**Part 1**

1. The critical path of the circuit is from *f* to *n*, which consists of two combinational logics and one DFF.

tclk ≥ 4+2+5+3+2= 16 ns = shortest possible clock period.

Consequently, the fastest possible clock frequency = 1/16ns ≈ 62.5 MHz

If the hold time was 11ns instead, problems will occur at *j* and *n* at the two DFFs (possible a, b, and c), since now the hold time is greater than tc2q + tcomb (9 and 8 ns respectively). One way to fix this hold time violation is to add delays to the combinational logic.

1. This logic will not synthesize as a combinational circuit because it has inferred latches for the output signals. For each *sel* pattern, only one of the outputs is assigned *a*, while the other three are assumed to hold their old values, which requires memory units. To fix this, one thing we could do is to assign 0 to all the outputs prior to the case statement; this way, the output signals will be 0 when they are not supposed to take the input value.
2. In this logic, output *b* is assigned multiple values in the always\_comb blocks. Thus, its actual value will be unknown during simulation. To fix this problem, remove the assignment in one of the always\_comb blocks or simply combine them into one always\_comb block and modify *b* only once.

**Part 2**

1. **How your testbench works, and why you think it is a sufficient way to test the design. Do you have any ideas of how you could have designed a more robust testbench?**

Our testbench loads input values and signals from a file generated using a C program. Besides the basic multiply and addition of some large numbers, it also tests the systems when one or both inputs are negative. The testbench also tests when the reset signal is asserted in the middle of computation to make sure the registers are cleared and function properly after a reset signal is asserted.

One way to improve this testbench is to try to have more test cases(in our case we had 100 random tests). In addition to random input values, we could also generate random valid\_in signals.

1. **Create a test that causes the accumulator to overflow. Explain how you did it and what you observed. If you wanted the system that could detect when overflow happened, how would you do so?**

This was accomplished by adding consecutive 1023\*1023 until the MAC unit overflowed to a negative number. Once overflow happened, -1024\*1023 was added to cause an underflow. This test case is marked in the testbench. If we want to accommodate this, one way is to keep a flag, whenever the operands are of the same sign, we set it to remind us to check to see if the result’s sign bit is different from the operands. If they are, then a over/underflow has happened.

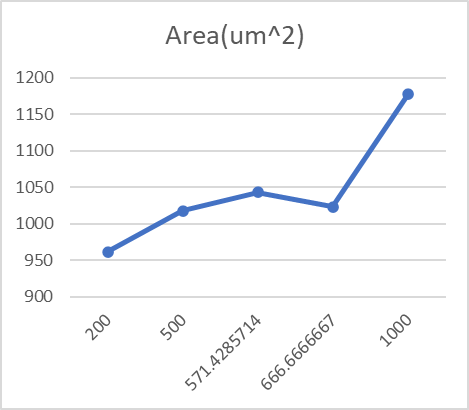
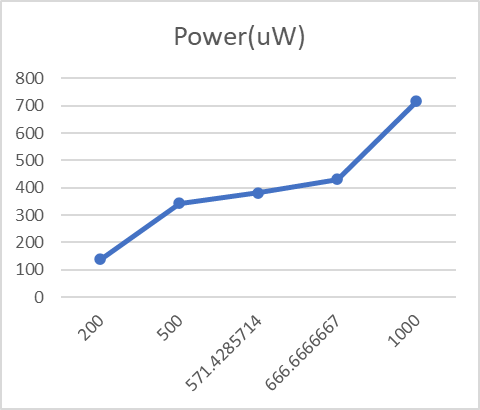
1. **Report the area, power, and critical path locations you determined for different clock frequencies. Make sure you include units (e.g., um2). Explain why you chose these frequencies. Make sure you found the maximum reachable frequency. When you report the critical path location, make sure you explain where in the logic the location is. (Don’t just copy/paste the location given in the report—explain it in a few words or a picture.)**

The frequencies are chosen to better observe the changes in the Area, Power, and Critical Path of the synthesized design. The initial tclk was set to 1ns, which turned out to violate the slack time requirement. Thus 1.5ns was tried and the slack time turned out to be 0.0, which just barely met the requirement and turned out to be the minimum possible clock period. The other three frequency values were observed to find a general correlation between clock frequency and the area, power, and critical path of the design.

There are tow critical paths observed: register *a* to *f* and register *b* to *f.* The paths should be the same since both of the input register outputs go through a multiplier and an adder. The critical path includes a/b register, which loads inputs a/b on each cycle, a multiplier, an adder, and finally the setup time of the output register f.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Frequency(MHz) | Clock T (ns) | Area (um2) | Power (uW) | Critical Path |
| 200 | 5 | 961.589997 | 117.6423 + 21.3307 | a\_reg -> f\_reg |
| 500 | 2 | 1017.449999 | 321.2531 + 21.0148 | a\_reg -> f\_reg |
| 571.43 | 1.75 | 1042.719998 | 359.5861 + 21.8070 | a\_reg -> f\_reg |
| 666.67 | 1.5 | 1023.301993 | 408.9877 + 21.8447 | b\_reg -> f\_reg |
| 1000 (violated) | 1 | 1177.847986 | 687.6696 + 27.4026 | b\_reg -> f\_reg |

1. **Make graphs that show the relationships you found between clock frequency and both area and power. Explain the trends that you observed and explain why they occur. (Make graphs with clock frequency on the x-axis and area or power on the y-axis.)**



With an exception at f=666.67MHz, in general both area and power increases as the clock frequency increases. This is expected since with higher frequencies, more computation jobs are done per second and thus the energy consumption increases. To accommodate higher frequencies, the synthesizer also needs to utilize more modules/logics to meet the short clock period.

1. **For the design you found with the maximum clock frequency, how much energy would your system consume if your system were to process a sequence of 50 cycles of input values? Assume you have to wait until the final output comes out of the system.**

To process 50 cycles of input values, the system consumes 50(408.9877 + 21.8447) ≈ 21541.62 uW = 21.542 mW

1. **Would the energy you computed in question e. change if you change the clock frequency? Why or why not?**

The energy consumption will increase as the clock increases or decrease as the clock frequency decreases since the power consumption as shown before is approximately linearly correlated with the clock frequency.

1. **The directions above told you to include reset signals on the registers. Is it necessary for you to do so for the system to work correctly? For all registers? Explain.**

In this case it is necessary to have a reset signal to initialize the registers to avoid unknown values. If the registers are not reset/initialized, for example, *f <= f + a\*b* will just be unknown all the time since *f* is never initialized: ‘X’ plus any value is still ‘X’.