Problem 1 (20pts).
To solve this problem we apply Amdahl’s law several times.

(a) Let F be the “percent vectorization” and S=12.5.

\[
\text{Net speedup} = \frac{1}{(1-F) + F \times S/12.5} = \frac{1}{1 - 0.92F}.
\]

(b) To have a net speedup of 2 we need \(1-0.92F = 1/2\) or \(F=0.5/0.92=0.5435\)

(c) The maximum speedup is 12.5. To get half the maximum speedup we need \(1-0.92F = 1/6.25\) or \(F=0.84/0.92=0.9130\). We need that more than 90% of the code is vectorizable.

(d) Hardware solution: net speedup =

\[
\frac{1}{(1-F) + 0.08xF/2} = \frac{1}{1 - 0.96xF} = \frac{1}{1 - 0.96 \times 0.7} = 3.04
\]

Software solution: We’d need \(\frac{1}{1 - 0.9xF} = 3.04\) or \(1/3.04 = 1 - 0.92F\) or \(F=.73\)

Thus a relatively small software effort may be better than a large hardware effort, as “percent vectorization” only has to increase from .7 to .73 to yield performance as good as a doubling of FP hardware speed.
Note however that vectorizing more code may not be always possible.

**Problem 2 (20pts).**

First let’s look at the original machine.

\[
\text{CPI}_{\text{original}} = 0.48x1 + 0.24x2 + 0.12x2 + 0.16x1 = 1.36 \text{ and } T_{\text{original}} = 1.36 \times \text{IC}_{\text{original}} \times T_c
\]

In the new machine, we are able to effectively use the new ALU instruction format for 25% of ALU instructions. These new ALU instructions now take each two cycles, while the rest of ALU instructions still take one cycle. Branches now take three cycles.

Another effect is that the number of loads is reduced from \(0.24 \times \text{IC}_{\text{original}}\) to \((0.24 - 0.48 \times 0.25)\times \text{IC}_{\text{original}} = 0.12\times \text{IC}_{\text{original}}\)

Therefore, the total time (in cycles) to execute the new mix is:

\[
\text{Time}_{\text{load}} + \text{Time}_{\text{store}} + \text{Time}_{\text{ALU}} + \text{Time}_{\text{Branch}} = 0.12 \times 2 \times \text{IC}_{\text{original}} + 0.12 \times 2 \times \text{IC}_{\text{original}} + 0.48 \times 2.5 \times 2 \times \text{IC}_{\text{original}} + 0.48 \times 0.75 \times 1 \times \text{IC}_{\text{original}} + 0.16 \times 2 \times \text{IC}_{\text{original}} = 1.4 \times \text{IC}_{\text{original}}
\]

We see that the new instruction is not a good idea in the context of this mix.

**Problem 3 (20pts)**

In the optimized code 1/3 of the load/stores are removed. Thus the optimized instruction count is \((0.3 \times 2) / (3 + 7) = 0.9\) times the unoptimized instruction count. Since all instructions take one cycle, the number of cycles in the optimized machine is also 90% of the number of cycles in the non-optimized machine.

On the other hand the clock RATE of the unoptimized machine is 1.05 times the clock RATE of the optimized machine.

We have: \(T_{\text{exe}} = (\# \text{ of clock cycles}) \times \text{clock cycle} = (\# \text{ of clock cycles}) / \text{clock rate}\).

Thus, \(T_{\text{exe}}_{\text{opt}} = (0.9 \times 1.05) \times T_{\text{exe}}_{\text{unopt}} = 0.945 T_{\text{exe}}_{\text{unopt}} \) and the speedup of the optimized machine is \(1/0.945 = 1.0582\).

Thus the optimized machine is 6% faster than the unoptimized machine. Good idea!

**Problem 4 (20pts).**

(a) We first find the number of dies per wafer by using the formula on page 12 with (wafer diameter) = 20cm and (Die area) = 256mm\(^2\); the result is (Dies per wafer) = 94.

Then we compute the (Die yield) from the formula on page 12, with (wafer yield)=.95, alpha=3, (die area)=256mm\(^2\) and defect per unit area of 1/cm\(^2\). The result is (Die yield) = 0.15. The number of (good dies per wafer) is 94\times 0.15=14.

(b) (Die cost)= (wafer cost)/(good die perwafer) =4000/14=285.70

(c) (IC cost) = (Die cost) + (Testing cost) + (packaging cost) = 285.70 + 320x10/(.15x3600) + 30 = 321.70 (since final test yield is 1).

(d) Using the same equations as above, we can find die yeild for defect densities 0.6/cm\(^2\) and 1.2/cm\(^2\) as 0.27 and 0.11 respectively. No of good dies per wafer will be 27.26 and 11.28 respectively. Putting these values in the equation of b we get Die cost as 157.60$ and 386.80$ respectively.
Testing cost will be 3.30$ and 8.10$.
Adding them up IC costs will be 190.90$ and 424.90$ respectively for .6/cm2 and 1.2/cm2 defect density.

The difference in cost come from the die yield which is much higher for lower defect densities as expected, since less defects mean less bad chips, which in turn mean that good chips have to cover less of the cost of bad chips.

**Problem 5 (10pts).**
(a) Native MFLOPS = (# of FP operations)/(execution time in second)\times10^6
In this case, native MFLOPS = 109.97/94 = 1.17

(b) For the normalized MFLOPS, we need to count 4 FP ops per divide. There are no functions in spice.
The result is: 157.017/94 = 1.67.

Thus the normalized MFLOPS is almost 50% higher than the native MFLOPS.

**Problem 6 (20pts)**
(a) From the statement of the problem we shall classify instructions in three groups:

1. Load/store have 16 bits + 8 or 16 for the offset and constitute 36% of all instructions.
Looking at Fig. 2.32, we see that 17% will be 16 bits long, 40% (line 7 - line0) will be 24 bits long and 43% (line 15 - line 7) will be 32 bits long.
2. Branch and jumps have 16 bits + 8 or 16 bits of displacement and constitute 16% of all instructions.
Looking at Fig. 2.32, we see that 93% (line 7 (i.e. 8 - 1 for the sign bit)) will be 24 bits long and 7% will be 32 bits.
3. ALU instructions are 16 bits and constitute 48% of all instructions

Overal, we have the following size distribution:

16 bits: .36x.17 (load/stores) + .48 (ALU) = .5412
24 bits: .36x.4 (load/stores) + .16x.93 (Branch) = .2928
32 bits: .36x.43 (load/stores) + .16x.07 (Branch) = .166

The average instruction size is: .5412x16 + .2928x24 + .166x32 = 20.9984

(b) Again, we classify instructions in three categories:

1. Load/store: 43% of load/store instructions need two words of 24 bits.
2. Branch and jumps: 7% of branch/jumps require two 24-bit words
3. ALU: one word only.

So, the average number of bits per instructions is:
24 + .43x.36x24 (loads/stores) + .07x.16x24 (branch) = 27.984 bits.

This length is worse that in (a). This is expected as the minimum size of an instruction is now longer than in (a).

(c) Now all load/store and branch/jump instructions need 32 bits while all other need 16 bits.
The average number of bytes for each instruction is now:
.52x32 (load/store/branch/jump) + .48x16 (other) = 24.48 bits

This solution has shorter code sequences than (b).

**Problem 7 (10pts).**

(a) In this problem it is stated that the CPI is not changed between the two machines.

\[ \text{Texe} = \text{IC} \times \text{CPI} \times Tc \]

Since CPI is unchanged, \( \frac{T\text{exe,new}}{T\text{exe,old}} = \frac{\text{IC}_{\text{new}}}{\text{IC}_{\text{old}}} \times \frac{Tc_{\text{new}}}{Tc_{\text{old}}} = \frac{\text{IC}_{\text{new}}}{\text{IC}_{\text{old}}} \times 1.1 = 1 \)

Thus \( \frac{\text{IC}_{\text{new}}}{\text{IC}_{\text{old}}} = \frac{1}{1.1} = 0.91 \).

Now there are 24% of loads. Since only loads can be removed from the code, we must have that:

\[ \text{IC}_{\text{new}} = [0.24x(1-\text{Fld}) + (1 - 0.24)] \times \text{IC}_{\text{old}} \]

where \( \text{Fld} \) is the fraction of loads removed

Thus we must have \( [0.24x(1-\text{Fld}) + (1 - 0.24)] = 0.91 \) or \( 1-\text{Fld} = 0.15/0.24 = 0.625 \)

37.5% of loads must be removed from the code, not an easy task!

(b) Sequences of code where the same register is used in both instructions will not work.

For example, in the old machine, if we had:

\[ \text{LW} \ r1,0(\ r1) \]
\[ \text{ADD} \ r1,\ r1,\ r1 \]

we could not replace this with:

\[ \text{ADD} \ r1,0(\ r1) \]

in the new machine, because in the first case the memory value affects both input operand of the ADD, whereas in the second case it only affects one of them.

**Problem 8 (25pts)**

Consider the following sequence:

\[ \text{B}=\text{A}+\text{B} \]
\[ \text{C} = \text{B}-\text{C} \]
\[ \text{D} = \text{C}+\text{B} \]

(a) Accumulator:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Code bytes</th>
<th>Data bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load A</td>
<td>(A) in accu.;</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Add B</td>
<td>add (B) from accu;</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Store B</td>
<td>store acc to (B);</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Sub C</td>
<td>/3 code bytes;</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Store C</td>
<td>/3 code bytes;</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Add B</td>
<td>/3 code bytes;</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Store D</td>
<td>/3 code bytes;</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

The total for code is 21 bytes. The total for data is 28 bytes. The total memory bandwidth (data+instructions) is 49 bytes.
(b) Memory-Memory

\[
\begin{align*}
\text{ADD } B,A,B & \quad /A+B\rightarrow B; \quad 7 \text{ code bytes}; \quad 12 \text{ data bytes} \\
\text{SUB } C,B,C & \quad /7 \text{ code bytes}; \quad 12 \text{ data bytes} \\
\text{ADD } D,C,B & \quad /7 \text{ code bytes}; \quad 12 \text{ data bytes}
\end{align*}
\]

The total for code is 21 bytes. The total for data is 36 bytes. The total memory bandwidth (data+instructions) is 57 bytes.

(c) Stack

\[
\begin{align*}
PUSH A & \quad /A \text{ on TOS}; \quad 3 \text{ code bytes}; \quad 4 \text{ data bytes} \\
PUSH B & \quad /B \text{ on TOS}; \quad 3 \text{ code bytes}; \quad 4 \text{ data bytes} \\
ADD & \quad /\text{remove copies of } A \& B \text{ from TOS, sub and put result on TOS}; \quad 1 \text{ code byte} \\
POP B & \quad /\text{store result in } B; \quad 3 \text{ code bytes}; \quad 4 \text{ data bytes} \\
PUSH B & \quad /3 \text{ code bytes}; \quad 4 \text{ data bytes} \\
PUSH C & \quad /3 \text{ code bytes}; \quad 4 \text{ data bytes} \\
SUB & \quad /1 \text{ code byte} \\
POP C & \quad /3 \text{ code bytes}; \quad 4 \text{ data bytes} \\
PUSH C & \quad /3 \text{ code bytes}; \quad 4 \text{ data bytes} \\
PUSH B & \quad /3 \text{ code bytes}; \quad 4 \text{ data bytes} \\
ADD & \quad /1 \text{ code byte} \\
POP D & \quad /3 \text{ code bytes}; \quad 4 \text{ data bytes}
\end{align*}
\]

The total for code is 30 bytes. The total for data is 36 bytes. The total memory bandwidth (data+instructions) is 66 bytes.

(d) Load/store

\[
\begin{align*}
\text{load } r1,A & \quad /A\rightarrow r1; \quad 4 \text{ code bytes}; \quad 4\text{ data bytes} \\
\text{load } r2,B & \quad /B\rightarrow r2; \quad 4 \text{ code bytes}; \quad 4 \text{ data bytes} \\
\text{add } r2,r1,r2 & \quad /r1+r2\rightarrow r2; \quad 3 \text{ code bytes}; \\
\text{store } r2,B & \quad /r2\rightarrow B; \quad 4 \text{ code bytes}; \quad 4 \text{ data bytes} \\
\text{load } r3,C & \quad /4 \text{ code bytes} \quad 4 \text{ data bytes} \\
\text{sub } r3,r2,r3 & \quad /3\text{code bytes} \\
\text{store } r3,C & \quad /4 \text{ code bytes}; \quad 4 \text{ data bytes} \\
\text{add } r4,r2,r3 & \quad /r3+r2\rightarrow r4; \quad 3\text{code bytes} \\
\text{store } r4,D & \quad /4 \text{ code bytes}; \quad 4 \text{ data bytes}
\end{align*}
\]

The total for code is 34 bytes. The total for data is 24 bytes. The total memory bandwidth (data+instructions) is 58 bytes.
Let’s compare the 4 machines for the code fragment.

### Table 1: Number of bytes transmitted between processor and memory

<table>
<thead>
<tr>
<th>Memory bandwidth</th>
<th>Accumulator</th>
<th>Memory-to-memory</th>
<th>Stack</th>
<th>Load/store</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>21</td>
<td>21</td>
<td>30</td>
<td>34</td>
</tr>
<tr>
<td>DATA</td>
<td>28</td>
<td>46</td>
<td>36</td>
<td>24</td>
</tr>
<tr>
<td>CODE+DATA</td>
<td>49</td>
<td>57</td>
<td>66</td>
<td>58</td>
</tr>
</tbody>
</table>

The table shows that the best machine for code sizes are the accumulator and memory-to-memory machines. On the other hand the load/store architecture has the best memory traffic for data. For this particular code, Load/Store loses to Accumulator and MtoM because of big code size. (This result should not be generalized, actually in practise Load/Store architectures are able to reduce the number of instructions a good deal, this is possible due to compiler optimizations that are available in the flexible Load/Store environment)

### Problem 9 (20pts).
The following DLX code is a possible translation of the C code.

```assembly
start:
  ADDI R1,R0,#0 /init i
  ADDI R3,R0,#0 /base offset of A
  ADDI R4,R0,#5000 /base offset of B
  LW R5,1500(R0) /LD value of C
loop:
  MULT R2,R1,#4 /compute word offset
  ADD R6,R2,R4 /compute address of B[i]
  LW R7,0(R6) /load value of B[i]
  ADD R8,R7,R5 /B[i]+c
  ADD R9,R2,R3 /compute address of A[i]
  SW 0(R9),R8 /A[i]<-B[i]+c
  ADDI R1,R1,#1 /increment i
  ADDI R10,R1,#-101 /is counter at 101?
  BNEZ R10,loop /if not 101, repeat
out:
  SW 2000(R0),R1 /store final value of i
```

The total number of instructions executed is the number of setup instructions plus the number of instructions in the loop times the number of iterations plus the number of instructions after the loop, i.e., 4 + (9x101) + 1 = 914.

Data references: 1 + (2x101) + 1 = 204.

Code size=(# of instructions in the code)x4 = 14x4 = 56 bytes.