Selected Topics on Hardware for Security
VHDL Assignment 2
ECC Hardware Design Project

Group: 1

Onno de Gouw
s1025613

Stefan Popa
s1027672

January 24, 2022

1 Introduction

Throughout the process of working on this project, for the exercises that we did not have to implement ourselves, we made sure that we understood the implementation provided properly. We did this by following the schemes that were given as well as diving into the code thoroughly. Moreover, we decided that it would be a nice idea to both simulate and manually verify - e.g. with online tools and calculators [2, 3] - whether certain parts of the simulation were correct for each exercise. In this way we made extra sure that no weird or unexpected behavior was present on our machines.

Due to the COVID-19 pandemic we were again forced to work online, and thus we had to make use of online conference tools and screen sharing to cooperate on the project. We were used to this by now and thus it did not affect our workflow. We also discussed possible uncertainties and explained them to each other.

For the drawings of our schemes, we decided to make use of the tool Draw.io [1]. This tool contained the necessary features and provided various options for the architecture designs that we were asked to create. It took some time to get used to the program and it was a tedious process, but we believe we ended up professional looking images.

2 Exercise explanations

2.1 Exercise 4: Design and simulate an n-bit modular adder architecture

For this exercise we designed the n-bit modular adder, by looking into the 4-bit modular adder given in Exercise 3 and by also looking at how the 4-bit adder was changed into an n-bit adder in exercise 1 & 2. This way, we were able to simulate the design, first with $n = 32$ bits and afterwards with $n = 64$ bits.
For both amounts of bits, we first determined different numbers ourselves and performed calculations manually using online tools that can do e.g. number conversions in decimal, binary and hexadecimal [3] and modular arithmetic [2]. This way we were able to verify whether our module did what it was supposed to do, in addition to what was given in the test bench. Finally, we decided to comment out the case of $n = 32$ bits in the test bench file. However, we tried and tested it and it is possible to uncomment the block of code to see it working.

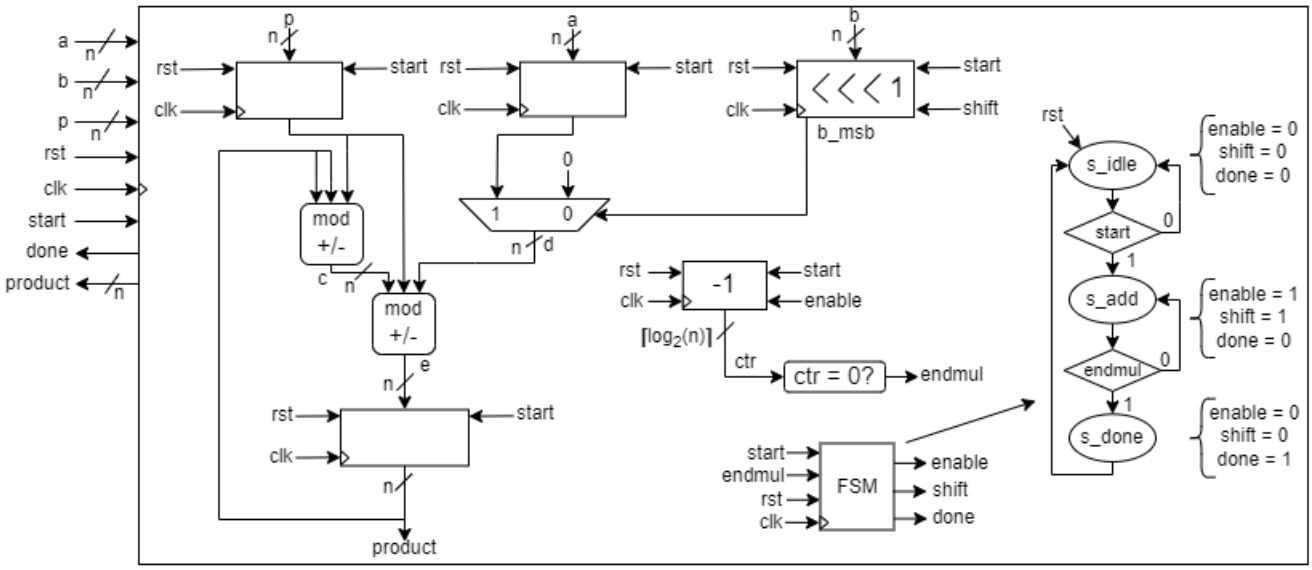
2.2 Exercise 7: Design and simulate an n-bit multiplier (by consecutive modular additions) and simulate the behavior

For this exercise, we based ourselves on Exercise 6 as well as the given schema in order to implement the n-bit multiplier. Many components and signals remained the same. However, for the counter implementation, things were slightly different in this case. Firstly, we needed to change the bit-size of the counter from 3 to n and we also had to add a check inside the counter which verifies whether `ctr = b - 1`. If this check holds, it sets `endmul` to 1 and then the state will move to `s_done`, as is reflected in the Finite State Machine. This is different from what we tried in the first place, namely creating a separate process for the counter check. Moreover, as given in the schema, we also had to add a 'b' register to our design in order to store the value with which to multiply.
We also noticed that for Exercise 7 the size of the counter is not $\lceil \log_2(n) \rceil$ but just `n`. This is a small mistake in the diagram that was given and which was also mentioned during the Q&A session held on January 5th. Finally, we are aware that the check that we implemented is actually determining if `ctr = b - 2`, and that is because of the one clock cycle delay caused by the Finite State Machine. There were other options possible in order to resolve this issue, like a conditional statement that does not need a process, but we were told via e-mail that performing the check in this way is also a good alternative.

2.3 Exercise 8: Design and simulate an n-bit maodular multiplier (through a left-to-right modular double-and-add algorithm)

We were very happy to see that our implementation for this exercise worked on first try, and that overall this exercise was not too different from the previous one.

The design schema:
First, we provide the schema that we designed for the implementation of this exercise:



The idea of this design is very similar to the one given in the previous exercise. The main change is that we now use the double-and-add algorithm which required us to use a different type of register for the 'b' value and also a different way to implement the counter functionality. The register shifts the 'b' value one bit to the left and takes out the most significant bit before the shift, while the counter now has a size of $\lceil \log_2(n) \rceil$, since its values now starts from 'n - 1' down to '0'. This is the way the double-and-add algorithm works on which we based our design. The pseudo-code for this algorithm was given in the assignment itself.

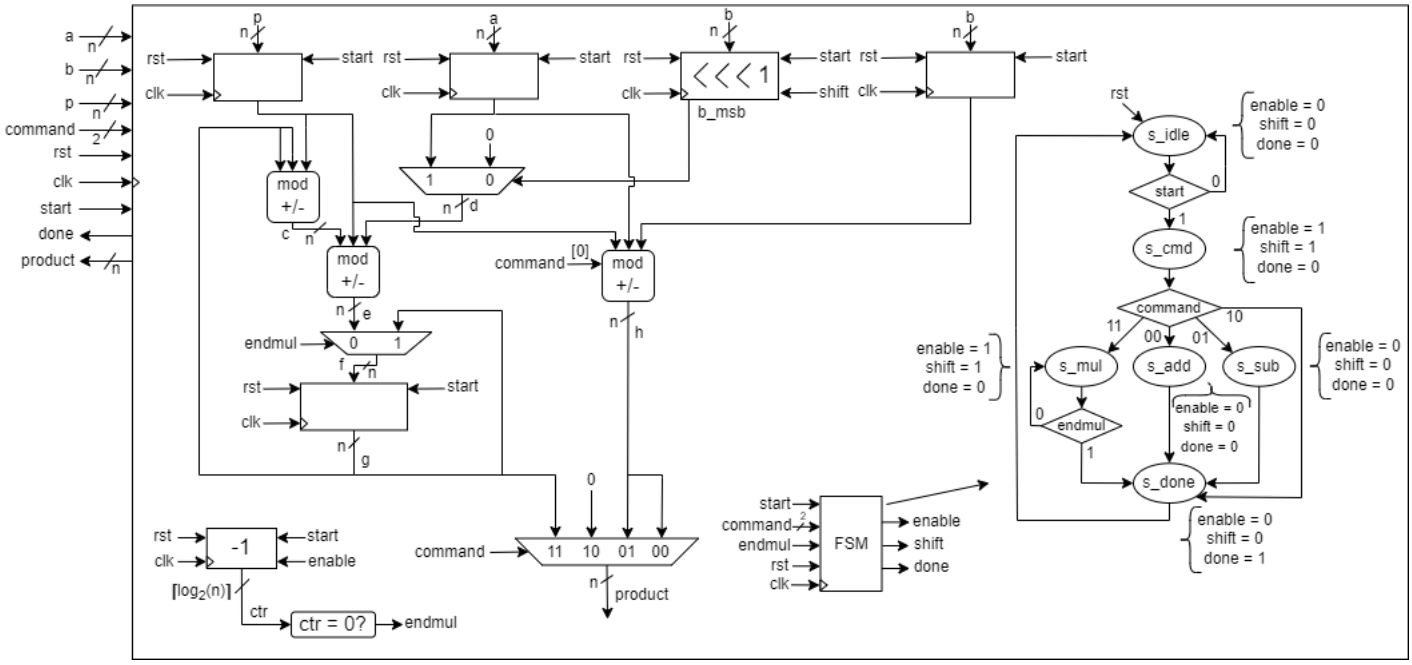
The code explanation:
When it comes to the code of the design that we just described, it is also quite similar to the implementation created for Exercise 7. The only differences are the counter and the 'b' register, designs that were explained in the previous paragraph. As was the case for Exercise 7, instead of stopping at `ctr = 0`, we stop at `ctr = 1`. This is again due to the one clock cycle delay caused by the Finite State Machine.

Extended Deadline Update: A mistake that we made was that our counter was underflowing after the last step, thus after counter becomes 0. We solved this by adding a conditional statement that checks whether counter is greater than 0, and if so, it subtracts 1 from the counter values.

2.4 Exercise 9: Draw, design and simulate a n-bit multiplier/addition/subtraction

This was one of the exercise we struggled with when working towards the first deadline, because we could not figure out why the test bench was raising a flag when the result that was computed was actually correct. After quite some debugging and testing we finally discovered the reason for this: the addition and subtraction module kept computing after the `endmul` flag was raised. This behavior takes place in the already given designs as well, e.g., Exercise 6 and 7, and thus we thought it is also an intended behavior. It turns out that it is not, and how we fixed this will be explained in the explanation of the schema.

The design schema:
First, we provide the schema that we designed for the implementation of this exercise:



The idea of this design is that we add another modular addition/subtraction component, such that it also becomes possible to do addition and subtraction. From here we looked into what other components were necessary to be able to implement this, and we came up with the idea of adding a multiplexer in order to decide which of the outputs to choose depending on the 'command' signal. We also had to add a second, normal register for 'b' that we could use for addition and subtraction. This is because we did not need the shifting register for these two operations. Secondly, we also

had to modify the final state machine accordingly. This was done by adding three more states, such that every possible operation is now represented in the FSM as well. Moreover, the state to which the program should move depends on the value of 'command', which follows from the new `s_cmd` state.

Extended Deadline Update: In order to solve the previously mentioned behavior causing issues with the test bench, we had to make use of a multiplexer that we placed right before the register storing the product value. This multiplexer takes as input the result coming from the addition and subtraction module if we are still computing the result, otherwise it keeps the already stored value.

While working on this new multiplexer, we had to decide what we would use as selector, and what made most sense was to use `endmul`. This is because the moment its value becomes 1, it means we need to stop taking new results from the addsub module. However, `endmul` was changing its values one step earlier (for the reasons mentioned in the previous exercises) and thus it did not work properly at first. Due to the underflow fix, we realized that we did not have to make the check one step earlier for `endmul`, and thus now we indeed change its value when `ctr` is 0.

As a last modifications, we now also had to slightly change the signals names to match the new design. In the end, the test bench error was fixed.

The code explanation:
When it comes to the code, we were happy to see that the test bench was already implemented, since this saved us much time. Similarly, not that much is different in the code compared to the previous two exercises. We had to add code for a second 'b' register, for an instance of the addition/subtraction module and for a multiplexer. Moreover, we had to add the new states to our previous Finite State Machine and modify its outputs as well. A first mistake we did while writing the code is that we forgot that it is not possible check for '11' in an if statement, but that we have to convert to the right type, e.g., `to_unsigned(3, 2)`, which is equal to the value of 3. Secondly, we again had to deal with the one clock cycle delay and thus we set 'enable' and 'shift' to 1 from the `s_cmd` state already. Finally, as mentioned in the introduction of this section already, we are aware of the fact that the test bench does say the implementation is not correct for multiplication. However, as can be seen from the simulation, the results that our implementation give are correct. Therefore, we do not know why the test bench says otherwise.

2.5 Exercise 11: Design and simulate a two ports memories system

The implementation that we created for this exercise is quite straightforward. We simply had to copy the code from Exercise 10 and modify it such that it now takes two addresses, instead of only a single one. Moreover, we needed to make sure that it is able to read two values in one clock cycle. The latter is done by adding a statement for `dout_b` under the `enable=1` conditional check.

2.6 Exercise 12: Design and simulate the arithmetic unit with the double memory, and instructions

For this exercise we first tried to understand the schema that was given, since there was not much information offered along with it. It took us some time, however we did manage to figure it out. In short, the `oper_o`, `oper_a` and `oper_b` registers store the addresses of the output value, the `a` value and the `b` value respectively. The fact that the value of an input is stored at the address with the same value in the RAM is simply a design decision that came with the assignment. Moreover, signals that start with `m_` as prefix are used during initialization and final state, while the others are used during internal computations. The rest of the signals and components were quite straightforward to understand. With this in mind, it did not take too long to implement to code for this design, following the schema. However, even after solving some of the bugs we noticed within our implementation, we still could not get it to work properly before the first deadline.

Extended Deadline Update: After solving the problem that we encountered as explained in Exercise 9, the design for this exercise worked as well without additional changes.

2.7 Exercise 13: Design and simulate a point addition/doubling architecture

This exercise did not have new things that we had to get a grasp of and thus we directly started working on implementing a design for it. We were glad that we had freedom with regards to how to implement it and we decided that the best course of action is to have the 77 steps hard coded into the finite state machine. This is especially due to the time we still had available when we started working on it. Moreover, it is also because we have experience with regard to how difficult it is to get a CPU fetcher right and working properly, because we had to work on something similar for a different course during our Bachelor.

The design schema:
Due to the size of the schema, you can find it included in the Appendix [4]. The FSM has a lot of states, but it is not that different from the FSM in Exercise 12. What has changed is the following:

- Since we do not have the command and the addresses given as inputs anymore, we had to add those as outputs of the FSM. These values change depending on the state we are in.
- We do not make a decision based on `reg_com[2]` anymore (due to the previously mentioned point) and we go directly to `s_load_p`. We also have a new state called `s_wait_p` to simulate the signal back to `s_idle` from the previous exercise.
- We now have a loading, computing, and writing state, as well as a conditional based on `a_done` for each of the 77 computations. This is because an addition or doubling in ECC consists of multiple operations in modular arithmetic (as seen in the algorithm given in the exercise), compared to the previous exercise where we were doing one operation per test. **Note:** Because we only realized later that we need to add a loading and writing state, we had to add those as a legend in the schema. Please assume that each loading state goes before each respective computation state, and each writing state goes after each respective computation state. Moreover, the transition to the writing state is also done based on the `a_done` conditional for each computation state.
- An `s_done` state has also been added.

Note: We realize that it is possible to use the `ecc_base` as a sub module and thus shorten our current FSM a lot, but we came to that conclusion quite far into our current implementation and we decided it is not worth it to scratch everything and start over, after working on this for a couple of days already.

The code explanation:
There is not much we can say about our code, since we follow the code from the previous exercise in addition to our schema, adjusted with the following changes:

- We removed the registers for the RAM addresses and for the command, since now we can directly use those from the output given by our FSM.
- We had to add all the states present in our FSM schema to the code, which took quite some time.

We managed to solve some bugs within our code, but there are still a few present. We will describe how we solved the former, and what we think causes the latter.

- We were only changing the RAM addresses when `start` was equal to 1, and thus later in the program a lot of the values were undefined. This was fixed by removing the said conditional.
- There were some delays in address assignments which again caused for undefined values, including for the prime number. This was fixed by removing the address registers completely and use the values directly from the FSM.
- There is currently a bug where after the computations the values are not stored in the output address. We believe that this is because we need one more state which allows for this value to be changed and stored in the RAM, before we move to the next operation. We tried this for a couple of states and we saw that it indeed solves this issue. However, due to the overhead it was not feasible to add so many more states right before the deadline.
- There is currently a bug where the arithmetic module stops earlier in the computation and also outputs a value of 0, causing wrong results. We are not really sure why that is, but it might be that we have some wrong assignments throughout our code for this exercise. The sub module should work as intended since it was tested in the previous exercises.

Note: Unfortunately we were not able to completely finish this exercise, also because the simulator was crashing due to having so many states and variables. We do however think that we are quite close to a working implementation.

Question: *can you think of an alternative way of implementing these formulas? Is there a way you could optimize the size, speed or energy consumption of your design?*

Another way of implementing these formulas could be by for example implementing the CPU fetcher, and thus you can save some memory (due to not having as many states stored) as well as increase the speed (the amount of operations you can do per clock cycle should increases, since you do not need to wait for a certain state change).
Moreover it would be possible to make use of parallelization in order to compute the different formulas, e.g. by means of using pipeline registers. These can be applied on formulas that do not depend on each other at the time of computing. This might improve the speed of our design, but perhaps take more space.

2.8 Exercise 14: Design and simulate a scalar multiplication through Montgomery Power ladder

Unfortunately, we again did not have enough time to start this exercise.

2.9 Exercise 15: Design and simulate an advanced unit

Unfortunately, we again did not have enough time to start this exercise.

3 Conclusion

In conclusion, we would like to mention that we really appreciate the fact that we were offered more time to work on this project. It allowed us to resolve many of the issues that occurred within our code and therefore we were also able to better understand how VHDL and the implementation works.

Unfortunately, we were still not able to completely finish every exercise, but this was mainly due to the time-consuming task of the schema we created in exercise 13 and the complexity of debugging the different kinds of bugs that occurred during the implementation processes. This does not take away from the fact that we are proud of what we were able to achieve thanks to the extended deadline.

Next to these comments, there are also some other takeaways from this project:

- While it can take time, we should not be afraid of hardware design and implementation! Most things are quite understandable and intuitive.
- When encountering a problem within the implementation of your design, debugging this problem can be complex and very time-consuming in VHDL since we had to use the simulation tools for this. Note that there might be some other way to debug, however we could not find these.
- We learned how to work with VHDL, which was a completely new experience. We can definitely say that we really enjoyed working on it, even though we did not finish the project entirely.

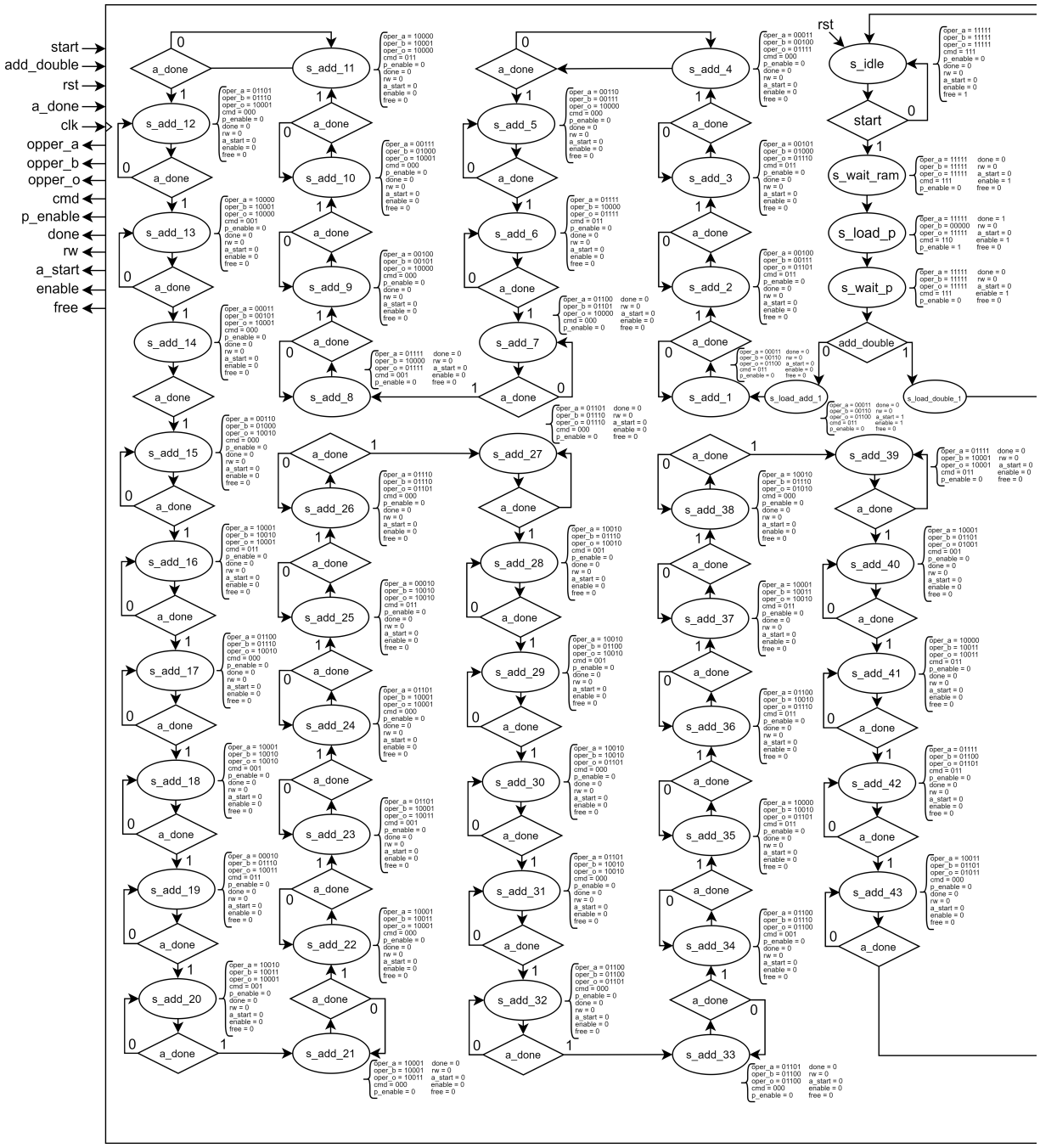
References

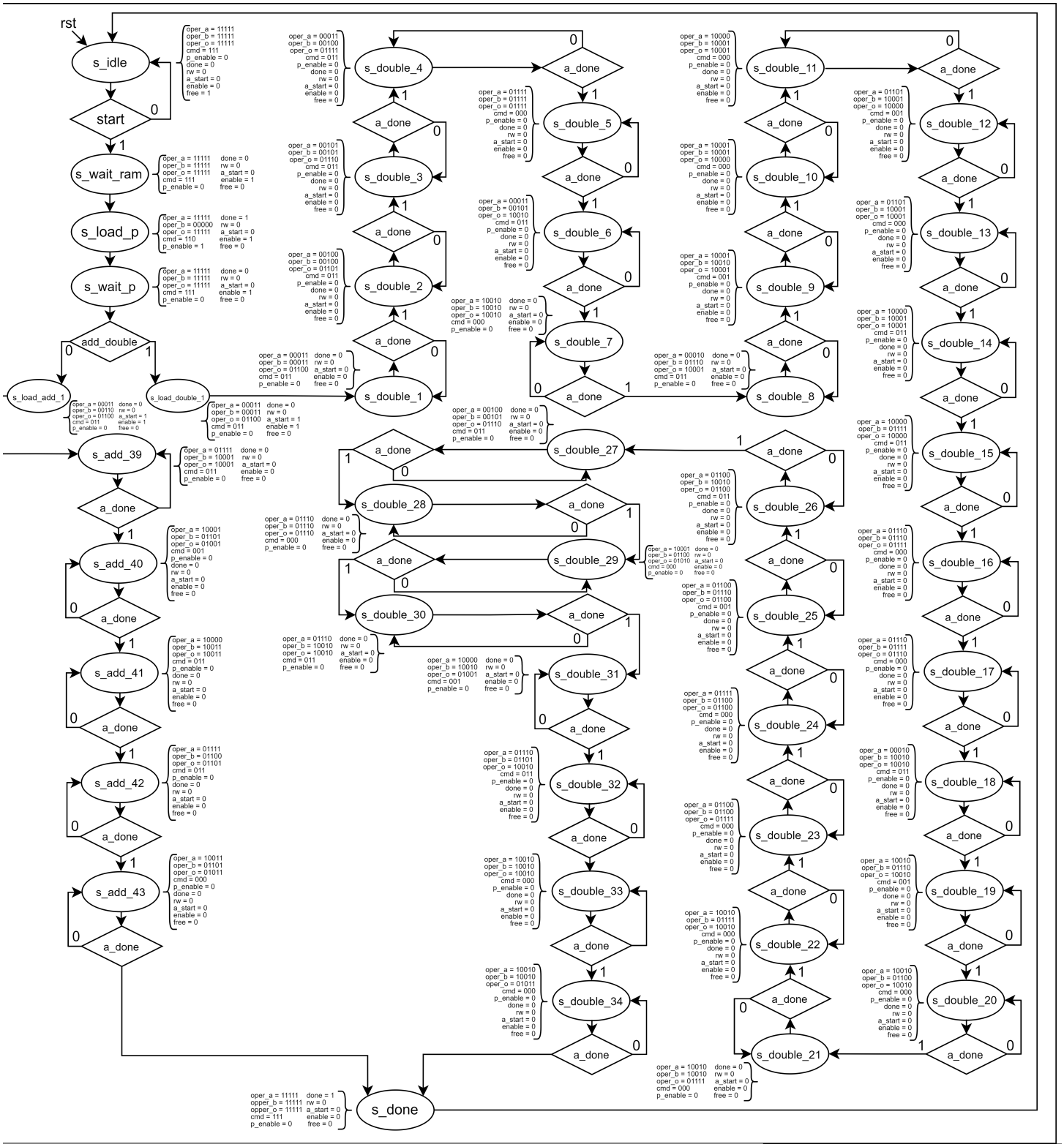
[1] Draw.io. Diagrams.net. URL: https://app.diagrams.net/ (visited on January 7, 2021).

[2] Wolfram Alpha LLC. Wolfram—alpha: computational intelligence. 2021. URL: https://www.wolframalpha.com/ (visited on January 6, 2021).

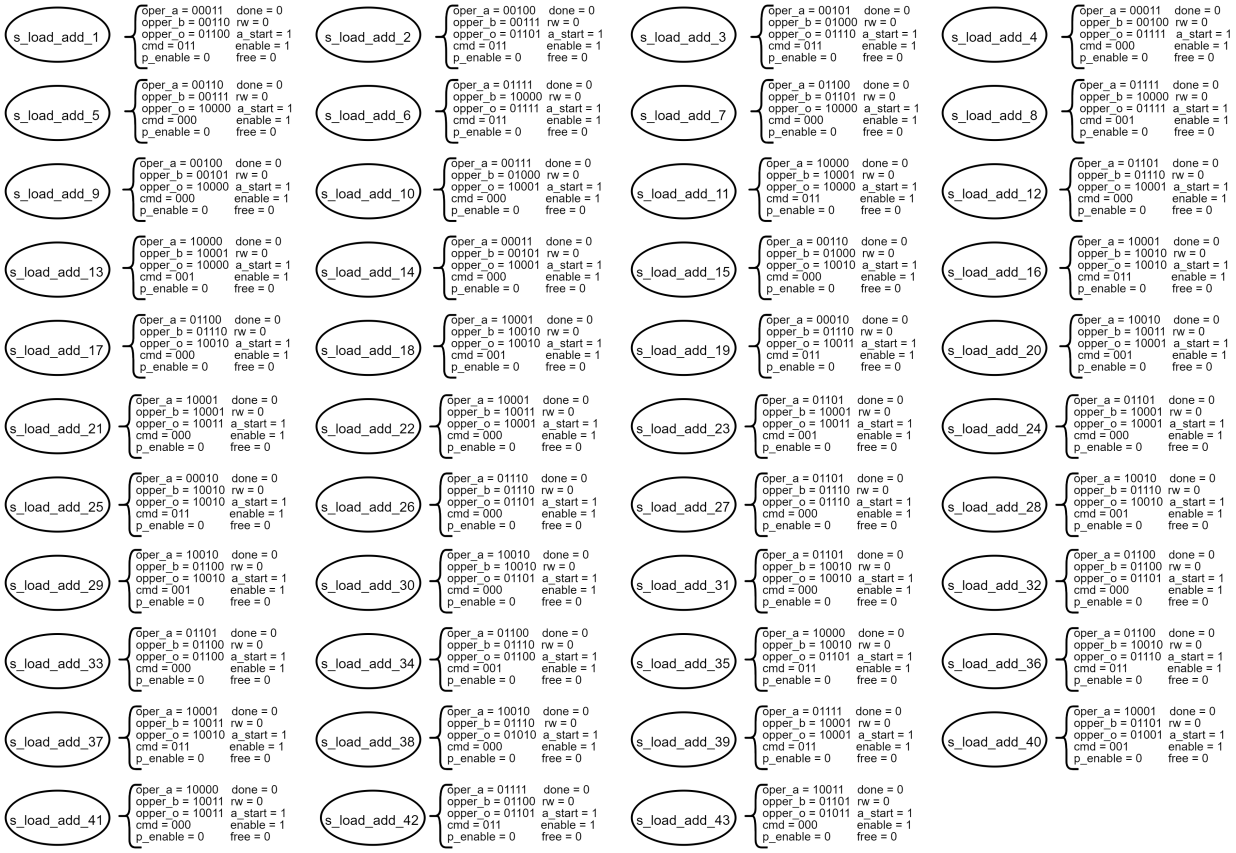
[3] Rapidtables. 2021. URL: https://www.rapidtables.com/ (visited on January 9, 2021).

4 Appendix

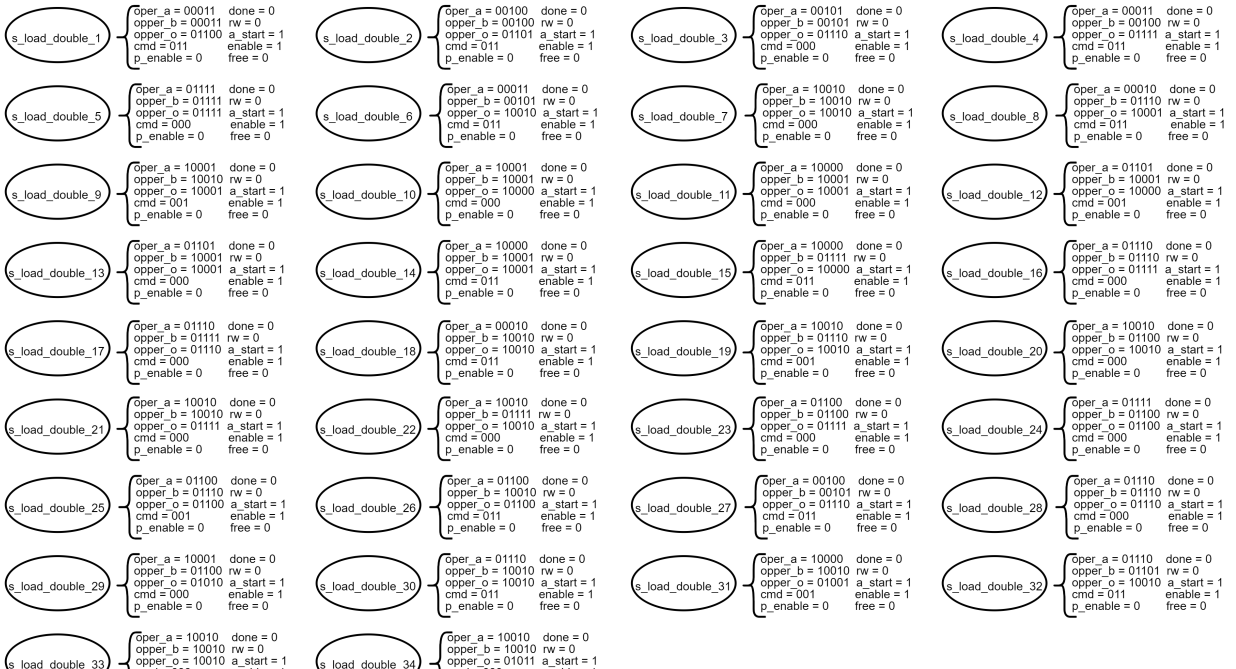

start
add_double
rst
a_done
clk
opper_a
opper_b
opper_o
cmd
p_enable
done
rw
a_start
enable
free
rst
s_idle
start
s_wait_ram
s_load_p
s_wait_p
add_double
s_load_add_1
s_load_double_1
s_add_1
s_add_2
s_add_3
s_add_4
s_add_5
s_add_6
s_add_7
s_add_8
s_add_9
s_add_10
s_add_11
s_add_12
s_add_13
s_add_14
s_add_15
s_add_16
s_add_17
s_add_18
s_add_19
s_add_20
s_add_21
s_add_22
s_add_23
s_add_24
s_add_25
s_add_26
s_add_27
s_add_28
s_add_29
s_add_30
s_add_31
s_add_32
s_add_33
s_add_34
s_add_35
s_add_36
s_add_37
s_add_38
s_add_39
s_add_40
s_add_41
s_add_42
s_add_43
a_done


rst
s_idle
start
s_wait_ram
s_load_p
s_wait_p
add_double
s_load_add_1
s_load_double_1
s_add_39
s_add_40
s_add_41
s_add_42
s_add_43
s_double_1
s_double_2
s_double_3
s_double_4
s_double_5
s_double_6
s_double_7
s_double_8
s_double_9
s_double_10
s_double_11
s_double_12
s_double_13
s_double_14
s_double_15
s_double_16
s_double_17
s_double_18
s_double_19
s_double_20
s_double_21
s_double_22
s_double_23
s_double_24
s_double_25
s_double_26
s_double_27
s_double_28
s_double_29
s_double_30
s_double_31
s_double_32
s_double_33
s_double_34
a_done
s_done

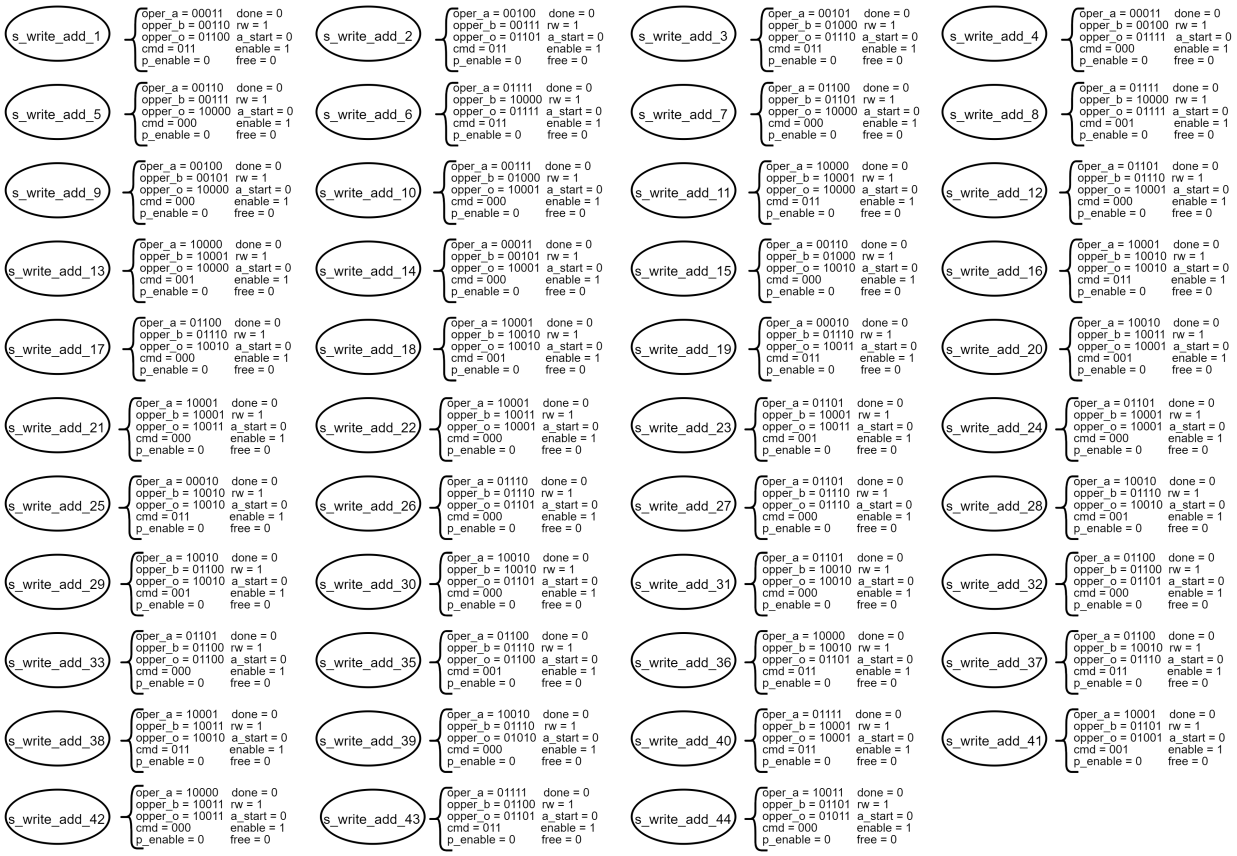
Loading States Addition:
(These go before each s_add_x state)



Loading States Doubling:
(These go before each s_double_x state)



Writing States Addition:
(These go after each s_add_x state)



Writing States Doubling:
(These go after each s_double_x state)