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.

```
`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
  , input [3:0]alufn // ADD, SUB, SLL, SLT, XOR, OR, SRA, AND, ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRLI, SRAI
  );

  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 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

Now we have an easy way of immediately identifying code to instruction relations. With this classification, we can build a JSON file that represent this.

```
{
  "module": "ALU"
  "args": [
    {
      "type": "input", "length": 32, "name": "a",
      "instructions": ["ADD", "SUB", "SLL", "SLT", "XOR", "OR", "SRA", "AND", "ADDI", "
 SLTI", "SLTIU", "XORI", "ORI", "ANDI", "SLLI", "SRLI", "SRLI", "SRAI"]
    },
    {
      "type": "input", "length": 32, "name": "b",
      "instructions": ["ADD", "SUB", "SLL", "SLT", "XOR", "OR", "SRA", "AND", "ADDI", "
 SLTI", "SLTIU", "XORI", "ORI", "ANDI", "SLLI", "SRLI", "SRLI", "SRAI"]
    },
    {
      "type": "input", "length": 5, "name": "shamt",
      "instructions": ["SLL", "SLT", "SLTU"]
    },
    {
      "type": "output reg", "length": 32, "name": "out",
      "instructions": ["all"]
    },
    {
      "type": "input", "length": 1, "name": "cf",
      "instruction": ["all"]
    },
    {
      "type": "output", "length": 4, "name": "alufn",
      "instructions": ["ADD", "SUB", "SLL", "SLT", "XOR", "OR", "SRA", "AND", "ADDI", "
 SLTI", "SLTIU", "XORI", "ORI", "ANDI", "SLLI", "SRLI", "SRLI", "SRAI"]
    }
  ]
  "declarations": [
    {
      "type": "wire", "length": 32, "name": "add"
      "instructions": ["ADD", "SUB"]
    },
    {
      "type": "wire", "length": 32, "name": "op_b",
      "instructions": ["SUB"]
    }
  ]
  "always": [
    "trigger": "*",
    "case": {
      "condition": "alufn"
      "assigns": [
        {
          "code": "`ALU_ADD : out = add;",
          "req": ["ADD"]
        },
        {
          "code": "`ALU_SUB : out = add;",
          "req": ["SUB"]
        }
        ...etc
      ]
    }
  ]
}

```

Listing 2: example JSON

Using this structure as well as a disjoint set, we can identify only the lines we need and properly output them.

Here comes the hard part. How to "jump through hoops" to completely remove certain components. To do that we can first create a graph to represent where each "logical path" leads. Then according to that, we can just assign wires to where they are meant to be without modules in the middle.

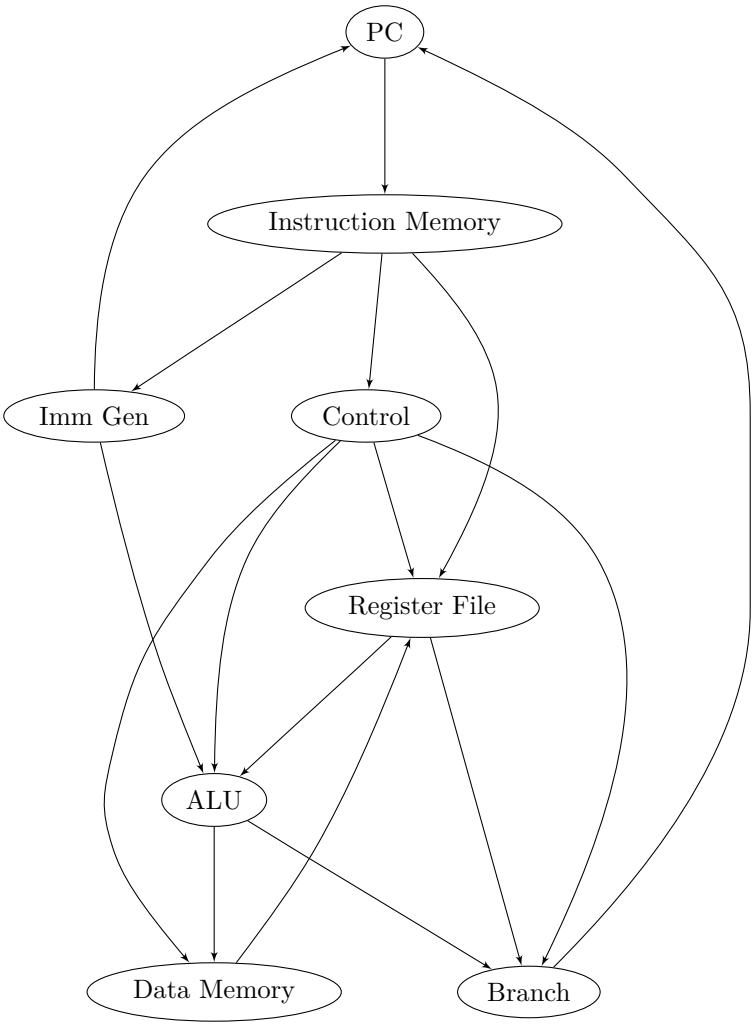

Figure 1:

A possible way to think about this is to make a sort of optimization passes. For instance, first we remove unnecessary modules. then we remove unused multiplexer.

A good example of this would be how we might not need the data memory module. Then we can realize through the JSON files that we do not need any of the module's internals internals. Then using the graph of the datapath, we can determine which wires need to jump to where so that we retain remaining functionality.