# ECE 385

Fall 2020

**Experiment # 5**

**An 8-Bit Multiplier in SystemVerilog**

Lian Xinyu & Liu Tianyu

TA: Li Shuren

**Introduction**

Following on from the previous task of designing an adder, this experiment requires us to design an 8-bit 2’s complement multiplier using CAD. We designed the multiplier architecture by dividing the modules, then completed the programming of each module, and finally verified the correctness of the multiplier's operation with test results. Moreover, we use some analysis tools in the software to finish the discussion for efficiency and energy consumption.

**Prelab**

1. This is the result of rework for using Multiplier B = 7, and Multiplicand S = -59.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Functions** | **X** | **A** | **B** | **M** | **Comments for the next step** |
| Clear A, Load B | 0 | 00000000 | 00000111 | 1 | M=1, so add S to A |
| ADD | 1 | 11000101 | 00000111 | 1 | Shift XAB |
| SHIFT | 1 | 11100010 | 10000011 | 1 | M=1, so add S to A |
| ADD | 1 | 10100111 | 10000011 | 1 | Shift XAB |
| SHIFT | 1 | 11010011 | 11000001 | 1 | M=1, so add S to A |
| ADD | 1 | 10011000 | 11000001 | 1 | Shift XAB |
| SHIFT | 1 | 11001100 | 01100000 | 0 | M=0, so shift XAB |
| SHIFT | 1 | 11100110 | 00110000 | 0 | Shift XAB |
| SHIFT | 1 | 11110011 | 00011000 | 0 | Shift XAB |
| SHIFT | 1 | 11111001 | 10001100 | 0 | Shift XAB |
| SHIFT | 1 | 11111100 | 11000110 | 0 | Shift XAB |
| SHIFT | 1 | 11111110 | 01100011 | 1 | END |

1. Block diagram

**图示

描述已自动生成**

**Operation**

Referring to our design, we divide the multiplier into five modules: top-level, 9bitADD, 8bit-register, D-flipflop and control unit.

1. **Top-level**: This part is the core module for the multiplier. We connect the input signal with other modules in the top-level part. In addition, we prepare Hex-Driver to input the logic result to DE2 board and synchronizer unit.
2. **9bitADD**: In this part, we connect nine 1-bit full adders to create a 9-bit adder. Moreover, this adder can not only perform add operation, but achieve subtract task. In addition, we provide one control signal fn in design to determine the model that the 9bitADD will execute.
3. **8-bit registe**r: In this part, we use asynchronized design to build an 8-bit register module. Notice that we need to provide two register instances, A and B, for the top-level design.
4. **D-flipflop**: In fact, it is a 1-bit register, and we also use asynchronized design to build this module. In the multiplier, it will be used to store X.
5. **Control unit**: This module is used to derive the operation that the multiplier should perform. In total, we need to provide execute signal for these operations: add, sub, shift, resetA\_loadB, resetB, stop. Above all, we use the Moore machine to finish our design for control unit part.

For the process of testing part using DE2 board, after we finish the planning for pins, we need open the source at first. Then we need to set the proper value for data B, then trigger ClearA\_LoadB signal. After we finish this step, data B will be loaded in to register B, then we need to input the corresponding value of data A into register A. When we prepare the input data for computing, we can trigger the run signal and check the result. If we finish computing, we need trigger reset signal to clear two registers.

**State Diagram**

图片包含 形状

描述已自动生成

Here is the description for the functions of different states, and for the output of each state, the description follows this form:

**clearA /loadA /loadB/add\_sub/shift**

**Resetstate**: This state is used for performing reset operation and clearA\_loadB operation. For reset operation, the output signal will control the processor to clear register A and B. For clearA\_loadB operation, the output signal will control the processor to clear register A and load data into the register B.

**Out: clearA\_loadB/0/clearA\_loadB /0/0**

**Beginrun:** This state informs that the process begins, and it will output the signal to clear the content in register A.

**Out: 1/0/0/0/0**

**A1,A2,A3,A4,A5,A6,A7**: This state (, is used to perform add operation. If the input data m (Bval[0], the right most bit in data B ) is 1, the state will control the processor to add the data in S into the register A, otherwise it will not affect the current work of the processor.

**Out: 0/m/0/0/0**

**S1,S2,S3,S4,S5,S6,S7,S8**: This state (, is used to control the shift operation. If the processor meets this state, it will perform execute shift function.

**Out: 0/0/0/0/1**

**A8**: This state is used to perform subtract operation. If m (Bval[0], the right most bit in data B) is 1, it will control the processor to subtract the value in register A using the, otherwise it will not affect the current work of the processor.

**Out: 0/m/0/m/0**

**Hold**: This stat will not affect the current work of the processor, it means “keep the same work”.

**Out: 0/0/0/0/0**

**Schematics for the multiplier design**

**8-bit register**

图示

描述已自动生成

**1-bit register**

图示

描述已自动生成

**Adder/Subtracter**

图示

描述已自动生成

**Control unit**

图示, 示意图

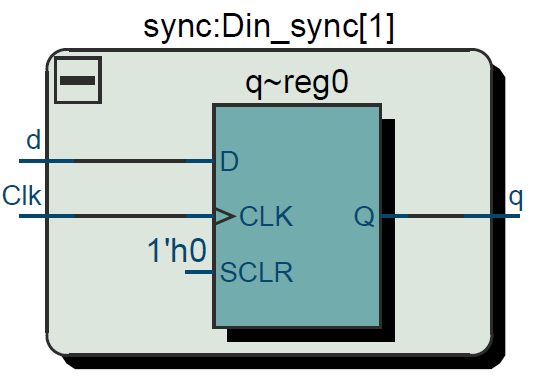
描述已自动生成

**HexDriver**

图示

描述已自动生成

**Synchronizer**

****

**Written Description of .sv Modules**

**Module:** toplevel.sv

**Inputs:** Clk, Reset, Run, ClearA\_LoadB, [7:0]S

**Outputs:** [7:0]Aval, [7:0]Bval,X,[6:0]AhexL, [6:0]AhexU, [6:0]BhexL, [6:0]BhexU

**Description:** This is a top-level file that operates on two 8-bit 2’s compliment numbers and produce a 16-bit result. When Reset is high, the data in the register will be set to the initial value, the machine will go back to the initial state. When Run is high, the multiplier will execute the process. When ClearA\_LoadB is high, the data in register A will be cleared and data in S will be loaded to register B.

**Purpose:** This module is used as a top-level platform to create all the circuit.

**Module:** full\_adder.sv

**Inputs:** x, y, z

**Outputs:** s, c

**Description:** This is a 1-bit adder that have two input (x and y) and a carry-in bit (z) and outputs the sum (s) with the carry-out bit (c).

**Purpose:** This module is used to as the underlying logic.

**Module:** ADD\_SUB9.sv

**Inputs:** [7:0]A, [7:0]B, fn

**Outputs:** [8:0]S

**Description:** This is a 9-bit carry-select adder. This module extend to input A and B to 9 bits. Then adds them together to produce a 9-bit sum using the carry-select method. When fn is 1, the module will do the subtraction.

**Purpose:** This module add or subtract the two 8-bit data and produce a 9-bit result with the 9th bit indicate the sign.

**Module:** control.sv

**Inputs:** clk, reset, run, calb, m

**Outputs:** clearA, loadA, loadB, shift, add\_sub

**Description:** This is a finite state machine to control the current and the next state. The state will change at the rising edge of the clk. The run signal will determine whether the first state should jump to the next state. The other input calb and m are used to determine the outputs. There are 19 states in the state machine.

**Purpose:** This module is used to create a finite state machine to control the multiplier to implement the add-shift algorithm to calculate the product.

**Module:** D\_filpflop.sv

**Inputs:** clk, load, reset, datain

**Outputs:** dataout

**Description:** This is a 1-bit register. When reset is high, the register will be cleared. When load is high, the register will load the datain in.

**Purpose:** This module is used to create a 1-bit register to hold X in the multiplier circuit.

**Module:** register\_8bit.sv

**Inputs:** clk, reset, shiftin, load, enable, [7:0]datain

**Outputs:** shiftout, [7:0]dataout

**Description:** This is a 8-bit register. When reset is high, the register will be cleared . When load is high, the register will load datain in. When enable is high, the register will perform a right-shift operation. It will take shiftin as the new most significant bit and shift out its original least significant bit to shiftout.

**Purpose:** This module is used to create 8-bit registers like A and B in the multiplier circuit.

**Module:** synchronizers.sv

**Inputs:** Clk, d

**Outputs:** q

**Description:** This is a 1-bit synchronizer. It synchronizes input s to output q.

**Purpose:** This module is used to synchronize the inputs from the FPGA board to the logic inside the circuit.

**Module:** HexDriver.sv

**Inputs:** [3:0]In0

**Outputs:** [6:0]Out0

**Description:** This is a converter that transforms the binary value in In0 to seven-segment form.

**Purpose:** This module is used to display the values in A and B to FGPA board.

**Module:** testbench.sv

**Inputs:** none

**Outputs:** none

**Description:** This is a testbench used to test the functionality of our multiplier circuit. It tests -2\*7, 6\*7, -3\*5, 2\*(-7) and (-6)\*(-7).

**Purpose:** This module is used to test the functionality of our multiplier.

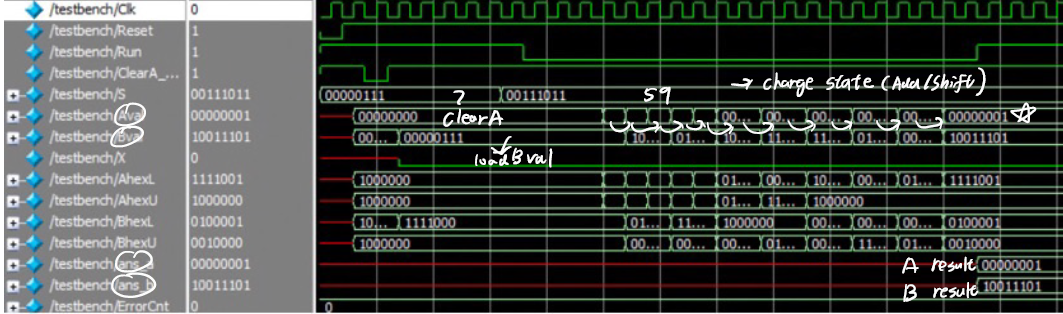
**Diagram for the control unit**

图片包含 形状

描述已自动生成

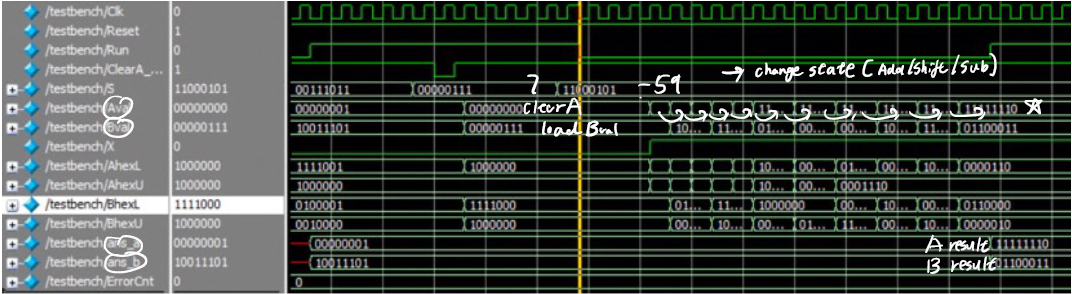
**Prelab Graph**

**These graphs are the results with annotation of RTL simulation based on testbench.sv file.**

****

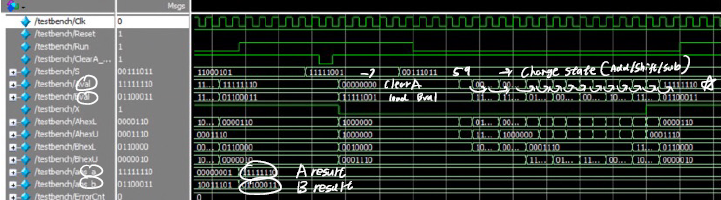
**7\*59**

**Both 7 and 59 are positive numbers, so we do not have to perform the subtraction operation. We just need to clear the register A, load data 7 into register B, input 59, and then follow the state machine instructions to calculate 7\*59=413.**

****

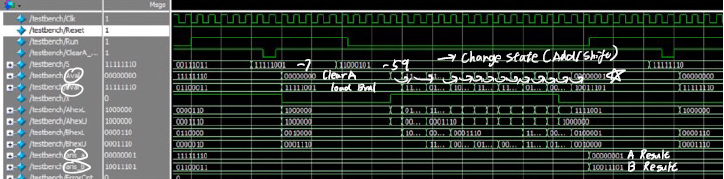
**7\*(-59)**

**-59 is a negative number, so we need to perform the subtraction operation. This time, we need to clear the register A, load data 7 into register B, input -59, and then follow the state machine instructions to calculate 7\*(-59)=-413.**

****

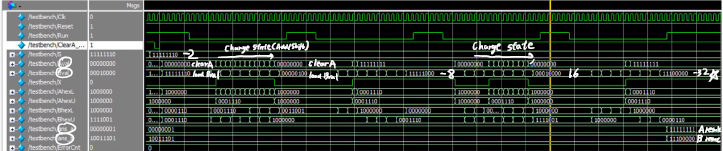
**-7\*59**

**-7 is a negative number, so we need to perform the subtraction operation. This time, we need to clear the register A, load data -7 into register B, input 59, and then follow the state machine instructions to calculate -7\*59=-413.**

****

**-7\*(-59)**

**-7 is a negative number, but -59 is also a negative number. Therefore, we do not need to perform subtraction operation. This time, we need to clear the register A, load data -7 into register B, input -59, and then follow the state machine instructions to calculate -7\*(-59)=413.**

****

**-2\*(-2)****\*(-2) \*(-2) \*(-2)**

**This operation is used to prove the correct result for continuous multiplication. If can be divided into five single multiplication steps. The final result is -32.**

**Post lab**

**1.**

|  |  |
| --- | --- |
| **LUT** | 95 |
| **DSP** | No DSP |
| **Memory (BRAM)** | 0 |
| **Flip-Flop** | 47 |
| **Frequency(MHz)** | 73.05 |
| **Static Power(mW)** | 98.51 |
| **Dynamic Power(mW)** | 3.86 |
| **I/O Thermal Power(mW)** | 43.85 |
| **Total Power(mW)** | 146.22 |

To optimize our design, firstly we can try to find the minimum requirement for the number of states transformation, since the transfer between states cost plenty of time. Secondly, we can improve the efficiency of the add/sub step, which means that we can choose adders with higher frequency and efficiency, such as CLA and CSA, to finish the tasks.

**2.**

**What is the purpose of the X register. When does the X register get set/cleared?**

X represents the sign bit of the multiplication result. X register is used to store it. When Reset or ClearA\_LoadB instruction is executed, the register will be cleared. When fn signal is used to perform add/sub operation, the register will be set.

**What are the limitations of continuous multiplications? Under what circumstances will the implemented algorithm fail?**

Continuous multiplication needs to be careful about overflow situation. That is, taking continuous multiplication as one example, if the previous product is not in the interval (-128,127), then the result of the next operation will be problematic. In other words, the multiplication algorithm will fail.

**What are the advantages (and disadvantages?) of the implemented multiplication algorithm over the pencil-and-paper method discussed in the introduction?**

Advantages: Manual processing (pencil-and-paper) of binary numbers is obviously more difficult, especially for the subtraction of the last digit, while multiplication algorithm of binary numbers has natural advantages in this aspect and the multiplication can be accomplished by relying on schemes that automate the processing of state machines. Moreover, the frequency of the multiplier reaches 73.05 MHz, which is still very efficient.

Disadvantages: If we calculate small add computing such as 1+6, we need to shift 8 bits for B so that we can get the final result. But in binary representation, 6 is just 00000110, so using pencil-and-paper method we do not need to run so many states. This is the lack of multiplication algorithm.

**Conclusion**

1. **Bugs/Debugs**

Thanks to Pro. Li, we learnt a lot in the lecture and we can finish out lab smoothly. In this lab, we encounter some errors. Almost the errors are caused by cacography. Fortunately, this kind error will be indicated when do the simulation. We can solve the error easily.

1. **Suggestions/Problems**

The instruction for lab5 is very perfect. I do not have any extra suggestions.

1. **Summary**

In this experiment, we construct a multiplier. This multiplier can calculate the multiplication of two 8-bit numbers. The multiplier can also multiply evenly, as long as the previous product is within a certain range. To design this multiplier, we use registers to store the data and state machines to control the computing process. This improved our understanding to how the add-shift method works. Moreover, we gained a lot about the knowledge related to state machines and System Verilog language.