Q1

As CSE-BUBBLE has a total of 32 registers. Following are the registers and their usage protocol:

- Registers $r_0 - r_{31}$: These are general-purpose registers and can be used for storing intermediate results or values that are required by the instructions during execution. Among these, the following have special roles:
  - r_0: This register always holds the value 0. Writing to this register has no effect.
  - $r_{29} - r_{31}$: These registers are reserved for use by the operating system.
- PC: This is the program counter register and it holds the address of the next instruction to be executed.
- HI and LO: These are special-purpose registers used for storing the results of certain arithmetic and multiplication instructions. Specifically, HI is used to store the high-order 32 bits of a 64-bit result, while LO is used to store the low-order 32 bits of a 64-bit result.
- r_a: This is the return address register and is used to store the address of the instruction that follows a jal (jump and link) instruction.

The usage protocol for the registers is as follows:

- The general-purpose registers can be used by any instruction for any purpose.
- PC, HI, LO, and ra registers are used only by certain instructions as specified in Table 1.
- r_0 always holds the value 0 and cannot be written to.
- Registers $r_{29} - r_{31}$ are reserved for use by the operating system and should not be used by application programs.

Q2

The size of the instruction and data memory in VEDA should be determined based on the maximum number of instructions and data that need to be stored in the memory.

Assuming that we have 32-bit instructions and 32-bit data, we can calculate the size of the memory as follows:

- Instruction memory size: If we have n instructions, the instruction memory size will be n x 4 bytes (since each instruction is 4 bytes long).
- Data memory size: If we have m data values to store, the data memory size will be m x 4 bytes (since each data value is 4 bytes long).
  The actual values of n and m will depend on the specific requirements of the application and the program being executed on the processor. Therefore, these values need to be decided before determining the size of the memory.

This is the data memory used by the CSE-BUBBLE.

```
module veda #(
  parameter SIZE = 32,
  parameter ADDRESS_WIDTH = 5
  ) (
  input clk,
  input rst,
  input [5:0] opcode,
  input [ADDRESS_WIDTH:0] pc,
  input [ADDRESS_WIDTH:0] addr,
```

```
  input [31:0] datain,
  input mode,
  output [31:0] dataout
);

  integer i;
  reg [31:0] cells [SIZE:0]; // first signifies data bits, second is number of
such cells
  assign dataout = cells[addr];

  initial begin
    for (i = 0; i < (SIZE); i = i + 1)
      cells[i] <= 0;
  end

  always @(posedge clk) begin
    if (rst == 1) begin : reset
      for (i = 0; i < (SIZE); i = i + 1)
        cells[i] <= 0;
    end else if (opcode == 6'b010110) begin : scribble
      cells[addr] = datain;
    end
  end

endmodule
```

Q3

The instruction set for CSE-BUBBLE includes three types of instructions: R-type, I-type, and J-type instructions. The encoding methodology for each type of instruction is as follows:

R-Type Instructions:

R-type instructions use three register operands and an operation code. The layout of the R-type instruction is as follows:

6-bit Opcode	5-bit rd	5-bit rs	5-bit rt	5-bit shift amount	6-bit funct
Opcode	rd	rs	rt	Shift Amount	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

The fields in the R-type instruction layout are defined as follows:

- Opcode: a 6-bit operation code that specifies the type of R-type instruction.
- rd: a 5-bit field that specifies the source register.
- rs: a 5-bit field that specifies the second source register.
- rt: a 5-bit field that specifies the destination register.
- Shift amount: a 5-bit field that specifies the number of bits to shift the value in the rt register (only used in shift instructions).
- Funct: a 6-bit function code that specifies the specific operation to be performed.
  The encoding methodology for R-type instructions is as follows:

I-Type Instructions:

I-type instructions use two register operands and an immediate value. The layout of the I-type instruction is as follows:

6-bit Opcode	5-bit rd	5-bit rs	16-bit immediate value
Opcode	rd	rs	Immediate Value
6 bits	5 bits	5 bits	16 bits

The fields in the I-type instruction layout are defined as follows:

- Opcode: a 6-bit operation code that specifies the type of I-type instruction.
- rs: a 5-bit field that specifies the source register.
- rt: a 5-bit field that specifies the destination register.
- Immediate value: a 16-bit field that specifies the immediate value.
  The encoding methodology for I-type instructions is as follows:

6-bit Opcode	5-bit rs	5-bit rt	16-bit immediate value
Opcode	rs	rt	Immediate Value

J-Type Instructions:

J-type instructions use a 26-bit jump target address. The layout of the J-type instruction is as follows:

6-bit Opcode	26-bit jump target address
Opcode	Jump Target Address
6 bits	26 bits

The fields in the J-type instruction layout are defined as follows:

J-type instruction encoding:

- opcode (6 bits): specifies the operation type, for example, 000010 for jump
- jump address (26 bits): specifies the address to jump to

Q4

To implement the instruction fetch phase, we need to perform the following steps:

Get the value of the program counter (PC) from the PC register.
Access the instruction memory using the PC value and retrieve the instruction at that address.
Increment the PC by the size of the instruction (32 bits).
Store the retrieved instruction in the instruction register.
Here's how we can implement the instruction fetch phase in Verilog:

```
module instruction_fetch #(
  parameter SIZE = 32, // size of the instruction memory
  parameter INSTR_SIZE = 32, // size of each instruction
  parameter ADDRESS_WIDTH = 5 // width of the address bus
```

```
)(
  input clk,
  input rst,
  input [ADDRESS_WIDTH:0] pc,
  output [31:0] instruction
);

  // VEDA memory
  // fetches instruction at addr pc and stores to instruction
  reg [31:0] cells [SIZE:0]; // first signifies data bits, second is number of
such cells
  reg [31:0] dataout_reg;
  assign instruction = dataout_reg;

  // to store data in memory
  integer i;
  initial begin
    $readmemb("bubble.bin", cells);
  end

  always @(posedge clk) begin
    if (rst == 1) begin : reset
      for (i = 0; i < (SIZE); i = i + 1)
        cells[i] <= 0;
    end
    else begin : normal
      dataout_reg <= cells[pc];
    end
  end
endmodule
```

This section has inbuilt instruction memory. The size of the instruction memory is taken as a parameter, the address width needs to be set accordingly. ALl the instructions are stored initially from the file named `bubble.bin`. Next on the positive edge of clock, the intruction pointed by the pc is fetched and passed on. If reset is set high, all the instructions stored is cleared.

Q5

Here is a possible implementation for the instruction decode module:

```
module inst_decode (
  input [31:0] instruction,
  output [5:0] opcode,
  output [4:0] rs,
  output [4:0] rt,
  output [4:0] rd,
  output [4:0] shift,
  output [5:0] funct,
  output [15:0] imm,
  output [25:0] jump
);
  // General
  assign opcode = instruction[31:26];
  assign rd = instruction[25:21];
  assign rs = instruction[20:16];
```

```
  // R type
  assign rt = instruction[15:11];
  assign shift = instruction[10:6];
  assign funct = instruction[5:0];

  // I Type
  assign imm = instruction[15:0];

  // J type
  assign jump = instruction[25:0];

endmodule
```

The module takes in a 32-bit instruction and outputs all the possible fields which includes `opcode`, `rd`, `rs`, `rt`, `shift`, `funct`, `imm` & `jump`. The next procedures is then decided by the control unit.

Q6

This is an implementation of the ALU which is responsible for all the arithematic and logical operations:

```
module ALU(
  input clk,
  input [5:0] opcode,
  input [31:0] a,
  input [31:0] b,
  input [15:0] imm,
  input [5:0] funct,

  output reg [31:0] ot
);

  always @(posedge clk) begin
    case (opcode)
      6'b000000: begin
        case (funct)
          // arithmetic
          6'b000000: ot = a + b; // signed addition

          6'b000001: ot = a - b; // signed subtraction

          6'b000010: ot = a+b; // unsigned addition
          6'b000011: ot = a-b; // unsigned subtraction

          // logical
          6'b000100: ot = a & b; // bitwise AND
          6'b000101: ot = a | b; // bitwise OR
          6'b000110: ot = a << b[4:0]; // left shift
          6'b000111: ot = a >> b[4:0]; // right shift

          // comparison
          6'b001000: ot = a < b; // less than
          default: ot = a + b; // unsigned addition by default
        endcase
      end
```

```
      // immediate values
      6'b000001: ot = a + imm; // signed immediate addition
      6'b000010: ot = a - imm; // signed immeidate subtraction
      6'b000011: ot = a & {16'b0, imm}; // immeidate and
      6'b000100: ot = a | {16'b0, imm}; // bitwise OR
      6'b000101: ot = a << imm[4:0]; // left shift
      6'b000110: ot = a >> imm[4:0]; // right shift
    endcase
  end
endmodule
```

This module takes in the `opcode`, `a`, `b`, `imm` & `funct` and fills in the appropriate value in `ot` based on the `opcode` and `funct` values.

Q7

This is an implementation of the branching operations:

```
module branch #(
  parameter ADDRESS_WIDTH = 5
) (
  input clk,
  input [5:0] opcode,
  input [15:0] imm,
  input [31:0] rd,
  input [31:0] rs,
  input [ADDRESS_WIDTH:0] pc,

  output reg [ADDRESS_WIDTH:0] next_pc
);

  always @(*) begin
    case (opcode)
      // conditional branch
      6'b001000: next_pc = ($signed(rd) == $signed(rs)) ? imm[ADDRESS_WIDTH:0]
: pc; // beq
      6'b001001: begin
        next_pc = ($signed(rd) != $signed(rs)) ? imm[ADDRESS_WIDTH:0] : pc; //
bne
      end
      6'b001010: next_pc = ($signed(rd) > $signed(rs)) ? imm[ADDRESS_WIDTH:0] :
pc; // bgt
      6'b001011: next_pc = ($signed(rd) >= $signed(rs)) ? imm[ADDRESS_WIDTH:0]
: pc; // bge
      6'b001100: next_pc = ($signed(rd) < $signed(rs)) ? imm[ADDRESS_WIDTH:0] :
pc; // blt
      6'b001101: next_pc = ($signed(rd) <= $signed(rs)) ? imm[ADDRESS_WIDTH:0]
: pc; // ble
    endcase
  end
endmodule
```

This module takes in the two values `rd` and `rs` and compares the values based on the opcode and outputs the `next_pc` which is then sent to the `cu` (control unit) which handles this appropriately.

Q8

This the control unit and the topmost unit of the processor:

```
`include "veda.sv"
`include "inst_fetch.sv"
`include "inst_decode.sv"
`include "alu.sv"
`include "branch.sv"

module CU #(
  parameter ADDRESS_WIDTH = 5,
  parameter INSTRUCTION_SIZE = 32,
  parameter DATA_SIZE = 64
  ) (
  input clk,
  input verbose,
  input rst
);
  wire [31:0] instruction; // instruction fetched from instruction memory
  reg [ADDRESS_WIDTH:0] pc; // always points the address of the stored
instruction, would always be accesed in instruction memory
  wire [ADDRESS_WIDTH:0] next_pc;

  reg [31:0] inplace_memory [31:0] ; // 32-bit 32 registers
  reg [31:0] ra_reg; // return address register

  // instruction decode registers
  wire [5:0] opcode;
  wire [4:0] rs;
  wire [4:0] rt;
  wire [4:0] rd;
  wire [4:0] shift;
  wire [5:0] funct;
  wire [15:0] imm;
  wire [25:0] jmp;

  // storage memory registers
  reg write_enable = 1'b0, mode = 1'b0;
  reg [ADDRESS_WIDTH:0] addr;
  wire [31:0] dataout;
  reg [31:0] datain;

  // ALU registers
  wire [31:0] alu_out;

  instruction_fetch insr(
    .clk(clk),
    .rst(rst),
    .pc(pc),
    .instruction(instruction)
  );

  veda memory(
    .clk(clk),
    .rst(rst),
```

```
    .opcode(opcode),
    .pc(pc),
    .addr(addr),
    .datain(datain),
    .mode(mode),
    .dataout(dataout)
  );

  // decodes instruction and stores to opcode, rs, rt, rd, shift, funct, imm,
jmp
  inst_decode decode(
    .instruction(instruction),
    .opcode(opcode),
    .rs(rs),
    .rt(rt),
    .rd(rd),
    .shift(shift),
    .funct(funct),
    .imm(imm),
    .jump(jmp)
  );

  // performs ALU operation and stores to result
  ALU alu(
    .clk(clk),
    .opcode(opcode),
    .a(inplace_memory[rs]),
    .b(inplace_memory[rt]),
    .imm(imm),
    .funct(funct),
    .ot(alu_out)
  );

  branch branch(
    .clk(clk),
    .opcode(opcode),
    .imm(imm),
    .rd(inplace_memory[rd]),
    .rs(inplace_memory[rs]),
    .pc(pc),

    .next_pc(next_pc)
  );

  integer i;
  initial begin
    for (i=0 ;i< 32 ;i=i+1)
      inplace_memory[i] <= 0;
    pc = 0;
    ra_reg = 0;
  end


  always @(posedge clk) begin
    if (rst)
      pc <= 0;
    else begin
      if (opcode < 6'b001000) begin : aluOps
```

```
        // storing back to register
        inplace_memory[rd] = alu_out;
        if (verbose)
          $display("[%d]. [ALU]: rd = %d, rs = %d, rt = %d, imm = %d", pc+1,
inplace_memory[rd], inplace_memory[rs], inplace_memory[rt], imm );
        pc = pc + 1;
      end
      else if (opcode >= 6'b001000 && opcode < 6'b010000) begin :
conditionalBranch
        if (verbose)
          $display("[%d]", pc+1);
        pc = next_pc + 1;
        if (verbose)
          $display("[BRANCH]: pc = %d, next_pc = %d", pc, next_pc);
      end
      // Unconditional branch
      else begin
        case (opcode)
          6'b010000: pc = jmp[ADDRESS_WIDTH:0]; // jump to register I type
          6'b010001: pc = inplace_memory[jmp[4:0]];
          6'b010010: begin : JAL
          pc = jmp[ADDRESS_WIDTH:0]; // return from subroutine I type
          ra_reg = pc + 1; // return address in $ra
          end

          6'b010011: begin
            inplace_memory[rd] = inplace_memory[rs] < inplace_memory[rt] ? 1 :
0;  // slt (set less than)
            pc = pc + 1;
          end

          6'b010100 : begin
            inplace_memory[rd] = inplace_memory[rs] < imm ? 1 : 0;  // slti
(set less than)
            pc = pc + 1;
          end

          6'b010101: begin // load word
            write_enable = 1'b0;
            mode = 1'b0;
            addr = inplace_memory[rs] + imm[ADDRESS_WIDTH:0];
            inplace_memory[rd] = dataout;
            if (verbose)
              $display("[%d]. [LOAD]: dataout = %d, source= %d, dest = %d",
pc+1, dataout, addr, rd);
            pc = pc + 1;
          end

          6'b010110: begin // store word
            write_enable = 1'b1;
            mode = 1'b0;
            datain = inplace_memory[rd];
            addr = inplace_memory[rs] + imm[ADDRESS_WIDTH:0];
            pc = pc + 1;
          end
        endcase
      end
    end
```

```
  end
endmodule
```

This is the top level module and integrates each of the individual submodules. The data pipeline is as follows:

1. instruction is read by the instruction_fetch module and is passed in the `instruction` wire.
2. `instruction` is then decoded by the instruction_decode modules which essentially slices the bits.
3. `opcode` is then used to differentiate between the type of instruction to be followed. It then switches to `alu`, `branch` or `memory` as per the requirement.

This also has the 32 registers as described initially. It also has a verbose input, which when enabled , prints all the operations done by the processor. Otherwise only the data memory upto 10 places are printed on each data write.

Q9

This is the MIPS code which is then converted to machine code:

```
.data

arr: .word 310, 230, 30, 50, 50, 60, 70, 80, 90, 100;
n: .word 10;
new_line: .asciiz "\n"

.text
.globl main
main:
  addi $t0, $zero, 10;
  addi $t1, $zero, 10;
  addi $a1, $zero, address of arr = 0;
  outer_loop:
    blt $t1, $zero, exit;
    inner_loop:
      lw $t3, ($t6), 0[16BIT];
      addi $t4, $t2, 1;
      add $t6, $t4, $a1;
      lw $t5, ($t6);
      blt $t5, $t3, swap;
  after_condition:
    addi $t2, $t2, 1;
    j inner_loop;
  swap:
    add $t6, $t4, $a1;
    sw $t3, ($t6);
    add $t6, $t2, $a1;
    sw $t5, ($t6);
    j after_condition;
  exit:
    syscall
```

Q10

```
// main:
000001 01000 00000 0000000000001010
000001 01001 00000 0000000000001010
000001 00101 00000 0000000000000000

// outer loop:
000010 01001 01001 0000000000000001
001100 01001 00000 0000000000011111
000000 01010 00000 00000 00000 000000

// inner loop:
001011 01010 01001 0000000000000011
000000 01110 01010 00101 00000 000000
010101 01011 01110 0000000000000000
000001 01100 01010 0000000000000001
000000 01110 01100 00101 00000 000000
010101 01101 01110 0000000000000000
001100 01101 01011 0000000000001111

// after condition:
000001 01010 01010 0000000000000001
010000 00000000000000000000000110

// swap:
000000 01110 01100 00101 00000 000000
010110 01011 01110 0000000000000000
000000 01110 01010 00101 00000 000000
010110 01101 01110 0000000000000000
010000 00000000000000000000001101
```

This cannot be directly fed to the processor and needs some preprocessing. After preprocessing the code becomes:

```
00000101000000000000000000001010
00000101001000000000000000001010
00000100101000000000000000000000
00001001001010010000000000000001
00110001001000000000000000011111
00000001010000000000000000000000
00101101010010010000000000000011
00000001110010100010100000000000
01010101011011100000000000000000
00000101100010100000000000000001
00000001110011000010100000000000
01010101101011100000000000000000
00110001101010110000000000001111
00000101010010100000000000000001
01000000000000000000000000000110
00000001110011000010100000000000
01011001011011100000000000000000
00000001110010100010100000000000
01011001101011100000000000000000
01000000000000000000000000001101
```

This is the machine code for bubble sorting 10 numbers stored from address 0-9 in the data memory.

Testbench

This is the testbench for the processor:

```
`include "cu.sv"

module cu_tb();

  reg clk, reset, verbose;

  // Instantiate the Unit Under Test (UUT)
  CU uut (
    .clk(clk),
    .rst(reset),
    .verbose(verbose)
  );


  initial begin
    clk = 0;
    reset = 0;
    verbose = 0
  end
  always #5 clk = ~clk;

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(0, cu_tb);
    # 10000
    $finish;
  end
endmodule
```