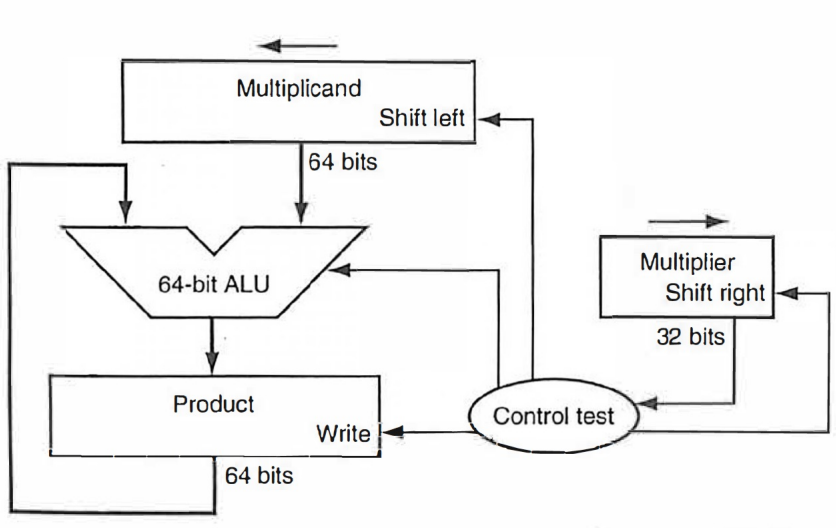
cs281: Introduction to Computer Systems

Lab07: Hardware Multiplication FSM

Assigned: November 1, Due: November 8

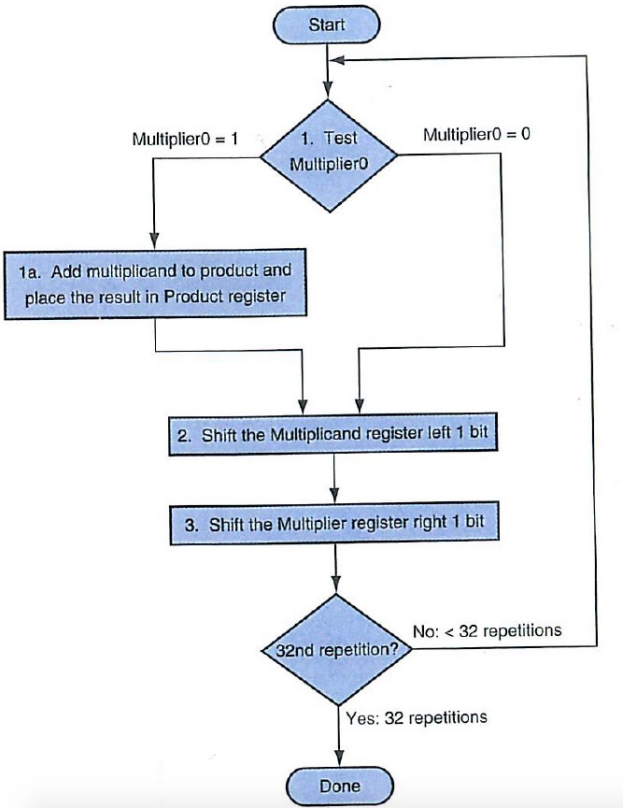
1. The purpose of this laboratory is to:
   - reinforce our experience in designing and implementing sequential logic circuits,
   - learn how hardware can accomplish our goal of multiplying binary numbers,
   - develop expertise with Logisim.
2. This lab may be completed in teams of at most two. You will only submit a copy of your Logisim file and a PDF of the elements of your design; there will be no Lab Report beyond these two elements. You will be given a template `multiply.circ` file so that you have a framework from which to build your binary multiplication circuit.
3. Abstract view of the hardware:



This design mimics the algorithm we learned in grammar school. We have drawn the hardware so that data flows from top to bottom to resemble more closely the paper-and-pencil method. Unlike the picture, we will develop an 8 bit times 8 bit multiply circuit, and so the three locations that refer to 64 bits will be 16 bits for us, and where the picture says 32 bits is 8 bits for us.

Let's assume that the multiplier is in the 8-bit Multiplier register and that the 16 bit Product register is initialized to 0, and the 16 bit Multiplicand is initialized with the 8-bit operand and zero filled to the left. Analogous to what we do with pencil and paper, at each step of the algorithm, we shift the multiplicand left one digit (bit) as, for each instance of a 1 in the multiplier, we add the multiplicand to a partial sum.

4. Abstract view of the control flow:



The next figure depicts the flow of the hardware algorithm. The least significant bit of the multiplier M_0 determines whether the multiplicand is added to the Product register. If 0, we do not add the multiplicand. Next, the Multiplicand register is shifted left by 1 bit, to position it for the next addition, and the multiplier is shifted right by 1 bit, to determine the next bit to check for 1 or 0. We need a mechanism to determine how many iterations to complete ... 8 total iterations for an 8-bit by 8-bit multiplication.

5. Finite State Machine: Given the provided template in `multiply.circ`, your goal is to design and implement the finite state machine providing this functionality; the realization of the registers, the wiring, and dataflow between your FSM and the rest of the circuitry is already in place. The FSM has two binary inputs (beyond the clock and an asynchronous reset line), and four binary outputs as follows:

In/Out	Name	Description
Input	M_0	Least Significant Bit of the Multiplier
Input	C	Continue indicator, 1 to continue, 0 when 8 iterations are complete
Output	`DoAdd`	1 if the ALU should add the current multiplicand to the current (partial) product
Output	`DoShift`	1 if the hardware should shift the Multiplicand, Multiplier, and Steps registers
Output	`DoLoad`	1 if the hardware should load initial values into the Multiplicand, Multiplier, Product, and Steps registers
Output	`DoDone`	1 if the hardware has completed the multiply operation

6. Design1: Given the above inputs and outputs, design a finite state machine that implements the hardware multiply algorithm.
    - Since the output of the FSM has output bits that control the loading of the registers in the datapath and for signaling the completion of operation, the reference solution has states for each: a start state that signals the `DoLoad` output (but none of the other outputs), and a final state that signals the `DoDone` output.
    - It is OK to have state(s) that have **no** outputs.
    - It is OK to have transitions that are *always* taken, regardless of the values of the inputs, or that are taken for a particular value of one of the inputs, and are "don't care" for the other input. We note this with an "X" for "don't care" on the transition arc for each appropriate input.
    - The reference solution used a total of five state.
7. Design2: Consider your state assignment carefully. This is where you map the logical states of the FSM to the bit representations of your 1-bit flip flops representing the state. For the reference solution, because five logical states required 3 bits of storage, we used one of the storage bits that, when 1, was identified with the initial state, and the other four states were represented when the first bit was 0 and using a combination of the other two bits.
8. Design3: Build the NextState truth table, determining the next logical state from the current state and the values of the inputs. If you followed the advice above in state assignment, you can have 16 lines in the truth table for everything except the initial state, and can have single line abstracting what is required for the initial state.
9. Design4: Build the Output truth table. Recall that, since outputs are determined only by the current state, the number of rows in the Output truth table is only based on the number of bits of storage. Don't forget to give the boolean expression for each Output based on the bits of state storage.
10. Implementation1: If we use D Flip Flops instead of J-K Flip Flops, the desired next state of a D Flip Flop is achieved with a single value, instead of two. This can simplify the K-Maps needed for each Flip Flop (1 per, instead of 2 per).
11. Implementation2: You are welcome to use gates that take 2 *or more* inputs, and can specify on each input whether it should be negated or not.
12. Submission: Use **submitbox** to upload your `multiply.circ` circuit file, and a PDF with your FSM picture, the state assignment, the NextState truth table, and K-Map to boolean algebra for each Flip Flop input. Only one submission per team. Make sure your names are included at the top level circuit.