asn1.2020-01

February 23, 2020

Calculations for CompSci 330 assignment 1.
The first block defines a couple of utility functions.
Subsequent blocks provide solutions for numbered problems from Patterson and Hennessey, COD - ARM

```
In [1]: '''
        Utility functions for calculating and formatting solutions to CompSci330 asn.ch01
        2020-02-21
        '''
        import math
        def engfmt(n, threshold=3):
            ''' Represent numbers in "engineering notation",
            i.e. with an exponent that is a multiple of 3.
            '''
            p10 = math.log(n,10)
            p = int((p10 // 3) * 3)
            if p > threshold or p < -threshold:
                excess = p10 - p
                return '{:.3f}e{:d}'.format(10**excess, p)
            else:
                return '{:.3f}'.format(n)

        def binfmt(n, units='bytes', threshold=3):
            ''' Represent numbers in "power-of-2 notation".
            '''
            prefixes = {0:'', 1:'Kibi', 2:'Mebi', 3:'Gibi', 4:'Tebi', 5:'Pebi', 6:'Exbi'}
            lgn = math.log(n,2)
            intpart = int(lgn/10)
            fracpart = lgn - 10*intpart
            return '{:.3g} {:s}{:s}'.format(2**fracpart, prefixes[intpart], units)

        def test_binfmt():
            x = int(input('value? '))
            print( binfmt(x) )
```

1 —————————————————————————-

2 question #1

2.0.1 1.4

Assume a color display using 8 bits for each of the primary colors (red, green, blue) per pixel and a frame size of 1280 x 1024.

a. What is the minimum size in bytes of the frame buffer to store a frame?
b. How long would it take, at a minimum, for the frame to be sent over a 100 Mbit/s network?

In [2]:
```
# 1.4 a.
pixels = 1280*1024
totalBytes = pixels*3
totalBits = totalBytes*8
print('{:s} ({:d} Bytes),  -or-  {:s} ({} bits)'.format(
        binfmt(totalBytes), totalBytes, binfmt(totalBits, units='bits'), totalBits))

#2.4 b.
bps = 100e6     # 100Mbits/second
total_time = totalBits / bps
print('{:s} ({}) seconds'.format(engfmt(total_time), total_time,))
```

```
3.75 Mebibytes (3932160 Bytes),  -or-  30 Mebibits (31457280 bits)
0.315 (0.3145728) seconds
```

3 —————————————————————————-

4 question #2

4.0.1 1.5

In [3]:
```
'''
1.5
Consider three different processors P1, P2, and P3 executing the same
instruction set.

    P1 has a 3 GHz clock rate and a CPI of 1.5.
    P2 has a 2.5 GHz clock rate and a CPI of 1.0.
    P3 has a 4.0 GHz clock rate and has a CPI of 2.2.

    a) Which processor has the highest performance expressed in
    instructions per second?

    b) If the processors each execute a program in 10 seconds,
    find the number of cycles and the number of instructions.
    c) We are trying to reduce the execution time by 30% but this leads
    to an increase of 20% in the CPI.  What clock rate should we have
    to get this time reduction?
'''
# instructions / second = clocks/second  /  clocks/instruction
def ips(cps, cpi): return cps/cpi

# a)
cps_p1 = 3e9   # 3GHz clock
cpi_p1 = 1.5   # cycles per instruction
ips_p1 = ips(cps_p1, cpi_p1)   # instructions per second

cps_p2 = 2.5e9   # 2.5GHz clock
cpi_p2 = 1.0   # cycles per instruction
ips_p2 = ips(cps_p2, cpi_p2)   # instructions per second

cps_p3 = 4e9   # 4GHz clock
cpi_p3 = 2.2   # cycles per instruction
ips_p3 = ips(cps_p3, cpi_p3)   # instructions per second

print('a) ', engfmt(ips_p1), engfmt(ips_p2), engfmt(ips_p3), ': P2 is largest')
print()

# b)
# Answers converted to "engineering units" for display:
inst_1 = ips_p1*10
inst_2 = ips_p2*10
inst_3 = ips_p3*10
print('b) P1: ', engfmt(3e9*10), 'cycles,', engfmt(inst_1), 'instructions')
print('   P2: ', engfmt(2.5e9*10), 'cycles,', engfmt(inst_2), 'instructions')
print('   P3: ', engfmt(4e9*10), 'cycles,', engfmt(inst_3), 'instructions')
print()

# c)
# execution times proportional to 1 / ips
# --> new execution times proportional to 0.7 / ips
# ips/0.7 = (X*cps) / (1.2*cpi)
# --> X = 1.2/0.7 * ips*cpi / cps = 1.2/0.7 * cps/cpi * cpi /cps
# --> X = 1.2/0.7

new_x = 1.2/0.7
ncps_p1 = new_x * cps_p1
ncps_p2 = new_x * cps_p2
ncps_p3 = new_x * cps_p3
print('c) X: {:.3f} clock speed increase'.format(new_x))
print('   P1: {:s} Hz ({})'.format(engfmt(ncps_p1), ncps_p1))
print('   P2: {:s} Hz ({})'.format(engfmt(ncps_p2), ncps_p2))
print('   P3: {:s} Hz ({})'.format(engfmt(ncps_p3), ncps_p3))
```

```
a)  2.000e9 2.500e9 1.818e9 : P2 is largest

b) P1:  30.000e9 cycles, 20.000e9 instructions
   P2:  25.000e9 cycles, 25.000e9 instructions
   P3:  40.000e9 cycles, 18.182e9 instructions

c) X: 1.714 clock speed increase
   P1: 5.143e9 Hz (5142857142.857143)
   P2: 4.286e9 Hz (4285714285.714286)
   P3: 6.857e9 Hz (6857142857.142858)
```

5 ——————————————————————-

6 question #3

6.0.1 1.7

In [4]:
```
'''
1.7
Compilers can have a profound impact on the performance of an application.
Assume that for a program, compiler A results in a dynamic instruction count
of 1.0E9 and has an execution time of 1.1 s, while compiler B results in a
dynamic instruction count of 1.2E9 and an execution time of 1.5 s.

a) Find the average CPI for each program given that the processor has a clock
   cycle time of 1 ns.
b) Assume the compiled programs run on two different processors.  If the execution
   times on the two processors are the same, how much faster is the clock of the
   processor running compiler A's code versus the clock of the processor running
   compiler B's code?
c) A new compiler is developed that uses only 6.0E8 instructions and has an
   average CPI of 1.1.  What is the speedup of using this new compiler versus
   using compiler A or B on the original processor?
'''
# a)
# clock frequency = cps = 1 / (seconds/cycle)
cps = 1/1e-9  # clock frequency = 1/(cycle time)

# cpi = cps * seconds / instructions
cpi_A = cps * 1.1 / 1.0e9
cpi_B = cps * 1.5 / 1.2e9
print('a) A:', engfmt(cpi_A), 'CPI;  B:', engfmt(cpi_B),'CPI')
print()

# b)
# cps = cpi * instructions / seconds
```

```
        # seconds = instructions * cpi / cps
        # seconds = 1.0e9 * cpi_A/cps_A
        # seconds = 1.2e9 * cpi_B/cps_B
        # 1.2e9 * cpi_B/cps_B = 1.0e9 *cpi_A/cps_A
        # cps_A / cps_B = 1.0e9/1.2e9 * cpi_A/cpi_B = speed ratio
        speedup = 1.0e9/1.2e9 * cpi_A/cpi_B
        print('b) cps_A = {:} * cps_B\n   cps_B = {:} * cps_A'.format(
            engfmt(speedup), engfmt(1/speedup)))
        print()

        # c)
        # cycles = cpi * instructions
        cycles_A = cpi_A * 1.0e9
        cycles_B = cpi_B * 1.2e9
        cycles_new = 1.1 * 6.0e8
        print('c) cycles A:', engfmt(cycles_A),
              ';  cycles B:', engfmt(cycles_B),
              'cycles new:', engfmt(cycles_new))
        print('   speedup A:', engfmt(cycles_A/cycles_new),
              '; speedup B:', engfmt(cycles_B/cycles_new))
```

```
a) A: 1.100 CPI;  B: 1.250 CPI

b) cps_A = 0.733 * cps_B
   cps_B = 1.364 * cps_A

c) cycles A: 1.100e9 ;  cycles B: 1.500e9 cycles new: 660.000e6
   speedup A: 1.667 ; speedup B: 2.273
```

7 ———————————————————————————-

8 question #4

8.0.1 1.8

```
In [5]: '''
        1.8
        The Pentium 4 Prescott processor, released in 2004, had a clock rate of 3.6 GHz and
        voltage of 1.25 V.
        Assume that, on average, it consumed 10 W of static power and 90 W of dynamic power.

        The Core i5 Ivy Bridge, released in 2012, had a clock rate of 3.4 GHz and voltage
        of 0.9 V.  Assume that, on average, it consumed 30 W of static power and 40 W of
        dynamic power.
        '''
        fp4p = 3.6e9
        Vp4p = 1.25
        staticp4p = 10
        dynamicp4p = 90

        fi5 = 3.4e9
        Vi5 = 0.9
        statici5 =30
        dynamici5 = 40
```

```
In [6]: '''
        a) [5] <COD ğ1.7> For each processor find the average capacitive loads.
        '''
        print('1.8 a)')
        capp4p = 2 * dynamicp4p / (Vp4p**2 * fp4p)
        capi5 = 2 * dynamici5 / (Vi5**2 * fi5)
        print('a) P4P capacitive load =', engfmt(capp4p))
        print('    i5 capacitive load =', engfmt(capi5))
```

```
1.8 a)
a) P4P capacitive load = 32.000e-9
    i5 capacitive load = 29.049e-9
```

```
In [7]: '''
        b) [5] <COD ğ1.7> Find the percentage of the total dissipated power comprised by
           static power and the ratio of static power to dynamic power for each technology.
        '''
        print('1.8 b)')
        totalp4p = staticp4p + dynamicp4p
        pctstaticp4p = staticp4p / totalp4p * 100
        ratiop4p = staticp4p / dynamicp4p
        totali5 = statici5 + dynamici5
        pctstatici5 = statici5 / totali5 * 100
        ratioi5 = statici5 / dynamici5
        print('b) P4P % static power =', engfmt(pctstaticp4p))
        print('       static:dynamic =', engfmt(ratiop4p))
        print('    i5 % static power =', engfmt(pctstatici5))
        print('       static:dynamic =', engfmt(ratioi5))
```

```
1.8 b)
b) P4P % static power = 10.000
       static:dynamic = 0.111
    i5 % static power = 42.857
       static:dynamic = 0.750
```

```
In [8]: '''
        c) [15] <COD ğ1.7> If the total dissipated power is to be reduced by 10%,
            how much should the voltage be reduced to maintain the same leakage current?
            Note: power is defined as the product of voltage and current.
```

```
static power = Voltage * leakage current --> s = V * I
I = s/V

dynamic power = 0.5 * Capacitive_load * V**2 * frequency --> d = (C * V**2 * f)/2
V**2 = 2 * d / (C*f)

C = 2 * d / (V**2 * f)
V**2 = 2 * d / (C * f)

total power = dynamic power + static power --> t = d + s

(dnew + snew) = 0.9 * (dold + sold)
snew = Vnew * I
     = Vnew * sold/Vold

dnew = 0.9 * (dold+sold) - snew
     = 0.9 * (dold+sold) - (Vnew * I)
     = 0.9 * (dold+sold) - (Vnew * sold/Vold)

Vnew**2 = 2 * dnew / (C * f)
(C*f) * Vnew**2 = 2 * dnew
                = 2 * ( 0.9 * (dold+sold) - (sold/Vold * Vnew) )
                = 2 * 0.9 * (dold+sold) - 2 * sold/vold * Vnew

(C*f) * Vnew**2 + 2*sold/Vold * Vnew - 1.8*(dold+sold) = 0

For the quadratic equation:
    a = C*f
    b = 2*sold/Vold
    c = -1.8*(dold+sold)
'''
print('1.8 c)')

def quadratic(a,b,c):
    t = (b**2 - 4*a*c)**0.5
    yp = (-b + t)/(2*a)
    ym = (-b - t)/(2*a)
    return (yp,ym)

leakagep4p = staticp4p / (Vp4p**2)
leakagei5 = statici5 / (Vi5**2)
print('leakages:', engfmt(leakagep4p), engfmt(leakagei5))

ap4p = capp4p * fp4p
bp4p = 2*staticp4p/Vp4p
cp4p = -1.8 * (dynamicp4p+staticp4p)
Vnewp4p = quadratic(ap4p, bp4p, cp4p)
```

```
        ai5 = capi5 * fi5
        bi5 = 2*statici5/Vi5
        ci5 = -1.8 * (dynamici5+statici5)
        Vnewi5 = quadratic(ai5, bi5, ci5)

        print('   P4P new voltage:', engfmt(Vnewp4p[0]))
        print('    i5 new voltage:', engfmt(Vnewi5[0]))
```

```
1.8 c)
leakages: 6.400 37.037
   P4P new voltage: 1.182
    i5 new voltage: 0.841
```

9 ——————————————————————————-

10 question #5

10.0.1 1.9

```
In [9]: '''
        1.9
        Assume for arithmetic, load/store, and branch instructions, a processor has CPIs
        of 1, 12, and 5, respectively.  Also assume that on a single processor a program
        requires the execution of 2.56E9 arithmetic instructions, 1.28E9 load/store
        instructions, and 256 million branch instructions. Assume that each processor has
        a 2 GHz clock frequency.

        Assume that, as the program is parallelized to run over multiple cores,
        the number of arithmetic and load/store instructions per processor is divided
        by 0.7 x p (where p is the number of processors) but the number of branch
        instructions per processor remains the same.
        '''
        cpiA = 1
        cpiL = 12
        cpiB = 5
        arith = 2.56e9
        ls = 1.28e9
        branch = 256e6
        freq = 2e9  # cycles/second

        def cycles(a, l, b, A=cpiA, L=cpiL, B=cpiB):
            return a*A + l*L + b*B

        def exectime(a, l, b, A=cpiA, L=cpiL, B=cpiB, frequency=freq):
            return cycles(a, l, b, A=A, L=L, B=B) / frequency
```

```
    def parallel(a, l, b, n_processors, A=cpiA, L=cpiL, B=cpiB):
        return exectime( a/(0.7*n_processors), l/(0.7*n_processors), b, A=A, L=L, B=B )
```

```
In [10]: '''
         1.9
         a)  [5] <COD §1.7> Find the total execution time for this program on 1, 2, 4, and 8
             processors, and show the relative speedup of the 2, 4, and 8 processor result
             relative to the single processor result.
         '''
         print('# a)')

         print('#      arith      ld/st      branch      cycles  ')
         print('#------------------------------------')
         print('s1   {:9.3e}  {:9.3e}  {:9.3e}'.format(arith, ls, branch), end='')
         print('   {:>9s}  {:>9s}'.format(
                 engfmt(arith*cpiA + ls*cpiL + branch*cpiB), engfmt(exectime(arith, ls, branch
         print('#------------------------------------')
         for p in (1,2,4,8):
             factor = 1/(0.7*p)
             print('p{:d}   {:9.3e}  {:9.3e}  {:9.3e}'.format(
                 p, factor*arith, factor*ls, branch), end='')
             print('   {:>9s}'.format(engfmt(factor*arith*cpiA + factor*ls*cpiL + branch*cpiB)
         print('#------------------------------------')

         print()

         single = exectime(arith, ls, branch)
         print('a)  Single-processor cycles:', engfmt(cycles(arith, ls, branch)))
         print('      Single-processor time:', engfmt(single))

         one = parallel(arith, ls, branch, 1)
         two = parallel(arith, ls, branch, 2)
         four = parallel(arith, ls, branch, 4)
         eight = parallel(arith, ls, branch, 8)
         print('      one:', engfmt(one), engfmt(one/single), engfmt(one/one))
         print('      two:', engfmt(two), engfmt(two/single), engfmt(two/one))
         print('     four:', engfmt(four), engfmt(four/single), engfmt(four/one))
         print('    eight:', engfmt(eight), engfmt(eight/single), engfmt(eight/one))

         fourOrig = four  # used in part c)
```

```
# a)
#      arith      ld/st      branch      cycles
#------------------------------------
s1   2.560e+09  1.280e+09  2.560e+08    19.200e9      9.600
#------------------------------------
p1   3.657e+09  1.829e+09  2.560e+08    26.880e9
p2   1.829e+09  9.143e+08  2.560e+08    14.080e9
```

```
p4   9.143e+08  4.571e+08  2.560e+08     7.680e9
p8   4.571e+08  2.286e+08  2.560e+08     4.480e9
#-----------------------------------

a)  Single-processor cycles: 19.200e9
      Single-processor time: 9.600
      one: 13.440 1.400 1.000
      two: 7.040 0.733 0.524
     four: 3.840 0.400 0.286
    eight: 2.240 0.233 0.167
```

```
In [11]: '''
         1.9
         b)  [10] <COD ğğ1.6, 1.8> If the CPI of the arithmetic instructions was doubled,
             what would the impact be on the execution time of the program on 1, 2, 4, or 8
             processors?
         '''
         # b)
         single2 = exectime(arith, ls, branch, A=(2*cpiA))
         print('b)  Single-processor cycles:', engfmt(cycles(arith, ls, branch, A=(2*cpiA))))
         print('      Single-processor time:', engfmt(single2))

         one2 = parallel(arith, ls, branch, 1, A=2*cpiA)
         two2 = parallel(arith, ls, branch, 2, A=2*cpiA)
         four2 = parallel(arith, ls, branch, 4, A=2*cpiA)
         eight2 = parallel(arith, ls, branch, 8, A=2*cpiA)
         print('      one:', engfmt(one2), engfmt(one2/single2))
         print('      two:', engfmt(two2), engfmt(two2/single2))
         print('     four:', engfmt(four2), engfmt(four2/single2))
         print('    eight:', engfmt(eight2), engfmt(eight2/single2))
```

```
b)  Single-processor cycles: 21.760e9
      Single-processor time: 10.880
      one: 15.269 1.403
      two: 7.954 0.731
     four: 4.297 0.395
    eight: 2.469 0.227
```

```
In [12]: '''
         c)  [10] <COD ğğ1.6, 1.8> To what should the CPI of load/store instructions be
             reduced in order for a single processor to match the performance of four
             processors using the original CPI values?
         '''
         # c)
         '''
         fourOrig = arith/(0.7*4) * cpiA + ls/(0.7*4) * cpiL + branch * cpiB
```

```
    singleNew = arith*cpiA + ls*cpiLnew + branch*cpiB = fourOrig

    arith*cpiA + ls*cpiLnew = arith/(0.7*4) * cpiA + ls/(0.7*4) * cpiL
    ls * cpiLnew = arith*(1/(0.7*4) - 1) * cpiA + ls/(0.7*4) * cpiL
    cpiLnew = (  arith*(1/(0.7*4) - 1) * cpiA + ls/(0.7*4) * cpiL ) / ls

    arith*cpiA + ls*cpiLnew = (arith*cpiA + ls*cpiL) / (0.7*4)
    ls * cpiLnew =  (arith*cpiA + ls*cpiL) / (0.7*4) - arith*cpiA
    cpiLnew = ( (arith*cpiA + ls*cpiL) / (0.7*4) - arith*cpiA ) / ls
    '''
    cpiLnew = ( (arith*cpiA +ls*cpiL)/(0.7*4) - arith*cpiA ) / ls
    print('c)  new load/store CPI =', engfmt(cpiLnew))
```

```
c)  new load/store CPI = 3.000
```

11 ——————————————————————————————-

12 question #6

See:

> ... based on empirical observations of yields at integrated circuit factories, with the exponent related to the number of critical processing steps.
>
> Hence, depending on the defect rate and the size of the die and wafer, costs are generally not linear in the die area.

12.0.1 1.10

1.10 Assume a 15 cm diameter wafer has a cost of 12, contains 84 dies, and has 0.020 defects/cm2. Assume a 20 cm diameter wafer has a cost of 15, contains 100 dies, and has 0.031 defects/cm2.
1.10.1 [2] <COD ğ1.5> Find the yield for both wafers.
1.10.2 [2] <COD ğ1.5> Find the cost per die for both wafers.
1.10.3 [2] <COD ğ1.5> If the number of dies per wafer is increased by 10% and the defects per area unit increases by 15%, find the die area and yield.
1.10.4 [2] <COD ğ1.5> Assume a fabrication process improves the yield from 0.92 to 0.95. Find the defects per area unit for each version of the technology given a die area of 200 mm2.

```
In [13]: class Wafer:
             def __init__(self, diameter=0, cost=0, count=0, defect_rate=0.0):
                 self.diameter = diameter   # in centimeters
                 self.area = math.pi * (self.diameter/2)**2

                 self.cost = cost   # relative cost
                 self.count = count
                 self.defect_rate = defect_rate
```

```
            def die_area(self):
                return self.area / self.count

            def die_yield(self):
                '''fraction of good dies (determined empirically?)'''
                return 1 / (1 + (self.defect_rate * self.die_area()/2))**2
        #--------

        wafer_A = Wafer(15, 12, 84, 0.020)
        wafer_B = Wafer(20, 15, 100, 0.031)

        #--------------------------------------------------------------------

        print('# 1.10.1')

        print('wafer A ({} cm): {} sqcm, {} yield'.format(
                wafer_A.diameter, engfmt(wafer_A.die_area()), engfmt(wafer_A.die_yield()))
             )
        print('wafer B ({} cm): {} sqcm, {} yield'.format(
                wafer_B.diameter, engfmt(wafer_B.die_area()), engfmt(wafer_B.die_yield()))
             )
```

```
# 1.10.1
wafer A (15 cm): 2.104 sqcm, 0.959 yield
wafer B (20 cm): 3.142 sqcm, 0.909 yield
```

```
In [14]: print('# 1.10.2')
         print('  For good dies:')
         wafer_A.cost_per_die = wafer_A.cost / (wafer_A.count * wafer_A.die_yield())
         wafer_B.cost_per_die = wafer_B.cost / (wafer_B.count * wafer_B.die_yield())
         print('    wafer A cost per die:', engfmt(wafer_A.cost_per_die))
         print('    wafer B cost per die:', engfmt(wafer_B.cost_per_die))

         print('  For all dies:')
         wafer_A.cost_per_die = wafer_A.cost / wafer_A.count
         wafer_B.cost_per_die = wafer_B.cost / wafer_B.count
         print('    wafer A cost per die:', engfmt(wafer_A.cost_per_die))
         print('    wafer B cost per die:', engfmt(wafer_B.cost_per_die))
```

```
# 1.10.2
  For good dies:
    wafer A cost per die: 0.149
    wafer B cost per die: 0.165
  For all dies:
    wafer A cost per die: 0.143
    wafer B cost per die: 0.150
```

```
In [15]: print('# 1.10.3')
         wafer_Anew = Wafer(
                     wafer_A.diameter,
                     wafer_A.cost,
                     wafer_A.count * 1.10,
                     wafer_A.defect_rate * 1.15
         )
         print('wafer A-new ({} cm): {} sqcm, {} yield'.format(
                 wafer_Anew.diameter, engfmt(wafer_Anew.die_area()), engfmt(wafer_Anew.die_yiel
              )

         wafer_Bnew = Wafer(
                     wafer_B.diameter,
                     wafer_B.cost,
                     wafer_B.count * 1.10,
                     wafer_B.defect_rate * 1.15
         )
         print('wafer B-new ({} cm): {} sqcm, {} yield'.format(
                 wafer_Bnew.diameter, engfmt(wafer_Bnew.die_area()), engfmt(wafer_Bnew.die_yiel
              )
```

```
# 1.10.3
wafer A-new (15 cm): 1.912 sqcm, 0.957 yield
wafer B-new (20 cm): 2.856 sqcm, 0.905 yield
```

```
In [16]: print('# 1.10.4')
         # Per the solution set:
         def defects_per_area(yld, die_area):
             sqrtyld = yld**0.5
             return (1 - sqrtyld) / (sqrtyld * die_area / 2)
         #--------

         die_area = 2   # sq cm (200 sq mm)
         for die_yield in [0.92, 0.95]:
             print('defects w/ {:.3f} yield: {:.3f}'.format(die_yield, defects_per_area(die_yi
```

```
# 1.10.4
defects w/ 0.920 yield: 0.043
defects w/ 0.950 yield: 0.026
```

13 ——————————————————————-

14 question #7

14.0.1 1.14

```
In [17]: '''
         Assume a program requires the execution of 50e6 FP instructions,
         110e6 INT instructions, 80e6 L/S instructions, and 16e6 branch instructions.
         The CPI for each type of instruction is 1, 1, 4, and 2, respectively.
         Assume that the processor has a 2 GHz clock rate.

         1.14.1 [10] <COD ğ1.10> By how much must we improve the CPI of FP instructions
             if we want the program to run two times faster?

         1.14.2 [10] <COD ğ1.10> By how much must we improve the CPI of L/S instructions
             if we want the program to run two times faster?

         1.14.3 [5] <COD ğ1.10> By how much is the execution time of the program improved
             if the CPI of INT and FP instructions is reduced by 40% and the CPI of
             L/S and Branch is reduced by 30%?

         '''
         print('# 1.14')

         pgm = {'fp':50e6, 'int':110e6, 'ls':80e6, 'br':16e6}
         cpis = {'fp':1, 'int':1, 'ls':4, 'br':2}
         clock = 2e9    # 2 GHz

         c_fp = pgm['fp'] * cpis['fp']
         c_int = pgm['int'] * cpis['int']
         c_ls = pgm['ls'] * cpis['ls']
         c_br = pgm['br'] * cpis['br']
         c_baseline = c_fp + c_int + c_ls + c_br
         t_baseline = c_baseline / clock
         print('{:g} fp cycles:', c_fp)
         print('baseline execution: {:g} cycles / {:g} secs'.format(c_baseline, t_baseline))
```

```
# 1.14
{:g} fp cycles: 50000000.0
baseline execution: 5.12e+08 cycles / 0.256 secs
```

```
In [18]: # 1.14.1
         print('\n# 1.14.1')
         c_fp_new = (c_baseline / 2) - c_int - c_ls - c_br
         cpi_fp_new = c_fp_new / pgm['fp']

         print('old FP cycles: {:.3g}  CPI: {:d}'.format(c_fp, cpis['fp']))
```

```
        if c_fp_new < 0:
            print('Impossible speedup', end=' --- ')
        print('new FP cycles: {:.3g}  CPI: {:.3g}'.format(c_fp_new, cpi_fp_new))
```

```
# 1.14.1
old FP cycles: 5e+07  CPI: 1
Impossible speedup --- new FP cycles: -2.06e+08  CPI: -4.12
```

```
In [19]: # 1.14.2
         print('# 1.14.2')
         c_ls_new = (c_baseline / 2) - c_fp - c_int - c_br
         cpi_ls_new = c_ls_new / pgm['ls']

         print('old L/S cycles: {:.3g}  CPI: {:d}'.format(c_ls, cpis['ls']))
         if c_ls_new < 0:
             print('Impossible speedup', end=' --- ')
         print('new L/S cycles: {:.3g}  CPI: {:.3g}'.format(c_ls_new, cpi_ls_new))
```

```
# 1.14.2
old L/S cycles: 3.2e+08  CPI: 4
new L/S cycles: 6.4e+07  CPI: 0.8
```

```
In [20]: # 1.14.3
         print('# 1.14.3')
         c_fp = pgm['fp'] * (cpis['fp'] * 0.6)
         c_int = pgm['int'] * (cpis['int'] * 0.6)
         c_ls = pgm['ls'] * (cpis['ls'] * 0.7)
         c_br = pgm['br'] * (cpis['br'] * 0.7)
         c_baseline = c_fp + c_int + c_ls + c_br
         t_baseline = c_baseline / clock

         print('new execution: {:g} cycles / {:g} secs'.format(c_baseline, t_baseline))
```

```
# 1.14.3
new execution: 3.424e+08 cycles / 0.1712 secs
```

15 ————————————————————————-

16 question #8

16.0.1 1.15

```
In [21]: '''
         1.15 [5] <COD ğ1.8>
```

```
When a program is adapted to run on multiple processors in
a multiprocessor system, the execution time on each processor
is comprised of computing time and the overhead time required
for locked critical sections and/or to send data from one
processor to another.

Assume a program requires t = 100 s of execution time on one
processor. When run p processors, each processor requires t/p s,
as well as an additional 4 s of overhead, irrespective of the
number of processors. Compute the per-processor execution time
for 2, 4, 8, 16, 32, 64, and 128 processors.
For each case, list the corresponding speedup relative to
a single processor and the ratio between actual speedup versus
ideal speedup (speedup if there was no overhead).
'''
print('# 1.15')
t_one = 100

def execution_time(t_one, p):
    time_per_processor = 4 + t_one / p
    return time_per_processor

for nprocs in (2,4,8,16,32,64,128):
    time = execution_time(t_one, nprocs)
    speedup = t_one / time
    theory = t_one / nprocs
    speedup_theory = t_one / theory
    print('%7.2f  %5.2f    %5.2f  %6.2f    %.4f' \
              % (time, speedup, theory, speedup_theory,
                 speedup / speedup_theory))
```

```
# 1.15
  54.00   1.85    50.00    2.00    0.9259
  29.00   3.45    25.00    4.00    0.8621
  16.50   6.06    12.50    8.00    0.7576
  10.25   9.76     6.25   16.00    0.6098
   7.12  14.04     3.12   32.00    0.4386
   5.56  17.98     1.56   64.00    0.2809
   4.78  20.92     0.78  128.00    0.1634
```