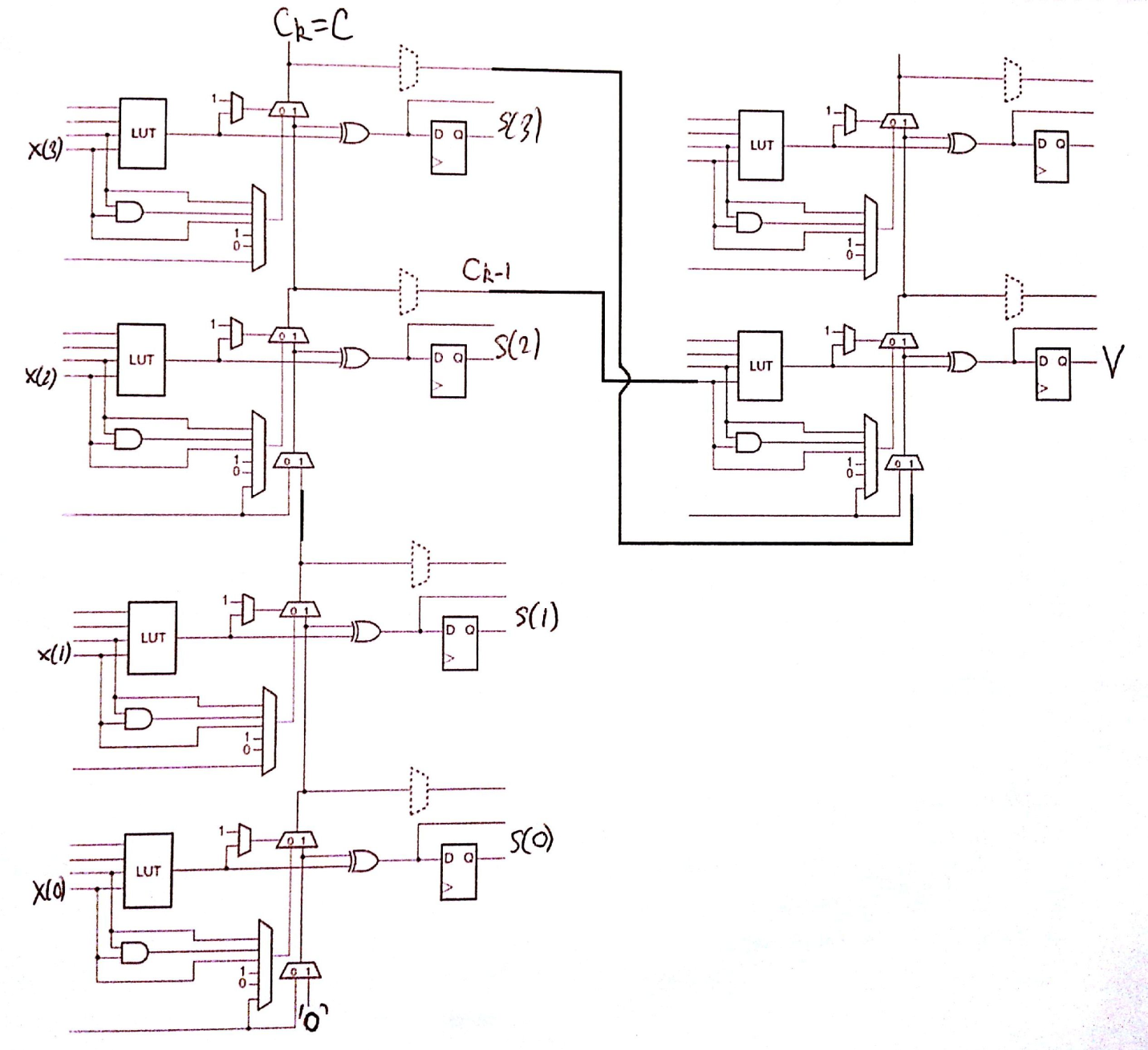
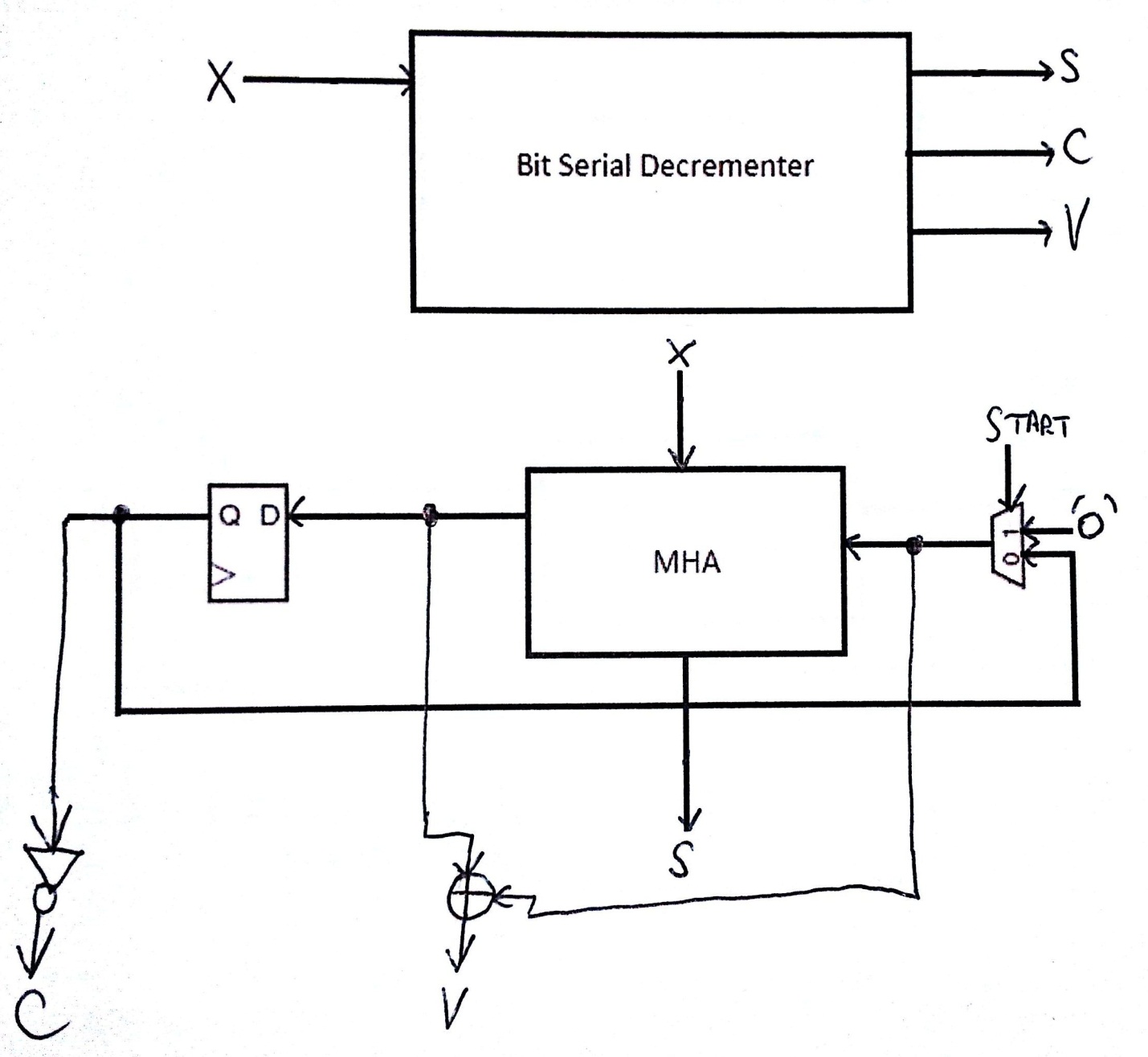
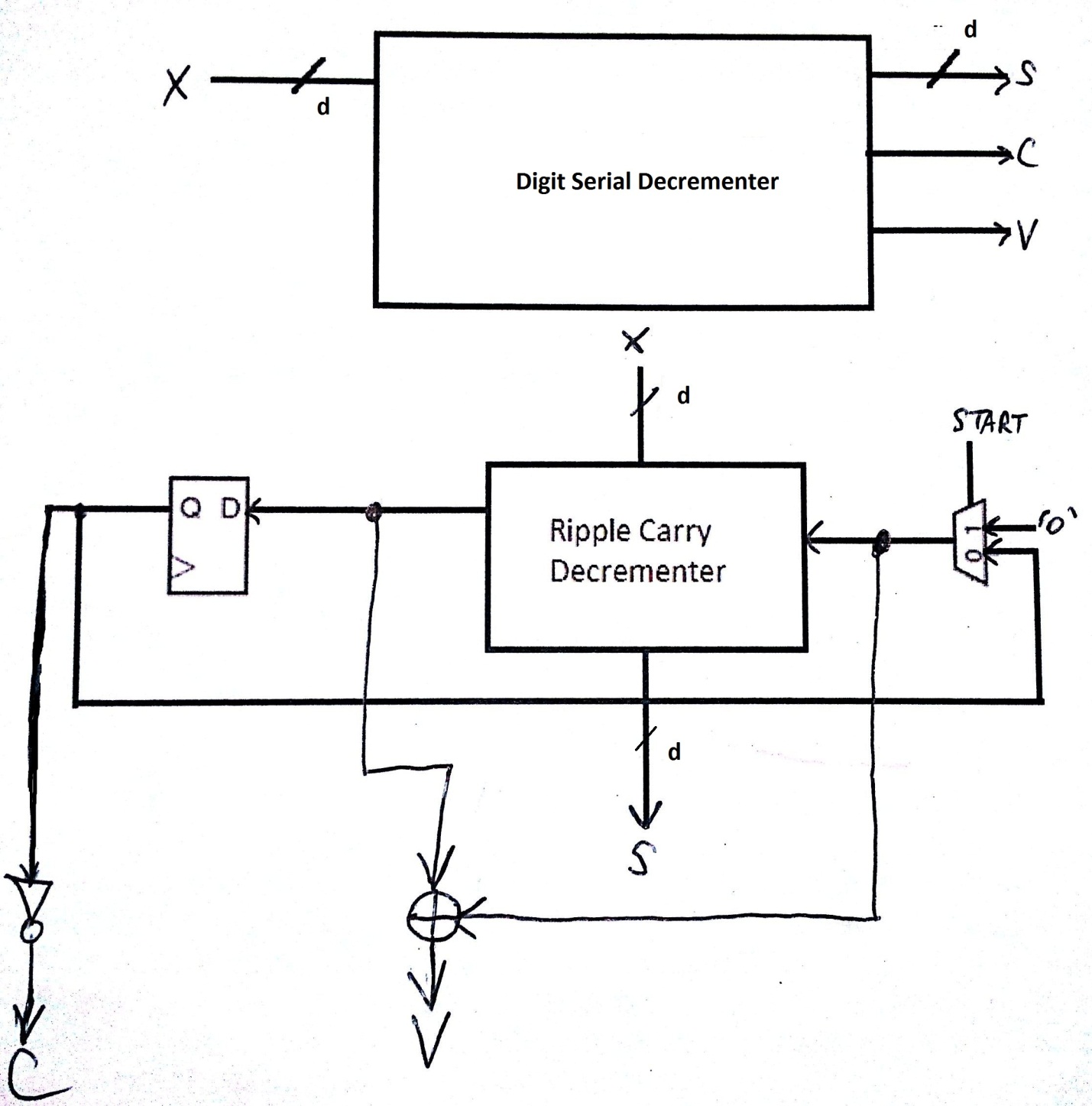
**Problem 1  
Task 1  
Below is a modified half adder table, along with a propagate table (Prop). We consider Y to be our input, and X to be CIN. Y passes through a LUT, which is the “Prop” table, which acts as an inverter. After this, it passes through an XOR with the CIN to produce its respective S bit.  
To pass Cout, we use the Prop signal for the mux select signal.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **X** | **Y** | **Cout** | **S** | **Prop.** |
| **0** | **0** | **0** | **1** | **1** |
| **0** | **1** | **1** | **0** | **0** |
| **1** | **0** | **1** | **0** | **1** |
| **1** | **1** | **1** | **1** | **0** |

**Each of the first four LUTs must be an inverter. In other words, we read in a 0 from Y, the output is a 1, and vice versa. This signal then propagates to the XOR, where we take CIN and XOR this with the signal “Prop.” The result is the respective sum bit for that logic cell. After we have Cout, the final carry output (Ck), we must then XOR this with Ck-1 in order to determine whether or not there has been an overflow. So the LUT for this cell will read in Cout from the “YB” multiplexer as input and pass a 0 if it’s a 0, and 1 if it’s a 1. From here, Cout (Ck) will be XORed with Ck-1 to determine V.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Task 2  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Task 3  
  
  
  
  
  
  
  
  
  
Task 4[Code Attached]  
  
Task 5**RCD:

|  |  |  |
| --- | --- | --- |
| k | #LUTs | #Slices |
| 4 | 5 | 3 |
| 8 | 11 | 6 |
| 16 | 18 | 10 |
| 32 | 34 | 18 |

#LUTs = (29/28\*)d + 1  
#Slices = (15/28)\*d + 1  
BSD:  
We are taking one bit and processing it at once clock cycle for every bit from the total input.  
The number of LUTs and CLB slices required for this circuit will always be the same, and thus:  
#LUTs = 2  
#Slices = 1  
  
DSD:  
Only changing the size of the input “digit” d will modify the hardware. Otherwise, it just takes longer to process. Below is a table of a few values tested:

|  |  |  |  |
| --- | --- | --- | --- |
| k | d | #LUTs | #Slices |
| 256 | 4 | 5 | 3 |
| 256 | 8 | 11 | 6 |
| 256 | 16 | 18 | 10 |
| 256 | 32 | 34 | 18 |
| 512 | 4 | 5 | 3 |
| 512 | 8 | 11 | 6 |
| 512 | 16 | 18 | 10 |
| 512 | 32 | 34 | 18 |

A rough linear approximation of the formulas are as follows:  
#LUTs = (29/28\*)d + 1  
#Slices = (15/28)\*d + 1

**Task 6**RCD: k=256, w=8  
With an 8 to 256 SIPO at the input and a 256 to 8 PISO at the output S, we will have the following, provided with registers at each one bit output:  
TCLKMIN = 10.6  
Minimum Latency: (dSIPO + dPISO)\*(dFF + 4\*dMHA)\* 10.6 = 2.9us  
ns  
Area: 15 slices  
Product Latency: 43.7us\*slices  
BSD: k=256

TCLKMIN = 2.4 ns  
(dFF + 2\*dFF + dMHA + dmux)\*256\*2.4 ns =  
Minimum Latency: 1.78us  
Area: 20 slices  
Product Latency: 35.6 us\*slices

DSD: k=256, d=4  
TCLKMIN = 18.74ns

(2\*dFF + 4\*dMHA)\*64\*18.74 =  
Minimum Latency: 5.51us  
Area: 16 slices  
Product Latency: 88.16 us\* slices

DSD: k=256, d=8  
TCLKMIN = 18.74ns  
(2\*dFF + 8\*dMHA)\*32\*18.74 =

Minimum Latency: 5.2us  
Area: 20 slices  
Product Latency: 104 us\* slices

DSD: k=256, d=16  
TCLKMIN = 18.74ns

(2\*dFF + 16\*dMHA)\*16\*18.74 =

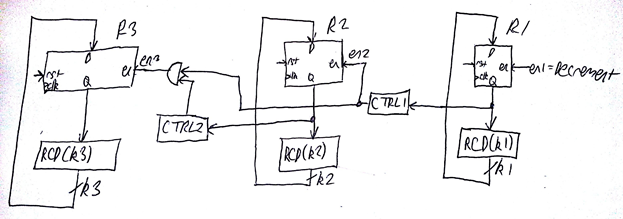
Minimum Latency: 4.977us  
Area: 24 slices  
Product Latency: 119.5 us\* slices

DSD: k=256, d=32  
TCLKMIN =18.74ns

(2\*dFF + 32\*dMHA)\*8\*18.74 =

Minimum Latency: 4.798 us  
Area: 31 slices  
Product Latency: 148.7 us\* slices

Best in terms of area: RCD  
Best in terms of Latency: BSD  
Best in terms of product Latency \* Area: DSD

**Problem 2  
  
CTRL1 and CTRL2 are composed of LUTS. The output will be a 1 whenever the inputs are all zeroes, and otherwise the output will be a 0. When CTRL2’s output AND CTRL1’s output are both ones, then R3 will be enabled.  
  
TCLK >= dFF + dRCD(k1) + tsetup 🡪  
TCLK >= 0.5 + 0.1k1 + 1.7 🡪  
Setting k1 = 4,  
TCLKMIN = 2.6 ns  
  
dFF + tsetup + dRCD(k2) <= 24 \* TCLK  
0.5 + 0.1k2 + 1.7 <= 16\*3  
0.1k2 <= 16\*3 – 2.2  
k2 <= 458  
We already know that the maximum number of bits is 256, so we will have less than this by far for k2. Let’s assume k2 = 16 for now, and test if it works later.  
  
dFF + dRCD(k3) + tsetup <= 2k1+k2\* TCLK  
0.5 + 0.1k3 + 1.7 <= 220 \* TCLK  
k3 = 236 if we select k1 = 4, k2 = 16.  
0.5 + 23.6 + 1.7 <= 220 \* 2.6  
25.8 <= 220 \* 2.6, which is clearly true, but we still need to factor in the other conditions:  
dFF + dCTRL1 + tsetup <= TCLK  
0.5 + 1 <= 2.6?  
Yes, so we test;  
dFF + dCTRL2 + dAND + tsetup <= TCLK  
0.5 + 1 + 2 <= TCLK, where we assume an AND gate has a delay of 1 ns, and our control logic requires 2 ns for the four LUTs running in parallel as well as the final LUT to process.  
3.5 > TCLK, so let’s just use one LUT with 4 bits input and assume k2 = 4.  
 dFF + dRCD(k3) + tsetup <= 2k1+k2\* TCLK  
0.5 + 24.8 + 1.7 <= 28 \* TCLK  
 27 <= 256 \* 2.6  
Now we test again under the conditions.  
Since we know that the control logic did not delay too much for CTRL1, we can try now with CTRL2 being just 1 LUT:  
dFF + dCTRL2 + dAND + tsetup <= TCLK  
0.5 + 1 + 1 <= TCLK  
2.5 <= 2.6, so our minimum acceptable TCLK is 2.5ns, and for k1, k2, and k3, I have selected the following values: 4, 4, and 256-8 = 248.  
  
CTRL1 and CTRL2 now just have 1 LUT inside, and both LUTs will only have ‘1’ as output if the input is “0000”, as we want to reach the value of all 0’s before we carry over to all 1’s again for the lower bits.**