

Part A

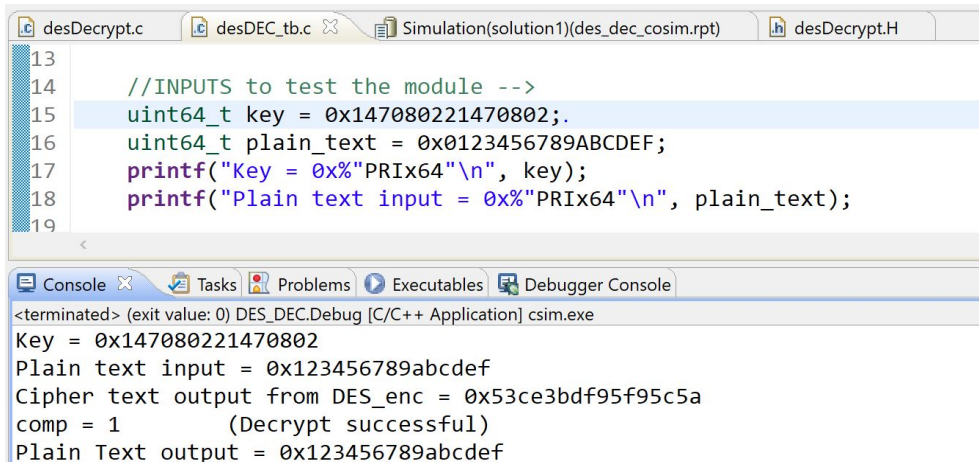
Question 2:

Prepare a C-based testbench to simulate the design. Run the simulation and add relevant screenshots (that show that the algorithm works and that you understand the output) to your report. Your simulation should include printing the test inputs and outputs of your testbench. [1 pt]

Test bench:

```
9 int main(){
10     //Outputs of encryption and decryption modules
11     uint64_t enc_out;
12     uint64_t dec_out;
13
14     //INPUTS to test the module -->
15     uint64_t key = 0x147080221470802;
16     uint64_t plain_text = 0x0123456789ABCDEF;
17     printf("Key = 0x%"PRIx64"\n", key);
18     printf("Plain text input = 0x%"PRIx64"\n", plain_text);
19
20     //Outputs True if plain_text input to enc == plain_text output from dec
21     int comp;
22
23     enc_out = des_enc(plain_text, key);
24     printf("Cipher text output from DES_enc = 0x%"PRIx64"\n", enc_out);
25     dec_out = des_dec(enc_out, key);
26
27     if(dec_out == plain_text){
28         comp = 1;
29         printf("comp = %d", comp);
30         printf("    (Decrypt successful)\n");
31     }
32     else{
33         comp = 0;
34         printf("comp = %d\n", comp);
35     }
36     printf("Plain Text output = 0x%"PRIx64"\n", dec_out);
37 }
38
```

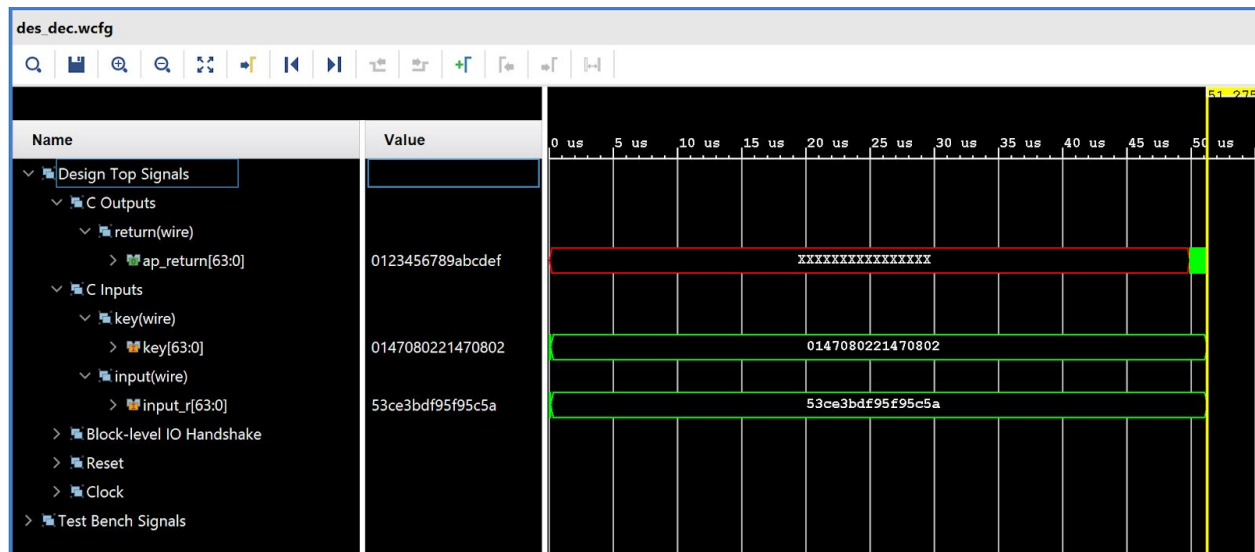
C-Simulation results:



The screenshot shows a C simulation environment with a code editor and a console window. The code editor displays the testbench code from the previous block, with lines 14-19 highlighted. The console window shows the output of the simulation, which matches the expected results.

```
desDecrypt.c | desDEC_tb.c | Simulation(solution1)(des_dec_cosim.rpt) | desDecrypt.H
13
14 //INPUTS to test the module -->
15 uint64_t key = 0x147080221470802;
16 uint64_t plain_text = 0x0123456789ABCDEF;
17 printf("Key = 0x%"PRIx64"\n", key);
18 printf("Plain text input = 0x%"PRIx64"\n", plain_text);
19
<terminated> (exit value: 0) DES_DEC.Debug [C/C++ Application] csim.exe
Key = 0x147080221470802
Plain text input = 0x123456789abcdef
Cipher text output from DES_enc = 0x53ce3bdf95f95c5a
comp = 1    (Decrypt successful)
Plain Text output = 0x123456789abcdef
```

Simulation results:



We verify our results with the C-Simulation results we did earlier. The input to our top function (des_dec) is the output ciphertext we got from DES encrypt.

Therefore, input = 0x53ce3bdf95f95c5a

Key = 0x0147080221470802

Output = 0x0123456789abcdef (Same plain_text we input in our testbench).

Question 3:

Synthesize the given baseline design and explain the initial synthesized hardware's resource usage, describing which parts of the C code are mapped to what kind of FPGA resources. State and explain your reasons for the mappings. [2 pts]

• Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2202	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	3	-	38	33	0
Multiplexer	-	-	-	312	-
Register	-	-	1222	-	-
Total	3	0	1260	2547	0
Available	100	90	41600	20800	0
Utilization (%)	3	0	3	12	0

- Our board XC7A35TLCPG236 has 100 BRAM_18k (block RAM), 90 DSP48E, and so resources available out of which we only use about 3% of BRAM, 3% of Flip Flops, and 12% of Lookup tables making a total resource utilization of 6%.

Memory

Memory	Module	BRAM_18K	FF	LUT	URAM	Words	Bits	Banks	W*Bits*Banks
E_U	des_dec_E	0	6	5	0	48	6	1	288
IP_U	des_dec_IP	0	7	7	0	64	7	1	448
P_U	des_dec_P	0	6	3	0	32	6	1	192
PC1_U	des_dec_PC1	0	6	6	0	56	6	1	336
PC2_U	des_dec_PC2	0	6	5	0	48	6	1	288
PI_U	des_dec_PI	0	7	7	0	64	7	1	448
S_U	des_dec_S	1	0	0	0	512	4	1	2048
sub_key_U	des_dec_sub_key	2	0	0	0	16	64	1	1024
Total	8	3	38	33	0	840	106	8	5072

- The figure above shows memory utilization with respect to different modules for example initial permutation module (IP_C) uses 7 flip flips and 7 lookup tables, permutation combination 1 (PC1_C) uses 6 FF and 6 LUTs, S-box uses block RAM of size 18k bits, and so on.
- The number of FFs and LUTs are the resources used to implement our configurable logic block (CLB) and thus depend on our design logic.
- Similarly, the report provides utilization w.r.t to each variable name (Expression) in the design, different multiplexers used in the circuit (used for both if-else loop and LUTs), and different registers defined in our design.
- Note that our initial design does not include pragmas, i.e. no pipelining or loop unrolling have performed.

Question 4:

Perform C/RTL co-simulation of your design and add screenshot(s) to your report. Explain how co-simulation works and annotate the screenshot to show that the co-simulation works as expected. [2 pts]

C/RTL co-simulation results:

Cosimulation Report for 'des_dec'

Result							
RTL	Status	Latency			Interval		
		min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	5109	5109	5109	NA	NA	NA

Results from testbench:

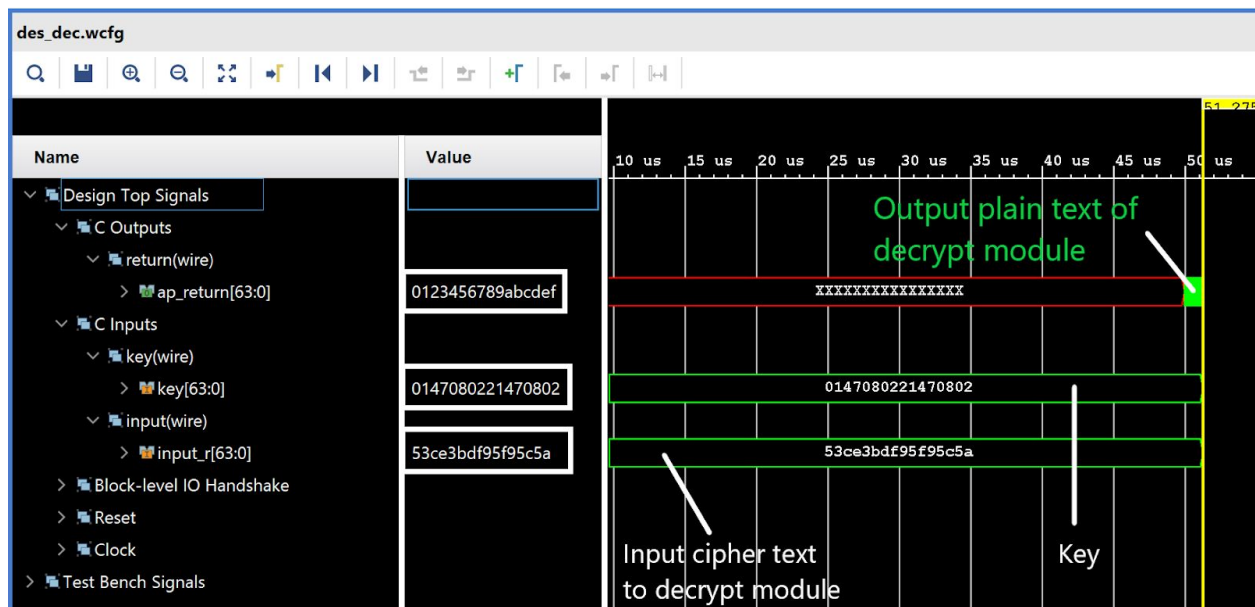
Input plain text to DES encrypt = **0x123456789abcdef**

Input cipher text to DES decrypt = 0x53ce3bdf95f95c5a

Key = 0x147080221470802

Output = **0x123456789abcdef** (Same plain_text we input in our testbench).

Simulation results:

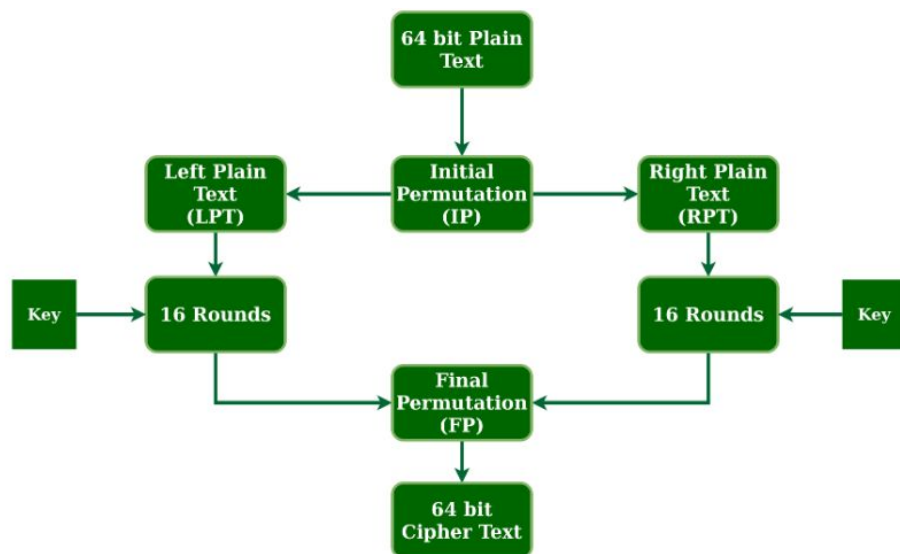


- As we see in the figure above, our simulation results from the C/RTL co-simulation complements the results from C-simulation thus confirming that our DES decrypt module work.

Question 5:

What part(s) of the design are dominating the latency? Explain your justifications using screenshots of the scheduling analysis and synthesis report. [1 pt]

Block-level diagram of DES encrypt/decrypt algorithm:



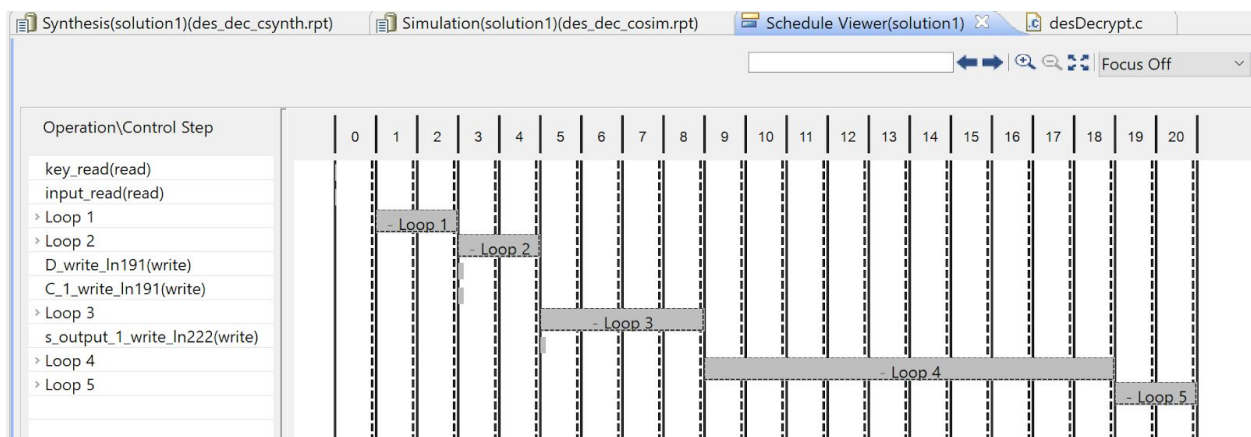
- According to the code, the for loop used for the round function $f(R, K)$ should have the highest latency as it runs for 16 rounds.
- Each round the transformed 48-bit key is XOR with an expanded permutation combination2 (PC_2: Right plain text) which is passed through an S-box. The result is permuted and XOR with the left plain text.
- This round function is loop 4 in our code. The results of the scheduling analysis are shown below.

```

219     }
220
221     for (i = 0; i < 16; i++) {
222         /* f(R,k) function */
223         s_input = 0;
224
225         for (j = 0; j < 48; j++) {
226             s_input <<= 1;
227             s_input |= (uint64_t) ((R >> (32-E[j])) & LB32_MASK);
228         }
229         s_input = s_input ^ sub_key[15-i];
230
231         /* S-Box Tables */
232         for (j = 0; j < 8; j++) {
233             // 00 00 RCCC CR00 00 00 00 00 s_input
234             // 00 00 1000 0100 00 00 00 00 row mask
235             // 00 00 0111 1000 00 00 00 00 column mask

```

Performance Profile			
	Pipelined	Latency	Iteration Latency
▼ des_dec	-	5109	-
● Loop 1	no	128	2
● Loop 2	no	112	2
> ● Loop 3	no	1584	99
> ● Loop 4	no	3152	197
● Loop 5	no	128	2



Question 6:

Prepare a list of potential structures in the code that provide opportunities for optimization (e.g., what are the loops/datatypes/arrays which you can apply pragmas to). For these structures, list which pragmas are likely to be applicable. Are there any pragmas that do not work? Explain your answer. [2 pt]

List of structures where pragmas can be applied:

- Loop1: Used for the initial permutation.
- Loop2: Used for the initial key schedule calculation.
- Loop3: Used for calculations of 16 keys used in the round function.
- Loop4: Used for the round function calculations.
- Arrays PI (initial permutation table), IP (inverse initial permutation combination), E (expansion table), P (post-S-box permutation table), PC1, PC2 (permutation choice table 1 & 2) and S (S-box table).

List of pragmas applicable:

- Loop unrolling (to loops 1, 2, 3, and 4)
- Loop pipelining (to loops 1, 2, 3, and 4)
- Array partition ()
- Array map
- Loop optimization (flatten and merge)
- Task level pipelining (Dataflow and Stream)
- Kernel optimization (Allocation, resource)

Pragmas that do not work:

- Array mapping won't work for the arrays PI, IP, S, E since none of them are being used for calculation inside loops.
- HLS Allocation with operation instances like srem, udiv, urem e.t.c, won't change anything since none of them are being performed in our design.

Part B

Question 1:

Given the pragmas you identified in A-(6), identify pragmas/settings that will:

- (a) Increase the latency of the synthesized hardware (compared to the baseline) [0.2 pt]
- (b) Decrease the latency of the synthesized hardware (compared to the baseline) [0.2 pt]
- (c) Increase the resource usage of the synthesized hardware (compared to the baseline) [0.2 pt]
- (d) Decrease the resource utilization of the synthesized hardware (compared to the baseline) [0.2 pt]
- (e) Increase the throughput of the synthesized hardware (compared to the baseline) [0.2 pt]

Latency and Resource Utilization without pragmas:

- Latency (clock cycles)
 - Summary

Latency		Interval		Type
min	max	min	max	
5109	5109	5109	5109	none

Utilization Estimates

- Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2202	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	3	-	38	33	0
Multiplexer	-	-	-	312	-
Register	-	-	1222	-	-
Total	3	0	1260	2547	0
Available	100	90	41600	20800	0
Utilization (%)	3	0	3	12	0

- A).** Pragma = **HLS Allocation**
Instance = lshr (Logical Shift Right)
Limit = 1
Type = operation
Total number of LUTs used by the lshr operation before adding pragma = 1508.
Latency without pragma = 5109
Total number of LUTs used by the lshr operation after adding pragma = 332
Latency with pragma = 5237

This pragma limits the number of combination logical shift right to 1. This reduces resource utilization but increases latency since all the lshr calculations are performed on just 1 core.

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
5237	5237	5237	5237	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1024	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	3	-	38	33	0
Multiplexer	-	-	-	434	-
Register	-	-	1280	-	-
Total	3	0	1318	1491	0
Available	100	90	41600	20800	0
Utilization (%)	3	0	3	7	0

- B).** Pragma: **HLS Dataflow**
Percentage utilization before pragma = 6.08%
Latency without pragma = 5109
Percentage utilization after pragma = 8.06%
Latency with pragma = 3154

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
4984	4984	3154	3154	dataflow

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	40	-
FIFO	0	-	50	608	-
Instance	1	-	1532	2754	-
Memory	2	-	0	0	0
Multiplexer	-	-	-	54	-
Register	-	-	6	-	-
Total	3	0	1588	3456	0
Available	100	90	41600	20800	0
Utilization (%)	3	0	3	16	0

- C).** Pragma: **HLS Unroll on loop 1 & 2.**
Percentage utilization before pragma = 6.08%
 Latency without pragma = 5109
Percentage utilization after pragma = 24%
 Latency with pragma = 4878

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
4878	4878	4878	4878	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	13359	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	3	-	393	33	0
Multiplexer	-	-	-	335	-
Register	-	-	1192	-	-
Total	3	0	1585	13727	0
Available	100	90	41600	20800	0
Utilization (%)	3	0	3	65	0

With unrolling the latency decreases but resource utilization increases. To decrease the latency even further we can unroll the nested loops 3 & 4. However, doing so will increase resource utilization.

- D).** Pragma: **HLS Pipeline on loop 1 & 2 (Factor = 32 for both loops)**
Percentage utilization before pragma = 6.08%
Latency without pragma = 5109
Percentage utilization after pragma = 5.93%
Latency with pragma = 1976

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
4991	4991	4991	4991	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2210	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	3	-	38	33	0
Multiplexer	-	-	-	342	-
Register	-	-	1089	-	-
Total	3	0	1127	2585	0
Available	100	90	41600	20800	0
Utilization (%)	3	0	2	12	0

- E). Pragma: **HLS Pipeline on loop 4.**
Percentage utilization before pragma = 6.08%
Latency without pragma = 5109
Percentage utilization after pragma = 4.75%
Latency with pragma = 1976

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1976	1976	1976	1976	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1786	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	6	-	26	25	0
Multiplexer	-	-	-	230	-
Register	-	-	901	-	-
Total	6	0	927	2041	0
Available	100	90	41600	20800	0
Utilization (%)	6	0	2	9	0

With pipelining, the latency decreases but resource utilization increases. We can also try pipelining the entire DES function. The results of it are shows decryption done in just 16 cycles. Pipelining on the entire DES decrypt function:

Latency (clock cycles)

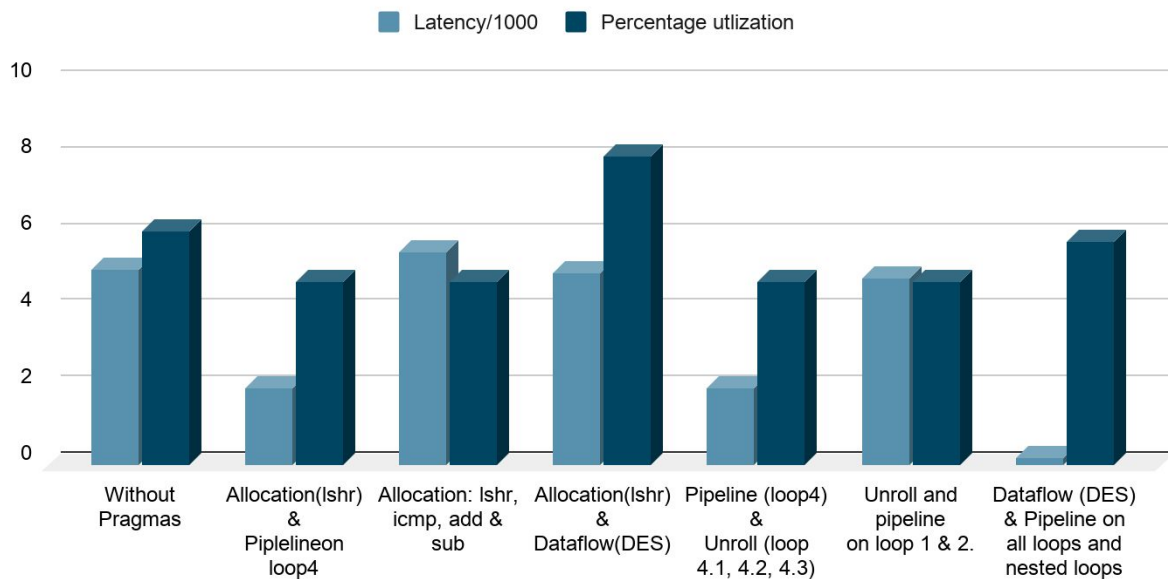
Summary

Latency		Interval		
min	max	min	max	Type
16	16	1	1	function

Question 2:

Implement combinations of pragmas used in the above question. Report the resource usage, latency, timing of every solution generated. Make a graph where x-axis is latency and y-axis is area, and plot every solution to show trade-offs between the solutions. Explain the graph in your own words, what are the tradeoffs and the reasons ? [3 pts]

Points scored



- The graph above shows the percentage area and latency used by different designs.
- Limiting resources using HLS Allocation increases latency and decreases resource utilization however when used with pipelining throughput increases by decreasing both latency and resource utilization.
- Dataflow decreasing the latency by task level pipelining. When used with limited resources, the area utilization increases however the latency decreases.
- Performing task level pipelining (Dataflow) on the entire DES function and performing pipelining on all the loops and nested loops will result in the lowest latency whereas the area utilization is almost equal to that of the baseline design.
- The general trend followed is:
 1. Dataflow & Pipelining decreases the latency but increases the area by a small amount.
 2. Allocation increases latency by limiting the resource used.
 3. Unroll decreases the latency but increases the resource utilization by making several copies and running them parallelly.

Question 3:

Add resource constraints to the C design (make sure that the number of resource constraints are lower than reported in the baseline design synthesis report). Add the resulting resource usage and scheduling screenshot to your report. Explain how this impacts synthesis of your baseline design. [2 pts]

- Constraints have been added on the following operations:
 Lshr (Logical Right Shift)
 Lcmpr (Integer Compare)
 Add (Addition)
 Sub (Subtraction)
 Dadd (Double-Precision Floating-Point Addition)
 Dsub (Double-Precision Floating-Point Subtraction)
- For each of the following operations, we limit the number of operations to 1.
- The results (resource utilization) after synthesis is shown below.

Utilization Estimates

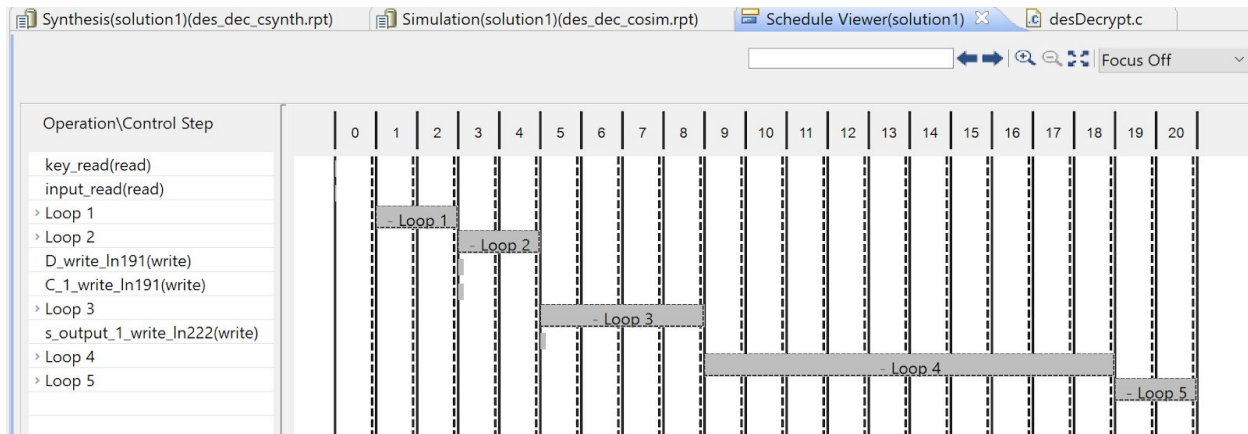
Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	902	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	3	-	38	33	0
Multiplexer	-	-	-	752	-
Register	-	-	1258	-	-
Total	3	0	1296	1687	0
Available	100	90	41600	20800	0
Utilization (%)	3	0	3	8	0

- The figure below shows the resource utilization based on the operations for different variables for **baseline design**:

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
add_ln245_1_fu_1154_p2	+	0	0	18	11	11
add_ln245_fu_1132_p2	+	0	0	16	9	9
i_5_fu_547_p2	+	0	0	15	6	1
i_7_fu_622_p2	+	0	0	15	5	1
i_8_fu_1241_p2	+	0	0	15	7	1
i_9_fu_845_p2	+	0	0	15	5	1
i_fu_486_p2	+	0	0	15	7	1
i_4_fu_798_p2	+	0	0	15	6	1
i_5_fu_953_p2	+	0	0	13	4	1
i_6_fu_1187_p2	+	0	0	15	6	1
i_fu_865_p2	+	0	0	15	6	1
sub_ln173_fu_515_p2	-	0	0	15	8	7
sub_ln183_fu_590_p2	-	0	0	15	8	7
sub_ln217_fu_813_p2	-	0	0	15	5	6
sub_ln228_fu_887_p2	-	0	0	15	7	6
sub_ln231_fu_876_p2	-	0	0	15	4	5
sub_ln239_1_fu_1007_p2	-	0	0	8	6	6
sub_ln239_2_fu_1013_p2	-	0	0	8	6	6
sub_ln239_fu_989_p2	-	0	0	15	7	7
sub_ln242_1_fu_1025_p2	-	0	0	8	6	6
sub_ln242_fu_1019_p2	-	0	0	8	6	6
sub_ln253_fu_1208_p2	-	0	0	15	7	6
sub_ln269_fu_1256_p2	-	0	0	15	8	7
and_ln239_fu_1048_p2	and	0	0	48	48	48
and_ln242_fu_1096_p2	and	0	0	47	47	47
empty_11_fu_628_p2	icmp	0	0	11	5	4
empty_12_fu_634_p2	icmp	0	0	11	5	4
empty_14_fu_646_p2	icmp	0	0	11	5	1
empty_16_fu_658_p2	icmp	0	0	11	5	1
icmp_ln171_fu_480_p2	icmp	0	0	11	7	8
icmp_ln181_fu_541_p2	icmp	0	0	11	6	5
icmp_ln191_fu_616_p2	icmp	0	0	11	5	6
icmp_ln215_fu_792_p2	icmp	0	0	11	6	6
icmp_ln222_fu_839_p2	icmp	0	0	11	5	6
icmp_ln226_fu_859_p2	icmp	0	0	11	6	6
icmp_ln234_fu_947_p2	icmp	0	0	11	4	5
icmp_ln251_fu_1181_p2	icmp	0	0	11	6	7
icmp_ln266_fu_1235_p2	icmp	0	0	11	7	8
lshr_ln173_fu_525_p2	lshr	0	0	182	64	64
lshr_ln183_fu_600_p2	lshr	0	0	182	64	64
lshr_ln217_fu_823_p2	lshr	0	0	166	56	56
lshr_ln228_fu_897_p2	lshr	0	0	101	32	32
lshr_ln239_1_fu_1056_p2	lshr	0	0	150	48	48
lshr_ln239_fu_1042_p2	lshr	0	0	150	48	48
lshr_ln242_1_fu_1104_p2	lshr	0	0	148	47	47
lshr_ln242_fu_1090_p2	lshr	0	0	148	45	47
lshr_ln253_fu_1218_p2	lshr	0	0	101	32	32
lshr_ln269_fu_1266_p2	lshr	0	0	182	64	64
empty_13_fu_640_p2	or	0	0	2	1	1
empty_15_fu_652_p2	or	0	0	2	1	1
empty_17_fu_664_p2	or	0	0	2	1	1
or_ln240_fu_1074_p2	or	0	0	2	1	1
R_1_fu_1202_p2	xor	0	0	32	32	32
xor_ln231_1_fu_941_p2	xor	0	0	48	48	48
xor_ln231_fu_935_p2	xor	0	0	47	47	47
Total		55	0	2202	948	899

- The figure below shows the scheduling of our **baseline design** with all operation completing in 20 cycles:

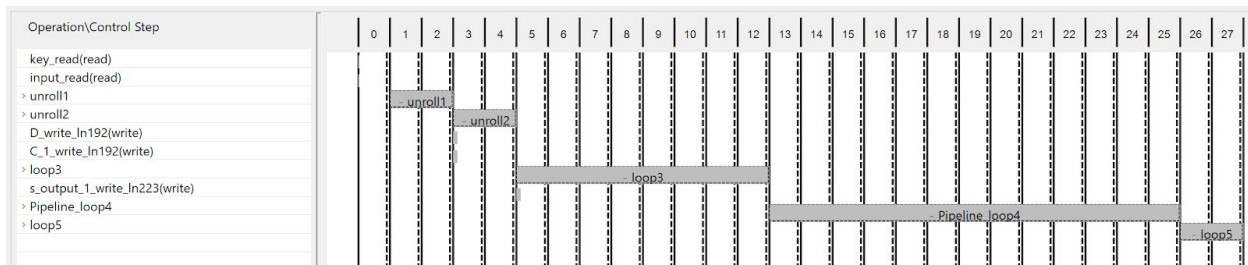


- The figure below shows the resource utilization based on the operations for different variables for our **constrained design**:

Expression

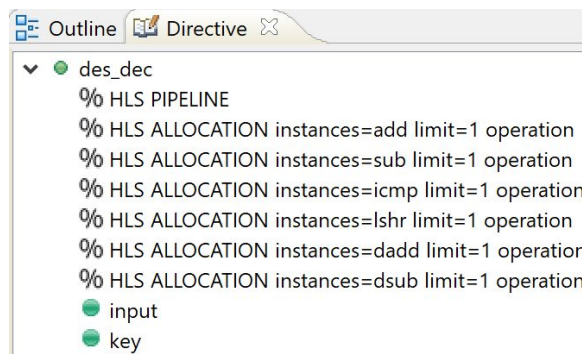
Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
grp_fu_512_p2	+	0	0	18	11	11
grp_fu_746_p2	-	0	0	15	7	7
and_In240_fu_1296_p2	and	0	0	48	48	48
and_In243_fu_1336_p2	and	0	0	47	47	47
grp_fu_868_p2	icmp	0	0	11	7	7
grp_fu_874_p2	lshr	0	0	182	64	64
lshr_In240_1_fu_895_p2	lshr	0	0	150	48	48
lshr_In243_1_fu_904_p2	lshr	0	0	148	47	47
lshr_In243_fu_899_p2	lshr	0	0	148	45	47
empty_13_fu_1019_p2	or	0	0	2	1	1
empty_15_fu_1023_p2	or	0	0	2	1	1
empty_17_fu_1028_p2	or	0	0	2	1	1
or_In241_fu_1321_p2	or	0	0	2	1	1
R_1_fu_1420_p2	xor	0	0	32	32	32
xor_In232_1_fu_1237_p2	xor	0	0	48	48	48
xor_In232_fu_1231_p2	xor	0	0	47	47	47
Total		16	0	902	455	457

- The figure below shows the scheduling of our **constrained design** with all operations completing in 27 cycles. This shows how adding constraints on the resources increases the latency of the design.



Impact: Adding constraints decreased the resource utilization percentage from 6.08% to 4.77% however, the latency increases by 448. Adding constraints might be useful when using it with an HLS pipeline to decrease both latency and area to increase the throughput of the design.

Results from a pipelined structure with resource constraints:



Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
112	112	112	112	function

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	14488	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	1	-	0	0	-
Multiplexer	-	-	-	1644	-
Register	-	-	4957	-	-
Total	1	0	4957	16132	0
Available	100	90	41600	20800	0
Utilization (%)	1	0	11	77	0

Question 4:

Currently, the C implementation is not amenable to pipelining with an II=1, modify the C code to achieve this. For this part of the question, please create a new project with your modified C code. Explain the changes made. [2 pts]

- Pipelined design with II = 1 is shown below:



- The results of synthesis (latency and resource utilization) are shown below. No changes were made in order for the C implementation to be amenable to pipelining with an II=1. The DES code provided is amenable to pipelining with II = 1.

Latency (clock cycles)

☐ Summary

Latency		Interval		Type
min	max	min	max	
16	16	1	1	function

Utilization Estimates

☐ Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	16486	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	64	-	0	0	-
Multiplexer	-	-	-	-	-
Register	0	-	13991	3104	-
Total	64	0	13991	19590	0
Available	100	90	41600	20800	0
Utilization (%)	64	0	33	94	0

Part C

Latency and Resource Utilization without pragmas:

- Latency (clock cycles)
 - Summary

Latency		Interval		Type
min	max	min	max	
5109	5109	5109	5109	none

Utilization Estimates

- Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2202	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	3	-	38	33	0
Multiplexer	-	-	-	312	-
Register	-	-	1222	-	-
Total	3	0	1260	2547	0
Available	100	90	41600	20800	0
Utilization (%)	3	0	3	12	0

Question 1:

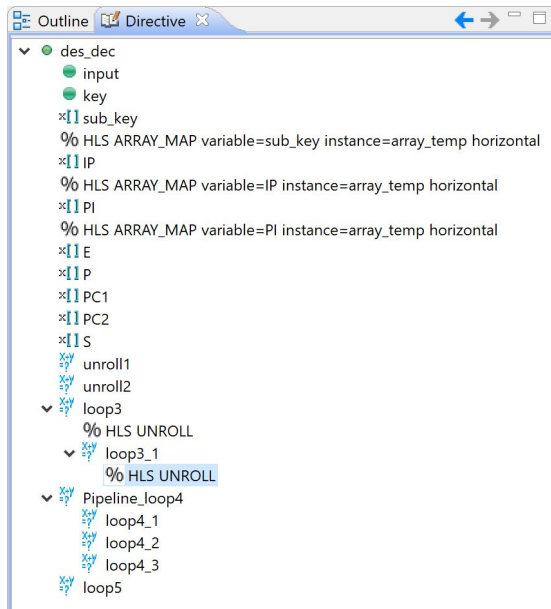
If your design has a loop and the **loop bound is a variable**, does your design synthesize ? Explain in detail.
[0.5 pt]

- If the loop bound is a variable, then the design won't synthesize since the number of loops is calculated at compile time. It cannot be a variable.
- In terms of hardware, the design won't synthesize since keeping loop bound as variable means there can be a variable number of hardware blocks being placed at run-time. This is not possible.
- This is the same reason why in Verilog we cannot write a for loop inside an always block since that will place N number of hardware blocks in the design at runtime.

Question 2:

Consider a loop in your design which uses an array. If you map the array to a BRAM and unroll loop fully, what do you observe ? Write the pragmas you used. [0.5 pt]

- Mapping the array to a BRAM (using HLS Array_Map) and unrolling the loop fully that uses that array (using HLS Unroll), it is observed that both latency and the area utilization decreases.



Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
3531	3531	3531	3531	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1977	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	4	-	18	14	0
Multiplexer	-	-	-	437	-
Register	-	-	1016	-	-
Total	4	0	1034	2428	0
Available	100	90	41600	20800	0
Utilization (%)	4	0	2	11	0

- As seen from the results above, it is observed that the total number of BRAM used increases while the number of FFs and LUTs decrease. This is compliant with the pragmas we used.
- BRAM increases because we mapped all 3 arrays to a BRAM named array_temp and the FFs and LUTs who were previously utilized by the arrays are now free thus decreasing the number of FF and LUT used.

Memory

Memory	Module	BRAM_18K	FF	LUT	URAM	Words	Bits	Banks	W*Bits*Banks
E_U	des_dec_E	0	6	5	0	48	6	1	288
P_U	des_dec_P	0	6	3	0	32	6	1	192
PC1_U	des_dec_PC1	0	6	6	0	56	6	1	336
S_U	des_dec_S	1	0	0	0	512	4	1	2048
array_temp_U	des_dec_array_temp	3	0	0	0	144	48	1	6912
Total	5	4	18	14	0	792	70	5	9776

- Latency decreases because we unrolled the loop for faster processing. This can be seen in the figure below where loop3 has not been mentioned since it has been completely unrolled.

Loop

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- unroll1	128	128	2	-	-	64	no
- unroll2	112	112	2	-	-	56	no
- Pipeline_loop4	3152	3152	197	-	-	16	no
+ loop4_1	96	96	2	-	-	48	no
+ loop4_2	32	32	4	-	-	8	no
+ loop4_3	64	64	2	-	-	32	no
- loop5	128	128	2	-	-	64	no

Question 3:

Synthesize the C design with a variable as ap_start signal. Is it possible to synthesize without the handshake signal ? [0.5 pt]

- It is not possible to synthesize without the handshake signal and the interface time ap_start will have no effect since the process already starts from the 1st cycle.

Question 4:

Compare and contrast the differences between traditional VHDL/Verilog based design and C-based HLS.
[0.5 pt]

Verilog/ VHDL:

- We can perform sequential and simultaneous operations using always blocks.
- Need hardware understanding to understand.
- We can describe the circuit by both structural and behavioral modeling.

C based HLS:

- It can only perform sequential operations.
- Easier to understand and to write since it's a higher-level language.
- We can code only using a behavioral model but it is easier to write algorithms ins C.

Question 5:

When you add the loop unroll pragma with increasing unroll factor, would one always expect the resource utilization to increase ? Explain your answer. [0.5 pt]

- Increasing the unroll factor(1 to N-1) increases hardware utilization since the factor number of copies are made of the loop (equal to the factor number of hardware) which are processed parallely.
- However, it is not always necessary that latency decreases linearly with increasing factor since it depends on data dependencies between loop iterations and the available hardware.
- It is also noticed that when performing loop unroll fully (factor = 64) on loop1 in DES, the resource and latency are both less than that of the baseline design, however, a logical result for that should have increased the area utilization since we make 64 copies of a loop that runs 64 times and all operations should have run parallely. One explanation for that might be the deficiency of available hardware resources.

Question 6:

Consider the loop in your design which uses an array. How does array partitioning impact the resulting latency? [0.5 pt]

- Array partitioning results in increased latency than that of the baseline design. It also increases resource utilization since partitioned BRAM array will have to be implemented using flip-flops and lookup tables.
- Array partitioning on sub_key used in loop3. The result of the latency is shown below:

Latency (clock cycles)

☐ **Summary**

Latency		Interval		Type
min	max	min	max	
6629	6629	6629	6629	none

Question 7:

Pipelining a loop with II=1 is expected to reduce latency. Does this optimization always result in reduction of latency? [0.5 pt]

- No, this optimization does not always result in a reduction of latency.
- Pipelining a loop with II=1 means that the pipelined design starts processing right from the 1st cycle however, this depends on the data dependencies from the previous loops and also between loop iterations.
- Pipelining will not increase the latency but may always decrease the latency.

Question 8:

Contrast the difference in resource usage and latency, between (a) an array resource mapped as registers and, (b) an array resource mapped as BRAM. State reasons for this difference. [0.5 pt]

- Array resource mapped as a BRAM: (BRAM is faster than registers and when mapped using registers it increases resource utilization)

```
▼ des_dec
  ● input
  ● key
  *1 sub_key
  % HLS ARRAY_MAP variable=sub_key instance=array_BRAM horizontal
```

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
5109	5109	5109	5109	none

Utilization Estimates

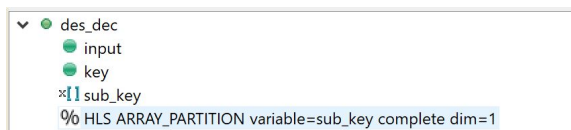
Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2202	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	3	-	38	33	0
Multiplexer	-	-	-	312	-
Register	-	-	1222	-	-
Total	3	0	1260	2547	0
Available	100	90	41600	20800	0
Utilization (%)	3	0	3	12	0

Memory

Memory	Module	BRAM_18K	FF	LUT	URAM	Words	Bits	Banks	W*Bits*Banks
E_U	des_dec_E	0	6	5	0	48	6	1	288
IP_U	des_dec_IP	0	7	7	0	64	7	1	448
P_U	des_dec_P	0	6	3	0	32	6	1	192
PC1_U	des_dec_PC1	0	6	6	0	56	6	1	336
PC2_U	des_dec_PC2	0	6	5	0	48	6	1	288
PI_U	des_dec_PI	0	7	7	0	64	7	1	448
S_U	des_dec_S	1	0	0	0	512	4	1	2048
sub_key_U	des_dec_sub_key	2	0	0	0	16	64	1	1024
Total		8	38	33	0	840	106	8	5072

- Array resource mapped as register:



Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
6629	6629	6629	6629	none

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2191	-
FIFO	-	-	-	-	-
Instance	-	-	0	1170	-
Memory	1	-	38	33	-
Multiplexer	-	-	-	436	-
Register	-	-	2247	-	-
Total	1	0	2285	3830	0
Available	100	90	41600	20800	0
Utilization (%)	1	0	5	18	0

Question 9:

Identify 4 c constructs that are not synthesizable. Explain your reasoning. [0.5 pts]

- The 4 C constructs that are non-synthesizable are:
 1. Pointers
 2. Recursive functions
 3. Malloc
 4. Alloc
- All of these constructs are system calls that cannot be synthesized. This is because we cannot place hardware blocks in our design during runtime. That is supposed to be calculated at compile time. That is the functionality of the system must be defined at compile time.
- It is the same reason why we cannot have loop bound as a variable.

Question 10:

Take 2 constructs from (9) and explain how you can make them synthesizable. Give an example for each [0.5 pts]?

- Design using Malloc and Alloc can be converted into synthesizable designs by writing them as functions of fixed size and bounds. This way they can be compiled at runtime.