|  |
| --- |
| 台科大 |
| PA1 |
| Computer Organization |

|  |
| --- |
| 周 柏宇  2024/3/28 |

目錄

[Multiplier 3](#_Toc163960744)

[CompMultiplier 3](#_Toc163960745)

[Description 3](#_Toc163960746)

[Details 4](#_Toc163960747)

[ALU 6](#_Toc163960748)

[Description 6](#_Toc163960749)

[Detail 6](#_Toc163960750)

[Multiplicand 7](#_Toc163960751)

[Description 7](#_Toc163960752)

[Detail 7](#_Toc163960753)

[Control 8](#_Toc163960754)

[Description 9](#_Toc163960755)

[Detail 9](#_Toc163960756)

[Product 10](#_Toc163960757)

[Description 10](#_Toc163960758)

[Detail 11](#_Toc163960759)

[Divider 12](#_Toc163960760)

[CompDivider 12](#_Toc163960761)

[Description 13](#_Toc163960762)

[Detail 13](#_Toc163960763)

[ALU 14](#_Toc163960764)

[Description 14](#_Toc163960765)

[Detail 14](#_Toc163960766)

[Divisor 14](#_Toc163960767)

[Description 15](#_Toc163960768)

[Detail 15](#_Toc163960769)

[Control 16](#_Toc163960770)

[Description 17](#_Toc163960771)

[Detail 17](#_Toc163960772)

[Remainder 18](#_Toc163960773)

[Description 18](#_Toc163960774)

[Detail 19](#_Toc163960775)

[Test 20](#_Toc163960776)

[Multiplier 20](#_Toc163960777)

[tb\_CompMultiplier 20](#_Toc163960778)

[tb\_ALU 21](#_Toc163960779)

[tb\_Control 22](#_Toc163960780)

[tb\_Multiplicand 23](#_Toc163960781)

[tb\_Product 23](#_Toc163960782)

[Divider 25](#_Toc163960783)

[tb\_Control 25](#_Toc163960784)

[tb\_ALU 26](#_Toc163960785)

[tb\_Divisor 27](#_Toc163960786)

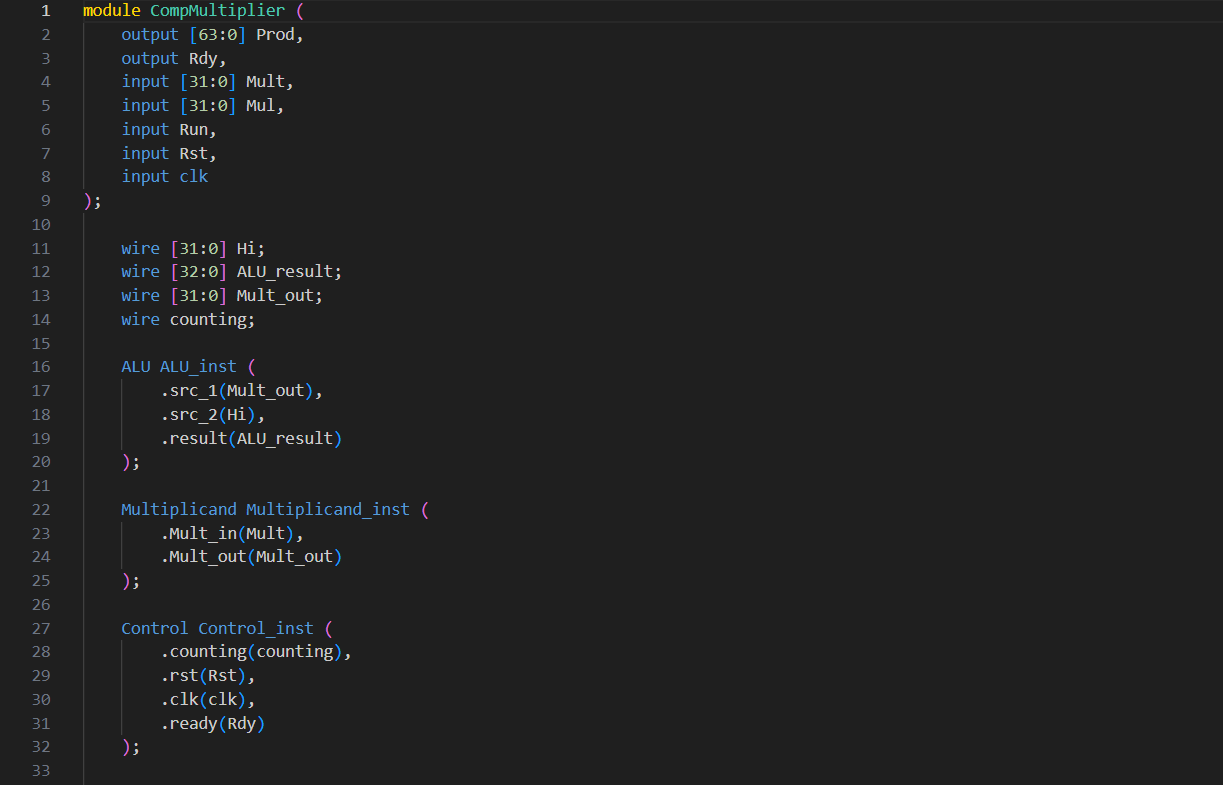
[tb\_Remainder 28](#_Toc163960787)

[tb\_CompDivider 29](#_Toc163960788)

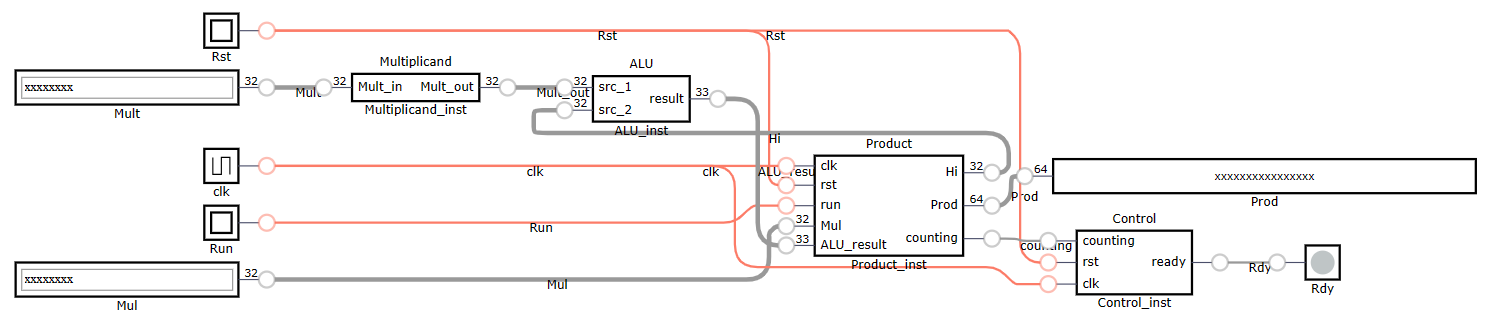
[Conclusion and Insights 31](#_Toc163960789)

# Multiplier

## CompMultiplier







### Description

This module is to connect all the small components to form a functional multiplier.

### Details

Line 1~9

I/O Interface.

|  |  |  |
| --- | --- | --- |
| Signal Symbol | Signal Name | Signal Description |
| Prod | 64-bit calculation result | This signal is set as an output before  the [Rdy](#Rdy) signal is activated to forbid the  testbench to read the previous calculation  result. |
| Rdy | completion signal | The “high” level represents the system has  completed multiplication and maintained the  result. |
| Mult | 32-bit multiplicand | The signal is generated by the testbench and  updated when the [Rst](#Rst) signal is “high”. |
| Mul | 32-bit multiplier | The signal is generated by the testbench and updated when the [Rst](#Rst) signal is “high”. |
| Run | execution signal | The system performs multiplication as the Run signal is “high” level. |
| Rst | initialization signal | The “high” level indicates that the system is initialized before multiplication, and the output results are set to zero. This signal has the highest priority and is generated by the testbench. |
| clk | clock signal | Periodic square waves are generated by the testbench for synchronizing each signal with the drive system. |

Line 11~14

Wires used in this module.

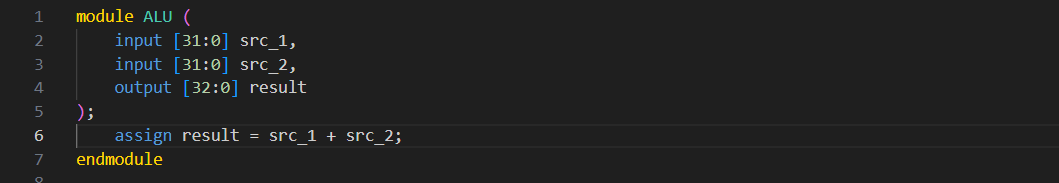
|  |  |  |
| --- | --- | --- |
| Wire Symbol | Wire Name | Wire Description |
| Hi | HI register | Upper 32 bits of the product. Used to pass data to ALU. |
| ALU\_result | 33 bits ALU result. This sees overflow as the 33th bit. | The computation result of ALU. |
| Mult\_out | 32-bit multiplicand | The signal is generated by the testbench and  updated when the [Rst](#Rst) signal is “high”. |
| counting | Counting flag | Tell Control to start. This will avoid racing condition. |

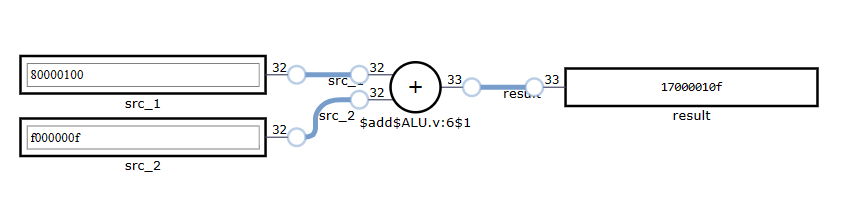
Line 16~43

Component instances.

|  |  |  |
| --- | --- | --- |
| Instance Name | Component Name | Instance Description |
| ALU\_inst | [ALU](#_ALU) instance | src\_1 gets data from [Mult\_out](#Mult_out), src\_2 gets data from [Hi](#Hi), result is the output of this ALU, connected to [ALU\_result](#ALU_result). |
| Multiplicand\_inst | [Multiplicand](#_Multiplicand) instance | Mult\_in gets data from [Mult](#Mult), Mult\_out sets output to [Mult\_out](#Mult_out). |
| Control\_inst | [Control](#_Control) instance | counting gets data from Product, rst gets data from [Rst](#Rst), clk gets data from [clk](#clk), ready sets output to [Rdy](#Rdy). |
| Product\_inst | [Product](#_Product) instance | clk gets data from [clk](#clk), rst gets data from [Rst](#Rst), run gets data from [Run](#Run), Mul gets data from [Mul](#Mul) on the posedge of [Run](#Run), ALU\_result gets data from [ALU\_result](#ALU_result), Hi outputs data to [Hi](#Hi), Prod generates output to [Prod](#Prod). And last but not least it gives counting to Control. |

## ALU





### Description

This module provides basic arithmetic functions that fulfills multiplier’s needs.

### Detail

Line 1~5

I/O Interface.

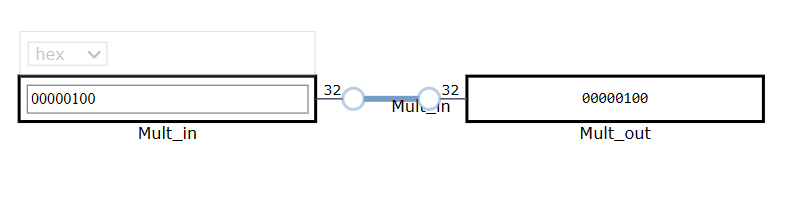
|  |  |  |
| --- | --- | --- |
| Signal Symbol | Signal Name | Signal Description |
| src\_1 | 32-bits Source 1 | addend |
| src\_2 | 32-bits Source 2 | augend |
| result | 33-bits addition result | Sum with 1 overflow bit |

Line 6

Wire connection that represents result = src\_1 + src\_2.

## Multiplicand





### Description

Dummy module. Mult signal is given all along between the 2 posedge of Rst. There is no need to do any other operation.

### Detail

Line 1~4

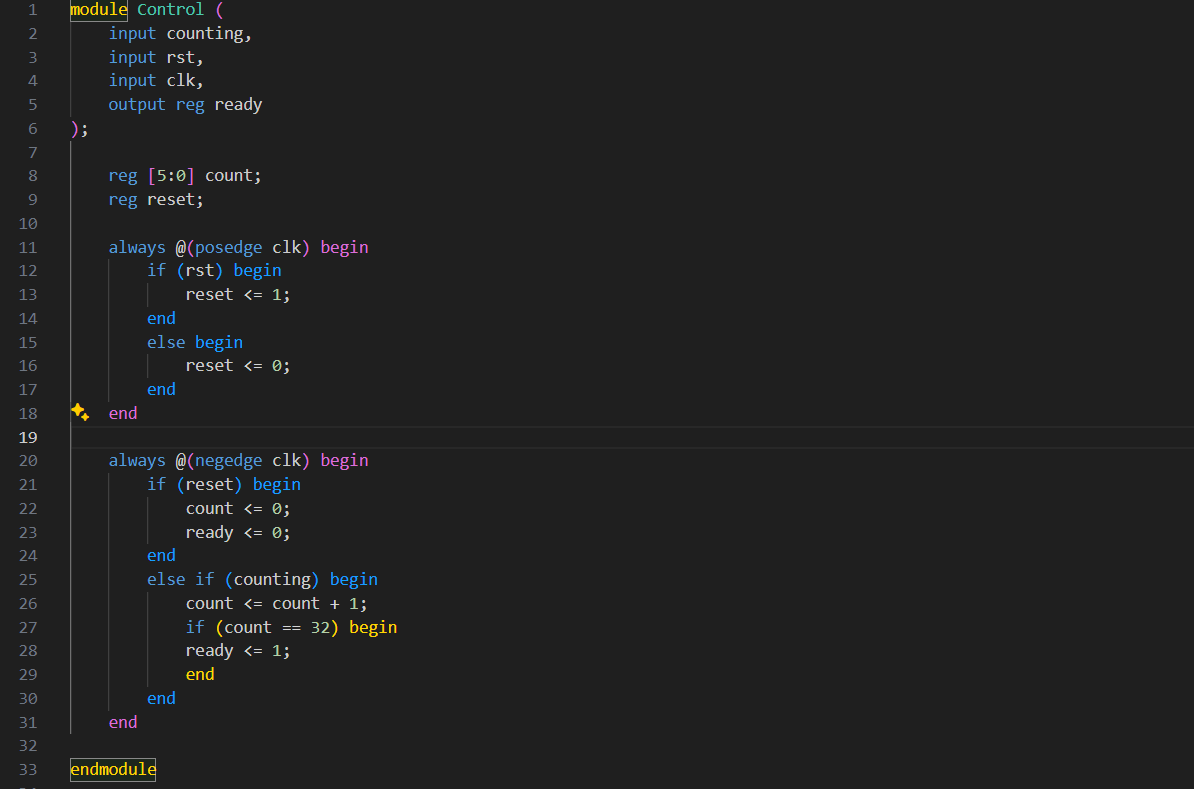
I/O Interface.

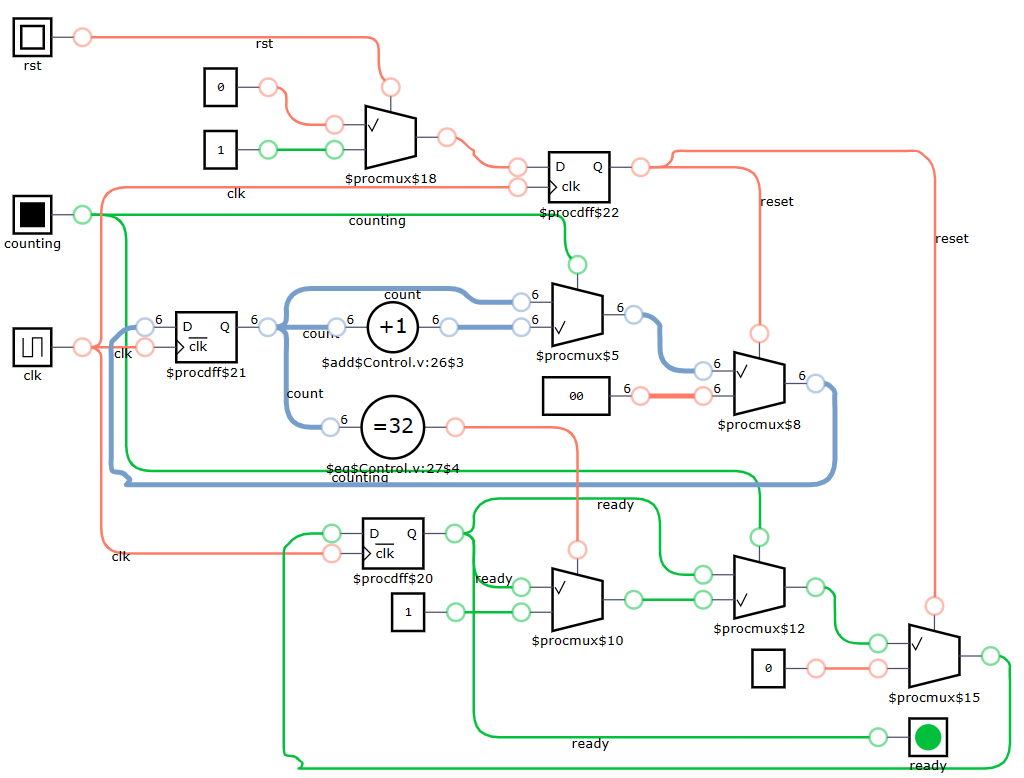
|  |  |  |
| --- | --- | --- |
| Signal Symbol | Signal Name | Signal Description |
| Mult\_in | In direction of Mult | In |
| Mult\_out | Out direction of Mult | out |

Line 5

Wire connection that denotes Mult\_out = Mult\_in.

## Control





### Description

A counter with a flag.

### Detail

Line 1~6

I/O Interface.

|  |  |  |
| --- | --- | --- |
| Signal Symbol | Signal Name | Signal Description |
| counting | Counting flag | The flag telling Control to count. |
| rst | initialization signal | The “high” level indicates that the system is initialized before multiplication, and the output results are set to zero. This signal has the highest priority and is generated by the testbench. |
| clk |  | Periodic square waves are generated by the testbench for synchronizing each signal with the drive system. |
| ready | Ready flag | Signify the end of the sequence of operation. |

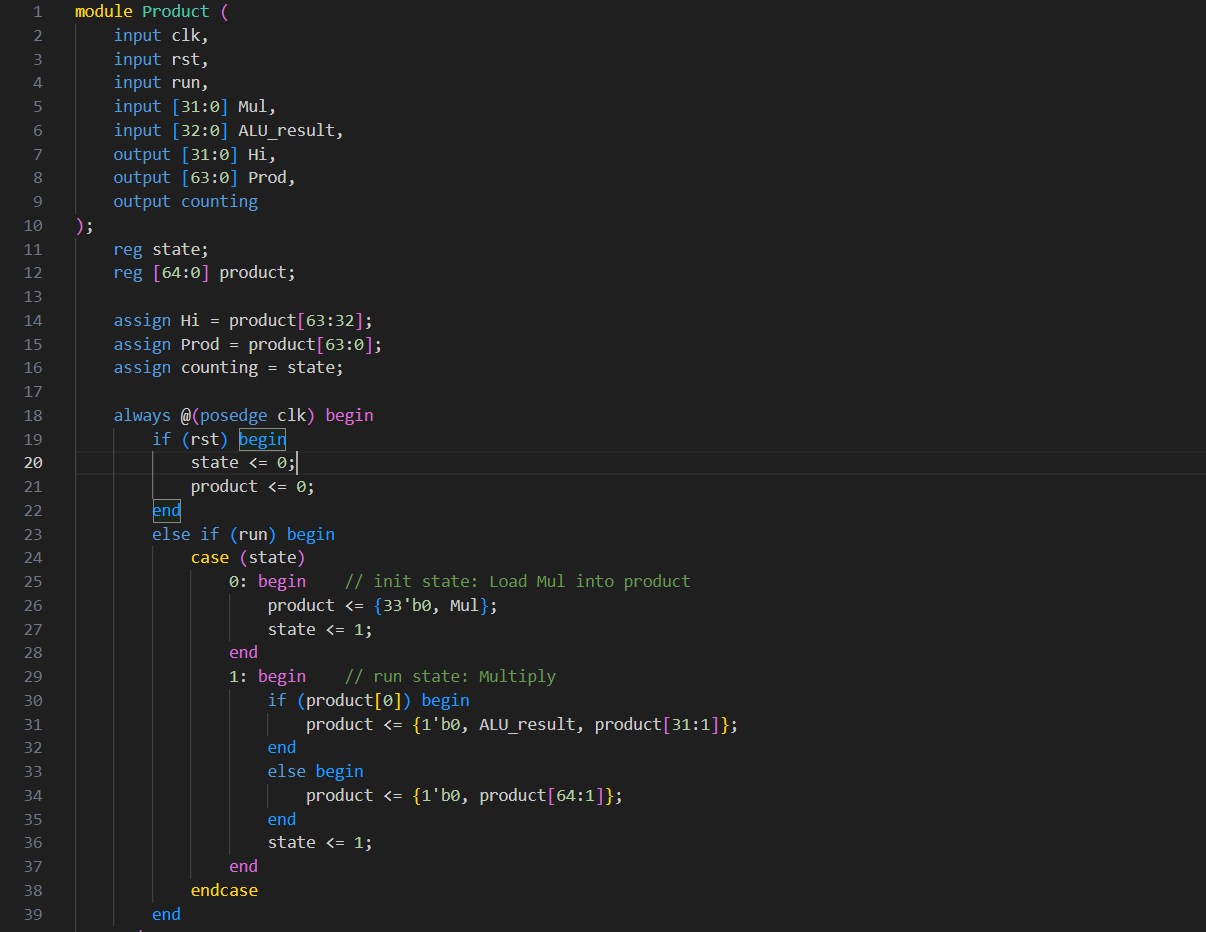
Line 11~18

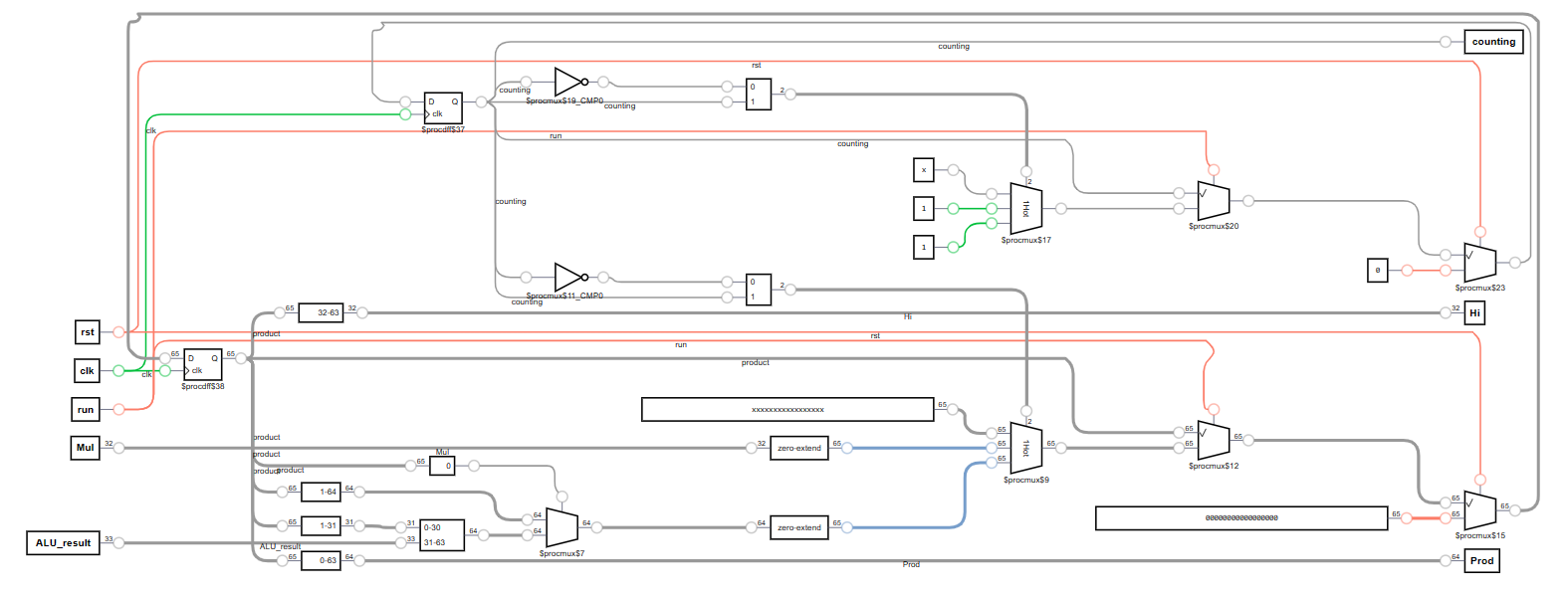
Use reset register to hold rst signal between 2 posedge clk to avoid racing.

Line 20~31

When running count 32 negedges of clk then set ready flag to notify other component to stop or fetch data.

## Product





### Description

Main module. I use 65 bits representing product in order to handle overflowing result from [ALU](#_ALU). The computation is executed on posedge clk to avoid racing condition. And I use counting, which is a flag to tell Control to count. This flag will effective avoid racing.

### Detail

Line 1~10

I/O Interface.

|  |  |  |
| --- | --- | --- |
| Signal Symbol | Signal Name | Signal Description |
| clk | clock | Clock |
| rst | reset | Reset the init state |
| run | Execution signal | Run operations when high |
| Mul | 32-bits multiplicand | Init will put this to LO |
| ALU\_result | 33-bits ALU result | With 1 more carry bit |
| Hi | HI | Upper 32 bits |
| Prod | production | 64-bits result |
| counting | Counting flag | Used to avoid racing |

Line 11~12

Registers

|  |  |  |
| --- | --- | --- |
| Register Symbol | Register Name | Register Description |
| state | state | FSM state |
| product | 65-bits product | 1 more bit for overflow result generated by [ALU](#_ALU). |

Line 14~16

Wire connection of [Hi](#PHi), [Prod](#PProd) and counting.

Line 19~22

Reset.

Line 24~38

FSM

|  |  |  |
| --- | --- | --- |
| State Symbol | State Name | State Description |
| 0 | Init | HI = 0, LO = [Mul](#PMul) |
| 1 | run | Examine LSB to do predefined [operations](#Pop). |

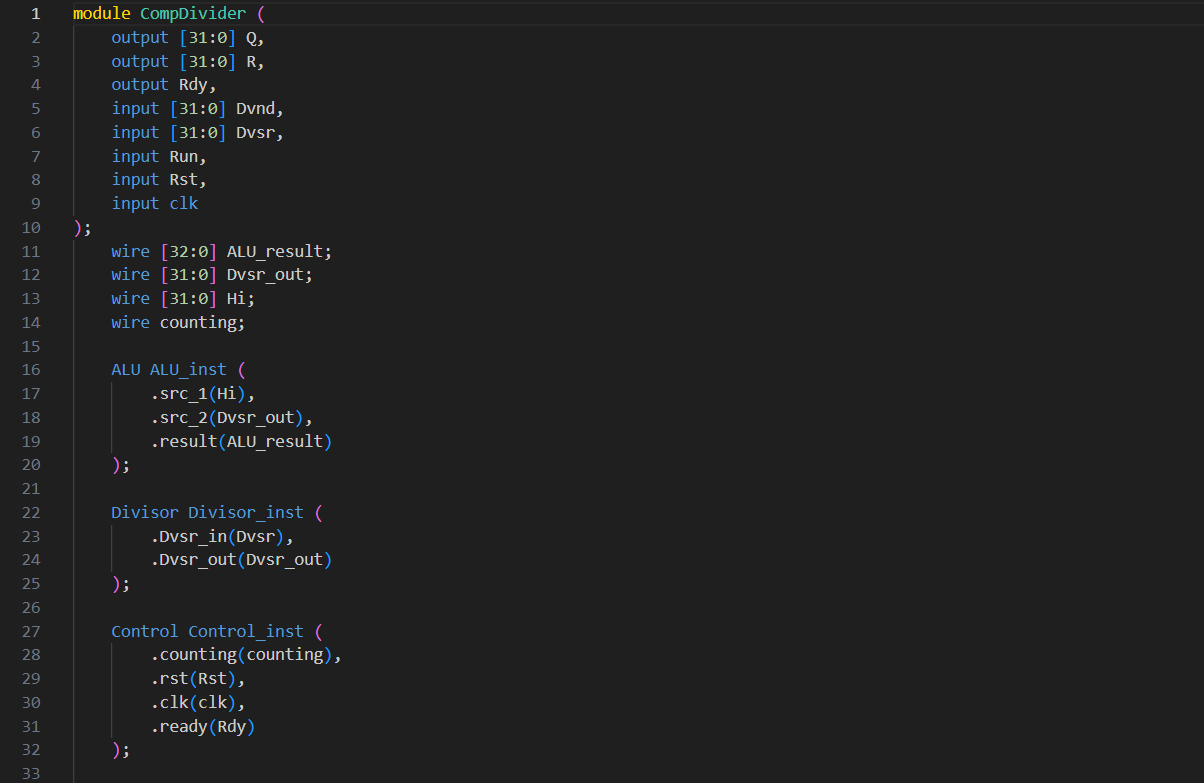
Line 30~36

If LSB is set, write ALU\_result to HI and shift right 1 bit.

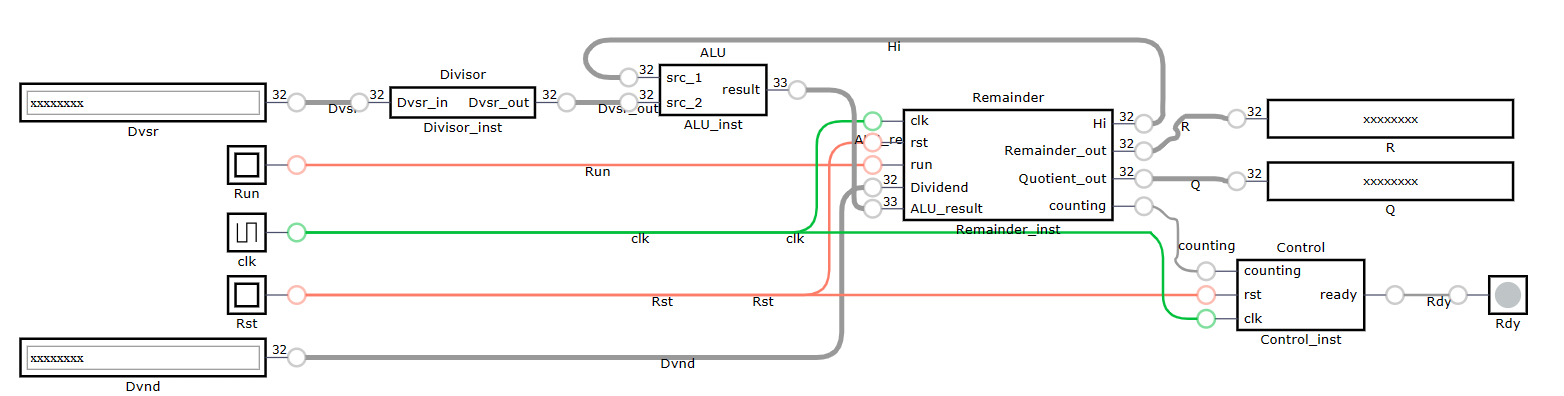
Else, shift right 1 bit.

# Divider

## CompDivider







### Description

This module is to connect all the small components to form a functional divider.

### Detail

Line 1~10

I/O Interface.

|  |  |  |
| --- | --- | --- |
| Signal Symbol | Signal Name | Signal Description |
| Q | 32-bit Quotient | Quotient result |
| R | 32-bit Reminder | Remainder result |
| Rdy | completion signal | Iff it’s set the, result is valid |
| Dvnd | 32-bit Dividend | Input |
| Dvsr | 32-bit Divisor | Input |
| Run | execution signal | Start execute when high |
| Rst | initialization signal | Reset |
| clk | clock signal | clock |

Line 11~14

Wires used in this module.

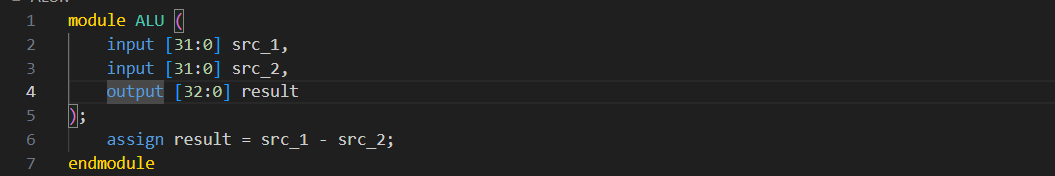
|  |  |  |
| --- | --- | --- |
| Wire Symbol | Wire Name | Wire Description |
| ALU\_result | 33-bits ALU out | With 1 carry bit |
| Dvsr\_out | Divisor out | Dummy connection |
| Hi | HI register | Upper 32 bits |
| counting | Counting flag | Used to avoid racing. |

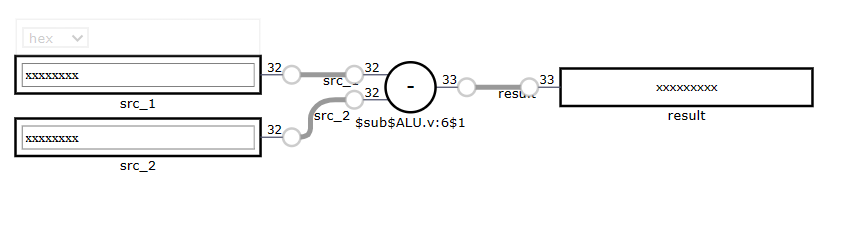
Line 16~44

Component instances.

|  |  |  |
| --- | --- | --- |
| Instance Symbol | Instance Name | Instance Description |
| ALU\_inst | [ALU](#_ALU_1) instance | Output to [ALU\_result](#DALU_result) considering [R](#R), [Dvsr\_out](#Dvsr_out) |
| Divisor\_inst | [Divisor](#_Divisor) instance | Dummy |
| Control\_inst | [Control](#_Control_1) instance | Counter with [Rdy](#RRdy) flag. |
| Remainder\_inst | [Remainder](#_Remainder) instance | 1 clk, 2 flags, 2 in, 4 out. For more detail [click me](#_Remainder). |

## ALU





### Description

This module provides basic arithmetic functions that fulfills divider’s needs.

### Detail

Line 1~5

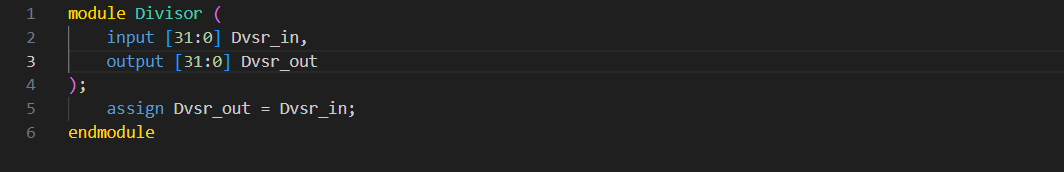
I/O Interface.

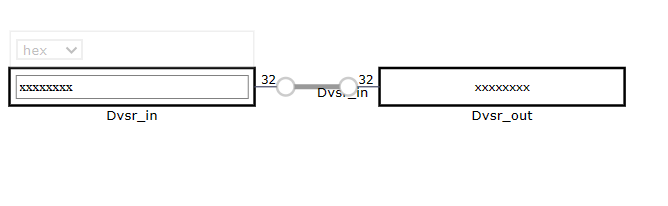
|  |  |  |
| --- | --- | --- |
| Signal Symbol | Signal Name | Signal Description |
| src\_1 | 32-bits source 1 | Minuend |
| src\_2 | 32-bits source 2 | subtrahend |
| result | 33-bits result | Difference, 1 more borrow bit |

Line 6

Wire connection symbolize result = src\_1 – src2.

## Divisor





### Description

Dummy

### Detail

Line 1~4

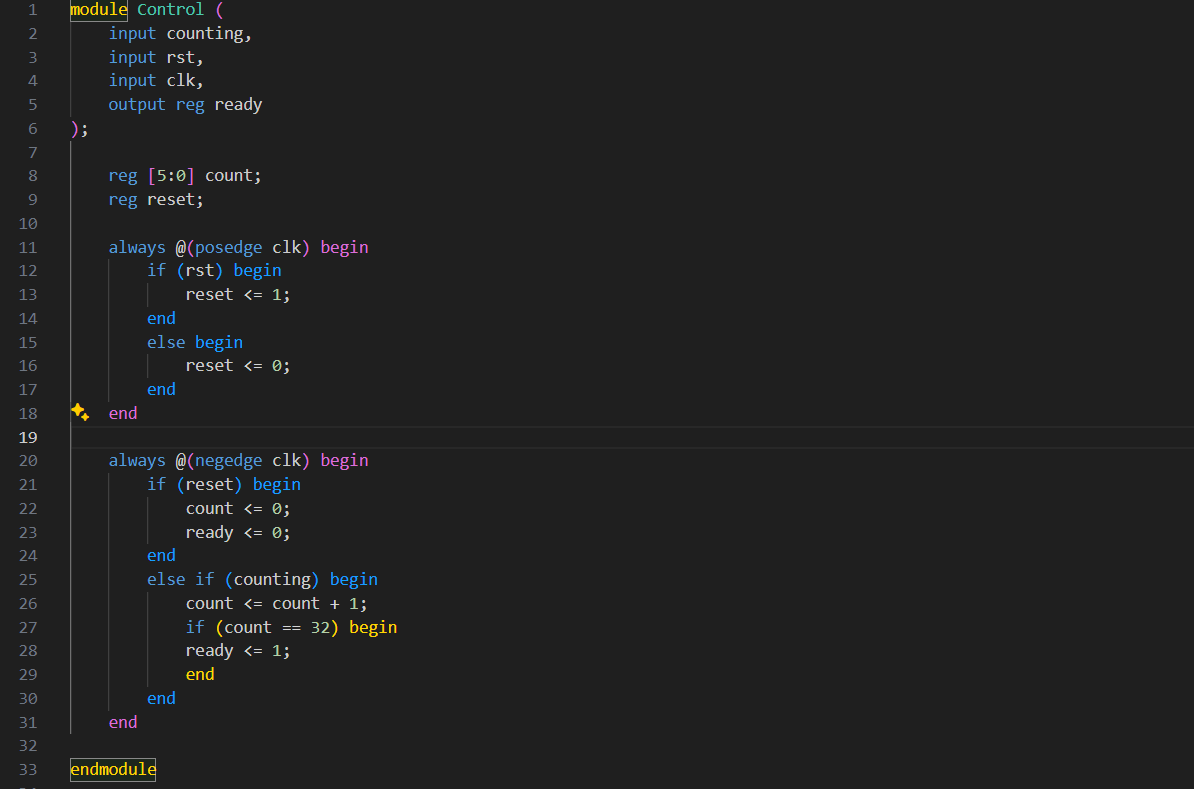
I/O Interface.

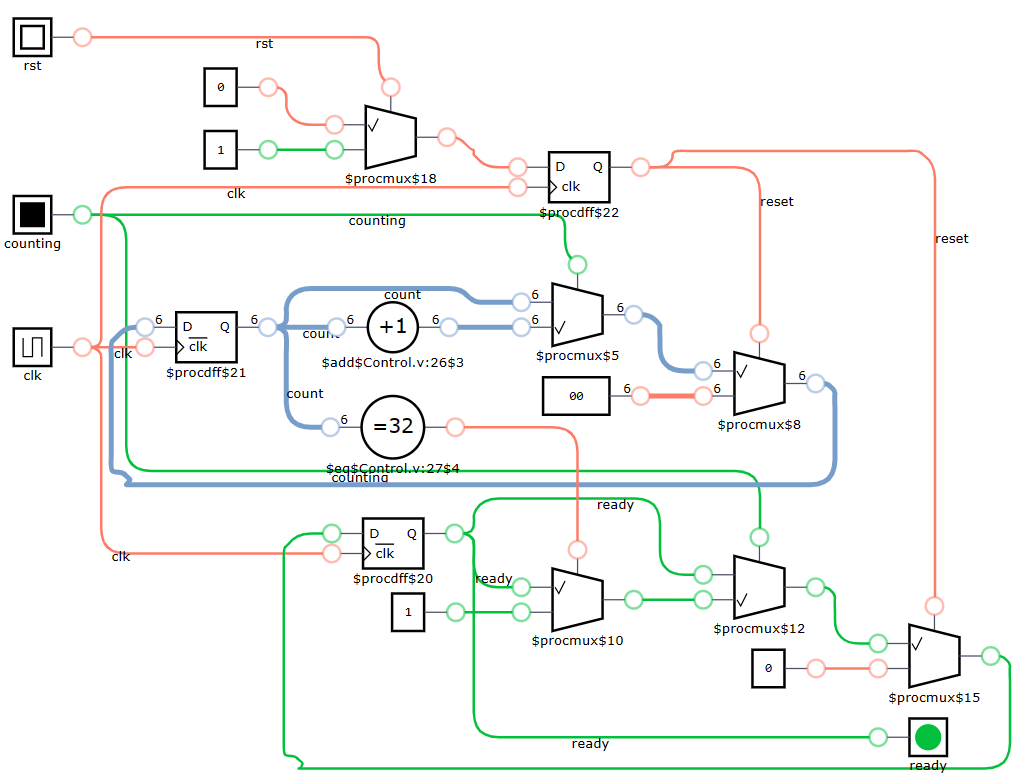
|  |  |  |
| --- | --- | --- |
| Signal Symbol | Signal Name | Signal Description |
| Dvsr\_in | Divisor in | In |
| Dvsr\_out | Divisor out | out |

Line 5

Wire connection to make out = in.

## Control





### Description

A counter with 2 in flags, 1 out flag.

### Detail

Line 1~6

I/O Interface.

|  |  |  |
| --- | --- | --- |
| Signal Symbol | Signal Name | Signal Description |
| counting | Counting flag | Input flag represent start |
| rst | Reset flag | Input flag represent reset |
| clk | Clock | Clock |
| ready | Ready flag | Signify the end of operation. |

Line 8~9

Registers

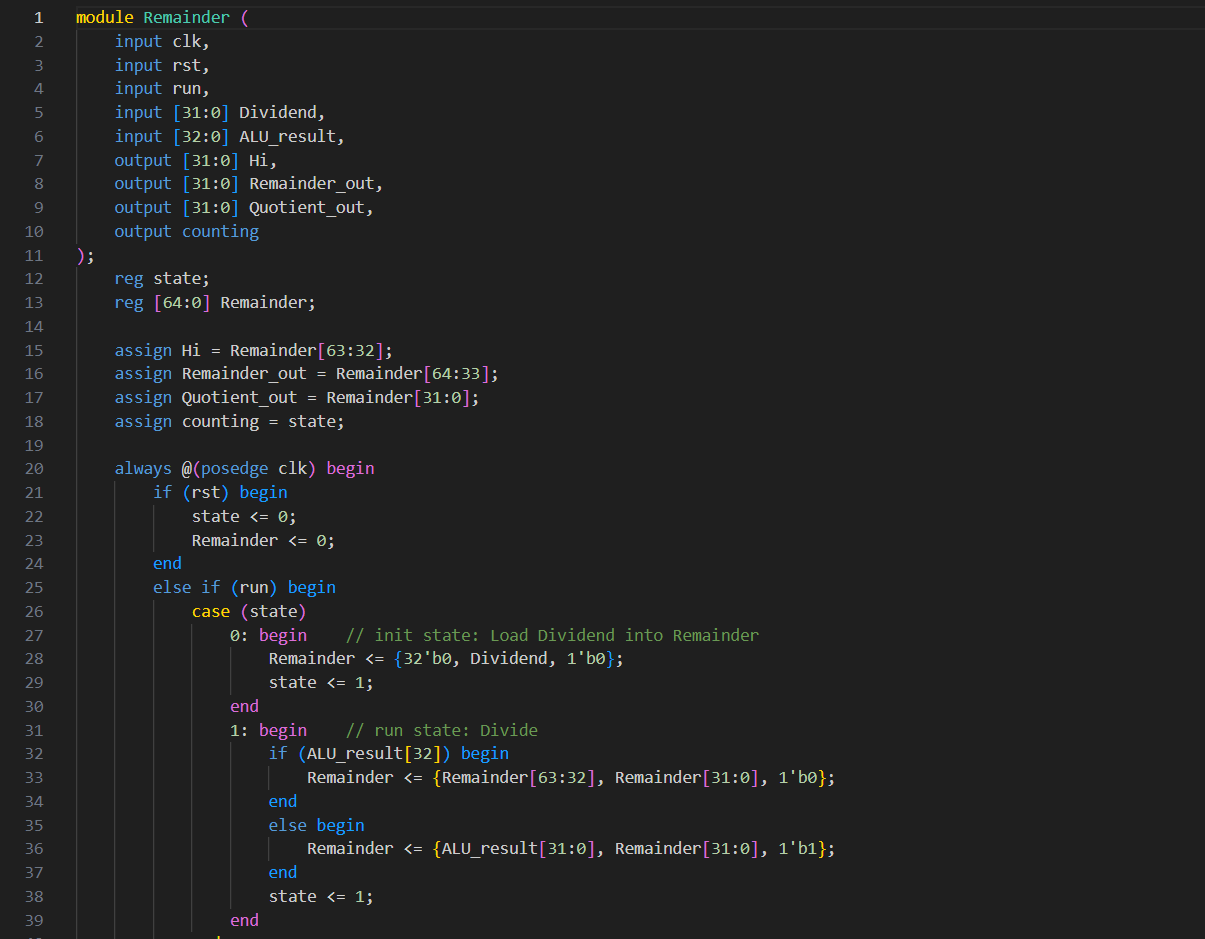
Line 11~18

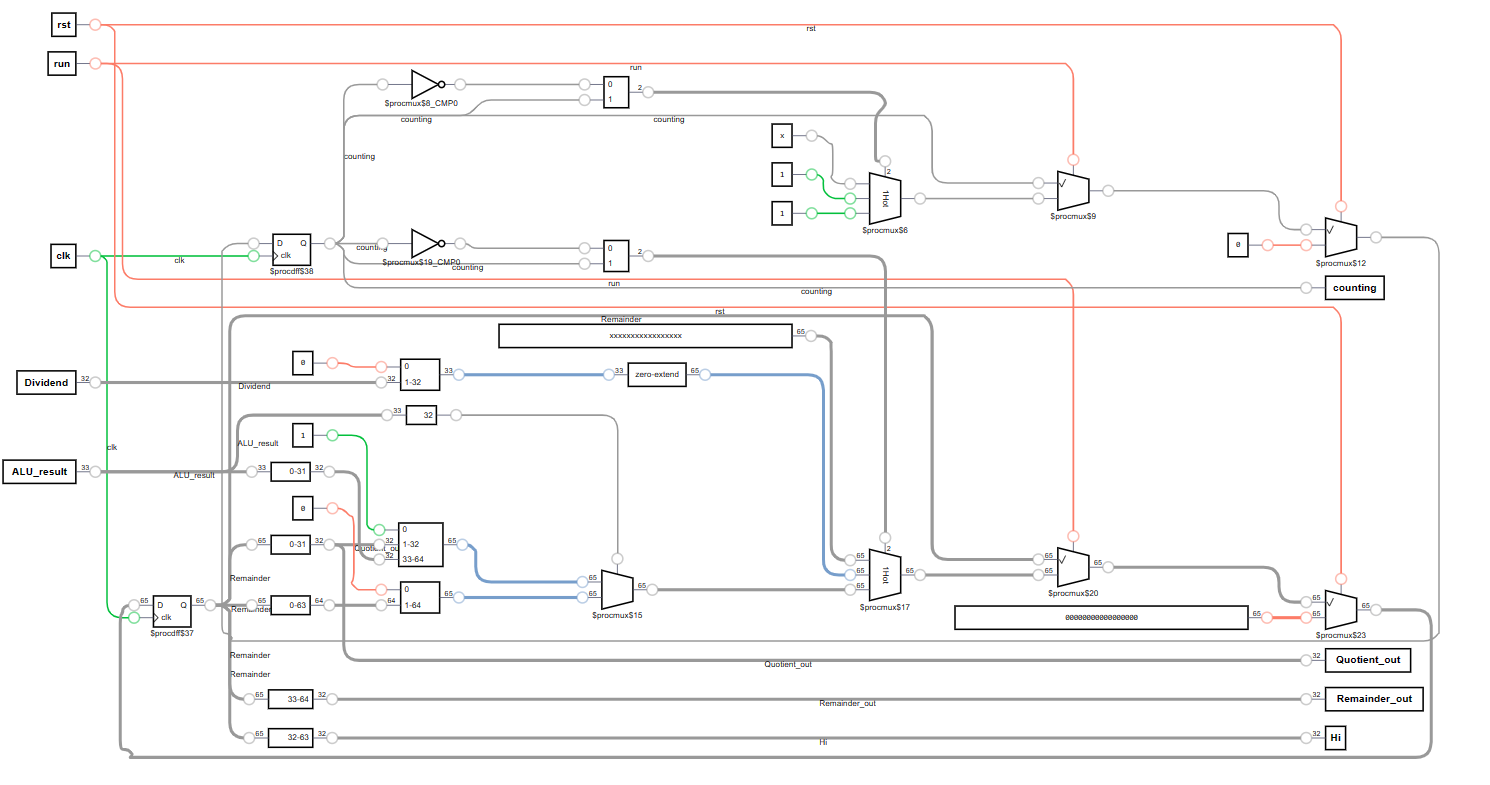
Use reset register to hold rst signal between 2 posedge clk to avoid racing.

Line 20~31

When running count 32 negedges of clk then set ready flag to notify other component to stop or fetch data.

## Remainder





### Description

Main module. I use 65 bits representing product in order to handle overflowing result from [ALU](#_ALU_1). The computation is executed on posedge clk to avoid racing condition.

### Detail

Line 1~11

I/O Interface.

|  |  |  |
| --- | --- | --- |
| Signal Symbol | Signal Name | Signal Description |
| clk | Clock | Clock |
| rst | Reset flag | Reset to init state |
| run | Run flag | Run when high |
| Dividend | 32-bits dividend | In when init |
| ALU\_result | 33-bits [ALU](#_ALU_1) result | With 1 more borrow bit |
| Hi | Hi register | Upper 32 bits |
| Remainder\_out | 32-bits Remainder | 34th bit to 65th bit to avoid the right shifting in last iteration |
| Quotient\_out | 32-bits Quotient | Output |
| counting | Counting flag | To avoid racing |

Line 12~13

Registers.

|  |  |  |
| --- | --- | --- |
| Register Symbol | Register Name | Register Description |
| state | State | For FSM |
| Remainder | 65-bits remainder | 1 more bit to handle borrow bit from [ALU](#_ALU_1) |

Line 15~18

Wire connection to [Remainder\_out](#Remainder_out), [Quotient\_out](#Quotient_out), HI, and counting.

Line 21~24

Reset.

Line 26~40

FSM

|  |  |  |
| --- | --- | --- |
| State Symbol | State Name | State Description |
| 0 | Init | Init Remainer by put Dividend to LO and shift left 1 bit. |
| 1 | run | Try to divide. |

Line 31~39

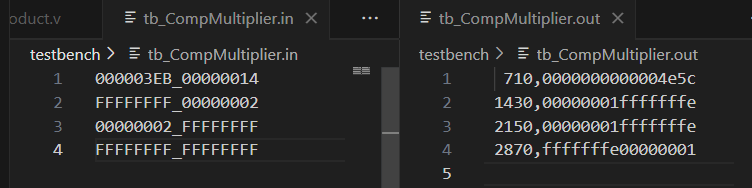
If ALU\_result is negative, shift left 1 bit.

Else, HI = ALU\_result, then shift left 1 bit. Finally set Remainder[0] to 1.

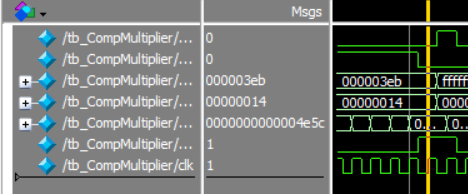
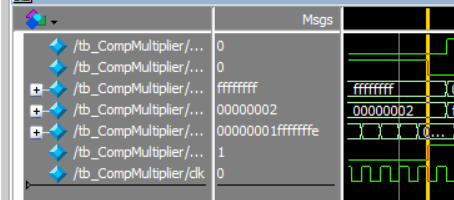
# Test

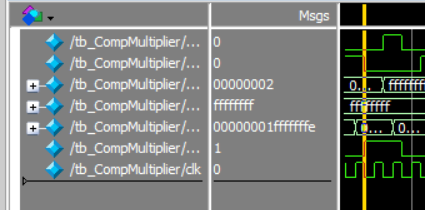
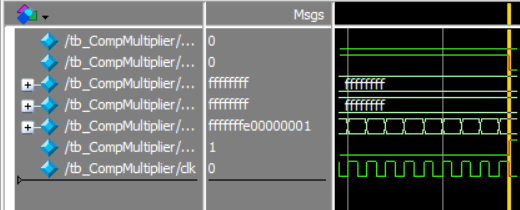
## Multiplier

### tb\_CompMultiplier

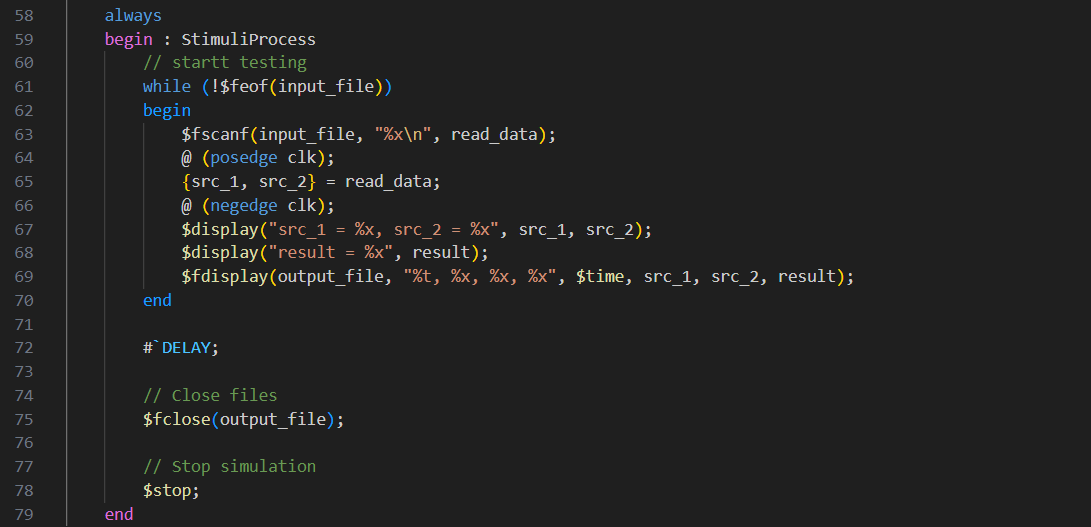


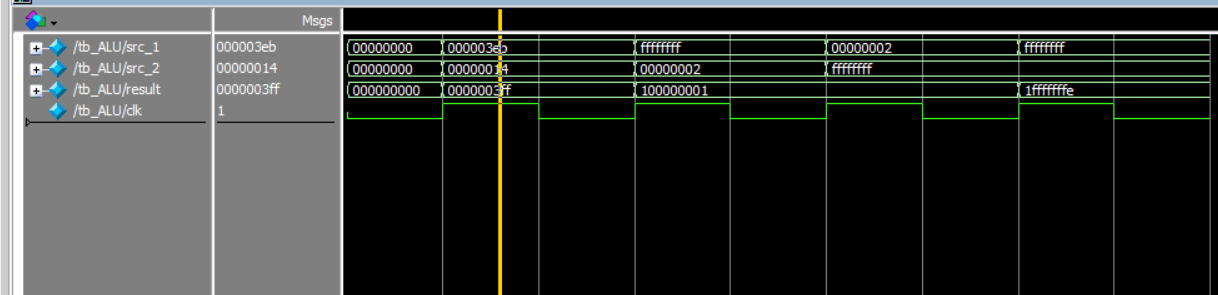
|  |  |  |
| --- | --- | --- |
| Multiplicand | Multiplier | Description |
| 000003EB | 00000014 | Common case |
| FFFFFFFF | 00000002 | Multiplicand is extreme value. |
| 00000002 | FFFFFFFF | Multiplier is extreme value. |
| FFFFFFFF | FFFFFFFF | Both are extreme value. |

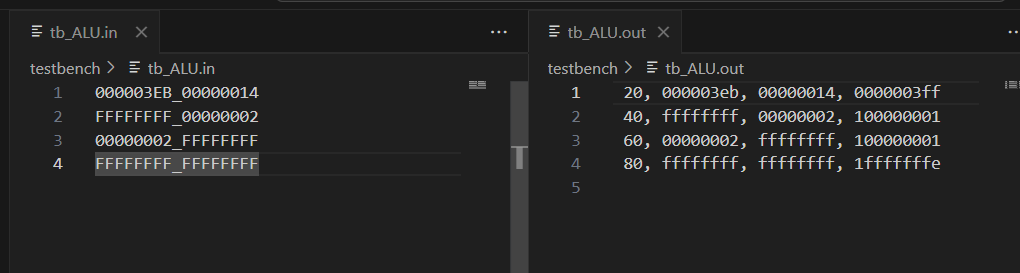
 

### tb\_ALU

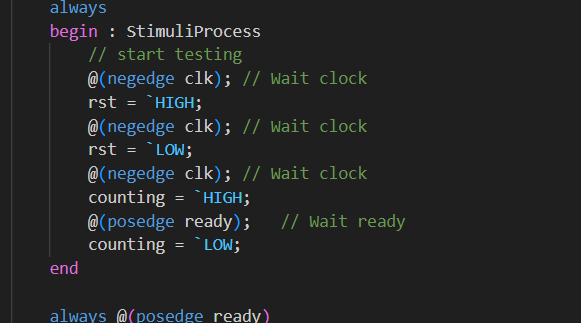


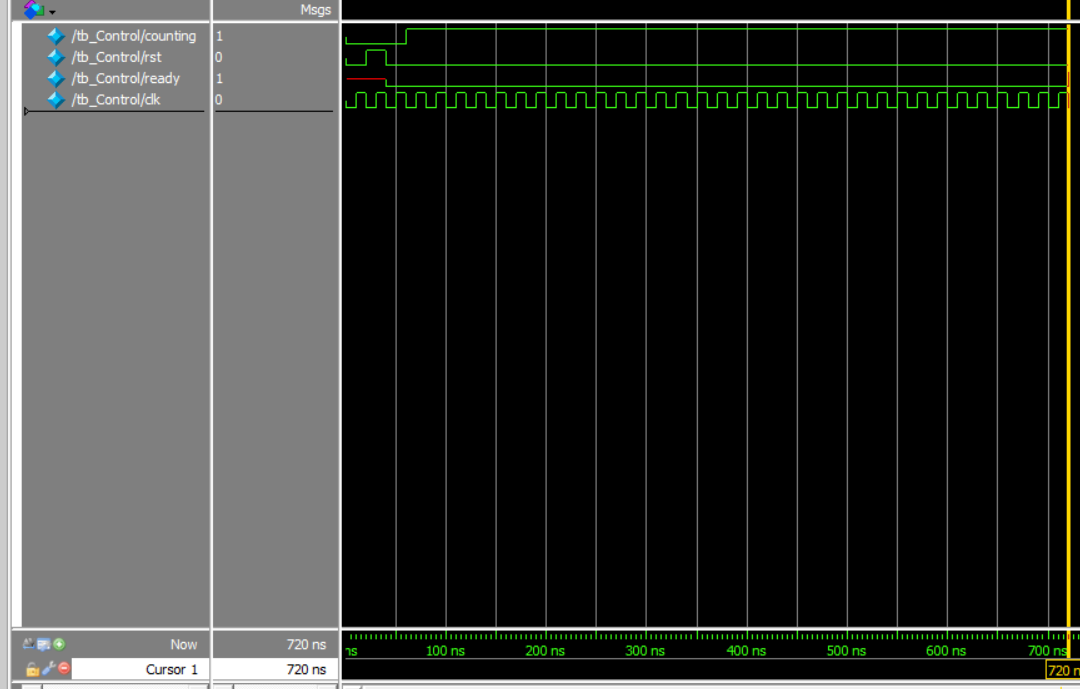




Test for functionality of ALU(only addition).

### tb\_Control

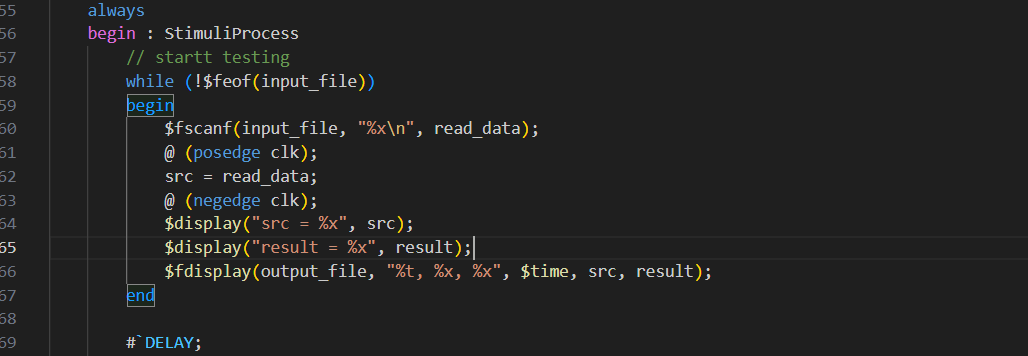




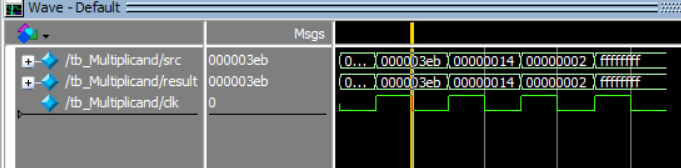
Test for functionality of Control.

(720-60)/20 = 33, 0.5 more for start and 0.5 more for end to avoid racing.

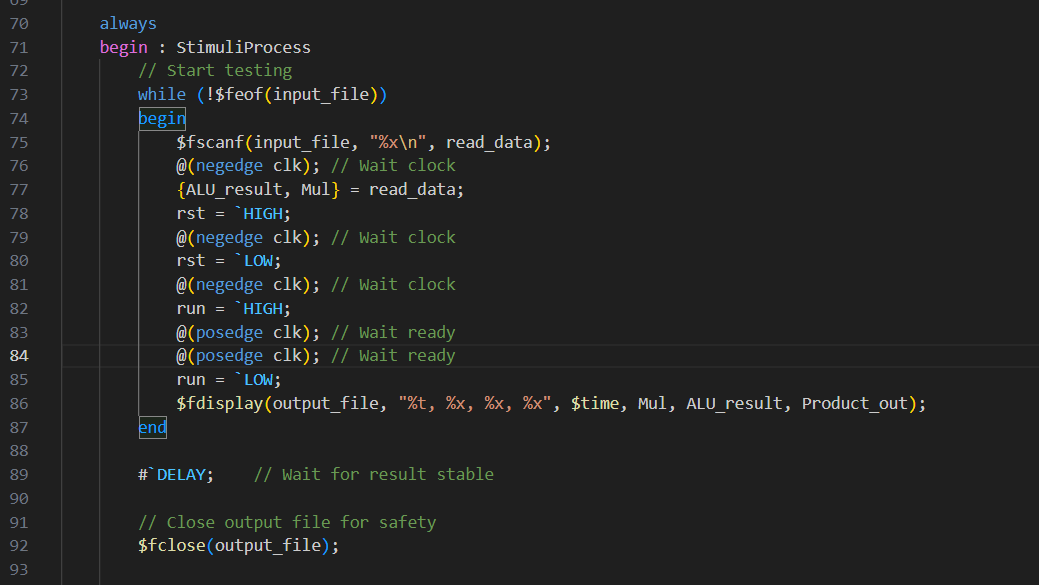
### tb\_Multiplicand



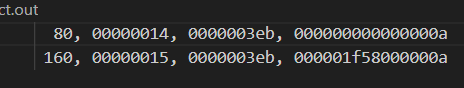
Test for the functionality of Multiplicand.

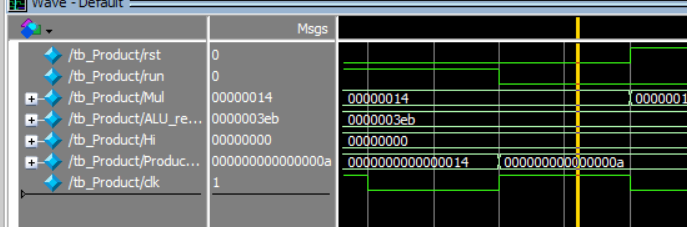


### tb\_Product

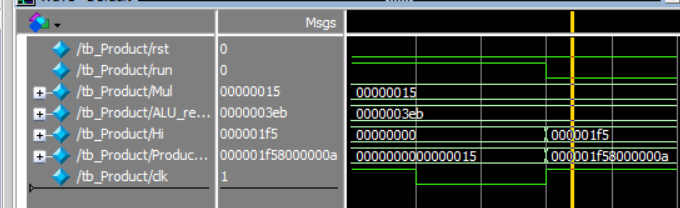


Test for functionality.





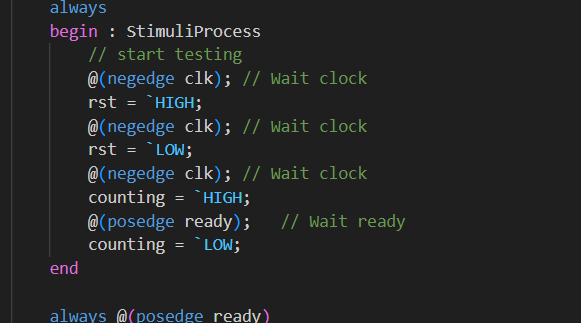
14h LSB is not 1, shift right.

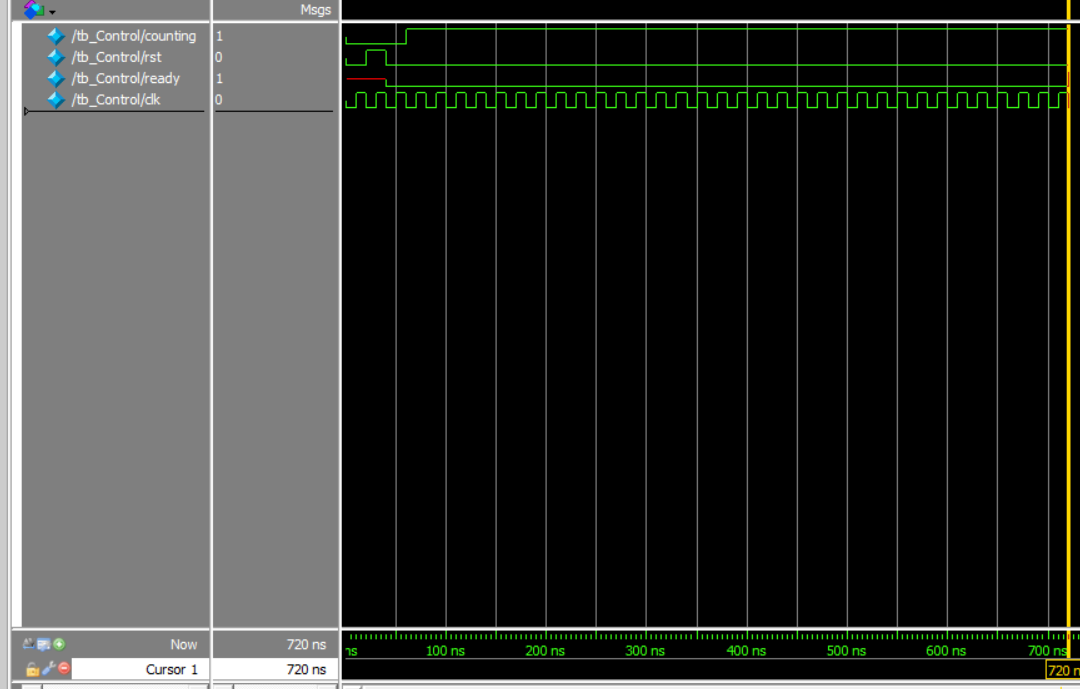


15h LSB is 1, add to HI, then shift right.

## Divider

### tb\_Control

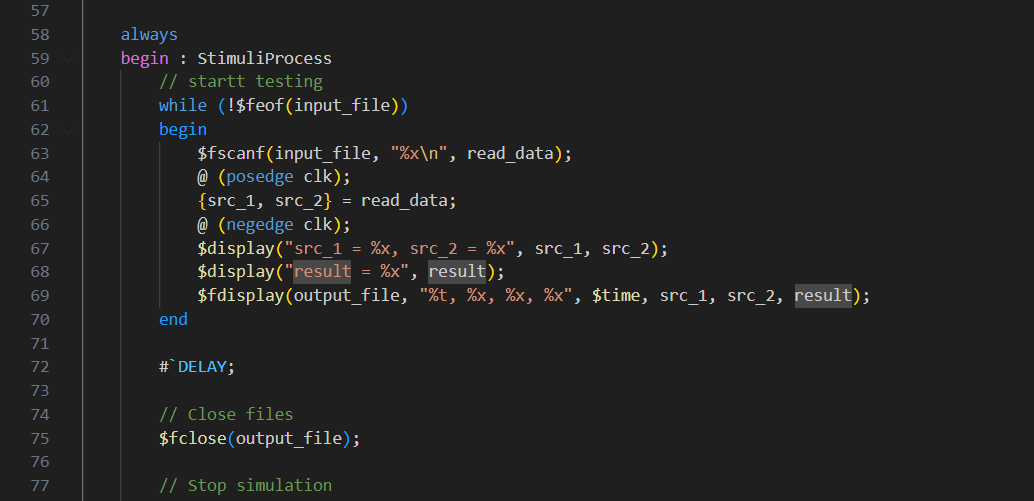




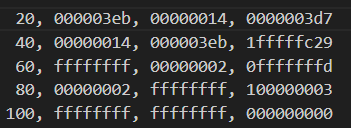
Test for functionality of Control.

(720-60)/20 = 33, 0.5 more for start and 0.5 more for end to avoid racing.

### tb\_ALU



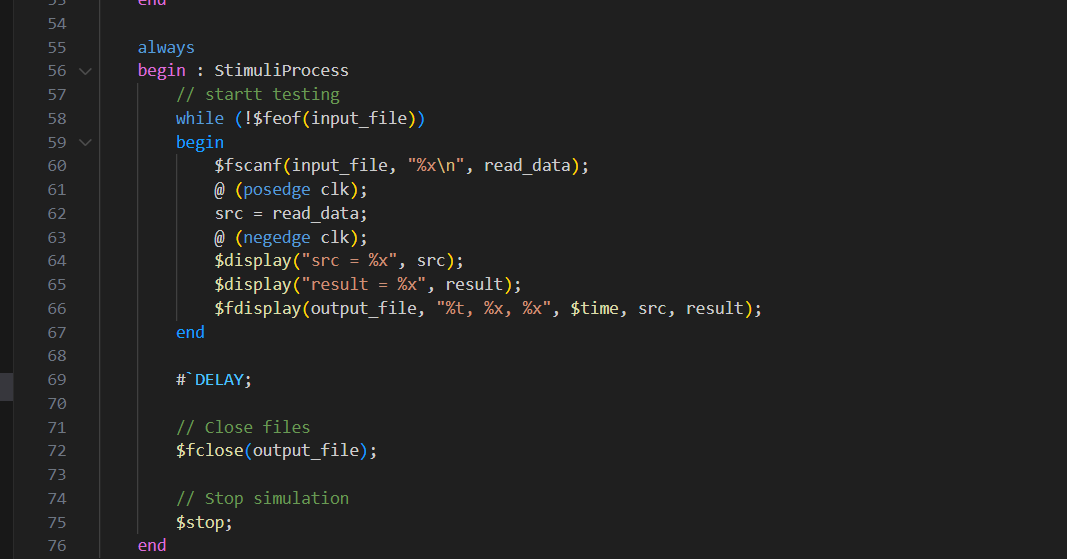
Test for functionality



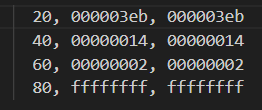


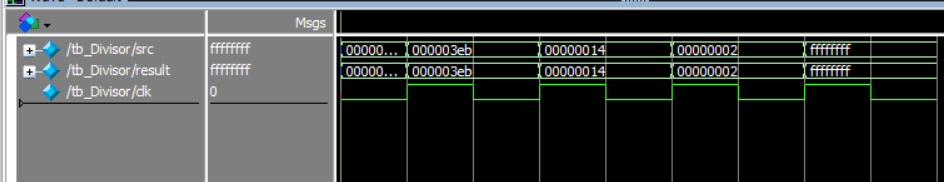
Note that result is 33-bits.

### tb\_Divisor

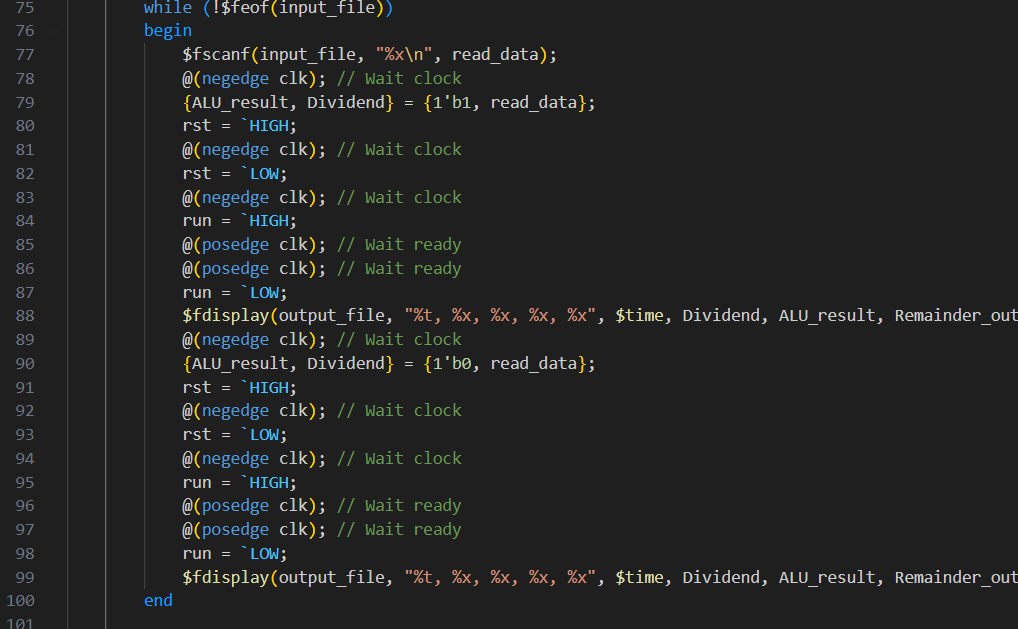


Test for functionality.

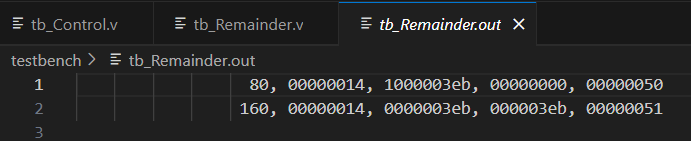


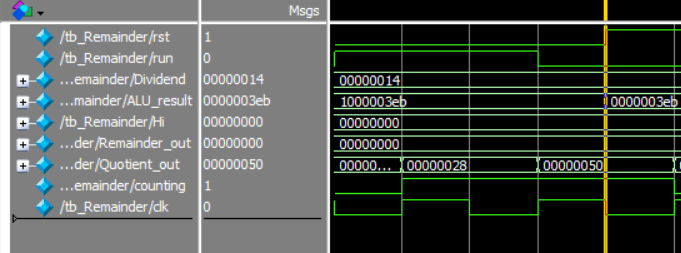


### tb\_Remainder

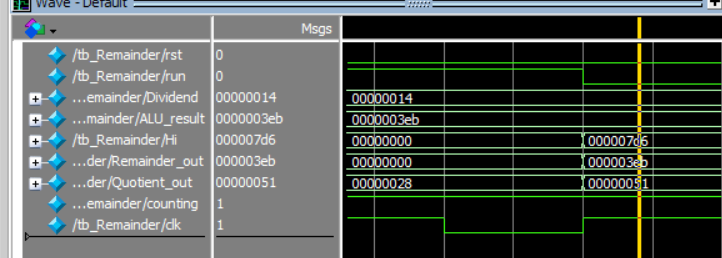


Test for functionality.



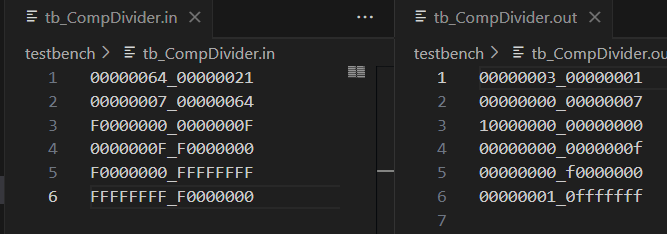


The first time ALU\_result[32] = 0 -> 28h \* 2h = 50h

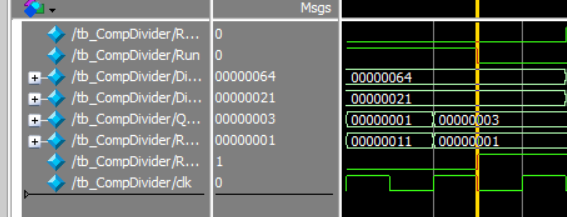
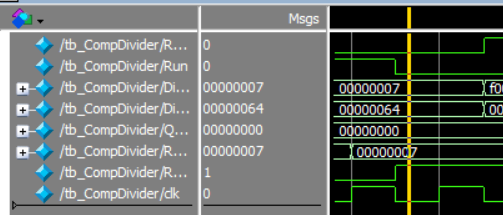


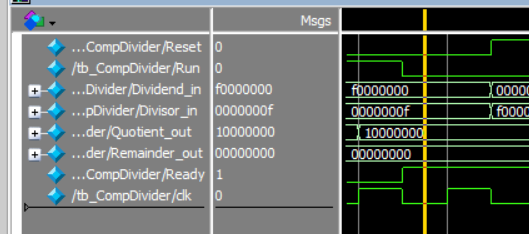
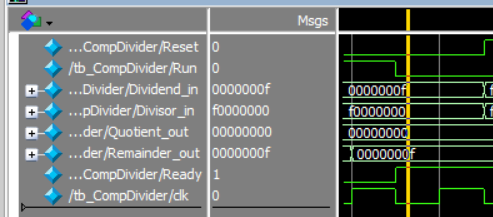
The first time ALU\_result[32] = 1 -> 28h \* 2h + 1h = 51h

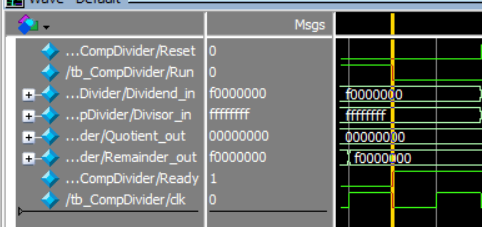
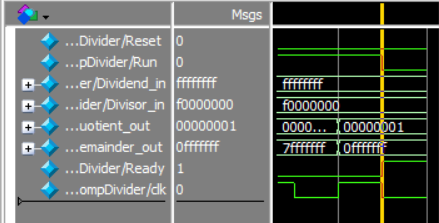
### tb\_CompDivider



|  |  |  |
| --- | --- | --- |
| Dividend | Divisor | Description |
| 00000064 | 00000021 | Common case |
| 00000007 | 00000064 | Common case |
| F0000000 | 0000000F | Dividend is extreme value |
| 0000000F | F0000000 | Divisor is extreme value |
| F0000000 | FFFFFFFF | Both are extreme |
| FFFFFFFF | F0000000 | Both are extreme |

# Conclusion and Insights

The most valuable lesson I learnt from this assignment is making report is the most tedious thing in the world. I think whom it may concern will feel the same when reading our naïve reports.

Since Run flag will be set on negedge of clk, it’ll be better if the mul/div operation is on posedge of clk because of the racing problem. Also, because of the counter should be count on negedge to make the last iteration’s output stable (setup time constraint,) this will require 1 more clk to make this happen. Considering the above observation, I make Product/Remainder to tell the control when to start counting. **This will 100% make no racing.** My structure is not the best in performance, but I think it’ll be relative stabler among all the other students’ structures in this assignment.

During the implementation, I found out that control can just sit there doing nothing but counting, so I did. It’ll need modification if considering merge multiplication and division or even merge all component to a complete ALU, but whatever. The requirement of this assignment doesn’t forbid us to do this.

And there is one more trick I used. I make multiplication/division specific ALU to further simply the control path. As a result, the control just sitting there waiting for set up flags.