COEN 313 Review WS

1. Write using RISC V assembly a small for loop that will add double (8 bytes) elements of arrays A, B into C. That is the loop will execute C[i] = A[i]+B[i] for N elements from i= 0 to N-1.

   Assume N (or N-1 if that suits you better) is stored in X2. Assume the base of arrays A, B, and C are stored in X3, X4, and X5. Assume that i (or 8*i if you want) is stored in X6. You can have separate registers for both i and 8*i (no problem at all). Remember is a loop and not a complete function.

   High-level code (pseudo):

   ```
   double [] A, B, C;
   for i in N:
       C[i] = A[i] + B[i];
   ```

   Assembly:

   ```
   Loop:
       // assume that X2 starts with value N, X6 starts with value i = 0
       // assume that X3, X4, X5 hold double arrays A, B, C
       ADD X10, X3, X6 // reg10 stores curr pos in A during traversal
       ADD X11, X4, X6 // reg11 stores curr pos in B during traversal
       ADD X12, X5, X6 // reg12 stores curr pos in C during traversal
       LD X7, X10 // load double val from array B to reg7
       LD X8, X11 //  load double val from array B to reg8
       ADD X9, X7, X8 // add these 2 values together and store in reg9
       SD X9, X5 // store sum as double in array C
       ADDI X6, X6, 8 // increment i by 8
       BLT X2, X6, Loop // loop if i < N
   ```

2. In your favorite language write a function unsigned int index(unsigned int cSize, unsigned int lSize, unsigned int assoc, unsigned int address): the function takes the cache size in bytes, line size in bytes, associativity and address. It returns the index of the set where the address contents might be present. The syntax given in the question is c-like. But you can choose your language. You don't have to run the program (but I personally would out of curiosity).

   ```
   findIndex(cSize, lSize, assoc, addr):
           indSize = cSize / lSize / assoc; // obtain the size of the index we are looking for
           offset = log(lSize, 2);  //obtain the size of the offset by computing $\log_2$(lSize)
           ind = addr >> offset; // right-shift to remove the offset
           ind = ind << len(ind) - indSize; // left-shift to isolate the index of the set
           return ind;
   ```

3. For the special case where assoc == 1, use the function of Q2 to check whether there is a hit or a miss. Assume there is a function bool isValid (index) which states whether the cache line is valid or not, and a function unsigned int getTag(index) which returns the tag of the line.

   ```
   ind = findIndex(cSize, lSize, 1, addr);
   if !isValid(ind): print ‘miss’; // print a miss if the valid tag is false
   cTag = getTag(ind);
   aTagSize = addr - len(index) - log(lSize, 2); // obtain the size of the tag of the address by subtracting the size of the index and offset from the address
   aTag = aTag >> addr - aTagSize; // left shift to isolate the address tag
   if (aTag == cTag): print ‘hit’ // check if the tags are equal
   else: print ‘miss’
   ```

   In both Q2, and Q3, assume that input parameters are already valid, i.e. everything is a power of 2, cache size >= (line size x associativity) and so on. In other words don't worry about wrong input.

4. What is the whole deal with Scoreboarding? What does it add beyond basic pipelining?

   According to the textbook, Scoreboarding is “a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependencies.” Scoreboarding monitors data dependencies and only allows instructions to pass through once it is determined that they won’t cause hazards, such as RAW.