Central Connecticut State University

**BUILDING A PIPELINED DATAPATH WITH 5-STAGES (IF,ID,EX, MEM,WB)**

**Final Project**

prepared by

Thi Minh Huyen Le

Brendan Manzolli

Erik Marrero

Fall - 2021

1. **Tasks:**

Thi Minh Huyen Le: Combine Code. Results

Brendan Manzolli: Multiplexes, Diagrams

Erik Marrero: MainControl, Instructions

1. **Description**

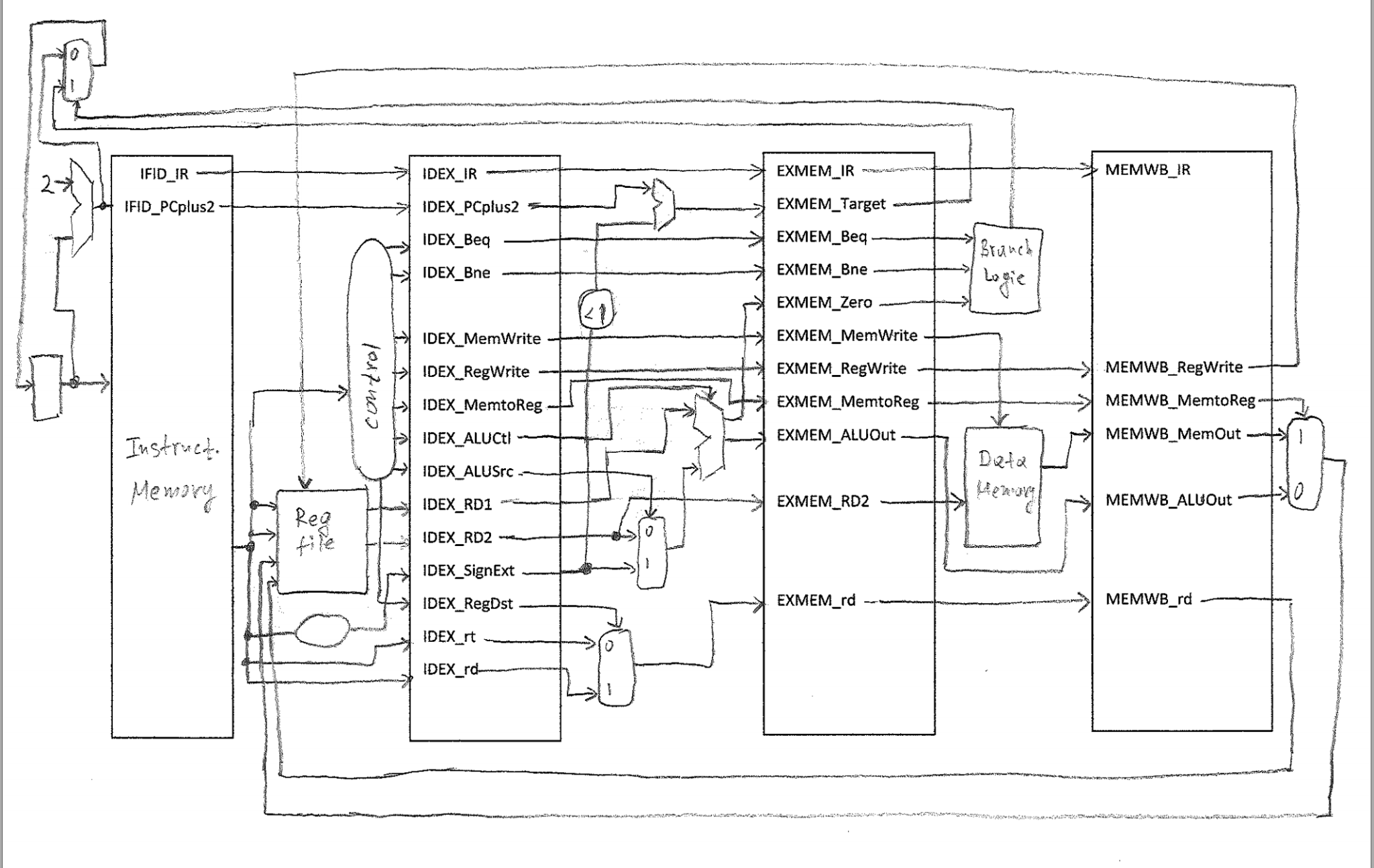
Instructions:

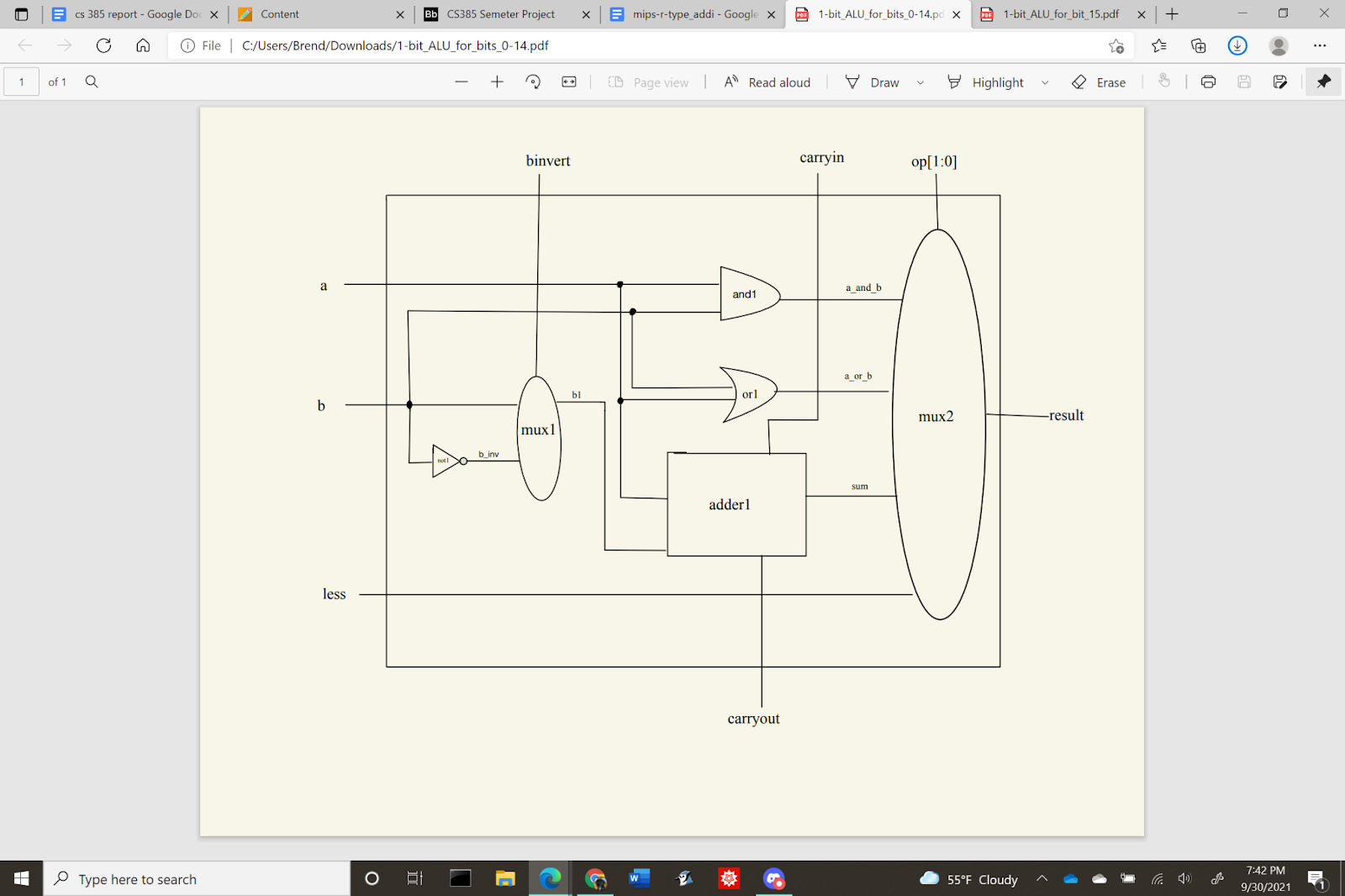
| Instruction (I-Type: addi) | Opcode  (4 bits) [15-12] | Rs  (2 bits) [11-10] | Rt (2 bits) [9-8] | Value or Address  (8 bits) [7-0] | Binary Instruction |
| --- | --- | --- | --- | --- | --- |
| nop | 0000 | 00 | 00 | 00000000 | 0000\_00\_00\_00000000 |
| lw $1, 0($0) | 1001 | 00 | 01 | 00000000 | 1001000100000000 |
| lw $2, 2($0) | 1001 | 00 | 10 | 00000010 | 1001001000000010 |
| beg $3,$0,IMemory[15] | 1011 | 00 | 11 | 00000101 | 1011001100000101 |
| sw $1,2($0) | 1010 | 00 | 01 | 00000010 | 1010000100000010 |
| sw $2,0($0) | 1010 | 00 | 10 | 00000000 | 1010001000000000 |
| lw $1,0($0) | 1001 | 00 | 01 | 00000000 | 1001000100000000 |
| lw $2,2($0) | 1001 | 00 | 10 | 00000010 | 1001001000000010 |
| bne $3,$0,IMemory[6] | 1100 | 00 | 11 | 00000010 | 1100110000000010 |

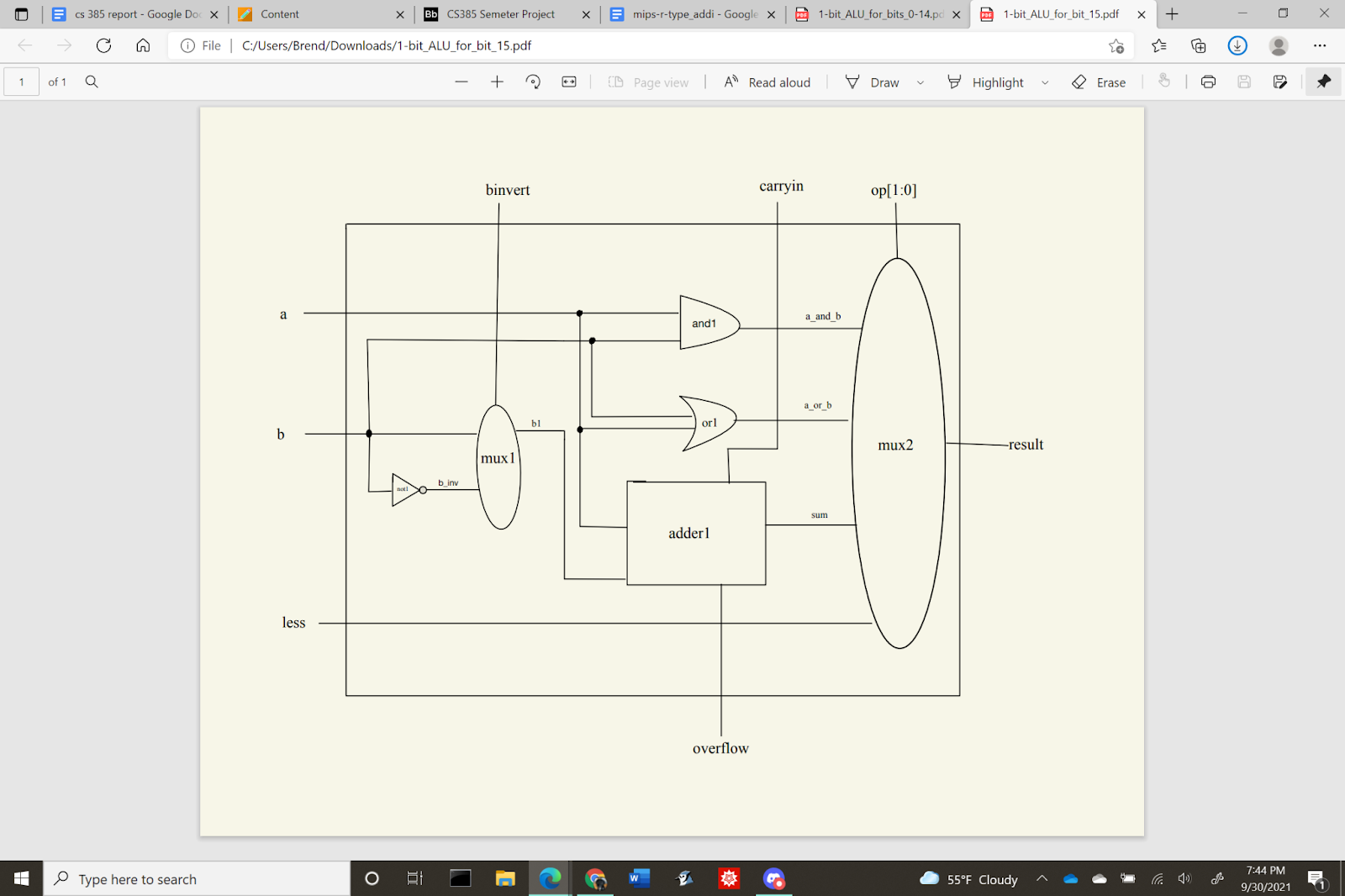
| Instruction (R-Type) | Opcode  (4 bits) [15-12] | Rs  (2 bits) [11-10] | Rt  (2bit) [9-8] | Rd  (2bits) [7-6] | Unused  (6 bits) [5-0] | Binary Instruction |
| --- | --- | --- | --- | --- | --- | --- |
| slt $3, $1, $2 | 0100 | 01 | 10 | 11 | 000000 | 0100011011000000 |
| sub $3, $1, $2 | 0001 | 01 | 10 | 11 | 000000 | 0001011011000000 |

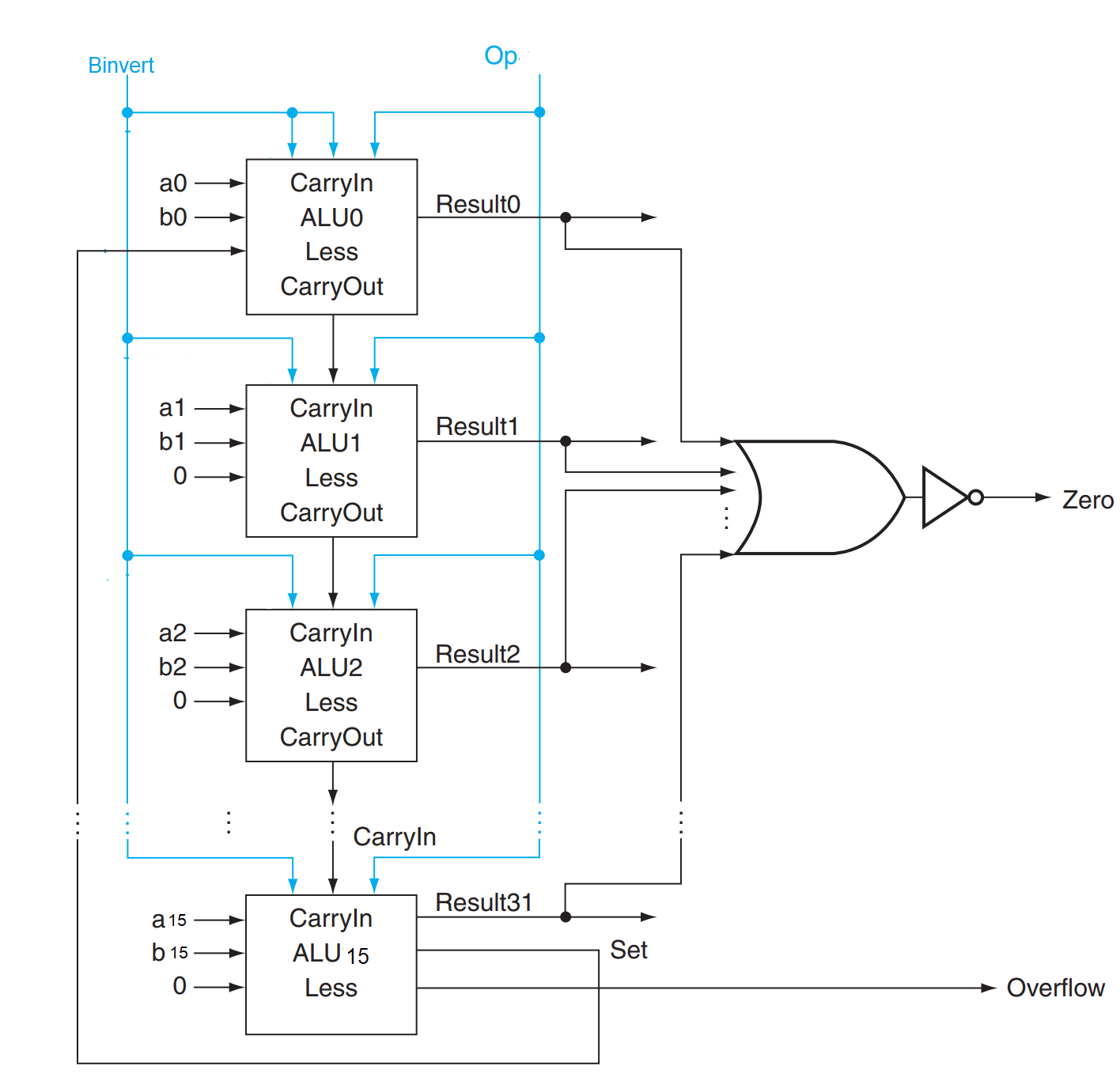
1. **Block logic diagrams**

**CPU :**

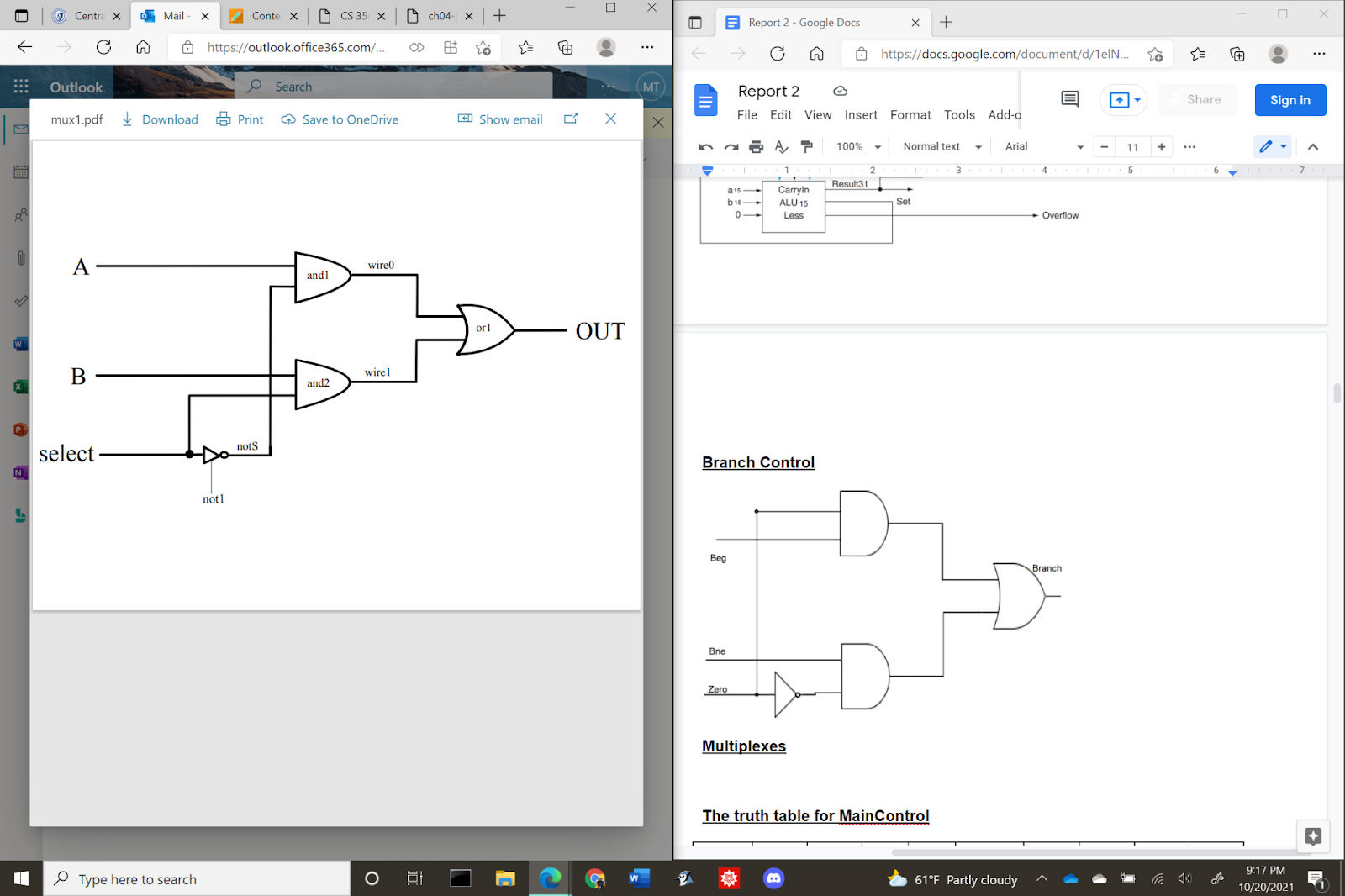
****

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

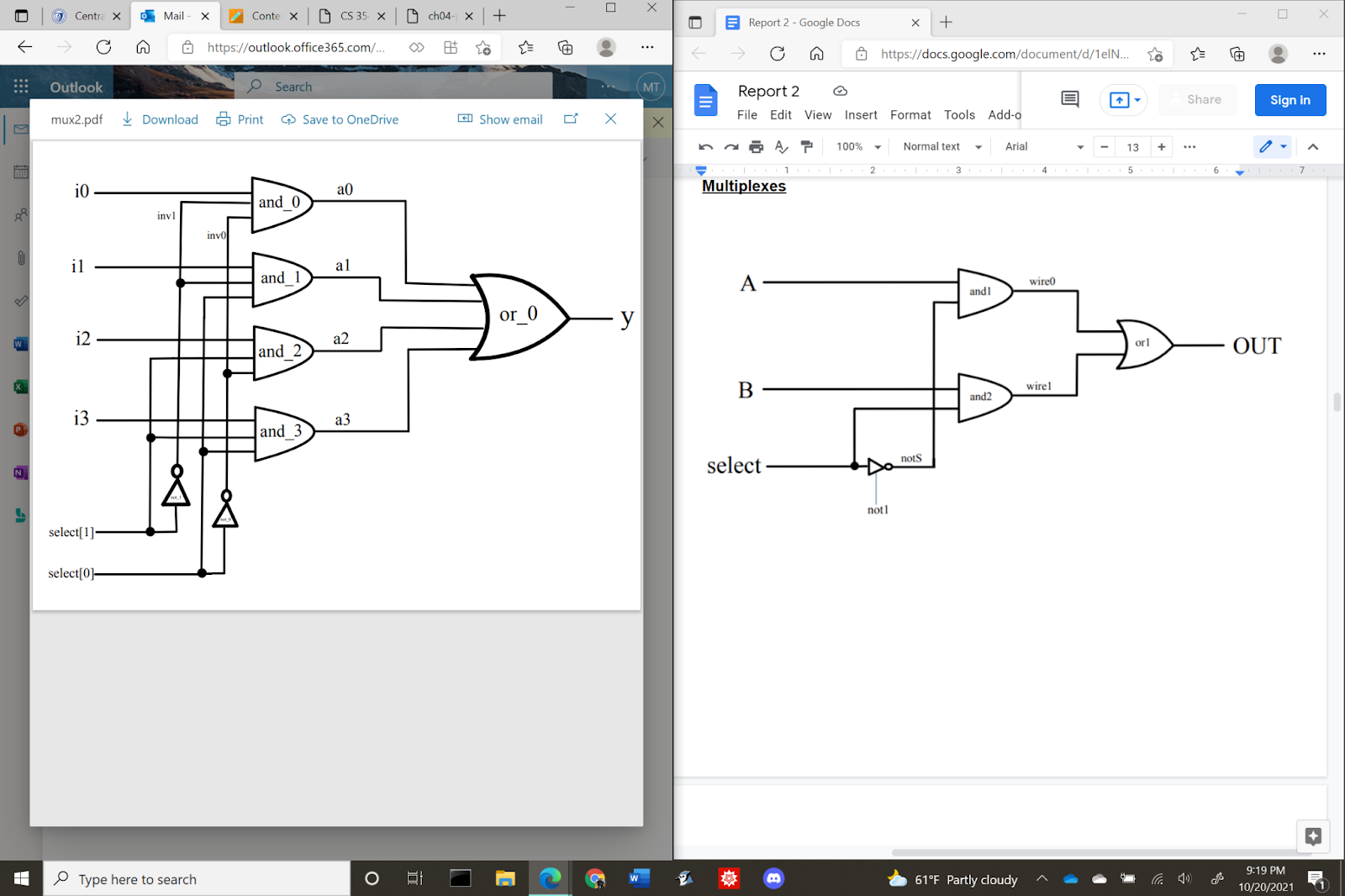
**1-bit ALU for bit 15**

**16bit-ALU** 

**2x1 Multiplexer**

****

**4x1 Multiplexer**

****

**The truth table for MainControl**

| **Instruction** | **OpCode** | **RegDst** | **Beg** | **Bne** | **MemtoReg** | **MemWrite** | **ALUSrc** | **RegWrite** | **ALUCtl** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **add** | **0000** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **010** |
| **sub** | **0001** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **110** |
| **and** | **0010** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **000** |
| **or** | **0011** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **001** |
| **slt** | **0100** | **1** | **0** | **0** | **0** | **0** | **0** | **1** | **111** |
| **addi** | **0101** | **0** | **0** | **0** | **1** | **0** | **1** | **1** | **010** |
| **slti** | **0110** | **0** | **0** | **0** | **1** | **0** | **1** | **1** | **111** |
| **andi** | **0111** | **0** | **0** | **0** | **1** | **0** | **1** | **1** | **000** |
| **ori** | **1000** | **0** | **0** | **0** | **1** | **0** | **1** | **1** | **001** |
| **lw** | **1001** | **0** | **0** | **0** | **1** | **0** | **1** | **1** | **010** |
| **sw** | **1010** | **0** | **0** | **0** | **0** | **1** | **1** | **0** | **010** |
| **beg** | **1011** | **0** | **1** | **0** | **0** | **0** | **0** | **0** | **110** |
| **bne** | **1100** | **0** | **0** | **1** | **0** | **0** | **0** | **0** | **110** |

**Verilog source code**

| // Behavioral model of MIPS - pipelined implementation  // --------------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 [9:0] ctrl;  // ctrl bits: RegDst, Beg, Bne, MemtoReg, MemWrite, ALUSrc, RegWrite, ALUCtl(3bits)  always @(op) case (op)  4'b0000: ctrl <= 10'b1000\_001\_010; // ADD  4'b0001: ctrl <= 10'b1000\_001\_110; // SUB  4'b0010: ctrl <= 10'b1000\_001\_000; // AND  4'b0011: ctrl <= 10'b1000\_001\_001; // OR  4'b0100: ctrl <= 10'b1000\_001\_111; // SLT  4'b0101: ctrl <= 10'b0001\_011\_010; // ADDI  4'b0110: ctrl <= 10'b0001\_011\_111; // SLTI  4'b0111: ctrl <= 10'b0001\_011\_000; // ANDI  4'b1000: ctrl <= 10'b0001\_011\_001; // ORI  4'b1001: ctrl <= 10'b0001\_011\_010; // LW  4'b1010: ctrl <= 10'b0001\_110\_010; // SW  4'b1011: ctrl <= 10'b0100\_000\_110; // BEG  4'b1100: ctrl <= 10'b0010\_000\_110; // BNE  endcase  endmodule  //------------END MAIN CONTROL----------------  //------------BRANCH CONTROL---------------  module BranchLogic(Zero,Beg,Bne,Branch);  input Zero,Beg,Bne;  output Branch;  wire invZero;  not not1 (invZero,Zero);  and a1 (a1,Zero,Beg),  a2 (a2,invZero,Bne);  or or1 (Branch,a1,a2);  endmodule  //------------END BRANCH CONTROL---------------  module CPU (clock,PC,IFID\_IR,IDEX\_IR,EXMEM\_IR,MEMWB\_IR,WD);  input clock;  output [15:0] PC,IFID\_IR,IDEX\_IR,EXMEM\_IR,MEMWB\_IR,WD;  initial begin  // Program: swap memory cells (if needed) and compute absolute value |5-7|=2  // // BEG WITH NO-OP --------------------------------------  // IMemory[0] = 16'b1001\_00\_01\_00000000; // lw $1, 0($0) $1 <= DM[0] = 5  // IMemory[1] = 16'b1001\_00\_10\_00000010; // lw $2, 2($0) $2 <= DM[1] = 7  // IMemory[2] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[3] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[4] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[5] = 16'b0100\_01\_10\_11\_000000; // slt $3, $1, $2 $1 < $2 => $3 =1  // IMemory[6] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[7] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[8] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[9] = 16'b1011\_11\_00\_00000101; // beq $3, $0, IMemory[15] $3 != $0 => continue to SWAP, don't JUMP  // IMemory[10] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[11] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[12] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[13] = 16'b1010\_00\_01\_00000010; // sw $1, 2($0) DM[1] <= $1 = 5  // IMemory[14] = 16'b1010\_00\_10\_00000000; // sw $2, 0($0) DM[0] <= $2 = 7  // IMemory[15] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[16] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[17] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[18] = 16'b1001\_00\_01\_00000000; // lw $1, 0($0) $1 <= DM[0] = 7  // IMemory[19] = 16'b1001\_00\_10\_00000010; // lw $2, 2($0) $2 <= DM[1] = 5  // IMemory[20] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[21] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[22] = 16'b0000\_00\_00\_00000000; // nop  // IMemory[23] = 16'b0001\_01\_10\_11\_000000; // sub $3, $1, $2 $3 = $1 -$2 = 7-5 =2  // // BNE WITH NO-OP ----------------------------------------  IMemory[0] = 16'b1001\_00\_01\_00000000; // lw $1, 0($0) $1 <= DM[0] = 5  IMemory[1] = 16'b1001\_00\_10\_00000010; // lw $2, 2($0) $2 <= DM[1] = 7  IMemory[2] = 16'b0000\_00\_00\_00000000; // nop  IMemory[3] = 16'b0000\_00\_00\_00000000; // nop  IMemory[4] = 16'b0000\_00\_00\_00000000; // nop  IMemory[5] = 16'b0100\_01\_10\_11\_000000; // slt $3, $1, $2 $1 < $2 => $3 = 1  IMemory[6] = 16'b0000\_00\_00\_00000000; // nop  IMemory[7] = 16'b0000\_00\_00\_00000000; // nop  IMemory[8] = 16'b0000\_00\_00\_00000000; // nop  IMemory[9] = 16'b1100\_11\_00\_00000101; // bne $3, $0, IMemory[15] $3 != $0 => continue to JUMP, skip swap  IMemory[10] = 16'b0000\_00\_00\_00000000; // nop  IMemory[11] = 16'b0000\_00\_00\_00000000; // nop  IMemory[12] = 16'b0000\_00\_00\_00000000; // nop  IMemory[13] = 16'b1010\_00\_01\_00000010; // sw $1, 2($0)  IMemory[14] = 16'b1010\_00\_10\_00000000; // sw $2, 0($0)  IMemory[15] = 16'b0000\_00\_00\_00000000; // nop  IMemory[16] = 16'b0000\_00\_00\_00000000; // nop  IMemory[17] = 16'b0000\_00\_00\_00000000; // nop  IMemory[18] = 16'b1001\_00\_01\_00000000; // lw $1, 0($0) $1 <= DM[0] = 5  IMemory[19] = 16'b1001\_00\_10\_00000010; // lw $2, 2($0) $2 <= DM[1] = 7  IMemory[20] = 16'b0000\_00\_00\_00000000; // nop  IMemory[21] = 16'b0000\_00\_00\_00000000; // nop  IMemory[22] = 16'b0000\_00\_00\_00000000; // nop  IMemory[23] = 16'b0001\_01\_10\_11\_000000; // sub $3, $1, $2 $3 = $1 -$2 = 5-7 = -2  // // BEG WITHOUT NO-OP --------------------------------------  // IMemory[0] = 16'b1001\_00\_01\_00000000; // lw $1, 0($0) $1 <= DM[0] = 5  // IMemory[1] = 16'b1001\_00\_10\_00000010; // lw $2, 2($0) $2 <= DM[1] = 7  // IMemory[2] = 16'b0100\_01\_10\_11\_000000; // slt $3, $1, $2 $1 < $2 => $3 =1  // IMemory[3] = 16'b1011\_11\_00\_0000010; // beq $3, $0, IMemory[6] $3 != $0 => continue to SWAP  // IMemory[4] = 16'b1010\_00\_01\_00000010; // sw $1, 2($0) DM[1] <= $1 = 5  // IMemory[5] = 16'b1010\_00\_10\_00000000; // sw $2, 0($0) DM[0] <= $2 = 7  // IMemory[6] = 16'b1001\_00\_01\_00000000; // lw $1, 0($0) $1 <= DM[0] = 7  // IMemory[7] = 16'b1001\_00\_10\_00000010; // lw $2, 2($0) $2 <= DM[1] = 5  // IMemory[8] = 16'b0001\_01\_10\_11\_000000; // sub $3, $1, $2 $3 = $1 -$2 = 7-5 = 2  // // BNE WITHOUT NO-OP ----------------------------------------  // IMemory[0] = 16'b1001\_00\_01\_00000000; // lw $1, 0($0) $1 <= DM[0] = 5  // IMemory[1] = 16'b1001\_00\_10\_00000010; // lw $2, 2($0) $2 <= DM[1] = 7  // IMemory[2] = 16'b0100\_01\_10\_11\_000000; // slt $3, $1, $2 $1 < $2 => $3 =1  // IMemory[3] = 16'b1100\_11\_00\_00000010; // bne $3, $0, IMemory[6] $3 != $0 => continue to JUMP, skip swap  // IMemory[4] = 16'b1010\_00\_01\_00000010; // sw $1, 2($0)  // IMemory[5] = 16'b1010\_00\_10\_00000000; // sw $2, 0($0)  // IMemory[6] = 16'b1001\_00\_01\_00000000; // lw $1, 0($0) $1 <= DM[0] = 5  // IMemory[7] = 16'b1001\_00\_10\_00000010; // lw $2, 2($0) $2 <= DM[1] = 7  // IMemory[8] = 16'b0001\_01\_10\_11\_000000; // sub $3, $1, $2 $3 = $1 -$2 = 5-7 = -2  // Data  DMemory [0] = 16'b0101; // switch the cells and see how the simulation output changes  DMemory [1] = 16'b0111; // (beq is taken if [0]=32'h7; [1]=32'h5, not taken otherwise)  end  // Pipeline  // IF  wire [15:0] PCplus2, NextPC;  reg [15:0] EXMEM\_Target;  wire Branch;  reg[15:0] PC, IMemory[0:1023], IFID\_IR, IFID\_PCplus2;  alu fetch (3'b010,PC,16'b10,PCplus2,Unused1);  mux16bitx2x1 muxBranch (PCplus2,EXMEM\_Target,Branch,NextPC);  // ID  wire [9:0] Control;  reg IDEX\_RegWrite,IDEX\_MemtoReg,  IDEX\_Beq, IDEX\_Bne, IDEX\_MemWrite,  IDEX\_ALUSrc, IDEX\_RegDst;  reg [2:0] IDEX\_ALUctl;  wire [15:0] RD1,RD2,SignExtend, WD;  reg [15:0] IDEX\_PCplus2,IDEX\_RD1,IDEX\_RD2,IDEX\_SignExt,IDEXE\_IR;  reg [15:0] IDEX\_IR; // For monitoring the pipeline  reg [1:0] IDEX\_rt,IDEX\_rd;  reg MEMWB\_RegWrite; // part of MEM stage, but declared here before use (to avoid error)  reg [1:0] MEMWB\_rd; // part of MEM stage, but declared here before use (to avoid error)  reg\_file rf (IFID\_IR[11:10],IFID\_IR[9:8],MEMWB\_rd,WD,MEMWB\_RegWrite,RD1,RD2,clock);  mainCtrl MainCtr (IFID\_IR[15:12],Control);  assign SignExtend = {{8{IFID\_IR[7]}},IFID\_IR[7:0]};    // EXE  reg EXMEM\_RegWrite,EXMEM\_MemtoReg,  EXMEM\_Beq, EXMEM\_Bne, EXMEM\_MemWrite;  wire [15:0] Target;  reg EXMEM\_Zero;  reg [15:0] EXMEM\_ALUOut,EXMEM\_RD2;  reg [15:0] EXMEM\_IR; // For monitoring the pipeline  reg [1:0] EXMEM\_rd;  wire [15:0] B,ALUOut;  wire [1:0] WR;  alu branch (3'b010,IDEX\_SignExt<<1,IDEX\_PCplus2,Target,Unused2);  alu ex (IDEX\_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  // MEM  reg MEMWB\_MemtoReg;  reg [15:0] DMemory[0:1023],MEMWB\_MemOut,MEMWB\_ALUOut;  reg [15:0] MEMWB\_IR; // For monitoring the pipeline  wire [15:0] MemOut;    assign MemOut = DMemory[EXMEM\_ALUOut>>1];  always @(negedge clock) if (EXMEM\_MemWrite) DMemory[EXMEM\_ALUOut>>1] <= EXMEM\_RD2;  BranchLogic bra (EXMEM\_Zero,EXMEM\_Beq,EXMEM\_Bne,Branch); // notice  // WB  mux16bitx2x1 muxWD (MEMWB\_ALUOut, MEMWB\_MemOut, MEMWB\_MemtoReg, WD); // MemtoReg Mux  //assign WD = (MEMWB\_MemtoReg) ? MEMWB\_MemOut: MEMWB\_ALUOut; // MemtoReg Mux  initial begin  PC = 0;  // Initialize pipeline registers  IDEX\_RegWrite=0;IDEX\_MemtoReg=0;IDEX\_Beq=0;IDEX\_Bne=0;IDEX\_MemWrite=0;IDEX\_ALUSrc=0;IDEX\_RegDst=0;//IDEX\_ALUOp=0;  IFID\_IR=0;  EXMEM\_RegWrite=0;EXMEM\_MemtoReg=0;EXMEM\_Beq=0;EXMEM\_Bne=0;EXMEM\_MemWrite=0;  EXMEM\_Target=0;  MEMWB\_RegWrite=0;MEMWB\_MemtoReg=0;  end  // Running the pipeline  always @(negedge clock) begin  // IF  PC <= NextPC;  IFID\_PCplus2 <= PCplus2;  IFID\_IR <= IMemory[PC>>1];  // ID  IDEX\_IR <= IFID\_IR; // For monitoring the pipeline  {IDEX\_RegDst,IDEX\_Beq,IDEX\_Bne,IDEX\_MemtoReg,IDEX\_MemWrite,IDEX\_ALUSrc,IDEX\_RegWrite,IDEX\_ALUctl} <= Control;  IDEX\_PCplus2 <= IFID\_PCplus2;  IDEX\_RD1 <= RD1;  IDEX\_RD2 <= RD2;  IDEX\_SignExt <= SignExtend;  IDEX\_rt <= IFID\_IR[9:8];// change from 11:10  IDEX\_rd <= IFID\_IR[7:6];// change from 9:8  // EXE  EXMEM\_IR <= IDEX\_IR; // For monitoring the pipeline  EXMEM\_RegWrite <= IDEX\_RegWrite;  EXMEM\_MemtoReg <= IDEX\_MemtoReg;  EXMEM\_Beq <= IDEX\_Beq;  EXMEM\_Bne <= IDEX\_Bne;  EXMEM\_MemWrite <= IDEX\_MemWrite;  EXMEM\_Target <= Target;  EXMEM\_Zero <= Zero;  EXMEM\_ALUOut <= ALUOut;  EXMEM\_RD2 <= IDEX\_RD2;  EXMEM\_rd <= WR;  // MEM  MEMWB\_IR <= EXMEM\_IR; // For monitoring the pipeline  MEMWB\_RegWrite <= EXMEM\_RegWrite;  MEMWB\_MemtoReg <= EXMEM\_MemtoReg;  MEMWB\_MemOut <= MemOut;  MEMWB\_ALUOut <= EXMEM\_ALUOut;  MEMWB\_rd <= EXMEM\_rd;  // WB  // Register write happens on neg edge of the clock (if MEMWB\_RegWrite is asserted)  end  endmodule  // Test module  module test ();  reg clock;  wire [15:0] PC,IFID\_IR,IDEX\_IR,EXMEM\_IR,MEMWB\_IR,WD;  CPU test\_cpu(clock,PC,IFID\_IR,IDEX\_IR,EXMEM\_IR,MEMWB\_IR,WD);  always #1 clock = ~clock;    initial begin  $display ("time PC IFID\_IR IDEX\_IR EXMEM\_IR MEMWB\_IR WD");  $monitor ("%2d %3d %b %b %b %b %d", $time,PC,IFID\_IR,IDEX\_IR,EXMEM\_IR,MEMWB\_IR,WD);  clock = 1;  #56 $finish;  end  endmodule  /\* Output:  time PC IFID\_IR IDEX\_IR EXMEM\_IR MEMWB\_IR WD  0 0 00000000 xxxxxxxx xxxxxxxx xxxxxxxx x  1 4 8c010000 00000000 xxxxxxxx xxxxxxxx x  3 8 8c020004 8c010000 00000000 xxxxxxxx x  5 12 00000000 8c020004 8c010000 00000000 0  7 16 00000000 00000000 8c020004 8c010000 5  9 20 00000000 00000000 00000000 8c020004 7  11 24 0022182a 00000000 00000000 00000000 0  13 28 00000000 0022182a 00000000 00000000 0  15 32 00000000 00000000 0022182a 00000000 0  17 36 00000000 00000000 00000000 0022182a 1  19 40 10600005 00000000 00000000 00000000 0  21 44 00000000 10600005 00000000 00000000 0  23 48 00000000 00000000 10600005 00000000 0  25 52 00000000 00000000 00000000 10600005 1  27 56 ac010004 00000000 00000000 00000000 0  29 60 ac020000 ac010004 00000000 00000000 0  31 64 00000000 ac020000 ac010004 00000000 0  33 68 00000000 00000000 ac020000 ac010004 4  35 72 00000000 00000000 00000000 ac020000 0  37 76 8c010000 00000000 00000000 00000000 0  39 80 8c020004 8c010000 00000000 00000000 0  41 84 00000000 8c020004 8c010000 00000000 0  43 88 00000000 00000000 8c020004 8c010000 7  45 92 00000000 00000000 00000000 8c020004 5  47 96 00221822 00000000 00000000 00000000 0  49 100 xxxxxxxx 00221822 00000000 00000000 0  51 104 xxxxxxxx xxxxxxxx 00221822 00000000 0  53 108 xxxxxxxx xxxxxxxx xxxxxxxx 00221822 2  55 112 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx X  \*/ |
| --- |

1. **Test result**

***Sample Test 1 : DMemory[0] = 5 DMemory[1] = 7 with Beg instruction***

***lw $1, 0($0) => $1 = 5***

***lw $2, 2($0)=> $2 = 7***

***slt $3, $1, $2=> $3 =1***

***beg $3, $0, IMemory[6]=> branch does not take***

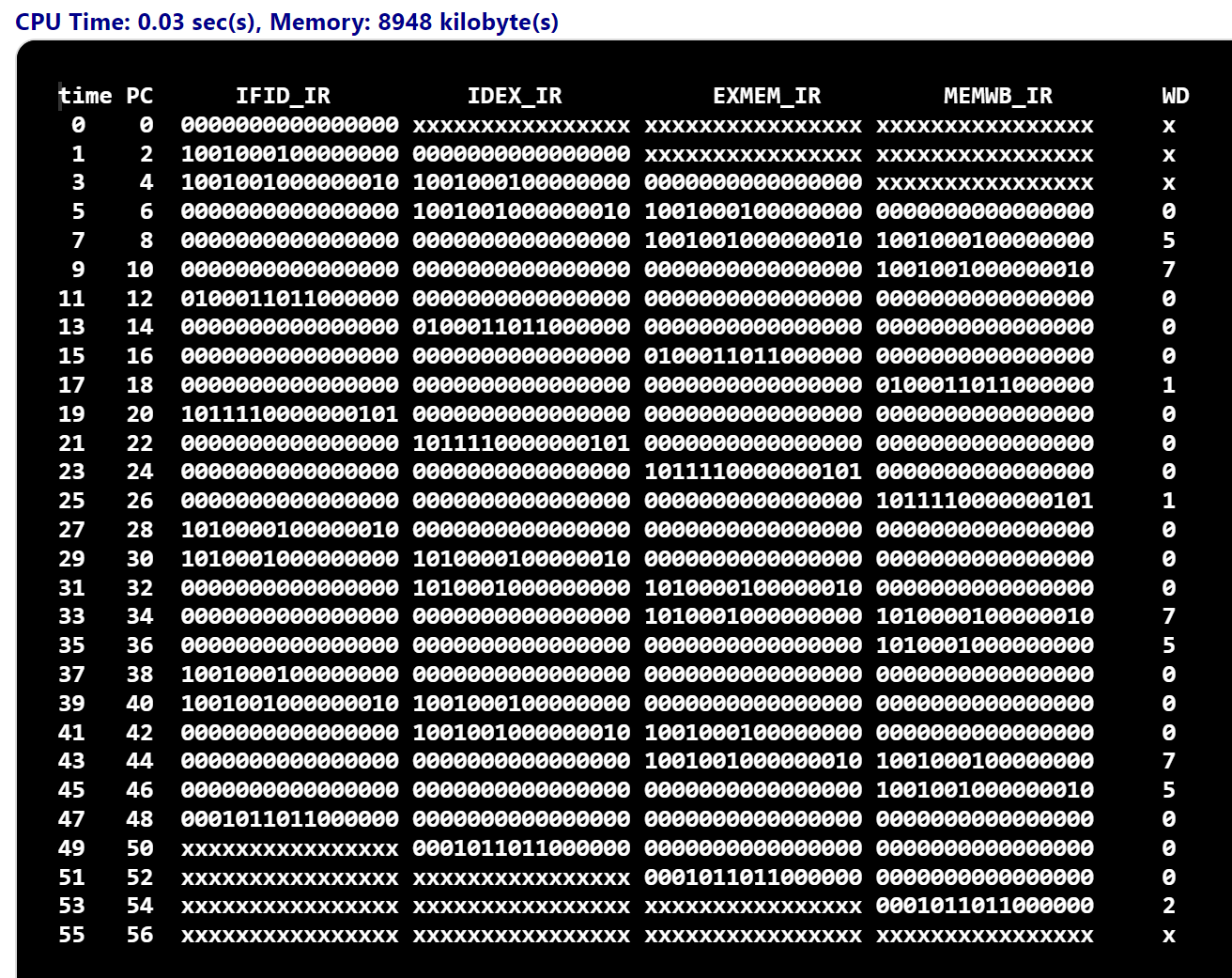
***sw $1, 2($0) => 2($0)=7***

***sw $2, 0($0) => 0($0)=5***

***lw $1, 0($0) => $1=5***

***lw $2, 2($0) => $2=7***

***sub $3, $1, $2 => $3 = 7-5 = 2 in binary 0000\_0000\_0000\_0010***



***Sample Test 2 : DMemory[0] = 7 DMemory[1] = 5 with Beg instruction***

***lw $1, 0($0) => $1 = 7***

***lw $2, 2($0)=> $2 = 5***

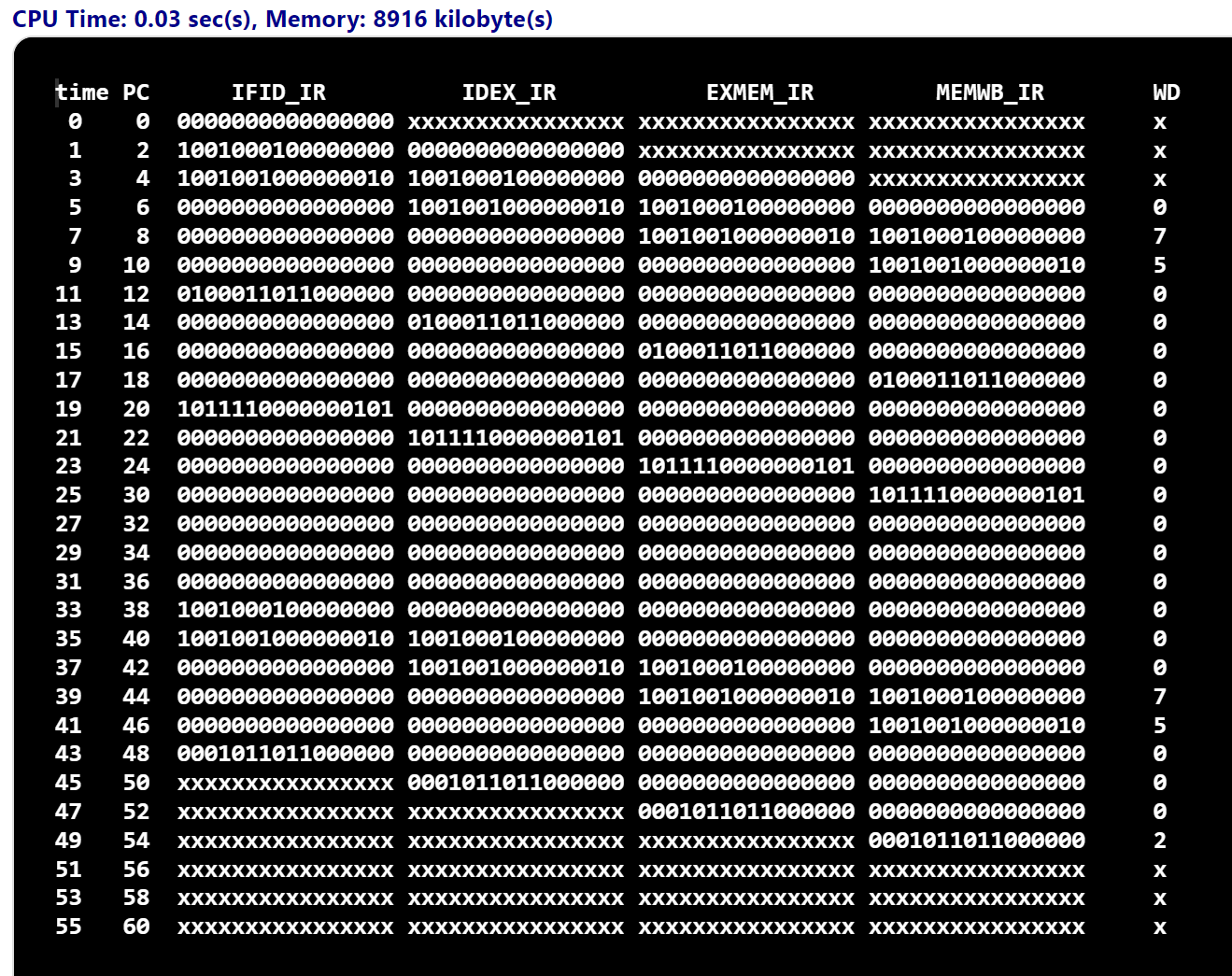
***slt $3, $1, $2=> $3 =0***

***beg $3, $0, IMemory[6]=> branch taken***

***lw $1, 0($0) => $1=7***

***lw $2, 2($0) => $2=5***

***sub $3, $1, $2 => $3 = 7-5 = 2 in binary 0000\_0000\_0000\_0010***



***Sample Test 3 : DMemory[0] = 5 DMemory[1] = 7 with Bne instruction***

***lw $1, 0($0) => $1 = 5***

***lw $2, 2($0)=> $2 = 7***

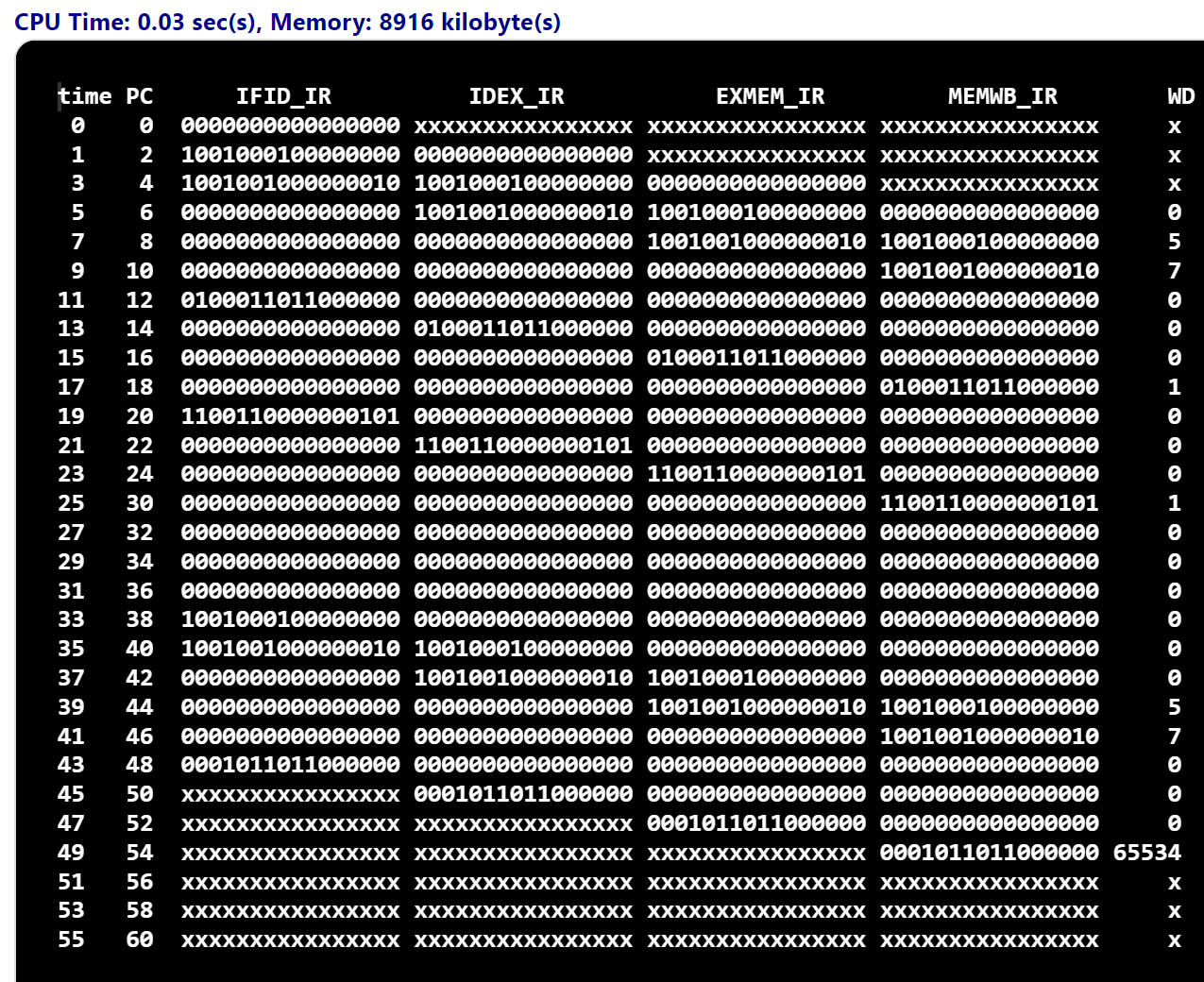
***slt $3, $1, $2=> $3 =1***

***bne $3, $0, IMemory[6]=> branch taken***

***lw $1, 0($0) => $1=5***

***lw $2, 2($0) => $2=7***

***sub $3, $1, $2 => $3 = 5-7 = -2 in binary 1111\_1111\_1111\_1110***



***Sample Test 4 : DMemory[0] = 7 DMemory[1] = 5 with Bne instruction***

***lw $1, 0($0) => $1 = 7***

***lw $2, 2($0)=> $2 = 5***

***slt $3, $1, $2=> $3 =0***

***bne $3, $0, IMemory[6]=> branch does not take***

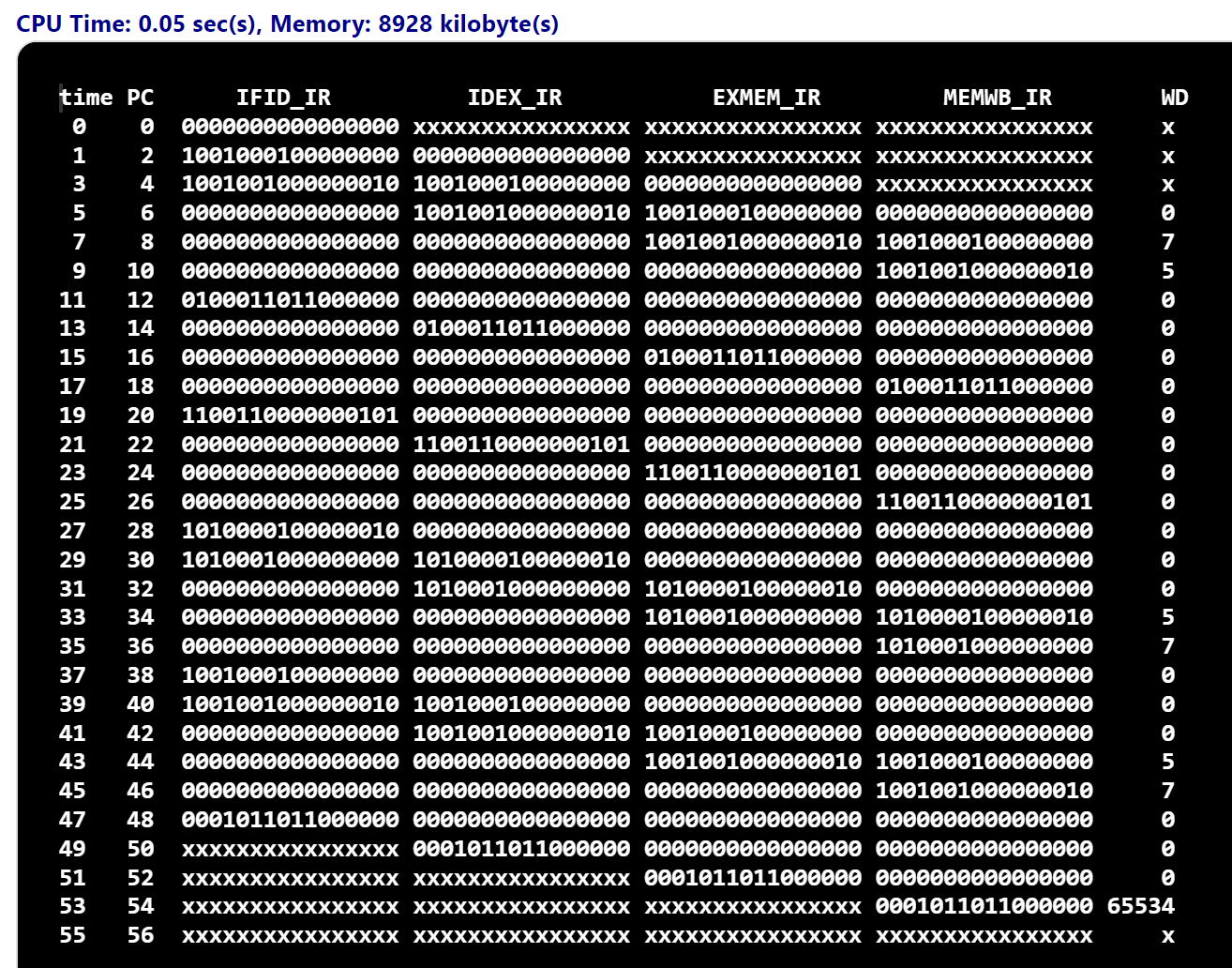
***sw $1, 2($0) => 2($0)=7***

***sw $2, 0($0) => 0($0)=5***

***lw $1, 0($0) => $1=5***

***lw $2, 2($0) => $2=7***

***sub $3, $1, $2 => $3 = 5-7 = -2 in binary 1111\_1111\_1111\_1110***



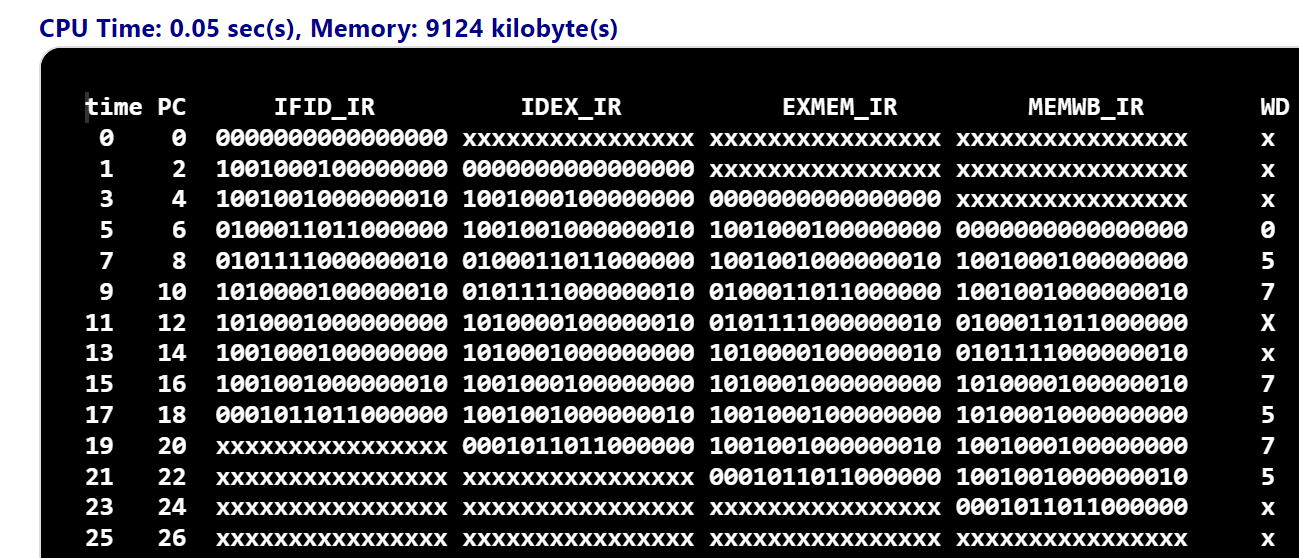
***Sample Test 5 : DMemory[0] = 5 DMemory[1] = 7 with Beg instruction without noop***

There are two hazards happen in this sample test. Therefore, CPU cannot compute the value for two instruction slt $3, $1, $2 and sub $3,$1,$2

| ***IFID\_IR*** | ***IDEX\_IR*** | ***EXMEM\_IR*** | ***MEMWB\_IR*** |
| --- | --- | --- | --- |
| beg $3,$0, IMemory[6] | slt $3, $1, $2 | lw $2, 2($0) | lw $1 , 0($0) |
|  | sub $3, $1, $2 | lw $2, 2($0) | lw $1, 0($0) |

The instruction **slt $3,$1,$2** need value from register **$1**,but  **$1** do not store value. Then, CPU cannot compute this instruction.

Similarly, the instruction **sub $3, $1, $2** need value of registers $1, $2 from the previous instructions : lw $2, 2($0) and lw $1 , 0($0). However, the instructions is not done.

******

***Sample Test 6 : DMemory[0] = 5 DMemory[1] = 7 with Bne instruction without noop***

Like the sample test 5. There are two hazards happen in this sample test. Therefore, CPU cannot compute the value for two instruction slt $3, $1, $2 and sub $3,$1,$2

