1 Objectives

The objectives of this laboratory were:

- Properly understanding hazard detection mechanisms.
- Developing methods to reduce stalls due to data Hazards, mainly through the development of a forwarding unit.
- Implementing the Hazard detection and forwarding unit in the built single cycle processor in order to create a full running example of a pipelined processor.

2 Design

2.1 Hazard unit

The Hazard unit was modeled as a Finite State Machine with three outputs:

- PCWrite: Update the PC if no stall
- IFWrite: Fetch the next instruction.
- Buble: Signaling to introduce a No-op in the control unit and freeze the movement of signals through latches.

The Hazard Unit was designed to have the ability to detect load hazards and issue one stall when they happen, Load Hazards are fired when the last RW matches current Rs or Rstand that the last RWwas not 0 and memory was being read. When a Load Hazard is detected a bubble is issued once and PC is not updated. For Braches, if a branch is detected, we will go into a branch 0 state after one stall, here if the branch is not taken, operation will continue as normal, if the branch is taken, then an extra stall will be issued in order to give time to update nextPC with branch address. This second branch stall will now have PCWrite and IFWrite enabled. Finally if Jump is detected, a stall is issued once and then we return to no hazard state. Each state has a different output for a 2 bit select. The logic for the fsm is shown in figure1. This module is shown in the code bellow.

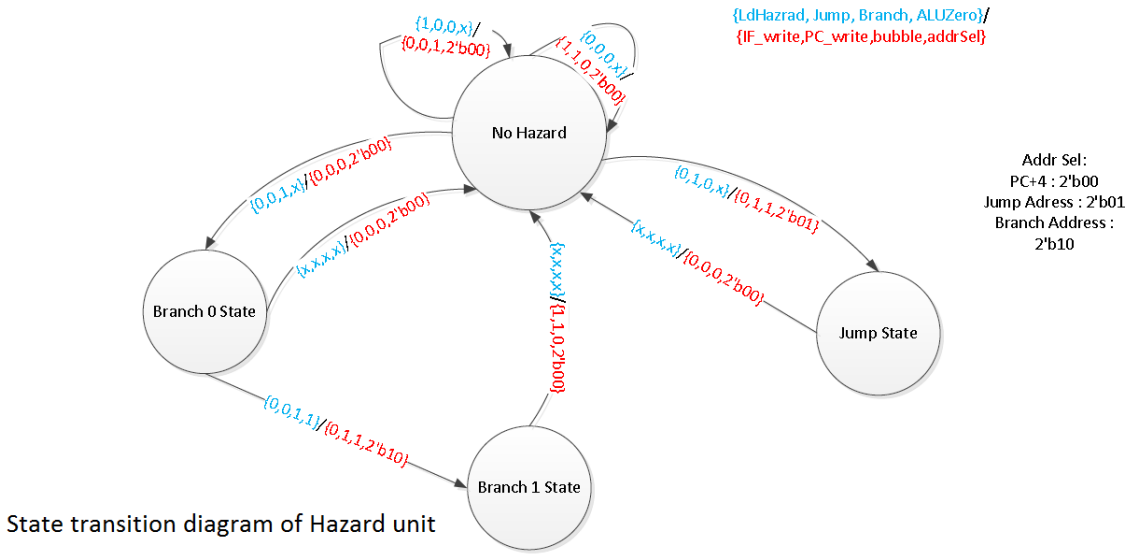

Figure 1: Hazard Detection Unit FSM

```
`timescale 1ns / 1ps
`define NoHazard_state  3'b000
`define bubble0_state   3'b001
`define bubble1_state   3'b010
`define Branch0_state   3'b011
`define Branch1_state   3'b100
`define Branch2_state   3'b101
`define Jump_state     3'b110

module HazardUnit( IF_write, PC_write, bubble, addrSel, Jump, Branch, ALUZero, memReadEX, c
        output reg PC_write, IF_write, bubble;
        output reg [1:0] addrSel;
        input Jump, Branch, ALUZero, memReadEX, Clk, Rst;
        input UseShamt, UseImmed;
        input   [4:0]   prevRt;
        input   [4:0]   currRs;
        input   [4:0]   currRt;




        /*internal signals*/
        wire cond1, cond2, cond3, cond4;
        wire LdHazard;

        /*internal state*/
        reg [2:0] FSM_state, FSM;

        /*create compare logic*/
        assign  cond1 = prevRt!=0;      //if Rt we checking is 0, no dependency
        assign  cond2 = (((currRs==prevRt)|(currRt==prevRt)) &(memReadEX)&(~(UseShamt|UseIm
        assign  cond3 = (((currRs==prevRt)&(UseShamt))&(memReadEX)); //? 1:0;
        assign  cond4 = (((currRs==prevRt)&&(UseImmed))&(memReadEX)); //? 1:0;
        assign  LdHazard = cond2?1:cond3?1:cond4?1:0;//(((currRs==rw3)||(rt==rw3))&&(rw3!=
        /* finds the dependancy btween current instruction and the
                        one before */
//      assign  LdHazard = ((currRs==prevRt)|(currRt==prevRt)) &memReadEX& UseImmed & UseSh
        /*FSM state register*/
        always @(negedge Clk) begin
                if(Rst == 0)
                        FSM_state <= 0;
                else begin
                        FSM_state <= FSM;
                        //$display("%b",FSM);
                end
                //if(LdHazard) $display("Load Hazard@ fsmstate");

        end

        /*FSM next state and output logic*/
        //always @(FSM_state or LdHazard or Jump or Branch or ALUZero ) begin //combinatorg
        always @(* ) begin
                //$display("%b",FSM_state);
                case(FSM_state)
```

```
`NoHazard_state: begin
//$display("Nh");
        //load hazard detected
        if({LdHazard,Jump, Branch}==3'b100) begin //bubble
                //if(LdHazard) $display("Load Hazard");
                {IF_write,PC_write,bubble,addrSel} <= #2 5'b00100;
                FSM <= #2 `NoHazard_state;

        end
        else if({LdHazard,Jump, Branch}==3'b010) begin //jump
                //$display(LdHazard,Jump, Branch,ALUZero);
                {IF_write,PC_write,bubble,addrSel} <= #2 5'b01001;
                FSM <= #2 `Jump_state;
        end
        else if({LdHazard,Jump, Branch}==3'b001) begin //branch
                {IF_write,PC_write,bubble,addrSel} <= #2 5'b00000;
                FSM <= #2 `Branch0_state;
        end
        //normal state
        else if({LdHazard,Jump, Branch}==3'b000) begin //normal

                {IF_write,PC_write,bubble,addrSel} <= #2 5'b11000;
                FSM <= #2 `NoHazard_state;
        end
end

`Branch0_state: begin
        //$display("branch0");
        if(ALUZero) begin //branch taken

                {IF_write,PC_write,bubble,addrSel} <= #2 5'b01110;
                FSM <= #2 `Branch1_state;
        end
        else begin      //branch not taken
                {IF_write,PC_write,bubble,addrSel} <= #2 5'b11100;
                FSM <= #2 `NoHazard_state;
        end
end

`Branch1_state: begin //return eventually
        //$display("branch1");
        {IF_write,PC_write,bubble,addrSel} <= #2 5'b11100;
        FSM <= #2 `NoHazard_state;
end


`Jump_state: begin      //return
        //$display("jump");
        {IF_write,PC_write,bubble,addrSel} <= #2 5'b11100;
        FSM <= #2 `NoHazard_state;

end

default: begin
```

```
                                        //FSM <= #2 NoHazard_state;
                                        PC_write <= #2 1'bx;
                                        IF_write <= #2 1'bx;
                                        bubble  <= #2 1'bx;
                                        addrSel <= #2 2'bx;
                                end
                        endcase
                end
endmodule
```

2.2 Forwarding Unit

The forwarding unit was designed to be the unit that controls wether Data gets forwarded to ALU A or ALU B from the output of the execute stage or the output of the Memory stage. If the current Rs or Rt match with the previous RW address, and immediates/shamts are not used, the data is brought back. The operation is showin in the code bellow.

```
`define sel_mem 2'b00
`define sel_ex 2'b01
`define sel_normal 2'b10
`define sel_sh_im 2'b11

module ForwardingUnit(AluOpCtrlA, AluOpCtrlB, DataMemForwardCtrlEX, DataMemForwardCtrlMEM,
        input UseShamt, UseImmed;
        input [4:0] IDRs, IDRt, EXRw, MEMRw;
        input EXRegWrite, MEMRegWrite;
        output  [1:0] AluOpCtrlA, AluOpCtrlB;
        output DataMemForwardCtrlEX, DataMemForwardCtrlMEM;

        wire conda1, conda2, conda3; //control selection logic for ALUA

        assign conda1 = (~UseShamt)  ;// not using shift value
        assign conda2 = (IDRs==MEMRw)/* & (MEMRw!=EXRw)*/ & MEMRegWrite &(MEMRw!=0);     //
        assign conda3 = (IDRs==EXRw) & EXRegWrite& (EXRw!=0);              // forward a from m

        assign AluOpCtrlA = conda1 ?
                                (conda2 ? `sel_mem: conda3 ? `sel_ex: `sel_normal ) :
                                UseShamt? `sel_sh_im: `sel_normal;

        wire condb1, condb2, condb3; //control selection logic for ALUB

        assign condb1 = (~UseImmed) ; //not using immediate value
        assign condb2 = (IDRt==MEMRw)/* & (MEMRw!=EXRw) */& MEMRegWrite & (MEMRw!=0);    //
        assign condb3 = (IDRt==EXRw) & EXRegWrite & (EXRw!=0);             //forward b from m

        assign AluOpCtrlB = condb1 ?
                                (condb2 ? `sel_mem: condb3 ? `sel_ex: `sel_normal ) :
                                UseImmed? `sel_sh_im: `sel_normal;

        wire condc1, condc2;     //wires for forward control signals
        assign DataMemForwardCtrlMEM = MEMRegWrite &(IDRt==MEMRw);         //forward from mem
        assign DataMemForwardCtrlEX = EXRegWrite & (IDRt==EXRw);           //forward from ex
        //assign  {DataMemForwardCtrlEX, DataMemForwardCtrlMEM} =
        //                condc2? 2'b10:condc1 ? 2'b01: 2'b00;
```

endmodule

2.3 Pipelined Cycle Processor

The pipelined processor was built using the template given. The Hazard unit was connected to its corresponding singals in their corresponding stages as seen in the code bellow. As given the Hazard units sits in the ID stage, every EX stage register or signal read is has appended a 3 and everything from the mem stage has appended a 4. The same thing was done with the forwading unit, that according to its signals, it selected between the outpus of the mem stage or the reg stage.

```
‘timescale 1ns / 1ps

‘define sel_mem 2'b00
‘define sel_ex 2'b01
‘define sel_normal 2'b10
‘define sel_sh_im 2'b11
‘define adsel_pc4 2'b00
‘define adsel_jmp 2'b01
‘define adsel_brn 2'b10

module PipelinedProc(CLK, Reset_L, startPC, dMemOut);
        input CLK;
        input Reset_L;
        input [31:0] startPC;
        output [31:0] dMemOut;


        //Hazard
        wire Bubble;
        wire PCWrite;
        wire IFWrite;

        //Stage 1
        wire    [31:0]  currentPCPlus4;
        wire    [31:0]  jumpDescisionTarget;
        wire    [31:0]  nextPC;
        wire    [31:0]  tentPC;
        reg     [31:0]  currentPC;
        wire    [31:0]  currentInstruction;

        //Stage 2
        reg     [31:0]  currentInstruction2;
        wire    [5:0]   opcode;
        wire    [4:0]   rs, rt, rd;
        wire    [15:0]  imm16;
        wire    [4:0]   shamt;
        wire    [5:0]   func;
        wire    [31:0]  busA, busB, ALUImmRegChoice, signExtImm;
        wire    [31:0]  jumpTarget;


        //Stage 2 Control Wires
        wire    regDst, aluSrc, memToReg, regWrite, memRead, memWrite, branch, jump, signEx
```

```
wire     UseShiftField;
wire     rsUsed, rtUsed;
wire     [4:0]    rw;
wire     [3:0]    aluOp;
wire     [31:0]   ALUBIn;
wire     [31:0]   ALUAIn;
wire     [1:0]    addrSel;
wire     [1:0] AluOpCtrlA, AluOpCtrlB;


//Stage 3
reg      [31:0]   ALUAIn3, ALUBIn3, busB3, signExtImm3;
reg      [4:0]    rw3;
wire     [5:0]    func3;
wire     [31:0]   shiftedSignExtImm;
wire     [31:0]   branchDst;
wire     [3:0]    aluCtrl;
wire     aluZero;
wire     [31:0]   aluOut;
wire DataMemForwardCtrlEX, DataMemForwardCtrlMEM;

//Stage 3 Control
reg      regDst3, memToReg3, regWrite3, memRead3, memWrite3, branch3;
reg      [3:0]    aluOp3;
reg DataMemForwardCtrlEX3, DataMemForwardCtrlMEM3;

//Stage 4
reg      aluZero4;
reg      [31:0]   branchDst4, aluOut4, busB4;
reg      [4:0]    rw4;
wire     [31:0]   memOut;
reg  DataMemForwardCtrlEX4;


assign dMemOut = memOut;

//Stage 4 Control
reg memToReg4, regWrite4, memRead4, memWrite4, branch4;

//Stage 5
reg      [31:0]   memOut5, aluOut5;
reg      [4:0]    rw5;
wire     [31:0]   regWriteData;

//Stage 5 Control
reg memToReg5, regWrite5;


//Stage 1 Logic
/*Below is a special case.  If we are doing a jump, and IFWrite is set to true,
then we have completed the jump.  That means we are not jumping anymore.
Same is true of a branch that we just took.
*/
//assign #1 jumpDescisionTarget = (jump & ~IFWrite) ? jumpTarget : currentPCPlus4;
```

```
//assign #1 nextPC = (branch4 & aluZero4 & ~IFWrite) ? branchDst4 : jumpDescisionT
/*`define adsel_pc4 2'b00
`define adsel_jmp 2'b01
`define adsel_brn 2'b10 */
assign signExtImm = {{16{(signExtend ? 1'b0:imm16[15])}},imm16};
assign jumpTarget = {currentPC[31:28], currentInstruction2[25:0], 2'b00};
assign #1 nextPC = (addrSel== `adsel_jmp ) ? jumpTarget:
                        (addrSel== `adsel_brn) ? branchDst:
                        currentPCPlus4;
//assign #1 nextPC = PCWrite ? tentPC : currentPC;

always @ (negedge CLK) begin
        if(~Reset_L)
                currentPC <= startPC;
        else if(PCWrite)
                currentPC <= nextPC;

end
assign #2 currentPCPlus4 = currentPC + 4;
InstructionMemory instrMemory(currentInstruction, currentPC);

//Stage 2 Logic
always @ (negedge CLK or negedge Reset_L) begin
        if(~Reset_L)
                currentInstruction2 <= 32'b0;
        else if(IFWrite) begin
                currentInstruction2 <= currentInstruction;
        end
end
assign {opcode, rs, rt, rd, shamt, func} = currentInstruction2;
assign imm16 = currentInstruction2[15:0];
PipelinedControl controller(regDst, aluSrc, memToReg, regWrite, memRead, memWrite,

assign #1 rw = regDst ? rd : rt;
assign #2 UseShiftField = ((aluOp == 4'b1111) && ((func == 6'b000000) || (func ==
assign #2 rsUsed = (opcode != 6'b001111/*LUI*/) & ~UseShiftField;
assign #1 rtUsed = (opcode == 6'b0) || branch || (opcode == 6'b101011/*SW*/);
/*Hazard hazard(PCWrite, IFWrite, Bubble, branch, aluZero4, jump,
                        regWrite ? rw : 5'b0,
                        rsUsed ? rs : 5'b0,
                        rtUsed ? rt : 5'b0,
                        Reset_L, CLK);*/

HazardUnit hazard( .IF_write(IFWrite),
        .PC_write(PCWrite),
        .bubble(Bubble),
        .addrSel(addrSel),
        .Jump(jump),
        .Branch(branch),
        .ALUZero(aluZero),
        .memReadEX(memRead3),
        .currRs( rs),
        .currRt(rt ),
```

```
        .prevRt(rw3),
        .UseShamt(UseShiftField),
        .UseImmed(aluSrc),
        .Rst(~Reset_L),
        .Clk(CLK)    );

ForwardingUnit fwd( .AluOpCtrlA(AluOpCtrlA),
                    .AluOpCtrlB(AluOpCtrlB),
                    .DataMemForwardCtrlEX(DataMemForwardCtrlEX),
                    .DataMemForwardCtrlMEM(DataMemForwardCtrlMEM),
                    .UseShamt(UseShiftField),
                    .UseImmed(aluSrc),
                    .IDRs(rs),
                    .IDRt(rt),
                    .EXRw(rw3),
                    .MEMRw(rw4),
                    .EXRegWrite(regWrite3),
                    .MEMRegWrite(regWrite4) );

RegisterFile Registers(busA, busB, regWriteData, rs, rt, rw5, regWrite5, CLK);


assign #2 ALUImmRegChoice = aluSrc ? signExtImm : busB;


assign #2 ALUAIn = (AluOpCtrlA == `sel_mem)? memRead4? memOut:aluOut4: //UseShiftF
                        (AluOpCtrlA == `sel_ex)? aluOut:
                        (AluOpCtrlA == `sel_sh_im)?  busB: busA;
                        //if memory is read forward memory, else forward old resul
assign #2 ALUBIn = (AluOpCtrlB == `sel_mem)? memRead4 ? memOut : aluOut4:
                        (AluOpCtrlB == `sel_ex)? aluOut :
                        UseShiftField ? {27'b0, shamt} :
                        (AluOpCtrlB == `sel_sh_im)? signExtImm: busB;

//Stage 3 Logic
always @ (negedge CLK or negedge Reset_L) begin
        if(~Reset_L) begin
                ALUAIn3 <= 0;
                ALUBIn3 <= 0;
                busB3 <= 0;
                signExtImm3 <= 0;
                rw3 <= 0;
                regDst3 <= 0;
                memToReg3 <= 0;
                regWrite3 <= 0;
                memRead3 <= 0;
                memWrite3 <= 0;
                branch3 <= 0;
                aluOp3 <= 0;
                DataMemForwardCtrlEX3 <= 0;
                DataMemForwardCtrlMEM3 <=0;
        end
        else if(Bubble) begin
                ALUAIn3 <= 0;
```

```
                ALUBIn3 <= 0;
                busB3 <= 0;
                signExtImm3 <= 0;
                rw3 <= 0;
                regDst3 <= 0;
                memToReg3 <= 0;
                regWrite3 <= 0;
                memRead3 <= 0;
                memWrite3 <= 0;
                branch3 <= 0;
                aluOp3 <= 0;
                DataMemForwardCtrlEX3 <= 0;
                DataMemForwardCtrlMEM3 <=0;
        end
        else begin
                ALUAIn3 <= ALUAIn;
                ALUBIn3 <= ALUBIn;
                busB3 <= busB;
                signExtImm3 <= signExtImm;
                rw3 <= rw;
                regDst3 <= regDst;
                memToReg3 <= memToReg;
                regWrite3 <= regWrite;
                memRead3 <= memRead;
                memWrite3 <= memWrite;
                branch3 <= branch;
                aluOp3 <= aluOp;
                DataMemForwardCtrlEX3 <= DataMemForwardCtrlEX;
                DataMemForwardCtrlMEM3 <=DataMemForwardCtrlMEM;
        end
end

assign func3 = signExtImm3[5:0];
ALUControl mainALUControl(aluCtrl, aluOp3, func3);
ALU mainALU(aluOut, aluZero, ALUAIn3, ALUBIn3, aluCtrl);

assign shiftedSignExtImm = {signExtImm3[29:0], 2'b0};
assign #2 branchDst = currentPC + shiftedSignExtImm;

//Stage 4 Logic
always @ (negedge CLK or negedge Reset_L) begin
        if(~Reset_L) begin
                aluZero4 <= 0;
                branchDst4 <= 0;
                aluOut4 <= 0;
                busB4 <= 0;
                rw4 <= 0;
                memToReg4 <= 0;
                regWrite4 <= 0;
                memRead4 <= 0;
                memWrite4 <= 0;
                branch4 <= 0;
                DataMemForwardCtrlEX4 <= 0;
        end
```

```
            else begin
                    aluZero4 <= aluZero;
                    branchDst4 <= branchDst;
                    aluOut4 <= aluOut;
                    busB4 <=DataMemForwardCtrlMEM3? regWriteData :  busB3;
                    rw4 <= rw3;
                    memToReg4 <= memToReg3;
                    regWrite4 <= regWrite3;
                    memRead4 <= memRead3;
                    memWrite4 <= memWrite3;
                    branch4 <= branch3;
                    DataMemForwardCtrlEX4 <= DataMemForwardCtrlEX3;
            end
    end
    //DataMemory dmem(memOut, aluOut4, busB4, memRead4, memWrite4, CLK);
    DataMemory dmem (.ReadData(memOut),
             .Address(aluOut4),
             .WriteData(DataMemForwardCtrlEX4? regWriteData :busB4),
             .MemoryRead(memRead4),
             .MemoryWrite(memWrite4),
             .Clock(CLK) );

    //Stage 5 Logic
    always @ (negedge CLK or negedge Reset_L) begin
            if(~Reset_L) begin
                    memOut5 <= 0;
                    aluOut5 <= 0;
                    rw5 <= 0;
                    memToReg5 <= 0;
                    regWrite5 <= 0;
            end
            else begin
                    memOut5 <= memOut;
                    aluOut5 <= aluOut4;
                    rw5 <= rw4;
                    memToReg5 <= memToReg4;
                    regWrite5 <= regWrite4;
            end
    end
    assign #1 regWriteData = memToReg5 ? memOut5 : aluOut5;

endmodule
```

3 Results

3.1 Hazard unit

As seen in figure 2 the Hazard unit was built to spec as a sinthesizable module. As shown on 3, it performedto the sprecification referenced in figure 1.

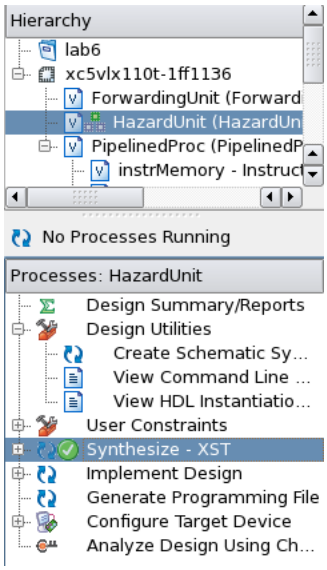

Figure 2: Hazard Detection Synthesis result

Figure 3: Hazard Test results

3.2 Forwarding Unit

Similar to the Hazard unit, the Forwarding unit was built to be sinthesizable as shown in figure 4.

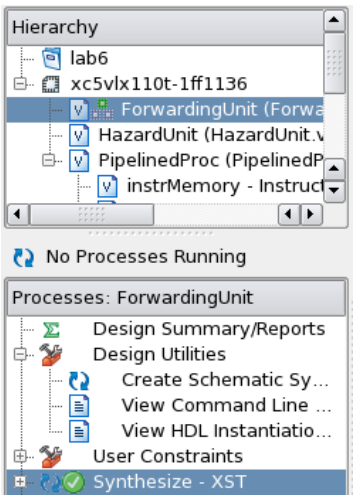

Figure 4: Forwarding Synthesis result

3.3 Pipelined processor

Finally, the Pipelined procesor was built as specified in previous section, figure 5 shows all the tests of PipelinedProcTest passing.

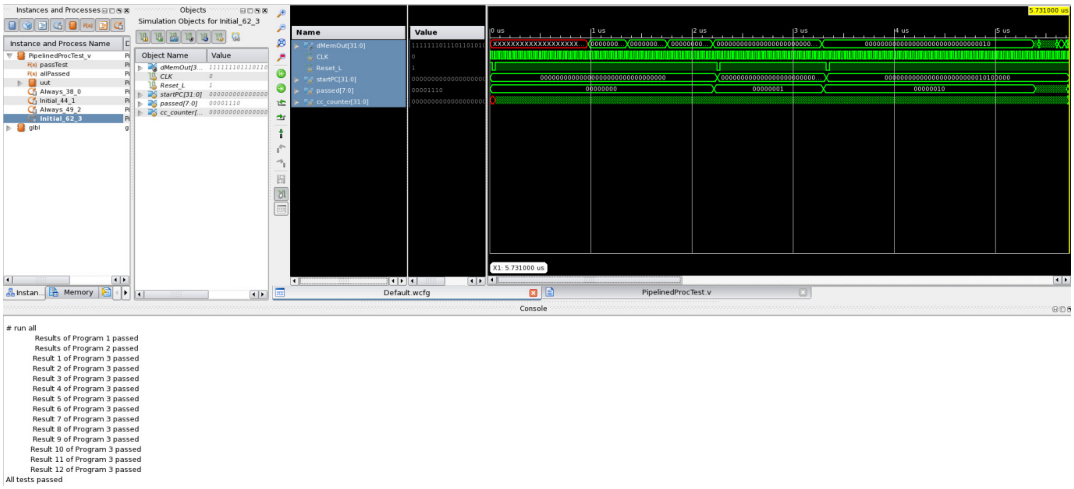

Figure 5: Processor Test

3.4 Single Cycle Processor

4 Questions

Instead of stalling the pipeline while waiting for a branch to be evaluated, we could simply continue to fetch from PC+4. What name is commonly given to this technique? What changes would have to be made to our existing design to accommodate this improvement? Be sure to

include modifications to the Hazard Detection Unit and pipeline registers.

This is called static branch prediction in where the branch is assumed to be a non taken. In order to accomodate this improvement the hazard unit should not issue a stall at branch 0 and output PCWrite and IFWrite, however at branch 0 the full evaluation would have to be done and if the branch is taken a flush and one stall would have to occur. For the rest, a flush signal needs to be added, when a flush is issued, all data and signals will be reset and IF will have to begin from the branch target.

What clock rate did the synthesis process estimate your overall design would run at? Look through the synthesis report and locate the section where the design timing is estimated. From that, determine which components are in the critical path. What changes could you make to the design to improve the clock rate without adding additional pipeline stages?

Processor is running at 120ns. Memory, jumps and branches are in the critical path as they can stall the pipeline. Although fillup and drain made the processor's CPI worse (as seen in the next question) the point of pipelining is gaining the ability to increase the frequency, so that's the first change to be made. In addition, as seen in last question, doing branch prediction can decrease the conditional branches to half a stall on average (assuming 50% of branches taken). Finally, the processor could be implemented as an out of order processor with the same stages but that would be very complicated.

Determine the CPI for each of the three programs executing on your processor. Compare those results to the CPI of the single-cycle processor. Also, provide the average CPI of all three programs.

In order to streamline the process, a program counter was added to a testbench. The timing was obtaining by adding a counter that resets with reset. The number of instructions were calculated by making a sum of all the times that PC changes excluding addresses of NOPs and addresses of don't cares, the results are in table 1.

Program	Instructions	Clock cycle Single	Clock Cycles Pipelined	Avg CPI Single	Avg CPI pipelined
1	33	33	46	1	1.39
2	11	11	34	1	3.09
3	37	37	40	1	1.08
total	81	81	120	1	1.48

Table 1: CPI comparison

We can see that the single cycle processor a larger CPI, meaning it has a worse performance, this because the processor has to spend time filling up and draining as well as the required stalls, in a real processor, the lower latency of each stay will allow for overclocking the processor at 5x the speed and still obtaining the same results.

Given the clock rate and average CPI, compute the MIPS (Million Instructions Per Second)rating for both versions of processor. Which one is faster and why?

The clock rate the same for both processors 1s/120ns or 8,333,333 clocks per second. At an average CPI of 1, the single clycle processor can do 8.33 MIPS. On the other hand at an average of 1.48 CPI, at this clock frequency, the processor has a MIPS of 8.33/1.48 or **5.63**. If they share the same clock frequency the single cycle processor is faster because it doesn't stall, however in an ideal world the clock rate could be increased by around 5x, making the pipelned processor more than 3x faster than the non pipelined one.