Integer MULT – Project Final Report

Hua, Mengli

Li, Junlei

Metkar, Piyush Vijay

Mu, Xingyu

Vyas, Dhwani Rakesh

Xiao, Yi

Zhu, Tianyi

December 7th, 2018

CSE 7381: Computer Architecture

Southern Methodist University

Author Note

Mengli Hua, Junlei Li, Piyush Vijay Metkar, Xingyu Mu, Dhwani Rakesh Vyas, Yi Xiao and Tianyi Zhu are now students in Computer Science, Southern Methodist University.

Contact: [mhua@smu.edu](mailto:mhua@smu.edu), [junleil@smu.edu](mailto:junleil@smu.edu), [pmetkar@smu.edu](mailto:pmetkar@smu.edu), [xingyum@smu.edu](mailto:xingyum@smu.edu), [drvyas@smu.edu](mailto:drvyas@smu.edu), [yixiao@smu.edu](mailto:yixiao@smu.edu) and [zhut@smu.edu](mailto:zhut@smu.edu).

**General Project Description**

Using Verilog, expand the design of the basic ALU of our previous Verilog homework assignment to handle integer multiply operations. Add appropriate registers and other hardware as needed. Assume 32-bit two’s-complement numbers will be used, same as for the current ADD and SUB operations. Test using both positive and negative integer numbers.

### **Team Duties**

Initial Assignment was as follow:

|  |  |  |
| --- | --- | --- |
| **No.** | **Roles** | **Name** |
| **1** | Project Manager | Hua, Mengli |
| **2** | Technical Writer | Li, Junlei |
| **3** | Hardware Designer(s) | Xiao, Yi, Zhu, Tianyi |
| **4** | Software Designer(s) | Metkar, Piyush Vijay, Vyas, Dhwani Rakesh |
| **5** | Test Engineer | Mu, Xingyu |

But during the project course, some schedules, duties and workloads had to be changed to meet our project specific requirements which have been covered and mentioned in the following sections individually.

Project Manager 🡪 Mengli Hua:

As the manager of this project, I will make the tasks plan with milestone and deliverable for the whole team. Also, I will always pay attention on the project processing to make sure that everyone is catching up to the schedule by complete their specific tasks on time. In addition, I will help the writer to notify other designers to give the information he needs to complete all the documents. The last specific duty of mine is to communicate with “upper management” and let all the members know the information given by Dr. Manikas.

Technical Writer 🡪 Junlei Li:

The main duty for a technical writer is to develop proposal, report drafts, including final draft. But due to changes in schedules and workloads, Dhwani Vyas and Piyush Metkar were involved with me during the writing of Final Report of the project. I would also pay more attention to how to deliver the information in a clear, concise and correct way to reader. I choose APA Formatting and Style guide (Purdue Online Writing Lab) as format to write this paper. I would also double check every teammates’ paper before adding them to reports, making sure the information they deliver is correct.

Hardware Designers 🡪 Yi Xiao and Tianyi Zhu:

As a hardware designer, I will develop the Verilog code and handle the component designs for the project. I will also write clear comments in my code to make it easier for teammates to understand and develop tests.

Software Designers 🡪 Dhwani Vyas and Piyush Metkar:

As a software designer, we were in charge of designing and developing the MIPS code to implement and run the Integer MULT operations on hardware. But during the project implementation, we later realized that our project is not Instruction based as we need just one instruction to be taken care of that is: MULT, which has been implemented by the hardware designers using Verilog and has been tested by the Test Engineer using test bench code. The requirements of the project are met using the hardware code itself i.e. integer multiply operations of 32-bit two’s-complement numbers, using both positive and negative integer numbers. Hence, we did not invest time in developing MIPS software code, instead used the Software designers: Metkar, Piyush Vijay and Vyas, Dhwani Rakesh for writing the Final Report along with the Technical Writer.

Test Engineer 🡪 Xingyu Mu:

Test is a very important part to ensure the reliability of the code we design. Test Engineer will do the test after software and hardware design completed, which means it is the last part of coding work. So, it is necessary to remind the schedule of the whole project as early as possible and other members’ work process. To test the validity of their MIPS and Verilog code. The main duty is to design methods and codes to test MIPS and Verilog code respectively. In MIPS field, initial plan is to directly assume values with the MIPS code to ensure its correctness. On the other hand, Verilog testing needs to create testbenches to verify the design functionality of hardware design, and we can see the result in the Verilog console. After coding work done, test engineer should give suggestions and take measures to correct or improve the functionality of the software and hardware code.

**Design Implementation and Testing**

In this part, we are going to introduce the similar Algorithm thinking we find, and we will also post the relevant code in different programming languages. In order to deeply understand the working principle of hardware, it is very necessary to include the logic circuit, and specific detail about it. After that, we will give the specific design. According to the topic we choose, we find MULT is not instruction based, we only need one instruction. Thus, we don’t have MARS code for developing and testing our design.

**Design Implementation**

Algorithm thinking： Shift addition

The algorithm can be divided into three parts: Firstly, multiply the binary multiply and each bit of the multiplier to obtain the partial product which has the same number of digits as the multiplier. Secondly, according to the weighted value, we should add the partial product and the partial product compression. Lastly, a super forward bit adder module is usually used. The design of the multiplier is naturally concentrated on the performance optimization of each module to achieve the purpose of fast speed and small area.

Here are the relevant sample codes:

**High-level algorithm:**

**Python:**

def multiply ( multiplicand, multiplier ):

result = 0

for i in range(32):

if multiplier & 0x1 == 1: // if the i-th bit of multiplier is 1, then add multiplicand to result

result += multiplicand

multiplicand = multiplicand << 1 // multiplicand shift left 1 bit

multiplier = multiplier >> 1 // multiplier shift right 1 bit

return result

C:

signed int multiply( signed int multiplier, signed int multiplicand ) {

signed int product = 0;

while (multiplier != 0) {

if ((multiplier & 1) != 0) {

product = product + multiplicand;

}

multiplier = multiplier >> 1;

multiplicand = multiplicand << 1;

}

return product;

}

**Hardware Components used:**

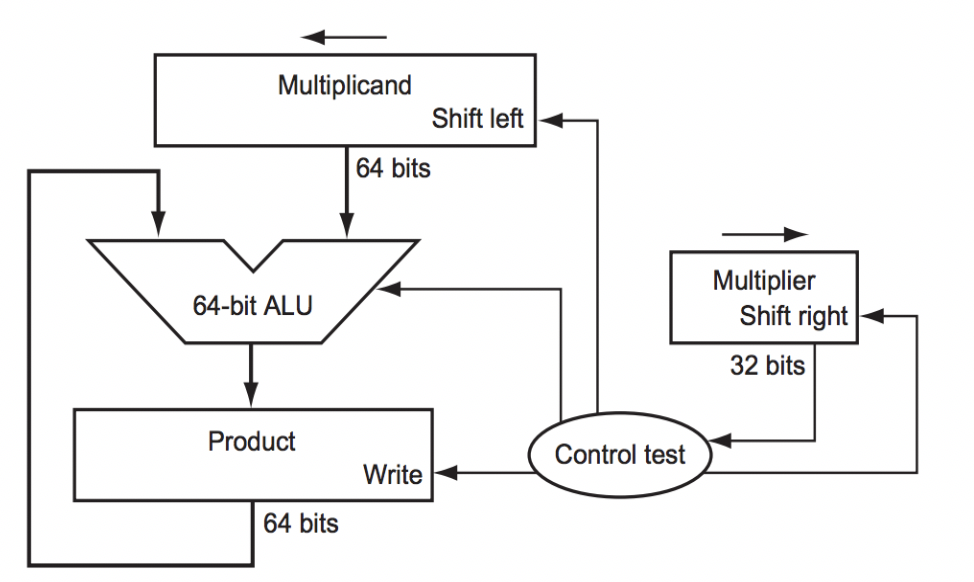


Figure 1. First version of the multiplication hardware

According to the Figure 1(please see Appendix A for first version of hardware), we can understand how the first version of hardware works, the hardware had been drawn to resemble the method of paper-and-pencil, which shows that the data flows from top to bottom. In order to understand this method clearly, we assume the multiplier is in the register with the size of 32-bits, and the 64-bit product register is initialized to 0.

There is no doubt that we should move the multiplicand from right to left for one digit in each step, since it has likely to be added to the intermediate products. A multiplicand with 32 bits would move from right to left when it passes over 32 steps, in this case, a Multiplicand register with 64 bits is needed, and this Multiplicand register should be initialized with 32 bits in right side and 0 bit in the other side. After that, this register is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 64-bit Product register (Patterson, D. A., Hennessy, J. L,2015).

As Figure 2 mentioned (Please see Appendix B for the first Multiplication Algorithm), we can find there are three basic steps needed for each bit. The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register. The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying with paper and pencil (Koren. 2010).

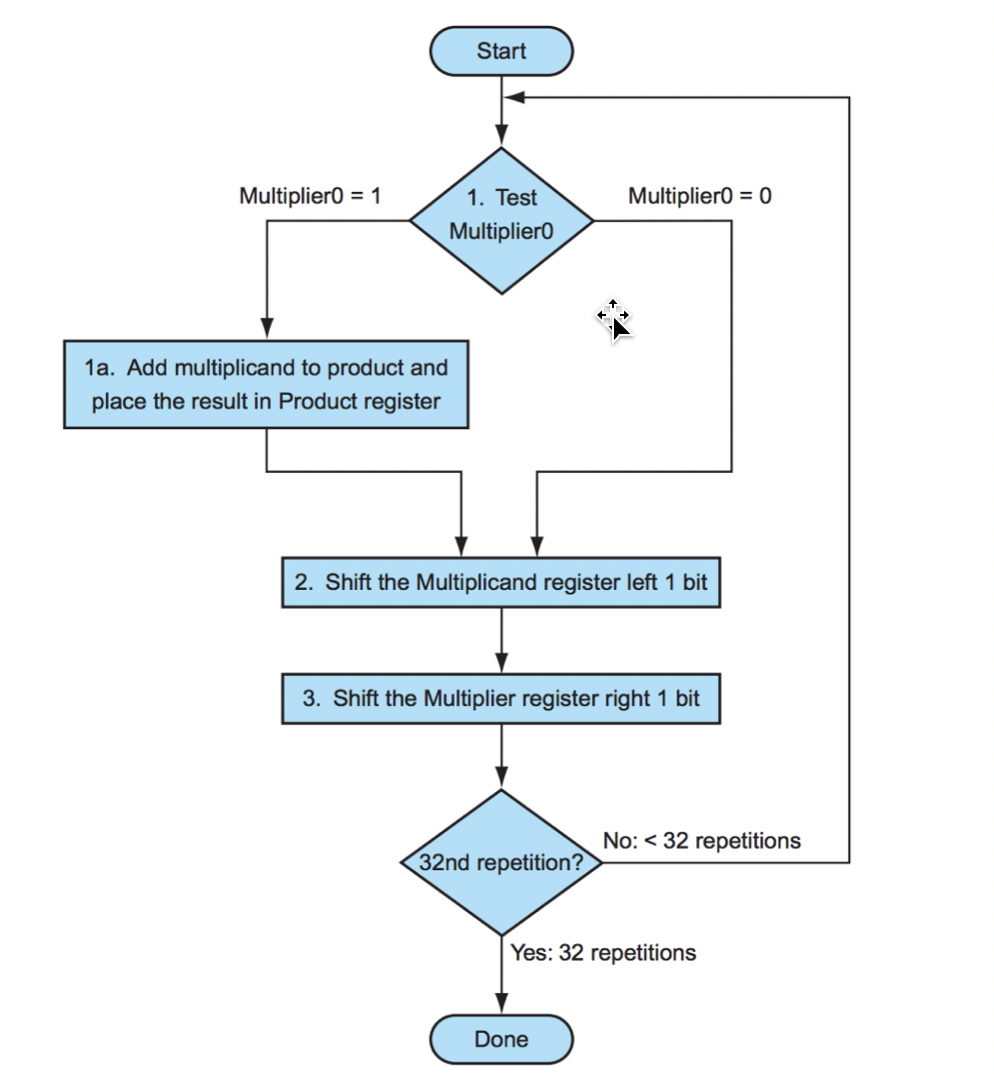
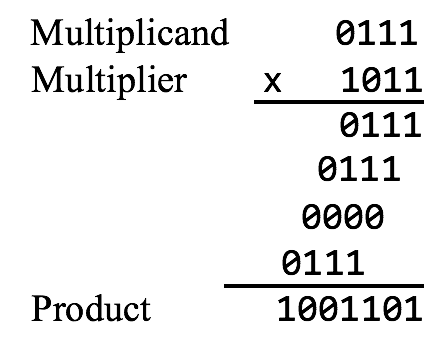


Figure 2. The first Multiplication Algorithm

**Method to handle signed integers (Positive integers and Negative integers):**

Because the numbers we accept are two’s complement numbers, in order to handle multiplication for both positive and negative integer numbers, we reverse the sign of negative numbers by taking the two’s complement operation for their binary representation, then we get the positive values for the negative integer numbers. Therefore, we can continue doing the multiplication by using shift-add method, which was discussed previously. In the last step, we take the XOR operation on the leading bit of the original values of both inputs to determine if the sign for the final product is positive or negative. That is, if one input has a leading bit with “0” and another input has a leading bit with “1”, after taking XOR operation, we will get “1” as the leading bit for the final product which represents a negative number. If both of them have the same leading bit “0” or the same leading bit “1”, we will get “0” as the leading bit for the final product which represents a positive number. Because the multiplication used two positive values, we will definitely get a positive temporary result. We will need to take the two’s complement operation for the temporary result to make it negative to get the correct final product if we determined that the final product should be a negative number after the XOR operation taken before. Otherwise, the final product would be the temporary result.

**Example for the shift-add multiplication for unsigned integers: (7\*11=77)**

****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Step | Product | Multiplier | Multiplicand | Operation |
| 0 | 0000 0000 | 1011 | 0000 0111 | Initialization |
| 1 | 0000 0111  0000 0111  0000 0111 | 1011  1011  0101 | 0000 0111  0000 1110  0000 1110 | Add Multiplicand to Product  Shift left Multiplicand 1 bit  Shift right Multiplier 1 bit |
| 2 | 0001 0101  0001 0101  0001 0101 | 0101  0101  0010 | 0000 1110  0001 1100  0001 1100 | Add Multiplicand to Product  Shift left Multiplicand 1 bit  Shift right Multiplier 1 bit |
| 3 | 0001 0101  0001 0101 | 0010  0001 | 0011 1000  0011 1000 | Shift left Multiplicand 1 bit  Shift right Multiplier 1 bit |
| 4 | 0100 1101  0100 1101  0100 1101 | 0001  0001  0000 | 0011 1000  0111 0000  0111 0000 | Add Multiplicand to Product  Shift left Multiplicand 1 bit  Shift right Multiplier 1 bit |

The example above shows the processes of shift-add multiplication for unsigned integers. In this example, we accept two 4-bits positive integer numbers which are 7(0111) as the multiplicand and 11(1011) as the multiplier and finally get an 8-bits product of 77(0100 1101). The reason that the multiplicand register has 8 bits here is because it needs to move from right to left when it passes over 4 steps as discussed before. In each step, if the last bit of the multiplier is 1, we need to add the current multiplicand to the current product. Then we shift the multiplicand left 1 bit and shift the multiplier right 1 bit. Otherwise, if the last bit of the multiplier is 0, we only need to shift the multiplicand left 1 bit and shift the multiplier right 1 bit. When the value of multiplier become 0, we are done.

## Code List

1. multiply.v (the hardware code)

2. multiply\_testbench.v (the testbench code for the hardware)

## Specific Implementation for Hardware Design:

We have designed a module “multiply” with 5 arguments, namely the two input arguments are multiplier and multiplicand which basically would be the 32-bit two’s complement numbers that are supposed to be multiplied by our module and the third input argument is clk for clock input of the design. The product of the two 32 bits numbers would be a 64-bit number which will be the output argument of the module named as product and the last output argument is a state flag, ready.

We are also creating registers of corresponding size for multiplier, multiplicand and product for temporary purpose as multiplier\_copy, multiplicand\_copy and product\_temp respectively.

isNeg register is basically used for determining whether the number is positive or negative by XORing the leftmost bit of multiplier and multiplicand. Bit Register is used for each bit and initialized to zero. As ours is a positive edge triggered design, the block of code will execute only on positive edges with respect to clock.

We have two conditions here, if and else. If condition runs first checking for the product’s sign and else condition is taking care of the multiplication bit by bit (1 bit is taken care in each cycle which is run 32 times for 32-bit multiplicand and multiplier)Using if else condition we have implemented that whenever the bit is greater than 0 i.e. till the last bit (it will run 32 times), the else portion explained below will execute and once the bit equals 0 i.e. all bits are over, if condition explained below will execute

If block explained🡪

Bit registered is initialized with a decimal of 6 bit as 32 as we are wanting to run the code ofr 32 times for each bit of 32-bit number and product and product\_temp registers are initialized to 0. Then we are getting the absolute value of multiplicand. This will be 32 bit so left part of multiplicand\_copy would be needed to be added with 0s as it is 64 bits.

Similarly, we are getting the absolute value of multiplier. We will check for the leftmost bit of the multiplier, if it’s 1, which mean it’s a negative number. multiplier\_copy will equal the 2’s complement of multiplier. Both the above conditions are checked using ternary conditional operators where 2’s complement number is turned into its absolute form making it ready for multiplication. Both the multiplier and multiplicand are XORed for checking the sign of the number as positive or negative.

Else block explained🡪

Whenever the rightmost bit of current multiplier\_copy is 1 which is checked using if condition, in that scenario the current multiplicand\_copy is added to product\_temp. If the number is negative that is if isNeg register equals 1 which basically means that the product of both numbers should be a negative number, in that case product register is equal to the 2’s complement of product\_temp. But when the product of both numbers should be a positive number that is when isNeg register=0, in that case product is equal to product\_temp.

In case of negative product, we are converting it to 2’s complement form but in case of positive product we are not making any changes to the product value. This again is checked using ternary conditional operator. Then we are shifting multiplier right by 1 bit and multiplicand left by 1 bit making it ready for the next cycle which will execute on positive edge of clock. We are reducing the bit by 1 in each cycle for the block to run for 32 times.

**Commented Verilog Code for our project is as follows:**

module multiply(ready,product,multiplier,multiplicand,clk);

input clk;

input [31:0] multiplier, multiplicand; // The inputs should be twos complement numbers

output [63:0] product;

output ready;

reg [63:0] product, product\_temp;

reg [31:0] multiplier\_copy; // Temporary registers for the multiply

reg [63:0] multiplicand\_copy;

reg isNeg; // Determine the sign by XOR the leftmost bit of multiplier and multiplicand

reg [5:0] bit; // For repeating 32 times

wire ready = !bit;

// Initialization

initial bit <= 0;

initial isNeg <= 0;

always @( posedge clk )

if( ready ) begin

bit <= 6'd32;

product <= 0;

product\_temp <= 0;

// Get the absolute value of multiplicand. Add 0 to the

// left half because multiplicand\_copy is 64 bits long.

multiplicand\_copy <= multiplicand[31] ?

{ 32'd0, ~multiplicand + 1'b1 } :

{ 32'd0, multiplicand };

// Get the absolute value of multiplier. If the leftmost bit

// of multiplier is 1 which represent a negative number,

// multiplier\_copy equals to the twos complement of multiplier.

multiplier\_copy <= multiplier[31] ? (~multiplier + 1'b1) : multiplier;

// XOR operation to determine the sign of the product

isNeg <= multiplicand[31] ^ multiplier[31];

end

else if ( bit > 0 ) begin

// If the rightmost bit of current multiplier\_copy is 1,

// then add the current multiplicand\_copy to product\_temp

if( multiplier\_copy[0] == 1'b1 )

product\_temp <= product\_temp + multiplicand\_copy;

// If isNeg = 1, which means that the product should be

// a negative number, then product equals to the

// twos complement of product\_temp. If isNeg = 0, product

// equals to product\_temp.

product <= isNeg ? (~product\_temp + 1'b1) : product\_temp;

// Shift right multiplier 1 bit

multiplier\_copy <= multiplier\_copy >> 1;

// Shift left multiplicand 1 bit

multiplicand\_copy <= multiplicand\_copy << 1;

bit <= bit - 1'b1; // repeat 32 times

end

endmodule

**Test Plan:**

After all other works done, which means the hardware and software designs are ready. Then the test can be processing with the MARS and Verilog, to ensure the Verilog and MIPS code can run correctly, respectively.

To test MIPS code, we can give values directly to the code, and generate results to identify whether the code is successful. If failed, by a list of test set, it seems possible to have the solution to correct the code.

For the Verilog code, we can create a new Verilog file as the testbench. The function of a testbench is to apply inputs to the design while testing, and then produce the outputs in a readable and user-friendly format. All the Verilog we plan on using in our hardware design must be synthesizable, meaning it has a hardware equivalent. The Verilog we write in a test bench does not need to be synthesizable because we will only ever simulate it (Justin Rajewski, January 10, 2018).

After tests has done, we can use the test results to see whether the hardware and software design is valid and try to find the way to correct or improve them.

**Test Cases for Design:**

We have identified the following test cases to ensure that the code runs properly for all the possible scenarios. The test cases are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Case# | Multiplier | Multiplicand | Expected Result | Actually Result |
| 1 | Positive | Positive | Positive | Positive |
| 2 | Positive | Negative | Negative | Negative |
| 3 | Negative | Positive | Negative | Negative |
| 4 | Negative | Negative | Positive | Positive |
| 5 | Zero | Non-zero | Zero | Zero |
| 6 | Zero | Zero | Zero | Zero |
| 7 | 32-bit highest positive | 32-bit highest positive | 64-bit (No overflow) | 64-bit (No overflow) |

Based on the test case design above, we created the testbench as following:

Separate Code:

begin

// case1 (Positive\*Positive): 3 \* 20 using complement code

multiplier = 32'b00000000000000000000000000000011;

multiplicand = 32'b00000000000000000000000000010100;

#90

$display($time, " multiplier = %b, multiplicand = %b, product=%b", multiplier, multiplicand, product);

// product should be 0000000000000000000000000000000000000000000000000000000000111100 = 60

// case2 (Positive\*Negative): 3 \* -2

multiplier = 32'b00000000000000000000000000000011;

multiplicand = 32'b11111111111111111111111111111110;

#90

$display($time, " multiplier = %b, multiplicand = %b, product=%b", multiplier, multiplicand, product);

// product should be 1111111111111111111111111111111111111111111111111111111111111010 = -6

// case3 (Negative\*Positive)-2 \* 1

multiplier = 32'b11111111111111111111111111111110;

multiplicand = 32'b00000000000000000000000000000001;

#99

$display($time, " multiplier = %b, multiplicand = %b, product=%b", multiplier, multiplicand, product);

// product should be 1111111111111111111111111111111111111111111111111111111111111110 = -2

// case4 (Negative\*Negative): -1 \* -23

multiplier = 32'b11111111111111111111111111111111;

multiplicand = 32'b11111111111111111111111111101001;

#90

$display($time, " multiplier = %b, multiplicand = %b, product=%b", multiplier, multiplicand, product);

// product should be 0000000000000000000000000000000000000000000000000000000000010111 = 23

// case5 (Positive\*Zero): 1 \* 0

multiplier = 32'b00000000000000000000000000000001;

multiplicand = 32'b00000000000000000000000000000000;

#90

$display($time, " multiplier = %b, multiplicand = %b, product=%b", multiplier, multiplicand, product);

// product should be 0000000000000000000000000000000000000000000000000000000000000000 = 0

// case6 (Negative\*Zero): -2 \* 0

multiplier = 32'b11111111111111111111111111111110;

multiplicand = 32'b00000000000000000000000000000000;

#90

$display($time, " multiplier = %b, multiplicand = %b, product=%b", multiplier, multiplicand, product);

// product should be 0000000000000000000000000000000000000000000000000000000000000000 = 0

// case7 (Zero\*Zero): 0 \* 0

multiplier = 32'b00000000000000000000000000000000;

multiplicand = 32'b00000000000000000000000000000000;

#90

$display($time, " multiplier = %b, multiplicand = %b, product=%b", multiplier, multiplicand, product);

// product should be 0000000000000000000000000000000000000000000000000000000000000000 = 0

// case8 (Zero\*Positive): 0 \* 2

multiplier = 32'b00000000000000000000000000000000;

multiplicand = 32'b00000000000000000000000000000010;

#90

$display($time, " multiplier = %b, multiplicand = %b, product=%b", multiplier, multiplicand, product);

// product should be 0000000000000000000000000000000000000000000000000000000000000000 = 0

// case9 (Zero\*Negative): 0 \* -1

multiplier = 32'b00000000000000000000000000000000;

multiplicand = 32'b11111111111111111111111111111111;

#90

$display($time, " multiplier = %b, multiplicand = %b, product=%b", multiplier, multiplicand, product);

// product should be 0000000000000000000000000000000000000000000000000000000000000000 = 0

// case10 (32 bit-highest\*32 bit-highest): 2147483647 \* 2147483647

multiplier = 32'b01111111111111111111111111111111;

multiplicand = 32'b01111111111111111111111111111111;

#500

$display($time, " multiplier = %b, multiplicand = %b, product=%b", multiplier, multiplicand, product);

// product should be 0011111111111111111111111111111100000000000000000000000000000001 = 4611686014132420609

$stop;

The output of this testbench can be showing as following:

91 multiplier = 00000000000000000000000000000011, multiplicand = 00000000000000000000000000010100, product=0000000000000000000000000000000000000000000000000000000000111100

181 multiplier = 00000000000000000000000000000011, multiplicand = 11111111111111111111111111111110, product=1111111111111111111111111111111111111111111111111111111111111010

280 multiplier = 11111111111111111111111111111110, multiplicand = 00000000000000000000000000000001, product=1111111111111111111111111111111111111111111111111111111111111110

370 multiplier = 11111111111111111111111111111111, multiplicand = 11111111111111111111111111101001, product=0000000000000000000000000000000000000000000000000000000000010111

460 multiplier = 00000000000000000000000000000001, multiplicand = 00000000000000000000000000000000, product=0000000000000000000000000000000000000000000000000000000000000000

550 multiplier = 11111111111111111111111111111110, multiplicand = 00000000000000000000000000000000, product=0000000000000000000000000000000000000000000000000000000000000000

640 multiplier = 00000000000000000000000000000000, multiplicand = 00000000000000000000000000000000, product=0000000000000000000000000000000000000000000000000000000000000000

730 multiplier = 00000000000000000000000000000000, multiplicand = 00000000000000000000000000000010, product=0000000000000000000000000000000000000000000000000000000000000000

820 multiplier = 00000000000000000000000000000000, multiplicand = 11111111111111111111111111111111, product=0000000000000000000000000000000000000000000000000000000000000000

1320 multiplier = 01111111111111111111111111111111, multiplicand = 01111111111111111111111111111111, product=0011111111111111111111111111111100000000000000000000000000000001

**Code Files and Information**

**Multiply.v**

Multiply.v contains the Verilog code to process 32-bit multiplication of integer. The documentation of the file and contents are as follows:

We are using 2 32-bit registers, multiplier & multiplicand, to store the 2 operands, multiplier and multiplicand. The product of the two operands will be stored in a 64-bit registers, product & product\_temp, to avoid data overflow. Both the operands are copied into 32 and 64-bit registers, multiplier\_copy & multiplicand\_copy, to be used in algorithm.

isNeg register is used to store the sign of the operands. This is done by taking the XOR of leftmost bits of the operands. 5-bit Bit register is used to loop 32 times (25).

The algorithm begins by initialiazing bit to 32 and isNeg to zero. The left half of the multiplicand\_copy register is set to zero bits because it’s size is 64 bit while the operand is 32-bit. Then the absolute value of multiplicand is copied into right half of the multiplicand\_copy register.

If the leftmost bit of multiplier is 1, it indicates the operand is negative. In this case we take two’s complement of the multiplier and copy it into the multiplier\_copy register. However, if the operand is positive, we just copy the absolute value in the multiplier\_copy register.

The sign of the multiplicand is determined by XOR on leftmost bit and the isNeg register is update accordingly.

Now we begin the for loop which has condition the bit is greater than zero. Following algorithm is repeated for 32 times using the bit buffer:

1. If the rightmost bit of current multiplier\_copy is 1 then add the current multiplicand\_copy to product\_temp.
2. If isNeg is 1, the product needs to be negative. Take two’s complement of product\_temp register.
3. Right shift the multiplier\_copy register by 1 bit.
4. Left shift the multiplicand\_copy register by 1 bit.
5. Subtract 1 from bit register.
6. Repeat until bit is greater than zero.

**Multiply\_testbench.v**

Multiply\_testbench.v contains the test cases as discussed as in section D. The testcases have been designed to thoroughly test the code with all possible combinations of inputs to Multiply.v.

# Reference

Patterson, D. A., Hennessy, J. L. (2015). Computer organization and design: The hardware/ software interface. Retrieved October 25, 2018, from <http://www4.comp.polyu.edu.hk/~comp2421/ComputerOrganizationAndDesign5thEdition2014.pdf>

Koren. (2010). Part 4: Datapath Design – Multiplication and Floating-point representations and operations. Lecture. Retrieved October 25, 2018, from <http://euler.ecs.umass.edu/ece232/pdf/04-MultFloat-11.pdf>

Baruch, Z. F. (2002). Structure of computer systems. Cluj-Napoca: U. T. Pres. Retrieved October 25, 2018, from <http://users.utcluj.ro/~baruch/book_ssce/SSCE-Shift-Mult.pdf>

Justin Rajewski, *Writing Test Benches*, January 10, 2018

Purdue Online Writing Lab. *APA Formatting and Style Guide* Retrieved from

<https://owl.purdue.edu/owl/research_and_citation/apa_style/apa_formatting_and_style_guide/general_format.html>