Vitis HLS <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction-to-Vitis-HLS>

Vitis HLS is a high level systhensis tool that allows C, C++ an OpenCL functions to become hardwired onto the device logic fabric and RAM/DSP block. Vitis HLS implements hardware kernels in the Vitis application development flow and uses C/C++ code for developing RTL IP for Xilinx device designs in the Vivado Design Suite.

RTL(Register transfer level) IP – The RTL Wizard page lets you define the RTL source for the IP.

<https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/RTL-IP-Definition>

Vitis HLS also supports customization of your code to implement different interface standards or specific optimizations to achieve your design objects.

Following is the Vitis HLS design flow:  
1.Compile, simulate and debug the C/C++ algorithm

2. View reports to analyza and optimize the design.

3. Synthesize the C algorithm into an RTT design

4.Verify the RTL implementation using RTL co-simulation

5.Package the RTL implementation into e compiled object file extension, or export an RTL IP.

Vitis HLS Memory Layout Model <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Vitis-HLS-Memory-Layout-Model>

The Vitis application acceleration development flow provides a framework for developing and delivering FPGA accelerated applications using standard programming languages for both software and hardware components. The software component or host program is developed using C/C++ to run on x86 embedded processors with OpenCl/Native XRT API. The hardware components , or kernel (that runs on the actual FPGA card/platform), can be developed using C/C++, OpenCl C or RTL.

It becomes important to define the exact memory model that is used so that the data that is being read/write can be correctly processed. The memory model defines the way data is arranged and accessed in computer memory. It consists of two separate but related issues: data alignment and data structure padding. In addition the Vitis HLS compiler supports the specification of special attributes (and pragmas) to change the default data alignment and data structure padding rules.

Data Alignment <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Data-Alignment>

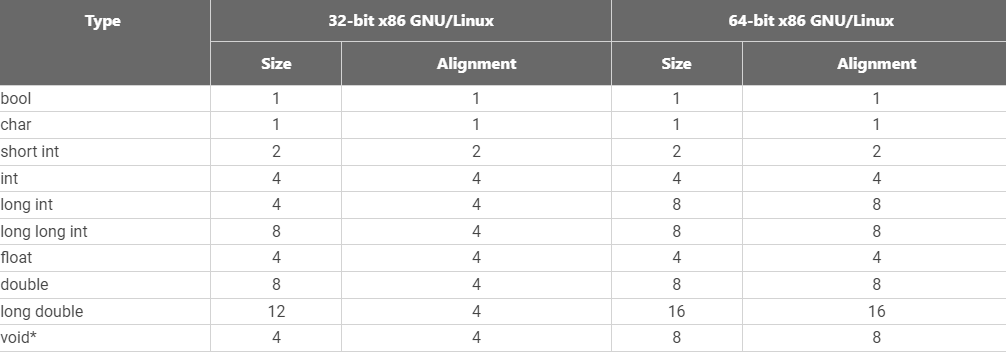
Software programmers are conditioned to think of memory as a simple array of bytes and basic data types are composed of one or more blocks of memory. Today’s modern CPU access memory in 2 4 8 16 or even 32 byte chunks at a time -although 32 bit and 64 but instruction set architecture (ISA) architectures are the most common.

If you don’t understand the address alignment issues in your software, the following situations are all possible:

* Your software will run slower
* Your application will lock/hang
* Your operating system can crash
* Your software will silently fail, yielding incorrect results

To make manipulating variables of these types fast, the generated object code will try to use CPU instructions that read/write the whole data type at once. This in turn means that the variables of these types should be placed in memory in a way that makes their addresses suitably aligned.

The following table shows the size and alignment for the basic native data types in C/C++



Why does a programmer need to change the alignment?

The main reason will be to trade off between requirements and performance. Fortunately, the GCC C/C++ compiler provides the language extension **\_\_attribute\_\_((aligned(x)))** in order to change the default alignment for the variable, structures/classes, or a structure field, measured in bytes.

Ex: The following declaration causes the compiler to allocate the global variables x on a 16-byte boundary

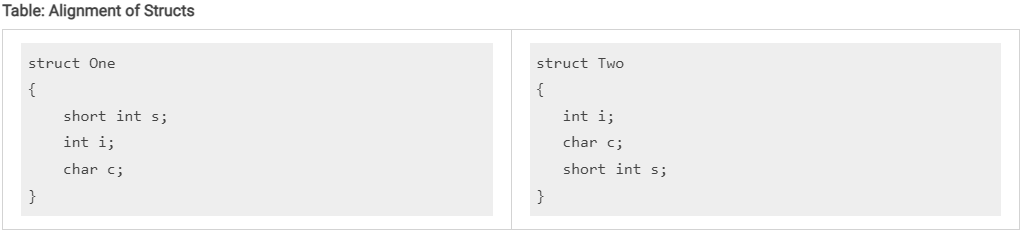
Int x \_\_attribute\_\_((aligned(16)))=0;

The \_\_attribute\_\_((aligned(X))) does not change the size of variables it is applied to, but may change the memory layout of structures by inserting padding between elements of the struct.

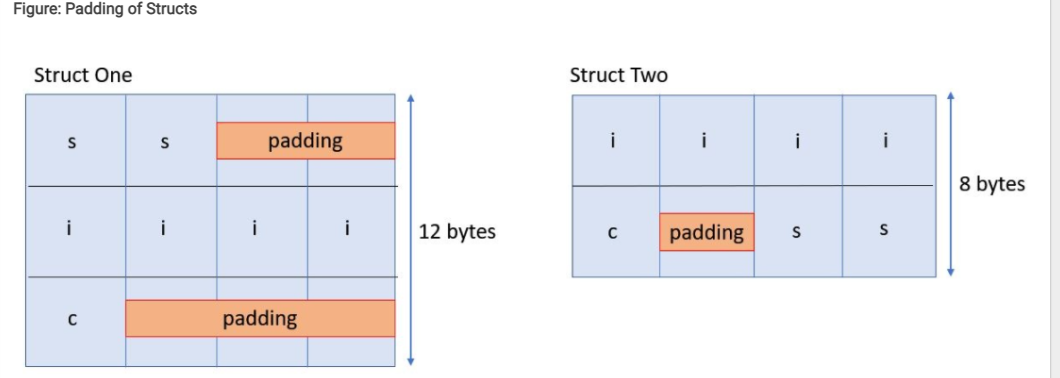
Data Structure Padding <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Data-Structure-Padding>

The C++ compiler also needs to make sure that all the member variables in a struct or class are properly aligned. In addition, to make sure that each element in an array of a user-defined type is aligned, the compiler may add some extra padding after the last data member.

Example:



Assuming the memory are on x86-64 alignment with short int having the alignment of 2 and int having an alignment of 4, to make the i data member of struct one suitably aligned, the compiler needs to insert two extra bytes of padding s and I to create alignment.



The GCC C/C++ compiler provides a language extension, \_\_attribute\_\_((packed)) which tells the compiler not to insert padding but rather allow the struct members to be misaligned.

Vitis HLS Alignment Rules and Semantics <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Vitis-HLS-Alignment-Rules-and-Semantics>

First, you need to understand the Aggregate and Disaggregate features in Vitis HLS

**Set\_directive\_aggregate**

This directive collects the data fields of a struct into a single wide scalar. Any arrays declared within the struct and Vitis HLS performs a similar operation as set\_directive\_array\_reshape, and completely partions and reshapes the array into a wide scalar and packs it with other elements of the struct.

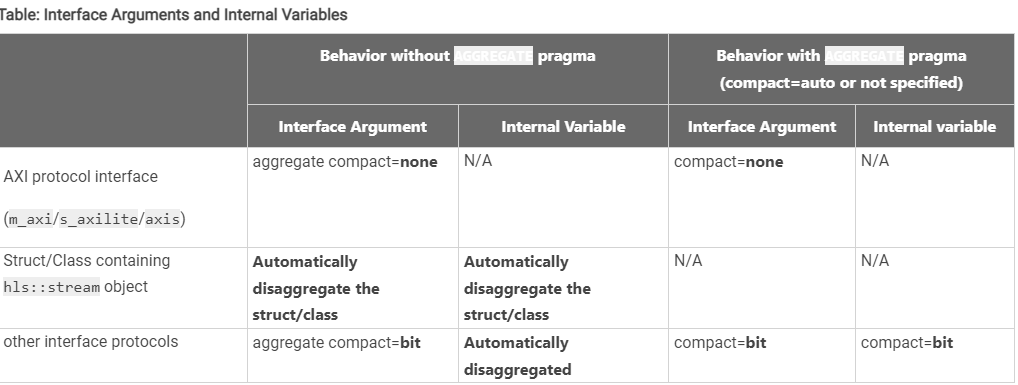
Set\_directive\_aggregate [OPTIONS] <location><variables>

**Set\_directive\_disaggregate**

Let’s you deconstruct a struct variable into its individual elements. The number and type of elements created are determined by the contents of struct itself.

Set\_directive\_diseggregate <location><variable>

The first element of the struct is aligned on the LSB of the vector and the final element of the struct is aligned with the MSB of the vector.



The goal of the default aggregation behavior in Vitis HLS is to use an x86\_64-gnu-linux memory layout at the top level hardware interface while optimizing the internal hardware for better quality of results.

Aggregate Memory Mapped Interface <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Examples-of-Aggregation>

This is an example of the AGGREGATE pragma or directive for an m\_axis interface:

struct A {

char foo; // 1 byte

short bar; // 2 bytes

};

int dut(A\* arr) {

#pragma HLS interface m\_axi port=arr depth=10

#pragma HLS aggregate variable=arr compact=auto

int sum = 0;

for (unsigned i=0; i<10; i++) {

auto tmp = arr[i];

sum += tmp.foo + tmp.bar;

}

return sum;

}

For the above example, the size of the m\_axis interface port arr is 3 byte (ot 24bits) but due to the AGGREGATE compact=auto pragma, the size of the port will be aligned to 4 bytes(or 32 bits) as this is the closest power of 2.

Aggregate Structs on the Interface

This is an example of the AGGREGATE pragma or directive for an ap\_fifo interface.

struct A {

int myArr[3]; // 4 bytes per element (12 bytes total)

ap\_int<23> length; // 23 bits

};

int dut(A arr[N]) {

#pragma HLS interface ap\_fifo port=arr

#pragma HLS aggregate variable=arr compact=auto

int sum = 0;

for (unsigned i=0; i<10; i++) {

auto tmp = arr[i];

sum += tmp.myArr[0] + tmp.myArr[1] + tmp.myArr[2] + tmp.length;

}

return sum;

}

For ap\_fifo interface, the struct will packed at the bit-level with or without aggregate pragma.

In the above example, the AGGREGATE pragma will create a port of size 119 bits for port arr. The array myArr will take 12 bytes (or 96 bits) and the element length will take 23 bits for a total of 119 bits.

Aggregate Nested Struct Port

This is an example of the AGGREGATE pragma or directive in the Vivado IP Flow.

#define N 8

struct T {

int m; // 4 bytes

int n; // 4 bytes

bool o; // 1 byte

};

struct S {

int p; // 4 bytes

T q; // 9 bytes

};

void top(S a[N], S b[N], S c[N]) {

#pragma HLS interface bram port=c

#pragma HLS interface ap\_memory port=a

#pragma HLS aggregate variable=a compact=byte

#pragma HLS aggregate variable=b compact=bit

#pragma HLS aggregate variable=c compact=byte

for (int i=0; i<N; i++) {

c[i].q.m = a[i].q.m + b[i].q.m;

c[i].q.n = a[i].q.n - b[i].q.n;

c[i].q.o = a[i].q.o || b[i].q.o;

c[i].p = a[i].q.n;

}

}

In the above example, the aggregation algorithm will create a port of size 104 bits for ports a and c as the compact=byte option was specified in the aggregate pragma but the compact bite default option is used for port b and its packed size will be 97 bits. The nested structure S and T are aggregated to encompass three 32 bit member variables (p,m and n) and one bit/byte member variable(o)

Examples of Disaggregation <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Examples-of-Disaggregation>

Disaggregate AXIS Interface

This is an example of the DISAGGREGATE pragma or directive for an axis interface

HLS Source Code

#define N 10

struct A {

char c;

int i;

};

void dut(A in[N], A out[N]) {

#pragma HLS interface axis port=in

#pragma HLS interface axis port=out

#pragma HLS disaggregate variable=in

#pragma HLS disaggregate variable=out

int sum = 0;

for (unsigned i=0; i<N; i++) {

out[i].c = in[i].c;

out[i].i = in[i].i;

}

}

Synthesized IP Module

module dut (

ap\_local\_block,

ap\_local\_deadlock,

ap\_clk,

ap\_rst\_n,

ap\_start,

ap\_done,

ap\_idle,

ap\_ready,

in\_c\_TVALID,

in\_i\_TVALID,

out\_c\_TREADY,

out\_i\_TREADY,

in\_c\_TDATA,

in\_c\_TREADY,

in\_i\_TDATA,

in\_i\_TREADY,

out\_c\_TDATA,

out\_c\_TVALID,

out\_i\_TDATA,

out\_i\_TVALID

);

In the above disaggregation example, the struct arguments in and out are mapped to AXIS interfaces and then disaggregated. This results in Vitis HLS creating two AXI streams for each argument: in\_c, in\_i and out\_c and out\_i. Each member of the struct A becomes a separate stream.

Disaggregate HLS::STREAM

This is an example of the DISAGGREGATE pragma or directive when used with the hls::stream type.

HLS Source Code

#define N 1024

struct A {,

hls::stream<int> s\_in;

long arr[N];

};

long dut(struct A &d) {

long sum = 0;

while(!d.s\_in.empty())

sum += d.s\_in.read();

for (unsigned i=0; i<N; i++)

sum += d.arr[i];

return sum;

}

Synthesized IP Module

module dut (

ap\_local\_block,

ap\_local\_deadlock,

ap\_clk,

ap\_rst,

ap\_start,

ap\_done,

ap\_idle,

ap\_ready,

d\_s\_in\_dout,

d\_s\_in\_empty\_n,

d\_s\_in\_read,

d\_arr\_ce0,

d\_arr\_q0,

ap\_return

);

Using an hls::stream object inside a structure that is used in the interface will cause the struct port to be automatically disaggregated by the Vitis HLS compiler.

Impact of Struct Size on Pipelining

The size of a struct used In a function interface can adversely impact pipelining of loops in that function that have access to the interface in the loop body.

struct A { /\* Total size = 192 bits (32 x 6) or 24 bytes \*/

int s\_1;

int s\_2;

int s\_3;

int s\_4;

int s\_5;

int s\_6;

};

void read(A \*a\_in, A buf\_out[NUM]) {

READ:

for (int i = 0; i < NUM; i++)

{

buf\_out[i] = a\_in[i];

}

}

void compute(A buf\_in[NUM], A buf\_out[NUM], int size) {

COMPUTE:

for (int j = 0; j < NUM; j++)

{

buf\_out[j].s\_1 = buf\_in[j].s\_1 + size;

buf\_out[j].s\_2 = buf\_in[j].s\_2;

buf\_out[j].s\_3 = buf\_in[j].s\_3;

buf\_out[j].s\_4 = buf\_in[j].s\_4;

buf\_out[j].s\_5 = buf\_in[j].s\_5;

buf\_out[j].s\_6 = buf\_in[j].s\_6 % 2;

}

}

void write(A buf\_in[NUM], A \*a\_out) {

WRITE:

for (int k = 0; k < NUM; k++)

{

a\_out[k] = buf\_in[k];

}

}

void dut(A \*a\_in, A \*a\_out, int size)

{

#pragma HLS INTERFACE m\_axi port=a\_in bundle=gmem0

#pragma HLS INTERFACE m\_axi port=a\_out bundle=gmem1

A buffer\_in[NUM];

A buffer\_out[NUM];

#pragma HLS dataflow

read(a\_in, buffer\_in);

compute(buffer\_in, buffer\_out, size);

write(buffer\_out, a\_out);

}

In the above example, the size of struct A is 192 bits, which is not a power of 2. Vitis HLS will automatically size the two M\_AXI interfaces to be size of 256

The way to fix such II issues is to pad struct A with 8 additional bytes such that you are always writing 256 bits at a time or by using the other alternative shown in the table below.



Basic of High-Level Synthesis

The Xilinx Vitis HLS tool synthesizes a C or C++ function into RTL code for acceleration in programmable logic

Some benefits of using a high-level synthesis (HLS) design methodology include:

+Developing and validating algorithms at the C-level for the purpose of designing at an abstract level from the hardware implementation details

+Using C-simulation to validate the design and iterate more quickly that with traditional RTL design

+Controlling the C-synthesis process using optimization pragmas to create high-performance implementations

+Creating multiple design solutions from the C source code and pragmas to explore the design space, and find an optimal solution.

+Quickly recompile the C-source to target different platforms and hardware devices.

HLS includes the following stages:

1.Scheduling determines which operations occur during each clock cycle based on:

When an operation’s dependencies have been satisfied or are available.

The length of the clock cycle or clock frequency

The time it takes for the operation to complete, as defined by the target device.

The available resource allocation

Incorporation of any user-specified optimization directives

2.Binding assigns hardware resources to implement each scheduled operation and maps operators.

3.Control logic extraction creates a finite state machine (FSM) that sequences the operations in the RTL design according to the defined schedule.

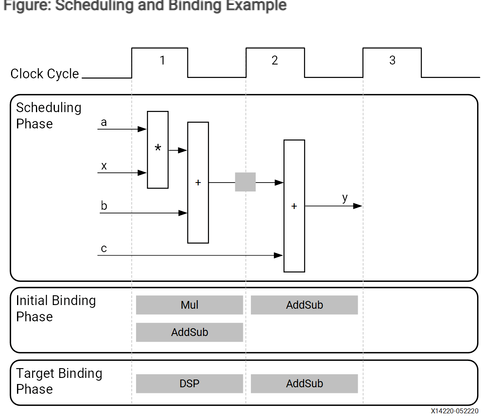
Scheduling and Binding Example <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Scheduling-and-Binding-Example>

int foo(char x, char a, char b, char c){  
char y;

y=x\*a+b+c;

return y;

}



In the final hardware implementation, high-level synthesis implements the arguments to the top-level function as input and output (I/O) ports. In this example, the arguments are simple data ports. Because each input variable is a char type, the input data ports are all 8-bits wide. The function return is a 32-bit int data type and the output data port is 32-bits wide.

Extracting Control Logic and Implementing I/O Ports Example <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Extracting-Control-Logic-and-Implementing-I/O-Ports-Example>

void foo(int in[3], char a, char b, char c, int out[3]){

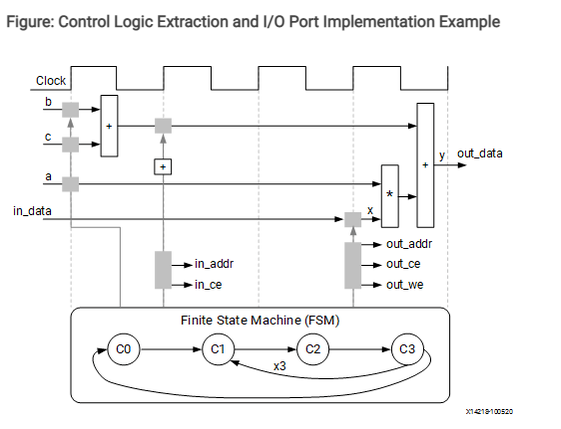
int x, y;

for(int i=0;i<3;i++){

x=in[i];

y=a\*x+b+c;

out[i]=y;  
}

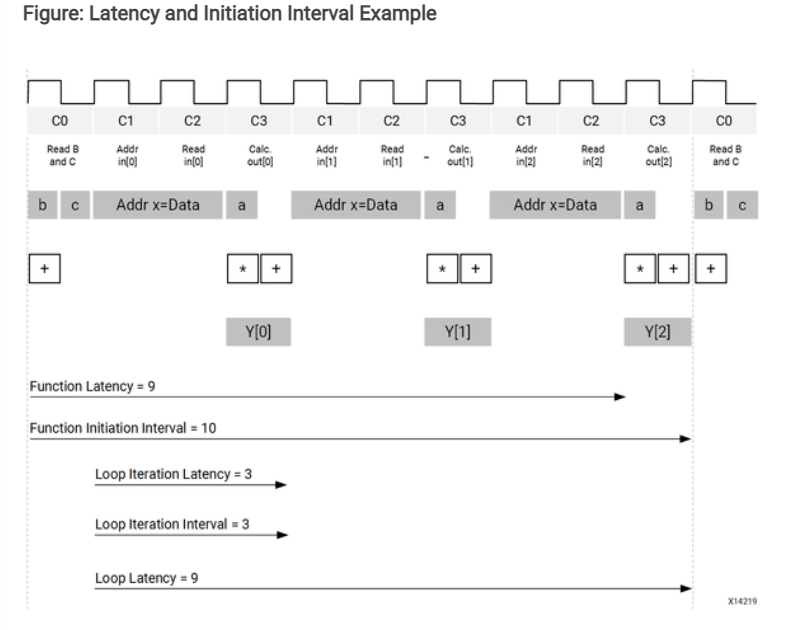


High level synthesis automatically extracts the control logic from the C code and creates an FSM in the RTL design to sequence these operations.

In high level synthesis, arrays are synthesized into block RAM by default, but other options are possible such as FiFOs, distributed RAM and individual register.

Performance Metrics Example <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Performance-Metrics-Example>

The following figure shows the complete cycle -by-cycle execution for the code in the previous example, including the states for each clock cycle, read operations, computation operations and write operations



Vitis HLS Process Overview <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Vitis-HLS-Process-Overview>

Vitis HLS is project based and can contain multiple variations called “solutions” to drive synthesis and simulation. Each solutions can target either the Vivado IP Flow or the Vitis Kernel Flow.

The following are the synthesis, analysis and optimization steps in the typical design flow.

1.Create a new Vitis HLS project.

2.Veify the source code with C simulation.

3.Run high-level synthesis to generate RTL files.

4.Analyze the results by examining latency, initiation interval (II), throughput, and resource utilization.

5.Optimize and repeat as needed.

6.Verify the results using C/RTL Co-simulation.

Enabling the Vivado IP Flow <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Enabling-the-Vivado-IP-Flow>

The flow selection is enabled with the open\_solution -flow\_target vivado

Vivado IP can support a wide variety of interface specifications and data transfer protocols, and does not naturally support the Xilinx runtime (XRT) requirements of the Vitis system. The Vivado IP flow provides much greater discretion in your design choices, however, leaves the integration and management of the IP up to you as well.

Enabling the Vitis Kernel Flow <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Enabling-the-Vitis-Kernel-Flow>

You are configurating Vitis HLS to generate the compiled kernel object (.xo) for the Vitis acceleration flow. The Vitis Kernel flow is more restrictive than the Vivado IP flow and the kernels produced by the HLS tool must meet the specific requirements of the platforms and Xilinx runtime (XRT).

When specifying open\_solution -flow\_target vitis or enabling the Vitis Kernel Flow in the IDE, Vitis HLS implements interface ports using the AXI standard

The solution is updated to include two new configuration

config\_rtl -registter\_reset\_num=3

and

config\_interface -default\_slave\_interface=s\_axilite -m\_axi\_latency=64 \

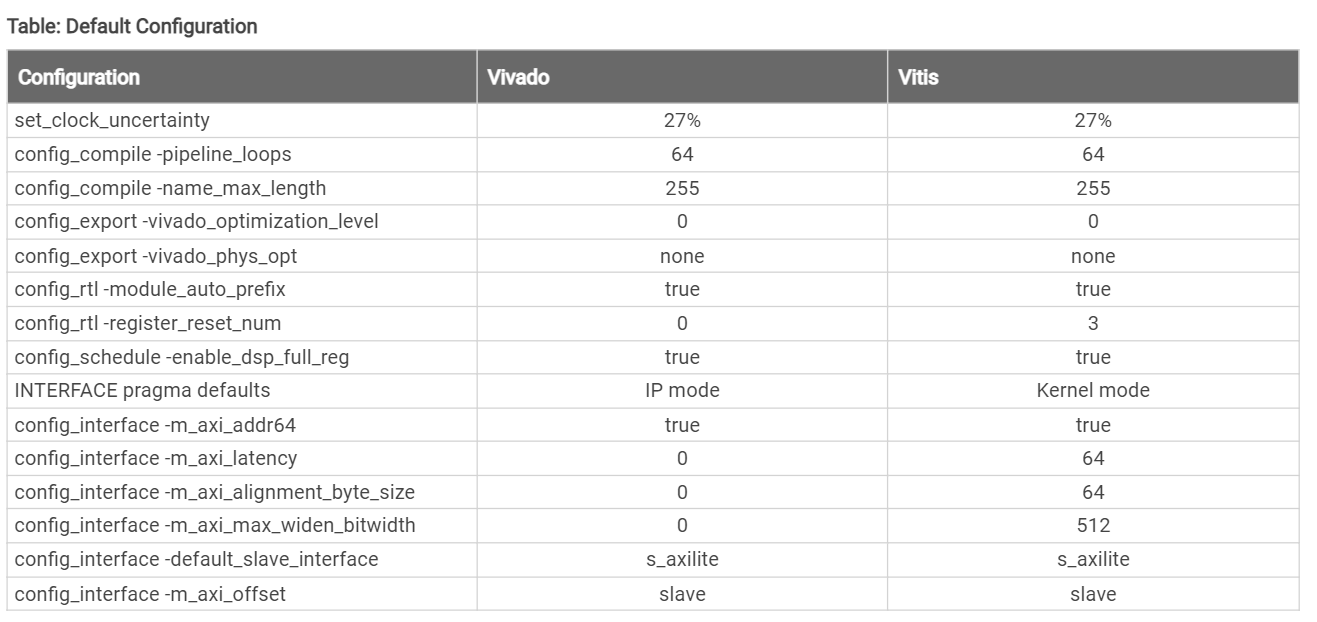
-m\_axi\_alignment\_byte\_size=64 -m\_axi\_max\_widen\_bitwidth=512 -m\_axi\_offset=slave

The config\_rtl command defines characteristics of the RTL code generated by Vitis HLS , specifically defining characteristics of the reset required by the Vitis application acceleration development flow.

The config\_interface command sets characteristics of the default interface protocols the tool assigns.

Default Settings pf Vivado/Vitis Flows <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Default-Settings-of-Vivado/Vitis-Flows>

The open\_solution target will configurate the compiler for either the Vivado IP flow or the Vitis Kernel Flow.



Launching Vitis HLS <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Launching-Vitis-HLS>

Setting Up the Environment <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Setting-Up-the-Environment?tocId=YM7vdK98YdukGQfR4AYbNg>

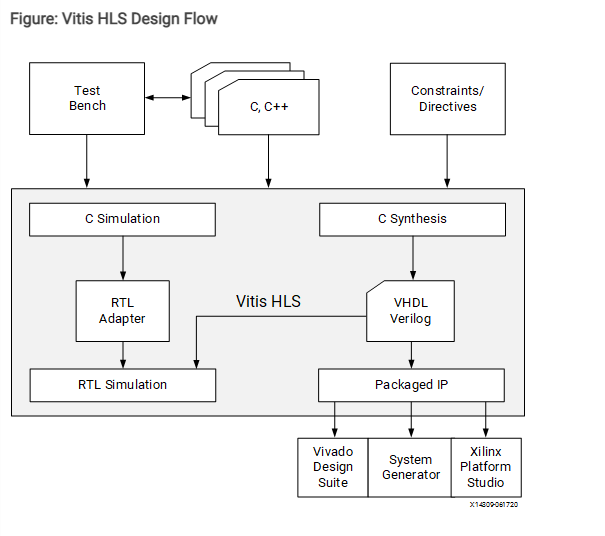
Overview of the Vitis HLS IDE <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Overview-of-the-Vitis-HLS-IDE>

Customizing the Vitis HLS IDE Behavior <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Customizing-the-Vitis-HLS-IDE-Behavior>

Creating a New Vitis HLS Project <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Creating-a-New-Vitis-HLS-Project>

Working with Sources <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Working-with-Sources>

The following figure illustrates the Vitis HLS design flow, showing the input and output files.



Vitis HLS inputs include:

C functions written in C and C++11/C++14. This is the primary input to Vitis HLS. The function can contain a hierarchy of sub-functions.

C functions with RTL blackbox

Design Constrains that specify the clock period, clock uncertainty and the device target

Directives are optional and direct the synthesis process to implement a specific behavior or optimization

C test bench and any associated files needed to simulate the C function prior to synthesis and to verify the RTL output using C/RTL Co-simulation

Coding C/C++ Functions <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Coding-C/C-Functions>

Only one function is allowed as the top-level function for synthesis

Any sub-functions in the hierarchy under the top-level function for synthesis are also synthesized

If you want to synthesize functions that are not in the hierarchy under the top-level function for synthesis, you must merge the functions into a single top-level function for synthesis

Accessing Source Files in Git Repositories <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Accessing-Source-Files-in-Git-Repositories>

Using Libraries in Vitis HLS <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Using-Libraries-in-Vitis-HLS>

Arbitrary Precision Data Types Library: Arbitrary precision data types let your C code use variables with smaller bit-widths that standard C or C++ data types, to enable improved performance and reduced are in hardware <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Arbitrary-Precision-Data-Types-Library>

Vitis HLS Math Library: Used to specify standard math operations for synthesis into RTL and implementation on Xilinx devices <https://docs.xilinx.com/r/u1ha7A~FnJAUGn1TvNNmSQ/5vFE6xWfqrzgx19vJra3IA>

HLS Stream Library: For modeling and compiling streaming data structures. <https://docs.xilinx.com/r/u1ha7A~FnJAUGn1TvNNmSQ/lHGz3jpTcYoPcmSMl0mDJQ>

<https://xilinx.github.io/Vitis_Libraries/>

<https://github.com/Xilinx/Vitis_Libraries>

Resolving Header File Problems <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Resolving-Header-File-Problems>

Resolving Comments in the Source Code <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Resolving-Comments-in-the-Source-Code>

Setting Configurations Options <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Setting-Configuration-Options>

Specifying the Clock Frequency <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Specifying-the-Clock-Frequency>

Clock and Reset Ports <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Clock-and-Reset-Ports>

Using the Flow Navigator <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Using-the-Flow-Navigator>

Verifying Code with C Simulation <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Verifying-Code-with-C-Simulation>

Write a Test Bench <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Writing-a-Test-Bench>

Example Test Bench <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Example-Test-Bench>

Design Files and Test Bench Files <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Design-Files-and-Test-Bench-Files>

Single File Test Bench and Design <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Single-File-Test-Bench-and-Design>

Using the Debug View Layout <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Using-the-Debug-View-Layout>

Output of C Simulation <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Output-of-C-Simulation>

Synthesis Control Flow <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Pre-Synthesis-Control-Flow>

Synthesizing the Code <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Synthesizing-the-Code>

Synthesis Summary <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Synthesis-Summary>

Output of C Synthesis <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Output-of-C-Synthesis>

Improving Synthesis Runtime and Capacity <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Improving-Synthesis-Runtime-and-Capacity>

Analyzing the Results of Synthesis <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Analyzing-the-Results-of-Synthesis>

Schedule Viewer <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Schedule-Viewer>

Function Call Graph Viewer <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Function-Call-Graph-Viewer>

Dataflow Viewer <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Dataflow-Viewer>

Timeline Trace Viewer <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Timeline-Trace-Viewer>

Optimizing the HLS Project <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Optimizing-the-HLS-Project>

Creating Additional Solutions <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Creating-Additional-Solutions>

Using Directive in Script vs Pragmas in Code <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Using-Directives-in-Scripts-vs.-Pragmas-in-Code>

Applying Directives to the Proper Scope <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Applying-Directives-to-the-Proper-Scope>

Applying Optimization Directives to Global Variables <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Applying-Optimization-Directives-to-Global-Variables>

Applying Optimization Directives to Class Objects <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Applying-Optimization-Directives-to-Class-Objects>

Applying Optimization Directives to Templates <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Applying-Optimization-Directives-to-Templates>

Using Constants with Pragmas <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Using-Constants-with-Pragmas>

Failure to Satisfy Optimization Directives <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Failure-to-Satisfy-Optimization-Directives>

C/RTL Co-Simulation in Vitis HLS <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/C/RTL-Co-Simulation-in-Vitis-HLS>

Output of C/RTL Co-Simulation <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Output-of-C/RTL-Co-Simulation>

Automatically Verifying the RTL <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Automatically-Verifying-the-RTL>

Interface Synthesis Requirements <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Interface-Synthesis-Requirements>

Verification of DATAFLOW and DEPENDENCE <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Verification-of-DATAFLOW-and-DEPENDENCE>

Unsupported Optimizations for Co-Simulation <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Unsupported-Optimizations-for-Co-Simulation>

Simulating IP Cores <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Simulating-IP-Cores>

Analyzing RTL Simulations <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Analyzing-RTL-Simulations>

Viewing Simulation Waveforms <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Viewing-Simulation-Waveforms>

Cosim Deadlock Viewer <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Cosim-Deadlock-Viewer>

Debuggging C/RTL Co-Simulation <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Debugging-C/RTL-Co-Simulation>

Setting Up the Environment <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Setting-Up-the-Environment?tocId=k~02cV2tmyHVdfcIfpTdYg>

Optimization Directives <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Optimization-Directives?tocId=EMPffriFrI~1pymEqb3k0g>

C Test Bench and C Source Code <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/C-Test-Bench-and-C-Source-Code>

Exporting the RTL Design <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Exporting-the-RTL-Design>

Running Implementation <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Running-Implementation>

Implementation Report <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Implementation-Report>

Output of RTL Export <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Output-of-RTL-Export>

Archiving the Project <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Archiving-the-Project>

Running Vitis HLS from the Command Line <https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Running-Vitis-HLS-from-the-Command-Line>