Homework assignments are to be completed individually. Hand-written submissions are fine, but they must be readable. Due at the beginning of class. Total points: 100, 3.3% of course grade.

1. [15 Points] Induction Variables. Perform induction variable detection, strength reduction and induction variable elimination on Figure 9.5 in book.

2. [15 Points] SSA. Translate Figure 9.3 into SSA. Perform copy propagation and dead code elimination on SSA.


   a = read()
   b = read()
   z = read()
   w = read()
   x = a - b
   y = a + b
   j = 0
   loop:
      z = a * b
      w = z
      if (w==0) goto L2
      w = a * b
   L2:
      x = a + b
      j = j + 1
      if (z < j) goto loop
   print w,x,y,z

   (a) Perform pessimistic global value numbering.
   (b) Transform the SSA code based on the pessimistic global value numbering results. After doing so, what optimization on SSA (hint problem 2 in this HW) should be performed? Perform it.
   (c) Rewrite the resulting code in 3-address code.

   Answer:
   For pessimistic global value numbering, we need to translate the code to SSA.

   a0 = read()
b0 = read()
z0 = read()
w0 = read()
x0 = a0 - b0
y0 = a0 + b0
j0 = 0

loop:
z1 = phi_n(z0, z2)
x1 = phi_n(x0, x2)
j1 = phi_n(j0, j2)
w1 = phi_n(w0, w2)
z2 = a0 * b0
w2 = z2
if (w2==0) goto L2
w3 = a0 * b0

L2:
w4 = phi_m(w3, w2)
x2 = a0 + b0
j2 = j1 + 1
if (z2<j2) goto loop
print w4, x2, y0, z2
(a) Pessimistic global value numbering.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Init Value Num</th>
<th>After visiting stmts</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0</td>
<td>#1</td>
<td>#1</td>
</tr>
<tr>
<td>b0</td>
<td>#2</td>
<td>#2</td>
</tr>
<tr>
<td>z0</td>
<td>#3</td>
<td>#3</td>
</tr>
<tr>
<td>w0</td>
<td>#4</td>
<td>#4</td>
</tr>
<tr>
<td>x0</td>
<td>#5</td>
<td>#1 - #2 → #18</td>
</tr>
<tr>
<td>y0</td>
<td>#6</td>
<td>#1 + #2 → #19</td>
</tr>
<tr>
<td>j0</td>
<td>#7</td>
<td>#7</td>
</tr>
<tr>
<td>z1</td>
<td>#8</td>
<td>phi_n ( #3, #12 ) → #20</td>
</tr>
<tr>
<td>x1</td>
<td>#9</td>
<td>phi_n ( #16, #116 ) → #21</td>
</tr>
<tr>
<td>j1</td>
<td>#10</td>
<td>phi_n ( #7, #17 ) → #22</td>
</tr>
<tr>
<td>w1</td>
<td>#11</td>
<td>phi_n ( #4, #13 ) → #23</td>
</tr>
<tr>
<td>z2</td>
<td>#12</td>
<td>#1 * #2 → #24</td>
</tr>
<tr>
<td>w2</td>
<td>#13</td>
<td>#24</td>
</tr>
<tr>
<td>w3</td>
<td>#14</td>
<td>#1 * #2 → #24</td>
</tr>
<tr>
<td>w4</td>
<td>#15</td>
<td>phi_n ( #24, #24 ) → #24</td>
</tr>
<tr>
<td>x2</td>
<td>#16</td>
<td>#1 + #2 → #19</td>
</tr>
<tr>
<td>j2</td>
<td>#17</td>
<td>#22 + 1 → #25</td>
</tr>
</tbody>
</table>

We end up with the following congruence classes:
- \( P_0 = \{ a_0 \} \)
- \( P_1 = \{ b_0 \} \)
- \( P_2 = \{ z_0 \} \)
- \( P_3 = \{ w_0 \} \)
- \( P_5 = \{ x_0 \} \)
- \( P_6 = \{ y_0, x_2 \} \)
- \( P_7 = \{ j_0 \} \)
- \( P_8 = \{ z_1 \} \)
- \( P_9 = \{ x_1 \} \)
- \( P_{10} = \{ j_1 \} \)
- \( P_{11} = \{ w_1 \} \)
- \( P_{12} = \{ z_2, w_2, w_3, w_4 \} \)
- \( P_{13} = \{ j_2 \} \)

After value numbering, copy propagation, and dead code elimination, we can rewrite the code as follows:

```plaintext
a0 = read()
b0 = read()
z0 = read()
w0 = read()
x0 = a0 - b0
y0 = a0 + b0
j0 = 0
```
loop:  
    z1 = phi_n(z0,z2)  
    x1 = phi_n(x0,y0)  
    j1 = phi_n(j0,j2)  
    w1 = phi_n(w0,z2)  
    z2 = a0 * b0

    if (w2==0) goto L2

L2:

    j2 = j1 + 1
    if (z2<j2) goto loop
print z2,y0,y0,z2

(c) Rewrite in 3-address code. The important points are that you can’t remove the SSA numbering of variables when you convert to three-address code and you need to put a copy on the incoming path associated with each entry in a phi function. We depend on the register allocation to perform coalescing to remove copies.

    a0 = read()  
    b0 = read()  
    z0 = read()  
    w0 = read()  
    x0 = a0 - b0  
    y0 = a0 + b0  
    j0 = 0  
    z1 = z0  
    x1 = x0  
    j1 = j0  
    w1 = w0

loop:  
    z2 = a0 * b0  
    if (w2==0) goto L2

L2:

    j2 = j1 + 1  
    z1 = z2  
    x1 = y0  
    j1 = j2  
    w1 = z2  
    if (z2<j2) goto loop
print z2,y0,y0,z2
4. [20 points] Data dependence analysis and unimodular transformations

```c
for (i=0; i<N; i++) {
    for (j=3; j<M; j++) {
        B[j-1][i] = sin(i * j);
    }
}
```

(a) For the above program, what is the direction vector for the output dependences between writes to A[i]? (Hint: Recall that $(\ast, \ast)$, $(\ast, =)$, and $(\ast, \ast)$ are not legal dependence vectors.)
(b) For the above program, what is the distance vector for the flow dependence?
(c) What is the unimodular transformation matrix that specifies a permutation of the i and j loops in the program for problem 4?
(d) Is the problem 4 loop fully permutable? Why or why not?
(e) Which loop carries each of the dependences? What is a possible parallelization strategy for the above loop?

5. [15 points] Loop Fission and the Kelly and Pugh Transformation Framework

(a) Show whether loop fission is legal or illegal for the following program using the K&P transformation framework.

```c
for (i=0; i<N; i++) {
    A[i] = ... ;
    ... = A[i+1];
}
```

(b) Show whether loop fission is legal or illegal for the following program using the K&P transformation framework.

```c
for (i=0; i<N; i++) {
    A[i] = ... ;
    ... = A[i-2];
}
```

6. [15 points] Loop transformations and Fourier Motzkin. Skew the loop to make it permutable and then permute the loop. Write the transformed code.

```c
for (i=1; i<N; i++) {
    for (j=1; j<(i+1); j++) {
```
\[
\]

Original bounds:
\[
\begin{bmatrix}
1 & 0 \\
-1 & 0 \\
1 & -1 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
i \\
j
\end{bmatrix}
\geq
\begin{bmatrix}
0 \\
1-N \\
0 \\
1
\end{bmatrix}
\]

Transformation \( T = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \) selected to modify dependence vector \((1, -1)\) so that loop is permutable.

\[
\begin{bmatrix}
1 & 1 \\
1 & 0
\end{bmatrix}
\begin{bmatrix}
i \\
j
\end{bmatrix} =
\begin{bmatrix}
i' \\
j'
\end{bmatrix}
\]

\[
T^{-1} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}
\]

Transformed iteration space:
\[
\begin{bmatrix}
0 & 1 \\
0 & -1 \\
-1 & 2 \\
1 & -1
\end{bmatrix}
\begin{bmatrix}
i' \\
j'
\end{bmatrix}
\geq
\begin{bmatrix}
0 \\
1-N \\
0 \\
1
\end{bmatrix}
\]

Array access \( A[i][j] \)
\[
F = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\]

\[
F' = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}
\]

New bounds after projecting out \( j' \) using Fourier Motzkin:
\[
\begin{bmatrix}
-1 & 0 \\
1 & 0 \\
1 & 0
\end{bmatrix}
\begin{bmatrix}
i' \\
j'
\end{bmatrix}
\geq
\begin{bmatrix}
2 - 2N \\
2 \\
1
\end{bmatrix}
\]

Transformed code:
\[
\text{for (i'=1; i<=N; i++) { for (j'=max(0,i'/2); j'<=min(N-1,i'-1); j'++) { A[ j' ][ i'-j' ] = A[j'-2,i'-j'] + A[j'-1,i'-j'+1]; }}}
\]