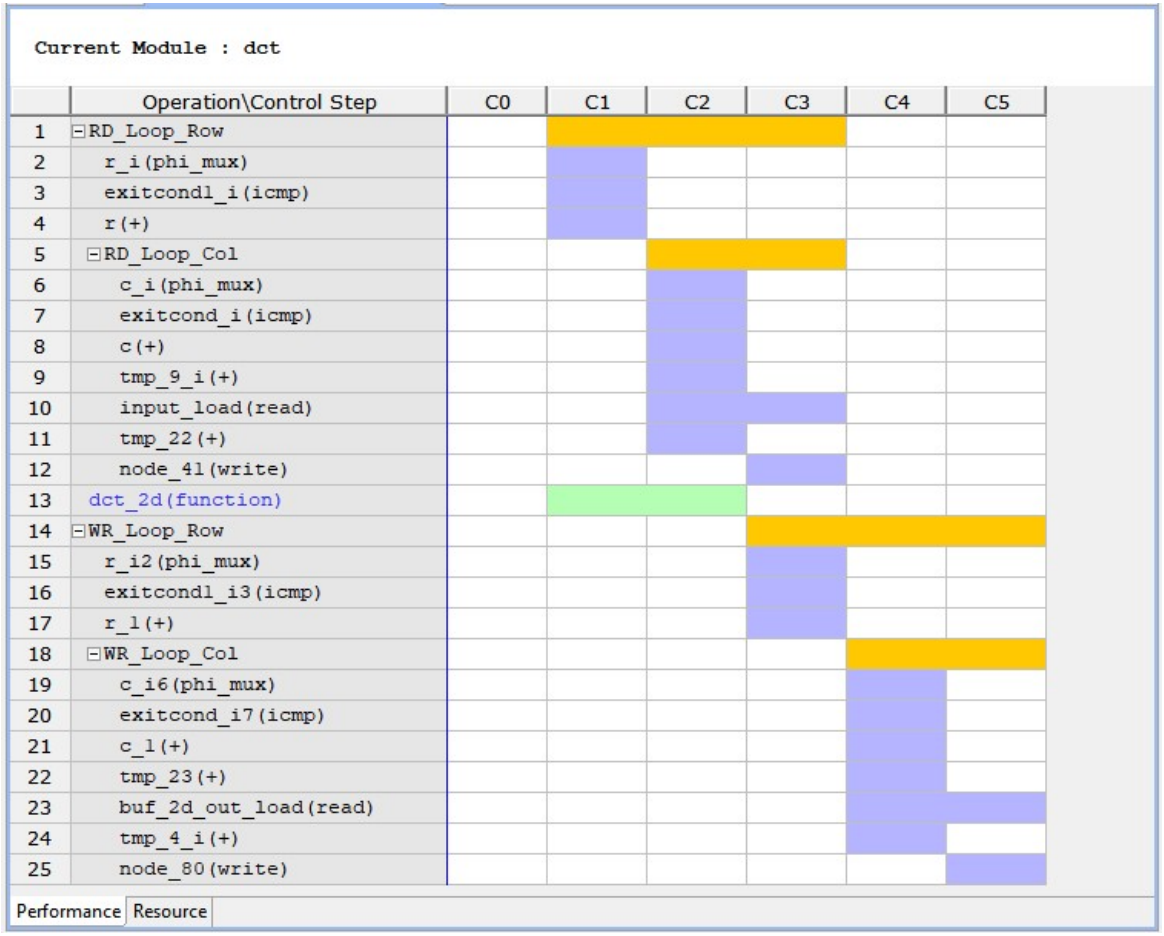


Analysis Perspective

- Looking at "dct" analysis, we see two loops even though we had only three functions
- Look at synthesis report on the console (switch perspectives for that)
- Notice how these functions were inlined

- Using data in performance profile, present latency breakdown of "RD_Loop_Row" and "WR_Loop_Row"

Performance View



Performance View

- Performance view color coding
  - Blue: basic operation
  - Yellow: loop
  - Green: Instance/function
- For "RD_Loop_Col" loop, explain each basic operation
  - Right-click on operation and "go to source" to get some insight
- What's the read latency? and the write latency?

Loop Pipelining

- Pipeline the loops to improve performance
  - Switch to synthesis directive
  - Create a new solution (copy from original solution)
  - For each of the inner most loops

    (DCT_Inner_Loop, Xpose_Row_Inner_Loop, Xpose_Col_Inner_Loopr, RD_Loop_Col, WR_Loop_Col)
    - Double click on the loop to insert directive
    - Pick pipeline directive
    - Keep directive destination in directives pane
    - (Adding to directive source creates issues with multiple solutions)
    - Leave options empty, II will be set to 1 (best case)
  - Synthesize

Loop Pipelining

- Compare performance with previous solution
  - Project → Compare Reports
  - How does pipelining decrease II?
  - How does pipelining affect the loop latency breakdown from slide15?

- Flattening is done automatically with pipelining
  - Synthesis report shows 4 flattened loops out of 5
  - Why "DCT_Outer_Loop" cannot be flattened?
  - Point to the reason by reading the synthesis report and relating it to the design code

Outer Loop Pipelining

- Pipelining outer loop causes inner loop to be unrolled
  - Solves issue with loops that can't be flattened
- Create new solution
  - Start from solution2
  - Remove pipeline directive on "DCT_Inner_Loop"
  - Add pipeline directive to "DCT_Outer_Loop"
  - Synthesize
- How does unrolling affect performance and resource utilization?

Note that design might not meet timing, though it is not major

Review for Bottlenecks

- Using the performance view in the analysis check which loop can still be further optimized, explain

Resource View

- Observe inner loop of "dct_1d2" in performance view
  - Notice operations in this loop are serialized even though loop is pipelined
  - Can you explain this behavior?
- Switch to resource view
  - "src" read ports are used in every single cycle
  - How can we solve the port bottleneck?

Resource View

Current Module : dct > dct_2d > dct_1d2

	Resource\Control Step	C0	C1	C2	C3	C4	C5	C6	C7
1-6	⊞I/O Ports								
7	⊟Memory Ports								
8	dct_coeff_table_1(p0)		read						
9	dct_coeff_table_3(p0)		read						
10	src(p1)		read	read	read	read			
11	src(p0)		read	read	read	read			
12	dct_coeff_table_0(p0)			read					
13	dct_coeff_table_5(p0)			read					
14	dct_coeff_table_2(p0)			read					
15	dct_coeff_table_7(p0)			read					
16	dct_coeff_table_4(p0)			read					
17	dct_coeff_table_6(p0)			read					
18	dst(p0)								write
19-41	⊞Expressions								

BRAM Partitioning

- Trace inputs used for each instance of "dct_1d2"
- Partition these inputs to allow better pipelining
  - For each input add an "array_partition" directive
  - Set type to "complete"
  - Synthesize
- How does partitioning affect performance and resource utilization?