Dual Fixed-Point Calculator

GABRIEL RAMIREZ | AUSTIN NOLEN

ECE 5736

SUMMER 2021

DFX Overview

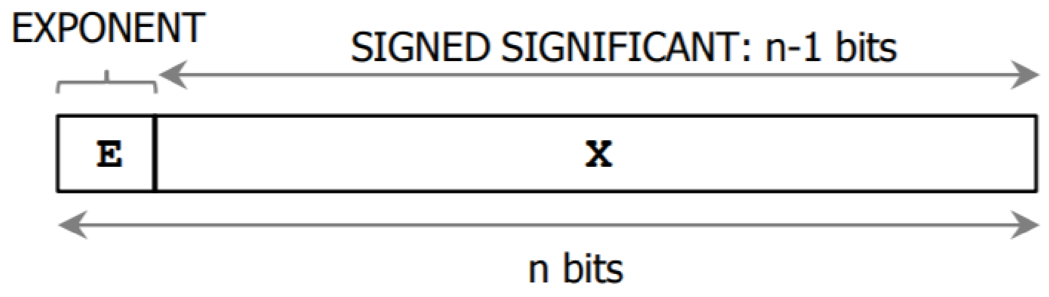
Dual fixed point is an alternative way of representing a fixed-point number

It utilizes two scalings n0 and n1 that greatly increase the dynamic range compared to FX

Uses less resources compare to floating point

Written in format [n p0 p1] where

p0 > p1



Project Overview

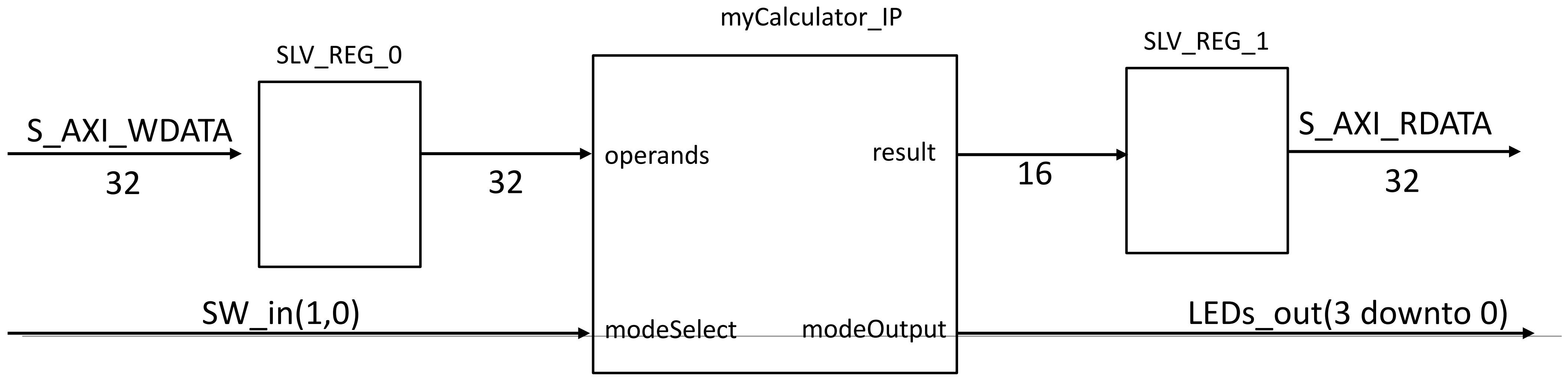
Dual Fixed-point calculator with Adder, subtractor, multiplier, divider

AXI Lite peripheral built around calculator and programmed onto Zybo board

Using selection switches on Zybo board to select between calculator functions

Input two 16-bit DFX operands in [16 8 4] format and receive 16-bit DFX output result based on selected calculator function

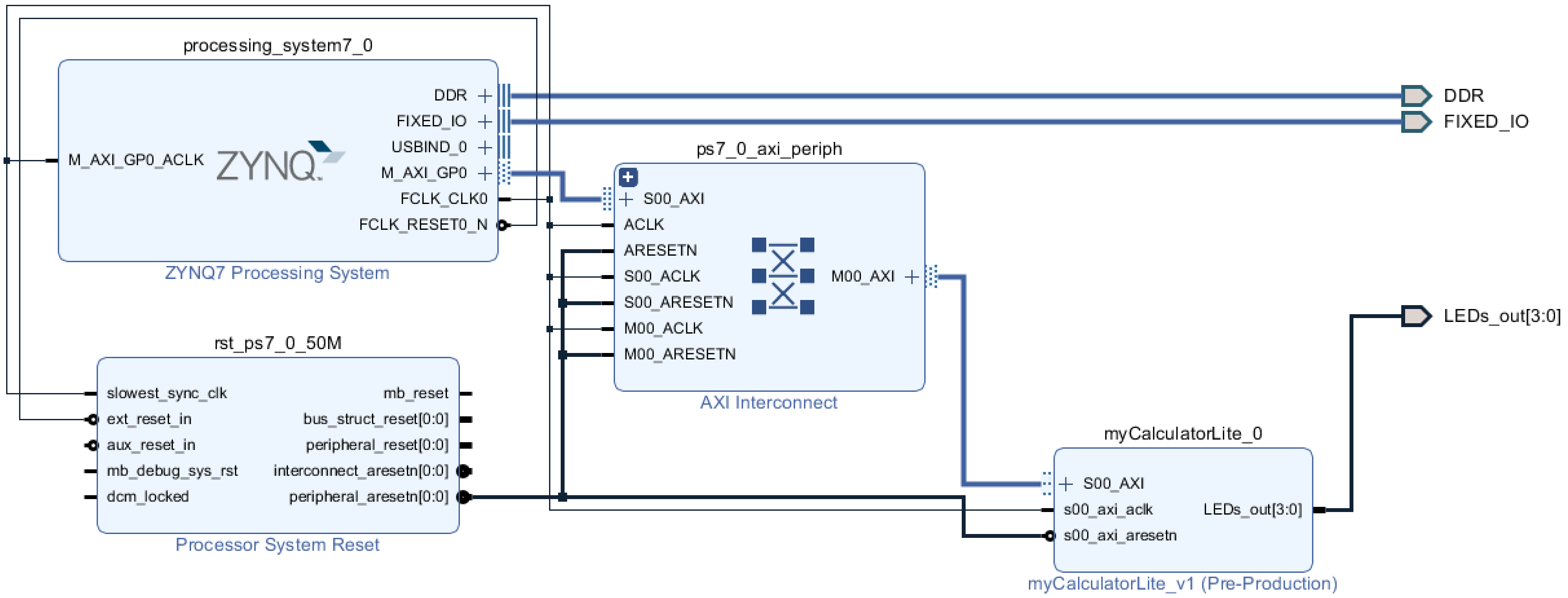
AXI_LITE interface

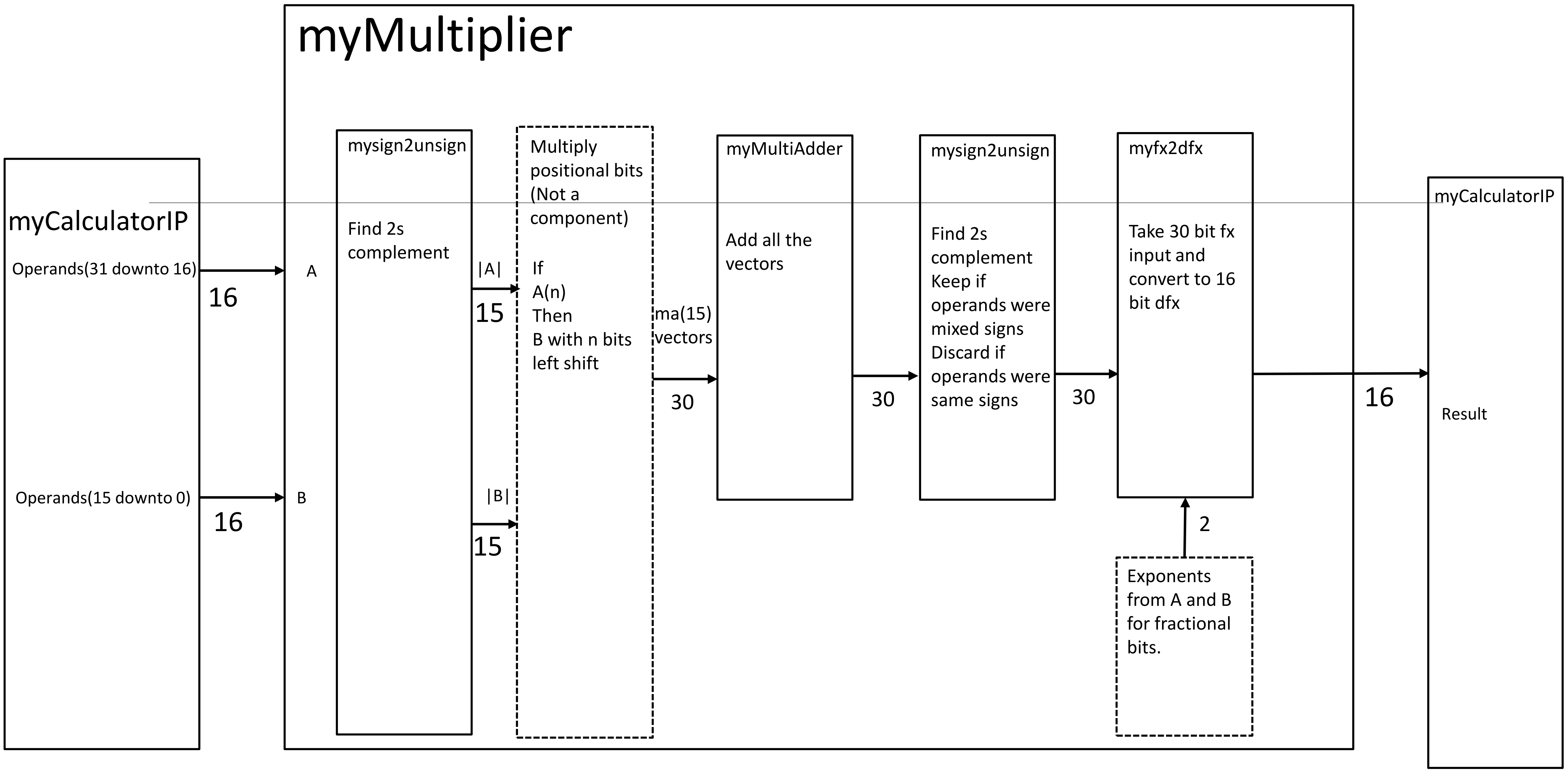


Hierarchy:

myCalculator_IP
- dfx_addsub
- mymultiplier
- mydivider

Block Diagram




myMultiplier
myCalculatorIP
Operands(31 downto 16)
16
A
Operands(15 downto 0)
16
B
mysign2unsign
Find 2s complement
|A|
15
|B|
15
Multiply positional bits (Not a component)
If A(n) Then B with n bits left shift
ma(15) vectors
30
myMultiAdder
Add all the vectors
30
mysign2unsign
Find 2s complement Keep if operands were mixed signs Discard if operands were same signs
30
myfx2dfx
Take 30 bit fx input and convert to 16 bit dfx
2
Exponents from A and B for fractional bits.
16
myCalculatorIP
Result

mysign2unsign

Generates the 2s compliment of whatever you put in.

Used to take absolute value of negative inputs

Multiply positional bits

If A(n) Then B with n bits left shift.   Keep several vectors for processing.

```
Example
   1111
x 10101
   1111          Vectors
 111100
11110000
```

Mymultiadder

uses an array of fulladd components to add all of the vectors obtained previously

Myfx2dfx

converts the fx result of multiplying the two operands into the final dfx [16 8 4] result

Looks at the two exponent bits to determine if the multiplied result has 8, 12 or 16 fractional bits.

After determining how many integer bits, it checks to see if this can be shown with 7 integer bits for a num0. (To do this we check to see that all bits downto the seventh integer bit are the same as the seventh integer bit.)

If so we take the 7 integer bits with 8 fractional bits and truncate the rest.

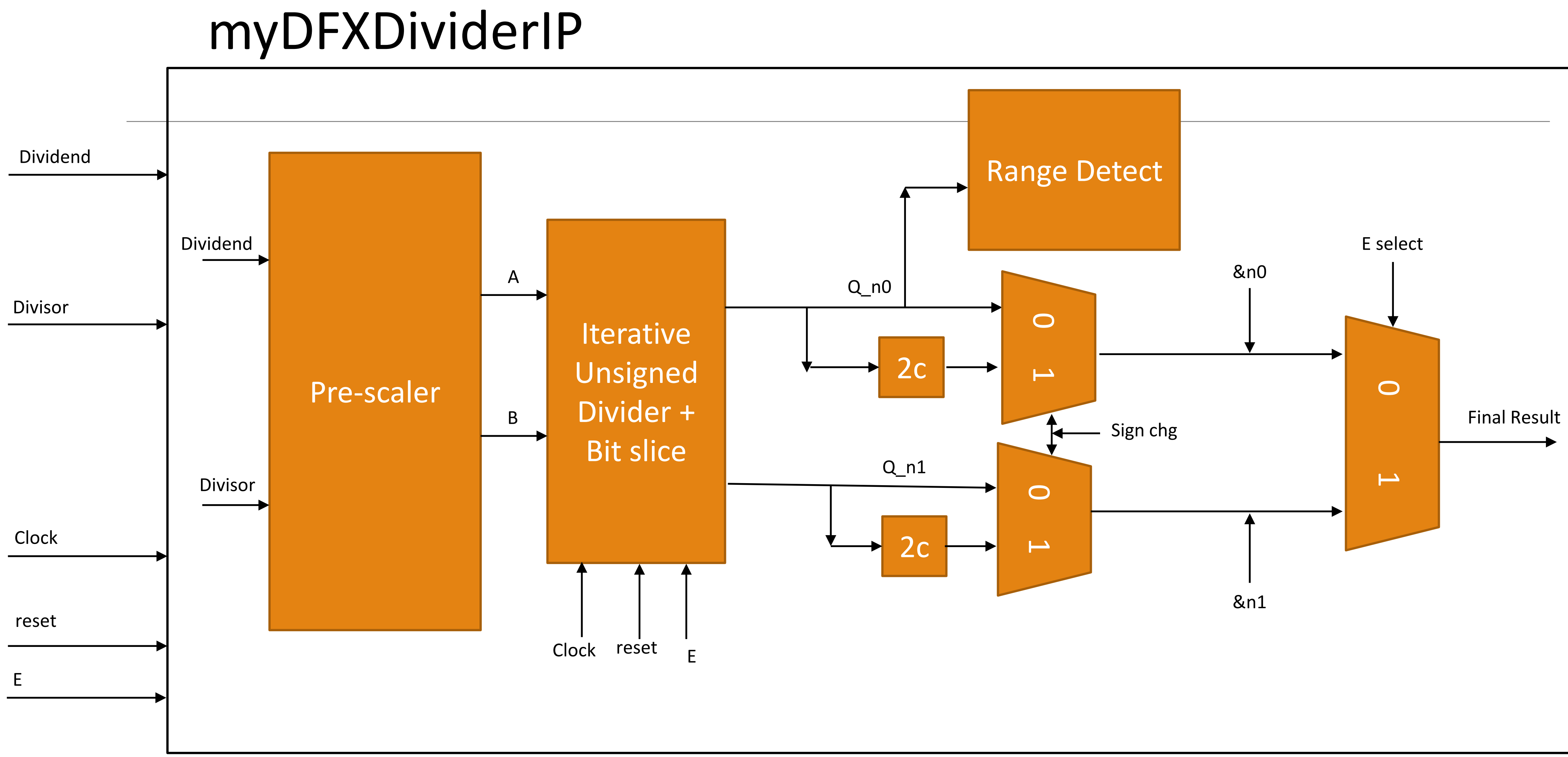
If not, we take 11 integer bits with 4 fractional bits and truncate the rest.

We then add in the exponent dependent on num0 or num1.

DFX Division Procedure

1. Convert from DFX-FX
2. Perform 2c if negative
3. Align fractional points to get integers
4. Perform Unsigned integer division
5. Place fractional point
6. Use range detector to determine if n0 or n1
7. Perform 2c if needed
8. Convert to DFX based on output of range detector

DFX Divider Block Diagram



Unsigned Iterative Divider Architecture

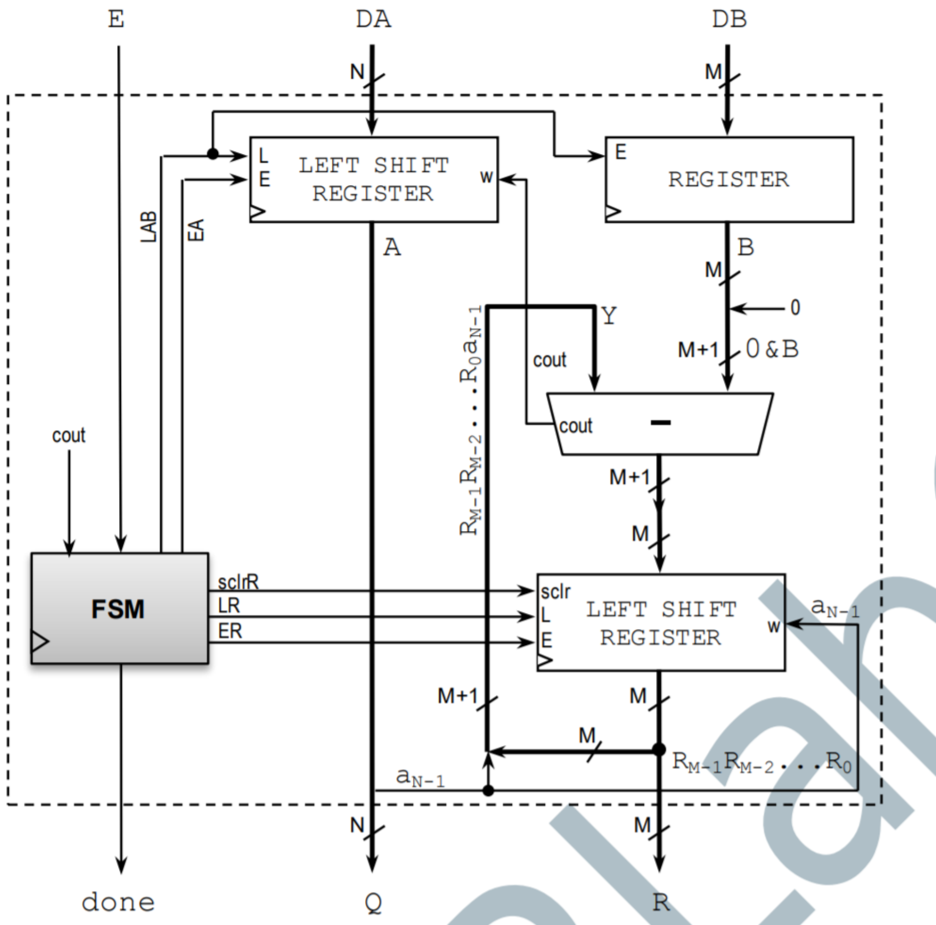
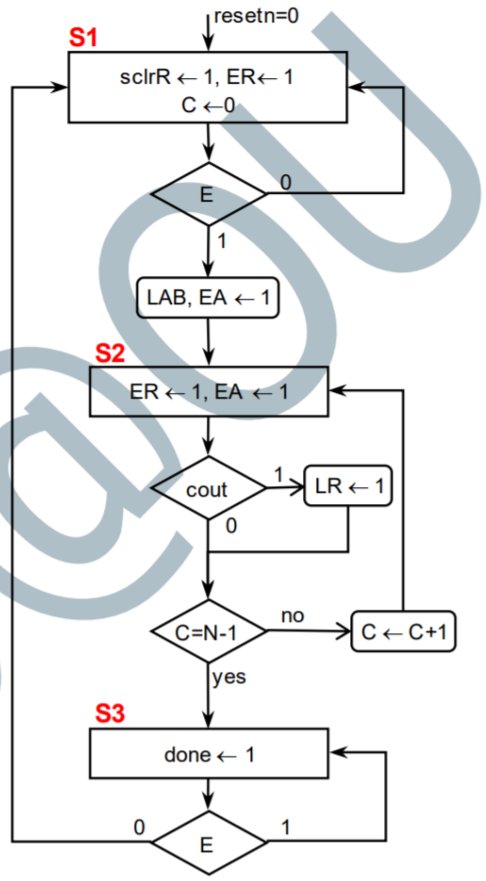
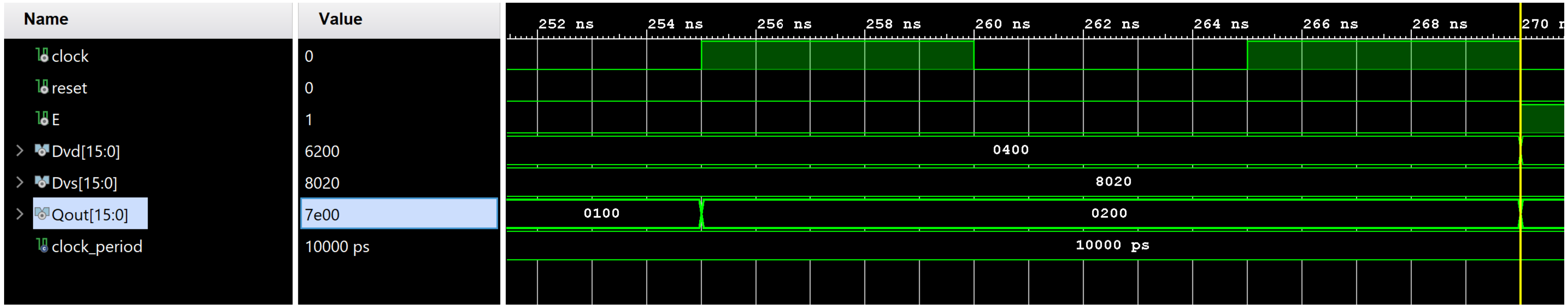


Figure 8. Iterative Divider



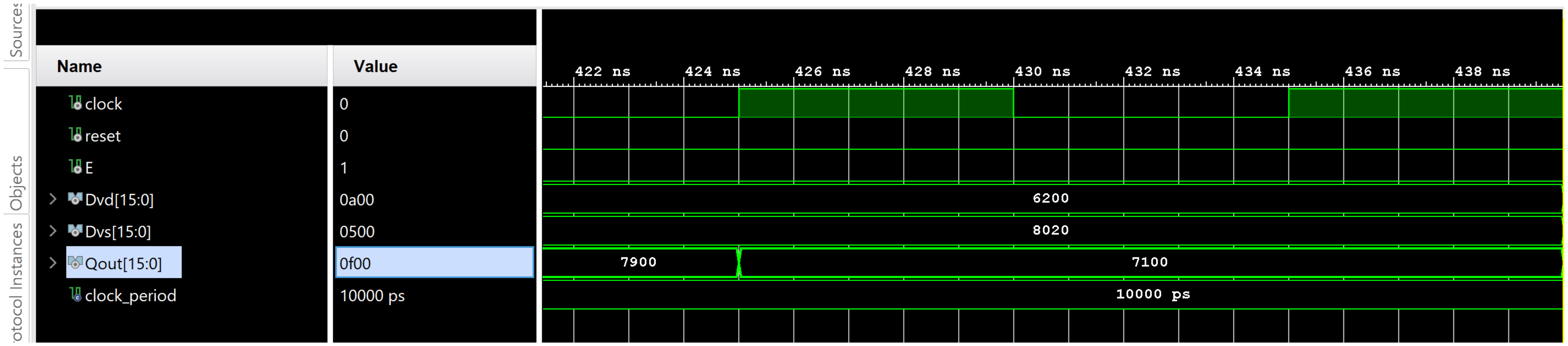
Divider Test

Dividend = 4 (n0) Divisor = 2 (n1)



Name	Value
clock	0
reset	0
E	1
Dvd[15:0]	6200
Dvs[15:0]	8020
Qout[15:0]	7e00
clock_period	10000 ps

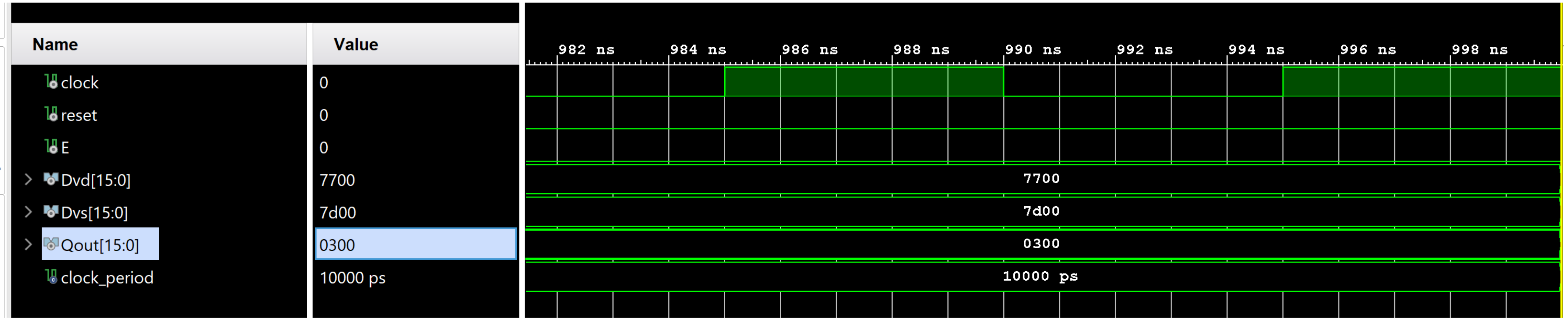
Dividend = -30 (n0) Divisor = 2 (n1)



Name	Value
clock	0
reset	0
E	1
Dvd[15:0]	0a00
Dvs[15:0]	0500
Qout[15:0]	0f00
clock_period	10000 ps

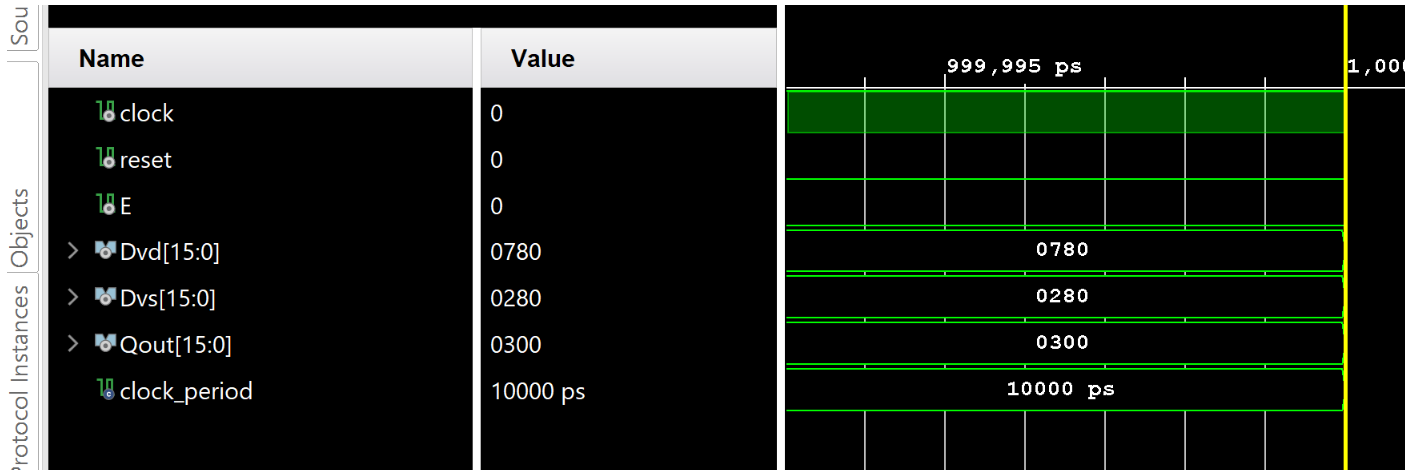
Divider Test

Dividend = -9 (n0)   Divisor = -3 (n0)



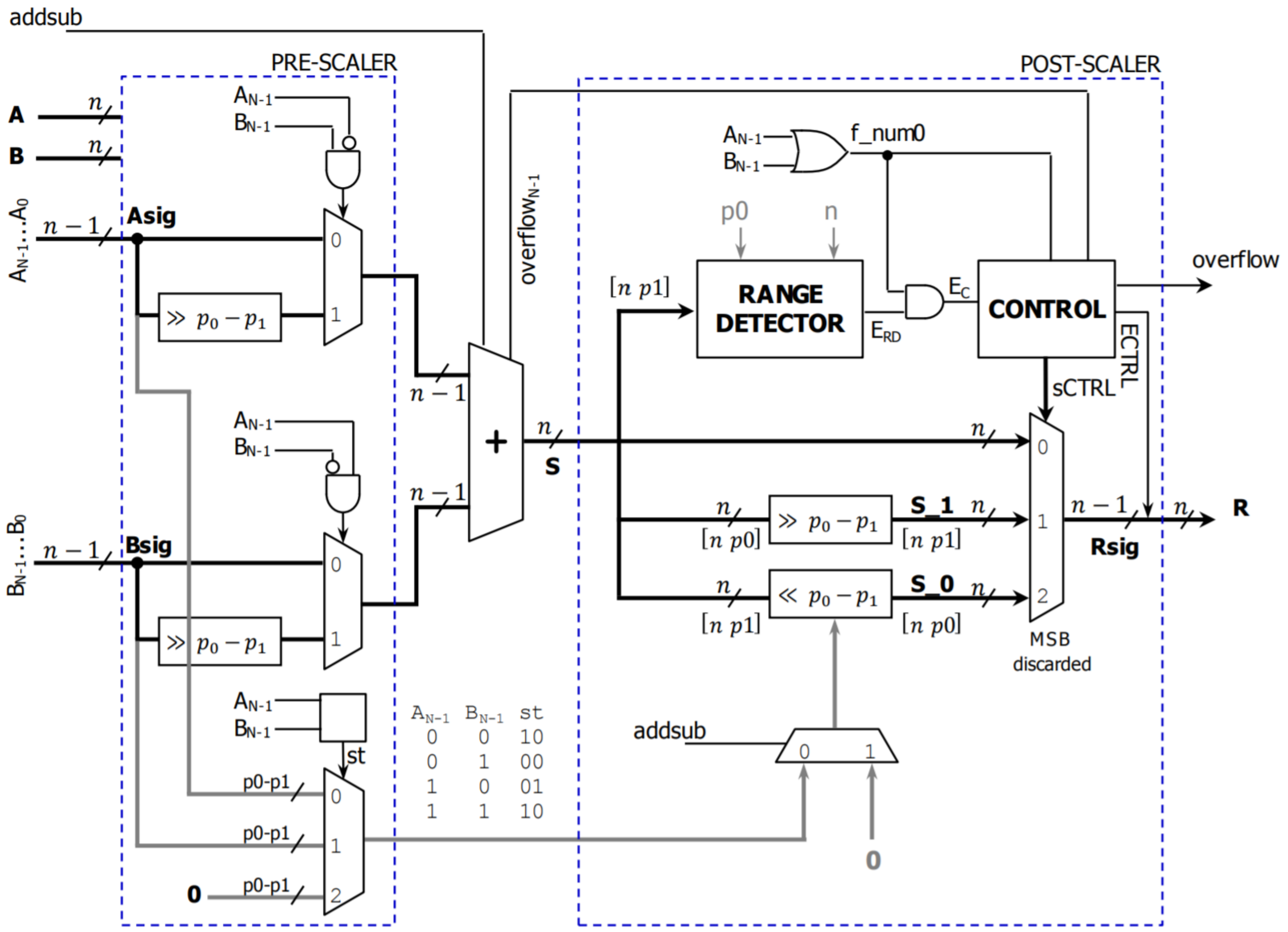
Name	Value
clock	0
reset	0
E	0
Dvd[15:0]	7700
Dvs[15:0]	7d00
Qout[15:0]	0300
clock_period	10000 ps

Dividend = 7.5 (n0) Divisor = 2.5 (n0)



Name	Value
clock	0
reset	0
E	0
Dvd[15:0]	0780
Dvs[15:0]	0280
Qout[15:0]	0300
clock_period	10000 ps

DFX Add/Subtract



A_{N-1}	B_{N-1}	st
0	0	10
0	1	00
1	0	01
1	1	10

Issues

The first design used switch inputs for mode selection. This conflicted with the adder/subtractor IP. The solution was to remove the switch inputs and use a second slave register write for the mode select.

The test bench was not providing outputs when checking the axi rdata. This was found to be caused by variables that were not initialized. It's worth noting that in implementation the variables were grounded so this did not interfere with SDK trials and was only noticed in the testbench.

Two variables had multiple assignments which prevented implementation.

The initial design had a mux controlling the inputs to the calculator functions. If the addition mode was selected than the mux would zero out the inputs as they were fed to subtraction, multiplication and division. This design was scrapped for a dmux that would take results from all functions and based on mode selection choose which to send out on the axi data bus. This does mean however that all functions are running even when only one function can be requested at a time.

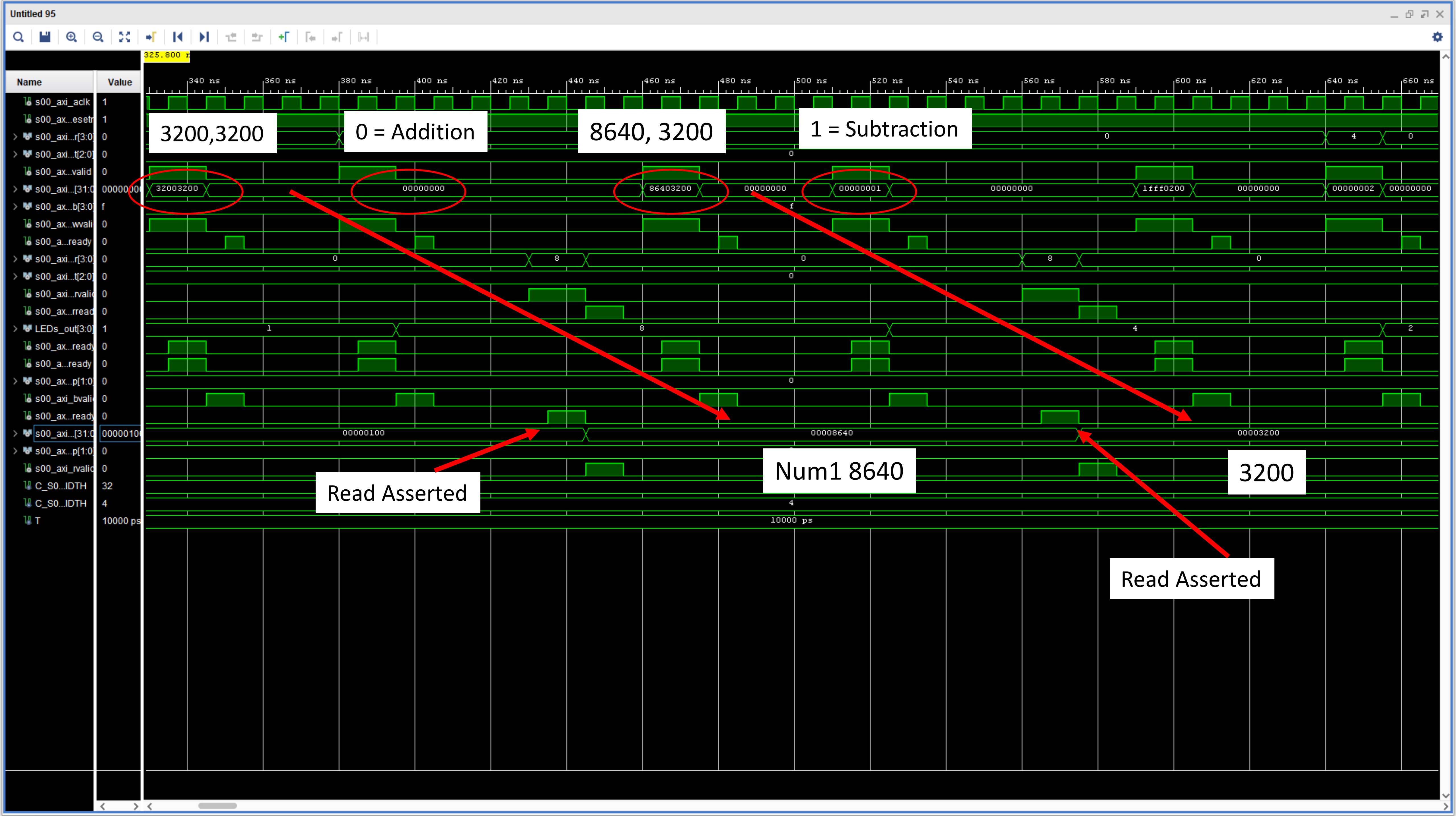
Limitations

Iterative circuit

Only two operands

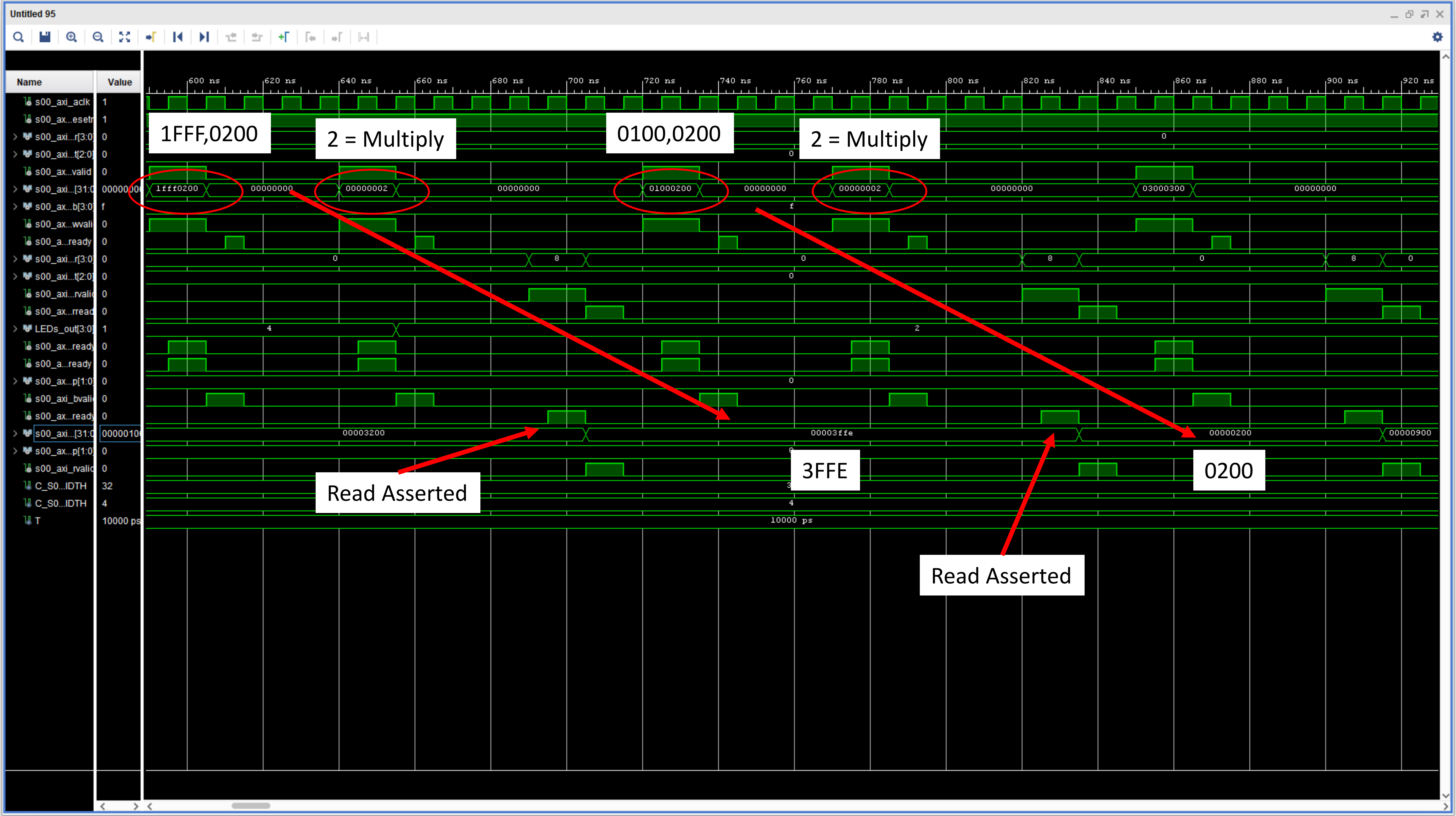
Does not validate inputs – An invalid num1 could be sent which would give inaccurate results.

Does not validate results – Two valid inputs could result in an overflow. This would be ignored and inaccurate results would be provided.

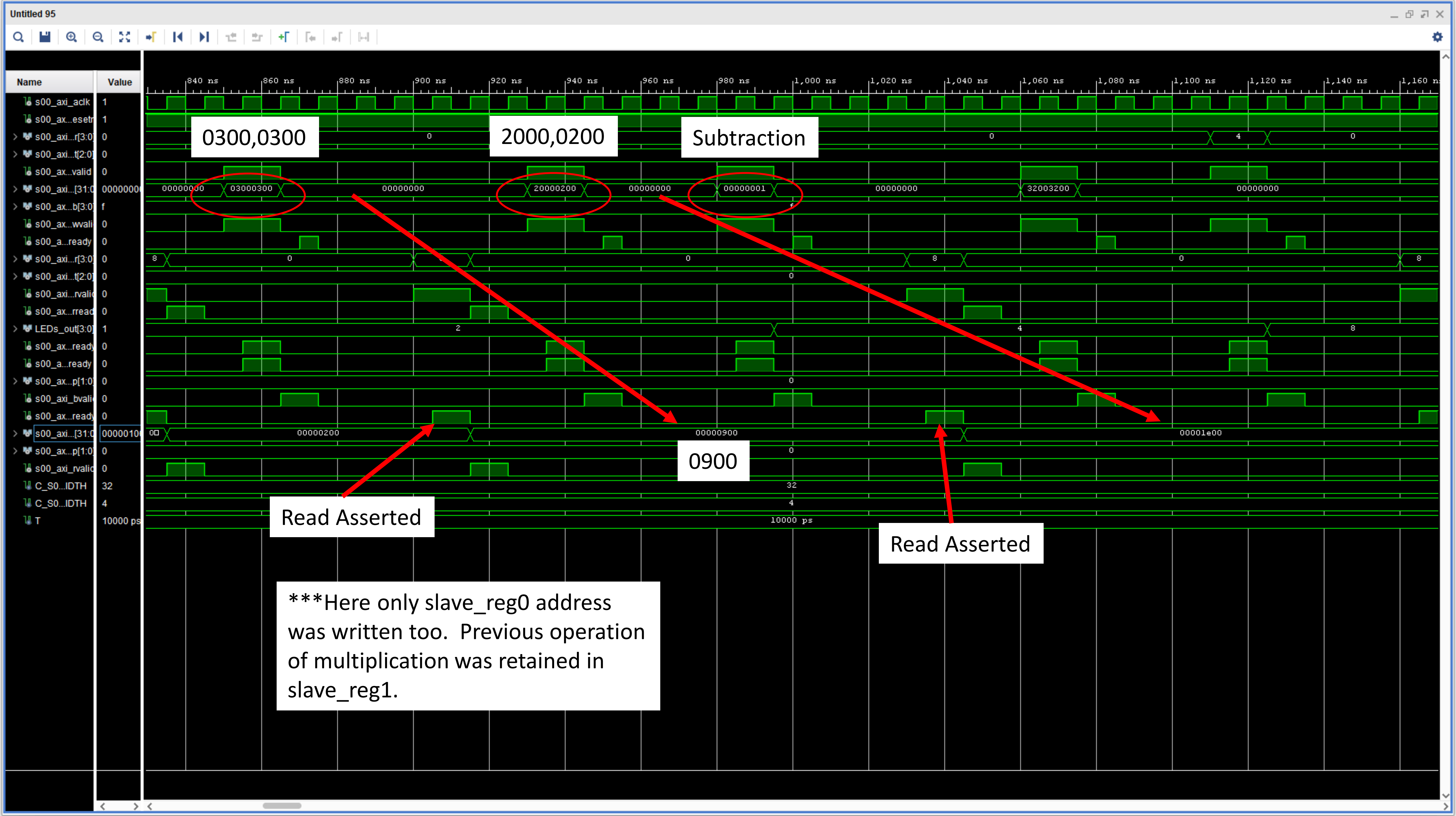


```
Name              Value
s00_axi_aclk      1
s00_ax...esetn    1
s00_axi...r[3:0]  0
s00_axi...t[2:0]  0
s00_ax...valid    0
s00_axi...[31:0]  0000000
s00_ax...b[3:0]   f
s00_ax...wvalid   0
s00_a...ready     0
s00_axi...r[3:0]  0
s00_axi...t[2:0]  0
s00_axi...rvalid  0
s00_ax...rready   0
LEDs_out[3:0]     1
s00_ax...ready    0
s00_a...ready     0
s00_ax...p[1:0]   0
s00_axi_bvalid    0
s00_ax...ready    0
s00_axi...[31:0]  0000010
s00_ax...p[1:0]   0
s00_axi_rvalid    0
C_S0...IDTH       32
C_S0...IDTH       4
T                 10000 ps
```

```
s00_axi...[31:0] (write data):  32003200  00000000  86403200  00000000  00000001  00000000  1fff0200  00000000  00000002  00000000
s00_axi...r[3:0] (write addr):  0  4  0
s00_axi...r[3:0] (read addr):   0  8  0  8  0
LEDs_out[3:0]:                  1  8  4  2
s00_axi...[31:0] (read data):   00000100  00008640  00003200
C_S0...IDTH:                    4
T:                              10000 ps
```



Name	Value
s00_axi_aclk	1
s00_ax...esetn	1
s00_axi...r[3:0]	0
s00_axi...t[2:0]	0
s00_ax...valid	0
s00_axi...[31:0]	0000000
s00_ax...b[3:0]	f
s00_ax...wvalid	0
s00_a...ready	0
s00_axi...r[3:0]	0
s00_axi...t[2:0]	0
s00_axi...rvalid	0
s00_ax...rready	0
LEDs_out[3:0]	1
s00_ax...ready	0
s00_a...ready	0
s00_ax...p[1:0]	0
s00_axi_bvalid	0
s00_ax...ready	0
s00_axi...[31:0]	0000010
s00_ax...p[1:0]	0
s00_axi_rvalid	0
C_S0...IDTH	32
C_S0...IDTH	4
T	10000 ps



Name	Value
s00_axi_aclk	1
s00_ax...esetr	1
s00_axi...r[3:0]	0
s00_axi...t[2:0]	0
s00_ax...valid	0
s00_axi...[31:0	0000000
s00_ax...b[3:0]	f
s00_ax...wvali	0
s00_a...ready	0
s00_axi...r[3:0]	0
s00_axi...t[2:0]	0
s00_axi...rvalid	0
s00_ax...rread	0
LEDs_out[3:0]	1
s00_ax...ready	0
s00_a...ready	0
s00_ax...p[1:0]	0
s00_axi_bvalid	0
s00_ax...ready	0
s00_axi...[31:0	0000010
s00_ax...p[1:0]	0
s00_axi_rvalid	0
C_S0...IDTH	32
C_S0...IDTH	4
T	10000 ps

0300,0300

2000,0200

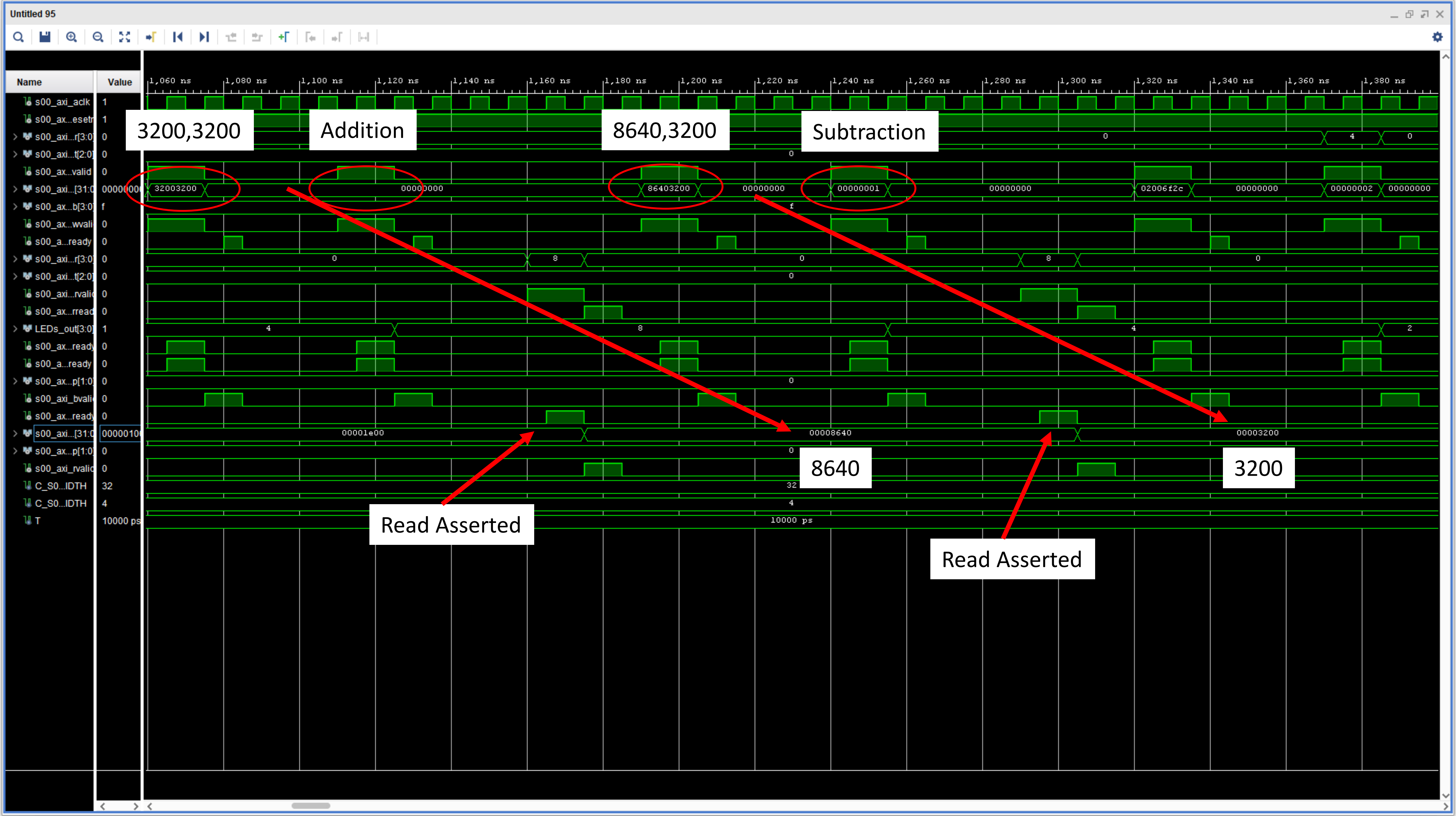
Subtraction

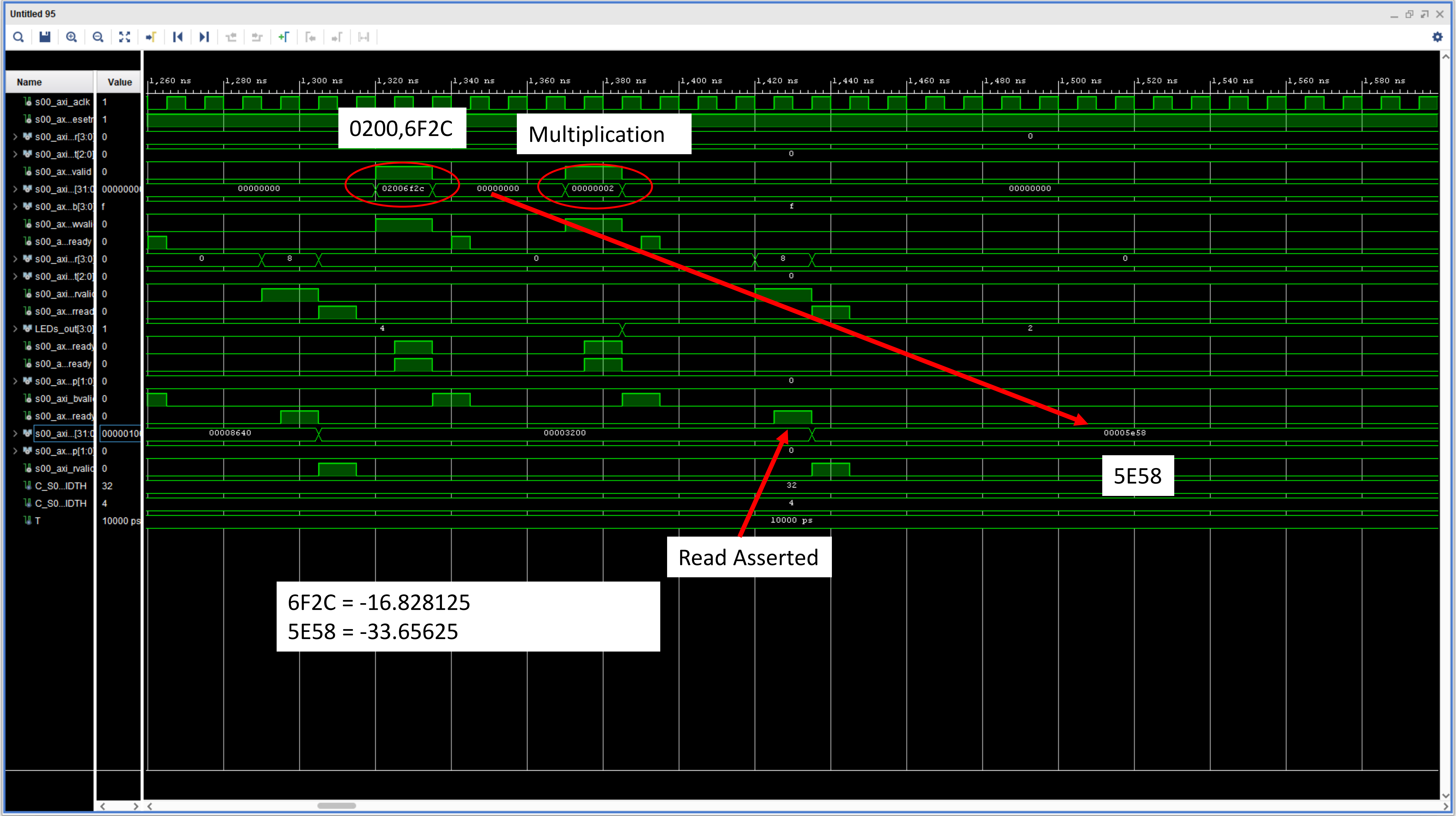
0900

Read Asserted

Read Asserted

***Here only slave_reg0 address was written too. Previous operation of multiplication was retained in slave_reg1.


Untitled 95
Name
Value
s00_axi_aclk
s00_ax...esetn
s00_axi...r[3:0]
s00_axi...t[2:0]
s00_ax...valid
s00_axi...[31:0]
s00_ax...b[3:0]
s00_ax...wvalid
s00_a...ready
s00_axi...r[3:0]
s00_axi...t[2:0]
s00_axi...rvalid
s00_ax...rready
LEDs_out[3:0]
s00_ax...ready
s00_a...ready
s00_ax...p[1:0]
s00_axi_bvalid
s00_ax...ready
s00_axi...[31:0]
s00_ax...p[1:0]
s00_axi_rvalid
C_S0...IDTH
C_S0...IDTH
T
3200,3200
Addition
8640,3200
Subtraction
32003200
00000000
86403200
00000001
02006f2c
00000002
00001e00
00008640
00003200
8640
3200
Read Asserted
Read Asserted
10000 ps



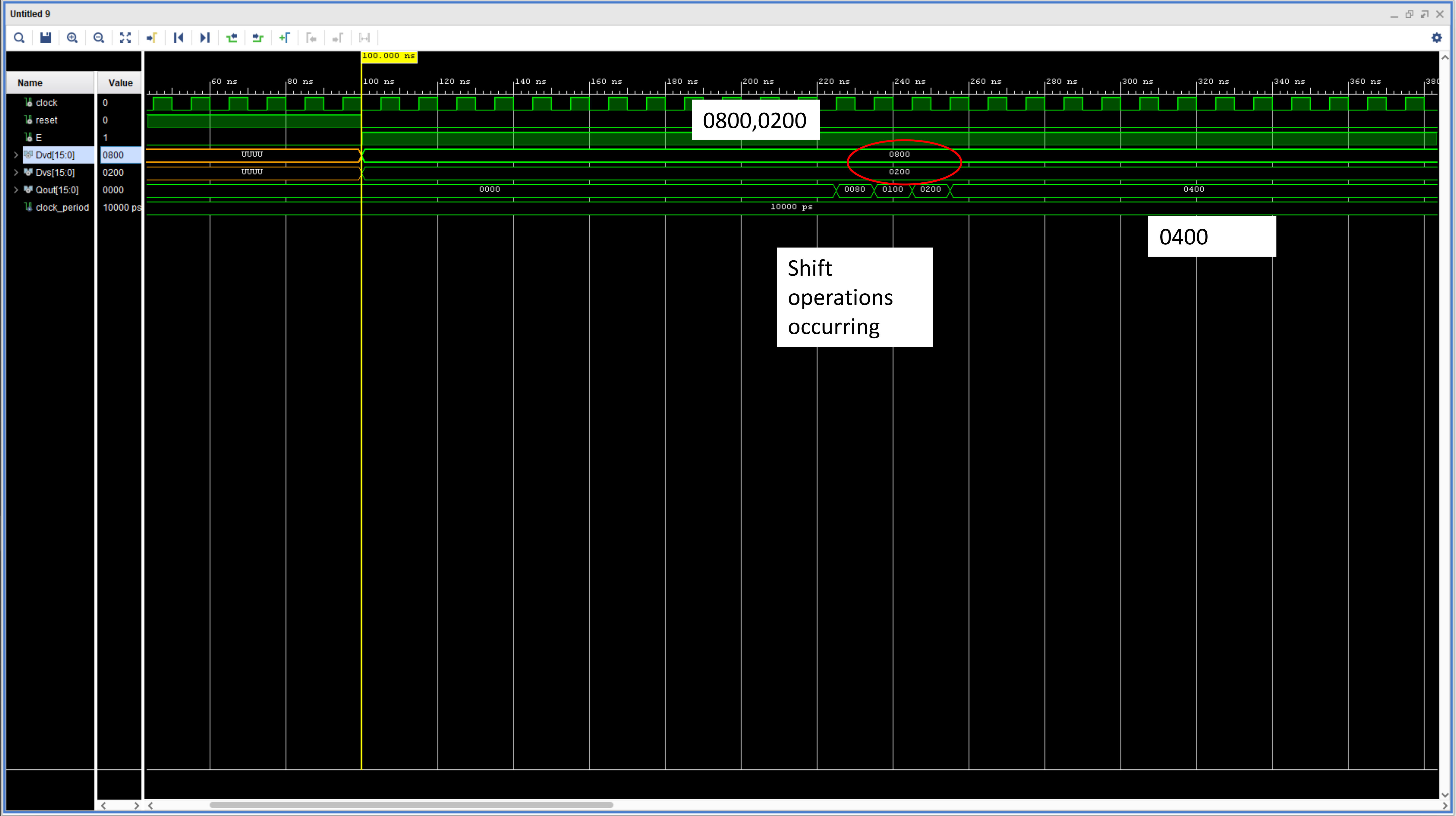
0200,6F2C

Multiplication

5E58

Read Asserted

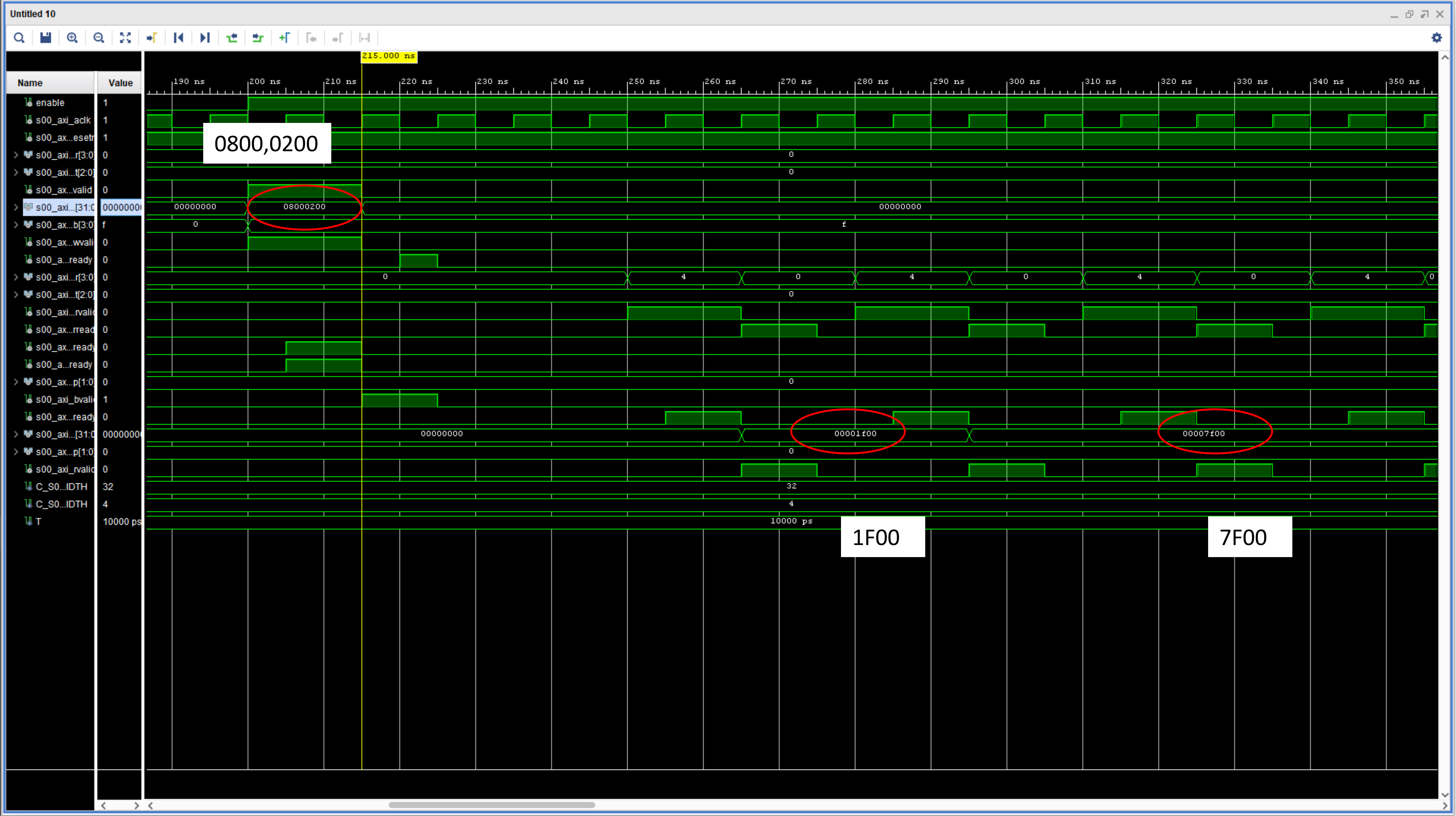
6F2C = -16.828125
5E58 = -33.65625



0800,0200

Shift operations occurring

0400



Name	Value
enable	1
s00_axi_aclk	1
s00_ax...esetr	1
s00_axi...r[3:0]	0
s00_axi...t[2:0]	0
s00_ax...valid	0
s00_axi...[31:0	0000000
s00_ax...b[3:0]	f
s00_ax...wvali	0
s00_a...ready	0
s00_axi...r[3:0]	0
s00_axi...t[2:0]	0
s00_axi...rvali	0
s00_ax...rread	0
s00_ax...ready	0
s00_a...ready	0
s00_ax...p[1:0]	0
s00_axi_bvali	1
s00_ax...ready	0
s00_axi...[31:0	0000000
s00_ax...p[1:0]	0
s00_axi_rvalid	0
C_S0...IDTH	32
C_S0...IDTH	4
T	10000 ps

0800,0200

1F00

7F00