Question 1:
In the IEEE 754 notation, a single precision floating point number is represented as

(a) [2 points]: represent $0.75 \times 2^{-5}$ as a single precision floating point number, showing the bit patterns. Make sure you delineate the various components carefully, to help the TA in grading!

$$0.75 \times 2^{-5} = 0.11 \times 2^{-5} = 1.1 \times 2^{-6}$$

$S = 0$

$E = -5 + 127 = 122 = 1001110$

$M = 10000000000000000000000$

(b) [2 points]: represent $-6.25 \times 2^{16}$ as a single precision floating point number

$$-6.25 \times 2^{16} = -110.01 \times 2^{16} = -1.1001 \times 2^{18}$$

$S = 1$

$E = 18 + 127 = 145 = 10010001$

$M = 10010000000000000000000$

$$-1.25 \times 2^{16} = -1.01 \times 2^{16}$$

$S = 1$

$E = 16 + 127 = 143 = 1001111$

$M = 01000000000000000000000$

(c) [2 points]: Describe the steps you would need to take to multiply 2 floating point numbers stored in the scheme above.

1) add exponents and correct bias
2) multiply significands
3) normalize product
4) round product significand
5) decide sign
(d)[4 points]: Find $r = -6.42 \times 10^1 \times 9.51 \times 10^2$ assuming that the output has only 3 significant digits. Do this first with guard and round digits [2 pts] and then without any extra digits[2 pts]. (Work in base-ten arithmetic)

\[-6.42 \times 10^1 \times (9.51 \times 10^2) = 6.10542 \times 10^4\]

with guard and round: $6.11 \times 10^4$

without: $6.10 \times 10^4$

**Question 2:**

During a typical compile, gcc profiles into the following classes of operations

<table>
<thead>
<tr>
<th>Operation Class</th>
<th>% time spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
</tr>
</tbody>
</table>

On our multi-cycle machine developed in class we know that the individual classes of operations mentioned above take

<table>
<thead>
<tr>
<th>Operation Class</th>
<th>Number of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>4</td>
</tr>
<tr>
<td>Load</td>
<td>5</td>
</tr>
<tr>
<td>Store</td>
<td>4</td>
</tr>
<tr>
<td>Branch</td>
<td>3</td>
</tr>
</tbody>
</table>

We have access to a machine running at 100MHz.

(a)[4 points]: For a specific compile, Gcc has to go through 1 million instructions. How long would this compile take ?

Assume the compile will take $X$ cycles,

$0.5X/4 + 0.2X/5 + 0.1X/4 + 0.2X/3 = 10^6$

$(0.125 + 0.04 + 0.025 + 0.067 ) X = 10^6$

$X = 3.89 \times 10^6$ cycles

$EX = 3.89 \times 10^6 / (100 \times 10^6) = 0.0389$ Seconds

(b)[3 points]: Your architecture team says that they can play some tricks and reduce the number of cycles that ALU operations need to 3. How fast is the compile now ?
¼ of the ALU time is reduced. ¼ * 50% = 12.5% of the original time

So New time is (1 – 12.5%) * 0.0389 = 87.85% * 0.0389 = 0.034 Seconds

(c)[3 points]: Your processing lab team says that for your current design, they can give you a clock speed that is slightly faster by playing with the memory architecture during the load cycle. However they cannot promise you any such speedups if the architecture people change the design as they propose in section(b). What clock speed would they have to deliver, for you to consider not changing the design and simply increasing the clock speed?

\[
\frac{100}{87.5\%} = 114 \text{ M}
\]

**Question 3:**
This problem generates a version of “pseudo-instructions”. Except we will go across architectures to do this.

3.a[3 pts] The PowerPC has a load-update instruction, which in a single cycle does the following

\[
\text{lwu, } $t1, 4($s0) \text{; } // \text{$t1} \leftarrow \text{Memory}[$s0+4]; \text{$s0} \leftarrow \text{$s0+4};
\]

(This permits a rather quick and painless way of doing loops). Explain how you would do the equivalent in MIPS assembly?

\[
\text{Lw } $t1, \text{ 4($s0)}
\]

\[
\text{Addi } $s0, \text{ $s0, 4}
\]

3.b[2 pts] The Java Virtual Machine (which represents an ISA in itself) permits a stack based operation called Swap, which swaps the top two integers on the stack. Viz.,

\[
\text{swap ; } // \text{ exchange $sp$ and $sp+4}
\]

Write the assembly code to expand this pseudo-instruction into MIPS assembly.

\[
\text{Lw } $t1, \text{ 0($sp)}
\]

\[
\text{Lw } $t2, \text{ 4($sp)}
\]
3.c [5 pts] The sparc assembly language permits an operation called a “population count” which tells you how many “one-bits” there are in a register. Viz.,

\[
\text{popc } \texttt{t1}, \texttt{t0} ; \quad / / \texttt{t1} \leftarrow \text{Number of Ones in } \texttt{t0}
\]

Write the equivalent in assembly. Assuming the register is 32 bits wide, and you have a single cycle instruction machine (viz., all instructions take one clock cycle) how many clocks does it take you to do this in MIPS? (hint: note the shift instructions in the MIPS-ISA)

There are many ways to do this, here is one way:

\[
\begin{align*}
\text{Add} & \quad \texttt{t1} & \texttt{zero} & \texttt{zero} \\
\text{Add} & \quad \texttt{S2} & \texttt{zero} & \texttt{zero} \\
\text{Addi} & \quad \texttt{S1} & \texttt{zero} & 32 \\
\text{Loop:} & \\
\text{Slti} & \quad \texttt{t3} & \texttt{t0} & 0 & \quad \text{# is the highest bit 1?} \\
\text{Add} & \quad \texttt{t1} & \texttt{t1} & \texttt{t3} \\
\text{Sll} & \quad \texttt{t0} & \texttt{t0} & 1 \\
\text{Addi} & \quad \texttt{S2} & \texttt{S2} & 1 \\
\text{Bne} & \quad \texttt{S1} & \texttt{S2} & \text{loop}
\end{align*}
\]

It will take 163 cycles to finish this

3.d [5 pts] The pentium permits a relatively complex instruction called load string (we shall call this LDS) which does the following:

\[
\text{LDS } \texttt{t1}, \texttt{t0} ; \quad / / \text{Loads 1/2/4-byte value from memory location refererred by } \texttt{t0}
\]

\[
\begin{align*}
& / / \texttt{t1} \leftarrow \text{Mem[}$\texttt{t1}$] \\
& / / \text{subject to register } \texttt{a0} \text{ being set to 1,2, 4} \\
& / / \text{and then increments or decrements } \texttt{t0}, \text{based on whether}
\end{align*}
\]

$\texttt{a1}$ is 0/1

viz.,

\[
\text{switch}($\texttt{a0}$) {
\begin{align*}
\text{case ‘1’: load 1 byte into } \texttt{t1} \text{ from Mem[}$\texttt{t0}$] \\
\text{if ( } \texttt{a1} \text{ == 0) } \{ \texttt{t0} \text{ += 1; } \}
\end{align*}
\]
And all this in a single assembly level instruction (remember, the Pentium is NOT RISC).

Write the equivalent in MIPS assembly.

Addi $t2 $zero 1
Addi $t3 $zero 2
Addi $t4 $zero 4
Beq $a0 $t2 case1
Beq $a0 $t3 case2
Beq $a0 $t4 case4

Case1:
    Lb $t1 0($t0)
    Add $t5 $t2
    J finish

Case2:
    LH $t1 0($t0)
    Add $t5 $t3
    J finish

Case4:
    LW $t1 0($t0)
    Add $t5 $t4

Finish:
    Beq $a1 $zero addn
    Beq $a1 $t2 subn
    J end

Addn:
    Add $t0 $t0 $t5
    J end

Subn:
    Sub $t0 $t0 $t5

End:
**Question 4**

4.a [2 pts] Explain why the last two bits of the PC are typically hard wired to 00?

Since all the instructions in MIPS are 32 bits, which is 4 bytes, the address for an instruction is always some multiples of 4, which means the last two bits are always 00. PC contains nothing but the address of instructions, thus its last two bits are typically hard wired to 00.

4.b [2 pts] Explain what *type* of instruction uses the adder on the top-right of the picture. The result of this adder goes into a MUX with PC_Src as its control line. What is the value of PC_Src for the *addi* (the add immediate) instruction? Note that the picture contains 0,1 entries in the MUXes corresponding to the values they will pass through for relevant values of the control lines.

Jump and branch instructions use the adder on the top-right to calculate the next instruction address.
For addi, no such change of pc is needed, so the MUX entry is 1, and selects pc=pc+4

4.c [3 pts] Explain what functional blocks get used when doing the *ori* (Or Immediate) instruction?

PC, Adder for PC + 4, Instruction memory, Register files, ALU, RegDst MUX, PCSrc MUX, ALUSrs MUX, MemtoRegMUX

4.d [3 pts] Explain the *path* of data during a *load* instruction. (feel free to draw on the picture?)

see the picture

**Question 5:**
Consider the single cycle datapath (and control) presented here. Modify this datapath to handle unknown opcodes as an exception. In case of an unknown opcode we want to simply jump to the *fixed* static address 0x4000. Note, we do not want to save state, save cause of the exception etc. This is a simplified unknown instruction exception. The net result of this exception would thus look like

If (unknown opcode) \{ $32 = PC; PC = 0x4000 \}. 
(a) [5 points]: Modify the datapath shown to add this functionality. Describe your modifications in detail including the intent of the modification and the actual form.

1) add a mux to provide another source of 0x4000 to pc
2) add a path to connect PC to register $32, and add control signal from instruction [26-31]

(b) [3 points]: The hardware team has really messed up and the data at 0x4000 is corrupt leading to the instruction at 0x4000 being an unknown opcode. Describe the behaviour of this machine under these circumstances.

The machine will go into infinite loop.

(c) [7 points]: Besides firing the hardware team (☹), How would you modify the processor to handle this additional problem? Describe any additional instructions you create as well as any additional modifications in the datapath.

(Any explanation that makes sense will be ok)

such as:

correct data at 0x4000
add control and data path to check if $32=0x4000 already. If so, jump to other instruction address instead of 0x4000
do a incremental jump. 0x4000, 0x4004, ....
Etc…