NWI-IMC039 Cryptographic Engineering (KW3)
Hardware Assignment

Job Douma (s1499734) and Maarten Dorrestijn (s4665198)[1]

[1]TRUe Master Cyber Security, Radboud University Nijmegen

July 6, 2020

Contents

1 Introduction

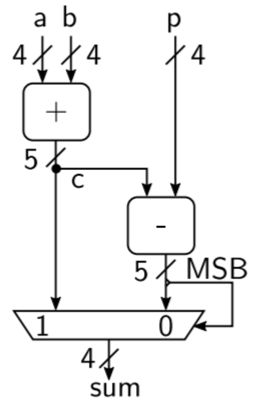
In this report, a design and simulation of a hardware architecture is going to be presented for elliptic curve cryptography. First, the datapath is build by simulating a simple n-bit modular adder architecture (2.1). Next, the n-bit modular adder will be extended with multiplier, addition and subtraction arithmetics. Furthermore, a memory unit and control logic is going to be added to implement an elliptic curve point scalar multiplication.

Additionally we chose to do one of the bonus exercises (2.9), namely the two hardware units, one computing point addition while the other computes point doubling. Note that the scope of this report is to only perform behavioral simulation using the VHDL design files which has been downloaded from the Cryptographic Engineering (KW3 V) Brightspace page.

2 Exercise Solutions

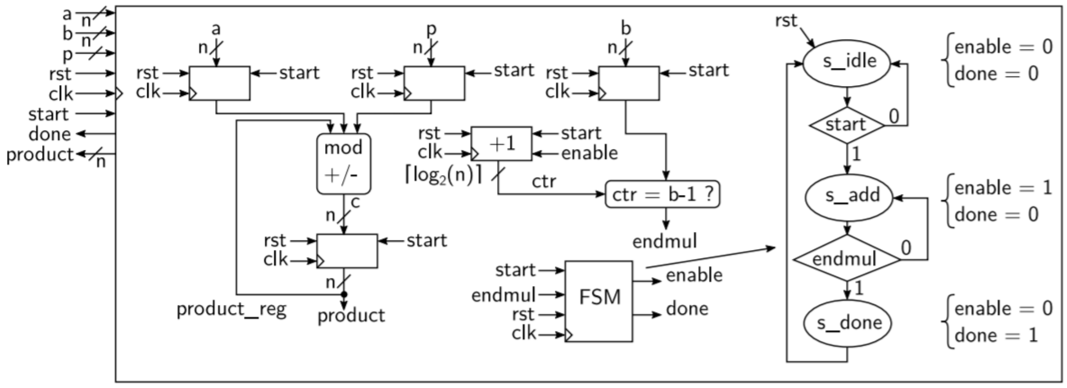
2.1 Exercise 4: n-bit Modular Adder

In this exercise a n-bit modular adder architecture has been designed starting from the design module template `modaddn.vhd`. The modular adder is created by modifying the `modadd4.vhd` architecture, from Exercise 3 of the tutorial, and replacing every instance of 4 with n and every instance of 3 with $n-1$.



2.2 Exercise 7: n-bit Modular Multiplier (addition)

Starting from the design module template `modaddn_mult.vhd` the goal is to build a modular multiplier by consecutive modular additions. The modular multiplication architecture is created by setting a counter to 0, and iteratively adding the variable a to an intermediate result register. After the counter reaches the value of b, the calculation is done. Note that this is a very inefficient way of implementing a modular multiplier, because it requires an impractical amount of additions to perform a simple modular multiplication.



2.3 Exercise 8: n-bit Modular Multiplier (double-and-add algorithm)

Starting from the design module template `modmultn.vhd` the goal is to design and simulate an n-bit modular multiplier through a left-to-right modular double-and-add algorithm.

The modular multiplication architecture is created by setting a counter to zero, and iteratively looping over the steps of the algorithm. The steps are executed by keeping track of a register s, which stores the value of s of the algorithm. Each loop, two additions are done. The first addition is the addition of s with itself, and the second addition is the result of the first addition and the output of a multiplex (mux). This mux either selects 0, or the value of a. The selection is done based on the most significant bit (MSB) of b. After the counter reaches n, the result of the multiplication is stored in s. The drawing shows a simple state machine that keeps track of whether were performing the loop of the algorithm, or whether we are idle or done. The drawing shows the loop of the algorithm being executed by two additions and a mux. The diagrams for this exercise can be viewed in appendices 3 and 3.

2.4 Exercise 9: n-bit Modular Multiplier/Addition/Subtraction

In this exercise the previous n-bit Modular Multiplier (2.3) that was able to perform multiplication is going to be expanded to additionally perform modular addition and subtraction. To choose the operation in the arithmetic unit an extra input port called `command` of two bits is added. This extra input will works as follows:

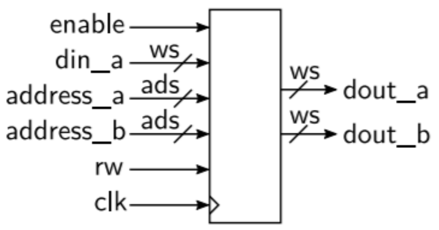
Command	
00	Do nothing
01	Modular multiplication ($a \times b \bmod p$)
10	Modular addition ($a + b \bmod p$)
11	Modular subtraction ($a - b \bmod p$)

This arithmetic unit works by feeding the inputs to a modular addition unit, a modular subtraction unit, and a modular multiplication unit. The outputs are selected based on the command. If addition or substraction is selected, the 'done' signal is raised immediately. Otherwise, the done signal is passed along from the modular multiplication unit. The drawing shows the output of the various arithmetic modules being muxed, and the 'done' output also being muxed. The diagram for this exercise can ben viewed in appendix 3.

2.5 Exercise 11: Double Ram

Given the `ram_simple.vhd`, from exercise 10 of the tutorial, the objective of this exercise is to construct a similar unit `ram_double`, which can perform two reads in one cycle instead of one. This concept represent the ram baseline for the ECC unit, since the operations are described by two inputs and one output.

The `double_ram` architecture is simply a copy of the `single_ram` unit, but with extra operations for the input and output. The two units differ in the fact that the `double_ram` unit has two address inputs and two outputs instead of one. On the rising edge of the clock and when enable is true, the addresses of both a and b, which are persistent in the memory, are written to their respective output. The `enable`, `din`, `rw` and `clk` variables are persistent throughout both units `single_ram` and `double_ram`.



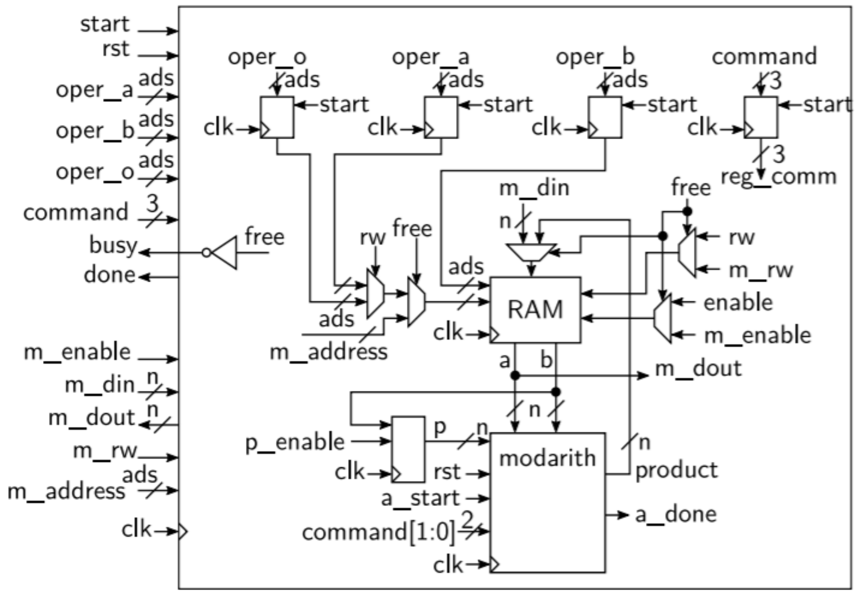
2.6 Exercise 12: Arithmetic Unit with Double Ram

The next step for the architecture is to join the arithmetic unit (2.4) with the the double ram unit (2.5). By joining them, the values from the ram can be loaded into the arithmetic unit. Furthermore, a computation of the desired operation can be performed, and finally the output can be rewritten into the ram. The rewriting of the ram will be the last step in the process, since we can iterate through all instructions necessary to compute the elliptic curve point addition and doubling.

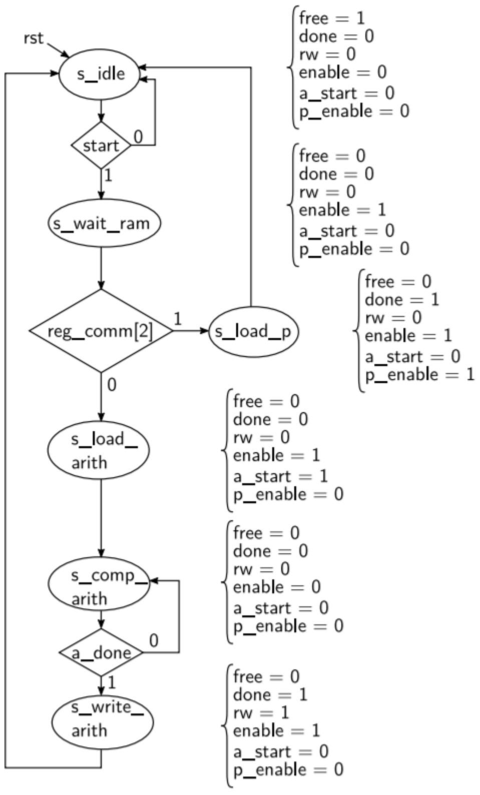
Command	
000	Do nothing
001	Modular multiplication ($a \times b \bmod p$)
010	Modular addition ($a + b \bmod p$)
011	Modular subtraction ($a - b \bmod p$)
100	Load p ($p \leftarrow b$)
101	Do nothing
110	Do nothing
111	Do nothing

Given the `ecc_base.vhd`, the first step was to construct components for the `modarith` and `ram_double` units and to map their respective ports to the ones already present in the `ecc_base` architecture. This way we could concurrently redefine what was already present in the architecture figure below. Furthermore, registers for operators o, a, b and `command` have been constructed, which define value should pass the register depending on the clock and the start value. If the rising edge of the clock is triggered and the start value is set to high then the register should pass the value of the respective operator down the line.

The definition of the muxes were a bit tricky as it was unclear which of the two input values should trigger and propagate through the mux at the trigger value zero or one. For instance take the mux where the variable `free` is the trigger value (right below register a). Would this figure imply that `m_address` is triggered on the expression `free = '0'` or on the expression `free = '1'`? The same question holds for all the remaining muxes. Through some careful use of the testbench we found the correct value for each of the mux as can be verified in our `ecc_base.vhd` submission.

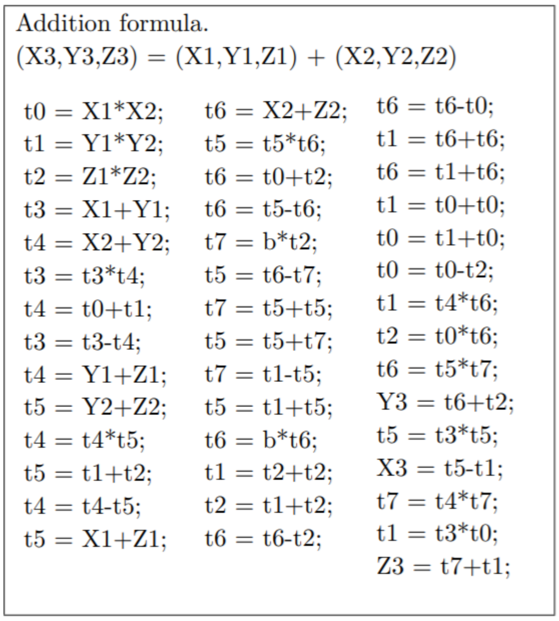


Finally, the states have been defined using the figure below. The states have been implemented as a switch case that sets every individual variable to the correct value depending on the state. During the debugging using the testbench, especially during the mux part, it was considerably easier to keep an overview of the states and in which state the error occurred, such that a solution was often found rather quickly.


rst
s_idle
start
0
1
s_wait_ram
reg_comm[2]
1
s_load_p
0
s_load_arith
s_comp_arith
a_done
0
1
s_write_arith
free = 1, done = 0, rw = 0, enable = 0, a_start = 0, p_enable = 0
free = 0, done = 0, rw = 0, enable = 1, a_start = 0, p_enable = 0
free = 0, done = 1, rw = 0, enable = 1, a_start = 0, p_enable = 1
free = 0, done = 0, rw = 0, enable = 1, a_start = 1, p_enable = 0
free = 0, done = 0, rw = 0, enable = 0, a_start = 0, p_enable = 0
free = 0, done = 1, rw = 1, enable = 1, a_start = 0, p_enable = 0

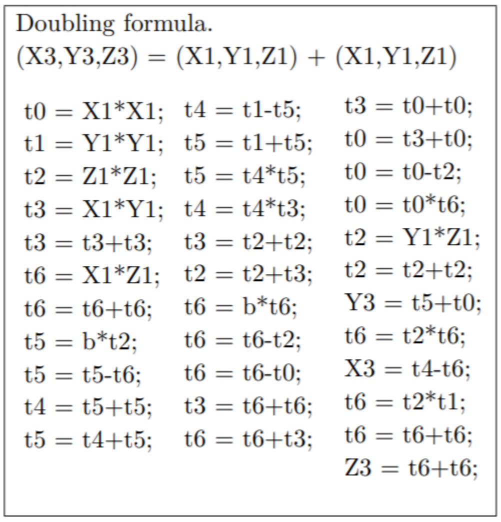
2.7 Exercise 13: Point Addition/Doubling Architecture

The consecutive step is to extend the previous `ecc_base` architecture (2.6) in order to be able to perform point addition or point doubling for Weierstrass elliptic curves. In our elliptic curve core we will be using the complete formulas of Renes, Batina and Costello [2]. Those formulas work with all elliptic curves with prime order and in Weierstrass format, but in this exercise we will try to focus on the NIST P-256 curve [1]. The formulas below are also only for Weierstrass curves with constant $a = -3$.



Addition formula.
(X3,Y3,Z3) = (X1,Y1,Z1) + (X2,Y2,Z2)

```
t0 = X1*X2;   t6 = X2+Z2;  t6 = t6-t0;
t1 = Y1*Y2;   t5 = t5*t6;  t1 = t6+t6;
t2 = Z1*Z2;   t6 = t0+t2;  t6 = t1+t6;
t3 = X1+Y1;   t6 = t5-t6;  t1 = t0+t0;
t4 = X2+Y2;   t7 = b*t2;   t0 = t1+t0;
t3 = t3*t4;   t5 = t6-t7;  t0 = t0-t2;
t4 = t0+t1;   t7 = t5+t5;  t1 = t4*t6;
t3 = t3-t4;   t5 = t5+t7;  t2 = t0*t6;
t4 = Y1+Z1;   t7 = t1-t5;  t6 = t5*t7;
t5 = Y2+Z2;   t5 = t1+t5;  Y3 = t6+t2;
t4 = t4*t5;   t6 = b*t6;   t5 = t3*t5;
t5 = t1+t2;   t1 = t2+t2;  X3 = t5-t1;
t4 = t4-t5;   t2 = t1+t2;  t7 = t4*t7;
t5 = X1+Z1;   t6 = t6-t2;  t1 = t3*t0;
                           Z3 = t7+t1;
```



Doubling formula.
(X3,Y3,Z3) = (X1,Y1,Z1) + (X1,Y1,Z1)

```
t0 = X1*X1;  t4 = t1-t5;  t3 = t0+t0;
t1 = Y1*Y1;  t5 = t1+t5;  t0 = t3+t0;
t2 = Z1*Z1;  t5 = t4*t5;  t0 = t0-t2;
t3 = X1*Y1;  t4 = t4*t3;  t0 = t0*t6;
t3 = t3+t3;  t3 = t2+t2;  t2 = Y1*Z1;
t6 = X1*Z1;  t2 = t2+t3;  t2 = t2+t2;
t6 = t6+t6;  t6 = b*t6;   Y3 = t5+t0;
t5 = b*t2;   t6 = t6-t2;  t6 = t2*t6;
t5 = t5-t6;  t6 = t6-t0;  X3 = t4-t6;
t4 = t5+t5;  t3 = t6+t6;  t6 = t2*t1;
t5 = t4+t5;  t6 = t6+t3;  t6 = t6+t6;
                          Z3 = t6+t6;
```

Given the `ecc_add_double.vhd`, the first step was to construct a component for the `ecc_base` unit and to map their respective ports to the ones already present in the `ecc_add_double` architecture. Next we added every single addition and doubling formula as shown in the picture above. We noticed after the creation of both formulas that we worked with numbers, but a better solution was to define variable names like $a = 00001$ and use these variables throughout our formula definitions. This solution will not reduce the number entries in each formula but will result in a better overview using less code for each formula.

The circuit works with a “start = 1” to initialize the operation, where point addition is done when “add_double = 0”, and doubling should occur when “add_double = 1”. The “busy” signal will be 1 when the unit is performing computations, and one cycle before it finishes the signal “done” will go from 0 to 1 and will go back to 0 in the next cycle. When no computations are being performed the signal “busy” is going to be 0. We implemented these functionalities in FSM processes for state and execution purposes.

2.8 Exercise 14: Scalar Multiplication Montgomery Power Ladder

The final exercise is comprised of a design and simulation of an architecture which performs scalar multiplication. In this exercise we created a testbench for the NIST P-256 with at the tests described in the tutorial. Note that we did not add any additional tests to the testbench. The code is stored in ecc_mont.vhd and tb_ecc_mont.vhd

The implementation of this exercise is executed a bit differently than was suggested in the assignment. Instead of changing the `ecc_add_double` element, we used this element as a simple primitive for point adding and doubling and extracted the results after every operation. To achieve this, we created many states that handled the different aspects of loading the variables and executing point addition and doubling. The states that has been implemented can be divided in roughly three categories:

- Preparation before the main loop of the algorithm.
- The main loop of the algorithm.
- The end state.

The preparation states handle the loading of the variables relating to the curve into the `ecc_add_double` element. After these states have been executed, the states that deals with the main loop are executed. By observing that each loop of the algorithm will consists of two operations, the states that deal with the main loop can also be divided into two groups; one group dealing with the first operation (`op1`) and one dealing with the second operation (`op2`). One operation consists of loading the right x, y and z values for the points involved, executing the operation and retrieving the values.

A counter was created to keep track of the iterations of the algorithm, and the inputs and outputs of the operations are determined based on the bit of the secret that is selected by the counter. This approach has the advantage that the `ecc_add_double` element can be kept as simple as possible, and that no secret-dependent memory locations are accessed. However, it did result in a lot of states, and as such, we have chosen not to draw the state diagram. The diagram shows many of the inputs to `ecc_add_double` as results of muxes based on the state, and for the precise actions that are taken per state, please consult `FSM_execute` in `ecc_mont.vhd`. The diagram is shown in appendix 3.

The compilation time of this file was very high on our hardware (50 minutes) and created a very large `.ghw` file. As such, we have used the `.fst` format for compilation with a reduced set of signals that are exported. The commands that were used to compile these files, as well as a compiled .fst file are included.

2.9 Exercise 15: Advanced Unit

In this exercise we expanded the previous design (2.8) by the optimization idea of two hardware units, one computing point addition while the other computes point doubling.

As an improvement, we chose to use two hardware units to speed up the computations by performing the point adding and point doubling in parallel. As such, we copied the design from 2.8 and duplicated the `ecc_add_double` elements and every signal that they were attached to. We changed the orders of the states in such a way that every iteration of the algorithm would start with loading the values required for the first operation and start that operation, followed by the loading of the second operation and the starting of that operation. After this, the states do not progress until both computations are finished. After storing the results, an iteration of the main loop is completed. By doing this, we almost reached a 2x speedup, going from a total of 19578940 nanoseconds to 10232380 nanoseconds.

Because copying every individual element of Exercise 14 and drawing it twice in the design would use too much space, we drew the design as the combination of twice the element that was created for this exercise. The diagram is shown in appendix 3. The files are saved as `ecc_mont_opt.v"` and `tb_ecc_mont_opt.vhd`.

3 Conclusion

Overall, the concepts of designing and programming hardware units was very interesting in our opinion. It was a nice change of phase to do something different than the usual imperative programming. However, in our opinion, VHDL is not the easiest languages to pick up, and given the somewhat difficult debugging lead to some difficulties. These difficulties have been overcome by persistent email contact and good corporation within the group. All in all, the project was conceptually very interesting but sometimes complex to design and implement, just as what we would expect from a master course.

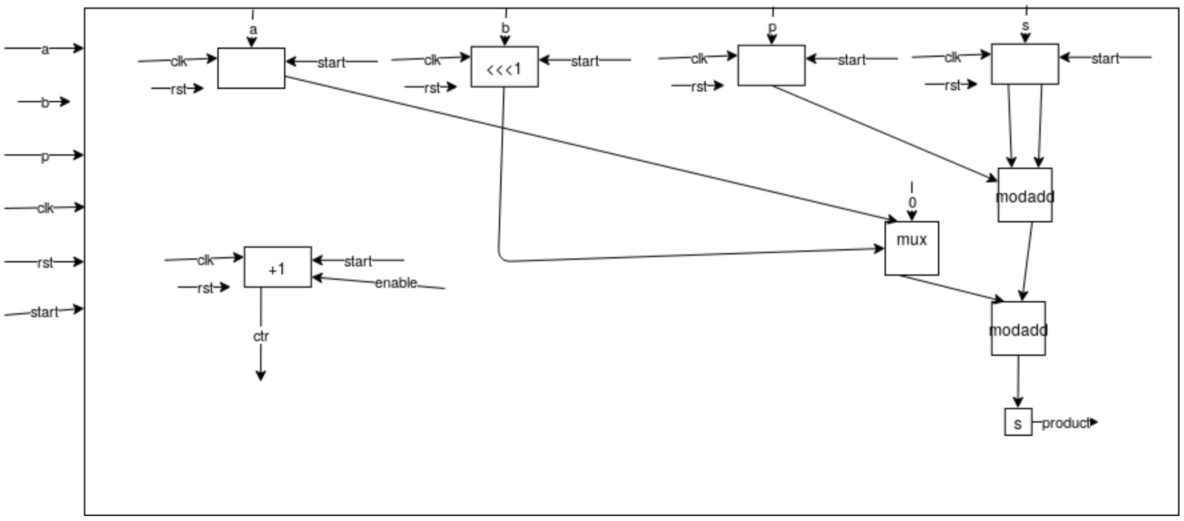
One last note to the lecturers of the Cryptographic Engineering Hardware part; it was unknown to us that exercise 15 is optional and used as bonus points. Only after email contact it became apparent to us the purpose of this exercise. Make sure to formulate or define this more clearly for the following years.

References

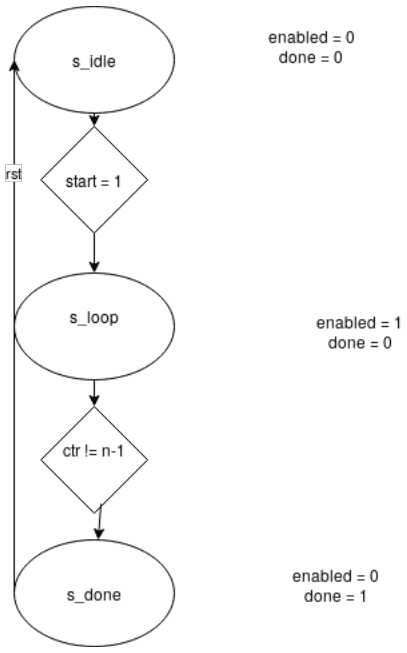
[1] Mehmet Adalier et al. Efficient and secure elliptic curve cryptography implementation of curve p-256. In Workshop on Elliptic Curve Cryptography Standards, volume 66, 2015.

[2] Joost Renes, Craig Costello, and Lejla Batina. Complete addition formulas for prime order elliptic curves. Cryptology ePrint Archive, Report 2015/1060, 2015. `https://eprint.iacr.org/2015/1060`.

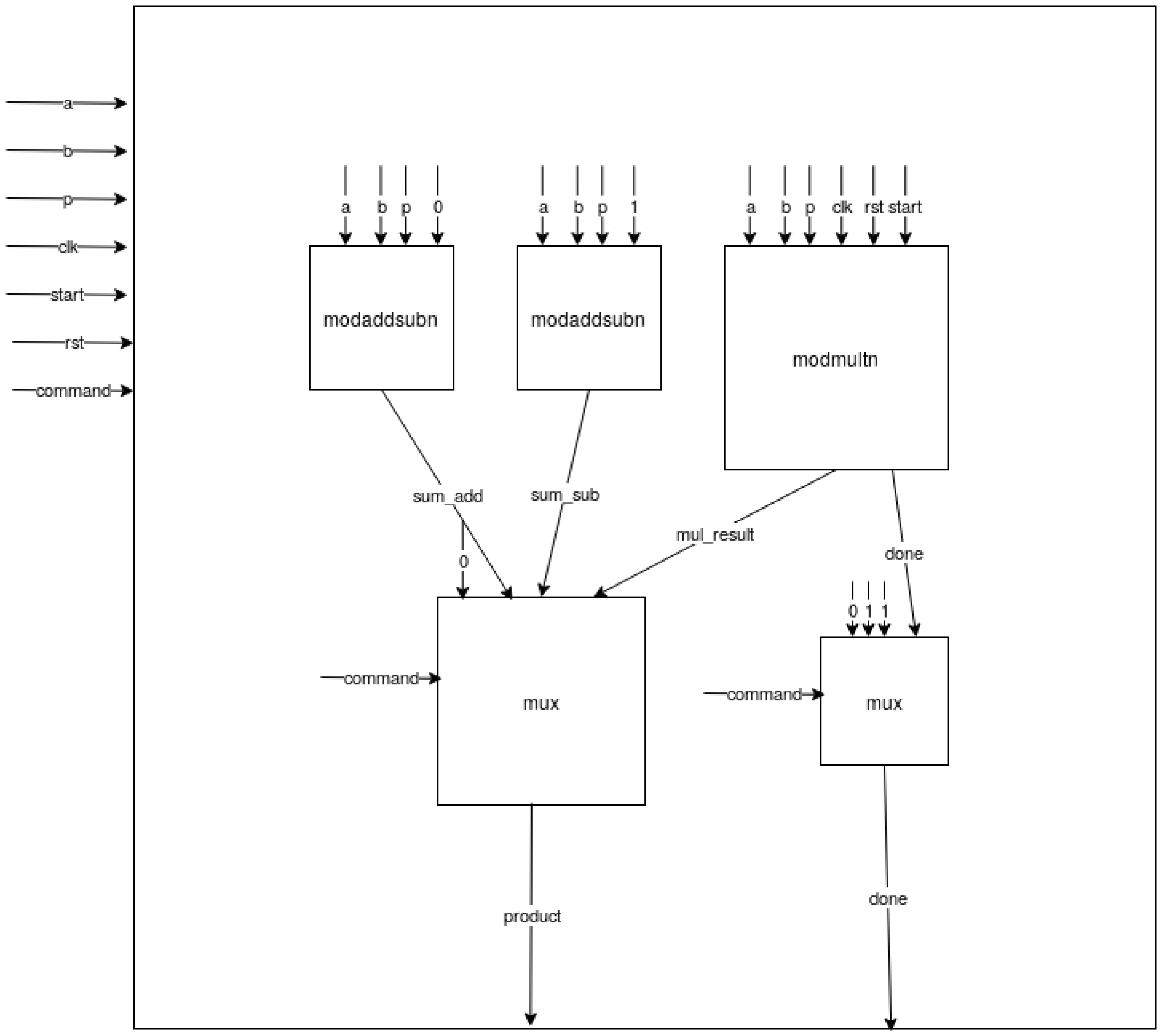
Appendix 1: Exercise 8.1



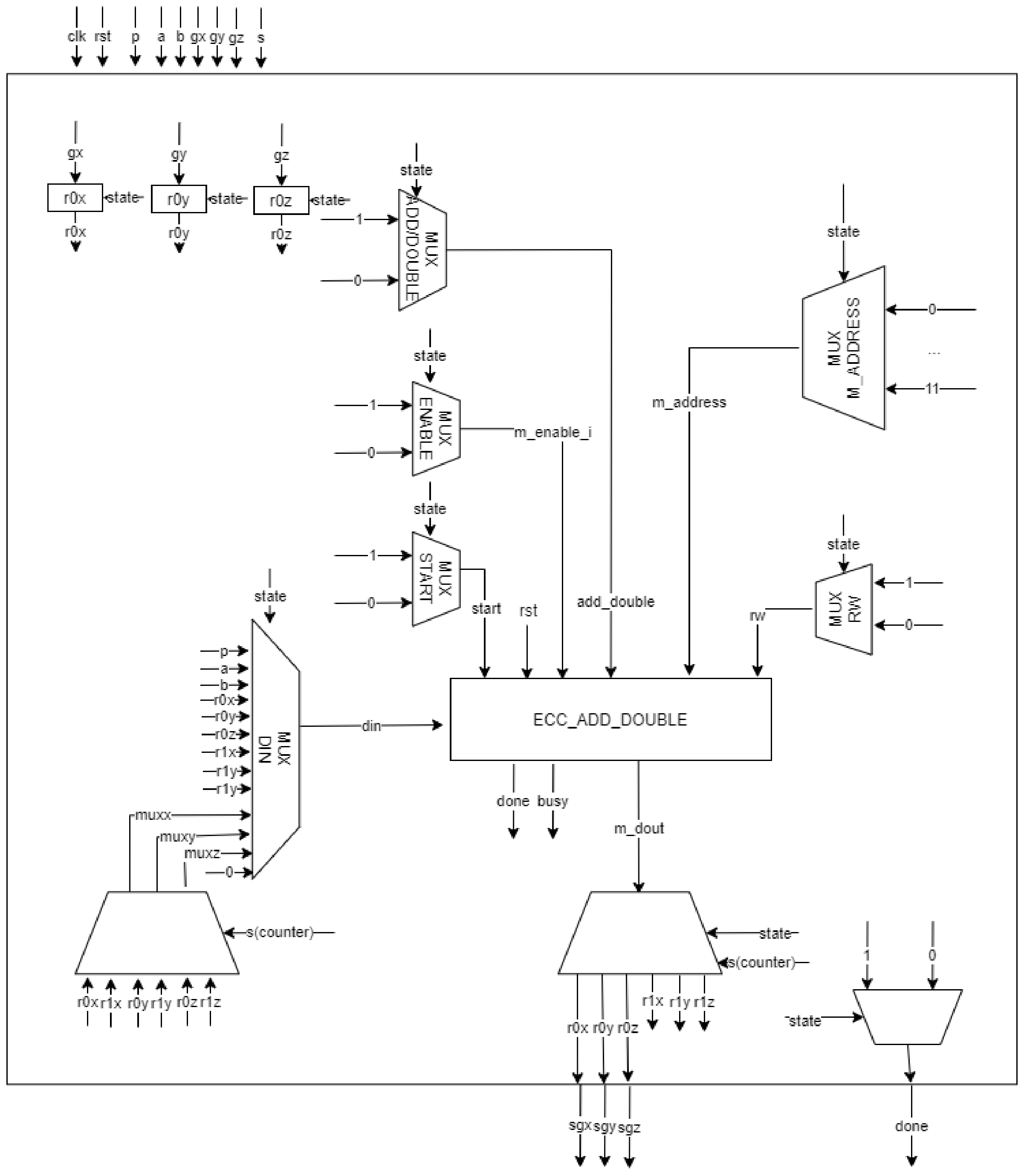
Appendix 2: Exercise 8.2



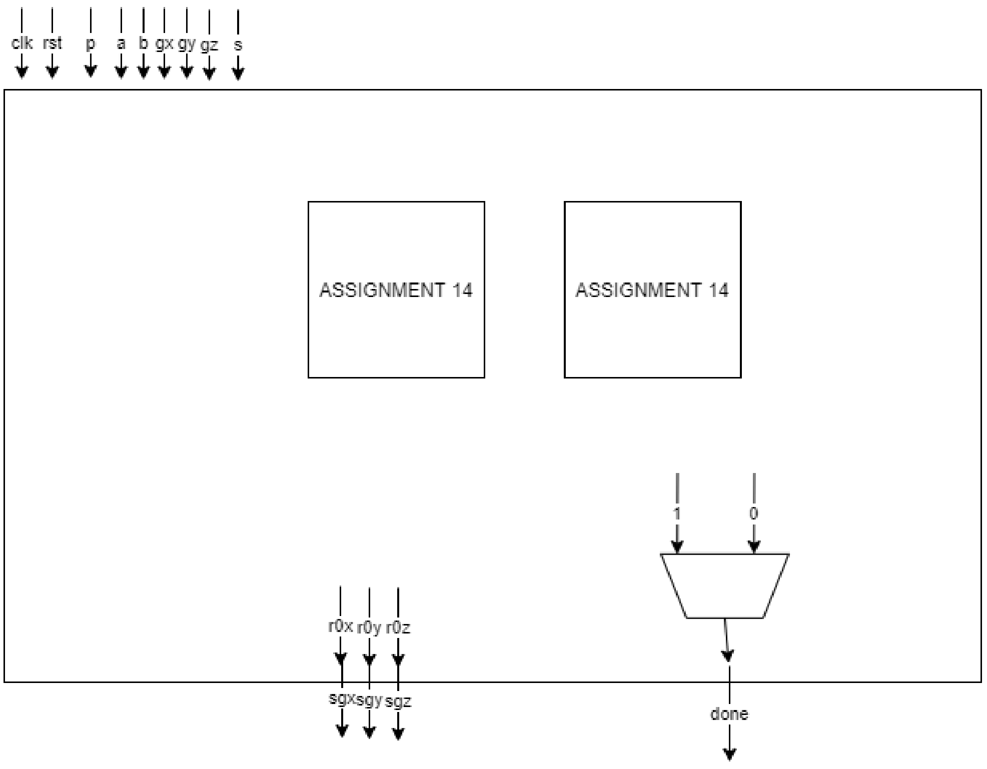
Appendix 3: Exercise 9



Appendix 4: Exercise 14



Appendix 5: Exercise 15


clk
rst
p
a
b
gx
gy
gz
s
ASSIGNMENT 14
ASSIGNMENT 14
1
0
r0x
r0y
r0z
sgx
sgy
sgz
done