# UNIVERSITY OF TEXAS AT DALLAS

800 W Campbell Rd, Richardson, TX 75080, USA


**EEDG 6370**
DESIGN AND ANALYSIS OF RECONFIGURABLE COMPUTING SYSTEMS
SPRING 2022

**DESIGN ASSIGNMENT FOR ACKERMANN FUNCTION**

**SUBMITTED BY**
**GROUP:** MRS

**Team Members:**
Gokul Sai Raghunath (GXR200021)
Abhishek Mahesh Kumar (AXM200255)
Aishwarya Surapuram (AXS210196)

## ACKNOWLEDGEMENT

We are grateful to Professor Dinesh Bhatia for providing us an opportunity to explore and conduct FPGA based projects. We also take this opportunity to express our gratitude to Mark Sears for his guidance in conducting the project.

## OBJECTIVE

In this project we must design and implement *Ackermann Function* in hardware utilizing BRAM on Zybo board along with analysis for memory requirement. We also must compare performance with software implemented on ARM core. Additionally, implement software version using a machine (computer). The Ackermann function *A (x, y)* is defined for integer x and y by

$$A(x,y)=\begin{cases} y+1 & \text{if } x=0 \\ A(x-1,1) & \text{if } y=0 \\ A(x-1,A(x,y-1)) & \text{otherwise} \end{cases}$$

## ACKERMANN FUNCTION – LOGIC ILLUSTRATION

Ackermann function is an example of a well-defined total function which is computable but not primitive recursive, providing a counterexample to the belief that every computable function was also primitive recursive. The function grows faster than multiple exponential function, depending on the first argument *x*, even for small inputs.

**Example:** Calculating A (1,5) takes 13 steps

= A(1,5)
= A(0,A(1,4))
= A(0,A(0,A(1,3)))
= A(0,A(0,A(0,A(1,2))))
= A(0,A(0,A(0,A(0,A(1,1)))))
= A(0,A(0,A(0,A(0,A(0,A(1,0))))))
= A(0,A(0,A(0,A(0,A(0,A(0,1))))))
= A(0,A(0,A(0,A(0,A(0,2)))))
= A(0,A(0,A(0,A(0,3))))
= A(0,A(0,A(0,4)))
= A(0,A(0,5))
= A(0,6)
= **7**

## HARDWARE IMPLMENTATION

## HIGH-LEVEL SYNTHESIS USING VIVADO HLS TOOL

### *1. CREATING NEW PROJECT*

Created a new project in Vivado HLS and added the C-based source i.e., "**ackremann.cpp**" and test bench file "**tb_ackermann.cpp**". Also select board part as "**xc7z010clg400-1**", the technical specifications of Zynq Zybo 7000 board were configured in solution configuration window. The source and testbench codes are included in the submission zip folder.

Following "pragmas" are used in HLS code:

**#pragma HLS INTERFACE bram port=Stack**

**#pragma HLS RESOURCE variable=Stack core=RAM_1P_BRAM**

**#pragma HLS ARRAY_RESHAPE variable=Stack complete dim=1**

The interface pragma specifies how RTL ports are created from the function definition during interface synthesis. BRAM Interface implements array arguments as standard RAM interface. Using this pragma on RTL design allows the memory interface to appear as single port in Vivado IP integrator. The library resource (core) pragma was used to specify the number of ports used to implement stack array variable. Vivado HLS implements the operations in the code using hardware cores. When multiple cores in the library can implement the operation, we can specify which core to use with. On absence of resource pragma, Vivado HLS determines the resource to use.

In the top function of the HLS, an array is used to iteratively compute the result of the Ackermann function. To utilize the onboard BRAM, the array is exposed as a function argument. This helps instruct the IP to perform all the array insertion and read from the BRAM.

**void ackermann(hls:: stream<int_side_ch> &inStream , hls:: stream<int_side_ch> &outStream, int Stack[LIMIT])**

### *2. RUNNING C SIMULATION*

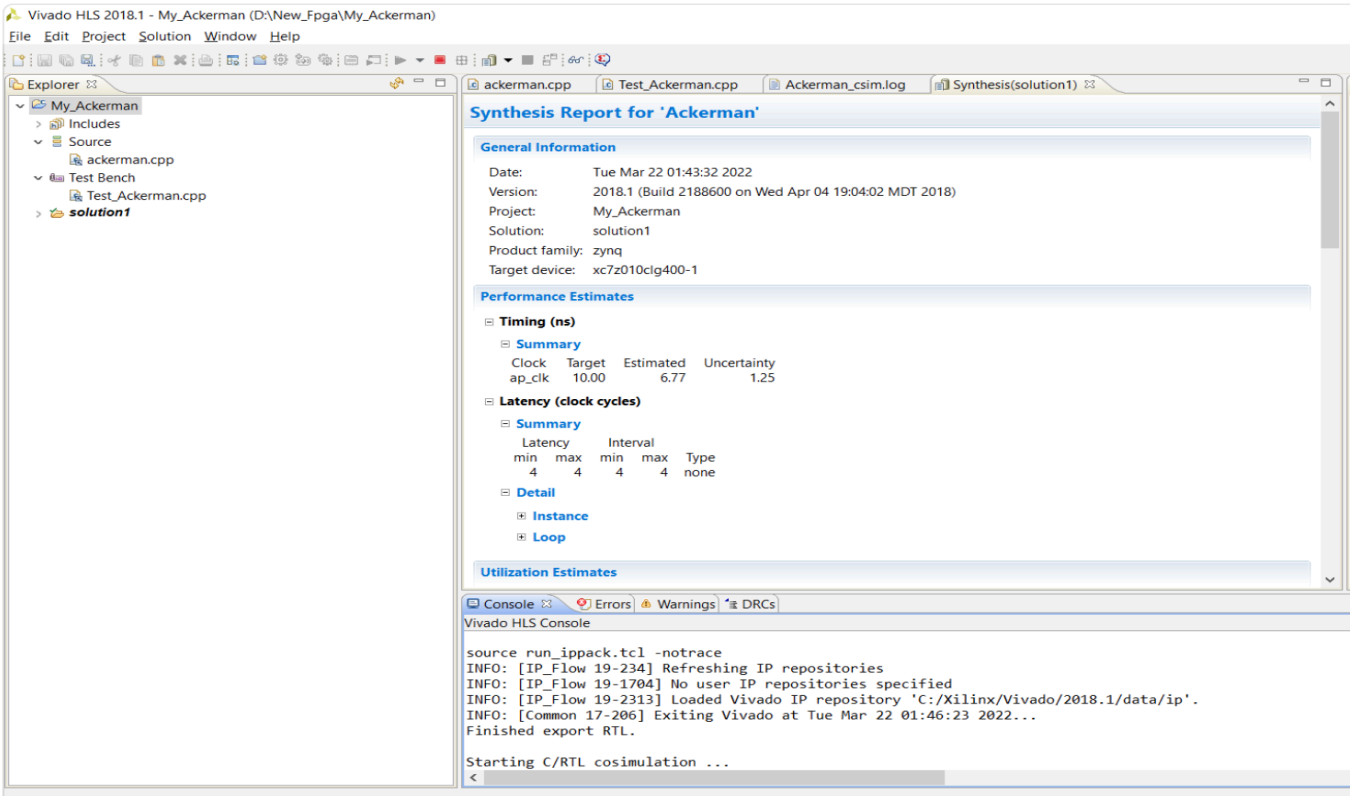
Confirmed the functionality (compilation and execution of the design application) by running C simulation and verified Ackermann Function works as expected

```
ackerman.cpp | tb_ackerman.cpp | Ackerman_csim.l
1 INFO: [SIM 2] *************** CSIM start ***************
2 INFO: [SIM 4] CSIM will launch GCC as the compiler.
3    Compiling ../../../../Ackerman/tb_ackerman.cpp in debug mode
4    Generating csim.exe
5  Input M = 3 Input N = 11 ack output val = 16381
6 INFO: [SIM 1] CSim done with 0 errors.
7 INFO: [SIM 3] *************** CSIM finish ***************
8
```

## *3. SYNTHESIZING THE DESIGN*

Synthesis was performed to compare the performance for ***9 different input variables***. Code functionality was substantiated in the console and synthesis report was generated to identify the timing and latency measures.

**EXAMPLE CASE: A = 3, B = 11**



## *4. RUN C/RTL CO-SIMULATION*

RTL Co-Simulation: High-Level synthesis can re-use the C test bench to verify the RTL using simulation. Co-Simulation was first performed for Verilog and analyzed the latency. Additionally, in the option for Dump Trace: all was selected.

***Co-Simulation was performed for Verilog***.

**Cosimulation Report for 'ackermann'**

**Result**

| | | Latency | | | Interval | | |
|---|---|---|---|---|---|---|---|
| RTL | Status | min | avg | max | min | avg | max |
| VHDL | NA | NA | NA | NA | NA | NA | NA |
| Verilog | Pass | 156448246 | 156448246 | 156448246 | NA | NA | NA |

Export the report(.html) using the Export Wizard

## *5. EXPORT RTL AND IMPLEMENT*

The final step in the High-Level Synthesis flow was to package the design as an IP block for use with other tools in the Xilinx Design Suite. The design was exported using VHDL as the language. The IP packaging process created a package for the Vivado IP catalog which would be used in Vivado IPI. By selecting the *"Evaluate Generated RTL",* the RTL underwent Synthesis, Placement and Routing evaluations also. Implementation results in Vivado HLS is shown below.

Project: Ackermann
Solution: solution1
Device target: xc7z010-clg400-1
Implementation tool: Xilinx Vivado v.2019.2

**Resource Usage**

| | Verilog |
|---|---|
| SLICE | 0 |
| LUT | 472 |
| FF | 504 |
| DSP | 0 |
| BRAM | 36 |
| SRL | 0 |

**Final Timing**

| | Verilog |
|---|---|
| CP required | 8.000 |
| CP achieved post-synthesis | 6.530 |

Timing met

*Resource Usage for example case A(3,11)*

We observed that the timing constraint was met with achieved period 6.530ns and also the resource usage – type and number of resources used for this implementation.
The IP package was created successfully and made ready to be added to the Vivado IP catalog.

## *6. BLOCK DIAGRAM (IPs & INTERCONNECTS)*

On creating a new project in Vivado IDE with appropriate board configuration settings, the Ackermann function IP was imported as a User-defined IP. We also added Zynq 7 processing system and other related IPs. The blocks were interconnected using IP integrator designer assistance tool – run block automation/auto-connect and further fine-tuned manually.

The run connection automation automatically instantiated two further IP blocks:

1. **Processor System Reset Module** – This provides customized rests for an entire processing system, including the peripherals, interconnect, and the processor itself.
2. **AXI interconnect** – This provides AXI interconnect for the system, allowing further IP and peripherals in the programmable logic to communicate with the main processing system.

The IP integrator automatically assigned memory map for all IP that was present in the design. For Zynq 7000 SoC processor, the Generic Interrupt Controller block handles the interrupts.

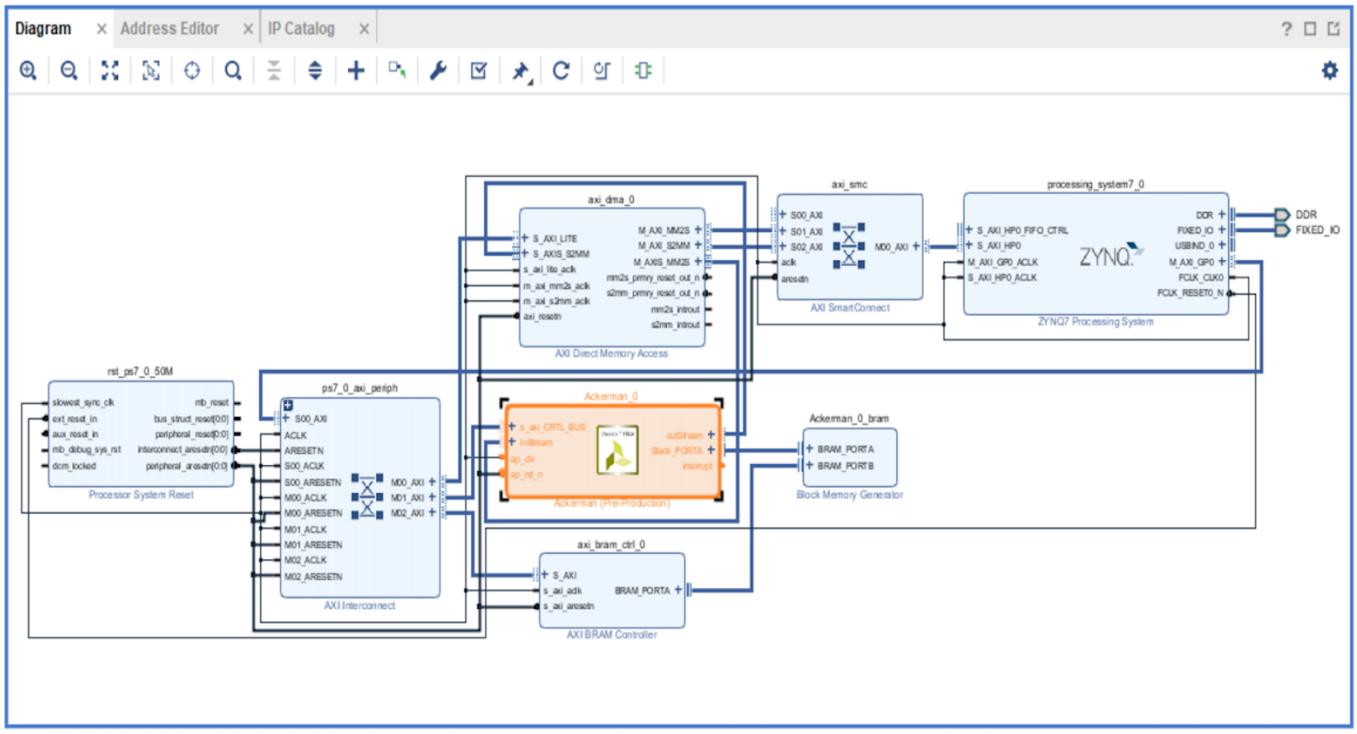
**BLOCK RAM** is a true dual port RAM module instantiated into FPGA fabric to provide on-chip storage for a relatively large set of data. The two types of BRAM memories available in a device can hold either 18k or 36k bits and the available amount of these memories is device specific.

**AXI BRAM Controller** – This IP core is designed as an AXI endpoint slave IP for integration with the AXI interconnect and system master devices to communicate to local BRAM.

**Block RAM Generator** – This IP core automates the creation of resource and power optimized block memories for the FPGA. The core enables us to create block memory functions to suit varied requirements in order to create minimized area, high performance or low power solution.

The block diagram is shown below:

*Block Diagram – Ackermann Function*

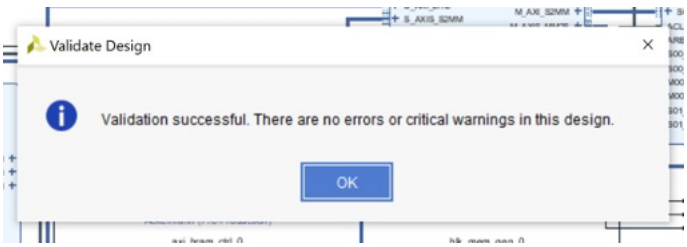



## *7. DESIGN VALIDATION AND CREATING HDL WRAPPER*

The design was validated to run a DRC (design-rule-check). The IP integrator runs basic checks in real time as the design in being assembled. On successful validation, the HDL wrapper was created around the top-level block design of the system.

All the source files for the IP blocks that were used in the IP integrator block diagram, as well as constraint file was generated during the synthesis process. As we specified Verilog as the target language when creating the project, all the generated source files were in Verilog language.

*Successful Design Validation*



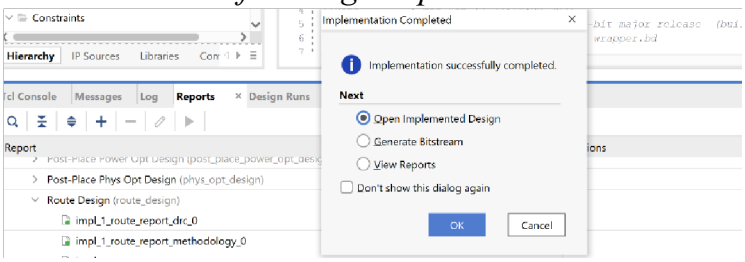
Vivado offered two choices on creating the HDL wrapper, we chose the default option. The other option allows us to create a modifiable script which required manually updating the wrapper every time the port-level changes were made in block design.

## *8. SYNTHESIS, IMPLEMENTATION AND BITSTREAM GENERATION*

The vivado IDE enabled editing the design constrains and netlists on the in-memory design. When the design was saved, the constraint changes were written back to the original source XDC files. This flexibility thus supports exploration of alternate timing and physical constrains while keeping the original source files intact.

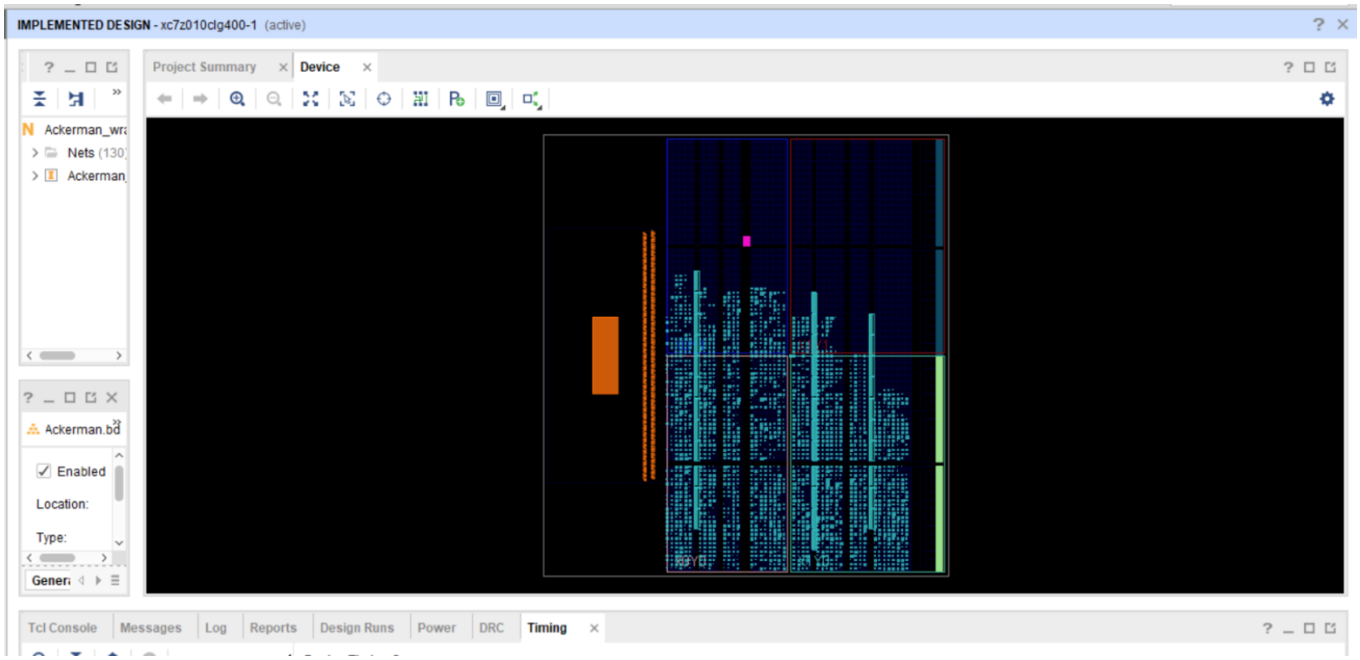
We ran the synthesis and implementation commands. Both the processes launched and were placed into background process after some initialization. Meanwhile, we analyzed the synthesized design using the command Windows > Reports.

*Successful design implementation*



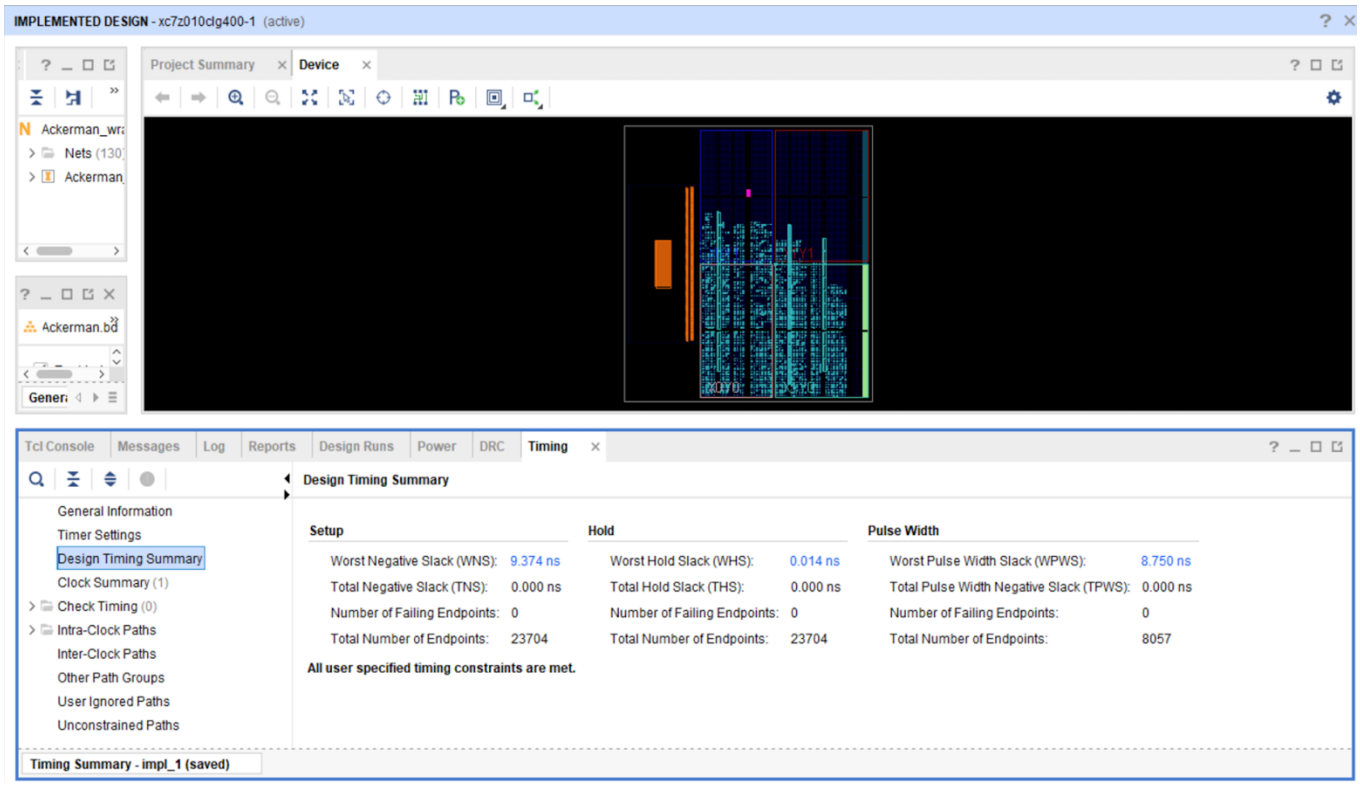

The implemented design after synthesis displaying placement and routing results is shown below:

***Design Implementation***

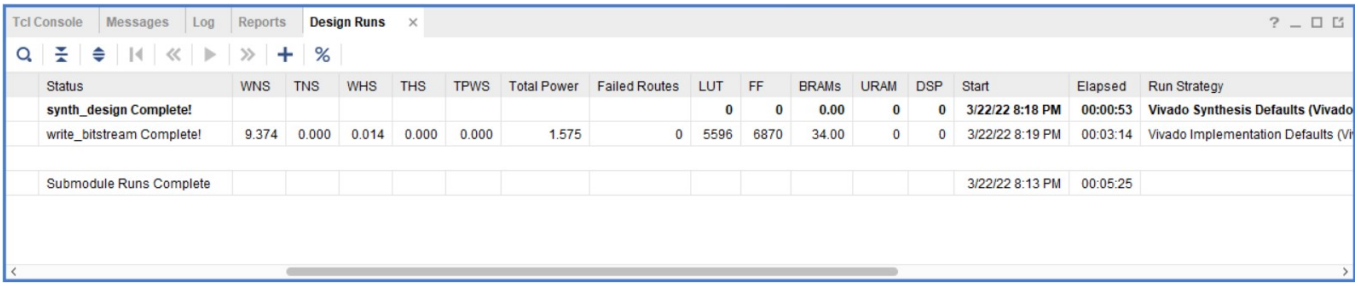


Once the implementation was completed, the tool provided various reports ranging from methodology, power utilization and timing summary.

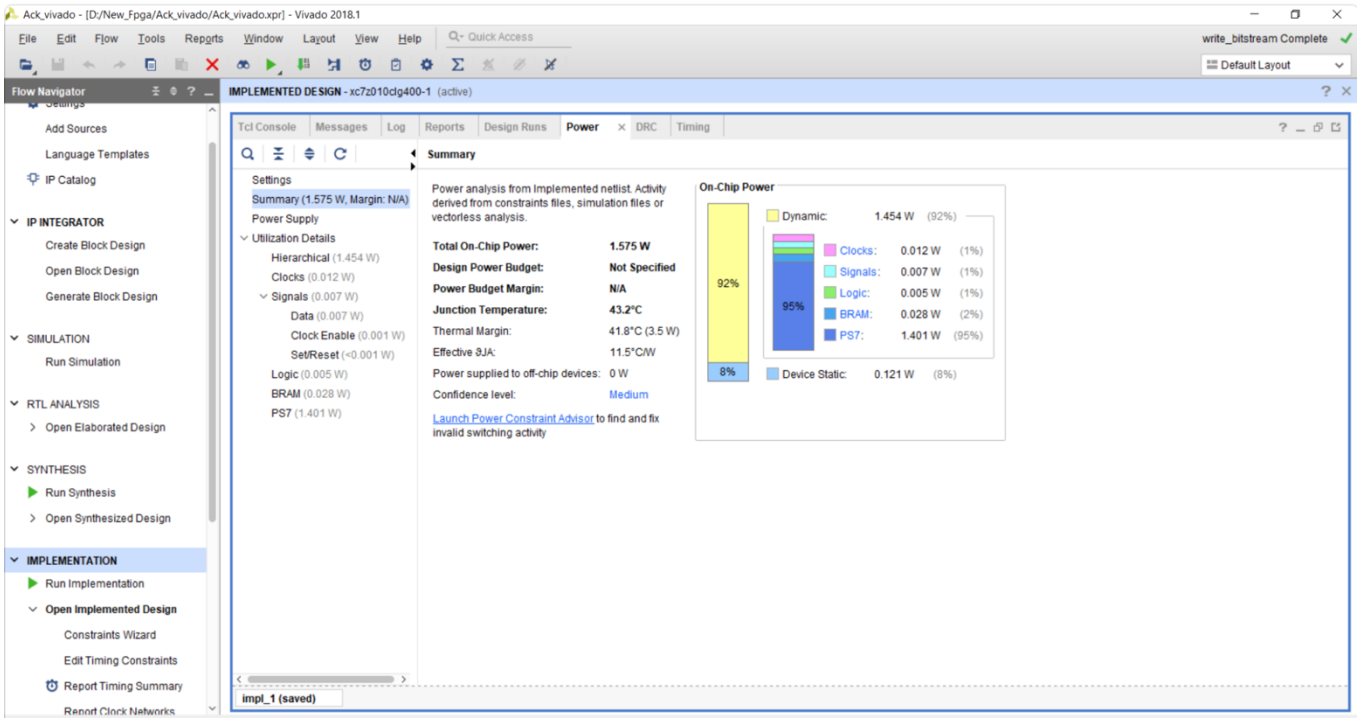
### *Design Timing Summary*




### *Implementation – design runs*



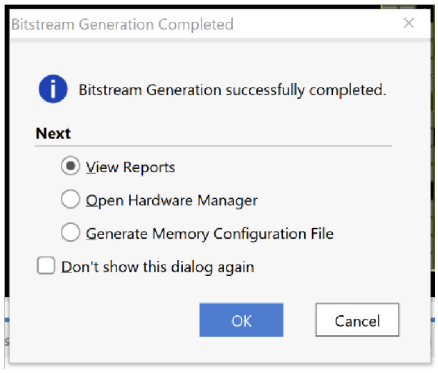
***Power analysis***




After successful implementation of the design, the timing report was generated by entering the below command in Vivado TCL console window

report_timing > timing.log

Finally, Bitstream was generated – and with that the building of hardware image was completed. It was then exported to software environment (SDK) where we built a software application to control and interact with our custom Ackermann function hardware. For this, we exported the hardware after including bitstream in the dialog box and launched the SDK in Vivado.



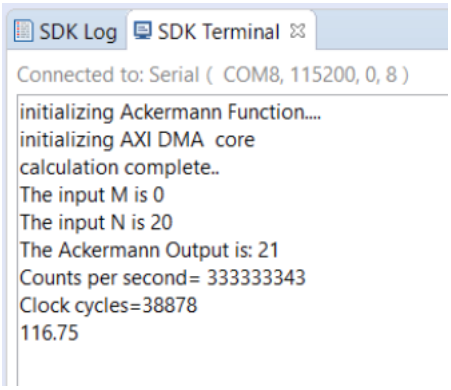

*Successful bitstream generation*

## 9. CREATING PROJECT ON SDK

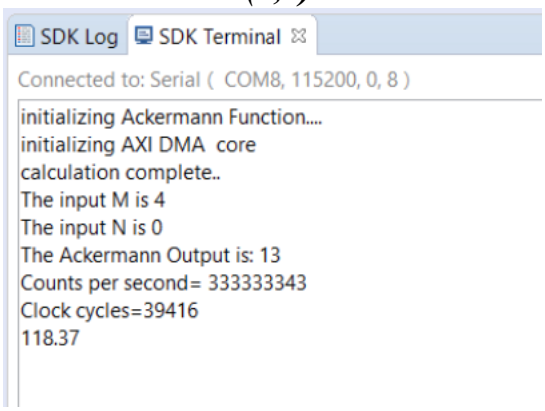
With the exported Ackermann Function hardware including bitstream, a new application project was created. The helloworld.c file was edited to implement Ackermann functionality. The bitstream and SDK code for hardware implementation, together with the README file for execution are included in the submission zip folder. The ZYBO board was connected to the PC using an UART USB cable. We set the baud rate in serial port setup to 115200 and programmed the FPGA. Finally, using the command *Launch on HW system debugger* the program was run and the results were viewed on the SDK terminal.

## *10. SDK TERMINAL OUTPUTS FOR HARDWARE IMPLEMENTATION USING FPGA*

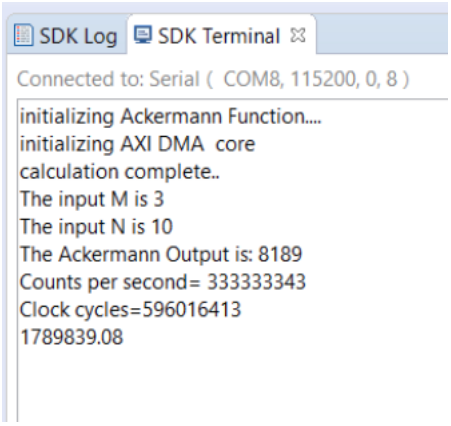
***A (0,20)***



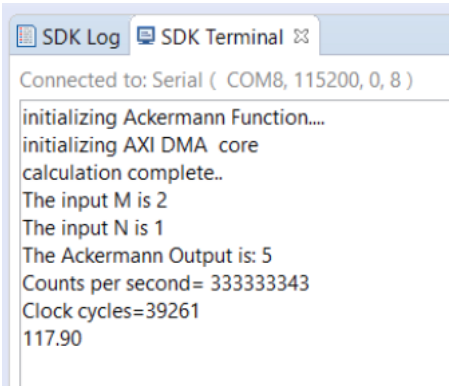

***A (4,0)***



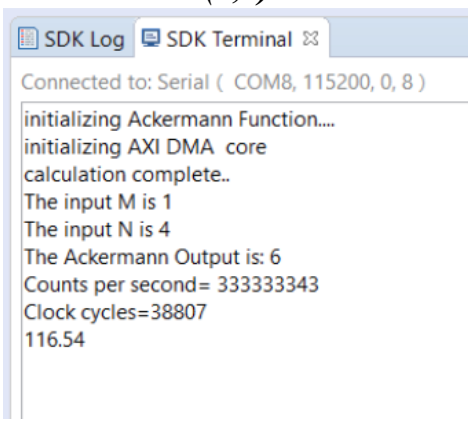

***A (3,10)***



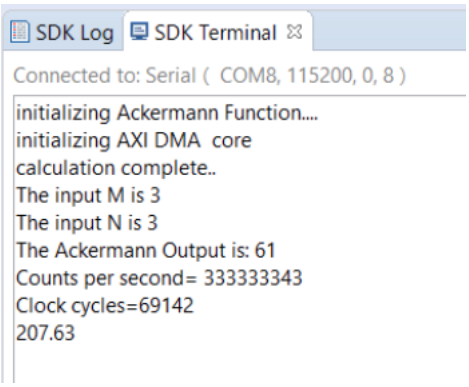

***A (2,1)***



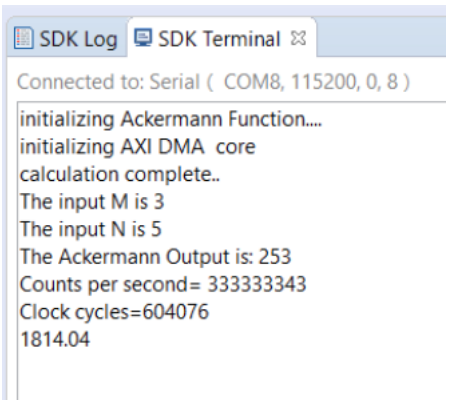
***A (1,4)***



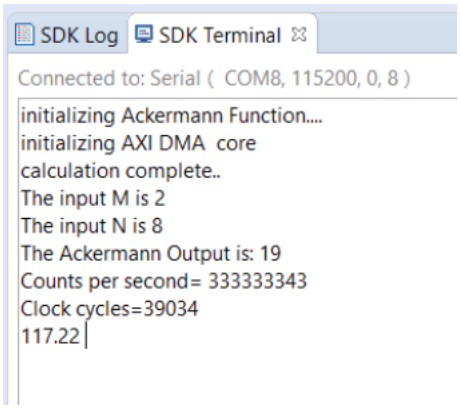

***A (3,3)***



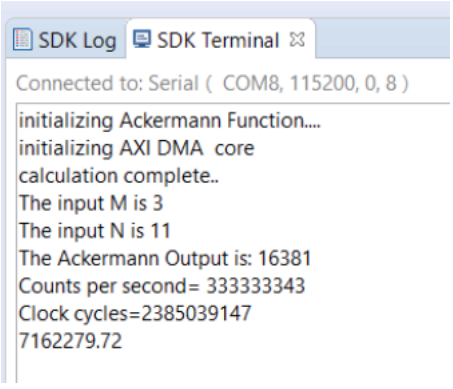

***A (3,5)***




***A (2,8)***




***A (3,11)***



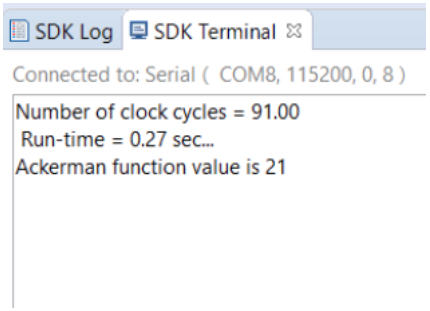
## SOFTWARE IMPLMENTATION ON PROCESSOR CORE

Implementation of Ackermann function code on the ARM core running via SDK application was carried out. This helped us to execute on the processor embedded inside the Zynq device. The application was burnt into the ARM processor by running using the command *Launch on Hardware (GDB).* The SDK code for software implementation, together with README file for execution are included in the submission zip folder.

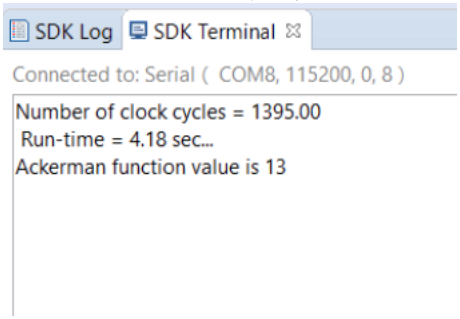
The Ackermann function was implemented using an array to iteratively compute the result. In order to compute the elapsed time and number of clock cycles, we have used *XTime_l.h* header file and called the *XTime_GetTime()* module at the start and end of the Ackermann implementation module for computation.

## *SDK TERMINAL OUTPUTS FOR* SOFTWARE IMPLMENTATION ON ARM PROCESSOR CORE

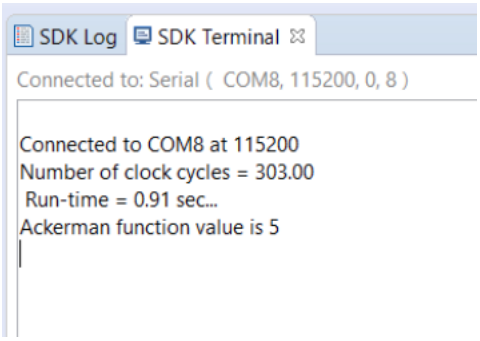
***A (0,20)***



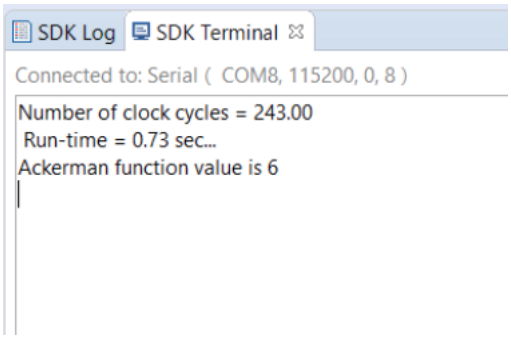

***A (4,0)***




***A (2,1)***

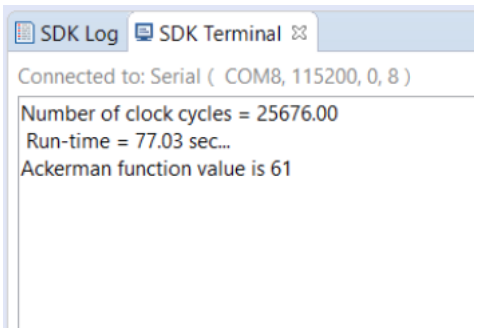


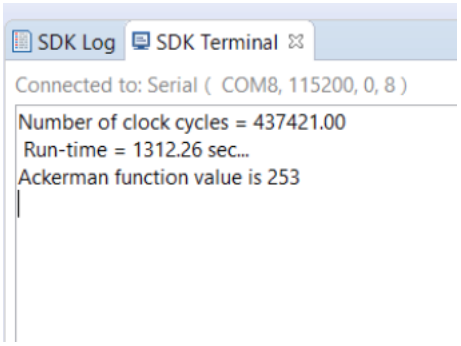

***A (1,4)***



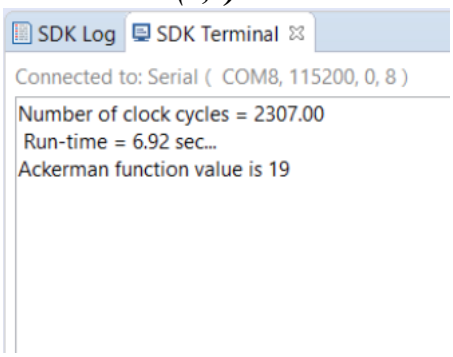

***A (3,3)***

***A (3,5)***




***A (2,8)***

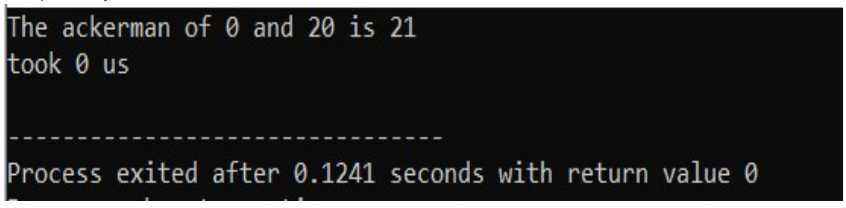



## SOFTWARE IMPLMENTATION ON A MACHINE

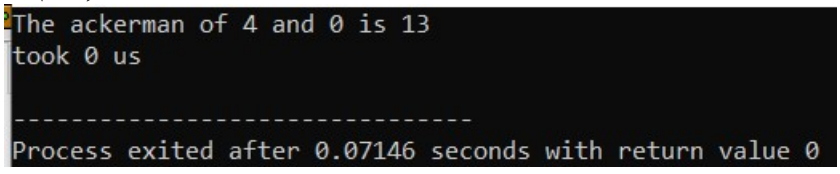
The Ackermann function was also analyzed by running in the computer. The software was written using CodeBlocks IDE and implemented in C. The function was implemented the same way as the software implementation using iterative loops.

To compute the CPU run time and number of clock cycles, we have used function clock () from library time.h at the start and end of the Ackermann function. The C code for machine implementation, together with README file for execution are included in the submission zip folder.

***A (0,20)***




***A (4,0)***



***A (3,10)***

```
The ackerman of 3 and 10 is 8189
took 100435 us

--------------------------------
Process exited after 0.331 seconds with return value 0
```

***A (2,1)***

```
The ackerman of 2 and 1 is 5
took 0 us

--------------------------------
Process exited after 0.05989 seconds with return value 0
```

***A (1,4)***

```
The ackerman of 1 and 4 is 6
took 0 us

--------------------------------
Process exited after 0.03258 seconds with return value 0
```

***A (3,3)***

```
The ackerman of 3 and 3 is 61
took 0 us

--------------------------------
Process exited after 0.05742 seconds with return value 0
```

***A (3,5)***

```
The ackerman of 3 and 5 is 253
took 0 us

--------------------------------
Process exited after 0.03558 seconds with return value 0
```

***A (2,8)***

```
The ackerman of 2 and 8 is 19
took 0 us

--------------------------------
Process exited after 0.02881 seconds with return value 0
```

*A (3,11)*

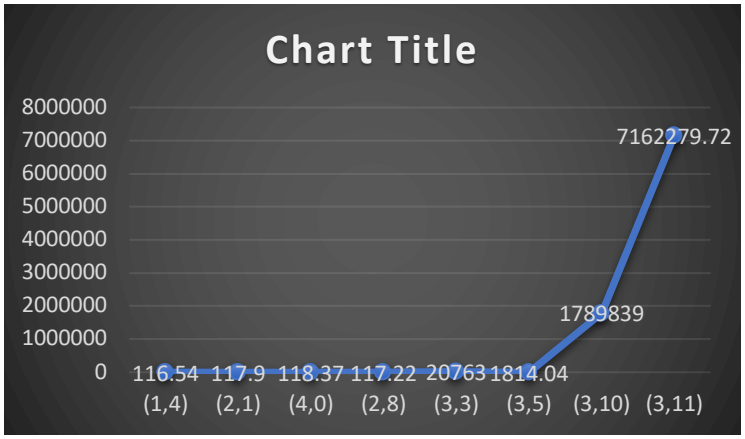
```
The ackerman of 3 and 11 is 16381
took 392975 us

--------------------------------
Process exited after 0.6141 seconds with return value 0
```

## PERFORMANCE COMPARISON

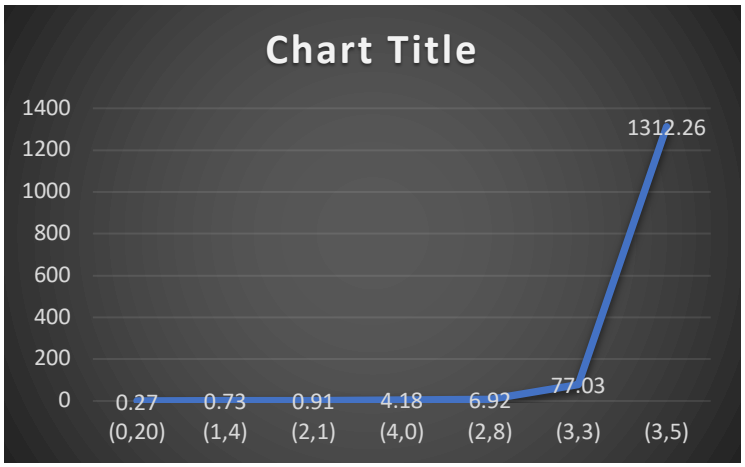
### a) RUN-TIME ESTIMATION

| COMPARISON: RUN-TIME REQUIREMENTS | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Case | Input M | Input N | Ackerman Function | Hardware Implementation (FPGA) | | Software Implementation (On-Chip ARM) | | Software Implementation (Local System) |
| | | | | Clock Cycle | Run-Time (in Milliseconds) | Clock Cycle | Run-Time (in Milliseconds) | Run-Time (in Milliseconds) |
| (1,4) | 1 | 4 | 6 | 38807 | 116.54 | 243 | 0.73 | ~0 |
| (0,20) | 0 | 20 | 21 | 38878 | 116.75 | 91 | 0.27 | ~0 |
| (2,1) | 2 | 1 | 5 | 39261 | 117.9 | 303 | 0.91 | ~0 |
| (2,8) | 2 | 8 | 19 | 39034 | 117.22 | 2307 | 6.92 | ~0 |
| (4,0) | 4 | 0 | 13 | 39416 | 118.37 | 1395 | 4.18 | ~0 |
| (3,3) | 3 | 3 | 61 | 69142 | 20763 | 25676 | 77.03 | ~0 |
| (3,5) | 3 | 5 | 253 | 604076 | 1814.04 | 437421 | 1312.26 | ~0 |
| (3,10) | 3 | 10 | 8189 | 596016413 | 1789839 | | | 100.435 |
| (3,11) | 3 | 11 | 16381 | 2385039147 | 7162279.72 | | | 392.975 |

**Run time plot for Hardware Implementation:**



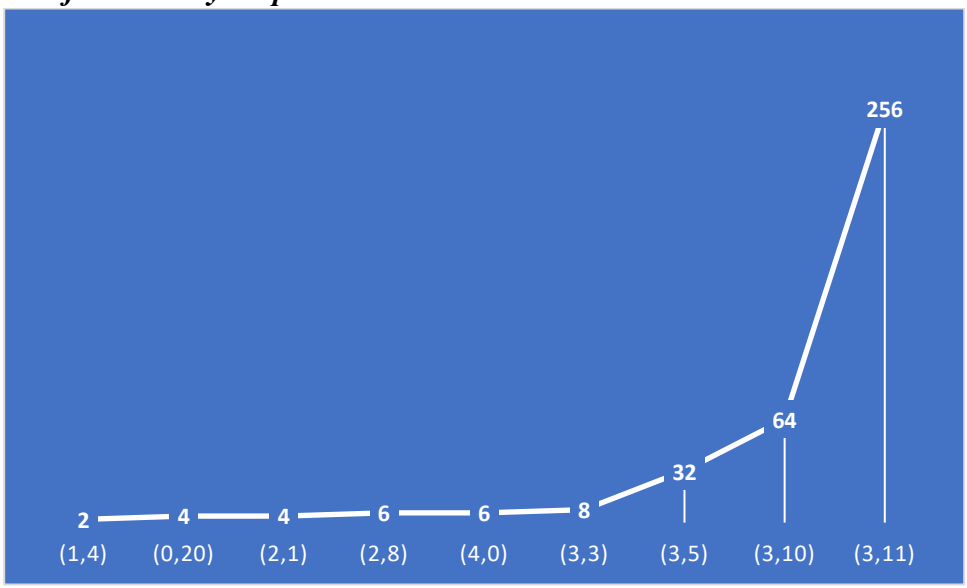
**Run time plot for Software Implementation (On- ARM Core):**



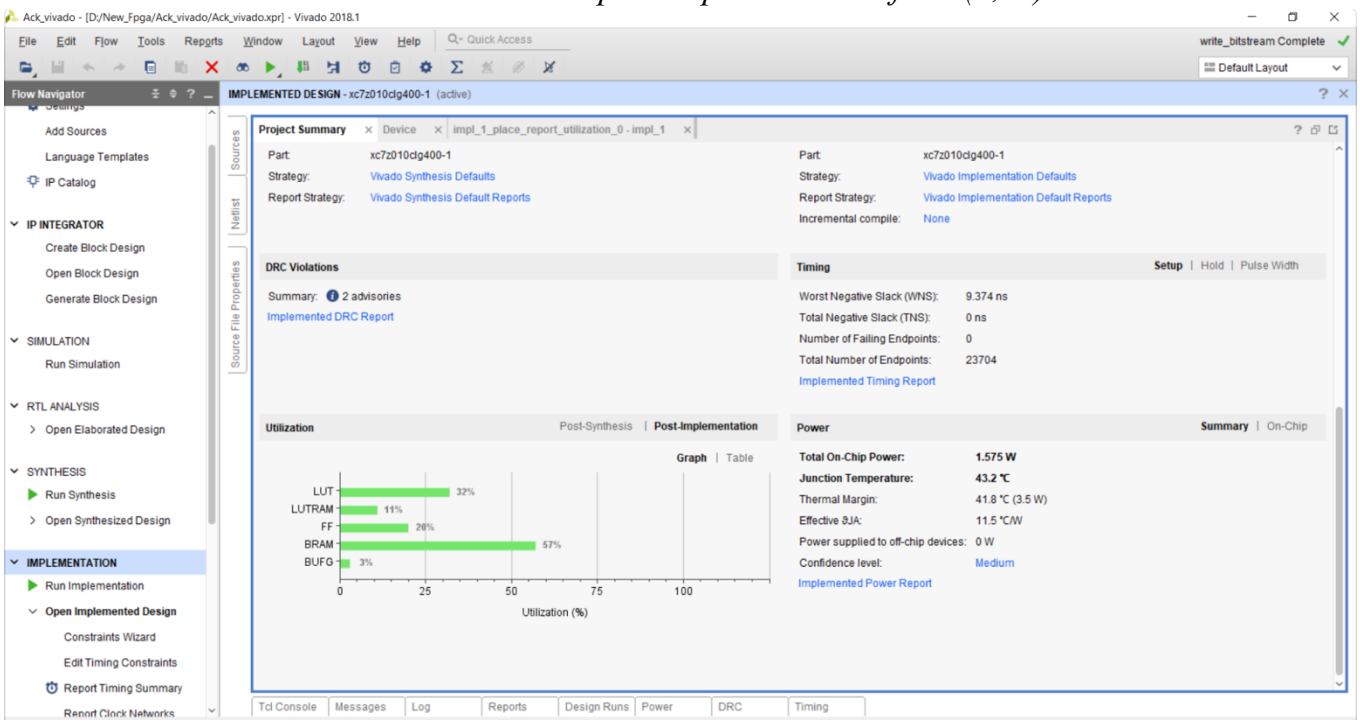

## b) MEMORY REQUIREMENT

| Case | Input M | Input N | Ackerman Function | BRAM AVAILABLE: 120 | |
|---|---|---|---|---|---|
| | | | | Resources Utilized | Utilization Percentage |
| (1,4) | 1 | 4 | 6 | 2 | 1% |
| (0,20) | 0 | 20 | 21 | 4 | 3% |
| (2,1) | 2 | 1 | 5 | 4 | 3% |
| (2,8) | 2 | 8 | 19 | 6 | 5% |
| (4,0) | 4 | 0 | 13 | 6 | 5% |
| (3,3) | 3 | 3 | 61 | 8 | 6% |
| (3,5) | 3 | 5 | 253 | 32 | 27% |
| (3,10) | 3 | 10 | 8189 | 64 | 57% |
| (3,11) | 3 | 11 | 16381 | 256 | 213% |

***Plot for Memory Requirement***




*Resource Utilization post-implementation for A(3,11)*




While HLS allows successful implementation for larger input values of the Ackermann Function, the BRAM utilization exceeds the available resources of Zybo Z7-10 device. This leads to implementation failure in IP integrator.

*C Synthesis for A (3,13) in HLS showing the exceeding Resource Utilization*

**Utilization Estimates**

**Summary**

| Name | BRAM_18K | DSP48E | FF | LUT | URAM |
|---|---|---|---|---|---|
| DSP | - | - | - | - | - |
| Expression | - | - | 0 | 273 | - |
| FIFO | - | - | - | - | - |
| Instance | 0 | - | 36 | 40 | - |
| Memory | 256 | - | 0 | 0 | 0 |
| Multiplexer | - | - | - | 152 | - |
| Register | - | - | 240 | - | - |
| Total | 256 | 0 | 276 | 465 | 0 |
| Available | 120 | 80 | 35200 | 17600 | 0 |
| Utilization (%) | 213 | 0 | ~0 | 2 | 0 |

**Detail**

**Instance**

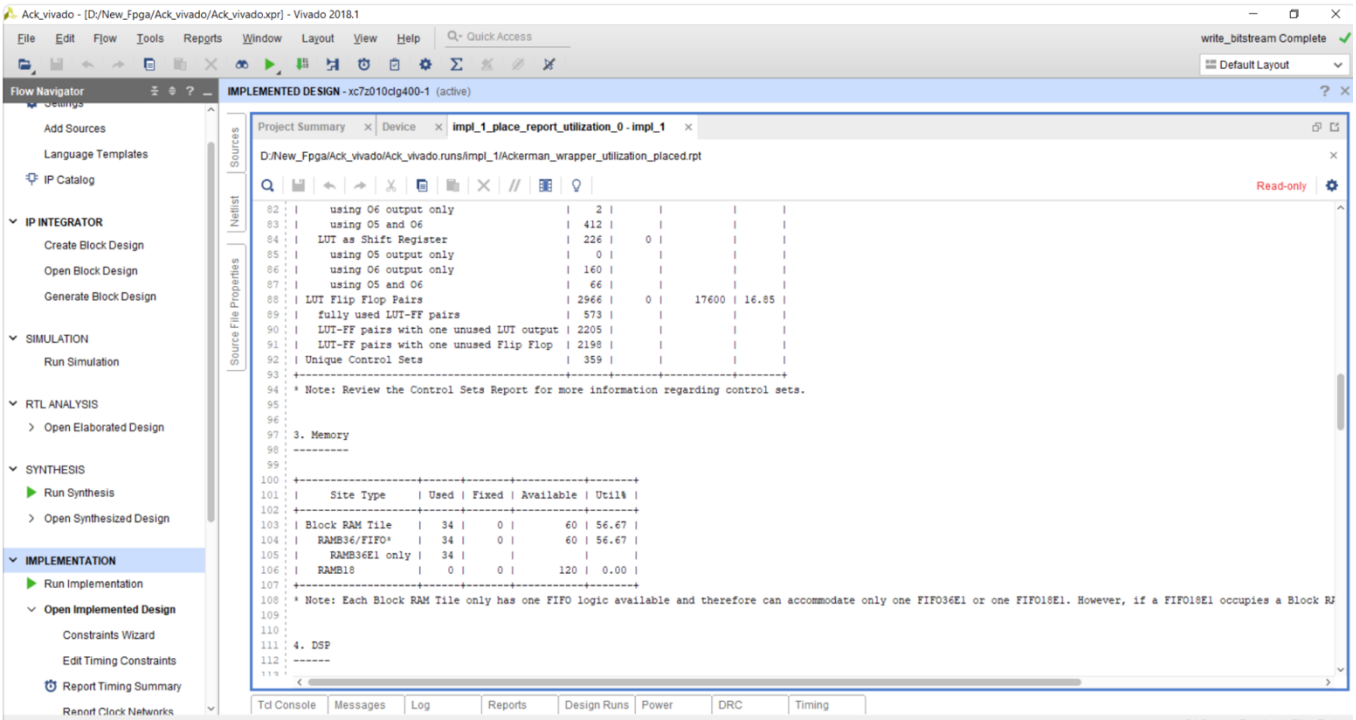
**DSP48E**

**Memory**

| Memory | Module | BRAM_18K | FF | LUT | URAM | Words | Bits | Banks | W*Bits*Banks |
|---|---|---|---|---|---|---|---|---|---|
| stack_arr_U | ackermann_stack_arr | 256 | 0 | 0 | 0 | 66000 | 32 | 1 | 2112000 |
| Total | 1 | 256 | 0 | 0 | 0 | 66000 | 32 | 1 | 2112000 |

After successful implementation of the design, the resource utilization report was generated by entering the report_utilization command in Vivado TCL console window.

*Memory Utilization post-implementation for A(3,11) within available BRAM resources*



*Memory Utilization post-implementation for A(3,13)* ***exceeding*** *available BRAM resources*
*Maximum Utilization % = 100*

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Block RAM Tile | 128 | 0 | 60 | 213.00 |
| RAMB36/FIFO* | 128 | 0 | 60 | 213.00 |
| RAMB36E1 only | 128 | | | |
| RAMB18 | 0 | 0 | 120 | 0.00 |

* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1

## CONCLUSION

We implemented Ackermann function in hardware integrating BRAM and compared performance with software implemented on ARM core. The ARM software will execute at a much lesser time when compared to hardware.

*Result Evaluation:*

1. HLS Simulation: Achieved results for the maximum input of (3,11) and (4,1).
2. Hardware Implementation: Achieved results for the maximum input value of (3,11), above which the implementation failed due to exceeding BRAM resource utilization.
3. Software Implementation: Achieved results for the maximum input value of (3,10).

Additionally, we implemented software version using Dev C++ in a machine. We evaluated results for different input combinations and observed that runtime is very less in the machine compared to hardware implementation on FPGA and software implementation on ARM Core. Run-Time and memory requirements were plotted for varying values of input variables for Ackermann's function.