Problem 1:  80 points, SEVERAL parts.

a. Using the intermediate, construct and "draw" a control flow graph (CFG) for gigo.

B1:

```plaintext
sym1001 = 0;       // sum
sym1002 = 1;       // prod
sym1003 = 0;       // diff
*C = 3;            // "simulated" store, C[0] = 3;
sym1004 = 1;       // i
```

B2:

```plaintext
sym1005 = sym1004-1;       // takes the place of computing offset;
sym1006 = B[sym1005];     // "simulated" load
sym1007 = sym1004;
sym1008 = A[sym1007];      // "simulated" load
sym1011 = sym1008 + sym1006;
sym1012 = sym1004;
C[sym1012] = sym1011;       // "simulated" store
sym1013 = sym1004;
sym1014 = B[sym1013];
sym1015 = sym1004-1;
sym1016 = A[sym1015];
sym1017 = sym1016 - sym1014;
sym1021 = sym1004;
B[sym1021] = sym1017;
sym1022 = sym1004-1;
sym1023 = C[sym1022];
sym1024 = sym1004;
sym1025 = C[sym1024];
sym1026 = sym1025 + sym1023;
sym1001 = sym1001 + sym1026;
sym1027 = sym1004;
sym1031 = C[sym1027];
sym1002 = sym1002 * sym1031;
sym1032 = sym1002 - sym1001;
sym1003 = sym1003 + sym1032;
sym1004 = sym1004 + 1;
if( sym1004 < 1024 ) goto L1;
B3:    return sym1003;
```

CFG edges are:  {B1-->B2, B2-->B2, B2->B3}

b. Show the dominator tree for the control flow graph of a.

Dominator edges are:  {B1-->B2, B2-->B3}
c. Perform the "standard" optimizations common subexpression elimination (CSE), copy propagation, and dead code elimination on the CFG for gigo. (You can do this by hand. No need to perform live variable analysis yet.) DO NOT change the algorithm or "data" structures of the program. Just do CSE, copy prop, and dead code elimination.

1:    sym1005 = sym1004-1;
2:    sym1006 = B[sym1005];
3:    sym1008 = A[sym1004];
4:    sym1011 = sym1008 + sym1006;
5:    C[sym1004] = sym1011;
6:    sym1014 = B[sym1004];
7:    sym1016 = A[sym1005];
8:    sym1017 = sym1016 - sym1014;
9:    B[sym1004] = sym1017;
10:   sym1023 = C[sym1006];
11:   sym1026 = sym1011 + sym1023;
12:   sym1001 = sym1001 + sym1026;
13:   sym1002 = sym1002 * sym1011;
14:   sym1032 = sym1002 - sym1001;
15:   sym1003 = sym1003 + sym1032;
16:   sym1004 = sym1004 + 1;
17:   if( sym1004 < 1024 ) goto L1;

d. Compute live variables for each intermediate statement. Describe how you go about doing this as well as showing the resulting sets.

1:    sym1005 = sym1004-1;                      {1001-1004}
2:    sym1006 = B[sym1005];                   {1001-1005}
3:    sym1008 = A[sym1004];                   {1001-1006}
4:    sym1011 = sym1008 + sym1006;           {1001-1005}
5:    C[sym1004] = sym1011;                   {1001-1005, 1011}
6:    sym1014 = B[sym1004];                   {1001-1005, 1011}
7:    sym1016 = A[sym1005];                   {1001-1004, 1011, 1014}
8:    sym1017 = sym1016 - sym1014;           {1001-1004, 1011}
9:    B[sym1004] = sym1017;                   {1001-1004, 1011}
10:  sym1023 = C[sym1006];                   {1001-1004, 1011}
11:  sym1026 = sym1011 + sym1023;           {1001-1004, 1011}
12:  sym1001 = sym1001 + sym1026;           {1001-1004, 1011}
13:  sym1002 = sym1002 * sym1011;           {1001, 1003-1004}
14:  sym1032 = sym1002 - sym1001;           {1001-1004}
15:  sym1003 = sym1003 + sym1032;           {1001-1002, 1004}
16:  sym1004 = sym1004 + 1;                 {1001-1003}
17:  if( sym1004 < 1024 ) goto L1;          {1001-1004}

Algorithm:

for each BB:
live = BB->live-out
for each stmt of BB, in reverse order:
stmt->live = live - stmt->def
live = stmt->live U stmt->use
e. Perform Briggs style graph-coloring register assignment on the function assuming that the "architecture" has 16 general purpose registers.

Here is a tabular view of the interference graph

<table>
<thead>
<tr>
<th></th>
<th>-01</th>
<th>-02</th>
<th>-03</th>
<th>-04</th>
<th>-05</th>
<th>-06</th>
<th>-08</th>
<th>-11</th>
<th>-14</th>
<th>-16</th>
<th>-17</th>
<th>-23</th>
<th>-26</th>
<th>-32</th>
</tr>
</thead>
<tbody>
<tr>
<td>-01</td>
<td>---</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
</tr>
<tr>
<td>-02</td>
<td>-X-</td>
<td>---</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
</tr>
<tr>
<td>-03</td>
<td>-X-</td>
<td>-X-</td>
<td>---</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
</tr>
<tr>
<td>-04</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>---</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
</tr>
<tr>
<td>-05</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>---</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
</tr>
<tr>
<td>-06</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>---</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
</tr>
<tr>
<td>-08</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>---</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
</tr>
<tr>
<td>-11</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>---</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
</tr>
<tr>
<td>-14</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>---</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
</tr>
<tr>
<td>-16</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>---</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
</tr>
<tr>
<td>-17</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>---</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
</tr>
<tr>
<td>-23</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>---</td>
<td>-X-</td>
<td>-X-</td>
</tr>
<tr>
<td>-26</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>-X-</td>
<td>---</td>
<td>-X-</td>
</tr>
</tbody>
</table>

Since each node has fewer than 16 edges, we can put nodes on the stack in any order, so

<table>
<thead>
<tr>
<th>Symbolic Reg</th>
<th>Physical Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>r1</td>
</tr>
<tr>
<td>02</td>
<td>r2</td>
</tr>
<tr>
<td>03</td>
<td>r3</td>
</tr>
<tr>
<td>04</td>
<td>r4</td>
</tr>
<tr>
<td>05</td>
<td>r5</td>
</tr>
<tr>
<td>06</td>
<td>r6</td>
</tr>
<tr>
<td>08</td>
<td>r7</td>
</tr>
<tr>
<td>11</td>
<td>r6</td>
</tr>
<tr>
<td>14</td>
<td>r7</td>
</tr>
<tr>
<td>16</td>
<td>r8</td>
</tr>
<tr>
<td>17</td>
<td>r5</td>
</tr>
<tr>
<td>23</td>
<td>r5</td>
</tr>
<tr>
<td>26</td>
<td>r5</td>
</tr>
<tr>
<td>32</td>
<td>r5</td>
</tr>
</tbody>
</table>
f. Build a precise, accurate data dependence graph (DDG) for the (single-block) loop. You should include latencies (all to be one cycle) and distance for each DDG edge.

The Loop statements are:

S1: \( C[i] = A[i] + B[i-1]; \)
S2: \( B[i] = A[i-1] - B[i]; \)
S3: \( \text{sum} += C[i] + C[i-1]; \)
S4: \( \text{prod} *= C[i]; \)
S5: \( \text{diff} += \text{prod} - \text{sum}; \)

The dependence edges are { True S1 --> S3, with latency, 1, and distance 0,
True S1 --> S4, with latency, 1, and distance 0,
True S2 --> S1, with latency, 1, and distance 1,
Anti S2 --> S2, with latency, 0, and distance 0,
True S3 --> S5, with latency, 1, and distance 0,
True S4 --> S5, with latency, 1, and distance 0 }

g. Assuming that each operation (load, store, arithmetic, if-test, branch) can be completed in a single cycle (hence no complicated latencies) and that we have 4 parallel functional units each capable of performing any single operation in one cycle

i) Show an instruction schedule for the acyclic loop. (i.e. a local schedule or a single iteration)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>7</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>8</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>17</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ii) Assume we wish to software pipeline the loop. What are the ResII, RecII, and minII for this loop?

ResII = 17 nodes / (4 nodes/instruction) = 5
RecII = 1, as there is no cycle in the Dependence Graph

BONUS: iii) Show a software pipelined schedule for the loop iff you can find a "legal" one that will lead to faster execution. If such a "shorter" schedule is not possible explain why.

We can’t come up with a shorter schedule than the "local" (acyclic) scheduled because the ResII = 5, which is also the length of the local schedule.
Problem 2: 10 points

A concern when implementing instruction scheduling (either acyclic or software pipelined) is how to treat the latency of memory accesses. Should one assume that each access is a cache hit, in order to form shorter schedules? Or should one assume a cache miss for a conservative approach? The conservative approach (assume miss) can lead to largely empty schedules since latency of a cache miss is considerable. On the other hand if we assume a cache hit, we have a long stall whenever a cache miss occurs, as the schedule has not allocated enough latency for the miss.

So, what are your thoughts? How should we decide which method to use? Should we use different policies for acyclic scheduling and software pipelining? Explain your reasoning process.

Since software pipelining can hide "any" latency, it makes sense to use the cache miss time. That way we know that the no cycles will be lost to stalls while waiting for resolution of a cache miss.

On the other hand, local scheduling likely will not have enough parallelism available to cover latencies, and since we expect more cache hits than misses, we should be optimistic and assume a cache hit. That way when we don’t get a miss we’ll get better performance. We get a miss we’ll stall but probably not much longer than the schedule would have been anyway if had assumed a miss.

(Actually, be using analysis and some loop manipulation we can make a very accurate guess as to the cache hits and misses and we get almost as good a pipelined schedules with MANY fewer registers needed than what the "miss" assumption leads to. See our "Hawaii" paper for details.)

Problem 3: 10 points.

It has been said that, for instruction-level (ILP) architectures, register assignment and instruction scheduling are basically antagonistic processes. Describe what is meant by that statement. What are the problem(s) inherent in defining separate compiler phases for register assignment and instruction scheduling? How would you recommend circumventing those problems?

If we do register assignment first, then scheduling is hindered by the anti-dependences inherent in mapping an "infinite" number of symbolic registers to a much smaller number of physical registers.

On the other hand, if we schedule first, the greater scheduling flexibility could lead to enough more registers needed spilling would occur. And any spilling at all can kill a programs execution time.

To circumvent these issues, you might try combining the two optimizations into a single pass. Some compilers actually do this.