ASSIGNMENT 5 ANSWERS

1. - Main idea: Starting at the entry, traverse the flowgraph looking for nodes that start EBBs. For each one of those, collect the remaining blocks that should be in the same EBB.

```
StarterNodes = {entry}   // (global) set of nodes with multiple predecessors.
                         // Initially it contains the flowgraph entry
-----------------------

Procedure FIND_ALL_EBBS
    EBBsets = { }

    while StarterNodes != { }
        n = an element of StarterNodes
        StarterNodes = StarterNodes - {n}
        EBBsets = EBBsets + FIND_EBB(n, {n})  // + is a union operation
End Procedure

---------------------

Procedure FIND_EBB(n, Ebb) // n is the "top" node of the EBB,
                           // Ebb is the set of n's EBB nodes
    while Ebb changes
        for each successor s of n
            if s has exactly one predecessor
                Ebb = Ebb + FIND_EBB(s)
            else if s is not already in StarterNodes
                StarterNodes = StaterNodes + {s}

---------------------
```

   - Performing iterative data flow analysis on extended basic blocks defeats the purpose. If we choose this method, we might as well go straight to global analysis. Instead, we will do the optimization the same way as with basic blocks by handling the collection of blocks in the EBB as a single basic block. This implies that we have to impose some kind of ordering. For example, if an EBB contains a block A that branches off to B and C, then the resulting "single" block could be A followed by B followed by C, or A followed by C followed by B. A preorder or topological ordering should be sufficient.

     A disadvantage of this method is that the ordering we pick may be such that certain opportunities for optimization are lost at this time. However, these will still be "caught" during global common subexpression elimination.
   - It is more beneficial. Basic blocks may be too small for significant improvements. If we work across basic blocks, we may be able to find candidates for delay slots from preceding (or following) blocks. Note that when looking at branch destinations for delay slot candidates, we would preferably look for an instruction that occurs in both and can be moved up. Or, we could find one performed only in one branch (here we could make good use of profiling information about execution frequencies), and move it up after renaming any registers it involves to avoid dependence-related problems.
   - We may be able to find more dead code.

2. Even though it is not necessary, since global CSE can identify and eliminate all common subexpressions, we can speed up the global one, by performing local CSE on the fly during intermediate code generation.

3. This optimization is called *downward store motion*. gcc performs it after global common subexpression elimination if the option `-fgcse-sm` is enabled. Note that we don't move the evaluation `d=b+i` outside the loop, but the instruction that stores the value to `d`.

   - We must know that the loop will be executed at least once. Note that in general, when performing this optimization, we will not need to place the store in both destinations of a branch, but only in the one where the lhs is still live.
   - Intermediate values of d are kept in a register and the final value is stored in memory just once, outside the loop. This means that the loop will require the use of an additional register (if the allocator didn't have one assigned already). In general, store motion will increase the demand for registers.
   - In order to identify candidates for store motion, we would need to collect aliasing information. In other words, we need to determine whether a storage location can be modified in more than one ways inside a loop (e.g. via a pointer). If it can be modified in only one way, then the `store` can be moved out. Note that references to the variable have no effect on our decision. For example, in the sample code, we could have had `d=d+i` and still performed store motion.

     Once we have a selection of candidates, we can traverse the loop and move them outside. As mentioned above, if we have live variable information, we may use it to determine whether we need to place the store in both branch destinations.

4. Refer to the flow graph in figure 1 for the names of the basic blocks.

   - Back edge (B4, B3) defines loop {B3, B4}
     Back edge (B5, B3) defines loop {B2, B3, B4, B5}
   - Figure 1 shows the GEN/KILL sets for each basic block as well as the initial values of the OUT sets. The IN/OUT sets after each pass are shown in the table. Note that there will also be a second pass which will not result in any changes in the set and thus cause the algorithm to terminate.
   - There are no opportunities for global common subexpression elimination. Unfortunately, this is due to a typo in block B3. The instruction $b = b * d$ was supposed to be $d = b * d$. Then $a + b$ would not be killed and we would evaluate it only once in B2 and then use the value in B3 and B5.
   - Figure 2 shows the DEF/USE sets for each basic block. The IN/OUT sets are shown in a table. Three passes are sufficient. The third pass will not make any changes and cause the algorithm to terminate. Note that we chose to traverse the blocks in an order that would minimize the total number of passes.

Block	Pass 1	
	in	**out**
B1	∅	∅
B2	∅	{a+b,c-a}
B3	{a+b,c-a}	{a-f,c-a}
B4	{a-f,c-a}	{a+b,a-f,c-a}
B5	{c-a}	{a-f,c-a,c-d}
B6	{a-f,c-a,c-d}	{a-d,c-d}
B7	{a-f}	{a-f,f+h}
B8	{a-d,c-d}	{a-d,c-d,f+h}

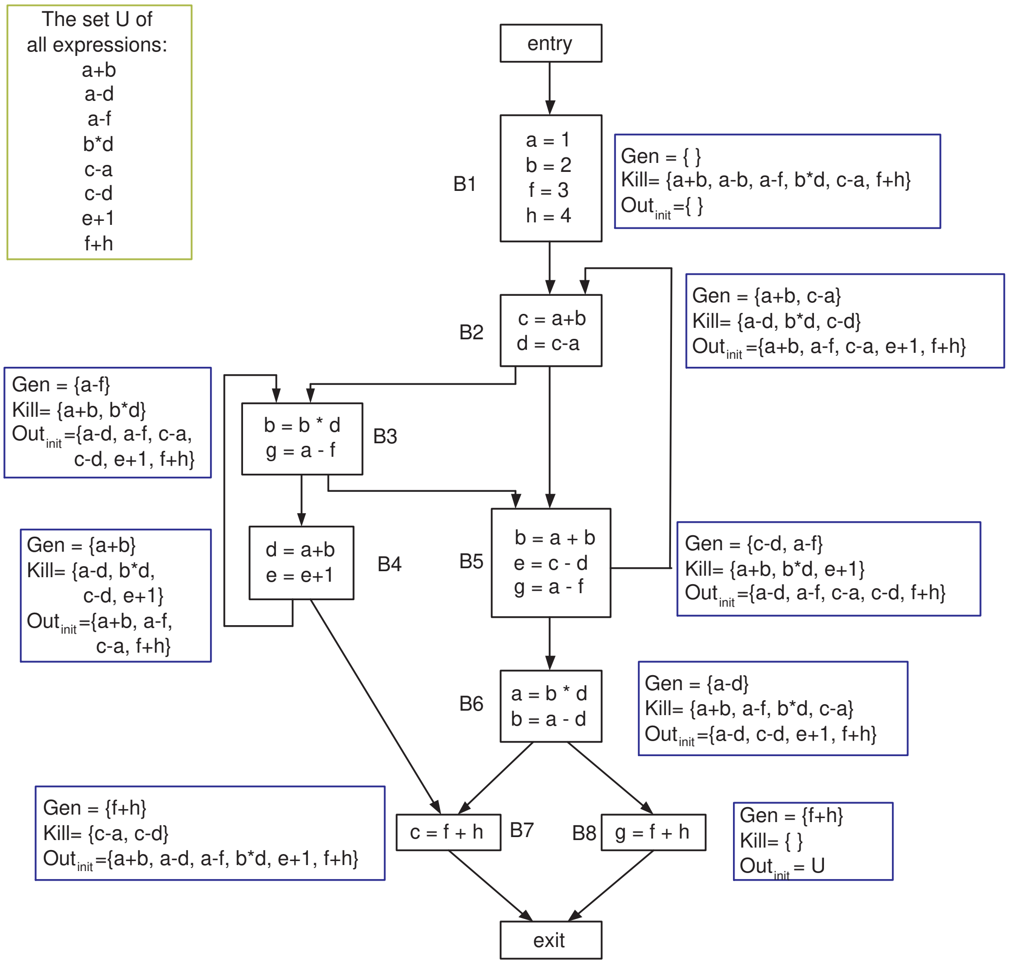

Figure 1: Problem 4, available expressions

Block	Pass 1		Pass 2	
	out	**in**	**out**	**in**
B8	∅	{f,h}	∅	{f,h}
B7	∅	{f,h}	∅	{f,h}
B6	{f,h}	{b,d,f,h}	{f,h}	{b,d,f,h}
B5	{b,d,f,h}	{a,b,c,d,f,h}	{a,b,d,e,f,h}	{a,b,c,d,f,h}
B4	{f,h}	{a,b,e,f,h}	{a,b,c,d,e,f,h}	{a,b,c,e,f,h}
B3	{a,b,c,d,e,f,h}	{a,b,c,d,e,f,h}	{a,b,c,d,e,f,h}	{a,b,c,d,e,f,h}
B2	{a,b,c,d,e,f,h}	{a,b,e,f,h}	{a,b,c,d,e,f,h}	{a,b,e,f,h}
B1	{a,b,e,f,h}	{e}	{a,b,e,f,h}	{e}

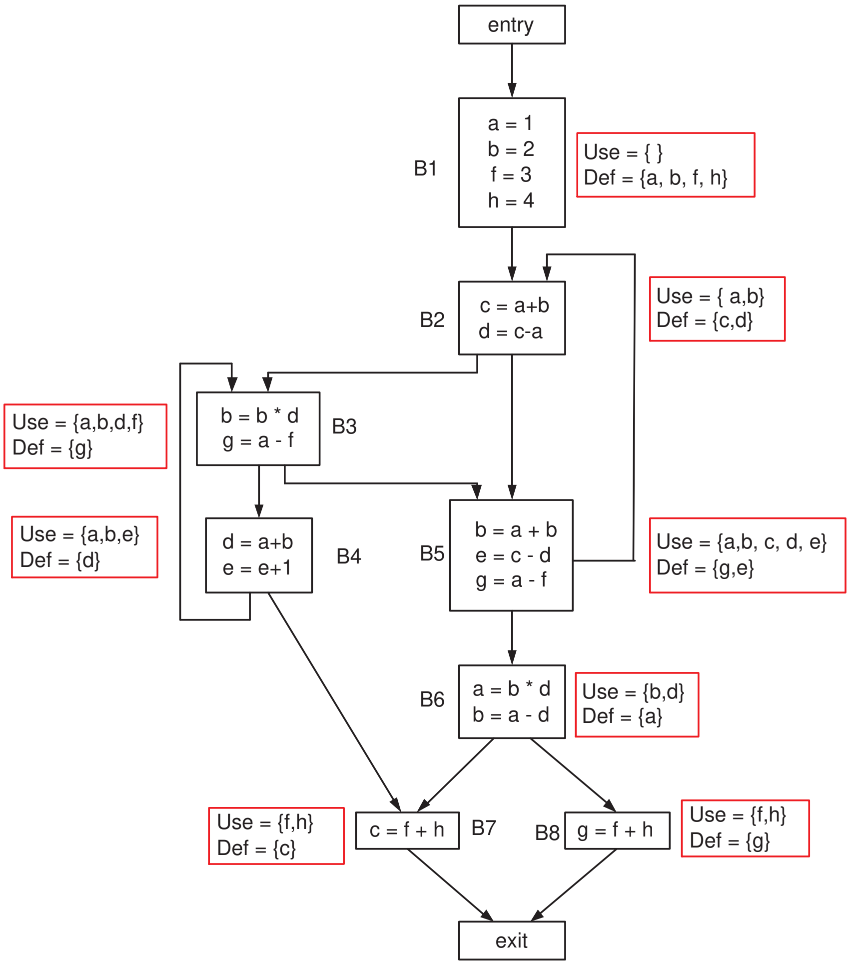

Figure 2: Problem 4, live variables