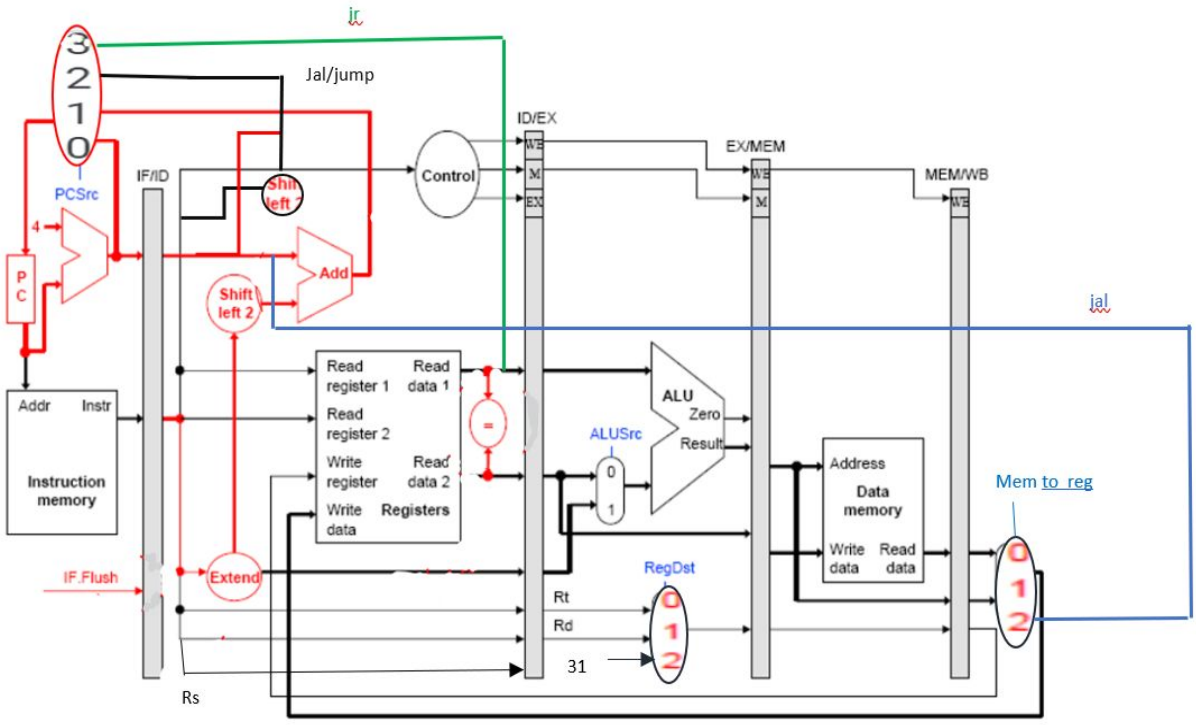
Computer Assignment #4 (Pipeline MIPS Processor)

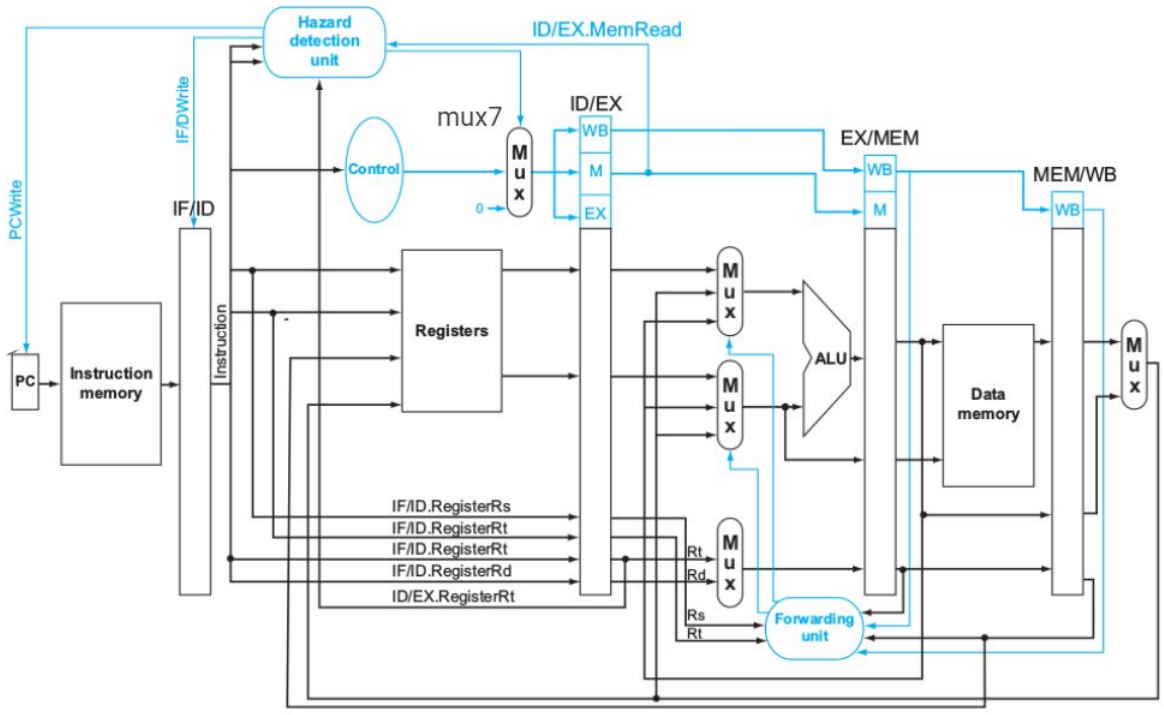
Helia Hosseini (810197491)
Daneshvar Amrollahi (810197685)

Datapath:

Block Diagram:



Datapath without forwarding unit and hazard detection unit



Datapath with Forwarding Unit and Hazard Detection Unit

Data Hazards caused by a LW (Load Word) instruction and also those caused between R-Type instructions are handled with the Hazard Unit in the datapath and a Forwarding Unit.

Verilog Code:

```
module datapath (clk,
                 rst,
                 inst_adr,
                 inst,
                 data_adr,
                 data_out,
                 data_in,
                 reg_dst,
                 mem_to_reg,
                 alu_src,
                 pc_src, //coming from controller
                 alu_ctrl,
                 reg_write,
```

```
                flush,
                mem_read,
                mem_write,
                forwardA,
                forwardB,
                mem_write_to_data_mem,
                mem_read_to_data_mem,
                pc_load, //coming from hazard detection unit
                IFID_Ld, //coming from hazard detection unit
                sel_signal, //coming from hazard detection unit
                IFIDopcode_out, //output to controller
                IFIDfunc_out, //output to controller
                zero_out, //output to controller (not used)
                operands_equal, //output to controller
                IDEX_mem_read,
                IDEX_Rt,
                IFID_Rt,
                IFID_Rs,
                IDEX_Rs,
                EXMEM_reg_write, EXMEM_Rd, MEMWB_reg_write, MEMWB_Rd
                );

   input  clk, rst;
   output [31:0] inst_adr; //going to istruction memory
   input  [31:0] inst; //coming from instruction memory to IF/ID
   output [31:0] data_adr; //from EX/MEM.alu_result to 'address' port of
DataMemory
   output [31:0] data_out; //from EX/MEM.mux3_out to 'write data' port of
DataMemory
   input  [31:0] data_in; //from DataMemory to MEM/WB
   input [1:0] mem_to_reg;
   input alu_src, reg_write;
   input [1:0] pc_src;
   input  [2:0] alu_ctrl;
   input flush;
   input mem_read,mem_write;
   input [1:0] forwardA, forwardB;
   input pc_load;
   input IFID_Ld;
   input sel_signal;
   output [5:0] IFIDopcode_out;
   output [5:0] IFIDfunc_out;
   output zero_out;
   output operands_equal;
   output IDEX_mem_read;
```

```
   output [4:0] IDEX_Rt, IFID_Rt, IFID_Rs, IDEX_Rs;
   output mem_read_to_data_mem, mem_write_to_data_mem;
   output EXMEM_reg_write, MEMWB_reg_write;
   output [4:0] EXMEM_Rd, MEMWB_Rd;

   input [1:0] reg_dst;

   ////////////////////////////////////////////////
   //Instruction Fetch
   wire [31:0] mux1_out;
   wire [31:0] pc_out;
   reg_32b PC(mux1_out, rst, pc_load, clk, pc_out);

   wire [31:0] adder1_out;
   wire cout1;
   adder_32b ADDER_1(pc_out , 32'd4, 1'b0, cout1 , adder1_out);
   assign inst_adr = pc_out; //output to inst. memory

   wire [31:0] IFIDinst_out, IFIDadder1_out;
   IFID IFIDReg(clk, rst, IFID_Ld, flush, inst, adder1_out, IFIDinst_out,
IFIDadder1_out);

   assign IFIDopcode_out = IFIDinst_out[31:26];
   assign IFIDfunc_out = IFIDinst_out[5:0];
   assign IFID_Rt = IFIDinst_out[20:16];
   assign IFID_Rs = IFIDinst_out[25:21];

   wire [31:0] adder2_out;
   wire [27:0] shl2_26b_out;
   wire [31:0] read_data1;
   mux4to1_32b MUX1(adder1_out, adder2_out, {IFIDadder1_out[31:28],
shl2_26b_out}, read_data1, pc_src, mux1_out);

   ////////////////////////////////////////////////////


   ////////////////////////////////////////////////////
   //Instruction Decode


   wire [31:0] sgn_ext_out;
   sign_ext SGN_EXT(IFIDinst_out[15:0], sgn_ext_out);
```

```
   wire [31:0] shl2_32_out;
   shl2_32b SHL2_32(sgn_ext_out, shl2_32_out);

   wire cout2;
   adder_32b ADDER_2(shl2_32_out, IFIDadder1_out, 1'b0, cout2, adder2_out);

   shl2_26b SHL2_26(IFIDinst_out[25:0], shl2_26b_out);

   wire [31:0] mux6_out;
   wire [4:0] MEMWB_mux5_out;
   wire [31:0] read_data2;
   wire MEMWB_reg_write_out;
   reg_file RF(mux6_out, IFIDinst_out[25:21], IFIDinst_out[20:16],
MEMWB_mux5_out, MEMWB_reg_write_out, rst, clk, read_data1, read_data2);

   assign operands_equal = (read_data1 == read_data2);
   ////////////////////////////////////////////////////


   ////////////////////////////////////////////////////
   //EX

   wire [31:0] IDEX_read1_out, IDEX_read2_out, IDEX_sgn_ext_out;
   wire [4:0] IDEX_Rt_out, IDEX_Rd_out, IDEX_Rs_out;
   wire [31:0] IDEX_adder1_out;
   IDEX_datas IDEX_DATAS(clk, rst, read_data1, read_data2, sgn_ext_out,
IFIDinst_out[20:16], IFIDinst_out[15:11], IFIDinst_out[25:21], IFIDadder1_out,
       IDEX_read1_out, IDEX_read2_out, IDEX_sgn_ext_out, IDEX_Rt_out,
IDEX_Rd_out, IDEX_Rs_out, IDEX_adder1_out);

   assign IDEX_Rt = IDEX_Rt_out;
   assign IDEX_Rs = IDEX_Rs_out;

   wire [2:0] IDEX_alu_ctrl_out;
   wire IDEX_reg_write_out;
   wire [1:0] IDEX_reg_dst_out;
   wire IDEX_mem_read_out, IDEX_mem_write_out;
   wire IDEX_alu_src_out;
   wire [1:0] IDEX_mem_to_reg_out;


   wire [10:0] mux7_out;
   mux2to1_11b MUX7(11'b0, {alu_ctrl, alu_src, reg_write, reg_dst, mem_read,
mem_write, mem_to_reg}, sel_signal, mux7_out);
```

```
   wire [2:0] IDEX_alu_ctrl_in;
   wire IDEX_alu_src_in;
   wire IDEX_reg_write_in;
   wire [1:0] IDEX_reg_dst_in;
   wire IDEX_mem_read_in;
   wire IDEX_mem_write_in;
   wire [1:0] IDEX_mem_to_reg_in;
   assign {IDEX_alu_ctrl_in, IDEX_alu_src_in, IDEX_reg_write_in,
IDEX_reg_dst_in, IDEX_mem_read_in, IDEX_mem_write_in, IDEX_mem_to_reg_in} =
mux7_out;
   IDEX_ctrl IDEX_CTRL(clk, rst, IDEX_alu_ctrl_in, IDEX_alu_src_in,
IDEX_reg_write_in, IDEX_reg_dst_in, IDEX_mem_read_in, IDEX_mem_write_in,
IDEX_mem_to_reg_in,
               IDEX_alu_ctrl_out, IDEX_alu_src_out, IDEX_reg_write_out,
IDEX_reg_dst_out, IDEX_mem_read_out, IDEX_mem_write_out, IDEX_mem_to_reg_out);

   assign IDEX_mem_read = IDEX_mem_read_out;

   wire [4:0] mux5_out;
   mux3to1_5b MUX5(IDEX_Rt_out, IDEX_Rd_out, 5'b11111, IDEX_reg_dst_out,
mux5_out);

   wire [31:0] mux3_out, mux4_out;
   mux2to1_32b MUX4(mux3_out, IDEX_sgn_ext_out, IDEX_alu_src_out, mux4_out);


   wire [31:0] EXMEM_alu_result_out;

   mux3to1_32b MUX3(IDEX_read2_out, mux6_out, EXMEM_alu_result_out, forwardB,
mux3_out);

   wire [31:0] mux2_out;
   mux3to1_32b MUX2(IDEX_read1_out, mux6_out, EXMEM_alu_result_out, forwardA,
mux2_out);

   wire [31:0] alu_result;
   wire alu_zero;
   alu ALU(mux2_out, mux4_out, IDEX_alu_ctrl_out, alu_result, alu_zero);

   assign zero_out = alu_zero;
   //////////////////////////////////////////////////
```

```
   ////////////////////////////////////////////////
   //MEM

   wire [31:0] EXMEM_adder1_out;
   wire EXMEM_zero_out;
   wire [31:0] EXMEM_mux3_out;
   wire [4:0] EXMEM_mux5_out;
   EXMEM_datas EXMEM_DATAS(clk, rst, IDEX_adder1_out, alu_zero, alu_result,
mux3_out, mux5_out,
           EXMEM_adder1_out, EXMEM_zero_out, EXMEM_alu_result_out,
EXMEM_mux3_out, EXMEM_mux5_out);


   assign EXMEM_Rd = EXMEM_mux5_out;

   wire EXMEM_mem_write_out, EXMEM_mem_read_out, EXMEM_reg_write_out;
   wire [1:0] EXMEM_mem_to_reg_out;
   EXMEM_ctrl EXMEM_CTRL(clk, rst, IDEX_mem_write_out, IDEX_mem_read_out,
IDEX_mem_to_reg_out, IDEX_reg_write_out,
               EXMEM_mem_write_out, EXMEM_mem_read_out, EXMEM_mem_to_reg_out,
EXMEM_reg_write_out);

   assign mem_write_to_data_mem = EXMEM_mem_write_out;
   assign mem_read_to_data_mem = EXMEM_mem_read_out;



   assign EXMEM_reg_write = EXMEM_reg_write_out;

   ////////////////////////////////////////////////




   ////////////////////////////////////////////////
   //WB

   wire [1:0] MEMWB_mem_to_reg_out;
   MEMWB_ctrl MEMWB_CTRL(clk, rst, EXMEM_mem_to_reg_out, EXMEM_reg_write_out,
       MEMWB_mem_to_reg_out, MEMWB_reg_write_out);


   assign MEMWB_reg_write = MEMWB_reg_write_out;
```

```
   wire [31:0] MEMWB_data_from_memory_out;
   wire [31:0] MEMWB_alu_result_out;
  //wire [4:0] MEMWB_mux5_out;
   wire [31:0] MEMWB_adder1_out;
   MEMWB_datas MEMWB_DATAS(clk, rst, data_in, EXMEM_alu_result_out,
EXMEM_mux5_out, EXMEM_adder1_out,
               MEMWB_data_from_memory_out, MEMWB_alu_result_out,
MEMWB_mux5_out, MEMWB_adder1_out);

   assign MEMWB_Rd = MEMWB_mux5_out;

   mux3to1_32b MUX6(MEMWB_alu_result_out, MEMWB_data_from_memory_out,
MEMWB_adder1_out, MEMWB_mem_to_reg_out, mux6_out);


   assign data_adr = EXMEM_alu_result_out; //output to data memory
   assign data_out = EXMEM_mux3_out;  //output to data memory
   ////////////////////////////////////////////////////////

endmodule
```

Some Instruction Formats:

Set Less Than Immediate:

```
SLTi   Rt   Rs   data
```

Opcode	R_s	R_t	data
001010	[25:21]	[20:16]	[15:0]

Jump and Link:

```
Jal adr
```

Opcode	adr
000011	[25:0]

Jump Register:

```
Jr Rs
```

Opcode	R_s	X
000110	[25:21]	X

Jump:

```
J adr
```

Opcode	adr
000010	[25:0]

Controller

```
module controller (opcode,
                   func,
                   zero,
                   reg_dst,
                   mem_to_reg,
                   reg_write,
                   alu_src,
                   mem_read,
                   mem_write,
                   pc_src,
                   operation,
                   IFflush,
                   operands_equal
                   );

   input [5:0] opcode;
   input [5:0] func;
   input zero;
   output reg reg_write, alu_src, mem_read, mem_write;
   output reg [1:0] pc_src;
   output reg [1:0] mem_to_reg;
   output [2:0] operation;
   output reg IFflush;
   input operands_equal;

   output reg [1:0] reg_dst;

   reg [1:0] alu_op;
   reg branch;

   alu_controller ALU_CTRL(alu_op, func, operation);

   always @(opcode)
   begin
       {reg_dst, mem_to_reg, reg_write, alu_src, mem_read, mem_write, pc_src,
alu_op, IFflush} = {2'b00, 2'b00, 1'b0, 1'b0, 1'b0, 1'b0, 2'b00, 2'b00, 1'b0};
       case (opcode)
           // RType instructions
           6'b000000 : {reg_dst, reg_write, alu_op} = {2'b01, 1'b1, 2'b10};

           // Load Word (lw) instruction
```

```
            6'b100011 : {alu_src, mem_to_reg, reg_write, mem_read} = {1'b1,
2'b01, 1'b1, 1'b1};

            // Store Word (sw) instruction
            6'b101011 : {alu_src, mem_write} = 2'b11;

            // Branch on equal (beq) instruction
            6'b000100 : {pc_src, IFflush} = {1'b0, operands_equal,
operands_equal}; //operands_equal ? 2'b01 : 2'b00

            // Add immediate (addi) instruction
            6'b001001: {reg_write, alu_src} = 2'b11;

            // Jump (j) instruction
            6'b000010: {pc_src, IFflush} = {2'b10, 1'b1};

            // Jump and link (jal) instruction
            6'b000011: {reg_dst, mem_to_reg, pc_src} = {2'b10, 2'b10, 2'b10};

            // Jump Register (JR) instruction
            6'b000110: {pc_src} = {2'b11};

            // Set Less Than immediate (SLTi) instruction
            6'b001010: {alu_src, reg_dst, reg_write, alu_op, mem_to_reg} =
{1'b1, 2'b00, 1'b1, 2'b11, 2'b00};

            //NOP (No Operation)
            //6'b111111: All signals are by default set to 0
        endcase
    end


endmodule
```

Testing

The processor is tested using the following test bench:

Load #20 32bit numbers in 2's complement format from a .mem file into Data Memory and find their minimum and it's index and finally store them at address 2000 and 2004 of Data Memory.

A NOP (no operation) is added between some assembly instructions in order to prevent some data hazards.

Pseudocode:

```
min = a[0]
min_idx = 0

for (int i = 1 ; i < 20 ; i++)
      if (a[i] < mn)
      {
            min = a[i]
            min_idx = i
      }
```

The pseudocode above is converted to an equivalent set of Assembly instructions:

Assembly Instructions:

```
LW R4, 1000(R0)
ADDi R5, R0, 0
ADDi R1, R0, 1
ADDi R2, R0, 20
```

```
            NOP
            NOP
            NOP

iLoop:      Beq R1, R2, AFTER_LOOP
            ADD R3, R0, R1
            ADD R3, R3, R3
            ADD R3, R3, R3

            LW R7, 1000(R3)
            SLT R6, R7, R4

            NOP
            NOP
            NOP

            Beq R6, R0, END_LOOP
            ADD R4, R0, R7
            ADD R5, R0, R1
END_LOOP:   ADDi R1, R1, 1

            NOP
            NOP
            NOP

            J iLoop
AFTER_LOOP:SW R4, 2000(R0)
            SW R5, 2004(R0)
```

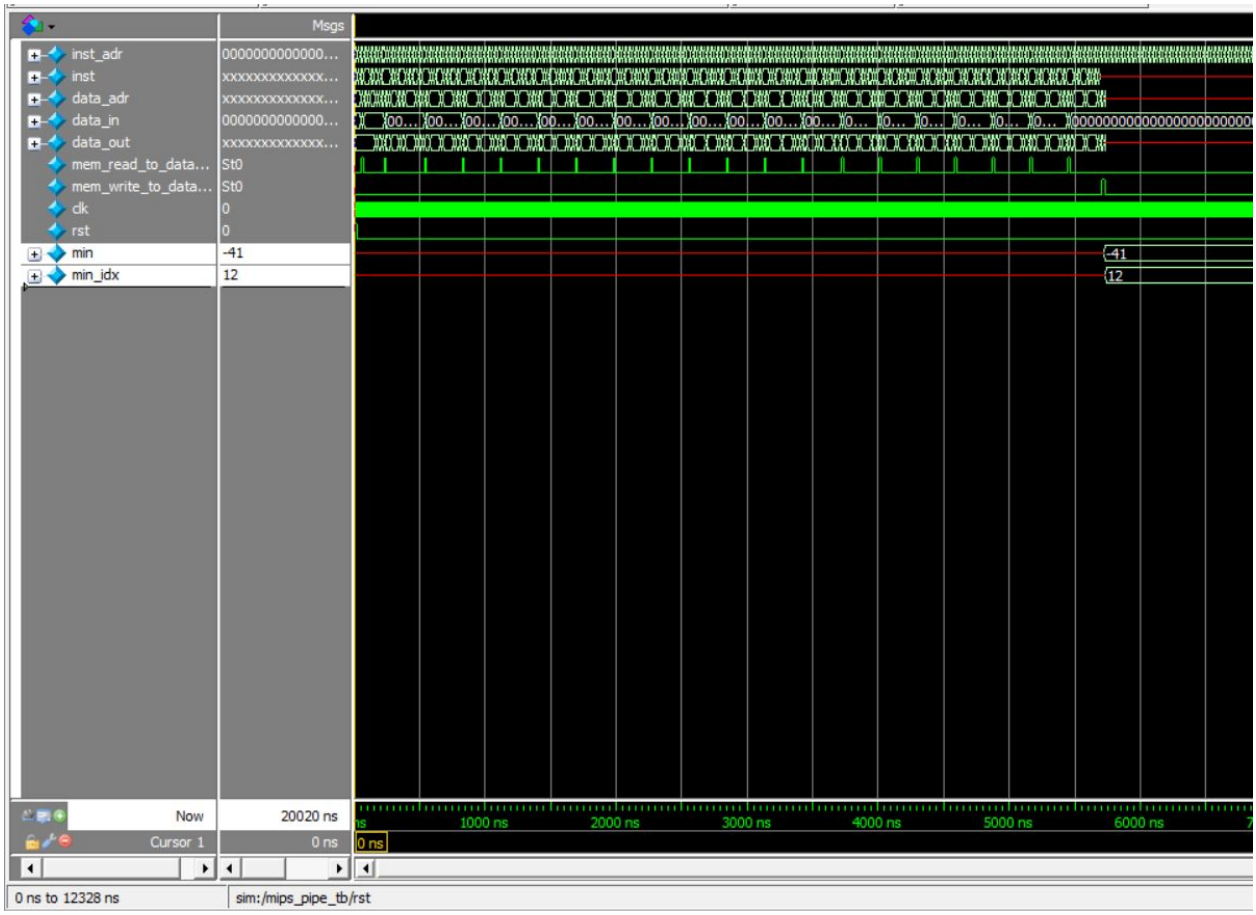
The loop variable (i) is stored in R1.
Minimum value (min) is stored in R4.
Minimum value's index (min_idx) is stored in R5.

The array of 20 random signed numbers loaded into Data Memory is as followed:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
37	27	49	94	42	28	83	113	62	109	73	82	-41	102	63	111	-27	25	68	32

The first row corresponds to the indices while the second row corresponds to the values stored in that index.

Waveforms:



As it can be seen, the minimum number in the test set is A[12] = -41 which is shown on *min* and *min_idx* in the waveforms after executing all instructions.