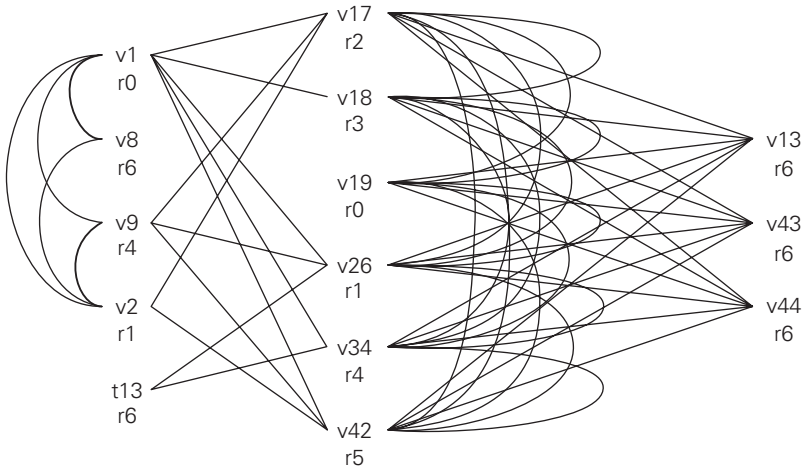


**Figure 17.19** Register interference graph for the software pipelined version of the `combinations` subroutine. Using architectural register names, we have indicated one of several possible seven-colorings.

```
Block 1:
    push {r4, r5, r6}
    *r1 := 1
    r6 := r0 << 2
    r4 := r1 + r6
    *r4 := 1
    r2 := 1
    r5 := r0 >> 1
    r3 := r0
    cc := r5 = 0
    if cc goto Block 4
    goto Block 3
Block 2:
    r6 := r0 div r2
    r2 := r2 + 1
    r3 := r3 − 1
    r1 := r1 + 4
    r4 := r4 − 4
    r0 := r6 × r3
    *r1 := r6
    *r4 := r6
Block 3:
    cc := r2 < r5
    if cc goto Block 2
Block 4a:
    r6 := r0 div r2
    *(r1+4) := r6
    *(r4−4) := r6
Block 4:
    pop {r4, r5, r6}
    goto *lr
```

**Figure 17.20** Final code for the `combinations` subroutine, after assigning architectural registers and eliminating useless copy instructions.

else; we have arbitrarily chosen to have both it and t13 share with the three registers on the right. ■

**EXAMPLE 17.43**

Optimized `combinations` subroutine

Final code for the `combinations` subroutine appears in Figure C-17.20. We have left v1/v19 and v2/v26 in r0 and r1, the registers in which their initial values were passed. Because our subroutine is a leaf, these registers are never needed for other arguments. Following ARM conventions (Section C-5.4.5), we have used