Impementing DES in Integrated Circuit Hardware

CSE463-563M Spring 2025
Washington University in St. Louis
Matt Kubas, Ethan Morton, Keeler Tardiff

Contents

1 Introduction

Simplified Data Encryption Standard Algorithm or DES, is an encryption algorithm that can take a specified key and input and generate an encrypted message. We were tasked with creating a CMOS chip that could compute an encrypted 8-bit message using an 8-bit plaintext input and a defined 10-bit key. This implementation must fit in an area of 1.5mm by 1.5mm and be powered by an external 10MHz clock. The system is to output the correct string when enable is HIGH and output all zeros when enable is LOW. The process of how DES is computed was found on the ESE463 canvas page [1].

The particular chip package we are tasked on designing into is a 40 pin ASIC package with pin mapping found in table 1. The particular design we went for was a small, custom, 8-bit processor with instructions instead of a purely combinational logic implementation. This meant the system would take more clock ticks to arrive at an answer, but would save space and could be clocked much higher. The DES encryption algorithm repeats the most complex steps twice, meaning only a single implementation of the complex functions was needed to complete the encryption. The system is described in detail in the following section.

Pins	Function	Note
1-8	Plaintext	8-bit String to encrypt
9, 23-26, 39	VDD	5V power input
10, 27-30, 40	GND	Ground
11-20	Key Input	10-bit input key
21	Clock	10 MHz system clock
22	Enable	Chip enabled when HIGH
31-38	Ciphertext	Encrypted value output

Table 1: Pin Mapping for the 40-pin chip package

The first step in DES is to generate the keys and complete the initial permutation. The

2 Architecture

The overall architecture in block diagram form can be found in fig. 1. This diagram shows the overall function of the processor we have constructed. Each block and its schematic construction will be described in both form and function. Each clock cycle a single instruction is completed. The following section will describe the operations of each block in the diagram.

2.1 IN

The IN block performs the Initial permutation on the input plaintext and also computes k_1 and k_2, since their computation is simply a binary rearrangement. The keys are generated by first performing the *P10* permutation, then for k_1, a left circular shift is performed on the right 5 and left 5 bits separately, and finally the *P8* permutation is performed to generate k_1. For k_2 a 3 bit left circular shift is performed for the left and right 5 bits separately and the *P8* permutation is performed resulting in k_2. The plain

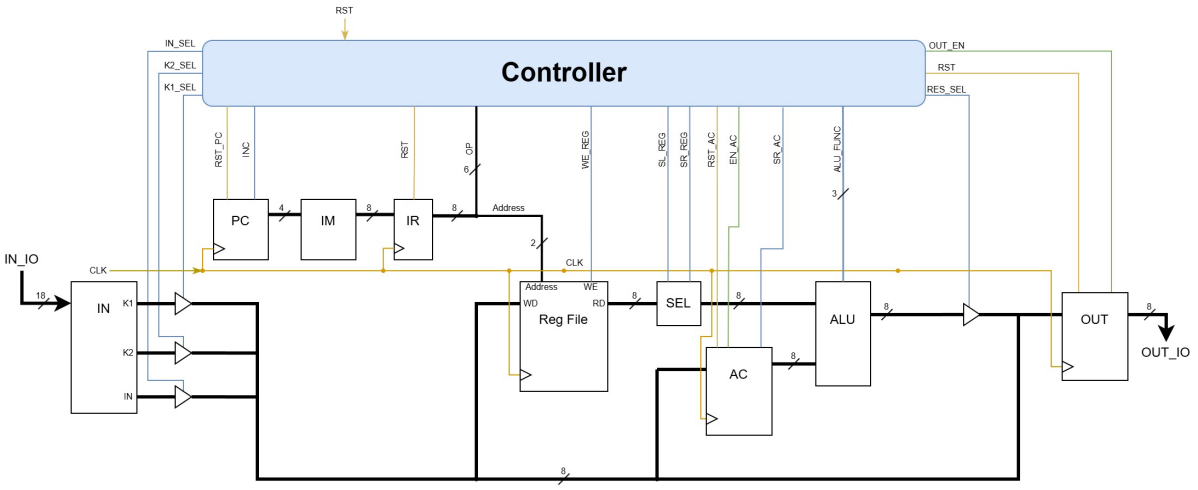

Figure 1: Block diagram of the architecture

	Arithmetic Operation	Assembly Instruction	OP Code
1		reg0 <- k1	10010000
2		reg1 <- k2	10110001
3		reg2 & AC <- IP(In)	11011010
4	E/P(AC[3:0])	AC <- Result	00001100
5	K1(reg0) $\oplus$ AC	AC <- Result	10001100
6	P4/SBOXES(AC)	AC <- Result	00101100
7	AC $\oplus$ reg_2[7:4]	AC <- Result	10101110
8	PAC{reg_2[3:0],AC[3:0]}	reg2 & AC <- Result	01011110
9	E/P(AC[3:0])	AC <- Result	00001100
10	K2(reg1) $\oplus$ AC	AC <- Result	10001101
11	P4/SBOXES(AC)	AC <- Result	00101100
12	AC $\oplus$ reg_2[7:4]	AC <- Result	10101110
13	SW{AC[3:0], reg_2[3:0]}	Out <-Result	01100110

Table 2: The processor operations for each clock cycle

text is permuted used the *IP* permutation (or initial permutation). These permutation definitions were found on the *Simplified DES algorithm* document found on the course canvas page.

2.2 PC

The program counter is a set of full adders that combinationally add one every clock tick, if the increment signal is on. The output of this addition is latched on every clock tick, with no overflow checking. Since we have 13 instructions, we implemented a 4 bit counter. The controller will reset the counter at the 13th clock cycle (synchronously resettable flip flops were used), or in the event of some unforeseen error, it will overflow and loop back to zero after 16 counts.

2.3 IM

The instruction memory block holds the specific set of instructions defined in table 2 with a set of 13 8-bit registers and 8 16:1 multiplexers, which are controlled by the program counter. The specifics of constructing the opcode from the necessary operation is described in the controller block description.

2.4 IR

The instruction register holds the current instruction. Holding the current instruction in a register provides a more stable output than the muxes and allows for easier debugging of control signals.

2.5 Regfile

The register file is a set of 3 8-bit registers allowing the reading and writing of 3 different values. Although 4 would be a more even number, our system only needs 3, so there is no point in wasting space by adding another. If this were a true microcontroller, we would add more registers for use.

2.6 SEL

The select module takes in an 8bit input, and selects one of three output formats: all 8-bits untouched; the bottom 4-bits untouched and the upper 4 output bits set to zero; or the top 4-bits moved to the bottom 4-bits, and the top 4 output bits set to zero. This block has inputs of *SL* (select left) and *SR* (select right). If either one is held HIGH the particular operation would happen, but if both wires are LOW, the block does not modify the value. Both control signals HIGH is an invalid state.

2.7 AC

The accumulator is a 8-bit register that holds a value for the bottom input of the ALU. The accumulator also has a SEL module. This block is synchronously reset and is triggered by the rising edge.

2.8 ALU

The arithmetic module consists of five operations: XOR, SBoxes/P4, E/P, SW, and PAC. The module has two 8-bit inputs, T (top) and B (bottom), 3-bit control input, and an 8-bit output.

XOR performs a bit-wise XOR operation, e.g., performing the operation $T \oplus B$ for ALU opcodes 100 and 101 (this was done to simplify the control logic).

The most complex operation was the P4(SBoxes). This operation uses all 8-bits of the B input, performs the SBoxes operation, then the *P4* permutation, and outputs a 4-bit value on the output. Because the values of the SBoxes are permanently static, they are implemented by simple 16:1 muxes connected to vdd or gnd. The *P4* permutation is performed with the output of the LUTs since it is simply a rearrangement of bits.

E/P is and expansion and permutation of the bottom 4-bits of the B input. This results in an 8-bit value on the output.

SW takes the lowest lowest 4-bits of B and the 4-bits of T and concatenates them together to form an 8-bit value. In this case the output has B as the upper-most bits and T has the lowest bits. This operation in Verilog notation would appear as: `{B[3:0],T[3:0]}`.

PAC does a similar concatenation to SW but places T as the uppermost bits and B as the lower bits. In Verilog the operation would appear as: `{T[3:0],B[3:0]}`.

2.9 OUT

The output module is a register that stores the values of the finished encryption in a register but also performs the inverse initial permutation or IP^{-1}.

2.10 CONTROLLER

The controller takes the instruction from the instruction register (IR) and outputs the correct control logic HIGH and LOW for each control wire. The 8-bit Opcode is constructed such that the upper six bits are the Op code to the controller and the lower 2-bits are the register address. The upper three bits of the OP code are also directly fed to the ALU to control which operation is performed but are also reused to select which input is fed onto the bus. The full timing diagram of which inputs are held HIGH at each point can be found at table 3. The instruction is constructed with a 6-bit OP code and a 2-bit address defined by the following Verilog code: `{OP5,OP4,OP3,OP2,OP1,OP0,A1,A0}`. The combinational logic equations can be found in table 4..

	Instructions	RST_PC	IN_SEL	K1_SEL	K2_SEL	A_REG	WE_REG	SL_REG	SR_REG	RST_AC	EN_AC	SR_AC	ALU_FUNC	RES_SEL	OUT_EN
1	10010000	0	0	1	0	00	1	0	0	0	0	0	000	0	0
2	10110001	0	0	0	1	01	1	0	0	0	0	0	000	0	0
3	11011010	0	1	0	0	10	1	0	0	0	1	0	000	0	0
4	00001100	0	0	0	0	00	0	0	0	0	1	1	000	1	0
5	10001100	0	0	0	0	00	0	0	0	0	1	0	100	1	0
6	00101100	0	0	0	0	00	0	0	0	0	1	0	001	1	0
7	10101110	0	0	0	0	10	0	1	0	0	1	0	100	1	0
8	01011110	0	0	0	0	10	1	0	1	0	1	1	010	1	0
9	00001100	0	0	0	0	00	0	0	0	0	1	1	000	1	0
10	10001101	0	0	0	0	01	0	0	0	0	1	0	100	1	0
11	00101100	0	0	0	0	00	0	0	0	0	1	0	001	1	0
12	10101110	0	0	0	0	10	0	1	0	0	1	0	100	1	0
13	01100110	1	0	0	0	10	0	0	1	1	0	1	011	1	1

Table 3: OP Code to Controller Outputs

$RST\_PC = (OP0 \wedge \overline{OP5} \wedge OP4 \wedge OP3) \vee \overline{RST}$	$K1\_SEL = \overline{OP0} \wedge OP5 \wedge \overline{OP4} \wedge OP3$
$IN\_SEL = (\overline{OP0} \wedge OP5 \wedge \overline{OP4} \wedge \overline{OP3}) \vee \overline{RST}$	$K2\_SEL = \overline{OP0} \wedge OP5 \wedge OP4 \wedge \overline{OP3}$
$A\_REG = \{A1, A0\}$	$WE\_REG = OP2$
$SL\_REG = OP0 \wedge OP5 \wedge \overline{OP4} \wedge OP3$	$SR\_REG = OP0 \wedge \overline{OP5} \wedge OP4$
$RST\_AC = OP0 \wedge \overline{OP5} \wedge OP4 \wedge OP3$	$EN\_AC = OP1$
$SR\_AC = OP0 \wedge \overline{OP5} \wedge (OP4 \vee \overline{OP3})$	$ALU\_FUNC = \{OP5, OP4, OP3\}$
$RES\_SEL = OP0$	$OUT\_EN = OP0 \wedge \overline{OP5} \wedge OP4 \wedge OP3$

Table 4: Equation for the combinational logic for each control wire

3 Implementation

To simplify the construction of each block in Cadence and to allow for asynchronous construction of the processor, each block was made separately and later combined in the master schematic which can be found at fig. 2 . The following sections will have screen captures of each component in schematic and layout form to perform the operation described by the above. The goal was to recreate the initial diagram in Cadence so that it was easier to debug the system.

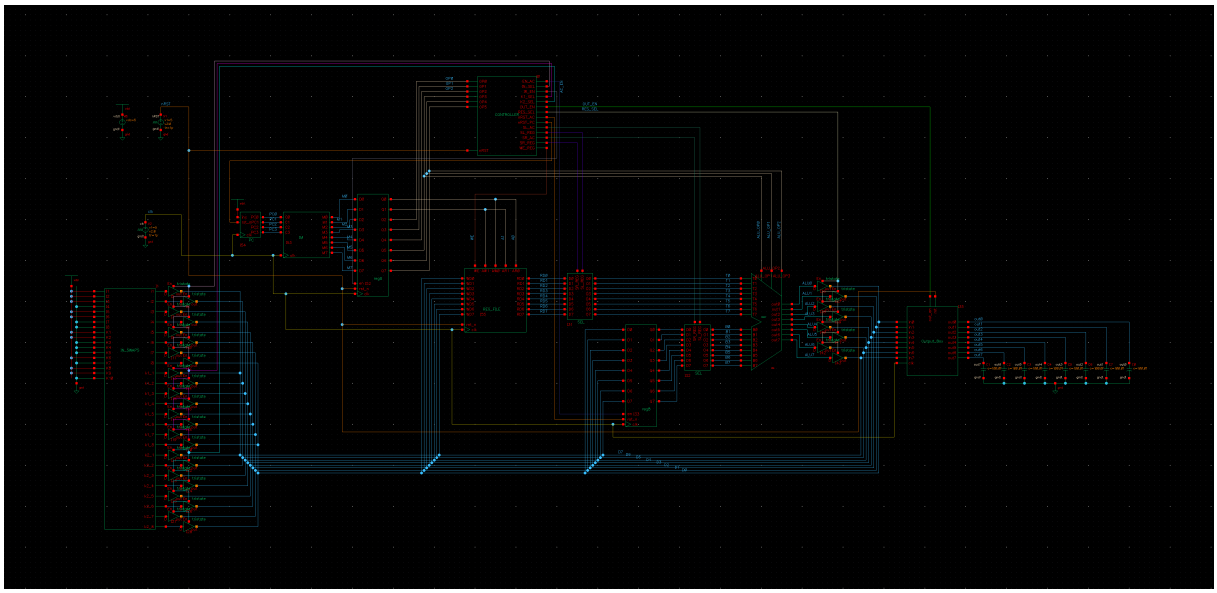

Figure 2: Processor Schematic in Cadence

3.1 Basic Components

Below are the schematics of the inverter and transmission gate. These are used on future designs and are simply introduced here.

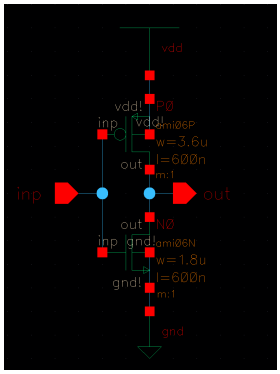

Figure 3: Inverter Schematic

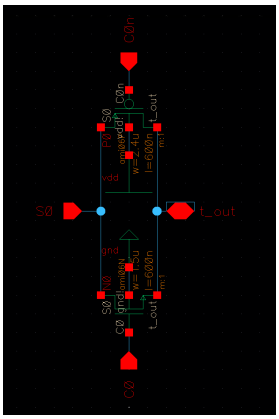

Figure 4: Transmission gate schematic

Again, the basic logic gates used include the NOR and NAND gates, as developed during the semester on homeworks.

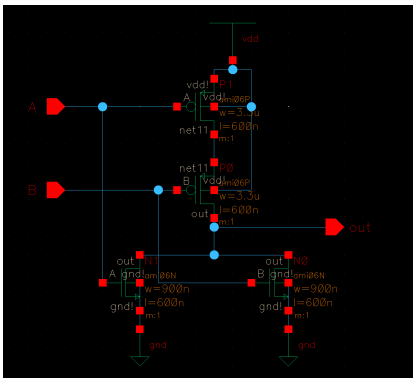

Figure 5: 2-input NOR gate Schematic

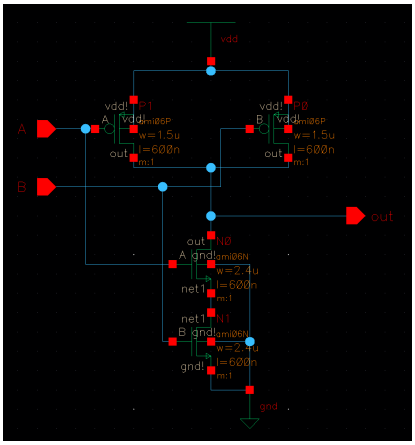

Figure 6: 2-input NAND gate schematic

3.2 Multiplexers

The multiplexers are designed using transmission gates. For ease of use in the schematic, we used the transmission gate shown in fig. 4 and used the transmission gate symbol in the schematic. For each level of multiplexer (2:1, 4:1, 8:1, 16:1) the previous level stage was used along with another 2:1 mux, as shown in figs. 8 to 10.

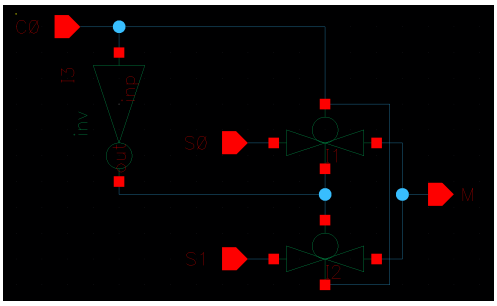

Figure 7: 2 to 1 MUX schematic

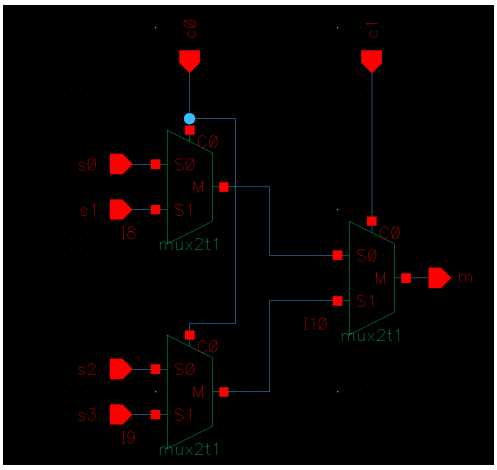

Figure 8: 4 to 1 MUX schematic

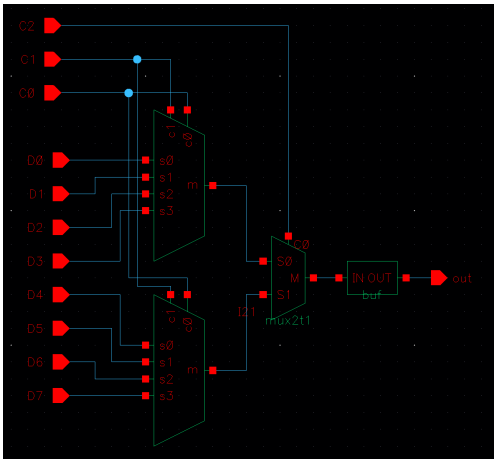

Figure 9: 8 to 1 MUX schematic

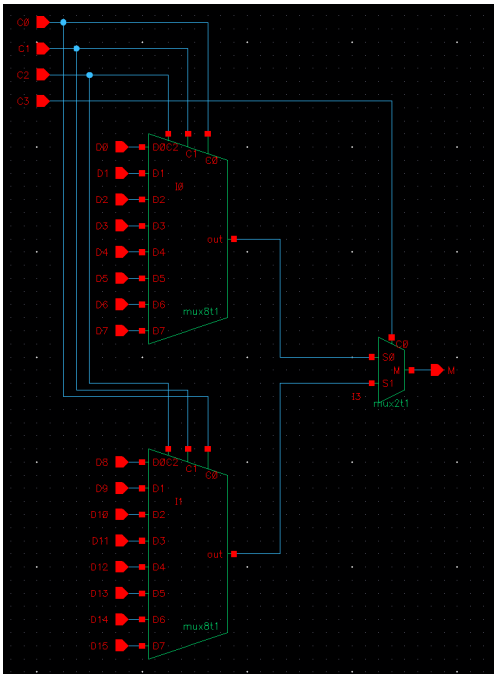

Figure 10: 16 to 1 MUX schematic

3.3 De-multiplexers

Originally we used transmission gates for 1:N De-Multiplexer but ran into issues with charge being trapped between gates. So, we switched to a combinational logic approach, where the 1:2 demux is defined with basic boolean logic, and the higher levels (1:4, 1:8, 1:16) are again built on top of the 1:2 demux in a hierarchical fashion.

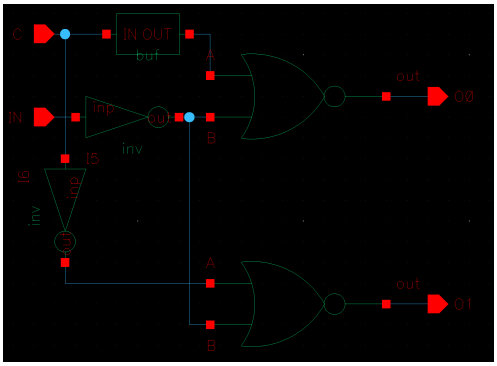

Figure 11: 1 to 4 De-MUX schematic

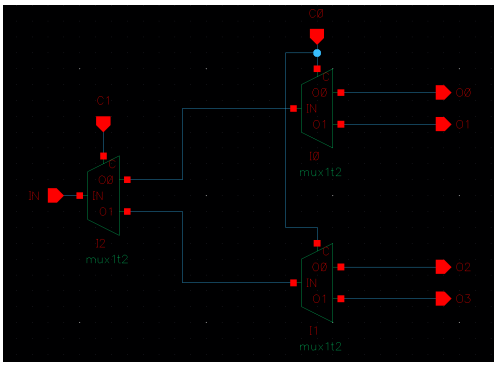

Figure 12: 4 to 1 De-MUX schematic

3.4 Buffers

We constructed a buffer using two inverters due to signal loss through the multiplexors. The buffer only add a delay of a few nanoseconds, so by using them sparingly, we stabilize crucial signals inside the processor.

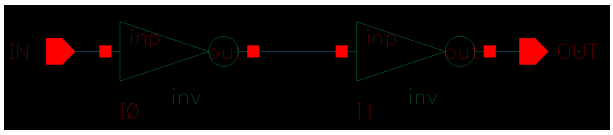

Figure 13: Buffer schematic

3.5 Registers

The registers were implemented using the master-slave D flip-flop from homework 7. Using this flip-flop along with two multiplexers allowed for the implementation of the enable and reset control signals, making a "proper" 1-bit register, as seen in fig. 14.

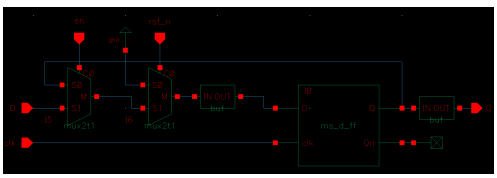

Figure 14: 1-bit register schematic

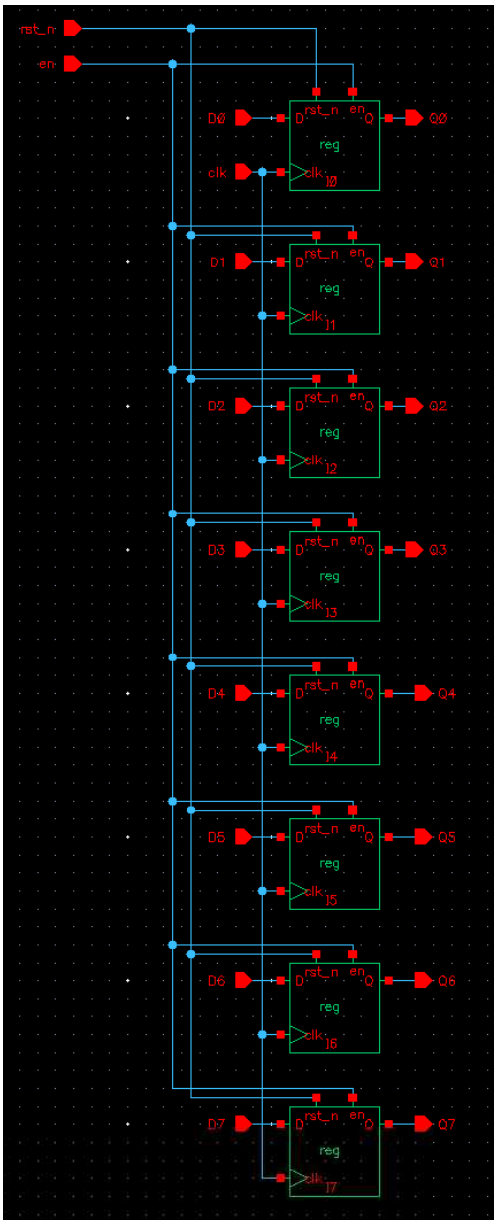

Figure 15: 8-bit register schematic

Multi-bit registers are implemented by simply arranging the 1-bit registers together and connecting their control bits. By doing this, they act as a single multi-bit register.

3.6 ALU Operations

The ALU was boken up into blocks, which will be shown shortly, but the final diagram had the following schematic.

3.6.1 XOR

By searching Google for XOR schematics, we were able to find a suitable schematic that simulated correctly in our implementation. The specific implementation can be seen in fig. 17. We built a bitwise XOR operation by placing 8 of these with the inputs connected to the two 8-bit inputs. This can be seen in fig. 18.

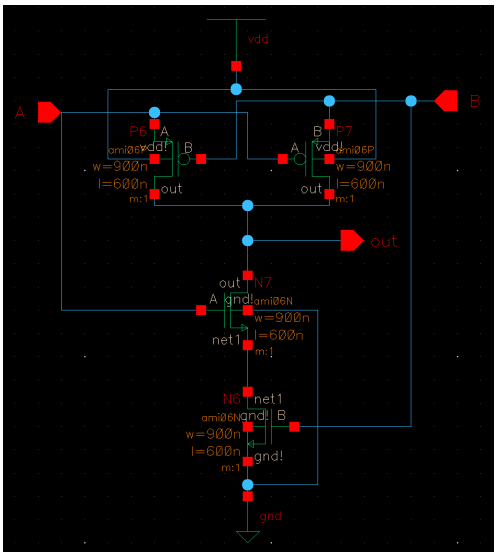

Figure 17: 2-input XOR gate schematic

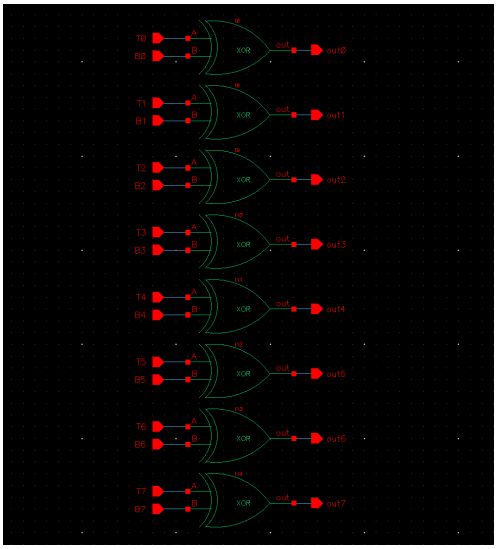

Figure 18: 8-bit bitwise XOR schematic

3.6.2 Sboxes/P4

Due to the fact that during the DES calculation, P4 is never used except directly after the SBoxes operation, we can combine them into a single operation. Instead of using registers, we used the 16-bit mux that we designed and wrote out the boolean equation for each bit in terms of the input bits. Since there are only four input bits per each set of two output bits, we only need 4 control signals to properly produce the output. The specific connections can be seen in fig. 19. Then, because P4 is always applied right after, we just rename the output pins to their appropriate value.

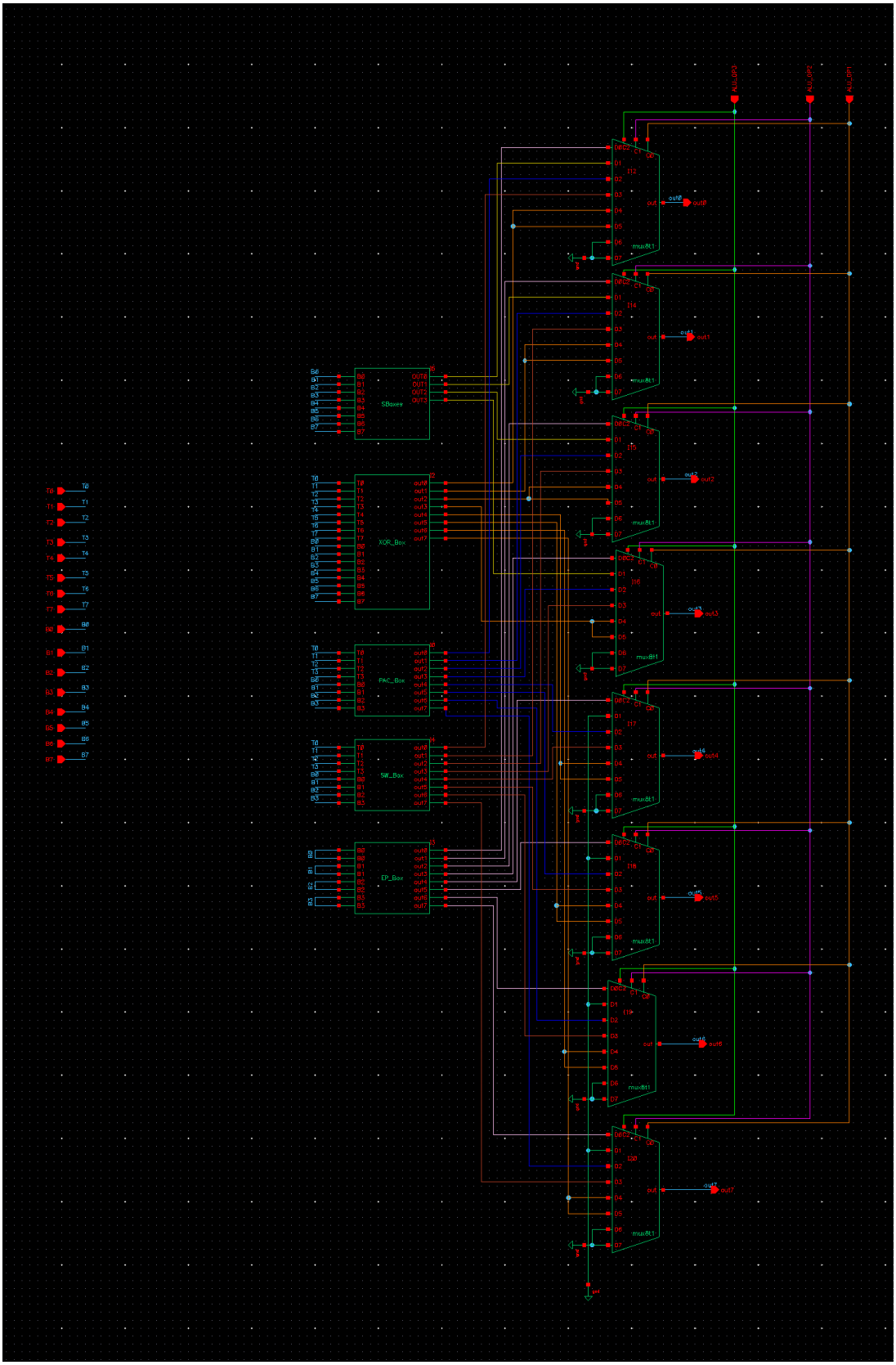

Figure 16: Schematic of the ALU

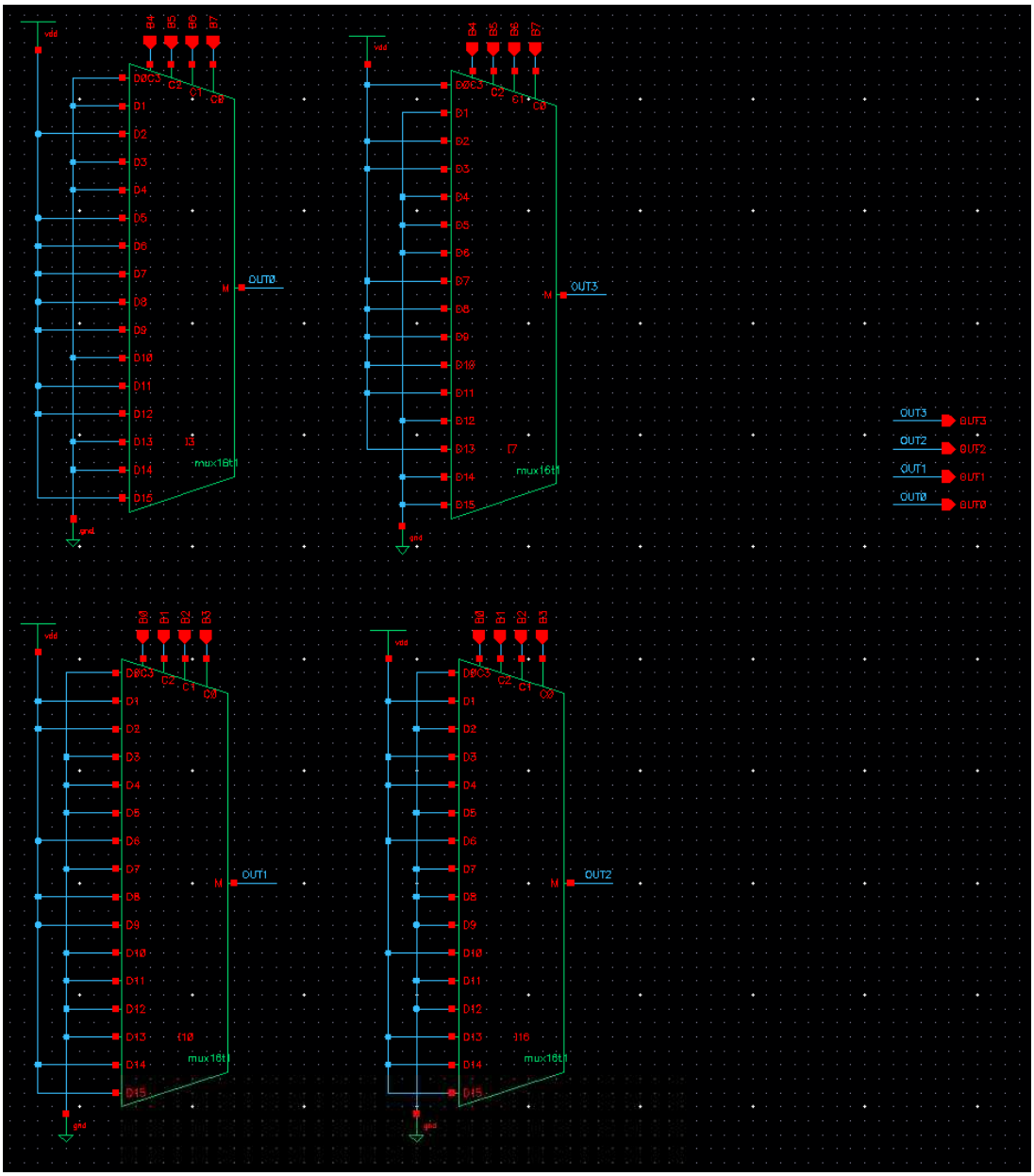

Figure 19: SBoxes/P4 Schematic

3.6.3 E/P

The E/P operation is a simple rearrangement of pins. It takes a 4-bit input, and produces an 8-bit output, made up of those 4 input bits. Our original schematic just shorted the correct input and output pins, but because Cadence throws an error for that, we placed buffers between them. This adds only a few nanoseconds and makes the signal a strong "1" or "0", if it was not previously. The rearrangement is seen in fig. 20

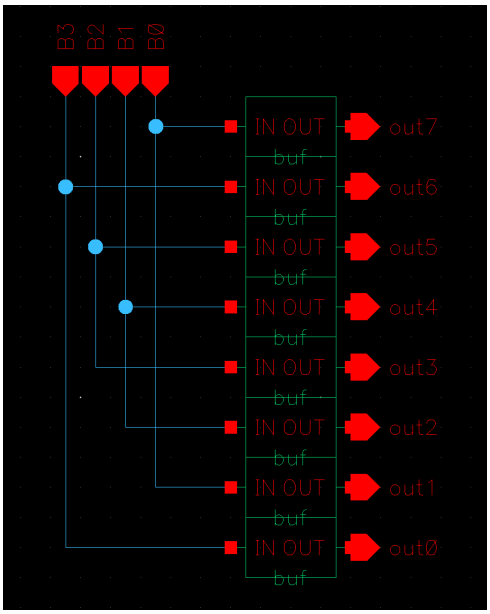

Figure 20: Expansion/Permutation box schematic

3.6.4 SW & PAC

Similar to E/P, the SW and PAC operations simply rearrange pins. We add buffers and connect them to the correct output, shown in figs. 21 and 22.

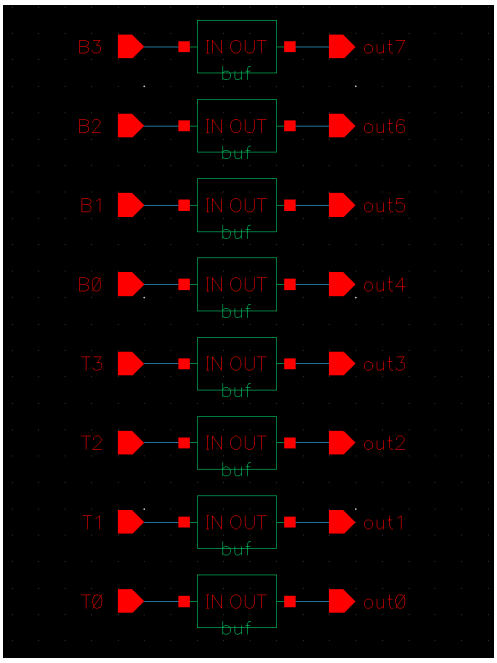

Figure 21: SWAP box schematic

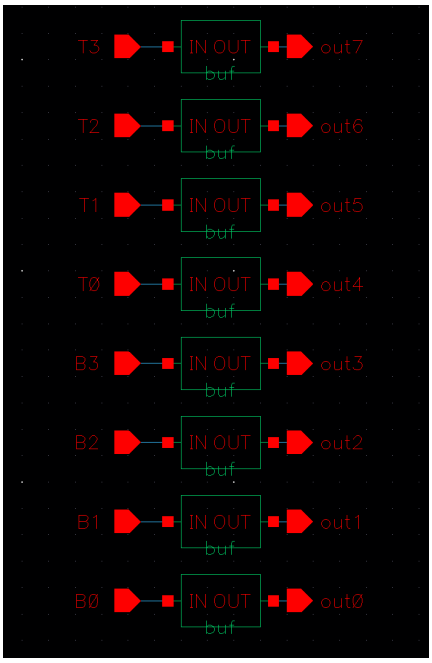

Figure 22: PAC box schematic

3.7 IN/OUT Blocks

Again, the IN and OUT permutations are simply static rearrangements of the pins. We again add a buffer and connect relevant outputs. The specific rearrangements can be seen in figs. 23 and 24.

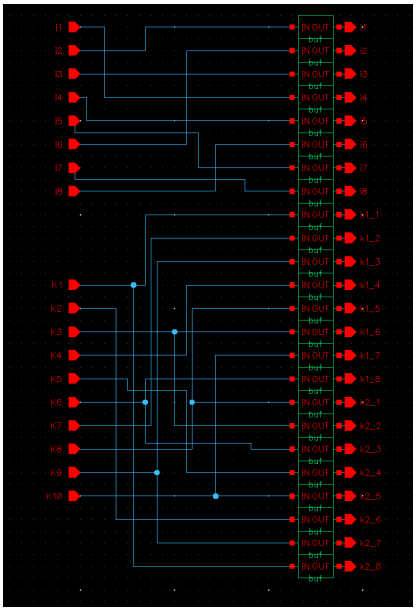

Figure 23: Input block schematic

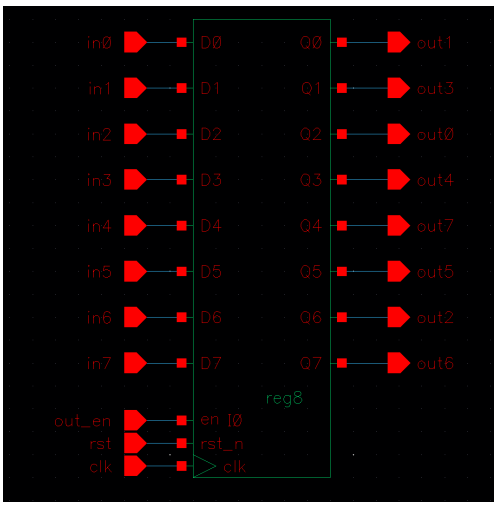

Figure 24: Output block schematic

3.8 Tri-State Buffers

Because we are using a bus-based architecture, we need to have tri-state buffers that present a high-impedance when they are not actively pushing or receiving from the bus.

The schematic pictured in fig. 25 was also found by searching google. The buffer functions by placing the logical not of the input into an inverter, which will be guaranteed to present a high impedance when the enable signal is low. With this, we can drive the main bus from multiple places, including the actual input pins and the ALU.

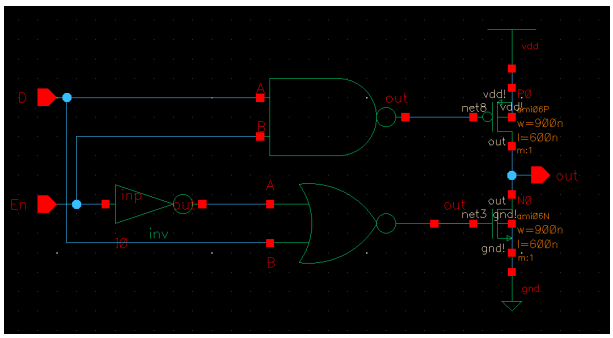

Figure 25: Tri-State buffer schematic

3.9 Controller Logic

The controller is just an implementation of multiple complex logic functions, as specified in the architecture section of this report. The actual implementation is exactly as discussed in chapter 7 of the textbook and the corresponding lectures. We form the pull-up and pull-down networks with the logical opposite of the function. Because all the functions are combinational. The specific implementations are seen in fig. 26

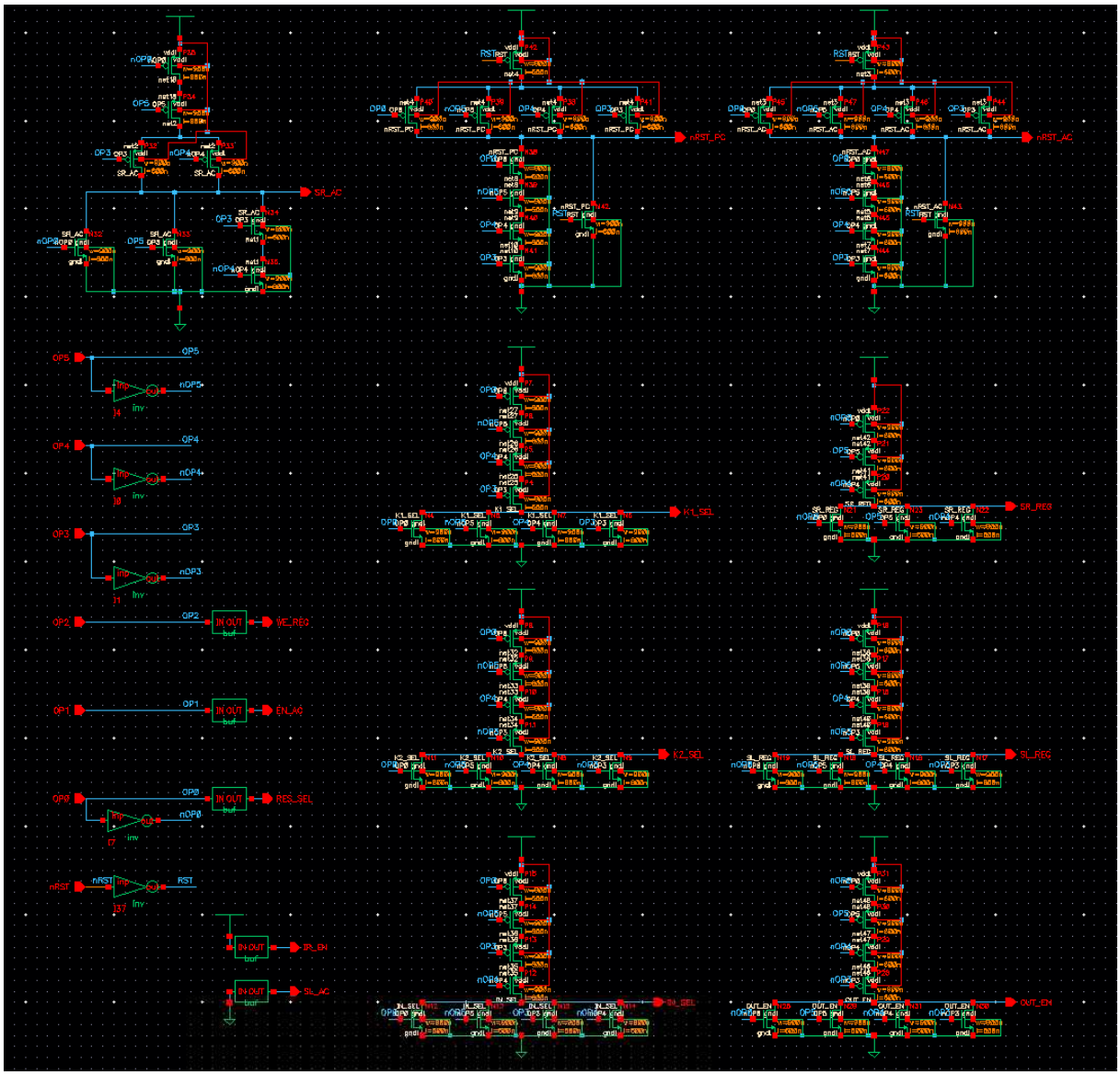

Figure 26: The Controller schematic

3.10 PC & IM

The Program Counter (PC) and Instruction Memory (IM) are implemented using registers and muxes. The program counter has a built-in 4-bit full adder (essentially a ripple counter), which counts up every clock period that the INC control signal is high.

The adder is implemented using a very simple full adder. It could be improved by implementing a carry look-ahead or a prefix sum adder, but because those are more complex, and we currently only need functionality, we stuck with what was easy. The adder can be seen in fig. 28.

The instruction memory was implemented using registers. Although this could've also been done solely with muxes and vdd/gnd, similar to how the SBoxes/P4 operation was done, we felt that the instructions are not static in the same way as the SBoxes operation. It makes more sense and is more easily interpretable by others to see registers for the instruction memory. The full instruction memory is seen in fig. 29. There are only 13 instructions, so we have only 13 registers. The last 3 values on the 16-bit muxes are tied

to ground.

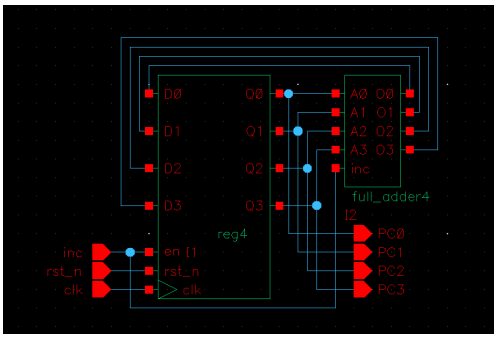

Figure 27: schematic

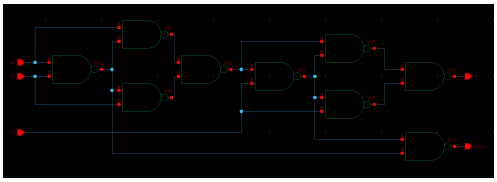

Figure 28: Full Adder schematic, four of these where chained together for the module seen in PC.

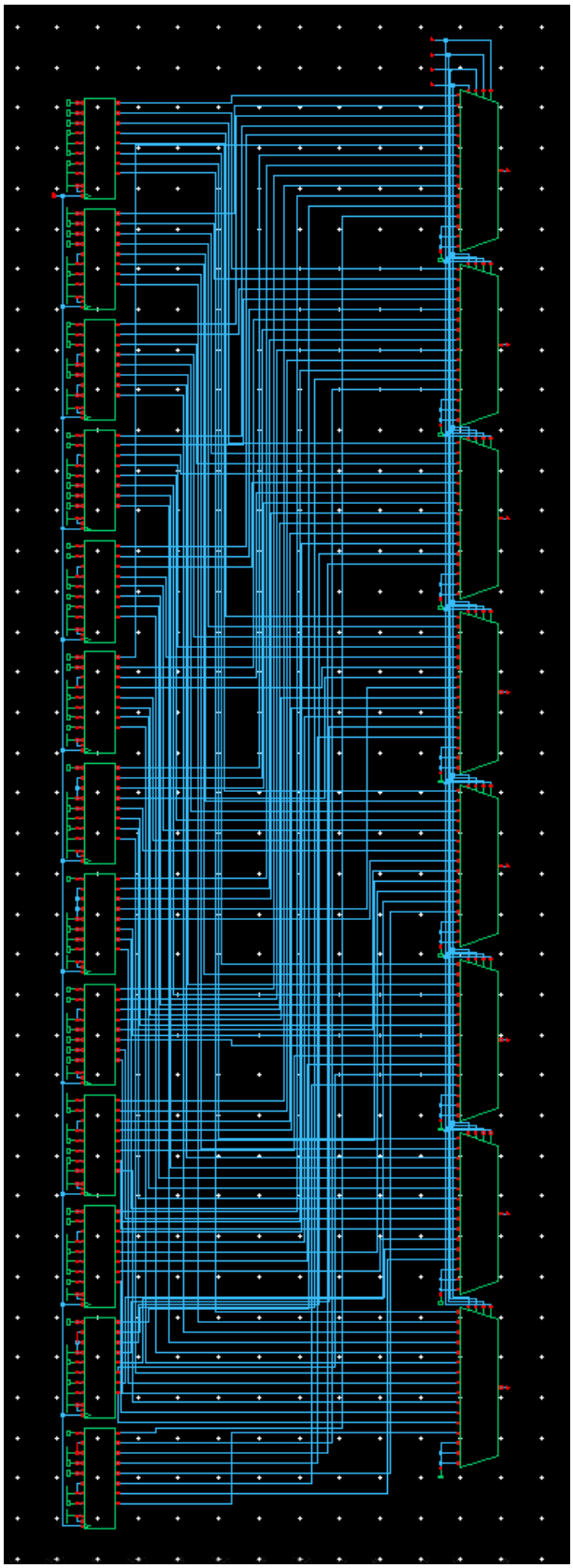

Figure 29: The Instruction Memory schematic

4 Results

The master schematic created in Cadence, seen in fig. 2, was simulated with multiple sets of inputs and the signals condensed to a `bus` for easy readability. The schematic simulation result can be found in fig. 30. Due to a error in implementation the enable wire was replaced with a Not Reset wire in the system. The overall functionality is very similar, but in this implementation the system is fully reset when the not reset wire is pulled low.

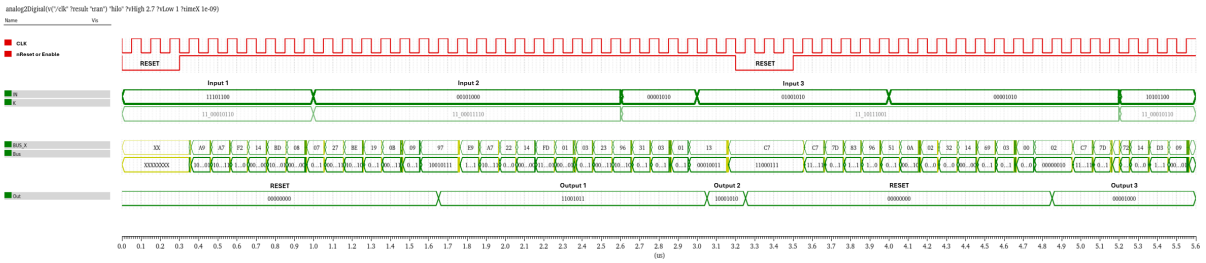

Figure 30: Schematic simulation results with three different sets of inputs. The graph shows the CLK (clock), nReset (not Reset or a sudo enable), IN (plaintext), K (key input), BUS_X (bus contents in Hex), Bus (bus contents in binary), and Out (the encrypted output).

Each unique input resulted in a correct corresponding output after the processor was given time to compute the correct value. The reset signal also behaved as expected and sets the outputs to zeros when pulled low.

4.1 Space Usage

Due to our use of a microprocessor architecture rather than a simple staged combinational logic method, there is likely more space used in general. The overall layout can be seen in fig. 31.

In the interest of saving some space and not adding a ton of pages, we will not paste the layout of every component the way we did for the implementation. However, the relevant components can be seen marked on the image, designating which component from the block diagram corresponds to which section of the layout.

We did not need to do much optimization in terms of space in order to fit everything in. Even with semi-random spacing between components and the internals having a lot of extra space, everything easily fit into the 1.5um x 1.5um space allotted to us, as can be seen in the figure.

We used a standard bus layout for routing wires between components since we were already using a bus-based architecture. This can be seen by the long strips of wire. The bus is an 8-bit bus, so it did not take up much space.

4.2 Performance

A key drawback of the processor style approach is the larger footprint and higher cost in terms of resources. However, it allows for flexibility in expanding the chip to perform more functions as needed in the future, simply by adding instructions and adjusting control and ALU logic. Much of the chip stays the same. The system also takes 13 clock

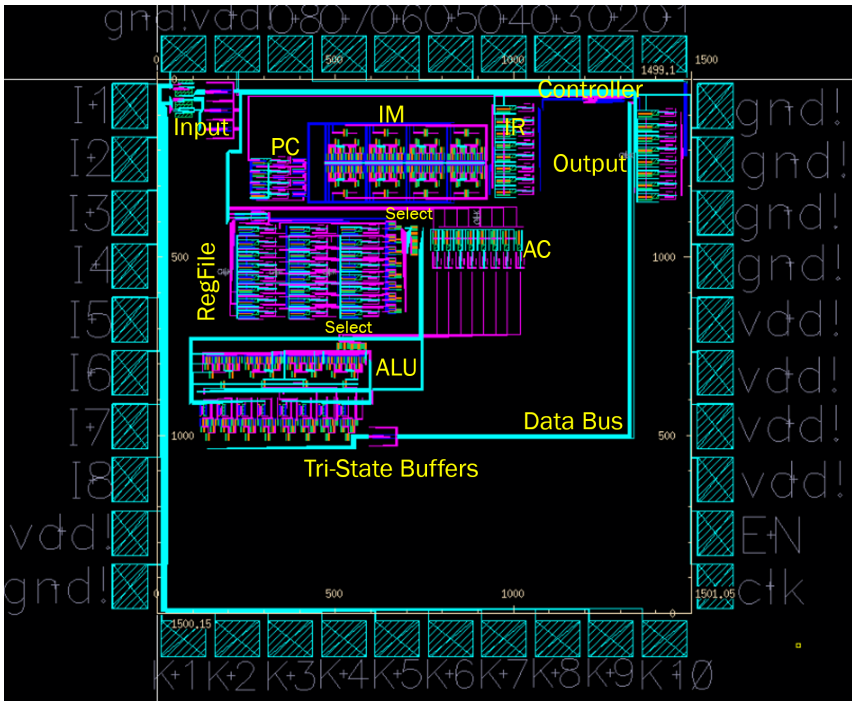

Figure 31: A picture of the full layout developed.

cycles from the reset(enable) signal transitioning from LOW to HIGH. The system would have been much more efficient in terms of space and transistor usage if built to be entirely combinational, but would not be as flexible for future changes, such as implementing a new encryption algorithm compatible with 8-bit computation. Pure combinational logic would reuse a much smaller amount of hardware, so if the algorithm was changed completely, it would require a full redesign, rather than just some adjustments to instructions and operations.

4.3 CSE563 Extension: Increasing Clock Speed (Ethan Morton)

With a full functional product developed, it is important to test the limits and see where it fails. For a small microprocessor architecture as we have used here, executing instructions blazingly fast is very important. Another important factor would be power consumption, which is not studied here, but even more critical than pure clock speed for small embedded devices such as this one.

The default clock speed that is tested earlier in this paper runs with a clock period of 100ns, or 10 MHz. My goal is to determine the frequency breakpoint where the chip stops functioning correctly, and see which specific component is the one failing in that scenario. Below in fig. 32 is a screen capture showing the correct output of the chip, 13 clock cycles after the transition of reset (enable) from LOW to HIGH.

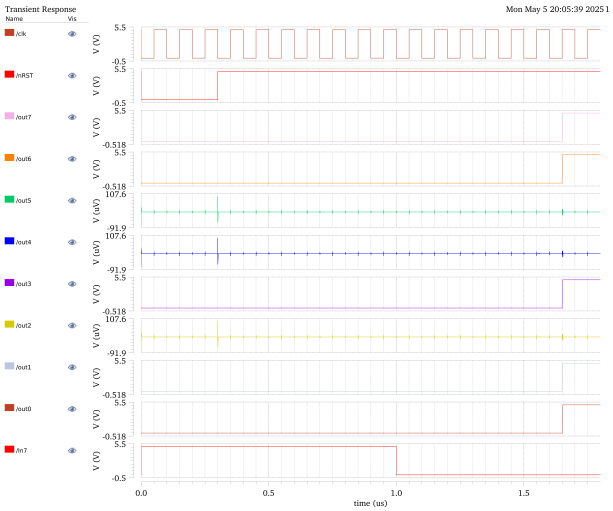

Figure 32: A waveform at 100ns clock period (10 MHz), showing a correct output from bits out0-7.

The lines to watch are out0-7, which represent the output seen from the output register at the end of our block diagram. The input to this run was a key K of

$$1, 1, 0, 0, 0, 1, 0, 1, 1, 0$$

and an input P of

$$1, 1, 1, 0, 1, 1, 0, 0$$

. The output, seen at roughly 500ns (13 clock cycles after reset was set to HIGH), is

$$1, 1, 0, 0, 1, 0, 1, 1$$

. That is the same as the output from the program developed for homework 7, seen in fig. 33.

```
========================
K1 ::
K                       || [1, 1, 0, 0, 0, 1, 0, 1, 1, 0]
P10(K)                  || [0, 0, 1, 0, 0, 0, 1, 1, 1, 1]
Shift(P10(K))           || [0, 1, 0, 0, 0, 1, 1, 1, 1, 0]
P8(Shift(P10(K)))       || [1, 0, 1, 0, 1, 0, 0, 1]
K1                      || [1, 0, 1, 0, 1, 0, 0, 1]
========================
K2 ::
K                       || [1, 1, 0, 0, 0, 1, 0, 1, 1, 0]
P10(K)                  || [0, 0, 1, 0, 0, 0, 1, 1, 1, 1]
Shift3(P10(K))          || [0, 0, 0, 0, 1, 1, 1, 0, 1, 1]
P8(Shift3(P10(K)))      || [1, 0, 1, 0, 0, 1, 1, 1]
K2                      || [1, 0, 1, 0, 0, 1, 1, 1]
========================
IP(P) ::
P                       || [1, 1, 1, 0, 1, 1, 0, 0]
IP(P)                   || [1, 1, 1, 1, 0, 0, 1, 0]
(L,R)                   || ([1, 1, 1, 1],[0, 0, 1, 0])
========================
First Round ::
R                       || [0, 0, 1, 0]
E/P(R)                  || [0, 0, 0, 1, 0, 1, 0, 0]
K1                      || [1, 0, 1, 0, 1, 0, 0, 1]
E/P(R) XOR K1           || [1, 0, 1, 1, 1, 1, 0, 1]
SBoxes(E/(R) XOR K1)    || [0, 1, 0, 0]
P4(SBoxes(E/(R) XOR K1)) || [1, 0, 0, 0]
FK1                     || [1, 0, 0, 0]
========================
f_K1 ::
FK1                     || [1, 0, 0, 0]
L                       || [1, 1, 1, 1]
L XOR FK1               || [0, 1, 1, 1]
f_K1                    || [0, 1, 1, 1, 0, 0, 1, 0]
========================
SW ::
SW(f_K1)                || [0, 0, 1, 0, 0, 1, 1, 1]
L2                      || [0, 0, 1, 0]
R2                      || [0, 1, 1, 1]
========================
Second Round ::
R                       || [0, 1, 1, 1]
E/P(R)                  || [1, 0, 1, 1, 1, 1, 1, 0]
K2                      || [1, 0, 1, 0, 0, 1, 1, 1]
E/P(R) XOR K2           || [0, 0, 0, 1, 1, 0, 0, 1]
SBoxes(E/(R) XOR K2)    || [1, 1, 1, 0]
P4(SBoxes(E/(R) XOR K2)) || [1, 0, 1, 1]
FK2                     || [1, 0, 1, 1]
========================
f_K2 ::
FK2                     || [1, 0, 1, 1]
L                       || [0, 0, 1, 0]
L XOR FK2               || [1, 0, 0, 1]
f_K2                    || [1, 0, 0, 1, 0, 1, 1, 1]
========================
IP^-1 ::
f_K2                    || [1, 0, 0, 1, 0, 1, 1, 1]
IP^-1(f_K2)             || [1, 1, 0, 0, 1, 0, 1, 1]
D                       || [1, 1, 0, 0, 1, 0, 1, 1]
========================
```

Figure 33: A guide to the encryption process using Python code, useful for checking every point of the computation for accuracy.

As we increase the clock rate, we expect to see the output waveforms remain the same shape, on the smaller timescales of fast frequencies. Through rigorous testing (and waiting a while for simulations to finish), I found that the fastest clock speed this can operate at is 66.7 MHz, or a clock period of 15ns. A graph of the chip working at this frequency is seen below in fig. 34.

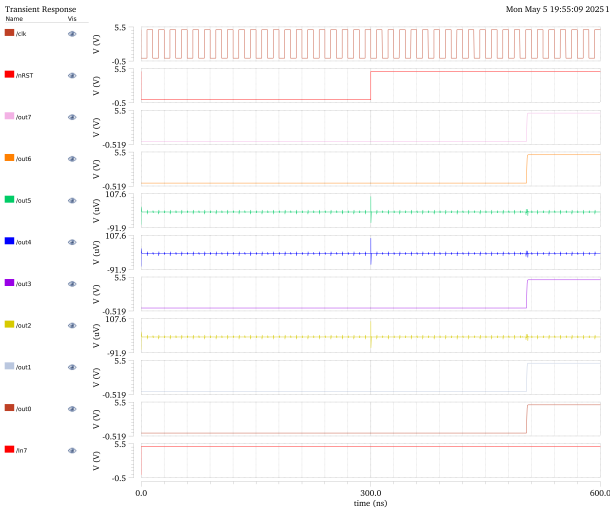

Figure 34: The resulting waveforms from running the clock at 66.7 MHz.

We see that the output remains the same, even at these smaller timescales. Any lower, and the output comes out wrong. I found that the first thing to fail was the ALU, which was not able to get through the first SBoxes operation before the clock period ended. This resulted in incorrect values being latched from the data bus, and the rest of the computation is crap. A good strategy to continue to improve this design would be to continually simulate for correct behavior, determine where the error is happening, and reduce that critical path until it no longer errors. Then, continue to simulate more extreme clock speeds until another error is hit, and so on. The goal of this would be to make the chip able to accept a clock speed at least 13 times as fast as the combinational versions. This would mean that it matches the combinational logic in terms of performance, and can compute the encryption in a similar amount of time.

5 Conclusion

The project posed a good design challenge, where we were tasked on making a chip to perform the DES algorithm calculation by any means possible, with only the chip space being the limitation. Our approach of making our chip a processor posed an even greater challenge, working to create a working processor, to gain the experience in making a processor. This approach also allowed us to reuse many components by significantly complicated the control side, but requiring fewer of the same components to complete the logic.

5.1 Improvements and Changes

If this project were to be done again, we would try to implement the logic in combination to understand the drawbacks of that method. We believe that this approach is simpler and thus there is less to learn from this approach. If the processor architecture were to be kept, we would first redesign the system to work without an instruction memory since it takes up a significant amount of space, and the combinational logic would, in turn, take up less space. The implementation of the SBoxes in the current design was done using LUTs, and, if done with combinational logic, it would be more complex to initially put together but would take up significantly less space.

References

[1] D. Ivanovich, “Des tutorial,” 2025.