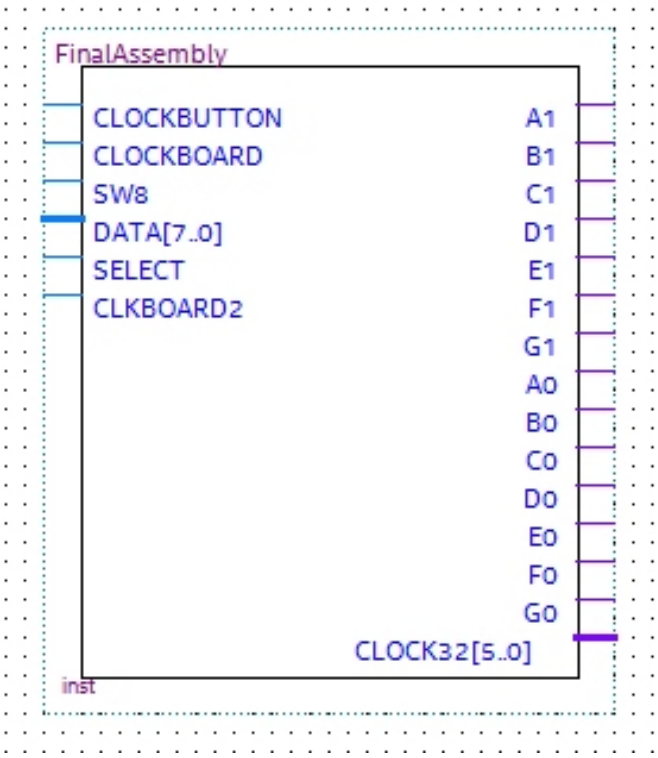
Max Van de Wille
490227758
Lab Section F

CprE 281 Final Project Report

For my final project in CprE 281, I elected to implement an encryptor/decryptor that utilizes the tiny encryption algorithm. My overall circuit for the project was to contain a finite state machine, a register file, and combinational logic in order to perform encryption using an Altera FPGA board. The circuit takes in a 16-bit integer value and a 32-bit integer key to be used in the algorithm.

For a top-level view of my circuit it is easiest to look at a block symbol file for the overall circuit:



The input functions are as follows:

- CLOCKBUTTON – A pushbutton on the board assigned as a manual clock for the user to cycle

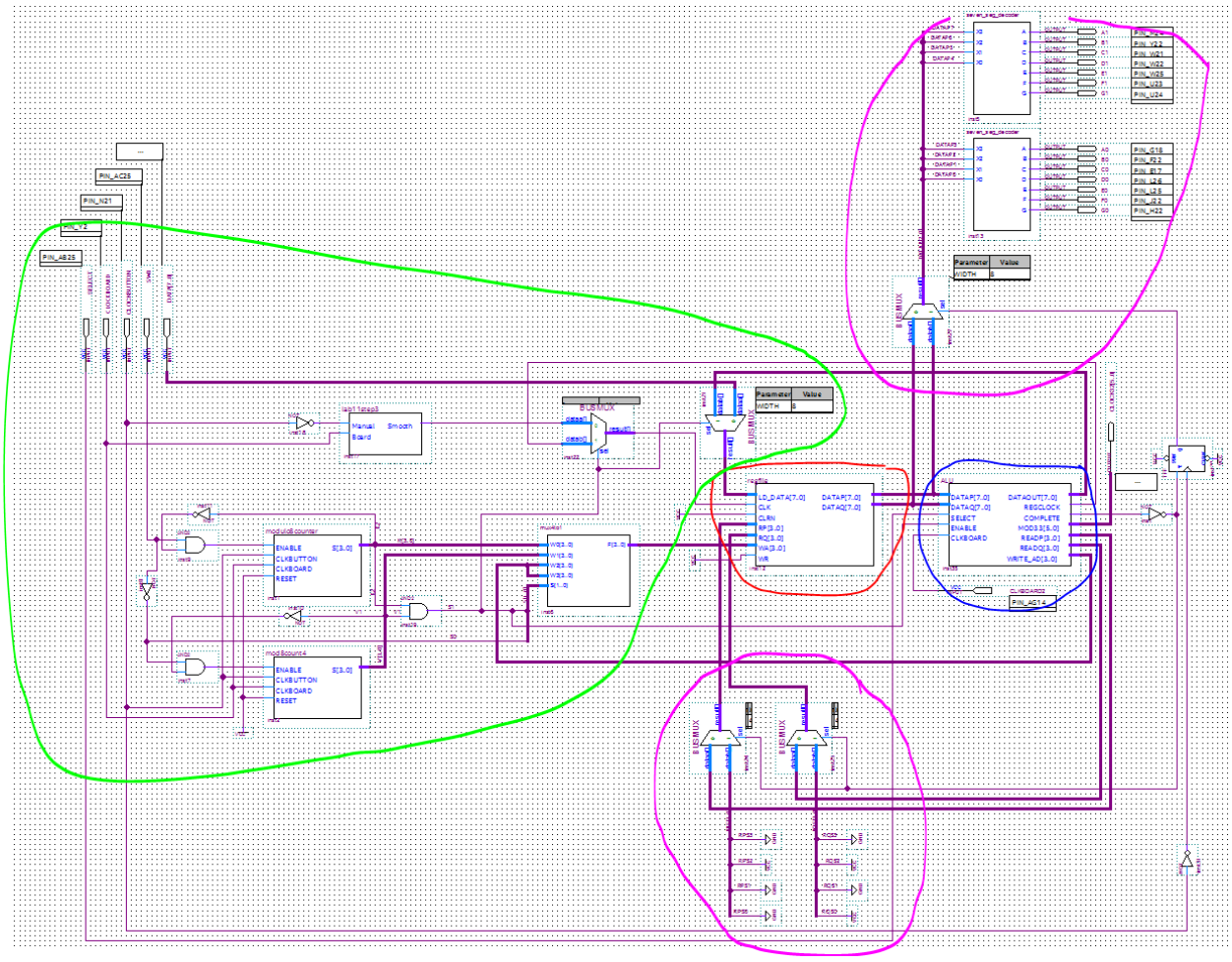
- CLOCKBOARD & CLKBOARD2 – Two 50 MHz clocks provided by the FPGA board
- DATA[7..0] – An 8-bit input for the circuit, assigned by 8 switches on the board to be toggled by the user
- SW8 – A switch to determine if the user is loading the key or the value to be encrypted using the DATA[7..0] inputs
- SELECT – This is assigned to a single switch that determines if the circuit is to encrypt the given value, or decrypt it using the inputted key

The output functions are as follows:

- A1-G1 & A0-G0 – These are to be assigned as two seven segment displays, where the circuit will display the encrypted/decrypted value one byte at a time
- CLOCK32[5..0] – A visual representation of the clock within the arithmetic and logical unit(ALU) to show when the circuit is nearing completion

The total circuit can be difficult to see the individual components in the following picture, so I will be breaking up the total circuit into four segments; the register file depicted in red, the control circuit shown in green, the ALU shown in blue, and the output shown in pink.

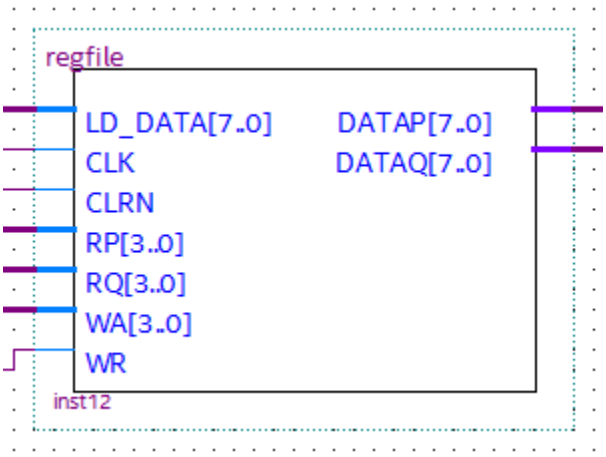
Max Van de Wille
490227758
Lab Section F



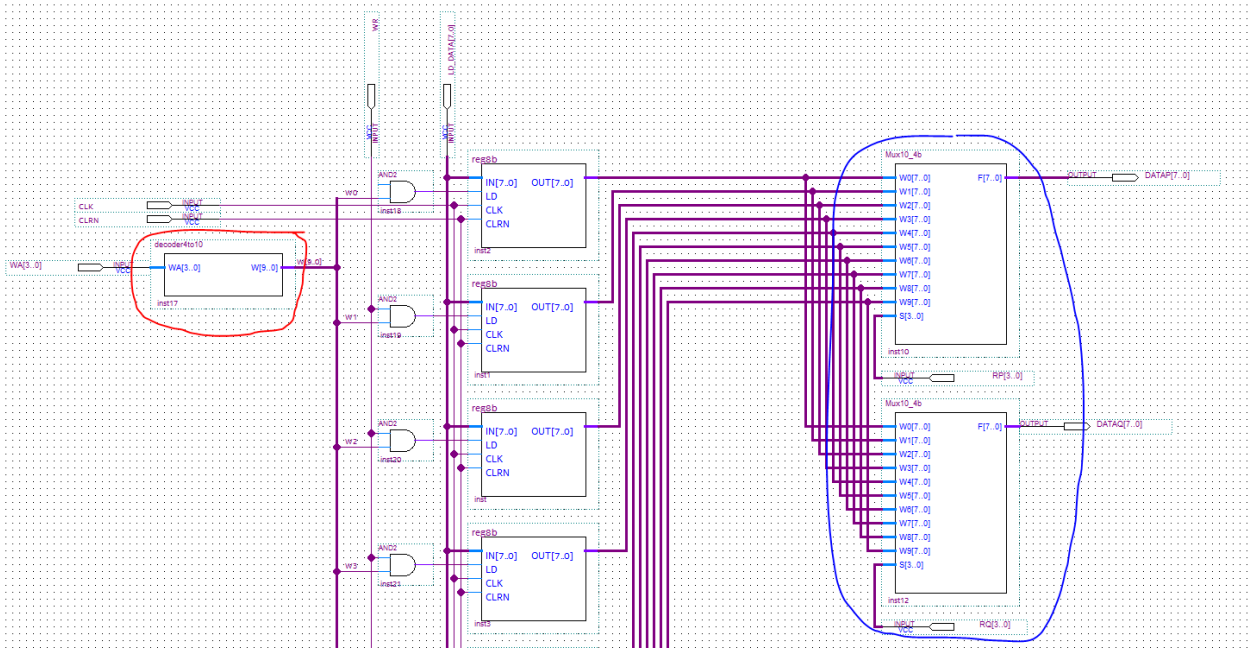
Section 1: The Register File

The register file acts as storage for the circuit, both permanent and temporary. The file contains 10 individual registers capable of storing 8 bits each. The register file is fairly standard as far as register files go, with two 8-bit read ports and one 8-bit write port. The register file has a CLRN function to clear all of the registers and a WR function to enable writing to the register. For our purposes both of these will remain at VCC so the register file does not clear and is always writeable. Read address port RP[3..0] corresponds to the read address for DATAP[7..0], and likewise for RQ and DATAQ.

Max Van de Wille
490227758
Lab Section F



The register file utilizes a 4 to 10 decoder, circled in red, and two 11 to 1 multiplexers with 4 bit select lines, circled in blue.



Both of these blocks are implemented in Verilog, with the decoder and multiplexer code shown below:

Max Van de Wille
490227758
Lab Section F

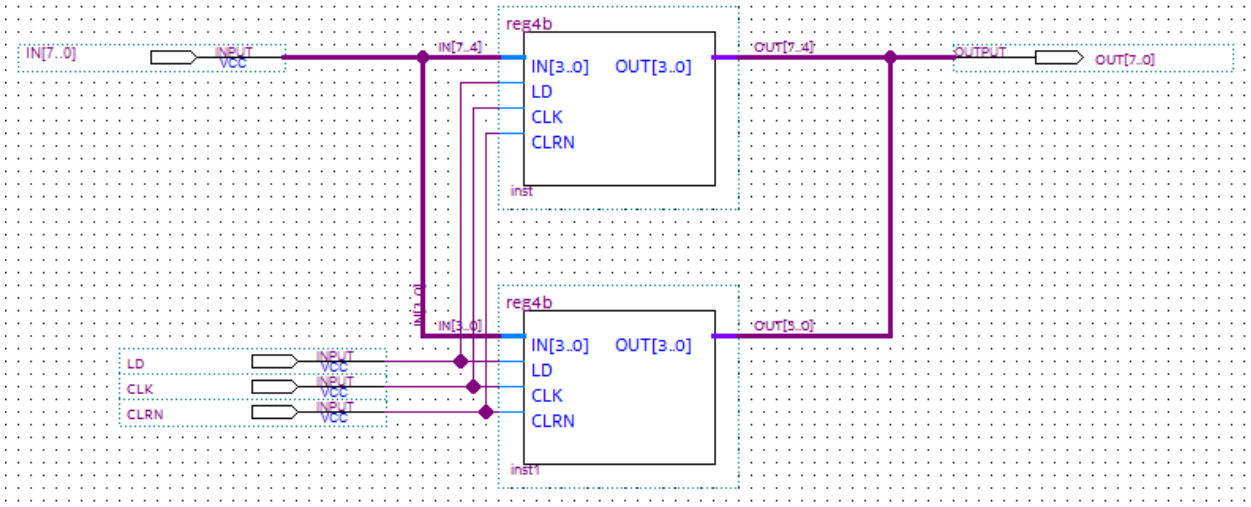
```
module decoder4to10( WA, W );
    input [3:0] WA;
    output [9:0] W;
    reg [9:0]W;

    always @(WA)
    case({WA})

        4'b0000: W = 10'b0000000001;
        4'b0001: W = 10'b0000000010;
        4'b0010: W = 10'b0000000100;
        4'b0011: W = 10'b0000001000;
        4'b0100: W = 10'b0000010000;
        4'b0101: W = 10'b0000100000;
        4'b0110: W = 10'b0001000000;
        4'b0111: W = 10'b0010000000;
        4'b1000: W = 10'b0100000000;
        4'b1001: W = 10'b1000000000;
    endcase
endmodule
```

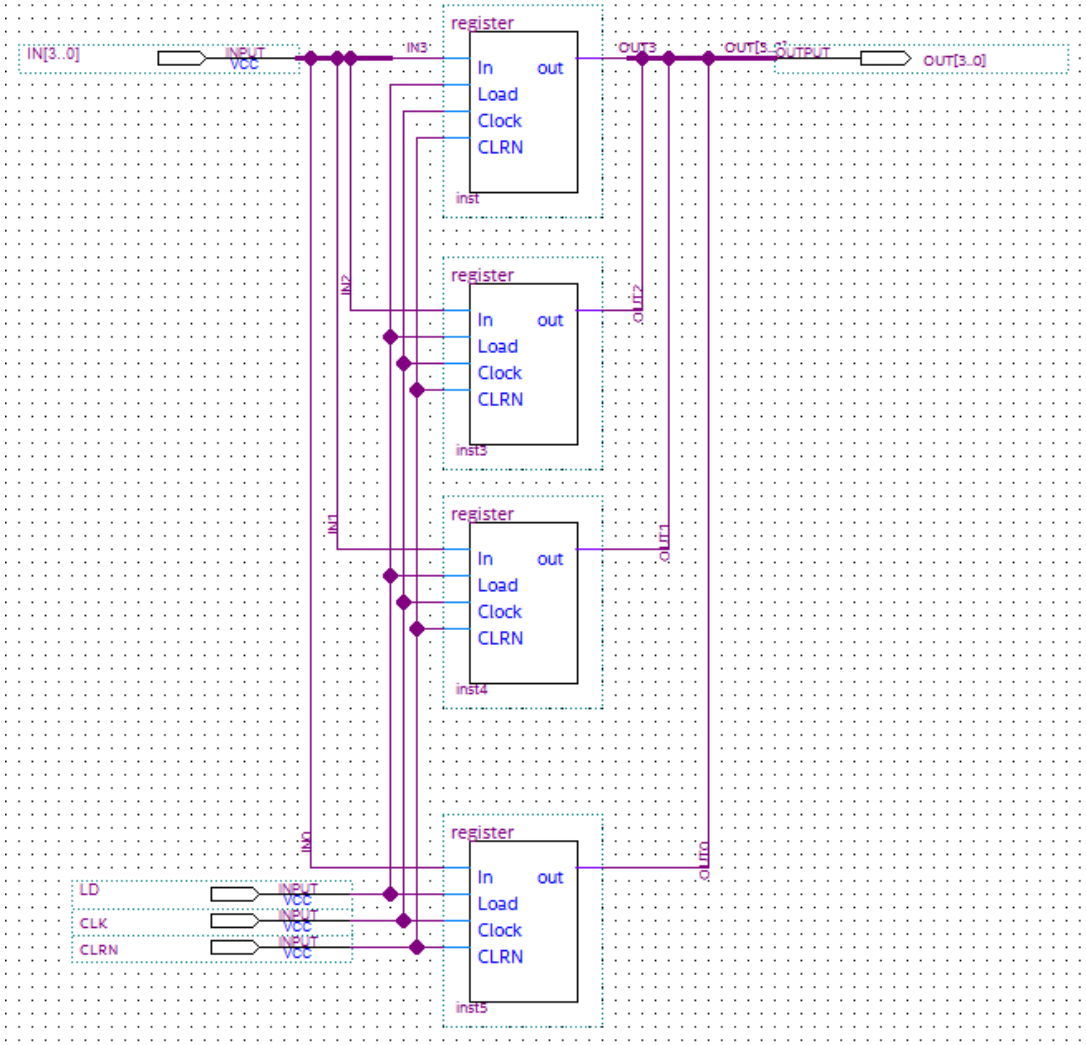
```
module Mux10_4b( W0, W1, W2, W3, W4, W5, W6, W7, W8, W9, S, F );
   input [3:0] S;
   input [7:0] W0, W1, W2, W3, W4, W5, W6, W7, W8, W9;
   output [7:0] F;
   reg [7:0] F;

   always @ ( S ) begin
      if( S == 'b1010 ) begin
         F = 'b10110111;
      end else begin
         F = S[3] ? (S[0] ? W9 : W8) : (S[2] ? (S[1] ? (S[0] ? W7 : W6) : (S[0] ? W5 : W4)) : (S[1] ? (S[0] ? W3 : W2) : (S[0] ? W1 : W0)));
      end
   end
endmodule
```

The multiplexers are 11 to 1 rather than 10 to 1 for the 10 register due to the register file requiring storage of a “magic constant” that will be utilized in the ALU later on for encryption. This magic constant is hardcoded into the multiplexer at read address 1010, past the actual registers.

Each of the individual registers is made up of two four bit registers that are in turn made up by four 1-bit registers, as shown:

Max Van de Wille
490227758
Lab Section F





Max Van de Wille
490227758
Lab Section F

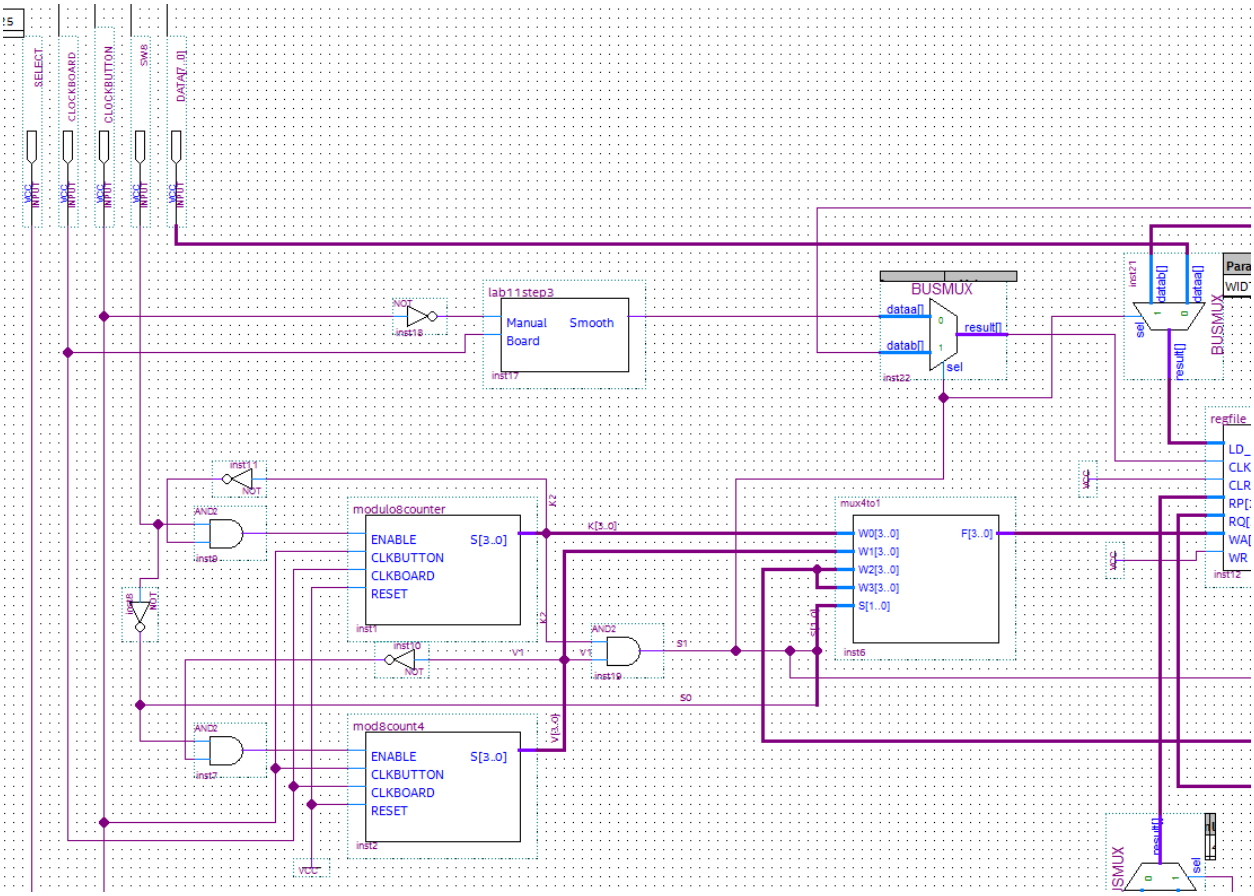
The ten individual registers are as follows:

K0	0000
K1	0001
K2	0010
K3	0011
V0	0100
V1	0101
sum	0110
tmp0	0111
tmp1	1000
tmp2	1001

Section 2: The Control Circuit

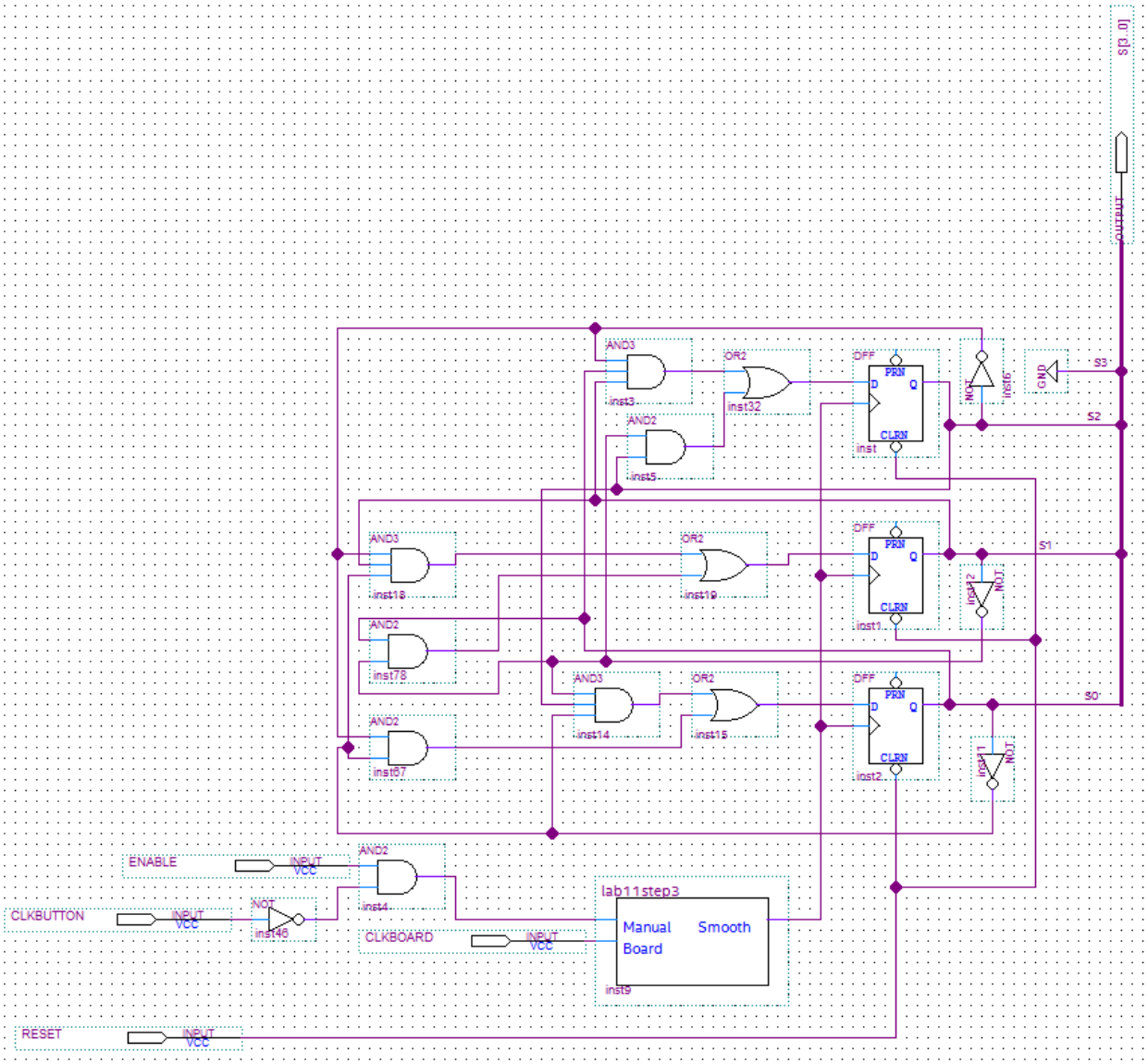
The control circuit serves the role of allowing the user to control what the circuit performs by using the designated inputs. In the case of our encryption algorithm, the user is expected to load the value integer and key 8-bits at a time using a line of switches. The circuit comes with the two additional select lines that control loading K versus V or encryption versus decryption. In order to be able to load K or V, the user loads them starting at K0 or V0 and loading up until K3 or V1, at which point encryption will begin.

Max Van de Wille
490227758
Lab Section F



Controlling the inputting of K and V is done using two counters, both with a selectable enable and shutoff feature so that the user cannot enter values past K3 and V1. The topmost counter controls counting for K and will count 0-4 and then shut itself off when it reaches the value 0100, as the 2nd most significant bit is inverted and sent into an AND gate with SW8. Thus, the counter for K is only enabled if switch8 is asserted and the counter itself has not reached 4. Alternatively, the counter for V is on the bottom, and it counts 4-6 and will automatically shut off when it reaches a value of 0101 using a similar AND gate setup for the enable. The input switch8 however is inverted so that the counter for V will only be enabled when switch8 is not asserted. Both counters are modeled after a modulo 8 counter, which originally counts from 0

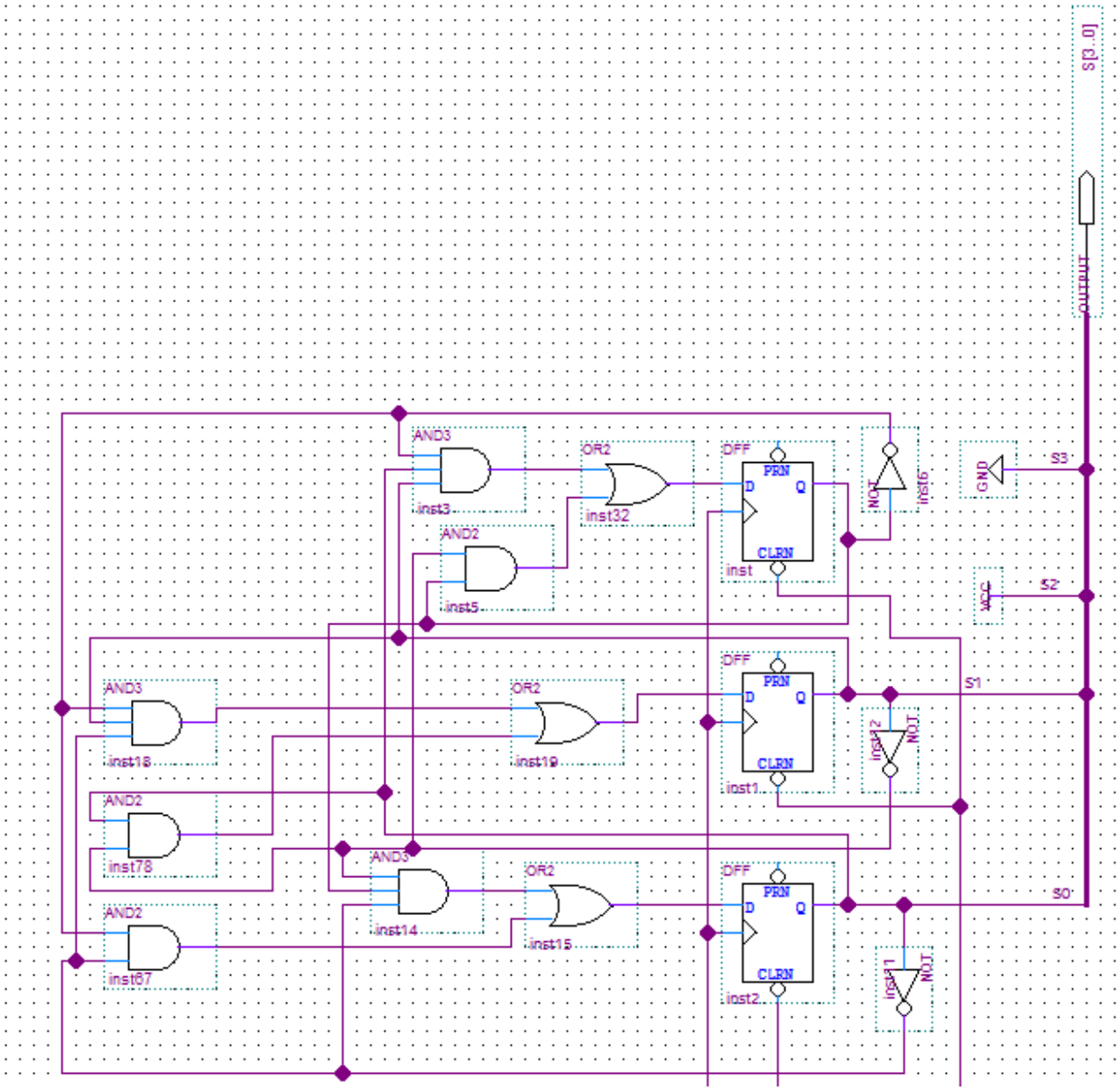
to 7. I used this counter since it allowed the appropriate range to load K0 through V1 at read addresses 0000 through 0101.



Having the counter for V set to count from 4 to 6 was done by using the exact same circuit as above, however setting the S2 wire for the output bus as VCC so it will start at 0100 rather than

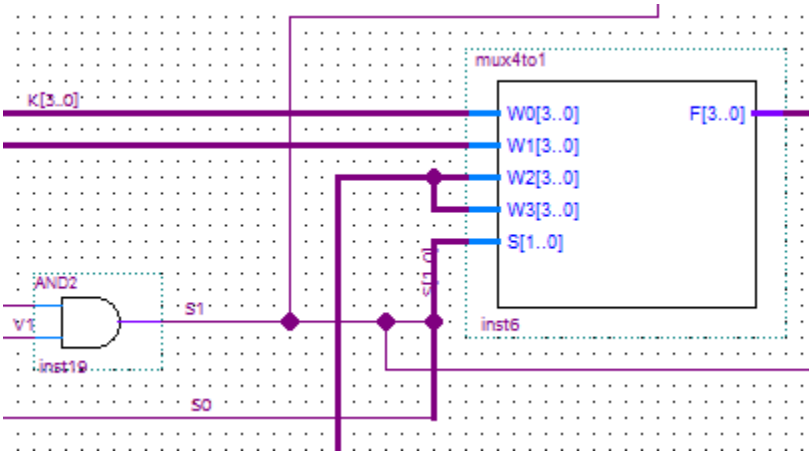
Max Van de Wille
490227758
Lab Section F

0000:



I wanted this clock to count starting at 4 so that both clocks could feed directly into the register file to use as a write address for loading the initial integers. Both of these clocks thus feed into a 4 to 1 multiplexer with 2 bit select line.

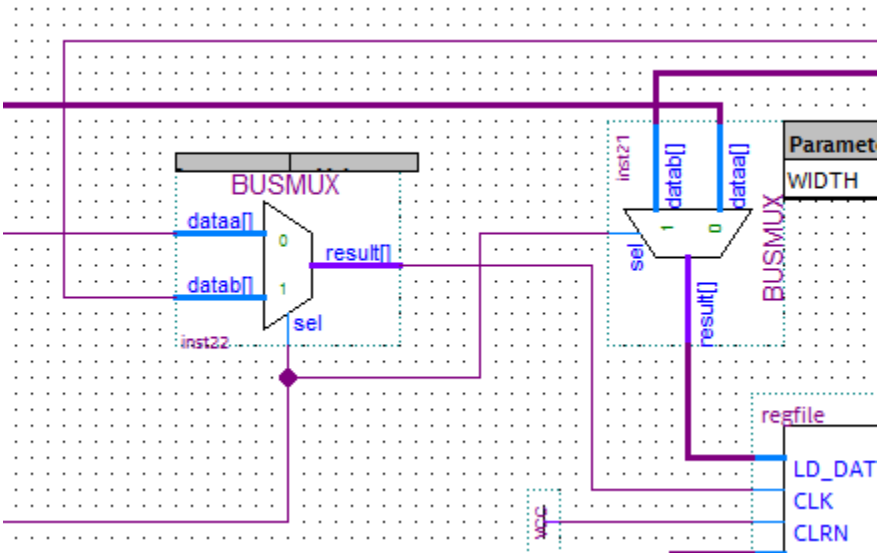
Max Van de Wille
490227758
Lab Section F



The selection bit S0 comes inverted from the SW8 input, so that if SW8 is asserted, the write address will source from the clock for K, and if it is not asserted, the write address will source from the clock for V, writing V into the register file. The second select line input, S1, comes from an AND gate that checks that both clocks have reached their final state and have shut off. Once this happens, the ALU will be activated and the register file will source its write address from the one coming out of the ALU. This is why the bus coming from the bottom of the above image accounts for W2 and W3 in the multiplexer, so that SW8 will not interfere with the ALU-sent write address.

The last portion of the control circuit involves the following two multiplexers, which control the clock and DATA input for the register file.

Max Van de Wille
490227758
Lab Section F

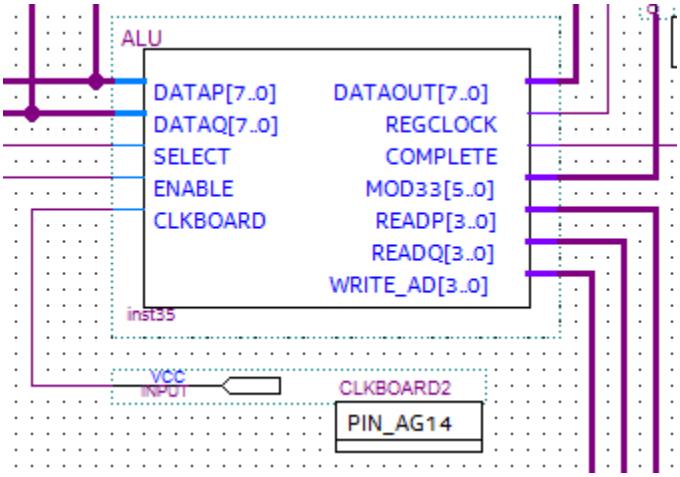


Both of these multiplexers are also selected using the previously mentioned AND gate checking if both counters are done. Thus, provided the counters have not completed we can infer that K and V have not been fully loaded, so the DATA input is sourced from the input switches on the lefthand side, and the clock is sourced from the physical clock button pushed by the user to load these values. However, once the counters have completed and the ALU is enabled, the clock and DATA inputs will come directly from the ALU, which will control reading and writing from the register file to perform the encryption.

Section 3: The ALU

It could be said that the rest of the circuit merely exists to support the ALU, similar to how the body exists to support the brain. The ALU truly is the brain of the circuit, as it performs the encryption and decryption operations.

Max Van de Wille
490227758
Lab Section F



The ALU takes in both DATAP and DATAQ sourced directly from the register file itself. It also takes in the last user-toggled switch, named SELECT. When SELECT is asserted, the circuit will be decrypting, whereas when SELECT is not asserted, the circuit is in encryption mode. This block also takes in another 50 MHz clock from the FPGA board itself. Lastly, the ALU takes in an ENABLE given by the AND gate deciding if the input clocks have completed their cycle. If they have, then all of the inputs have been loaded and the ALU may begin to process. The ALU outputs the following:

- DATAOUT – An 8-bit bus output that sends new data into the register file through a previously shown multiplexer
- REGCLOCK – A clock sourced from the ALU that is sent into the register so that the ALU itself can control writing to the register file
- COMPLETE – A signal that outputs 1 until the ALU has finished encrypting. This will be used further in the output segment of the circuit
- MOD33 – A 6-bit bus that outputs the current count of a modulo 33 clock located inside the ALU.

- READP & READQ – A new pair of read addresses set by the finite state machine in the ALU that are sent to the register file so that the ALU may read from the registers when necessary
- WRITE_AD – A 4-bit write address sent into the original 4 to 1 multiplexer of the control circuit and then sent into the register once the input clocks have completed

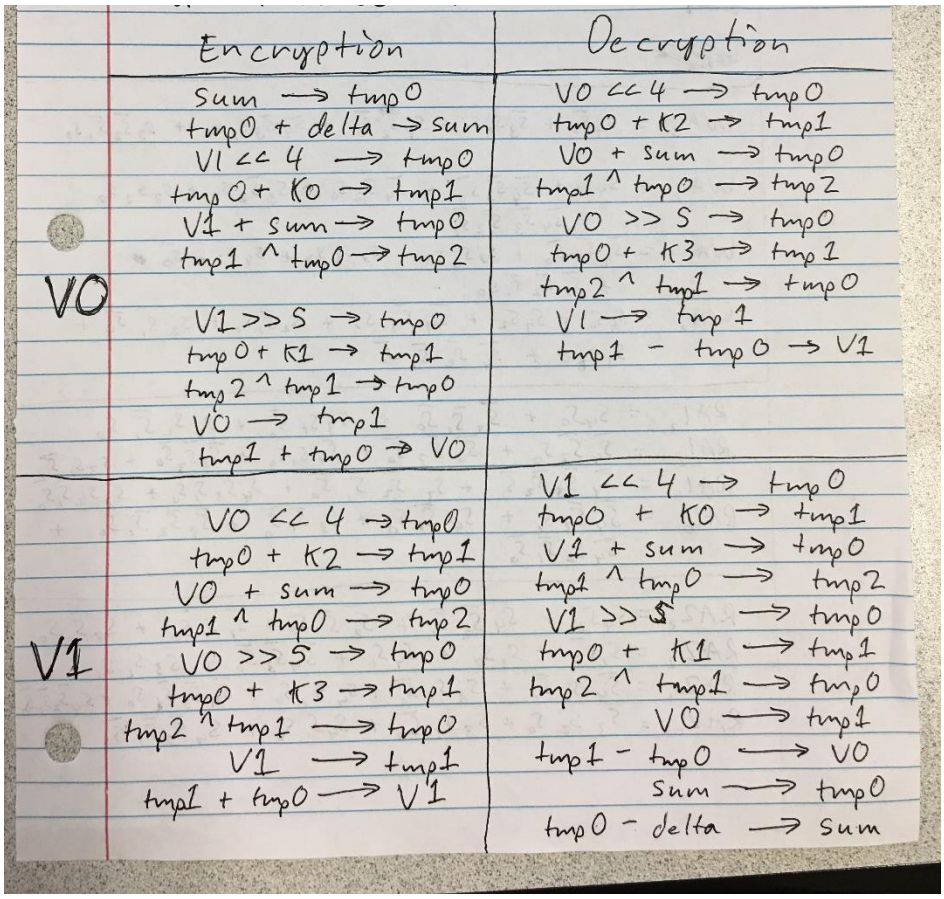
After that high level overview, we will dive into the complex circuit that makes up the ALU. As with the overall diagram, the ALU will be split into chunks that make it easier to understand, namely the control clocks, the FSM itself, and the processing blocks. Prior to that, however, I will go over my thought process behind implementing the ALU.

In the specification sheet for the project, we are given sample code that shows the functions of the tiny encryption algorithm:

```
void encrypt (uint8_t* v, uint8_t* k) {
    uint8_t v0=v[0], v1=v[1], sum=0, i;           /* set up */
    uint8_t delta=0xb7;                          /* a key schedule constant */
    uint8_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache key */
    for (i=0; i < 32; i++) {                      /* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }                                             /* end cycle */
    v[0]=v0; v[1]=v1;
}

void decrypt (uint8_t* v, uint8_t* k) {
    uint8_t v0=v[0], v1=v[1], sum=0xe0, i;  /* set up */
    uint8_t delta=0xb7;                     /* a key schedule constant */
    uint8_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache key */
    for (i=0; i<32; i++) {                        /* basic cycle start */
        v1 -= ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
        v0 -= ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        sum -= delta;
    }                                             /* end cycle */
    v[0]=v0; v[1]=v1;
}
```

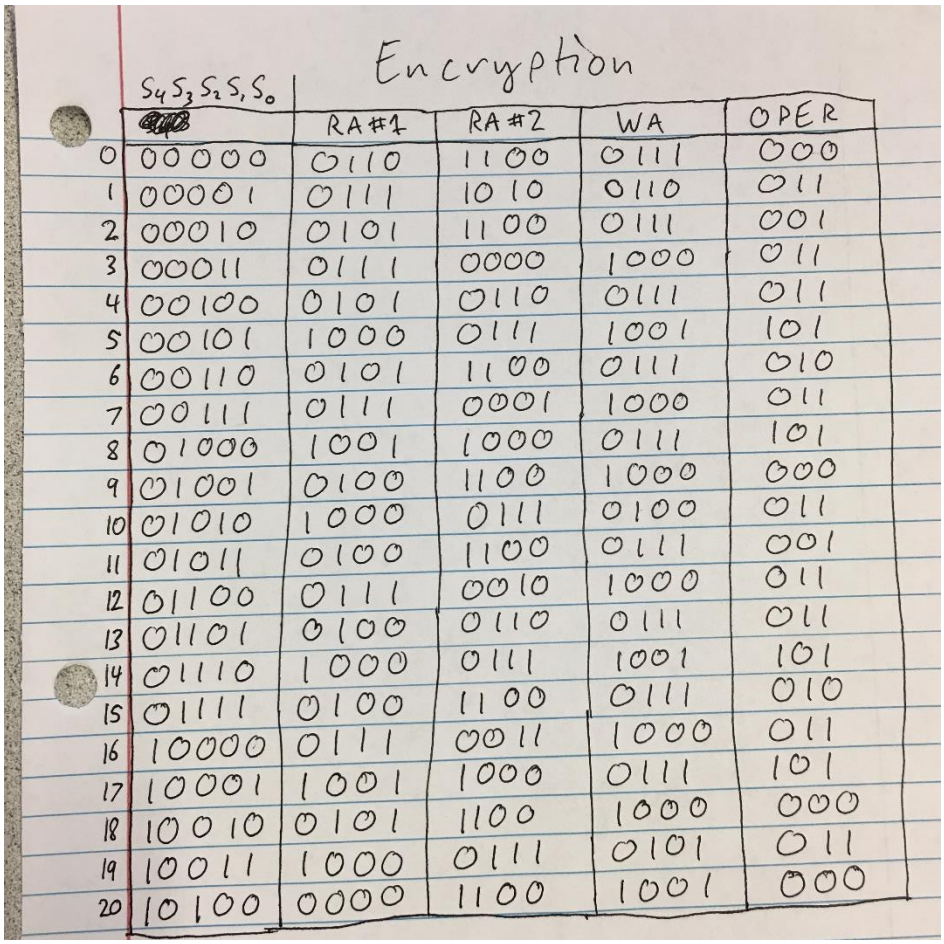
Max Van de Wille
490227758
Lab Section F

The issue that I faced with this code, however, is that it utilizes registers containing V0, V1, K0, K1, K2, K3, sum, and the magic constant all at once. Since our register file was specified to only have two read ports, however, it would be impossible to output all eight of the necessary values into an ALU as depicted by the code. Given this, I broke down the code into steps that only require reading from a maximum of two registers at a time.



From this, I created a state assigned tables for both encryption and decryption and the corresponding next state expressions and output expressions.

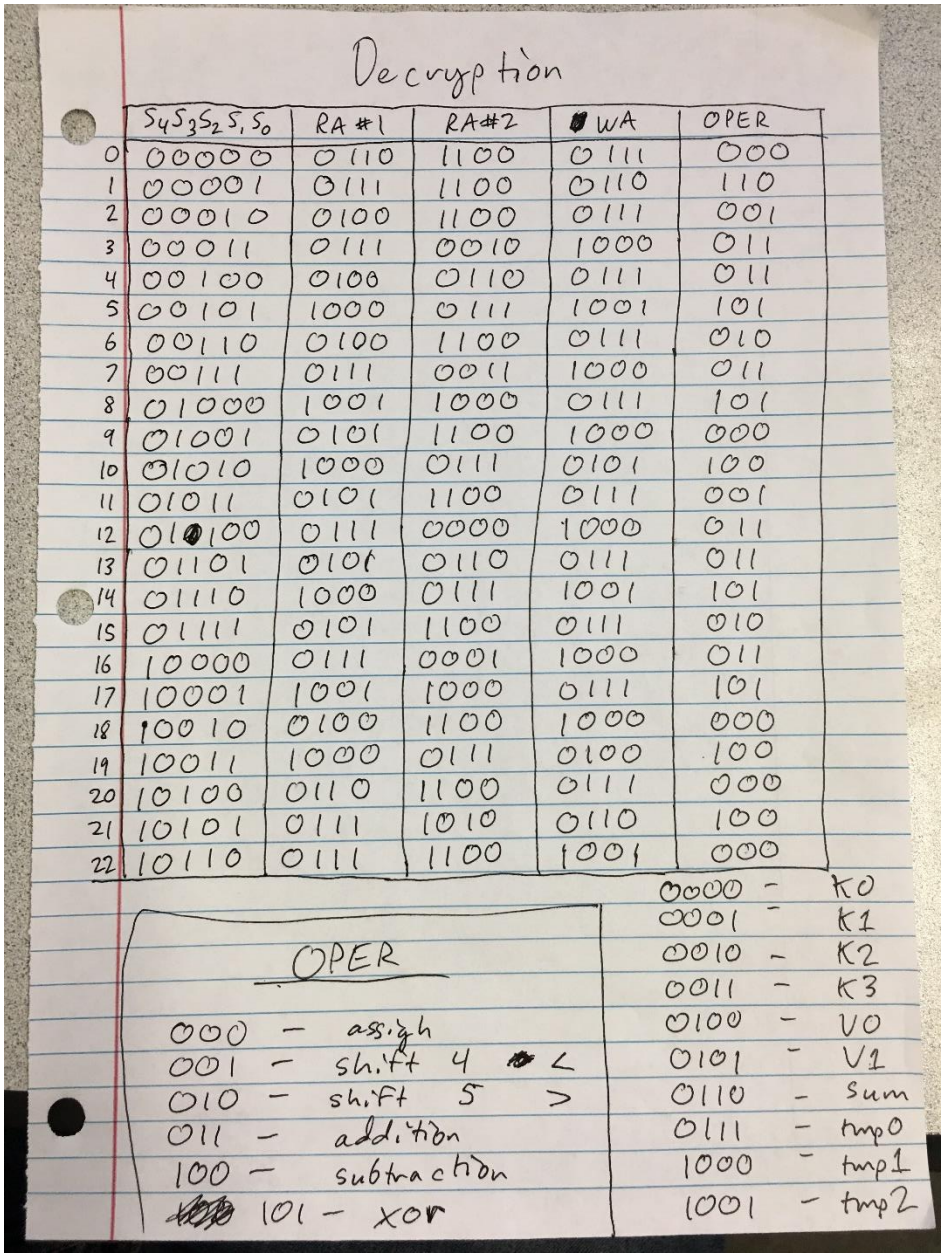
Max Van de Wille
490227758
Lab Section F



Encryption

	$S_4S_3S_2S_1S_0$	RA#1	RA#2	WA	OPER
0	00000	0110	1100	0111	000
1	00001	0111	1010	0110	011
2	00010	0101	1100	0111	001
3	00011	0111	0000	1000	011
4	00100	0101	0110	0111	011
5	00101	1000	0111	1001	101
6	00110	0101	1100	0111	010
7	00111	0111	0001	1000	011
8	01000	1001	1000	0111	101
9	01001	0100	1100	1000	000
10	01010	1000	0111	0100	011
11	01011	0100	1100	0111	001
12	01100	0111	0010	1000	011
13	01101	0100	0110	0111	011
14	01110	1000	0111	1001	101
15	01111	0100	1100	0111	010
16	10000	0111	0011	1000	011
17	10001	1001	1000	0111	101
18	10010	0101	1100	1000	000
19	10011	1000	0111	0101	011
20	10100	0000	1100	1001	000

Max Van de Wille
490227758
Lab Section F



Decryption

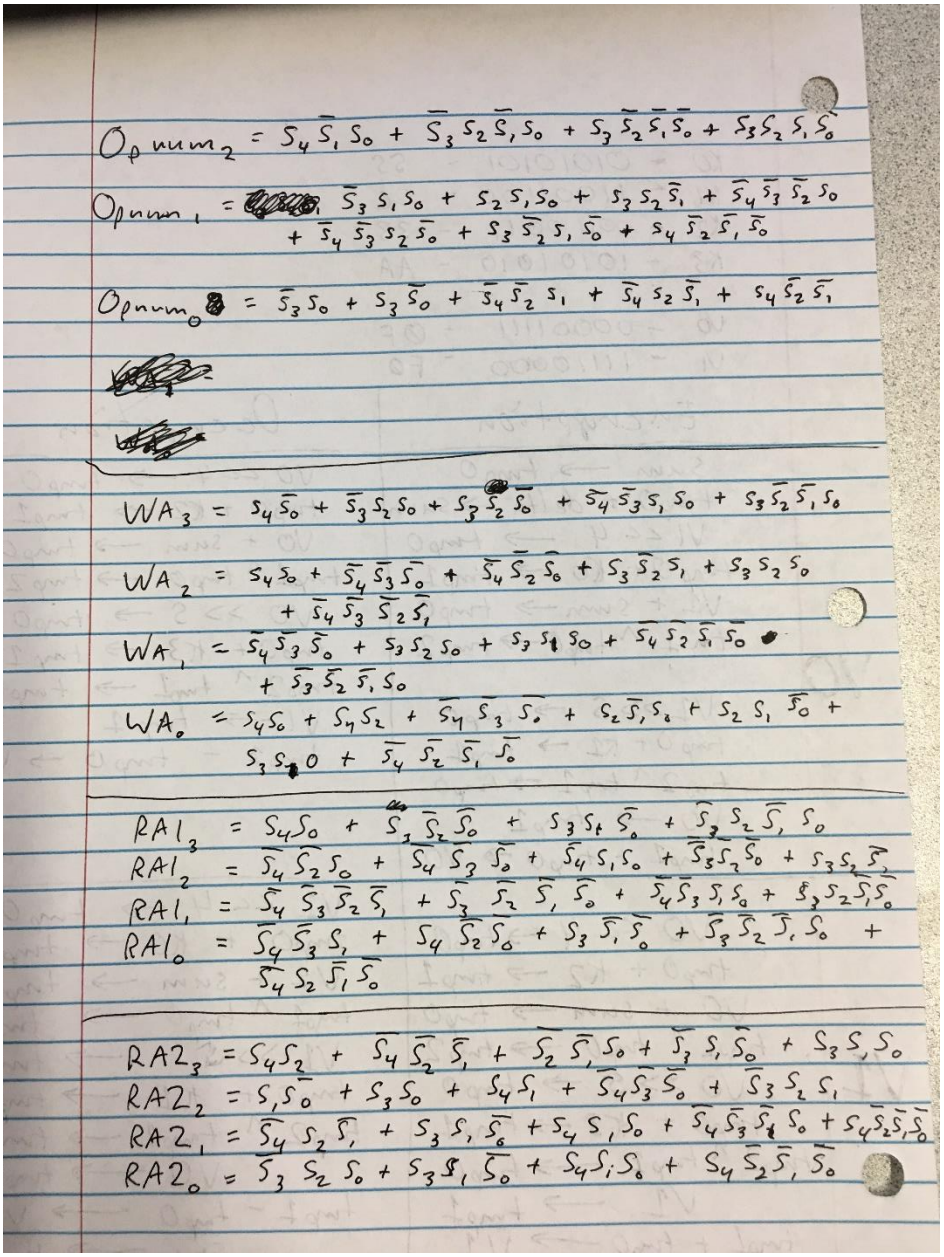
	$S_4S_3S_2S_1S_0$	RA #1	RA #2	WA	OPER
0	00000	0110	1100	0111	000
1	00001	0111	1100	0110	110
2	00010	0100	1100	0111	001
3	00011	0111	0010	1000	011
4	00100	0100	0110	0111	011
5	00101	1000	0111	1001	101
6	00110	0100	1100	0111	010
7	00111	0111	0011	1000	011
8	01000	1001	1000	0111	101
9	01001	0101	1100	1000	000
10	01010	1000	0111	0101	100
11	01011	0101	1100	0111	001
12	01100	0111	0000	1000	011
13	01101	0101	0110	0111	011
14	01110	1000	0111	1001	101
15	01111	0101	1100	0111	010
16	10000	0111	0001	1000	011
17	10001	1001	1000	0111	101
18	10010	0100	1100	1000	000
19	10011	1000	0111	0100	100
20	10100	0110	1100	0111	000
21	10101	0111	1010	0110	100
22	10110	0111	1100	1001	000

OPER

000 – assign
001 – shift 4 <
010 – shift 5 >
011 – addition
100 – subtraction
101 – xor

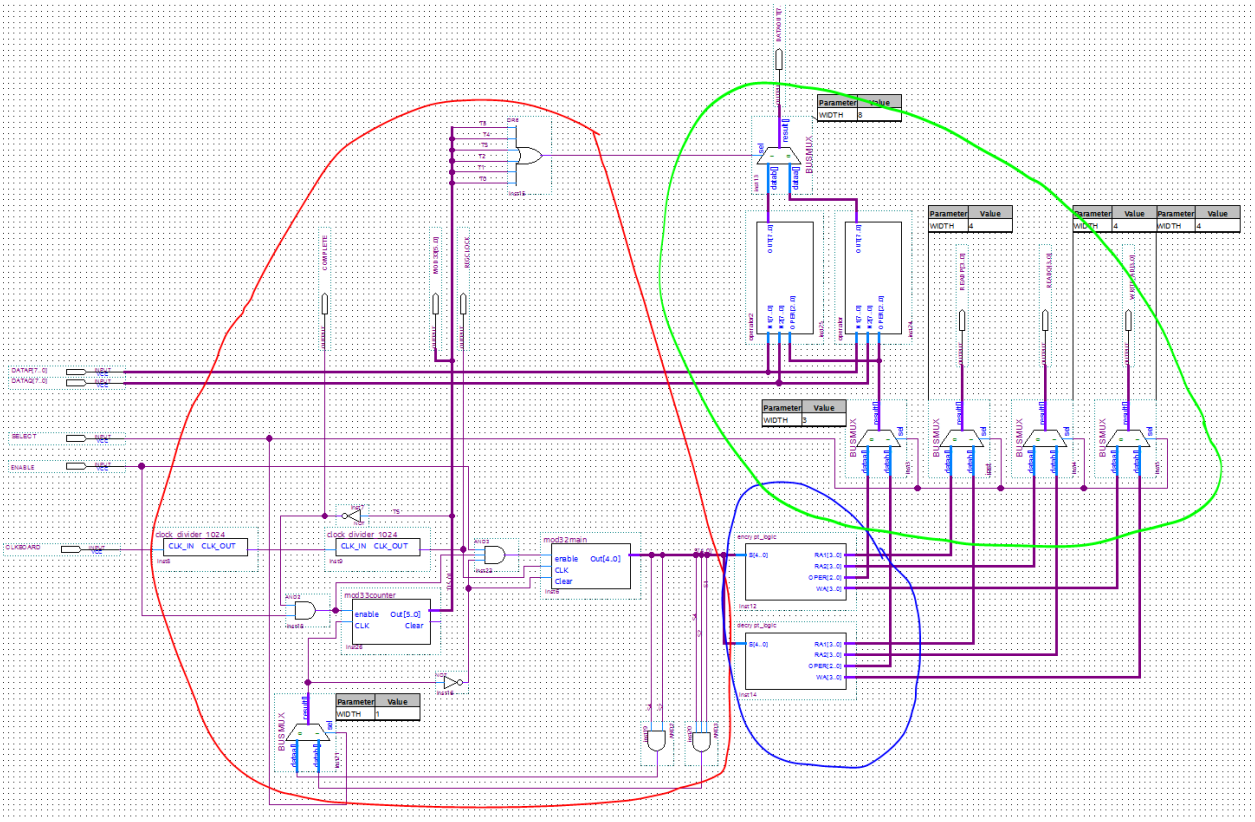
0000 – K0
0001 – K1
0010 – K2
0011 – K3
0100 – V0
0101 – V1
0110 – sum
0111 – tmp0
1000 – tmp1
1001 – tmp2

Max Van de Wille
490227758
Lab Section F



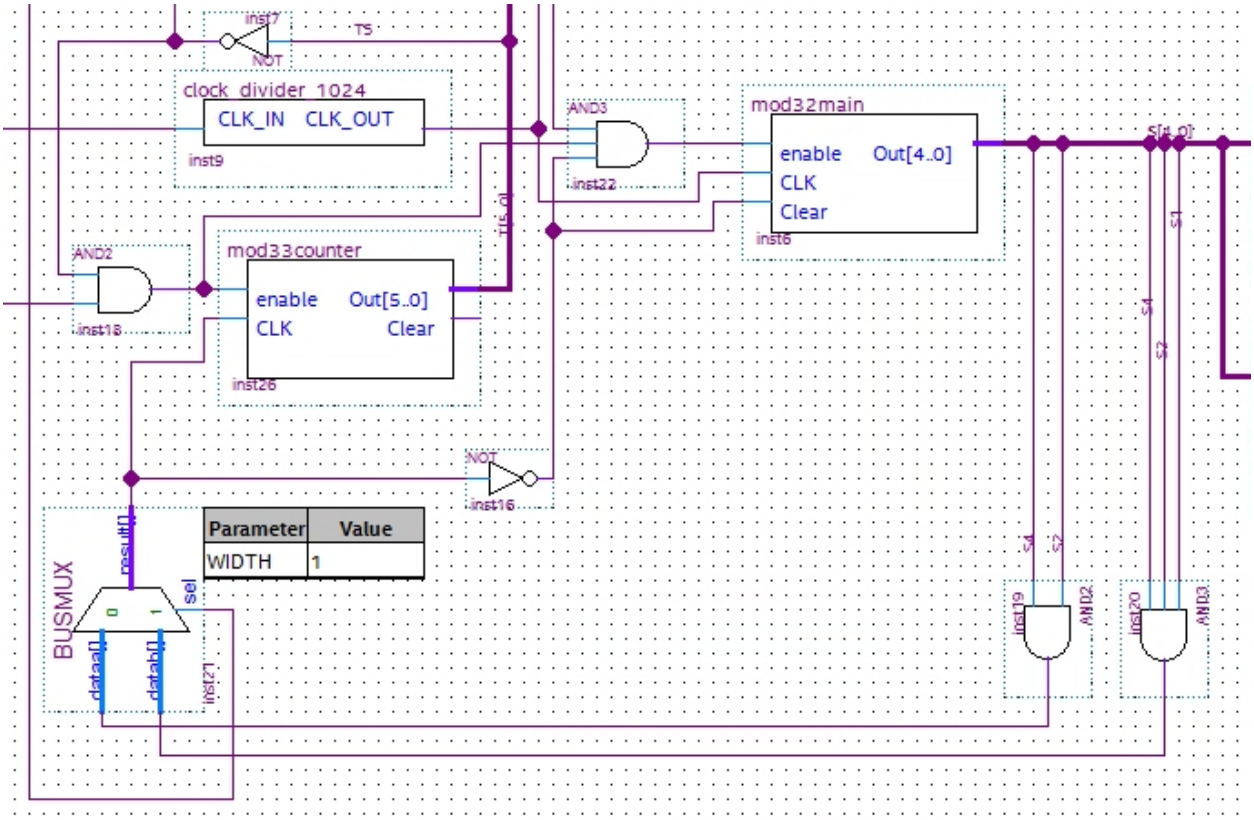
The idea behind these FSM state assigned tables is that they will take an input from a counter that can go up to 10110 and based on that clock cycle they will output two read addresses, a new write address, and an operation number that corresponds to a process in the code. So for counter cycle 0 of encryption, the process is assigning sum to tmp0, therefore the operation number is 000, which corresponds to assigning. All of this will have to be implemented in a for

loop that runs 32 times, however, as is clear in the code. Now I will give an overview of the ALU circuit itself.



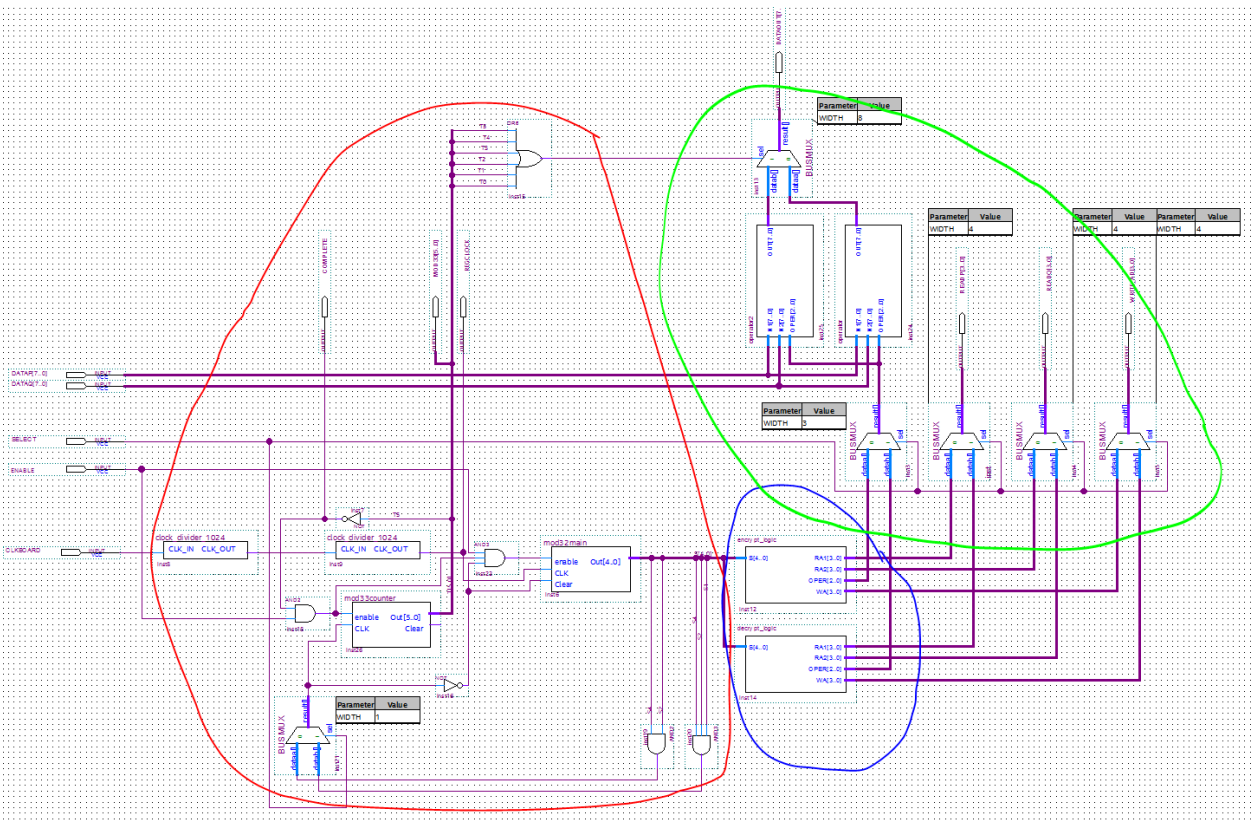
Firstly, we will go over the control counters and gates highlighted in red. Within this section of the code, the inputs enable the series of counters described above by first enabling a modulo 32 counter that is hooked up to the input board clock. This mod32 counter has an auto shutoff feature that accounts for the 21 steps of encryption and 23 steps of decryption by sending both shutoff signals into a multiplexer that is selected by the SELECT input, i.e. if the circuit is decrypting it will only send the decryption signal through once the clock hits its 23rd cycle. After this clock reaches its final state, it sends a signal into the clock of a modulo 33 counter that

enables for 32 cycles like the for loop in the code above. This step also resets the modulo 32 counter back to 0 and runs it again, imitating a for loop.



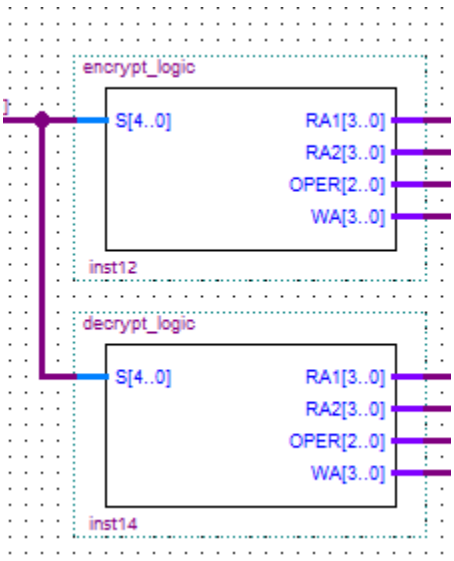
The modulo 33 clock has a shutdown feature as well, as can be seen by the NOT gate in the top of the above image. When the clock reaches 32, it will disable the counter and will send an output for COMPLETE to trigger outputting the results.

Max Van de Wille
490227758
Lab Section F



Next I will go over the function of the FSM that was touched upon earlier, highlighted in blue. The FSM is built into two separate Verilog blocks, encrypt_logic and decrypt_logic. Both of these Verilog files simply contain the output expressions for each bit of the desired outputs including the two read addresses, write address, and operation number. These blocks take in the counter signal directly from the mod32 counter and send their outputs into the actual operations portion of the ALU circuit, which is made up of some multiplexers and two Verilog blocks.

Max Van de Wille
490227758
Lab Section F

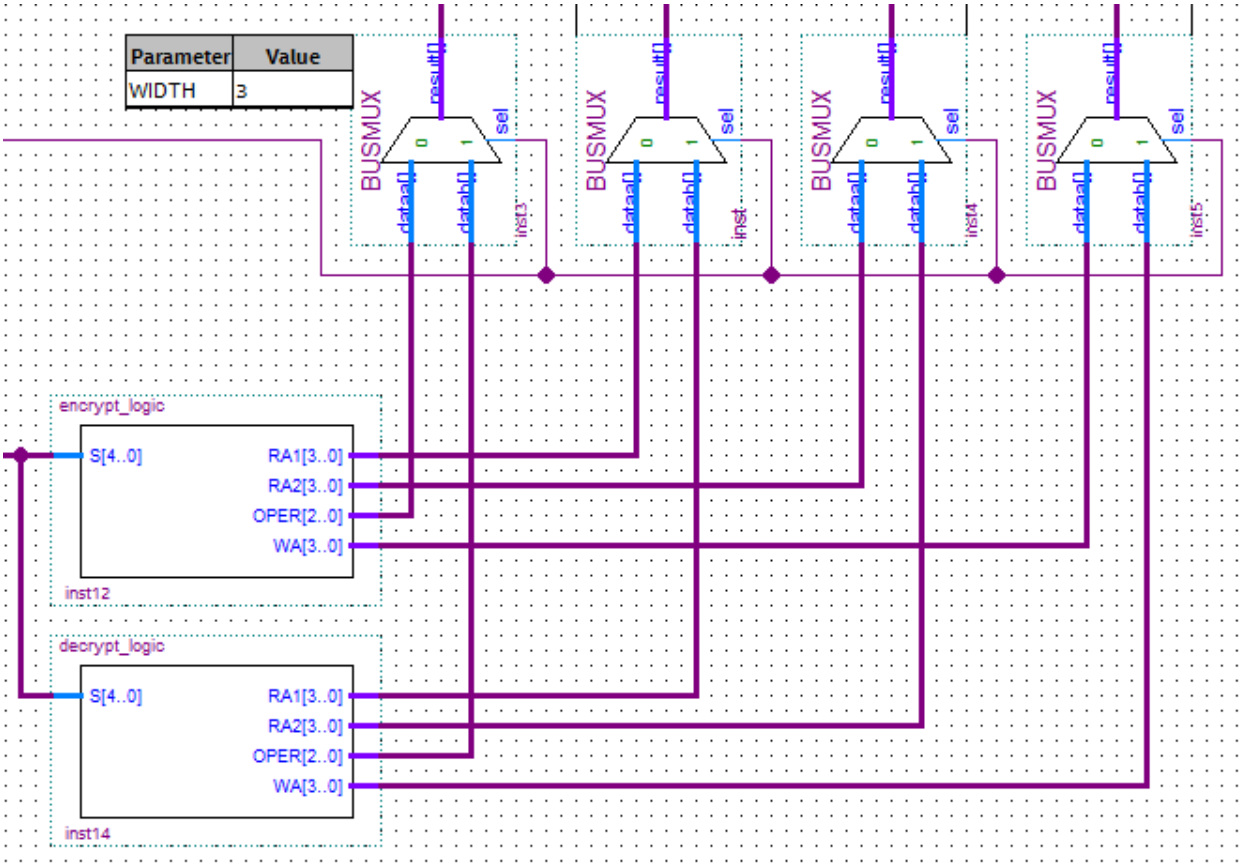



```
module encrypt_logic(S, RA1, RA2, OPER, WA);

   input [4:0] S;
   output [3:0] RA1, RA2, WA;
   output [2:0] OPER;

   assign RA1[3] = (S[4] & S[0]) | (S[3] & ~S[2] & ~S[0]) | (S[2] & S[1] & ~S[0]) | (~S[2] & S[2] & ~S[1] & S[0]);
   assign RA1[2] = (~S[4] & ~S[3] & S[0]) | (~S[4] & ~S[1] & ~S[0]) | (~S[3] & S[1] & S[0]) | (~S[3] & ~S[2] & ~S[0]) | (S[3] & S[2] & ~S[1]);
   assign RA1[1] = (~S[4] & ~S[3] & ~S[2] & ~S[1]) | (~S[3] & ~S[2] & ~S[1] & ~S[0]) | (~S[4] & ~S[3] & S[1] & S[0]) | (S[3] & S[2] & ~S[1] & ~S[0]);
   assign RA1[0] = (~S[4] & ~S[3] & S[1]) | (S[4] & ~S[2] & ~S[0]) | (S[3] & ~S[1] & ~S[0]) | (~S[3] & ~S[2] & ~S[1] & S[0]) | (~S[4] & S[2] & ~S[1] & ~S[0]);

   assign RA2[3] = (S[4] & S[3]) | (~S[4] & ~S[2] & ~S[1]) | (~S[2] & ~S[1] & S[0]) | (~S[3] & S[1] & ~S[0]) | (S[3] & S[1] & S[0]);
   assign RA2[2] = (S[3] & ~S[0]) | ( S[3] & S[0]) | (S[4] & S[1]) | (~S[4] & ~S[3] & ~S[0]) | ( ~S[3] & S[2] & ~S[1]);
   assign RA2[1] = (~S[4] & S[2] & ~S[1]) | (S[3] & S[1] & ~S[0]) | (S[4] & S[1] & S[0]) | (~S[4] & ~S[2] & ~S[1] & S[0]) | (S[4] & ~S[2] & ~S[1] & ~S[0]);
   assign RA2[0] = (~S[3] & S[2] & S[0]) | (S[3] & S[1] & ~S[0]) | (S[4] & S[1] & S[0]) | (S[4] & ~S[2] & ~S[1] & ~S[0]);

   assign OPER[2] = (S[4] & ~S[1] & S[0]) | (~S[1] & S[2] & ~S[1] & S[0]) | (S[3] & ~S[2] & ~S[1] & ~S[0]) | (S[3] & S[2] & S[1] & ~S[0]);
   assign OPER[1] = (~S[3] & S[1] & S[0]) | (S[2] & S[1] & S[0]) | (S[3] & S[2] & ~S[1]) | (~S[4] & ~S[3] & ~S[2] & S[0]) | (~S[4] & ~S[3] & S[2] & ~S[0]) | (S[3] & ~S[2] & S[1] & ~S[0]) | (S[4] & ~S[2] & ~S[1] & ~S[0]);
   assign OPER[0] = (~S[3] & S[0]) | (S[3] & ~S[0]) | (~S[4] & ~S[2] & S[1]) | (~S[4] & S[2] & ~S[1]) | (S[4] & ~S[2] & ~S[1]);

   assign WA[3] = (S[4] & ~S[0]) | (~S[3] & S[2] & S[0]) | (S[3] & S[1] & ~S[0]) | (~S[4] & ~S[3] & S[1] & S[0]) | (S[3] & ~S[2] & ~S[1] & S[0]);
   assign WA[2] = (S[4] & S[0]) | (~S[3] & ~S[2] & ~S[0]) | ( ~S[4] & ~S[2] & ~S[0]) | (S[3] & ~S[2] & S[1]) | (S[3] & S[2] & S[0]) | (~S[4] & ~S[3] & ~S[2] & ~S[1]);
   assign WA[1] = (~S[4] & ~S[3] & ~S[0]) | (S[3] & S[2] & S[0]) | (S[3] & S[1] & S[0]) | (~S[4] & ~S[2] & ~S[1] & ~S[0]) | (~S[3] & ~S[2] & ~S[1] & S[0]);
   assign WA[0] = (S[4] & S[0]) | (S[4] & S[2]) | (~S[4] & ~S[3] & ~S[0]) | ( S[2] & ~S[1] & S[0]) | (S[2] & S[1] & ~S[0]) | ( S[3] & S[1] & S[0]) | (~S[4] & ~S[2] & ~S[1] & ~S[0]);

endmodule
```

```
module decrypt_logic(S, RA1, RA2, OPER, WA);

   input [4:0] S;
   output [3:0] RA1, RA2, WA;
   output [2:0] OPER;

   assign RA1[3] = (S[3] & ~S[2] & ~S[0]) | (S[3] & S[1] & ~S[0]) | (S[4] & ~S[2] & S[0]) | (~S[4] & ~S[3] & S[2] & ~S[1] & S[0]);
   assign RA1[2] = (~S[3] & ~S[0]) | (S[4] & S[2]) | (~S[4] & S[1] & S[0]) | (~S[4] & ~S[2] & S[0]) | (S[3] & S[1] & ~S[1]);
   assign RA1[1] = (S[4] & S[2]) | (S[4] & ~S[1] & ~S[0]) | (~S[4] & ~S[3] & ~S[2] & ~S[1]) | (~S[4] & ~S[3] & S[1] & S[0]) | (S[3] & S[2] &~S[1] & ~S[0]);
   assign RA1[0] = (S[3] & ~S[1]) | (~S[4] & ~S[2] & S[0]) | (~S[4] & S[1] & S[0]) | (S[4] & ~S[2] & ~S[1]) | (S[4] & ~S[1] & S[0]) | (S[4] & S[2] & S[1]);

   assign RA2[3] = (S[4] & S[2]) | (~S[4] & ~S[2] & ~S[1]) | (~S[3] & S[1] & ~S[0]) | (~S[2] & ~S[1] & S[0]) | (S[3] & S[1] & S[0]);
   assign RA2[2] = (S[1] & ~S[0]) | (S[3] & S[0]) | (S[4] & S[1]) | (~S[4] & ~S[3] & ~S[1]) | (~S[3] & S[2] & ~S[0]);
   assign RA2[1] = (~S[3] & S[1] & S[0]) | (S[2] & ~S[1] & S[0]) | (S[3] & S[1] & ~S[0]) | (~S[4] & ~S[3] & S[2] & ~S[1]);
   assign RA2[0] = (S[3] & S[1] & ~S[0]) | (S[4] & S[1] & S[0]) | (~S[4] & ~S[3] & S[2] & S[0]) | (S[4] & ~S[2] & ~S[1] & ~S[0]);

   assign OPER[2] = (S[4] & S[0]) | (~S[3] & ~S[1] & S[0]) | (S[3] & ~S[2] & ~S[0]) | (S[3] & S[1] & ~S[0]);
   assign OPER[1] = (S[2] & S[1] & S[0]) | (S[3] & S[2] & ~S[1]) | (~S[4] & ~S[3] & ~S[2] & S[0]) | (~S[3] & S[2] & S[1]) | (~S[4] & ~S[2] & ~S[1] & S[0]);
   assign OPER[0] = (~S[4] & S[2] & ~S[1]) | (S[3] & ~S[1] & ~S[0]) | (S[3] & S[2] & ~S[0]) | (S[4] & ~S[2] & ~S[1]) | (~S[4] & ~S[3] & S[1] & S[0]) | (~S[4] & ~S[2] & S[1] & S[0]);

   assign WA[3] = (S[2] & S[2] & ~S[0]) | (S[4] & ~S[2] & ~S[0]) | (S[4] & S[1] & ~S[0]) | (~S[4] & ~S[3] & S[1] & S[0]) | (~S[4] & ~S[3] & S[2] & S[0]) | (S[3] & ~S[2] & ~S[1] & S[0]);
   assign WA[2] = (S[4] & S[0]) | (~S[4] & ~S[3] & ~S[0]) | (~S[4] & ~S[2] & ~S[0]) | (S[3] & ~S[2] & S[1]) | (S[3] & S[2] & S[0]) | (S[4] & S[2] & ~S[1]) | (~S[4] & ~S[3] & ~S[2] & ~S[1]);
   assign WA[1] = (~S[4] & ~S[3] & ~S[0]) | (S[3] & S[2] & S[0]) | (S[4] & S[2] & ~S[1]) | (S[3] & S[1] & S[0]) | (~S[3] & ~S[2] & ~S[1] & S[0]) | (~S[4] & ~S[2] & ~S[1] & ~S[0]);
   assign WA[0] = (S[3] & S[1]) | (~S[4] & ~S[2] & ~S[0]) | (~S[3] & S[2] & ~S[0]) | (~S[4] & S[2] & ~S[1] & S[0]) | (S[4] & ~S[2] & ~S[1] & S[0]);

endmodule
```

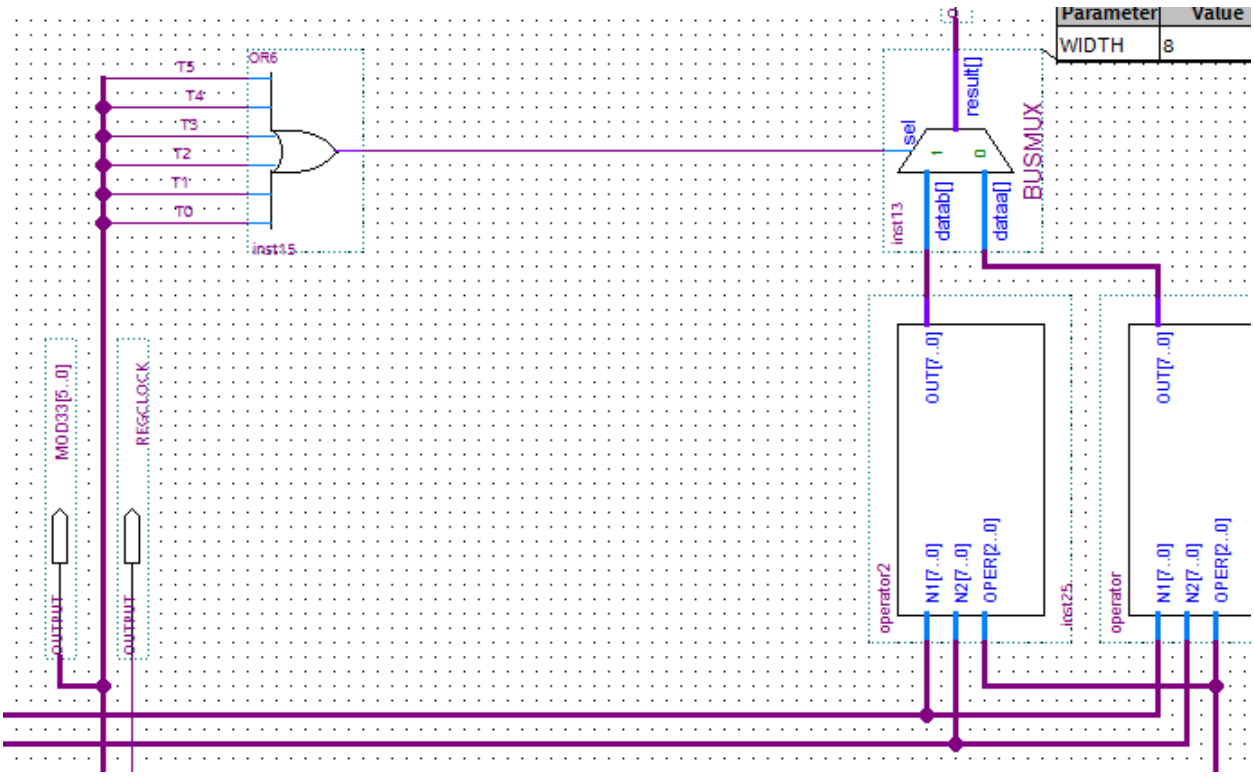
The set of multiplexers in the righthand side of the circuit are all controlled by the select line and all output one of the ALU outputs such as READP, READQ, or WA. If the SELECT line is set to encryption, only the values leaving encrypt_logic will make it through, and vice versa.

Max Van de Wille
490227758
Lab Section F



Finally, the last and most key component to the ALU is the operator blocks, which take in the 8-bit DATA input and the 3-bit operation number OPER from the FSM, and then performs functions based on the operation number. The reason for there being two of these operator blocks is that when decrypting, the sum value is set to hex e0 initially, however this cannot be part of the for loop function otherwise sum will constantly be reset to e0. Therefore, there is one operator block that performs the normal functions such as shifting, addition, subtraction, and one operator block that also has a case for setting sum equal to e0. This block is only called upon when the modulo 33 “for loop” counter is at its 0 state, and this is done in the control circuit using an OR gate to see if any of the bits in the modulo 33 have been asserted.

Max Van de Wille
490227758
Lab Section F




```
1   module operator(N1, N2, OPER, OUT);
2
3      input [7:0] N1, N2;
4      input [2:0] OPER;
5      output [7:0] OUT;
6      reg [7:0] OUT;
7
8      always @ (OPER or N1 or N2)
9         case(OPER)
10           3'b000: OUT = N1;
11           3'b001: OUT = N1<<4;
12           3'b010: OUT = N1>>5;
13           3'b011: OUT = N1 + N2;
14           3'b100: OUT = N1 - N2;
15           3'b101: OUT = N1 ^ N2;
16           3'b110: OUT = 8'b11100000;
17           default: OUT = N1;
18        endcase
19  endmodule
```

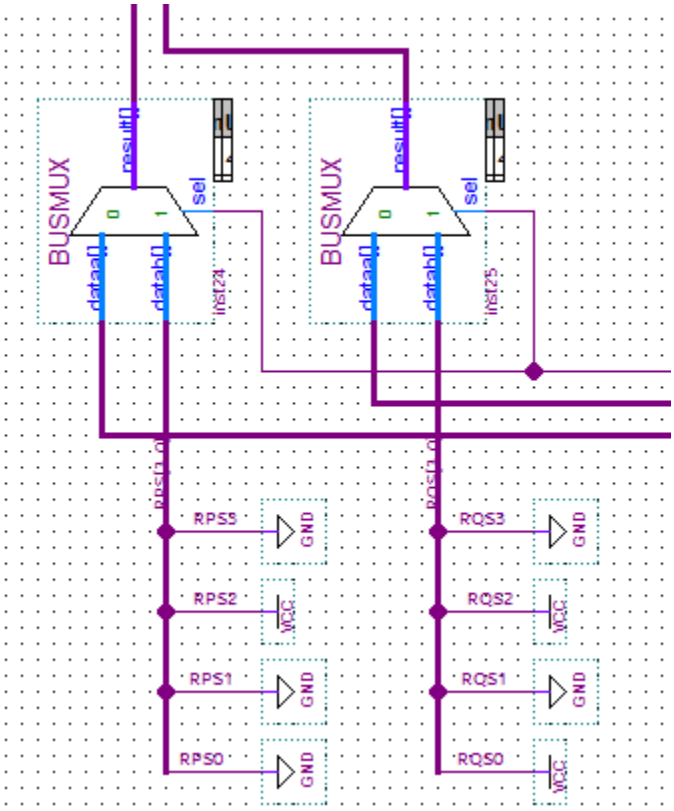
Max Van de Wille
490227758
Lab Section F

```
1   module operator2(N1, N2, OPER, OUT );
2
3      input [7:0] N1, N2;
4      input [2:0] OPER;
5      output [7:0] OUT;
6      reg [7:0] OUT;
7
8      always @ (OPER or N1 or N2)
9         case(OPER)
10           3'b000: OUT = N1;
11           3'b001: OUT = N1<<4;
12           3'b010: OUT = N1>>5;
13           3'b011: OUT = N1 + N2;
14           3'b100: OUT = N1 - N2;
15           3'b101: OUT = N1 ^ N2;
16           default: OUT = N1;
17        endcase
18  endmodule
19
```

Section 4: Output Mechanisms

The output mechanisms for the circuit are not extremely complex and consist primarily of two multiplexers, a T flip-flop, and two seven segment display decoders. The primary concern for this mechanism is that once the ALU is done encrypting, the new values for the input values have been placed in the V0 and V1 registers within the register file, so the read addresses for the register file must change to those read addresses. That is the purpose of the following mechanism, where the COMPLETE signal from the ALU is inverted and sent into the select line of the following multiplexers.

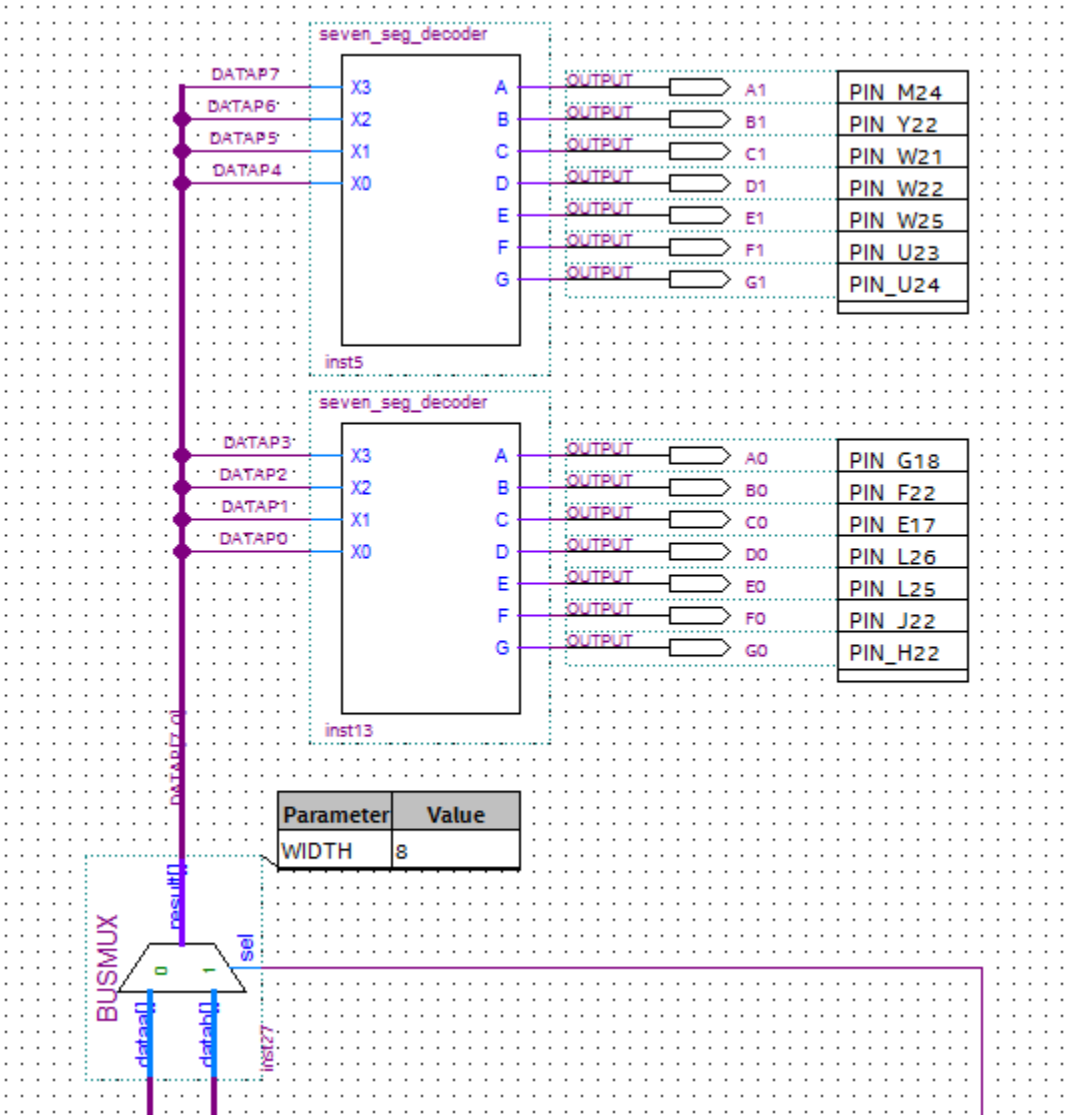
Max Van de Wille
490227758
Lab Section F



The signals coming from the right are the READP and READQ signals from the ALU, therefore when the ALU is not done, the read address will source from those set in the finite state machine, however once the ALU is done the multiplexers will switch to the hardcoded values for V0 and V1 registers.

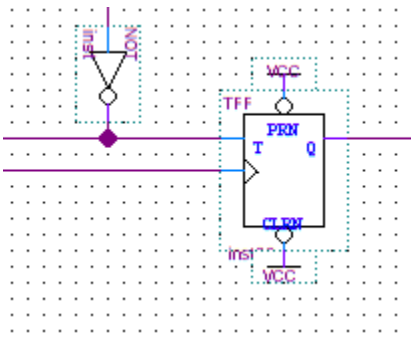
The read ports from the register file, which now output the processed V0 and V1, are connected to two seven segment displays, allowing the output integer to be displayed on two seven segment displays, with one byte showing at a time.

Max Van de Wille
490227758
Lab Section F



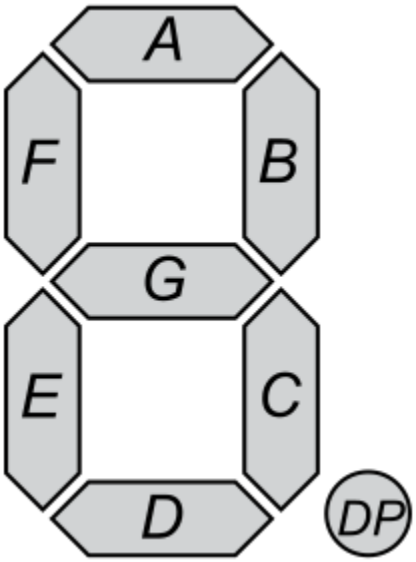
As per the specification sheet, pressing the manual clock button should toggle to the next byte in the integer, therefore I have set up a toggleable trigger using a T flip-flop to accomplish this. The read ports are hooked up to the multiplexer shown above, whose select line is connected to the output of the following T flip-flop.

Max Van de Wille
490227758
Lab Section F



The input for T comes from the inverted COMPLETE signal, which will be a 1 once the ALU has finished encrypting, therefore by pressing the manual pushbutton clock, connected into the clock input of the T flip-flop, it will then toggle which byte of data appears on the seven segment decoders.

These seven segment decoders operate on the basis of the following image, which shows how each piece of the seven segment display has its own pin that can be asserted.



Through the Verilog code below, the seven segment decoder takes the 4-bit input and expresses it using one seven segment display.

Max Van de Wille
490227758
Lab Section F

```
module seven_seg_decoder (A,B,C,D,E,F,G,X3,X2,X1,X0);

   input X3,X2,X1,X0;
   output reg A,B,C,D,E,F,G;

   always@({X3,X2,X1,X0})
   begin
      case({X3,X2,X1,X0})
         4'b0000: A='b0;
         4'b0001: A='b1;
         4'b0010: A='b0;
         4'b0011: A='b0;
         4'b0100: A='b1;
         4'b0101: A='b0;
         4'b0110: A='b0;
         4'b0111: A='b0;
         4'b1000: A='b0;
         4'b1001: A='b0;
         4'b1010: A='b0;
         4'b1011: A='b1;
         4'b1100: A='b0;
         4'b1101: A='b1;
         4'b1110: A='b0;
         4'b1111: A='b0;
      endcase
      case({X3,X2,X1,X0})
         4'b0000: B='b0;
         4'b0001: B='b0;
         4'b0010: B='b0;
         4'b0011: B='b0;
         4'b0100: B='b0;
         4'b0101: B='b1;
         4'b0110: B='b1;
         4'b0111: B='b0;
         4'b1000: B='b0;
         4'b1001: B='b0;
         4'b1010: B='b0;
         4'b1011: B='b1;
         4'b1100: B='b1;
         4'b1101: B='b0;
         4'b1110: B='b1;
         4'b1111: B='b1;
      endcase
      case({X3,X2,X1,X0})
         4'b0000: C='b0;
         4'b0001: C='b0;
         4'b0010: C='b1;
         4'b0011: C='b0;
         4'b0100: C='b0;
         4'b0101: C='b0;
         4'b0110: C='b0;
         4'b0111: C='b0;
         4'b1000: C='b0;
         4'b1001: C='b0;
         4'b1010: C='b0;
         4'b1011: C='b0;
         4'b1100: C='b1;
         4'b1101: C='b0;
         4'b1110: C='b1;
         4'b1111: C='b1;
      endcase
      case({X3,X2,X1,X0})
```

Max Van de Wille
490227758
Lab Section F

```
62          case({X3,X2,X1,X0})
63              4'b0000: D='b0;
64              4'b0001: D='b1;
65              4'b0010: D='b0;
66              4'b0011: D='b0;
67              4'b0100: D='b1;
68              4'b0101: D='b0;
69              4'b0110: D='b0;
70              4'b0111: D='b1;
71              4'b1000: D='b0;
72              4'b1001: D='b0;
73              4'b1010: D='b1;
74              4'b1011: D='b0;
75              4'b1100: D='b0;
76              4'b1101: D='b0;
77              4'b1110: D='b0;
78              4'b1111: D='b1;
79          endcase
80          case({X3,X2,X1,X0})
81              4'b0000: E='b0;
82              4'b0001: E='b1;
83              4'b0010: E='b0;
84              4'b0011: E='b1;
85              4'b0100: E='b1;
86              4'b0101: E='b1;
87              4'b0110: E='b0;
88              4'b0111: E='b1;
89              4'b1000: E='b0;
90              4'b1001: E='b1;
91              4'b1010: E='b0;
92              4'b1011: E='b0;
93              4'b1100: E='b0;
94              4'b1101: E='b0;
95              4'b1110: E='b0;
96              4'b1111: E='b0;
97          endcase
98          case({X3,X2,X1,X0})
99              4'b0000: F='b0;
100             4'b0001: F='b1;
101             4'b0010: F='b1;
102             4'b0011: F='b1;
103             4'b0100: F='b0;
104             4'b0101: F='b0;
105             4'b0110: F='b0;
106             4'b0111: F='b1;
107             4'b1000: F='b0;
108             4'b1001: F='b0;
109             4'b1010: F='b0;
110             4'b1011: F='b0;
111             4'b1100: F='b0;
112             4'b1101: F='b1;
113             4'b1110: F='b0;
114             4'b1111: F='b0;
115         endcase
116         case({X3,X2,X1,X0})
117             4'b0000: G='b1;
118             4'b0001: G='b1;
119             4'b0010: G='b0;
120             4'b0011: G='b0;
121             4'b0100: G='b0;
122             4'b0101: G='b0;
123             4'b0110: G='b0;
124             4'b0111: G='b1;
125             4'b1000: G='b0;
126             4'b1001: G='b0;
127             4'b1010: G='b0;
128             4'b1011: G='b0;
129             4'b1100: G='b1;
130             4'b1101: G='b0;
131             4'b1110: G='b0;
132             4'b1111: G='b0;
133         endcase
134     end
135 endmodule
```

All of these sections of the circuit combine to implement the tiny encryption algorithm.