**Question 1.** (14 points) This question involves **12-bit signed, 2's complement** binary numbers.

(a) Give the 12-bit 2's complement binary and hexadecimal representations of the decimal number -320.

**1110 1100 0000**

**0xEC0**

(b) Translate the 12-bit hexadecimal number 0xF00 to binary and then give its decimal value if we interpret it as a 2's complement binary integer.

**1111 0000 0000**

**-256**

Powers of 2 and 16 for reference.

| Number | Hex | Decimal |
|---|---|---|
| $2^0$ | $16^0$ | 1 |
| $2^1$ | | 2 |
| $2^2$ | | 4 |
| $2^3$ | | 8 |
| $2^4$ | $16^1$ | 16 |
| $2^5$ | | 32 |
| $2^6$ | | 64 |
| $2^7$ | | 128 |
| $2^8$ | $16^2$ | 256 |
| $2^9$ | | 512 |
| $2^{10}$ | | 1024 |
| $2^{11}$ | | 2048 |
| $2^{12}$ | $16^3$ | 4096 |

**Question 2.** (18 points) The pseudo-question. (But you have to provide a real answer)

Consider the following fragment of a MIPS assembly language program.

```
        bgt     $t3, $a0, there
        add     $t3, $t3, $a0
        move    $v0, $t3
there:
        li      $t0, 0x10010110
```

This fragment contains several assembler pseudo instructions as well as a real MIPS machine instruction or two (or maybe more, or maybe less).

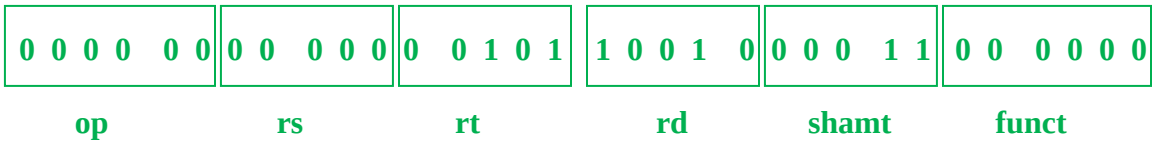
Re-write this fragment of code so it uses only MIPS machine language instructions. You should replace each pseudo-instruction in the code with machine language instructions, as would be done by a MIPS assembler like SPIM.

```
   slt  $at, $a0, $t3          # $at=$t3>$a0; must use $at
   bne  $at, $zero, there
   add  $t3, $t3, $a0
   or   $v0, $t3, $zero        # add ok here instead
there:
   lui  $t0, 0x1001            # could also use $at
   ori  $t0, $t0, 0x0110       # add ok here instead
```

**In grading we were flexible as long as the results were reasonable and would produce the right results. So, for example, `add` instructions would work instead of `or` in these particular cases. The `or` instruction above `there:` could have been an `ori`. Similarly the `lui` could put its result in `$at`. What would not be ok is, for instance, using some register other than `$at` for the result of the `slt` instructin, since that might clobber some needed value in that other register. There would also be problems with `add` instructions instead of `or` if the numbers had the high-order bit set, which would lead to problems with sign-extended negative immediate values. The answers given here are pretty much what a normal assembler would produce.**

**Question 3.** (14 points) Suppose we have a 32-bit MIPS word containing the value `0x000590C0`. We would like to know what MIPS machine instruction this represents.

(a) Write this instruction word in binary. Leave enough spaces between the digits for part (c) of the question.




(b) What is the format of this instruction? (circle)

R I J

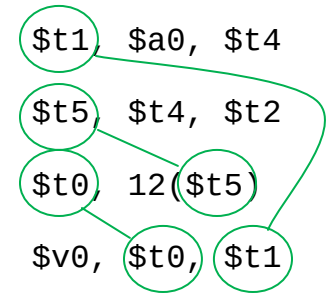
(c) In your answer to part (a), draw boxes around the bits that make up the different fields of the instruction and then label the instruction fields (opcode, rs, etc.)

(d) Translate this instruction to assembly language. Use symbolic register names like `$t0` instead of absolute register numbers like `$8`.

```
sll  $s2, $a1, 3
```

**Question 4.** (20 points) Suppose we have the following four instructions in a fragment of a program.

```
add  $t1, $a0, $t4
add  $t5, $t4, $t2
lw   $t0, 12($t5)
add  $v0, $t0, $t1
```



(a) Identify all of the data dependencies in the above code. You can either write an answer below, or circle the affected registers in the above code and draw lines to indicate the data dependencies between instructions.

(b) Fill in the following pipeline timing diagram to show how those instructions would execute on a machine **with** internal forwarding of data from one instruction to the next. You should also assume that a register can be written with a new value at the beginning of a cycle and the new value can be read out at the end of the same cycle. Indicate a stall by leaving an entry blank or writing "stall" or "NOP". The first three cycles of the first instruction (only) are filled in for you. (You may not need all of the columns.)

| cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `add` | IF | ID | EX | **MEM** | **WB** | | | | | | | | | |
| `add` | | **IF** | **ID** | **EX** | **MEM** | **WB** | | | | | | | | |
| `lw` | | | **IF** | **ID** | **EX** | **MEM** | **WB** | | | | | | | |
| `add` | | | | **IF** | **ID** | **stall** | **EX** | **MEM** | **WB** | | | | | |

(c) If there are any stall cycles in the timing diagram from part (b), is there any way to rearrange or alter these instructions to reduce the number of stalls? If so, how would you do it, and how many stall cycles (if any) are left in the final schedule?

**Yes. Move the first add instruction so it is between the `lw` and final `add`. That will keep the pipeline full while waiting for the MEM cycle in the `lw` to complete.**

**0 stall cycles in the final schedule.**

**Question 5.** (22 points) A little string programming. For this problem, implement a MIPS assembly language procedure `strcat` that appends a copy of one string to the end of another. For instance, if strings `s` and `t` are

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| s | i | c | e | \0 | *?* | *?* | *?* | *?* | *?* | *?* | *?* |
| t | c | r | e | a | m | \0 | *?* | *?* | *?* | *?* | *?* |

then after executing `strcat(s,t)` the strings should look as follows.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| s | i | c | e | c | r | e | a | m | \0 | *?* | *?* |
| t | c | r | e | a | m | \0 | *?* | *?* | *?* | *?* | *?* |

In the diagrams, `\0` indicates a byte containing binary 0 (`0x00`), and *?* indicates bytes whose contents are unknown.

Your code should use the standard MIPS calling conventions. You should assume that the first string has enough unused space at the end to hold a copy of the second string, and that both strings are properly terminated with a `\0` byte at the end. The `strcat` procedure has no return value (i.e., it would be a `void` function in C or Java).

Please supply reasonable comments so we can follow your use of registers and memory.

**(solution on next page)**

(additional room on the next page for the rest of your answer if needed)

**Question 5.** (cont) Additional room for the rest of your `strcat` procedure if needed.

```
# append a copy of the second string argument to the first
# string argument.

strcat:

# Input: $a0 = address of string s (destination)
#        $a1 = address of string t (source)


# Advance #a0 until it contains address of \0 byte

skip:
        lbu     $t0, 0($a0)
        beq     $t0, $zero, done
        addi    $a0, $a0, 1
        j       skip
done:

# copy t to s until \0 byte copied

copy:
        lbu     $t0, 0($a1)
        sb      $t0, 0($a0)
        addi    $t0, $t0, 1
        addi    $t1, $t1, 1
        bne     $t0, $zero, copy

# return to caller

        jr      $ra
```

**Obviously there are many possible solutions and we gave credit for any that worked properly and obeyed the standard conventions about register usage, etc. In particular, `lb` would work just as well as `lbu` in this code, although in general `lbu` would be best for loading 8-bit quantities that we do not want to treat as signed numbers.**

**One error that was surprisingly common was to use `lw`/`sw` instead of `lb`/`sb`. That would produce all sorts of trouble, including unaligned memory access errors, not to mention storing 4 bytes at a time.**

**Question 6.** (12 points) When most programs are executed, their memory references exhibit a property known as *temporal locality.*

(a) What does this term mean? (You can be very brief here.)

**If a program references a particular location in storage, it is likely to reference the same location again soon.**

(b) Why is this property useful when we design caches and memory hierarchies?

**The first reference to any memory location will be a cache miss and will require a slow access to main memory. Because of temporal locality, there is a high likelihood that subsequent references will occur while the value is still in the cache. The average access time for the multiple references will then be the average of the single main memory reference plus the (multiple) cache references, which will be significantly faster than multiple main memory references.**