IBM CAPI SNAP framework

Version 1.2

How to optimize a HLS action in SNAP environment.

The Guide describes how to measure and optimize the code of a function in SNAP environment.

In a first step, you have ported your code to a FPGA. In a second step, you need to see how to optimize it. The final performance depends on the way you will write your code. This is very similar to what we do for software. The difference is in the concepts used for optimizing FPGA performance.

Overview

Let's use the very simple example **hls_helloworld** to learn how we can measure the time spent by the function, also called "hardware action", and how to optimize it. This example is so basic that you will not get much from it, but you will hopefully understand the concept. As soon as you understand these basic points, you can try to port and optimize the sponge/SHA3 code which will give you more experience about issues you might face and how to solve them (see related Application Note in https://github.com/open-power/snap/doc).

The last part of this document is key to understanding the different "times" reported and what they represent. It will explain the overhead added to access the FPGA and should help you understand when offloading a function becomes interesting or not.

We will use the Nimbix cloud environment to illustrate the issues and their resolutions, but this can be easily translated to any other environment.

Notice that we use HLS (High Level Synthesis) tool in this example, because it provides a fast way to generate your own hardware implementation from a C/C+ code. Amongst others, the Xilinx Vivado HLS tool was selected since it is one that better enables those users who have less hardware knowledge. You can easily use the other HLS tools provided by other providers (Stratus from Cadence, …) if the tool can generate HDL (Verilog, VHDL, ...) code.

This document has been built using snap git release tag v1.3.2. It can certainly work also for new releases.

This document will successfully go through the following items:
- understand where to find and set parameters that can impact the time spent by your function.
- how to identify in which function most time is spent
- find a path for improvement
- try optimizations and compare performance results
- understand the difference in time reported by application and actions.

Document history

V1.0	February 15th, 2017	Initial release
V1.1	May 31st, 2018	Added chapter *4.8 Conclusion: is there a way to remove this overhead?*

Contents

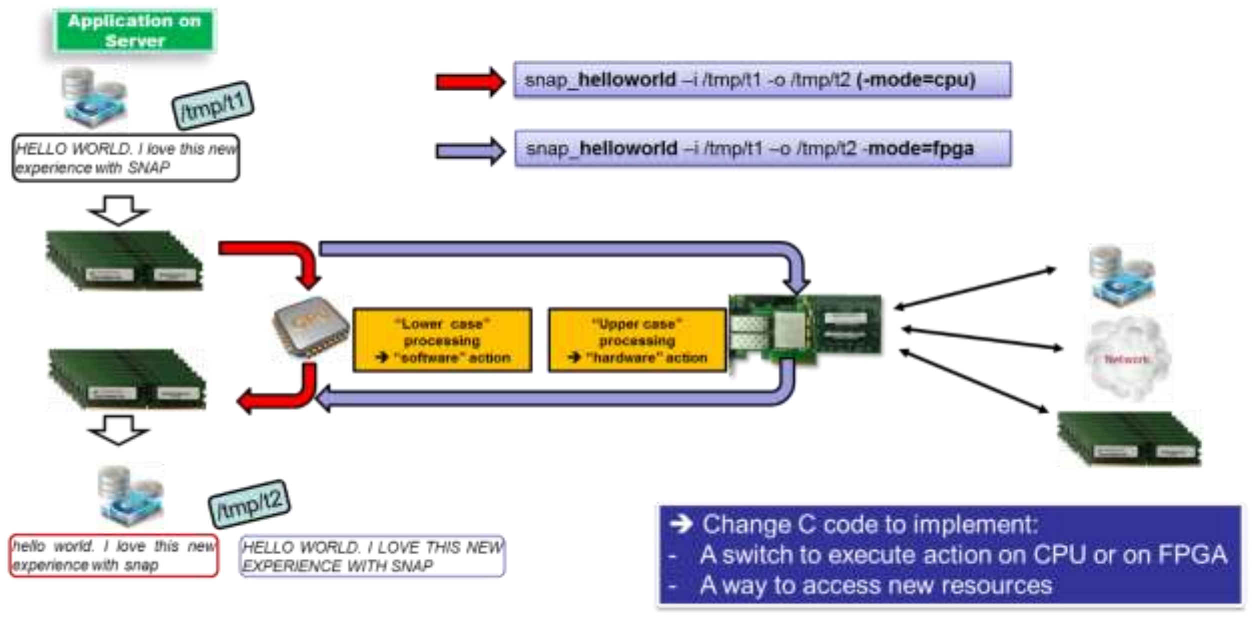
1. The helloworld example

Before starting to dive into details, let us quickly explain what the example we are using is doing.

The SNAP helloworld sample application consists of a software part running on the host as well as a SNAP action which is executed with on the FPGA. The code running on the host-system opens a file, places the data into an input memory buffer, opens a SNAP context and triggers the SNAP action on the card to perform the data processing.

Once the SNAP hardware action receives the order to start the job, it will transfer the data from the input memory buffer into the FPGA, does the requested processing, e.g. converting characters to upper case, and transfer the results back into the provided output memory on the host.

The host application waits for the SNAP hardware action to complete, e.g. by using an interrupt or polling a special register. Once the hardware action signals completion, the action has processed the data and already written it into the provided output memory buffer. From there it is written into a new file. It can now be further processed.


Application on Server
/tmp/t1
HELLO WORLD. I love this new experience with SNAP
snap_helloworld –i /tmp/t1 -o /tmp/t2 (-mode=cpu)
snap_helloworld –i /tmp/t1 –o /tmp/t2 -mode=fpga
"Lower case" processing → "software" action
"Upper case" processing → "hardware" action
Network
/tmp/t2
hello world. I love this new experience with snap
HELLO WORLD. I LOVE THIS NEW EXPERIENCE WITH SNAP
→ Change C code to implement:
- A switch to execute action on CPU or on FPGA
- A way to access new resources

In the following sections it is explained how the data processing is done inside the FPGA and which steps can be used to analyze the performance as well as methods to improve it.

2. How and where to find key data?

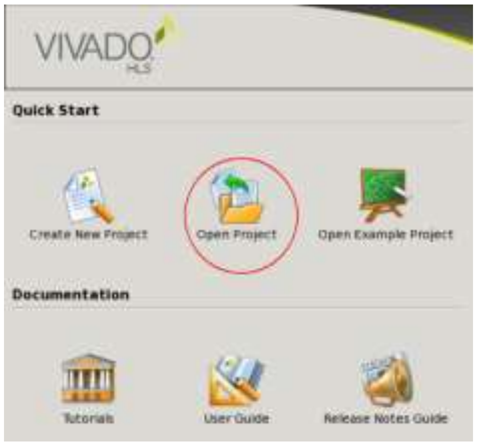
2.1 Opening Vivado HLS

The optimization of the code is essentially done on the hardware action.

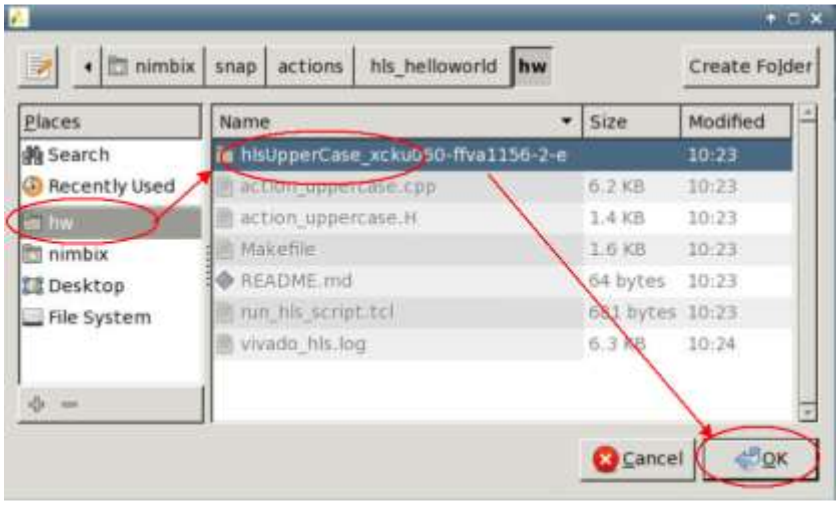
Go into ***hls_helloworld/hw*** directory, compile the hardware action and open vivado_hls tool. The compilation is necessary since this will set all parameters needed for the tool.

```
cd $SNAP_ROOT/actions/hls_helloworld/hw
make
vivado_hls
```

Select Open Project



Select then **hw** directory, **hlsUpperCasexxx** directory and then Select **OK**

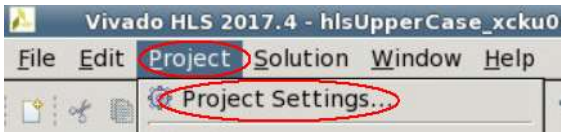


Name	Size	Modified
hlsUpperCase_xcku060-ffva1156-2-e		10:23
action_uppercase.cpp	6.2 KB	10:23
action_uppercase.H	1.4 KB	10:23
Makefile	1.6 KB	10:23
README.md	64 bytes	10:23
run_hls_script.tcl	681 bytes	10:23
vivado_hls.log	6.3 KB	10:24

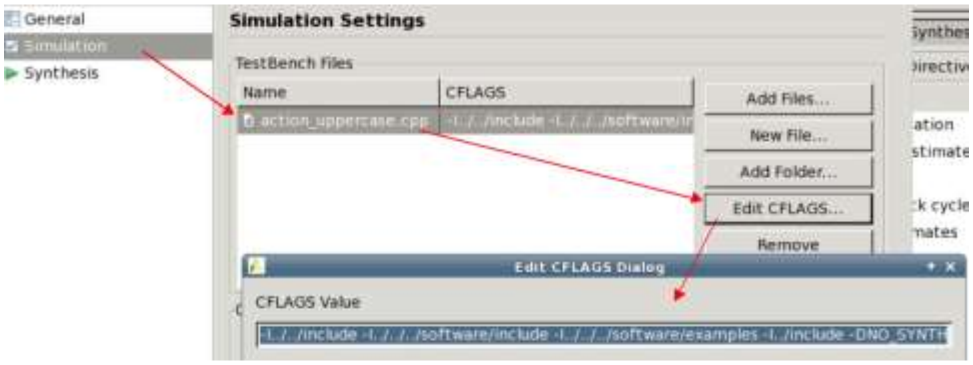
Several parameters have been set by SNAP scripts when compiling the hardware action. Let's see where they are so that you can better understand why they are set to these values.

In the menu, you have 2 important settings: one for the **project** and one for the **solution**. A **project** is related to the source code you are working on, while the **solution** is related to the FPGA you are working with.

Go to Project > Project Settings



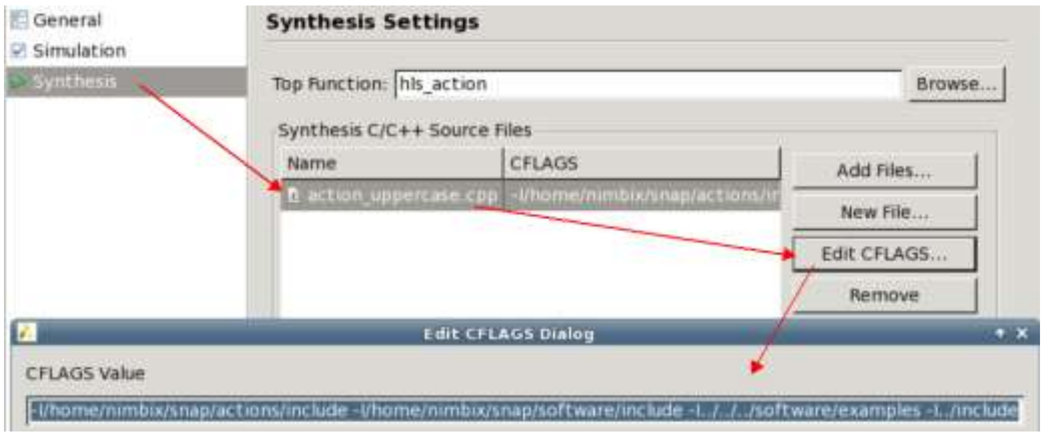
In **Simulation tab**, you will find the paths to all the header files used by your C/C++ hardware action code.



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

You will notice that the last parameter of the CFLAGS line is **-DNO_SYNTH**. This allows the user to add at the end of the hardware action code an area delimited by "**#ifdef NO_SYNTH / #endif"** which contains a unit test for the hardware action. This area will not be "synthesized" meaning that the code in this area will not be implemented into the FPGA.

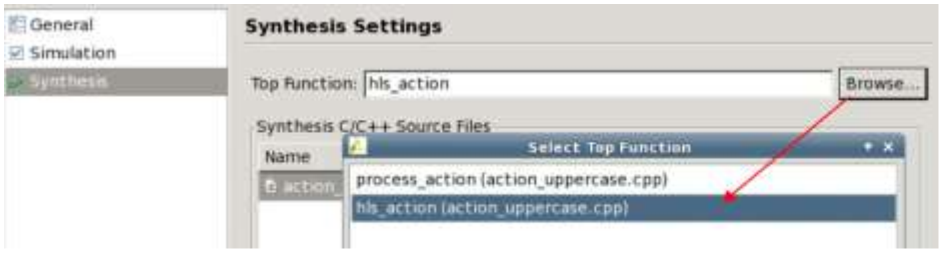
In **Synthesis tab,** you will find the same line but without the **-DNO_SYNTH.**



Top Function: hls_action

```
-I/home/nimbix/snap/actions/include -I/home/nimbix/snap/software/include -I../../../software/examples -I../include
```

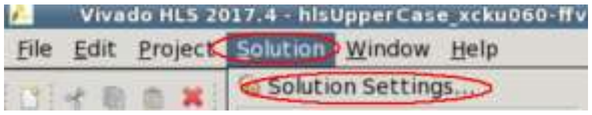
You will notice that at the top of this window, you can select a **Top function.** Push the **Browse** button and you will discover all the different functions contained in your hardware action. By default, the ***hls_action*** is selected since it is the top one. This feature gives you the ability to work with just part of your design instead of taking the whole function at once. This can be very useful when porting a function step by step.



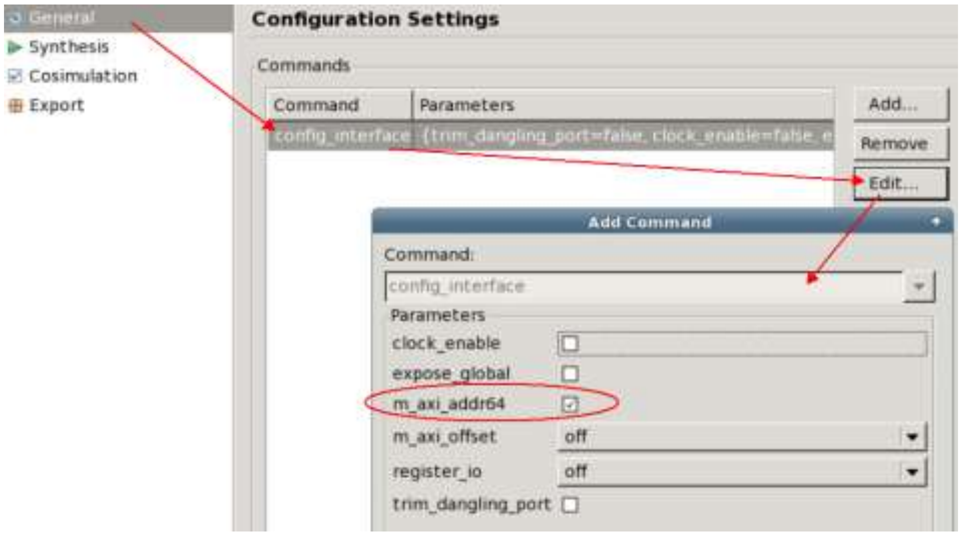
Top Function: hls_action

process_action (action_uppercase.cpp)
hls_action (action_uppercase.cpp)

Now let's have a look at the other part of the settings.
Go to **Solution > Solution Settings**



In the **General** tab, you can see a setting done on the ***config_interface***. This is to declare that all address busses used in SNAP are declared to be 64 bits wide (default is 32 bits).



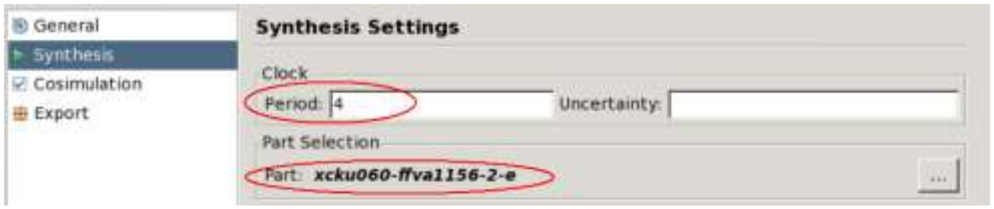
Command	Parameters
config_interface	{trim_dangling_port=false, clock_enable=false, e

Command: config_interface

Parameters

clock_enable	☐
expose_global	☐
m_axi_addr64	☑
m_axi_offset	off
register_io	off
trim_dangling_port	☐

In the **Synthesis** tab, you can select the **clock period** (250MHz = 4ns) and the **FPGA** exact **type** used by the card you are working with (XCKU060).



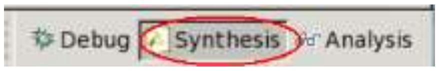
Period: 4 Uncertainty:

Part: *xcku060-ffva1156-2-e*

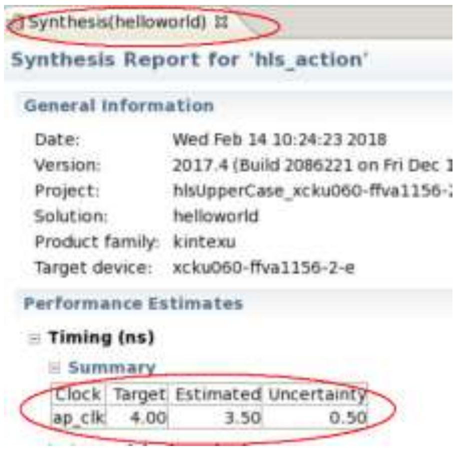
We will not go further into the other tabs since nothing has been changed in those.

2.3 Looking for key constraints

In the previous chapter, we have seen 2 important settings, which are the **constraints** given to the chip. The **clock period** is 4ns and the **FPGA** used is a KU060. The period gives the **speed of the logic** we are going to work with and the FPGA type will give you the **size of the FPGA**, meaning the amount of resources available for your hardware action.

Let’s stay first in the **Synthesis** view (right of your screen) 

The **clock period** constraint is displayed in the middle of your screen under the Synthesis tab. After every synthesis, you will need to check that the estimated value is below the “Target – Uncertainty” value.



Synthesis(helloworld)

Synthesis Report for 'hls_action'

General Information

Date:	Wed Feb 14 10:24:23 2018
Version:	2017.4 (Build 2086221 on Fri Dec 1
Project:	hlsUpperCase_xcku060-ffva1156-2
Solution:	helloworld
Product family:	kintexu
Target device:	xcku060-ffva1156-2-e

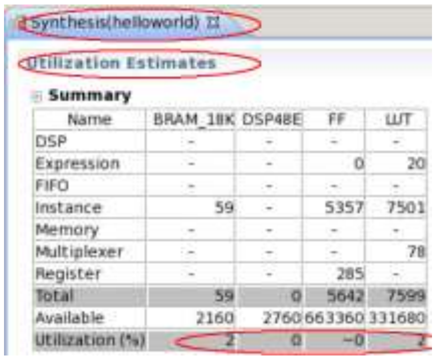
Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	4.00	3.50	0.50

In the same tab, a bit below, you will find the information about the **resources utilization** of your hardware action. Be careful, these values are not taking in account all the SNAP + PSL logic that is needed in the final design, but just your hardware action. To give you a rough idea, the “SNAP+PSL+memory drivers” can take between a third and a half of the FPGA. However, keep in mind that HLS gives very pessimistic estimation and the Vivado routing tool works efficiently, so you can easily try and reach 100% of your chip with your hardware action and fit into the FPGA!



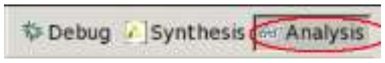
Synthesis(helloworld)

Utilization Estimates

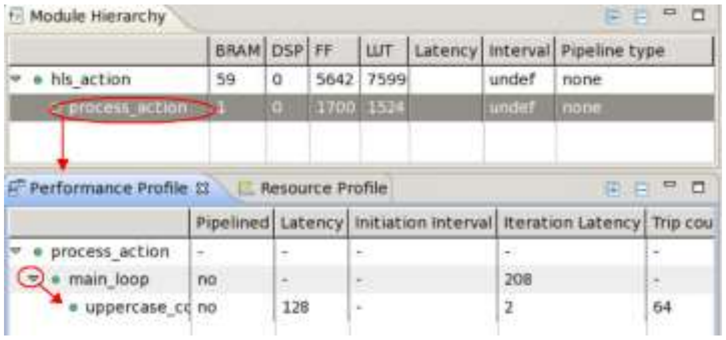
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	20
FIFO	-	-	-	-
Instance	59	-	5357	7501
Memory	-	-	-	-
Multiplexer	-	-	-	78
Register	-	-	285	-
Total	59	0	5642	7599
Available	2160	2760	663360	331680
Utilization (%)	2	0	~0	2

In the previous chapter, we have seen 2 important settings which are the constraints given to the design. Let's now see how to **measure the time** taken by your function, also called hardware action:

Let's go into the **Analysis** view (right of your screen) 

This will display all the data you need to do the measurement.
At the top of the screen, you will find the **amount of logic** used by every part of your functions (BRAM is FPGA internal memory, DSP (digital signal processing) is FPGA predefined math functionality, FF (flip-flop) and LUT (LookUp Table) is logic related).
At the bottom of the screen, you will find the **latency**, meaning the time taken by the logic described by your hardware action. (Select the functions above to get details on the functions below).



Module Hierarchy

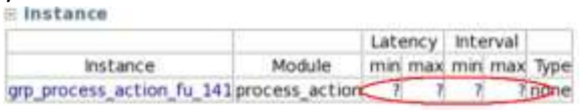
	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
hls_action	59	0	5642	7599		undef	none
process_action	1	0	1700	1524		undef	none

Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip cou
process_action	-	-	-	-	-
main_loop	no	-	-	208	-
uppercase_co	no	128	-	2	64

In this example, you can read that the loop called **main_loop** in your C code of the hardware action takes 208 cycles of 4ns clock period, meaning 820ns. It is due to a 64 iteration of a 2 cycles loop and overall logic outside the **uppercase_conversion** loop.

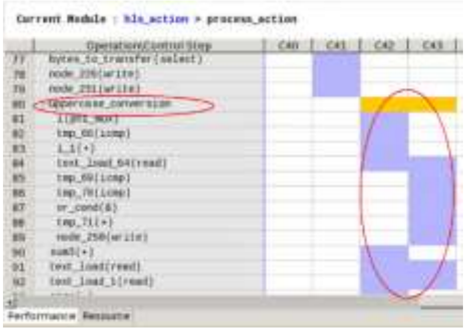
If no data are displayed, then it may be because you have loops with unknown min / max bounds. Refer to UG902 HLS guide to learn how to specify it. It will not impact your design, but gives you values for your measurements.



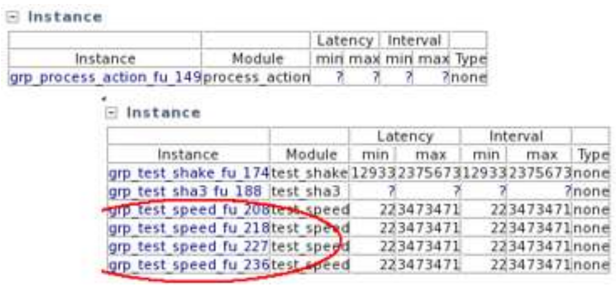
Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
grp_process_action_fu_141	process_action	?	?	?	?	none

The other view will give you a graphical representation of the operations done and if they are done sequentially or in parallel. This can be interesting to use to check in a glance what you think should be parallelized. For parallelization you can also look to the Synthesis view which wil display parallel calls.



or



Instance

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

Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
grp_test_shake_fu_174	test_shake	12933	2375673	12933	2375673	none
grp_test_sha3_fu_188	test_sha3	?	?	?	?	none
grp_test_speed_fu_208	test_speed	22	3473471	22	3473471	none
grp_test_speed_fu_218	test_speed	22	3473471	22	3473471	none
grp_test_speed_fu_227	test_speed	22	3473471	22	3473471	none
grp_test_speed_fu_236	test_speed	22	3473471	22	3473471	none

2.5 Looking for potential improvement in the code

As for a software application when looking to the CPU time used by the different functions, the goal is to see **where time is spent**.

Remember that in a FPGA, we are using pre-defined resources, but nothing is pre-connected together. In a FPGA, we can instantiate thousands of multipliers, and can use those to execute a calculation much quicker in parallel than sequential execution of the calculation on the CPU. The goal is to explain the task (hardware action) the right way to the tool, so the tool can implement the desired parallelism.

The two main constraints in a FPGA are the **availability** of the resources used (e.g. multipliers, RAM), and the physical **distance** between those resources, which may impact the clock period constraint you have set.

The Vivado HLS tool we are using is good at improving three specific pain-points of your code:

- Huge number of loops
- Math functions
- Parallelized processing

Identifying these 3 items in your code may help you when going through the obvious process:

- measure the latency of a part of the code
- try optimization
- measure the benefit immediately.

There are **two important recommendations** before starting any change in your code. These simple extra lines may save you a lot of time debugging and optimizing the code:

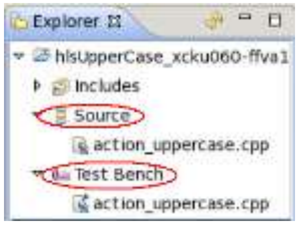
1) **Test your whole code without any optimization**. Do not insert any specific HLS optimization instructions (such as **"#pragma HLS xxx**") before your whole hardware action is tested and is functionally correct. Keep in mind that the insertion of these specific pragmas will constraint the compiler to use different algorithms which may break the functionality of your code!
2) **Build a unit test** and insert it at the bottom of your hardware action between the "#ifdef NO_SYNTH/#endif" flags so that you can test at any time if a change has broken your codes functionality

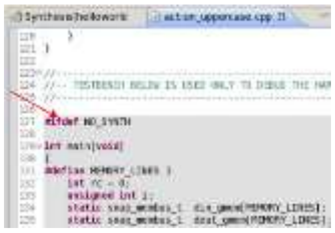
3. Measuring and optimizing hls_helloworld hardware action

3.1 Measure latency of a design with “No optimization”

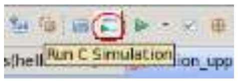
Let’s continue our work on the **hls_helloworld** hardware action named **action_uppercase.cpp.** As explained in the previous chapter, we have in the **Synthesis** tab

the same file used for the **Source code** (code synthesized) and for the **Test Bench** (used for Simulation only). Opening the Source code, you will see that the test between the “#ifdef NO_SYNTH / #endif” flags has been greyed, meaning that it will not be considered in synthesis

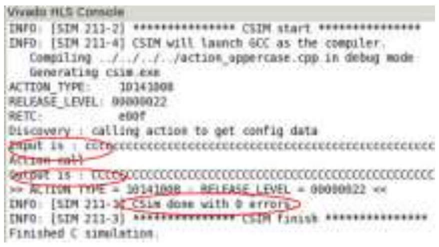




```
    }
}

//----------------------------------------------
//-- TESTBENCH BELOW IS USED ONLY TO DEBUG THE HAR
//----------------------------------------------

#ifdef NO_SYNTH

int main(void)
{
#define MEMORY_LINES 1
    int rc = 0;
    unsigned int i;
    static snap_membus_t  din_gmem[MEMORY_LINES];
    static snap_membus_t  dout_gmem[MEMORY_LINES];
```

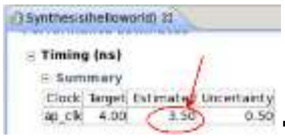
Let’s run the Simulation 

and you will get the result of your test bench:



```
Vivado HLS Console
INFO: [SIM 211-2] *************** CSIM start ***************
INFO: [SIM 211-4] CSIM will launch GCC as the compiler.
   Compiling ../../../../action_uppercase.cpp in debug mode
   Generating csim.exe
ACTION_TYPE:    10141008
RELEASE_LEVEL:  00000022
RETC:           e00f
Discovery : calling action to get config data
Input is : cccccccccccccccccccccccccccccccccccccccccccccccc
Action call
Output is : CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
>> ACTION TYPE = 10141008 - RELEASE LEVEL = 00000022 <<
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

As seen in 2.4, the 3 key data we will be looking every time are:



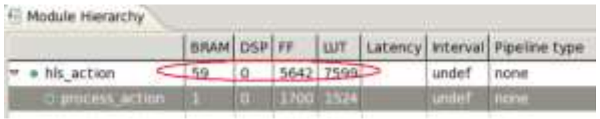
Synthesis(helloworld)

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	4.00	3.50	0.50

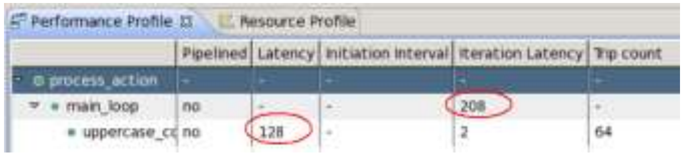
➔ to check that the clock constraint is met



Module Hierarchy

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
hls_action	59	0	5642	7599		undef	none
process_action	1	0	1700	1524		undef	none

➔ to check the amount of logic used by the change



Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
process_action	-		-	-	-
main_loop	no	-	-	208	-
uppercase_co	no	128	-	2	64

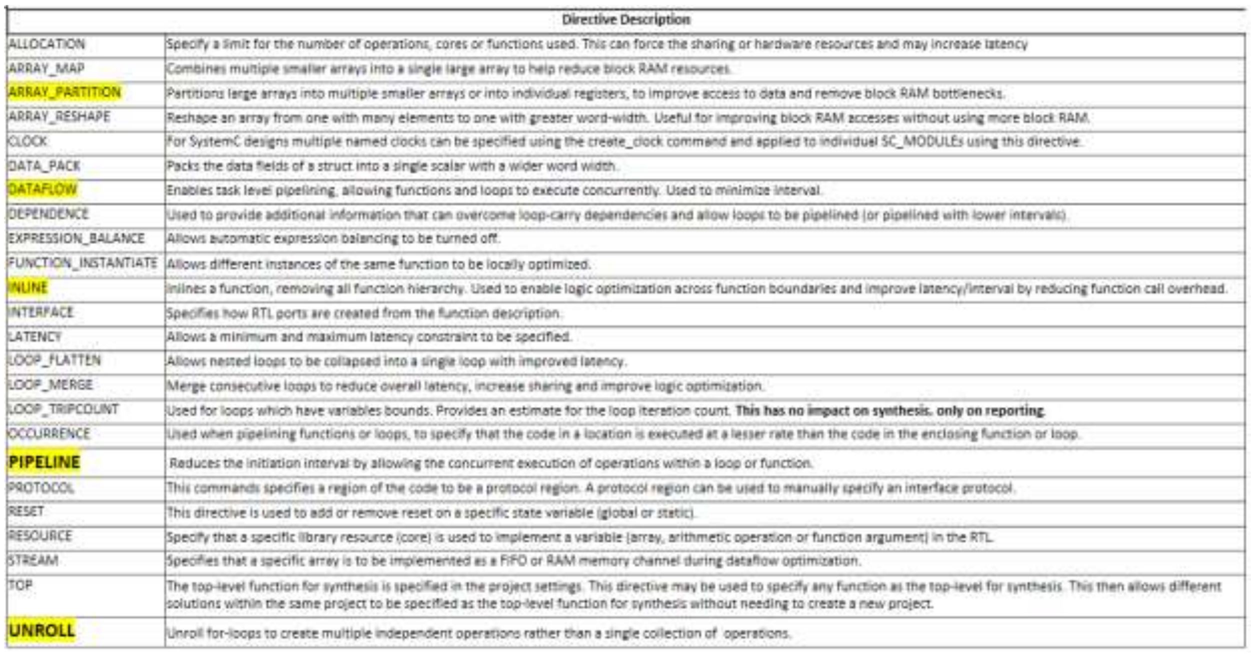
➔ to measure the overall latency

Case 1: No optimization
Main loop :
- 208 x 4ns = **832ns**
- 7599 LUTs – 5642 FF

3.2 Optimization instructions provided by Vivado HLS

A full reference of these instructions is detailed in HLS User Guide UG902 (HLS 2017.4 release).

24 optimization directives are listed. 5 major ones are highlighted below but we will use only 2 basic ones for this example.



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.

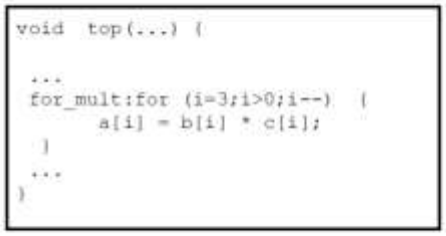
Important to know: adding a pragma may reorder all the generated RTL code so that you won't recognize your variables in debug mode.

3.2.1 #pragma HLS UNROLL (factor=2)

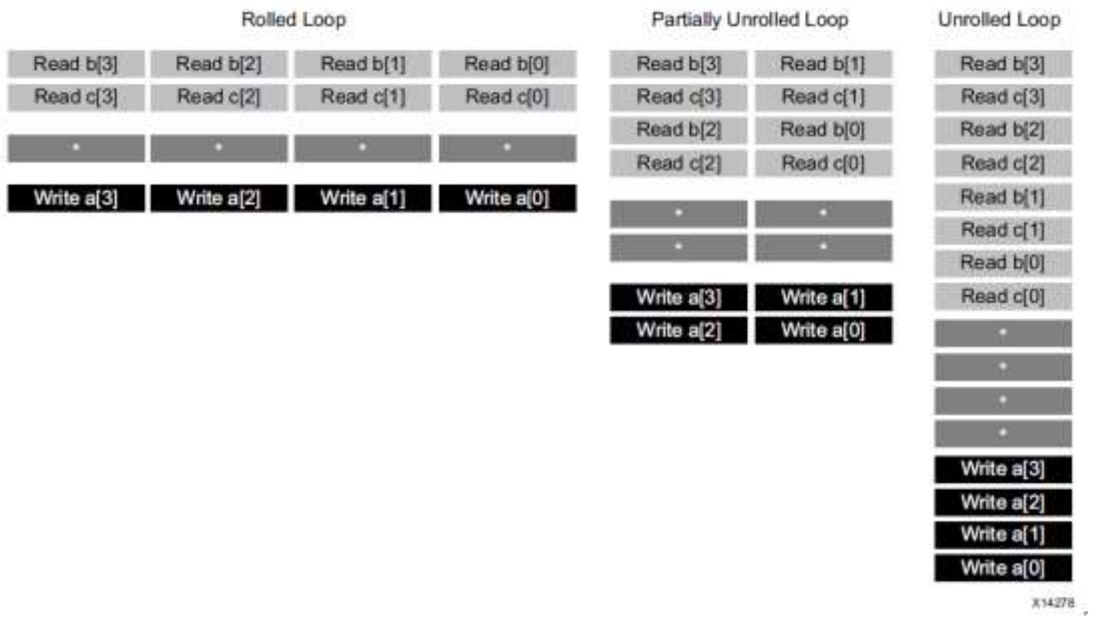
UNROLLING means flattening a loop so that all iterations are executed in one cycle. The factor value controls the unroll; default is a maximum unroll. But as nothing is magic, unrolling a loop means that you are duplicating the logic used, so potentially your design takes much more FPGA resources. This UNROLL pragma is inserted into the loop.

This feature is very useful to parallelize high level functions as soon as they are independent, meaning not waiting for the value of the previous iteration to start a new one.

This pragma is inserted in the code at the top of a function or inside the loop



```
void  top(...) {

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



Xilinx HLS UG902 (v2017.4) December 20, 2017: figure 1-60

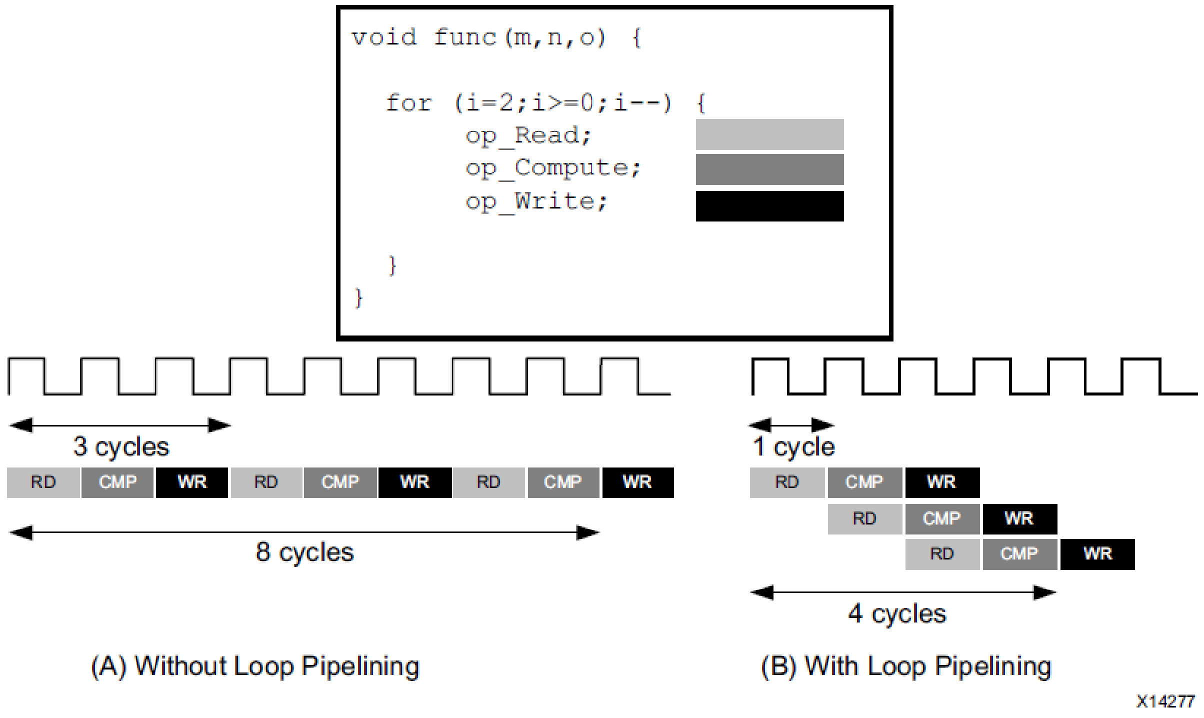
3.2.2 #pragma HLS PIPELINE

A PIPELINE directive will first flatten the design (unrolling loops). Then it will look at 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.

In this example, the compiler with this pipeline instruction will understand that the next **RD** can be done during the **CMP** of the previous data. There is no reason to wait for completion of the full sequence. The **initiation interval** (II) which is the time between 2 reads will so be reduced from 3 to 1 in this case. The **overall latency** will so be reduced from 8 to 4 cycles.

PIPELINE is a recursive function, so handle it with care since it may add a huge amount of logic for minimally better latency!

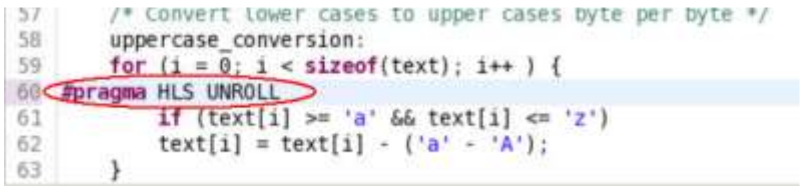
PIPELINE can be very good for math processing. If it's not, that can be due to the way your code is written.



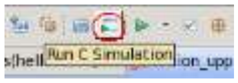
Xilinx HLS UG902 (v2017.4) December 20, 2017: figure 1-51

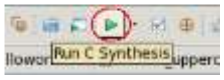
3.3 Measure latency of a design with an UNROLL instruction in a loop

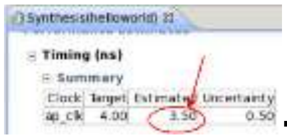
Let's start a first optimization to understand the effect of a simple UNROLL instruction. Open the ***action_uppercase.cpp*** file located in Source and uncomment the #pragma HLS UNROLL on line 60.



```
57          /* Convert lower cases to upper cases byte per byte */
58          uppercase_conversion:
59          for (i = 0; i < sizeof(text); i++ ) {
60      #pragma HLS UNROLL
61              if (text[i] >= 'a' && text[i] <= 'z')
62              text[i] = text[i] - ('a' - 'A');
63          }
```

Save the file, run the C simulation  and check that your test bench gives you the right result.

Now run the C synthesis  and get the key numbers as for 3



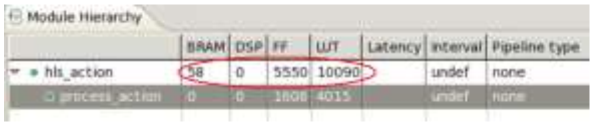
Synthesis(helloworld)

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	4.00	3.50	0.50

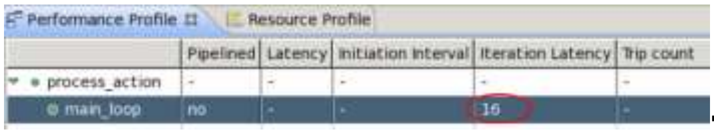
➔ ok



Module Hierarchy

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
hls_action	58	0	5550	10090		undef	none
process_action	0	0	1808	4015		undef	none

➔ x1.3 more LUTS – a bit less FFs➔ more logic used



Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
process_action	-	-	-	-	-
main_loop	no	-	-	16	-

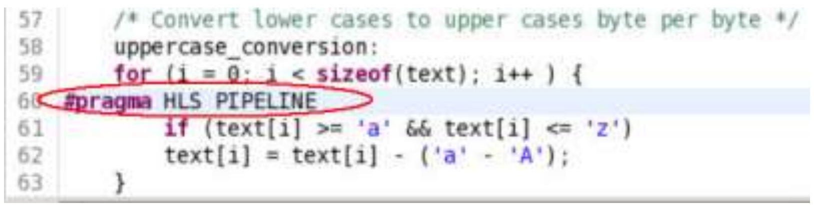
➔ The "uppercase_conversion" loop has disappeared, and the overall latency was reduced by 13!

Case 1: No optimization
Main loop :
- 208 x 4ns = **832ns**
- 7599 LUTs – 5642 FF

Case 2: UNROLL in uppercase_conversion loop
Main loop :
- 16 x 4ns = **64ns (x13)**
- 10090 LUTs – 5550 FF (x1.3)

➔ The UNROLL instruction has been able to reduce the latency a lot but is using one third more logic. Using this case will significantly improve the performance of the action but depending on the size of the overall function to implement, this amount of logic added may become a constraint.

Let's try now the PIPELINE instruction at the same location, replacing the UNROLL instruction.



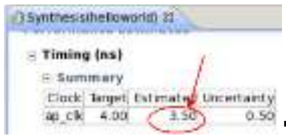
```
    /* Convert lower cases to upper cases byte per byte */
    uppercase_conversion:
    for (i = 0; i < sizeof(text); i++ ) {
#pragma HLS PIPELINE
        if (text[i] >= 'a' && text[i] <= 'z')
        text[i] = text[i] - ('a' - 'A');
    }
```

Save the file, run the C simulation

and check that your test bench gives you the right result.

Now run the C synthesis

and get the key numbers as for 3.



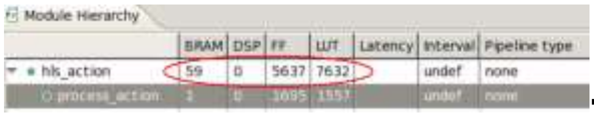
Synthesis(helloworld)

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	4.00	3.50	0.50

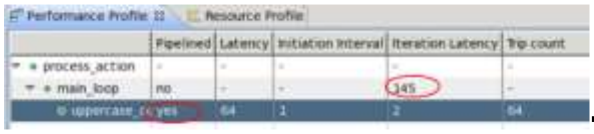
➔ ok



Module Hierarchy

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
hls_action	59	0	5637	7632		undef	none
process_action	1	0	1695	1557		undef	none

➔ very similar to case 1 ➔ no loss



Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
process_action	-	-	-	-	-
main_loop	no	-	-	145	-
uppercase_c	yes	64	1	2	64

➔ The overall latency was reduced by 1.4

Case 1: No optimization
<u>Main loop:</u>
- 208 x 4ns = **832ns**
- 7599 LUTs – 5642 FF

Case 2: UNROLL in uppercase_conversion loop
<u>Main loop:</u>
- 16 x 4ns = **64ns (13x better)**
- 10090 LUTs – 5550 FF (x1.3)

Case 3: PIPELINE in uppercase_conversion loop
<u>Main loop:</u>
- 145 x 4ns = **580ns (1.4x better)**
- 7632 LUTs – 5637 FF (#same)

➔ Note that comparing to case 1, the PIPELINE instruction has been able to reduce the latency a bit without taking more logic! Depending on the performance you need and the size of your design, it can be interesting to use this option which slightly improves your performance without taking more logic in the FPGA.

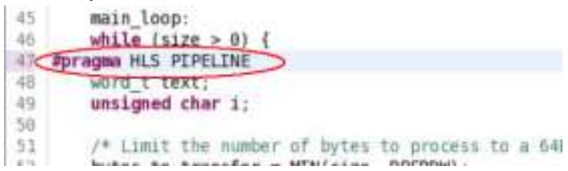
You may notice the following (false) warning during the synthesis. It is related to "main_loop".

```
WARNING: [XFORM 203-542] Cannot flatten a loop nest 'main_loop' (action_uppercase.cpp:46:23) in function 'process_action'
the outer loop is not a perfect loop because there is nontrivial logic in the loop latch.
```

This is a side effect of the PIPELINE directive but no reason to worry, Result is confirmed as ok:

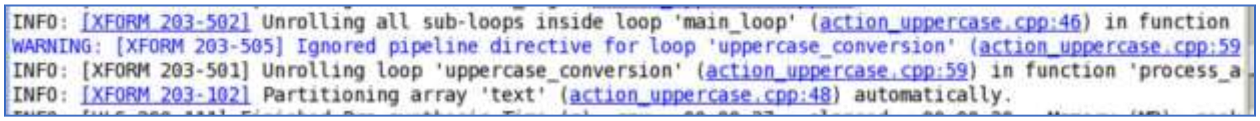
```
INFO: [SCHED 204-61] Pipelining loop 'uppercase_conversion'.
INFO: [SCHED 204-61] Pipelining result : Target II = 1, Final II = 1, Depth = 2.
```

Let’s try now the PIPELINE instruction inside the **main_loop** instead of inside a sub loop.

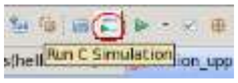


```
    main_loop:
    while (size > 0) {
#pragma HLS PIPELINE
    word_t text;
    unsigned char i;

    /* Limit the number of bytes to process to a 64
```

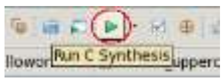
As the PIPELINE is recursive, keeping or not the pragma defined ealier in **uppercase_conversion** loop will have no effect.



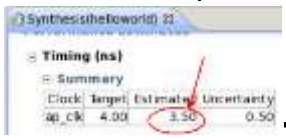
```
INFO: [XFORM 203-502] Unrolling all sub-loops inside loop 'main_loop' (action_uppercase.cpp:46) in function
WARNING: [XFORM 203-505] Ignored pipeline directive for loop 'uppercase_conversion' (action_uppercase.cpp:59
INFO: [XFORM 203-501] Unrolling loop 'uppercase_conversion' (action_uppercase.cpp:59) in function 'process_a
INFO: [XFORM 203-102] Partitioning array 'text' (action_uppercase.cpp:48) automatically.
```

Save the file, run the C simulation 

and check that your test bench gives you the right result.

Now run the C synthesis 

and get the key numbers as for 3



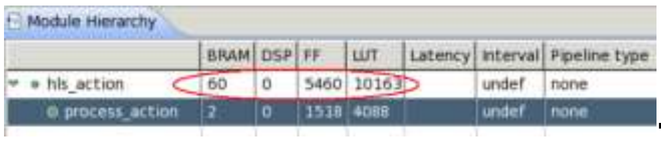
Synthesis(helloworld)

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	4.00	3.50	0.50

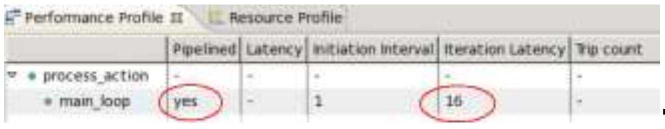
➔ ok



Module Hierarchy

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
hls_action	60	0	5460	10163		undef	none
process_action	2	0	1518	4088		undef	none

➔ very similar to case 2 ➔ x1.3 more LUTs



Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
process_action	-	-	-	-	-
main_loop	yes	-	1	16	-

➔ As for case 2, the uppercase_conversion loop has disappeared and the overall latency was reduced by 13 !

Case 1: No optimization
<u>Main loop:</u>
- 208 x 4ns = **832ns**
- 7599 LUTs – 5642 FF

Case 2: UNROLL in uppercase_conversion loop
<u>Main loop:</u>
- 16 x 4ns = **64ns (13x better)**
- 10090 LUTs – 5550 FF (x1.3)

Case 3: PIPELINE in uppercase_conversion loop
<u>Main loop:</u>
- 145 x 4ns = **580ns (1.4x better)**
- 7632 LUTs – 5637 FF (#same)

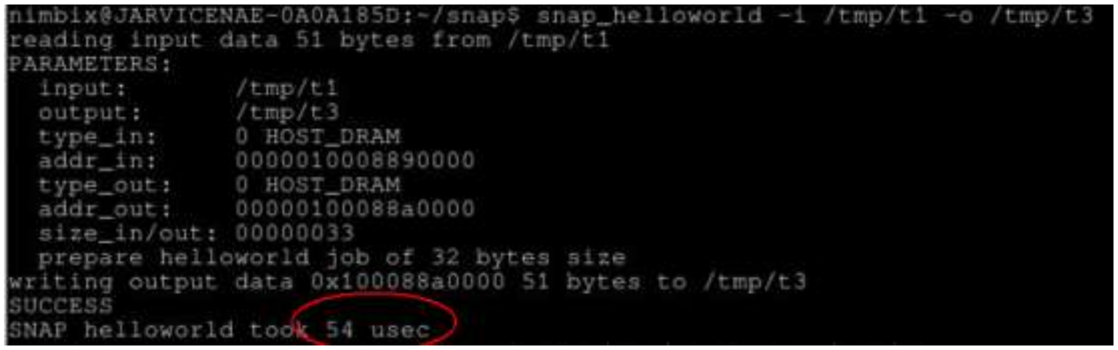
Case 4: PIPELINE in main loop
<u>Main loop:</u>
- 16 x 4ns = **64ns (13x better)**
- 10163 LUTs – 5460 FF (x1.3)

➔ The PIPELINE instruction inserted inside the top "main_loop" or at the top of the function will be able to have the same effect as a simple UNROLL instruction located below in the logic (Case 2). Inserting a PIPELINE directive can drive to situations where the latency and the size of the logic generated gives opposite results than the one you would expect. This is due to the way the algorithm tries to optimize your code if he does a bad analysis path. Rewriting the code differently or placing the PIPELINE instruction elsewhere may solve the issue.

4. What represents the times values measured?

4.1 What is included in the time reported by the application running the action on a real FPGA?

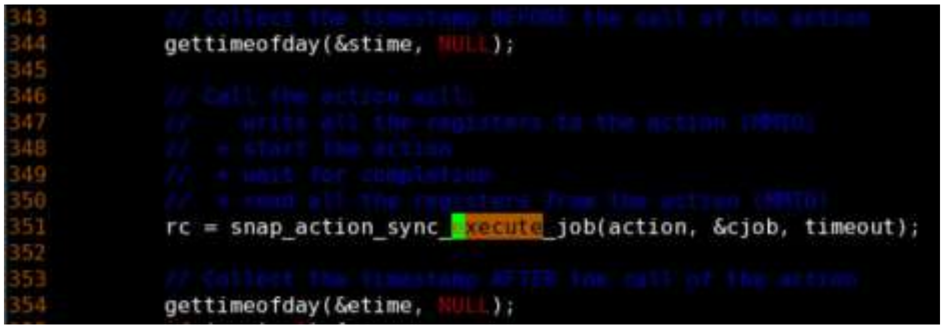
Let’s use the non-optimized release and look to the results returned by the execution of the hls_helloworld on a real FPGA:



```
nimbix@JARVICENAE-0A0A185D:~/snap$ snap_helloworld -i /tmp/t1 -o /tmp/t3
reading input data 51 bytes from /tmp/t1
PARAMETERS:
  input:       /tmp/t1
  output:      /tmp/t3
  type_in:     0 HOST_DRAM
  addr_in:     0000010008890000
  type_out:    0 HOST_DRAM
  addr_out:    00000100088a0000
  size_in/out: 00000033
  prepare helloworld job of 32 bytes size
writing output data 0x100088a0000 51 bytes to /tmp/t3
SUCCESS
SNAP helloworld took 54 usec
```

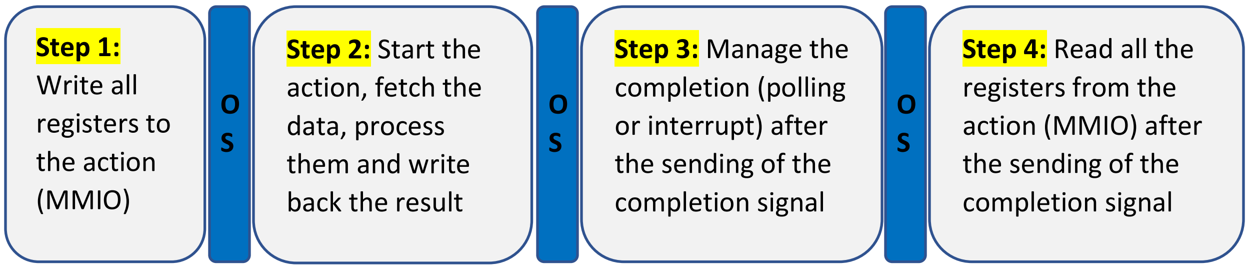
Result is 54µs, while execution in HLS shows 832ns!! That’s a great gap! **So, let’s try and understand these different numbers and what we are measuring.**

Looking to the application code ***hls_helloworld/sw/snap_helloworld.c,*** we measure the time the “function” is called and returns a return code. This “function” can either be the software or the hardware action depending on the switch we used. In this case, we used the hardware action.



```
        // Collect the timestamp BEFORE the call of the action
        gettimeofday(&stime, NULL);

        // Call the action will:
        //    write all the registers to the action (MMIO)
        //  + start the action
        //  + wait for completion
        //  + read all the registers from the action (MMIO)
        rc = snap_action_sync_execute_job(action, &cjob, timeout);

        // Collect the timestamp AFTER the call of the action
        gettimeofday(&etime, NULL);
```

The snap_action_sync_execute_job, will successively go through 4 different steps:



Step 1: Write all registers to the action (MMIO)

Step 2: Start the action, fetch the data, process them and write back the result

Step 3: Manage the completion (polling or interrupt) after the sending of the completion signal

Step 4: Read all the registers from the action (MMIO) after the sending of the completion signal

These 4 steps are done successively but are completely independent actions, meaning that **they will be handled by SNAP libraries through the Operating systems as 4 independent tasks.**

To have a better view, let’s use the Simulator capabilities which gives a better view of all steps.

Before starting the explanation, **let's understand what we measure with a simulator**. The simulator is working with a model named **PSLSE** (PSL Simulation Engine) which provides answers to the hardware action, as if we have a real Power8 + PSL answering. This model allows the simulator to see exact answers that could be provided by an application running on a Power8 through a PSL, but **this model is NOT simulating** in any case the time taken by the Operating System nor all levels between the application and the hardware action.

In other words, we can see how long read or writes to registers takes, how long the processing of the hardware action takes but we are missing the real timing of the real OS and firmware.

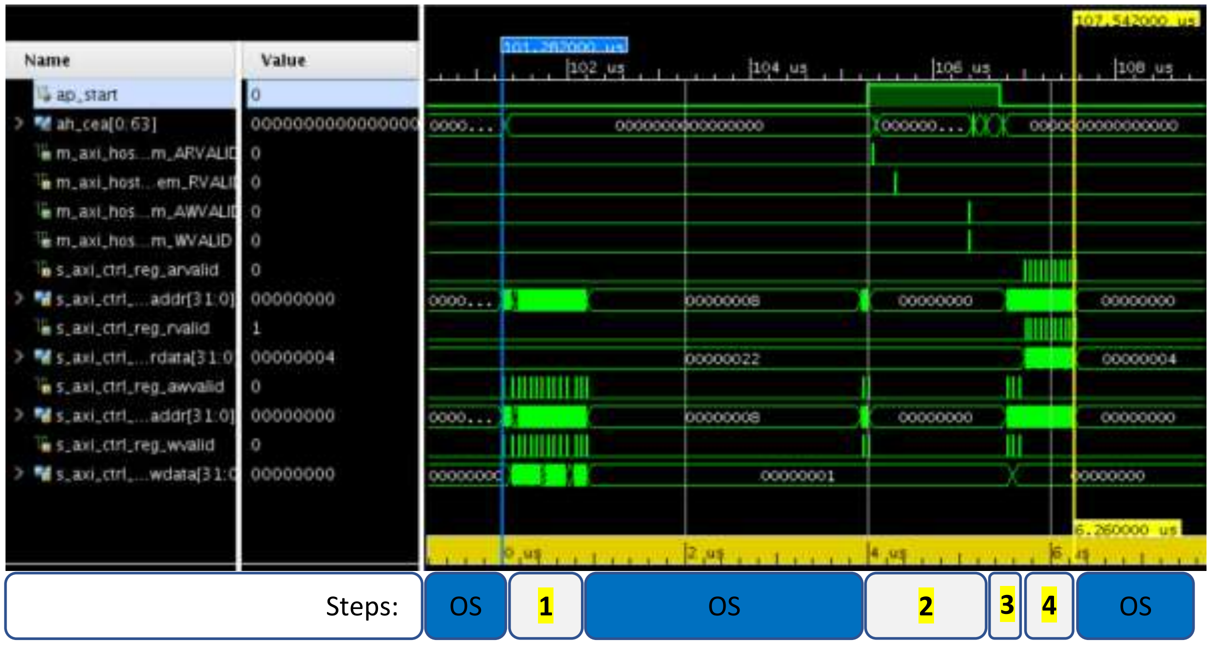
After having run a full simulation, we can display the waveforms using different simulators. Let's use the default Xilinx Vivado simulator (available on Nimbix) using the following command:

```
cd $SNAP_ROOT/hardware/sim
xsim -gui xsim/latest/top.wdb
```

Choosing specific signals allows you to see all the operations in one glance. Let's explain them:

- The first signal ***ap_start*** shows a value "set to 1" when the hardware action is enabled.
- The second signal ***ah_cea*** shows the access by the hardware to the server memory (through the PSLSE simulator).
- All signals starting by ***m_axi_host*** shows the activity to and from the server memory memory (through the PSLSE simulator).

All signals starting by ***s_axi_ctrl_reg*** shows the MMIO activity, meaning the read and write of the MMIO registers.

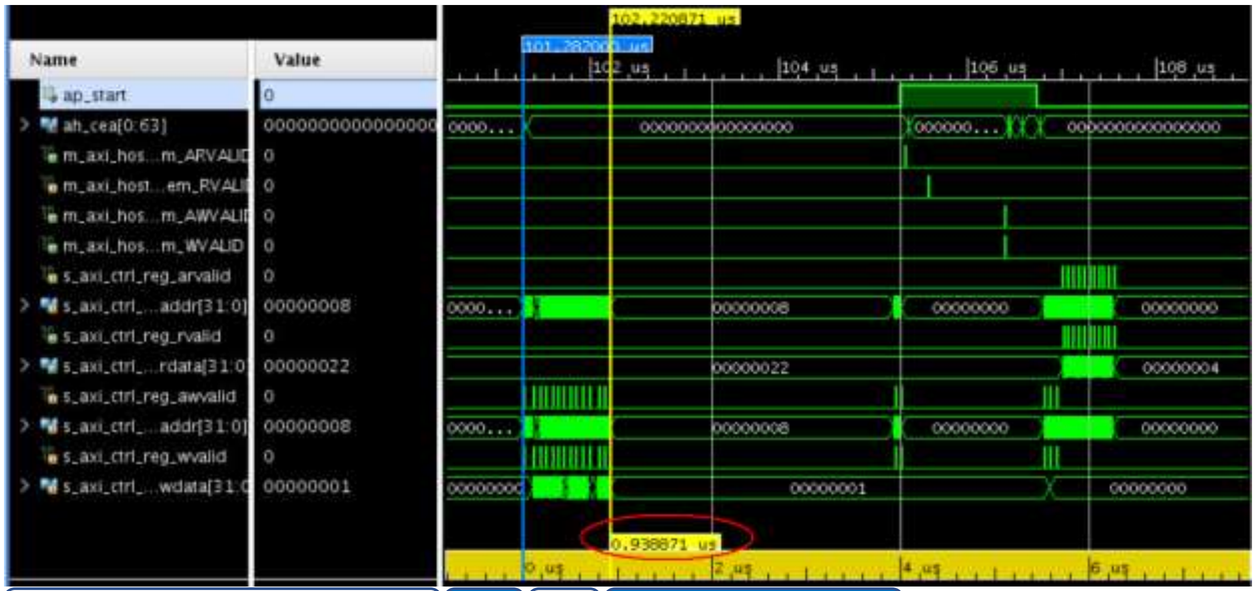


Name	Value
ap_start	0
ah_cea[0:63]	0000000000000000
m_axi_hos...m_ARVALID	0
m_axi_host...em_RVALID	0
m_axi_hos...m_AWVALID	0
m_axi_hos...m_WVALID	0
s_axi_ctrl_reg_arvalid	0
s_axi_ctrl_...addr[31:0]	00000000
s_axi_ctrl_reg_rvalid	1
s_axi_ctrl_...rdata[31:0]	00000004
s_axi_ctrl_reg_awvalid	0
s_axi_ctrl_...addr[31:0]	00000000
s_axi_ctrl_reg_wvalid	0
s_axi_ctrl_...wdata[31:0]	00000000

4.3 Step 1: Write all registers to the action (MMIO)

Writing all registers to the action (MMIO) takes roughly 1µs. Once this is done, the hardware action is all set but not started. The Operating System will send in Step 2 the “START” order.

This step happens **once** to configure the hardware action.



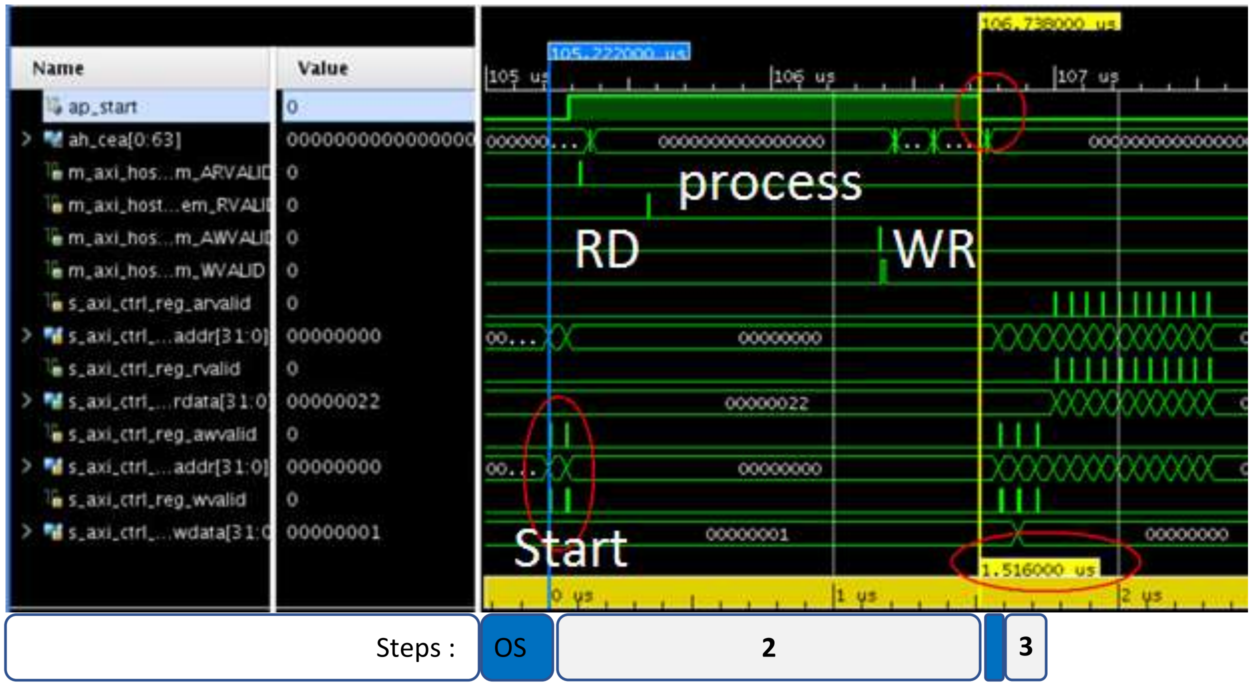
Name	Value
ap_start	0
ah_cea[0:63]	0000000000000000
m_axi_hos...m_ARVALID	0
m_axi_host...em_RVALID	0
m_axi_hos...m_AWVALID	0
m_axi_hos...m_WVALID	0
s_axi_ctrl_reg_arvalid	0
s_axi_ctrl_...addr[31:0]	00000008
s_axi_ctrl_reg_rvalid	0
s_axi_ctrl_...rdata[31:0]	00000022
s_axi_ctrl_reg_awvalid	0
s_axi_ctrl_...addr[31:0]	00000008
s_axi_ctrl_reg_wvalid	0
s_axi_ctrl_...wdata[31:0]	00000001

Steps: OS 1

4.4 Step 2: Start the action, and process the data with the hardware action

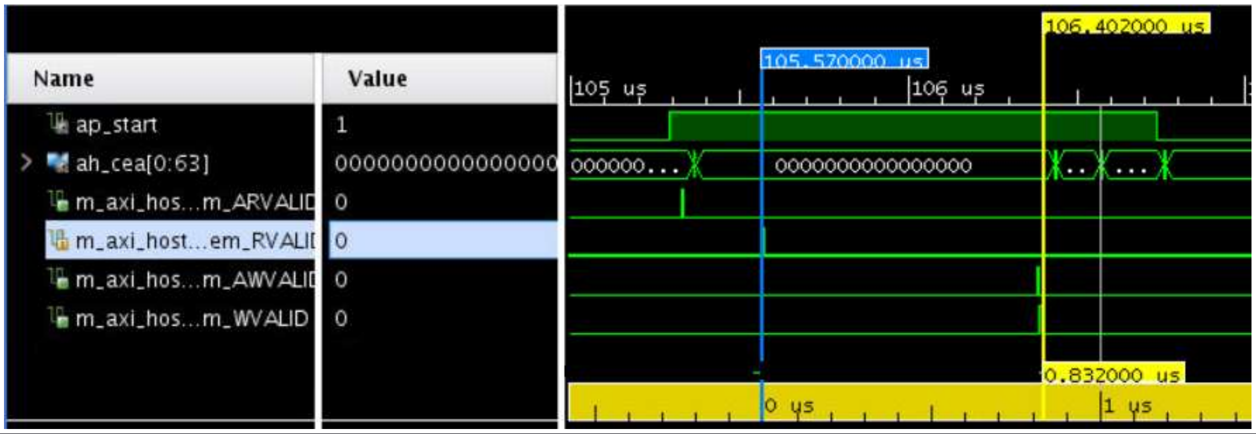
The different actions done in Step 3 takes roughly 1.5µs split as follow:

- The START signal is written by the application in the MMIO register.
  ➔ This triggers the start of the hardware action
- Data are read (between rising edge of ap_start and RVALID)
- Data are processed (between RVALID and AWVALID)
- Result is written (between AWVALID and falling edge of ap_start)



Name	Value
ap_start	0
ah_cea[0:63]	0000000000000000
m_axi_hos...m_ARVALID	0
m_axi_host...em_RVALID	0
m_axi_hos...m_AWVALID	0
m_axi_hos...m_WVALID	0
s_axi_ctrl_reg_arvalid	0
s_axi_ctrl_...addr[31:0]	00000000
s_axi_ctrl_reg_rvalid	0
s_axi_ctrl_...rdata[31:0]	00000022
s_axi_ctrl_reg_awvalid	0
s_axi_ctrl_...addr[31:0]	00000000
s_axi_ctrl_reg_wvalid	0
s_axi_ctrl_...wdata[31:0]	00000001

➔ The time given by HLS (832ns) is perfectly coherent with the time given by the simulator when not including the time to access the data.

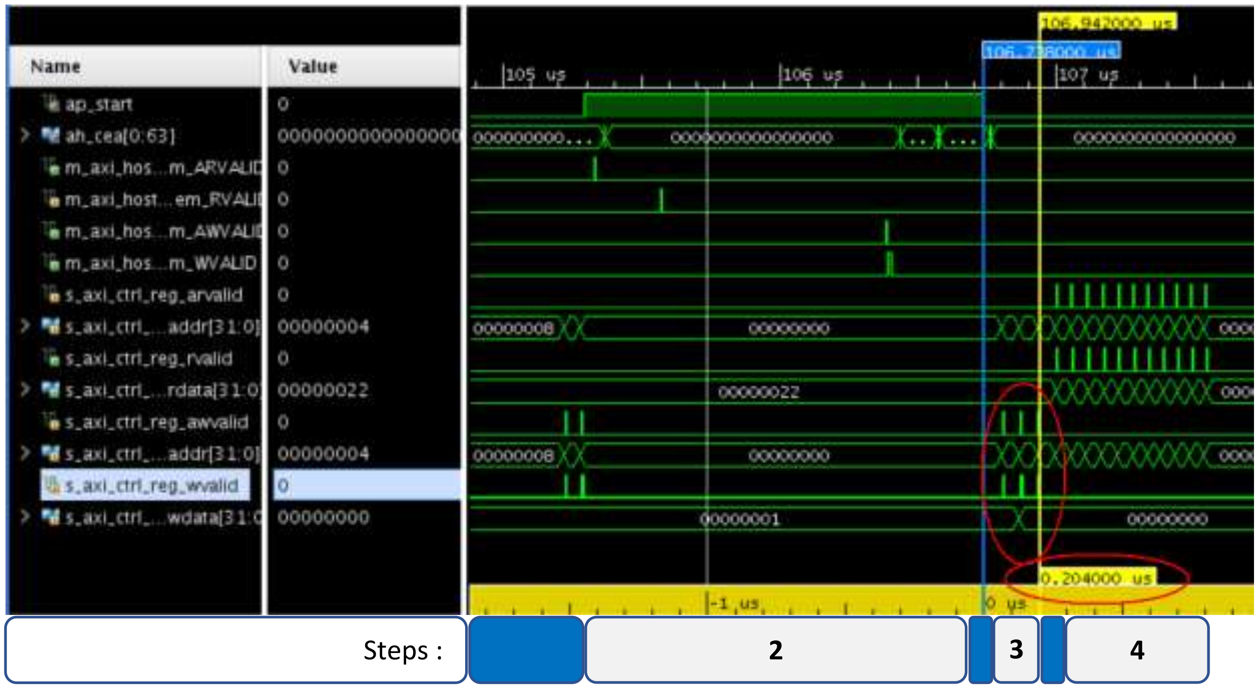


Name	Value
ap_start	1
ah_cea[0:63]	0000000000000000
m_axi_hos...m_ARVALID	0
m_axi_host...em_RVALID	0
m_axi_hos...m_AWVALID	0
m_axi_hos...m_WVALID	0

In this example, we have chosen the default interrupt mode for handling the completion of the hardware action. This option can be modified in the application to use the polling mode.

When the action is completed, the ap_start signal is disabled. This sends an interrupt to the Operating System which will identify the source of this interrupt and the SNAP library will clear the interrupt.

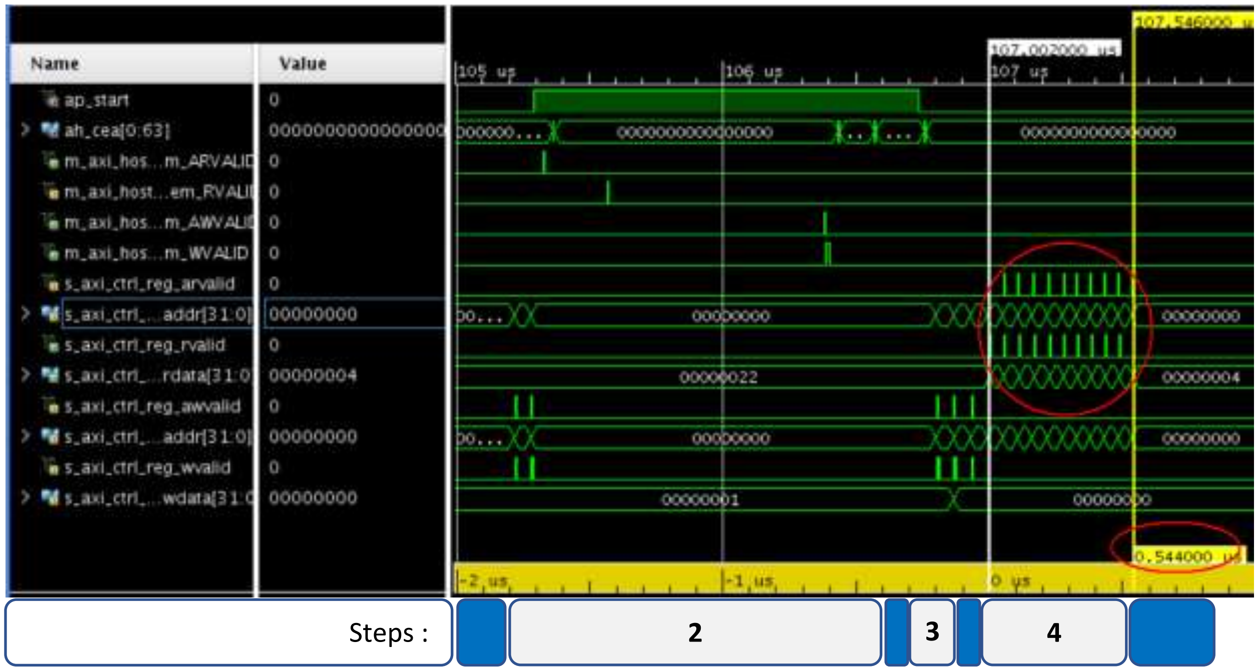
This is done **once** after the execution of the hardware action.



Name	Value
ap_start	0
ah_cea[0:63]	0000000000000000
m_axi_hos…m_ARVALID	0
m_axi_host…em_RVALID	0
m_axi_hos…m_AWVALID	0
m_axi_hos…m_WVALID	0
s_axi_ctrl_reg_arvalid	0
s_axi_ctrl_…addr[31:0]	00000004
s_axi_ctrl_reg_rvalid	0
s_axi_ctrl_…rdata[31:0]	00000022
s_axi_ctrl_reg_awvalid	0
s_axi_ctrl_…addr[31:0]	00000004
s_axi_ctrl_reg_wvalid	0
s_axi_ctrl_…wdata[31:0]	00000000

4.6 Step 4: Read all registers of the action (MMIO)

The last operation done by the SNAP libraries is a complete read of the action registers. This step takes roughly 0.5µs and is done **once** at the end of the end of the action.

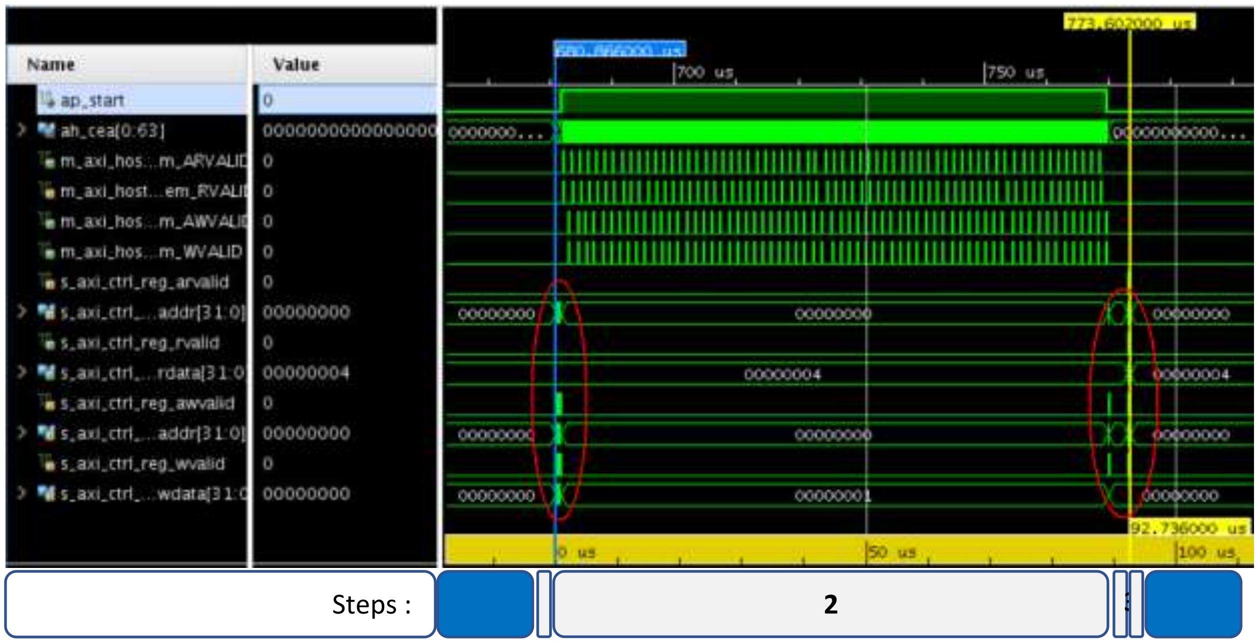


Name	Value
ap_start	0
ah_cea[0:63]	0000000000000000
m_axi_hos...m_ARVALID	0
m_axi_host...em_RVALID	0
m_axi_hos...m_AWVALID	0
m_axi_hos...m_WVALID	0
s_axi_ctrl_reg_arvalid	0
s_axi_ctrl_...addr[31:0]	00000000
s_axi_ctrl_reg_rvalid	0
s_axi_ctrl_...rdata[31:0]	00000004
s_axi_ctrl_reg_awvalid	0
s_axi_ctrl_...addr[31:0]	00000000
s_axi_ctrl_reg_wvalid	0
s_axi_ctrl_...wdata[31:0]	00000000

4.7 Conclusion: is the overhead always harmful?

This hls_helloworld example is going through all the SNAP process with just a few bytes read and written. Indeed, an overhead of 53µs seems huge versus the 1µs of processing. Now, this example is not reflecting the reality of offloading large processing tasks (meaning there is no interest to offload such little processing).

The overhead (Step1+Step3+Step4) can become totally negligible if the processing (Step 2) becomes more important. Just increasing the size of the text processed by hls_helloworld confirms this.



Name	Value
ap_start	0
ah_cea[0:63]	0000000000000000
m_axi_hos...m_ARVALID	0
m_axi_host...em_RVALID	0
m_axi_hos...m_AWVALID	0
m_axi_hos...m_WVALID	0
s_axi_ctrl_reg_arvalid	0
s_axi_ctrl_...addr[31:0]	00000000
s_axi_ctrl_reg_rvalid	0
s_axi_ctrl_...rdata[31:0]	00000004
s_axi_ctrl_reg_awvalid	0
s_axi_ctrl_...addr[31:0]	00000000
s_axi_ctrl_reg_wvalid	0
s_axi_ctrl_...wdata[31:0]	00000000

We can notice that processing the data by bursts rather than by reading and writing single words would improve a lot the behavior of this hls_helloworld example.

4.8 Conclusion: is there a way to remove this overhead?

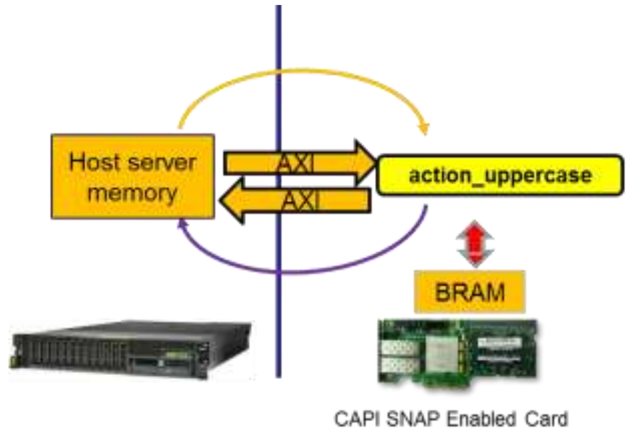
Indeed, hls_helloworld is using the standard – and easy to use – ***snap_action_sync_execute_job*** API in the application code. This API sets the registers, start the action and wait for the completion of the action. As we can see, it is not optimized for exchanging data quickly and efficiently between the application and the action. Having an application reporting a 54µs latency while action processing is taking only 1µs, could be seen as a waste of time taken by what we called the OS in this chapter.

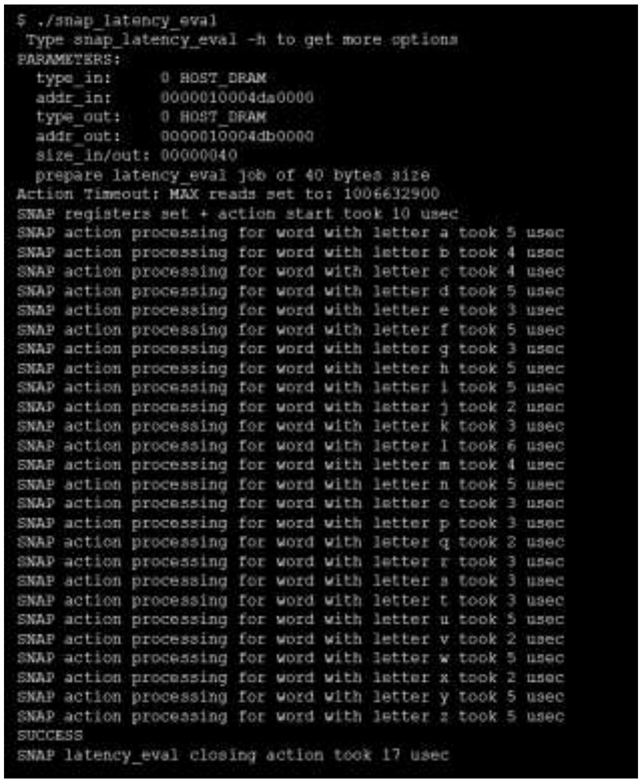
Now there are plenty of ways to optimize this latency. We just need to:

- code the application such as:
  - data are sent directly to the host memory without cache delays ➔ use volatile variables
  - no system calls are done in the code that needs OS ➔ this would impact the latency
  - ***snap_action_sync_execute_job*** API is replaced by the 4 following steps
    - snap_action_sync_execute_job_set_regs
    - snap_action_start
    - implement/measure the following sequence: test action status + fill host memory + poll host memory and compare the data read to the expected data
    - snap_action_sync_execute_job_check_completion
- code the action such as:
  - action data are read and written continuously from/to the host memory
  - action ends successfully on a specific sequence
  - implement a timeout that ends the action independently from the processing

The way to implement that is described in the example https://github.com/open-power/snap/tree/master/actions/hls_latency_eval

Measurements shows an average of 3µs for an application to have processed data into the FPGA




```
$ ./snap_latency_eval
 Type snap_latency_eval -h to get more options
PARAMETERS:
  type_in:      0 HOST_DRAM
  addr_in:      0000010004da0000
  type_out:     0 HOST_DRAM
  addr_out:     0000010004db0000
  size_in/out:  00000040
  prepare latency_eval job of 40 bytes size
Action Timeout: MAX reads set to: 1006632900
SNAP registers set + action start took 10 usec
SNAP action processing for word with letter a took 5 usec
SNAP action processing for word with letter b took 4 usec
SNAP action processing for word with letter c took 4 usec
SNAP action processing for word with letter d took 5 usec
SNAP action processing for word with letter e took 3 usec
SNAP action processing for word with letter f took 5 usec
SNAP action processing for word with letter g took 3 usec
SNAP action processing for word with letter h took 5 usec
SNAP action processing for word with letter i took 5 usec
SNAP action processing for word with letter j took 2 usec
SNAP action processing for word with letter k took 3 usec
SNAP action processing for word with letter l took 6 usec
SNAP action processing for word with letter m took 4 usec
SNAP action processing for word with letter n took 5 usec
SNAP action processing for word with letter o took 3 usec
SNAP action processing for word with letter p took 3 usec
SNAP action processing for word with letter q took 2 usec
SNAP action processing for word with letter r took 3 usec
SNAP action processing for word with letter s took 3 usec
SNAP action processing for word with letter t took 3 usec
SNAP action processing for word with letter u took 5 usec
SNAP action processing for word with letter v took 2 usec
SNAP action processing for word with letter w took 5 usec
SNAP action processing for word with letter x took 2 usec
SNAP action processing for word with letter y took 5 usec
SNAP action processing for word with letter z took 5 usec
SUCCESS
SNAP latency_eval closing action took 17 usec
```

5. Summary

Before a real application modified to use an FPGA for acceleration, analysis must be done to judge potential performance benefits of FPGA usage, such that expected parallelism and advantages by using pipelined data processing. The analysis needs also to consider latency overhead involved when starting the FPGA action and waiting for its completion.

This document explained how the performance of the hardware accelerated action on the FPGA can be analyzed and further optimized. The SNAP framework helps to cut out performance critical parts of an application and enables their execution on the FPGA.