Exercise 1

1- There can be two different approaches to solve this problem.

a) The first approach is using the integer multiplication, which is based on the position value of each digit in a decimal number.

Let's assume that the integer value n is represented by four digits $d, c, b\ , a$ as n=dcba. For example, $n = 6342$ where $d = 6$, $c = 3$, $b = 4$, and $a = 2$.

Then we have

$$n = 10^3 d + 10^2 c + 10b + c$$

We can use this expression to convert a 4-digit BCD code into its equivalent binary. This code shows the design top-function

```
******************************************************/
#include "bcd2binary_mult.h"


void bcd2binary_mult(uint16 packed_bcd, uint14 *output_bcd) {
#pragma HLS INTERFACE ap_none port=packed_bcd
#pragma HLS INTERFACE ap_none port=output_bcd
#pragma HLS INTERFACE ap_ctrl_none port=return


    uint14 bcd = 0;

    uint4 digit_1 = packed_bcd(3, 0);
    uint4 digit_2 = packed_bcd(7, 4);
    uint4 digit_3 = packed_bcd(11, 8);
    uint4 digit_4 = packed_bcd(15, 12);



    bcd = bcd + digit_1 * 1;
    bcd = bcd + digit_2 * 10;
    bcd = bcd + digit_3 * 100;
    bcd = bcd + digit_4 * 1000;

    *output_bcd = bcd;
}
```

b) The second approach utilises the reverse function of the double-dabble algorithm that is shown here

```
uint32 reverse_double_dabble(uint32 scp) {
	uint32 s;
	s = scp;
	s = scp >> 1;
	if (s(19, 16) > 7)
	  s(19, 16) = s(19, 16) - 3;
	if (s(23, 20) > 7)
	  s(23, 20) = s(23, 20) - 3;
	if (s(27, 24) > 7)
	  s(27, 24) = s(27, 24) - 3;


	return s;
    }



void binary2bcd_rdd(uint16 packed_bcd, uint16 *in_binary) {
#pragma HLS INTERFACE ap_none port=in_binary
#pragma HLS INTERFACE ap_none port=packed_bcd
#pragma HLS INTERFACE ap_ctrl_none port=return

    uint32 scratch_pad = 0;
    scratch_pad(31, 16) = packed_bcd;

    scratch_pad = reverse_double_dabble(scratch_pad);
    scratch_pad = reverse_double_dabble(scratch_pad);
    scratch_pad = reverse_double_dabble(scratch_pad);
    scratch_pad = reverse_double_dabble(scratch_pad);

    scratch_pad = reverse_double_dabble(scratch_pad);
    scratch_pad = reverse_double_dabble(scratch_pad);
    scratch_pad = reverse_double_dabble(scratch_pad);
    scratch_pad = reverse_double_dabble(scratch_pad);

    scratch_pad = reverse_double_dabble(scratch_pad);
    scratch_pad = reverse_double_dabble(scratch_pad);
    scratch_pad = reverse_double_dabble(scratch_pad);
    scratch_pad = reverse_double_dabble(scratch_pad);

    scratch_pad = reverse_double_dabble(scratch_pad);
    scratch_pad = reverse_double_dabble(scratch_pad);
    scratch_pad = reverse_double_dabble(scratch_pad);
    scratch_pad = scratch_pad >> 1;

    *in_binary   = scratch_pad(15, 0);
}
```

Here, I did these changed to the original double-dabble algorithm

1- Right-shift
2- If a digit is greater than 7, then minus 3
3- Initialise the scratch-pad register with packed-bcd
4- Right-shift
5- Return the lower 16-bits as the integer number.

The complete codes of these implementations are attached to this lecture and can be found in the Resources folder.

Exercise 2

To solve this problem, our design code should implement three tasks:

- Extract digits
- Find 7-segment code
- Display digit

The following function describes these three tasks.

```
void four_digit_hex_7segments(
  ap_uint<16> a,
  ap_uint<4>  push_buttons,
  ap_uint<8> *segment_data,
  ap_uint<4> *segment_ctrl,
  ap_uint<16> *led) {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=a
#pragma HLS INTERFACE ap_none port=push_buttons
#pragma HLS INTERFACE ap_none port=segment_data
#pragma HLS INTERFACE ap_none port=segment_ctrl
#pragma HLS INTERFACE ap_none port=led


  *led = a;
  ap_uint<4> digit_1 = a(3, 0);
  ap_uint<4> digit_2 = a(7, 4);
  ap_uint<4> digit_3 = a(11, 8);
  ap_uint<4> digit_4 = a(15, 12);

  ap_uint<8> segment_data_0 = seven_segment_digit_code(digit_1);
  ap_uint<8> segment_data_1 = seven_segment_digit_code(digit_2);
  ap_uint<8> segment_data_2 = seven_segment_digit_code(digit_3);
  ap_uint<8> segment_data_3 = seven_segment_digit_code(digit_4);



  if (push_buttons == 0b0001) {
    *segment_data = segment_data_0;
    *segment_ctrl = 0b1110;
  } else if (push_buttons == 0b0010) {
    *segment_data = segment_data_1;
    *segment_ctrl = 0b1101;
  } else if (push_buttons == 0b0100) {
    *segment_data = segment_data_2;
    *segment_ctrl = 0b1011;
  } else if (push_buttons == 0b1000) {
    *segment_data   = segment_data_3;
    *segment_ctrl = 0b0111;
  } else {
    *segment_data   = seven_segment_off;
```

```
42-       *segment_ctrl = 0b1111;
43-     }
44-   }
```

Lines 16 t0 19, extract the hex digits. Note that every 4 bits constitute a hex digit.

Lines 21 to 22 find the 7-segment code corresponding to each digit. These lines call the *seven_segment_digit_code* sub-function that its code is attached to this lecture.

Lines 28 to 43 represent the hex digit on a 7-segment based on the pressed push-buttons.

The following figure shows parts of the high-level synthesis report. As can be seen, the design frequency period constraint is $20\ ns$, and the propagation delay of the design is about $12.562\ ns$. Also, the design utilises 1261 LUTs.

Targ... ...ce: xc7a35t-cpg236-1

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	20.00 ns	12.562 ns	2.50 ns

Latency

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	101	-
FIFO	-	-	-	-	-
Instance	-	-	0	1160	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	-	-
Register	-	-	-	-	-
Total	0	0	0	1261	0
Available	100	90	41600	20800	0
Utilization (%)	0	0	0	6	0

Detail

Interface

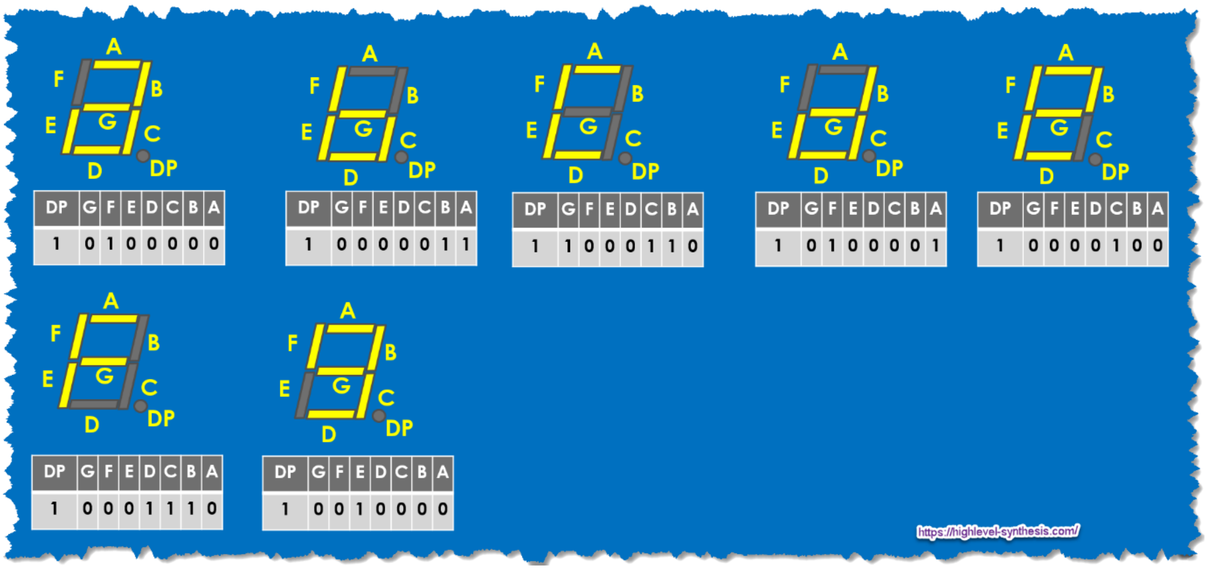
Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
a_V	in	16	ap_none	a_V	scalar
push_buttons_V	in	4	ap_none	push_buttons_V	scalar
segment_data_V	out	8	ap_none	segment_data_V	pointer
segment_ctrl_V	out	4	ap_none	segment_ctrl_V	pointer
led_V	out	16	ap_none	led_V	pointer

Export the report(.html) using the Export Wizard

Exercise 2

The following figure shows the 7-segment codes for letters from **a** to **g**. We can save these codes in an array to be used in our design.



This code snippet shows such an array.

```
const unsigned int seven_segment_code[16] = {
//-----------    code --- letter --- index
             0b11000000,  // 0      //0
             0b10101000,  // a      //1
             0b10000011,  // b      //2
             0b11000110,  // c      //3
             0b10100001,  // d      //4
             0b10000100,  // e      //5
             0b10001110,  // f      //6
             0b10010000,  // g      //7
};
```

As can be seen from the following table, the first three bits and of an ascii code from **a** to **g** can be used as index to extract the corresponding 7-segment code from the above array.

Letter	ASCII Code		
	DEC	HEX	BIN
a	97	61	01100001
b	98	62	01100010
c	99	63	01100011
d	100	64	01100100
e	101	65	01100101
f	102	66	01100110
g	103	67	01100111

Therefore, the following code solve the problem

```
void ascii_on_7segment(
             ap_uint<8> a,
             ap_uint<8> *segment_data,
             ap_uint<4> *segment_ctrl,
             ap_uint<8> *led ) {
#pragma HLS INTERFACE ap_none port=a
#pragma HLS INTERFACE ap_none port=segment_data
#pragma HLS INTERFACE ap_none port=segment_ctrl
#pragma HLS INTERFACE ap_none port=led
#pragma HLS INTERFACE ap_ctrl_none port=return
      *led = a;
      ap_uint<3> index = a(2,0);

      switch(index) {
      case 1:
             *segment_data = seven_segment_code[1];
             break;
      case 2:
             *segment_data = seven_segment_code[2];
             break;
      case 3:
             *segment_data = seven_segment_code[3];
             break;
      case 4:
             *segment_data = seven_segment_code[4];
             break;
      case 5:
             *segment_data = seven_segment_code[5];
             break;
      case 6:
             *segment_data = seven_segment_code[6];
```

```
            break;
        case 7:
            *segment_data = seven_segment_code[7];
            break;
        default:
            *segment_data = seven_segment_code[0];
            break;
    }

    *segment_ctrl = 0b1110;
}
```

The following code shows parts of the high-level synthesis report.

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	1.449 ns	1.25 ns

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	44	-
Register	-	-	-	-	-
Total	0	0	0	44	0
Available	100	90	41600	20800	0
Utilization (%)	0	0	0	~0	0

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
a_V	in	8	ap_none	a_V	scalar
segment_data_V	out	8	ap_none	segment_data_V	pointer
segment_ctrl_V	out	4	ap_none	segment_ctrl_V	pointer
led_V	out	8	ap_none	led_V	pointer