

CS152 Lab 0: Multiplier Design Doc – 9/15/2003

Jack Kang (cs152-ta),
John Gibson (c152-tb),
Kurt Meinz (cs152-td)
TA: Dave

ABSTRACT:

The purpose of this assignment is to design and implement a 32bit x 32bit hardware multiplier module using variable-cycle algorithm.

We will start by defining the necessary submodules, datapath, and control in verilog and schematic, then we will test each module using automated, intelligent verilog behavioral tests, and finally we will push the design onto the board. Using timing analysis, we will see if the speedup over a predesigned fixed-cycle multiplier is as expected.

DESIGN DECISIONS:

We have decided to stick close to the provided framework, and will keep all signals as described in the framework. We plan to create a variable-cycle multiplier, where each bit of the multiplier will take 1 cycle to process if it is a 0 (just a shift in the product), and two to process if it is a 1 (one to shift the product and start the adder, one extra cycle to let the adder finish). Our multiplier will perform these steps once for each bit in the multiplier, starting at the left-most bit and shifting the multiplier (and product) left 32 times.

In this scenario, we think that multipliers with lots of zeros (the common case) will run considerably faster since there will be no waiting for the adder.

HIGH LEVEL IMPLEMENTATION

Submodules:

1. module **Reg64_RCE** (clk, rst, write_enable, [63:0]d, [63:0]q)

A basic 64 bit register with synchronous reset and clock enable. This will hold the product.

2. module **Reg32_RCE** (clk, rst, write_enable, [31:0]d, [31:0]q)

A basic 32 bit register with synchronous reset and clock enable. This will hold the multiplier and multiplicand.

3. module **Count32_RCE** (clk, rst, clock_enable, done)

A fast 32-bit counter (using shift registers) that has an synchronous reset and clock_enable. This module will start counting down from 32 whenever *clock* and *reset* are both asserted, and will assert done after $AND(clock, \overline{clock_enable})$ has been asserted exactly 32 times. This module will track how many bits of the multiplier we have processed already. It must be done using a shifter since an adder would be too slow.

4. module **ShiftL32** ([31:0]in, [31:0]out)

This module will put $in \ll I$ on the out wires (no overflow). It will be used to shift the multiplier register as necessary.

5. module **ShiftL64** ([63:0]in, [63:0]out)

This module will put $in \ll I$ on the out wires (no overflow). It will be used to shift the product register as necessary.

6. module **Mux2** (sel, in0, in1, out)

A basic 2->1 multiplexor. This will be used to select which input to put into the product register: the result of the add (for a 1 bit in the multiplier) or the product shifted by 1 (for a zero in the multiplier). It will also be used to select which input to use for the D of multiplier register: the input lines or the shifter.

7. module **Mult32_R** (clk, rst, mcand, mplier, product, done)

Our toplevel module will take a clock, synchronous reset and two 32 bit inputs (*mplier* and *mcand*) and will put the product in *product*. *done* will be asserted after the calculation has finished (which can be a variable number of cycles due to our design)

Datapath

Please see our enclosed Xilinx schematic or hand-drawn picture of our datapath. We have included all major modules as well as all of the interconnections between them (just like in the book).

Control

Our control is a finite state machine written behaviorally. Below is a rough sketch of the states. We have also included a FSM diagram. The state priority is in parens, with lower numbers meaning higher priority.

RESET_ASSERTED (0):

1. Action: Latch in inputs, reset *counter* and *product*, unassert *done*.

2. Entered if: *reset* asserted.

DONE_ALREADY (1):

1. Action: Do nothing and wait for next *reset*. *product* will already be correct.
2. Entered if: shift-counter output going high.

MULT_BIT_IS_0 (2):

1. Action: Select the straight-shift line on the multiplexor to the *product* register. Select the shift line on the input to the *multiplier* register. Assert clock enable on the *counter*, *product*, and *multiplier*.
2. Entered if: Left-most bit of *multiplier* is a 0.

MULT_BIT_IS_1_START (4):

1. Action: Do nothing. Allow entire cycle for the *shifted product* and the *multiplicand* to be added together.
2. Entered if: Left-most bit of *multiplier* is a 1;

MULT_BIT_IS_1_FINISH (3):

1. Action: Select the adder line on the multiplexor to the *product* register. Select shift line on input to the *multiplier* reg. Assert clock enable on the *counter*, *product*, and *multiplier*.
2. Entered if: Last state == MULT_BIT_IS_1_FINISH

TESTING:

We will write test benches for all of the subcomponents using automated-testing methodology.

We will do minor, non-automated testing of the datapath (sans control) by presetting and checking values using Modelsim, but we think that it will be advantageous to insert the control as quickly as possible.

For functional testing of the multiplier as a whole, we plan to do two types of cases:

1. Corner cases, where we feed interesting values to the multiplier, let it run, and make sure that it receives the right answers. We plan to test every combination of {0, FFFF FFFF, 8000 0000, 0000 0001, and 8000 0001} PICK 2 for our multiplier and multiplicand.
2. Probabilistic testing, where we have verilog generate 1000 random multipliers and multiplicands, and runs the multiplier on each one.

TIMING

A normal multiplier that requires 32 10ns clock cycles to perform the multiplication will require 320ns in total.

Assuming that our multiplier has clock cycles that are twice as fast (5ns; a reasonable assumption since we have allotted 2 clocks for the adder), then if we assume that, on average, $\frac{1}{2}$ of the digits in the multiplier are 0s, we get the following results:

FRACTIONenhanced = 0.5 (The zero cycles are sped up)

SPEEDUPenhanced = 2 (The zero cycles complete twice as quickly)

By Amdahl's Law, the total speedup is $1 / ((1 - \text{FRACTIONenhanced}) + (\text{FRACTIONenhanced} / \text{SPEEDUPenhanced})) \rightarrow 33\%$

However, if we assume that $\frac{3}{4}$ of the bits of multiplier are 0s (all of the upper 32, and $\frac{1}{2}$ of the lower 32), we get the following numbers:

FRACTIONenhanced = 0.75 (The zero cycles are sped up)

SPEEDUPenhanced = 2 (The zero cycles complete twice as quickly)

SPEEDUPoverall $\rightarrow 60\%$

TIME LINE AND DIVISION OF LABOR

Tuesday: submodules (3 hours), submodule testing (3 hours) and datapath (1 hour)

Wednesday: datapath testing (1 hour)

Thursday: Control (4 hours)

Friday: Testing (4 hours)

Saturday: Project Writeup and Submission (2 hours)

This will give us 2 full days (Sunday and Monday) to fall back on.

Kurt will be our designated spokesperson for this lab. He will communicate with the TAs any questions we may have, and schedule any needed checkoff times.

We will divide the submodules and their testing by person (Jack gets the regs, Kurt gets the shifters and muxes, John gets the counter). Then we will all sit down and hash out the datapath implementation and testing as well as control implementation. Jack and Kurt will then write the corner-case tests, while John write the probabilistic checker.

END OF DESIGN DOC