Please answer questions 1 and 2 on this paper and put all other answers in the blue book.

1. True/False. Please circle the correct response.
   a. T In the C and assembly calling convention that we used for this class, arguments to a function call are pushed in reverse (i.e. right-to-left) order.
   b. F In C, for the expression \( x | \text{THE\_MASK} \), where \( \text{THE\_MASK} \) has at least one bit that is not zero, the result will be zero if all the bits of \( x \) are zero.
   c. T In x86 assembly, the instruction \texttt{mov eax, [ebp+12]} (or \texttt{movl 12(%ebp), %eax} in AT&T syntax) will add 12 to the value contained in the ebp register in order to compute a memory address.
   d. T In x86 assembly on a 32-bit machine, the instruction \texttt{pop eax} adds 4 to the value of the esp register.
   e. F For a read operation on a direct-mapped cache, if the tag in the specified address matches the tag in the corresponding cache entry, then a cache hit results – no matter what the status of any other bit in the cache entry is.
   f. F In the NRU (not-recently-used) cache replacement algorithm that was used in the L1 cache in the class project, it was impossible for the reference bit of a cache entry to be 0 but the dirty bit to be 1.
   g. F The use of a write-through cache is slower than a write-back cache because, using a write-through cache, the CPU must wait until the data is written to main memory before proceeding with the next instruction.
   h. T A multiplexer uses N select lines to select from \( 2^N \) input lines.
   i. T A 32-bit adder can be used for subtraction if each bit of the second operand is first sent through a (32-bit) NOT gate and the carry-in to the adder is set to 1.
   j. T In an unclocked latch, setting both the S and the R inputs to 0 will cause the output Q to retain its current value.

2. Write the number AB31 hex in binary: 1010 1011 0011 0001.
   b. Write the number 73 decimal in hex: 49 and in binary: 0100 1001.
   c. \( \log 16G = \underline{34} \).
   d. In order to access all bytes in a 64MB memory, an address must have at least \underline{26} bits.

3. Write a C procedure, \texttt{int foo(int x)} that returns the index of the most significant bit of \( x \) whose value is 1. For example, if bit 15 of \( x \) is the most significant bit of \( x \) whose value is 1, then \texttt{foo} should return 15. If no bits are 1, then \texttt{foo} should return -1.
int foo(int x)
{
    int i;
    for(i=31;i>=0;i--) {
        if (x & (1<<i))
            return i;
    }
    return -1;
}

4. Write an x86 assembly procedure bar that takes two parameters, an integer n and a pointer p to an integer array (i.e. p is the address of the start of the array), and adds up the first n integers of the array and returns the result.

Intel Syntax

    _bar:
    push    ebp
    mov     ebp,esp
    mov     eax,0          #result is initially 0
    mov     ecx,[ebp+8]    #ecx holds n
    mov     edx,[ebp+12]   #edx holds p
    TOP:
         cmp     ecx,0    #if n=0
         je       DONE    #jump out of loop
         add     eax,[edx] #result = result + *p
         dec     ecx      #n--
         add     edx,4    #p++
         jmp     TOP      #jump to top of loop
    DONE:
         pop     ebp
         ret

AT&T Syntax

    _bar:
    push    %ebp
    mov     %esp,%ebp
    mov     $0,%eax          #result is initially 0
    mov     8(%ebp),%ecx     #ecx holds n
    mov     12(%ebp),%edx    #edx holds p
    TOP:
         cmp     $0,%ecx      #if n=0
         je       DONE       #jump out of loop
         add     (%edx),%eax  #result = result + *p
         dec     %ecx        #n--
         add     $4,%edx      #p++
         jmp     TOP         #jump to top of loop
5.
   a. In the class project, when running the code in test_memory_subsystem.c, the simulated cache performance was much worse when addresses were generated randomly than when sequences of consecutive addresses were generated. Why is that?

   Having multi-word cache lines (as in the class project) exploits spatial locality, i.e. the property that when a location in memory is accessed, a neighboring location (i.e. in the same cache line) is likely to be accessed soon. In the test code that generated addresses randomly, there was no spatial locality. Successive random memory accesses are likely to generate cache misses, whereas as successive accesses to consecutive memory locations are likely to result in cache hits (being in the same cache line).

   b. On a 32-bit machine, suppose there is a 2MB, 2-way set associative cache with 16 words per cache line. Further, suppose that an instruction issues a memory address for loading a single byte from memory into a register. Indicate how the various bits of the address are used by the cache hardware to determine whether a cache hit has occurred and, if so, to return the requested byte.

Since each set contains two cache lines, and each cache line contains 16 words, a set occupies 32 words = 128 bytes = 2^7 bytes. Since the entire cache contains 2MB = 2^{21} bytes, there are 2^{21}/2^7 = 2^{14} sets in the cache. Therefore, log 2^{14} = 14 bits are required to for the set index. Since there are 16 words per cache line, log 16 = 4 bits are required to select the desired word from the cache line. Since there are 4 bytes per word, log 4 = 2 bits are required to select the desired byte from the word. The rest of the bits are used for the tag. Therefore, an address is partitioned as follows:

Bits 0-1: byte offset within word

Bits 2-5: word offset within cache line

Bits 6-19: set index

Bits: 20-31: tag

   c. Suppose there is a computer with a memory hierarchy consisting of an L1 cache, an L2 cache, and main memory. Suppose further that an access to L2 cache or main memory reads or writes an entire cache line at a time, but an access to L1 cache reads or writes a single word at a time. If an instruction generates the address of a word in memory to load into a register, describe how the caches and main memory are accessed in order to retrieve the specified word. That is, describe every possible scenario (depending on a hit or miss at the various levels of the hierarchy).

   (Note: These are the steps described in class and implemented in the class project)

Step 1. Issue the read to the L1 cache. If it results in a cache hit, then the desired word is returned by the L1 cache and the read is completed. Otherwise, if a cache miss results, go to step 2.
Step 2. Issue a read to the L2 cache. If an L2 cache hit occurs, then the L2 cache hit returns an entire cache line. This cache line is inserted into the L1 cache and the read operation on the L1 cache is re-issued and is sure to generate a cache hit and the read will complete. Otherwise, if an L2 cache miss occurs, go to step 3.

Step 3. Issue the read to main memory. Main memory will return a cache line, which is then inserted into L2 cache and into L1 cache. The read to L1 cache is then re-issued and is sure to generate a cache hit and the read will complete.

6.  
a. Build, from AND, OR, and NOT gates, a circuit that represents the two-bit “less-than” function. That is, it has two two-bit inputs, A and B, and a single one-bit output, R, such that R is true when A < B. [Hint: enumerate the possible inputs for which the output is true.]

One way to do this is to write out the truth table, where the four inputs are A1 A0 B1 and B0, then generate the Boolean formula for those rows where the result is 1, simplify if possible, and then build the circuit according to the Boolean formula. Another way is to notice:
- If A = 00, then A < B if either bit of B is 1
- If A = 01, then A < B if B1 is 1
- If A = 10, then A < B if B=11

Encoding the above observations as a Boolean formula gives:
\[
R = (\neg A_1 \land \neg A_0 \land (B_1 \lor B_0)) \lor
(\neg A_1 \land A_0 \land B_1) \lor
(A_1 \land (\neg A_0) \land B_1 \land B_0)
\]
Although this could be simplified a bit further, it is easy enough to build as is:

![Circuit Diagram](image)

b. As you saw in class, a clocked latch is built from an unclocked latch as shown below. Why are flip-flops used for storing bits in a CPU rather than clocked latches?

![Flip-Flop Diagram](image)
With a clocked latch, the value being stored (and output at Q) can change as long as the clock is up. In a system, such as the one illustrated below,

![Diagram of sequential and combinational circuits](image)

where the data stored in a sequential circuit is being used as input to a combinational circuit (such as an ALU) and the output of the combinational circuit is being sent back to the combinational circuit for storage, if the sequential circuit consisted of clocked latches then, as long as the clock was up, the data coming in to the sequential circuit could overwrite the data that was stored—while the combinational circuit is still performing its computation. What is needed instead, and provided by flip-flops, is for the data coming in to the sequential circuit to be prevented from overwriting the old data until the clock falls (and presumably, the combinational circuit has completed the computation).

c. Build from gates, multiplexers, decoders, and/or adders (including 32-bit versions of each) a circuit that takes two 32-bit inputs, X and Y, and outputs the value of the larger of X and Y. That is, it computes the equivalent of the following C code:

\[
\text{output} = \text{X} > \text{Y} ? \text{X} : \text{Y};
\]

The comparison is accomplished by performing a subtraction, Y-X, and then testing if the result is negative by using the highest bit of the result as the selector for a multiplexer whose inputs are X and Y.