Central Connecticut State University

**BUILDING A PIPELINED DATAPATH WITH 3-STAGES (IF,ID,EX)**

**Report 3**

prepared by

Thi Minh Huyen Le

Brendan Manzolli

Erik Marrero

Fall - 2021

1. **Tasks:**

Thi Minh Huyen Le: Instruction Memory, Combine Code.

Brendan Manzolli: Multiplexes, Diagrams

Erik Marrero: MainControl, Instructions

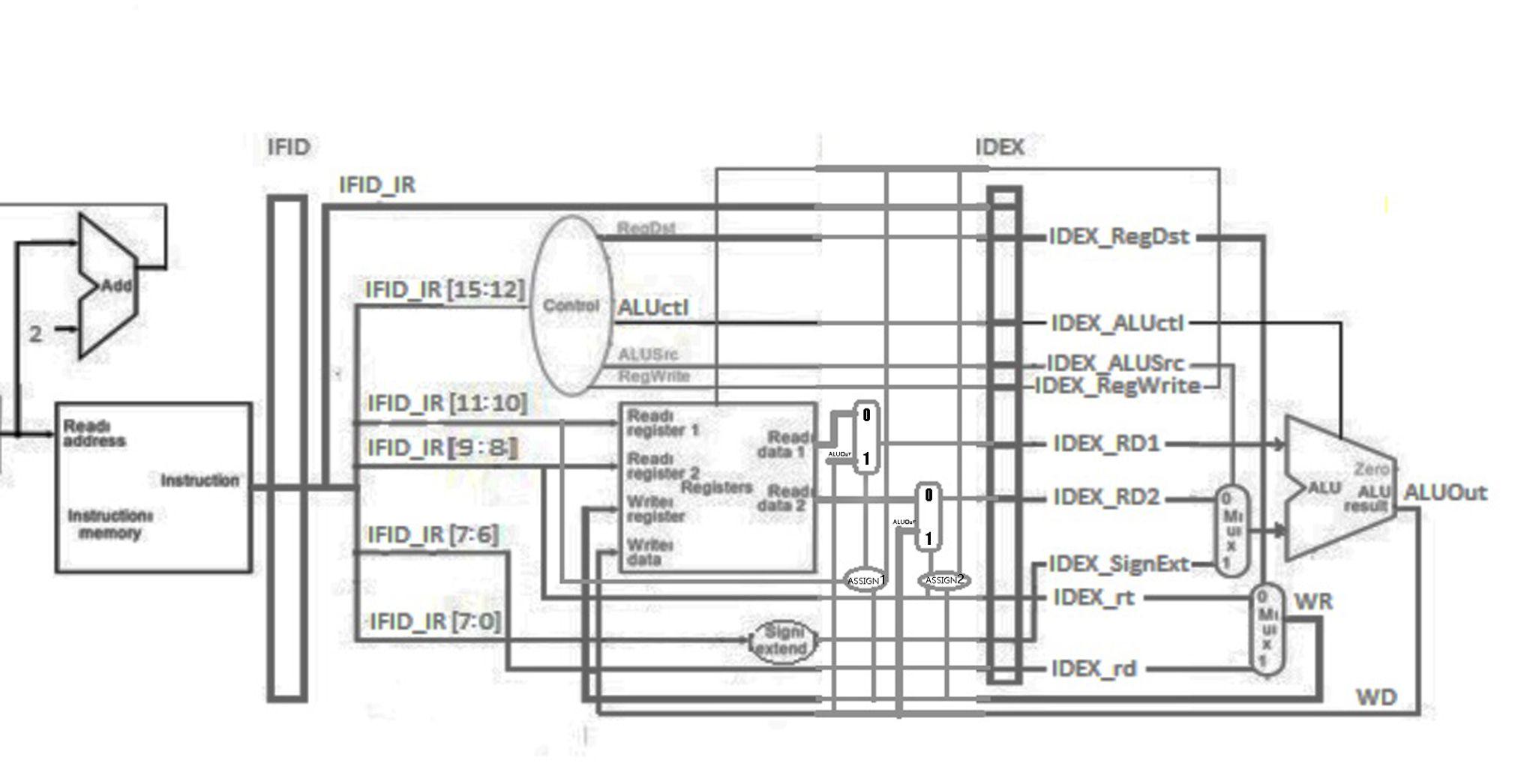
1. **Description**

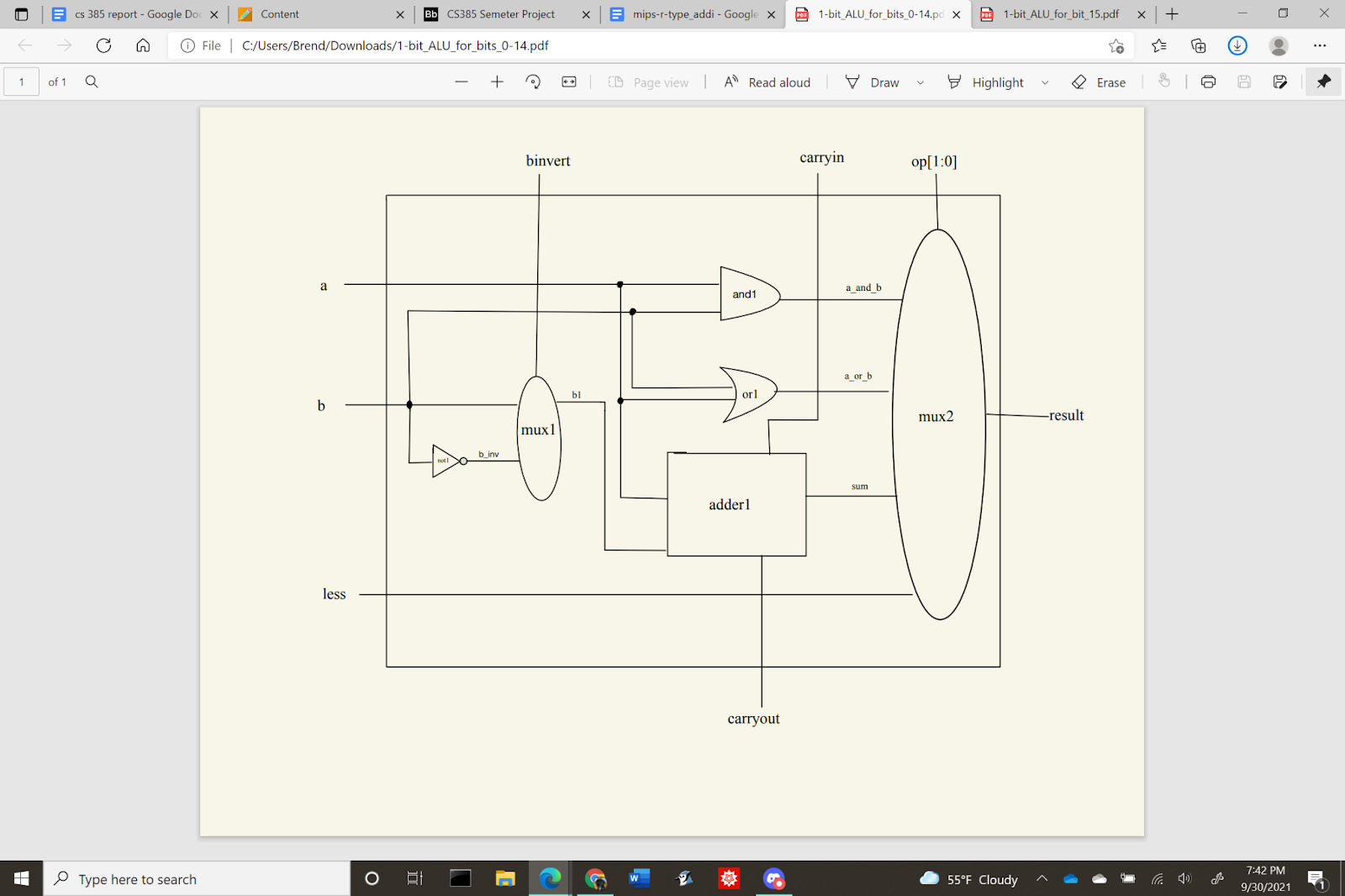
Instructions:

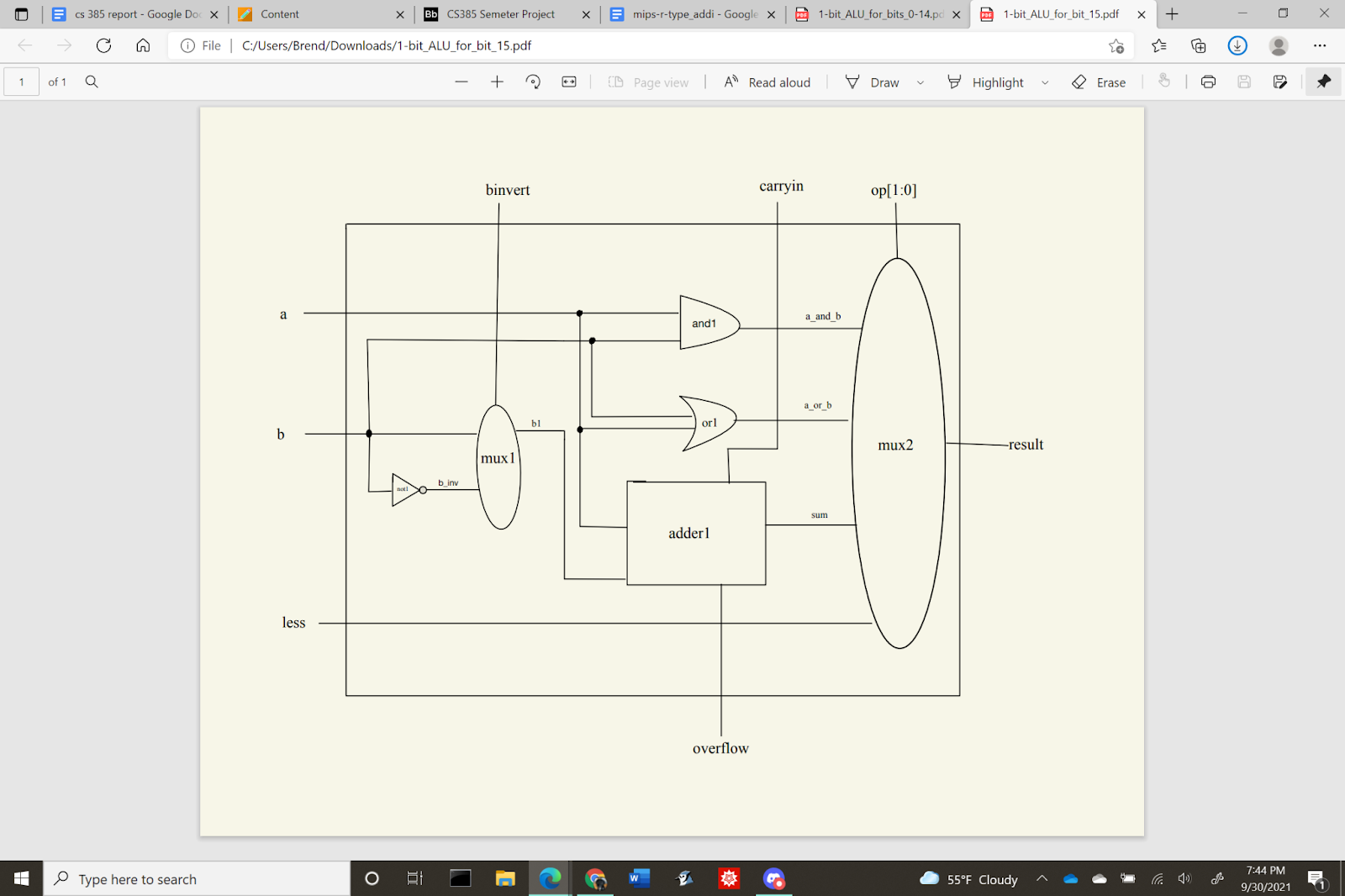
| Instruction (I-Type: addi) | Opcode  (4 bits) [15-12] | Rs  (2 bits) [11-10] | Rt (2 bits) [9-8] | Value or Address  (8 bits) [7-0] | Binary Instruction |
| --- | --- | --- | --- | --- | --- |
| addi $t1,$0,15 | 0101 | 00 | 01 | 00001111 | 0101\_00\_01\_00001111 |
| addi $t2,$0,7 | 0101 | 00 | 10 | 00000111 | 0101\_00\_10\_00000111 |
| nop | 0000 | 00 | 00 | 00000000 | 0000\_00\_00\_00000000 |

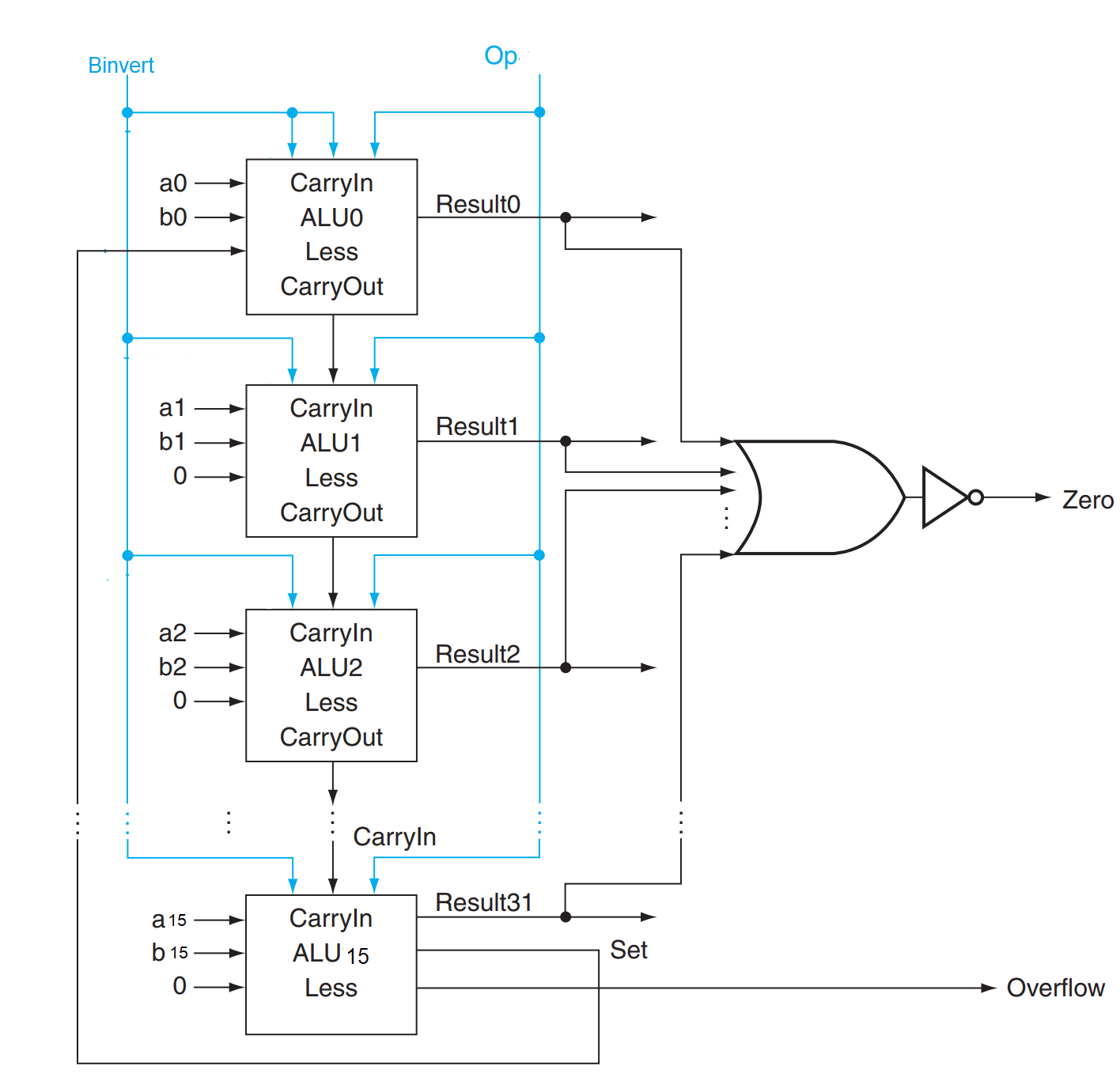
| Instruction (R-Type) | Opcode  (4 bits) [15-12] | Rs  (2 bits) [11-10] | Rt  (2bit) [9-8] | Rd  (2bits) [7-6] | Unused  (6 bits) [5-0] | Binary Instruction |
| --- | --- | --- | --- | --- | --- | --- |
| and $t3,$t1,$t2 | 0010 | 01 | 10 | 11 | 000000 | 0010\_01\_10\_11\_000000 |
| sub $t2,$t1,$t3 | 0001 | 01 | 11 | 10 | 000000 | 0001\_01\_11\_10\_000000 |
| or $t2,$t2,$t3 | 0011 | 10 | 11 | 10 | 000000 | 0011\_10\_11\_10\_000000 |
| add $t3,$t2,$t3 | 0000 | 10 | 11 | 11 | 000000 | 0000\_10\_11\_11\_000000 |
| slt $t1,$t3,$t2 | 0100 | 11 | 10 | 01 | 000000 | 0100\_11\_10\_01\_000000 |
| slt $t1,$t2,$t3 | 0100 | 10 | 11 | 01 | 000000 | 0100\_10\_11\_01\_00000 |

1. **Block logic diagrams**

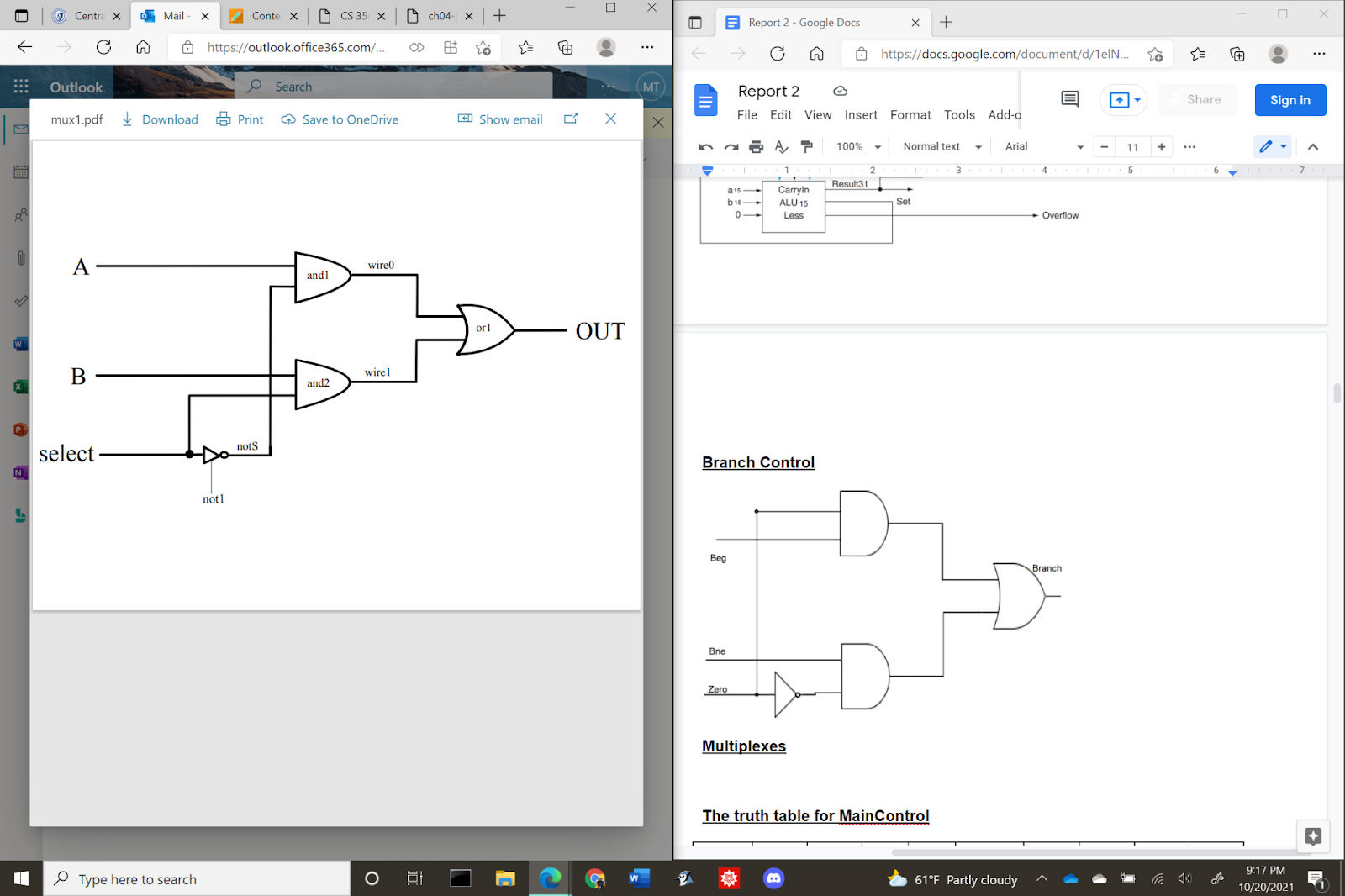
**CPU** 

**1-bit ALU for bits 0-14**

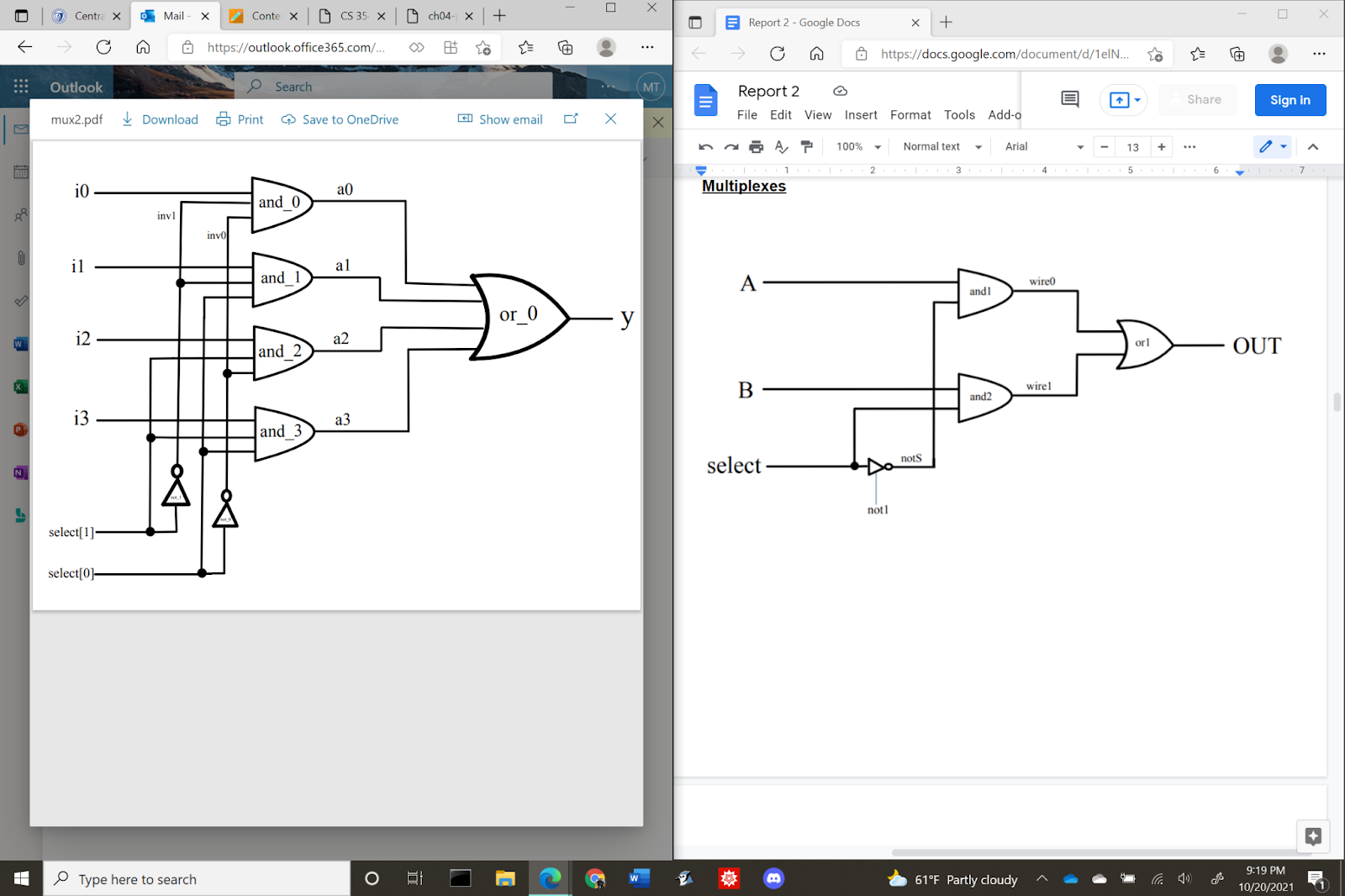
**1-bit ALU for bit 15**

**16bit-ALU** 

**2x1 Multiplexer**

****

**4x1 Multiplexer**

****

**The truth table for MainControl**

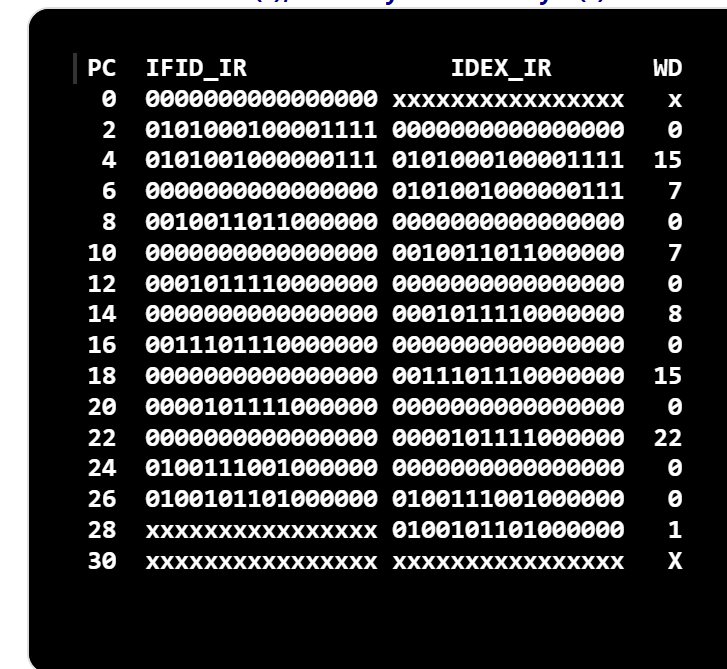
| **Instruction** | **OpCode** | **RegDst** | **ALUSrc** | **RegWrite** | **ALUCtl** |
| --- | --- | --- | --- | --- | --- |
| **add** | **0000** | **1** | **0** | **1** | **010** |
| **sub** | **0001** | **1** | **0** | **1** | **110** |
| **and** | **0010** | **1** | **0** | **1** | **000** |
| **or** | **0011** | **1** | **0** | **1** | **001** |
| **slt** | **0100** | **1** | **0** | **1** | **111** |
| **addi** | **0101** | **0** | **1** | **1** | **010** |
| **slti** | **0110** | **0** | **1** | **1** | **111** |
| **andi** | **0111** | **0** | **1** | **1** | **000** |
| **ori** | **1000** | **0** | **1** | **1** | **001** |

**Verilog source code**

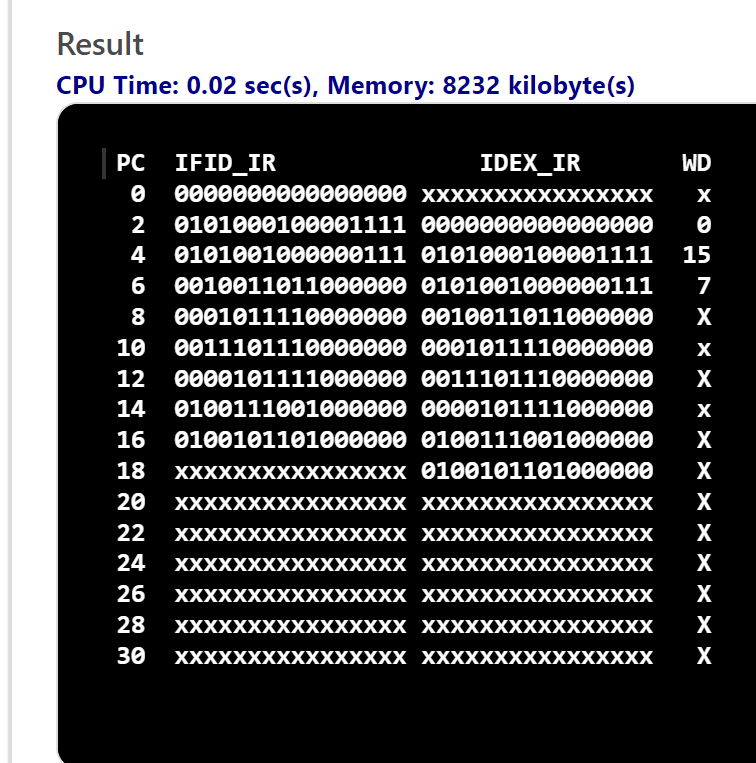
| //Group : Thi Minh Huyen Le, Brendan Manzolli, Erik Marrero  // Behavioral model of MIPS - single cycle implementation, R-types and addi,slti,anddi,ori  // --------------REGISTER FILE ---------------  // Assign RR1, RR2 to RD1, RD2  // When RegWrite is on, assign WD to WR  module reg\_file (RR1,RR2,WR,WD,RegWrite,RD1,RD2,clock);  input [1:0] RR1,RR2,WR;  input [15:0] WD;  input RegWrite,clock;  output [15:0] RD1,RD2;  reg [15:0] Regs[0:3];  assign RD1 = Regs[RR1];  assign RD2 = Regs[RR2];  initial Regs[0] = 0;  always @(negedge clock)  if (RegWrite==1 & WR!=0)  Regs[WR] <= WD;  endmodule  //-----------END REGISTER FILE-----------  //--------------ALU----------------------  // A 32-bit ALU constructed from the 31 copies of the 1-bit ALU in the top and one 1-bit ALUmsb in the bottom.  module ALU (op,a,b,result,zero);  input [15:0] a;  input [15:0] b;  input [2:0] op;  output [15:0] result;  output zero;  wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16;    ALU1 alu0 (a[0],b[0],op[2],op[1:0],set, op[2], c1,result[0]);  ALU1 alu1 (a[1],b[1],op[2],op[1:0],1'b0, c1, c2,result[1]);  ALU1 alu2 (a[2],b[2],op[2],op[1:0],1'b0, c2, c3,result[2]);  ALU1 alu3 (a[3],b[3],op[2],op[1:0],1'b0, c3, c4,result[3]);  ALU1 alu4 (a[4],b[4],op[2],op[1:0],1'b0, c4, c5,result[4]);  ALU1 alu5 (a[5],b[5],op[2],op[1:0],1'b0, c5, c6,result[5]);  ALU1 alu6 (a[6],b[6],op[2],op[1:0],1'b0, c6, c7,result[6]);  ALU1 alu7 (a[7],b[7],op[2],op[1:0],1'b0, c7, c8,result[7]);  ALU1 alu8 (a[8],b[8],op[2],op[1:0],1'b0, c8, c9,result[8]);  ALU1 alu9 (a[9],b[9],op[2],op[1:0],1'b0, c9, c10,result[9]);  ALU1 alu10 (a[10],b[10],op[2],op[1:0],1'b0, c10, c11,result[10]);  ALU1 alu11 (a[11],b[11],op[2],op[1:0],1'b0, c11, c12,result[11]);  ALU1 alu12 (a[12],b[12],op[2],op[1:0],1'b0, c12, c13,result[12]);  ALU1 alu13 (a[13],b[13],op[2],op[1:0],1'b0, c13, c14,result[13]);  ALU1 alu14 (a[14],b[14],op[2],op[1:0],1'b0, c14, c15,result[14]);  ALUmsb alu15 (a[15],b[15],op[2],op[1:0],1'b0, c15, c16,result[15],set);    or or1(or01, result[0], result[1], result[2], result[3], result[4], result[5], result[6], result[7], result[8], result[9], result[10], result[11], result[12], result[13], result[14], result[15]);  not not1(zero, or01);  endmodule  // 1-bit ALU for bits 0-2  // A 1-bit ALU that performs AND, OR, addition , subtraction on a and b,  module ALU1 (a,b,binvert,op,less,carryin,carryout,result);  input a,b,less,carryin,binvert;  input [1:0] op;  output carryout,result;  wire sum, a\_and\_b, a\_or\_b, b\_inv;    not not1(b\_inv, b);  mux2x1 mux1(b,b\_inv,binvert,b1);  and and1(a\_and\_b, a, b);  or or1(a\_or\_b, a, b);  fulladder adder1(sum,carryout,a,b1,carryin);  mux4x1 mux2(a\_and\_b,a\_or\_b,sum,less,op[1:0],result);  endmodule  // 1-bit ALU for the most significant bit  // A 1-bit ALU that performs AND, OR, addition , subtraction on a and b, GIVE THE MOST SIGNIFICANT BIT AS AN OUTPUT SUM  module ALUmsb (a,b,binvert,op,less,carryin,overflow,result,sum);  input a,b,less,carryin,binvert;  input [1:0] op;  output overflow,result,sum;  wire sum, a\_and\_b, a\_or\_b, b\_inv;    not not1(b\_inv, b);  mux2x1 mux1(b,b\_inv,binvert,b1);  and and1(a\_and\_b, a, b);  or or1(a\_or\_b, a, b);  fulladder adder1(sum,overflow,a,b1,carryin);  mux4x1 mux2(a\_and\_b,a\_or\_b,sum,less,op[1:0],result);  endmodule  module halfadder (S,C,x,y);  input x,y;  output S,C;  xor (S,x,y);  and (C,x,y);  endmodule  module fulladder (S,C,x,y,z);  input x,y,z;  output S,C;  wire S1,D1,D2;  halfadder HA1 (S1,D1,x,y),  HA2 (S,D2,S1,z);  or g1(C,D2,D1);  endmodule  // Multiplexor two 1bit-inputs  module mux2x1(A,B,select,OUT);  input A,B,select;  output OUT;  wire wire0, wire1;  not not1 (notS,select);  and and1 (wire0, A,notS),  and2 (wire1, B,select);  or or1 (OUT,wire0,wire1);  endmodule  // Multiplexor four 1bit-inputs  module mux4x1(i0,i1,i2,i3,select,y);  output y; // Output  input i0, i1, i2, i3; // Input ports.  input [1:0] select; // Select lines.    // intermediate wires  wire inv0, inv1; // Inverter outputs.  wire a0, a1, a2, a3; // AND gates outputs.    // Inverters.  not not\_1 (inv1, select[1]);  not not\_0 (inv0, select[0]);    // 4-input AND gates.  and and\_0 (a0, i0, inv1, inv0);  and and\_1 (a1, i1, inv1, select[0]);  and and\_2 (a2, i2, select[1], inv0);  and and\_3 (a3, i3, select[1], select[0]);    // 4-input OR gate.  or or\_0 (y, a0, a1, a2, a3);  endmodule  //--------------END ALU-----------------  //--------------MULTIPLEXORS-------------  // Multiplexor two 16-bit inputs.  module mux16bitx2x1 (A,B,Select,Out);  input [15:0] A,B;  input Select;  output [15:0] Out;  mux2x1 m0 (A[0],B[0],Select,Out[0]),  m1 (A[1],B[1],Select,Out[1]),  m2 (A[2],B[2],Select,Out[2]),  m3 (A[3],B[3],Select,Out[3]),  m4 (A[4],B[4],Select,Out[4]),  m5 (A[5],B[5],Select,Out[5]),  m6 (A[6],B[6],Select,Out[6]),  m7 (A[7],B[7],Select,Out[7]),  m8 (A[8],B[8],Select,Out[8]),  m9 (A[9],B[9],Select,Out[9]),  m10 (A[10],B[10],Select,Out[10]),  m11 (A[11],B[11],Select,Out[11]),  m12 (A[12],B[12],Select,Out[12]),  m13 (A[13],B[13],Select,Out[13]),  m14 (A[14],B[14],Select,Out[14]),  m15 (A[15],B[15],Select,Out[15]);  endmodule  // Multiplexor two 2-bit inputs.  module mux2bitx2x1 (A,B,Select,Out);  input [1:0] A,B;  input Select;  output [1:0] Out;  mux2x1 m0 (A[0],B[0],Select,Out[0]),  m1 (A[1],B[1],Select,Out[1]);  endmodule  //--------------END MULTIPLEXORS------------  //--------------MAIN CONTROL---------------  /\*\*\* 16-bit CPU control source code \*\*\*/  module mainCtrl (op, ctrl);  input [3:0] op;  output reg [5:0] ctrl;  // ctrl bits: RegDst, ALUSrc, RegWrite, ALUCtl(3bits)  always @(op) case (op)  4'b0000: ctrl <= 6'b101010; // ADD  4'b0001: ctrl <= 6'b101110; // SUB  4'b0010: ctrl <= 6'b101000; // AND  4'b0011: ctrl <= 6'b101001; // OR  4'b0100: ctrl <= 6'b101111; // SLT  4'b0101: ctrl <= 6'b011010; // ADDI  4'b0110: ctrl <= 6'b011111; // SLTI  4'b0111: ctrl <= 6'b011000; // ANDI  4'b1000: ctrl <= 6'b011001; // ORI  endcase  endmodule  //------------END MAIN CONTROL----------------  //------------CPU--------------  module CPU (clock,PC,IFID\_IR,IDEX\_IR,WD);  input clock;  output [15:0] PC,IFID\_IR,IDEX\_IR,WD;  /\*  initial begin  // Program with nop's - no hazards  IMemory[0] = 16'b0101\_00\_01\_00001111; // addi $t1, $0, 15 ($t1=15)  IMemory[1] = 16'b0101\_00\_10\_00000111; // addi $t2, $0, 7 ($t2= 7)  IMemory[2] = 16'b0000\_00\_00\_00000000; // nop  IMemory[3] = 16'b0010\_01\_10\_11\_000000; // and $t3, $t1, $t2 ($t3= 7)  IMemory[4] = 16'b0000\_00\_00\_00000000; // nop  IMemory[5] = 16'b0001\_01\_11\_10\_000000; // sub $t2, $t1, $t3 ($t2= 8)  IMemory[6] = 16'b0000\_00\_00\_00000000; // nop  IMemory[7] = 16'b0011\_10\_11\_10\_000000; // or $t2, $t2, $t3 ($t2=15)  IMemory[8] = 16'b0000\_00\_00\_00000000; // nop  IMemory[9] = 16'b0000\_10\_11\_11\_000000; // add $t3, $t2, $t3 ($t3=22)  IMemory[10] = 16'b0000\_00\_00\_00000000; // nop  IMemory[11] = 16'b0100\_11\_10\_01\_000000; // slt $t1, $t3, $t2 ($t1= 0)  IMemory[12] = 16'b0100\_10\_11\_01\_000000; // slt $t1, $t2, $t3 ($t1= 1)  end  \*/  initial begin  // Program without nop's - wrong results due to data hazards  IMemory[0] = 16'b0101\_00\_01\_00001111; // addi $t1, $0, 15 ($t1=15)  IMemory[1] = 16'b0101\_00\_10\_00000111; // addi $t2, $0, 7 ($t2= 7)  IMemory[2] = 16'b0010\_01\_10\_11\_000000; // and $t3, $t1, $t2 ($t3= 7)  IMemory[3] = 16'b0001\_01\_11\_10\_000000; // sub $t2, $t1, $t3 ($t2= 8)  IMemory[4] = 16'b0011\_10\_11\_10\_000000; // or $t2, $t2, $t3 ($t2=15)  IMemory[5] = 16'b0000\_10\_11\_11\_000000; // add $t3, $t2, $t3 ($t3=22)  IMemory[6] = 16'b0100\_11\_10\_01\_000000; // slt $t1, $t3, $t2 ($t1= 0)  IMemory[7] = 16'b0100\_10\_11\_01\_000000; // slt $t1, $t2, $t3 ($t1= 1)  end  // Pipeline stages  //=== IF STAGE ===  wire [15:0] NextPC;  reg[15:0] PC, IMemory[0:1023];  //--------------------------------  reg[15:0] IFID\_IR;  //--------------------------------  ALU fetch (3'b010,PC,16'b10,NextPC,Unused);  //=== ID STAGE ===  wire [5:0] Control;  wire [15:0] RD1,RD2,SignExtend,WD;  wire [15:0] FWD\_RD1,FWD\_RD2; // Outputs of the forwarding muxes  //----------------------------------------------------  reg [15:0] IDEX\_IR; // For monitoring the pipeline  reg IDEX\_RegWrite,IDEX\_ALUSrc,IDEX\_RegDst;  reg [15:0] IDEX\_RD1,IDEX\_RD2,IDEX\_SignExt;  reg [1:0] IDEX\_rt,IDEX\_rd;  wire [1:0] WR;  //----------------------------------------------------  reg\_file rf (IFID\_IR[11:10],IFID\_IR[9:8],WR,WD,IDEX\_RegWrite,RD1,RD2,clock);  mainCtrl MainCtr (IFID\_IR[15:12],Control);  assign SignExtend = {{8{IFID\_IR[7]}},IFID\_IR[7:0]};    //=== EXE STAGE ===  wire [15:0] B,ALUOut;  reg [2:0] ALUctl;  reg condition;  ALU ex (ALUctl, IDEX\_RD1, B, ALUOut, Zero);  mux16bitx2x1 muxB (IDEX\_RD2, IDEX\_SignExt, IDEX\_ALUSrc, B); // ALUSrc Mux  mux2bitx2x1 muxWR (IDEX\_rt, IDEX\_rd, IDEX\_RegDst, WR); // RegDst Mux  assign WD = ALUOut;  // Forwarding multiplexers  assign assign1 = (IDEX\_RegWrite && WR==IFID\_IR[11:10]);  assign assign2 = (IDEX\_RegWrite && WR==IFID\_IR[9:8]);  mux16bitx2x1 ReadD1 (RD1,ALUOut,assign1,FWD\_RD1);  mux16bitx2x1 ReadD2 (RD2,ALUOut,assign2,FWD\_RD2);  // assign FWD\_RD1 = (IDEX\_RegWrite && WR==IFID\_IR[11:10]) ? ALUOut: RD1;  // assign FWD\_RD2 = (IDEX\_RegWrite && WR==IFID\_IR[9:8]) ? ALUOut: RD2;  initial begin  PC = 0;  IFID\_IR = 0; // clear pipeline register to avoid forwarding from empty pipeline  IDEX\_RegWrite =0;  end  // Running the pipeline  always @(negedge clock) begin  // Stage 1 - IF  PC <= NextPC;  IFID\_IR <= IMemory[PC>>1];  // Stage 2 - ID  IDEX\_IR <= IFID\_IR; // For monitoring the pipeline  {IDEX\_RegDst,IDEX\_ALUSrc,IDEX\_RegWrite,ALUctl} <= Control;  // No Forwarding  IDEX\_RD1 <= RD1;  IDEX\_RD2 <= RD2;  // Forwarding (NOTE: clear IFID\_IR and IDEX\_RegWrite, or add another instruction, e.g. a nop in the beginning of the program)  IDEX\_RD1 <= FWD\_RD1;  IDEX\_RD2 <= FWD\_RD2;  IDEX\_SignExt <= SignExtend;  IDEX\_rt <= IFID\_IR[9:8];  IDEX\_rd <= IFID\_IR[7:6];  // Stage 3 - EX  // No transfers needed here - on negedge WD is written into register WR  end  endmodule  // Test module  module test ();  reg clock;  wire [15:0] PC,IFID\_IR,IDEX\_IR,WD;  CPU test\_cpu(clock,PC,IFID\_IR,IDEX\_IR,WD);  always #1 clock = ~clock;    initial begin  $display (" PC IFID\_IR IDEX\_IR WD");  $monitor ("%3d %b %b %3d", PC,IFID\_IR,IDEX\_IR,WD);  clock = 1;  #29 $finish;  end  endmodule  /\* Compiling and simulation  Program with nop's  ------------------------------  PC IFID\_IR IDEX\_IR WD  0 xxxxxxxx xxxxxxxx 0  4 2009000f xxxxxxxx 0  8 200a0007 2009000f 15  12 00000000 200a0007 7  16 012a5824 00000000 0  20 00000000 012a5824 7  24 012b5022 00000000 0  28 00000000 012b5022 8  32 014b5025 00000000 0  36 00000000 014b5025 15  40 014b5820 00000000 0  44 00000000 014b5820 22  48 016a482a 00000000 0  52 014b482a 016a482a 0  56 xxxxxxxx 014b482a 1  60 xxxxxxxx xxxxxxxx X  Program without nop's  ------------------------------  PC IFID\_IR IDEX\_IR WD  0 xxxxxxxx xxxxxxxx 0  4 2009000f xxxxxxxx 0  8 200a0007 2009000f 15  12 012a5824 200a0007 7  16 012b5022 012a5824 X  20 014b5025 012b5022 x  24 014b5820 014b5025 X  28 016a482a 014b5820 x  32 014b482a 016a482a X  36 xxxxxxxx 014b482a X  40 xxxxxxxx xxxxxxxx X  44 xxxxxxxx xxxxxxxx X  48 xxxxxxxx xxxxxxxx X  52 xxxxxxxx xxxxxxxx X  56 xxxxxxxx xxxxxxxx X  60 xxxxxxxx xxxxxxxx X  \*/ |
| --- |

1. **Test result**

Test1 : Code with the instructions no-opp



Test2 : Code without the instructions no-opp



Test3 : Code without the instructions no-opp and using forwarding

