PDS 1

- Zero is hard-coded into register number 0
- One is hard-coded into register number 1
- Caller-saved registers $s0 - $s7 are allotted registers numbered 2-9.
- Callee-saved Temporary registers $t0 - $t7 are allotted registers numbered 10-18.
- Procedure value registers $v0-$v1 are allotted registers numbered 19-20.
- Argument registers $a0-$a3 are allotted registers numbered 21-24.
- OS registers $k0, $k1 are 25-26.
- Global pointer $gp is register number 28.
- Stack pointer $sp is register number 29.
- Frame pointer $fp is register number 30.
- Return address $ra is register number 31.

PDS 2

The instruction size is 1 word = 4 bytes = 32 bits. Each instruction is addressed using it's first byte. Thus, every instruction address is a multiple of 4. The data memory has 256 words capable of storing 256 32-bit values. The size is 256 x 32.

PDS 3

Note :- The opcodes and their labels of type of instruction are clearly mentioned in the "ins_fetch" file. We have followed the MIPS methodology of instructions. The opcode and funct together are used to identify the exact instruction to be performed by the ALU.

R type :-

We have followed the MIPS methodology.

Format : {6'b opcode, 5'b rs, 5'b rt, 5'b rd, 5'b shamt, 6'b funct}

Source registers :- rs, rt.

Destination register :- Rd

Shift amount :- shamt which is required in sll, srl instructions.

Function code :- funct, used to identify the instruction.

I type :-

Format : {6'b opcode, 5'b rs, 5'b rt, 16'b constant }

Base address register :- rs

Destination register :- rt

Constant :- offset, used to specify exact address.

Note :- To expand our range of addresses, we used WORD addressing rather than BYTE addressing for instructions like lw, sw, branch and jump.

For example, lw $t0, 4($t1) means :- Load the value of data_memory [$t1 + 4 * 4] into register $t0.

Branching uses a similar format.
For example, beq $t0, $t1, L means :- compare values in registers t0 and t1. If equal, branch to statement labelled L. We have used PC - relative addressing for branching, that is, PC = PC + (16'b const >> 2), reminiscent of MIPS. Note that the 16'b const is multiplied by 4 since we use WORD addressing rather than BYTE addressing

Important note :- For convenience, we have made a separate instruction for branching BACKWARDS. This is labelled bneb in ins_fetch. 4 * word_distance is subtracted from the current PC to reach the desired branch address.

J type :-

Format : {6'b opcode, 26'b address}

Opcode is used to identify jump instruction.
The PC is then given the value of (26'b address >> 2), again due to word addressing. Here we use ABSOLUTE addressing - if we want to jump to address 8000, 26'b address = 2000.

Jal is written similar to MIPS. On calling jal, $ra is given value of PC + 4. Then, we jump to address provided in instruction, which usually takes us to a procedure call. Once the procedure call is completed, PC gets the value of $ra, which is the next instruction in sequence.

IMP NOTE:
Since we have to implement FSM, we have written the code for the instruction fetch phase, decoding phase, ALU phase and memory write phase together in file ‘CSEBubble_final.v’.