Tutorial for Vivado HLS

This tool converts a C/C++/System C code into an RTL code using High level synthesis (HLS). In this tutorial, we will see the C coding style, interface management, several optimizations that can be performed, and the RTL generation. C functions execute in orders of magnitude faster than RTL simulations. Using C to develop and validate the algorithm prior to synthesis is more productive than developing at RTL. C/C++ constructs to RTL mapping would be as shown below

Functions	→	**Modules**
Arguments	→	**Input/output ports**
Operators	→	**Functional units**
Scalars	→	**Wires or registers**
Arrays	→	**Memories**
Control flows	→	**Control logics**

C coding style

We will have a C code with the top-level function that needs to be synthesized, a header file, and a C test bench file. The test bench is used to validate the behavior of the top-level function to be synthesized. Generally, it is good design practice to separate the top-level function for synthesis from the test bench and to make use of header files. The following code shows a sample design that calls a function.

```
#include "hier_func.h"
int sumsub_func(din_t *in1, din_t *in2, dint_t *outSum, dint_t *outSub)
{
*outSum = *in1 + *in2;
*outSub = *in1 - *in2;
}
void hier_func(din_t A, din_t B, dout_t *C, dout_t *D)
{
dint_t apb, amb;
sumsub_func(&A,&B,C,D);
}
```

The types din_t, dint_t and dout_t are defined using in the header file shown below. typedef can make the code more portable and readable.

```
#ifndef _HIER_FUNC_H_
#define _HIER_FUNC_H_
#include <stdio.h>
#define NUM_TRANS 40
typedef int din_t;
typedef int dint_t;
typedef int dout_t;
```

```
void hier_func(din_t A, din_t B, dout_t *C, dout_t *D);
#endif
```

The first step in the synthesis of any block is to validate that the C function is correct. This is performed by the test bench. The key to taking advantage of C development times is to have a test bench that checks the results of the function against known good results. This allows any code changes to be validated before synthesis. Vivado HLS can re-use the C test bench to verify the RTL design (no RTL test bench needs to be created when using Vivado HLS). An example of a test bench is shown below.

```
#include "hier_func.h"
int main() {
// Data storage
int a[NUM_TRANS], b[NUM_TRANS];
int c_expected[NUM_TRANS], d_expected[NUM_TRANS];
int c[NUM_TRANS], d[NUM_TRANS];
//Function data (to/from function)
int a_actual, b_actual;
int c_actual, d_actual;
int retval=0, i, i_trans, tmp;
for (i=0; i<NUM_TRANS; i++){
a[i] = 1;
}
for (i=0; i<NUM_TRANS; i++){
b[i] = 1;
}
// Execute the function multiple times (multiple transactions)
for(i_trans=0; i_trans<NUM_TRANS-1; i_trans++){
//Apply next data values
a_actual = a[i_trans];
b_actual = b[i_trans];
hier_func(a_actual, b_actual, &c_actual, &d_actual);
c[i_trans] = c_actual;
d[i_trans] = d_actual;
}
for (i=0; i<NUM_TRANS; i++){
c_expected[i] = 2;
}
for (i=0; i<NUM_TRANS; i++){
d_expected[i] = 0;
}
// Check outputs against expected
for (i = 0; i < NUM_TRANS-1; ++i) {
if(c[i] != c_expected[i]){
retval = 1;
}
if(d[i] != d_expected[i]){
retval = 1;
}
}
// Print Results
if(retval == 0){
printf(" Results are good \n");
} else {
```

```
printf(" Mismatch: retval=%d \n", retval);
}
// Return 0 if outputs are correct
return retval;
}
```

Unsupported C Language Constructs

In order to be synthesized, the C function must contain the entire functionality of the design. System calls such as printf() or fprintf() cannot feature in the hardware design and so cannot be synthesized. Pointer casting or using the recursive functions is not supported.

Any system calls which manage memory allocation within the system, for example malloc(), alloc() and free(), must be removed from the design code prior to synthesis. The reason for this is that they are implying resources which are created and released during runtime: to be able to synthesize a hardware implementation the design must be fully self-contained, specifying all required resources.

Arbitrary Precision Types with C

To use arbitrary precision integer data types in a C function,
• Add header file "ap_cint.h" to the source code. (Vivado also supports adding external libraries)
• Change the bit types to intN or uintN, where N is a bit-size from 1 to 1024.
• Compile using the apcc compiler
° Select the Use APCC for Compiling C Compiler option in the GUI.
° Use apcc in place of gcc at the command prompt

The apcc compiler can be enabled in the project setting using menu Project > Project Settings > Simulation and select Use APCC for Compiling C Files.

The following example shows how the header file is added and the two variables are implemented to use 9-bit integer and 10-bit unsigned integer types:

```
#include ap_cint.h
void foo_top (…) {
int9 var1; // 9-bit
uint10 var2; // 10-bit unsigned
```

Multiple access pointer interfaces:

When multi-access pointers are used, the following must be performed:
• It is a requirement to use the volatile qualifier.
• Specify the number of accesses on the port interface if verifying the RTL with cosim_design. You can specify it as depth to a port using a directive command shown below: top_design is your top function and port is your port name. Depth is 32 here.

set_directive_interface -mode ap_fifo -depth 32 "top_design" port

The following example shows the use of volatile, and will ensure that it will read 4 unique values from the test bench.

```
#include "fifo.h"
void fifo (volatile int *d_o, volatile int *d_i) {
static int acc = 0;
int cnt;
acc += *d_i;
acc += *d_i;
*d_o = acc;
acc += *d_i;
acc += *d_i;
*d_o = acc;
}
```

Function Optimization

Function in-lining: There is typically a cycle overhead to enter and exit functions and removing the function hierarchy can mean improved latency and throughput. Function in-lining can be used to remove function hierarchy, often at the expense of area.

Options:

-region - All functions in the specified region are to be in-lined.

-recursive - By default only one level of function in-lining is performed: the functions within the specified function are not in-lined. The **-recursive** option in-lines all functions recursively down the hierarchy.

-off - This disables function in-lining and is used to prevent particular functions from being in-lined. For example, if the **-recursive** option is used in a caller function, this option can prevent a particular callee function from being in-lined when all others are in-lined.

The following commands can be applied to the top level function foo_top. We can prevent foo_sub from being in-lined using –off.

```
set_directive_inline –region -recursive foo_top
set_directive_inline -off foo_sub
```

Pragma

The pragma should be placed in the C source within the boundaries of the required location. The pragma needs to be written in the C-code while all the directives can be written in your directives file. The pragma equivalent of the above 2 directive commands can be written as follows

```
#pragma AP inline region recursive
#pragma AP inline off
```

You need to write these two pragmas in the top-level function and in the foo_sub function respectively.

Function instantiation:

By default
• Functions remain as separate hierarchy blocks in the RTL.
• All instances of a function, at the same level of hierarchy, will use the same RTL implementation (block).
The **set_directive_function_instantiate** command is used to create a unique RTL implementation for each instance of a function, allowing each instance to be optimized. By default, the following code would result in a single RTL implementation of function foo_sub for all three instances.

```
char foo_sub(char inval, char incr)
{
return inval + incr;
}
void foo(char inval1, char inval2, char inval3,
char *outval1, char *outval2, char * outval3)
{
*outval1 = foo_sub(inval1, 1);
*outval2 = foo_sub(inval2, 2);
*outval3 = foo_sub(inval3, 3);
}
```

For the example code shown above, the following Tcl (or pragma placed in function foo_sub) allows each instance of function foo_sub to be independently optimized with respect to input **incr.**

```
set_directive_function_instantiate incr foo_sub
#pragma AP function_instantiate variable=incr
```

Function dataflow Pipelining: Dataflow pipelining takes a sequential functional description and creates a parallel process architecture from it. Dataflow pipelining is a very powerful method for improving design throughput.

-interval: An integer value specifying the desired initiation interval (II): the number of cycles between the first function or loop executing and the start of execution of the next function or loop.

In the following command, dataflow is specified in function My_Func, with a target initiation interval of 3.

```
set_directive_dataflow -interval 3 My_func
#pragma AP dataflow interval=3
```

Function pipelining: Where dataflow pipelining allows the optimization of the communication between functions to improve throughput, function pipelining optimizes the operations within a function and has a similarly positive effect on throughput. The following command (tcl or pragma) will pipeline the function “foo” with II as1.

Tcl command: `set_directive_pipeline foo`

```
#pragma AP pipeline
```

-enable_flush - This option implements a pipeline that can flush pipeline stages if the input of the pipeline stalls. This feature implements additional control logic, has greater area and is optional.

Here, loop loop_1 in function foo is pipelined with an initiation interval of 4 and pipelining flush is enabled.

```
set_directive_pipeline -II 4 -enable_flush foo/loop_1
#pragma AP pipeline II=4 enable_flush
```

Loop optimizations

Unrolling: Unroll for-loops to create multiple independent operations rather than a single collection of operations. We can specify the unrolling factor. Here it is 2. This command unrolls the loop L1 (you can name the loops) in a function foo. You can write this pragma inside loop L1.

Tcl command: `set_directive_unroll -skip_exit_check -factor 4 foo/L1`

```
#pragma AP unroll skip_exit_check factor=4
```

Merging: When there are multiple sequential loops this can sometimes create additional unnecessary clock cycles. So, we merge those loops. This example merges all consecutive loops in function foo into a single loop.

```
set_directive_loop_merge foo
#pragma AP loop_merge
```

Flattening nested loops: It requires additional clock cycles to move between rolled nested loops. It requires one clock cycle to move from an outer loop to an inner loop and from an inner loop to an outer loop. The inner loop L1, which has the body, and the foo function are specified.

Tcl command: `set_directive_loop_flatten foo/L1`

```
#pragma AP loop_flatten
```

Loop dataflow pipelining: Dataflow pipelining can be applied to loops in similar manner as it can be applied to functions. It allows loops which are sequential in nature to operate concurrently at the RTL. The –interval option can be used to specify exactly how many cycles that can allowed between the start of one loop implementation and the next. Here “foo” is the functions that has loops.

Tcl command: `set_directive_dataflow -interval 2 foo`

```
#pragma AP dataflow interval=2
```

Array Optimizations

Arrays in a C language description are typically mapped to memories and so the optimizations performed on arrays have a great impact on both area and performance.

Read and write -> RAM, constant array -> ROM

Horizontal mapping: Two arrays arrray1 and array2 concatenated into array3

```
set_directive_array_map -instance array3 -mode horizontal top array1
set_directive_array_map -instance array3 -mode horizontal top array2

#pragma AP array_map variable=array1 instance=array3 horizontal
#pragma AP array_map variable=array2 instance=array3 horizontal
```

Vertical mapping:

```
set_directive_array_map -instance array3 -mode vertical top array2
set_directive_array_map -instance array3 -mode vertical top array1

#pragma AP array_map variable=array1 instance=array3 vertical
#pragma AP array_map variable=array2 instance=array3 vertical
```

Array partitioning: Partitions an array into smaller arrays. This increases throughput.

The following command (the equivalent pragma is also shown) partitions array AB[13] in function foo into four arrays. Because 4 is not an integer multiple of 13, three of the arrays will have 3 elements and one will have 4 (containing elements AB[9:12]).

```
set_directive_array_partition -type block -factor 4 foo AB
#pragma AP array_partition variable=AB block factor=4
```

Specifying a core to be used for a variable:

We can specify the type of RAM which supplies data to the port. RAM_1P must have more than 16 elements (addresses) and each element must be greater than 8-bits. RAM_2P must have more than 28 elements and each element must be more than 12-bits.

In this example, variable coeffs[128] is an argument to top-level function foo_top. This directive specifies coeffs be implemented with core RAM_1P from the library. The ports created in the RTL to access the values of coeffs, will be those defined in the core RAM_1P.

```
set_directive_resource -core RAM_1P foo_top coeffs
#pragma AP resource variable=coeffs core=RAM_1P
```

Interface Management

We can specify the interface behavior explicitly in the input source code. This allows any arbitrary IO protocol to be used, hence allows the function to interface with any hardware resource.

Interface types:

Standard port level interface synthesis is specified by applying the appropriate interface mode to a function argument. A function argument which is both read from and written to (an RTL inout port) is synthesized in the following manner

• For interface types ap_none, ap_stable, ap_ack, ap_vld, ap_ovld and ap_hs, separate input and output ports are created. For example, if function argument arg1 was both read from and written to, it would be synthesized as RTL input data port arg1_i and output data port arg1_o and any specified or default IO protocol is applied to each port individually.

• For interface types ap_memory and ap_bus, a single interface is created. Both these RTL interfaces support read and write.

• For interface type ap_fifo, read and write are not supported for ap_fifo interfaces. Only streaming is allowed.

ap_hs (ap_ack, ap_vld and ap_ovld)

An ap_hs interface provides both an acknowledge signal to say when data has been consumed and a valid signal to indicate when data has been read. This interface type is a superset of types ap_ack, ap_vld and ap_ovld

• Interface type ap_ack only provides an acknowledge signal.

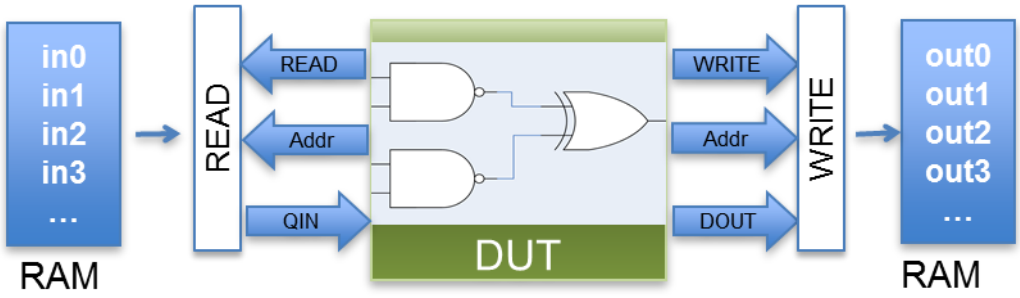
• Interface type ap_vld only provides a valid signal.

• Interface type ap_ovld only provides a valid signal and only applies to output ports or the output half of an inout pair.

ap_memory

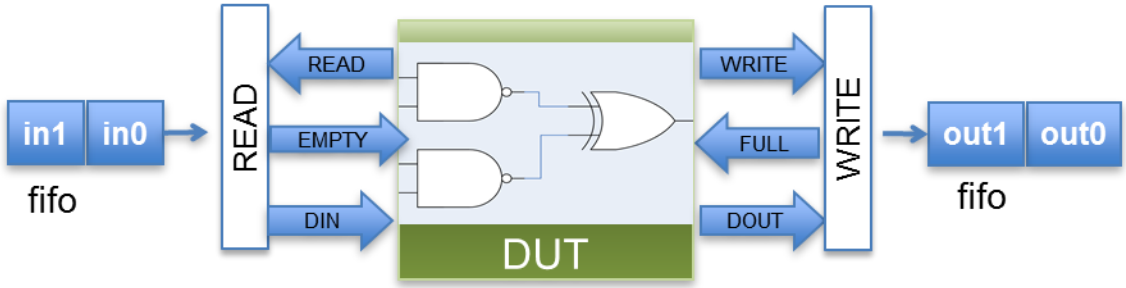
Array arguments are typically implemented using the ap_memory interface. This type of port interface is used to communicate with memory elements (RAMs, ROMs) when the implementation requires random accesses to the memory address locations. Array arguments are the only arguments which support a random access memory interface.

Command: **set_interface -type memory -port {in out}**



ap_fifo: If access to a memory element is required and the access is only ever performed in a sequential manner (no random access) an ap_fifo interface is the most hardware efficient.

Command: **set_interface -type fifo -port {in out}**




```
void foo(int* in1, ...) {
int data1, data2, data3;
...
data1= *in1;
data2= *(in1+2);
data3= *(in1-1);
...
}
```

In the above example, if "in1" is specified as an ap_fifo interface, High-Level Synthesis will check the accesses and determine the accesses are not in sequential order, and issues an error and halt. To read from non-sequential address locations use an ap_memory interface as this random accessed or use an ap_bus interface.

ap_bus

An ap_bus interface can be used to communicate with a bus bridge. The interface does not adhere to any specific bus standard but is generic enough to be used with a bus bridge which in-turn arbitrates with the system bus. The bus bridge must be able to cache all burst writes.

Controlling Interface Synthesis

Interface synthesis is controlled by the INTERFACE directive or by using a configuration setting. Configuration settings can be used to specify the default operation for creating RTL ports and interfaces. The INTERFACE directive is used to specify the explicit interface type of a particular port and overrides any default or global configuration. Configuration settings can be used to create RTL ports for any global variables and to set a default interface type for all ports of the specified type. The following Tcl command sets the interface type on all input ports to type ap_vld and the default type on all output ports to type ap_memory.

```
config_interface -mode in ap_vld
config_interface -mode out ap_memory
```

Specifying port interface: Here, argument InData in function `foo` is specified to have a **ap_vld** interface.

```
set_directive_interface -mode ap_vld -register foo InData
#pragma AP interface ap_vld register port=InData
```

Specifying bus interfaces

In addition to the standard interfaces explained in the Interface Synthesis section, Vivado HLS can also automatically add bus interfaces to the RTL design. The primary difference between bus interface ports and the RTL ports created by interface synthesis (ap_none, ap_hs, etc) is that the bus interfaces are added to the design during the Export RTL process.

- Bus interfaces are not reported in the synthesis reports.
- Bus interfaces are not present in the RTL written after synthesis.

We can group several ports into a common bus interface. Another important aspect of bus interfaces is that the type of bus interface which can be used depends on the protocol of RTL port (ap_none, ap_hs, ap_bus etc). Each type of RTL interface can only be connected to certain bus interfaces. For example, an AXI4 Stream bus interface can only be added to ports of type ap_fifo.

Eg: The following example shows how multiple RTL ports are bundled into a common AXI4 slave interface, allowing multiple RTL ports to be accessed through a single bus interface

```
int foo_top (int *a, int *b, int *c, int *d) { // Define the RTL interfaces
#pragma AP interface ap_hs port=a
#pragma AP interface ap_none port=b
#pragma AP interface ap_vld port=c
#pragma AP interface ap_ack port=d
#pragma AP interface ap_ctrl_hs port=return register
// Define the pcore interfaces and group into AXI4 slave “slv0”
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=a
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv0" variable=b
// Define the pcore interfaces and group into AXI4 slave “slv1”
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=return
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=c
#pragma AP resource core=AXI4LiteS metadata="-bus_bundle slv1" variable=d
*a += *b;
return (*c + *d);

}
```

Now, let us see how to run generate an RTL file from our C/C++ design file.

First, access the Linux desktop of Linux lab. You can work in GUI by typing *vivado_hls* in the terminal. In this tutorial, we will see how we can work with the command lines from the terminal.

Step 1: Setting up the environment for simulation

- Copy the content of ".bashrc.cadence" file from http://aimlab.seas.wustl.edu/courses/.bashrc.cadence into an empty file in your *home directory* and rename the file to ".bashrc.cadence"
- In your terminal, type as follows from home directory
  "source .bashrc.cadence"
- You should observe that the *environment is loaded*.

Step 2: Create a project folder and write the following files in it.

- You need a main .cpp file, a test bench file (.cpp), a header file, a run_hls.tcl file, and directives.tcl file.
- We considered the example of optical flow that uses Lucas Kanade algorithm. You can find the codes in this link: https://drive.google.com/folderview?id=0B5QS328V9qFHfldQYmF5a3BQeUhHZTlsRHloRVVJU0RJVGIyNmFGSTMwaGlfWmd3TjZRY2c&usp=sharing

  Step 3: Let's see the details of run_hls.tcl
  This file is generally written as follows:

```
##create the project
open_project optical_new
## set the top level function of your c-code
set_top LucasKanade8_XY
## add design files
add_files LK.c
## add test bench file
add_files -tb optical_tb.c
## create a solution
open_solution "solution1"
## define the technology and the clock rate(ns) here
set_part {xq7z045rf900-2i}
create_clock -period 20 -name default
##  source the directives file.
source "./directives.tcl"
## run the set up. Validating the c code
csim_design -setup
## synthesizing C design into an RTL design
```

```
csynth_design
## cosimulation
cosim_design
## exporting the RTL for future use
export_design -format ip_catalog
exit
```

The directives file can be written as follows:

```
set_directive_interface -mode ap_fifo -depth 32 "LucasKanade8_XY" p_process
set_directive_interface -mode ap_memory -depth 32 "LucasKanade8_XY" ci
set_directive_interface -mode ap_memory -depth 32 "LucasKanade8_XY" li
set_directive_unroll -factor 2 "LucasKanade8_XY/loop1"
set_directive_loop_flatten "LucasKanade8_XY/loop6"
```

In cosimulation, RTL verification is done with the re-use of test bench. Finally, we export the RTL. You can also specify the optimization commands in the directives.tcl file.

Step 4: Type the following command after going to the project directory.

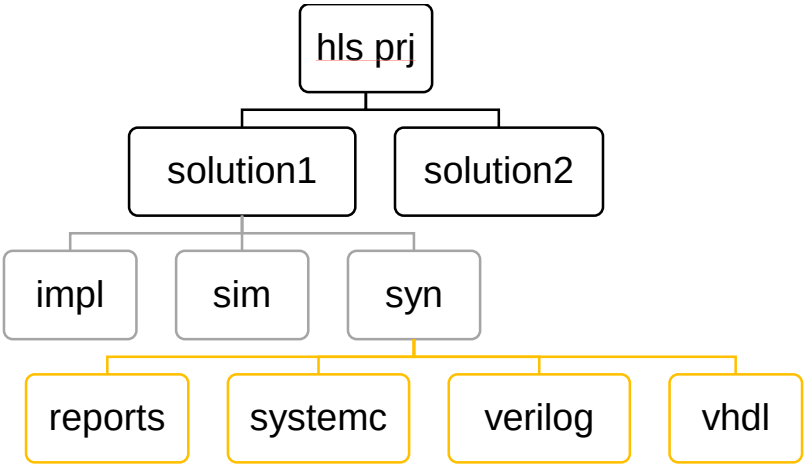
```
vivado_hls -f run_hls.tcl
```

This completes the process. You should also see whether the co-simulation has passed or not, and if the RTL has been exported or not.

You can also type the following command to invoke interactive mode, in which you can type Tcl commands one at a time.

```
vivado_hls -i
```

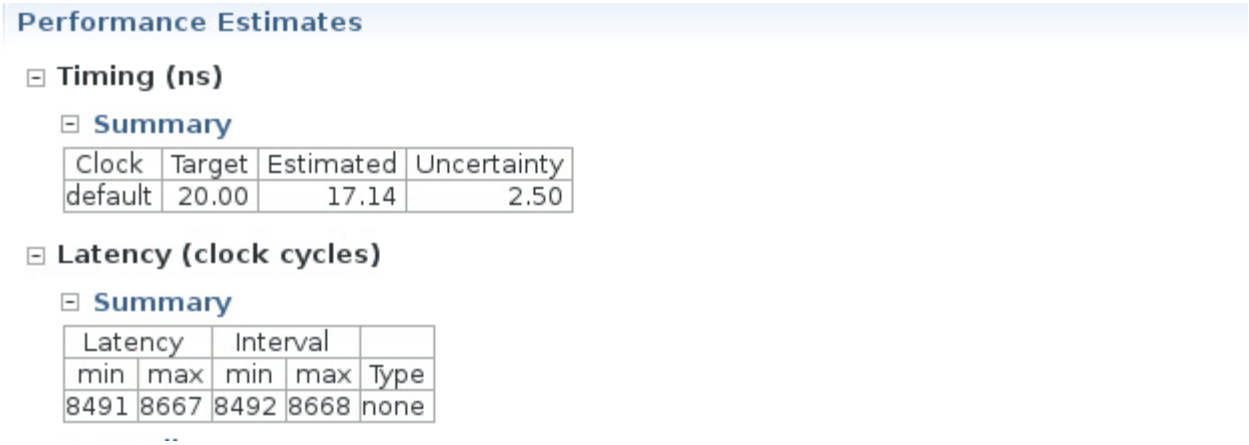
The synthesis directory structure would be as shown below.



The final output RTL design should be available in proj_main\solution1\syn\Verilog or VHDL. You can go with the file named with your top-level function. If you see a lot of other files named with dsp along with the top-level function, it says how heavily you want to use the DSP slices on the

FPGA. Also if there is any dat file named after a variable, it means that a ROM has been created.
The synthesis report would be available in syn\report directory.

After performing the optimizations, as shown in directives file, the latency has decreased. The following figure shows the latency part of the report.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	20.00	17.14	2.50

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
8491	8667	8492	8668	none