ECE 5722 Final

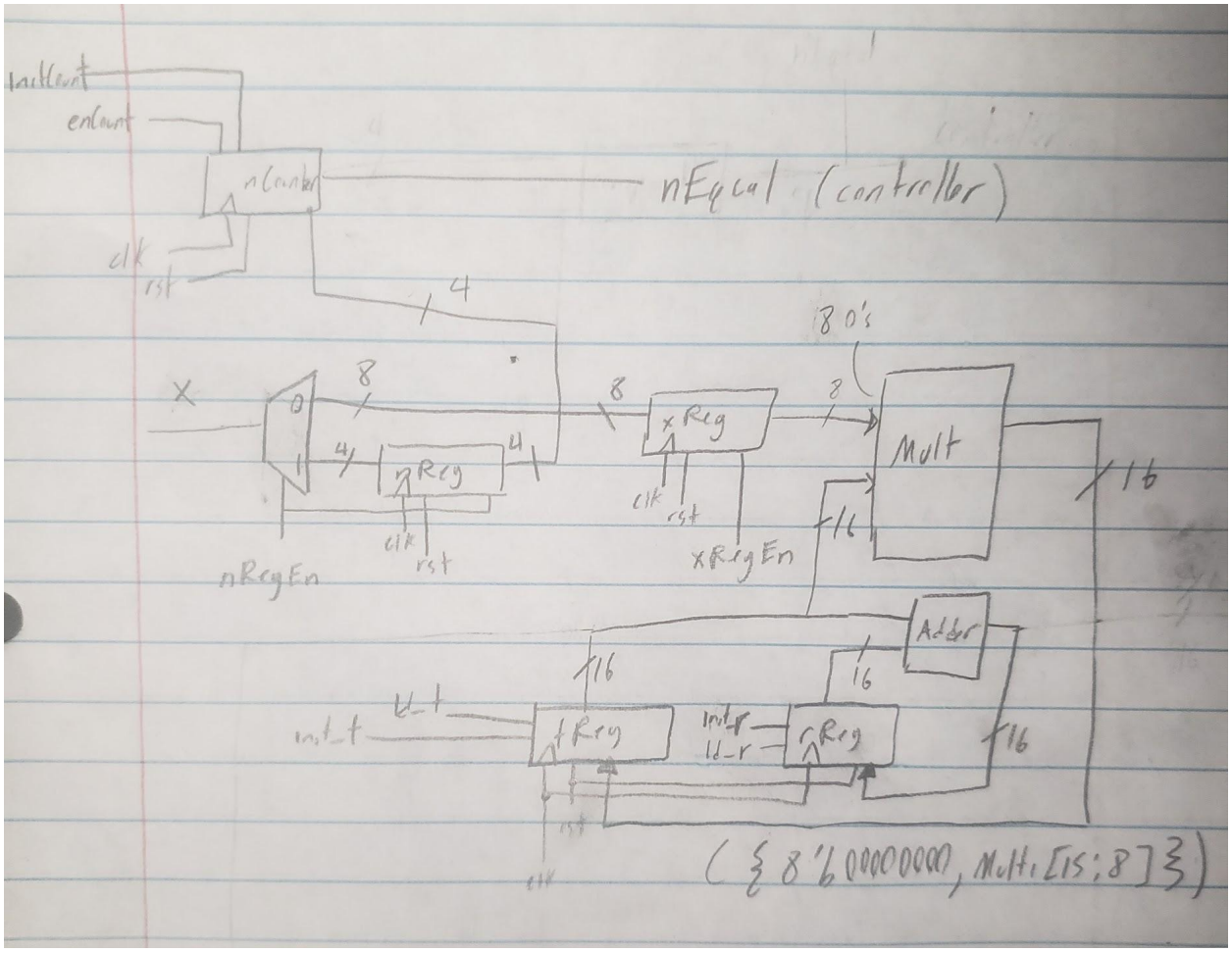
By Bruce Huynh

12/08/2021

Problem 1:

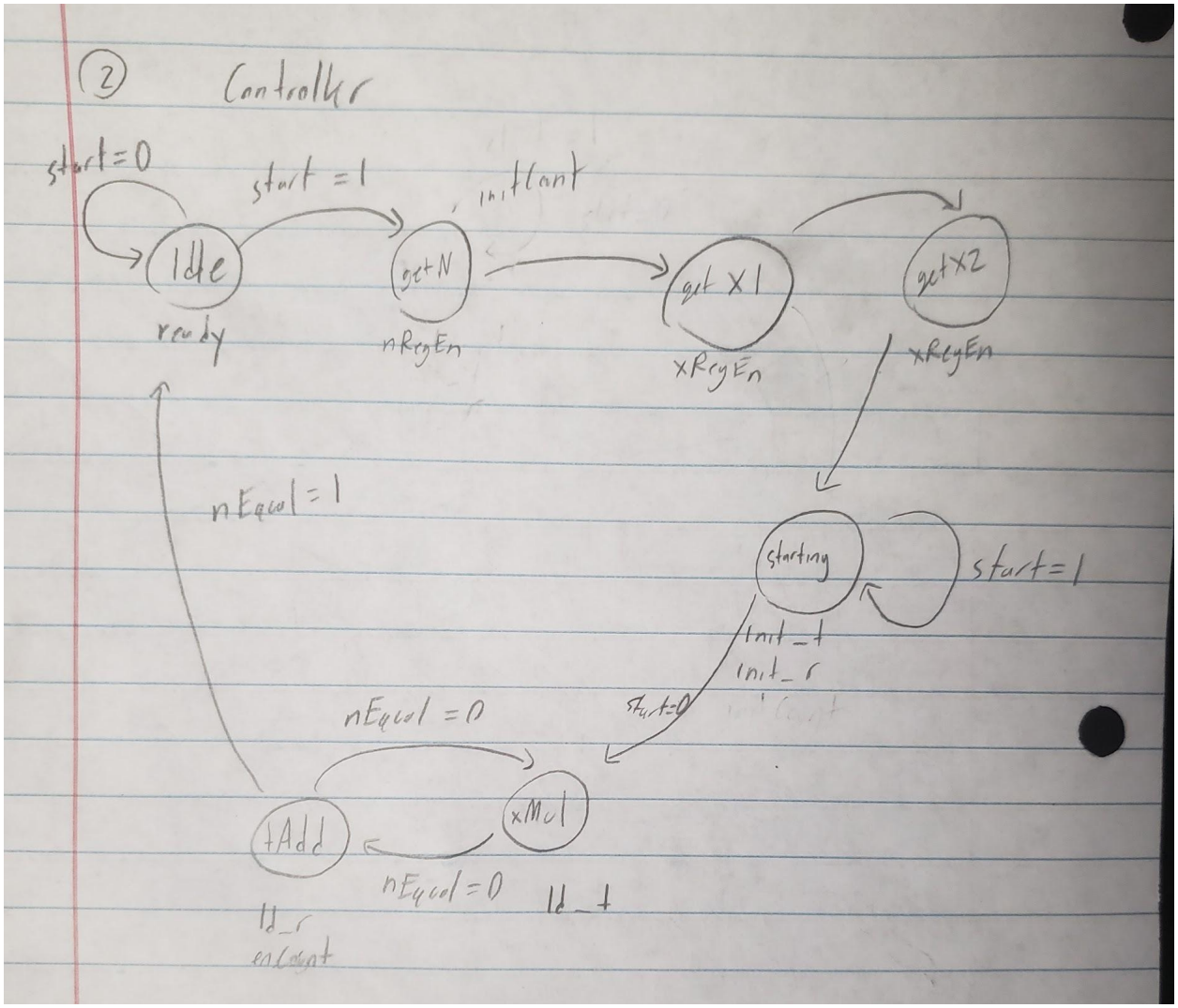
In this problem, we were tasked with doing a Taylor series approximation of 1/(1-x). For a fractional x, that specific Taylor series can be found with the following sequence: 1 + x + x^2 + x^3… x^n.

The following is the datapath for our circuit:



When a start pulse is detected, the first 4 bits are going to be put into the nReg which holds the *n* amount of iterations we are going to perform with our circuit. Afterwards, 2 nibbles of data (4 bits each) are going to determine our fractional x. We keep multiplying the *x* to get its exponent values and then add them together. rReg holds the sum while tReg holds the result of the multiplication. In order to correctly account for the 8-bit integer and 8-bit fractional part, we pad

the *x* input with 8 zeros on the left, we take the MSB of the multiplier product and then pad that with 8 zeros on the left as well.



The controller waits for a complete start pulse to happen, then it will get the value of *n* into a register and then the 8-bit value of *x,* two nibbles in two clock cycles so two states. Afterwards, the registers for holding the product and sum will be initiated and enabled. The process stops when the counter reaches *n* - 1 value of the nReg.

For the code of the datapath, refer to the .zip file for more information (as there are many modules on different files). The following is the implementation of the controller:

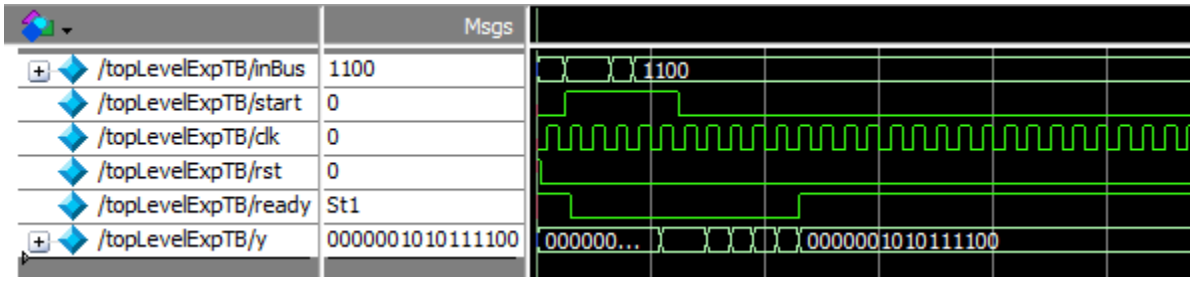
```
module controller(input start, nEqual, clk, rst, output reg nRegEn,
xRegEn, init_t, init_r, initCount, ld_r, ld_t, enCount, ready);

     reg [3:0] pstate, nstate;
     parameter idle = 3'b000, getN = 3'b001, getX1 = 3'b010, getX2 =
3'b011, starting = 3'b100, xMul = 3'b101, tAdd = 3'b110;

     always @(posedge clk or posedge rst) begin
          if (rst) pstate <= idle;
               else
                    pstate <= nstate;
     end

     always @(pstate or start or nEqual) begin

          case (pstate)

               idle: begin

                    if (start == 0)
                         nstate = idle;

                    else if (start == 1)
                         nstate = getN;

               end

               getN: begin
                    nstate = getX1;
               end

               getX1: begin
                    nstate = getX2;
               end

               getX2 : begin
                    nstate = starting;
               end
```

```
                starting : begin

                      if (start == 1)
                            nstate = starting;
                      else if (start == 0)
                            nstate = xMul;
                end

                xMul: begin

                      if (nEqual == 0)
                            nstate = tAdd;
                      else if (nEqual == 1) // even if it gets
interrupted after multiplication, the addition wont be affected.
                            nstate = idle;
                end

                tAd: begin

                      if (nEqual == 0)
                            nstate = xMul;
                      else if (nEqual == 1)
                            nstate = idle;

                end

          endcase

     end

          always @(pstate or start or nEqual) begin

                            nRegEn = 0;
                            xRegEn = 0;
                            init_t = 0;
                            init_r = 0;
                            initCount = 0;
                            ld_t = 0;
                            ld_r = 0;
```

```
                enCount = 0;
                ready = 0;

        case (pstate)

                idle: begin

                ready = 1;

                end

                getN: begin

                nRegEn = 1;
                initCount = 1;

                end

                getX1: begin

                xRegEn = 1;

                end

                getX2 : begin

                xRegEn = 1;

                end

                starting : begin

                init_t = 1;
                init_r = 1;


                end

                xMul: begin
```

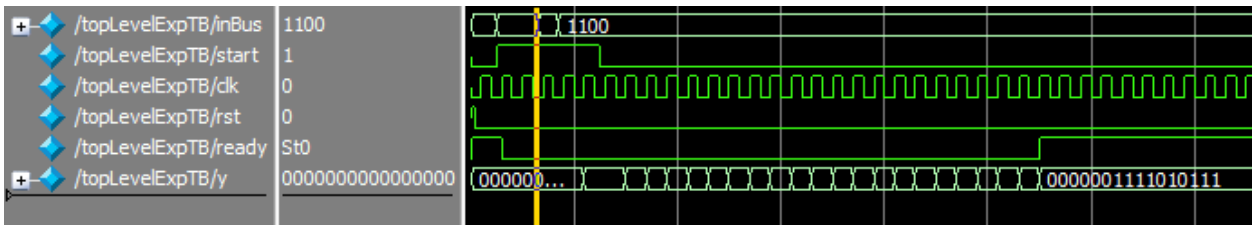
```
                        ld_t = 1;

                    end

                    tAdd: begin

                        ld_r = 1;
                        enCount = 1;

                    end
            endcase
        end
endmodule
```

Here are the results of our circuit:



This waveform is for an *n* value of 4 iterations and an *x* value of .75. This gives us a result of 10.10111100. This is equal to 2.734375 which is the exact same value of (1 + .75 + .75^2 + .75^3)

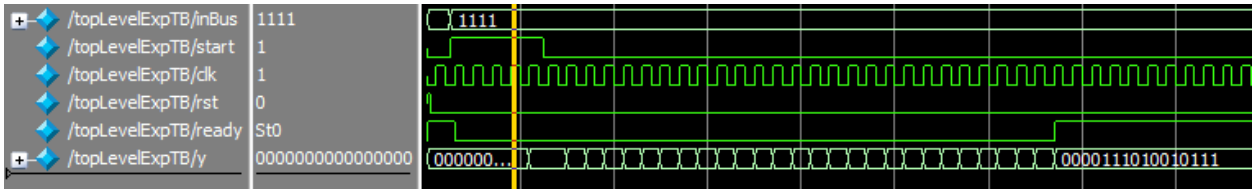
For a closer approximation, let's increase *n* to 12 since we are not close to the value of 1/(1-.75) or 4:



With 12 iterations we get a value of 11.11010111 which is equal to 3.83984375. This is very close to the end value of 4 and it can be achieved with more iterations.

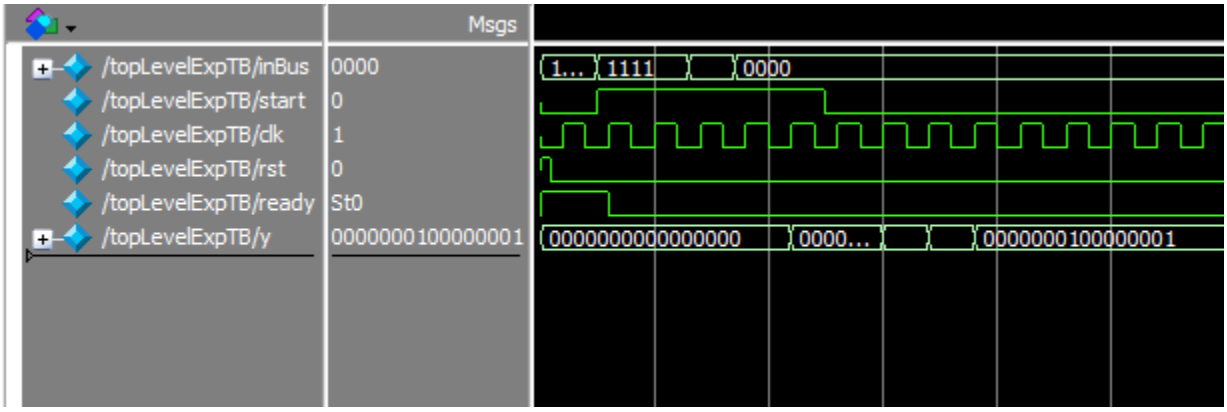
Next let's compute the maximum and minimum values with an *n* of 15. The largest value of *x* is .11111111 or 0.99609375. The smallest value of x is .00000001 or 0.00390625. Let's see what we get when we put both of these values in.

.11111111:



We get 14.58984375 from our circuit while the actual value is 14.59670 (1 + 0.99609375 + 0.99609375^2 + 0.99609375^3 + 0.99609375^4 + 0.99609375^5 + 0.99609375^6 + 0.99609375^7 + 0.99609375^8 + 0.99609375^9 + 0.99609375^10 + 0.99609375^11 + 0.99609375^12 + 0.99609375^13 + 0.99609375^14). The full value is expected to be 256 but this is not possible with only 4 bits for *n* amount of iterations.

.00000001:

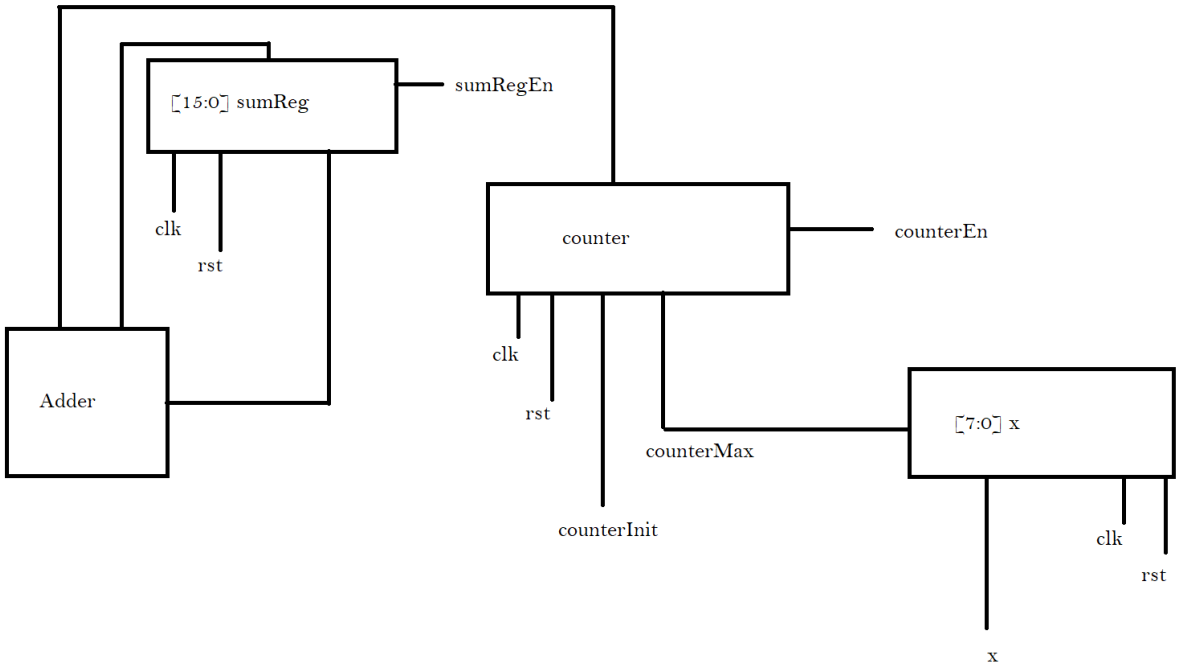


We get 1.00000001 for our output which is 1.00390625. 1.00392156863 is the full expected result and 1.00392156863 is the expected result of 15 iterations. So our circuit is still quite accurate.

Problem 2:

In this problem, we were tasked with creating a custom instruction that creates the sum of 1 to x with x being the input.

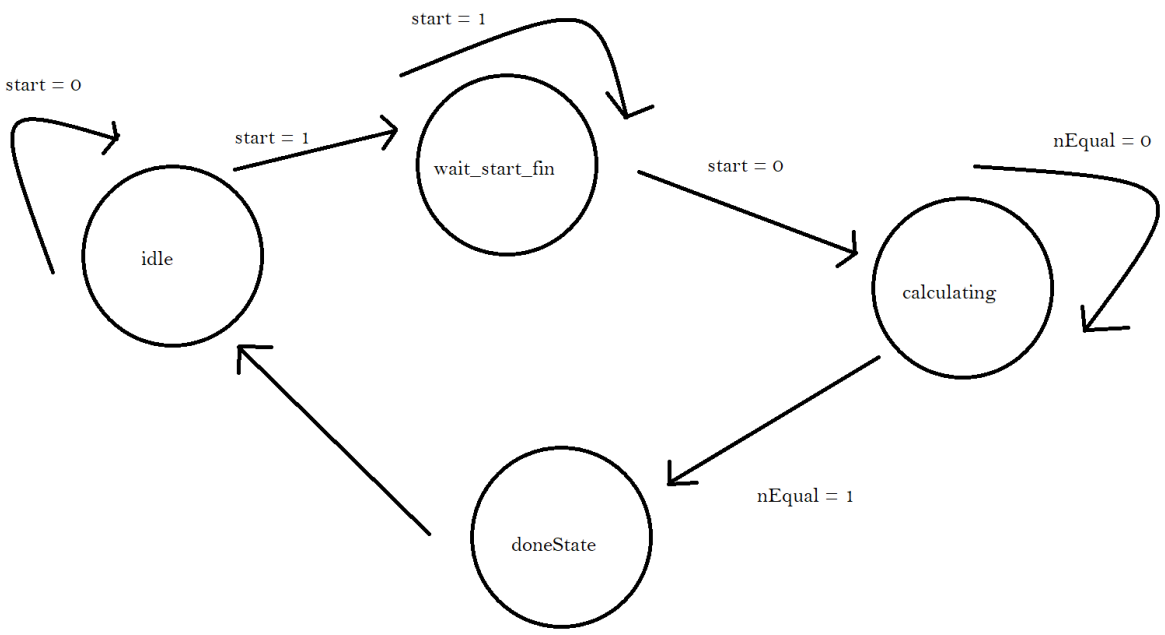
First we design the datapath for the circuit as follows:



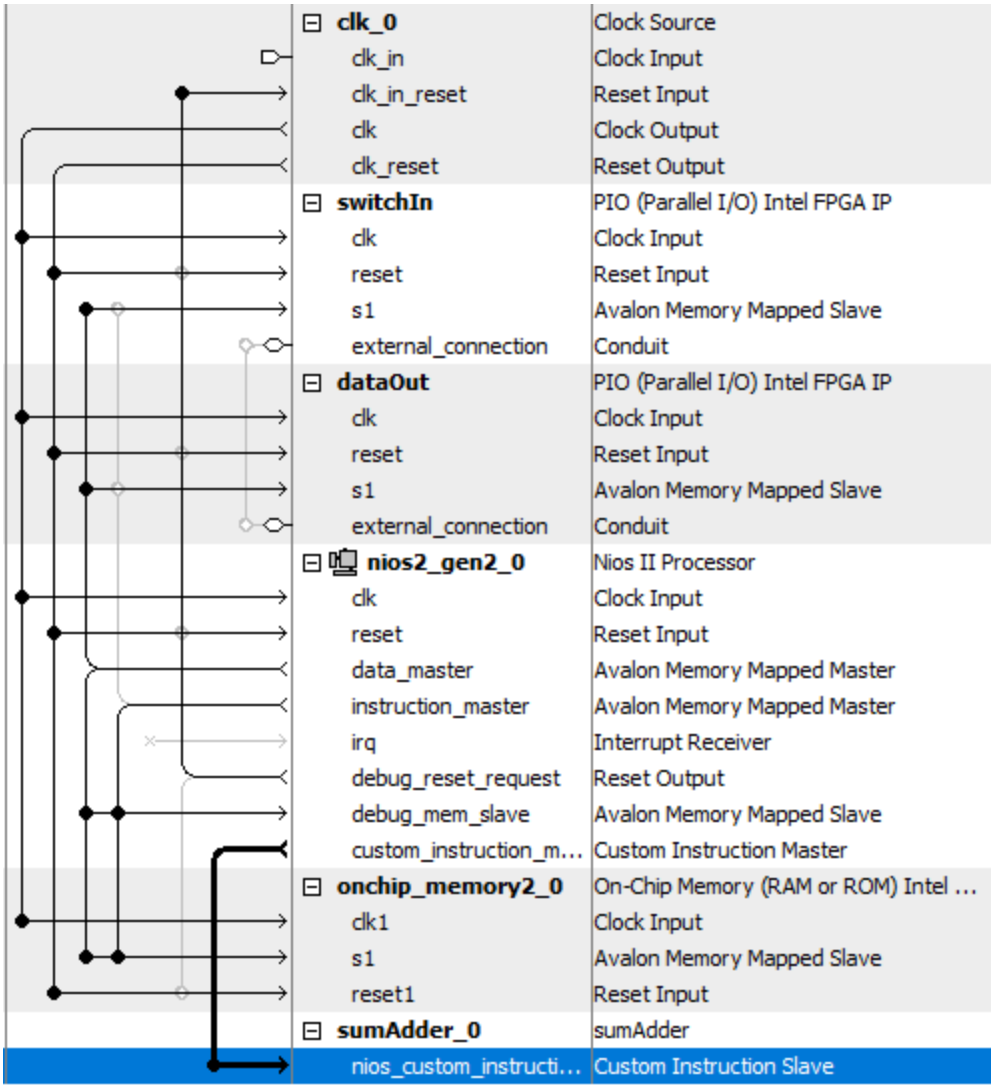
Note that the [7:0]x also has an enable signal not seen here

We have the counter that gets initialized to 1. Its value is added to the current sum until we reach our inputted x value.

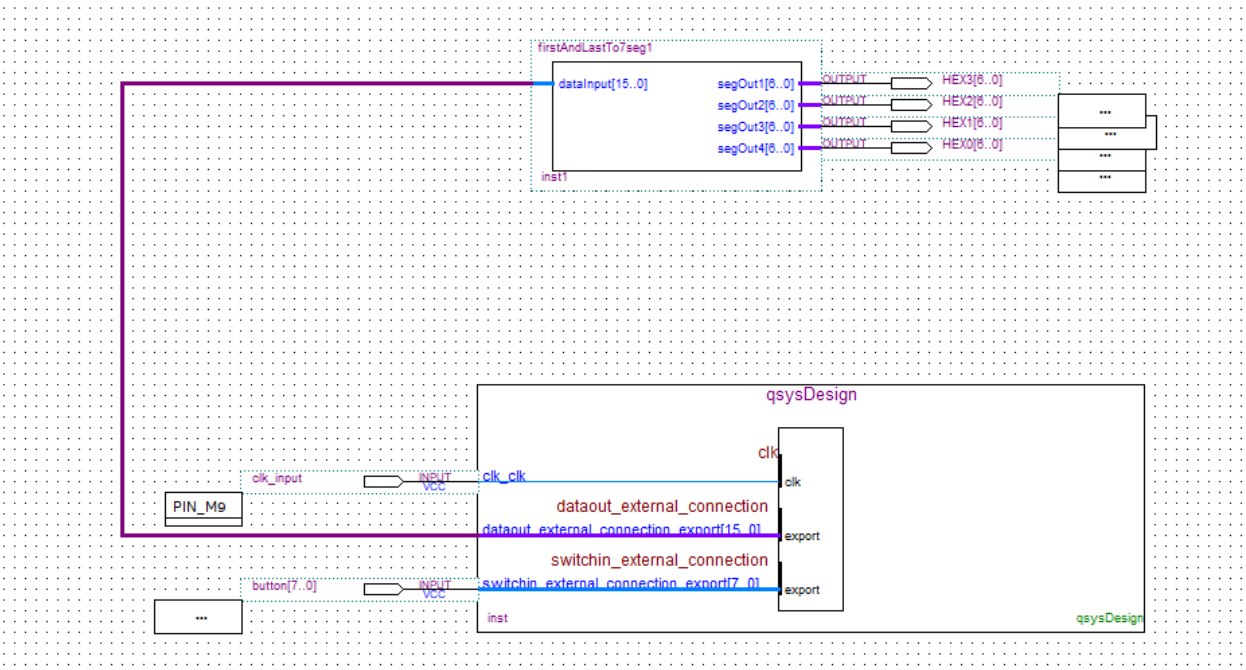
The sum register, counter, and x register signals are all controlled by the controller. After a start pulse has finished it goes to a state that calculates the value until the sum of all the numbers are completed.



We synthesize the design and test it in ModelSim to test its correctness, then we put it into our Qsys system as a custom instruction. The following is the Qsys design:



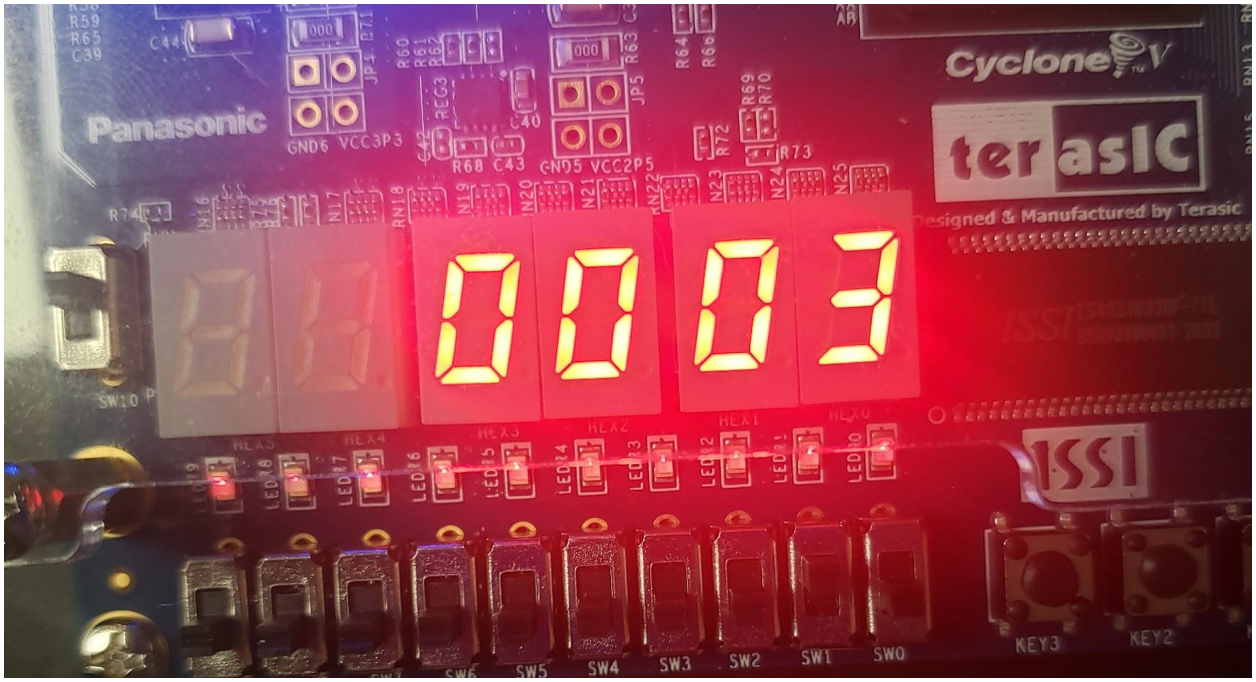
To display it, we combine the Qsys design with a BCD converter so we can see the output on our FPGA development board.



To use the custom instruction we use the following C code:

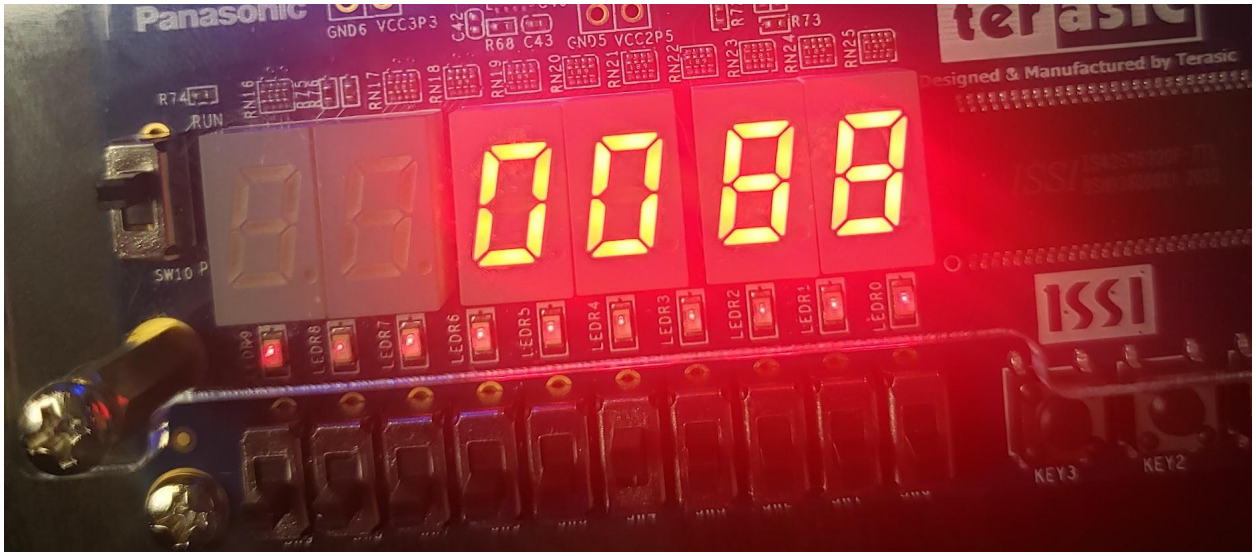
```c
#include <stdio.h>
#include "system.h"
#include "altera_avalon_pio_regs.h"

int main()
{
    while (1){
    int zero = 0;
    int inputData = IORD_ALTERA_AVALON_PIO_DATA(SWITCHIN_BASE);

    int result = ALT_CI_SUMADDER_0(inputData, zero);
    IOWR_ALTERA_AVALON_PIO_DATA(DATAOUT_BASE, result);
    }
}
```

The following are the results:

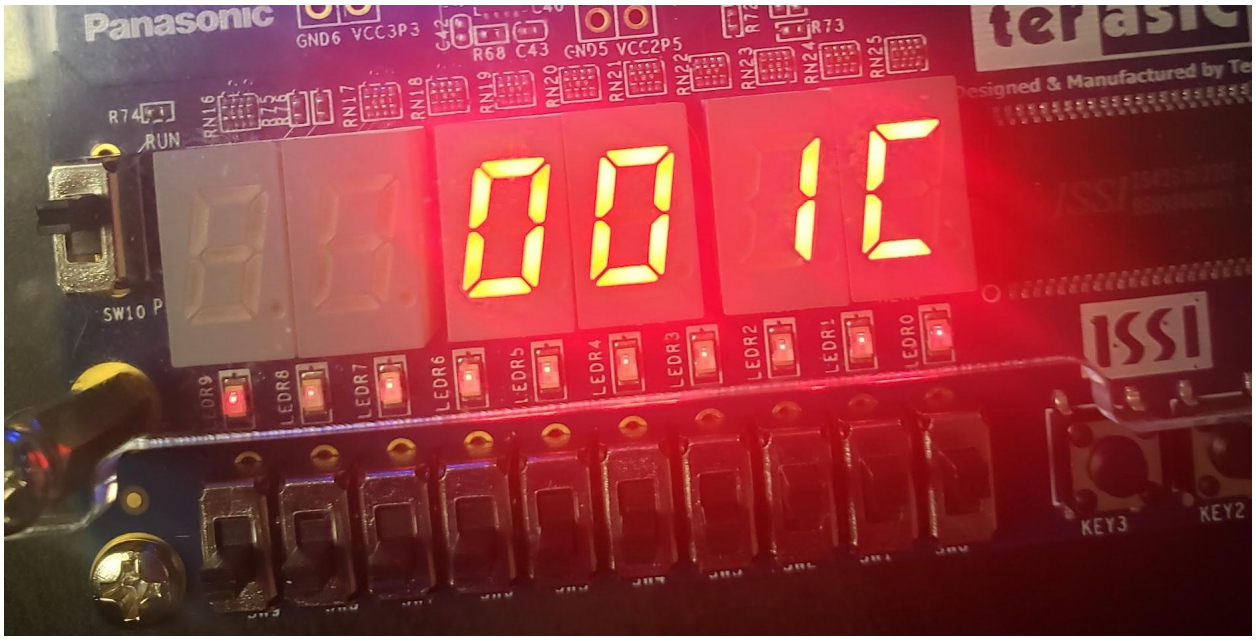


When inputting 2 into the circuit, we get 3 back (1 + 2 = 3).

When inputting 8 into the circuit we get 0x24 (36) (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36).



When inputting 16 into the circuit we get 0x88 (136) (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 + 16 = 136).

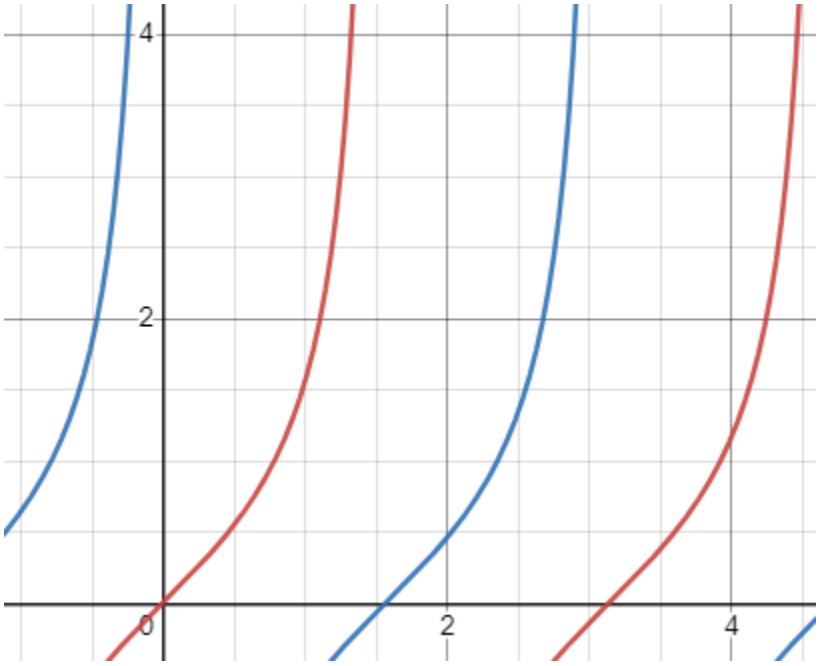


When inputting 7 into the circuit we get 0x1C (28) (1 + 2 + 3 + 4 + 5 + 6 + 7 = 28).

Problem 3:

In this problem, we were tasked with creating a software interface for our tangent hardware accelerator. The circuit can only take values between 0 and pi/2, but we want it to be able to calculate tan(x) for all values 0 <= x <= 2pi.

To do this, we need to separate these ranges and do different math on them. For the range 0 <= x < pi/2, we do not have to make any changes as it is the regular tan(x) region. For the range pi/2 <= x < pi, we use -cot(x) as it has the exact same shape but shifted to the right a bit.



The red line represents tan(x) while the blue represents -cot(x). To fix the offset, we subtract pi/2 from the initial x value. -cot will be applied after putting x through the accelerator. For pi <= x <= 3pi/2, we use tan again and subtract pi from x. Lastly, for the range 3pi/2 <= x < 2pi we use -cot at the end and we subtract 3pi/2 from x initially.

This is how we set up the code with if statements:

```
// adjusting input x before it goes into the accelerator

if (x >= 0 && x < PI/2){

    cot = 0;
    input = x;  // no changes needed
    break;
}

else if (x >= PI/2 && x < PI){
    cot = 1; // negative cot needed
    input = x - (PI/2);
    break;
}

else if (x >= PI && x < ((3*PI)/2)){
    cot = 0;
    input = x - PI; // everything is the same but we need to shift
    break;
}

else if (x >= ((3*PI)/2) && x < (2*PI)){
    cot = 1; // negative cot needed
    input = x - (3*PI/2);
    break;
}
```

Afterwards, we start the accelerator and then wait for the done signal. When a done pulse is detected, for the inputs that require a -cot(x) edit, we perform that on the result output.

```
if (done == 1){ //wait for calculation to be done
    if (cot == 1){ // if additional cotangent was needed
        result = -1/result; // perform -cot(x) to get correct result
    }
```

We then write the result on the PIO, which will be displayed on the 7-Segment Display. The following is the whole code:

```
#include <stdio.h>
#include "system.h"
#include "altera_avalon_pio_regs.h"

#define PI 3.14159265359
#define START_REG_OFFSET 1 // fake values (will be different when we
know addresses)
```

```
#define X_REG_OFFSET 2
#define DONE_REG_OFFSET 3
#define RESULT_REG_OFFSET 4
int main()
{

	while (1){
		int cot = 0;
		double result, input;

		x = IORD_ALTERA_AVALON_PIO_DATA(x_base);

	// adjusting input x before it goes into the accelerator

	if (x >= 0 && x < PI/2){

		cot = 0;
		input = x; // no changes needed
		break;
	}

	else if (x >= PI/2 && x < PI){
		cot = 1; // negative cot needed
		input = x - (PI/2);
		break;
	}

	else if (x >= PI && x < ((3*PI)/2)){
		cot = 0;
		input = x - PI; // everything is the same but we need to
shift
		break;
	}

	else if (x >= ((3*PI)/2) && x < (2*PI)){
		cot = 1; // negative cot needed
		input = x - (3*PI/2);
		break;
	}
```

```
	IOWR(TAN_ACCEL_BASE, X_REG_OFFSET, input); // putting new input
into the x input
	IOWR(TAN_ACCEL_BASE, START_REG_OFFSET, 1); // start signal
asserted

	IORD(TAN_ACCEL_BASE, DONE_REG_OFFSET);
	result = IORD(TAN_ACCEL_BASE, RESULT_REG_OFFSET);

	if (done == 1){ //wait for calculation to be done
		if (cot == 1){ // if additional cotangent was needed
			result = -1/result; // perform -cot(x) to get
correct result
		}
		IOWR_ALTERA_AVALON_PIO_DATA(RESULT_base, result); //
writing to PIO (7-Segment Display)
	}

	}
}
```

To show that the math works, I will do three examples with each being in the range of pi/2 - pi, pi - 3pi/2 and 3pi/2 - 2pi.

pi/2 - pi:

tan (1.4pi/2) = −1.37638192

(subtract pi/2)

tan (.4pi/2) = 0.72654252

(-cot)

-1/0.72654252 = −1.37638193

pi - 3pi/2:

tan (6pi/5) = 0.72654252

(subtract pi)

tan (pi/5) = 0.72654252

3pi/2 - 2pi:

tan (1.9pi) = −0.32491969

(subtract 3pi/2)

tan (.4pi) = 3.07768353

(-cot)

-1/3.07768353 = −0.32491969