SIMON FRASER UNIVERSITY

SCHOOL OF ENGINEERING SCIENCE

ENSC 350: Digital System Design
Spring 2009

Lab 2

Due the Week of February 23rd, 2009

Lab 2 will have you implement a ***structural*** design of a Data Encryption Standard (DES) decryption circuit on the DE2 board along with an appropriate testbed. Your success will be measured by having a functioning design as well as maximizing your (1/area*Max Throughput) product (to be explained later). DES was one of the most popular high-speed encryption algorithms. Although DES has been, for the most part, replaced by newer algorithms, such as Triple-DES and Rijndael (AES), these new algorithms all operate using principles similar to DES. In fact, Triple-DES is simply the application of the DES algorithm three times. Although DES can be performed by software, the fact that it is comprised of mainly shifts, bit permutations, and exclusive-or operations makes it far more suitable for a hardware implementation.

This lab will be a ***lot of work***, that's why you have three weeks. It is highly recommended that you get working on it early, otherwise you may fail to complete the lab. A suggested schedule of tasks is given at the end; we won't be checking your progress, so it is up to you to make sure you meet the milestones. If you don't, you'll have a hard time completing the lab for the due date.

The DES Algorithm

The DES algorithm encrypts or decrypts data in 64-bit "chunks" using a 64-bit key (versions using longer keys also exist). The algorithm performs a series of bit-shuffles and exclusive OR's on the 64-bit data; the exact pattern depends on the key. Once encrypted, the original data can only be recovered using the key used during the encryption. The encryption and decryption algorithms are similar, although not identical. In this project, you will create a decryption circuit. To test the design, we will give you some encrypted messages, and the key, and your design will decrypt these messages. Please note that during your lab demo, we will use ***different*** messages and keys, so you should thoroughly test your design, which means you may want to create more test vectors for yourself. In this section, the decryption algorithm will be described; if you want to know how the encryption works, then you can ask Google.[1]

a) Top Level

The top-level schematic is shown in Figure 1. The 64-bit encrypted data arrives on signals **DesDin[1:64]** and the 64-bit key arrives on signals **DesKey[1:64]**. The output data appears on the bus **DesDout[1:64]**. Note that we are numbering the wires in the bus differently than usual here (usually we would expect a 64 bit bus to be numbered [63:0]). We will do this throughout this project to be consistent with much of the published encryption/decryption literature that you might find on the web. The first wire in the **DesDin** bus is wire 1 (MSB), the second wire is wire 2, … and the last wire is wire 64 (LSB). You should follow this convention throughout the design.

Your core will have a single clock (**Clk**) and a synchronous reset (**Rst**). Data processed by your core will read by and written to First In First Out (FIFOs) to provide buffering. The necessary handshaking signals for the input FIFO (**InFIFO_Empty**, **InFIFO_Rd**) and the output FIFO (**OutFIFO_Full**, **OutFIFO_Wr**) are used by the controller to ensure that data is not lost or corrupted during decryption.

---

[1] Note: much of this description was modified from http://www.aci.net/kalliste/des.htm, but it is no longer a live link. http://www.thaiall.com/security/des.htm is another link you can try.

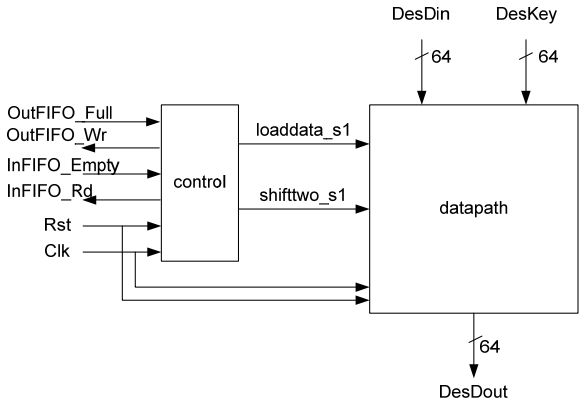

Figure 1: Top-Level Schematic

The controller should read in a new value from the input FIFO when it is not empty and no value is currently being processed. When the **InFIFO_Rd** signal is asserted during a clock cycle, it means that a valid key and encrypted data will be available on the **DesKey** and **DesDin** ports *in the next clock cycle*. To reiterate, if **InFIFO_Rd** is high in clock cycle *i*, the new data and key are accepted in cycle *i+1*. This might seem strange, but it actually makes the design of the controller slightly easier. Please do not change this behaviour so that your following lab (lab 3) is easier. Also, if you change the behaviour and have problems, your classmates and your TA probably won't be able to help you. Finally, this behaviour will be expected for the testbed you are asked to design (we will be providing you with testbenches for your testbed assuming that your circuit conforms to the specifications). Once the new data has been read, the decrypted data should be available on **DesDout** after 16 cycles. If you choose to increase the number of clock cycles required to provide valid output, this will likely decrease the Throughput of your design and thus it's performance as measured by the (1/area*Max Throughput) product.

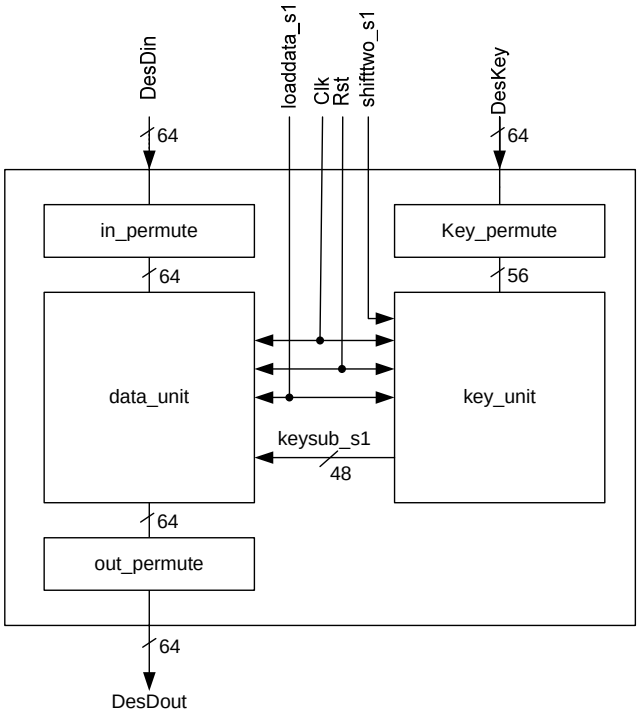

Figure 2: Datapath schematic

b) Datapath

Figure 2 shows the structure of the datapath. As the figure shows, the datapath is divided into two parts, a key unit and a data unit. First consider the data side. The data arrives on port **DesDin[1:64]** and is shuffled by the permutation block labelled **in_permute**. This rearranges the bits according to the following table, where the entries in the table show the new arrangement of the bits from their initial order. So, for example, the 58th bit of the input data becomes the first bit of input permute block output. The 50th bit of input data becomes the second bit of the input permute block output. The 7^{th} bit of input data becomes the last bit of the input permute block output.

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Table 1: Input Permute Block Pattern

So, as an example, if the input string of the permute block is:

input = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

then the output string of the permute block is:
output = 1100 1100 0000 0000 1100 1100 1111 1111 1111 0000 1010 1010 1111 0000 1010 1010

Note that this permute block is strictly combinational. In fact, it performs no computation, it only rearranges the bits in the input word.

The data is then shuffled for 16 cycles, in a manner to be described below. After the 16 cycles, the data bits are rearranged again, according to the pattern in Table 2.

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Table 2: Output Permute Block Pattern

This table is read the same way as Table 1. As an example, if the output permute block input is
input = 00001010 01001100 11011001 10010101 01000011 01000010 00110010 00110100

then the output permute block output is:
output = 10000101 11101000 00010011 01010100 00001111 00001010 10110100 00000101

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Table 3: Key Permute Block Pattern

Now consider the key side. The 64-bit key is fed to a **key_permute** block, which shuffles the bits within the key. The **key_permute** block rearranges the 64 bits of the key, and selects 56 of these bits for use in the algorithm (note that this means that 8 of the bits in the key are ignored. In a 64-bit key, only 56 bits are actually used). The rearrangement is performed according to pattern in Table 3.

This table is read the same as the other tables; Since the first entry in the table is "57", this means that the 57th bit of the original key becomes the first bit of the permuted key (the key permute block output). The 49th bit of the original key becomes the second bit of the permuted key. The 4th bit of the original key is the last bit of the permuted key. Again, note only 56 bits of the original key appear in the permuted key. As an example, if the input pattern key is:
input = 00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

Then the permuted key (the key permute block output) is:
output = 1111000 0110011 0010101 0101111 0101010 1011001 1001111 0001111

As will be described below, the data is shuffled within the **data_unit** block for 16 cycles. This shuffling is controlled by the 48 bit signal **keysub_s1**. These 48 bits are selected from the 56 bit permuted key computed by the **Key Unit** block, according to a predefined pattern. The way in which these bits are selected, and the manner in which these bits are used to shuffle the data are described next.

c) Key Unit

The Key Unit receives 56-bit key bits (remember, the Key Permute block selects 56 bits from the original 64 bit key). The Key Unit selects 48 bits from these 56 bits according to a predefined pattern. These 48 bits are sent to the Data Unit through the 48-bit wide port **keysub_s1**. Each cycle, a different set of 48 bits is sent to the Data Unit (remember that the entire decryption process takes 16 cycles, so 16 different patterns are produced by the Key Unit).

It would be possible to create a large ROM that would indicate which 48 bits of the 56 bit key (and the order of these bits) are produced each cycle. Alternatively, the bits can be created using some very simple logic that shuffles bits around. We will follow this latter approach. Figure 3 shows the logic that can be used to produce the 16 48-bit sequences.

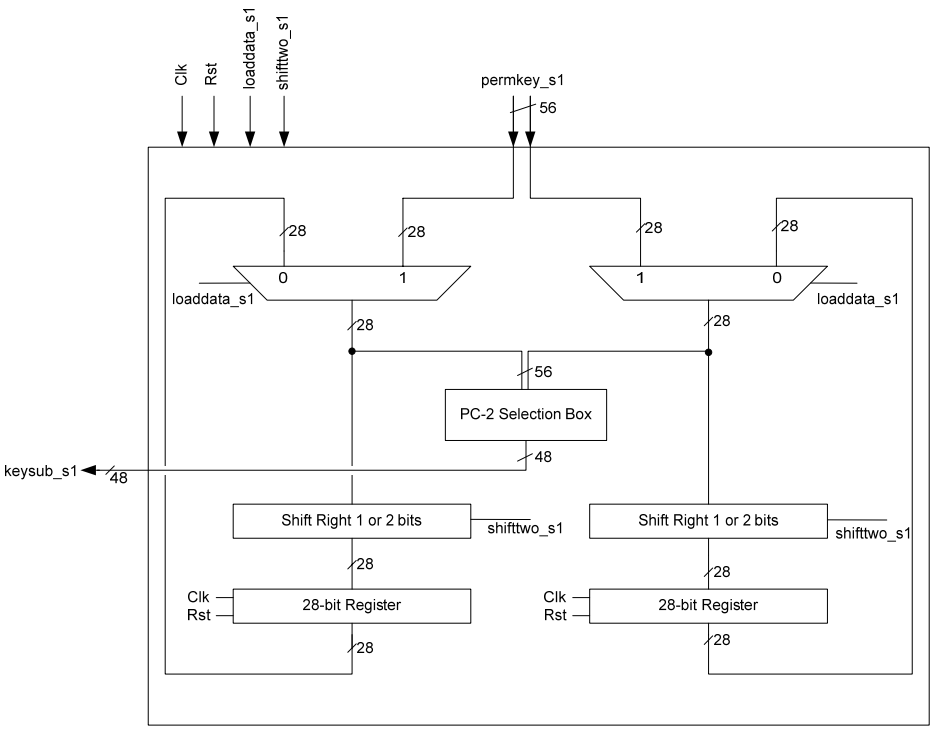

Figure 3: Key Unit Schematic (Decryption)

Figure 3 shows that the operation of the Key Unit is controlled by two control signals, **loaddata_s1** and **shifttwo_s1**. Both of these signals are generated by the controller block (see below). During the first cycle, the 56 bit key input is split into two 28-bit halves. As will be described below, during the first cycle, the **loaddata_s1** control signal is high, meaning the input key is selected by the multiplexers (in the remaining cycles, **loaddata_s1** is low, meaning the feedback signals are selected). Each of the 28-bit halves are then rotated by either 1 or 2 bits (note that, although the block is labeled "shift", it is actually "rotate"; with each shift, the right-most bit is rotated to the left-most position). The control signal **shifttwo_s1** indicates whether the data is shifted one bit or two bits (if the signal is high, the data is shifted two bits). Also note that the shift is combinational, not sequential. The controller asserts **shifttwo_s1** according to a predetermined pattern, as described below. The two halves are then register into two 28-bit registers. This is repeated for 15 more cycles.

In this way, during each of the 16 cycles, a new pattern of bits appears at the output at the output of each multiplexer. These bits are fed to a permutation block, labeled PC-2, which selects 48 of the 56 signals, according to the pattern in Table 4:

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Table 4: PC-2 Block Pattern

To read this table, imagine that the C is the left 28-bits and D is the right 28-bits. Then, {C, D} is a 56 bit number, which is used to index into Table 4. So, the first bit of the output **keysub_s1** is the 14th bit of {C, D}. The second bit of the output **keysub_s1** is the 17th bit of {C, D}. The third bit of the output **keysub_s1** is the 11th bit of {C, D}, etc. The last bit of **keysub_s1** is the 32nd bit of {C, D}.

These 48 bits are then passed to the Data Unit. The Data Unit then uses these 56 bits to control the decryption as explained below.

Interestingly, the only difference between the encryption and decryption algorithm is that the **keysub_s1** sequences provided to the Data Unit are provided in the reverse order. In other words, the 48-bit sequence provided during Cycle 1 in the decryption algorithm is provided during Cycle 16 in the encryption algorithm. The 48-bit sequence provided during Cycle 2 in the decryption algorithm is provided during Cycle 15 in the encryption algorithm, etc. Although not required for this lab, it is an interesting exercise to figure out how to modify the datapath in Figure 3 to produce keys in the reverse order (Hint: you shift left instead of right, but you need some other modifications too).

d) Data Unit

The Data Unit uses the sixteen 48-bit sequences generated by the Key Unit block to decrypt the permuted data from the in_permute block. Again, a very simple logic circuit is used to perform the decryption. Figure 4 shows the circuitry.

Like the key in the Key Unit, the data input to the Data Unit is split into two halves during the first cycle. The right half is first permuted (and extended) using an E-Box, according to pattern in Table 5.

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Table 5: Ebox Expansion Block Pattern

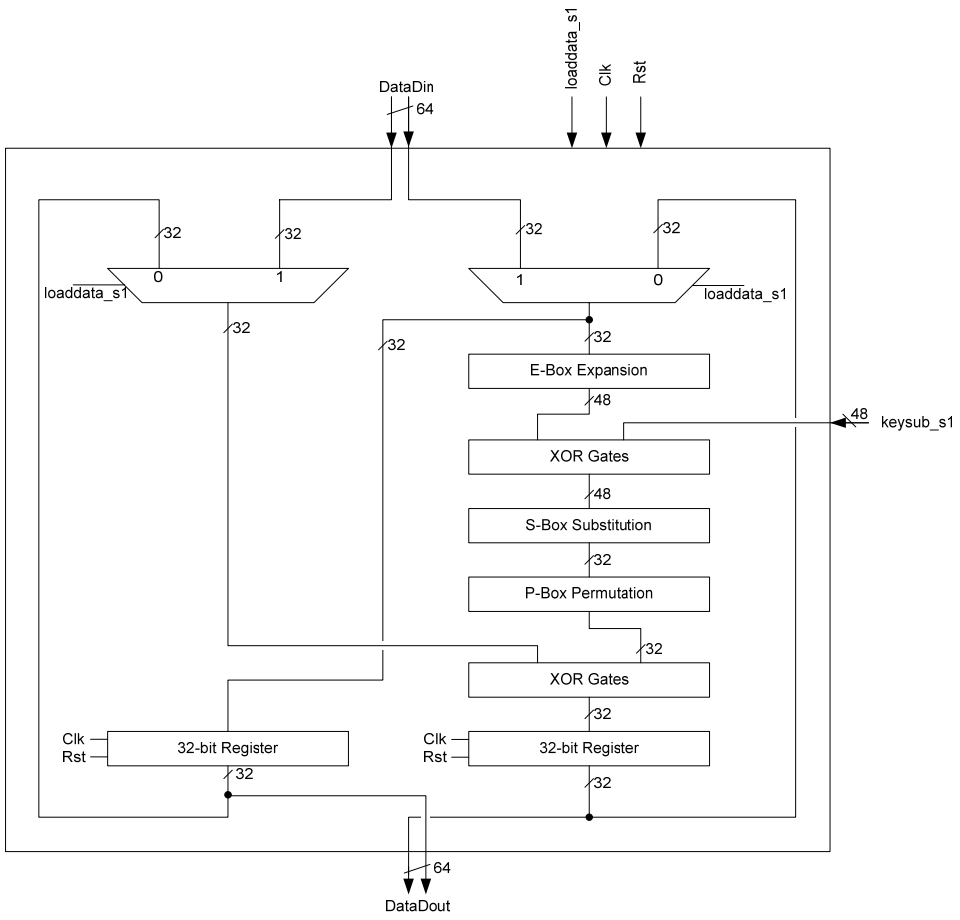

Figure 4: Data Unit Schematic

This table is read the same way as the previous tables. The Ebox has 32 inputs and 48 outputs. The first output (bit 1) is taken from input bit 32. The second output bit is taken from input bit 1. The third output bit is taken from input bit 2, etc. Note that some input bits (such as bit 4) feed two outputs; this shouldn't be surprising since the block has more outputs than inputs.

The 48 output bits from the Ebox are then exclusive-ored with the 48-bit **keysub_s1** signal from the Key Unit. Recall that these 48 **keysub_s1** bits are permuted bits of the key, and the **keysub_s1** bits change each cycle.

The 48 exclusive-or outputs then feed a Substitution Block. This block is different than the others. The other blocks simply rearrange bits; this block substitutes the input bits with new bits according to a predetermined pattern. Each unique pattern of the 48 input bits corresponds to a 32-bit output pattern. Think of the Substitution Block as a large ROM with 2^{48} words, each of which is 32 bits. The 48 input bits index into this ROM (the address inputs); the 32 bit word that is addressed is sent as the output of the block.

Of course, it is not feasible to implement a 2^{48} word ROM, since 2^{48} is a very large number. You might also imagine this block being implemented using 32 logic functions, each of which has 48 inputs. However, in general, a 48-input function is huge, meaning this isn't really a feasible solution either.

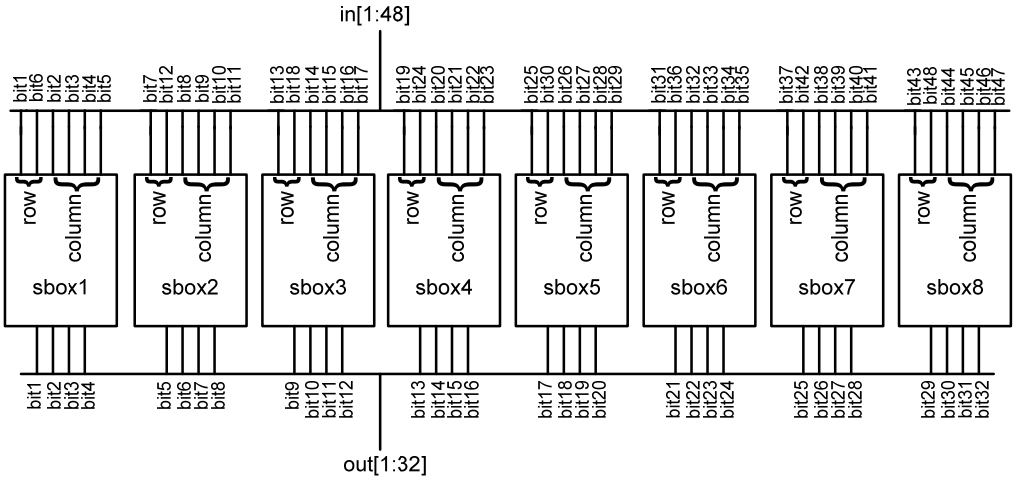

Figure 5: Substitution Block Schematic

Instead, because of the nature of the predetermined pattern that would be stored in the ROM, the ROM can be replaced with the simpler structure shown in Figure 5. Rather than one 2^{48}x32 ROM, this solution requires *eight* 2^6 x 4 bit ROMs. Each ROM, which is called an sbox, has 6 address inputs and 4 data outputs. The address lines of each ROM are driven by 6 of the 48 input bits, and the data lines of each ROM drive 4 of the 32 output bits.

Consider the first sbox (sbox1). This ROM is driven by bits 1 through 6 of the input, and it produces bits 1 to 4 of the output, as shown in Figure 5. Each unique 6 bit input pattern produces a 4-bit pattern, according to the pattern in Table 6.

	Column Number															
Row No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Table 6: Sbox1 Pattern

Table 6 is read differently than the other tables presented earlier. Suppose the number formed by combining bits 1 and 6 of the input is *i*, and the number formed by combining bits 2 to 5 of the inputs is *j*. Then, the output is determined by reading the number in row *i* and column *j* of Table 6. As an example, suppose bits 1 through 6 of the input are “111100”. Then, *i* is 2 (binary “10” which is bit 1 and bit 6 of the input) and *j* is 14 (binary “1110” which is the combination of bits 2 through 5 of the input). Thus, by reading the entry at row 2 and column 14, we can see that the sbox1 output is 5 (“0101”). As another example, suppose bits 1 through 6 of the input are “001100”. Then, *i* is 0 (binary “00” which is bit 1 and bit 6 of the input) and *j* is 6 (binary “0110” which is the combination of bits 2 through 5 of the input). Thus, by reading the entry at row 0 and column 6, we can see that the sbox1 output is 11 (“1011”).

The other sboxes operate in the same way. For each sbox, the subset of input and output bits is different, as shown in Figure 5. The output pattern is also different; Table 7 shows the output pattern for all eight sbox’s.

S1

14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S2

15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S3

10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S4

7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S5

2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S6

12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S7

4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S8

13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Table 7: All sbox patterns

The 32-bit output from the Substitution Block is then fed through another permutation block, called the Pbox Permutation Block. The permutation pattern for this block is shown in Table 8. Table 8 is read the same way as the other permutation blocks. Bit 1 of the output comes from bit 16 of the input. Bit 2 of the output comes from bit 7 of the input. Bit 3 of the output comes from bit 20 of the output, etc. Finally, the P-box output is exclusive-ored with the right-most 32 data bits and registered as shown in Figure 5.

These operations are repeated for 16 cycles.

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Table 8: P-box Permutation Block Pattern

e) Control Block

As shown in Figure 1, the controller is a state machine with one input (in addition to the two clock inputs) and two outputs. The input is **InFIFO_Rd** and is used by the user to initiate a decryption. When **InFIFO_Rd** goes high in a cycle, the encryption starts *in the next cycle*. Thus, the **InFIFO_Rd** signal can be thought of as a valid signal to start up your state machine. The first output is the **loaddata_s1**; this feeds the Data Unit and Key Unit, and causes these units to load in new data/key, as described earlier. (*Note: it is assumed that the value of **DesKey** may change, but not during the decryption of a data block. It is the responsibility of the user providing the inputs to ensure that this does not happen.*) The state machine asserts **loaddata_s1** during the first cycle of each decryption and keeps it low for all other cycles. The second output is **shifttwo_s1**. This signal is used to indicate whether the rotate blocks in the Key Unit should rotate by 1 bit or 2 bits. If this signal is 1, each rotate block rotates its input by 2 bits, otherwise it rotates the input by 1 bit. The shifttwo_s1 is asserted according to a predetermined pattern, as shown in Table 9.

Cycle	loaddata_s1	shifttwo_s1
1	1	0
2	0	1
3	0	1
4	0	1
5	0	1
6	0	1
7	0	1
8	0	0
9	0	1
10	0	1
11	0	1
12	0	1
13	0	1
14	0	1
15	0	0
16	0	0

Table 9: Controller State Machine Behaviour (Decryption)

The preceding should give you a good overview of the DES algorithm. You will be able to find many other references and examples, including sample code, on the web. Testbenches for each component will be provided, so you can verify your understanding of each module. Note, we will be using different testbenches to verify your overall design, so you may want to generate more test vectors of your own.

The Testbed

You will need to create a testbed to verify and evaluate your design. This testbed will be used in your demo to evaluate your design. The required testbed files are:

- des_testbed.vhd – top level testbed file, you will have to add to this.
- des.vhd – dummy DES that writes all inputs directly to outputs to debug the testbed itself (will be replaced with your actual circuit after you get the testbed working).
- expected_ram.vhd – wrapper vhdl file for 1024x64-bit ROM of expected output values
- expected_ram_inst.vhd – example VHDL for instantiation
- expected_ram.cmp – netlist file for 1024x64-bit ROM of expected output values
- expected_ram.mif – initialization file for 1024x64-bit ROM of expected output values. Same values as testvector_rom.mif except for address 0x03
- result_fifo.vhd – wrapper vhdl file for FIFOs (your DES circuit interacts with this)
- result_fifo.cmp– netlist file for the FIFOs your DES circuit interacts with
- testvector_rom.vhd – wrapper vhdl file for 1024x64-bit ROM of input test vectors
- testvector_rom_inst.vhd – example VHDL for instantiation
- testvector_rom.cmp – netlist file for 1024x64-bit ROM of expected input test vectors
- testvector_rom.mif – initialization file for 1024x64-bit ROM of input test vectors. Same values as expected_ram.mif except for address 0x003

Description:
The testbed will consist of two ROMs, two FIFOs, Comparison logic, LCD logic, and control logic for operating all these things. Below is a high level block diagram of the testbed.

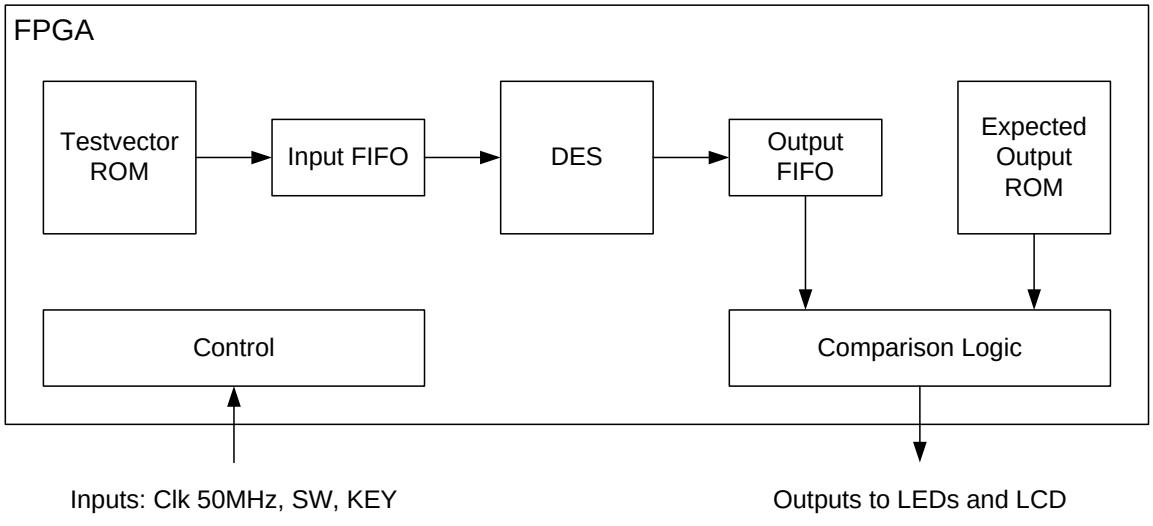

Figure 6: Testbed Block Diagram.

Testvectors are read from the Testvector ROM into the Input FIFO. The DES block reads from the Input FIFO when it is ready to decrypt a new word. Once the DES circuit is done decryption, it writes the result to the Output FIFO. This result is compared against an expected result from the Expected Output ROM. A

green LED is turned on if the result matches the expected value, otherwise a red LED is turned on and the comparison logic is stalled to let the user view the offending result on the LCD.

Note that this testbed uses a real 50MHz clock instead of a pushbutton as in lab 1.

FIFOs:

The DES circuit interacts with the testbed via the two FIFOs. Below is a description of how to read/write. Be sure to handle the FIFIO Full and Empty cases in your DES circuit control logic.

The following waveforms show the behavior of scfifo megafunction for the chosen set of parameters in design result_fifo.vhd. The design result_fifo.vhd has a depth of 64 words of 64 bits each. The output of the FIFO is unregistered. The FIFO is in legacy synchronous mode. The data becomes available after 'rdreq' is asserted; 'rdreq' acts as a read request.

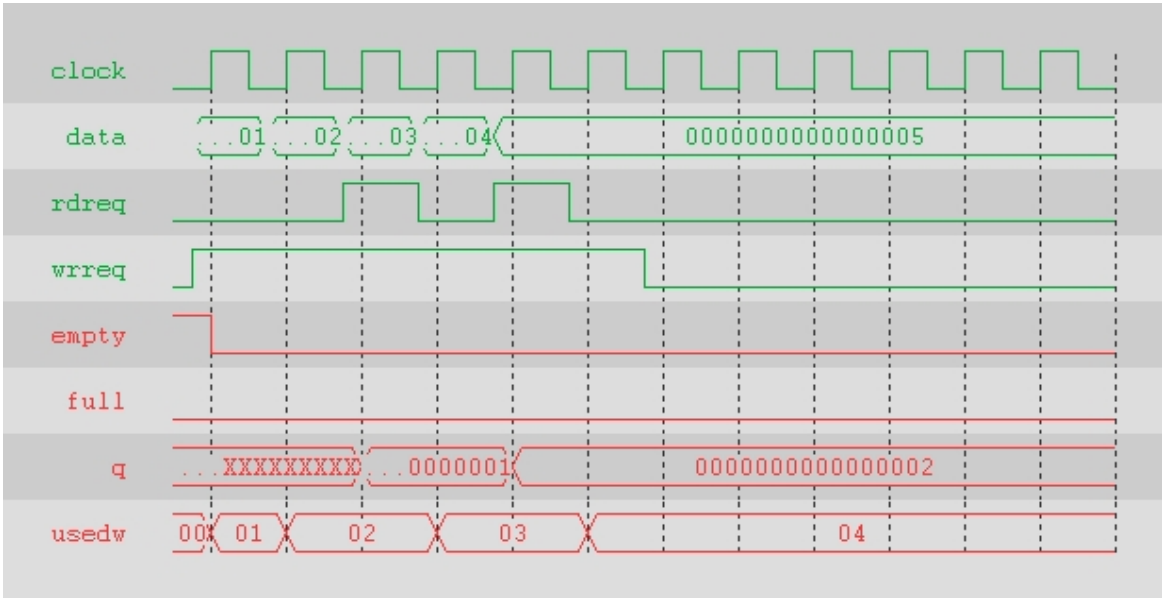

Figure 7: Wave showing read and write operation.

The above waveform shows the behavior of the design under normal read and write conditions .

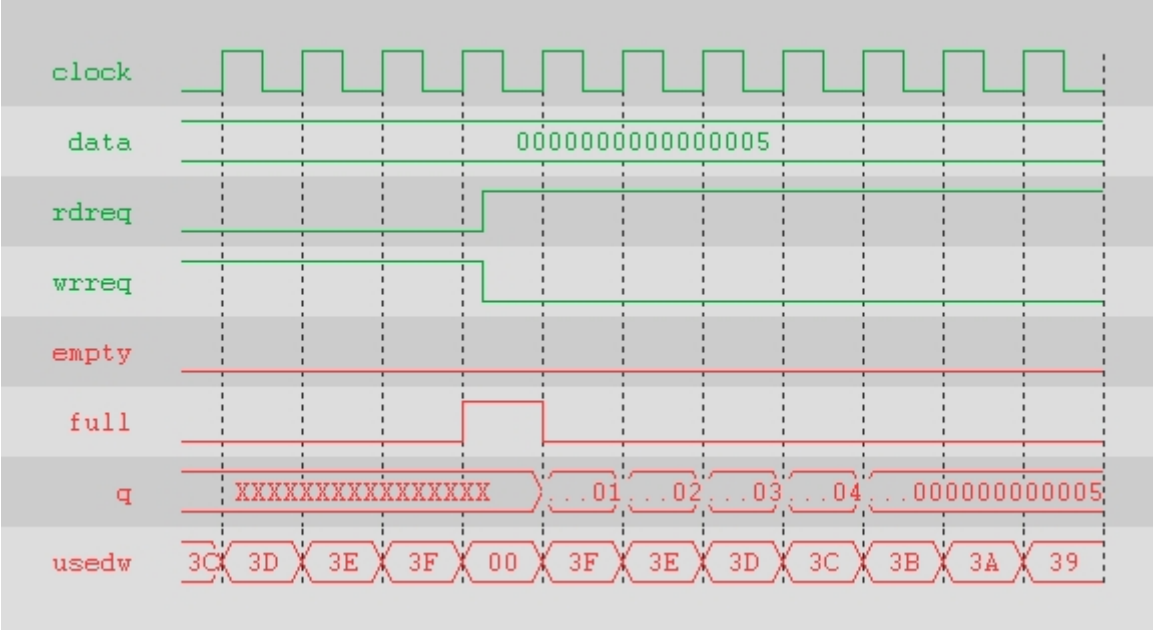

Figure 8: Wave showing FIFO full operation.

The above waveform shows the behavior of the design for FIFO full condition. In the example above, data is written into the FIFO till it is full, then data is read back.

ROMs:

The test vectors and expected outputs are stored in the Testvector ROM and the Expected ROM (this is a ROM, despite the expected_ram.vhd name). There should be some control logic to control addressing the ROMs and coordinating reading their outputs to the appropriate places (Input FIFO and Comparison Logic). Below is a description of how to interface with the ROMs (they both operate the same way).

The following waveforms show the behavior of altsyncram megafunction for the chosen set of parameters in design testvector_rom.vhd. For the purpose of this simulation, the contents of the memory at the start of the sample waveforms is assumed to be (00000000FFFFFFF0, 00000000FFFFFFF1, 00000000FFFFFFF2, 00000000FFFFFFF3, ...). The design testvector_rom.vhd has one read port. The read port has 1024 words of 64 bits each. The core uses a different clock enable than the input registers. The output of the read port is registered by clock.

The waveform on the next page shows the behavior of the design under normal read conditions. The read happens at the rising edge of the enabled clock cycle. The output from the RAM is undefined until after the first rising edge of the read clock. The clock enable on the read side input registers is disabled. The clock enable on the output registers is disabled.

The two ROMs are identical except their names and that each has its own initialization file. The initialization files have a .mif extension and contain an initial value for every memory address of the ROM. The provided .mif files are identical except for the data stored at address 0x003. This is different so that your can verify that your testbed can detect errors.

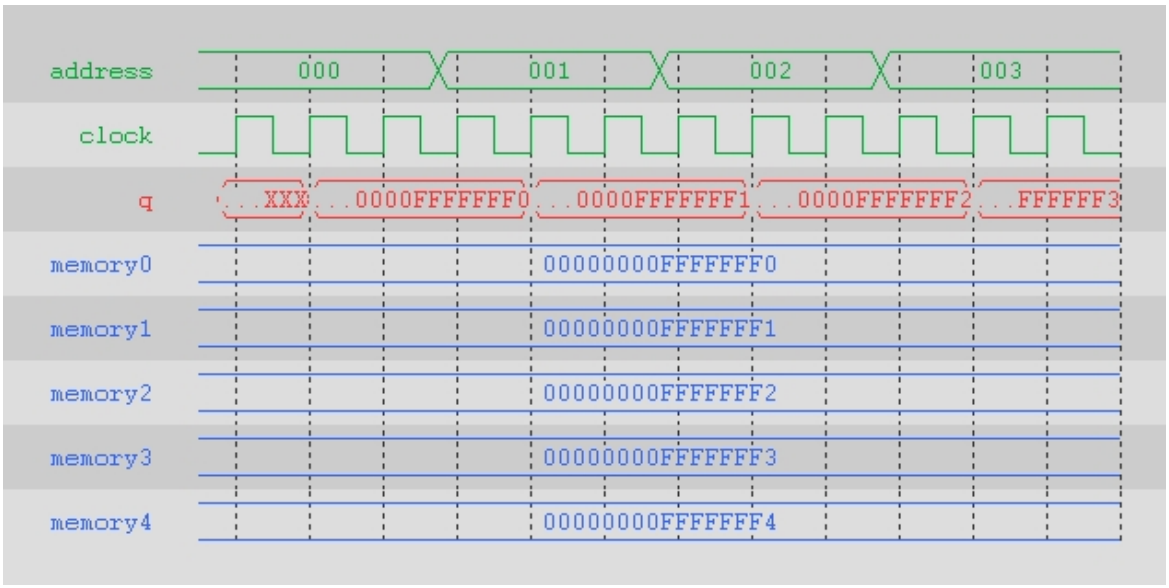

Fig. 9 : Wave showing read operation.

Setup the Testbed Project:

1. Make a new Quartus project for the testbed.
2. Include all of the testbed files in a new Quartus project directory. We will provide you with a template that includes some of the testbed components and you will be required to create the rest to complete the block diagram shown in Figure 6.
3. Add the files to your project.
4. Import the DE2-70 pin assignments to your project using the .cvs file from the course website and the "Assignments->Import Assignments…" command in Quartus.
5. Write the rest of the testbed.
6. Hit compile and download when finished.

Use the Testbed:

1. Before downloading set SW(1:0) to desired value to select one of four DES keys hardcoded in des_testbed.vhd. The .mif files you use should have valid inputs and expected outputs for this key. The key should not be changed during circuit operation.
2. KEY(0) is the circuit reset. This MUST be pressed after downloading to the board
3. LEDG(0) will light up as long as the output of your circuit matches that of the expected output ROM. If this stays lit then everything seems to be working fine.
4. LEDR(0) will light up the DES circuit output does not match the expected output.
5. If a mismatch is detected, the comparison circuit in the testbed will stall. No more comparisions will be done until after a system reset.
6. You can view the expected output, circuit output, and ROM address where mismatch occurred on the LCD. To view information on the LCD you must use SW(17:16) to select which value you want to display (00=circuit, 01=expected, 10 and 11 = address) and then push KEY(2) to update the LCD.

What you are going to do:

Task 1: Describe Behaviour in VHDL

The first thing to do for the project is to describe each of the subcomponents as separate modules using structural/behavioural VHDL as appropriate (remember it must synthesize). You might be tempted to describe the whole DES decryption circuit in one module, however, I strongly recommend that you break it up into sub-modules. The reason for this is that, if you make even a single bit-error in one of the lower-

level blocks (the S-boxes, for example), it will be really hard to trace the cause of the error. Describing each in a separate block will allow you to test each block separately, to help find any errors. I will give you test benches (along with test vectors) for the various permutation and expansion blocks; these will be useful only if you define these blocks as separate VHDL modules.

Here are the specifics. Templates are provided on the course webpage matching the I/O sequence given below. Below is a list of the major subcomponents listing the sequence of their I/O declarations (***Note that these are not proper VHDL entity declarations***).

Top Level: Filename des.vhd

I/O: (in this order):

DesDout	[1:64]	output
InFIFO_Rd		output
OutFIFO_Wr		output
DesDin	[1:64]	input
DesKey	[1:64]	input
InFIFO_Empty		input
OutFIFO_Full		input
Clk		clock input
Rst		reset input

Controller: Filename: control.vhd

I/O: (in this order):

shifttwo_s1	output
loaddata_s1	output
InFIFO_Rd	output
OutFIFO_Wr	output
InFIFO_Empty	input
OutFIFO_Full	input
Clk	clock input
Rst	reset input

Datapath: Filename: datapath.vhd

I/O: (in this order):

DesDout	[1:64]	output
DesDin	[1:64]	input
DesKey	[1:64]	input
loaddata_s1		input
shifttwo_s1		input
Clk		clock input
Rst		reset input

Data Unit: Filename: data_unit.vhd

I/O: (in this order):

dataout_s1	[1:64]	output
datain_s1	[1:64]	input
keysub_s1	[1:48]	input
loaddata_s1		input
Clk		clock input
Rst		reset input

Key Unit: Filename key_unit.vhd
I/O: (in this order):

keysub_s1	[1:48]	output
permkey_s1	[1:56]	input
shifttwo_s1		input
loaddata_s1		input
Clk		clock input
Rst		reset input

E-Box Filename: ebox.vhd
I/O: (in this order)

out_s1	[1:48]	output
in_s1	[1:32]	input

S-Sub Blocks Filename: ssub.vhd (collection of S-boxes)
I/O: (in this order)

out_s1	[1:32]	output
in_s1	[1:48]	input

Note that the ssub.vhd file may contain sub-modules for each of the sboxes as well.

Pbox: Filename: pbox.vhd
I/O: (in this order)

out_s1	[1:32]	output
in_s1	[1:32]	input

PC-2 Box: Filename: pc2.vhd
I/O: (in this order)

out_s1	[1:48]	output
in_s1	[1:56]	input

Shift28: Filename: shift28.vhd
I/O: (in this order)

out_s1	[1:28]	output
in_s1	[1:28]	input
shifttwo_s1		input

In-Permute Box: Filename: in_permute.vhd
I/O: (in this order)

out_s1	[1:64]	output
in_s1	[1:64]	input

Out-Permute Box:Filename: out_permute.vhd
I/O: (in this order)

out_s1	[1:64]	output
in_s1	[1:64]	input

Key-Permute Box:Filename: key_permute.vhd
I/O: (in this order)

out_s1	[1:56]	output
in_s1	[1:64]	input

The data and key units also contain registers, multiplexers, and exclusive or gates. You can either just include these inline within the data and key unit modules, or define separate modules for these components. If you decide to define separate modules, include them in the key_unit.vhd and data_unit.vhd files.

To help you test your specification, you can find testbenches/test vectors on the course webpage for the submodules like the s-box/e-box. At this point you will be using simulation so you can better see what's happening. Please remember that Timing simulation will provide more reliable feedback.

Important note: Those of you who are good with Google will likely find lots of VHDL descriptions of DES on the web. Feel free to use these as a ***reference***, but remember, your design has to match our specifications ***exactly***. Even if this were not a form of ***plagiarism***, it is probably more work to modify an existing DES implementation than to write your own given the information provided here. Also beware that at least a few of the on-line versions have bugs. Finally, remember any direct copying of on-line versions would be plagiarism resulting in the loss of 30% of your final mark (as specified during the first class)

Task 2: Integrate all the modules and do Preliminary Testing

Integrate the components from Task 1 to create your DES decryption circuit. Perform preliminary testing of the design using the supplied test vectors and Timing Simulation. Note, you will be required to submit a system block diagram for your circuit. If your VHDL and system block diagram are significantly different, this is a pretty good sign that you have plagiarized one of the two. Be sure to label the inputs and outputs of each module. Use the datapath techniques discussed in class and your understanding of FPGA architecture to maximize the (1/area*Max Throughput) product. Although your lab will be graded to scale based on this product, please note getting something to work is critical. If it's slow but working, you will get a significantly better mark than fast and small but not working.

Task 3: Create a Testbed

Although you will have *hopefully* ironed out most of the bugs based on your initial testing in Task 3. Real verification requires a testbed that can be configured with different *large* testbenches. You should create your testbed to match the structure described above.

Task 4: Thorough Testing of DES circuit with Testbed

The top-level test provided only test a few sample decryptions. This probably isn't sufficient. When we test your circuit, we will use a much more extensive set of test vectors (and we won't give you these test vectors). Thus, it would be a good idea to add enough test vectors until you are sure your specification is correct.

Using the Testbed created in Task 3, you will perform a thorough testing of your design. This means that they need to work together properly (you need to follow the controller format described). Note that you need to make sure your DES circuit and Testbed works for our specification and our test files (this is not unrealistic; in the real world, you will be given very exact specifications of what your design is supposed to do - in the real world, "off by one clock cycle" is just as bad as being completely wrong).

Task 5: Demonstrate your Design and Submit your Report.

During your demonstration, you will be asked to:

- Show your source code and answer questions about it
- Show a simulation of your DES circuit or any of the sub-circuits and explain what is going on in the waveform
- Show your circuit working on the DE2-70 board with the testbed and and different sets of input and output vectors.

Your written submission should include:

- A title page with your ***group number***, both students names, student numbers, course number, and date.
- A state diagram (what type of state machine did you use),
- A system block diagram,
- A summary of resource usage,
- The minimum clock period and maximum throughtput of your design,
- A hierarchy tree for your VHDL files
- Anything that you think is exceptional about the design of your DES decryption block or testbed.
- A zipped softcopy of all your source files. This must be messaged to the TA via WebCT. Source files ***must*** have comments, consistent indentation, adequate white spaces, descriptive names for signals/variables, one entity/architecture pair per file, and a maximum line length of 80 characters.

Please note that your reports are not marked by weight. Clearly written, short and succinct reports are highly preferable. There is no maximum or minimum page limit, but your target should be between 3-5 pages, not including title page. Also, a picture is worth a 1000 words, so if possible illustrate. Please do not include a background description of the generic DES algorithm, it is not necessary.

Your source files will be run through a comparison program to help identify copied segments of source code between groups and from the web. Copying any portion of another groups source files is considered cheating (and will result in an automatic 0/30 for your ***entire*** lab component in the course).

Marking Scheme

Demo

2 marks for working Simulations and reasonable responses to questions
2 marks for working on the DE2-70 and reasonable responses to questions

Total = 4

Report

1 mark for accurate State Diagram (must match VHDL)
1 mark for accurate System Block Diagram (must match VHDL)
1 mark for resource usage report
1 mark for min Clk period report and max throughput calculation
1 mark for hierarchy tree and VHDL coding practices
1 mark for readability of report

Total = 6

Suggested Milestones:

The following are some suggested milestones for this project. We aren't going to be checking your progress, but you can use this list to see if you are falling behind.

Start Project: February 1rst, 2009

Milestone 1: Build all DES components and test them to prove that they are functioning properly, February 8th, 2009

Milestone 2: Integrate all the components and do preliminary testing: February 11th, 2009

Milestone 3: Build Testbed: February 15th, 2009

Milestone 4: Finish a thorough testing of the DES module with your Testbed: February 22nd, 2009

Milestone 5: Submit your report on your design: February 26th, 2009

Bonus ONLY:

The design we've outlined here assumes 16 clock cycles are required to decrypt each data word. For those of you that are already comfortable with synthesizable VHDL, you can try to design a ***pipelined*** version of this circuit to increase throughput. This will be ***incredibly*** challenging and should not be considered by the inexperienced or faint of heart. If successful, you will be awarded a maximum of 2 bonus marks on your final mark.

What to hand in:

1) A copy of your lab report submitted into the ensc350 drop box by **3:30pm on February 26th, 2009**. Note the time, as we won't accept excuses (including jammed printers/no paper in the printers) and we won't be accepting soft copies.

2) A zip file with your VHDL files for both your DES decryption core and your testbed as well as a README file with hierarchy, group members and student IDs, group number. Please note this should be emailed to the TA's **WebCT** email account.

REMEMBER: Don't be late, as late submissions will be severely penalized (minimum of 50%) even if it is only by 1 hour.

Then you are done!!!