COMPUTER ARCHITECTURE

IMPLEMENTING RISC-V PROCESSOR

Pipelined Processing

By : Lama Imam - li06072 , Akeel Medina -am05427 , Ayeza nasir -an05918

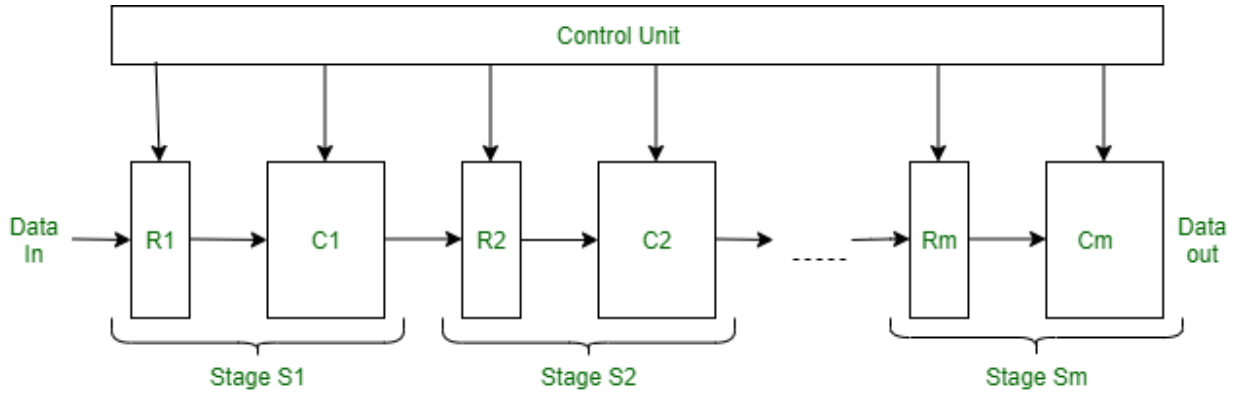


Figure - Structure of a Pipeline Processor

Introduction

Pipelining is the way toward collecting information from the processor through a pipeline. It permits putting away and executing instructions in a precise interaction. It is otherwise called pipeline processing. Pipelining is a strategy where numerous instructions are covered during execution.

Task 1:

To implement task 1 we implemented bubble sort by making alterations to our Instruction Memory Module .

Code for Task 1:

```
module Instruction_Memory
(
  input [63:0] Inst_Address,
  output reg [31:0] Instruction
);
  reg [7:0] Instruction_Memory[87:0];
  initial
    begin
      {Instruction_Memory[3], Instruction_Memory[2], Instruction_Memory[1], Instruction_Memory[0]} = 32'h00000913;//1
      {Instruction_Memory[7], Instruction_Memory[6], Instruction_Memory[5], Instruction_Memory[4]} = 32'h00000433;//2
      {Instruction_Memory[11], Instruction_Memory[10], Instruction_Memory[9], Instruction_Memory[8]} = 32'h04b40863;//3
      {Instruction_Memory[15], Instruction_Memory[14], Instruction_Memory[13], Instruction_Memory[12]} = 32'h00800eb3;//4
      {Instruction_Memory[19], Instruction_Memory[18], Instruction_Memory[17], Instruction_Memory[16]} = 32'h000409b3;//5
      {Instruction_Memory[23], Instruction_Memory[22], Instruction_Memory[21], Instruction_Memory[20]} = 32'h013989b3;//6
      {Instruction_Memory[27], Instruction_Memory[26], Instruction_Memory[25], Instruction_Memory[24]} = 32'h013989b3;//7
      {Instruction_Memory[31], Instruction_Memory[30], Instruction_Memory[29], Instruction_Memory[28]} = 32'h013989b3;//8
      {Instruction_Memory[35], Instruction_Memory[34], Instruction_Memory[33], Instruction_Memory[32]} = 32'h02be8663;//9
```

```
    {Instruction_Memory[39], Instruction_Memory[38], Instruction_Memory[37], Instruction_Memory[36]} = 32'h001e8e93;//10

    {Instruction_Memory[43], Instruction_Memory[42], Instruction_Memory[41], Instruction_Memory[40]} = 32'h00898993;//11

    {Instruction_Memory[47], Instruction_Memory[46], Instruction_Memory[45], Instruction_Memory[44]} = 32'h00092d03;//12

    {Instruction_Memory[51], Instruction_Memory[50], Instruction_Memory[49], Instruction_Memory[48]} = 32'h0009ad83;//13

    {Instruction_Memory[55], Instruction_Memory[54], Instruction_Memory[53], Instruction_Memory[52]} = 32'h01bd4463;//14

    {Instruction_Memory[59], Instruction_Memory[58], Instruction_Memory[57], Instruction_Memory[56]} = 32'hfe0004e3;//15

    {Instruction_Memory[63], Instruction_Memory[62], Instruction_Memory[61], Instruction_Memory[60]} = 32'h01a002b3;//16

    {Instruction_Memory[67], Instruction_Memory[66], Instruction_Memory[65], Instruction_Memory[64]} = 32'h01b92023;//17

    {Instruction_Memory[71], Instruction_Memory[70], Instruction_Memory[69], Instruction_Memory[68]} = 32'h0059a023;//18

    {Instruction_Memory[75], Instruction_Memory[74], Instruction_Memory[73], Instruction_Memory[72]} = 32'hfc000ce3;//19

    {Instruction_Memory[79], Instruction_Memory[78], Instruction_Memory[77], Instruction_Memory[76]} = 32'h00140413;//20

    {Instruction_Memory[83], Instruction_Memory[82], Instruction_Memory[81], Instruction_Memory[80]} = 32'h00890913;//21

    {Instruction_Memory[87], Instruction_Memory[86], Instruction_Memory[85], Instruction_Memory[84]} = 32'hfa000ae3;//22

  end


 always @ (Inst_Address)

  begin
```

```
    Instruction = {Instruction_Memory[Inst_Address[63:0]+3],
                   Instruction_Memory[Inst_Address[63:0]+2],
                   Instruction_Memory[Inst_Address[63:0]+1],
                   Instruction_Memory[Inst_Address[63:0]]};


  end
endmodule
```

Code for Branching Module in Task 1:

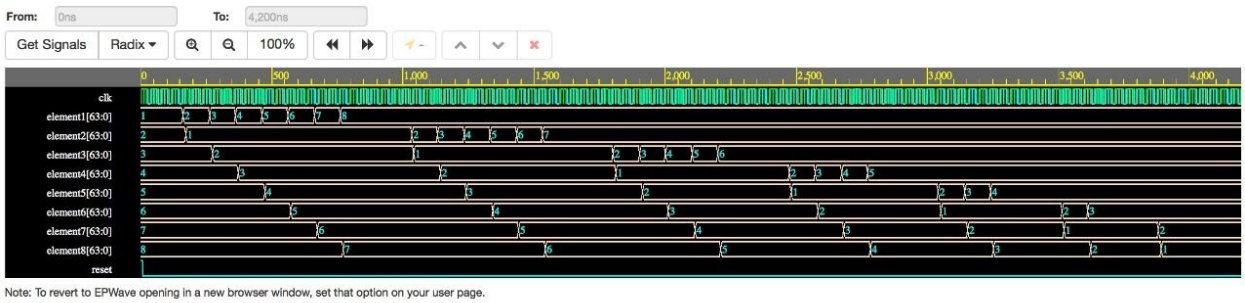
```
module branch
 (
  input [2:0] funct3,
   input [63:0] readData1,
   input [63:0] muxoutaluin,
  output reg addermuxselect
 );


 initial
   begin
    addermuxselect = 1'b0;
   end


 always @(*)
begin
    case (funct3)
      3'b000:
```

```
                begin
                  if (readData1 == muxoutaluin)
                    addermuxselect = 1'b1;
                  else
                    addermuxselect = 1'b0;
                  end
                3'b100:
                                begin
                  if (readData1 < muxoutaluin)
                    addermuxselect = 1'b1;
                  else
                    addermuxselect = 1'b0;
                  end
                3'b101:
                  begin
                  if (readData1 > muxoutaluin)
                        addermuxselect = 1'b1;
                   else
                    addermuxselect = 1'b0;
                  end
              endcase
            end
          endmodule
```

EP WAVE :



Link to our Risc-V processor performing bubble sort:

https://edaplayground.com/x/RcYe

Task 2:

Building on the module we designed in lab 11, for this task we added 4 pipeline registers, IFID, IDEX, EXMEM and MEMWB as well as two additional 3 by 1 muxes. We reconnected all modules that were previously in lab 11 to incorporate a Forwarding unit module which was also newly coded to make a 5 stage pipelined risc v processor :

Code for New modules of Task 2 :

Module ThreebyOneMux

```
module ThreebyOneMux (
  input[63:0] a,
  input[63:0] b,
```

```
  input[63:0] c,
  input [1:0] sel,
  output reg [63:0] out
);
always @(*)
  begin
    case(sel[1:0])
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
Endmodule
```

Module IFID:

```
module IFID(
  input clk,
  input reset,
  input [31:0] instruction,
  input [63:0] A, //a
  output reg [31:0] inst,//instruction out,
  output reg [63:0] a_out
);
always @(posedge clk)
  begin
```

```
    if (reset == 0)
      begin
        inst = instruction;
        a_out = A;
      end
    else
      begin
        inst = 32'b0;
        a_out = 64'b0;
      end
  end
endmodule
```

Module IDEX:

```
Module IDEX(
input clk,reset,
input [3:0] funct4_in,//funct4 of instruction from instruction memory
input [63:0] A_in,//adder input, output of IFID carried forward
input [63:0] readdata1_in, //from regwrite
input [63:0] readdata2_in,//from regwrite
input [63:0] imm_data_in,//from data extractor
input [4:0] rs1_in,//from instruction parser
input [4:0] rs2_in, //from instruction parser
input [4:0] rd_in, //from instruction parser
```

```
  input branch_in,memread_in,memtoreg_in,memwrite_in,aluSrc_in,regwrite_in, //from control unit
  input [1:0] Aluop_in,
  output reg [63:0] a,
  output reg [4:0] rs1,
  output reg [4:0] rs2,
  output reg [4:0] rd,
  output reg [63:0] imm_data,
  output reg [63:0] readdata1, //2bit mux
  output reg [63:0] readdata2, //2bit mux
  output reg [3:0] funct4_out,
  output reg Branch,Memread,Memtoreg, Memwrite, Regwrite,Alusrc,
  output reg [1:0] aluop
);

  always @ (posedge clk)
    begin
      if (reset == 1'b1)
        begin
          a = 64'b0;
          rs1 = 5'b0;
          rs2 = 5'b0;
          rd = 5'b0;
          imm_data = 64'b0;
          readdata1 = 64'b0;
          readdata2 = 64'b0;
          funct4_out = 4'b0;
```

```
        Branch = 1'b0;
        Memread = 1'b0;
        Memtoreg =1'b0;
        Memwrite = 1'b0;
        Regwrite = 1'b0;
        Alusrc = 1'b0;
        aluop = 2'b0;
      end
    else
      begin
        a = A_in;
        rs1 = rs1_in;
        rs2 = rs2_in;
        rd = rd_in;
        imm_data = imm_data_in;
        readdata1 = readdata1_in;
        readdata2 = readdata2_in;
        funct4_out = funct4_in; //when connecting in top module Funct4 is wire containing this section of 31 bit instruction {instruction[30],instruction[14:12]}
        Branch = branch_in;
        Memread = memread_in;
        Memtoreg = memtoreg_in;
        Memwrite = memwrite_in;
        Regwrite = regwrite_in;
        Alusrc = aluSrc_in;
        aluop = Aluop_in;
```

```
        end
    end
endmodule
```

Module EXMEM:

```
module EXMEM(
input clk,reset
input [63:0] Adder_out, //adder output
input [63:0] Result_in_alu,//64bit alu output
input Zero_in,//64bit alu output
input [63:0] writedata_in, //2 bit mux2by1 output
input [4:0] Rd_in, //IDEX output
input branch_in,memread_in,memtoreg_in,memwrite_in,regwrite_in, //IDEXX outputs
output reg [63:0] Adderout,
output reg zero,
output reg [63:0] result_out_alu,
output reg [63:0] writedata_out,
output reg [4:0]rd,
output reg Branch,Memread,Memtoreg, Memwrite, Regwrite);

always @ (posedge clk)
  begin
    if (reset == 1'b1)
      begin
        Adderout = 64'b0;
```

```
            zero = 1'b0;
            result_out_alu = 63'b0;
            writedata_out = 64'b0;
            rd = 5'b0;
            Branch = 1'b0;
            Memread = 1'b0;
            Memtoreg =1'b0;
            Memwrite = 1'b0;
            Regwrite = 1'b0;

          end
        else
          begin
            Adderout = Adder_out;
            zero = Zero_in;
            result_out_alu = Result_in_alu;
            writedata_out = writedata_in;
            rd = Rd_in;
            Branch = branch_in;
            Memread = memread_in;
            Memtoreg = memtoreg_in;
            Memwrite = memwrite_in;
            Regwrite = regwrite_in;
          end
      end
endmodule
```

Module MEMWB:

```
module MEMWB(
  input clk,reset,
  input [63:0] read_data_in,
  input [63:0] result_alu_in, //2 bit 2by1 mux input b
  input [4:0]Rd_in, //EX MEM output
  input memtoreg_in, regwrite_in, //ex mem output as mem wb inputs
  output reg [63:0] readdata, //1bit
  output reg [63:0] result_alu_out,//1bit
  output reg [4:0] rd,
  output reg Memtoreg, Regwrite
);

  always @(posedge clk)
    begin
      if (reset == 1'b1)
        begin
          readdata = 63'b0;
          result_alu_out = 63'b0;
          rd = 5'b0;
          Memtoreg = 1'b0;
          Regwrite = 1'b0;
```

```
        end
      else
        begin
         readdata = read_data_in;
          result_alu_out = result_alu_in;
          rd = Rd_in;
          Memtoreg = memtoreg_in;
          Regwrite = regwrite_in;
        end
    end
Endmodule
```

Module Forwarding Unit :

```
module ForwardingUnit (
  input [4:0] RS_1, //ID/EX.RegisterRs1
  input [4:0] RS_2, //ID/EX.RegisterRs2
  input [4:0] rdMem, //EX/MEM.Register Rd
  input [4:0] rdWb, //MEM/WB.RegisterRd

  input regWrite_Wb, //MEM/WB.RegWrite
  input regWrite_Mem, // EX/MEM.RegWrite
  output reg [1:0] Forward_A,
  output reg [1:0] Forward_B
);

 always @(*)
```

```
begin

        if ( (rdMem == RS_1) & (regWrite_Mem != 0 & rdMem !=0))

    begin

        Forward_A = 2'b10;

    end

        else

    begin

      // Not condition for MEM hazard

     if ((rdWb== RS_1) & (regWrite_Wb != 0 & rdWb != 0) & ~((rdMem == RS_1) &(regWrite_Mem != 0 &
rdMem !=0)  )  )

       begin

        Forward_A = 2'b01;

       end

      else

       begin

        Forward_A = 2'b00;

       end

    end


   if ( (rdMem == RS_2) & (regWrite_Mem != 0 & rdMem !=0) )

    begin

     Forward_B = 2'b10;

    end

   else

    begin

      // Not condition for MEM Hazard
```

```
        if ( (rdWb == RS_2) & (regWrite_Wb != 0 & rdWb != 0) &  ~((regWrite_Mem != 0 & rdMem !=0 ) &
(rdMem == RS_2) ) )
          begin
            Forward_B = 2'b01;
          end
        else
          begin
            Forward_B = 2'b00;
          end
      end
  end
Endmodule
```

****Here is our new instruction memory module which tests the following**

instructions of R type using 5 stage pipelining:

add x19, x8, x0

add x20, x19, x0

add x21, x19,x 20

add x22, x19, x2

```
module instruction_memory(
  input [63:0] inst_address,
  output reg [31:0] instruction);
  reg [7:0] inst_mem [15:0]; //initialising array
  initial
```

```
  begin
    {inst_mem[3], inst_mem[2], inst_mem[1], inst_mem[0]} = 32'h000409b3;//1
    {inst_mem[7], inst_mem[6], inst_mem[5], inst_mem[4]} = 32'h00098a33;//2
    {inst_mem[11], inst_mem[10], inst_mem[9], inst_mem[8]} = 32'h01498ab3;//3
    {inst_mem[15], inst_mem[14], inst_mem[13], inst_mem[12]} = 32'h01498b33;//4
  end
 always @ (inst_address)
  begin
    instruction[7:0] = inst_mem[inst_address+0];
    instruction[15:8] = inst_mem[inst_address+1];
    instruction[23:16] = inst_mem[inst_address+2];
    instruction[31:24] = inst_mem[inst_address+3];
  end
Endmodule
```

Results:

My expected result after running this module through my 5 stage pipelined processor is:

X19 initiated with 0

X8 initiated with 4

add x19, x8, x0 = 0

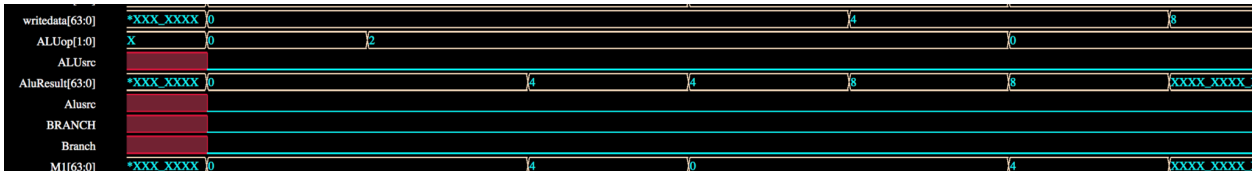
add x20, x19, x0 = 4

add x21, x19,x 20 = 4

add x22, x19, x2 = 4+4

This is successfully demonstrated by the ALUresult signal of the EPwave generated below:

EP Wave :



Link: https://edaplayground.com/x/9Nuh

Task 3:

Addition of pipeline hazard detection unit & Flushing unit is made to make load instruction work in our pipelined processor.Minor changes are made to ensure efficient running of the bubble sort implementation in Task 1 Memory Module.

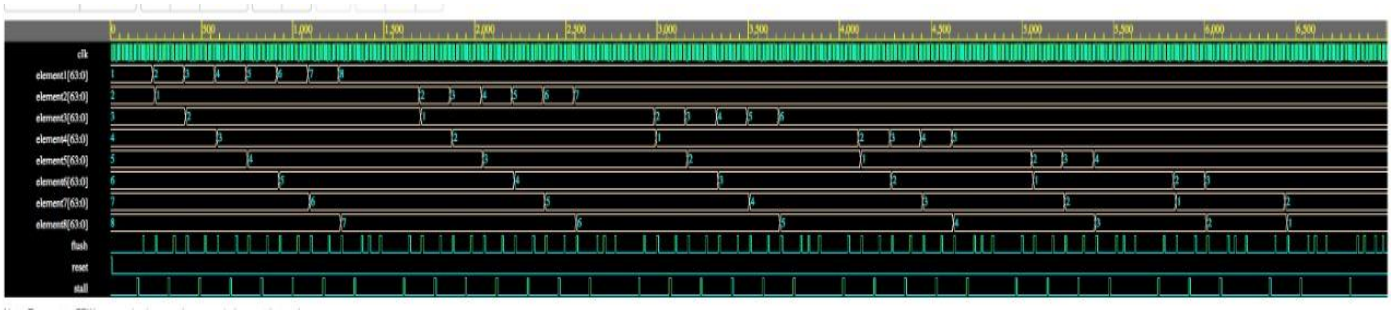
Module for Hazard Detection Unit:

```
module hazard_detection_unit (
  input Memread,
  input [31:0] inst,
  input [4:0] Rd,
  output reg stall
);
initial
  begin
    stall = 1'b0;
  end
always @(*)
  begin
    if (Memread == 1'b1 && ((Rd == inst[19:15]) || (Rd == inst[24:20])))
      stall = 1'b1;
    else
      stall = 1'b0;
  end
endmodule
```

Module for Pipelined Flushing Unit:

```
module pipeline_flush (
input branch,
  output reg flush
);
initial
  begin
    flush = 1'b0;
  end
  always @(*)
  begin
    if (branch == 1'b1)
      flush = 1'b1;
    else
      flush = 1'b0;
  end
endmodule
```

EP Wave:



Link : https://edaplayground.com/x/r3Er