Ackermann Function Implementation

- The Ackermann function is notable for being one of the simplest examples of a total, computable function that isn't primitive recursive.
- **Primitive Recursive functions:** functions that are computable whose upper bounds are known and which use only ‘for’ loops.
- We will use the definition of A(m,n) taking in two nonnegative integers where A(0,n) = n+1 A(m,0) = A(m-1,1) A(m,n) = A(m-1,A(m,n-1))
- The Ackermann function is one of the most important functions in computer science. Its most outstanding property is that it grows astonishingly fast.
- In fact, it gives rise to such large numbers very quickly that these numbers, called Ackermann numbers, are written in a special way known as Knuth's up-arrow notation.
- Two positive integers, m, and n are the input and A(m, n) is the output in the form of another positive integer.
- The function can be programmed easily in just a few lines of code.
- The problem isn't the complexity of the function but the awesome rate at which it grows. For example, the innocuous-looking A(4,2) already has 19,729 digits!
- The use of a powerful large-number shorthand system, such as the up-arrow notation, is indispensable as the following examples show:

  A(1, n) = 2 + (n + 3) - 3

  A(2, n) = 2 × (n + 3) - 3

  A(3, n) = 2^(n + 3) - 3

  A(4, n) = 2^(2^(2^ (...^2))) - 3 (n + 3 twos) = 2^^(n + 3) - 3

  A(5, n) = 2^^^(n + 3) – 3, etc.

- Intuitively, the Ackermann function defines generalizations of multiplication by two (iterated additions) and exponentiation with base 2 (iterated multiplications) to iterated exponentiation, iteration of this operation, and so on.

	n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10	n=11	n=12	n=13	n=14
m=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
m=1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
m=2	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
m=3	5	13	29	61	125	253	509	1021	2045	4093	8189	16381	32765	65533	131069
m=4	13	65533	-	-	-	-	-	-	-	-	-	-	-	-	-

Work Flow:

Step 1: Create a new project: Open a new project in Vivado HLS. Select the board part *xc7z010clg400-1* which corresponds to the Zybo board. Then, import the source code and its corresponding test bench and open the new project.

Step 2: C Simulation: Simulate the code by clicking on the Run C Simulation icon. This simulates the source code and checks its functionality using the values provided in the testbench.

Step 3: C/RTL Co-Simulation: The C/RTL Co-Simulation checks the behavior of the program with respect to the hardware RTL design.

Step 4: Synthesis: Synthesis plays a vital role in analyzing the performance of the designed RTL with respect to the inputs given. For analyzing the efficiency of the Ackerman function, all possible inputs need to be considered. The inputs considered for 'm' vary between 1 and 3 and for 'n' is between 1 and 15. Below cases are attached as only limited screenshots can be attached to the report.

- m = 0, n = 2
- m = 2, n = 14
- m = 3, n = 14

Step 5: Export RTL: The designed RTL is then exported as the custom IP so that it can be integrated with the Zynq PS. This process converts the C++ code into Verilog. This is then imported into Vivado IDE.

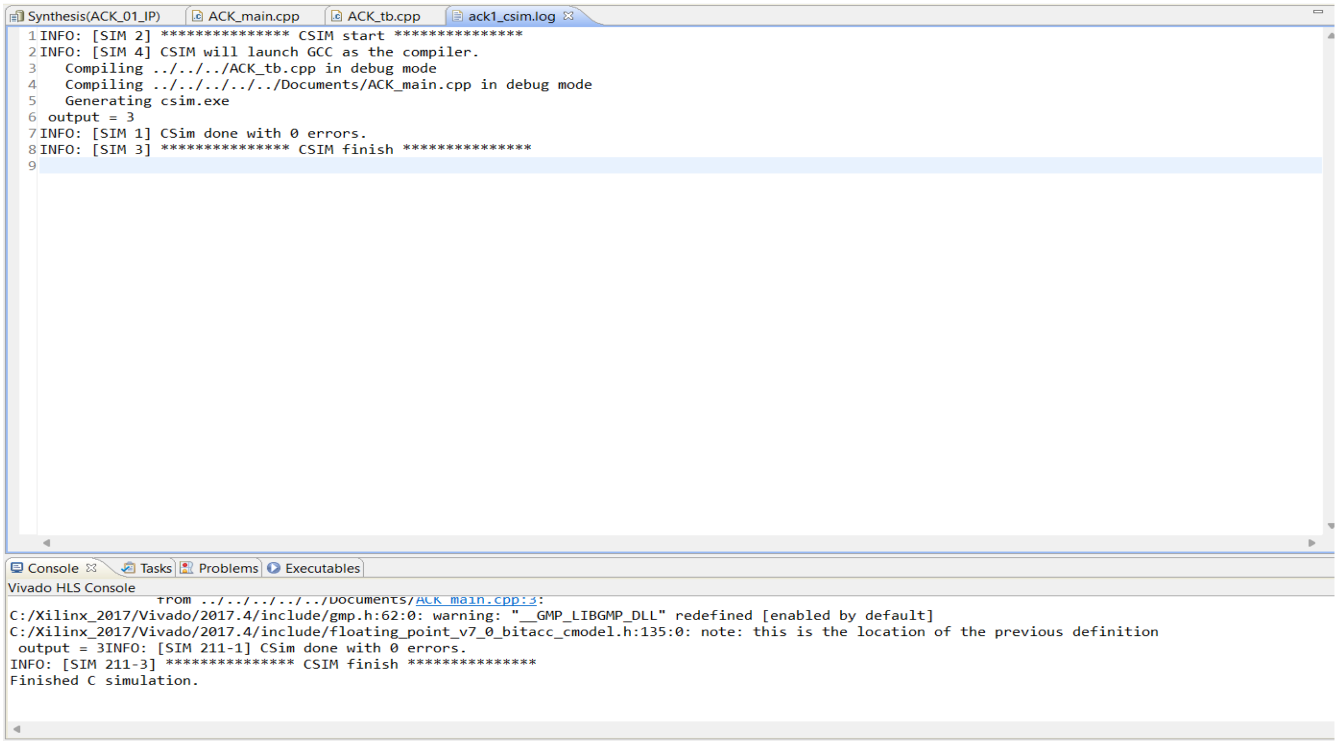
Step 6: IP Integrator Design using Vivado IDE: Create a new project in Vivado IDE specifying the Zybo Board file from the list of boards. Add the custom IP to the IP repository. Open a new block design and then place the newly exported custom IP along with the ZYNQ Processing System. Run Block and Connection automation to establish connections between the added IPs.

Step 7: Wrapper and Bitstream generation: The block design is then validated and followed by generating the wrapper. This wrapper maps the IO of the design with the onboard physical pins. Finally, the hardware file (.hdf) is obtained along with the bitstream generated for the design.

Step 8: Application Development: The design must then be checked on the Zybo board for which an application C code is needed. This source code runs an application for the designed hardware, and it is debugged on the Zybo Board

Vivado HLS Results:

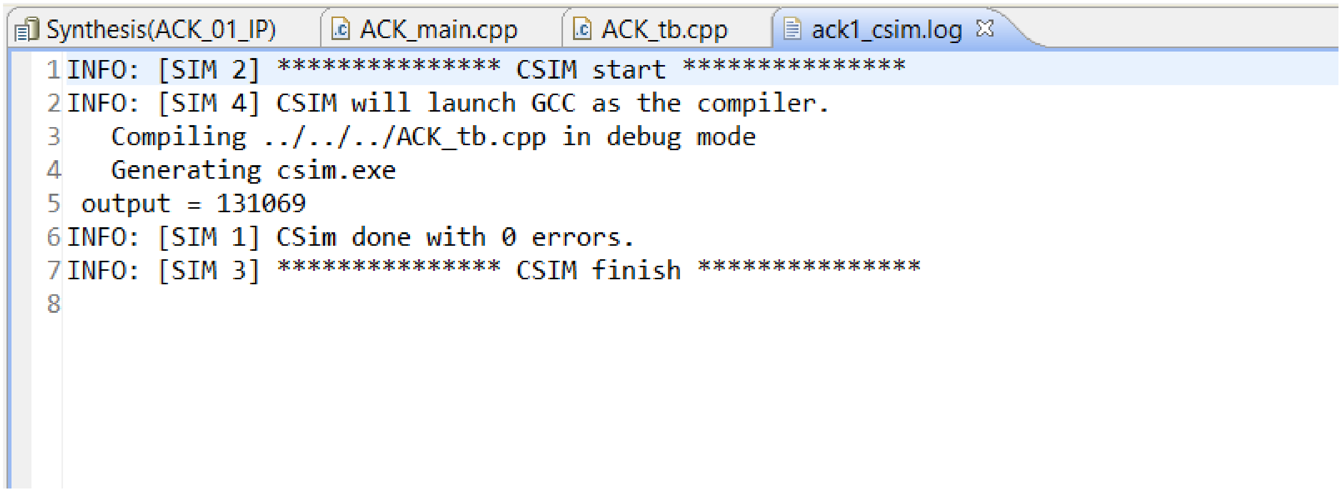
Simulation result:



```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../ACK_tb.cpp in debug mode
   Compiling ../../../../../Documents/ACK_main.cpp in debug mode
   Generating csim.exe
 output = 3
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

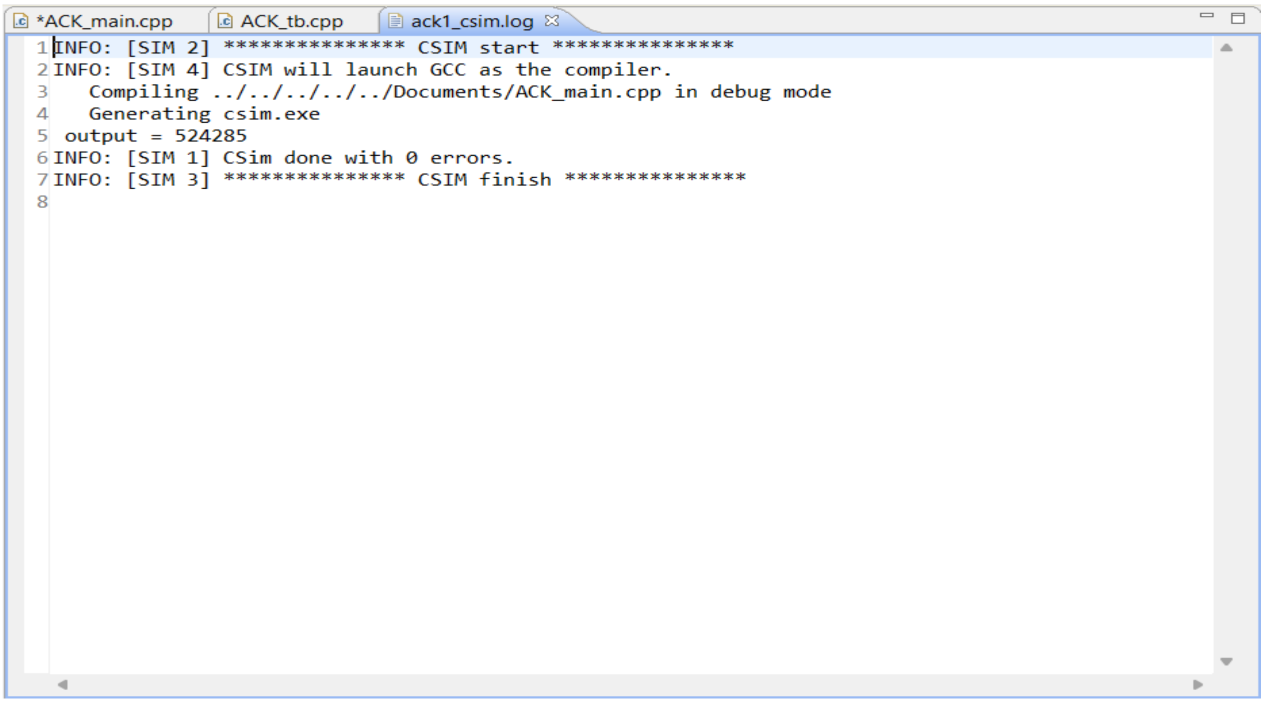
```
                 from ../../../../../Documents/ACK_main.cpp:3:
C:/Xilinx_2017/Vivado/2017.4/include/gmp.h:62:0: warning: "__GMP_LIBGMP_DLL" redefined [enabled by default]
C:/Xilinx_2017/Vivado/2017.4/include/floating_point_v7_0_bitacc_cmodel.h:135:0: note: this is the location of the previous definition
 output = 3INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

- The above result is generated with inputs m = 0; n = 2.
- It compiled successfully in Vivado HLS 2017.4.
- This is one of the above cases being considered.



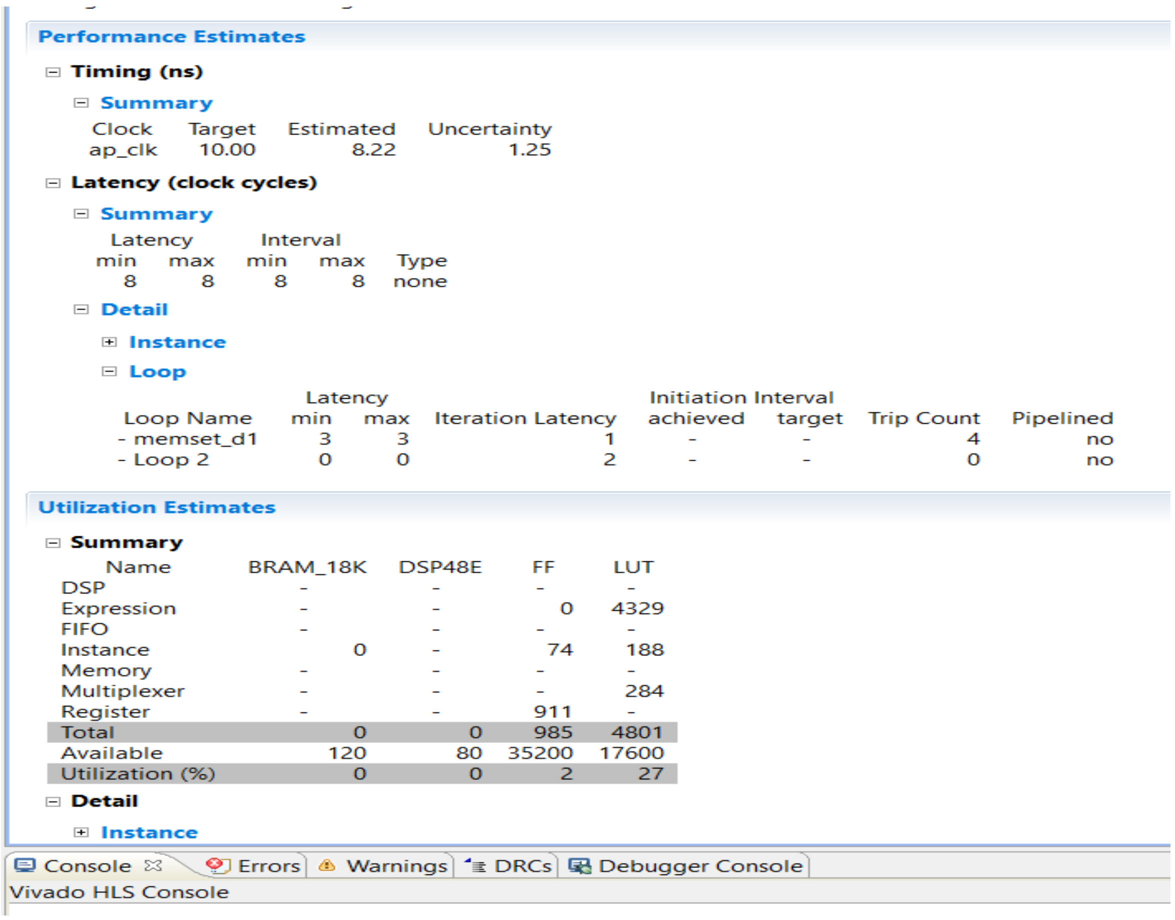
```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../ACK_tb.cpp in debug mode
   Generating csim.exe
 output = 131069
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

- The above output was given when m = 3 and n = 14, one of the proposed cases. The simulation was successful using Vivado HLS 2017.4.
- We have used a test case with m = 3 and n = 16 to check the computability and the simulation result is as follows,


*ACK_main.cpp ACK_tb.cpp ack1_csim.log

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../Documents/ACK_main.cpp in debug mode
   Generating csim.exe
 output = 524285
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Synthesis result:


Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.22	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
8	8	8	8	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- memset_d1	3	3	1	-	-	4	no
- Loop 2	0	0	2	-	-	0	no

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4329
FIFO	-	-	-	-
Instance	0	-	74	188
Memory	-	-	-	-
Multiplexer	-	-	-	284
Register	-	-	911	-
Total	0	0	985	4801
Available	120	80	35200	17600
Utilization (%)	0	0	2	27

Detail

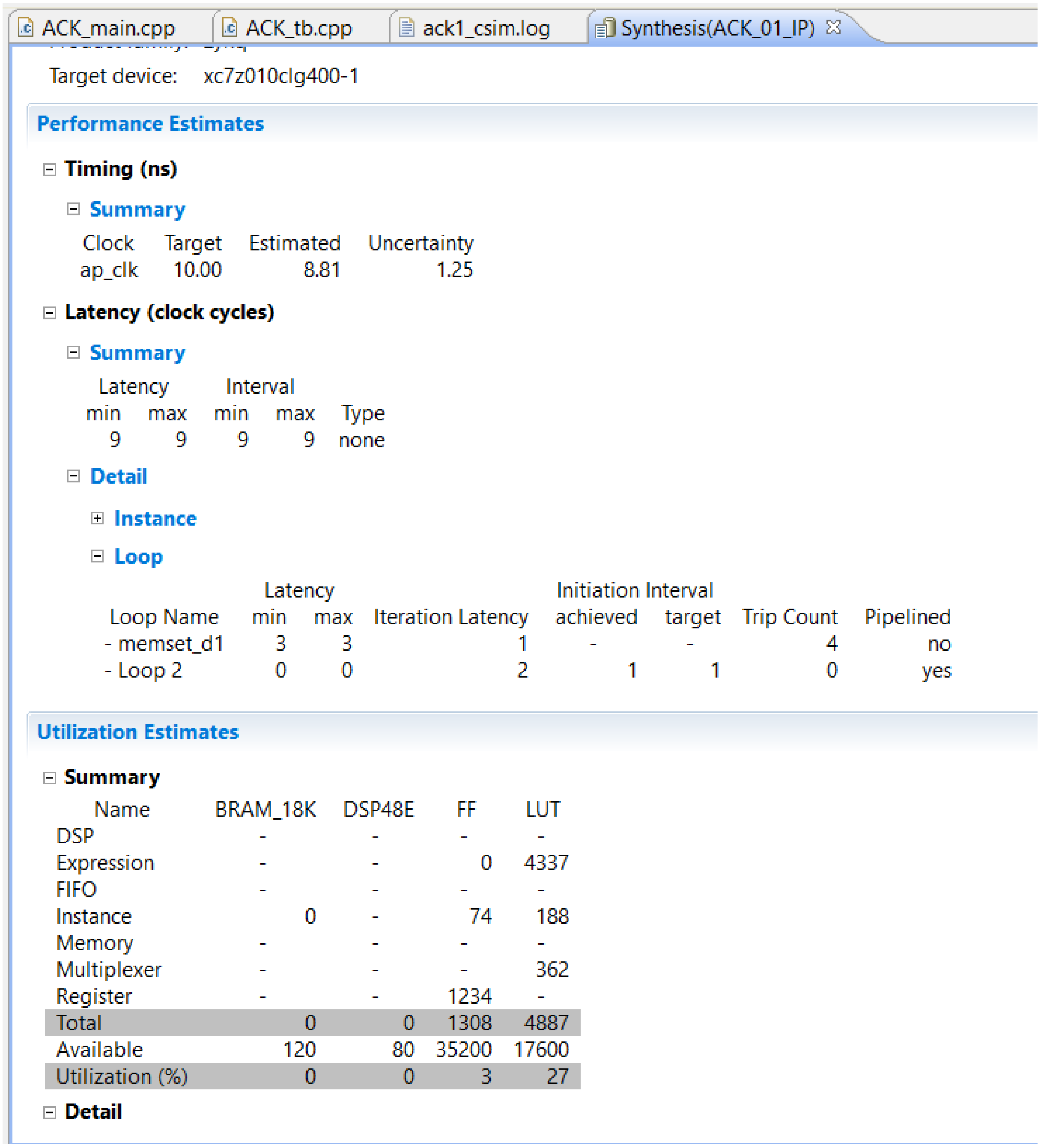
Instance

Console Errors Warnings DRCs Debugger Console

Vivado HLS Console

- The synthesis for the above-simulated code has been done and the output latency was calculated using the **#pragma HLS loop_tripcount min = 0.**

- The trip count is used for analyses purposes and does not affect any synthesis process.
- Since the loop variable has a bound variable which is determined by dynamic operations, the latency is not calculated.
- The loop synthesis without pipelining is shown above.



Target device: xc7z010clg400-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.81	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
9	9	9	9	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- memset_d1	3	3	1	-	-	4	no
- Loop 2	0	0	2	1	1	0	yes

Utilization Estimates

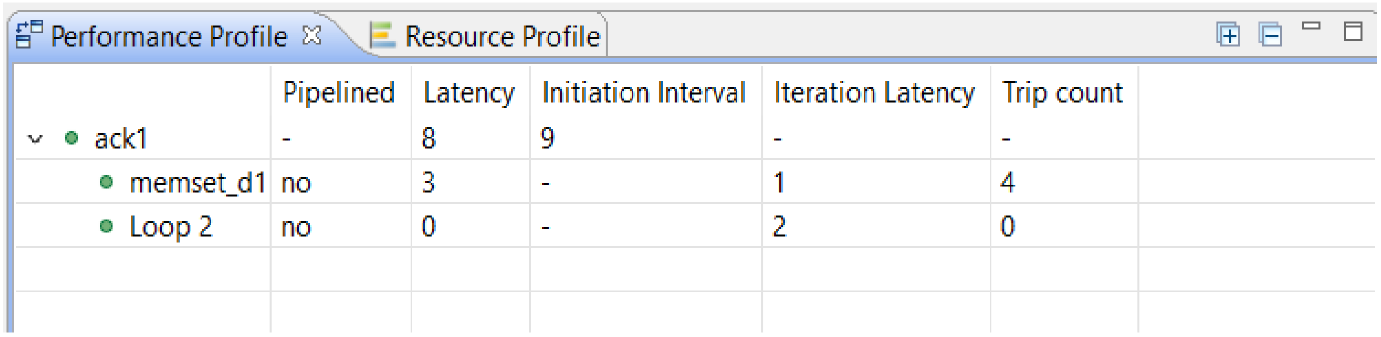
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4337
FIFO	-	-	-	-
Instance	0	-	74	188
Memory	-	-	-	-
Multiplexer	-	-	-	362
Register	-	-	1234	-
Total	0	0	1308	4887
Available	120	80	35200	17600
Utilization (%)	0	0	3	27

Detail

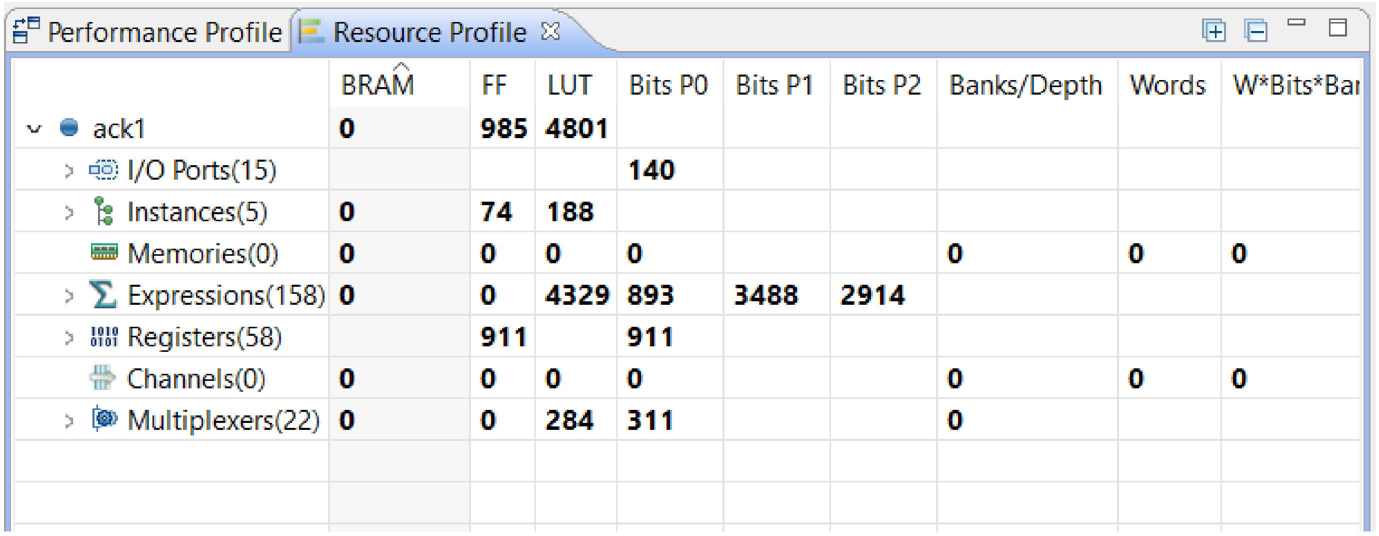
- Then #pragma HLS pipeline II =1 was implemented for the loop to have a pipelined process and the synthesis results are attached as above.

- Detailed analysis of the performance without pipelining is shown as follows,



Performance Profile | Resource Profile

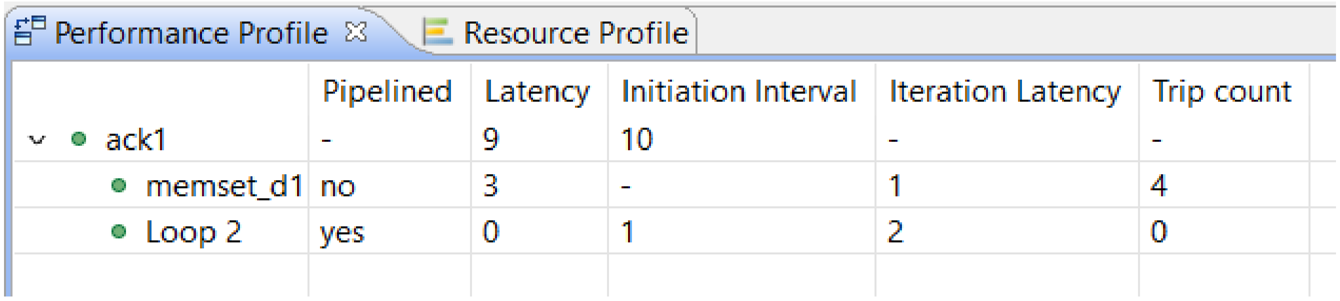
	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
ack1	-	8	9	-	-
memset_d1	no	3	-	1	4
Loop 2	no	0	-	2	0



Performance Profile | Resource Profile

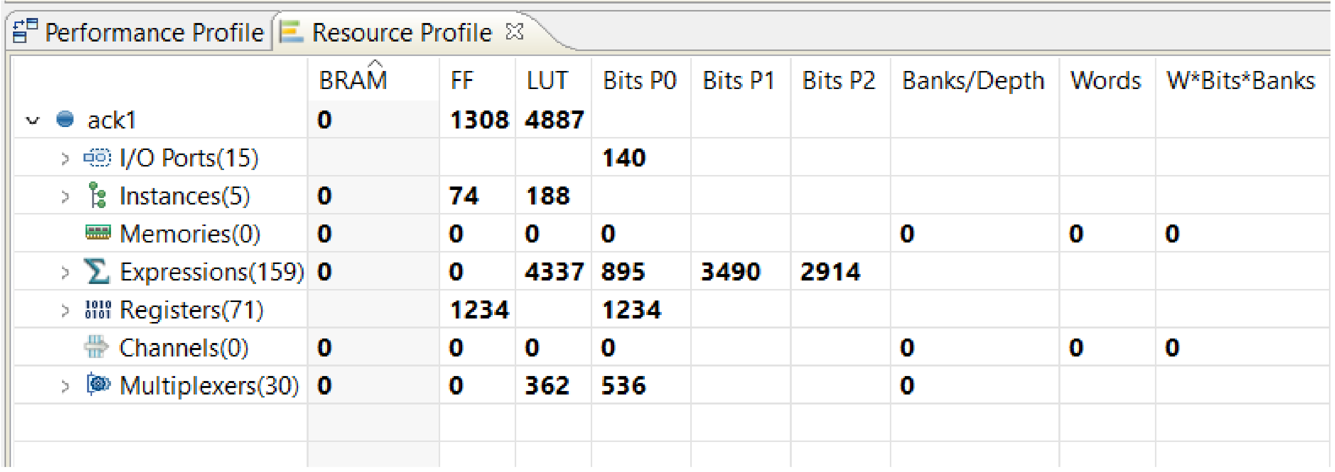
	BRAM	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Bar
ack1	0	985	4801						
I/O Ports(15)				140					
Instances(5)	0	74	188						
Memories(0)	0	0	0	0			0	0	0
Expressions(158)	0	0	4329	893	3488	2914			
Registers(58)		911		911					
Channels(0)	0	0	0	0			0	0	0
Multiplexers(22)	0	0	284	311			0		

- Detailed analysis of the performance with pipelining is shown as follows,



Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
ack1	-	9	10	-	-
memset_d1	no	3	-	1	4
Loop 2	yes	0	1	2	0

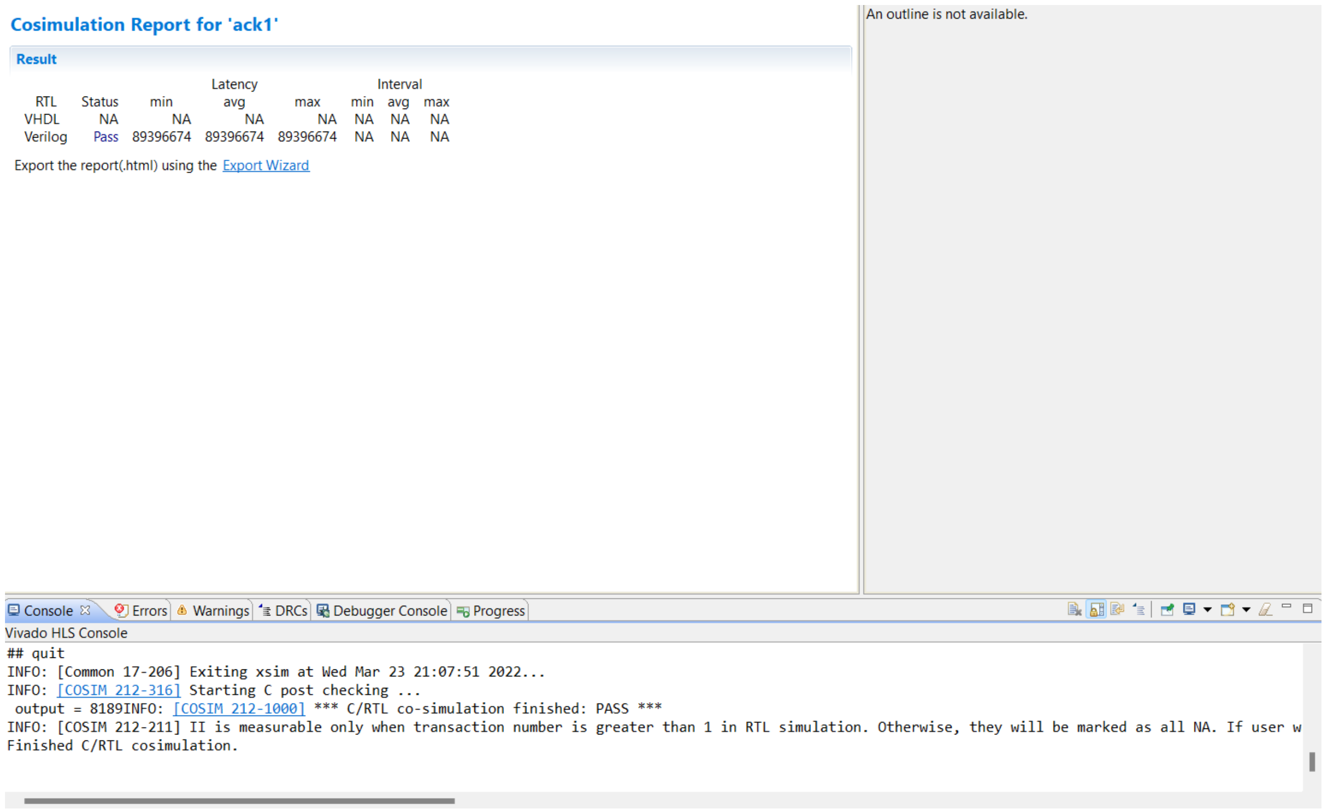


Performance Profile | Resource Profile

	BRAM	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Banks
ack1	0	1308	4887						
I/O Ports(15)				140					
Instances(5)	0	74	188						
Memories(0)	0	0	0	0			0	0	0
Expressions(159)	0	0	4337	895	3490	2914			
Registers(71)		1234		1234					
Channels(0)	0	0	0	0			0	0	0
Multiplexers(30)	0	0	362	536			0		

C/RTL Cosimulation Result:

- The cosimulation of the Ackermann, without pipelining, with m = 3 & n = 10 in Verilog is performed then and the report is as follows,



Cosimulation Report for 'ack1'

Result

RTL	Status	Latency min	Latency avg	Latency max	Interval min	Interval avg	Interval max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	89396674	89396674	89396674	NA	NA	NA

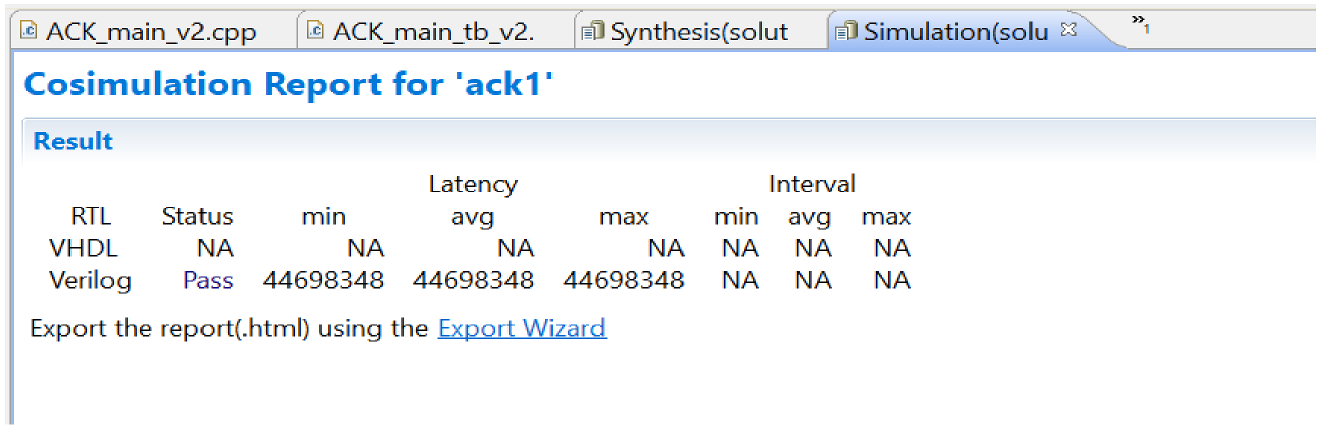
Export the report(.html) using the Export Wizard

An outline is not available.

Vivado HLS Console

```
## quit
INFO: [Common 17-206] Exiting xsim at Wed Mar 23 21:07:51 2022...
INFO: [COSIM 212-316] Starting C post checking ...
 output = 8189INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
INFO: [COSIM 212-211] II is measurable only when transaction number is greater than 1 in RTL simulation. Otherwise, they will be marked as all NA. If user w
Finished C/RTL cosimulation.
```

- The cosimulation of the Ackermann, with pipelining, with m = 3 & n = 10 in Verilog is performed then and the report is as follows,



Cosimulation Report for 'ack1'

Result

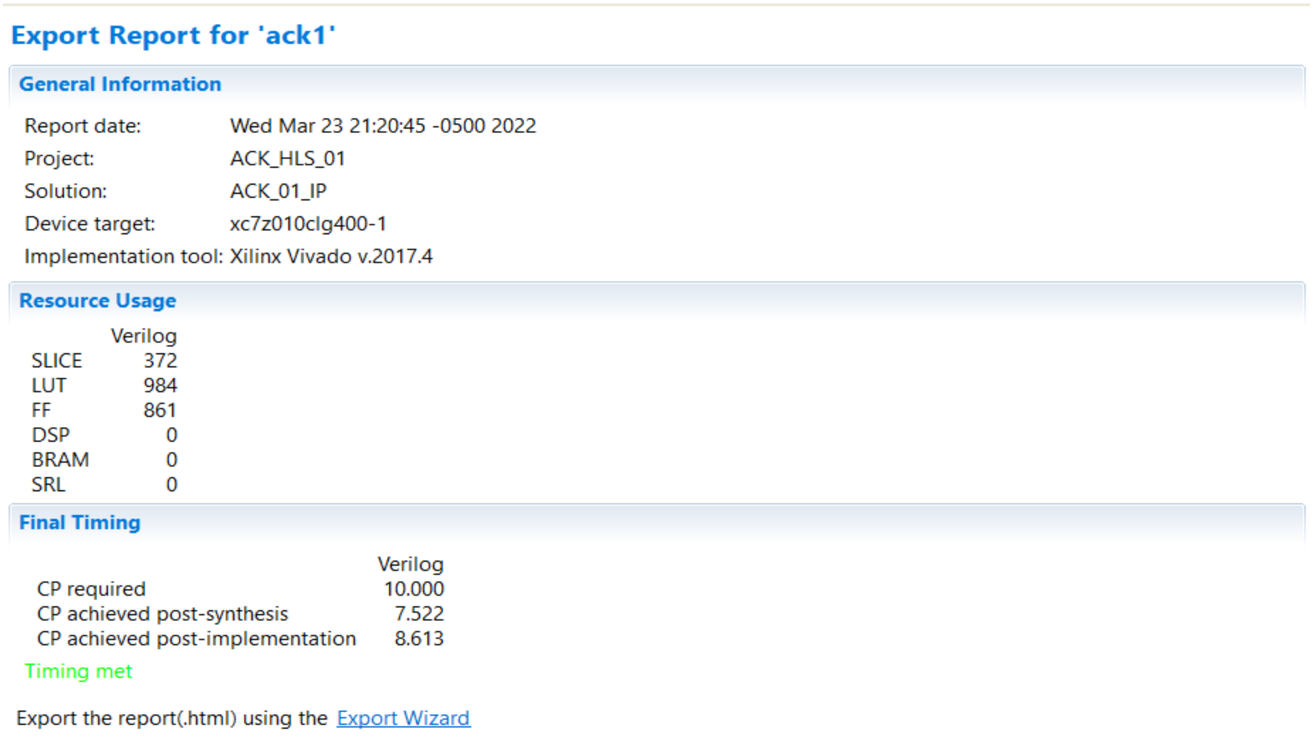
RTL	Status	Latency min	Latency avg	Latency max	Interval min	Interval avg	Interval max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	44698348	44698348	44698348	NA	NA	NA

Export the report(.html) using the Export Wizard

- After this step, export RTL step is performed, and the performance for Verilog RTL is compared with pipelining, and without pipelining.
- The timing constraint was met for Ackermann function with and without pipelining.
- The report is generated as follows,

Export RTL Result:

Without Pipeline:



Export Report for 'ack1'

General Information

Report date:	Wed Mar 23 21:20:45 -0500 2022
Project:	ACK_HLS_01
Solution:	ACK_01_IP
Device target:	xc7z010clg400-1
Implementation tool:	Xilinx Vivado v.2017.4

Resource Usage

	Verilog
SLICE	372
LUT	984
FF	861
DSP	0
BRAM	0
SRL	0

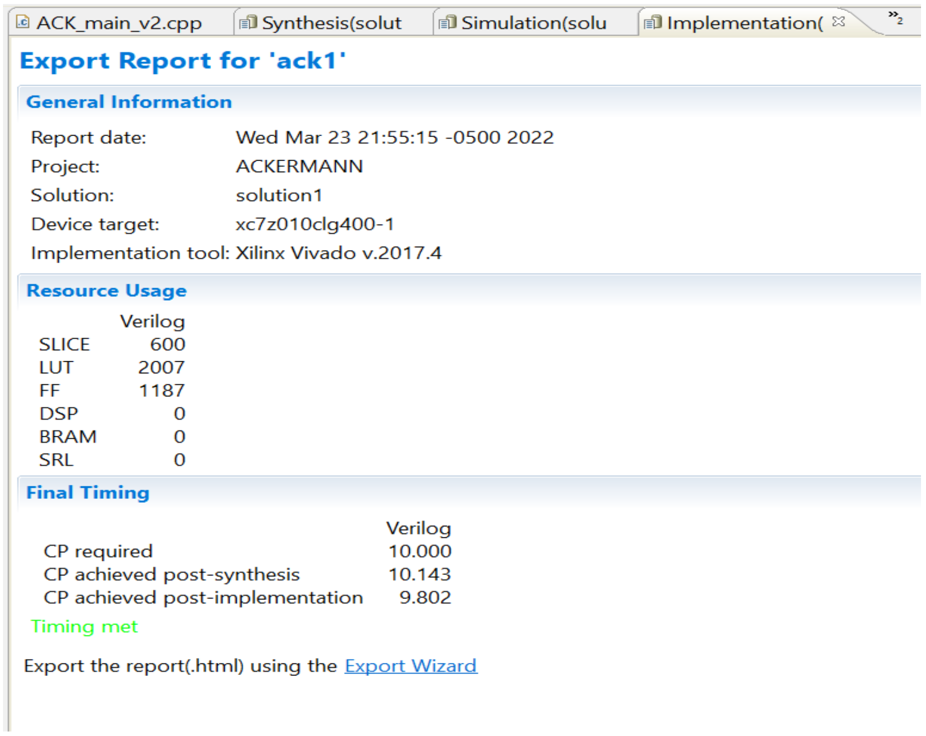
Final Timing

	Verilog
CP required	10.000
CP achieved post-synthesis	7.522
CP achieved post-implementation	8.613

Timing met

Export the report(.html) using the Export Wizard

With Pipeline:



Export Report for 'ack1'

General Information

Report date:	Wed Mar 23 21:55:15 -0500 2022
Project:	ACKERMANN
Solution:	solution1
Device target:	xc7z010clg400-1
Implementation tool:	Xilinx Vivado v.2017.4

Resource Usage

	Verilog
SLICE	600
LUT	2007
FF	1187
DSP	0
BRAM	0
SRL	0

Final Timing

	Verilog
CP required	10.000
CP achieved post-synthesis	10.143
CP achieved post-implementation	9.802

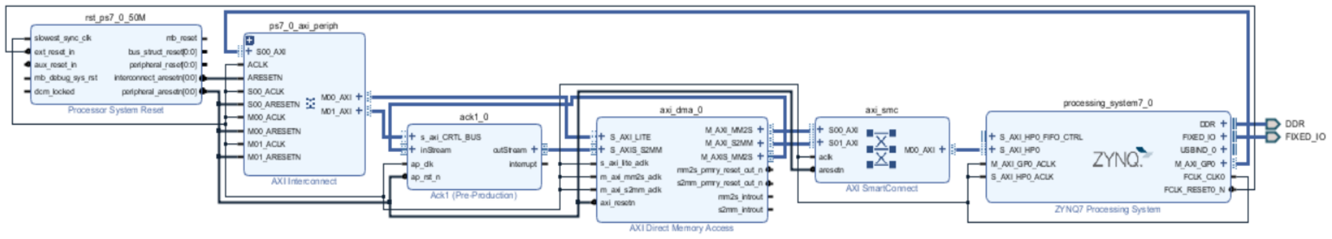
Timing met

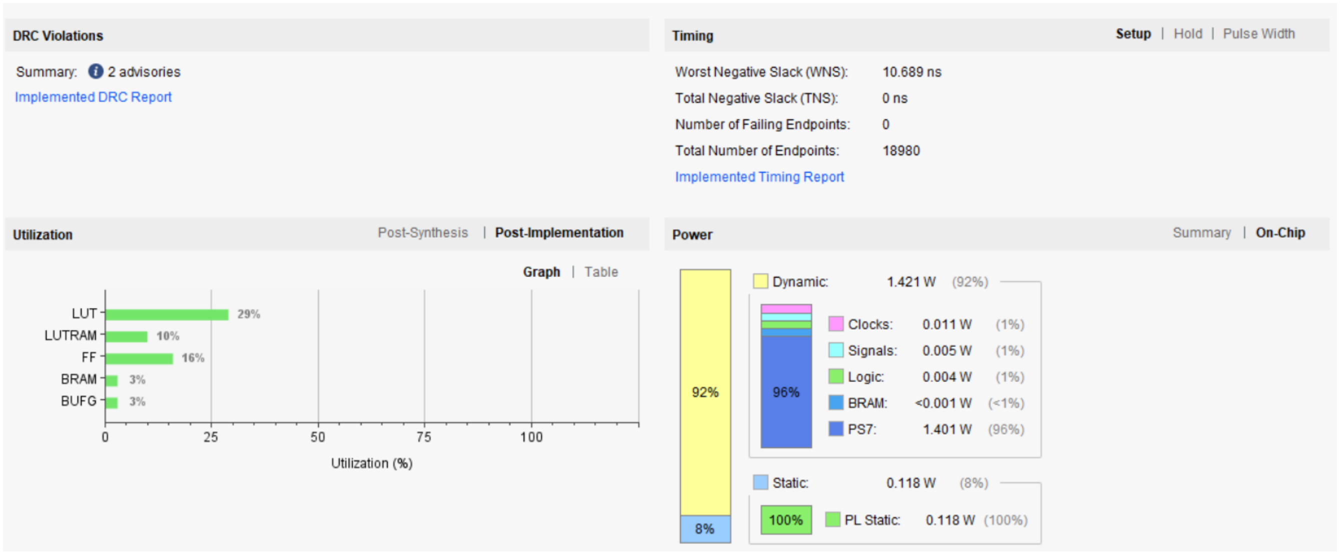
Export the report(.html) using the Export Wizard

Vivado Results:

- After we export RTL in Vivado HLS, we create the block diagram for customizing IP catalog.
- The following steps are to be taken.
- Block Diagram -> Design Validation -> Creating HDL Wrapper ->Synthesis -> Bitstream generation.
- After creating a new project in Vivado IDE with board configuration settings, the new block diagram is created under the IP integrator.

Block Diagram:





DRC Violations

Summary: 2 advisories
Implemented DRC Report

Timing Setup | Hold | Pulse Width

Worst Negative Slack (WNS):	10.689 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	18980

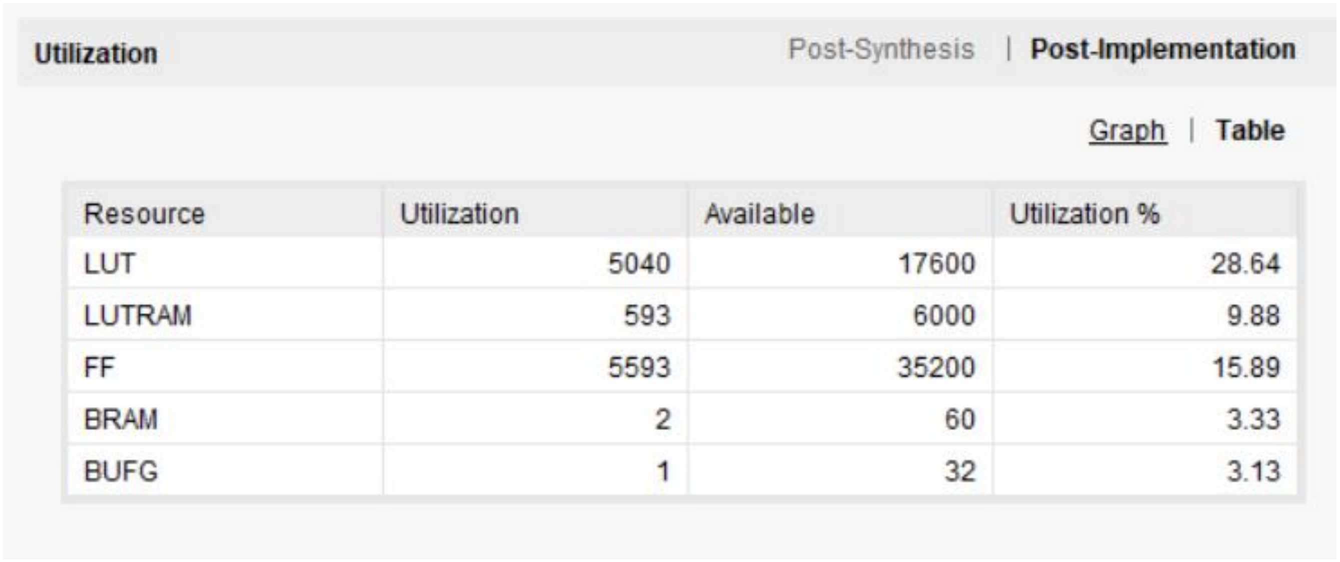
Implemented Timing Report

Utilization Post-Synthesis | Post-Implementation

Graph | Table

Power Summary | On-Chip

Dynamic:	1.421 W	(92%)
Clocks:	0.011 W	(1%)
Signals:	0.005 W	(1%)
Logic:	0.004 W	(1%)
BRAM:	<0.001 W	(<1%)
PS7:	1.401 W	(96%)
Static:	0.118 W	(8%)
PL Static:	0.118 W	(100%)



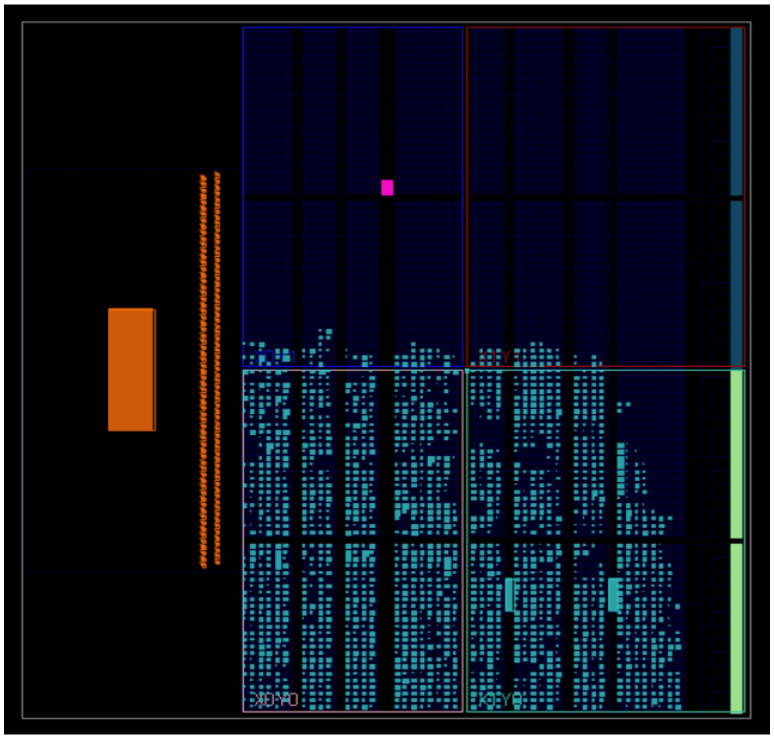
Utilization Post-Synthesis | Post-Implementation

Graph | Table

Resource	Utilization	Available	Utilization %
LUT	5040	17600	28.64
LUTRAM	593	6000	9.88
FF	5593	35200	15.89
BRAM	2	60	3.33
BUFG	1	32	3.13

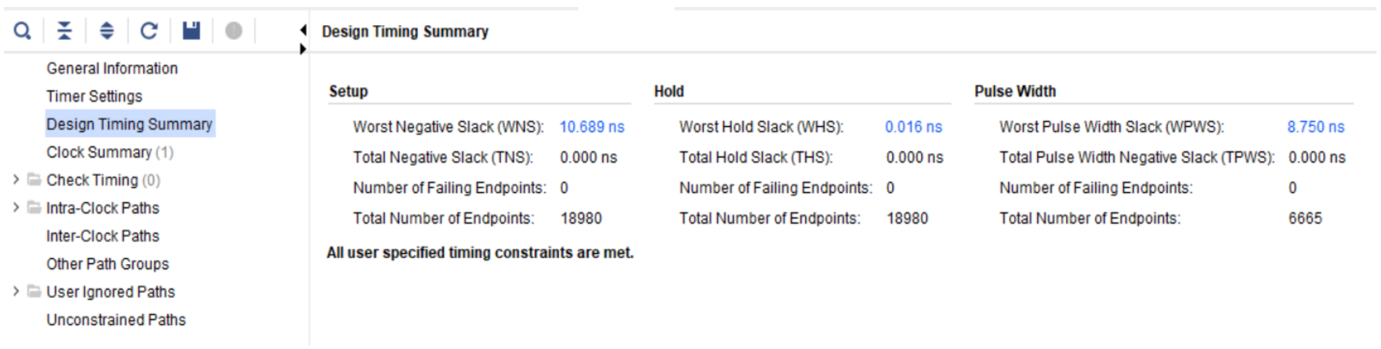
- Once the block diagram is created and the necessary interconnects have been given with the zynq processor and the Ackermann block and DMA controller, the design is validated.
- After validation, an HDL wrapper is created for the design.
- The implemented design after synthesis with placement and routing results are shown below,

Device Layout Report:



- After the implementation of the above Ackermann design, different reports were generated based on the design.
- The timing report is shown to show that the timing constraints were met.
- Detailed timing of the slack was reported in this detailed analysis.
- Additionally, power, DRC reports were also reported.

Timing Report:

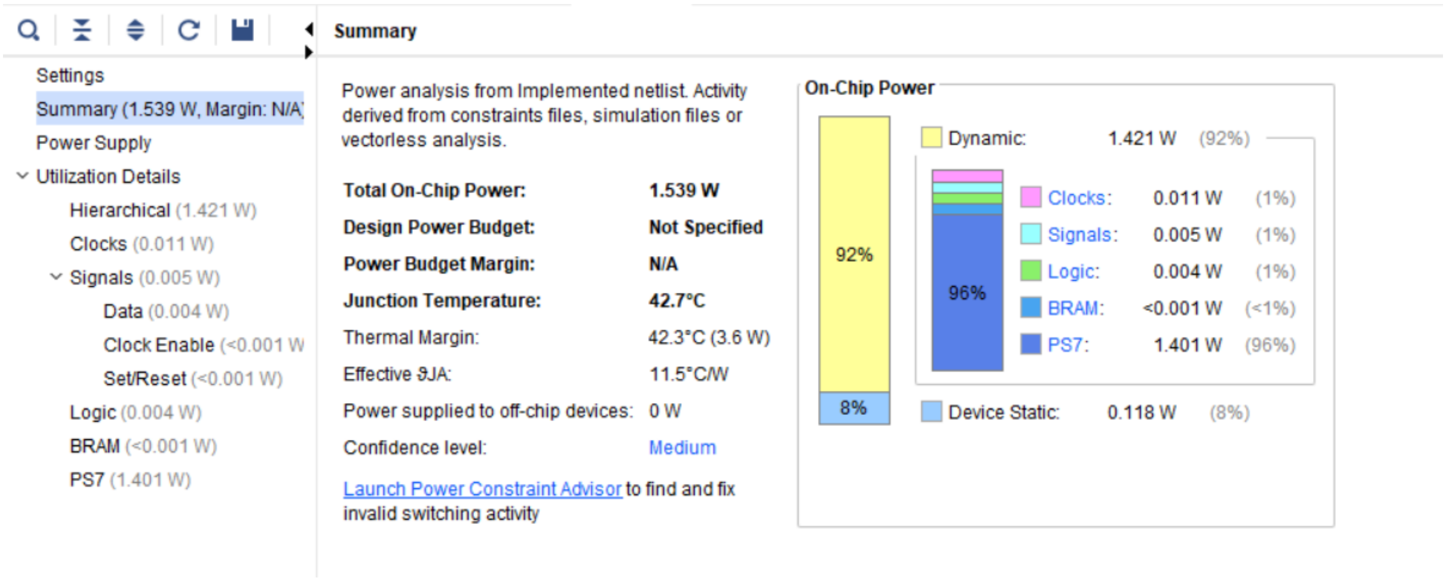


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	10.689 ns	Worst Hold Slack (WHS):	0.016 ns	Worst Pulse Width Slack (WPWS):	8.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	18980	Total Number of Endpoints:	18980	Total Number of Endpoints:	6665

All user specified timing constraints are met.

Power Report:



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

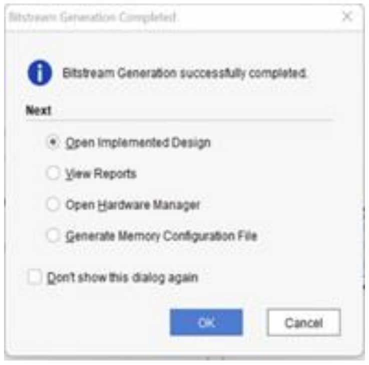
Total On-Chip Power:	**1.539 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**42.7°C**
Thermal Margin:	42.3°C (3.6 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

🟨	Dynamic:	1.421 W	(92%)
🟪	Clocks:	0.011 W	(1%)
🟦	Signals:	0.005 W	(1%)
🟩	Logic:	0.004 W	(1%)
🟦	BRAM:	<0.001 W	(<1%)
🟦	PS7:	1.401 W	(96%)
🟦	Device Static:	0.118 W	(8%)

Bitstream Generated report:

- Finally, the bitstream was generated successfully and exported to the software environment (SDK) where we can use the user-defined IP package to implement the Ackerman function on the Zynq hardware.
- The SDK can be launched from the Vivado 2017.4.



Bitstream Generation successfully completed.

[✓ Open Implemented Design]
View Reports
Open Hardware Manager
Generate Memory Configuration File

Don't show this dialog again

- Additionally, software with the Hello World application template can be written with Ackerman function to verify the results and compare the clock cycles taken for different inputs between the hardware implementation and the software implementation.

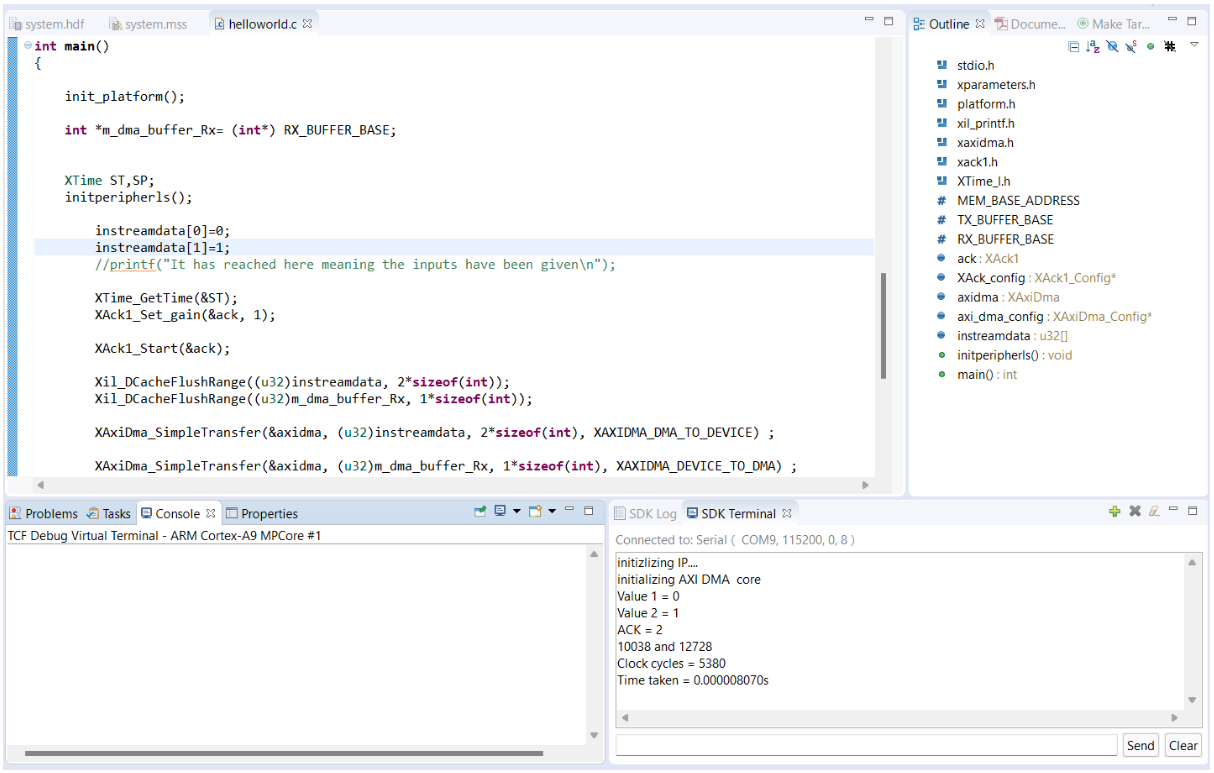
SDK terminal Output:

Hardware Result:

- With the exported Ackermann function bitstream, we created a new application project with HelloWorld template to implement the Ackermann functionality.
- The ZYBO board was connected to the system and set to a baud rate of 115200 and programmed the FPGA to flash the memory with the Ackermann program.
- Right-click on the program and run as *Launch on Hardware system debugger* and can be viewed on the putty terminal or tera term.

- In our case, we had used the putty terminal to display the result.
- The inputs were given through the USB from the user.
- We have attached three case scenarios but ran all the test cases as per the table attached,
    - m = 0; n = 1
    - m = 2; n = 10
    - m = 3; n = 14

- Case 1: m = 0 & n = 1; with Hardware implementation:



```
int main()
{

    init_platform();

    int *m_dma_buffer_Rx= (int*) RX_BUFFER_BASE;


    XTime ST,SP;
    initperipherls();

        instreamdata[0]=0;
        instreamdata[1]=1;
        //printf("It has reached here meaning the inputs have been given\n");

        XTime_GetTime(&ST);
        XAck1_Set_gain(&ack, 1);

        XAck1_Start(&ack);

        Xil_DCacheFlushRange((u32)instreamdata, 2*sizeof(int));
        Xil_DCacheFlushRange((u32)m_dma_buffer_Rx, 1*sizeof(int));

        XAxiDma_SimpleTransfer(&axidma, (u32)instreamdata, 2*sizeof(int), XAXIDMA_DMA_TO_DEVICE) ;

        XAxiDma_SimpleTransfer(&axidma, (u32)m_dma_buffer_Rx, 1*sizeof(int), XAXIDMA_DEVICE_TO_DMA) ;
```

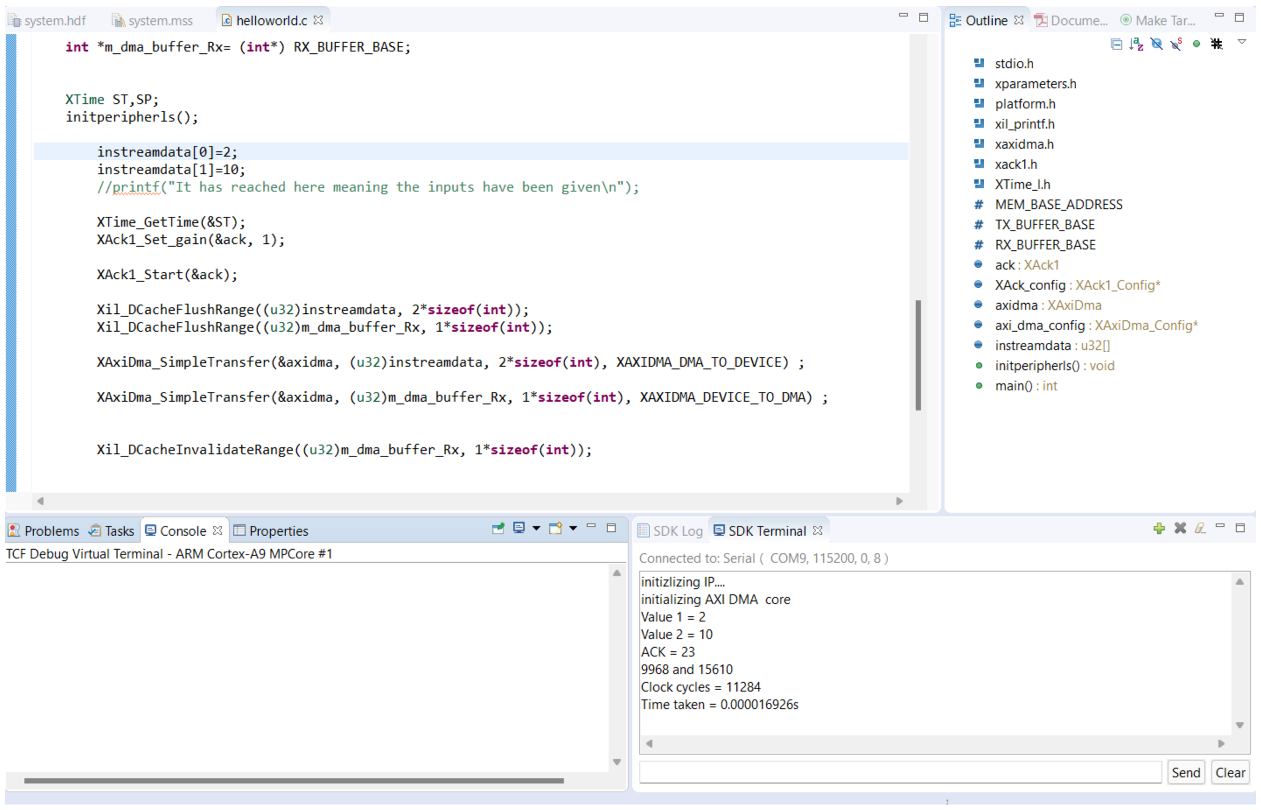
```
stdio.h
xparameters.h
platform.h
xil_printf.h
xaxidma.h
xack1.h
XTime_l.h
# MEM_BASE_ADDRESS
# TX_BUFFER_BASE
# RX_BUFFER_BASE
ack : XAck1
XAck_config : XAck1_Config*
axidma : XAxiDma
axi_dma_config : XAxiDma_Config*
instreamdata : u32[]
initperipherls() : void
main() : int
```

TCF Debug Virtual Terminal - ARM Cortex-A9 MPCore #1

Connected to: Serial (COM9, 115200, 0, 8)

```
initizlizing IP....
initializing AXI DMA  core
Value 1 = 0
Value 2 = 1
ACK = 2
10038 and 12728
Clock cycles = 5380
Time taken = 0.000008070s
```

- Case 2: m = 2 & n = 10; with Hardware implementation:



```
    int *m_dma_buffer_Rx= (int*) RX_BUFFER_BASE;


    XTime ST,SP;
    initperipherls();

        instreamdata[0]=2;
        instreamdata[1]=10;
        //printf("It has reached here meaning the inputs have been given\n");

        XTime_GetTime(&ST);
        XAck1_Set_gain(&ack, 1);

        XAck1_Start(&ack);

        Xil_DCacheFlushRange((u32)instreamdata, 2*sizeof(int));
        Xil_DCacheFlushRange((u32)m_dma_buffer_Rx, 1*sizeof(int));

        XAxiDma_SimpleTransfer(&axidma, (u32)instreamdata, 2*sizeof(int), XAXIDMA_DMA_TO_DEVICE) ;

        XAxiDma_SimpleTransfer(&axidma, (u32)m_dma_buffer_Rx, 1*sizeof(int), XAXIDMA_DEVICE_TO_DMA) ;


        Xil_DCacheInvalidateRange((u32)m_dma_buffer_Rx, 1*sizeof(int));
```

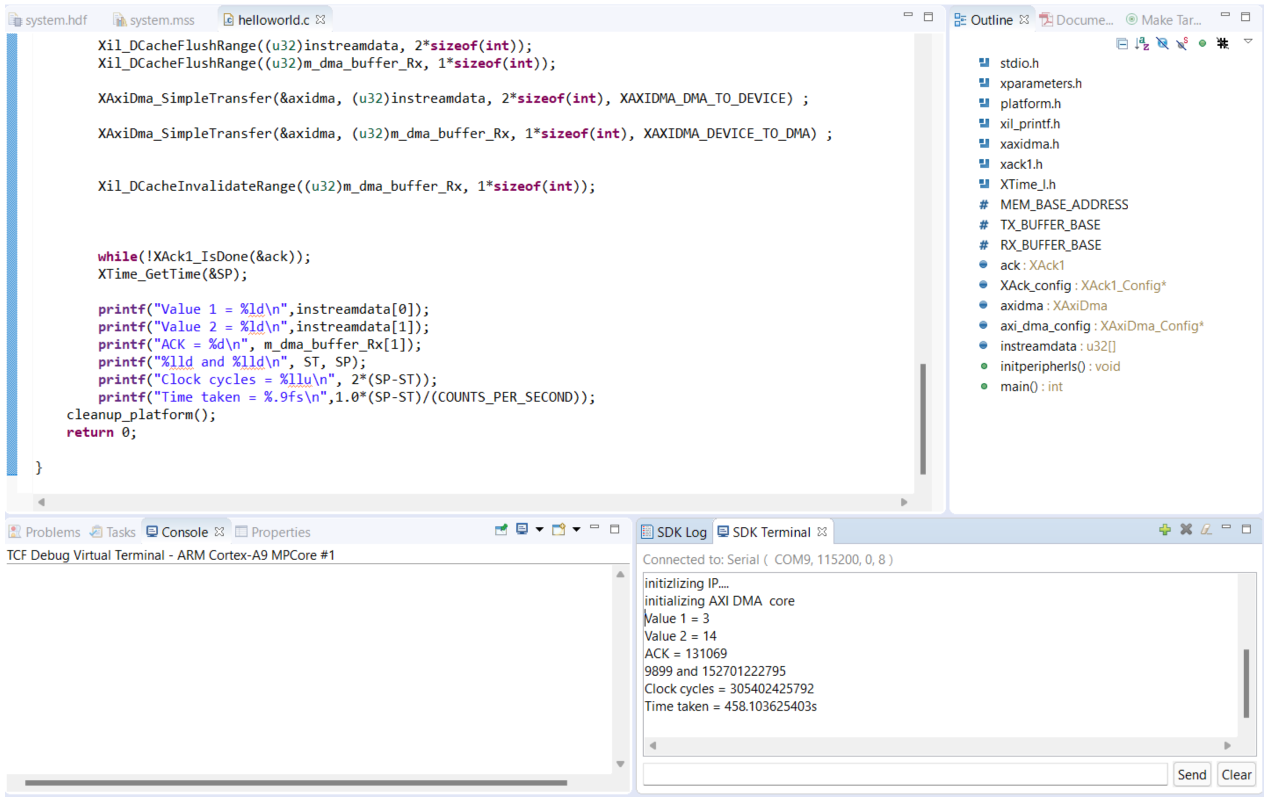
Outline: stdio.h, xparameters.h, platform.h, xil_printf.h, xaxidma.h, xack1.h, XTime_l.h, MEM_BASE_ADDRESS, TX_BUFFER_BASE, RX_BUFFER_BASE, ack : XAck1, XAck_config : XAck1_Config*, axidma : XAxiDma, axi_dma_config : XAxiDma_Config*, instreamdata : u32[], initperipherls() : void, main() : int

TCF Debug Virtual Terminal - ARM Cortex-A9 MPCore #1

Connected to: Serial (COM9, 115200, 0, 8)

```
initizlizing IP....
initializing AXI DMA  core
Value 1 = 2
Value 2 = 10
ACK = 23
9968 and 15610
Clock cycles = 11284
Time taken = 0.000016926s
```

- Case 3: m = 3 & n = 14; Hardware implementation:



```
        Xil_DCacheFlushRange((u32)instreamdata, 2*sizeof(int));
        Xil_DCacheFlushRange((u32)m_dma_buffer_Rx, 1*sizeof(int));

        XAxiDma_SimpleTransfer(&axidma, (u32)instreamdata, 2*sizeof(int), XAXIDMA_DMA_TO_DEVICE) ;

        XAxiDma_SimpleTransfer(&axidma, (u32)m_dma_buffer_Rx, 1*sizeof(int), XAXIDMA_DEVICE_TO_DMA) ;


        Xil_DCacheInvalidateRange((u32)m_dma_buffer_Rx, 1*sizeof(int));



        while(!XAck1_IsDone(&ack));
        XTime_GetTime(&SP);

        printf("Value 1 = %ld\n",instreamdata[0]);
        printf("Value 2 = %ld\n",instreamdata[1]);
        printf("ACK = %d\n", m_dma_buffer_Rx[1]);
        printf("%lld and %lld\n", ST, SP);
        printf("Clock cycles = %llu\n", 2*(SP-ST));
        printf("Time taken = %.9fs\n",1.0*(SP-ST)/(COUNTS_PER_SECOND));
    cleanup_platform();
    return 0;

}
```

Outline: stdio.h, xparameters.h, platform.h, xil_printf.h, xaxidma.h, xack1.h, XTime_l.h, MEM_BASE_ADDRESS, TX_BUFFER_BASE, RX_BUFFER_BASE, ack : XAck1, XAck_config : XAck1_Config*, axidma : XAxiDma, axi_dma_config : XAxiDma_Config*, instreamdata : u32[], initperipherls() : void, main() : int

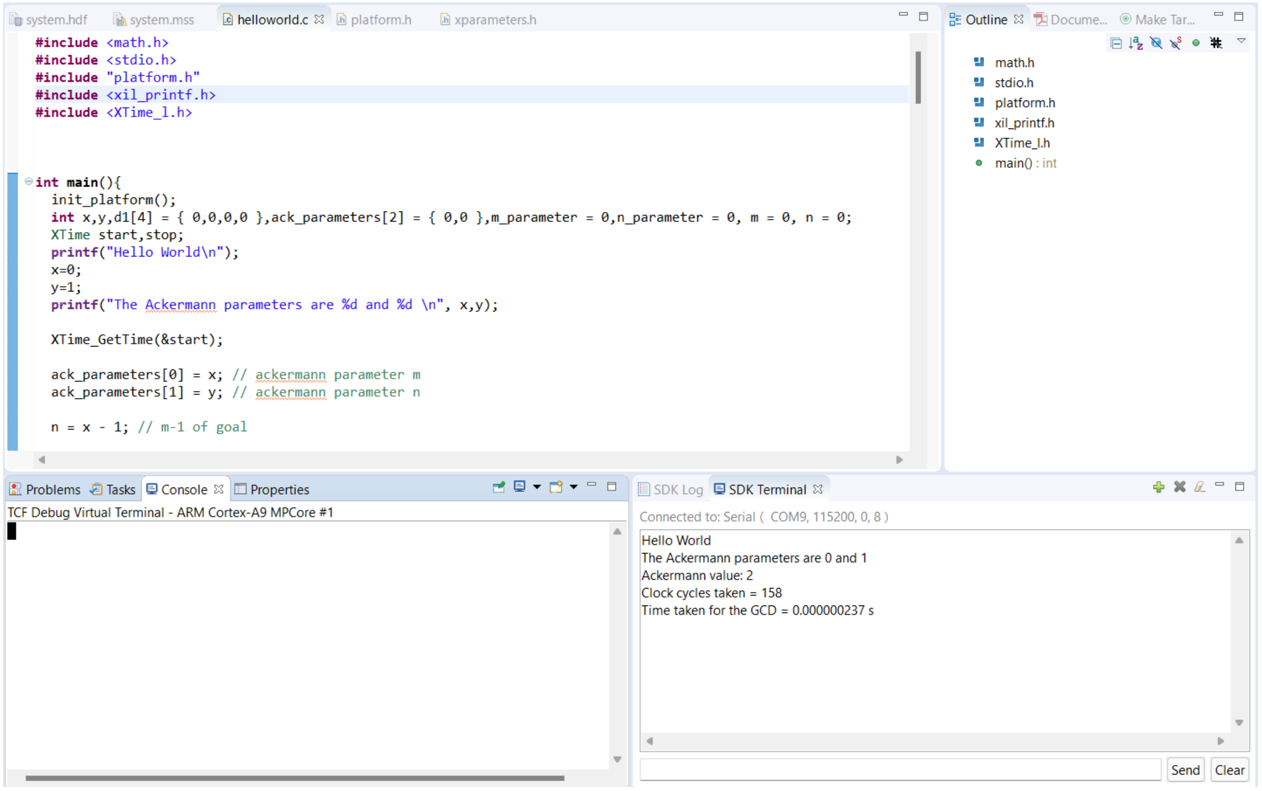
TCF Debug Virtual Terminal - ARM Cortex-A9 MPCore #1

Connected to: Serial (COM9, 115200, 0, 8)

```
initizlizing IP....
initializing AXI DMA  core
Value 1 = 3
Value 2 = 14
ACK = 131069
9899 and 152701222795
Clock cycles = 305402425792
Time taken = 458.103625403s
```

Software Result:

- Case 1: m = 0 & n = 1; software implementation on zynq processor;

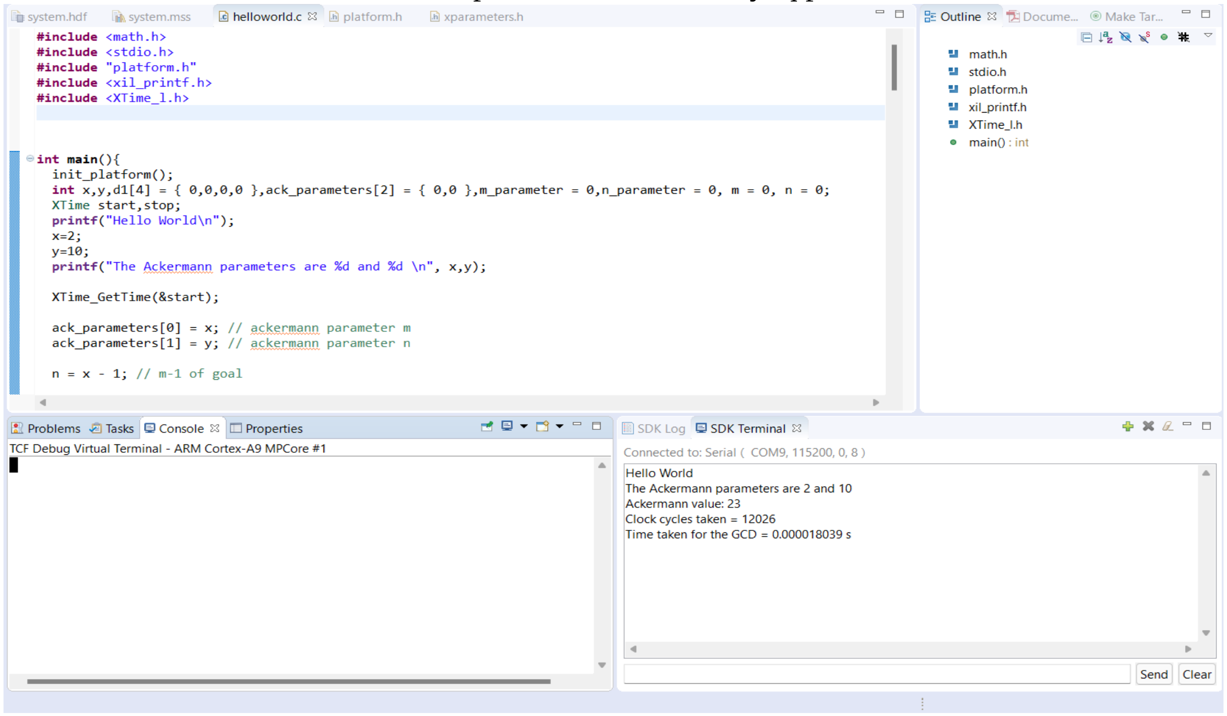


```
#include <math.h>
#include <stdio.h>
#include "platform.h"
#include <xil_printf.h>
#include <XTime_l.h>


int main(){
  init_platform();
  int x,y,d1[4] = { 0,0,0,0 },ack_parameters[2] = { 0,0 },m_parameter = 0,n_parameter = 0, m = 0, n = 0;
  XTime start,stop;
  printf("Hello World\n");
  x=0;
  y=1;
  printf("The Ackermann parameters are %d and %d \n", x,y);

  XTime_GetTime(&start);

  ack_parameters[0] = x; // ackermann parameter m
  ack_parameters[1] = y; // ackermann parameter n

  n = x - 1; // m-1 of goal
```

```
TCF Debug Virtual Terminal - ARM Cortex-A9 MPCore #1
```

```
Connected to: Serial ( COM9, 115200, 0, 8 )
Hello World
The Ackermann parameters are 0 and 1
Ackermann value: 2
Clock cycles taken = 158
Time taken for the GCD = 0.000000237 s
```

- Case 2: m = 2 & n = 10; software implementation on the zynq processor;

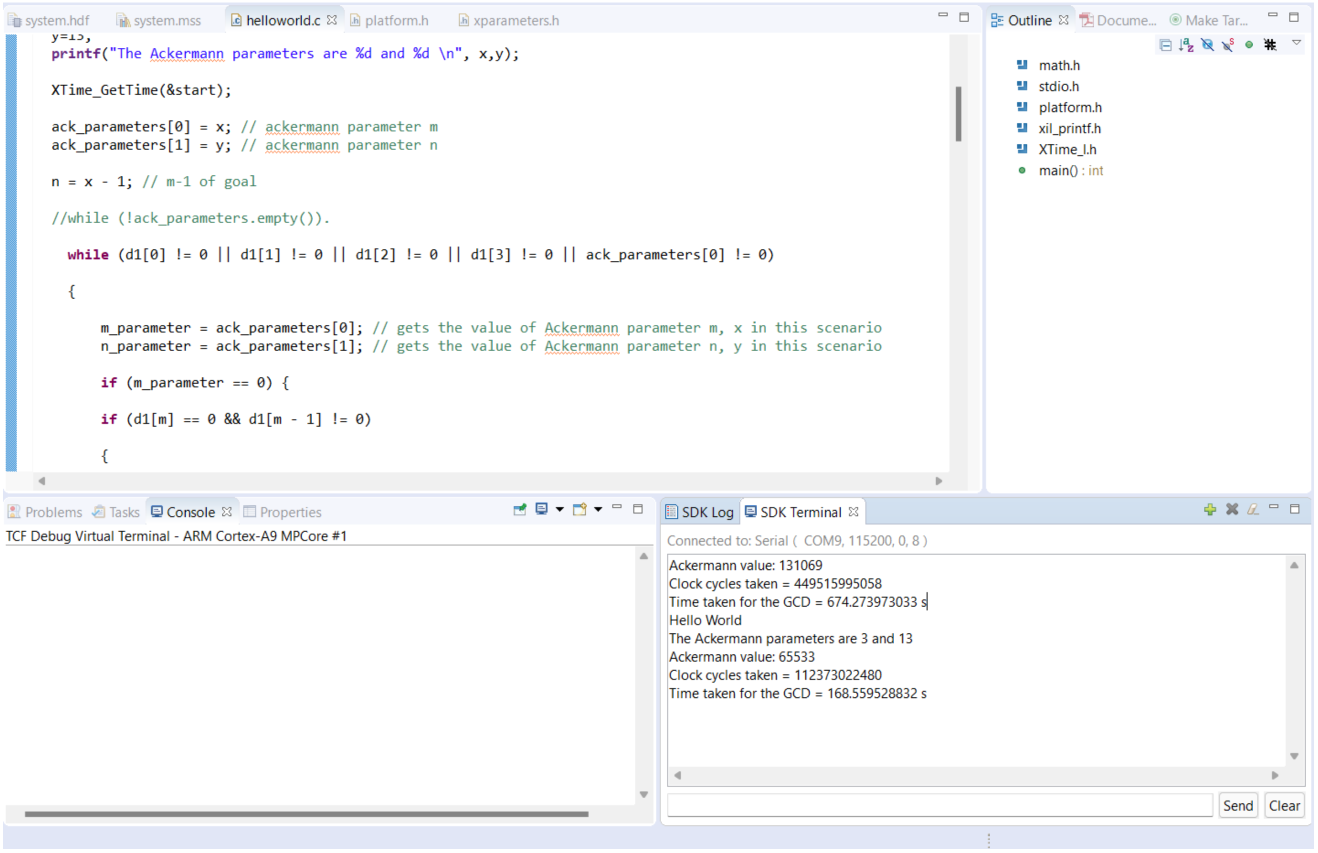


```
#include <math.h>
#include <stdio.h>
#include "platform.h"
#include <xil_printf.h>
#include <XTime_l.h>


int main(){
  init_platform();
  int x,y,d1[4] = { 0,0,0,0 },ack_parameters[2] = { 0,0 },m_parameter = 0,n_parameter = 0, m = 0, n = 0;
  XTime start,stop;
  printf("Hello World\n");
  x=2;
  y=10;
  printf("The Ackermann parameters are %d and %d \n", x,y);

  XTime_GetTime(&start);

  ack_parameters[0] = x; // ackermann parameter m
  ack_parameters[1] = y; // ackermann parameter n

  n = x - 1; // m-1 of goal
```

```
TCF Debug Virtual Terminal - ARM Cortex-A9 MPCore #1
```

```
Connected to: Serial ( COM9, 115200, 0, 8 )
Hello World
The Ackermann parameters are 2 and 10
Ackermann value: 23
Clock cycles taken = 12026
Time taken for the GCD = 0.000018039 s
```

- Case 3: m = 3 & n = 14; software implementation on zynq processor.



```
    y=13;
    printf("The Ackermann parameters are %d and %d \n", x,y);

    XTime_GetTime(&start);

    ack_parameters[0] = x; // ackermann parameter m
    ack_parameters[1] = y; // ackermann parameter n

    n = x - 1; // m-1 of goal

    //while (!ack_parameters.empty()).

      while (d1[0] != 0 || d1[1] != 0 || d1[2] != 0 || d1[3] != 0 || ack_parameters[0] != 0)

      {

          m_parameter = ack_parameters[0]; // gets the value of Ackermann parameter m, x in this scenario
          n_parameter = ack_parameters[1]; // gets the value of Ackermann parameter n, y in this scenario

          if (m_parameter == 0) {

          if (d1[m] == 0 && d1[m - 1] != 0)

          {
```

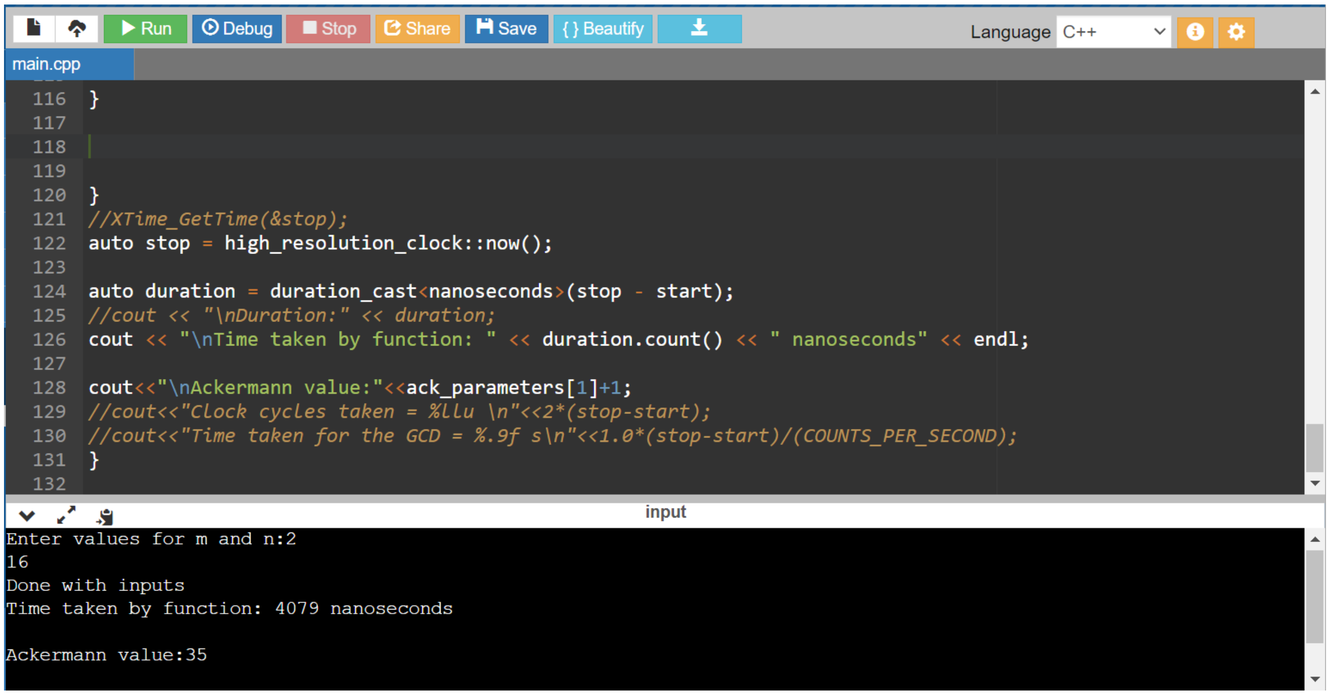
TCF Debug Virtual Terminal - ARM Cortex-A9 MPCore #1

Connected to: Serial (COM9, 115200, 0, 8)

```
Ackermann value: 131069
Clock cycles taken = 449515995058
Time taken for the GCD = 674.273973033 s
Hello World
The Ackermann parameters are 3 and 13
Ackermann value: 65533
Clock cycles taken = 112373022480
Time taken for the GCD = 168.559528832 s
```

System C Compiler Results:

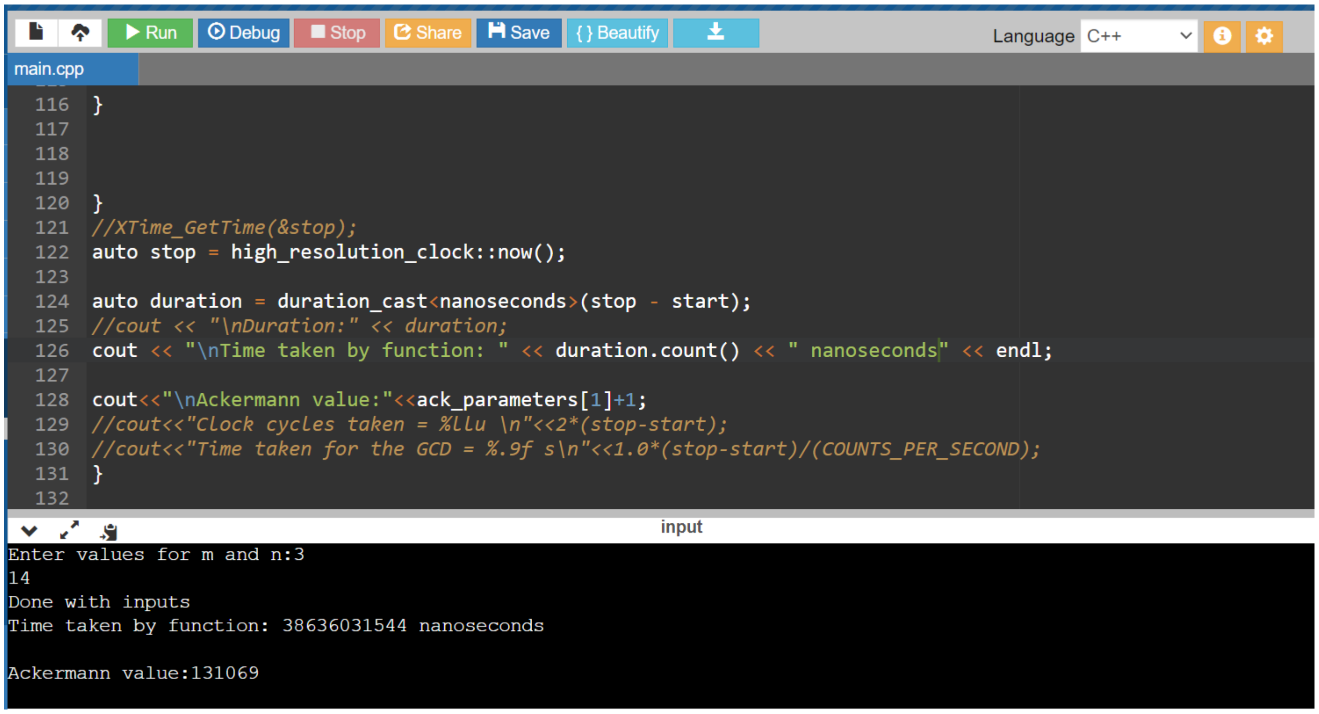
- Case 1: m = 2 & n = 16; System C Compiler



```
}


}
//XTime_GetTime(&stop);
auto stop = high_resolution_clock::now();

auto duration = duration_cast<nanoseconds>(stop - start);
//cout << "\nDuration:" << duration;
cout << "\nTime taken by function: " << duration.count() << " nanoseconds" << endl;

cout<<"\nAckermann value:"<<ack_parameters[1]+1;
//cout<<"Clock cycles taken = %llu \n"<<2*(stop-start);
//cout<<"Time taken for the GCD = %.9f s\n"<<1.0*(stop-start)/(COUNTS_PER_SECOND);
}
```

```
Enter values for m and n:2
16
Done with inputs
Time taken by function: 4079 nanoseconds

Ackermann value:35
```

- Case 2: m = 3 & n = 14; System C compiler



```
}



}
//XTime_GetTime(&stop);
auto stop = high_resolution_clock::now();

auto duration = duration_cast<nanoseconds>(stop - start);
//cout << "\nDuration:" << duration;
cout << "\nTime taken by function: " << duration.count() << " nanoseconds" << endl;

cout<<"\nAckermann value:"<<ack_parameters[1]+1;
//cout<<"Clock cycles taken = %llu \n"<<2*(stop-start);
//cout<<"Time taken for the GCD = %.9f s\n"<<1.0*(stop-start)/(COUNTS_PER_SECOND);
}
```

```
Enter values for m and n:3
14
Done with inputs
Time taken by function: 38636031544 nanoseconds

Ackermann value:131069
```

	n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10	n=11	n=12	n=13	n=14
m=0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
m=1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
m=2	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
m=3	5	13	29	61	125	253	509	1021	2045	4093	8189	16381	32765	65533	131069
m=4	13	65533	-	-	-	-	-	-	-	-	-	-	-	-	-

Hardware/Software Performance Comparison:

m	n	Ackermann value	Hardware Implementation		Software Implementation	
			Clock cycles	Runtime	Clock cycles	Runtime
0	1	2	5380	80.7ns	158	237ns
2	10	23	11284	16.9 us	12026	18.04us
3	14	131069	305402425818	458.103s	449515995058	674.274s

- It is observed that the hardware implementation takes a lesser number of clock cycles than the software implementation and hardware implementation is comparatively faster but with a longer clock period.

Conclusion:

- Hardware implementation was achieved for the Ackermann function for a maximum value of m = 3 and n = 14 with the above-reported cycles and time values.
- We were able to compute for a value of (m,n) = (3,15) but took a long time for the computation.

- Software Implementation achieved a maximum of (m,n) = (3,16) but reported for a value (m,n) = (3,14) due to the performance comparison with the hardware.
- Also, we implemented the above logic using a system C Compiler where the time is calculated and the max(m,n) = (3,14) was calculated and reported.
- The runtime and the cycles of various combinations are tabulated above and compared.