Project report on High level synthesis using C programming language and Vivado HLS

Name-Pragya Pusp

ID-101164933

SYSC807Z-Advanced topics in Computer systems

Table of contents-

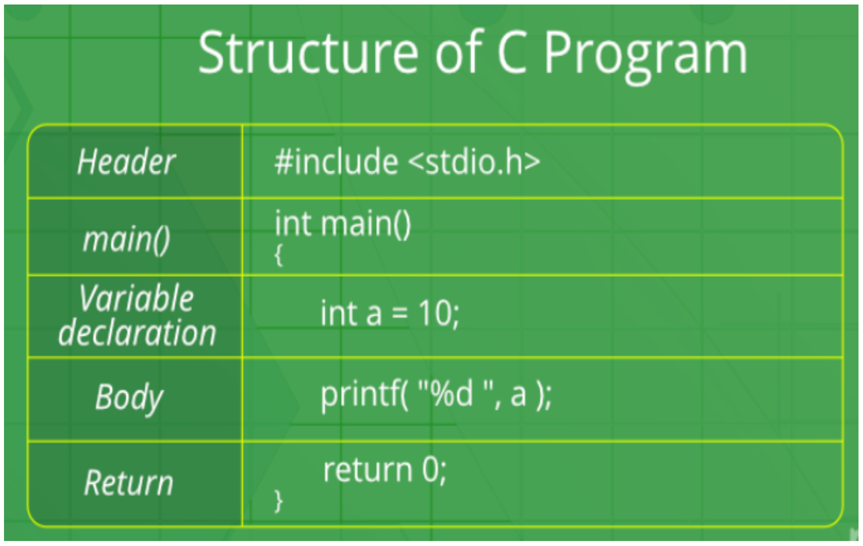
Chapter 1-Basics of C Programming

C programming is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system.

Text editors like CodeBlocks can be used to type, debug and observe the results of C language.

The files you create with your editor are called the source files and they contain the program source codes. The source files for C programs are typically named with the extension “**.c**”.

The source code written in source file is the human readable source for your program. It needs to be “compiled”, into machine language so that your CPU can execute the program as per the instructions given. The compiler compiles the source codes into final executable programs.



Structure of C Program

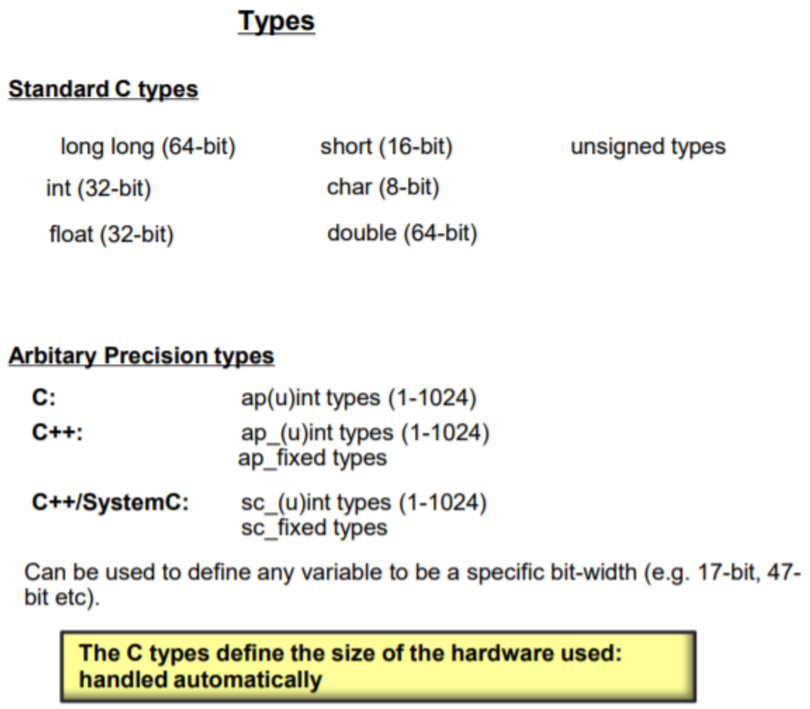
Header	#include <stdio.h>
main()	int main() {
Variable declaration	int a = 10;
Body	printf("%d ", a);

Return	return 0;
	}

Fig.1-Structure of C program

Fig.1 shows the basic structure of any C program. The first line of the program #include <stdio.h> is a preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation. The next line *int main()* is the main function where the program execution begins. Then, we declare variables and include body statements which might include calling functions, printf statements, expressions, comments, etc. The next line **return 0;** terminates the main() function and returns the value 0.

The smallest individual units in a C program are known as tokens. In a C source program, the basic element recognized by the compiler is the "token." A token is source-program text that the compiler does not break down into component elements.
C has 6 different types of tokens –
1. Keywords [e.g. float, int, while]
2. Identifiers [e.g. main, amount]
3. Constants [e.g. -25.6, 100]
4. Strings [e.g. "SMIT", "year"]
5. Special Symbols [e.g. {, }, [,]]
6. Operators [e.g. +, -, *]
C programs are written using these tokens and the general syntax.



Types

Standard C types

long long (64-bit) short (16-bit) unsigned types
int (32-bit) char (8-bit)
float (32-bit) double (64-bit)

Arbitary Precision types

C: ap(u)int types (1-1024)
C++: ap_(u)int types (1-1024)
ap_fixed types
C++/SystemC: sc_(u)int types (1-1024)
sc_fixed types

Can be used to define any variable to be a specific bit-width (e.g. 17-bit, 47-bit etc).

The C types define the size of the hardware used: handled automatically

Fig.2- Data types

Chapter 2- Concepts of Vivado HLS

The Xilinx Vivado High-Level Synthesis (HLS) tool transforms a C specification into a register transfer level (RTL) implementation that you can synthesize into a Xilinx field programmable gate array (FPGA).

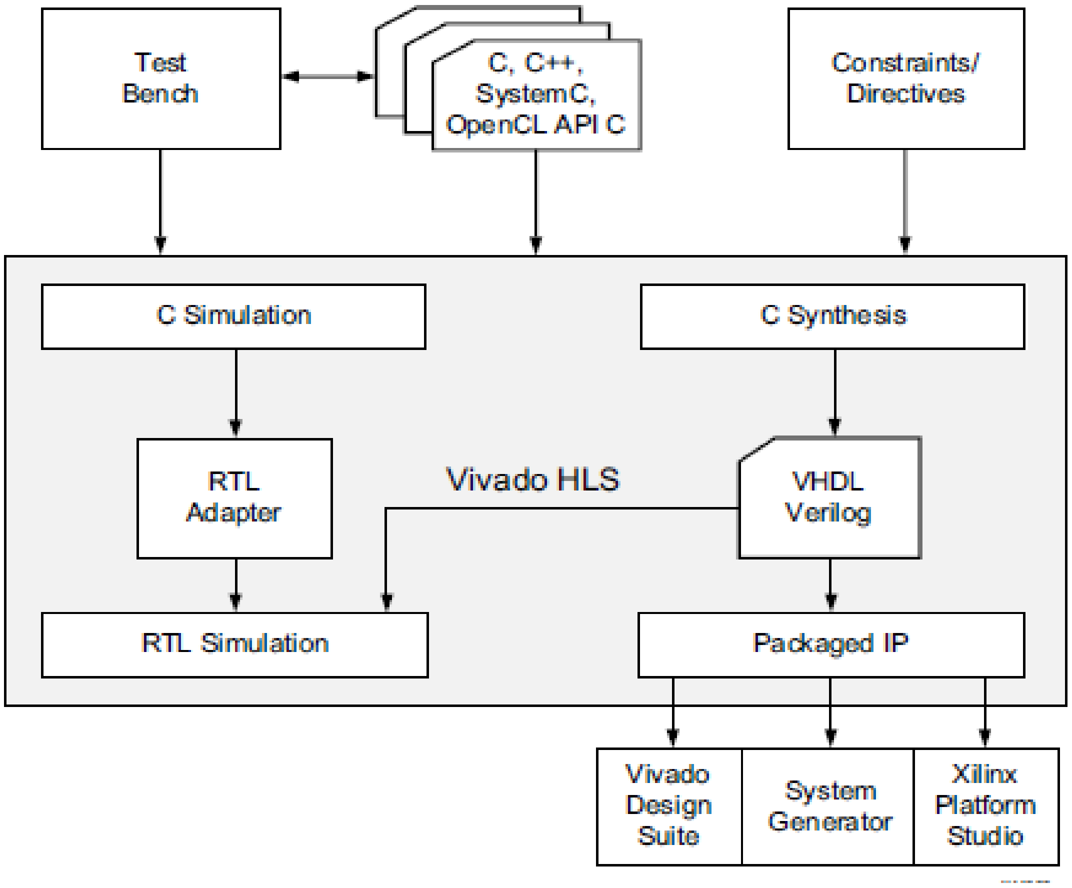


Fig.3-Vivado HLS design flow

High-Level Synthesis Benefits-

• Improved productivity for hardware designers
Hardware designers can work at a higher level of abstraction while creating high-performance hardware.

• Improved system performance for software designers
Software developers can accelerate the computationally intensive parts of their algorithms on a new compilation target, the FPGA.

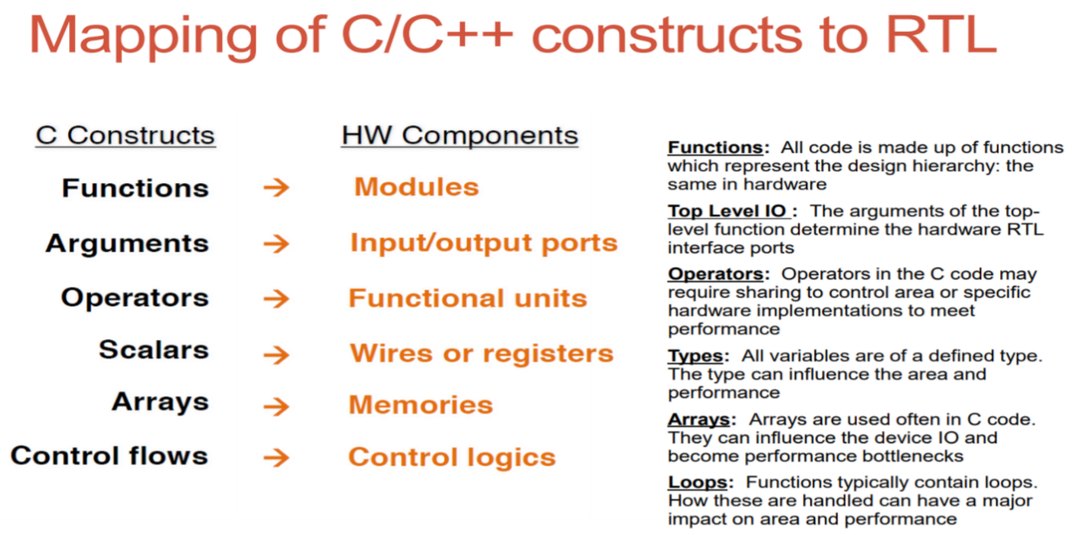
Using a high-level synthesis design methodology allows you to:
• Develop algorithms at the C-level
Work at a level that is abstract from the implementation details, which consume development time.
• Verify at the C-level
Validate the functional correctness of the design more quickly than with traditional hardware description languages.
• Control the C synthesis process through optimization directives
Create specific high-performance hardware implementations.
• Create multiple implementations from the C source code using optimization directives.
Explore the design space, which increases the likelihood of finding an optimal implementation.
• Create readable and portable C source code.
Retarget the C source into different devices as well as incorporate the C source into new projects.
High-level synthesis includes the following phases:

• Scheduling
Determines which operations occur during each clock cycle based on:
° Length of the clock cycle or clock frequency
° Time it takes for the operation to complete, as defined by the target device
° User-specified optimization directives
If the clock period is longer or a faster FPGA is targeted, more operations are completed within a single clock cycle, and all operations might complete in one clock cycle.
Conversely, if the clock period is shorter or a slower FPGA is targeted, high-level synthesis automatically schedules the operations over more clock cycles, and some operations might need to be implemented as multicycle resources.

• Binding
Determines which hardware resource implements each scheduled operation. To implement the optimal solution, high-level synthesis uses information about the target device.

• Control logic extraction
Extracts the control logic to create a finite state machine (FSM) that sequences the operations in the RTL design.

Performance Metrics:
• Area: Amount of hardware resources required to implement the design based on the resources available in the FPGA, including look-up tables (LUT), registers, block RAMs, and DSP48s.
• Latency: Number of clock cycles required for the function to compute all output values.
• Initiation interval (II): Number of clock cycles before the function can accept new input data.
• Loop iteration latency: Number of clock cycles it takes to complete one iteration of the loop.
• Loop initiation interval: Number of clock cycle before the next iteration of the loop starts to process data.
• Loop latency: Number of cycles to execute all iterations of the loop.


Mapping of C/C++ constructs to RTL

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

Functions: All code is made up of functions which represent the design hierarchy: the same in hardware

Top Level IO : The arguments of the top-level function determine the hardware RTL interface ports

Operators: Operators in the C code may require sharing to control area or specific hardware implementations to meet performance

Types: All variables are of a defined type. The type can influence the area and performance

Arrays: Arrays are used often in C code. They can influence the device IO and become performance bottlenecks

Loops: Functions typically contain loops. How these are handled can have a major impact on area and performance

Fig.4- Mapping of C constructs to RTL

In the scheduling phase of Fig.4, high-level synthesis schedules the following operations to occur during each clock cycle:

• First clock cycle: Multiplication and the first addition

• Second clock cycle: Second addition and output generation

In the initial binding phase of this example, high-level synthesis implements the multiplier operation using a combinational multiplier (Mul) and implements both add operations using a combinational adder/subtractor (AddSub).

In the target binding phase, high-level synthesis implements both the multiplier and one of the addition operations using a DSP48 resource. The DSP48 resource is a computational block available in the FPGA architecture that provides the ideal balance of high-performance and efficient implementation.

```
int foo(char x, char a, char b, char c) {
  char y;
  y = x*a+b+c;
  return y;
}
```

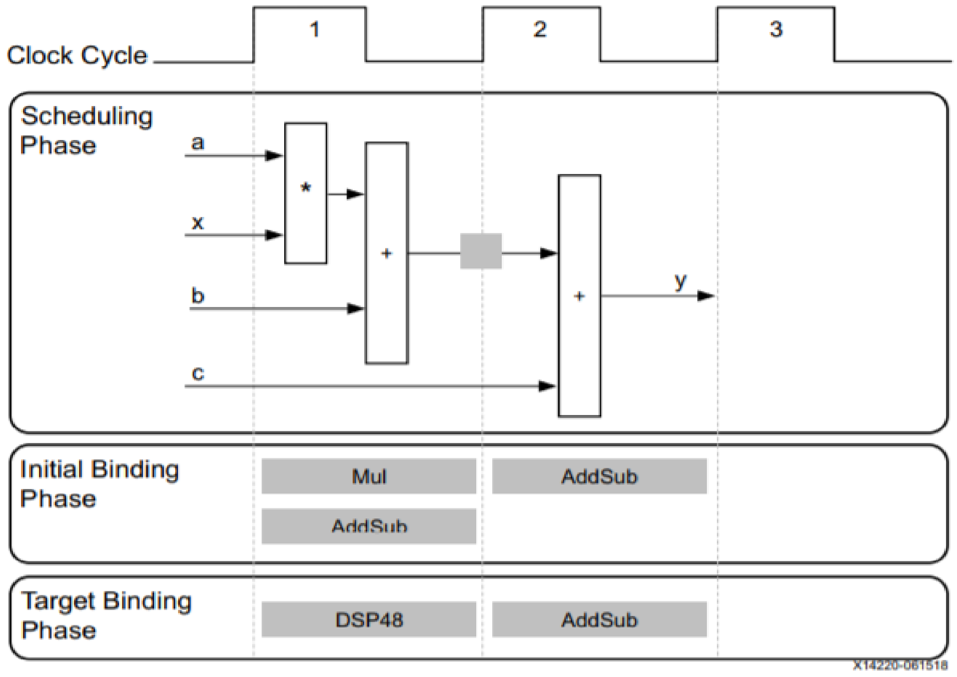
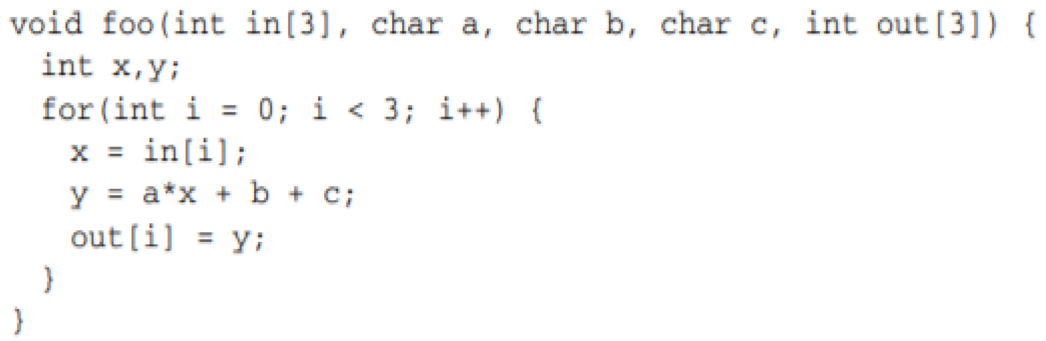


Fig.5- Scheduling and Binding example

Fig.6 performs the same operations as the previous example. However, it performs the operations inside a for-loop, and two of the function arguments are arrays.
The resulting design executes the logic inside the for-loop three times when the code is scheduled. High-level synthesis automatically extracts the control logic from the C code and creates an FSM in the RTL design to sequence these operations. High-level synthesis implements the top-level function arguments as ports in the final RTL design. The scalar variable of type char maps into a standard 8-bit data bus port. Array arguments, such as in and out, contain an entire collection of data.
In high-level synthesis, arrays are synthesized into block RAM by default, but other options are possible, such as FIFOs, distributed RAM, and individual registers. When using arrays as arguments in the top-level function, high-level synthesis assumes that the block RAM is outside the top-level function and automatically creates ports to access a block RAM outside the design, such as data ports, address ports, and any required chip-enable or write-enable signals.
The FSM controls when the registers store data and controls the state of any I/O control signals. The FSM starts in the state C0. On the next clock, it enters state C1, then state C2, and then state C3. It returns to state C1 (and C2, C3) a total of three times before returning to state C0.



```
void foo(int in[3], char a, char b, char c, int out[3]) {
  int x,y;
  for(int i = 0; i < 3; i++) {
    x = in[i];
    y = a*x + b + c;
    out[i] = y;
  }
}
```

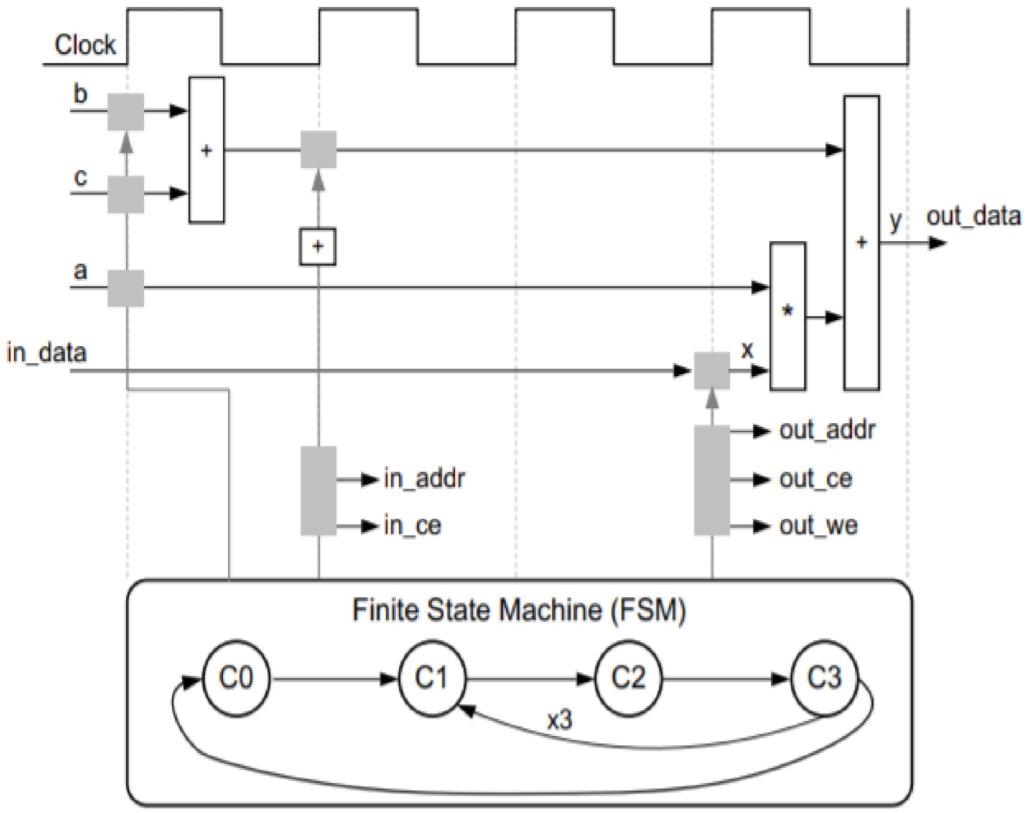


Fig.6- Control logic extraction

The design requires the addition of b and c only one time. High-level synthesis moves the operation outside the for-loop and into state C0. Each time the design enters state C3, it reuses the result of the addition.

The design reads the data from in and stores the data in x. The FSM generates the address for the first element in state C1. In addition, in state C1, an adder increments to keep track of how many times the design must iterate around states C1, C2, and C3. In state C2, the block RAM returns the data for in and stores it as variable x.

High-level synthesis reads the data from port a with other values to perform the calculation and generates the first y output. The FSM ensures that the correct address and control signals are generated to store this value outside the block. The design then returns to state C1 to read the next value from the array/block RAM in. This process continues until all output is written. The design then returns to state C0 to read the next values of b and c to start the process again.

Vivado HLS design flow:
1. Compile, execute (simulate), and debug the C algorithm.
2. Synthesize the C algorithm into an RTL implementation, optionally using user optimization directives.
3. Generate comprehensive reports and analyze the design.
4. Verify the RTL implementation using a pushbutton flow.
5. Package the RTL implementation into a selection of IP formats.

Inputs and Outputs-

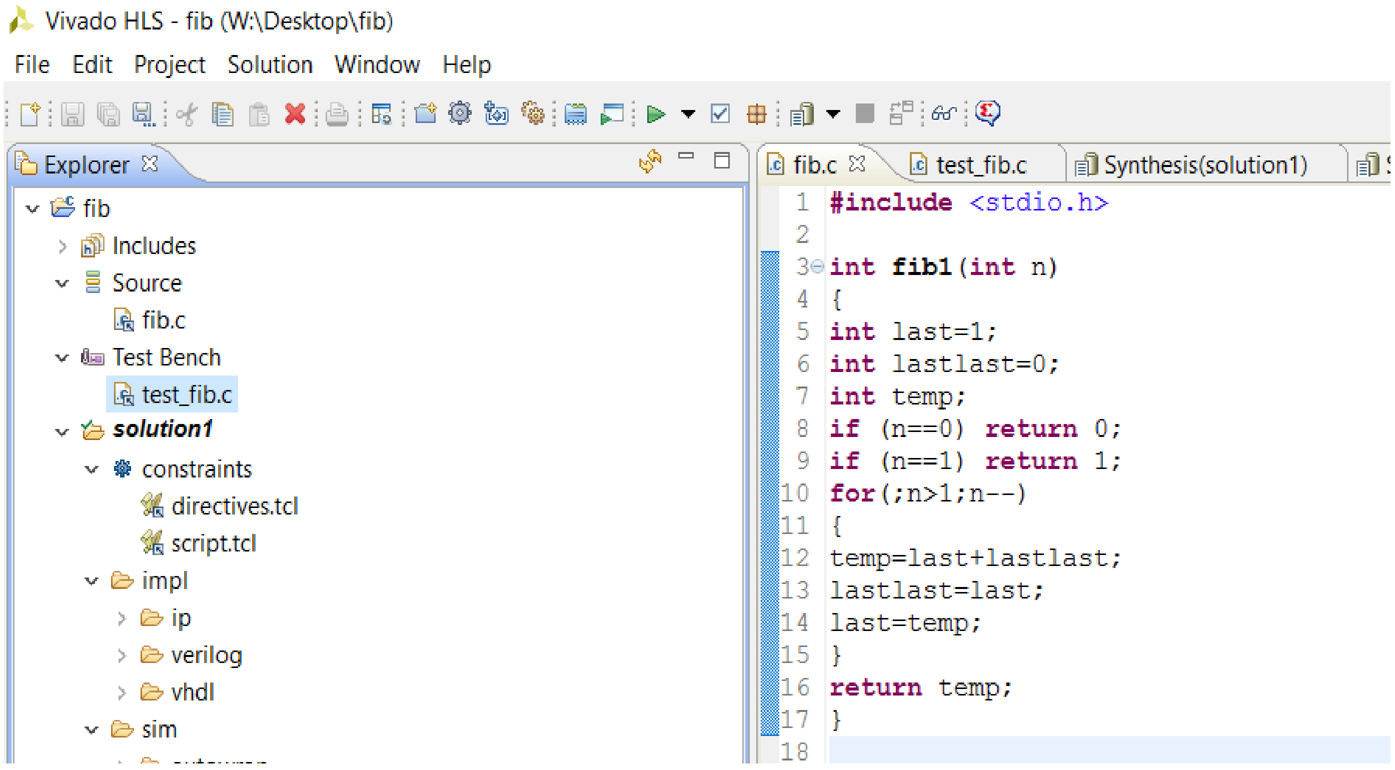
- Inputs to Vivado HLS-
  C function written in C, C++, or SystemC, Constraints, Directives, C test bench and any associated files.
- Outputs to Vivado HLS-
  RTL implementation files in hardware description language (HDL) formats and report files.

In any C program, the top-level function is called main(). In the Vivado HLS design flow, you can specify any sub-function below main() as the top-level function for synthesis. You cannot synthesize the top-level function main(). Following are additional rules:
• Only one function is allowed as the top-level function for synthesis.
• Any sub-functions in the hierarchy under the top-level function for synthesis are also synthesized.
• If you want to synthesize functions that are not in the hierarchy under the top-level function for synthesis, you must merge the functions into a single top-level function for synthesis.

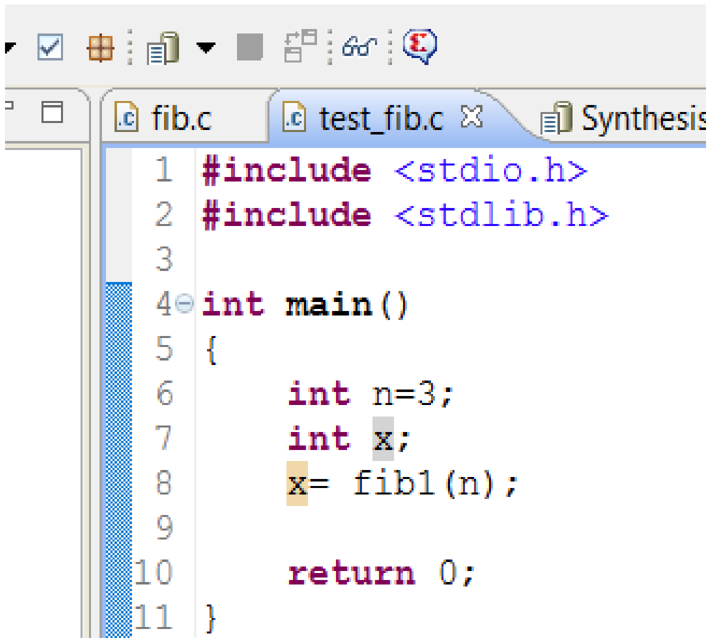
The C test bench includes the function main() and any sub-functions that are not in the hierarchy under the top-level function for synthesis. These functions verify that the top-level function for synthesis is functionally correct by providing stimuli to the function for synthesis and by consuming its output.

Vivado HLS uses the test bench to compile and execute the C simulation. During the compilation process, you can select the Launch Debugger option to open a full C-debug environment, which enables you to analyze the C simulation.

Chapter 3- Example based on Fibonacci Series

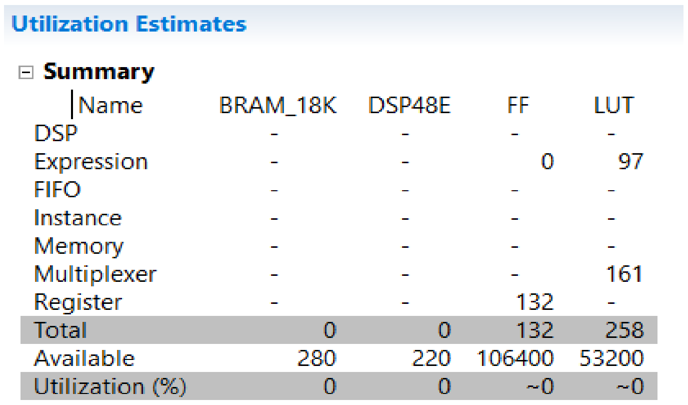


```c
#include <stdio.h>

int fib1(int n)
{
int last=1;
int lastlast=0;
int temp;
if (n==0) return 0;
if (n==1) return 1;
for(;n>1;n--)
{
temp=last+lastlast;
lastlast=last;
last=temp;
}
return temp;
}
```



```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
      int n=3;
      int x;
      x= fib1(n);

      return 0;
}
```

Fig.7- Top level function code and testbench code

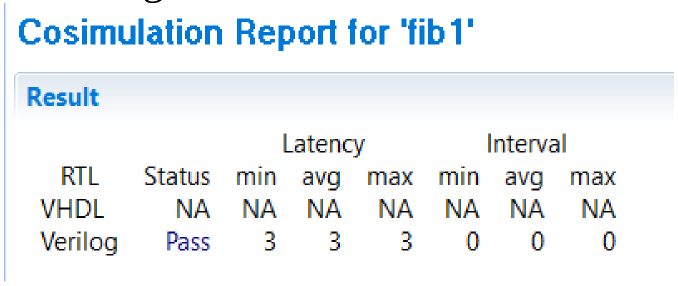


Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	97
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	161
Register	-	-	132	-
Total	0	0	132	258
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Fig.8- Utilization estimates

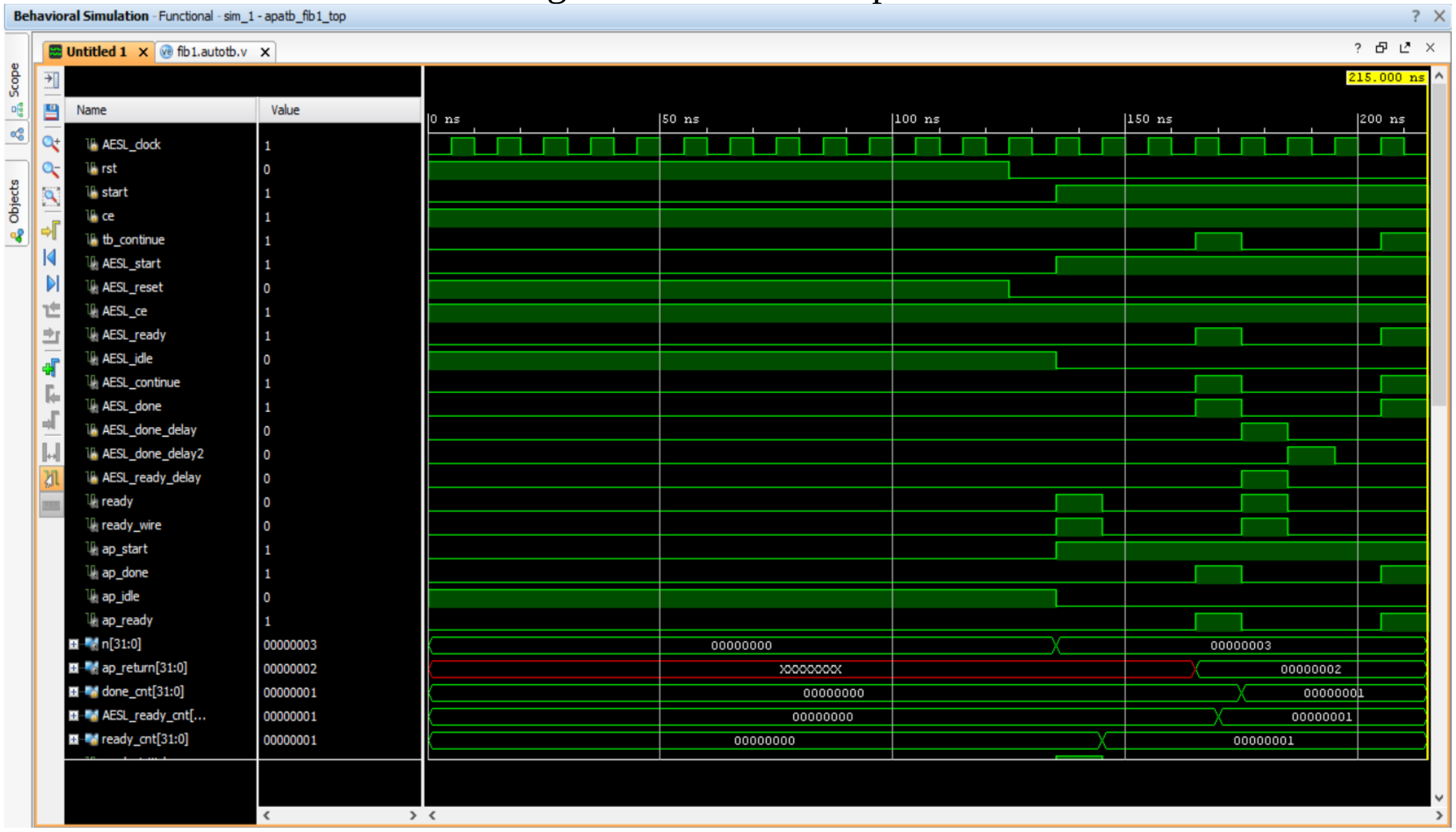


Cosimulation Report for 'fib1'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	3	3	3	0	0	0

Fig.9-Cosimulation report



Behavioral Simulation - Functional - sim_1 - apatb_fib1_top

Untitled 1 | fib1.autotb.v

Name	Value
AESL_clock	1
rst	0
start	1
ce	1
tb_continue	1
AESL_start	1
AESL_reset	0
AESL_ce	1
AESL_ready	1
AESL_idle	0
AESL_continue	1
AESL_done	1
AESL_done_delay	0
AESL_done_delay2	0
AESL_ready_delay	0
ready	0
ready_wire	0
ap_start	1
ap_done	1
ap_idle	0
ap_ready	1
n[31:0]	00000003
ap_return[31:0]	00000002
done_cnt[31:0]	00000001
AESL_ready_cnt[...	00000001
ready_cnt[31:0]	00000001

Fig.10-Simulation results

Chapter 4- Sobel filtering for edge detection

Edges in images are areas with strong intensity contrasts; a jump in intensity from one pixel to the next.
The process of edge detection significantly reduces the amount of data and filters out unneeded information, while preserving the important structural properties of an image.
There are many different edge detection methods, the majority of which can be grouped into two categories: Gradient and Laplacian.
The gradient method detects the edges by looking for the maximum and minimum in the first derivative of the image. The Laplacian method searches for zero crossings in the second derivative of the image .

Sobel Filtering falls under the category of gradient edge detection.
It works by calculating the gradient of image intensity at each pixel within the image. It finds the direction of the largest increase from light to dark and the rate of change in that direction.
The result shows how abruptly or smoothly the image changes at each pixel, and therefore how likely it is that that pixel represents an edge.
It also shows how that edge is likely to be oriented.
The result of applying the filter to a pixel in a region of constant intensity is a zero vector.
The result of applying it to a pixel on an edge is a vector that points across the edge from darker to brighter values.
The sobel operator uses two 3×3 kernels which are convolved with the original image to calculate approximations of the derivatives – one for horizontal changes, and one for vertical.
If we define Gx and Gy as two images that contain the horizontal and vertical derivative approximations respectively, the computations are:

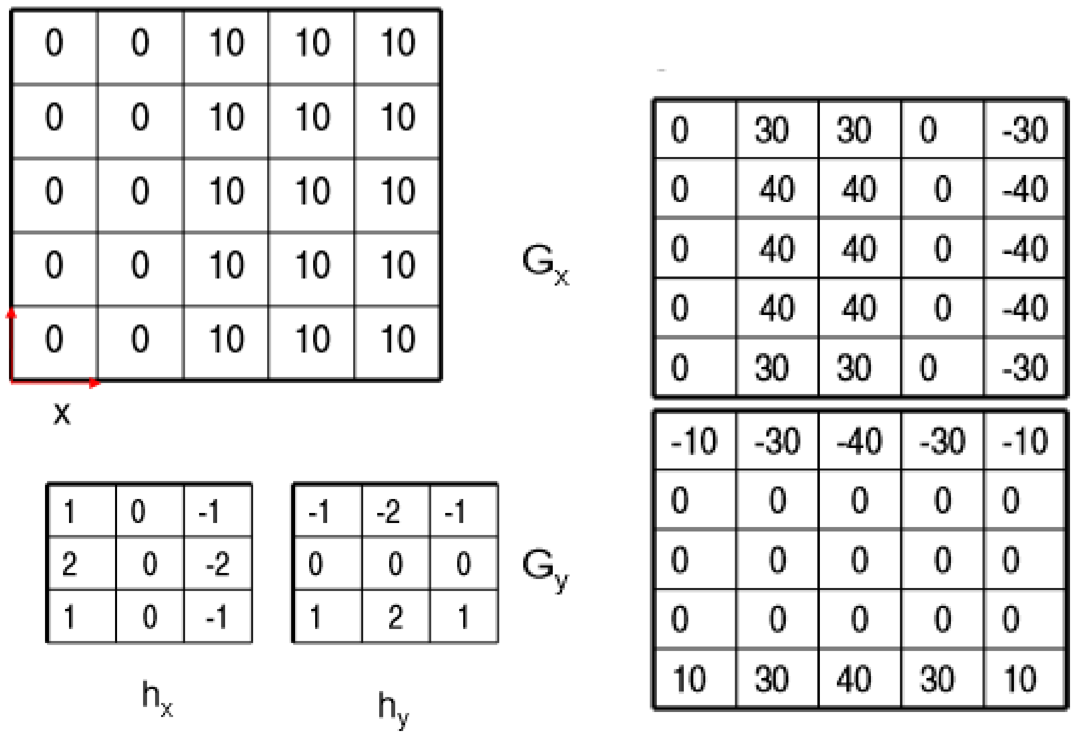
$$G_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} * A \quad \text{and} \quad G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} * A$$

where A is the original source image.

We move the appropriate kernel (window) over the input image, computing the value for one pixel and then shifting one pixel to the right. Once the end of the row is reached, we move down to the beginning of the next row.
At each pixel in the image, the gradient approximations given by Gx and Gy are combined to give the gradient magnitude, using:

$$G = \sqrt{G_x^2 + G_y^2}$$

We Compute Gx and Gy, gradients of the image performing the convolution of Sobel kernels with the image and use zero-padding to extend the image. After that linear transformation is required to transform the gradient values in the range of 0-255.



0	0	10	10	10
0	0	10	10	10
0	0	10	10	10
0	0	10	10	10
0	0	10	10	10

1	0	-1
2	0	-2
1	0	-1

-1	-2	-1
0	0	0
1	2	1

0	30	30	0	-30
0	40	40	0	-40
0	40	40	0	-40
0	40	40	0	-40
0	30	30	0	-30

-10	-30	-40	-30	-10
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
10	30	40	30	10

Fig.11- Example of sobel filtering

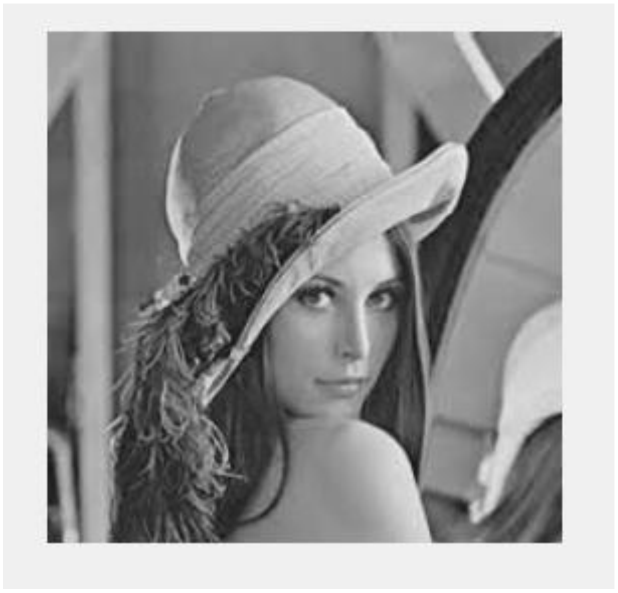


Fig.12- Input image

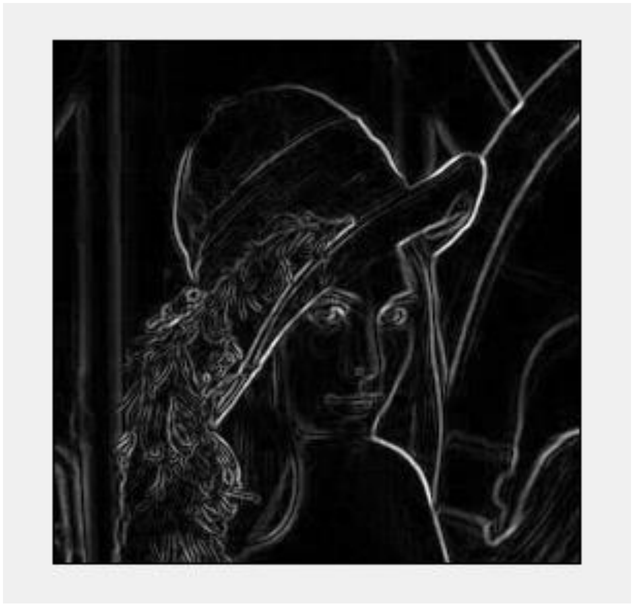
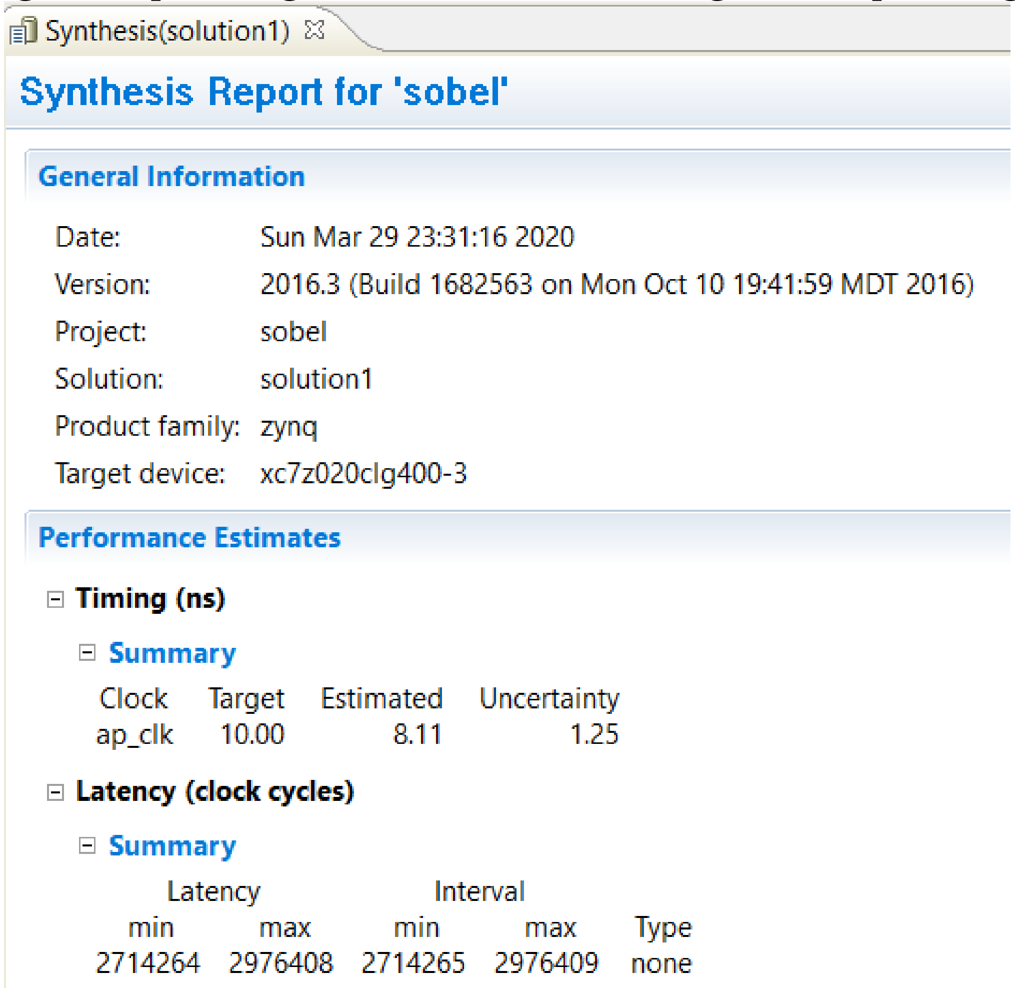


Fig.13- Output image



Synthesis(solution1)

Synthesis Report for 'sobel'

General Information

Date:	Sun Mar 29 23:31:16 2020
Version:	2016.3 (Build 1682563 on Mon Oct 10 19:41:59 MDT 2016)
Project:	sobel
Solution:	solution1
Product family:	zynq
Target device:	xc7z020clg400-3

Performance Estimates

Timing (ns)

Summary

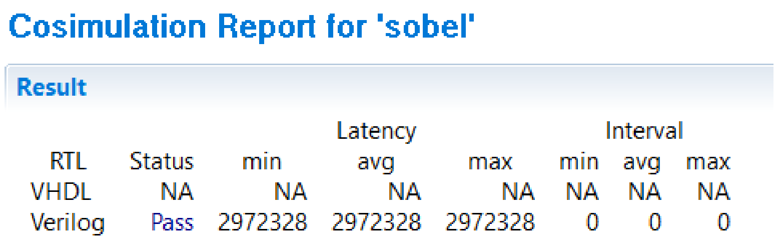
Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.11	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
2714264	2976408	2714265	2976409	none

Fig.14- Synthesis report

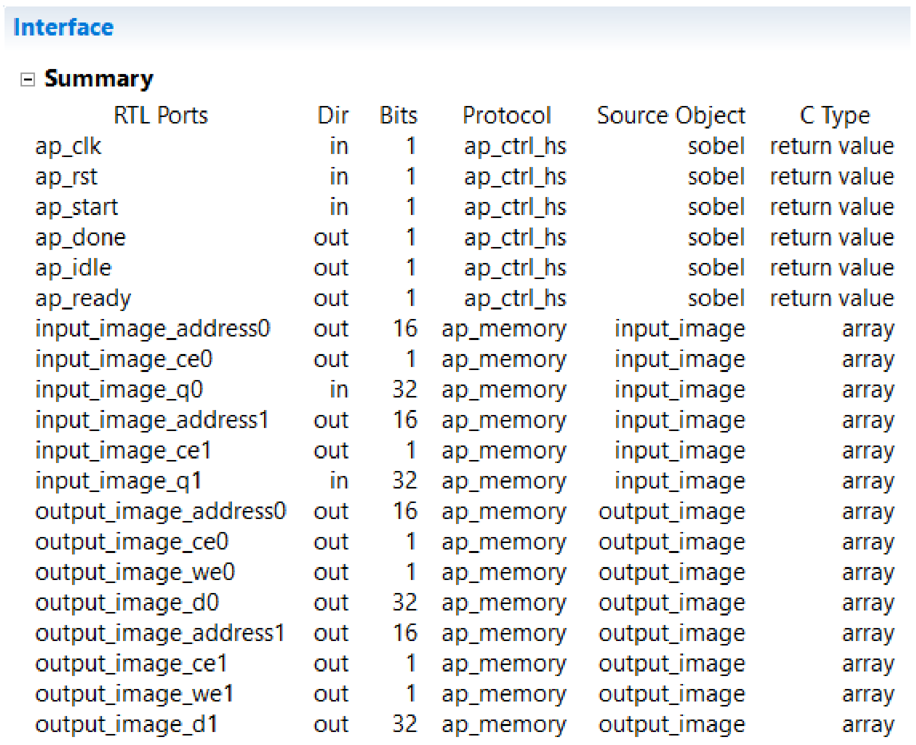


Cosimulation Report for 'sobel'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	2972328	2972328	2972328	0	0	0

Fig.15-Cosimulation report

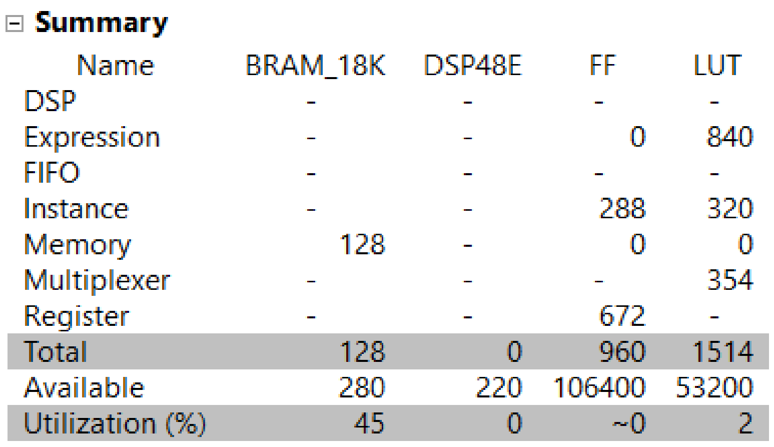


Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	sobel	return value
ap_rst	in	1	ap_ctrl_hs	sobel	return value
ap_start	in	1	ap_ctrl_hs	sobel	return value
ap_done	out	1	ap_ctrl_hs	sobel	return value
ap_idle	out	1	ap_ctrl_hs	sobel	return value
ap_ready	out	1	ap_ctrl_hs	sobel	return value
input_image_address0	out	16	ap_memory	input_image	array
input_image_ce0	out	1	ap_memory	input_image	array
input_image_q0	in	32	ap_memory	input_image	array
input_image_address1	out	16	ap_memory	input_image	array
input_image_ce1	out	1	ap_memory	input_image	array
input_image_q1	in	32	ap_memory	input_image	array
output_image_address0	out	16	ap_memory	output_image	array
output_image_ce0	out	1	ap_memory	output_image	array
output_image_we0	out	1	ap_memory	output_image	array
output_image_d0	out	32	ap_memory	output_image	array
output_image_address1	out	16	ap_memory	output_image	array
output_image_ce1	out	1	ap_memory	output_image	array
output_image_we1	out	1	ap_memory	output_image	array
output_image_d1	out	32	ap_memory	output_image	array

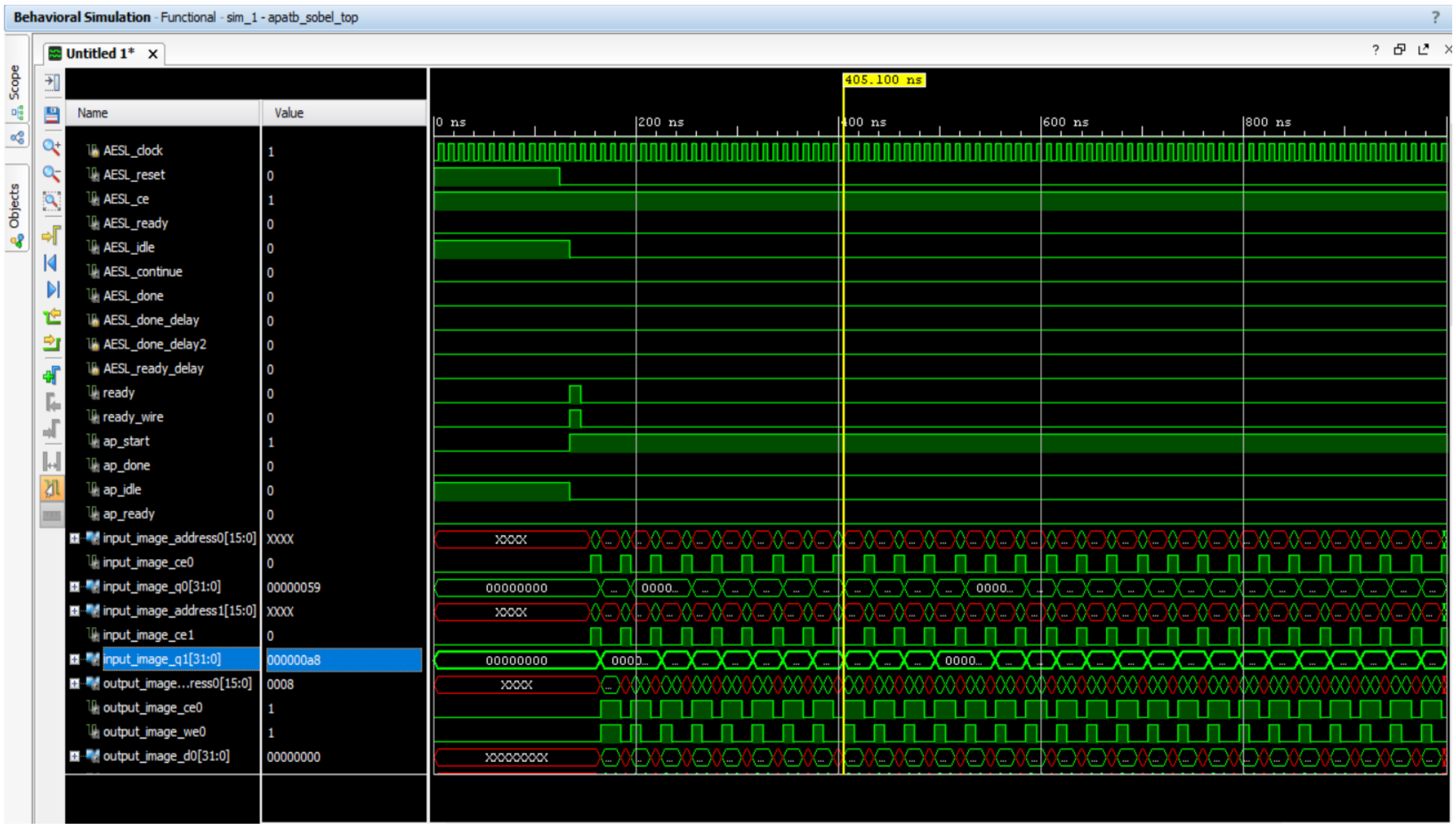
Fig.16- Interfaces



Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	840
FIFO	-	-	-	-
Instance	-	-	288	320
Memory	128	-	0	0
Multiplexer	-	-	-	354
Register	-	-	672	-
Total	128	0	960	1514
Available	280	220	106400	53200
Utilization (%)	45	0	~0	2

Fig.17- Utilization estimates



Name	Value
AESL_clock	1
AESL_reset	0
AESL_ce	1
AESL_ready	0
AESL_idle	0
AESL_continue	0
AESL_done	0
AESL_done_delay	0
AESL_done_delay2	0
AESL_ready_delay	0
ready	0
ready_wire	0
ap_start	1
ap_done	0
ap_idle	0
ap_ready	0
input_image_address0[15:0]	XXXX
input_image_ce0	0
input_image_q0[31:0]	00000059
input_image_address1[15:0]	XXXX
input_image_ce1	0
input_image_q1[31:0]	000000a8
output_image…ress0[15:0]	0008
output_image_ce0	1
output_image_we0	1
output_image_d0[31:0]	00000000

Fig.18- Simulation results

Chapter 5- K Means clustering for image segmentation

Image segmentation is the division or separation of an image into regions i.e. set of pixels, pixels in a region are similar according to some criterion such as colour, intensity or texture.
Image segmentation methods fall into different categories: Region based segmentation, Edge based segmentation, and Clustering based segmentation, Thresholding, Artificial neural network, feature-based segmentation.
The clustering algorithm aim is to develop the partitioning decisions based on initial set of clusters that is updated after each iteration.

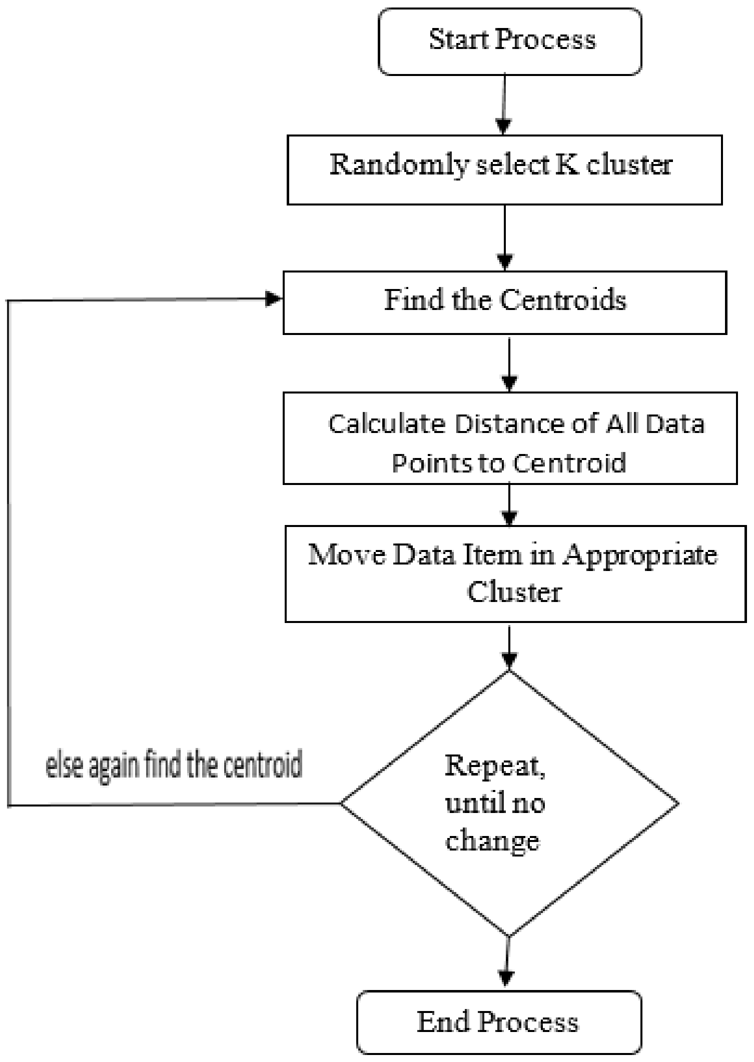

Start Process
Randomly select K cluster
Find the Centroids
Calculate Distance of All Data Points to Centroid
Move Data Item in Appropriate Cluster
Repeat, until no change
else again find the centroid
End Process

Fig.19- K means clustering algorithm

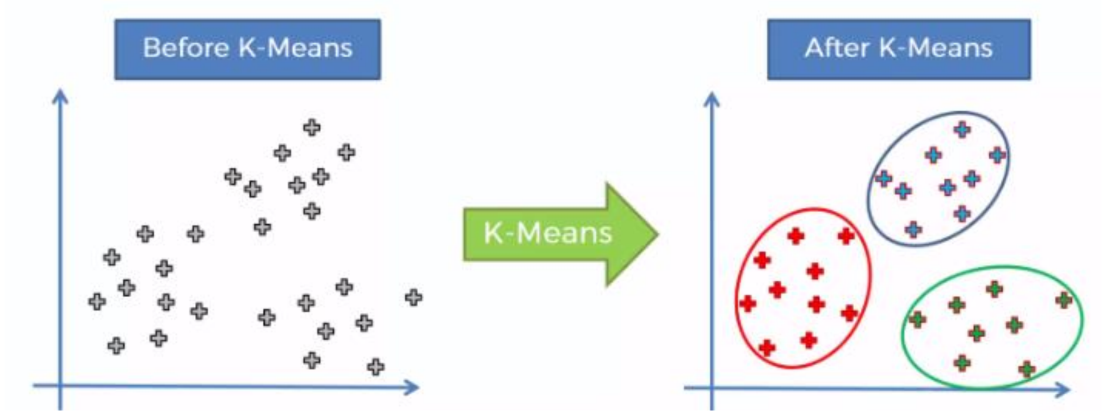


Fig.20- Example of K means clustering

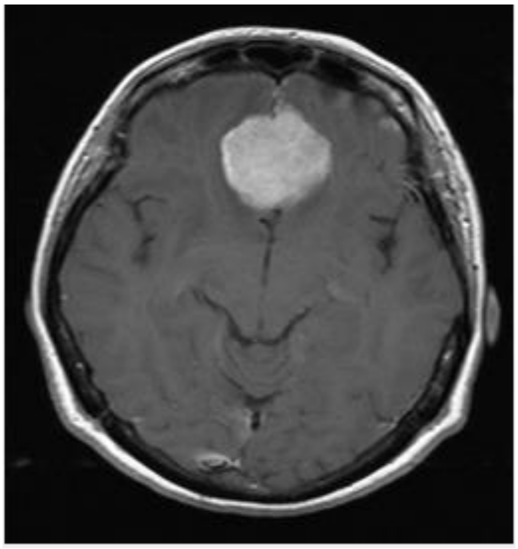


Fig.21- Input

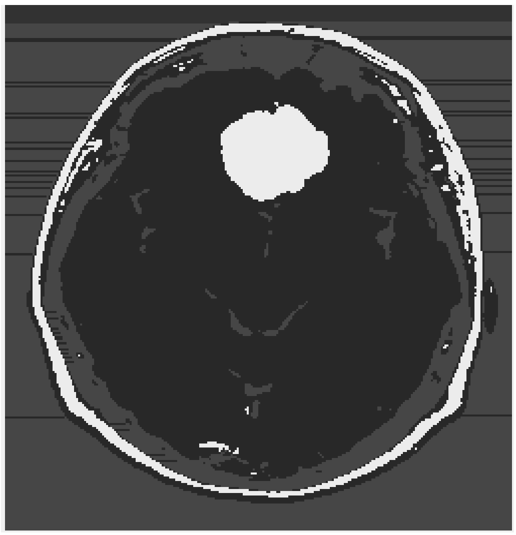
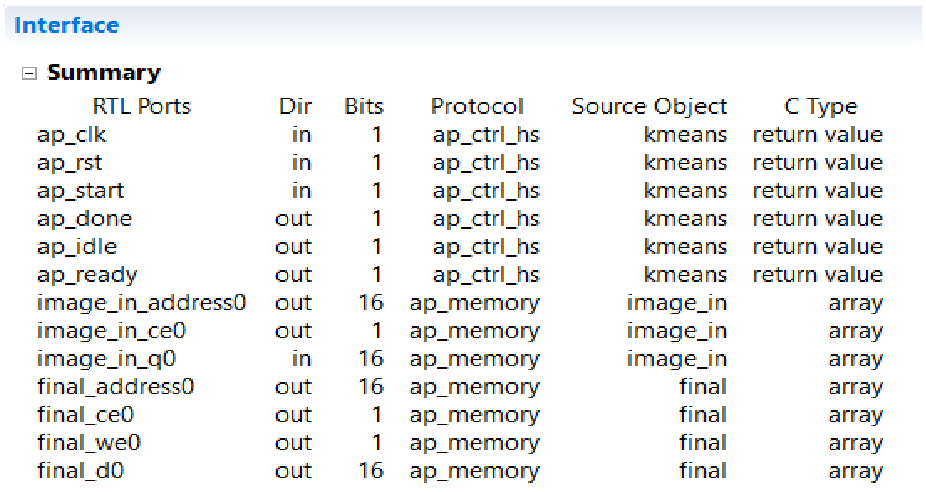


Fig.22- Output

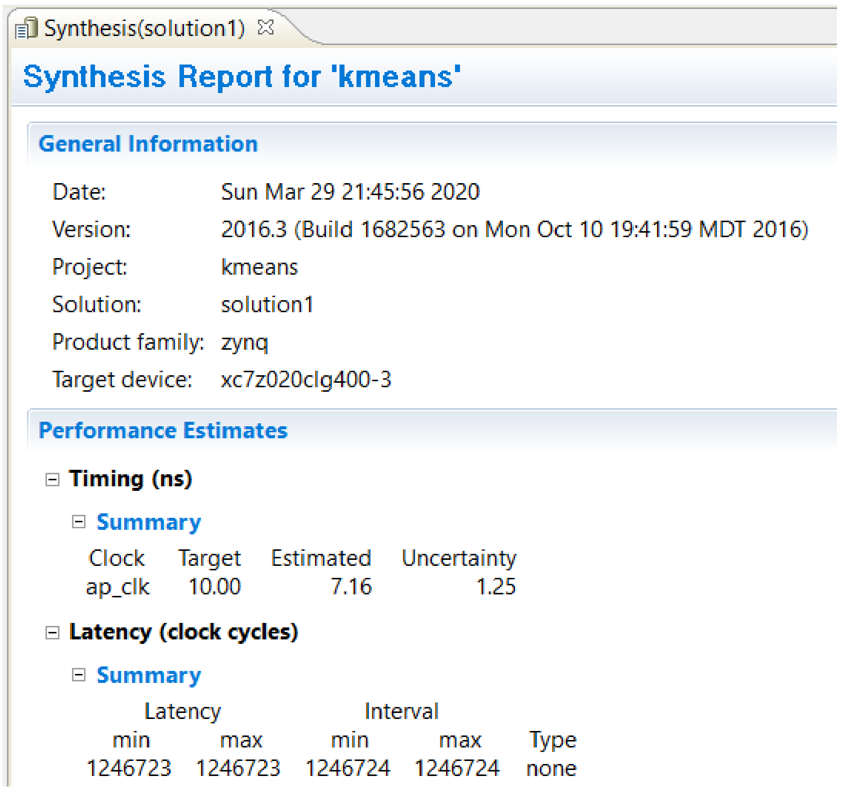


Interface

⊟ **Summary**

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	kmeans	return value
ap_rst	in	1	ap_ctrl_hs	kmeans	return value
ap_start	in	1	ap_ctrl_hs	kmeans	return value
ap_done	out	1	ap_ctrl_hs	kmeans	return value
ap_idle	out	1	ap_ctrl_hs	kmeans	return value
ap_ready	out	1	ap_ctrl_hs	kmeans	return value
image_in_address0	out	16	ap_memory	image_in	array
image_in_ce0	out	1	ap_memory	image_in	array
image_in_q0	in	16	ap_memory	image_in	array
final_address0	out	16	ap_memory	final	array
final_ce0	out	1	ap_memory	final	array
final_we0	out	1	ap_memory	final	array
final_d0	out	16	ap_memory	final	array

Fig.23- Interfaces



Synthesis(solution1)

Synthesis Report for 'kmeans'

General Information

Date:	Sun Mar 29 21:45:56 2020
Version:	2016.3 (Build 1682563 on Mon Oct 10 19:41:59 MDT 2016)
Project:	kmeans
Solution:	solution1
Product family:	zynq
Target device:	xc7z020clg400-3

Performance Estimates

Timing (ns)

Summary

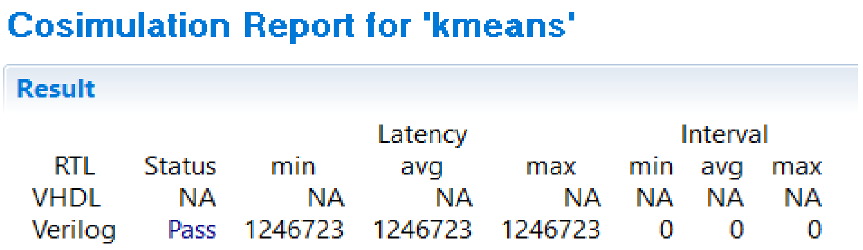
Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.16	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1246723	1246723	1246724	1246724	none

Fig.24- Synthesis report

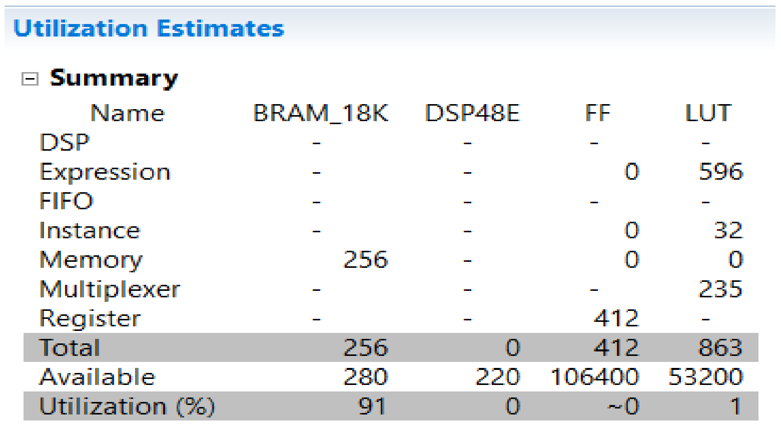


Cosimulation Report for 'kmeans'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	1246723	1246723	1246723	0	0	0

Fig.25- Cosimulation report



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	596
FIFO	-	-	-	-
Instance	-	-	0	32
Memory	256	-	0	0
Multiplexer	-	-	-	235
Register	-	-	412	-
Total	256	0	412	863
Available	280	220	106400	53200
Utilization (%)	91	0	~0	1

Fig.26- Utilization estimates

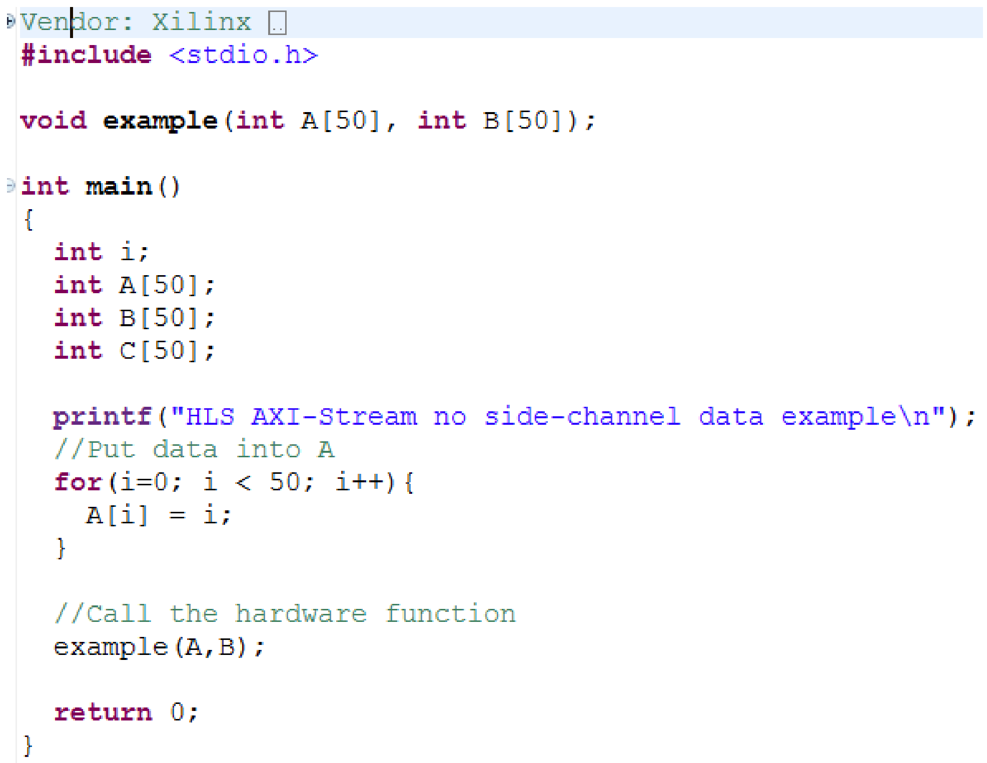
Chapter 6- Questions

1) How parameterizable a design/module is (e.g., can we specify number of bits or type of function as a parameter on sub-module instantiation?)

Yes we can specify the number of bits and type of function as a parameter on sub-module instantiation.

2)How much support is there for hardware/software interactions? None; one way (software calling hardware accelerators): two-way (hardware calling software functions as well)?

One way support is there for hardware/software interactions. Software functions can call hardware functions but hardware functions cannot call software functions, it gives an error when trying to call software function from hardware one.



```
Vendor: Xilinx
#include <stdio.h>

void example(int A[50], int B[50]);

int main()
{
  int i;
  int A[50];
  int B[50];
  int C[50];

  printf("HLS AXI-Stream no side-channel data example\n");
  //Put data into A
  for(i=0; i < 50; i++){
    A[i] = i;
  }

  //Call the hardware function
  example(A,B);

  return 0;
}
```

Fig.27- Software function calling hardware synthesizable function

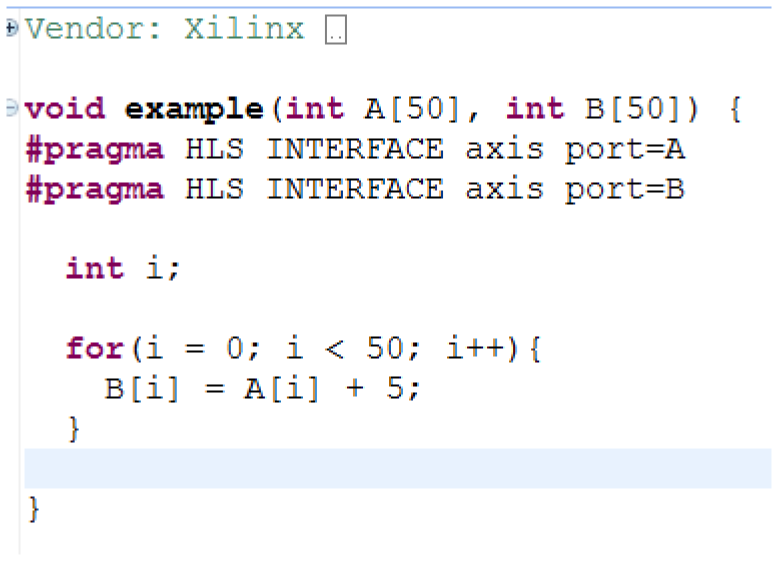

```
Vendor: Xilinx

void example(int A[50], int B[50]) {
#pragma HLS INTERFACE axis port=A
#pragma HLS INTERFACE axis port=B

  int i;

  for(i = 0; i < 50; i++){
    B[i] = A[i] + 5;
  }

}
```

Fig.28- Hardware function being called

3)Can it infer (implicitly or explicitly) tri-state buffers?

C code cannot infer tri-state buffers. It cannot be used to define high impedance states.

4)How much native support is there for interfacing with off-chip systems (e.g., external DRAM)?

We can interface external RAM using Vivado HLS.

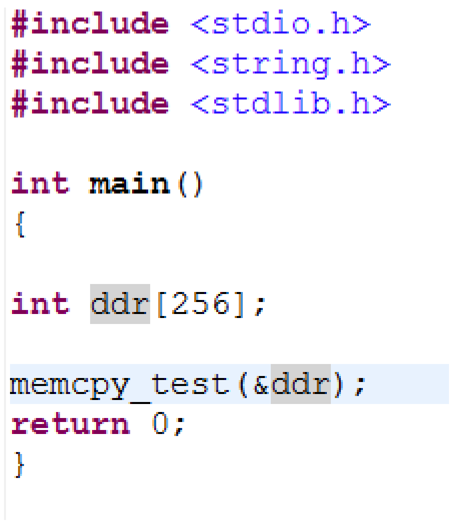
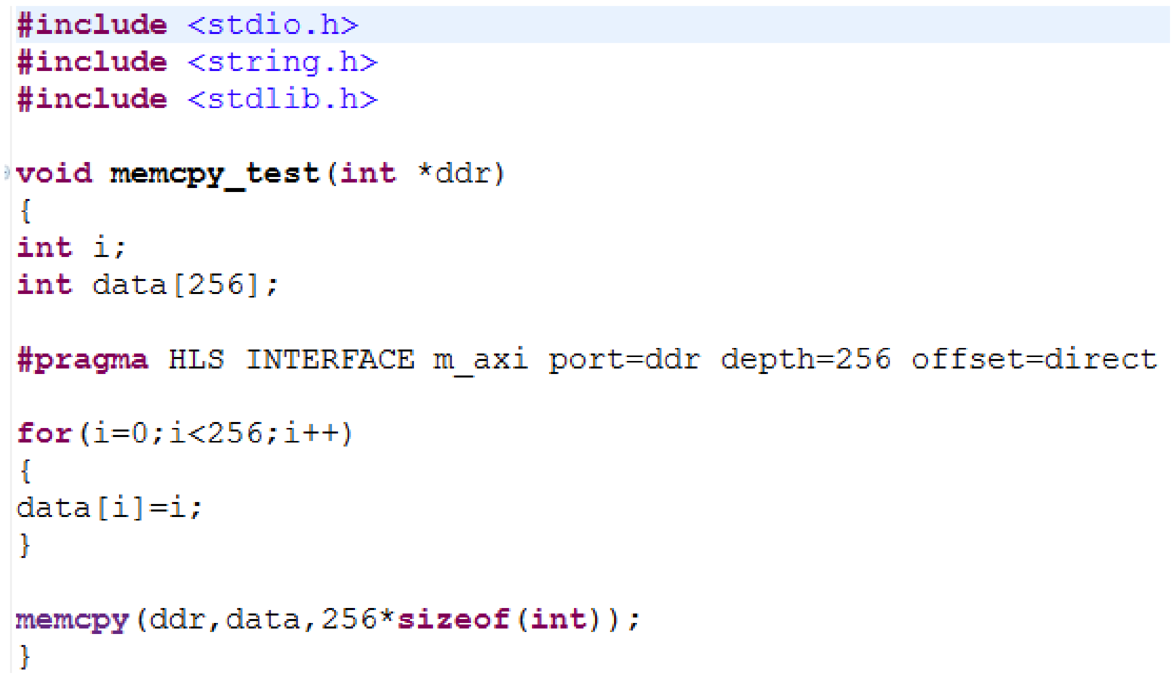
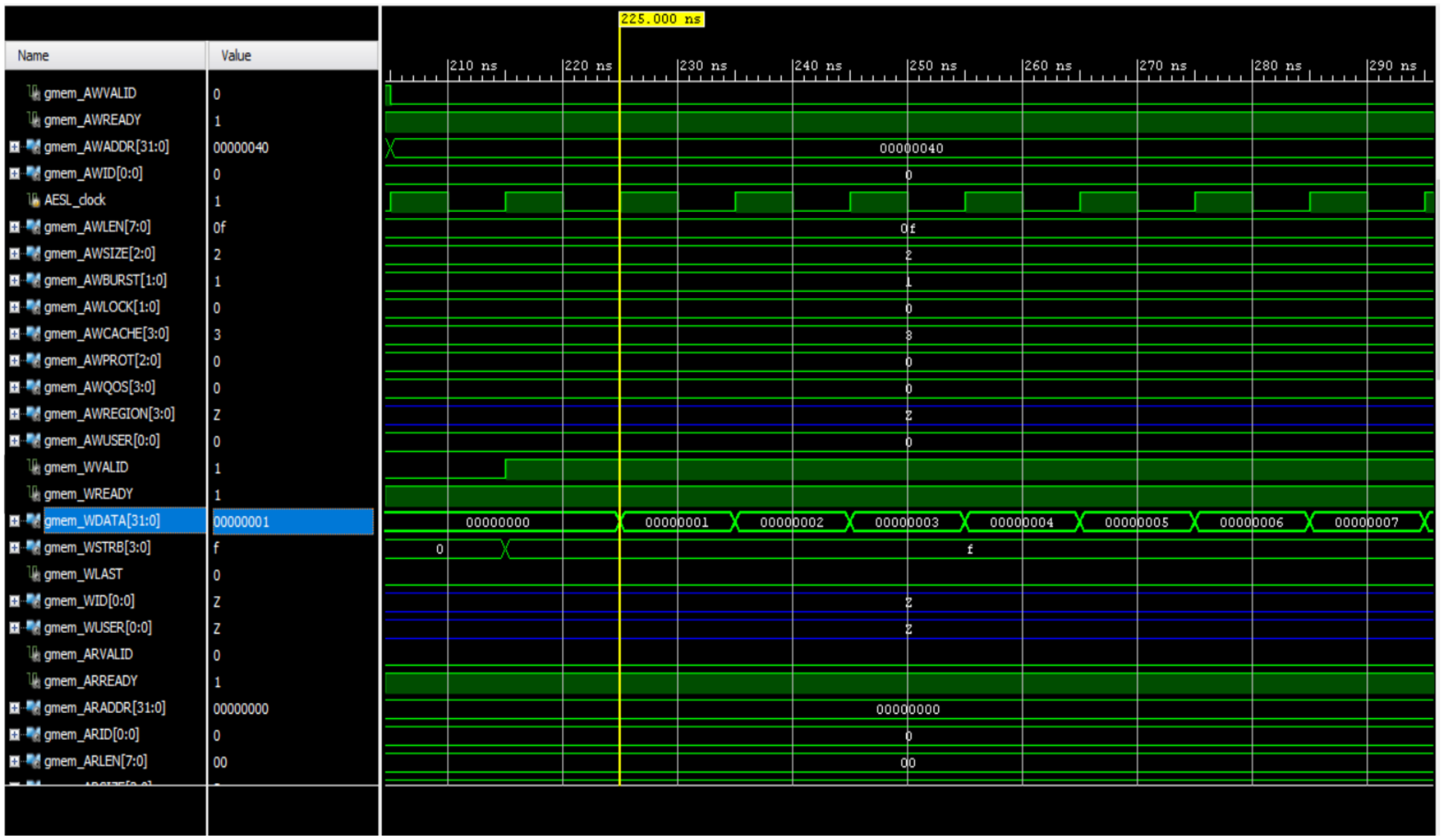

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main()
{

int ddr[256];

memcpy_test(&ddr);
return 0;
}
```

Fig.29- testbench code



```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void memcpy_test(int *ddr)
{
int i;
int data[256];

#pragma HLS INTERFACE m_axi port=ddr depth=256 offset=direct

for(i=0;i<256;i++)
{
data[i]=i;
}

memcpy(ddr,data,256*sizeof(int));
}
```

Fig.30- C code to interface external DDR RAM



Name	Value
gmem_AWVALID	0
gmem_AWREADY	1
gmem_AWADDR[31:0]	00000040
gmem_AWID[0:0]	0
AESL_clock	1
gmem_AWLEN[7:0]	0f
gmem_AWSIZE[2:0]	2
gmem_AWBURST[1:0]	1
gmem_AWLOCK[1:0]	0
gmem_AWCACHE[3:0]	3
gmem_AWPROT[2:0]	0
gmem_AWQOS[3:0]	0
gmem_AWREGION[3:0]	Z
gmem_AWUSER[0:0]	0
gmem_WVALID	1
gmem_WREADY	1
gmem_WDATA[31:0]	00000001
gmem_WSTRB[3:0]	f
gmem_WLAST	0
gmem_WID[0:0]	Z
gmem_WUSER[0:0]	Z
gmem_ARVALID	0
gmem_ARREADY	1
gmem_ARADDR[31:0]	00000000
gmem_ARID[0:0]	0
gmem_ARLEN[7:0]	00

Fig.31- Simulation result

5) How much support is there to implement interface protocols (e.g., can we create a system that interfaces with the outside world through a protocol that requires an enable signal to be high for one clock cycle, then an address burst followed by a read/write request?)

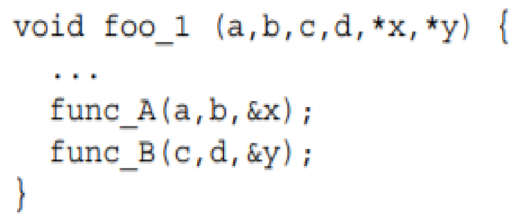
Yes we can implement interface protocols using Vivado HLS and c code. There are AXI interfaces available like AXI master, AXI slave and AXI stream protocols available on Vivado HLS.

6) How many pre-defined (default) interfaces does it support?

One clock per C design(ap_clk), one reset(ap_rst), one clock enable(ap_ce), one function return(ap_return), ap_start, ap_idle, ap_done and ap_ready are the default or pre-defined interfaces. After that depending on the design it can have data ports, pointers, FIFO ports, RAM ports or AXI interface ports.

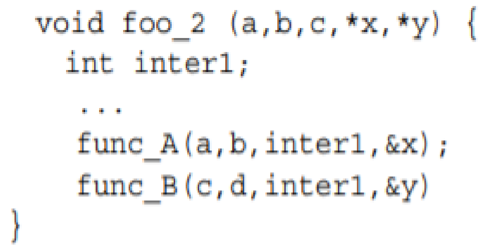
7) How much automatic parallelization is performed (versus manual parallelization)?

The following examples shows two functions, foo_1 and foo_2.



```
void foo_1 (a,b,c,d,*x,*y) {
  ...
  func_A(a,b,&x);
  func_B(c,d,&y);
}
```

Fig.32- Function foo_1



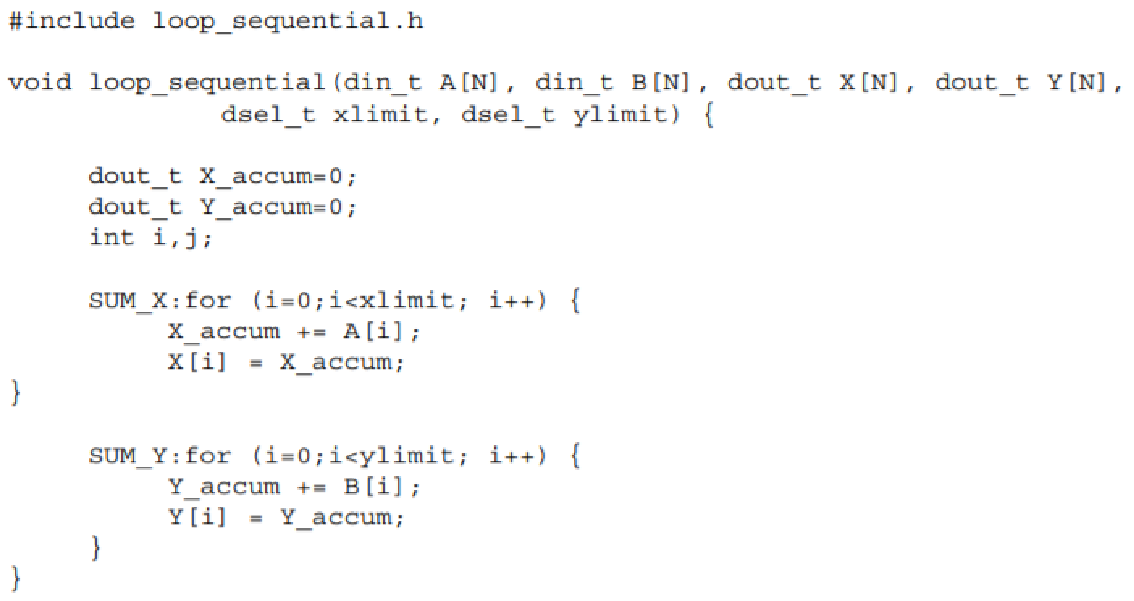
```
void foo_2 (a,b,c,*x,*y) {
  int inter1;
   ...
   func_A(a,b,inter1,&x);
   func_B(c,d,inter1,&y)
}
```

Fig.33- Function foo_2

In function foo_1, there is no data dependency between functions func_A and func_B. Even though they appear serially in the code, Vivado HLS will implement an architecture where both functions start to process data at the same time in the first clock cycle.

In function foo_2, there is a data dependency between the functions. Internal variable inter1 is passed from func_A to func_B. In this case, Vivado HLS must schedule function func_B to start only after function func_A is finished.

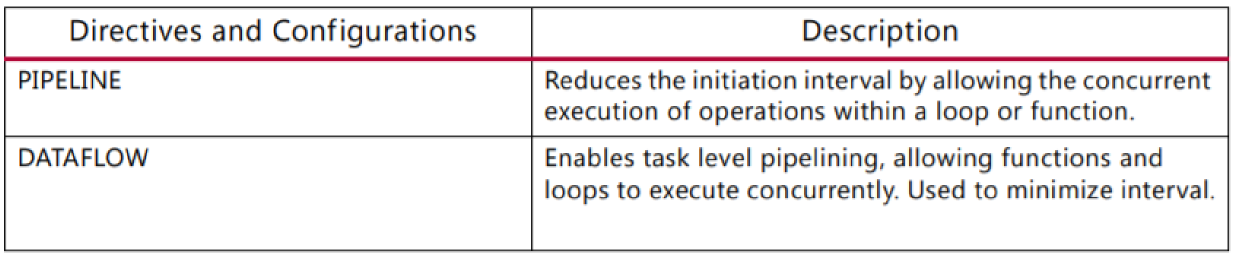
Loops are always scheduled to execute in order. In the following example, there is no dependency between loop SUM_X and SUM_Y, however they will always be scheduled in the order they appear in the code.



```
#include loop_sequential.h

void loop_sequential(din_t A[N], din_t B[N], dout_t X[N], dout_t Y[N],
              dsel_t xlimit, dsel_t ylimit) {

     dout_t X_accum=0;
     dout_t Y_accum=0;
     int i,j;

     SUM_X:for (i=0;i<xlimit; i++) {
          X_accum += A[i];
          X[i] = X_accum;
}

     SUM_Y:for (i=0;i<ylimit; i++) {
          Y_accum += B[i];
          Y[i] = Y_accum;
     }
}
```

Fig.34- Sequential loops

Loops by default are left “rolled”. This means Vivado HLS synthesizes the logic in the loop body once and then executes this logic serially until the loop termination limit is reached.

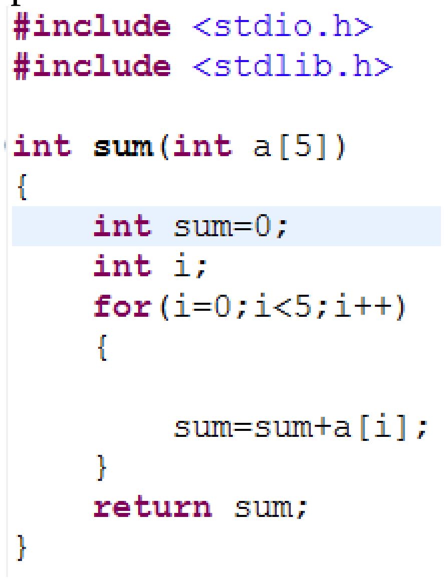


Directives and Configurations	Description
PIPELINE	Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.
DATAFLOW	Enables task level pipelining, allowing functions and loops to execute concurrently. Used to minimize interval.

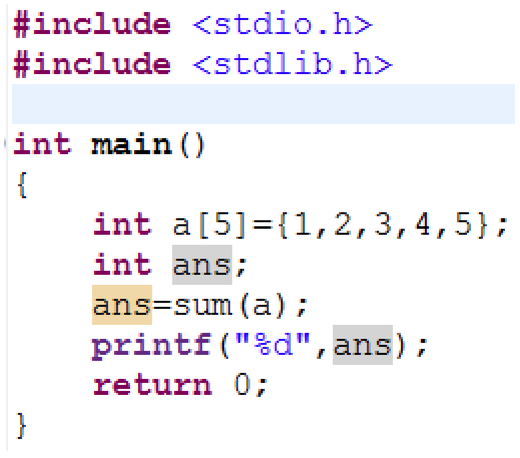
Fig.35- Parallelization using pipelining and dataflow

The PIPELINE directive can be applied to functions and loops. The DATAFLOW directive is used at the level containing the functions and loops to make them work in parallel.

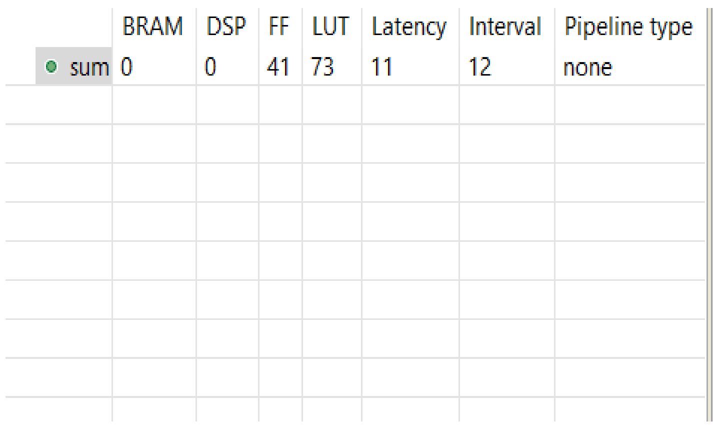
Example code-



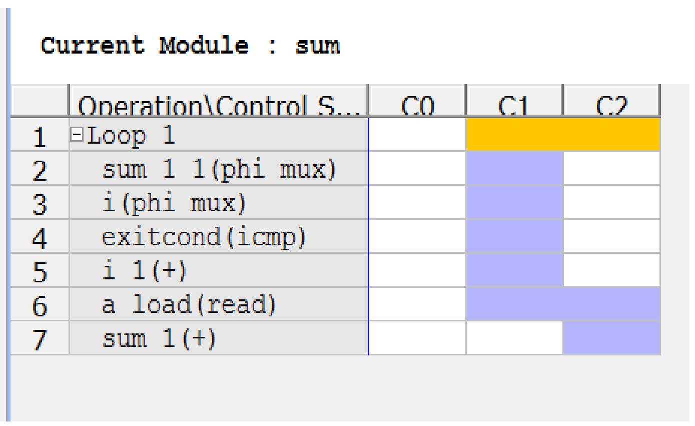
```
#include <stdio.h>
#include <stdlib.h>

int sum(int a[5])
{
    int sum=0;
    int i;
    for(i=0;i<5;i++)
    {

        sum=sum+a[i];
    }
    return sum;
}
```



```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int a[5]={1,2,3,4,5};
    int ans;
    ans=sum(a);
    printf("%d",ans);
    return 0;
}
```



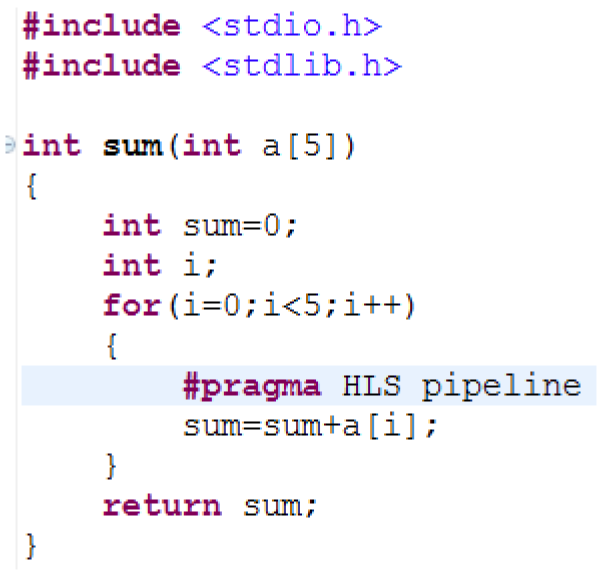
	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
● sum	0	0	41	73	11	12	none



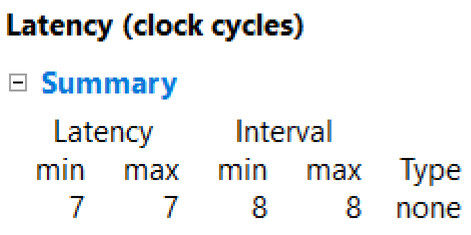
Current Module : sum

	Operation\Control S...	C0	C1	C2
1	⊟Loop 1		🟧	🟧
2	sum 1 1(phi mux)		🟪	
3	i(phi mux)		🟪	
4	exitcond(icmp)		🟪	
5	i 1(+)		🟪	
6	a load(read)		🟪	🟪
7	sum 1(+)			🟪

Fig.36- Without pipelining



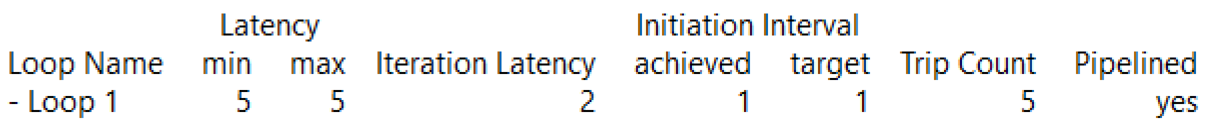
```
#include <stdio.h>
#include <stdlib.h>

int sum(int a[5])
{
    int sum=0;
    int i;
    for(i=0;i<5;i++)
    {
        #pragma HLS pipeline
        sum=sum+a[i];
    }
    return sum;
}
```



Latency (clock cycles)

⊟ **Summary**

Latency		Interval		
min	max	min	max	Type
7	7	8	8	none

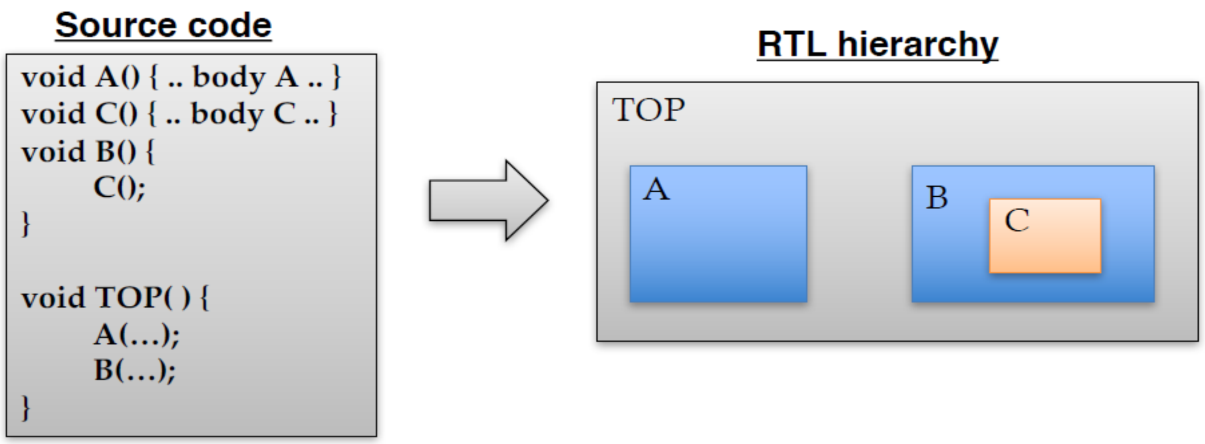


	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	5	5	2	1	1	5	yes

Fig.37- With pipelining

8)How well can it interface with Verilog (e.g., can we instantiate sub-modules designed in Verilog?)

Yes, we can instantiate sub-modules. We can create a top level function for synthesis which will be the top module and inside it call different functions which ultimately will become sub modules.



```
void A() { .. body A .. }
void C() { .. body C .. }
void B() {
        C();
}

void TOP( ) {
        A(...);
        B(...);
}
```

Fig.38- Function hierarchy

Chapter 7- Conclusion

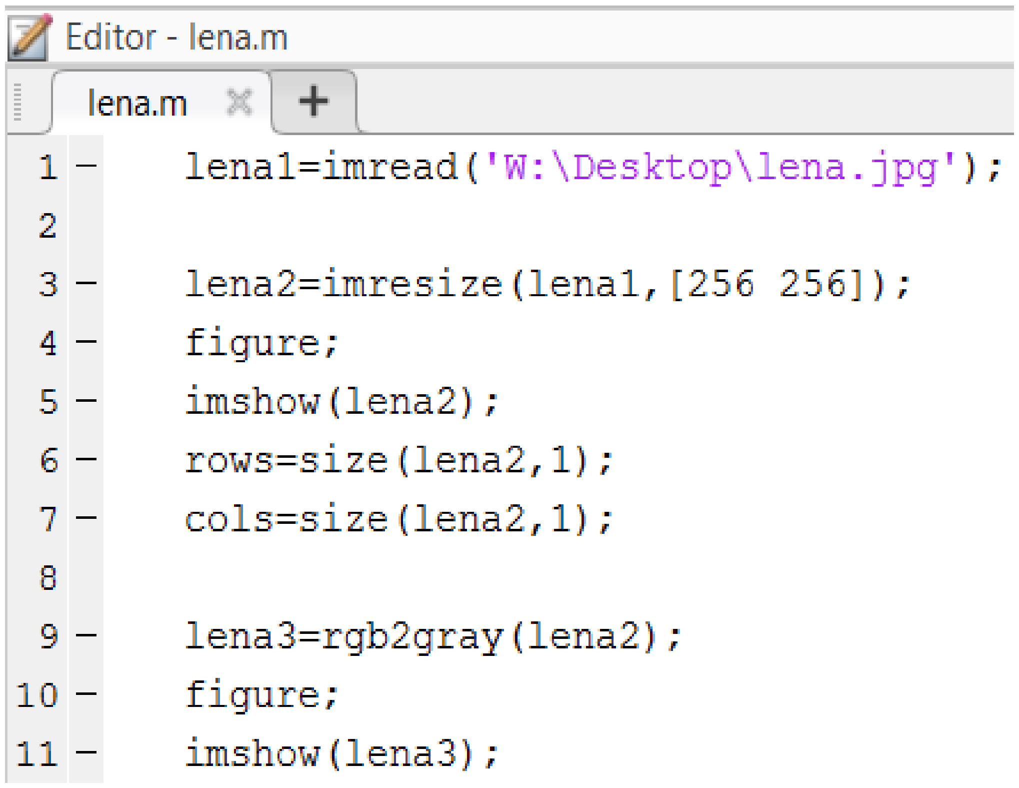
In this project, I implemented Sobel edge detection and K Means filtering using C programming language and Vivado HLS. I learnt about various language and Vivado HLS features while completing the final project.

For both image processing methods, I was able to generate results as expected. I observed the latency, simulation results, synthesis report, co-simulation report, utilization estimates , the default ports assigned and the other interfaces relevant to the method. I observed that by default there is no pipelining, but we can improve the interval estimate and latency estimate using pragmas for pipelining and dataflow.

I observed that there is one-way support which means the software function can call hardware functions but not reverse. We cannot infer tri-state buffers using C language. It is possible to interface external memory using AXI ports and using these ports we can establish connection between slave and master devices. I learnt about the parallelization of loops and functions defined.

Chapter 8- Appendix

1) MATLAB code for conversion of image from RGB to Gray-



```
lena1=imread('W:\Desktop\lena.jpg');

lena2=imresize(lena1,[256 256]);
figure;
imshow(lena2);
rows=size(lena2,1);
cols=size(lena2,1);

lena3=rgb2gray(lena2);
figure;
imshow(lena3);
```

Fig.39- MATLAB code

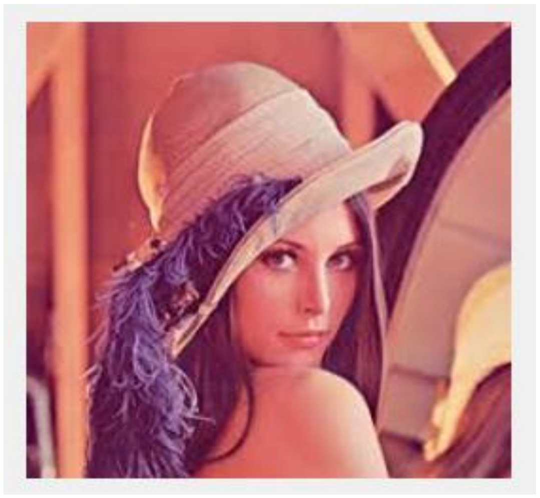


Fig.40- Input color image

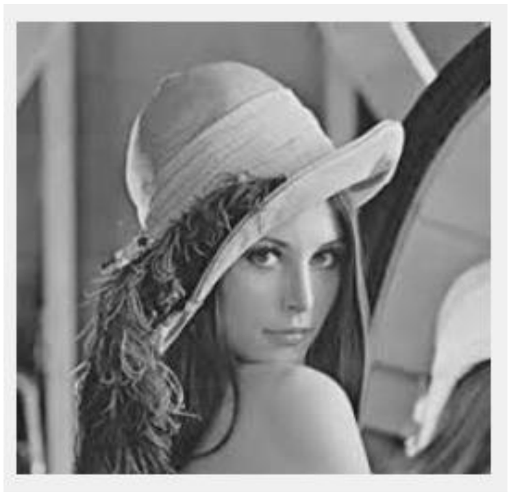
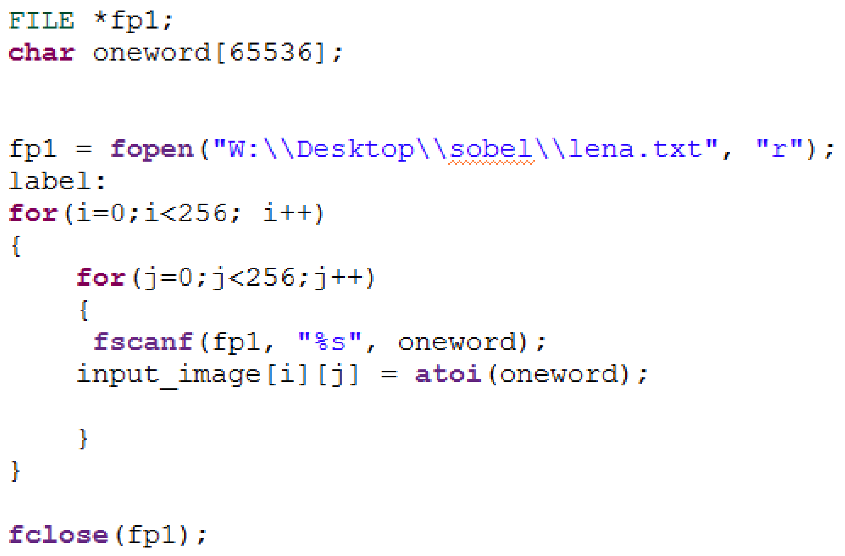


Fig.41- Output gray image

To write the pixel value to a file just copy the lena3 array to a file.

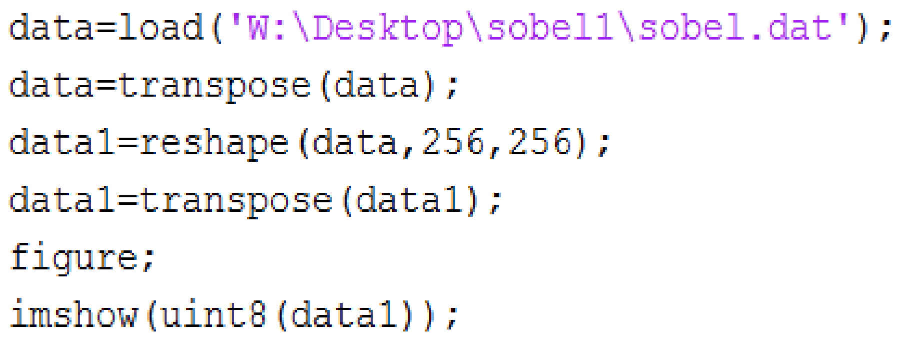
2) Copying the pixel values from a file to a matrix using c



```
FILE *fp1;
char oneword[65536];


fp1 = fopen("W:\\Desktop\\sobel\\lena.txt", "r");
label:
for(i=0;i<256; i++)
{
    for(j=0;j<256;j++)
    {
     fscanf(fp1, "%s", oneword);
    input_image[i][j] = atoi(oneword);

    }
}

fclose(fp1);
```

Fig.42- Demonstrating how to copy pixel values from a file to matrix

3) Copying the pixel values from .dat report file from the generated implementation results, storing it in another file and using them in MATLAB.



```
data=load('W:\Desktop\sobel1\sobel.dat');
data=transpose(data);
data1=reshape(data,256,256);
data1=transpose(data1);
figure;
imshow(uint8(data1));
```

Fig.43- MATLAB code

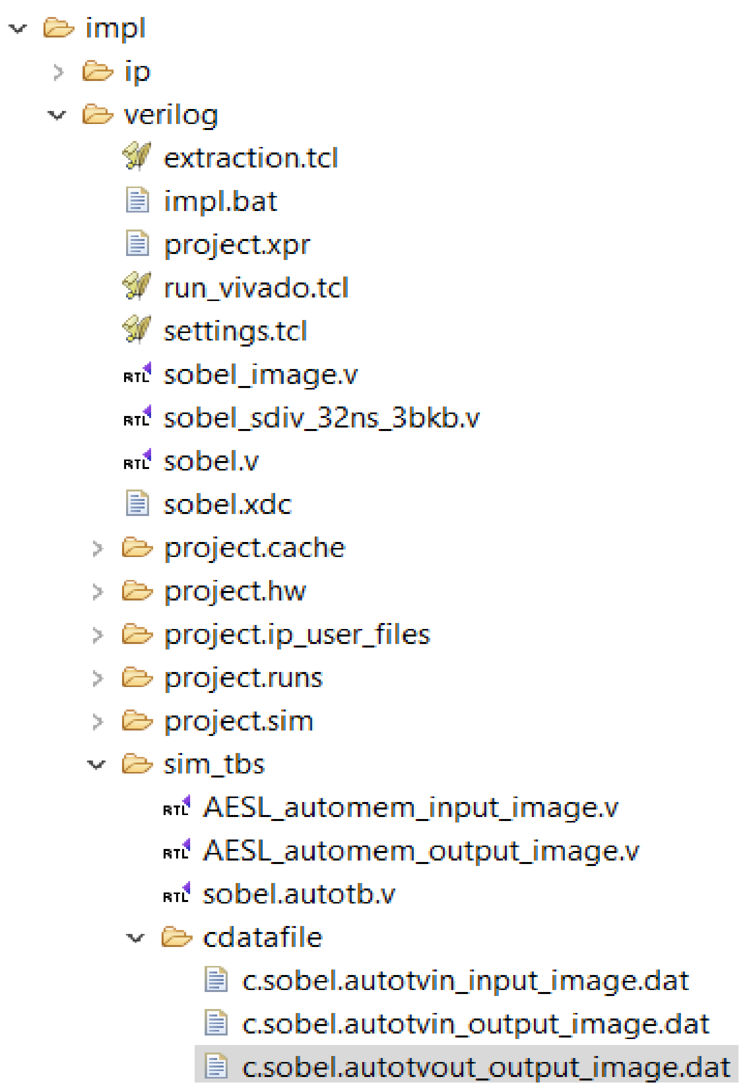

```
impl
  ip
  verilog
    extraction.tcl
    impl.bat
    project.xpr
    run_vivado.tcl
    settings.tcl
    sobel_image.v
    sobel_sdiv_32ns_3bkb.v
    sobel.v
    sobel.xdc
    project.cache
    project.hw
    project.ip_user_files
    project.runs
    project.sim
    sim_tbs
      AESL_automem_input_image.v
      AESL_automem_output_image.v
      sobel.autotb.v
      cdatafile
        c.sobel.autotvin_input_image.dat
        c.sobel.autotvin_output_image.dat
        c.sobel.autotvout_output_image.dat
```

Fig.44- Output file generated containing the output image pixels

In Fig.44, copy the contents of the selected .dat file into another file sobel.dat.

Fig.43 shows how to convert the pixel values in array format to a matrix format so that it is convenient to show the image using imshow function in MATLAB.

4) Sobel filtering code-

Source code-

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void sobel(int input_image[256][256],int output_image[256][256])
{

    int k,l,c,d;
    int image[256][256];
    int i;
    int j;
    int max,min;

    //int Gx[3][3] = {{-1,0,1},{-2,0,2},{-1,0,1}};//horizontal mask
    //int Gy[3][3] = {{1,2,1},{0,0,0},{-1,-2,-1}};//vertical mask
    int x_dir,y_dir,edge_weight;
    x_dir = 0;
    y_dir = 0;
//sobel edge detection algorithm
for(i=0;i<256;i++)
 {
    for(j=0;j<256;j++)
    {
            if((i > 0 && (i < (256-1)) && (j > 0) && (j < (256-1))))// to
exclude the first and last columns and first and last rows as we cannot
do convolution at these places
            {
                    x_dir=-1*input_image[i-1][j-1]+0*input_image[i-
1][j]+input_image[i-1][j+1]-2*input_image[i][j-
1]+0*input_image[i][j]+2*input_image[i][j+1]-1*input_image[i+1][j-
1]+0*input_image[i+1][j]+input_image[i+1][j+1];
```

```
                    y_dir=1*input_image[i-1][j-1]+2*input_image[i-
1][j]+input_image[i-1][j+1]+0*input_image[i][j-
1]+0*input_image[i][j]+0*input_image[i][j+1]-1*input_image[i+1][j-
1]-2*input_image[i+1][j]-1*input_image[i+1][j+1];
                    edge_weight = abs(x_dir) + abs(y_dir);
                    image[i][j] = edge_weight;
                    //printf("%d\n",image[i][j]);
            }
            if(i==0)//for first row
            {
                    output_image[i][j]=0;
            }
            if(i==255)//for last row
            {
                    output_image[i][j]=0;
            }

            if(j==0)//for first column
            {
                    output_image[i][j]=0;
            }
            if(j==255)//for last column
            {
                    output_image[i][j]=0;
            }
     }
 }

//linear transformation to convert the range to 0-255 pixel value
max = image[1][1];
min = image[1][1];
  //finding minimum and maximum values
for (c = 1; c < 255; c++)
     {
     for (d = 1; d < 255; d++)
```

```
    {
            if (image[c][d] > max)
            {
                    max = image[c][d];

            }
            if(image[c][d]<min)
            {
                    min=image[c][d];
            }
    }
    }
//printf("%d\n",max);
//printf("%d\n",min);
  //actual linear transformation equation
for(k=1;k<255;k++)
    {
            for(l=1;l<255;l++)
            {
                    output_image[k][l]=((255 * image[k][l])/(max-min));

            }
    }

//printf("output image pixel value at %d %d is %d\n",57,50,output_image[57][50]);
}
```

Testbench code-

```
#include <stdio.h>
#include <stdlib.h>

int main()
{

   int input_image[256][256],output_image[256][256];
   int i;
   int j;

   FILE *fp1;
   char oneword[65536];


   fp1 = fopen("W:\\Desktop\\sobel\\lena.txt", "r");
   label:
   for(i=0;i<256; i++)
   {
      for(j=0;j<256;j++)
      {
       fscanf(fp1, "%s", oneword);
      input_image[i][j] = atoi(oneword);

      }
   }

   fclose(fp1);
   sobel(input_image,output_image);
  //   printf("input image pixel value at %d %d is %d\n",57,50,input_image[57][50]);
   return 0;
}
```

5) K Means clustering code
Source code-

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void kmeans(short image_in[256][256],short final[256][256])
{
    short rows=256;
    short cols=256;
    short i,j,p,m,n,l,q,s,t;
    int k[4]={5,40,80,180};
    int mean[4];
    short ab[4];
    short r;
    short max_iterations=0;
    short min,mean1,mean2,mean3,mean4;
    short cluster=0;
    short cluster1[rows][cols];
    short cluster2[rows][cols];
    short cluster3[rows][cols];
    short cluster4[rows][cols];
    int size1=0;
    int size2=0;
    int size3=0;
    int size4=0;
    int sum1=0;
    int sum2=0;
    int sum3=0;
    int sum4=0;
    LOOP:
    i=0;
    j=0;
    p=0;
    m=0;
    n=0;
    l=0;
    q=0;
    s=0;
    t=0;
    for(s=0;s<256;s++)
    {
            for(t=0;t<256;t++)
            {
                    cluster1[s][t]=0;
                    cluster2[s][t]=0;
                    cluster3[s][t]=0;
                    cluster4[s][t]=0;
            }
    }
    size1=0;
    size2=0;
    size3=0;
    size4=0;
    sum1=0;
    sum2=0;
    sum3=0;
    sum4=0;
    for(i=0;i<rows;i++)
    {
            for(j=0;j<cols;j++)
            {
                    r=image_in[i][j];
                    //printf("input   image   pixel   value   at   %d   %d   is
%u\n",i,j,r);
                    for(p=0;p<4;p++)
                    {
                    ab[p]=abs((r-k[p]));
                    }
```

```
                min = ab[0];
                p=1;
                for (p = 1; p < 4; p++)
                {
                  if (ab[p] < min)
                     {
                         min = ab[p];
                         cluster=p;
                     }
            }

                if (cluster==0)
                {
                        cluster1[i][j]=r;
                        size1++;

                }
                if (cluster==1)
                {
                        cluster2[i][j]=r;
                        size2++;
                }
                if(cluster==2)
                {
                        cluster3[i][j]=r;
                        size3++;
                }
                if(cluster==3)
                {
                        cluster4[i][j]=r;
                        size4++;
                }

        }
}
```

```
printf("%d",size1);
for(m=0;m<rows;m++)
{
    for(n=0;n<cols;n++)
    {
            sum1=sum1+cluster1[m][n];
            sum2=sum2+cluster2[m][n];
            sum3=sum3+cluster3[m][n];
            sum4=sum4+cluster4[m][n];
    }
}

mean1=sum1/size1;
mean2=sum2/size2;
mean3=sum3/size3;
mean4=sum4/size4;
mean[0]=mean1;
mean[1]=mean2;
mean[2]=mean3;
mean[3]=mean4;
//printf("sum1=%d\n",sum1);
//printf("mean2=%d\n",mean2);
//if((mean[0]==k[0]) && (mean[1]==k[1]) && (mean[2]==k[2]) &&
(mean[3]==k[3]))
    for(l=0;l<rows;l++)
    {
            for(q=0;q<cols;q++)
            {

                            {
                            if((cluster1[l][q])>0)
                            {
                            cluster1[l][q]=50;
                            }
                            if((cluster2[l][q])>0)
```

```
                        {
                        cluster2[l][q]=70;
                        }
                        if((cluster3[l][q])>0)
                        {
                        cluster3[l][q]=40;
                        }
                        if((cluster4[l][q])>0)
                        {
                        cluster4[l][q]=220;
                        }


    final[l][q]=cluster1[l][q]+cluster2[l][q]+cluster3[l][q]+cluster4[l][q
];

                        }
                }
        }
/*
else
{
k[0]=mean[0];
k[1]=mean[1];
k[2]=mean[2];
k[3]=mean[3];
max_iterations++;
if(max_iterations<100)
{
goto LOOP;
}
}
*/
//printf("output image pixel value at %d %d is %d\n",0,0,final[0][0]);
}
```

Testbench code-

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   short input_image[256][256],output_image[256][256];
   int i;
   int j;

   FILE *fp1;
   char oneword[65536];


   fp1 = fopen("W:\\Desktop\\kmeans\\pic.txt", "r");
   label:
   for(i=0;i<256; i++)
   {
      for(j=0;j<256;j++)
      {
       fscanf(fp1, "%s", oneword);
      input_image[i][j] = atoi(oneword);

      }
   }

   fclose(fp1);
   //printf("input image pixel value at %d %d is 
%u\n",0,0,input_image[0][0]);
   kmeans(input_image,output_image);

   return 0;
}
```