CSC 462/562 Computer Architecture
Homework #1 answer key

1. Results shown below.

For F=20% and k=2, S=1.11
For F=30% and k=2, S=1.18
For F=40% and k=2, S=1.25
For F=50% and k=2, S=1.33
For F=60% and k=2, S=1.43
For F=70% and k=2, S=1.54
For F=80% and k=2, S=1.67
For F=90% and k=2, S=1.82
For F=40% and k=7, S=1.52
For F=40% and k=9, S=1.55
For F=40% and k=11, S=1.57
For F=40% and k=13, S=1.59
For F=40% and k=15, S=1.60
For F=40% and k=17, S=1.60
For F=40% and k=19, S=1.61
For F=40% and k=21, S=1.62
For F=75% and k=1.4, S=1.27
For F=75% and k=1.6, S=1.39
For F=75% and k=1.8, S=1.50
For F=75% and k=2.0, S=1.60
For F=75% and k=2.2, S=1.69
For F=75% and k=2.4, S=1.78
For F=75% and k=2.6, S=1.86
For F=75% and k=2.8, S=1.93

The greatest speedup occurs when F = 90% and k = 2 and when F = 75% and k = 2.8. When F = 40%, even a speedup of 21 doesn't come as close. The speedup generally has little impact when F is low (see when F < 50%) no matter what k is, while the impact when F is large (50%+ sees a significant speedup even if k is as low as 2.4). The best speedup occurs when F is very large or F is large and k is more than 2.

2. The following list shows for each instruction its usage in the two benchmarks, the original and new CPIs.

Instruction type	% in sjeng	% in mcf	original CPI	new CPI
Loads	19%	35%	6	6
Stores	7%	11%	5	5
ALU operations	56%	29%	4	3 (enh1)
Conditional branches	15%	24%	4	2 (enh2)
Uncond. branches	3%	1%	3	2 (enh2)

sjeng original CPI = .19 * 6 + .07 * 5 + .56 * 4 + .15 * 4 + .03 * 3 = 4.42
sjeng enhancement 1 CPI = .19 * 6 + .07 * 5 + .56 * 3 + .15 * 4 + .03 * 3 = 3.86
sjeng enhancement 2 CPI = .19 * 6 + .07 * 5 + .56 * 3 + .15 * 2 + .03 * 2 = 4.09
speedup of enhancement 1 = 1.15 (15%)
speedup of enhancement 1 = 1.08 (8%)
mcf original CPI = .35 * 6 + .11 * 5 + .29 * 4 + .24 * 4 + .01 * 3 = 4.80
mcf enhancement 1 CPI = .35 * 6 + .11 * 5 + .29 * 3 + .24 * 4 + .01 * 3 = 4.51
mcf enhancement 2 CPI = .35 * 6 + .11 * 5 + .29 * 4 + .24 * 2 + .01 * 2 = 4.31
speedup of enhancement 1 = 1.06 (6%)
speedup of enhancement 1 = 1.11 (11%)

The two enhancements favor one benchmark over the other. Amdahl's law says to make the common case faster and in this case, the more common case is usually ALU operations (mcf is an exception in that the branches almost equal the ALU operations). So, even though the improvement in branches is better (4 → 2, 3 → 2 over 4 → 3), the common case argues that enhancement 1 should be implemented.

3. The astar benchmark has the following instruction mix breakdown:

   28% loads, 6% stores, 18% branches, 2% jumps, 46% ALU

   The new ALU instructions will be used in place of 30% of loads, or 28% * 30% = 8.4% and in 15% of the stores, or 6% * 15% = .9%. Thus, we can remove 8.4% + .9% = 9.3% of the instructions, or IC reduces by 9.3% to 90.7%. With the removal of instructions from the program, we now have to recompute the breakdown of operations. We do so by dividing the previous fraction by 90.7%. In the case of loads and stores, we first deduct the 8.4% and .9% respectively. Our new breakdown is:

   (28% - 8.4%) / 90.7% = 21.6%
   (6% - .9%) / 90.7% = 5.6%
   18% / 90.7% = 19.8%
   2% / 90.7% = 2.2%
   46% / 90.7% = 50.7%

   We use the previous CPIs except that of the 50.7% ALU operations, 9.3% have a CPI of 7, so the remaining 50.7% – 9.3% = 41.4% have a CPI of 4. The other CPIs are as they were from problem #2 (6, 5, 4, 3 respectively for load, store, conditional branch, unconditional branch).

   a. Original CPI = .28 * 6 + .06 * 5 + .18 * 4 + .02 * 3 + .46 * 4 = 4.6
      New CPI = .216 * 6 + .056 * 5 + .198 * 4 + .022 * 3 + .414 * 4 + .093 * 7 = 4.741
      Speedup = old CPI * IC / (new CPI * new IC) = 1 * 4.6 / (.907 * 4.741) = 1.070, or a 7% speedup
   b. Factoring in the clock cycle time means we need to compare old clock * old CPI * old IC to new clock * new CPI * new IC, this amounts to 1 * 4.6 * 1.0 / (.907 * 4.741 * 1.10) = .972, or a slowdown of about 2.8%.

4. The number of cycles required to compute a multiplication in hardware is 8. The number of cycles to compute a multiplication in software is 2 + n * 3. This is 50 for n = 16 and 258 for n = 64. The speedup of the enhancement is 50 / 8 = 6.25 (k) for n = 16 and 258 / 8 = 32.25 for n = 64. The frequency of usage for this benchmark is 5% (f). The speedup is then S = 1 / (1 - .05 + .05 / 6.25) = 1.044 or 4.4% for n = 16 and S = 1 / (1 - .05 + .05 / 32.25) = 1.051 for n = 64.

5.
   a. We will use x0 when we want to clear a register as in add x1, x0, x0 to place 0 into x1. We will use x0 for direct memory referencing in a load or store as in lw x2, 5000(x0).
   b. To clear a register, we will subtract a value from itself as in sub x1, x2, x2. This places 0 into x1. For the load, we would first have to place 0 into the register we will use as the base. So, we can do lw x2, 5000(x1) after we did the sub instruction.

6.
```
        addw   x1, x0, x0          // x1 = i, set it to 0
        addiw  x2, x0, 500         // x2 = 500, used to determine end of loop
        addiw  x3, x0, 10000  // x3 is the address of a[i]
        lw     x4, 9988(x0)        // x4 = x
        lw     x5, 9992(x0)        // x5 = y
        lw     x6, 9996(x0)        // x6 = z
loop:   beq    x1, x2, out         // exit loop when i == 500
```

```
        lw     x7, 0(x3)         // x7 = a[i]
        slt    x8, x7, x4        // x8 ← 1 if a[i] < x meaning we want to do the else
        jeq    x8, x0, then      // if x8 is 0, it means a[i] >= x and we do the then
        addiw  x6, x6, 1         // else clause (z++)
        j      end               // skip then clause to finish loop
then:   addiw  x5, x5, 1         // y++
end:    addiw  x3, x3, 4         // move x3 onto next array location
        addiw  x1, x1, 1         // i++
        j      loop              // go back to top of loop for next iteration
out:    sw     x5, 9992(x0)      // store revised y back to memory
        sw     x6, 9996(x0)      // store revised z back to memory
```

7.

```
        addw   x1, x0, x0        // x1 = i, set i to 0
        addiw  x2, x0, 5000      // x2 will be used to access array A
        addiw  x3, x0, 6000      // x3 will be used to access array B
        ld     x4, 7000(x0)      // x4 ← C
        ld     x5, 7008(x0)      // x5 ← D
        addiw  x6, x0, 100       // x6 will be used to compare i<=100
loop:   slt    x7, x6, x1        // x7 ← 1 if i exceeds 100, 0 otherwise
        jne    x7, x0, out       // if x7 != 0 it equals 1, so leave loop
        ld     x8, 0(x3)         // x8 ← B[i]
        mul    x9, x8, x4        // x9 ← B[i] * C
        add    x10, x9, x5       // x10 ← B[i] * C + D
        sd     x10, 0(x2)        // A[i] ← x10
        addiw  x1, x1, 1         // i++
        addiw  x2, x2, 8         // x2 points at next element of A
        addiw  x3, x3, 8         // x3 points at next element of B
        j      loop
out:    …
```