Project 7b: Multiplier and Divider

E210/B441, Spring 2019

Version 2019.0

Autograder Due: 11:15am, Friday, April 26, 2019

Demo Due: 1:10pm, Friday, April 26, 2019

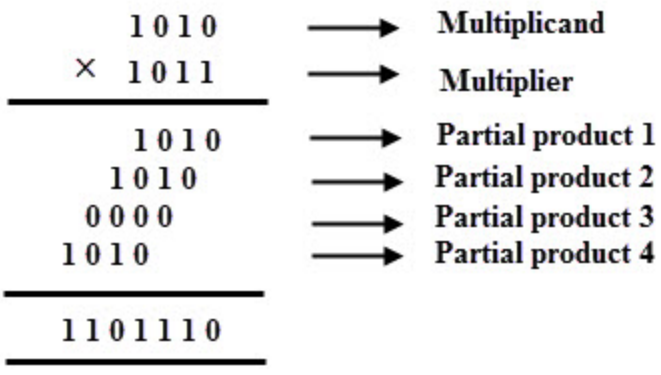
Overview

This is a continuation of the P7: Calculator Project. Here we will add a multi-cycle multiplier to the existing calculator. You can also optionally incorporate a multi-cycle divider for extra credit.

Background

Binary Multiplication

The basic process for binary multiplication consists of two steps, generating partial products and then adding them. This is illustrated in the figure below [2].



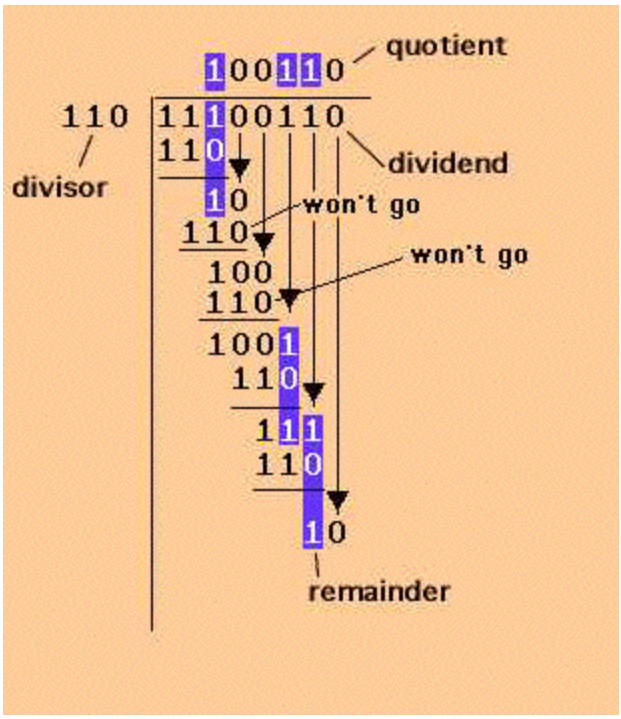
Recall that the partial products can be generated as a series of simple boolean equations, shown below.

```
                                        a3  a2  a1  a0
                                  ×     b3  b2  b1  b0
                                 ---------------------
                                    b0a3 b0a2 b0a1 b0a0
                               +  b1a3 b1a2 b1a1 b1a0    0
                               + b2a3 b2a2 b2a1 b2a0     0    0
                               + b3a3 b3a2 b3a1 b3a0     0    0    0
                                 ---------------------------------
                                  p7   p6   p5   p4   p3   p2   p1   p0
```

There are a number of ways to add the partial products. Provided these implementations are correct, we do not require a specific multiplier implementation. The only requirement is that whatever multiplier implementation you choose, it must meet the timing requirements specified below.

Binary Division

Below is the basic binary division algorithm illustrated graphically and with a brief algorithm for implementation [1].



There are numerous techniques or algorithms to implement division, we recommend following this simple algorithm:

- Set quotient to 0
- Align leftmost digits in dividend and divisor
- Repeat
    - If that portion of the dividend above the divisor is greater than or equal to the divisor
        - Then subtract divisor from that portion of the dividend and
        - Concatenate 1 to the right hand end of the quotient
        - Else concatenate 0 to the right hand end of the quotient
    - Shift the divisor one place right
- Until dividend is less than the divisor
- quotient is correct, dividend is remainder
- STOP

Assignment Description

Your assignment is to create the following new Verilog modules and testbenches as specified below. You will then integrate them into the existing Calculator infrastructure.

Multiplier (New)

```
module multiplier(
    input         clk,
    input         rst,

    input         start,
    input  [7:0]  multiplicand,
    input  [7:0]  multiplier,

    output        done,
    output [7:0] product
```

```
);
```

This module should compute the result of `product = multiplicand * multiplier`. The operation is started when the `start` signal is raised. Upon a `start` request, your implementation must take between 2-10 cycles to compute the product, and raise the `done` signal when the result is complete. `Done` should stay high for 1 clock cycle.

You may assume that `multiplicand` and `multiplier` will not change during the computation.

This unit must synthesize correctly at the full 100MHz clock frequency of the Basys3 board. However, it may take multiple cycles to complete the operation.

You may NOT use the built-in operator ('*') for this module!

Technically, the resulting product is actually 16 bits. However, our stack only stores 8-bit results. **Therefore, the upper 8 product bits may be discarded.**

Divider (New)

NOTE: This module is optional. It is extra credit.

```
module divider(
    input          clk,
    input          rst,

    input          start,
    input  [7:0]   dividend,
    input  [7:0]   divisor,

    output         done,
    output [7:0]   quotient,
    Output [7:0]   remainder
);
```

This module should compute the result of `quotient = dividend / divisor` and `remainder = dividend % divisor`. The operation is started when the `start` signal is raised. Upon a `start` request, your implementation must take between 2-10 cycles to compute the `quotient` and `remainder`, and raise the `done` signal when the result is complete. `Done` should stay high for 1 clock cycle.

You may assume that `dividend` and `divisor` will not change during the computation.

This unit must synthesize correctly at the full 100MHz clock frequency of the Basys3 board. However, it may take multiple cycles to complete the operation.

You may NOT use the built-in operators (‘%’ or ‘/’) for this module!

Calculator (Updated)

This module defines a postfix calculator. It remains largely unchanged, with two exceptions. First, two new operations must be supported, multiplication and modulo. Second, the calculator should instantiate multiplier and divider submodules internally.

```
module calculator(
    input        clk,
    input        rst,
    input  [7:0] in_data,
    input        in_req,
    output       out_ready,
    output [7:0] out_data,
    output       err
);

multiplier mult0 ( .clk(clk), .rst(rst), ...  );

//extra credit
divider div0( .clk(clk), .rst(rst), ...  );

endmodule
```

Byte (Hex)	Operation	Description
0xff	PEEK()	Return the top byte of the stack over UART
0xfe	AND (X Y &)	AND the top two values on the stack, push the result back onto the stack
0xfd	OR (X Y |)	OR the top two values on the stack, push the result back onto the stack
0xfc	NOT (Y ~)	NOT the top value on the stack, push the result back onto the stack
0xfb	XOR (X Y ^)	XOR the top two values on the stack, push the result

		back onto the stack
`0xfa`	`ADD (X Y +)`	Add the top two values on the stack, push the result (X + Y) back onto the stack
`0xf9`	`SUB (X Y -)`	Subtract the top two values on the stack, push the result (X - Y) back onto the stack
`0xf8`	`INC (X 1 +)`	Increment the top value on the stack by 1, push the result back onto the stack
`0xf7`	`MULT (X Y *)`	Multiply the top two values on the stack, push the result back onto the stack. *Overflow cases can be neglected.*
`0xf6`	`MOD (X Y %)`	Compute the integer modulo (division remainder) of the top two values on the stack, push the result back onto the stack. (X % Y) (EXTRA CREDIT)
`0xf5`	`POP()`	Remove the top value from the stack. Returns it over UART.
`0xf4 - 0x00`	`PUSH(value)`	Add a new value to the stack. The value added is the byte received, *e.g.* `0x02` = `PUSH(2)`.

Testbench

For this project, we do not require any testbenches for the autograder, since there are too many combinations of different operands and operators. HOWEVER, you should still write your own testbenches!

Remember to select "System Verilog" from the "File Type" drop-down menu.

Constraints

You will also need to reconfigure your constraints file to align with the top-level module declaration. The names should line up properly by default. A reference file is available in the Google Drive folder.

Python Helper Script

For demonstration, use the python script available at the link below. This allows us to communicate with the FPGA without having to convert between hex and ascii (the python script handles this for us)

Calculator.py

Evaluation

The evaluation will have two steps, first submission of your source code and testbench to the autograder. Second, you will need to synthesize your design, download it to the FPGA and do a demonstration for the TA.

Autograder (60%)

Log on to https://autograder.sice.indiana.edu and submit your code as per Project 1.

Demonstration (40%)

Program your FPGA with your calculator and demonstrate your working system to the TA. You will not receive full points until the TA has approved your demonstration.

References

[1] https://courses.cs.vt.edu/~cs1104/BuildingBlocks/divide.030.html

[2] https://www.electronicshub.org/binary-multiplication/