In essence what we are trying to do is breakup `verilog` code and decidedly put it back together omitting certain parts of it depending on which instructions are used and which `verilog` code is needed to support these instructions.

First we create classify how each group-able block of `verilog` code is used by each instruction. To communicate that with ourselves and each other, we can simply add comments to each block, or even line if need be, so that we can much more easily identify them.

```verilog
`include "defines.v"

module alu (
  input [31:0]a // ADD, SUB, SLL, SLT, XOR, OR, SRA, AND, ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRLI, SRAI
  , input [31:0]b // // ADD, SUB, SLL, SLT, XOR, OR, SRA, AND, ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRLI, SRAI
  , input [4:0]shamt // SLL, SLT, SLTU
  , output reg [31:0]out // all, requirement of the module
  , output cf
  , output zf
  , output vf
  , output sf //
  , input [3:0]alufn // all except immediate
  );

  wire [31:0] add, op_b; // ADD, SUB

  assign op_b = (~b); // SUB

  assign {cf, add} = alufn[0] ? (a + op_b + 1'b1) : (a + b); // ADD, SUB

  assign zf = (add == 0);
  assign sf = add[31];
  assign vf = (a[31] ^ (op_b[31]) ^ add[31] ^ cf);

  wire[31:0] sh; // SLLI, SRLI, SRAI, SLL, SLT, SLTU
  shifter shifter0(.a(a), .shamt(shamt), .type(alufn[1:0]),  .r(sh)); // SLLI, SRLI, SRAI, SLL, SLT, SLTU

  always @ * begin
    out = 0;
    (* parallel_case *)
    case (alufn)
      // arithmetic
      `ALU_ADD : out = add;
      `ALU_SUB : out = add;
      `ALU_PASS : out = b;
      // logic
      `ALU_OR:  out = a | b;
      `ALU_AND:  out = a & b;
      `ALU_XOR:  out = a ^ b;
      // shift
      `ALU_SRL:  out=sh;
      `ALU_SRA:  out=sh;
      `ALU_SLL:  out=sh;
      // slt & sltu
      `ALU_SLT:  out = {31'b0,(sf != vf)};
      `ALU_SLTU:  out = {31'b0,(~cf)};
    endcase
  end
endmodule
```

Listing 1: an example of an annotated file