Lab 3: Hardware Multiplier Design

In this lab, you are going to design your first Synchronous HDL design, including a Finite State Machine (FSM). You will implement a multiplication algorithm using digital hardware.

1. Multiplication Algorithm

First, let's review the multiplication algorithm, we can multiply 1000 by 1001 by performing the following steps:

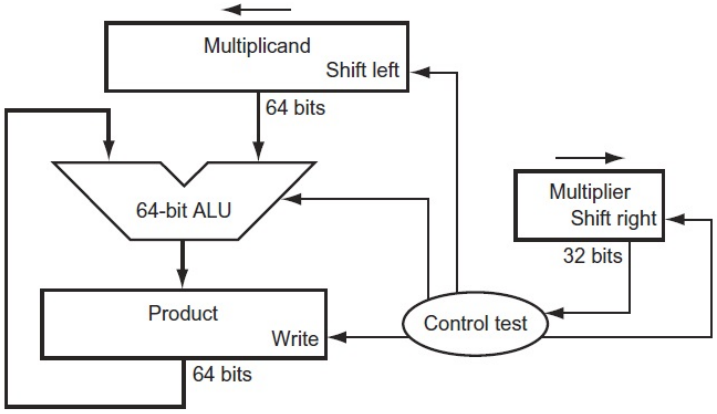
Multiplicand		1000_{ten}
Multiplier	x	1001_{ten}
		1000
		0000
		0000
		1000
Product		1001000_{ten}

The first operand is called the *multiplicand* and the second the *multiplier*. The final result is called the *product*. As you may recall, the algorithm learned in school is to take the digits of the multiplier one at a time from right to left, multiplying the *multiplicand* by the single digit of the *multiplier*, and shifting the intermediate product one digit to the left of the earlier intermediate products. The first observation is that the number of digits in the *product* is considerably larger than the number in either the *multiplicand* or the *multiplier*. In fact, if we ignore the sign bits, the length of the multiplication of an *n*-bit *multiplicand* and an *m*-bit *multiplier* is a *product* that is $n + m$ bits long. That is, $n + m$ bits are required to represent all possible products.

In this example, we restricted the decimal digits to 0 and l. With only two choices, each step of the multiplication is simple:

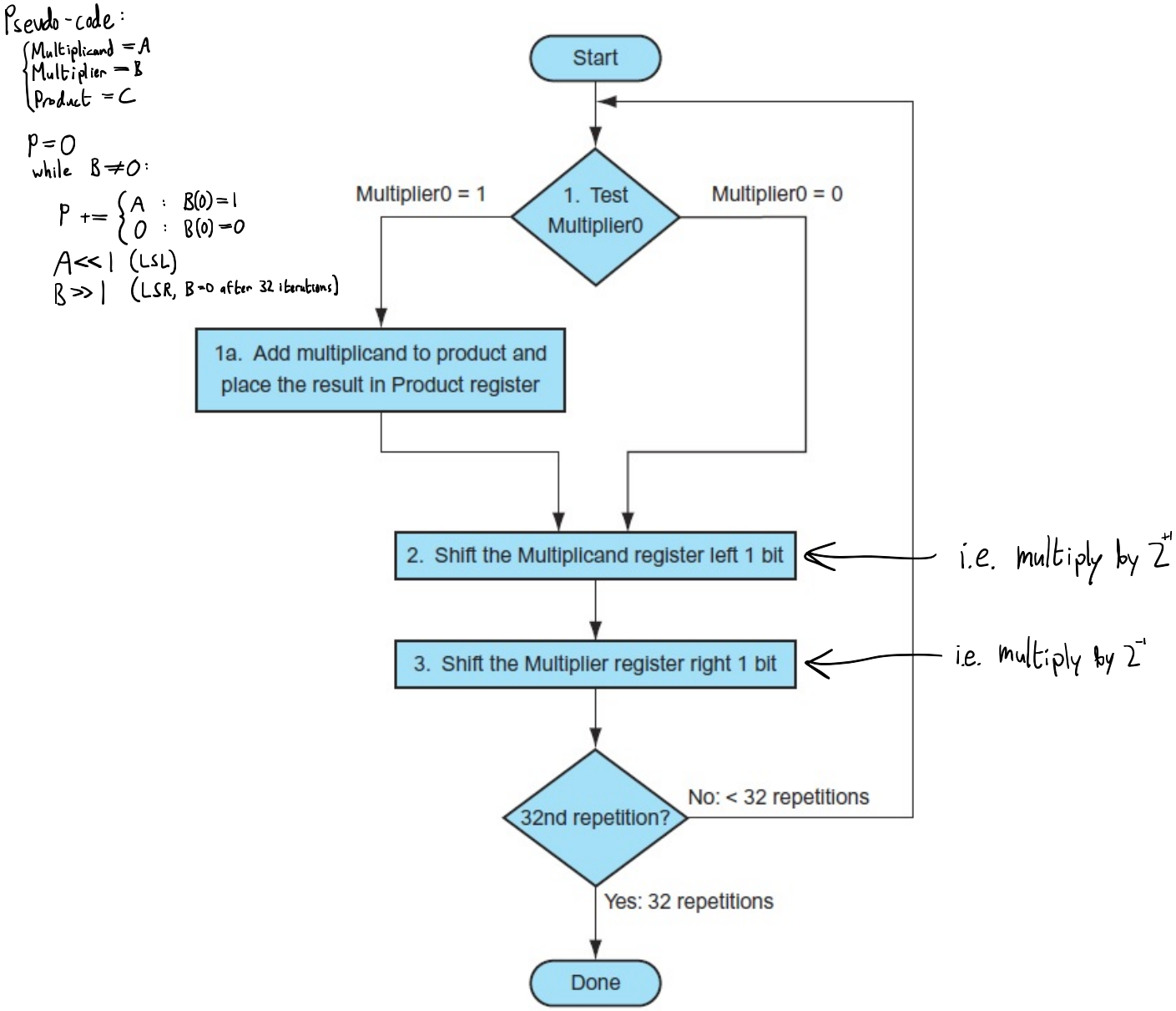
1. Just place a copy of the *multiplicand* (1 x *multiplicand*) in the proper place if the *multiplier* digit is a 1, or
2. Place 0 (0 x *multiplicand*) in the proper place if the digit is 0.
3. Add all the partial products till you get the final result.

One way (but not the only way) we can realize this algorithm in hardware as follows:



Where the *Multiplicand* register, ALU, and *Product* register are all 64 bits wide, with only the *Multiplier* register containing 32 bits. The 32-bit *multiplicand* starts in the right half of the *Multiplicand* register and is shifted left 1 bit on each step. The *multiplier* is shifted in the opposite direction at each step. The algorithm starts with the *Product* register initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the *Product* register.

Here is a flow chart that depicts the algorithm:



For this lab, you are going to design, synthesize, and test a 32-bit multiplier, according to the hardware proposed above. This is your first synchronous project which will contain many clock-based components interacting together. Please makes sure to design each component carefully, and fully test it first, before trying to put everything together.

It's also a good to familiarize yourself with the algorithm that you will translate into hardware, do a few small, 4-bit examples by hand.

2. Multiplier Components

This section will provide some guidelines on what components needed and how to implement them. Note that the information mentioned here is just a guideline, the same hardware can be implemented in different ways.

A. Adder:

This is just an asynchronous 64-bit adder, with no carry out. You can use your adder from lab 1.

B. Register:

This is a synchronous 64-bit register, with asynchronous reset. The register needs to have a load signal as it will help down the way. It's useful to use generic here as you will be needing this component in Lab 4. Here is an example of a 1-bit register (flip-flip):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY flipflop IS
   PORT(
      CLK : IN     std_logic;
      D   : IN     std_logic;
      EN  : IN     std_logic;
      RST : IN     std_logic;
      Q   : OUT    std_logic
   );

-- Declarations

END flipflop ;

--
ARCHITECTURE flipflop OF flipflop IS
BEGIN
    CLKD : process(CLK, RST)
    begin
       if(RST = '1') then
          Q <= '0';
       elsif(CLK'event AND CLK = '1') then
          if(EN = '1') then
             Q <= D;
          end if;
       end if;
    end process CLKD;
END flipflop;
```

Make sure to write a testbench to test its functionality thoroughly.
Note: writing a Tcl script might be easier than a VHDL testbench, because you can use add_force to easily for a clock.

C. Shift Registers: Using behavioral shift

This is a synchronous 64-bit and 32-bit shift registers, with asynchronous reset. The shift register needs to have a load signal and shift signal, as you will be loading the *multiplier* and *multiplicand* values before shifting them. You will be needing two flavors, a shift left for the *multiplicand* and a shift right for the *multiplier*.

Make sure to write a testbench to test its functionality thoroughly.
Note: writing a Tcl script might be easier than a VHDL testbench, because you can use add_force to easily for a clock.

D. Control Unit:

Implement the control unit as a Finite State Machine (FSM). You can either do Mealy or Moore State Machines. Check the files with this lab for an example code on each of them. Here are some tips on how to implement them:

a. Make a first pass at constructing the multiplier data path. A data path means all the components (registers, shift registers, adder, ….) that you have in your design, connected together, except for the control unit. Leave the controller as a "black box" that takes in inputs and produces the control bits that you will need. Please note, your data path will likely change as your design progresses.
b. Start implementing the control logic. As your FSM develops, you may need to make modifications to your data path to add or modify control signals and/or include additional data path components. For example, for an *n*-bit multiplier we could include *n* additional states to track the number of repetitions. However, we can also simplify the FSM by adding more hardware to our data path (a 5-bit counter for example).

Make sure to write a testbench to test its functionality thoroughly. The states have to transition correctly as you desire.
Note: writing a Tcl script might be easier than a VHDL testbench, because you can use add_force to easily force a clock.

What to do:

1. Design the 32-bit Multiplier Unit, per the hardware algorithm above. Your final top-level Multiplier Unit should have the following IO ports. Make sure that the ports' name in your entity matches the letter case and naming in the Table.

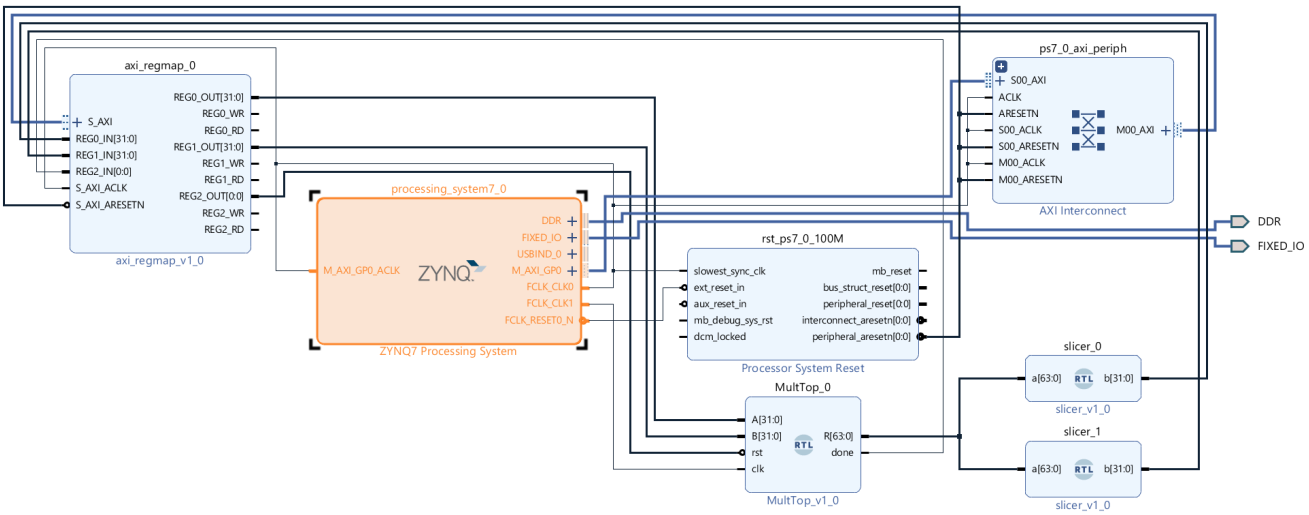
Port	Direction	Description
A(31:0)	In	32-bit input bus carrying the "A" operand.
B(31:0)	In	32-bit input bus carrying the "B" operand.
clk	In	A single bit carrying the clock signal.
rst	In	A single bit for the asynchronous active high reset signal.
R(63:0)	Out	64-bit output bus which will hold the *Product*
done	Out	A single bit which is only high when the multiplication is done and the product is ready, otherwise, it's low.

2. Write a VHDL or Tcl Script testbench to verify the functionality of the Multiplier Unit. You should test a wide range of cases.
3. Synthesize, implement and generate bitstream for your Multiplier Unit, then write C/C++ testbench to fully verify the functionality of your FPGA-configured design. You should test a wide range of cases and include random testing. You will be needing more `regmap` registers for this lab, so make sure to configure and take notes of their numbers and bit-widths.
   Note: You will need to use two `regmap` for *R*. Think of a way to split *R* in two 32-bit wide buses. Whatever your solution is, it should be separate from your Multiplier Unit. Your *clk* would be tied to the board clock source.

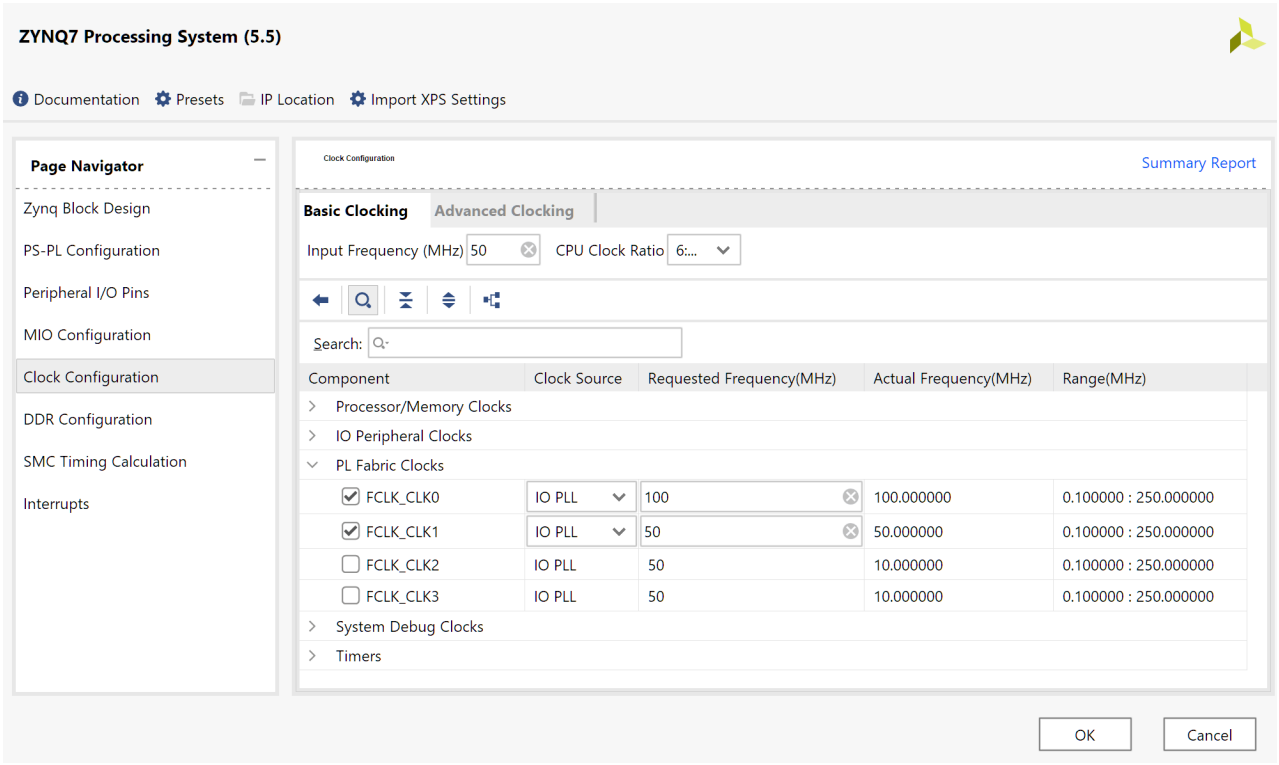
Clock Signal and C/C++ testbench

When making the final diagram for the multiplier to be synthesized, including the Zynq processor and the axi_regmap component, it's time to connect the clk signal of the Multiplier to an actual physical clock signal. The PYNQ-Z1 board offers multiple clock signal sources with variety of frequencies. You can check the pynq-reference-manual (uploaded to Canvas) on that topic for more information on what types of clocks the board has. For the sake of this project, we are going to use a 50 MHz, that we can generate from the Zynq processing system component to feed the Multiplier.

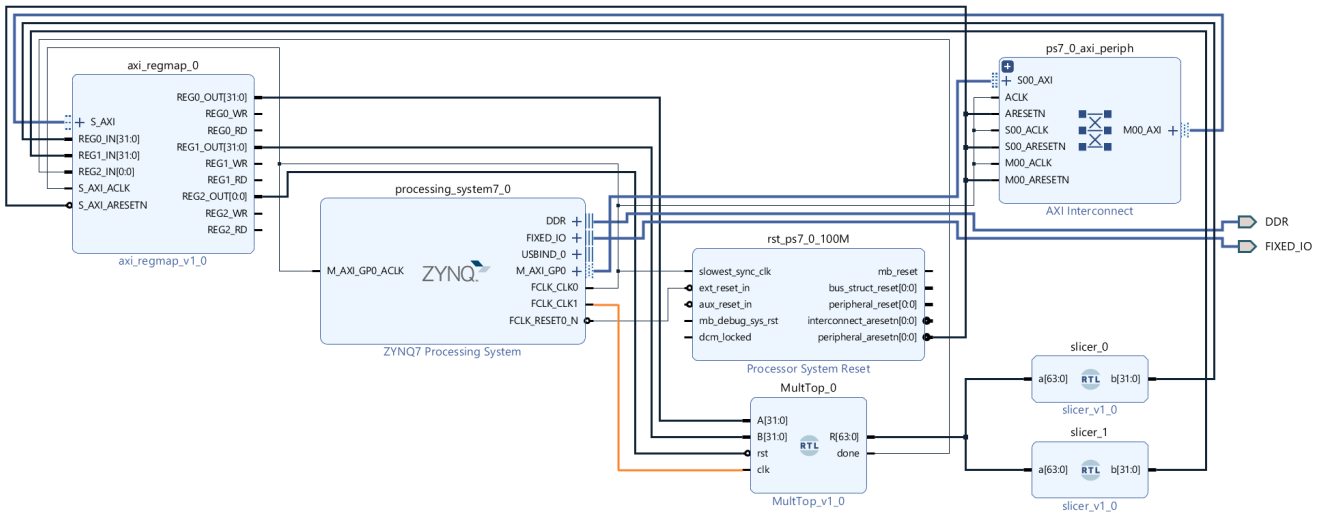
Double click on the ZYNQ7 Processing System component:



Then choose Clock Configuration, expand PL Fabric Clocks and check the box next to FCLK_CLK1. Change the Requested Frequency to 50, then click OK:



An extra pin will appear in the block for the FCLK_CLK1 that we just chose, now you can connect to the multiplier clock signal:



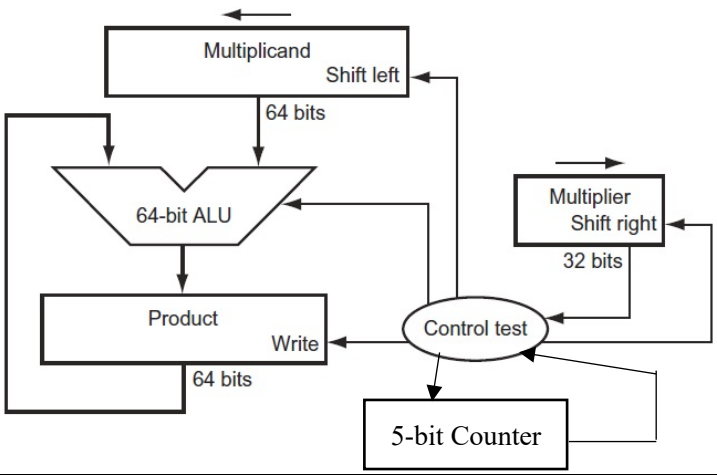
Now you can proceed as in the usual.

Deliverables:

1. You will be delivering all the VHDL and C codes that you have written, including the design files, testbenches, and the C codes. The submission will be on Canvas Assignment for Lab 3. You can just Zip the top Vivado project folder that has all the files that was mentioned above. (please make sure that it has all the files required).
2. Check-offs will be done on the due date for this assignment. The rubric for the check-offs will be provided later.

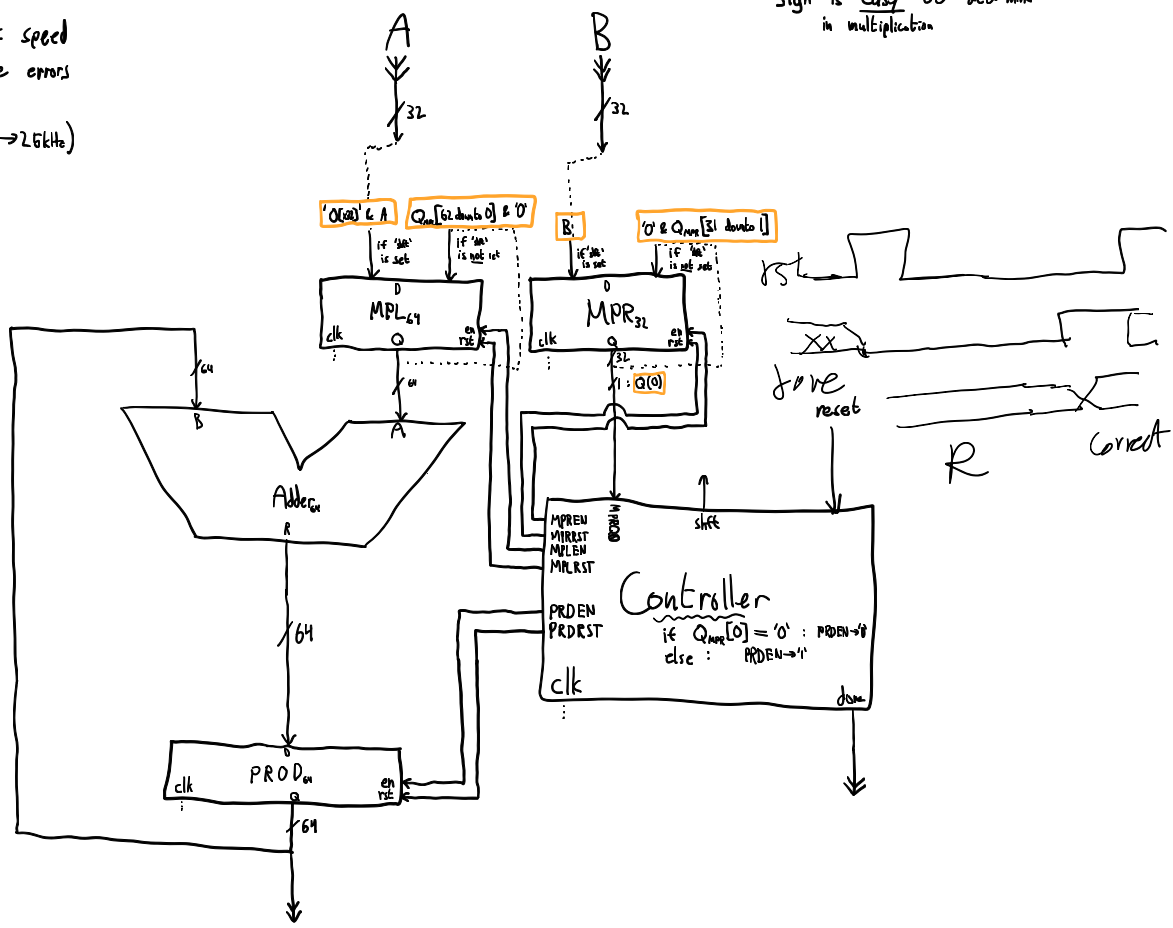
Bonus:

Students who design the Multiplier Unit using a 5-bit counter, as a separate block, included in their data path (see the diagram below) AND get the product ready in the least number of cycles possible from when the *rst* signal is asserted will get a 2% bonus toward their final grade.



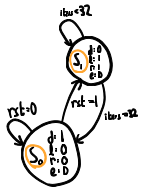
lower clock speed
if there are errors
in C tb.
(i.e. 50kHz → 25kHz)

sign is easy to determine
in multiplication



e.g. A B
1001 · 0101

instate	in MPL	in MPR	out PROD	out done	in rst	out shft	out prod enable	state
0	X	X	X	1	0	0	X	s0
1	00001001	0101	00000000	0	1	1	1	s1
2	00010010	0010	00001110	0	0	1	0	s1
3	00100100	0001	00001110	0	0	1	1	s1
4	01001000	0000	00101010	1	0	0	0	s0



$Q_{prod}[0]$

Use a mealy machine