Optimizing Naive Code

```
void vadd (int A[], int B[], int C[], int N)
{
    int i;
    for(i = 0; i < N; i++)
        A[i] = B[i] + C[i];
}
```

--- That COULD generate the following (hypothetical machine) assembler ---

```
L1:  r11 = load   8(fp)        # load local "i"
     r12 = load -16(fp)        # load parm  "N"
     r9  = r12 - r11           # N - i
     beq L3                    # branch on ==
```

```
L2:  r13 = load   8(fp)        # load local "i"
     r14 = r13 * 4             # t1 = i * 4
     r15 = r14 + B             # t2 = t1 + &B
     r25 = load 0(r15)         # load B[i]

     r16 = load   8(fp)        # load local "i"
     r17 = r16 * 4             # t3 = i * 4
     r18 = r17 + C             # t4 = t3 + &C
     r26 = load 0(r18)         # load C[i]

     r19 = r25 + r26           # t5 = B[i] + C[i]

     r20 = load   8(fp)        # load local "i"
     r21 = r20 * 4             # t6 = i * 4
     r22 = r21 + A             # t7 = t6 + &A
     store r19, 0(r22)         # store A[i]

     r23 = load   8(fp)        # load local "i"
     r24 = r23 + 1             # i = i + 1
     store r24, 8(fp)          # store i

     br  L1                    # branch to loop's top
```

```
L3:    # Whatever comes next
```

************ First Improvement --- Register Assignment *********

Assume i stored in r29, N in r30

****

```
L1:  r9  = r30 - r29            # N - i
     beq L3                     # branch on ==

L2:  r14 = r29 * 4              # t1 = i * 4
     r15 = r14 + B              # t2 = t1 + &B
     r25 = load 0(r15)          # load B[i]

     r17 = r29 * 4              # t3 = i * 4
     r18 = r17 + C              # t4 = t3 + &C
     r26 = load 0(r18)          # load C[i]

     r19 = r25 + r26            # t5 = B[i] + C[i]

     r21 = r29 * 4              # t6 = i * 4
     r22 = r21 + A              # t7 = t6 + &A
     store r19, 0(r22)          # store A[i]

     r29 = r29 + 1              # i = i + 1

     br  L1                     # branch to loop's top

L3:    # Whatever comes next
```

******** Next, note we compute i * 4 several times ********

```
L1:  r9  = r30 - r29          # N - i
     beq L3                   # branch on ==

L2:  r14 = r29 * 4            # t1 = i * 4
     r15 = r14 + B            # t2 = t1 + &B
     r25 = load 0(r18)        # load B[i]

     r18 = r14 + C            # t4 = t1 + &C
     r26 = load 0(r19)        # load C[i]

     r19 = r15 + r16          # t5 = B[i] + C[i]

     r22 = r14 + A            # t7 = t1 + &A
     store r19, 0(r22)        # store A[i]

     r29 = r29 + 1            # i = i + 1

     br  L1                   # branch to loop's top

L3:    # Whatever comes next
```

******** Next, note we don't need compute i * 4 in loop at all ********

```
L0:  r21 = A                    # r21 = &(A[0])
     r22 = B                    # r22 = &(B[0])
     r23 = C                    # r23 = &(C[0])

L1:  r9  = r30 - r29            # N - i
     beq L3                     # branch on ==

L2:  r25 = load 0(r22++)        # load B[i], auto-inc
     r26 = load 0(r23++)        # load C[i], auto-inc
     r19 = r25 + r26            # t5 = B[i] + C[i]
     store r19, 0(r21++)        # store A[i], auto-inc
     r29 = r29 + 1              # i = i + 1
     br  L1                     # branch to loop's top

L3:    # Whatever comes next
```

******** Finally, using compiler voodoo we get ********

```
L0:  r21 = A                    # r21 = &(A[0])
     r22 = B                    # r22 = &(B[0])
     r23 = C                    # r23 = &(C[0])
     r28 = r30 * 4              # r28 = offset per array
     r29 = r30 + A              # end of A, for test

L2:  r25 = load 0(r22++)        # load B[i]
     r26 = load 0(r23++)        # load C[i]
     r19 = r25 + r26            # t5 = B[i] + C[i]
     store r19, 0(r21++)        # store A[i]
     r9  = r21 - r29            # &(A[i]) - &(A[end])
     bgt L2                     # branch on >

L3:    # Whatever comes next
```