IBM CAPI SNAP framework

Version 1.0

How to Port and Optimize a C/C++ code in SNAP environment.

The Guide describes the process to port, evaluate and optimize an existing C code to a FPGA in SNAP environment. This will also detail the use of the Vivado HLS tools.

To understand without any ambiguity all the explanation provided in this document, let's recall the basics:

- an **application** is the program running on the host server and executed on a **CPU**
- an **action** is the program running on the **FPGA**

Overview

Let's imagine that you have a code written in C / C++ and want to port it in a FPGA to offload it and even accelerate it. You will need to start from a code which is functionally working and from which results are known so that we can confirm the results at every stage of the porting.

Multiple ways can be used to port a code, so let's choose to describe here the **bottom-up** method which allows the user to know very quickly if it's worth spending time to port an algorithm by evaluating its performance as soon as possible in the process.

Out of this first experience, you should be able to create any new action from an existing code understanding where the parameters are but also discover how to use the different tools.

This document has been built using snap git release tag v1.4.2. It can certainly work also for newer releases.

This document will successfully go through the following items:
- Simulate and synthesize user code
- Evaluate action code performance and optimize code (optional)
- Insert SNAP interface in the action code
- write the application code to work with

Useful documentation:

General documents can be found in ***~snap/doc***. It is highly recommended to start with the User Guide ***UG_CAPI_SNAP-QuickStart_on_a_General_Environment.pdf***, and then refer to the Application Notes "AN_"to get answers to specific questions. You can also find for every example, their user guide in the ***~snap/actions/hls_xxx/doc*** directory.

Document history

Rev.	Date	Changes
V1.0	May 23rd , 2018	Initial release

Contents

1. Pre-requisites

Development environment:

You need to have Vivado greater or equal to 2017.4 and Vivado HLS greater or equal to 2017.4 installed with associated licenses.

Vivado HLS (High Level Synthesis) is the first tool we are going to use. It takes C or C++ code, compiles it, optimizes it by trying to parallelize the code and generates VHDL, Verilog or SystemC code which is a low-level language understood by the FPGA synthesizers.

Vivado is then used to create the binary files from the Verilog or VHDL code. This binary will be then downloaded into the FPGA to configure it and have your algorithm work.

This can be executed only on a x86 environment due to Vivado tool not running on POWER systems. We use Ubuntu OS.

Deployment environment:

You need to have Ubuntu OS on a POWER8 or POWER9 environment and a FPGA card plugged in a "CAPI enabled" PCIe slot.

C/C++ Code:

Get a tested C program with its testbench. This will ensure that every time you change something in the code, or at every step of the process, you keep having your code functional.

New Action created:

You have the choice to start by the option you prefer, but we will start with Option 1 in this document.

1. Start porting your code in a standalone directory to estimate the performances of the algorithm in a FPGA, and then insert your code into a new SNAP action, or
2. Start creating a new action and then work in this structured environment.

2. Synthesize the code to port into the FPGA

2.1 Download a code and execute it as-is.

Whatever the option you have chosen, the first steps will be the same.

Let's use as an example a SHA3 example taken from the web. This example is a compute-only program which calculate the number of time a "keccak" function is processed by second on the system it is executed on.

This program doesn't use external data but use some values to parametrize the function called and return the key processed. This will simplify the way to explain the entire process. Adding access to external resources can then been done easily by looking to the different examples provided in library.

Let's first download, compile and execute this program as it is provided.

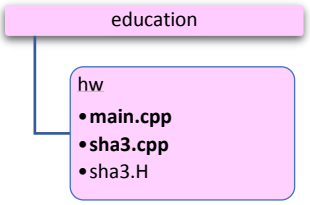
```
cd ~
mkdir education
cd education
git clone https://github.com/mjosaarinen/tiny_sha3
cd tiny_sha3
make
./sha3test
```

```
$ ./sha3test
FIPS 202 / SHA3, SHAKE128, SHAKE256 Self-Tests OK!
(1EB5D09736B9FBC5) 651465.798 Keccak-p[1600,24] / Second.
```

2.2 Prepare the action in the Vivado HLS environment and the way to test it.

Let's keep the **education/tiny_sha3** directory as a reference and copy all **c** and **h** files to a new directory. As we are working the action, then the program will be copied in for example **education/hw** directory.

If you have chosen to create a new action first, then copy all files to the **education/hw** directory you have created.



The main.cpp program prepare the data, and the sha3.cpp contains the different algorithms used.

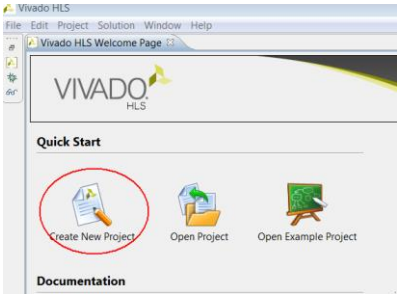
```
cd education
mkdir hw
cd hw
cp ../tiny_sha3/*.c
cp ../tiny_sha3/*.h
mv main.c main.cpp
mv sha3.c sha3.cpp
mv sha3.h sha3.H
```

We rename the **c** files to **cpp** even if this is only C code to get more functionalities from Vivado HLS.

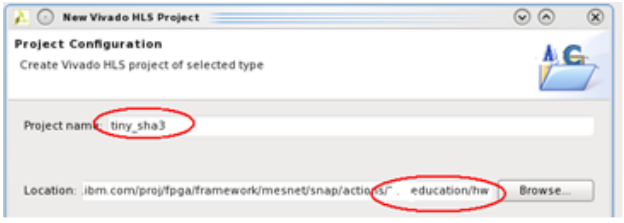
Start Vivado HLS GUI by typing:

```
vivado_hls
```

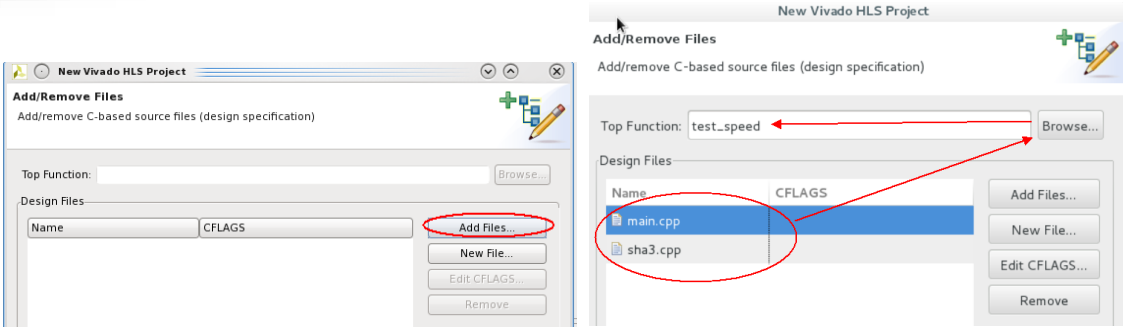
Create a new Project



Type a project name under **education/hw**



Add files, select the "**cpp"** files only and use **Browse** to find and select the function to test.



You will get the list of all the distinct functions available in the "**cpp"** files you have added with the caller function at the bottom of the list.

The **main** program calls 3 functions: "**test_sha3"**, "**test_shake"** and "**test_speed".**

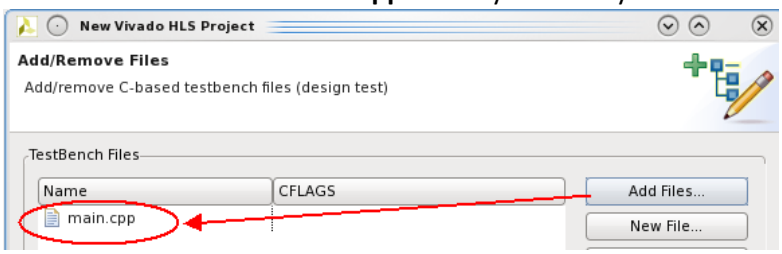
```
int main(int argc, char **argv)
{
    if (test_sha3() == 0 && test_shake() == 0)
        printf("FIPS 202 / SHA3, SHAKE128, SHAKE256 Self-Tests OK!\n");
    test_speed();

    return 0;
}
```

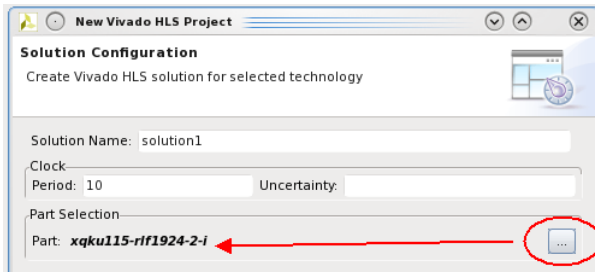
Let's select the "**test_speed"** function as the top function that we want to put in the FPGA since this is the core of the benchmark. We don't choose the **main** program since we will keep the **main** as the caller function which will be used as the "testbench", or later as the "application" executed on the server/CPU.

Now let's add the testbench that will be used to test the program at any time.

Add files and select the **main.cpp** file only since only this file will contain the testbench.

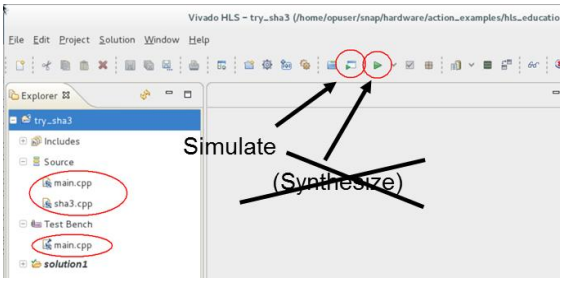


Then specify the clock period to 4 (period is specified in ns) and the FPGA you will use. If you have no idea at this stage, no worry, set the value to any of them. It will be set by the snap environment by scripts later.



The configuration of the environment is now completed.

Let's run the simulator, by pressing the simulate button and keep options proposed unchanged



This will give you the following error in the console window at the bottom of the screen

```
3    Compiling ../../../../main.cpp in debug mode
4 ../../../../main.cpp:7:18: fatal error: sha3.h: No such file or directory
5 compilation terminated.
6 make: *** [obj/main.o] Error 1
7 CRITICAL WARNING: [SIM 100] 'csim_design' failed: compilation error(s).
8 INFO: [SIM 3] *************** CSIM finish ***************
9
```

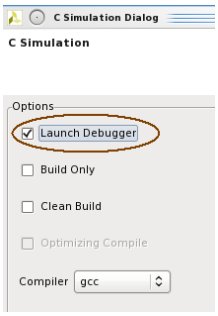
Correct the call of the header files extension ".h" → ".H" in the cpp files and re-rerun the simulator.

```
1 INFO: [SIM 2] *************** CSIM start ***************
2 INFO: [SIM 4] CSIM will launch GCC as the compiler.
3    Compiling ../../../../main.cpp in debug mode
4    Generating csim.exe
5 FIPS 202 / SHA3, SHAKE128, SHAKE256 Self-Tests OK!
6 (7ADBD935BE50D2D1) 86455.331 Keccak-p[1600,24] / Second.
7 INFO: [SIM 1] CSim done with 0 errors.
8 INFO: [SIM 3] *************** CSIM finish ***************
9
```

This step is now completed.

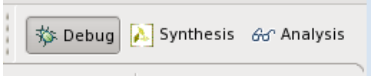
In this step, you have just setup the environment and confirmed that the program is running in the same condition than in a basic terminal.

You can also try and run this simulator with the debugger option selected



This will provide you a standard debugger that can help you going into your code.

You will notice that on top right of your code, the view has changed to the Debug view. Going back to the previous view can be done selecting the Synthesis view (and stopping the debugger simulation)



The synthesis step we are now going to start is the key point. In this step, we are going to compile the C code targeting a FPGA instead of a CPU. The main difference is that a FPGA has no Operating System and thus some obvious “features” available on a standard CPU are sometimes not implemented in a FPGA. As an example, the dynamic memory allocation doesn’t exist in a FPGA. Working with fix allocation will circumvent the issue.

To start your first synthesis, you have 2 options: start synthesizing the whole project or have a bottom-up approach. Let’s start with this second approach to progress an easier way.

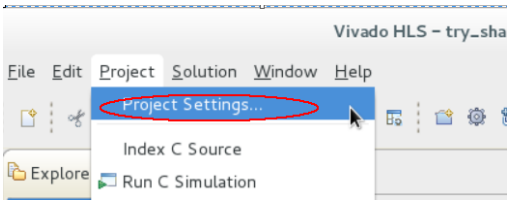
Analyzing quickly the code, we can see that the order of the calls is the following:

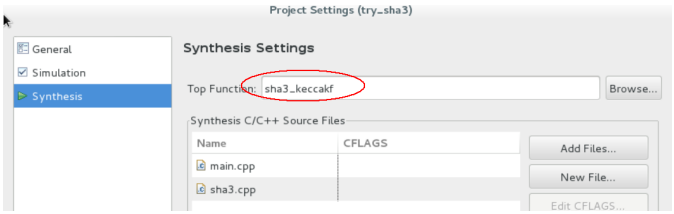
Main ➔ test_speed ➔ sha3_keccakf

2.3.1 Lowest level function synthesis

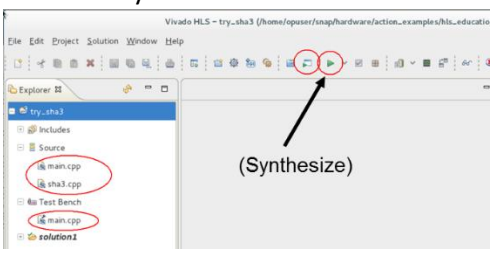
Let’s start synthesizing the **sha3_keccakf** function only

In the menu, select Project > Project Settings.
Then in the Synthesis tab, select Browse and the **sha3_keccakf** function

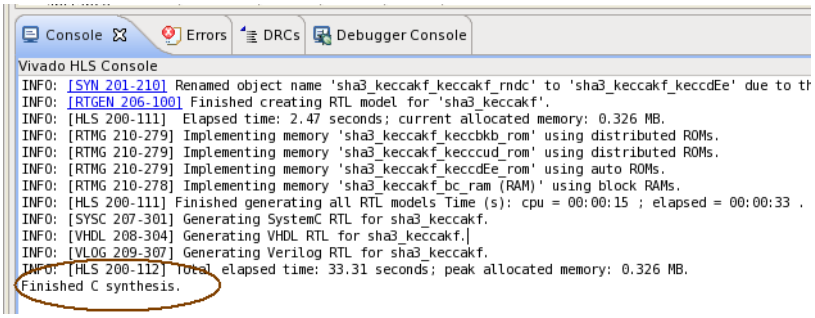




Press the Synthesis button

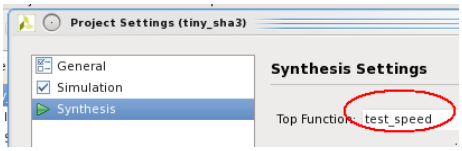


The result shows that the **synthesis is ok**



2.3.2 Upper level function synthesis

Now let's try to synthesize the upper level "**test_speed"** function.
In the menu, select Project > Project Settings. Then in the Synthesis tab, select Browse and the "**test_speed"** function.



Press the **synthesis** button.

The synthesis will fail and the console will display the following error:

```
ERROR: [SYNCHK 200-72] main.cpp:168: unsupported c/c++ library function 'clock'.
WARNING: [SYNCHK 200-77] The top function 'test_speed' (main.cpp:159) has no outputs.
INFO: [SYNCHK 200-10] 1 error(s), 1 warning(s).
ERROR: [HLS 200-70] Synthesizability check failed.
```

This is due to a **system call** in the "**test_speed"** function that is not recognized/supported by the FPGA.

```
159 void test_speed()
160 {
161     int i;
162     uint64_t st[25], x, n;
163     clock_t bg, us;
164
165     for (i = 0; i < 25; i++)
166         st[i] = i;
167
168     bg = clock();
169     n = 0;
170     do {
171         for (i = 0; i < 100000; i++)
172             sha3_keccakf(st);
173         n += i;
174         us = clock() - bg;
175     } while (us < 3 * CLOCKS_PER_SEC);
```

Analyzing the program, we can see that the **system call** is just used to calculate the number of time the **sha3_keccakf** function will be run. As the goal of this test is to display the number of time per seconds a function is called, we can call it any number of time and just divide the result by the final time.

Let's just remove these lines calling the **clock system call**. You will notice that the "printf" of the "**test_speed"** function is now not useable anymore since we removed the system time calls. We will later add a loop to replace it.

```
159 void test_speed()
160 {
161     int i;
162     uint64_t st[25], x, n;
163     //clock_t bg, us;
164
165     for (i = 0; i < 25; i++)
166         st[i] = i;
167
168     // bg = clock();
169     n = 0;
170     // do {
171         for (i = 0; i < 100000; i++)
172             sha3_keccakf(st);
173         n += i;
174     //     us = clock() - bg;
175     // } while (us < 3 * CLOCKS_PER_SEC);
176
177     x = 0;
178     for (i = 0; i < 25; i++)
179         x += st[i];
180
181     //printf("(%016lX) %.3f Keccak-p[1600,24] / Second.\n",
182     //     (unsigned long) x, (CLOCKS_PER_SEC * ((double) n)) / ((double) us));
183
184
185 }
```

Let's press the synthesis button, and the result will now report a successful synthesis of the function "**test_speed"**.

2.3.3 Summary – cleaning code

Let's summarize where we are at this moment:

- The main program (calling test_speed, test_shake, test_sha3) is executed ok (HLS simulation mode)
- ***test_speed (will recursively call the key calculation routines)*** function was fully synthesized.
- **test_shake, test_sha3 (one shot key calculations)** and have not been ported / synthesized yet

Now let's clean our work to go further:

- in a first approach, let's comment the call of the **test_shake, test_sha3** in the main program
- To check that result is correct, **test_speed** should send back the result which was displayed with the "printf" so that a testbench could automatically check that correct result is always returned ok
- As all data are generated in the **test_speed** function, let's just add the timing loop we have removed around this **test_speed** function so that we can parallelize the calls.

```
157 // test speed of the comp
158 #define NB_ROUNDS 100000
159 uint64_t test_speed()
160 {
161     int i;
162     uint64_t st[25], x, n;
163     //clock_t bg, us;
164 
165     for (i = 0; i < 25; i++)
166         st[i] = i;
167 
168     // bg = clock();
169     n = 0;
170     // do {
171         for (i = 0; i < NB_ROUNDS; i++)
172             sha3_keccakf(st);
173         n += i;
174     //    us = clock() - bg;
175     // } while (us < 3 * CLOCKS_PER_SEC);
176 
177     x = 0;
178     for (i = 0; i < 25; i++)
179         x += st[i];
180 
181     // return checksum to upper level
182     return x;
183     //printf("(%016lX) %.3f Keccak-p[1600,24] / Second.\n",
184     //    (unsigned long) x, (CLOCKS_PER_SEC * ((double) n)) / ((double) us));
185 
186 }
```

```
188 // main
189 #define NB_TESTS_RUNS 3
190 int main(int argc, char **argv)
191 {
192     unsigned int run_number;
193     int checksum;
194     clock_t bg, ms;
195 
196     // Let's comment the call to these 2 functions in a first approach
197     //if (test_sha3() == 0 && test_shake() == 0)
198     //    printf("FIPS 202 / SHA3, SHAKE128, SHAKE256 Self-Tests OK!\n");
199 
200     // Add the clock measurement
201     bg = clock();
202     for(run_number = 0; run_number < NB_TESTS_RUNS; run_number++)
203         checksum ^= test_speed();
204 
205     ms = (clock() - bg) / 1000.0;
206 
207     // Print the result as in the initial program
208     printf("(%016lX) %.3f Keccak-p[1600,24] / Second.\n",
209         (unsigned long) checksum, ((NB_TESTS_RUNS * NB_ROUNDS * 1000)/ ((double) ms)) );
210 
211     return 0;
212 }
```

```
1 INFO: [SIM 2] *************** CSIM start ***************
2 INFO: [SIM 4] CSIM will launch GCC as the compiler.
3    Compiling ../../../../main.cpp in debug mode
4    Generating csim.exe
5 (FFFFFFFFE7CC0386) 86956.522 Keccak-p[1600,24] / Second.
6 INFO: [SIM 1] CSim done with 0 errors.
7 INFO: [SIM 3] *************** CSIM finish ***************
```

This step is now completed.

You will notice that the checksum is not the same than the original test since the implementation of the loop was done differently. Even more setting an even number to NB_TEST_RUNS will always return a null checksum since the function always return the same result and doing a XOR of 2 same checksums will be equal to zero. In the final release, we have implemented a way to return a variable to checksum to control the results.

In this step, you have synthesized the code of the algorithm that will be placed in the FPGA and put in place the testbench to check that .

This example is very simple and only a **clock system call** was encountered while synthesizing the test_speed function. When you'll try to synthesize the 2 other functions, you will see that more adaptations are necessary such as a "**union"** which is not recognized by the synthesizer. You can also notice that standard function like "**memset"** are not supported, only if they do a cast on the top of setting a variable.

All constraints are listed in HLS user guide:

➔ Download the Xilinx HLS User Guide UG902 – Chapter 3 - **Unsupported C Constructs**

Some user experienced recommendations:

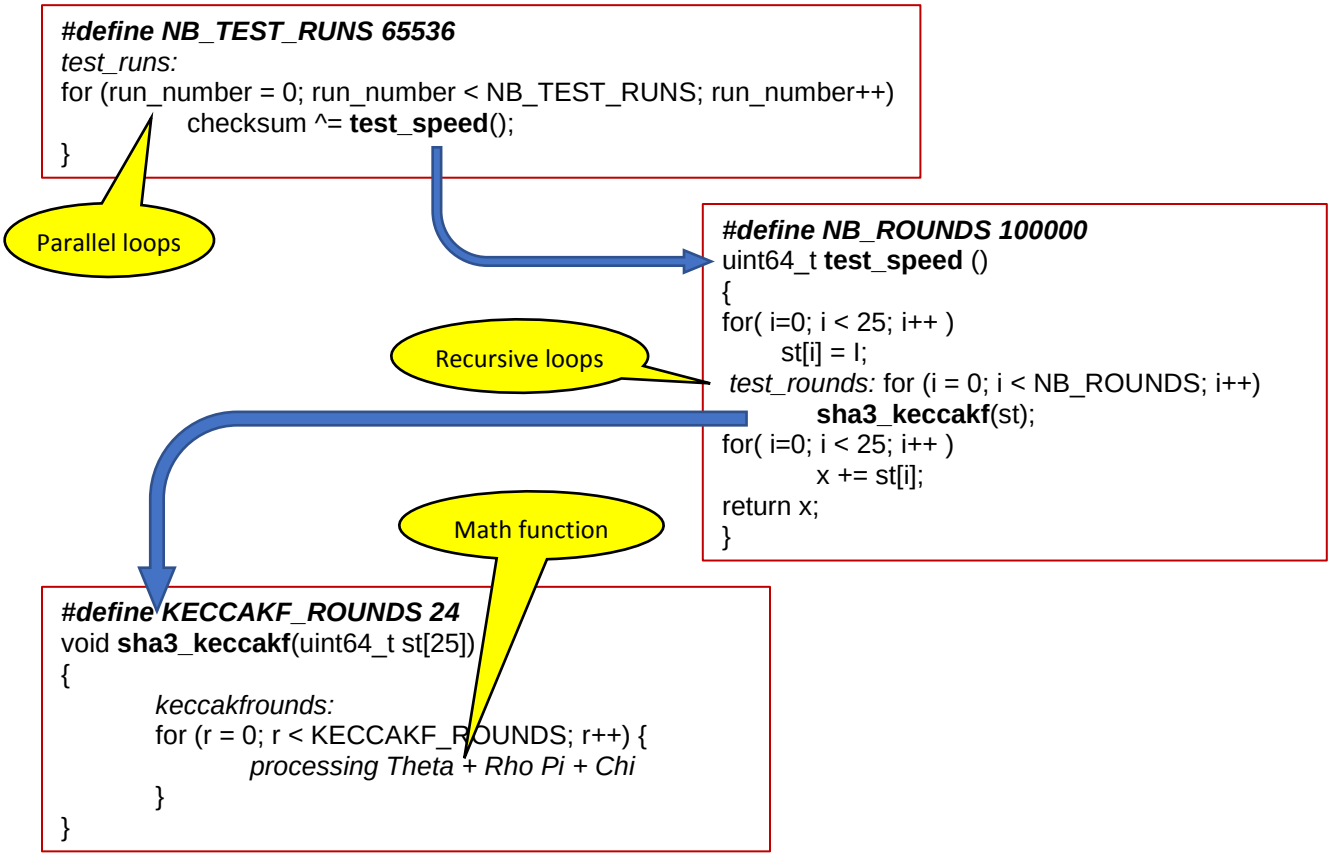
- Keep your code in small functions
    - Add #pragma HLS INLINE off to keep the hierarchy ➔ helpful for debug
- Set fixed bounds to loops
- Keep code simple to help HLS compiler being efficient
- "Switch/case" is often more efficient than "if/then/else"
- Always test your code **without** #pragma and test after each adding
- Unused logic is removed by Vivado:
    - Remove unused ports to remove the drivers if not needed

3. Evaluate the performance of the action (optional).

This step is to evaluate the possible performance we can reach with a FPGA by porting the algorithm to it. This is not mandatory but just let you discover the way to do measurements with the Vivado HLS tool. This step can be skipped to go directly to the next one.

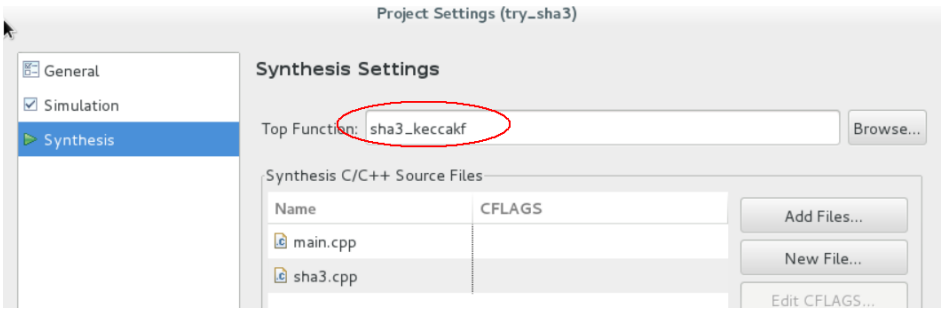
3.1 Understand the architecture of the code.

Now that your code is synthesizable, let's see if we can evaluate the time to process the main algorithm which is in the **sha3_keccakf** function. Indeed the 2 levels above this math function are loops that can may be parallelized, but the final execution time will mainly depend on the time the **sha3_keccakf** will take.



3.2 Understand the key numbers to analyze performances on Vivado HLS interface.

Let's start to synthesize only the **sha3_keccakf** function. Set **sha3_keccakf** as the top function**.**



Press the **synthesis** button. This will display the **Synthesis(solution1)** tab with all reports.

There are 3 key information to look to carefully. 2 in the Synthesis
The **first** essential information is the **Timing Summary**. We need to get an estimated number below the "target – uncertainty".
*(The **Target** clock has been set earlier but can be modified by pulling in the menu Solution > Solution Settings > Synthesis tab and **Period** field.)*

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	4.00	3.14	0.50

The **second** essential information is the utilization estimates. This shows the amount of logic taken by the algorithm. When working on optimizing a function, these numbers may change and need to be monitored to see if a change if affordable or not.
*(The type and amount of **resources** depend on the FPGA type set earlier but can be modified by pulling in the menu Solution > Solution Settings > Synthesis tab and **Part** field.)*

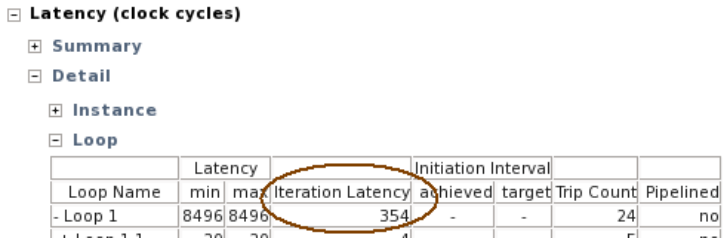
Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1559
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	6	-	11	5
Multiplexer	-	-	-	364
Register	-	-	761	
Total	6	0	772	1928
Available	4320	5520	1328720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	~0	0	~0	~0
Utilization SLR (%)	~0	0	~0	~0

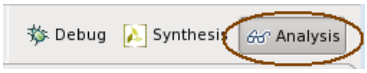
(FYI: FF stands for Flip-flop and LUT for LookUp Table)

The **third** essential information is the **latency** which means the time taken to execute a function

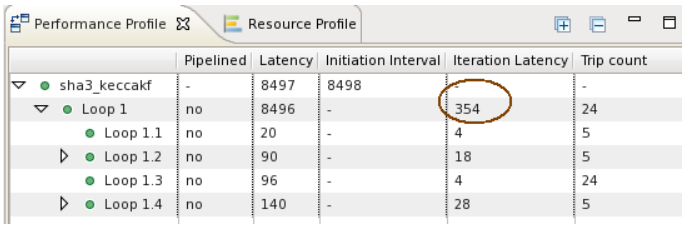


This information which is displayed in the **Synthesis(solution1)** tab of Synthesis view

can also be found in the Analysis view



with more details:



In summary, this means that the **sha3_keccakf** function:

- Can be synthesized within the clock constraints given (3.14 < 3.5ns)
- Will take 772FF and 1928LUT (whatever it means)
- Will take 354 x 4ns=1416ns=**1.416µs** to be executed.
  (This number 354 may change depending on Vivado HLS release)

Saying that another way, considering just this function, this means that we can run it **706 214 times per seconds.**

3.3 Understand what directives can be used to optimize the synthesis.

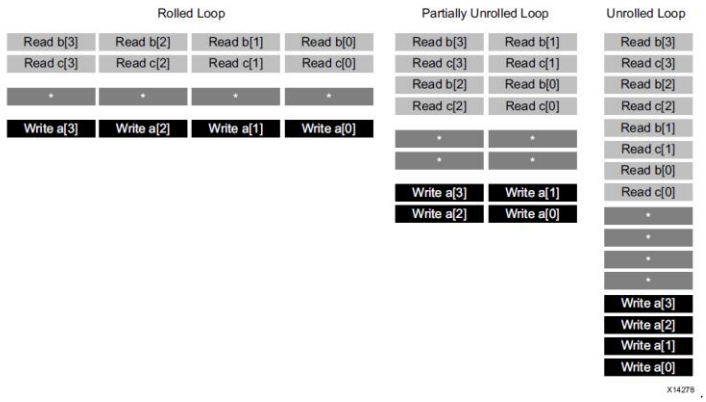
Amongst plenty of possible directives that may require hardware skills to implement them, there are 2 majors directives that can be tried easily to optimize loops. HLS User Guide UG902 lists the 24 "optimization directives" that can be used and how to use them.

	Directive Description
ALLOCATION	Specify a limit for the number of operations, cores or functions used. This can force the sharing or hardware resources and may increase latency
ARRAY_MAP	Combines multiple smaller arrays into a single large array to help reduce block RAM resources.
ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks.
ARRAY_RESHAPE	Reshape an array from one with many elements to one with greater word-width. Useful for improving block RAM accesses without using more block RAM.
CLOCK	For SystemC designs multiple named clocks can be specified using the create_clock command and applied to individual SC_MODULEs using this directive.
DATA_PACK	Packs the data fields of a struct into a single scalar with a wider word width.
DATAFLOW	Enables task level pipelining, allowing functions and loops to execute concurrently. Used to minimize interval.
DEPENDENCE	Used to provide additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals).
EXPRESSION_BALANCE	Allows automatic expression balancing to be turned off.
FUNCTION_INSTANTIATE	Allows different instances of the same function to be locally optimized.
INLINE	Inlines a function, removing all function hierarchy. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.
INTERFACE	Specifies how RTL ports are created from the function description.
LATENCY	Allows a minimum and maximum latency constraint to be specified.
LOOP_FLATTEN	Allows nested loops to be collapsed into a single loop with improved latency.
LOOP_MERGE	Merge consecutive loops to reduce overall latency, increase sharing and improve logic optimization.
LOOP_TRIPCOUNT	Used for loops which have variables bounds. Provides an estimate for the loop iteration count. **This has no impact on synthesis, only on reporting**.
OCCURRENCE	Used when pipelining functions or loops, to specify that the code in a location is executed at a lesser rate than the code in the enclosing function or loop.
PIPELINE	Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.
PROTOCOL	This commands specifies a region of the code to be a protocol region. A protocol region can be used to manually specify an interface protocol.
RESET	This directive is used to add or remove reset on a specific state variable (global or static).
RESOURCE	Specify that a specific library resource (core) is used to implement a variable (array, arithmetic operation or function argument) in the RTL.
STREAM	Specifies that a specific array is to be implemented as a FIFO or RAM memory channel during dataflow optimization.
TOP	The top-level function for synthesis is specified in the project settings. This directive may be used to specify any function as the top-level for synthesis. This then allows different solutions within the same project to be specified as the top-level function for synthesis without needing to create a new project.
UNROLL	Unroll for-loops to create multiple independent operations rather than a single collection of operations.

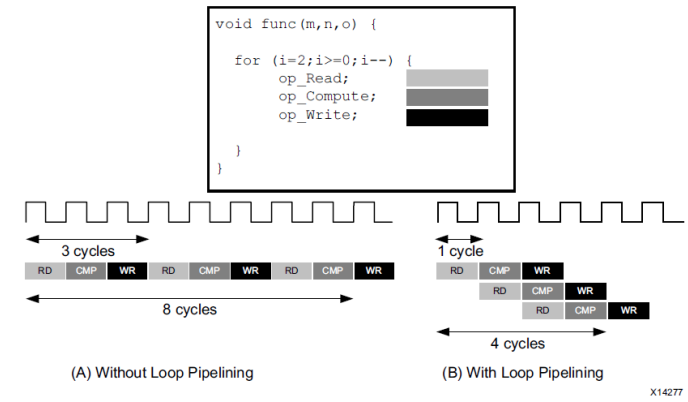
Remember that when trying an optimization directive, always run a simulation before and after to ensure that the synthesis didn't misunderstood your code and broke its functionality.

The **first** directive is a **#pragma HLS UNROLL *(factor = 2)*.** This is flattening the loop so that it can be executed in one clock period. This also means that the logic used is duplicated N times if N loops.

```
void  top(...) {
 ...
 for_mult:for (i=3;i>0;i--)  {
       a[i] = b[i] * c[i];
  }
 ...
}
```



The **second** directive is a **#pragma HLS PIPELINE.** This is first flattening the loop. Then the synthesizer will look to the relationship between all variables and find which processing can be done before the end of another processing to understand what can be parallelized / pipelined



This pipeline will so manage that the next data starts the next loop iteration with an interval of 1 cycle (default value but can be parametrized).

3.4 Using directives in the code and measure the effect.

In the **sha3_keccakf** function, let's try to add a simple **pragma** directive in the main loop

```
48 #endif
49
50     // actual iteration
51     for (r = 0; r < KECCAKF_ROUNDS; r++) {
52 #pragma HLS PIPELINE
53         // Theta
54         for (i = 0; i < 5; i++)
55             bc[i] = st[i] ^ st[i + 5] ^ st[i + 10] ^ st[i + 15] ^ st[i + 20];
56
57         for (i = 0; i < 5; i++) {
58             t = bc[(i + 4) % 5] ^ ROTL64(bc[(i + 1) % 5], 1);
59             for (j = 0; j < 25; j += 5)
60                 st[j + i] ^= t;
61         }
62
63         // Rho Pi
64         t = st[1];
```

Run first a simulation to check that your result is still as expected. This is confirmed as working ok:

```
1 INFO: [SIM 2] *************** CSIM start ***************
2 INFO: [SIM 4] CSIM will launch GCC as the compiler.
3    Compiling ../../../../main.cpp in debug mode
4    Compiling ../../../../sha3.cpp in debug mode
5    Generating csim.exe
6 (FFFFFFFFE7CC0386) 86956.522 Keccak-p[1600,24] / Second.
7 INFO: [SIM 1] CSim done with 0 errors.
8 INFO: [SIM 3] *************** CSIM finish ***************
9
```

Then run a synthesis and look to the console results. Before looking to the 3 key numbers, let's look to the logs. You may notice a WARNING that is displayed and need to be taken in consideration to get what you want:

```
Vivado HLS Console
INFO: [HLS 200-42] -- Implementing module 'sha3_keccakf'
INFO: [HLS 200-10] ----------------------------------------------------------------
INFO: [SCHED 204-11] Starting scheduling ...
INFO: [SCHED 204-61] Pipelining loop 'Loop 1'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependence constraint (II = 1, distance = 1, offset = 1)
   between 'store' operation (sha3.cpp:81) of variable 'tmp_2', sha3.cpp:81 on array 'st' and 'load' operation ('st_load', sha3.cpp:55) on array 'st'.
WARNING: [SCHED 204-69] Unable to schedule 'load' operation ('st_load_1', sha3.cpp:55) on array 'st' due to limited memory ports. Please consider using a memory core with more ports or partitioning the array 'st'.
INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 27, Depth = 27.
INFO: [SCHED 204-11] Finished scheduling.
INFO: [HLS 200-111]  Elapsed time: 14.57 seconds; current allocated memory: 0.326 MB.
```

This means that the pipeline couldn't be done correctly due to a load/store issue on the "st" array. *(Explanation about how to correct such issues can be found in Vivado HLS UG902).* This array "st" is the input and output argument of the **sha3_keccakf** function. The synthesizer has found a contention using the same argument as input and output.
Let's correct that simply by duplicating this entry argument as follow. We have decided to change the arguments names rather than modifying anything in the algorithm:

⇨ sha3.cpp file

```
// update the state with given number of rounds

void sha3_keccakf(uint64_t st_in[25], uint64_t st_out[25])
{
    // constants
    const uint64_t keccakf_rndc[24] = {
        0x0000000000000001, 0x0000000000008082, 0x800000000000808a,

    // variables
    int i, j, r;
    uint64_t t, bc[5];
    uint64_t st[25];

    //separate entry port from logic
    for (i = 0; i < 25; i++)
#pragma HLS UNROLL
        st[i] = st_in[i];

#if __BYTE_ORDER__ != __ORDER_LITTLE_ENDIAN__
    uint8_t *v;

        v[6] = (t >> 48) & 0xFF;
        v[7] = (t >> 56) & 0xFF;
    }
#endif
    //separate entry port from logic
    for (i = 0; i < 25; i++)
#pragma HLS UNROLL
        st_out[i] = st[i];

}
```

⇨ sha3.H

```
    int pt, rsiz, mdlen;                    // these don't overflow
} sha3_ctx_t;

// Compression function.
void sha3_keccakf(uint64_t st_in[25], uint64_t st_out[25]);

// OpenSSL - like interfece
int sha3_init(sha3_ctx_t *c, int mdlen);    // mdlen = hash output in bytes
```

⇨ main.c file

```
// test speed of the comp
#define NB_ROUNDS 100000
uint64_t test_speed()
{
    int i;
    uint64_t st[25], x, n;
    //clock_t bg, us;

    for (i = 0; i < 25; i++)
        st[i] = i;

    // bg = clock();
    n = 0;
    // do {
        test_rounds: for (i = 0; i < NB_ROUNDS; i++)
            sha3_keccakf(st, st);
        n += 1;
    //    us = clock() - bg;
    // } while (us < 3 * CLOCKS_PER_SEC);
```

Correct it also everywhere in the sha3.cpp file where the **sha3_keccakf** function is called (4 occurrences)

Rerun the simulation until everything is corrected, and first see that the performance has increased.

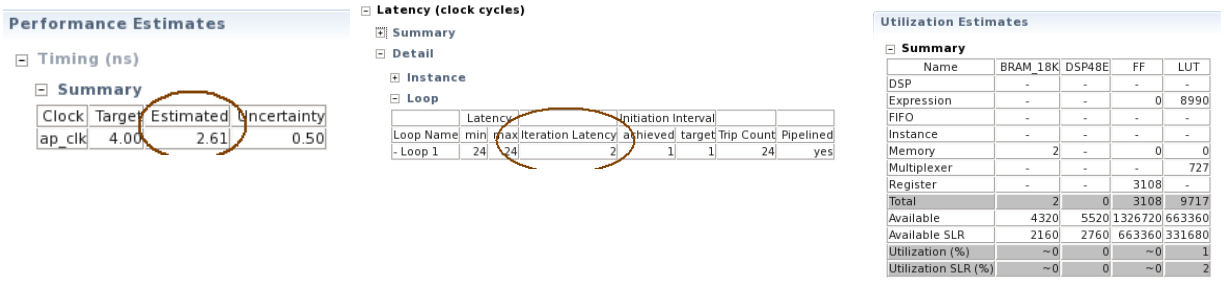
```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../sha3.cpp in debug mode
   Generating csim.exe
(FFFFFFFFE7CC0386  102040.816 Keccak-p[1600,24] / Second.
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Now let's synthesize the design and look if the console shows a better pipeline behavior

```
INFO: [HLS 200-10] ----------------------------------------------------------
INFO: [HLS 200-42] -- Implementing module 'sha3_keccakf'
INFO: [HLS 200-10] ----------------------------------------------------------
INFO: [SCHED 204-11] Starting scheduling ...
INFO: [SCHED 204-61] Pipelining loop 'Loop 1'.
INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 2.
INFO: [SCHED 204-11] Finished scheduling.
INFO: [HLS 200-111]  Elapsed time: 13.41 seconds; current allocated memory: 0.326 MB.
INFO: [RTND 205-100] Starting micro-architecture generation ...
```

All warnings have been removed and the pipelining result is as expected.

This means that the whole sha3_keccakf can be done in 2 clock cycles (depth) and that the Initiation Interval (II) is 1 meaning that new data can be entered every clock cycle.
Let's have a look to our 3 key numbers:



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	4.00	2.61	0.50

Latency (clock cycles)

Summary

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	24	24	2	1	1	24	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	8990
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	2	-	0	0
Multiplexer	-	-	-	727
Register	-	-	3108	-
Total	2	0	3108	9717
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	~0	0	~0	1
Utilization SLR (%)	~0	0	~0	2

In summary, this means that the **sha3_keccakf** function:

- Can be synthesized within the clock constraints given (2.61 < 3.5ns)
- Will take 3100FF (vs 772FF) and 9717LUT (vs 1928LUT) ➔ 4 to 5 more logic
- Will take 2 x 4ns=8ns (vs 1416ns) to be executed. ➔ 177x better

This means that just considering this function we can execute it in now 125 **000 000 times per seconds!**

Taking 4 to 5 time more logic may certainly be affordable in this case since the execution time (latency) is interesting to get more performances.

And this is just by considering this math function and not the loops at upper level. We will see later that parallelizing the loops at the top level will allow to run the checksum calculation in parallel depending on the size of the FPGA.
Do you think that a CPU could run this math function in only 2 cycles?

With very few changes we now know that this function may be efficient to be ported into a FPGA. It's worth trying to spend time to go to the end of the integration in SNAP.

A document has been published that can help you understanding more about optimization:
https://github.com/open-power/snap/blob/master/doc/AN_CAPI_SNAP-How_to_optimize_a_hardware_action.pdf

In this step, you have checked that the code of the algorithm can be optimized with very few instructions and modifications.

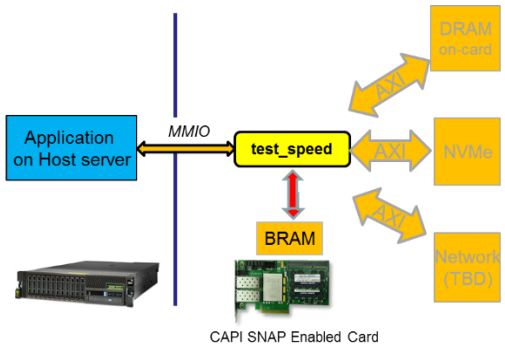
You can now close the Vivado HLS GUI.

4. Adapt your code to a SNAP action.

Now that your code is synthesizable, let's insert it into the SNAP formalism which will connect it easily to the entire design. This action can be found as an example with more options in https://github.com/open-power/snap/tree/master/actions/hls_sponge

4.1 Creating the SNAP new action.

The simpler may be to create a new action in SNAP environment by copying an existing action which has the same interfaces and then to insert your code in the action. As this **test_speed** program is not exchanging any data with any resources (DDR, ethernet, Flash…), then let's choose an example which has just the basic host interface access to exchange parameters and results with the user: hls_helloworld for example.



Let's follow the process described in https://github.com/open-power/snap/blob/master/doc/AN_CAPI_SNAP-How_to_create_a_New_Action.pdf

Let's call the files as follow, and add the 3 files main.cpp, sha3.cpp and sha3.H initially located in **education/hw** directory in the hls_tinysha3/hw directory.

As you are going to work on a newly created SNAP action, be sure to **close the VIVADO HLS GUI** so that the project opened previously is closed and that you can get all automatic parameters (Clock period, FPGA type, paths) automatically set correctly.

If you want to use the Vivado HLS GUI and have all the parameters set automatically, ensure to:

- *Go into **hls_tinysha3/hw** directory*
- *Execute almost once a **make***
- *Call **vivado_hls** to open GUI.*
- *Select Open Project*
- *Just select (no enter it) the "hlstinysha3_<fpgatype>" directory found in actions/hls_tinysha3/hw directory*

Let's start by working only on the action files located in the **snap/actions/hls_tinysha3/hw** and **snap/actions/hls_tinysha3/include** directories.

Two types of files can be found: the **header files** and the file **hw_action_tinysha3.cpp** (taken from the hls_helloworld example in last step) which has 3 functions: **process_action**, **hls_action** and **main**. Let's see in detail what they are about and what we will add in it.

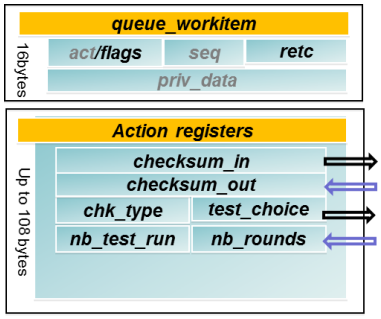
4.1.1 The header files

The header files located in **snap/actions/hls_tinysha3/hw** directory listed as **hw_action_tinysha3.H** and **sha3.H** shouldn't need any modification.

The file that will need some changes will be the header file **tinysha3_commonheader.h** located in **snap/actions/hls_tinysha3/include.** It will contain, per default, all settings or declaration that will be used by the action and by the application. It is here that you will add all #define values that can parametrized the code: NB_ROUNDS, NB_TEST_RUNS… but also the ACTION_TYPE value. You will often not find the RELEASE_LEVEL in this file since it is only action related and should be preferably in a header file in /hw used only by the action.

An important declaration here is the **tinysha3_job_t** structure declaration. This structure is the way to exchange parameters and/or small results between the application and the action using the asynchronous MMIO interface (no need to know more to use it). This structure will be sent by the application to the action before starting the action and will be read at the completion of the action or at any time if the application needs it.
Place in this structure all arguments that the action could need to report or to be parametrized. As an example, we only use the **chk_out** to report to the application, the checksum that the action has calculated. We can add a **chk_in**, if in a future we want to check or process a checksum provided by the application. We can also add a **chk_type** if we think other algorithms may be implemented in the action.

```
typedef struct tinysha3_job {
    uint64_t chk_in;      /* in:  checksum input */
    uint64_t chk_out;     /* out: checksum output */
    uint32_t chk_type;    /* in:  SPONGE, CRC32 */
    uint32_t test_choice; /* in:  use of you want to select test to execute */
    uint32_t nb_test_runs; /* out: report parameter – useful for debug */
    uint32_t nb_rounds;   /* out: report parameter – useful for debug */
} tinysha3_job_t;
```



VERY IMPORTANT:
A very important point is that this structure **tinysha3_job_t** can take **up to 108 Bytes**, never more. If user needs more space for registers, then he can use the host memory server space which can be any size. If he needs less, then an automatic padding will be done to add 0 to 108 bytes.

Some more information about how to use these data are described in: https://github.com/open-power/snap/blob/master/doc/AN_CAPI_SNAP-How_is_data_managed.pdf

4.1.2 The file **hw_action_tinysha3.cpp**: **process_action** function

```
#include <string.h>
#include "ap_int.h"
#include "hw_action_tinysha3.H"

//-----------------------------------------------------------------------
//--- MAIN PROGRAM ------------------------------------------------------
//-----------------------------------------------------------------------
static int process_action(snap_membus_t *din_gmem,
              snap_membus_t *dout_gmem,
              /* snap_membus_t *d_ddrmem, *//* not needed */
              action_reg *act_reg)
{
```

The ***process_action*** function will prepare the data and call the user code. This is where typically the loop calling the **test_speed** function will be located (moved from the initial main function). We will add in this function the read of the different parameters that we want to use to tune the tests, and the write of the results (checksum) to the registers so that the application can check and display the results.

We will add between the existing **#include** and before the **process_action** function, all the specific **include** and functions that are contained in the initial main.cpp file.

The process_action can so contain for example the following code:

```
void process_action (action_reg *Action_Register) {
    int rc = 1;
    uint64_t run_number, j;
    uint64_t checksum = 0;

    switch(Action_Register->Data.test_choice) {
    case(0):
        test_runs: for (run_number = 0; run_number < NB_TEST_RUNS; run_number++)
                checksum ^= test_speed();
        Action_Register->Data.chk_out = checksum;
        Action_Register->Data.nb_test_runs = NB_TEST_RUNS;
        Action_Register->Data.nb_rounds  = NB_ROUNDS;
        rc = 0;
        break;
    case(1):   /* use other cases to call other functions such as:  rc = test_sha3();  rc = test_shake(); */
        rc = test_sha3();
        Action_Register->Data.chk_out = rc;
        Action_Register->Data.nb_test_runs = 0;
        Action_Register->Data.nb_rounds  = 0;
        break;
…/…
    default:
        rc = 1;
        break;
    }
    /* Final output register writes */
  if (rc == 0) {
    printf("Test completed OK !!\n");
    Action_Register->Control.Retc = SNAP_RETC_SUCCESS;
  }
  else
    Action_Register->Control.Retc = SNAP_RETC_FAILURE;
}
```

You will notice that, as we don't need to get or send any data from application (host server) the 2 arguments din_gmem and dout_gmem which are the way to read and write data from the host memory server has been removed.

4.1.3 The file **hw_action_tinysha3.cpp**: **hls_action** function

```
//--- TOP LEVEL MODULE -------------------------------------------------
void hls_action(snap_membus_t *din_gmem,
        snap_membus_t *dout_gmem,
        /* snap_membus_t *d_ddrmem, // CAN BE COMMENTED IF UNUSED */
        action_reg *act_reg,
        action_RO_config_reg *Action_Config)
{
```

This hls_action contains the different interfaces. You shouldn't need to change anything in that function since it has the interfaces you need for this example and is just calling the **process_action** function.

Just check that your TINYSHA3_TYPE (or the one you have chosen) was correctly set when creating the new action.

```
void hls_action(snap_membus_t *din_gmem,
                snap_membus_t *dout_gmem,
                //snap_membus_t *d_ddrmem,
                action_reg *Action_Register,
                action_RO_config_reg *Action_Config)
{
// Host Memory AXI Interface
#pragma HLS INTERFACE m_axi port=din_gmem bundle=host_mem offset=slave depth=512
#pragma HLS INTERFACE s_axilite port=din_gmem bundle=ctrl_reg          offset=0x030

#pragma HLS INTERFACE m_axi port=dout_gmem bundle=host_mem offset=slave depth=512
#pragma HLS INTERFACE s_axilite port=dout_gmem bundle=ctrl_reg          offset=0x040

//DDR memory Interface
//#pragma HLS INTERFACE m_axi port=d_ddrmem bundle=card_mem0 offset=slave depth=512
//#pragma HLS INTERFACE s_axilite port=d_ddrmem bundle=ctrl_reg           offset=0x050

// Host Memory AXI Lite Master Interface
#pragma HLS DATA_PACK variable=Action_Config
#pragma HLS INTERFACE s_axilite port=Action_Config bundle=ctrl_reg      offset=0x010
#pragma HLS DATA_PACK variable=Action_Register
#pragma HLS INTERFACE s_axilite port=Action_Register bundle=ctrl_reg    offset=0x100
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl_reg

      /* Hardcoded numbers */
        switch (Action_Register->Control.flags) {
        case 0:
            Action_Config->action_type   = (snapu32_t) TINYSHA3_TYPE;
            Action_Config->release_level = (snapu32_t) RELEASE_LEVEL;
            Action_Register->Control.Retc = (snapu32_t)0xe00f;
            return;
            break;
        default:
            Action_Register->Control.Retc = (snapu32_t)0x0;
            process_action(Action_Register);
            break;
```

Ports declaration

Registers declaration

Code for discovery phase

Code calling action

4.1.4 The file **hw_action_tinysha3.cpp**: **main** function

```
//------------------------------------------------------------------------
//-- TESTBENCH BELOW IS USED ONLY TO DEBUG THE HARDWARE ACTION WITH HLS TOOL --
//------------------------------------------------------------------------

#ifdef NO_SYNTH

int main(void)
{
```

As the comment highlights it, it is the function that is used as a testbench when running in standalone in HLS GUI. It is required only when working with the HLS GUI; it is not used in the SNAP process.

Having this testbench is not mandatory BUT will help you in the optimization phase to ensure very quickly that the optimization you try leaves the functionality of you code ok. This is why it is highly recommended to spend some time writing this testbench. Even more it will quickly help you isolating the issue when you will need to do some debugging just on the action without considering the application.

The main function can so contain for example the following code:

```
#ifdef NO_SYNTH
int main(void)
{
    short i, j, rc=0;
    snap_membus_t din_gmem[1];     // declared but unused
    snap_membus_t dout_gmem[1];    // declared but unused
    //snap_membus_t d_ddrmem[1];   // declared but unused
    action_reg Action_Register;
    action_RO_config_reg Action_Config;

    // Get Config registers to read Action Type
    Action_Register.Control.flags = 0;
    hls_action(din_gmem, dout_gmem, //d_ddrmem,
                &Action_Register, &Action_Config);
    printf(">> ACTION TYPE = %8lx - RELEASE_LEVEL = %8lx <<\n",
            (unsigned int)Action_Config.action_type,
            (unsigned int)Action_Config.release_level);

    // Process the action
    Action_Register.Control.flags = 1;

    //********SPEED TESTS*******
    Action_Register.Data.test_choice = 0;   //speed test

    hls_action(din_gmem, dout_gmem, //d_ddrmem,
                &Action_Register, &Action_Config);
    printf(" ==> 2 test calls : checksum = %016llx", (long long) Action_Register.Data.chk_out);

    if (Action_Register.Data.chk_out == 0x83f5b14f2a6a5d8e)
        printf(" => checksum OK\n");
    else
        printf(" => WRONG checksum => expecting : 0x83f5b14f2a6a5d8e\n");       /* Final output register writes */
    if (Action_Register.Control.Retc == SNAP_RETC_FAILURE) {
                printf(" ==> RETURN CODE FAILURE <==\n");
                return 1;
    }
    else
        printf(" ==> RETURN CODE OK <==\n");
    return rc;
}
#endif // end of NO_SYNTH flag
```

You will notice that this **main** function is contained between a **#ifdef NO_SYNTH** and a **#endif**. This function will not be synthesized thanks to a NO_SYNTH parameter added automatically by SNAP scripts into the simulation settings. You can check this parameter in Project > Project Settings > **Simulation** tab > select the **hw_action_tinysha3.cpp** file and press Edit CFLAGS button. You should get the following line:

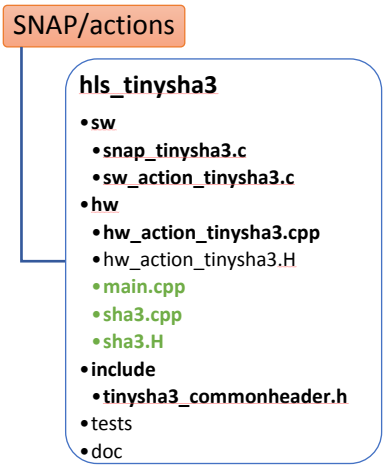
-I../../include -I../../../software/include -I../../../software/examples -I../include -DNO_SYNTH

This -DNO_SYNTH will not be found in the Synthesis tab CFLAGS. This is how the HLS tool will differentiate the files that are called from source and from testbench, even if they have the same name like we do.

In this step, you have adapted the action code to your algorithm code and with the testbench you can simulate and synthesize the whole action.

5. Adapt the application to call the Action code

5.1 Files architecture.



Now that the action code is ready, let's adapt the so-called "software code" (because executed on the CPU) and which is in **snap/actions/hls_tinysha3/sw** directory. Remember that the software code will use the settings defined in /include/tinysha3_cmmonheader.h file.

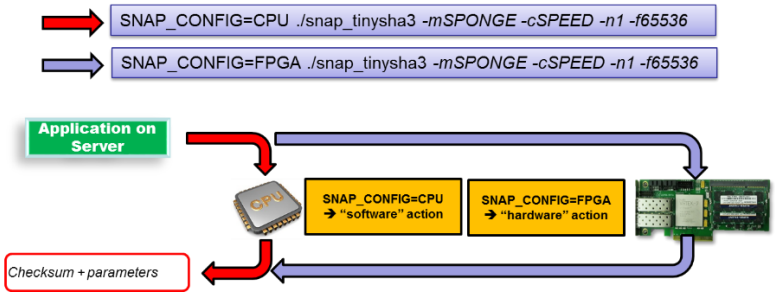
These files should have been created following the process described in paragraph "4.1 **Creating the SNAP new action.**".

You should so find in this /sw directory:

- the application named **snap_tinysha3.c** and
- the "software action" named **sw_action_tinysha3.c** file.

You will notice that the extension of these files can be ".c" and does not need to be ".cpp".

This "software action" should contain the same algorithm and processing than the "hardware action". It is very useful if we want to compare the time of a process executed on a CPU and the same algorithm executed on the FPGA, or if we want to isolate an issue understanding if the problem comes from the application or from the hardware action. Switching from the "software action" to the "hardware action" will be done using a parameter set on the command line SNAP_CONFIG:



Let's first have a look to the application which is common to all types of actions.

5.2 Adapting the application to call the action.

The snap_tinysha3.c file contains the whole application code.

Starting by the **main** function, you will have to:

- change and adapt the arguments options depending on your needs.
- read data from files and/or allocate (*host server = SNAP_ADDRTYPE_HOST_DRAM*) memory
  → *this is not needed with our tiny_sha3 example*
- allocate the card using the SNAP API *returning the card number*
  *card = **snap_card_alloc_dev**(device, SNAP_VENDOR_ID_IBM, SNAP_DEVICE_ID_SNAP);*
- attach the action using the SNAP API returning the action_id
  *action = **snap_attach_action**(card, TINYSHA3_ACTION_TYPE, action_irq, timeout);*
- fill the structure of data exchanged with the action using the ***snap_prepare_tinysha3(**…/…)* function.
- Write the structure of data, then start the action and wait until completion using the SNAP API
  *rc = **snap_action_sync_execute_job**(action, &cjob, timeout);*
- Test the return code and check/display the results
- Clean before exiting detaching the action, freeing the card and freeing the memory allocated

You will find before this main function all the functions needs:

- Usage to provide on-line help
- ***snap_prepare_tinysha3()*** *function setting the parameters using SNAP API* ***snap_addr_set*** which sets **snap_addr** types (*not needed with our tiny_sha3 example) and* ***snap_job_set*** to assign the settings structure to the job

5.3 Adapting the software action.

The **sw_action_tinysha3.c** file contains the whole software action code.

This file is structured as follow from top to bottom:

- 2 functions **mmio_write32** and **mmio_read32** needed for the trace/debug only. *Adding SNAP_TRACE=0x0/to/0xF before the command line will activate the trace (default is 0x0 value= no trace)*
- Code of the algorithm ➔ all the code from the hardware action (**hw_action_tinysha3.cpp** file) except the process_action and the hls_action can be added here

- A function selecting the **test_speed** test (as the code of the process_action)

```
static uint64_t sha3_main(uint32_t test_choice)
{
	uint32_t run_number;
	uint64_t checksum=0;

	act_trace("  sw: NB_TEST_RUNS=%d NB_ROUNDS=%d\n", NB_TEST_RUNS, NB_ROUNDS);
	switch(test_choice) {
	case(CHECKSUM_SPEED):
	{
		for (run_number = 0; run_number < NB_TEST_RUNS; run_number++) {
			uint64_t checksum_tmp;
			act_trace("  run_number=%d\n", run_number);
			checksum_tmp = test_speed(run_number);
			checksum ^= checksum_tmp;
			act_trace("    %016llx %016llx\n",
				(long long)checksum_tmp,
				(long long)checksum);
		}
	}
		break;
	case(CHECKSUM_SHA3):
		checksum = (uint64_t)test_sha3();
		break;
…/…
	default:
		checksum = 1;
		break;
	}

	act_trace("checksum=%016llx\n", (unsigned long long)checksum);
	return checksum;
}
```

- The **action_main** function selecting the type of checksum you want to calculate

```
static int action_main(struct snap_sim_action *action, void *job,
                unsigned int job_len)
{
      struct checksum_job *js = (struct checksum_job *)job;
      void *src;

      act_trace("%s(%p, %p, %d) [%d]\n", __func__, action, job, job_len,
            (int)js->chk_type);

      switch (js->chk_type) {
      case CHECKSUM_SPONGE: {
            act_trace("test_choice=%d", js->test_choice);
            if(js->test_choice == CHECKSUM_SPEED) {
               js->nb_test_runs = NB_TEST_RUNS;
               js->nb_rounds = NB_ROUNDS;
            }
            else {

               js->nb_test_runs = 0;
               js->nb_rounds = 0;
            }
            js->chk_out = sha3_main(js->test_choice);
            break;
      }
      case CHECKSUM_CRC32:
…/…
      default:
            return 0;
      }
      action->job.retc = SNAP_RETC_SUCCESS;
      return 0;
}
```

- And the action switch that shouldn't need any modification

```
static struct snap_sim_action action = {
        .vendor_id = SNAP_VENDOR_ID_ANY,
        .device_id = SNAP_DEVICE_ID_ANY,
        .action_type = CHECKSUM_ACTION_TYPE,

        .job = { .retc = SNAP_RETC_FAILURE, },
        .state = ACTION_IDLE,
        .main = action_main,
        .priv_data = NULL,      /* this is passed back as void *card */
        .mmio_write32 = mmio_write32,
        .mmio_read32 = mmio_read32,

        .next = NULL,
};

static void _init(void) __attribute__((constructor));

static void _init(void)
{
        snap_action_register(&action);
}
```

5.4 Compiling the whole software.

From the software directory /sw you should compile executing a **make** and then call the action as follow:

(SNAP_CONFIG=FPGA and SNAP_TRACE=0x0 are default values)

- Application calling the **software** action:
  SNAP_CONFIG=CPU ./snap_tinysha3 -vv -t2500 -mSPONGE -N -cSPEED -n1 -f65536
- Application calling the **hardware** action:
  ./snap_tinysha3 -vv -t2500 -mSPONGE -N -cSPEED -n1 -f65536
- Application calling the **hardware** action with the **trace** of all MMIO exchanges between the application and the action :
  SNAP_TRACE=0xF SNAP_CONFIG=FPGA ./snap_tinysha3 -vv -t2500 -mSPONGE -N -cSPEED -n1 -f65536

6. Measure the Action performance and optimize it

In paragraph "4 Overview**Evaluate the performance of the action (optional).**", we have seen that optimizing a math function has shown promising results. Let's see, now that the whole code is implemented, if we can measure the time taken by the action. Several ways to do that:

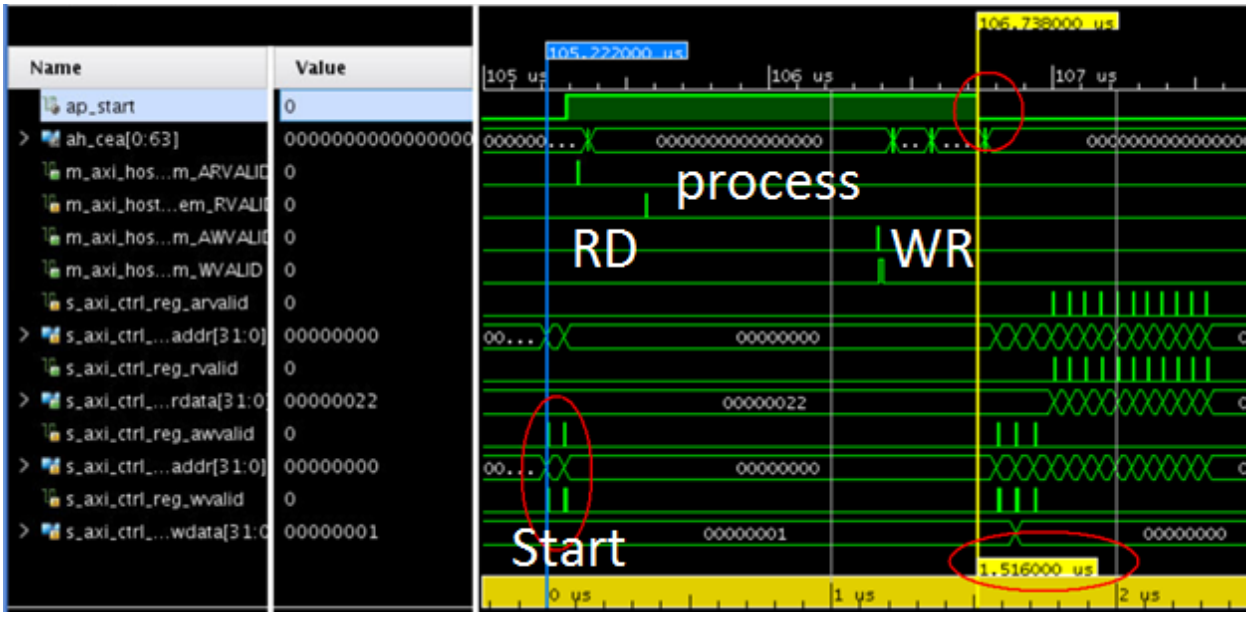
6.1 Option 1: An overall view using simulation of the whole application + code

A way to measure the real time taken by an action is to use the simulator graphical view ran after a "**make sim" command.** This may not be the preferred way for a software guy since it looks too much a "hardware view" of things, but this can be helpful to understand where time is spent.

For more information, refer to paragraph "1.9 Debugging the hardware action with the application" of https://github.com/open-power/snap/blob/master/doc/AN_CAPI_SNAP-How_to_debug_an_issue.pdf

This will show you all details about the real timing of the action run by the application.

The ap_start signal (not seen in the action code) will be enable (set to 1) during all the action activity.



6.2 Option 2: A detailed view using Vivado HLS GUI

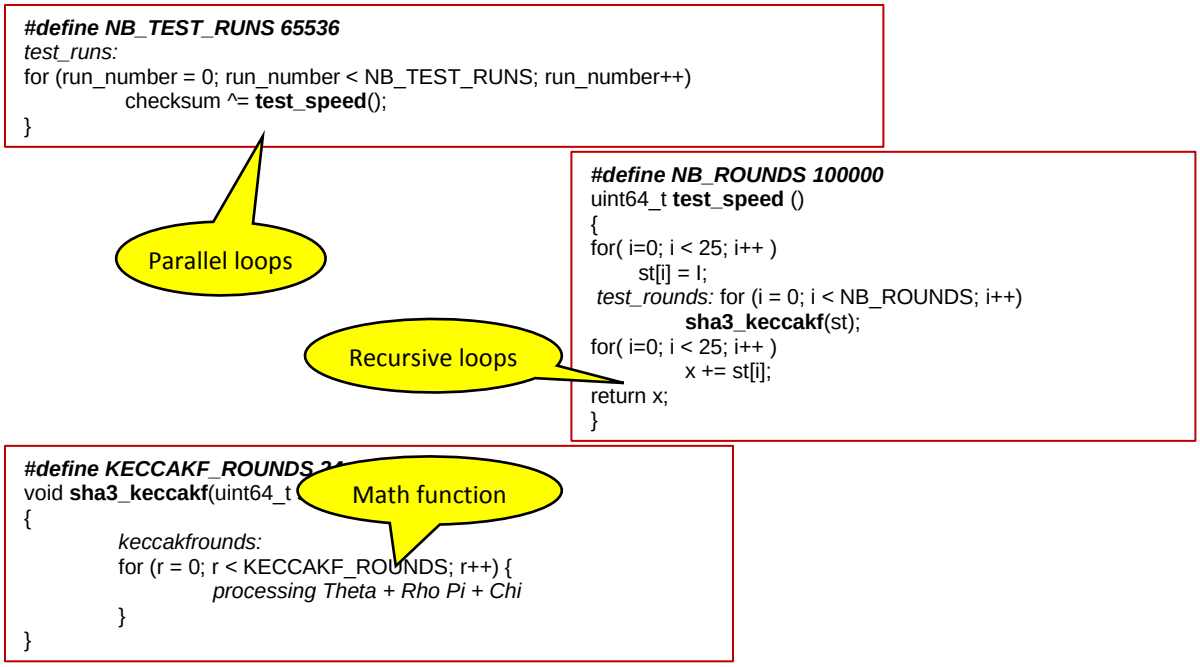
As already done previously, let's open the Vivado HLS GUI and load the action. It is mandatory to have built a testbench at the bottom of the hardware action code.

As a reminder, if you want to use the Vivado HLS GUI and have all the parameters set automatically, ensure to:

- *Go into **hls_tinysha3/hw** directory*
- *Execute almost once a **make***
- *Call **vivado_hls** to open GUI.*
- *Select Open Project*
- *Just select (no enter it) the "hlstinysha3_<fpgatype>" directory found in actions/hls_tinysha3/hw directory*

Remember the 3 sorts of function:

- **sha3_keccakf** which is the Math function was optimized with a **#pragma HLS PIPELINE** into the *keccakfrounds* loop. This function can be executed within 2 clock cycles.
- **test_speed** contains **recursive loops** meaning that the function needs the result of the previous value of "st" to start a new call of the function. Adding some pragma may add a lot of logic with a very small performance increase.
- **Main** function contains a loop calculating checksums. The good thing is that the XOR of 2 checksums can be done in any order, meaning that we could parallelize the call of the test_speed with a basic **#pragma HLS UNROLL** added in the **test_runs** loop**.** Depending on the number of NB_TEST_RUNS loops, the FPGA may not be big enough to contain all the duplicated logic added by this unroll, so adding a **factor=4/16/32** after the UNROLL instruction can help managing the parallelization.



To illustrate this parallelization , Vivado HLS will show you that the function are physically duplicated

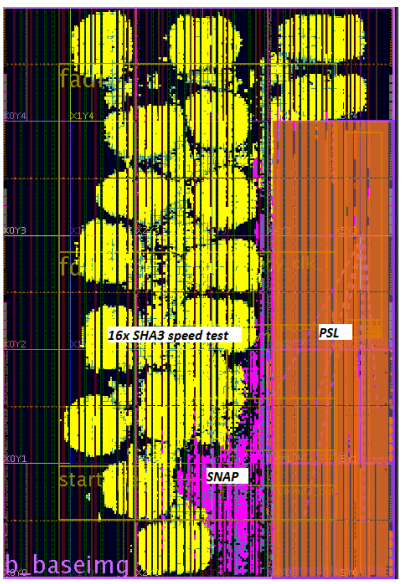
Instance

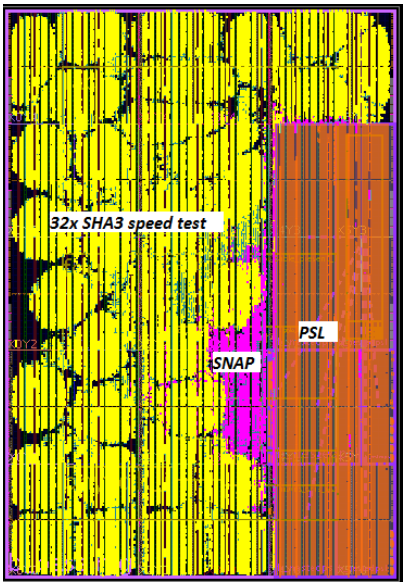
Instance	Module	Latency min	Latency max	Interval min	Interval max	Type
grp_process_action_fu_149	process_action	?	?	?	?	none

Instance

Instance	Module	Latency min	Latency max	Interval min	Interval max	Type
grp_test_shake_fu_174	test_shake	1293323	7567313	1293323	7567313	none
grp_test_sha3_fu_188	test_sha3	?	?	?	?	none
grp_test_speed_fu_208	test_speed	223473471		223473471		none
grp_test_speed_fu_218	test_speed	223473471		223473471		none
grp_test_speed_fu_227	test_speed	223473471		223473471		none
grp_test_speed_fu_236	test_speed	223473471		223473471		none

A picture of the duplication of the test_speed can be physically seen when the FPGA has been fully routed.





This means that between 16 and 32 duplications, you have twice the channels to process your checksum, so dividing the time of your overall calculation by 2!

						slices/16	slices/16	slices/32	slices/32	CPU (antipode) 16 cores - 160 threads	CPU (antipode) 16 cores - 160 threads
						FPGA KU060-16//	FPGA KU060-16//	FPGA KU060-32//	FPGA KU060-32//	System P	System P
NB_ROUNDS	NB_TEST_RUNS	nb_elmts	freq	test_speed calls	Checksum	(keccak per sec)	(msec)	(keccak per sec)	(msec)	(keccak per sec)	(msec)
100,000	65,536	1	65,536	100,000	3e05f34be7cc0386	4,624,491	22	**4,666,573**	21	**149,575**	669
100,000	65,536	2	65,536	200,000	2ccef6d61b67ad2f	9,248,983	22	**9,334,453**	21	**295,786**	676
100,000	65,536	4	65,536	400,000	0796ca863ac8273f	18,498,821	22	**18,668,036**	21	**488,441**	819
100,000	65,536	8	65,536	800,000	0018c0972c9227d2	36,990,799	22	**37,330,845**	21	**865,289**	925
100,000	65,536	16	65,536	1,600,000	5bd139d5bf8dad3a	73,995,283	22	**74,672,143**	21	**1,572,084**	1,018
100,000	65,536	32	65,536	3,200,000	a0c267468cf1e051	74,722,709	43	**143,568,576**	22	**2,539,064**	1,260
100,000	65,536	128	65,536	12,800,000	05c290e99ff8b7ae	75,279,062	170	**149,900,457**	85	**3,699,211**	3,460
100,000	65,536	4,096	65,536	409,600,000	ed3ff1c664125abb	75,465,691	5,428	**150,837,950**	2,715	**4,267,759**	95,975
100,000	65,536	8,192	65,536	819,200,000	cfd69627069b3e3e	75,468,917	10,855	**150,900,077**	5,429	**4,303,717**	190,347
100,000	65,536	32,767	65,536	3,276,700,000	eb4c1384fa60e252	75,468,889	43,418	**150,937,573**	21,709	**4,344,618**	754,198
100,000	65,536	65,536	65,536	6,553,600,000	38c7143fc6c46500	75,471,578	86,835	**150,941,821**	43,418	**4,352,266**	1,505,790