CS 740, Fall 1995
Homework Assignment 3
Assigned: Oct. 17
Due: Thurs., Oct 26, 1:00PM

1. Policy

You may work in a group of up to 3 people in solving the problems for this assignment. However, you must prepare your own write-up, and you must identify your collaborators in your write-up.

2. Logistics

Any clarifications and revisions to the assignment will be posted on Web page *assigns.html* in the class WWW directory.

For this assignment, you will want to retrieve the files in the directory

/afs/cs.cmu.edu/academic/class/15740-f95/public/asst/asst3

You only need to hand in a hard copy document for this assignment. Formatted text is preferred to hand written. If some question arises during grading regarding the correctness of your code, you may be asked to submit an electronic version.

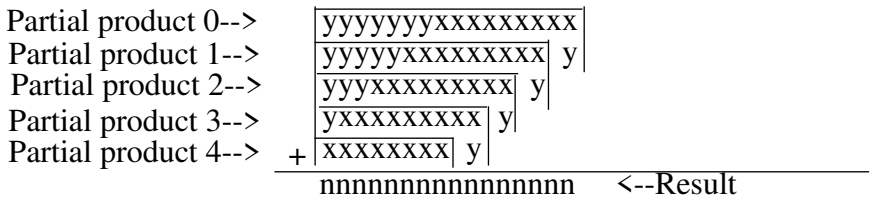
3. Task 1: Multiplier Design Based on Booth Recoding.

In homework assignment 1, you used the **multu** assembly code to do unsigned 32x32 bit multiplication and get the 64-bit result. Now, your task is to implement an **unsigned** 32x32 multiplier based on radix-4 Booth recoding. Suppose the multiplier is $Y(y_{31}y_{30}....y_2y_1y_0)$ and the multiplicand is X. Figure A.25 in H&P text book shows the multiples of X to be used for radix-4 Booth recoding. Now, "i" is from 0 to 16 and y_{-1}, y_{32}, y_{33} are 0.

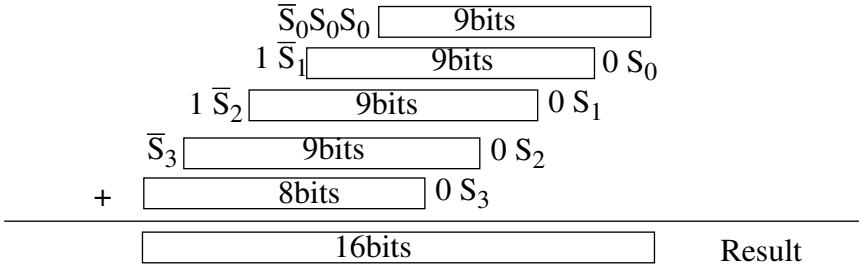
(A) Sign Extension.

When you use Booth recoding to generate the partial products (in **two's complement** form), you have to consider the sign of the partial product and do the sign extension. The following figure shows the partial products for an 8x8 multiplier, where y represents the sign of the partial product. If y=0, the vector xxxxxxxxx represents the partial product; Otherwise it represents the 1's complement of the partial product. Here, the partial products, except the last one which is always positive, are represented by an "L" shape, because of 2's complement representation of the partial products. When the partial product is negative, its 2's complement representation can be obtained

by adding 1 to the 1's complement of the partial product. In other words, we are computing the 2's complement form of the partial products by injecting the sign bit (1 if negative) into the next addition stage.



There is a method to simplify the signe extension and represents the partial product by less than 37 bits (rather than 64 bits). The method will reduce the size of CSAs, if you use the Wallace tree to sum up the partial products. For a 8x8 multiplier, the sign extension method is shown as following:



where the S_i represents the sign of partial product *i, i=0 to 3,* and $\overline{S}_i$ is the negation of S_i. The 9-bit vector can represent the partial product 0, the multiplicand, or 2 times the multiplicand. For example, if the partial product is -2 times the multiplicand, the sign is 1 and the 9-bit vector contains the 1's complement of 2 times multiplicand. Using this method, it will product the correct result for multiplication. You can extend it to a 32x32 multiplier by increasing the 9 bits to 33 bits and the number of partial products from 5 to 17.

Now, your task is to justify why this method works.

(B) 32x32 Multiplier Based Radix-4

Based on the sign extension method in task 1.A, your task is to write a C program to implement an unsigned 32x32 multiplier based on radix-4 Booth recoding, a CSA which can be used to sum the 17 partial products, generating two vectors carry (C) and sum (S), and a conventional ripple carry adder which produces the final result by adding C and S. In other word, you should implement some basic elements as procedures (e.g., a 64-bit adder, a recoding block, etc.), and then implement multiplication using a loop that iterates 15-17 times (which depends on your design) to generate C and S. Each iteration would recode some bits of the multiplier, generate a partial product (represented by your sign extension method), and update the appropriate bits of C and S using the CSA. After exiting the loop, C and S would be added to form the final product.

You should write a write a resizeable multiplier such that you can fully test your multiplier by resizing it to 8x8 multiplier and verifying it with all of the possible input combinations. File asst3/m_8x8.test contains all of the possible input combinations for 8x8 multiplier. You can use the file *asst3/m100.test* to test your implementation for 32x32 multiplier.

4. Task 2: Floating-Point Rounding

Do all of the questions in the Exercise A.16.

5. Task 3: Cache Measurements

In this part of the assignment, you will devise a series of programming experiments to determine the following properties about the data cache for the workstation of your choice:

1) Is it physically or virtually addressed?

2) What is the associativity? If it is physically indexed and set associative, it would be very difficult to determine the cache parameters.

3) What is the line size?

4) What is the miss penalty for reads?

5) If not direct mapped, what can you determine about the replacement policy? It may be difficult to precisely determine the replacement policy. Most caches implement LRU (Least recently used), but some only approximate LRU or use a pseudorandom replacement policy. If your cache uses LRU or FIFO(First-in First-out), you should be able to verify it.

6) What is the cache size?

You can use whatever machine you want, but the problem is much harder on some machines than others and the series of tests used depend somewhat on the particular cache design. You are free to consult any relevant technical material, but this should only be to confirm your experimental results.

For your convenience, the files *cycle-time.c* and *cycle-time.h* package up the timing facilities in a helpful way. These files can be found in the *asst3* directory. Comments in the .h file describe how to use this code.

Hints:

You can determine these characteristics by writing short C programs that create worst or best case behavior near the various cutoffs in the cache parameters. In other words, you can determine these parameters based on timing code that can run purely at the user level. You may also make use of the Unix ***pagesize*** command to determine the virtual memory page size. The order of the tests listed above provides a possible sequence for determining the parameters -- the values from prior

tests can be useful in determining the remaining ones. Note that these figures may vary from one model workstation to another. Note also that you must be very careful in your measurement technique. You should make your measurements on a lightly loaded machine.

The biggest challenge in this task is that you have no control over the mapping of virtual memory pages to physical ones. In fact, you have no way of even knowing what the mapping is! The trick then is to take a large sample of pages and determine their partitioning into "equivalence classes" i.e., groups of pages that map to the same cache region. This is not difficult to do with a direct-mapped cache. For a cache that is virtually indexed, or that (like the PowerPC) uses just the low order page offset bits to index the cache, it would be reasonable (and interesting!) to measure the performance of a set-associative cache. If your machine has 2-level caches, you should consider the effects of level 2 cache.

Hand In

Explain how you test for the different parameters, what the measured results were, how you infer the different parameters from your measurements, and how they compare to the answers you may have found in reference materials. You must also specify the exact CPU model in your writeup.