Question 1. (9 points). Suppose we have the following MIPS instructions stored in memory starting at location `0x2400`.

```
2400:          add    $v0,$t1,$t3
2404:   loop:  sll    $t3,$a2,12
2408:          or     $s2,$t3,$a1
240c:          bne    $s2,$t8,loop
2410:          addi   $s5,$zero,17
```

(a) (3 points) Re-write the `bne` instruction at location `240c` replacing the register names and branch label (`loop`) with the actual register numbers and an integer branch offset.

`bne    $18, $24, -3`

(b) (1 point) What is the format of the `bne` instruction (R, I, or J)?

I

(c) (5 points) Re-write the `bne` instruction in binary. Your answer should diagram the fields of the instruction and show the bits in each field. Label the fields of the instruction (opcode, etc.).

```
000101 10010 11000 1111111111111101
opcode  rs    rt         imm
```

Question 2. (12 points) For each of the following instructions, (i) indicate whether it is a real MIPS machine instruction or an assembler pseudo-instruction, and (ii) if it is a pseudo-instruction, write the corresponding machine instruction or instructions that would be generated by an assembler to perform the given operation. You can still use symbolic register names like `$a5`, `$t2`, etc., in your re-written instructions; you do not need to translate those to the actual register numbers. If the original instruction is a real machine instruction you do not need to write anything under (ii).

(a) `li $v0, 0xdeadbeef` (i) Instruction type: real (pseudo)

(ii) Rewritten machine instruction(s) if original code is a pseudo-instruction:

```
lui  $v0, 0xdead
ori  $v0, $v0, 0xbeef
```

Note: `addi` could not be used instead of `ori` here because when `0xbeef` is sign-extended to a 32-bit number it becomes `0xffffbeef` and the resulting sum would be `0xdeacbeef`. `addi` could be used if the operand of the `lui` instruction was `0xdeae` and the second instruction was `addi` with an operand of `0xbeef`.

(b) `bge $s5, $a0, somewhere` (i) Instruction type: real (pseudo)

(ii) Rewritten machine instruction(s) if original code is a pseudo-instruction:

```
slt  $at, $s5, $a0            # at=0 if $s5>=$a0
beq  $at, $zero, somewhere
```

Note: `$at` is the only register that can be used to hold the result of `slt`. Other registers cannot be used unless the assembler knows that the value in the chosen register is not needed later, and there is no way it can know that.

(c) `addi $sp, $s3, -17` (i) Instruction type: (real) pseudo

(ii) Rewritten machine instruction(s) if original code is a pseudo-instruction:

Question 3. (25 points) MIPS Hacking. If we have a sorted list of integers, one operation we might want to perform is to insert a new value into the proper place in the list. Here is a function to do that.

```
/* Insert val in the correct place in sorted int array A.  */
/* Preconditions: There are n elements currently stored in */
/* A[0]..A[n-1], and n >= 0.  The array is sorted in non-  */
/* decreasing order, A[0] <= A[1] <= ... <= A[n-1].        */
/* The array size is at least n+1, so there is room for a  */
/* new value.                                              */
void insert(int A[], int n, int val) {
   int k = n;
   while (k > 0 && A[k-1] > val) {
      A[k] = A[k-1];
      k--;
   }
   A[k] = val;
}
```

Translate this function into MIPS assembly language. You should use the standard MIPS calling and register conventions. You do not need to allocate a stack frame if you do not need one. You must implement this algorithm as given, but you may use either explicit subscripting operations or pointers or both to reference array elements, whichever is more convenient. Feel free to use assembler pseudo-instructions in your solution.

Hint: Even though there are four references to array elements in the code, you probably don't need to include lots of duplicate code to recalculate each of the array element locations from scratch, one at a time.

Reminder: Remember that when evaluating `k>0&&A[k-1]>val`, the second part of the condition is not evaluated if the first part (`k>0`) is false.

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 3. (cont.) Write your MIPS version here. C code repeated for reference.

```
   void insert(int A[], int n, int val) {
      int k = n;
      while (k > 0 && A[k-1] > val) {
         A[k] = A[k-1];
         k--;
      }
      A[k] = val;
   }
```

```
insert:                     # register assignments:
                            # $a0 = &A[0]  (input arguments)
                            # $a1 = n
                            # $a2 = val
                            # $t0 = k      (temporaries)
                            # $t1 = &A[k]
                            # $t2 = A[k-1]

      move  $t0,$a1         # k = n
      sll   $t1,$t0,2       # $t1 = k*4
      add   $t1,$t1,$a0     # $t1 = &A[k]
loop:
      ble   $t0,$zero,done  # exit if k<=0
      lw    $t2,-4($t1)     # load A[k-1]
      ble   $t2,$a2,done    # exit if A[k-1]<=val
      sw    $t2,0($t1)      # A[k] = A[k-1]
      addi  $t0,$t0,-1      # decrement k
      addi  $t1,$t1,-4      # adjust &A[k] to match
      j     loop            # repeat
done:
      sw    $a2,0($t1)      # A[k] = val
      jr    $ra             # return
```

Notes: This is a leaf function that does not call other functions and there are enough temporary registers available for the code, so there is no need to allocate a stack frame or save/restore registers.

Obviously there are many ways to write the code. This solution is a straight-forward translation of the original C code.

Question 4. (22 points) Suppose we want to add support for a new set of branch instructions to our **single-cycle** MIPS data path. These instructions branch depending on the value of a word in memory and a register, rather than a pair of registers. The new instructions are MBEQ, MBNE, MBLEZ, and MBGTZ and they work like their register counterparts. For example:

"`MBEQ $rs, $rt, some_label`" will branch to `some_label` if M[R[$rs]] = $rt.
"`MBLEZ $rs, $zero, some_label`" will branch to `some_label` if M[R[$rs]] <= 0

(a) Below is skeleton code from the PCAddressComputer.v module from your Lab processor. Add support for the four new branches (MBEQ, MBNE, MBLEZ, and MBGTZ) to the Verilog. For simplicity, assume they have the same values of Inst[27:26] as regular branches.

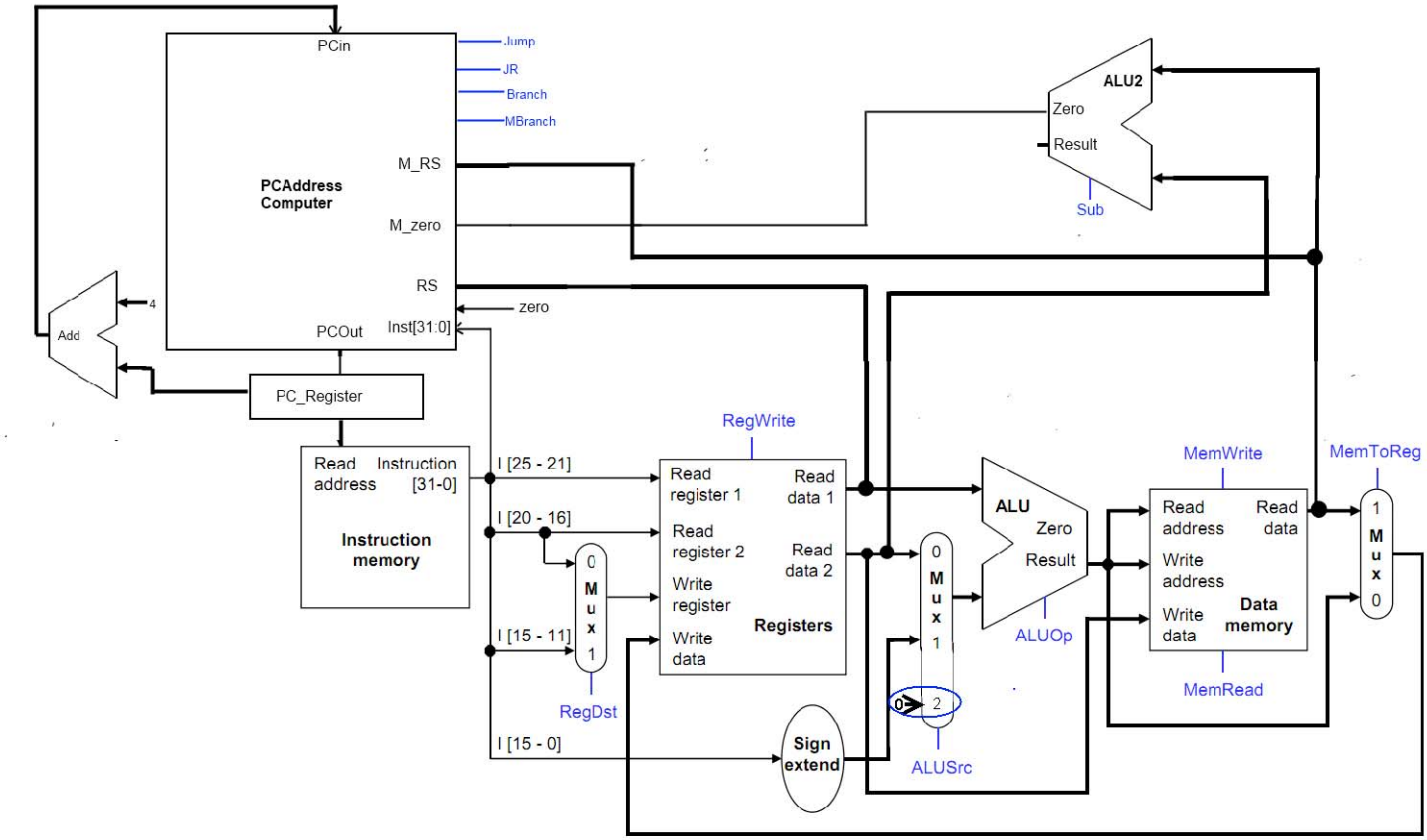
```
module PCAddressComputer(
    input [31:0] PCIn, // Address of Next Instruction (PC + 4)
    input [31:0] Inst, // Instruction
    input [31:0] RS,   // value of register specified by RS field of Inst
    input zero,        // set if ALUOut == 0
    input Jump,        // set if Inst indicates a jump operation
    input JR,          // set if Inst is JR or JALR
    input Branch,      // set if Inst is BEQ,BNE,BLEZ,BGTZ
    input MBranch,     // set if Inst is MBEQ, MBNE, MBLEZ, MBGTZ
    input [31:0] M_RS, // value read from memory for MBranch
    input M_zero,
    output reg [31:0] PCOut

    // Relevant opcodes
    parameter BEQ = 2'b00;      // Branch if Equal
    parameter BGTZ = 2'b11;     // Branch if Greater Than Zero
    parameter BLEZ = 2'b10;     // Branch if Less than or Equal to Zero
    parameter BNE = 2'b01;      // Branch if Not Equal

    wire [31:0] BranchAddress = {{14{Inst[15]}}, Inst[15:0], 2'b0}; // target for a branch
    wire [31:0] JumpAddress = {PCIn[31:28], Inst[25:0], 2'b0};      // target for a jump

    always @ (*) begin
        PCOut = PCIn;         // non-jump, non-branch default
        if (Jump)
                PCOut = (JR) ? RS : JumpAddress;
        else if (Branch) begin
                case (Inst[27:26])
                    BEQ : PCOut = (zero) ? PCIn + BranchAddress : PCIn;
                    BGTZ: PCOut = (!RS[31] && !zero) ? PCIn + BranchAddress: PCIn;
                    BLEZ: PCOut = (RS[31] || zero) ? PCIn + BranchAddress : PCIn;
                    BNE : PCOut = (!zero) ? PCIn + BranchAddress : PCIn;
                    default: PCOut = 32'bX;
                endcase
        end
        else if (MBranch) begin /* Your code goes below here */
                case (Inst[27:26])
                    BEQ : PCOut = (M_zero) ? PCIn + BranchAddress : PCIn;
                    BGTZ: PCOut = (!M_RS[31] && !M_zero) ? PCIn + BranchAddress: PCIn;
                    BLEZ: PCOut = (M_RS[31] || M_zero) ? PCIn + BranchAddress : PCIn;
                    BNE : PCOut = (!M_zero) ? PCIn + BranchAddress : PCIn;
                    default: PCOut = 32'bX;
                endcase
        end
    end
endmodule
```

Question 4 (cont.) (b) Now add the modified PCAddressComputer to the single cycle datapath. A modified version of this datapath is shown below, including the new PCAddressComputer. Wire-up the two new inputs to the PCAddressComputer to support the four Memory Branch instructions. You are free to add additional functional units (e.g. ALUs, muxes, etc.) to the datapath to implement the new instructions.



Question 5. (18 points) Pipeline hazards. Consider this sequence of MIPS instructions.

a) `lw $t1, 40($t2)`

b) `add $t2, $t3, $a0`

c) `add $t1, $t1, $t2`

d) `sw $t1, 20($t2)`

(a) (6 points) Identify all of the data dependencies in the above instructions, whether or not they cause any hazards or stalls. You can either draw arrows in the instructions or describe the dependencies below. The instructions are labeled (a)-(d), which might be useful in identifying them in your answer.

(a) generates value in `$t1` used in (c)
(b) generates value in `$t2` used in (c) and (d)
(c) generates new value in `$t1` used in (d)

(b) (4 points) Suppose we execute these instructions on a processor with a 5-stage pipeline with forwarding as described in class. Are there any hazards in the above sequence of instructions that will require the pipeline to stall because they can't be handled by the forwarding mechanisms? If so, where are they and why is the stall needed?

No stalls needed. If forwarding is available, the only possible need for a stall is if the result of the `lw` instruction is needed before it is available in the pipeline datapath. However, when these instructions are executed, the `lw` result is available at the end of cycle 4 (the `lw mem` stage). The `add` instruction (c) needs that value at the beginning of cycle 5 so it is available and no stall is required.

(continued next page)

Question 5. Pipeline hazards (cont). Instruction sequence repeated for reference.

```
a)    lw   $t1, 40($t2)
b)    add  $t2, $t3, $a0
c)    add  $t1, $t1, $t2
d)    sw   $t1, 20($t2)
```

(c) (8 points) Now let's assume we're executing these instructions on a new, prototype processor, also with our usual 5-stage pipeline. Unfortunately in this new processor the hazard circuitry is completely broken and it does not detect hazards or properly forward results or insert stalls when necessary. We need to modify the code and insert `nop` instructions to delay the execution of later instructions when necessary. Re-write the above code and insert the minimum number of `nop` instructions needed to avoid hazards that would otherwise require forwarding or stalls to produce the correct results. You may not reorder the original instructions – just insert `nop`s where they are needed. (To insert a `nop`, just write `nop` on a line by itself. The assembler will know how to translate that into a machine instruction that does nothing except delay for a cycle.)

If no forwarding or stalls are available, we must insert nops to ensure that values are written on or before the decode cycle of any instruction that reads them. In this case we need to inert nops before instructions (c) and (d) to wait 2 cycles each for the previous instructions to write their results. With these delays, `lw` instruction (a) has already written its result in `$t1` before it is needed by `add` instruction (c).

```
a)    lw   $t1, 40($t2)
b)    add  $t2, $t3, $a0
      nop
      nop
c)    add  $t1, $t1, $t2
      nop
      nop
d)    sw   $t1, 20($t2)
```

Quesiton 6. (14 points) Pipeline performance. Suppose we have the following functional units with the given latencies in a processor:

IF	2 ns
ID	2 ns
EX	3 ns
MEM	6 ns
WB	2 ns

(a) If we use these units to build a **single-cycle** implementation, how long does it take to execute a single instruction?

15 ns

(b) If we use these units to build our usual 5-stage pipeline processor, what is the shortest possible cycle time?

6 ns (time needed by longest stage)

(c) How long does it take to execute N instructions using this pipeline, where N is some arbitrary large number?

6*(4+N) ns (4 cycles to fill the pipeline, then 1 cycle for each result)

(d) What is the speedup of this pipelined processor over the single-cycle implementation?

Ignoring the fill delay, 15/6 = 2.5x

(e) How does the speedup calculated in part (d) compare with the maximum speedup we could get from a 5-stage pipeline implementation? If it is not as good as it might be, explain what the problem is and suggest what might be done to improve it.

The maximum speed up of a 5-stage pipeline is 5x, so we are only doing half as well as we might.

The trouble is that the MEM stage takes twice as long as any other stage, which limits the speed at which the pipeline stages can execute. One reasonable fix would be to split the MEM stage into two cycles, MEM1 and MEM2, taking 3ns each. Then we could have a 6-stage pipeline with a 3ns cycle time. That could execute N instructions in 3*(5+N) cycles, for a speedup of 15/3 = 5x over the single-cycle implementation.