**Question 1.** (6 points) (a) Rewrite the instruction `sub $v0,$t8,$a2` using absolute register numbers instead of symbolic names (i.e., if the instruction contained `$at`, you would rewrite that as `$1`.)

**`sub $2, $24, $6`**

(b) Rewrite the instruction from part (a) in binary. Use vertical lines to show which fields of the instruction are which, and write the field names (op, rd, etc.) underneath the bits that make up that field.

| 000000 | 11000 | 00110 | 00010 | 00000 | 100010 |
|---|---|---|---|---|---|
| op | rs | rt | rd | shamt | funct |

**Question 2.** (6 points) What is the value of the decimal number -129 if it is converted to a 32-bit, 2's-complement binary integer? Write your answer in hexadecimal.

(Suggestion: It might be useful to show your intermediate steps so if there's a mistake it'll be easier to figure out what happened.)

**`0xffffff7f`**

**Question 3.** (6 points) Suppose we have the following 32-bit word labeled `n` located at address `0x1000001c` in an assembly language program.

```
[0x1000001c]   n:  .word    42
```

MIPS assembly language allows us to write `lw $t0,n` to load this word into register `$t0`. Although `lw` is a real machine instruction and not a pseudo-instruction, this line of assembly language code cannot be translated into a single machine instruction because the address `0x1000001c` does not fit into the 16-bit offset field in a single `lw` instruction. Give a sequence of two actual machine instructions that perform the desired `lw $t0,n` operation and that could be used in a program without interfering with surrounding instructions. Write your answer using symbolic assembly language notation, not binary.

```
lui  $t0, 0x1000
lw   $t0, 0x1c($t0)
```

**It would also be ok to use `$at` for the address register.**

**Question 4.** (22 points) MIPS Hacking. A fast way to search an unordered list is a *sentinel search*, where we place the item we are looking for at the end of the list after all the data currently in it, then search until we find it and report whether it was found at the sentinel location or earlier. For this problem, write a MIPS assembly language version of the following C sentinel search function.

```
/* Search for x in A[0..n-1].  Return 1 if found, else 0.  */
/* Precondition: A has capacity for n+1 or more int values */
int find(int x, int A[], int n) {
   A[n] = x;
   k = 0;
   while (A[k] != x) {
      k++
   }
   if (k < n)
      return 1;
   else
      return 0;
}
```

You should use the standard MIPS calling and register conventions (initially, `$a0` = value of x, `$a1` = address of array A, `$a2` = value of n). You do not need to allocate a stack frame if you do not need one.

You may use either explicit subscripting operations or pointers or both to reference array elements, whichever is more convenient.

Include brief comments to make it easier to follow your code.

Write

your

answer

on

the

next

page.

(You can tear this page out for reference while you work, if that is helpful.)

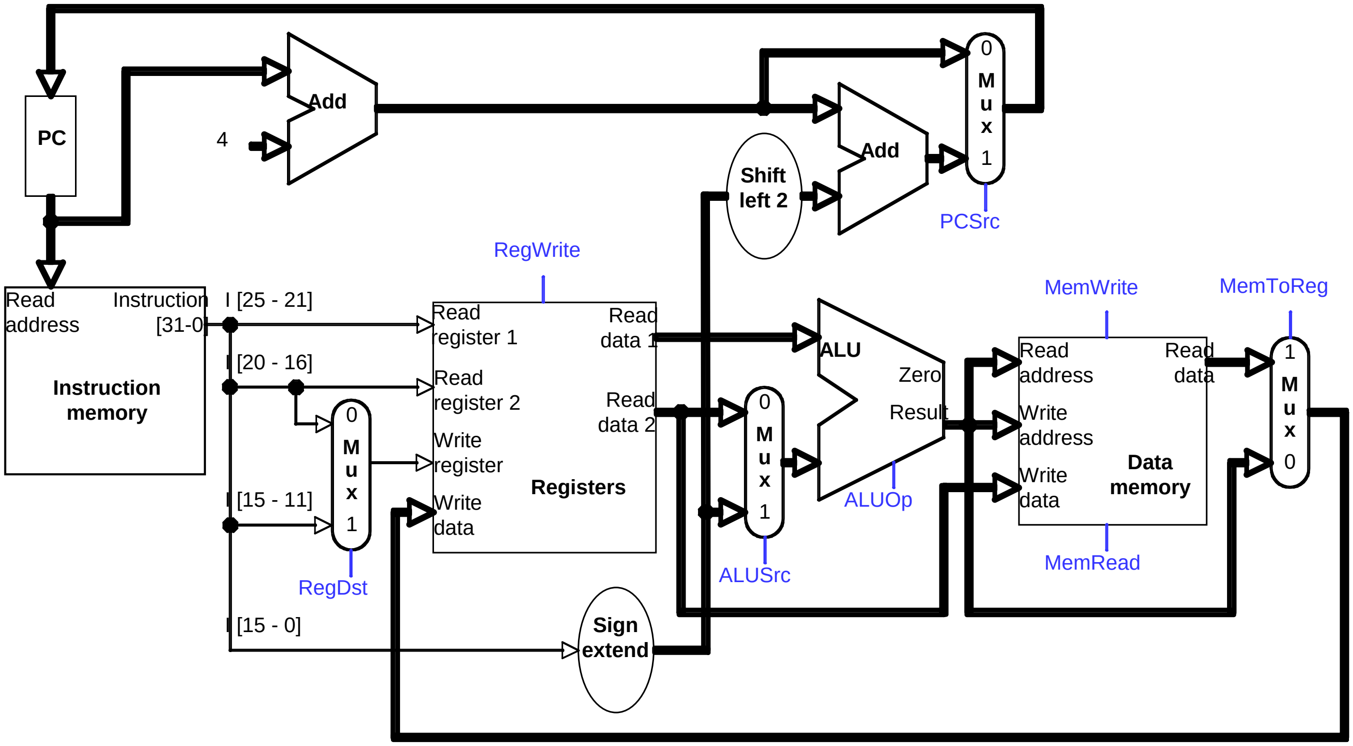
**Question 4.** (cont.) Write your answer here. C code repeated for reference (reformatted to save space).

```
/* Search for x in A[0..n-1].  Return 1 if found, else 0.  */
/* Precondition: A has capacity for n+1 or more int values */
int find(int x, int A[], int n) {
   A[n] = x;
   k = 0;
   while (A[k] != x) { k++ }
   if (k < n) return 1; else return 0;
}
```

```
find:
     sll  $t1, $a2, 2          # $t1 = &A[n]
     add  $t1, $t1, $a1
     sw   $a0, 0($t1)          # A[n] = x
loop:
     lw   $t0, 0($a1)          # $t0 = A[k]  (k=0 initially)
     beq  $a0, $t0, found      # branch if A[k] is x
     addi $a1, 4               # k++
     j    loop                 # repeat
found:
     slt  $v0, $a1, $t1        # result = &A[k] < &A[n]
     jr   $ra                  # return
```

**There are, of course, many other ways to solve the problem.**

The following question concerns the **single-cycle** MIPS datapath shown below.




**Question 5.** (20 points) For each of the following instructions, fill in the blanks in the table below to indicate the settings of the control signals to execute that instruction. Each control signal must be specified as 0, 1, or X (don't care). Writing a 1 or a 0 when an X is more accurate is *not* correct. If a control signal is a logical function of one or more other signals, write that function. For ALUOp, you can write things like "add" or "xor", since we don't expect you to have memorized the binary ALUOp signals.

| Opcode | RegDst | RegWrite | ALUSrc | ALUOp | MemWrite | MemRead | MemToReg | PCSrc |
|---|---|---|---|---|---|---|---|---|
| sub | 1 | 1 | 0 | sub | 0 | 0 | 0 | 0 |
| lw | 0 | 1 | 1 | add | 0 | 1 | 1 | 0 |
| bne | X | 0 | 0 | sub | 0 | 0 | X | !Zero |

**Question 6.** (20 points) Suppose we want to add a new R-format instruction to the MIPS ISA to compute the absolute value of a 32-bit integer. The new instruction is `abs rd,rs`, and it stores the absolute value of `rs` in `rd`.

A diagram of the simple 5-stage MIPS pipelined CPU we will modify is on the next page.

(a) To implement this instruction, we decide to add a new module to the datapath. Complete the following Verilog module `absComp` so that it generates the absolute value of `inVal` on the output `absVal`. (Don't be alarmed if the solution is quite short.)

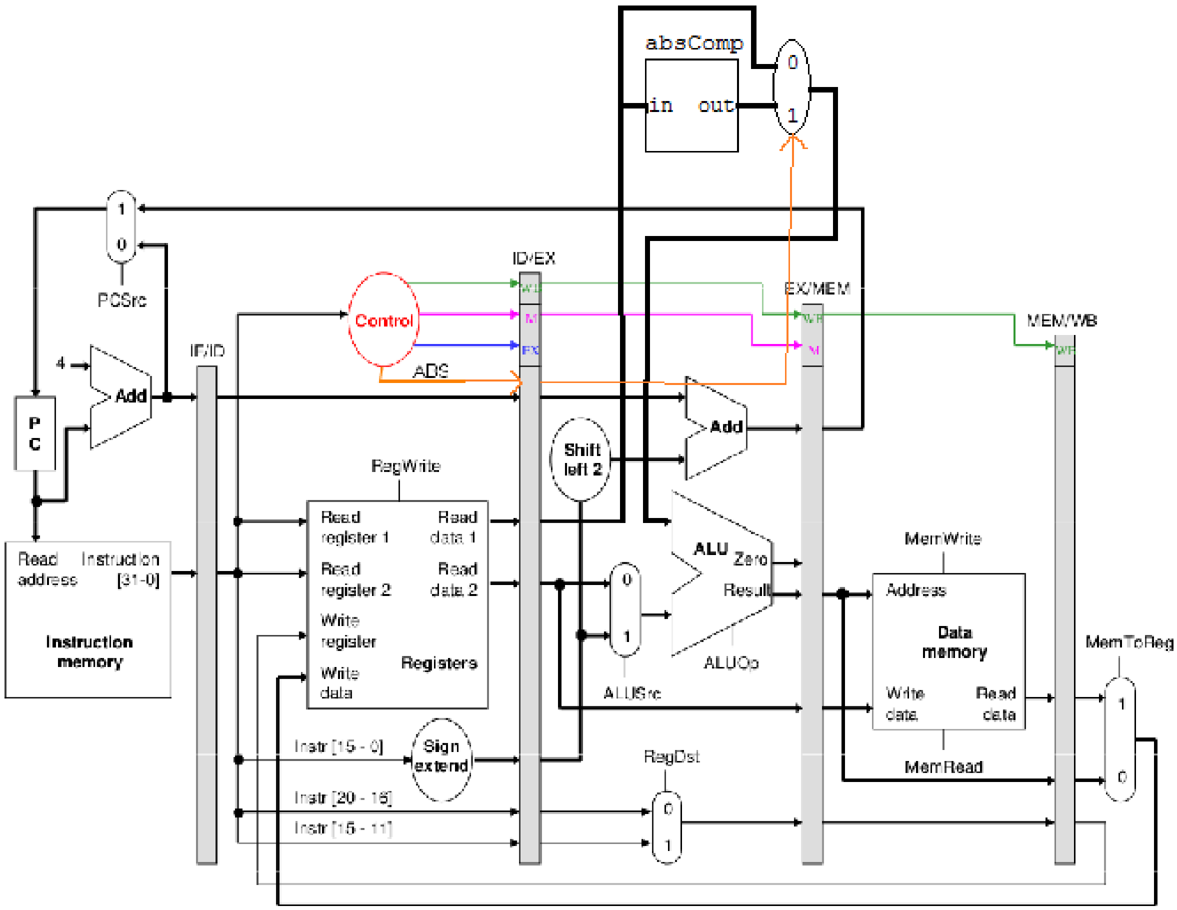
```
module absComp(inVal, absVal);
  input  wire [31:0] inVal;
  output wire [31:0] absVal;

    assign absVal = inVal[31] ? -inVal : inVal;

endmodule
```

(continued on next page)

**Question 6. (cont.)** (b) The following diagram is a simple version of the 5-stage MIPS pipelined CPU from class. The controller has a new output wire named ABS, which is high when the controller decodes an `abs` instruction and low otherwise. Add in and properly wire up the `absComp` module from part (a) into this datapath




<u>Note:</u> **This is only one of many possible solutions. All correct solutions involve adding the module, adding a multiplexer to choose between the module's output and the value of R[rs], and using a properly-forwarded ABS control line from the ID stage as the select for the added mux. It would also be ok to place the module between the ALU and the EX/MEM latch. In a real circuit, the best placement would need to take into account timing and delay issues, among other things.**

**Question 7.** (20 points) Pipeline hazards. Consider this sequence of MIPS instructions

```
addi    $t1, $zero, 17
lw      $t2, 0($a0)
add     $v0, $t1, $t2
lw      $t3, 4($a0)
add     $v0, $v0, $t3
```

(a) Describe all of the data dependencies in the above instructions. You can draw arrows on the instructions above to show your answer or you can write an explanation below.

(b) Assuming that we execute these instructions as written on a processor with a 5-stage pipeline with forwarding, fill in the diagram below to show how this sequence of instructions executes. Show stalls in the schedule, if any, by writing "stall" in that square. The first row for the `addi` instruction and the first two cycles of the next two instructions are written for you.

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

(c) Reorder the instructions into a new schedule (sequence) so they will execute without any stalls on a 5-stage pipeline with forwarding, if that is possible, or as few stalls as possible. Write the reordered instructions below.

**Any schedule that does not use the result of either load as an operand in the immediately following instruction will work. Here's one.**

```
lw      $t2, 0($a0)
lw      $t3, 4($a0)
addi    $t1, $zero, 17
add     $v0, $t1, $t2
add     $v0, $v0, $t3
```

How many stall cycles are there in your new schedule above, if any? ___**0**____