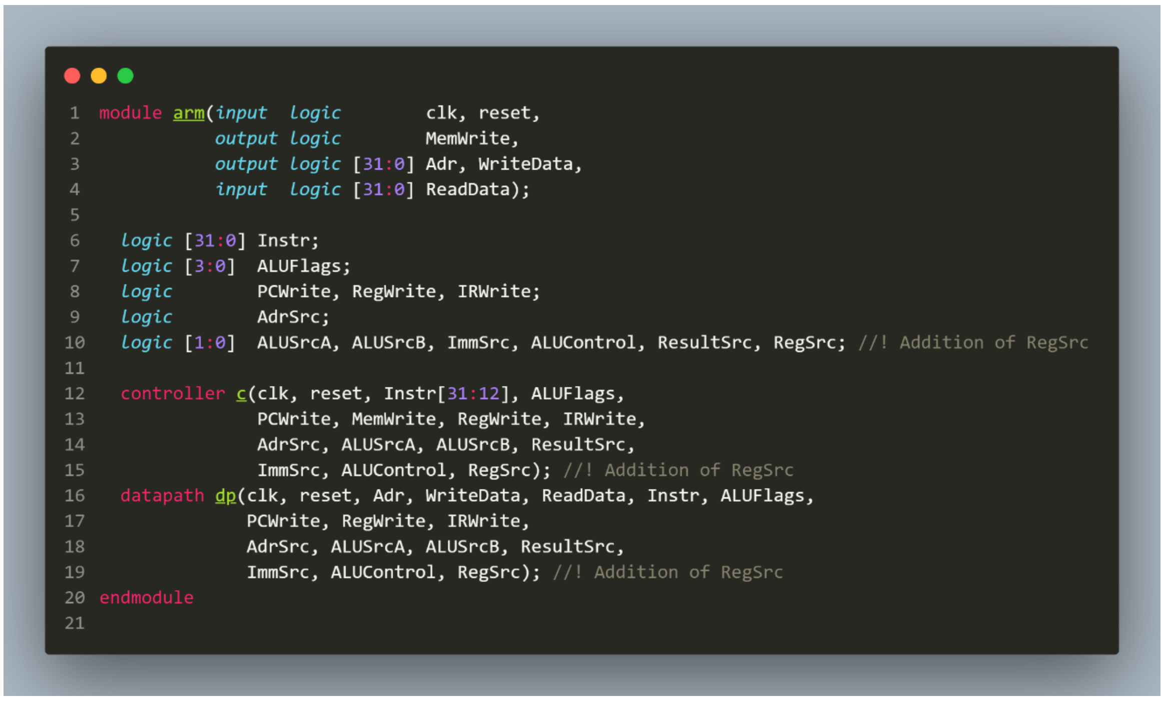
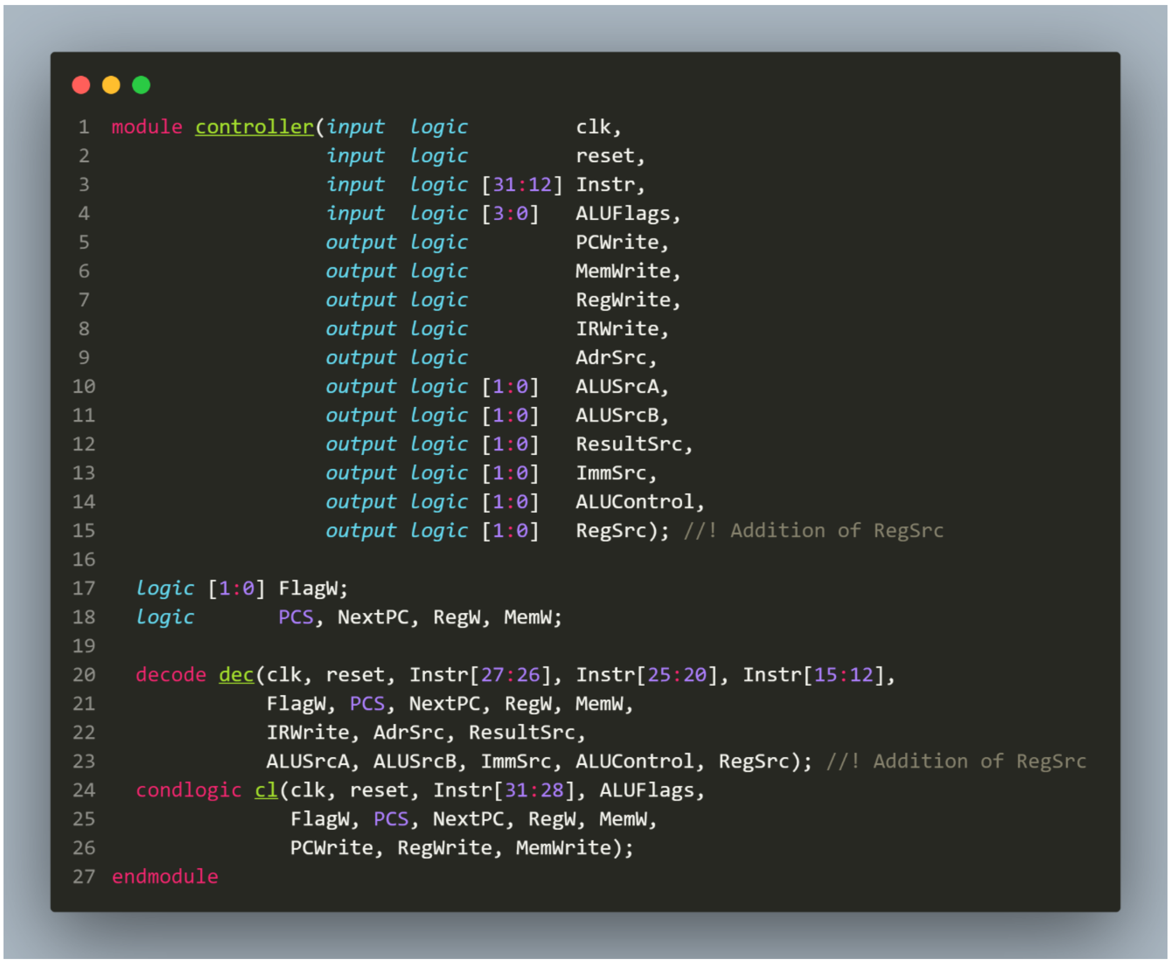


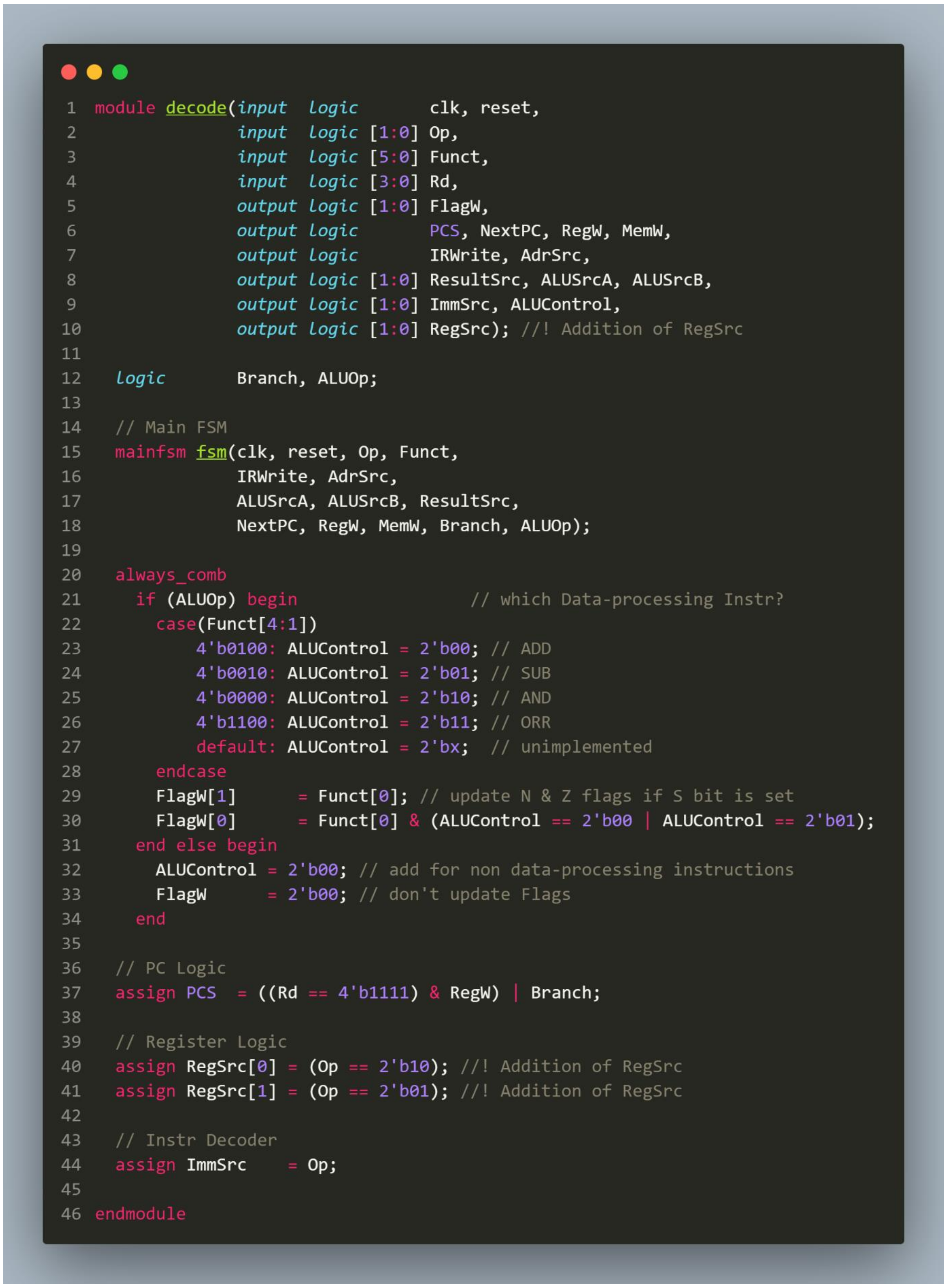
```
module arm(input  logic        clk, reset,
           output logic        MemWrite,
           output logic [31:0] Adr, WriteData,
           input  logic [31:0] ReadData);

  logic [31:0] Instr;
  logic [3:0]  ALUFlags;
  logic        PCWrite, RegWrite, IRWrite;
  logic        AdrSrc;
  logic [1:0]  ALUSrcA, ALUSrcB, ImmSrc, ALUControl, ResultSrc, RegSrc; //! Addition of RegSrc

  controller c(clk, reset, Instr[31:12], ALUFlags,
               PCWrite, MemWrite, RegWrite, IRWrite,
               AdrSrc, ALUSrcA, ALUSrcB, ResultSrc,
               ImmSrc, ALUControl, RegSrc); //! Addition of RegSrc
  datapath dp(clk, reset, Adr, WriteData, ReadData, Instr, ALUFlags,
              PCWrite, RegWrite, IRWrite,
              AdrSrc, ALUSrcA, ALUSrcB, ResultSrc,
              ImmSrc, ALUControl, RegSrc); //! Addition of RegSrc
endmodule
```

Addition of a new register RegSrc to control the two multiplexers for RA1 and RA2



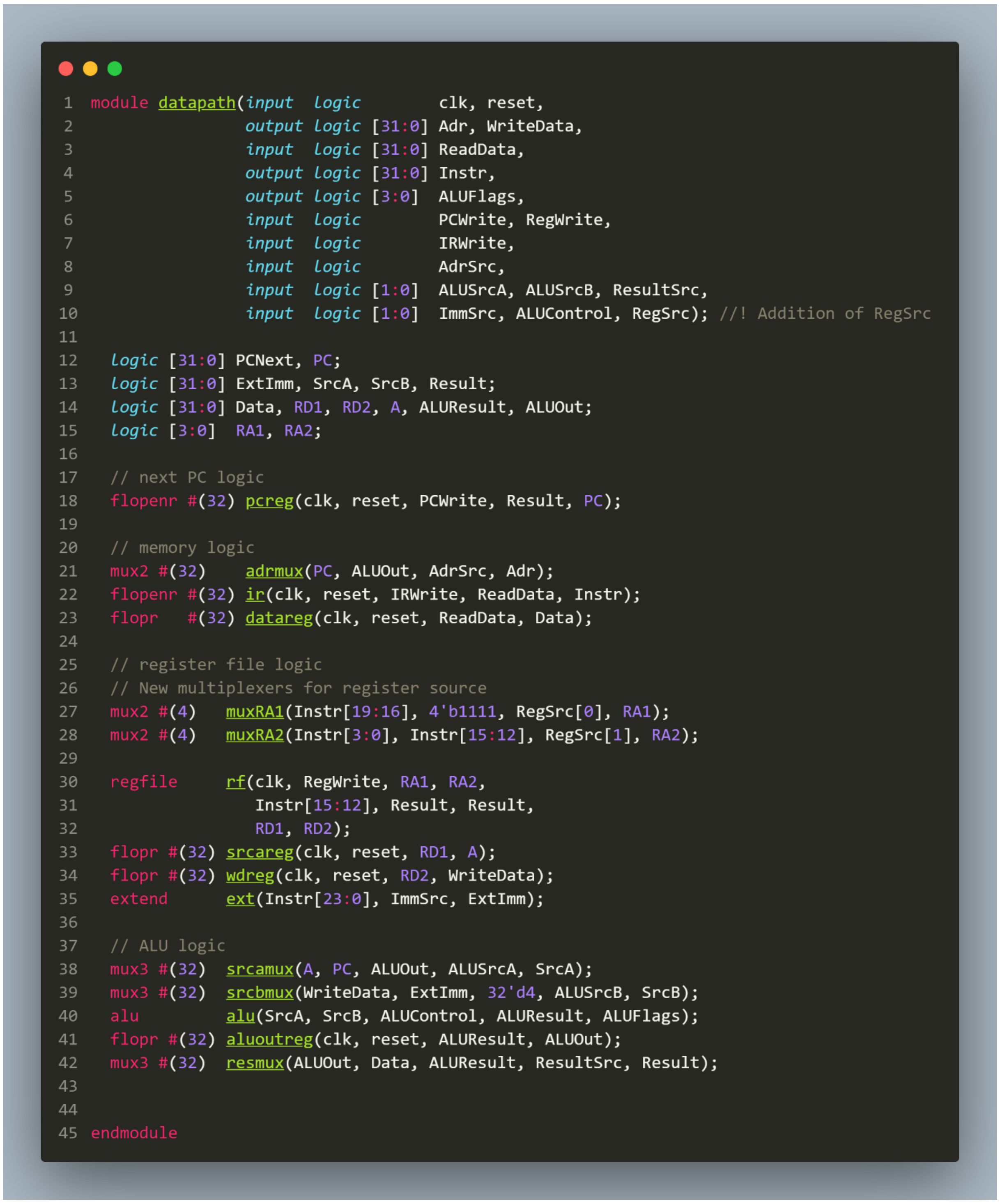
```
module controller(input  logic        clk,
                  input  logic        reset,
                  input  logic [31:12] Instr,
                  input  logic [3:0]   ALUFlags,
                  output logic         PCWrite,
                  output logic         MemWrite,
                  output logic         RegWrite,
                  output logic         IRWrite,
                  output logic         AdrSrc,
                  output logic [1:0]   ALUSrcA,
                  output logic [1:0]   ALUSrcB,
                  output logic [1:0]   ResultSrc,
                  output logic [1:0]   ImmSrc,
                  output logic [1:0]   ALUControl,
                  output logic [1:0]   RegSrc); //! Addition of RegSrc

  logic [1:0] FlagW;
  logic       PCS, NextPC, RegW, MemW;

  decode dec(clk, reset, Instr[27:26], Instr[25:20], Instr[15:12],
             FlagW, PCS, NextPC, RegW, MemW,
             IRWrite, AdrSrc, ResultSrc,
             ALUSrcA, ALUSrcB, ImmSrc, ALUControl, RegSrc); //! Addition of RegSrc
  condlogic cl(clk, reset, Instr[31:28], ALUFlags,
               FlagW, PCS, NextPC, RegW, MemW,
               PCWrite, RegWrite, MemWrite);
endmodule
```

Addition of this register in the controller



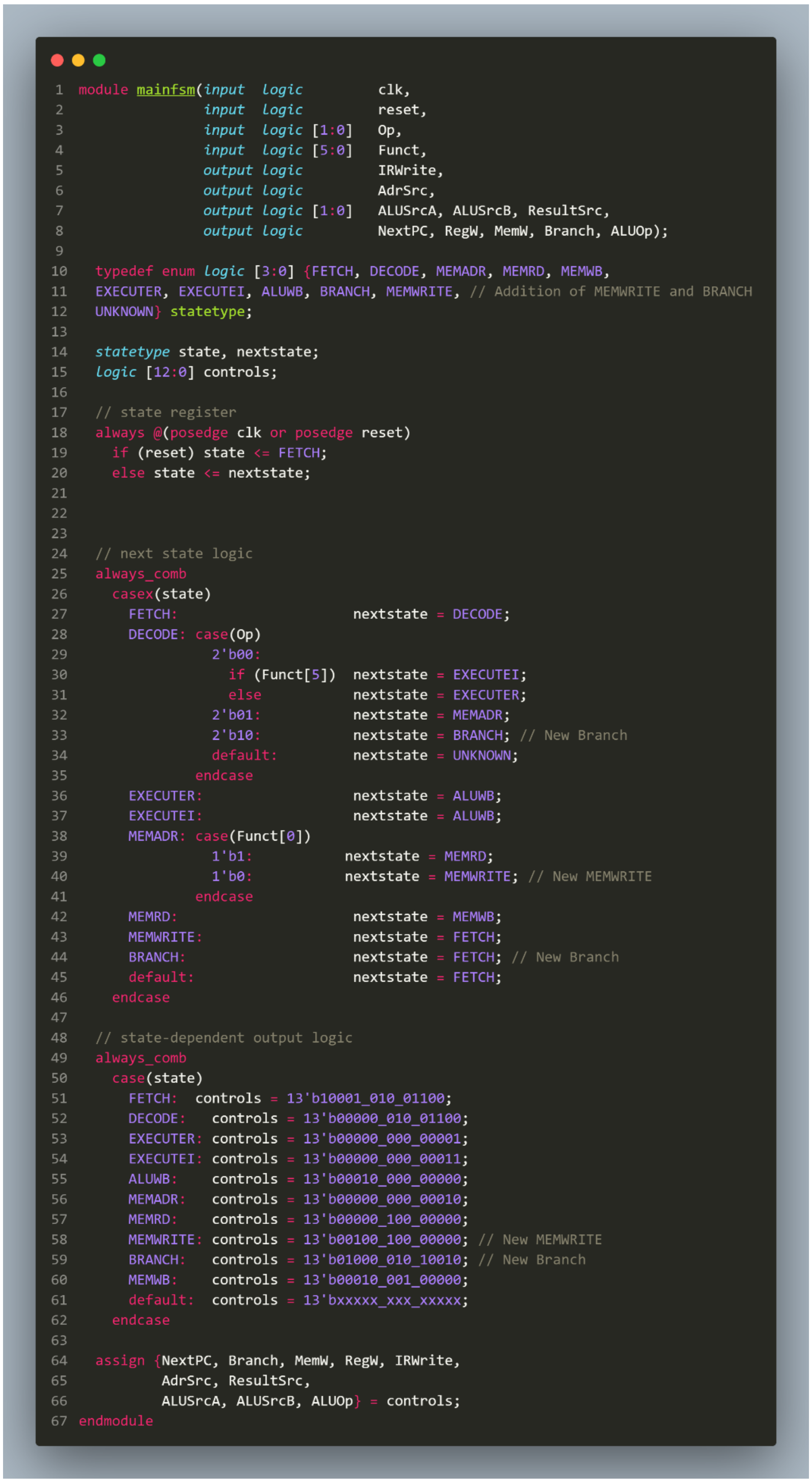
```
 1 module decode(input  logic       clk, reset,
 2               input  logic [1:0] Op,
 3               input  logic [5:0] Funct,
 4               input  logic [3:0] Rd,
 5               output logic [1:0] FlagW,
 6               output logic       PCS, NextPC, RegW, MemW,
 7               output logic       IRWrite, AdrSrc,
 8               output logic [1:0] ResultSrc, ALUSrcA, ALUSrcB,
 9               output logic [1:0] ImmSrc, ALUControl,
10               output logic [1:0] RegSrc); //! Addition of RegSrc
11 
12   logic       Branch, ALUOp;
13 
14   // Main FSM
15   mainfsm fsm(clk, reset, Op, Funct,
16               IRWrite, AdrSrc,
17               ALUSrcA, ALUSrcB, ResultSrc,
18               NextPC, RegW, MemW, Branch, ALUOp);
19 
20   always_comb
21     if (ALUOp) begin                 // which Data-processing Instr?
22       case(Funct[4:1])
23           4'b0100: ALUControl = 2'b00; // ADD
24           4'b0010: ALUControl = 2'b01; // SUB
25           4'b0000: ALUControl = 2'b10; // AND
26           4'b1100: ALUControl = 2'b11; // ORR
27           default: ALUControl = 2'bx;  // unimplemented
28       endcase
29       FlagW[1]      = Funct[0]; // update N & Z flags if S bit is set
30       FlagW[0]      = Funct[0] & (ALUControl == 2'b00 | ALUControl == 2'b01);
31     end else begin
32       ALUControl = 2'b00; // add for non data-processing instructions
33       FlagW      = 2'b00; // don't update Flags
34     end
35 
36   // PC Logic
37   assign PCS  = ((Rd == 4'b1111) & RegW) | Branch;
38 
39   // Register Logic
40   assign RegSrc[0] = (Op == 2'b10); //! Addition of RegSrc
41   assign RegSrc[1] = (Op == 2'b01); //! Addition of RegSrc
42 
43   // Instr Decoder
44   assign ImmSrc     = Op;
45 
46 endmodule
```

Addition of the register RegSrc in the decoder and addition of the logic for the register in line 40 & 41



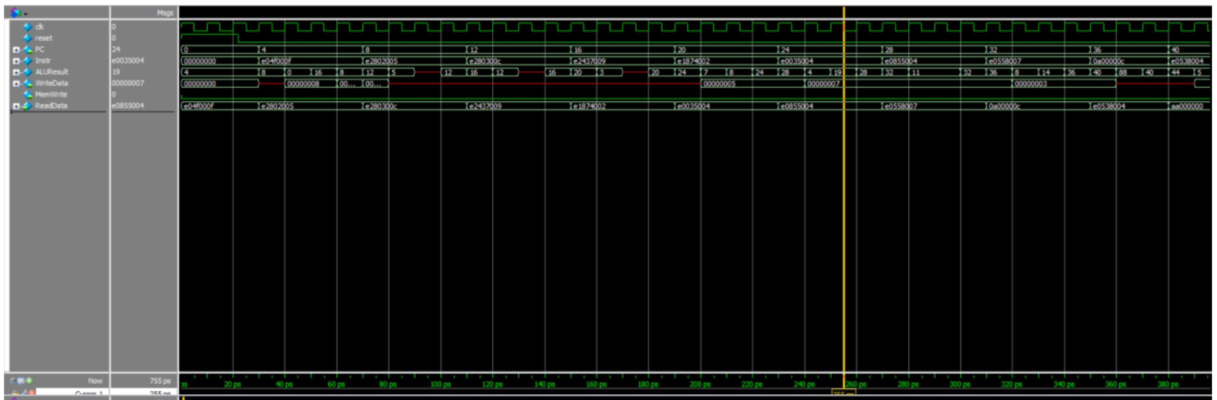
```
1  module datapath(input  logic        clk, reset,
2                  output logic [31:0] Adr, WriteData,
3                  input  logic [31:0] ReadData,
4                  output logic [31:0] Instr,
5                  output logic [3:0]  ALUFlags,
6                  input  logic        PCWrite, RegWrite,
7                  input  logic        IRWrite,
8                  input  logic        AdrSrc,
9                  input  logic [1:0]  ALUSrcA, ALUSrcB, ResultSrc,
10                 input  logic [1:0]  ImmSrc, ALUControl, RegSrc); //! Addition of RegSrc
11
12   logic [31:0] PCNext, PC;
13   logic [31:0] ExtImm, SrcA, SrcB, Result;
14   logic [31:0] Data, RD1, RD2, A, ALUResult, ALUOut;
15   logic [3:0]  RA1, RA2;
16
17   // next PC logic
18   flopenr #(32) pcreg(clk, reset, PCWrite, Result, PC);
19
20   // memory logic
21   mux2 #(32)    adrmux(PC, ALUOut, AdrSrc, Adr);
22   flopenr #(32) ir(clk, reset, IRWrite, ReadData, Instr);
23   flopr   #(32) datareg(clk, reset, ReadData, Data);
24
25   // register file logic
26   // New multiplexers for register source
27   mux2 #(4)   muxRA1(Instr[19:16], 4'b1111, RegSrc[0], RA1);
28   mux2 #(4)   muxRA2(Instr[3:0], Instr[15:12], RegSrc[1], RA2);
29
30   regfile     rf(clk, RegWrite, RA1, RA2,
31                  Instr[15:12], Result, Result,
32                  RD1, RD2);
33   flopr #(32) srcareg(clk, reset, RD1, A);
34   flopr #(32) wdreg(clk, reset, RD2, WriteData);
35   extend      ext(Instr[23:0], ImmSrc, ExtImm);
36
37   // ALU logic
38   mux3 #(32)  srcamux(A, PC, ALUOut, ALUSrcA, SrcA);
39   mux3 #(32)  srcbmux(WriteData, ExtImm, 32'd4, ALUSrcB, SrcB);
40   alu         alu(SrcA, SrcB, ALUControl, ALUResult, ALUFlags);
41   flopr #(32) aluoutreg(clk, reset, ALUResult, ALUOut);
42   mux3 #(32)  resmux(ALUOut, Data, ALUResult, ResultSrc, Result);
43
44
45 endmodule
```

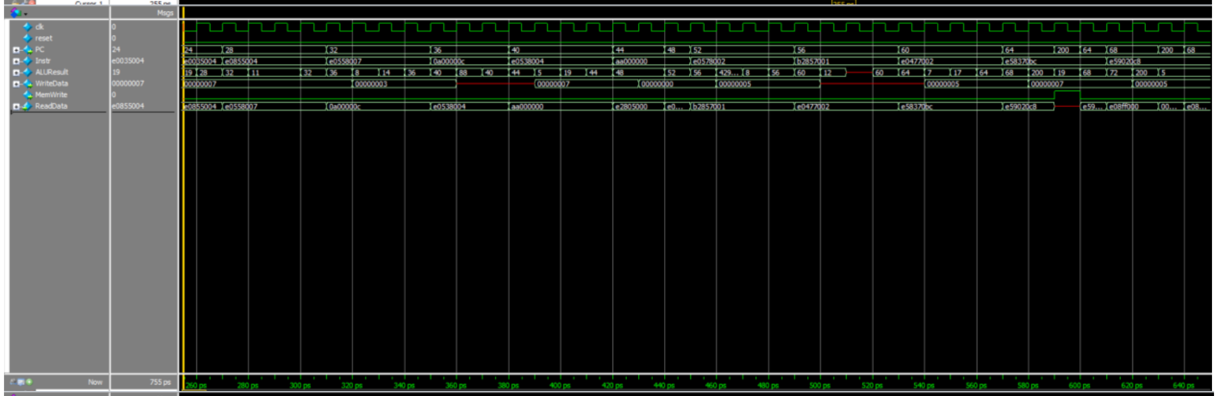
Addition of the register in the datapath and addition of the multiplexer at line 27 and 28

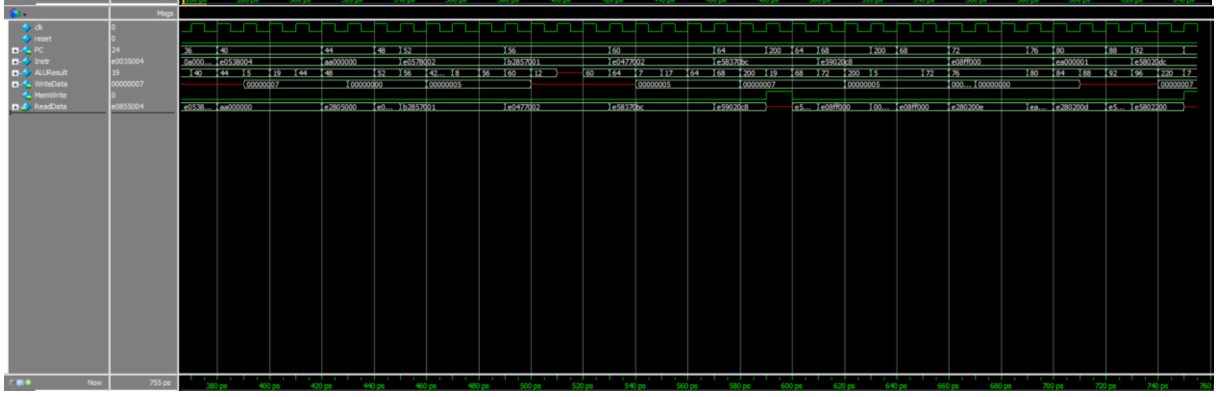


```systemverilog
module mainfsm(input  logic         clk,
               input  logic         reset,
               input  logic [1:0]   Op,
               input  logic [5:0]   Funct,
               output logic         IRWrite,
               output logic         AdrSrc,
               output logic [1:0]   ALUSrcA, ALUSrcB, ResultSrc,
               output logic         NextPC, RegW, MemW, Branch, ALUOp);

  typedef enum logic [3:0] {FETCH, DECODE, MEMADR, MEMRD, MEMWB,
  EXECUTER, EXECUTEI, ALUWB, BRANCH, MEMWRITE, // Addition of MEMWRITE and BRANCH
  UNKNOWN} statetype;

  statetype state, nextstate;
  logic [12:0] controls;

  // state register
  always @(posedge clk or posedge reset)
    if (reset) state <= FETCH;
    else state <= nextstate;



  // next state logic
  always_comb
    casex(state)
      FETCH:                     nextstate = DECODE;
      DECODE: case(Op)
                2'b00:
                  if (Funct[5])  nextstate = EXECUTEI;
                  else           nextstate = EXECUTER;
                2'b01:           nextstate = MEMADR;
                2'b10:           nextstate = BRANCH; // New Branch
                default:         nextstate = UNKNOWN;
              endcase
      EXECUTER:                  nextstate = ALUWB;
      EXECUTEI:                  nextstate = ALUWB;
      MEMADR: case(Funct[0])
                1'b1:           nextstate = MEMRD;
                1'b0:           nextstate = MEMWRITE; // New MEMWRITE
              endcase
      MEMRD:                     nextstate = MEMWB;
      MEMWRITE:                  nextstate = FETCH;
      BRANCH:                    nextstate = FETCH; // New Branch
      default:                   nextstate = FETCH;
    endcase

  // state-dependent output logic
  always_comb
    case(state)
      FETCH:  controls = 13'b10001_010_01100;
      DECODE:   controls = 13'b00000_010_01100;
      EXECUTER: controls = 13'b00000_000_00001;
      EXECUTEI: controls = 13'b00000_000_00011;
      ALUWB:    controls = 13'b00010_000_00000;
      MEMADR:   controls = 13'b00000_000_00010;
      MEMRD:    controls = 13'b00000_100_00000;
      MEMWRITE: controls = 13'b00100_100_00000; // New MEMWRITE
      BRANCH:   controls = 13'b01000_010_10010; // New Branch
      MEMWB:    controls = 13'b00010_001_00000;
      default:  controls = 13'bxxxxx_xxx_xxxxx;
    endcase

  assign {NextPC, Branch, MemW, RegW, IRWrite,
          AdrSrc, ResultSrc,
          ALUSrcA, ALUSrcB, ALUOp} = controls;
endmodule
```

Addition of the cond logic for the Branch and MemWrite instructions


Msgs
clk 0
reset 0
PC 24
Instr e0035004
ALUResult 19
WriteData 00000007
MemWrite 0
ReadData e0855004
Now 755 ps
Cursor 1 255 ps


Msgs
clk 0
reset 0
PC 24
Instr e0035004
ALUResult 19
WriteData 00000007
MemWrite 0
ReadData e0855004
Now 755 ps


Msgs
clk 0
reset 0
PC 24
Instr e0035004
ALUResult 19
WriteData 00000007
MemWrite 0
ReadData e0855004
Now 755 ps