Lab 4: Count ‘em up

Gavin Chen
CE100 UCSC
5/9/18

Description:

The purpose of this lab was to familiarize the logic behind flip-flop state storage circuits to implement a binary counter. The main part of the lab was to implement a 16-bit counter by chaining several smaller bit counters together. In creating the logic for the counters we gained insight on combinational logic for computation, and how it can be used to change the behavior of a circuit based on its current state.

Methods:

3-bit counter:

We began the lab by creating a 3-bit counter with functionality to both increment as well as decrement. The entire module utilized 3 flip flops to hold the current bit state, with input pins to control increment, decrement, or the ability to load in a bit value. The 2 truth tables shown to the right illustrate the change in state given the current state and an increment or decrement event. Given the 2 truth tables, karnaugh maps were then created to implement the logic as feed-in to the respective flip flop. The following logic statements were derived, with Dn being the input and Qn the output of flip flop *n*. Two equations were obtained per flip flop which represented combinational logic input. To ensure the flip flop acknowledged the change only on a respective up or down event, the logic was ANDed with its respective event pin, and the flip flop was enabled on the OR of all three input pins.

Current state			Next state increment			Next state decement		
Q0	Q1	Q2	Q0	Q1	Q2	Q0	Q1	Q2
0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	0	0	1
0	1	0	0	1	0	0	1	0
0	1	1	0	1	1	0	1	1
1	0	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0	1
1	1	0	1	1	0	1	1	0
1	1	1	1	1	1	1	1	1

Flip flop	Increment	Decrement
D0	$-Q0$	$-Q0$
D1	$-Q1{*}Q0 + -Q0{*}Q1$	$-Q1{*} - Q0 + Q1{*}Q0$
D2	$Q2(-Q0{*}Q1 + -Q1) + -Q2{*}Q1{*}Q0$	$Q2(-Q0{*}Q1 + Q0) + -Q2{*} - Q1{*} - Q0$

Implementing the load in values was similar, only instead of combinational logic as a result of the current state, input was taken directly from a bus of 3 bits that was fed as input. This corresponded to a 3 bit number that was ANDed with the load enable bit. The flip flops were also sent into the enable state on the event of a high on the load bit. The result was that all logic was combined into a single input, and the correct change only activated when the bit mask (i.e. its input enable pin) was driven high.

The next piece was to make the counter modular through an over/underflow bit, allowing it to be chained with other counters in the 16 bit counter. The overflow bit simply output a high on the event that all current bits were set high AND there was an event to increment the counter. In this case the counter would roll over to become all zeros and the up count event would signify an event to increment the subsequent counter. The dual of this same principle was implemented for underflow, with an event being all current bits set to zero AND a down count event. Overall the module took in 5 inputs: 3 for a counter event (up, down, load), a 3 bit bus for loadable bit numbers, and a clock to synchronize all flip flops. It output a 3 bit bus corresponding to the current count, and 2 event counters for overflow or underflow when counting.

5-bit counter:

A 5-bit counter was then implemented in the same fashion, with the only difference being the way in which the logic was computed. As creating the input logic for a 5 bit counter involved creating at some point a 5 input K-map, this was deemed too difficult. Instead a formula was used to calculate for which inputs a given flip flop would be 1 for a given event. It was noticed that in the event of an up count event, a given flip flop *n*

remained 1 if there was at least one zero in all preceding flip flops, or switched to 1 if it was currently a 0 and all previous flip flops were 1. In either case the feed in for Dn given an up count event should be 1. The same idea was applied to the decrement logic, only the bits were inverted and changes to a 0 state were highlighted. This reduced to the formulas:

$$\text{Up count:} \quad -Q_n{*}((Q_{n-1}{:}\,Q_0)) + Q_n{*}\{(Q_{n-1}{:}\,Q_0)\}$$
$$\text{Down count:} \quad Q_n{*}(((Q_{n-1}{:}\,Q_0)) + -\{|(Q_{n-1}{:}\,Q_0)\}$$

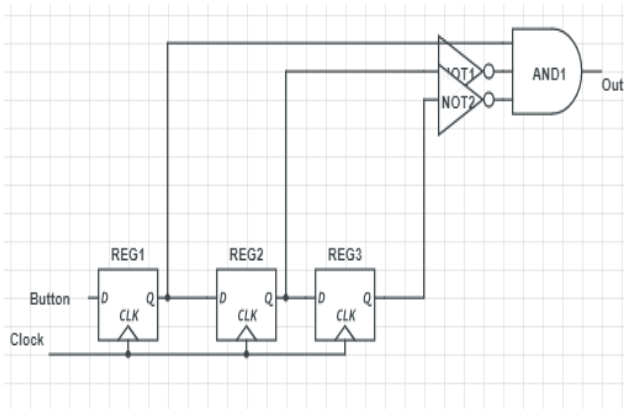
Where the & or | represented a folding of all bits into a single bit using the AND or OR operation respectively, and bits in a range (a,b). This logic was fed into the *n* flip flop in the same fashion as the 3-bit counter logic, with enable pins wired the same way. The overflow/underflow lines were calculated the same way as before, and the resulting module was similar to the 3-bit counter with the only difference being the combinational logic and the width of the output bus.

16-bit counter:

The 16-bit counter was created simply as a module that coordinated the combination of the smaller bit counters. It consisted of two 5-bit counters and two 3-bit counters. The 3-bit counters were responsible for the upmost and lowermost bits, and the 5-bit counters the middle 10. Input was taken as up, down, and load events, along with a 16-bit bus from the onboard switches for loading input. The output was a 16 bit number with over/underflow support as well. The input switches were fed into the counter modules responsible for the respective bits. The up/down count event for the 16-bit module was fed in as input to the lowermost 3-bit counter only, with all overflow/underflow count cascading into the subsequent counter. The final overall overflow/underflow counter event for the 16-bit counter was taken as the AND of all bits, and the inverse of the OR of all bits respectively. The flow output was slightly different from the individual counters in that it was independent of the up/down input event to the 16-bit counter.

Edge detector:

The edge detector took in the state of a button as input, and output a high signal for a single clock cycle on a state change of the button. The detector was implemented using 3 flip flops wired sequentially, with the output of one becoming the input of another. The button state input was fed into the first flip flop, and the output was chained to another, which was then chained to another. The output lines of all flip flops were tied to an AND gate. The 2 flip flops not connected to the button input were inverted, and the remaining initial flip flop was left untouched. The output of the AND gate became the module output. A schematic of the design is shown to the right.



Ring counter:

The ring counter was a simple bit rotator, and was implemented in the same fashion as the edge detector. As it consisted of 4 bits, it was implemented with 4 flip flops all wired sequentially. The initial flip flop was initialized to a value of 1, and each flip flop was wired into the input of the next. The last flip flop in the line was tied back into the beginning. The module took in as input the clock and an advance signal that shifted the bit on high, and output a 4-bit bus with only one bit set.

Selector:

A module to select 4-bit sections of a 16-bit input was implemented using a bitmask dependent on a 4-bit input. The 4-bit input from the ring counter had guaranteed one bit high at one time, with each bit corresponding to a 4-bit section in the 16-bit number. Each bit from the 4-bit bus was extended to a 4-bit bus itself. Each extended bus was then ANDed with sections of the 16-bit input. The table with the bit and its corresponding bit field is presented below.

Ring counter bit position	[0]	[1]	[2]	[3]
16-bit field	3:0	4:7	8:11	12:15

Top:

The top module combined each individual module described, as well as a provided clock module and a previously implemented 7-segment display converter, to create a counter which incremented and decremented based on the input from buttons on the board. There were 5 inputs into the module from the hardware: 4 buttons up, down, left, and center; and the 16-bit switch. The up and down buttons were input into edge detector modules, with the output piped as respective up and down events into the 16-bit counter. The center button was first combined with logic before being fed into the 16-bit up event: bits 2:15 from the output were ANDed together, and the result was inverted then ANDed with the center button. The left button and 16 switch bus was fed into the load and load input ports of the 16-bit counter, respectively.

The 16-bit output of the counter was fed into the 16-bit input of the selector. The ring counter was then connected to the dig_sel output of the clock module, and the 4-bit rotator ring was set as input into the selector. The selector chose a 4-bit field of the selector based on the ring counter, and fed the field into the 7-segment converter to create an LED pattern for the display. The selector was connected to the segment cathode connection, and the anode bus was set as the inverse of the ring counter output. This created an updating LED display that output the 16-bit counter as 4 hex digits. The output of the counter was also piped into the onboard LED's in a direct one-to-one fashion.

The overflow/underflow of the 16-bit counter was synchronized with the ring counter to illuminate either the leftmost or rightmost decimal point when highs, respectively. The overflow bit was ANDed with the most significant bit of the ring counter, and the underflow the lowest. The two outputs were then ORed together and fed into the decimal point cathode.

Results:

3-bit counter:

The 3-bit counter worked as expected, after some bug fixing. There were some syntax issues with the initial logic in calculating high values for increment. After correction of the logic, it was observed that the input logic to the lowest bit of the counter, in both increments and decrements, is simply an inversion of the bit. This corresponds to intuition for the bit in the ones place of a counter, with events simply being a toggle of that bit. Looking at the logic for the 2^{nd} and 3^{rd} bits of the counter, it can also be seen that the logic implemented for the 5-bit counter is also mirrored in the 3-bit counter. This relation was found during creation of the 5-bit logic detailed earlier, but after the 3-bit logic.

It was also noticed that synchronizing the clock cycles of all flip-flops together and changing only on edges, we could rely on our combinational logic outputting a stable value before being set by the flip flop. If inputs/outputs changed ad hoc as it is for normal gates, different flops would hold inconsistent values at different time, making design impossible. Since all flip flops changed at the same instant, anomaly cases such as number jumping because of constantly changing input values, doesn't happen.

5-bit counter:

The 5-bit counter worked as expected after significant debugging, counting from the full range of 0 to 31 with increments taking a full clock cycle to propagate. It was beneficial beginning with the 3-bit counter and creating K-maps to implement the logic first before moving onto the 5-bit counter. By first seeing the logic implemented with a standard formula, deriving a general formula for any *n* flip flop was easier to visualize as well as verify. The logic for the 5-bit counter became more intuitive after the fact. For increment, a flop stayed 1 if there was a previous 0 to "absorb" the increment, or switched to 1 in the event of a roll over. For decrement, a flop stayed 0 if there was a previous 1 to "absorb" the subtraction, or switched to 0 if it was carried out from the subtraction. Again, synchronizing flip flops maintained consistency when switching between states.

16-bit counter:

The 16-bit counter successfully implemented the combination of the all n-bit counters. By feeding in the overflow/underflow outputs of a previous counter, it effectively became an event for the current counter. Therefore, each subsequent counter after first 3-bit could be thought of as a counter of overflow/underflow events from the previous counter. By having the up/down count event only toggle the lowermost bit of the lowermost counter as opposed to all up/down events, it simulated the desired event of at most a single value increment per clock cycle. The only observable difference between the 16-bit counter and the n-bit counters was

the behavior of the flow output. Instead of the flow event remaining high for only one clock cycle, the flow counter remained high until another input event. Observe that for the n-bit counters an overflow/underflow event exists for at most a single clock cycle, and is immediately followed by a reset of all bits to either all 1's or all 0's. This is not the case for the 16-bit counter.

Edge detector:

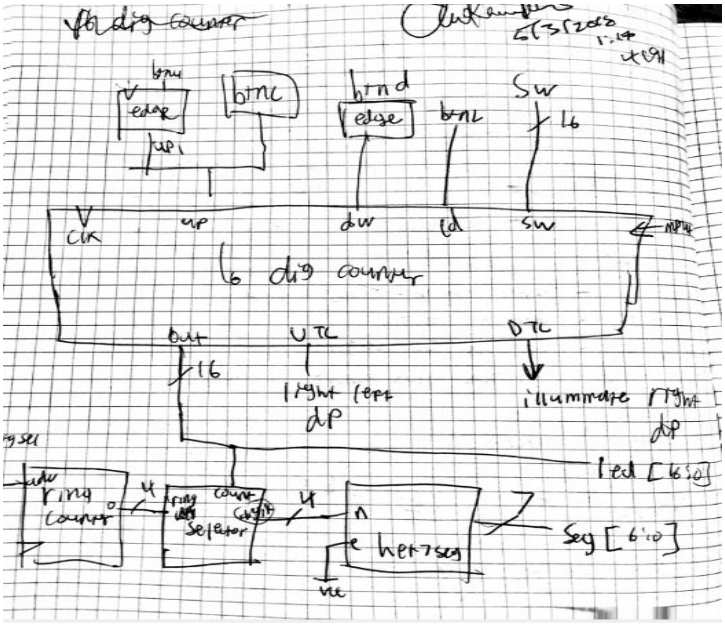
By implementing the edge detector to filter button input, it ensured itempotency for button events. Observe that the edge detector only outputs high for the specific sequence of input 100. Also note that the button state propagates through the edge detector one clock cycle at a time. Since the detector only outputs high for a sequence of inputs that exists for only one clock cycle, it registers essentially only the event the button changes from a 0 to a 1. Observe also that since the sequence is on for a single 1 followed by two 0's, the edge detector also filters out oscillations in the connection when the button is pressed. Care must be taken in choosing the correct sample rate of the button to produce the desired effect. However that exact balance of clock time and input sequence to filter out the bounce was not found in this lab, but rather was chosen by the great overlord Martine. It is noted that the edge detector could have been implemented using only 2 flip flops instead of 3, where the output is a combination of the flip flop outputs and the button input as opposed to 3 flip flops.

Ring counter:

Implementing the ring counter proved to be more challenging than first assumed. Due to initialization errors with the Verilog code, the ring counter logic was implemented correctly however with no initial 1 value to be shifted. Therefore, the output of the counter remained 0 until the error was found. Since the output of the ring counter controlled the refresh rate of the segment display, careful attention needed to be paid to select a goldilocks advance signal to prevent flickering or uneven refresh rate. If the refresh rate were too slow, noticeable flickering in the lights as they turned off for clock cycles would have been observed. In the case of the clock cycle being set too fast there was an effect of an afterglow on the display. We were able to force this error by increasing the overall clock rate of the top module. When we did so, we observed an afterglow of the segment displays, as the segment immediately on the left was ghosted onto the display to its right, with the rightmost segment wrapping around and ghosting onto the leftmost segment. This ghosting affect is in line with the rotation direction of the ring counter, and was attributed to the output of the selector and segment display converter not being able to synchronize and stabilize its output before being output on the display.

Selector:

The bit field selector worked as advertised. By extending each input bit from the ring counter to become 4-bits wide, it effectively created a bit mask that either let in or blocked all values of a 4-bit field of the input. Since it was guaranteed by the ring counter that only one bit would be activated per cycle, the logic for the selector did not need to account for isolating singular bit field of the output. A benefit of implementing a ring counter and selector was that only one segment converter was needed to display 4 numbers. Since the converter implemented several mux's to create the logic, using only one and creating a selector and ring counter which had minimal logic significantly reduced the overall gate count of the design.



Top:

A block diagram of the top module is shown to the left. The top diagram was responsible for putting together all of the base modules into working hardware, and interfacing with the board itself. The main section of the module was the 16-bit counter. All other modules, including the edge detectors, ring counter, selector, and segment converter all helped to facilitate and show the logic being implemented with the counter "under the hood". In the end the top module worked as hoped, however with some testing necessary. It was found that first testing the individual components before implementing them in the top module was a better design approach. After testing the parts, simulation of the top module focused mostly on correct connection of component input/output into each other. The only logic needed to be implemented in the top module was disabling the center button when the counter reached a range between 0xFFFC and 0xFFFF. To do this it was noticed that only in this range of binary numbers, the top 14 bits all corresponded to a high value. Therefore, by ANDing all top 14 bits together the result would only be high in the desired range. Therefore inverting it and ANDing it with the input from the center button essentially disabled it during that range.

A similar logic for the decimal points was implemented, wherein the cathode that controlled it was activated only in the cases where the overflow bit was high AND the ring counter was in the most significant bit, OR when the underflow bit was high AND the ring counter was in the least significant bit. This switched on the decimal point for cases where the 16-bit counter was at a maximum or minimum and ready to turn over, and the ring counter was at the correct anode we wanted to display. In testing the physical functionality of the overall program, we checked the reset condition i.e. when the right button was pressed. When this was done it was observed that the counter value was reset to 0, and the output on the segment display showed only the rightmost digit. The counter value was because the initial condition for all flip flops used in the counters was reset to 0. The display was observed to be high only on the right most digit because the flip flops used for the ring counter only initialized the rightmost flop to a value of 1. Therefore, when the flip flops were in reset state only the rightmost flop output high, and so only the segment associated with that bit was activated.

Testing and simulation:

As discussed before, it was found to be easier to first test individual components before implementing them in the top module. We began with the 3-bit counter. There were several cases we needed to cover to ensure full functionality of the module: increment, decrement, load value, rollover, and finally roll under. If the device was able to count normally, load in a specified value when needed, and reset to the top or bottom value while outputting the correct flow value for one clock cycle, it was functional and able to be implemented in the top module.

We next tested the 5-bit module in the same fashion. We found some bugs in the logic on the increment and decrement. However, through testing it was clear to see the error was on the roll over and it was quickly fixed and adjusted.

We then moved onto the 16-bit module. In this case too we tested the basic functionality. However extra attention was paid to the flow counter. It was made sure that the flow counter output a high value for the duration that the output was all high or all low, as opposed to just a single clock cycle.

Next we tested the edge detector. We tested that for an input that began as 0 and switched to 1, the detector output only a single high value for one clock cycle. We also checked in the case that the input button state oscillated between on and off multiple times, with a period of less than 3 clock cycles. This was to ensure the detector would filter out any unwanted button bounce.

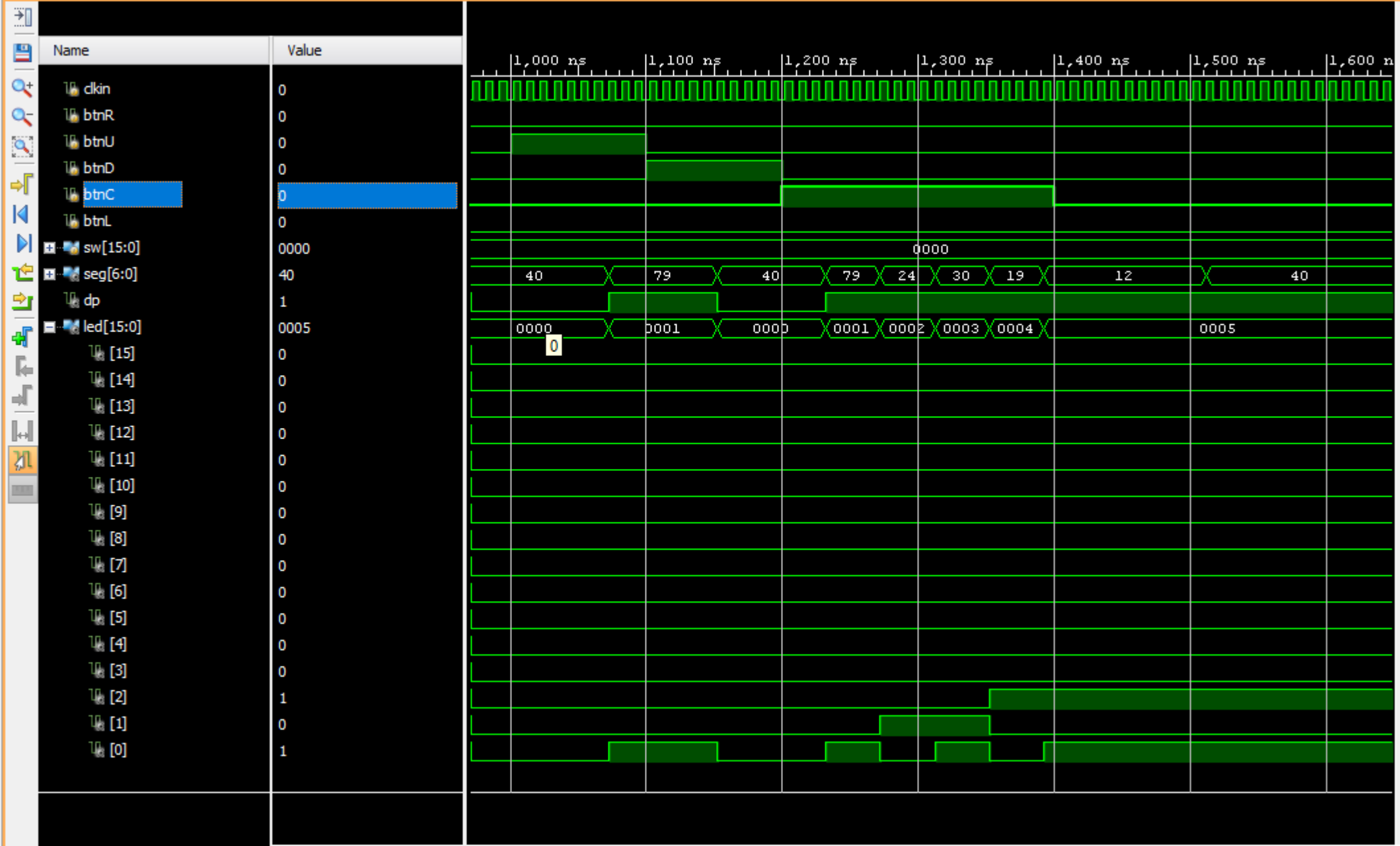
Finally, the ring counter and 4-bit selector were tested in conjunction to make sure they rotated a bit on the clock cycle. We input a 16-bit value which broke down into unique 4-bit values, and checked that the selector output the correct value on sequential clock cycles.

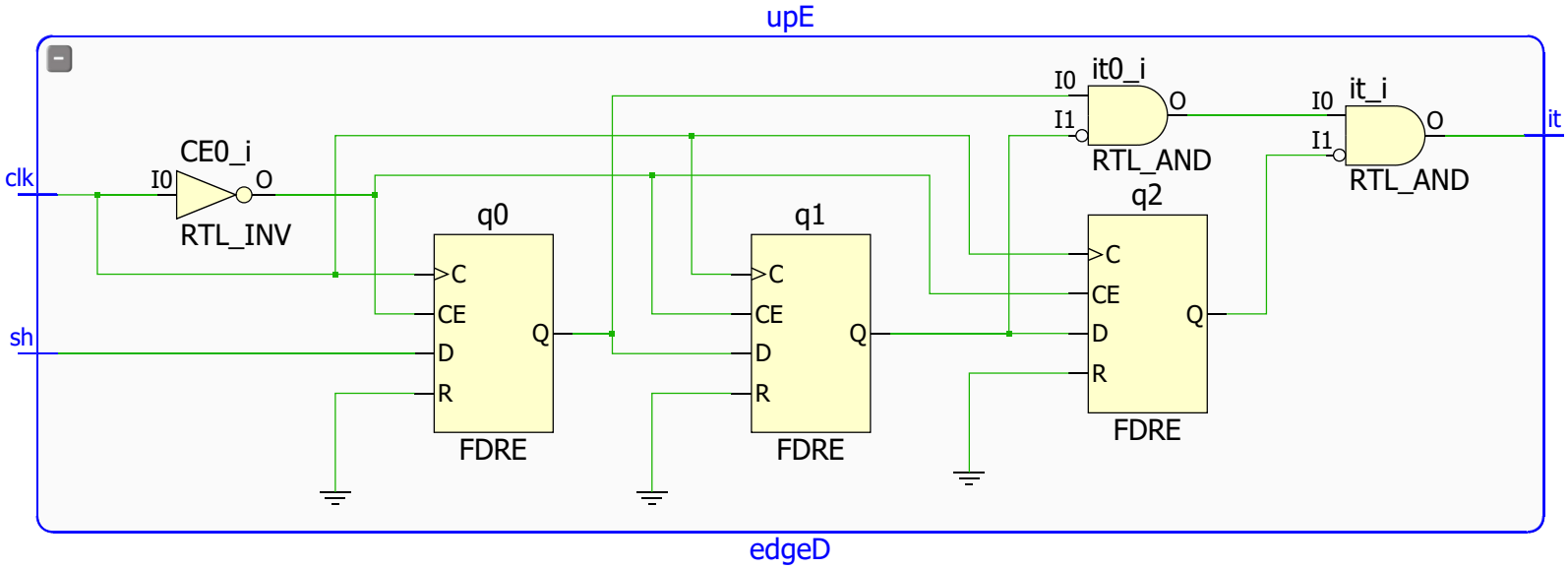
Conclusion:

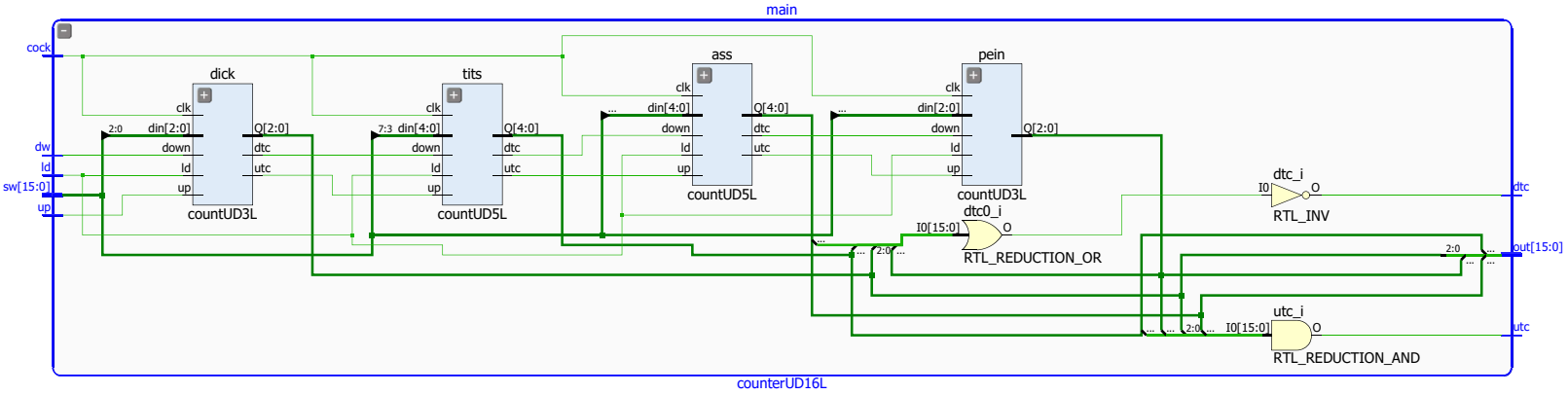
The lab was fun, and it was interesting to implement a working counter. We learned how to use flip flops to store states, and how synchronous programming ensured stable and steady evolution of a circuit. The most difficult part was creating the logic for the n-bit counters. Working with them for the first time it was difficult wrapping our heads around the idea of constant input being fed into a discrete device. If this lab were done again, the only change would be in the testing. As opposed to creating all of the modules then testing them in the end, creating them one-by-one and testing as we went along would have helped us catch and learn from our mistakes earlier. In terms of optimization, the edge detector and some of the logic for the counters could have been optimized. One less flip flop could have been used for the detector, and the counters could have been implemented with fewer gates. Overall the lab was interesting and we had fun staying in the lab until 4am rushing to make up for our procrastination!

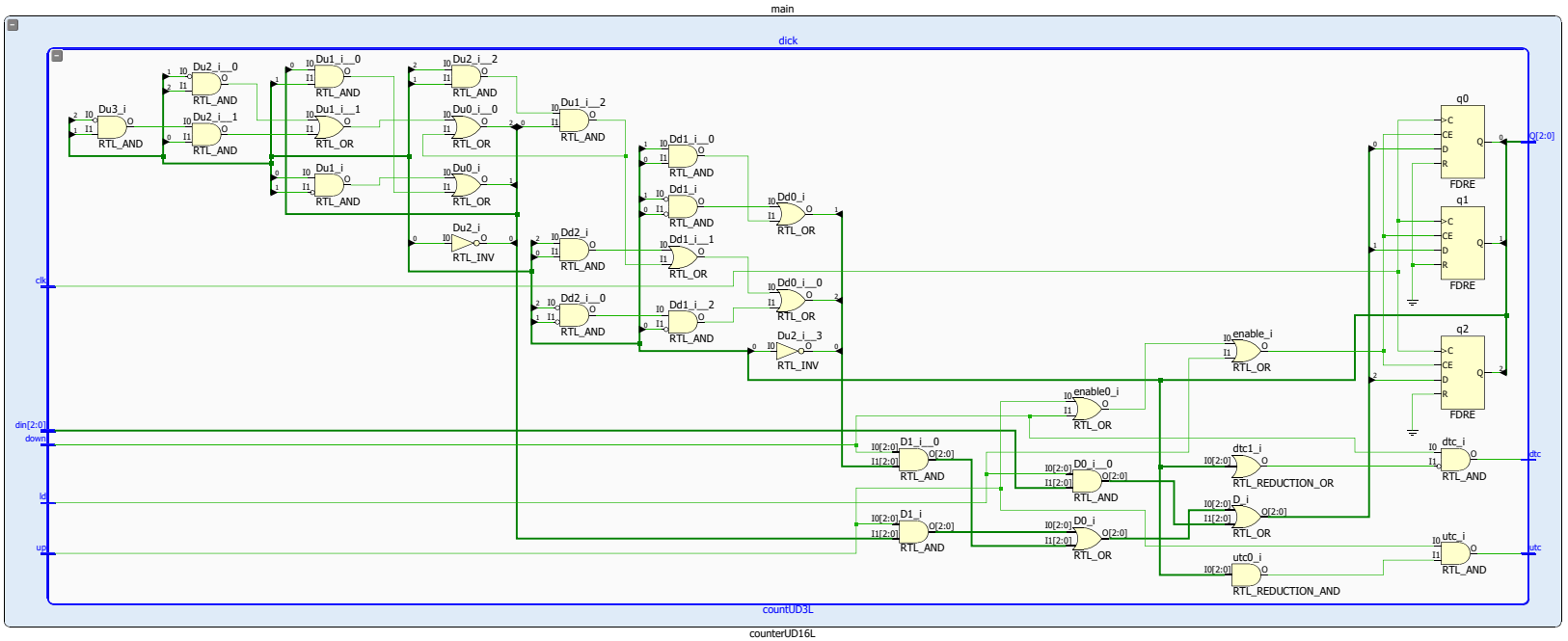
Appendix

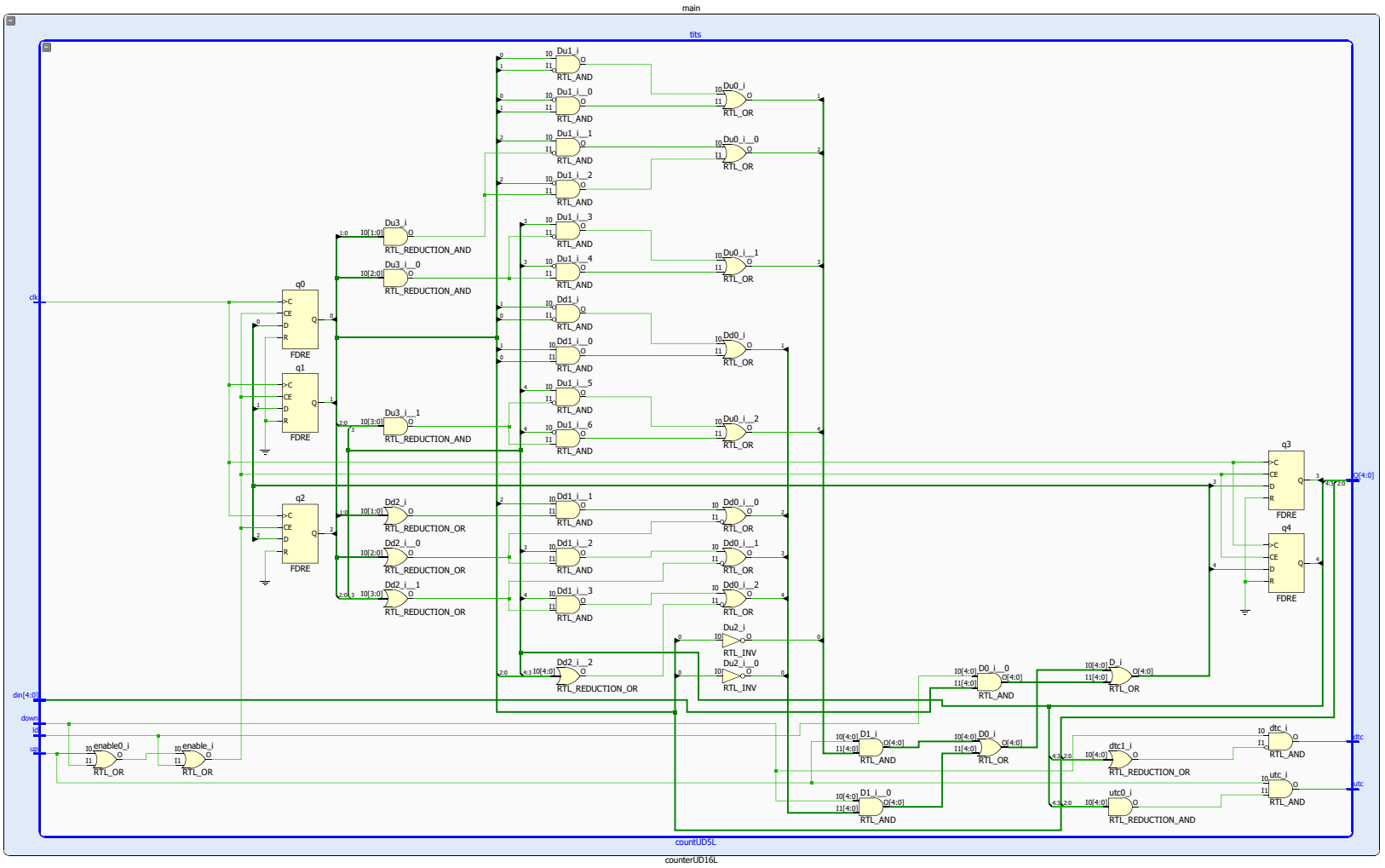

Name
Value
clkin
0
btnR
0
btnC
0
btnU
1
btnD
0
btnL
0
sw[15:0]
9034
seg[6:0]
40
an[3:0]
d
dp
1
UTC
0
TX_ERROR[31:0]
00000003
PERIOD[31:0]
0000000a
DUTY_CYCLE
0.5
OFFSET[31:0]
00000002
2,707.620003 us
2,700 us
2,702 us
2,704 us
2,706 us
2,708 us
2,710 us
9034
46
0e
46
0e
06
0e
40
79
40
79
40
7
9034
d
b
7
e
d
b
7
e
d
b
7
e
d
b
7
e
d
b
00000000
00000
00000003
0000000a
0.5
00000002

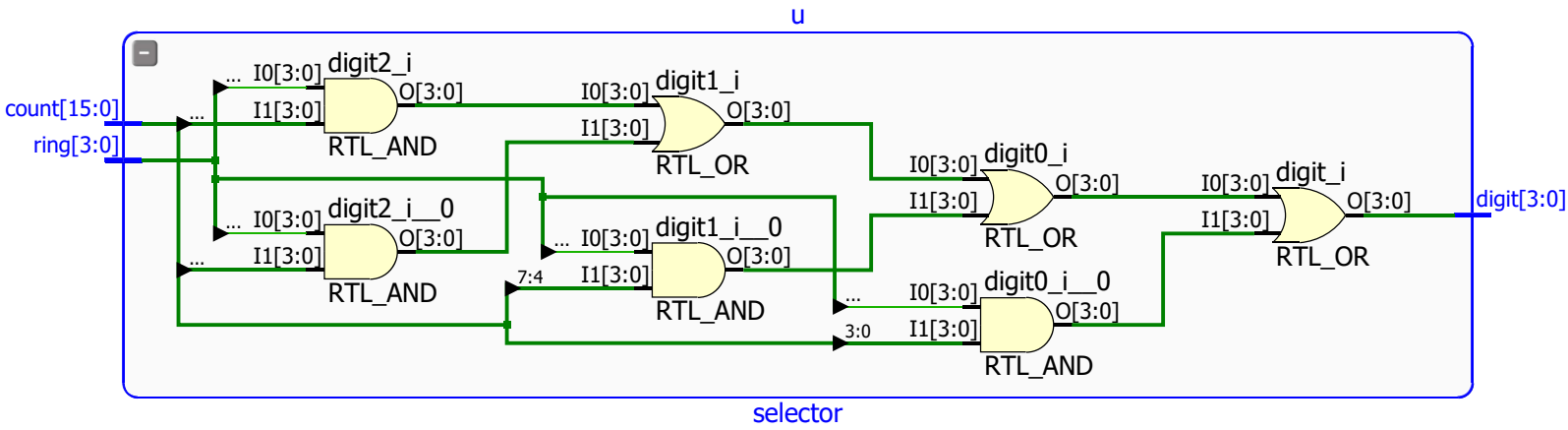

Name
Value
clkin
0
btnR
0
btnU
0
btnD
0
btnC
0
btnL
0
sw[15:0]
0000
seg[6:0]
40
dp
1
led[15:0]
0005
[15]
0
[14]
0
[13]
0
[12]
0
[11]
0
[10]
0
[9]
0
[8]
0
[7]
0
[6]
0
[5]
0
[4]
0
[3]
0
[2]
1
[1]
0
[0]
1
1,000 ns
1,100 ns
1,200 ns
1,300 ns
1,400 ns
1,500 ns
1,600 n
0000
40
79
40
79
24
30
19
12
40
0000
0001
0000
0001
0002
0003
0004
0005
0

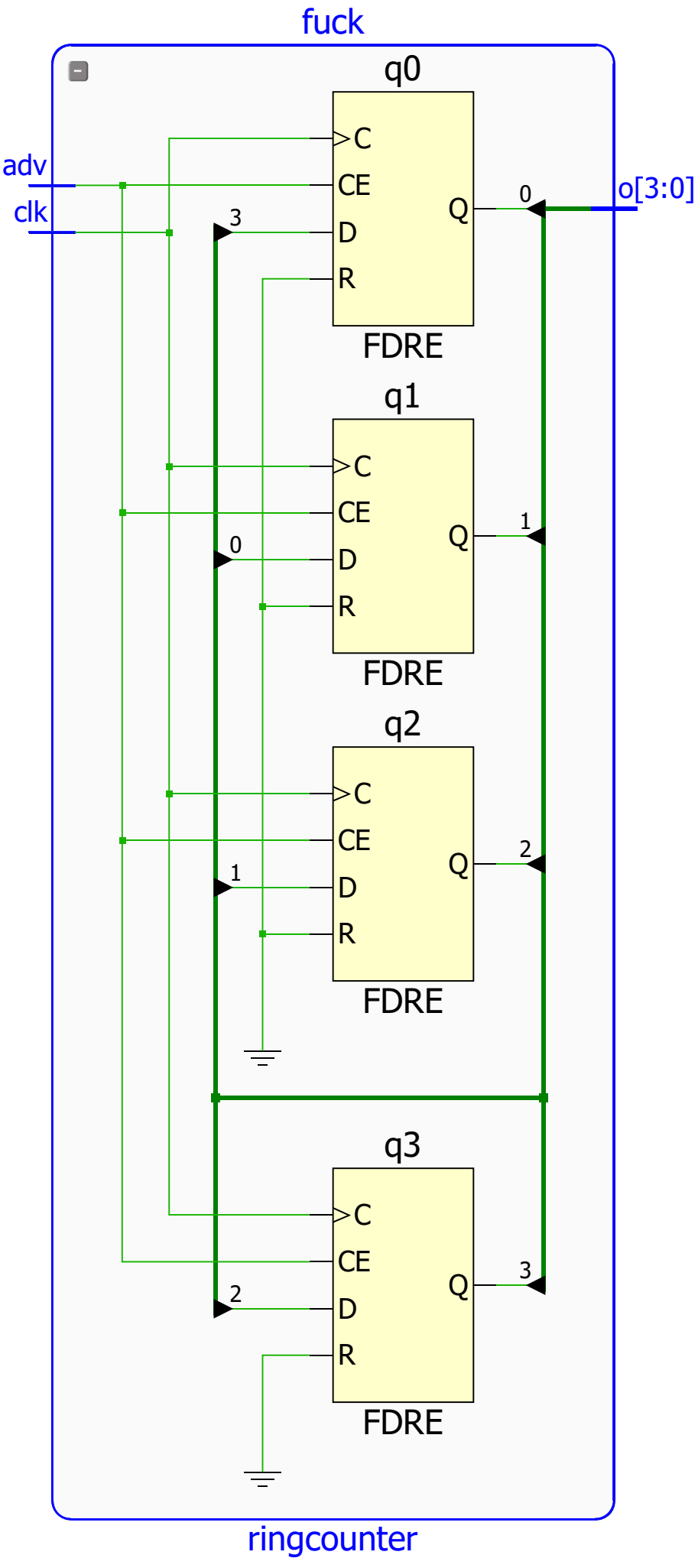

upE
clk
sh
CE0_i
I0
O
RTL_INV
q0
C
CE
D
R
Q
FDRE
q1
C
CE
D
R
Q
FDRE
it0_i
I0
I1
O
RTL_AND
q2
C
CE
D
R
Q
FDRE
it_i
I0
I1
O
RTL_AND
it
edgeD

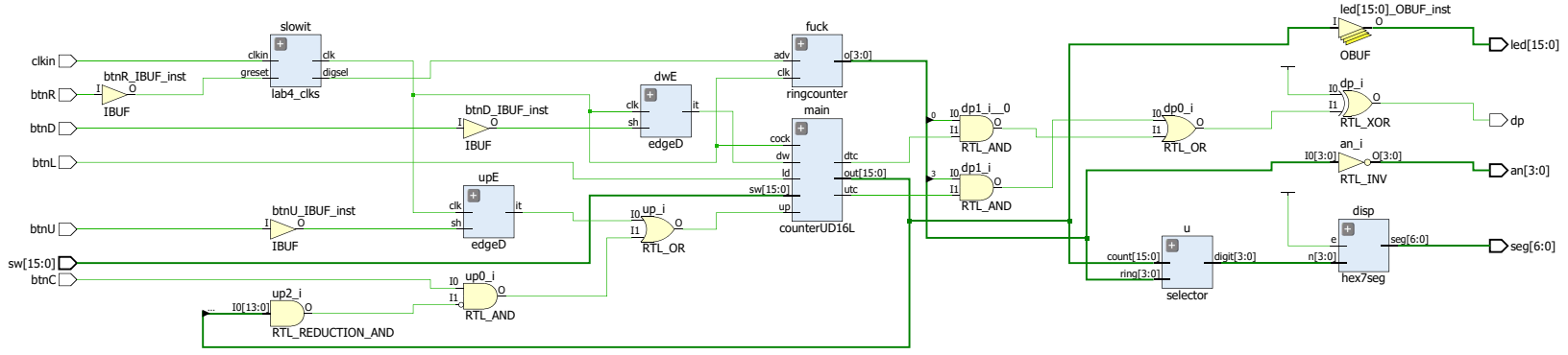

main
cock
dw
ld
sw[15:0]
up
dick
clk
din[2:0]
down
ld
up
Q[2:0]
dtc
utc
countUD3L
tits
din[4:0]
Q[4:0]
countUD5L
ass
pein
dtc_i
I0
O
RTL_INV
dtc0_i
I0[15:0]
RTL_REDUCTION_OR
out[15:0]
utc_i
RTL_REDUCTION_AND
counterUD16L


main
dick
countUD3L
counterUD16L
clk
din[2:0]
down
ce
utc
Q[2:0]
dtc
utc
Du3_i
RTL_AND
Du2_i__0
RTL_AND
Du2_i__1
RTL_AND
Du1_i__0
RTL_AND
Du1_i__1
RTL_OR
Du1_i
RTL_AND
Du2_i__2
RTL_AND
Du0_i__0
RTL_OR
Du0_i
RTL_OR
Du2_i
RTL_INV
Du1_i__2
RTL_AND
Dd2_i
RTL_AND
Dd2_i__0
RTL_AND
Dd1_i__0
RTL_AND
Dd1_i
RTL_AND
Dd1_i__1
RTL_OR
Dd1_i__2
RTL_AND
Dd0_i
RTL_OR
Dd0_i__0
RTL_OR
Du2_i__3
RTL_INV
enable0_i
RTL_OR
enable_i
RTL_OR
D1_i__0
RTL_AND
D1_i
RTL_AND
D0_i__0
RTL_AND
D0_i
RTL_OR
dtc1_i
RTL_REDUCTION_OR
D_i
RTL_OR
utc0_i
RTL_REDUCTION_AND
dtc_i
RTL_AND
utc_i
RTL_AND
q0
FDRE
q1
FDRE
q2
FDRE


main
tits
countUD5L
counterUD16L
clk
din[4:0]
down
ld
up
Q[4:0]
dtc
utc
q0
q1
q2
q3
q4
FDRE
Du1_i
Du1_i_0
Du1_i_1
Du1_i_2
Du1_i_3
Du1_i_4
Du1_i_5
Du1_i_6
Du0_i
Du0_i_0
Du0_i_1
Du0_i_2
Du3_i
Du3_i_0
Du3_i_1
Dd1_i
Dd1_i_0
Dd1_i_1
Dd1_i_2
Dd1_i_3
Dd0_i
Dd0_i_0
Dd0_i_1
Dd0_i_2
Dd2_i
Dd2_i_0
Dd2_i_1
Dd2_i_2
Du2_i
Du2_i_0
D_i
D0_i
D0_i_0
D1_i
D1_i_0
dtc_i
dtc1_i
utc_i
utc0_i
enable_i
enable0_i
RTL_AND
RTL_OR
RTL_INV
RTL_REDUCTION_AND
RTL_REDUCTION_OR


u
count[15:0]
ring[3:0]
digit2_i
I0[3:0]
I1[3:0]
O[3:0]
RTL_AND
digit1_i
RTL_OR
digit2_i__0
digit1_i__0
digit0_i
digit0_i__0
digit_i
digit[3:0]
7:4
3:0
selector


fuck
q0
C
CE
D
R
Q
FDRE
adv
clk
3
0
o[3:0]
q1
C
CE
D
R
Q
FDRE
0
1
q2
C
CE
D
R
Q
FDRE
1
2
q3
C
CE
D
R
Q
FDRE
2
3
ringcounter


clkin
btnR
btnD
btnL
btnU
sw[15:0]
btnC
slowit
clkin
greset
clk
digsel
lab4_clks
btnR_IBUF_inst
IBUF
btnD_IBUF_inst
IBUF
btnU_IBUF_inst
IBUF
dwE
clk
sh
it
edgeD
upE
clk
sh
it
edgeD
up_i
RTL_OR
up0_i
RTL_AND
up2_i
I0[13:0]
RTL_REDUCTION_AND
fuck
adv
clk
o[3:0]
ringcounter
main
cock
dw
ld
sw[15:0]
up
dtc
out[15:0]
utc
counterUD16L
dp1_i__0
RTL_AND
dp1_i
RTL_AND
dp0_i
RTL_OR
u
count[15:0]
ring[3:0]
digit[3:0]
selector
led[15:0]_OBUF_inst
OBUF
led[15:0]
dp_i
RTL_XOR
dp
an_i
O[3:0]
RTL_INV
an[3:0]
disp
e
n[3:0]
seg[6:0]
hex7seg
seg[6:0]

Lab 4

4/25/18

3 bit flip flop counter

input = System clock, increment, & decrement controls (1)

- also need method to load in some input
  - > also some 3 bit vector that will be loaded in when LD is high
  - > for the button states

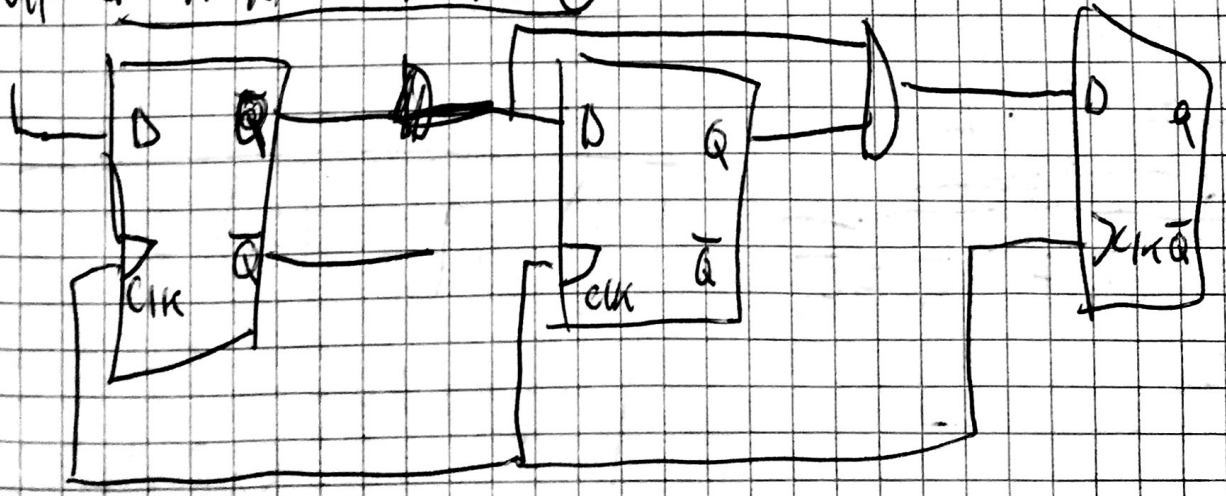
output 3 bit bus of current counter value

up count - signal when all bits are high
- tells subsequent counters to perform up count

down count high when all bits are zero

Counter design

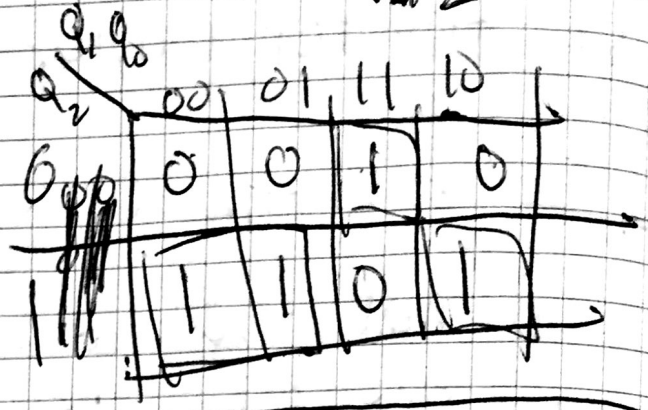
input increment decrement (1)



3 bit Counter truth table

Q_2	Q_1	Q_0	out
0	0	0	0 0 1
0	0	1	0 1 0
0	1	0	0 1 1
0	1	1	1 0 0
1	0	0	1 0 1
1	0	1	1 1 0
1	1	0	1 1 1
1	1	1	0 0 0

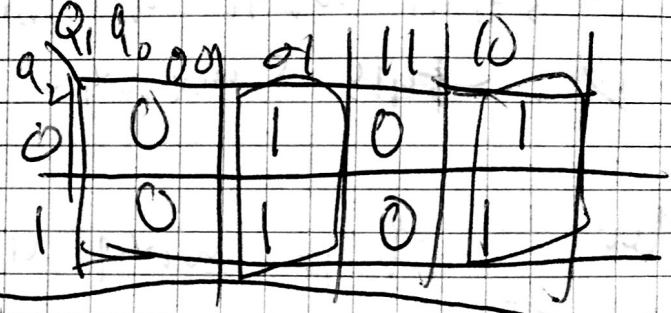
up count Q_2



$up_{on} = \bar{Q}_1 Q_2 + \bar{Q}_2 Q_1 Q_0 + Q_2 \bar{Q}_1 \bar{Q}_0$

Q_2

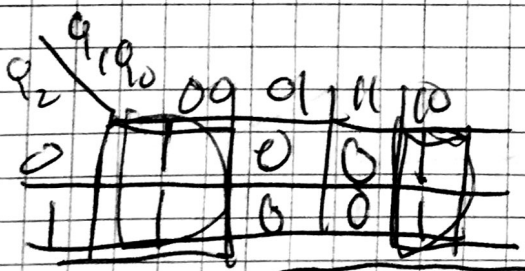
up count Q_1



$up_{on} = \bar{Q}_1 Q_0 + Q_1 \bar{Q}_0$

Q_1

up count Q_0



$up_{on} = \bar{Q}_1 Q_0 + Q_1 \bar{Q}_0 = \overline{Q_0}$

Q_0

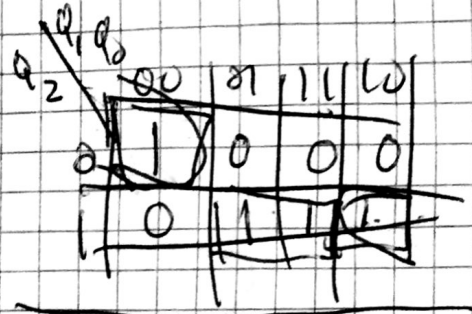
D, N (0)1, 2

normal enable on up high

3 bit down count TT

Q_2 Q_1 Q_0	out Q_2 Q_1 Q_0
0 0 0	1 1 1
0 0 1	0 0 0
0 1 0	0 0 1
0 1 1	0 1 0
1 0 0	0 1 1
1 0 1	1 0 0
1 1 0	1 0 1
1 1 1	1 1 0

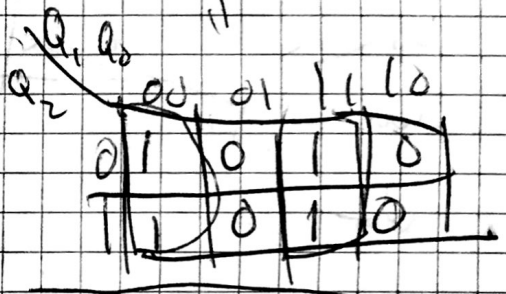
Q_2



$D_n = Q_0Q_2 + Q_2Q_1\bar{Q}_0 + \bar{Q}_2Q_1Q_0$

Q_2

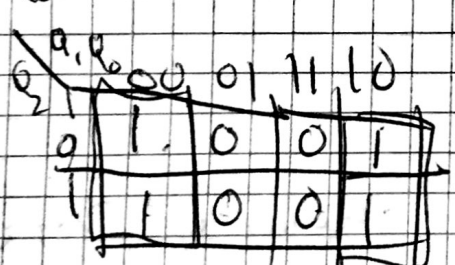
Q_1



$D_n = \bar{Q}_1Q_0 + Q_1\bar{Q}_0$

Q_1

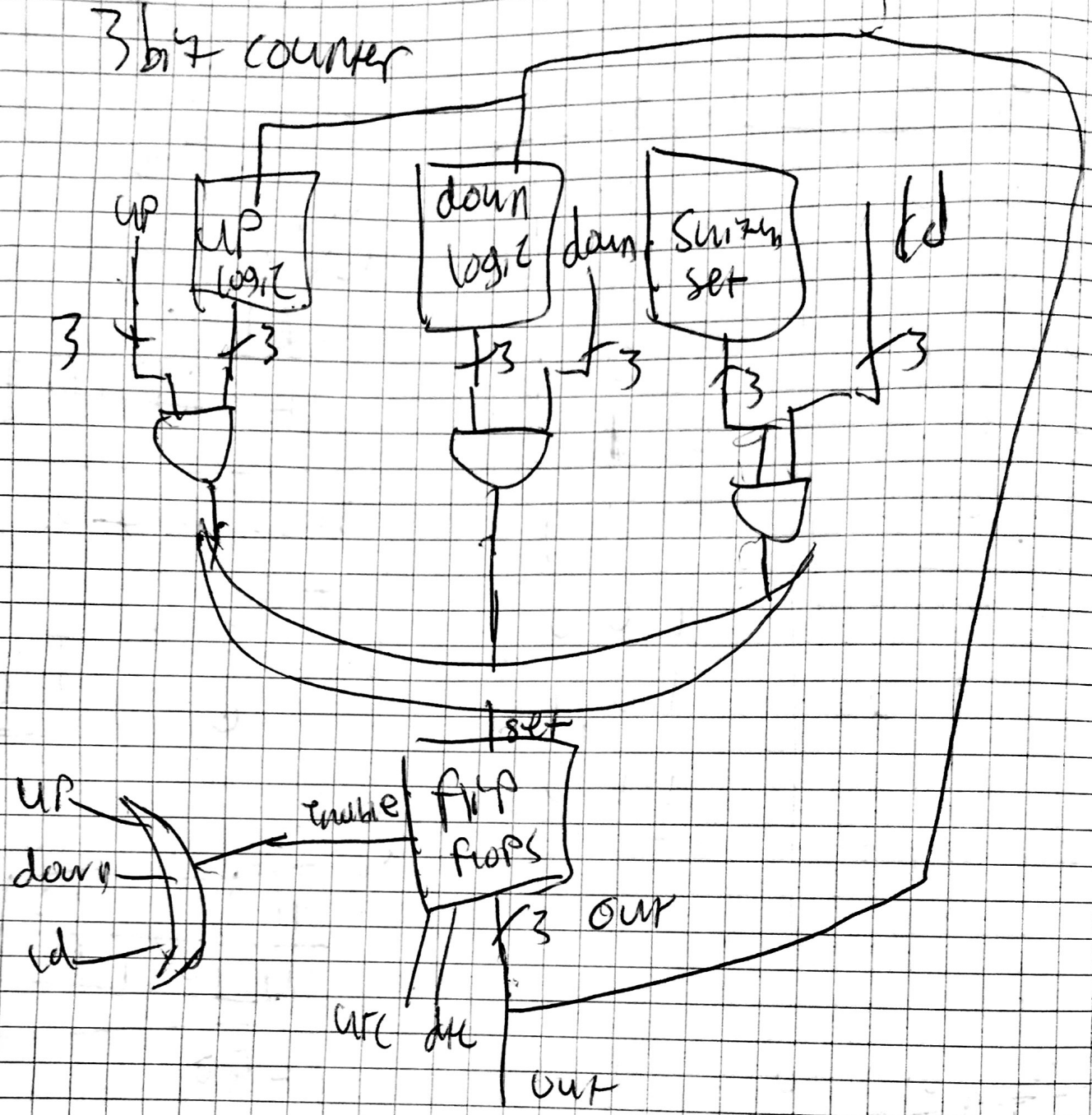
Q_0



$D_n = \bar{Q}_0$

Enable down when down count high

3 bit counter



- up down & ld logic always running there
- select which are using the extended bit
  - i.e. bit masked shift
- allow flip flops to be changed only on enable bit
  - will only change when trying to do up, down, or load on clk edge

4 bit counter

No	n_3 n_2 n_1 n_0	n_3 n_2 n_1 n_0
0	0 0 0 0	0 0 0 1
1	0 0 0 1	0 0 1 0
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 1 0 0
4	0 1 0 0	0 1 0 1
5	0 1 0 1	0 1 1 0
6	0 1 1 0	0 1 1 1
7	0 1 1 1	1 0 0 0
8	1 0 0 0	1 0 0 1
9	1 0 0 1	1 0 1 0
10	1 0 1 0	1 0 1 1
11	1 0 1 1	1 1 0 0
12	1 1 0 0	1 1 0 1
13	1 1 0 1	1 1 1 0
14	1 1 1 0	1 1 1 1
15	1 1 1 1	

4/16

Decimal point

Left most on if

(0) FFFF

Right most on if

(0) 0000

down count event

q_n is 1 if $Q_n(or(Q_0 : Q_{n-1}))$ or

$\sim or(Q_0 \ldots Q_n)$

Im 1 & at least one 1

were all 0

q_n is 1 if

Up count

$q_n \leftarrow (\in \{Q_0 : Q_{n-1}\})$

Start 1 if Im 1 & Evenore before me isn't all 1

$\bar{q}_n \leftarrow \{\in Q_n\}$

Rest of Im 0 & evenore else is 1

If Q_n not is 0 automatically ~~reset~~ shift

16 bit counter

- input,
  up event - bit
  down event - bit
  rst event - bit
  clk - clock
  16 switch input = 16 bit #

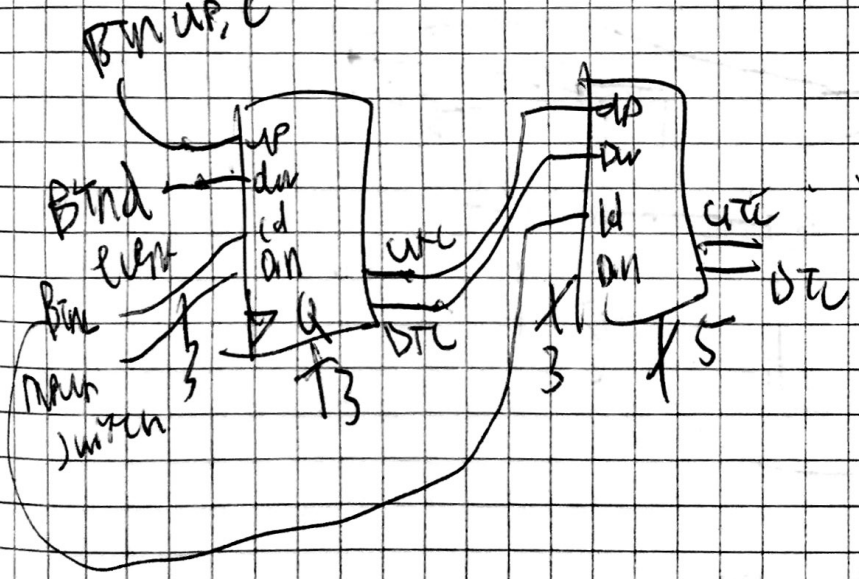
- output
  16 bit # → current count
  utc → roll over
  dtc → down under

  2 instances of 3 bit
  2 instances of 5 bit

4/26

up/down events controlled by button

- only affects lowest bit
  - i.e up count signal good only for bottom 3 bits
- UTC & DTC events propagate up the counter
- upper counters only implement count events
  - if all bits in lower counter are 1
  - i.e wire DTC & UTC to subsequent up down event
- can one has set of switches as load value
- clock is fed together



edge detector - used for btnU & btnD

↓ i.e. single events - one button push

output q1 for 0 0 1
first last

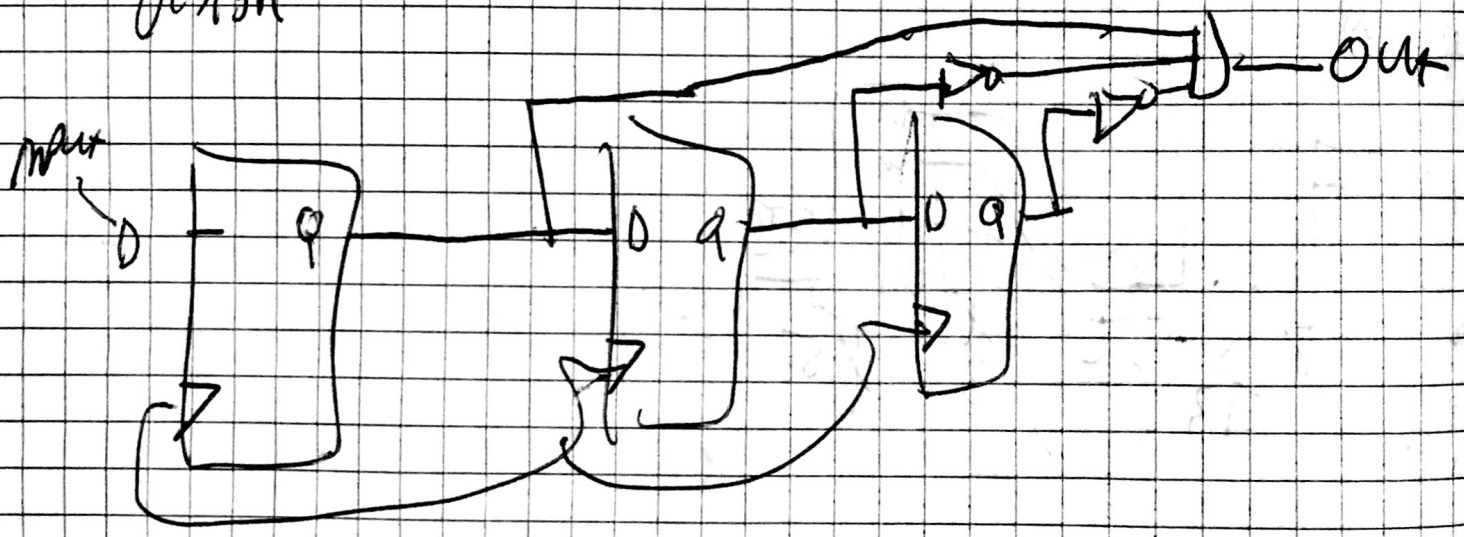
i.e. 2 0's then a 1

- 3 FDRE's

input = btn, clock,

output = high signal

design



- feed flip flops into each other
- will hold previous state for 1 cycle
- circuit will only output high when most recent 2, of 2
- enable on clock cycle

Selector 4/30

input = 16 bit # from counter
- represents 4 digits to display on segment display
- also some ring counter
  - bit shifter that highlights active LED
    - ie 0001 first display
      0010 second "
      0100 third "
      1000 fourth "
- bit extend certain bit to create bitmask
  - and w specified bit range to create "mux"

output = 4 bit # that represents digit to show on 7seg
↳ input to 7 seg display

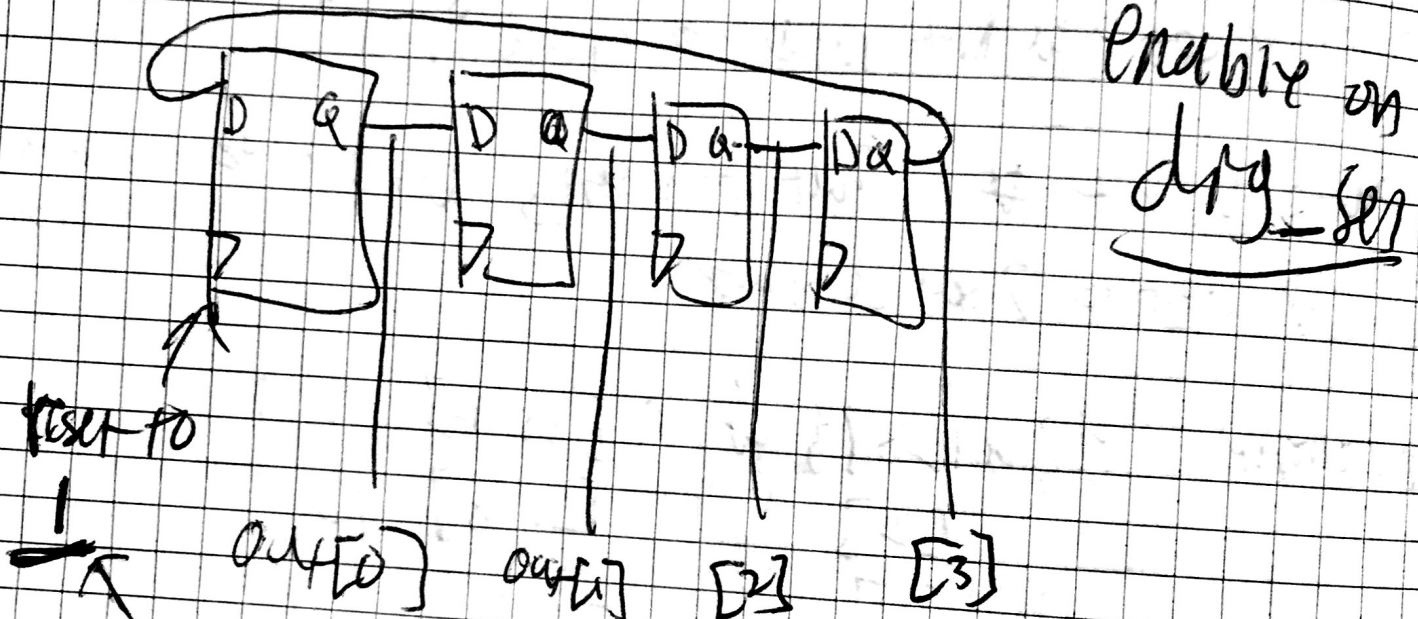
implement: 4{sel[3]} & in[15:12] | or
{sel[2]} & in[11:8] | or
{sel[1]} & in[7:4] | or
{sel[0]} & in[3:0] | or

ring counter

- bit rotater
- advance on drg sel input

input: drg sel ← advance
clk ← clock

output: 4 bit # w/ one | active bit



enable on drg_sel

clock puts an initial 1 in ring on reset

- each flop represents a single bit

Top Schematic

Input: CLK_in - 100MHz CLK module
- BTNR - right btn → reset of FPGA's
- BTNU - increment counter by 1
- BTND - decrement counter by 1
- BTNC - change continuously except from
- BTNL - load switch values FFFC to FFFF
- sw - 16 switches #16

Out:
- #8 Seg - 8bit 7 seg display
- 1 dp - decimal point
- #4 an - controls seg digits
- #16 led - Controls leds above switches

edge detector x2

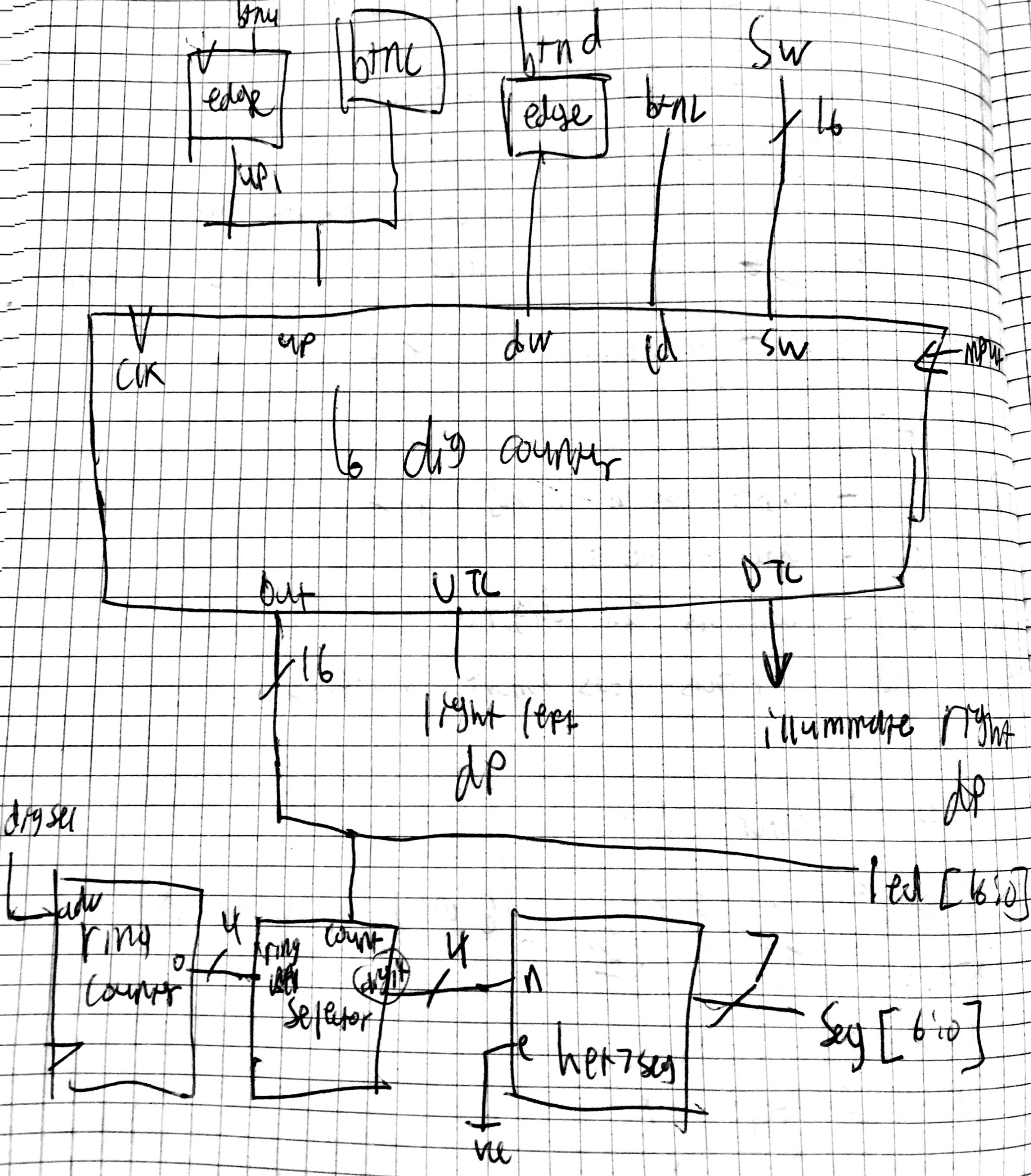
~~3digit counter x2~~
~~5 digit counter x2~~

16 digit counter x1

clock clks x1
ring counter x1
selector x1
hex 7 seg x1

16 dig counter

6/3/2018



log2 for binc

0 for ~~[illegible]~~ ~&[2:15]&

↑ 0 when all upper bits on

binc & ~(&{out[2:15]})

(binc works until upper 14 bits are 1

log2 for Seg display

- Set an equal to ring counter
- will highlight correct segment

decimal point on for

decimal = 0

~~out~~ o[3] & ~dec | o[0] & dec

↑ on upper segment | ↑ on lower segment

turn on if illuminating first digit & overflow on counter | turn on if illuminating last dig & underflow on counter

Testing Checklist

3bit ✓

5bit ✓

6bit ✓

edge detector ✓

ring

selector

input fast clk:

LEDs have constant glow

- 1, 2 3, 4 ghosted image between adjacent displays
- brightest display on 4
  - spends most clock cycles on 4, connected wrong
- bcnt immediately shoots to FFFC
  - ghosted display for F.F.F.F

    dp gets gradually fainter
- digits moving too fast for LEDs to look seamless