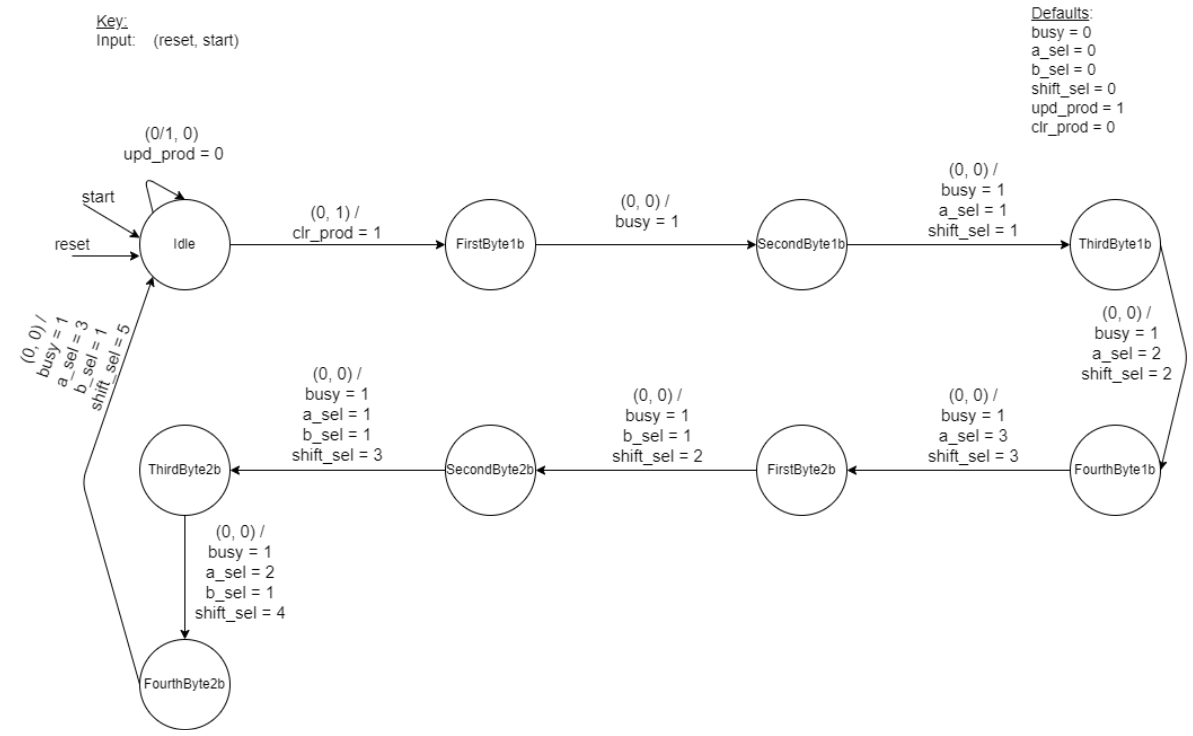
Simulation 2
Dry Part

341312304	שרה גריפית
207223066	טל שמיר

2.1: Diag

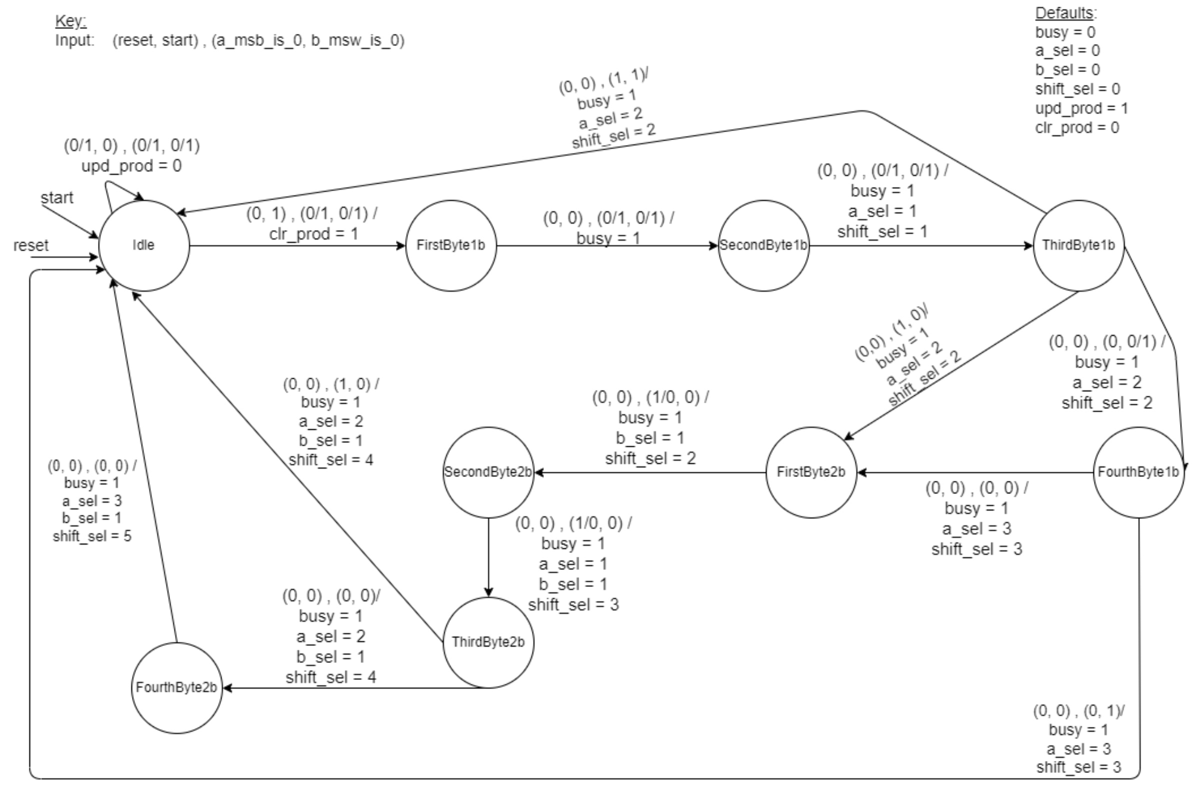
The following is a diagram of the FSM we will implement:



Each multiplication takes 8 clock cycles, from the moment busy goes up to 1, until it goes down to 0.

Simulation 2
Dry Part

2.2:



The fastest calculation of the multiple will be given when both a_msb_is_0 and b_msw_is_0 are 1. In this case, the calculations after the state ThirdByte1b are irrelevant, and so the calculations will be completed after 3 clock cycles, from the moment busy goes up until it goes back down to 0.

If a_msb_is_0 is 1 while b_msw_is_0 is 0, the calculation will continue to the state FirstByte2b after ThirdByte1b, skipping the FourthByte1b. Likewise, the calculation will skip over the state FourthByte2b and instead return to Idle after ThirdByte2b. The calculation in this case will take 6 clock cycles, from the moment busy goes up until it goes back down to 0.

If a_msb_is_0 is 0 while b_msw_is_0 is 1, the calculation will continue to Idle after the state ourthByte1b. The calculation in this case will take 4 clock cycles, from the moment busy goes up until it goes back down to 0.

In every other instance (when both a_msb_is_0 and b_msw_is_0 are 0), the calculation time will remain the same as it was in the previous question, 8 clock cycles, from the moment busy goes up until it goes back down to 0.

2.3: Algorithm for Implementation of Multiplier 8Nx8N

Let ‘a’ represent the first number given and let ‘b’ represent the second number given.

The program will begin by checking that the given N is not equal to 0. When it is, we will have received an illegal number, and will therefore return 0 as the result.

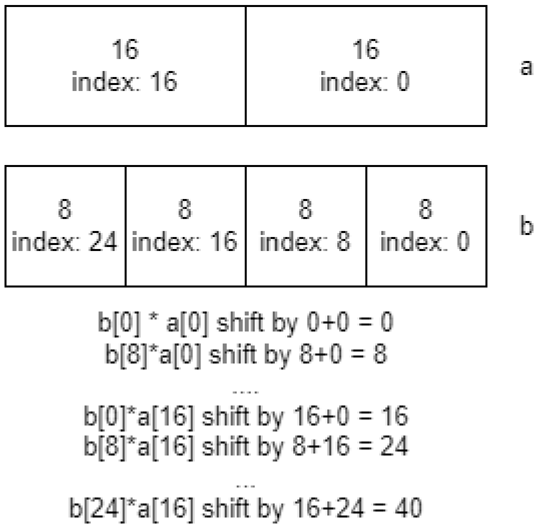
Simulation 2
Dry Part

Otherwise, we will break the given numbers, both of which are of size 8N > 0, into pieces that will allow us to use the given multiplier (16x8). We will follow the steps below in order to calculate the desired product:

- Define two variables that act as bit counters in the numbers given, one for each of them. Initiate both bit counters (indices) to 0.
- Define a variable that represents the product to be returned and is initialized to the value of 0.
- While the index of a is less than its size (8N):
    - While the index of b is less than its size (8N):
        - Multiply 8 bits of b by 16 bits of a using the 16x8 multiplier provided
        - Save the result in a local temporary variable
        - Shift the result left by (a's index + b's index) bits
        - Add the shifted result to the product
        - Increase b's index by 8
    - Increase a's index by 16
- At the end of the loop, return the calculated product.

Further explanation regarding the shift:

As seen in the previous questions, shifting the result of the multiplier is necessary in order to receive the correct final product. This shift must happen similarly as it does in long multiplication and in the previous questions. In our case, the shift done on the first set of 8 bits from b is of the size 0, 8, 16, … as seen in long multiplication in our daily lives. Thereafter, the initial shift (on the product of the first set of 8 bits from b with the next 16 bits from a) is 16 bits more than the previous initial shift. In our case, throughout the entire calculation, this value is equal to the sum of the two indices of a and b. For example:


16
index: 16
16
index: 0
a
8
index: 24
8
index: 16
8
index: 8
8
index: 0
b
b[0] * a[0] shift by 0+0 = 0
b[8]*a[0] shift by 8+0 = 8
....
b[0]*a[16] shift by 16+0 = 16
b[8]*a[16] shift by 8+16 = 24
...
b[24]*a[16] shift by 16+24 = 40

Time Complexity of the Algorithm:

The calculations below rely on the assumption that basic Assembly operations take place in time complexity of O(1). This includes addition, multiplication, and shift.

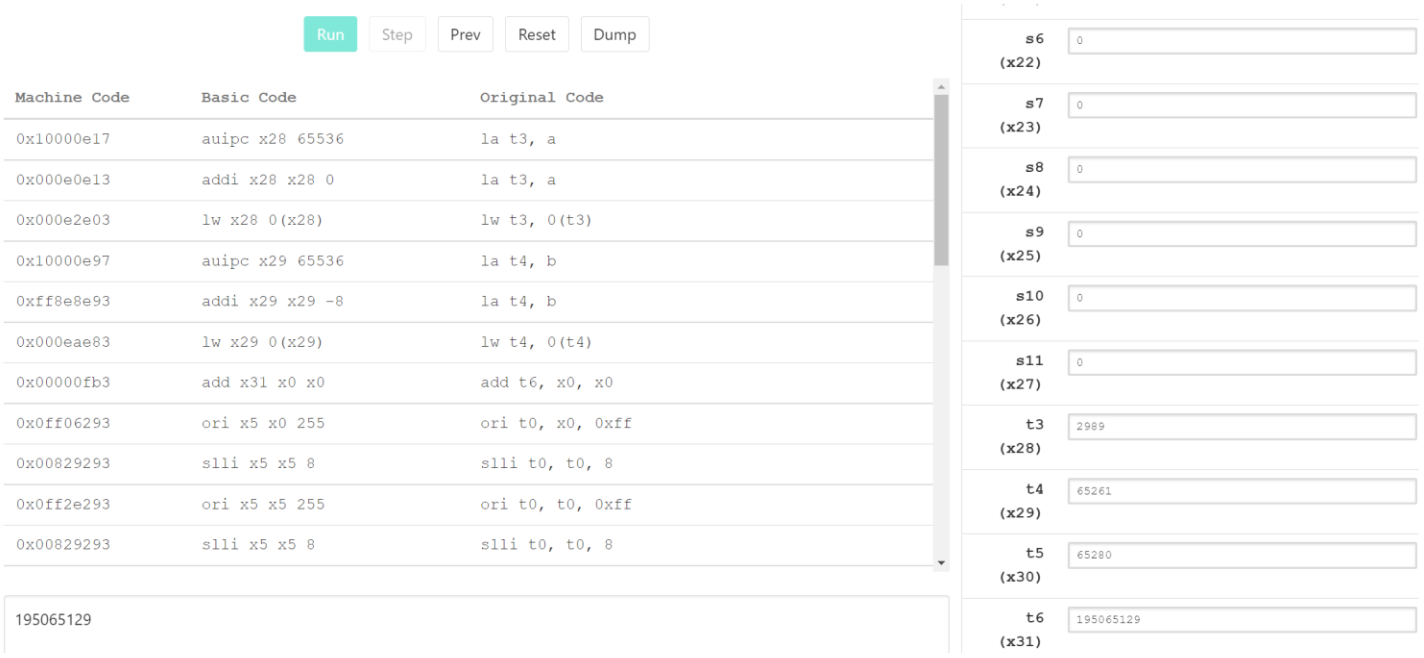
Simulation 2
Dry Part

- Defining and initializing several variables to be used throughout the calculation: O(1)
- External loop that runs on the length of the number a in jumps of 16:
    - Internal loop that runs on the length of the number b in jumps of 8:
        - Multiplying each of the relevant values: O(1)
        - Shifting the result: O(1)
        - Adding the shifted result to the product: O(1)
        - Increasing the necessary index by a constant value of 8: O(1)
        - Due to the fact that the loop runs on the length of the number in jumps of 8, the total time complexity of this loop is: $O(\log_8 N * 1) = O(\log_8 N)$
    - Increasing the necessary index by a constant value of 16: O(1)
    - The external loop runs on the length of a in jumps of 16. Therefore, the total time complexity of the nested loops is: $O(\log_8 N * \log_{16} N) = O(\log^2 N)$
- Returning the product received: O(1)

Therefore, the total time complexity of the algorithm presented is: $O(1) + O(\log^2 N) = O(\log^2 N)$

2.4: Implementation

The output of the multiplication was as follows:



Machine Code	Basic Code	Original Code
0x10000e17	auipc x28 65536	la t3, a
0x000e0e13	addi x28 x28 0	la t3, a
0x000e2e03	lw x28 0(x28)	lw t3, 0(t3)
0x10000e97	auipc x29 65536	la t4, b
0xff8e8e93	addi x29 x29 -8	la t4, b
0x000eae83	lw x29 0(x29)	lw t4, 0(t4)
0x00000fb3	add x31 x0 x0	add t6, x0, x0
0x0ff06293	ori x5 x0 255	ori t0, x0, 0xff
0x00829293	slli x5 x5 8	slli t0, t0, 8
0x0ff2e293	ori x5 x5 255	ori t0, t0, 0xff
0x00829293	slli x5 x5 8	slli t0, t0, 8

```
195065129
```

s6 (x22)	0
s7 (x23)	0
s8 (x24)	0
s9 (x25)	0
s10 (x26)	0
s11 (x27)	0
t3 (x28)	2989
t4 (x29)	65261
t5 (x30)	65280
t6 (x31)	195065129

The calculation takes 23 clock cycles:

The loop runs twice throughout the calculation because the input is 16x16. Each run of the loop is 10 operations, and as defined, each operation takes 1 clock cycle. Additionally, there are three variable definitions before the loop, each taking 1 clock cycle. In total, the calculation takes 23 clock cycles.

2.5: Fast Implementation

Simulation 2
Dry Part

In order to create an implementation that deals with scenarios in which the first byte of a and/or b is equal to 0, it is necessary to isolate and check those bytes first.

In our implementation, we first check the first byte of a. If the first byte of a is equal to 0, we then swap the values of a and b, so that the loop that calculates the product will run only once. This is because there is no need to divide b (which is now equal to a) into two separate parts due to the first byte being 0. Therefore, the calculation ends after a single run on the loop. However, if we find that the first byte of b is also equal to 0, we do not continue to the loop we instead preform a single calculation that returns the product necessary.

If a is not equal to 0, we continue with the regular loop, and check before repetition whether the first byte of b is equal to 0. In this case, as before, if this is true, the program ends.

```
        ori t5, x0, 0xff # shift variable

        srli a5, t3, 8 # isolates the first byte of a
        bne a5, x0, Start # if the first byte of a is not equal to 0, we'll
continue the regular loop
        # Else, swap the values of a and b, so that b is left as 16 in the
multiplication
        add t2, t4, x0
        add t4, t3, x0
        add t3, t2, x0
        srli a6, t3, 8 # isolates the first byte of b
        bne a6, x0, Start
        and a4, t5, t4
        mul t6, a4, t3 # multiply 8 bits from b with the 16 from a
        and t6, t6, t0
        j Exit

Start:
        add t2, x0, x0 # index of b
        addi t1, x0, 16 # size of numbers given
Loop1:
        sll t5, t5, t2 # shift the shift variable over
        and a4, t5, t4 # isolates 8 bits from b
        srl a4, a4, t2 # 8 bits from b
        mul a4, a4, t3 # multiply 8 bits from b with the 16 from a
        and a4, a4, t0
        add a3, x0, t2 # add the two indices together
        sll a4, a4, a3 # shift the product by the sum of the indices
        add t6, t6, a4 # add the product to the output
        srli a6, t4, 8 # isolates the first byte of b
        beq a6, x0, Exit # if the first byte of b is equal to 0, skip to the end
and leave the loop
        addi t2, t2, 8 # move the index on b by 8
```

Simulation 2
Dry Part

```
        blt t2, t1, Loop1
Exit:
```

Prior to the changes above, the calculation took 23 clock cycles. There are a number of cases possible with the new code, and in each case the clock cycles necessary to complete the calculation are different:

1. Worst case: neither the first byte of a nor the first byte of b are 0:
   5 operations outside of the loop, and two runs of the loops where each run is 12 operations
   Total: 29 clock cycles
2. The first byte of a is equal to 0, and the first byte of b is not equal to 0:
   10 operations outside of the loop, and one run of the loop containing 10 operations
   Total: 20 clock cycles
3. The first byte of b is equal to 0, and the first byte of a is not equal to 0:
   5 operations outside of the loop, and one run of the loop containing 10 operations
   Total: 15 clock cycles
4. Both the first byte of b and the first byte of a are equal to 0: 12 operations in total
   Total: 12 clock cycles

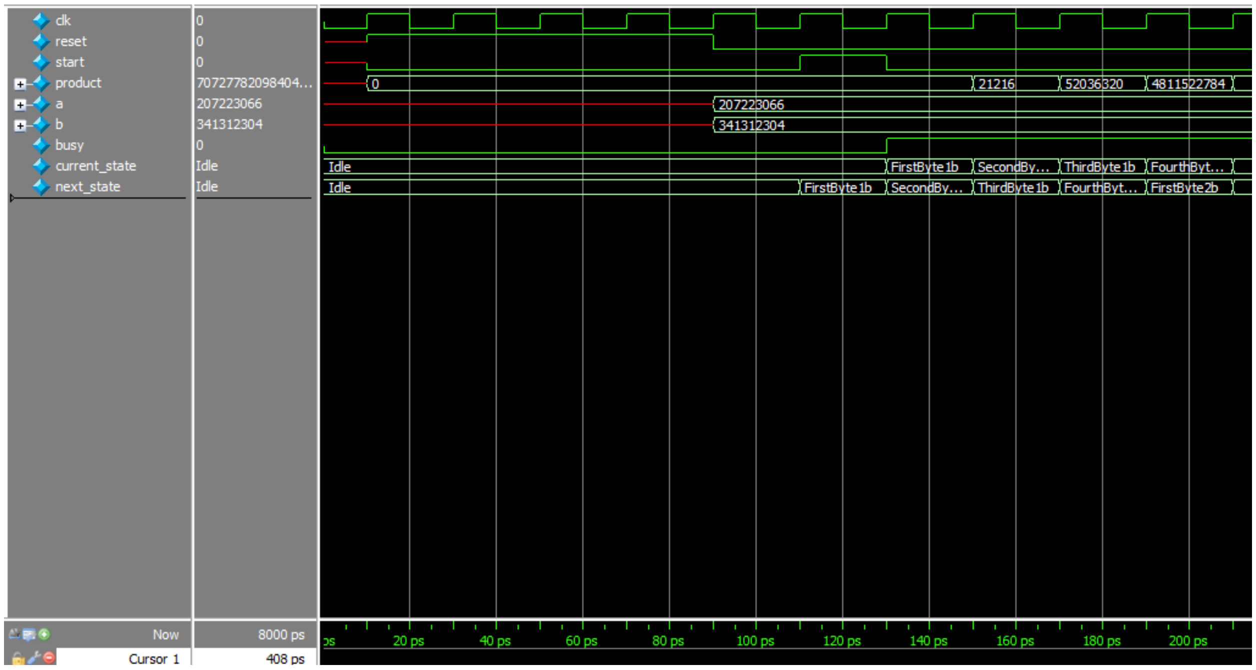
In general, this implementation is more efficient because it saves time in three of the four possible scenarios by reducing the number of operations the program executes. However, in the worst case scenario, this implementation added 6 clock cycles to the run time. Therefore, this implementation may be more efficient when given the correct set of data but may also be significantly less efficient if this is not the case.

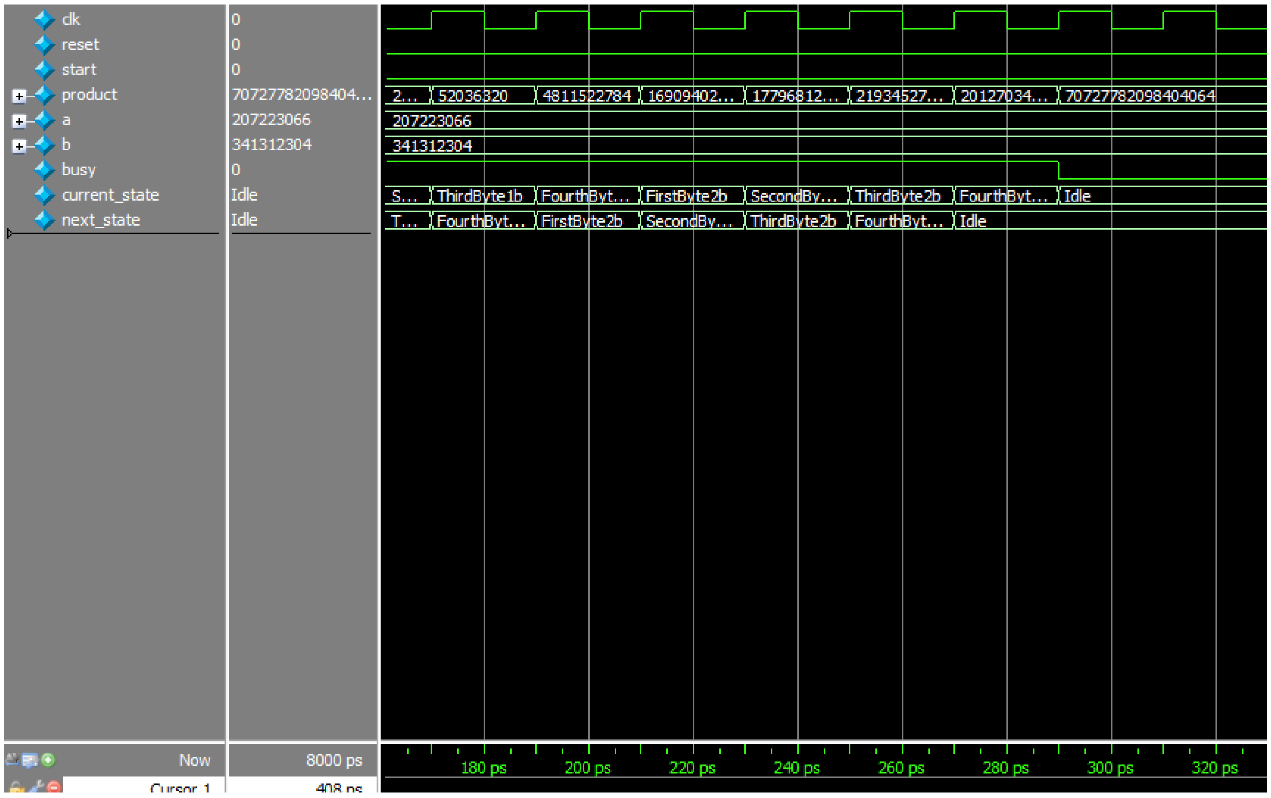
3.4: Simulation of Multiplier 32x32

The following are the wave lines of the simulation and test bench created:

Simulation 2

Dry Part





The clock starts at the value of 0 and changes its value every 10 units of time. Once the clock changes to 1 for the first time, the values of product, start, and reset are initialized: product and start are equal to 0, and reset is equal to 1 as requested. This lasts for 4 cycles of the clock.

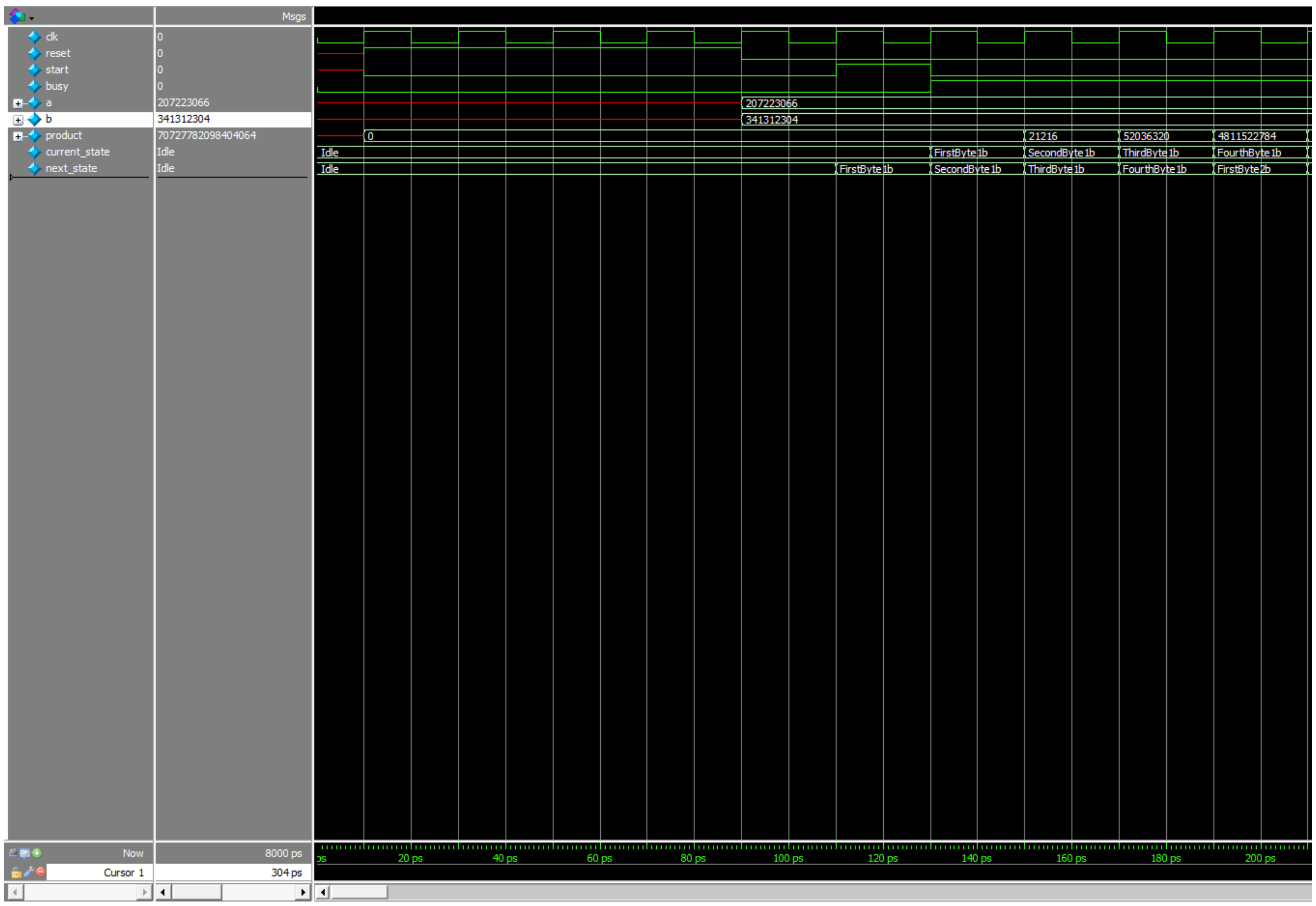
The values of a and b are initialized after these 4 cycles, and thereafter, start is changed to 1 for one cycle, starting the multiplication process. One cycle after start changes, the value of busy is changed to

Simulation 2
Dry Part

1, in accordance with the requirements of the exercise. As seen in the diagrams, with each clock cycle, the next_state and current_state change in the order described in the diagram from question 2.1. Finally, after 8 clock cycles (from the moment busy goes up until it goes back down to 0), busy goes down to the value of 0, the current and next states return to Idle, and the product from the multiplication of our ID numbers is received.

As shown, the result is: 341312304 x 207223066 = 70727782098404064

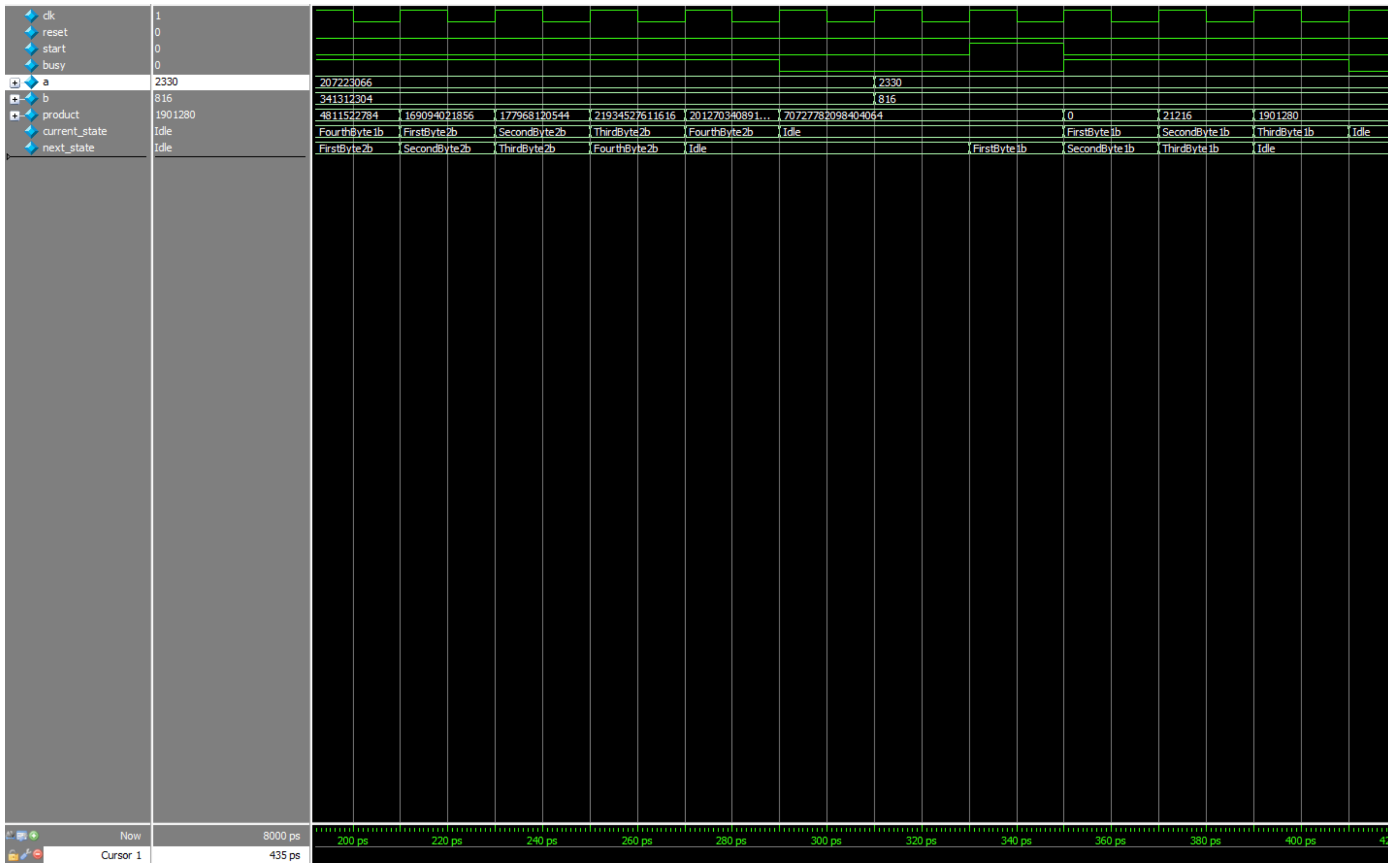
3.7: Simulation of Fast Multiplier 32x32



```
dk             0
reset          0
start          0
busy           0
a              207223066
b              341312304
product        70727782098404064
current_state  Idle
next_state     Idle
Now            8000 ps
Cursor 1       304 ps
```

```
a              207223066
b              341312304
product        0                    21216          52036320       4811522784
current_state  Idle                 FirstByte1b    SecondByte1b   ThirdByte1b    FourthByte1b
next_state     Idle   FirstByte1b   SecondByte1b   ThirdByte1b    FourthByte1b   FirstByte2b
```

Simulation 2
Dry Part



```
clk            1
reset          0
start          0
busy           0
a              2330
b              816
product        1901280
current_state  Idle
next_state     Idle
Now            8000 ps
Cursor 1       435 ps

a:             207223066 | 2330
b:             341312304 | 816
product:       4811522784 | 169094021856 | 177968120544 | 21934527611616 | 201270340891... | 70727782098404064 | 0 | 21216 | 1901280
current_state: FourthByte1b | FirstByte2b | SecondByte2b | ThirdByte2b | FourthByte2b | Idle | FirstByte1b | SecondByte1b | ThirdByte1b | Idle
next_state:    FirstByte2b | SecondByte2b | ThirdByte2b | FourthByte2b | Idle | FirstByte1b | SecondByte1b | ThirdByte1b | Idle
```

The clock starts at the value of 0 and changes its value every 10 units of time. Once the clock changes to 1 for the first time, the values of product, start, and reset are initialized: product and start are equal to 0, and reset is equal to 1 as requested. This lasts for 4 cycles of the clock.

After these four cycles, the values of a and b are initialized to our ID numbers. Start is then changed to 1 for one clock cycle, after which busy becomes 1 and the multiplication process begins. The current and next states change throughout the calculation in accordance with the state diagram as shown in part 2.2. The calculation process in this case is identical to the process shown in the regular multiplier from the previous question. After 8 clock cycles, measured from the moment busy is equal to 1 until busy is equal to 0, the product of the multiplication is received. As in the previous simulation, the result is 341312304 x 207223066 = 70727782098404064.

After one clock cycle, a and b are initialized to new values: our ID numbers, such that the first two bytes of each number in their binary representation are 0. After one clock cycle, start is changed to the value of 1 for one clock cycle. Thereafter, busy changes to 1, and after 3 clock cycles, returns to the value of 0, and the product of the multiplication is received. The next and current states are as shown in the state diagram from question 2.2. The time this calculation takes is in accordance with our predictions given the ability to skip certain states as described in question 2.2. This is indeed significantly faster than if the calculation were done with the original implementation. The result received from this calculation is: 2330 x 816 = 1901280.