Sample Solution

1. Test for dependences on S. Write down the subscripts. Which positions are separable, which are coupled? Which dependence test would you apply to each position?

a)

```
     for (k=0; k<100; k++) {
        for (j=0; j<100; j++) {
          for (i=0; i<100; i++) {
S           A[i+1,j+1,k+1] = A[i,j,1] + c;
          }
        }
      }
```

<i+1, i>: separable, strong SIV test
<j+1, j>: separable, strong SIV test
<k+1, 1>: separable, weak-zero SIV test

b)

```
      for (k=0; k<100; k++) {
        for (j=0; j<100; j++) {
          for (i=0; i<100; i++) {
S           A[i+1,j+k+1,k+1] = A[i,j,k] + c;
          }
        }
      }
```

<i+1, i>: separable, strong SIV test ($\Delta i = 1 \rightarrow$ direction vector "<")
<j+k+1, j>, <k+1, k>: coupled, Delta test:
<k+1,k>: strong SIV test: $k+1 = k + \Delta k \rightarrow \Delta k = 1 \rightarrow$ direction vector "<"
<j+k+1,j>: $j + k + 1 = j + \Delta j \rightarrow \Delta j = k + 1$. $k >= 0 \rightarrow \Delta j > 0 \rightarrow$ direction vec. "<"

c)

```
      for (k=0; k<100; k++) {
        for (j=0; j<100; j++) {
          for (i=0; i<100; i++) {
S           A[i+1,j+k+1,i] = A[i,j,2] + c;
          }
        }
      }
```

<i+1,i>, <i, 2>: coupled, Delta test:
<i, 2>: weak-zero test: $i = 2$
<i+1, i>: $\Delta i = 1$ $\rightarrow$ dependence between i=2 and $i' = i + \Delta i = 3$.
<j+k+1, j>: separate, solve as in (b) above.

Note that k is unconstrained, i.e., the direction vector for k is (*)

2. Compute the entire set of direction vectors for all potential dependences in the loop. What type of dependence are we dealing with? What test would we apply to test for dependence?

```
        for (k=0; k<100; k++) {
          for (j=0; j<100; j++) {
            for (i=0; i<100; i++) {
S1            A[i+1,j+4,k+1] = B[i,j,k] + c;
S2            B[i+j,5,k+1] = A[2,k,k] + c;
            }
          }
        }
```

<u>S1 → S2 on A:</u>

<i+1,2>, <j+4,k>, <k+1,k>

<i+1, 2>: separable, the weak-zero test yields i = 1, direction unbound.

<j+4, k>, <k+1, k>: coupled, applying Delta test:

<k+1, k>: the strong SIV test yields a distance vector of d = 1.

<j+4, k>: propagating the distance constraint for k yields

$j + 4 = k + \Delta k = k + 1$

$j = k - 3$, direction unbound.

→ D = (<, *, *). This is in any possible case a loop-carried level-1 true dependence.

<u>S2 → S1 on B:</u>

<i+j, i>, <5, j>, <k+1, k>

<k+1, k>: separable, the strong SIV test yields a distance of $\Delta k = 1$.

<i+j, i>, <5, j>: coupled, applying Delta test:

<5, j>: the weak-zero SIV test yields 5 = j' (unconstrained direction)

<i+j,i>: $i + j = i + \Delta i$

$\Delta i = j$. Since j >= 0, the direction is either = or <.

→ D = (<, *, =/<). Again, this is in any possible scenario a loop-carried level-1 true dep.

3. Construct valid breaking conditions for the following examples

a)

```
    for (i=0; i<100; i++) {
S     A[i+ix] = A[i] + c;
    }
```

we have dependence if i + ix = i'. Taking the loop bounds into consideration this is the case if $-100 < ix < 100$. This is the breaking condition, i.e.,

```
if ((-100 < ix) && (ix < 100)) {
     // the dependence manifests
} else {
     // loop can be directly parallelized
}
```

b)

```
    for (k=0; k<100; k++) {
      for (j=0; j<100; j++) {
        for (i=0; i<100; i++) {
S         A[i+1,j+1,k+1] = A[i,jx,1] + c;
        }
      }
    }
```

Determining the distance vector for S:

<i+1,i>, <j+1,jx>, <k+1, 1>

<k+1, 1>: dependence only if k = 0 (from the weak-zero SIV test)
<i+1, i>: dependence if i' = i+1, i.e., the distance is 1 (strong SIV test)
<j+1, jx>: dependence if j+1 = jx. Considering the loop bounds this is only possible if jx = [1...100].

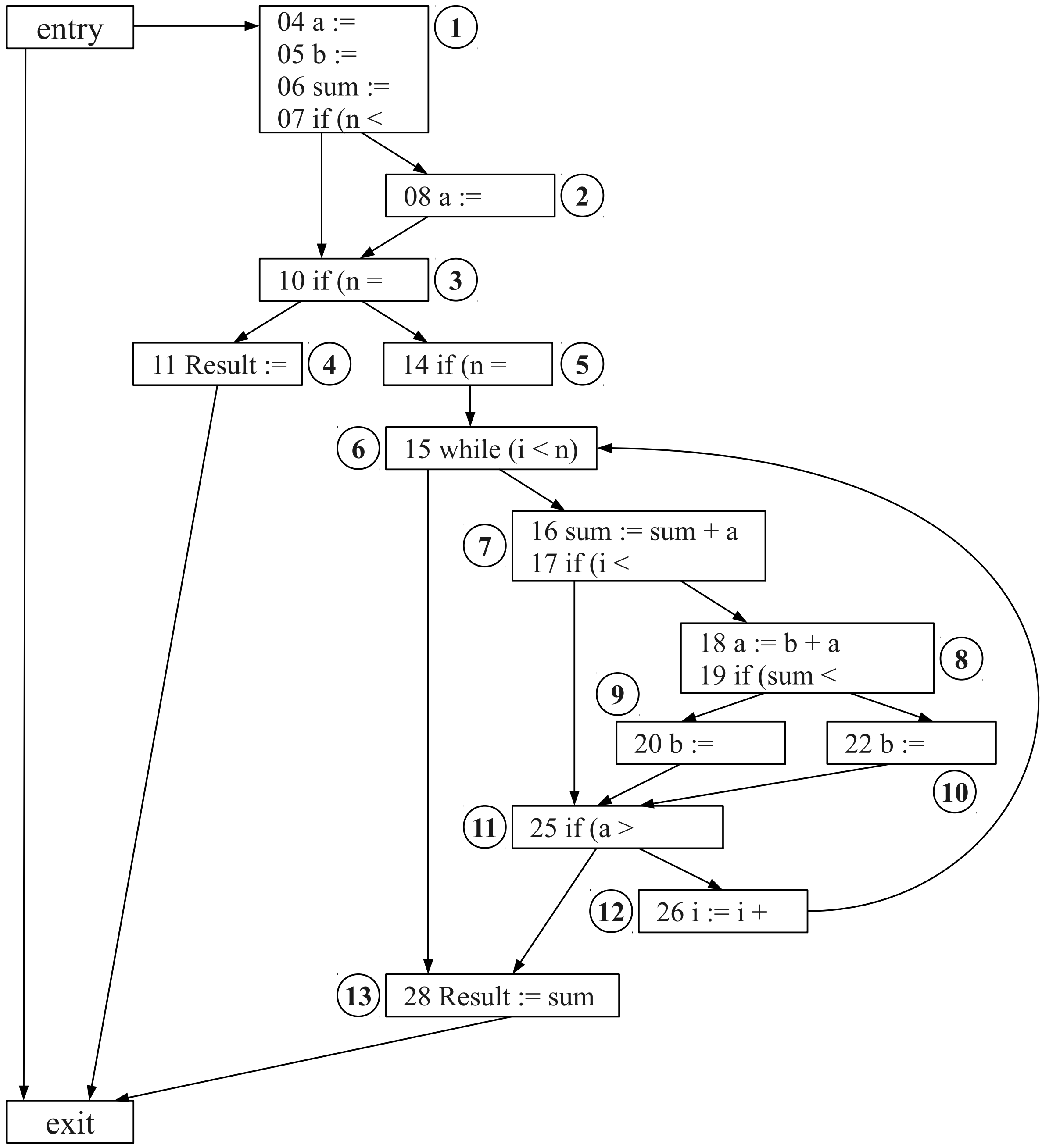
Here, it is best to treat the breaking condition on k separately, and only test for jx if the breaking condition for k does not hold:

```
if (k > 0) {
        // no dependence possible, parallelize at will
} else {
        // A level-1 loop-carried dependence exists, but the level-2 dependence may or
        // may not manifest itself
        if ((0 < jx) && (jx < 101)) {
                // the level-2 dependence exists
        } else {
                // no level-2 dependence, parallelize the j/i loop at will
        }
}
```

4. Construct the CFG for the following program. Which blocks are included in the dominance frontier of variable a defined at line 18?

```
01 procedure foo(n: integer): integer;
02 var i,sum: integer;
03 begin
04  a := 0;
05  b := 1;
06  sum := 0;

07  if (n < 0) then begin
08    a := 5
09  end;

10  if (n = 1001) then begin
11    Result := 0;
12    Exit
13  end;

14  i := 0;
15  while (i < n) do begin
16    sum := sum + a;
17    if (i < 5) then begin
18      a := b + a;
19      if (sum > 100) then begin
20        b := 2
21      end else begin
22        b := 3
23      end
24    end;
25    if (a > 20) then break;
26    i := i+1
27  end;

28  Result := sum;
29 end;
```

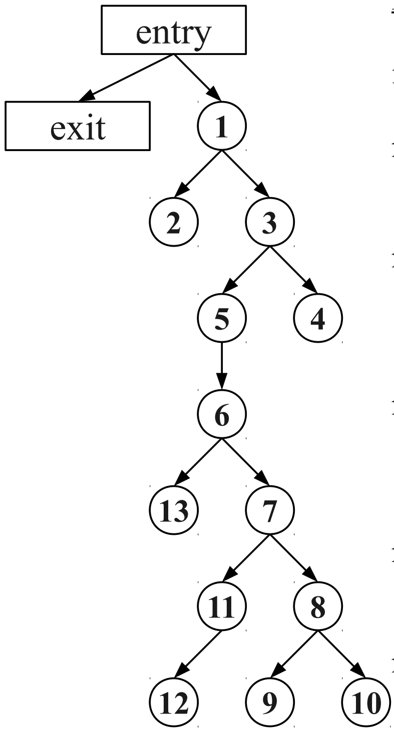
<u>CFG:</u> (number in round circles denote block numbers)



Block 8 (which contains the statement 18 a := b + a) only dominates blocks 9 and 10. The only successor of these blocks is block 11, the only block in the dominance frontier of block 8.

5. Construct the SSA graph for the program shown in the previous problem.

Dominator tree:



Placement of Φ nodes for variable a:

init: worklist = { 1, 2, 8 }

node = 1; worklist = { 2, 8 }
    DF(1) = { exit }

node = 2; worklist = { 8 }
    DF(2) = { 3 }
    insert Φ node into 3, add 3 to worklist ({ 8, 3 })

node = 8; worklist = { 3 }
    DF(8) = { 11 }
    insert Φ node into 11, add 11 to worklist ({ 3, 11 })

node = 3; worklist = { 11 }
    DF(3) = { exit }

node = 11; worklist = {}
    DF(11) = { 6, 13 }
    insert Φ node into 6, 13, add 6, 13 to worklist ({ 6, 13 })

node = 6; worklist = { 13 }
    DF(6) = { exit }

node = 13; worklist = {}
    DF{13} = { exit }

Renaming variables (only considering a):
count = 0; stack = {}
search(entry);

entry: no assigments, no Φ nodes in successors, search children { exit, 1 }
exit: no assigments, no successors, no children
1: assigment to a (line 4): a → a0, count = 1; stack = { 0 }
Φ node in successor 3: replace first operand by a0: a = Φ(a0, a)
search children { 2, 3 }
2: assigment to a (line 8): a → a1, count = 2; stack = { 0, 1 }
Φ node in successor 3: replace second operand by a1: a = Φ(a0, a1)
no children, pop stack = { 0 }
3: Φ node assignment: a → a2 = Φ(a0, a1), count = 3, stack = {0, 2 }
no Φ nodes in successors, search children { 4, 5 }
4: no assigments, no Φ nodes in successors, no children
5: no assignments
Φ node in successor 6: replace first operand by a2: a = Φ(a2, a)
search children (6)
6: Φ node assignment a → a3 = Φ(a2, a), count = 4; stack = { 0, 2, 3 }
Φ node in successor 13: replace first operand by a3: a = Φ(a3, a)
search children { 7, 13 }
13: Φ node assignment a → a4 = Φ(a3, a), count = 5, stack = { 0, 2, 3, 4 }
no Φ nodes in successors, no children, pop stack = { 0, 2, 3 }
7: replace use of a (line 16) by a3: sum := sum + a3, no assigment to a,
Φ node in successor 11: replace first operand by a3: a = Φ(a3, a, a)
search children (8, 11)
8: replace use of a (line 18) by a3: a := b + a3
assignment to a (line 18): a → a5, count = 6, stack = { 0, 2, 3, 5 }
no Φ nodes in successors, search children {9, 10}
9: no uses of, assigments to a
Φ node in successor 11: replace 2nd operand by a5: a = Φ(a3, a5, a)
no children
10: no uses of, assignments to a
Φ node in successor 11: replace 3rd operand by a5: a=Φ(a3, a5, a5)
pop stack = { 0, 2, 3 }
11: Φ node assignment: a → a6, count = 7, stack = { 0, 2, 3, 6 }
replace use of a (line 25) by a6: if (a6 > 20)
Φ node in successor 13: replace 2nd operand by a6: a4 = Φ(a3, a6)
search children { 12 }
12: no uses of, assignments to a
Φ node in successor 6: replace 2nd operand by a6: a3 = Φ(a2, a6)
no children
pop stack = { 0, 2, 3 }
(node 6) pop stack = { 0, 2 }
(node 3) pop stack = { 0 }
(node 1) pop stack = {}

<u>Final code (only variable a in SSA form):</u>

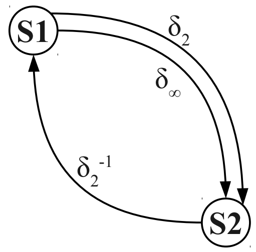
```
procedure foo(n: integer): integer;
var i,sum: integer;
begin
 a0 := 0;
 b := 1;
 sum := 0;

 if (n < 0) then begin
   a1 := 5
 end;

 a2 = Φ(a0, a1)
 if (n = 1001) then begin
   Result := 0;
   Exit
 end;

 i := 0;
 a3 = Φ(a2, a6);
 while (i < n) do begin
   sum := sum + a3;
   if (i < 5) then begin
     a5 := b + a3;
    if (sum > 100) then begin
       b := 2
     end else begin
       b := 3
     end
   end;
   a6 = Φ(a3, a5, a5);
   if (a6 > 20) then break;
   i := i+1
 end;

 a4 = Φ(a3, a6);
 Result := sum;
end;
```

6. Use the algorithm *vectorize* from chapter 4 to vectorize the following code:

```
    for (k=0; k<100; k++) {
      for (j = 0; j<100; j++) {
S1      B[1,j,k] = A[1,j-1,k];
        for (i=0; i<100; i++) {
S2        A[i+1,j,k] = B[i,100-j,k] + c;
        }
      }
    }
```

1. construct dependence graph

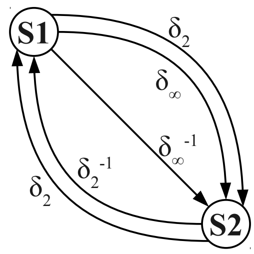
<u>S1 → S2 on B</u>
<k, k>, <j, 100-j>, <1, i>



<k, k>: strong SIV, D = (=)
<j, 100-j>: weak-crossing SIV: D = (*)
<1, i>: ZIV: loop-independent true dependence

D = (=, *) = { (=, <), (=, =), (=, >) }

<u>S2 → S1 on A</u>
<k, k>, <j, j-1>, <i+1, 1>



<k, k>: strong SIV, D = (=)
<j, j-1>: strong SIV, D = (<)
<i+1, 1>: ZIV: loop-independent anti-dependence

D = (=, <)

2. call vectorize() for the whole loop nest

<u>`vectorize(loop nest, level=1, D)`</u>
{ S1, S2 } are strongly connected and the only such segment, hence π_1 = { S1,S2 }
π_1 is cyclic:

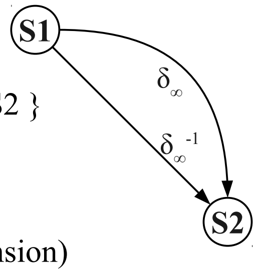
```
"for (k=0; k<100; k++) {"
  vectorize( π1, level=2, D)
"}"
```

<u>`vectorize(j,i-loop, level=2, D)`</u>
strip all dependences < 2 (none), so the only π block is still { S1, S2 }
π_1 is still cyclic:

```
"for (j=0; j<100; j++) {"
  vectorize( π1, level=3, D)
"}"
```

<u>`vectorize(i-loop, level=3, D)`</u>

strip all dependences < 2, yielding the dependence graph



no stronly connected components, two π blocks: { S1 }, { S2 }
topological sorting: π_1 = { S1 }, π_2 = { S2 }

π_1 not cyclic, enclosing loops: 2
    vectorize in 2 – 3 + 1 = 0 dimensions (i.e., no dimension)
    `"B[1,j,k] = A[1,j-1,k];"`

π_2 not cyclic, enclosing loops: 3
    vectorize in 3 – 3 + 1 = 1 dimension:
    `"A[1:100,j,k] = B[0:99,100-j,k] + c;"`

<u>complete code:</u>

```
   for (k=0; k<100; k++) {
     for (j = 0; j<100; j++) {
S1     B[1,j,k] = A[1,j-1,k];
S2       A[1:100,j,k] = B[9:99,100-j,k] + c;
     }
   }
```