UNIVERSITY OF BRITISH COLUMBIA
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

EECE 479: Introduction to VLSI Systems
Fall 2007

Final Course Project

Due: November 30th, 2007: 11:59pm
Work on this project in groups of two or three[1]

In this project, you will design an *RC6 decoder*. RC6 was designed as a potential successor for the aging (and weak) DES encoding scheme. Although a competing scheme, Rijndael, was finally selected as the Advanced Encryption Standard (AES) by the U.S. Government, RC6 still is an efficient but strong encryption scheme, that will have applications when conformance to AES is not required. Although it is possible to implement an RC6 decoder in software, a VLSI implementation will give much higher performance, allowing it to keep up with faster communication rates.

A good description of RC6 can be found in http://people.csail.mit.edu/rivest/Rc6.pdf. RC6 is a parameterized standard, meaning the bit width and encryption strength can be adjusted depended on the available processing power. In this project, we will create a decoder that uses four *rounds* (each round is one clock cycle in our implementation). Each set of four rounds decrypts four eight-bit bytes (in the above document, which you do *not* have to read, this corresponds to w=8 and r=4). The above document describes both encryption and decryption, but in this project, we will only implement the decryption circuit (I will give you encoded messages, and your circuit should be able to decrypt them). We will hard-code the key in the chip (a more flexible implementation would allow us to set the key during operation, but that would require more simulation effort).

This handout first describes the decryption circuit in enough detail that you will be able to implement and test it. Then, the handout gives some guidance in exactly how to go about your implementation. This project will be a lot of work, so it is highly recommended that you get working on it early. A suggested schedule of tasks will be given; 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 project.

The RC6 Decryption Algorithm

Our decoder will decrypt four bytes at a time. The decryption is governed by a multi-bit key. This key is used to pre-calculate a set of constants that are used through the decryption (once the constants are calculated, the key is no longer needed). For this project, you do not need to worry about how the constants are calculated; I have already done that and will just give you the constants, and you will hard-code them in your chip. For a decoder with four rounds (such as ours) there are 12 constants; eight inside the datapath and four outside the datapath. If you want to read more about the constant calculation, see the above-referenced document on the web.

An algorithmic description of the algorithm is shown in Figure 1. You are not going to implement the algorithm in software, but it is provided here for completeness, and because it provides some insight into the amount of computation required. The four input bytes (containing the encoded text) appear in **A**, **B, C**, and **D**. Through a set of subtractions, multiplications, and rotations, the data is decoded, and the results are available in **A, B, C**, and **D** at the end of the algorithm. The constants described above are in the **S** array. Note that *all* mathematical operations are "modulo-256". This means that only the 8 least significant bits of the results are maintained. For an addition, this means discarding the carry-out. For a subtraction, it means discarding the borrow-out. As an example, if you subtract 00000001 – 00000010, you would get 11111111. For a multiplier, it means discarding bits 9 to 15 of the result (keeping only bits 0 to 7). The algorithm also involves "rotate" operations. Recall that a rotate operation is different than a shift, in that in a rotate, bits that are shifted out of the word are "rotated" back to the other end. So, if we rotate the number 01110000 to the left by 3 bits, we would get 10000011. Keep this in mind when you are working on this project; when I did it, I often forgot to implement the rotate or modulo-256 arithmetic properly, and spent a lot of time debugging these sorts of errors.

[1] If anyone wants to work alone, they need to implement the entire RC6 circuit as described here (I don't recommend this).

Inputs: Ciphertext stored in four 8-bit input registers *A,B,C,D*
Twelve 8-bit constants *S[0..11]*

Output: Plaintext stored in *A,B,C,D*

Procedure:
C = C - S[11]
A = A - S[10]
for *i=4* downto 1 do {
    (A,B,C,D) = (D,A,B,C)
    w = (D x (2xD+1)) rol 3
    t = (B x (2xB+1)) rol 3
    C = ((C - S[2i+1]) ror lsb3(t)) xor u
    A = ((A - S[2i]) ror lsb3(u)) xor t
}
D = D - S[1]
B = B - S[0]

Legend:
- : modulo-256 subtraction
+ : modulo-256 addition
x : modulo-256 mult.
rol n: rotate to left *n* bits
ror n: rotate to right *n* bits
xor : exclusive or
lsb3(n): the three least-significant bits of n

Figure 1: Algorithmic (software-like) description of the algorithm
(modified from R. Rivest et al, "The RC6™ Block Cipher", http://people.csail.mit.edu/rivest/Rc6.pdf)

In this project, we will create a hardware implementation of this algorithm. The rest of this section describes one possible hardware implementation (you can use your own design, but I highly recommend following the suggestions here, since then our test scripts will work with your sub-units).

A top-level diagram of the circuit is shown in Figure 2. The four input (decrypted) bytes come in on **inA_s1, inB_s1, inC_s1**, and **inD_s1** (each is 8 bits wide). Inputs A and C are then pre-conditioned by subtracting pre-determined constants (as described above, these constants were calculated based on a key). Remember that the subtraction is modulo-256, meaning we ignore the borrow-out.

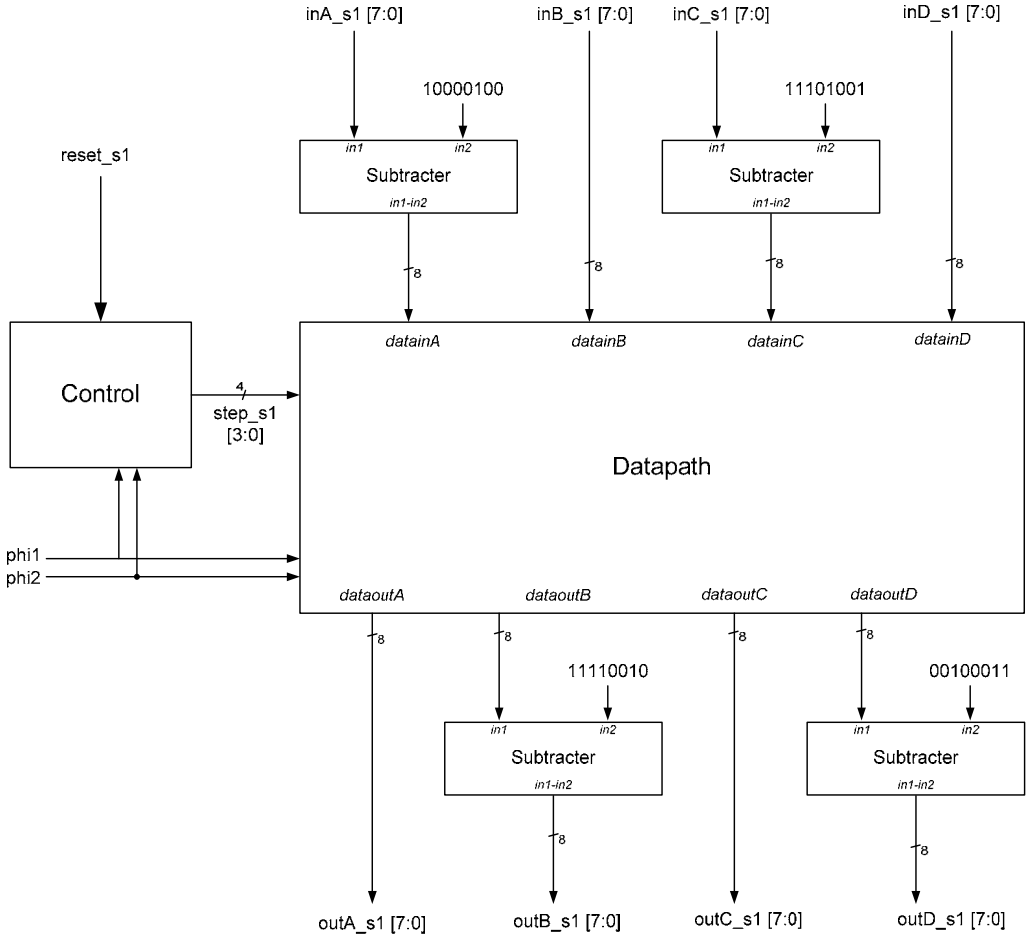

Figure 2: Top-Level Schematic

Inputs B, D, as well as the results of the two subtractions are provided to the datapath. The datapath operates for four cycles, and then produces four bytes on pins **dataoutA, dataoutB, dataoutC,** and **dataoutD** at the end of the fourth cycle (each cycle is called a *round*). Outputs B and D are then post-conditioned by subtracting two other constants using a modulo-256 subtracter as above. The final decrypted bytes are then available on **outA_s1, outB_s1, outC_s1,** and **outD_s1**. Four new encrypted data bytes can then be provided to the inputs; thus, every four cycles, four bytes can be decrypted.

The datapath is controlled by a control block. The control block produces a single 4-bit control word (called **step_s1**) that cycles through 0001, 0010, 0100, 1000 and then repeats. When **reset_s1** goes high, the control block output will go to 0001 during the *next* cycle (this is a synchronous reset). The four-bit control word is used by the datapath so that it knows which of the four rounds that make up each decryption we are in. Note: we could represent the round number using only two bits. Although it seems that this would minimize the number of wires, it turns out that it would make the implementation of the controller and datapath more complex (this will become clear later). The behaviour of the datapath when the control word is anything other than 0001, 0010, 0100, or 1000 is undefined; you need to make sure your control block never generates such outputs.

Figure 3 shows the schematic for the datapath block. The upper row of multiplexers select the four input bytes during the first round, and the feedback from the registers during each other round (remember that bit 0 of the **step_s1** input from the controller is high during the first round (cycle) of the decryption. After some shuffling, the bytes then are fed into one of four blocks (two are fed into **pathAC** blocks and two are fed into **pathBD** blocks). The internal structure of these blocks will be described later. Two of the outputs from the **pathBD** blocks (labeled **t_s1** and **u_s1**) are fed into the **pathAC** blocks as inputs. The **pathAC** blocks also each receive a constant as input. The left-most pathAC block receives one of 0xFB, 0x96, 0x76, or 0xA3, depending on the value of the step_s1 signals (0xFB is provided in the first cycle, 0x96 in the second cycle, etc.). The right-hand **pathAC** block receives four different constants, again depending on which cycle of the decryption we are in. Note these constants were pre-calculated based on the key; you will hardcode these constants in your design.

The output of the **pathAC** and **pathBD** blocks are then registered (meaning they pass through a register) and fed back to the input multiplexers. At the end of the fourth cycle, the outputs of these registers are sent out of the datapath, and after post-conditioning, as described above, are sent outside the chip.

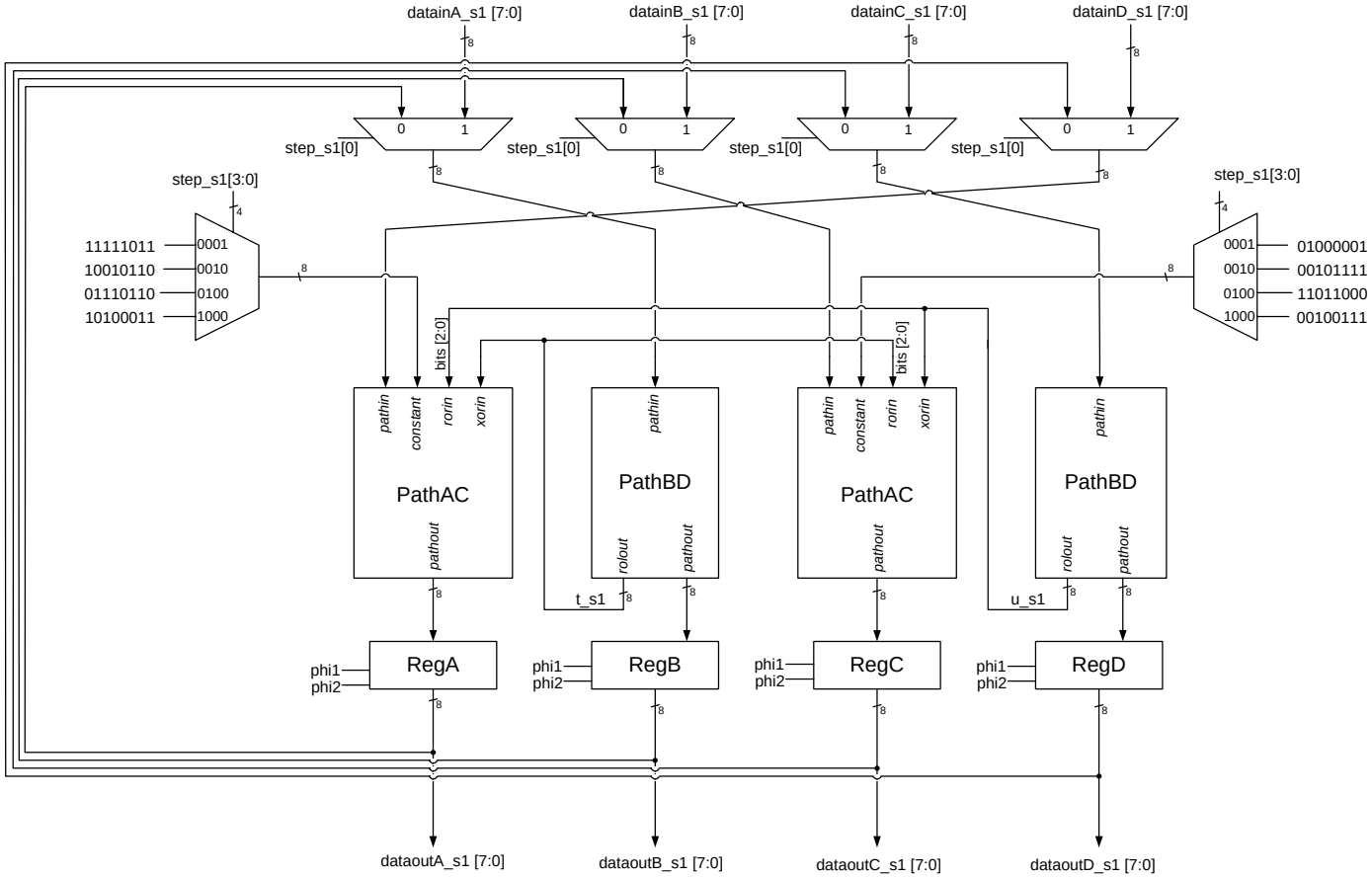

Figure 3: Datapath Schematic

The structure of each **PathAC** block appears in Figure 4. This block consists of a modulo-8 subtracter, a rotate right block, and eight exclusive-OR gates. Note that the "Rotate Right" block will rotate the input word by *n* bits, where *n* is any number from 0 to 7 (you can probably imagine how this could be implemented using 8 multiplexers, although other implementations are possible). Remember that this block is purely combinational (no clock input).

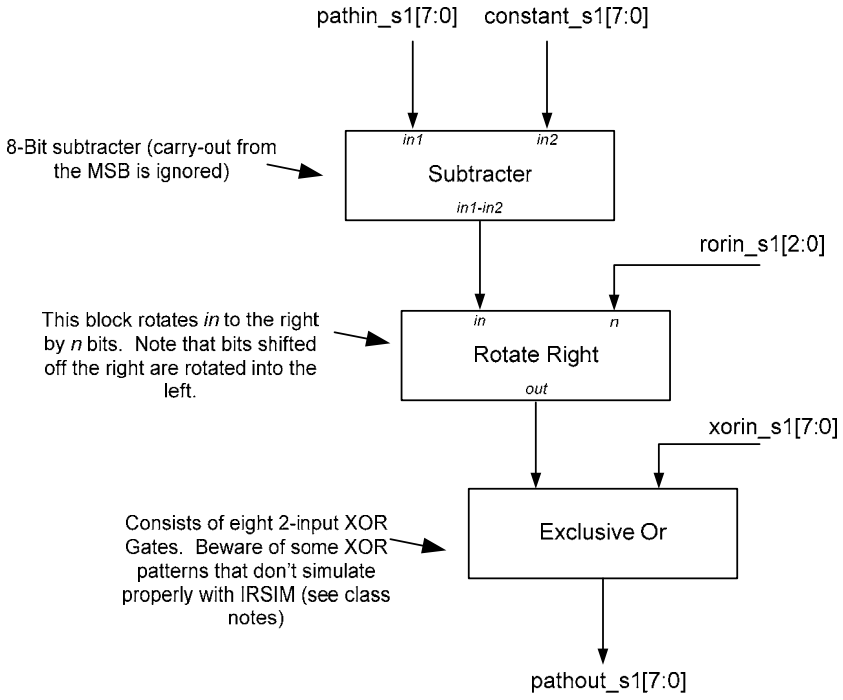

Figure 4: **PathAC** Schematic

The structure of each **PathBD** block is in Figure 5. The upper block implements the function *in*(1+2*in) modulo-256*. This will be explained in more detail below. The lower block rotates the input too the left by 3 bits. Note that this rotate block is very simple, and can be implemented without any transistors! Again, the **PathBD** block is purely combinational.

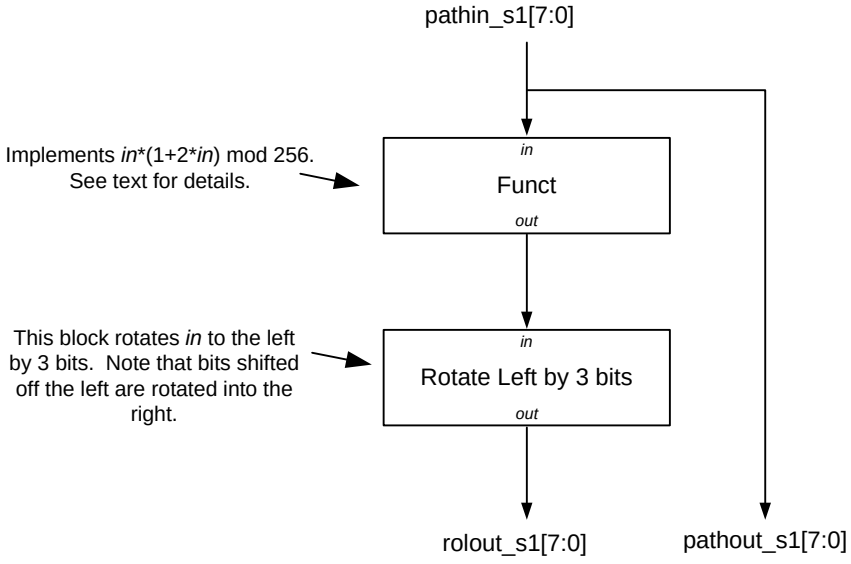

Figure 5: PathBD Schematic

A simple implementation of the **Funct** block would require a multiplier or shifter (for the *2), and adder (for the +1) and another multiplier. However, we can simplify this significantly. First, consider the operation *1+2*b*. Clearly, the *2*b* can be

implemented by a left-shift (discarding the upper order bit; remember, this is modulo-256 arithmetic). The addition can be implemented by simply setting the LSB of the shift result to a 1 (in other words, shift in a 1 instead of a 0). So, if the input is "$b_7b_6b_5b_4b_3b_2b_1b_0$" then the result of *(1+2*b)* is simply "$b_6b_5b_4b_3b_2b_1b_01$". No transistors are required to implement this!

Now consider the multiplication within the **Funct** block. From EECE 353, we learned how to make an 8x8 array multiplier using half adders and full adders, as shown in Figure 6. We can optimize this implementation significantly. Remember that since we are using modulo-256 arithmetic, we are going to discard the upper-order bits (bits 8 through 15) of the result. Thus, we can eliminate all the circuitry needed to produce those bits. As shown in Figure 6, this gets rid of about half of the gates in the multiplier.

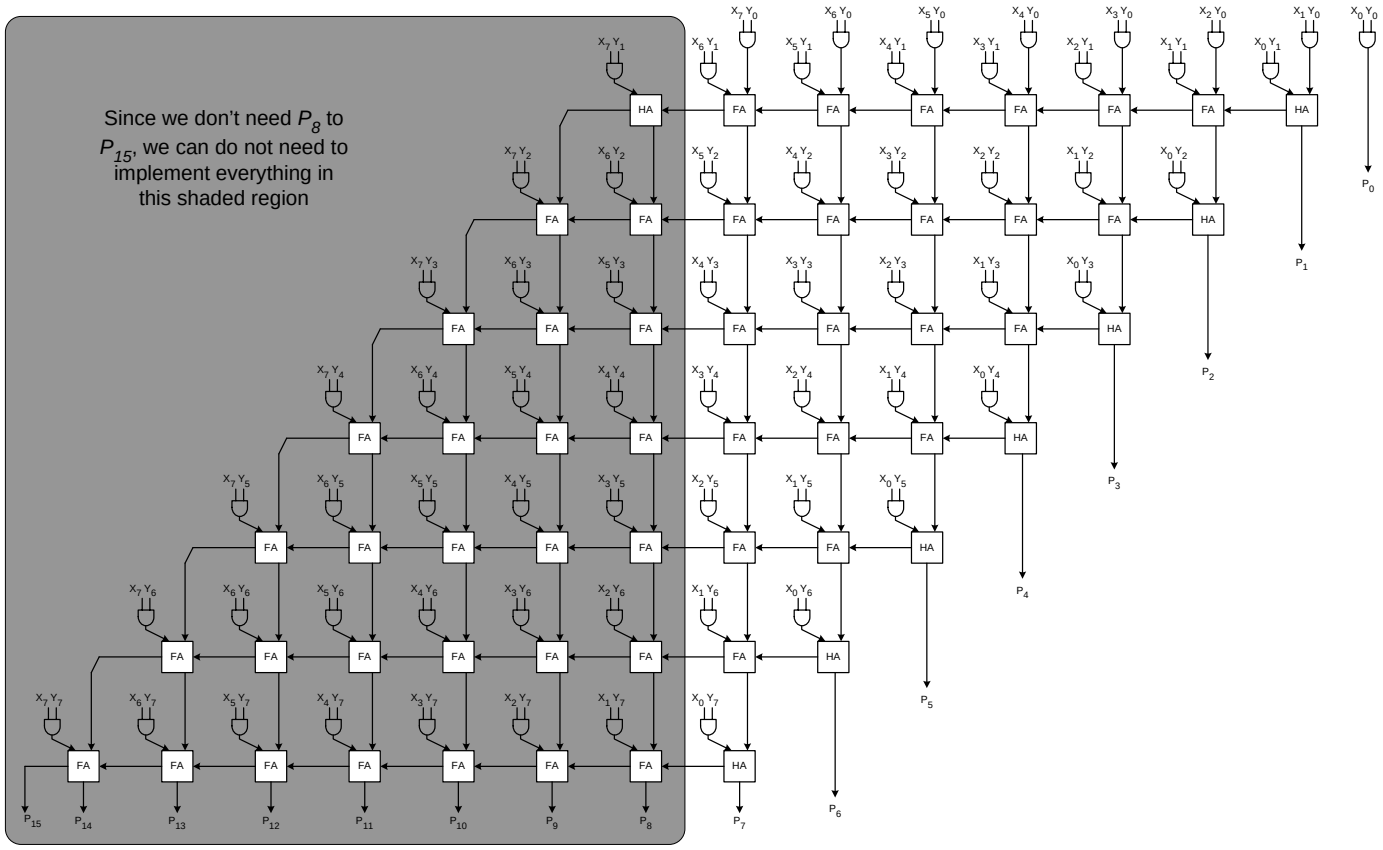

Figure 6: An 8x8 Array Multiplier

When it comes to lay this out, a few other optimizations are possible. First, notice that we can implement a half-adder out of a full-adder by tying the carry-in input to 0 (saves layout time). Second, at least one of the input bits is always 1. Finally, there is correlation in the input bits (remember, we are multiplying "$b_7b_6b_5b_4b_3b_2b_1b_0$" with "$b_6b_5b_4b_3b_2b_1b_01$"). You might find ways to take advantage of this, or you might decide that it is not worth your time to save just a few transistors (when I did it, I decided the latter). However, getting rid of half of the multiplier as shown in Figure 6 is certainly worthwhile.

Returning to Figure 2, we can see that the datapath is controlled by a **control** block. The operation of this block has already been explained -- the control block produces a single 4-bit control word that cycles through 0001, 0010, 0100, 1000 and then repeats. When **reset_s1** goes high, the control block output will go to 0001 during the *next* cycle (this is a synchronous reset). This is shown as a state machine in Figure 7. Of course, this doesn't mean it has to be implemented using normal state-machine techniques; one could imagine a four-bit shift register at the heart of the control block (this is up to you, you can do it however you want).

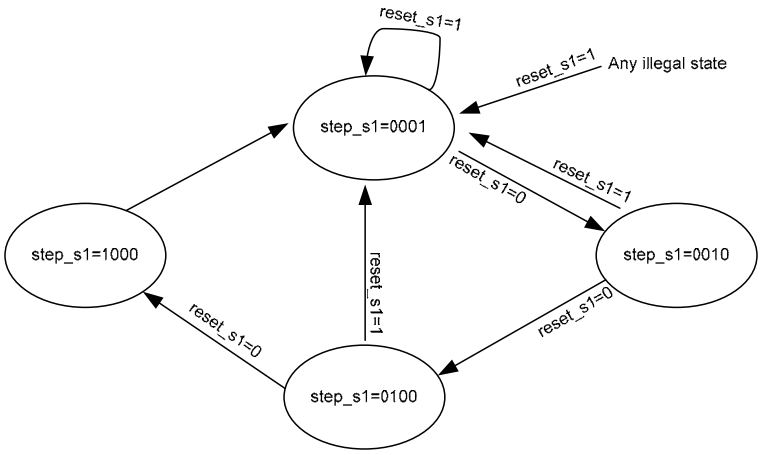

Figure 7: Control Block State Machine

The preceding should give you an overview of the RC6 algorithm. You will be able to find many other references and examples, including sample code, on the web (be aware that the sample code may assume other keys (constants) and different data widths and number of rounds). Testbenches for each component will be provided, so you can verify your understanding of each module.

What you are going to Do:

Task 1: Set up your Account

The files you need are on the department file system in ~eece479/proj_files. Copy these files to your own directory using:

cp –r ~eece479/proj_files ~/eece479/project

(the –r copies the directories recursively). In this directory, you will find stubs you should use for your Verilog specification, and testbenches you can use to test both the Verilog specification and the transistor-level design. Be sure to read all the README files carefully (they are text files).

Task 2: Describe Behaviour in Verilog

The first thing to do for the project is to describe the behaviour of the circuit using Verilog. You should have separate modules for the datapath block, control block, and top-level structure. You may want to break up your datapath into smaller blocks (using a structural description); that is up to you. Note that, in general, the lower-level you write your Verilog, the easier it will be (later) to be sure your transistor-level diagram is correct.

There is one thing that will help you here: if you specify your data elements as [7:0], then whenever you use the built-in "+", "-", and "*" operations in Verilog, modulo-256 arithmetic is automatically assumed. Thus, if you write something like:

```
assign x = y*(1+2*y)
```

it will automatically use modulo-256 arithmetic for all operations (which is what you want here).

You can find templates in **~eece479/proj_files/verilog** (you should use these template files to ensure that your description works with our tester). The I/O ports you should use are shown below:

Top Level: Filename: **top.v**

I/O: (in this order):

outA_s1	[7:0]	output
outB_s1	[7:0]	output
outC_s1	[7:0]	output
outD_s1	[7:0]	output
inA_s1	[7:0]	input
inB_s1	[7:0]	input
inC_s1	[7:0]	input
inD_s1	[7:0]	input
reset_s1		input
phi1		input
phi2		input

Controller: Filename: **control.v**

I/O: (in this order)

step_s1	[3:0]	output
reset_s1		input
phi1		input
phi2		input

Datapath: Filename: **datapath.v**

I/O: (in this order)

dataoutA_s1	[7:0]	output
dataoutB_s1	[7:0]	output
dataoutC_s1	[7:0]	output
dataoutD_s1	[7:0]	output
datainA_s1	[7:0]	input
datainB_s1	[7:0]	input
datainC_s1	[7:0]	input
datainD_s1	[7:0]	input
step_s1	[3:0]	input
phi1		input
phi2		input

*note: you may choose to use a reset signal for your datapath signals. If so, it is ok; just modify the **datapath.v** and **top.v** stubs appropriately.

*note: you can include any other files you want (if you want to make a structural description). Make sure they are appropriately named.

Place all your Verilog code in the directory **~/eece479/project/verilog**.

To help you test your specification, you can find a testbench file in **~eece479/proj_files/testbenches/verilog/test_top.v**. This file uses test vectors from the file **test_top.vec** (the vector file name is hard-coded within **test_top.v**). The testbench decodes the message and prints the ascii decoded message to the screen. Note that, in initial debugging, you probably want to display more than just the final ascii decoded value; you can modify the testbench as you see fit. Note that you need to make sure your specification works with 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).

There are four vector files (**message1.vec** to **message4.vec**), each corresponding to a different encoded message. In the initial distribution, **test_top.vec** is a copy of **message1.vec**. To test your design with the other test vector files, you can either rename the file to **test_top.vec**, or change the filename in **test_top.v**. When we test your circuit, we will use a much more extensive set of test vectors.

Important note: those of you good with Google might find lots of Verilog descriptions of RC6 on the web. Feel free to use these as a reference, but remember, your design has to match our specifications exactly. It is probably more work to modify an existing RC6 implementation than to write your own. Also beware that on-line versions may have bugs.

Task 3: Floorplan your Design

In the next few tasks, you are going to lay out this circuit. In order to plan your wiring, you should first create a floorplan. On a piece of paper, draw an outline of how you plan to lay out the cells. You should not provide the details of each cell, rather, each cell should be drawn as a box. At the periphery of each box, label the inputs and outputs of each cell, along with the metal/poly layers that these signals will use. Use the datapath techniques discussed in class, and attempt to make sure that as many connections as possible can be made by abutting cells. Also draw in the wires that must travel over the cells. Think about how you plan to lay out the controller logic. Also think about how you will distribute power, ground and the clocks.

Don't feel limited by the structure of the schematics earlier in this handout. It may be that you want to combine or interleave blocks (such as the **pathAC** and **pathBD** blocks) in some way. Be creative.

Note that even though you will not hand in this floorplan, the quality of the layout you produce will be significantly impacted by how carefully you plan your design in this task. In the 2006 offering of this course (when we did a DES decoder), the chip area varied by *a factor of more than 10* between the smallest and largest chip in the class (I'm not kidding!). In the worst case, if you don't plan well, you might find yourself laying out cells more than once if you get the wrong aspect ratio/IO locations the first time. This would be really bad.

Task 4: Layout Datapath Cells

Note: the order in which you lay out your datapath may vary, depending on your floorplan above. Each of the cells should be placed in the file **~/eece479/project/layout**. If your **datapath.magic** cell works, then we won't care about the individual sub-units within the datapath when we are marking, however, it is strongly urged to work through the layout carefully and test all cells as you go. Depending on your design, you may need other primitive cells not listed here, in that case, choose meaningful filenames for each cell.

a) Lay out the 8-bit subtracter. Remember that in EECE 353, we talked about how to implement a subtracter using an adder, and we talked about how to build an 8-bit adder using eight full-adders. I would start by laying out a full-adder, replicating eight times, and add any extra logic you might need to make the subtracter. Call your design **sub8.magic**. You can use the test file **sub8.cmd** to test your design using IRSIM; you will find this file in **~/eece479/project/testbenches/irsim/subunits/sub8/sub8.cmd**. If IRSIM does not report any errors when you use this .cmd file, that suggests that your design might be correct (you may want to run extra tests until you are completely sure your design is correct).

b) Lay out an 8-bit XOR block and an 8-bit two-input multiplexer. Use the filenames **xor8.magic** and **mux2_8.magic**. You can test your design using the files **~/eece479/project/testbenches/irsim/subunits/xor8/xor8.cmd** and **~/eece479/project/testbenches/irsim/subunits/mux2_8/mux2_8.cmd**.

c) Lay the 8-bit Rotate-Right block. Remember this can rotate by any number of bits (0 to 7). A common error here will likely to create a shift block rather than a rotate block (when bits are shifted off the LSB of the word, they should "wrap-around" to the MSB, as described earlier). You might implement this using a set of eight eight-to-1 multiplexers, or perhaps you can think of some other interesting design. Use the filename **ror8.magic**. You can test your design using **~/eece479/project/testbenches/irsim/subunits/ror8/ror8.cmd**.

[Note: steps (d) and (e) may be done together, depending on how you are optimizing your multiplier. You may not have a separate multiplier layout, but you should definitely have a working **Funct** block layout].

d) Using the circuit described earlier in the handout, layout the modulo-256 multiplier. Be sure to take advantage of any optimizations you can think of. You can find a testbench for the multiplier in **~/eece479/project/testbenches/irsim/subunits/mult8/mult8.cmd**, however, beware that this expects a general 8-bit modulo-256 multiplier (if your design assumes that bit 0 of one of the numbers is a 1, for example, then this testbench will not be useful for you… in that case, you can write your own fairly easily).

e) Use the multiplier to create the **Funct** block (this is the block that implements $b*(2b+1)$). Your layout should be named **func8.magic**. Use the optimizations described earlier in this handout, and any others you might think of. You can find a testbench in **~/eece479/project/testbenches/irsim/subunits/func8/func8.cmd**. Be sure to spend some time testing this block, since it is one of the more complex in the project.

f) Using the above submodules, and any others you might need, create the **pathAC** and **pathBD** modules. We have not provided testbenches for these sub-units, but you can create your own. Note: depending on your floorplan, you may not have separate **pathAC** and **pathBD** files; that is ok.

g) Everything up to now is combinational. Next, layout the registers. Remember that these are two-phase clocked registers (**phi1** and **phi2**) consisting of two latches; the first is clocked by one of the clocks, and the second is clocked by the other clock (which is which? You should be able to figure this out). The inputs and outputs of the registers are all _s1 signals. Use the filename **reg8.magic**. You can test your design using **~/eece479/project/testbenches/irsim/subunits/reg8.cmd**.

h) Now, combine all the sub-units, and any others that you might need, to create the datapath. Call your layout **datapath.magic**. You can test your datapath using **~/eece479/project/testbenches/irsim/subunits/datapath/datapath.cmd**.

I expect that, for most people, the datapath will not work correctly the first time. To help you debug, consider adding "w" statements to the **datapath.cmd** files to observe some of the internal signals within the datapath. This will help you trace through the design to find your errors. I would not be surprised if the bulk of your debugging time is spent debugging the complete datapath; once you get this working, the rest should go smoothly.

Task 5: Layout of the Controller

Layout the controller. I can imagine creating the controller using a 4-bit rotate register with some extra logic to work with the synchronous reset, however, if you want to use the normal state-machine design techniques from 2nd year, you can do that. Remember that the output is 4 bits wide as described earlier in this handout.

You should call your design **control.magic,** and you can test it using **~/eece479/project/testbenches/irsim/subunits/control/control.cm**d.

Do not go on until the both the controller and the datapath works.

Task 6: Putting it all Together

This is it! Create a final layout in cell **~/eece479/project/layout/top.magic**. Include your controller and datapath cells as well as the two pre-conditioning subtracters and two post-conditioning subtracters (see Figure 2). The inputs and outputs of the chip should be placed on the periphery of the chip, and given *exactly the same names as in Task 2*. Note that only one instance of each input/output should appear. You will also have **Vdd** and **GND** inputs; you can have multiple **Vdd** and **GND** inputs if you want.

Extract your chip, and test it using IRSIM. Some sample .cmd files are provided in **~/eece479/project/testbenches/irsim/top**. Four control files are provided, each corresponds to a different encoded message. When you run IRSIM with these .cmd files, you will see the ASCII output. To see the text version of the output, you can run it through the perl script "print_output.pl" as follows:

```
irsim scmos50.prm –top_message1.cmd | perl print_output.pl
```

(the | is a pipe operator… it sends the output of IRSIM to the input of Perl).

If you can read them all, it seems pretty likely that your design works properly. Feel free to create additional test files if you want to be extra sure that your design is correct.

(continued on next page…)

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: October 25th, 2007

Milestone 1: Finish Tasks 1, 2 and 3: November 1st, 2007

Milestone 2: Finish Task 4: November 15th, 2007

Milestone 3: Finish Task 5: November 22th, 2007

Milestone 4: Finish Task 6: November 30th, 2007

What to hand in:

If you have followed the above instructions correctly, all your files will be in the right place. The only thing you need is an info.txt file in **~/eece479/project/info.txt**. This file has four parts. In the first part, include the names, numbers, and email addresses of each of your group members. Be sure your email addresses are valid, because that is where your mark will be sent. Separate your names from your email address by a comma.

The second part of the file asks you to specify whether your design works or not. You should enter a **Y** or an **N** (only **Y** or **N**). If your design worked, use **Y**. If your design didn't work, use **N**. Note that if you say **Y,** and your design doesn't work, you will loose many more marks than if it doesn't work and you say **N**, so make sure you test your design thoroughly before saying **Y**. Also, notice you have to say **Y** or **N**; no Maybe's. You can *not* get full marks if you leave this line blank.

The third part of the file can contain a text message explaining either why the design did not work (what you think the problem is: the more specific you can be, the more part marks you will get) and/or anything exceptional about your design you want the markers to know. Filling out this part of the file is optional.

The fourth part of the file asks you to disclose any help you might have received in completing the assignment. If you received any help (other than from the professor, the TA, or other members of your group) disclose it here. If you have not received any help, enter **NONE**. Do not leave this section blank. We will be carefully comparing assignments to look for unauthorized collaboration. Any un-disclosed collaboration will be reported to the Dean for possible referral to the discipline committee. Disclosing any help you may have received protects you from this (of course, you may not receive full marks if you copied someone's cell, but that is better than having the case referred to the Dean). If this is unclear, please contact the professor or TA *before* handing in your assignment.

Once you are done, and have created the info.txt file, use the handin program:

```
handin eece479 project
```

Then you are done!!!