Contents

1 Objective

In this project, we need to model both single cycle and pipelined implementation of MIPS computer in Verilog. Such MIPS implementation should support a subset of MIPS instruction set including:

- The memory-reference instructions load word (lw) and store word (sw)
- The arithmetic-logical instructions add, addi, sub, and, andi, or, and slt
- The jumping instructions branch equal (beq), branch not equal (bne), and jump (j)

2 The Top-level Block Diagram

2.1 Top-level Block Diagram of Single Cycle Implementation

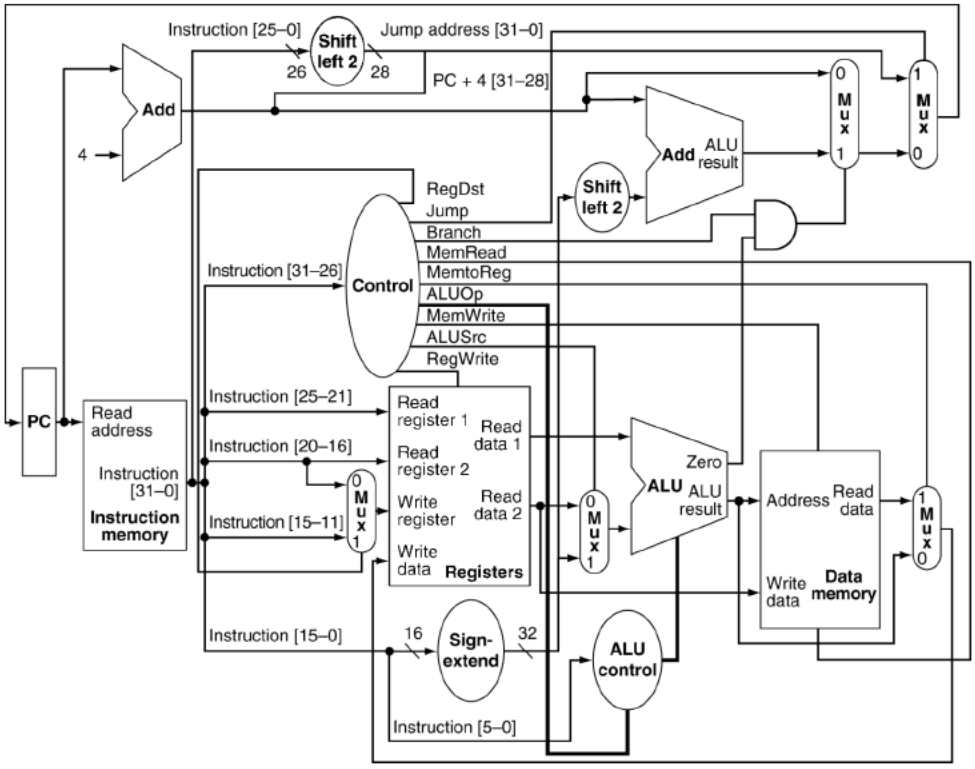

Figure 1: top-level block diagram of single cycle implementation

2.2 Top-level Block Diagram of Pipeline Implementation

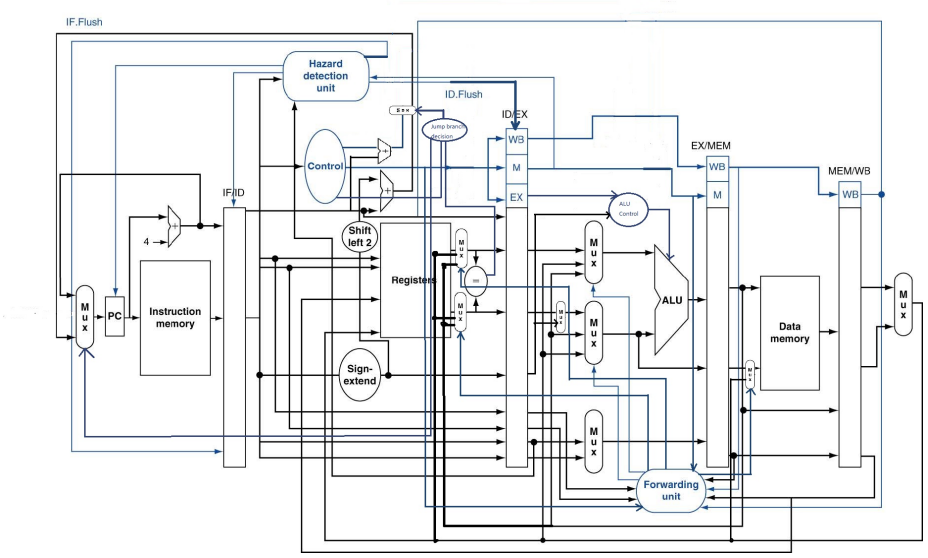

Figure 2: top-level block diagram of pipeline implementation

3 The RTL Schematic of the Verilog Design

3.1 Design for Single Cycle Processor

The following are the general RTL design of the single cycle processor.

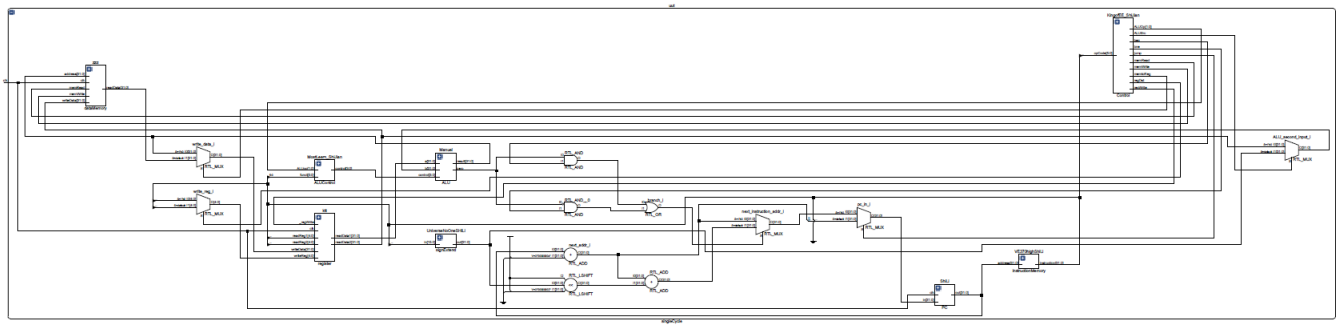

Figure 3: RTL diagram of single cycle implementation

3.2 Design for Pipeline Processor

The following are the general RTL design of the pipeline processor.

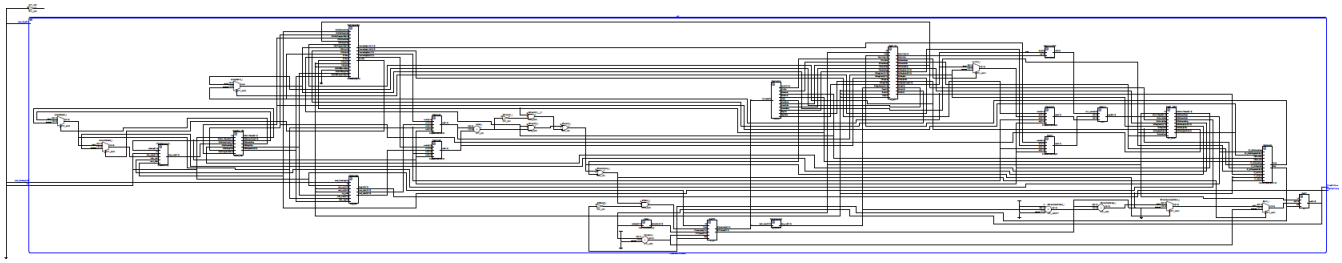

Figure 4: RTL diagram of pipeline implementation

4 Components Design and Explanations

4.1 ALU

For ALU, we use the signal *ALUControl* to control the output zero and result. At the beginning of our simulation, we initialize the result to 0. The figure below shows the code implementation of ALU. The complete version of code is in Appendix.

```
assign zero = (result == 0);

initial begin
    result = 32'b0;  //to initialize the result
end

always @ (a or b or ALU_control) begin
    case (ALU_control)
        4'b0000:
            result = a & b;  // and
        4'b0001:
            result = a | b;  // or
        4'b0010:
            result = a + b;  // add
        4'b0110:
            result = a - b;  // sub
        4'b0111:
            result = (a < b) ? 1:0;  // slt
        default:
            result = a;
    endcase
```

Figure 5: Part of the code of ALU

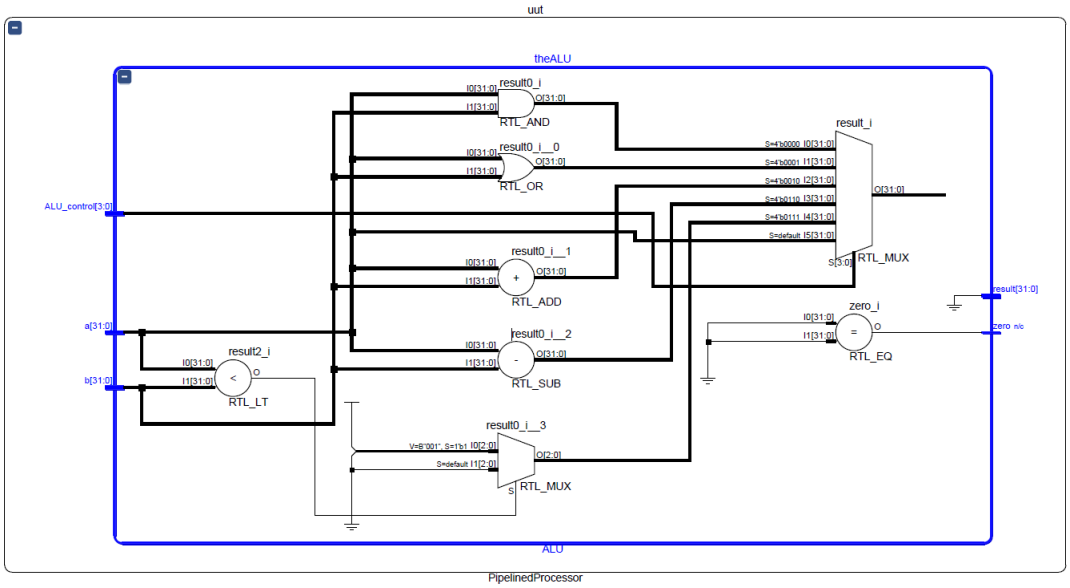

Figure 6: RTL diagram of ALU

4.2 Register File

In register file, we just create 32 32-bit registers and then use them to store, read or write data. It is controlled by the *reg_write* signal. Code implementation is shown in figure below. The complete version of code is in Appendix.

```
reg [31:0] registers [0:31];
integer i;

initial begin
    for (i = 0; i < 32; i = i + 1)
        registers[i] = 32'b0;  // initialize the registers
end

assign read_data1 = registers[read_reg1];
assign read_data2 = registers[read_reg2];
assign RegOut = registers[Input_Readreg];
always @(negedge clk) begin
    if (reg_write == 1)
        registers[write_reg] <= write_data;
end
```

Figure 7: Part of the code of Register File

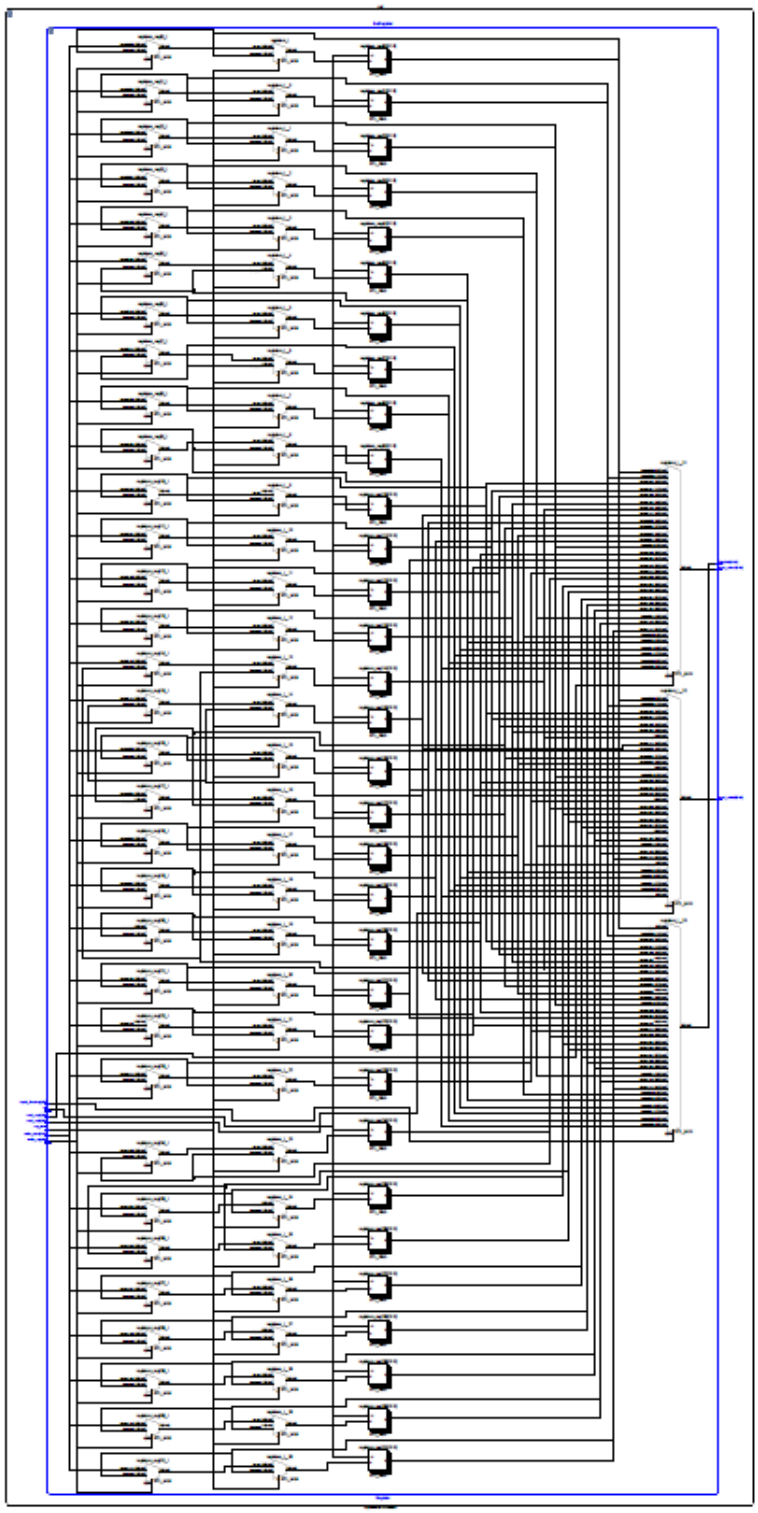

Figure 8: RTL diagram of Register File

4.3 Instruction Memory

In the instruction memory part, we use the **include** command to load the instructions from the given text file. Such way of loading instructions is easier to be implemented and less likely of making mistakes. Memory should be initialized as 0 first before we get the input address and output instructions. Related codes are shown in figure below.

```
initial begin
    for (i = 0; i <= 63; i = i + 1)
        memory[i] = 32'b0;  // initialize the instruction memory
   `include "InstructionMem_for_P2_Demo_updated.txt"  // load the instructions
end
```

Figure 9: Part of the code of Instruction Memory

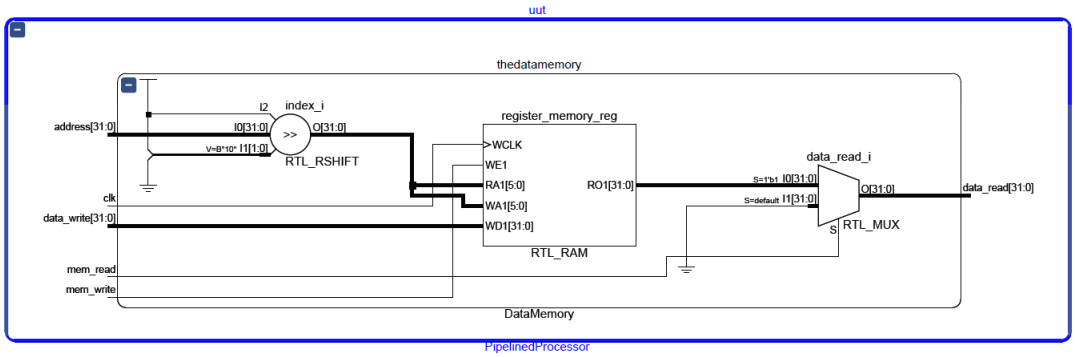

Figure 10: RTL diagram of Instruction Memory

4.4 Data Memory

Compared with the instruction memory, implementation of the data memory is more tricky. We need to get the index of data by multiplying the address of the data to 4. Then we write data one by one in case that the signal *mem_write* is 1. Initialization is included for this part, too. Related codes can be seen in picture below.

```
assign index = address >> 2;
initial begin
    for (i = 0; i < size; i = i + 1)
        register_memory[i] = 32'b0;  // initialize the memory
end

always @ ( posedge clk ) begin
    if (mem_write == 1'b1) begin
        register_memory[index] = data_write;  //write data one by one
    end
end
```

Figure 11: Part of the code of Data Memory

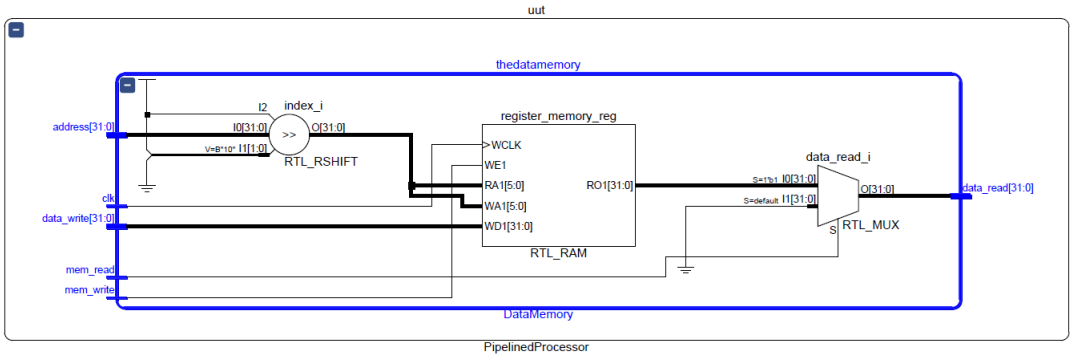

Figure 12: RTL diagram of Data Memory

4.5 Forwarding Unit

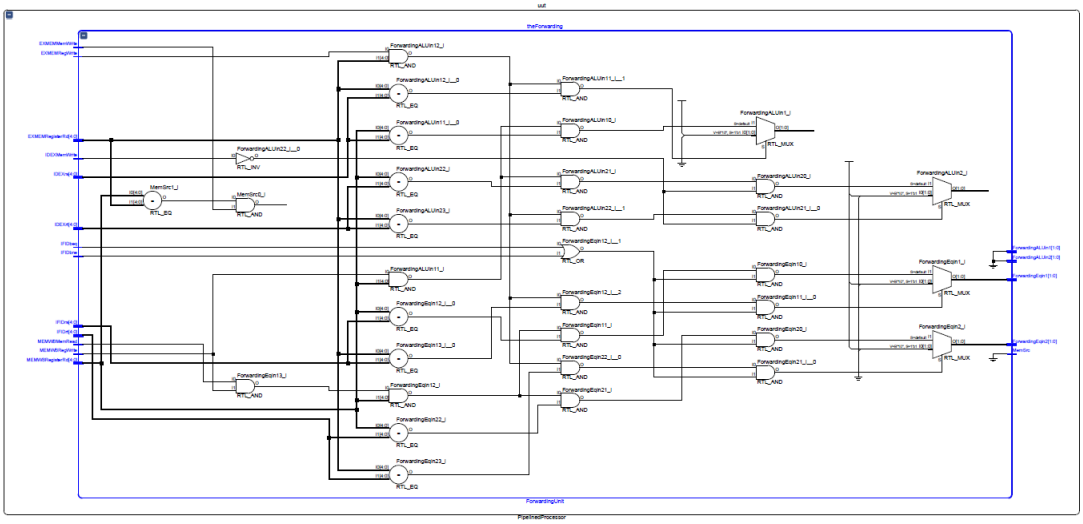

Figure 13: RTL diagram of Forwarding Unit

Forwarding part is one of the most challenging parts to us. We need to detect EX hazzard and MEM hazzard by comparing the value in different registers. What's more, the hazzard that caused by **beq**, **bne** or **jump** instruction should be taken into consideration. This unit helps to path the data we need "ahead of time". Combined with the hazard detection unit, we can take care of these basic hazards successfully and efficiently. All these works are done by using the $if...else...$ conditions and the related codes are shown at the following.

```
//Memsource
    if(MEMWBRegisterRd == EXMEMRegisterRd && EXMEMMemWrite)
        MemSrc = 1'b1;
    else
        MemSrc = 1'b0;
```

Figure 14: Part of the code of Forwarding Unit to detect MemSrc Hazard

```
//ForwardingA
    //EX
    if(EXMEMRegWrite && EXMEMRegisterRd && EXMEMRegisterRd == IDEXrs)
        ForwardingALUin1 = 2'b10;
    //MEM
    else if(MEMWBRegWrite && MEMWBRegisterRd && MEMWBRegisterRd == IDEXrs)
        ForwardingALUin1 = 2'b01;
    else
        ForwardingALUin1 = 2'b00;

//ForwardingB
     //EX
    if(EXMEMRegWrite && EXMEMRegisterRd && EXMEMRegisterRd == IDEXrt && !IDEXMemWrite)
        ForwardingALUin2 = 2'b10;
     //MEM
    else if(MEMWBRegWrite && MEMWBRegisterRd && MEMWBRegisterRd == IDEXrt && !IDEXMemWrite)
        ForwardingALUin2 = 2'b01;
    else
```

Figure 15: Part of the code of Forwarding Unit to detect EX/MEM Hazzard

```
        //BEQ using the ALU result or mem result of last instruction

            if(EXMEMRegWrite && EXMEMRegisterRd &&
    EXMEMRegisterRd == IFIDrs && (IFIDbeq || IFIDbne))
                ForwardingEqin1 = 2'b10;
            else if (MEMWBMemRead && MEMWBRegWrite &&
    MEMWBRegisterRd && MEMWBRegisterRd == IFIDrs && (IFIDbeq || IFIDbne))
                ForwardingEqin1 = 2'b01;
            else
                ForwardingEqin1 = 2'b00;

             if(EXMEMRegWrite && EXMEMRegisterRd &&
    EXMEMRegisterRd == IFIDrt && (IFIDbeq || IFIDbne))
                ForwardingEqin2 = 2'b10;
            else if (MEMWBMemRead && MEMWBRegWrite &&
    MEMWBRegisterRd && MEMWBRegisterRd == IFIDrt && (IFIDbeq || IFIDbne))
                ForwardingEqin2 = 2'b01;
            else
                ForwardingEqin2 = 2'b00;
        end
```

Figure 16: Part of the code of Forwarding Unit to detect beq/bne data Hazzard

5 Hazzard Detection Unit

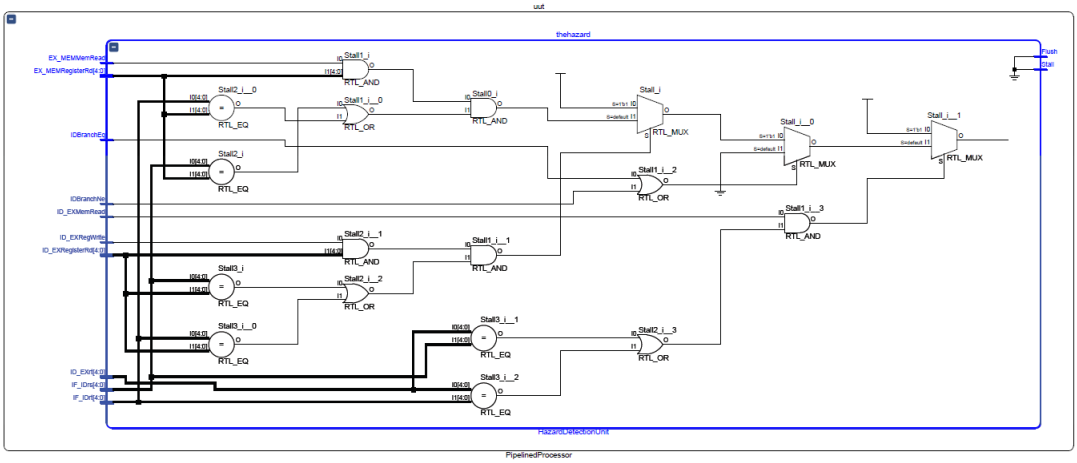

Figure 17: RTL diagram of Forwarding Unit

5.1 Hazard Detection Unit

Hazzard detection part is also one of the most challenging parts. This part mostly is used to generate "bubbles" or nop instruction in the pipeline. When we face a load use hazard or a data hazard caused by **beq/bne**, we need to stall the later instruction as well as flushing the space between the previous instruction and stalled instruction. Also, we carefully design the detection part to avoid the stalled instruction being flushed due to the successful **jump** or **beq/bne** instruction.

```
    always @(*) begin
        if(ID_EXMemRead && (ID_EXrt == IF_IDrs || ID_EXrt == IF_IDrt)) begin
            Stall = 1'b1;
            Flush = 1'b1;
        end

        else if (IDBranchEq || IDBranchNe) begin
        //If a comparison register is a destination of immediately preceding ALU instruction (R type)
            if(ID_EXRegWrite && ID_EXRegisterRd && (IF_IDrs == ID_EXRegisterRd
|| IF_IDrt == ID_EXRegisterRd)) begin
                Stall = 1'b1;
                Flush = 1'b1;
            end
            //or 2nd preceding load instruction
            else if (EX_MEMMemRead && EX_MEMRegisterRd && (IF_IDrs == EX_MEMRegisterRd
|| IF_IDrt == EX_MEMRegisterRd ))begin
                Stall = 1'b1;
                Flush = 1'b1;
            end
            else begin
            Stall = 1'b0;
            Flush = 1'b0;
            end
        end

        //If a comparison register is a destination of immediately preceding load instruction
        //Dealt by the first hazard situation...
```

Figure 18: Part of the code of Hazzard Detection Unit

5.2 IF/ID flush

One of our flushing unit, which is used to deal with control hazzard caused by any kind of "jumping" instruction, flushing the IFID pipeline register when a **beq/bne** or **jump** is taken, is set outside the hazzard detection unit. Reason for setting outside hazard detection unit is that we only need it happen when any **jump** or **beq/bne** instruction is going to happen. So a wire called $ifbranchjump$ will detect this situation and flush the instruction after **beq/bne** or **jump**. Furthermore, a $!stall$ is used to prevent the stalled instruction being flushed when the equal accidentally output a one (when the actual result need longer time to get).

```
assign ifidflush = ifbranchjump && (!stall);
```

Figure 19: Part of the code of IFID Flush.

6 SSD and Top Module Design

In this part, we will introduce the design of our SSD and how we output the results on FPGA board. We can look at our FPGA board at first.

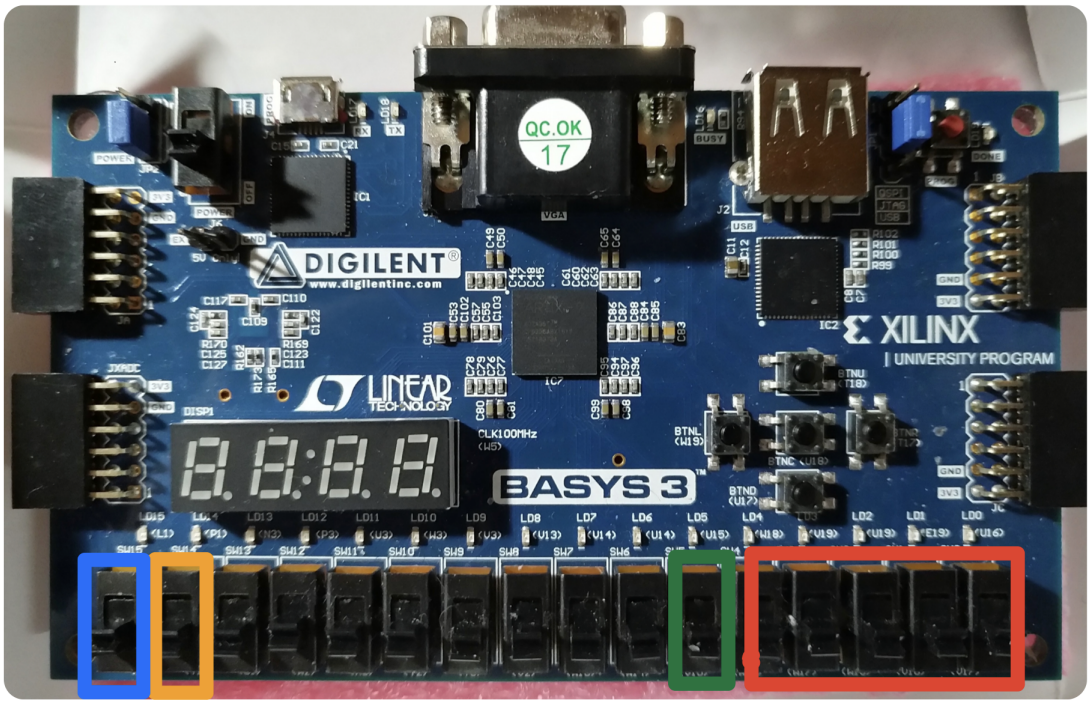

Figure 20: The picture of the FPGA board we use

To control this FPGA board, highlighted switches are needed.

- The switch in blue represents the input "clk" signal of the module pipelined processor. It can be controlled by ourselves.
- The switch in orange represents the "SSDoutupper" signal. When its value is 1, the SSD outputs the first 4-bit value in the register.When its value is 0, the SSD outputs the last 4-bit value in the register.
- The five switches in red are used to determine which register we want to read. It is the input 5-bit signal "Input_Readreg" of the module pipelined processor.
- The switch in green is used to determine whether we output the address of PC or the value in registers on SSD. When its value is 1, SSD will plot the address of PC no matter what value red switches are. When its value is 0, SSD will plot the value in the chosen register.

To make the numbers shown on the SSD correct, we use three new components: ring counter, clock divider and SSD. Some parts of their codes are shown as follow. The complete codes of the three components can be found in the Appendix.

```
module Clockdivider(input clk,
output reg clkdivide
    );
    reg [17:0] count;
    always@ (posedge clk)
    begin
    if (count==0) begin count<=count+1; clkdivide<=1;end
    else if (count<199999)begin count<=count+1;clkdivide<=0;end
    else count<=0;
    end
```

Figure 21: Part of the codes of the clock divider

```
module Ringcounter(input clk,
output reg [3:0] SSDdigitchoice

    );
    always@(posedge clk)begin
    if (SSDdigitchoice==4'b1110) SSDdigitchoice<=4'b1101;
    else if (SSDdigitchoice==4'b1101) SSDdigitchoice<=4'b1011;
    else if (SSDdigitchoice==4'b1011) SSDdigitchoice<=4'b0111;
    else  SSDdigitchoice<=4'b1110;
    end
```

Figure 22: Part of the codes of the ring counter

The ring counter and the clock divider used to help the number shown on the SSD stable. Their input signal clk is the clock cycle of the FPGA board.

```
case (SSDinput)
    4'b0000: SSDdigit=7'b1000000;
    4'b0001: SSDdigit=7'b1111001;
    4'b0010: SSDdigit=7'b0100100;
    4'b0011: SSDdigit=7'b0110000;
    4'b0100: SSDdigit=7'b0011001;
    4'b0101: SSDdigit=7'b0010010;
    4'b0110: SSDdigit=7'b0000010;
    4'b0111: SSDdigit=7'b1111000;
    4'b1000: SSDdigit=7'b0000000;
    4'b1001: SSDdigit=7'b0010000;
    4'b1010: SSDdigit=7'b0001000;
    4'b1011: SSDdigit=7'b0000011;
    4'b1100: SSDdigit=7'b1000110;
    4'b1101: SSDdigit=7'b0100001;
    4'b1110: SSDdigit=7'b0000110;
    4'b1111: SSDdigit=7'b0001110;
    endcase
end
```

Figure 23: Part of the codes of the SSD

The component SSD outputs a 7-bit output signal to control the display number of SSD.

By using these components, we succeed to plot our result of pipeline processor on the FPGA board.

7 Simulation Scenario

For single cycle processor, we use **10ns** as the half length of a clock in the testbench.

For pipeline cycle processor, we manually add a changing integer that will be add 1 whenever our clk changes as the half length of a clock in the testbench.

Details about test bench design can be found in Appendix.

8 Textual Result

In this part, we show part of the function of our program. To avoid repetition, for each design, we present only part of the results of the simulation. Further details are available in Appendix.

8.1 Textual Result of Single Cycle Design

The following PC value shows the jump instruction is successful.

```
------------------------------------------------------------
Time:         580 ns, Clock = 0, PC = 0x00000074
[$s0] = 0x00000037, [$s1] = 0x00000037, [$s2] = 0x00000037
[$s3] = 0x00000020, [$s4] = 0x00000001, [$s5] = 0x00000000
[$s6] = 0x00000000, [$s7] = 0x00000000, [$t0] = 0x00000020
[$t1] = 0x00000037, [$t2] = 0x00000000, [$t3] = 0x00000000
[$t4] = 0x00000000, [$t5] = 0x00000000, [$t6] = 0x00000000
[$t7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
------------------------------------------------------------
Time:         590 ns, Clock = 1, PC = 0x0000005c
[$s0] = 0x00000037, [$s1] = 0x00000037, [$s2] = 0x00000037
[$s3] = 0x00000020, [$s4] = 0x00000001, [$s5] = 0x00000000
[$s6] = 0x00000000, [$s7] = 0x00000000, [$t0] = 0x00000020
[$t1] = 0x00000037, [$t2] = 0x00000000, [$t3] = 0x00000000
[$t4] = 0x00000000, [$t5] = 0x00000000, [$t6] = 0x00000000
[$t7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
------------------------------------------------------------
```

And the following PC value change shows that the **beq** has been successfully taken as currently $\$s4$ is not one any more (the last picture shows it was one).

```
------------------------------------------------------------
Time:         680 ns, Clock = 0, PC = 0x0000006c
[$s0] = 0x00000037, [$s1] = 0x00000037, [$s2] = 0x00000037
[$s3] = 0x00000020, [$s4] = 0x00000000, [$s5] = 0x00000000
[$s6] = 0x00000000, [$s7] = 0x00000000, [$t0] = 0x00000020
[$t1] = 0x00000037, [$t2] = 0x00000000, [$t3] = 0x00000000
[$t4] = 0x00000000, [$t5] = 0x00000000, [$t6] = 0x00000000
[$t7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
------------------------------------------------------------
Time:         690 ns, Clock = 1, PC = 0x000000ac
[$s0] = 0x00000037, [$s1] = 0x00000037, [$s2] = 0x00000037
[$s3] = 0x00000020, [$s4] = 0x00000000, [$s5] = 0x00000000
[$s6] = 0x00000000, [$s7] = 0x00000000, [$t0] = 0x00000020
[$t1] = 0x00000037, [$t2] = 0x00000000, [$t3] = 0x00000000
[$t4] = 0x00000000, [$t5] = 0x00000000, [$t6] = 0x00000000
[$t7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
------------------------------------------------------------
```

8.2 Textual Result of Pipeline Design

The first *addi* instruction doesn't take place as soon as the program starts. The register $\$t0$ get the value when the pc jumps to $0x10$. This shows the property of the pipeline processor.

```
------------------------------------------------------------
Time:          3, CLK = 1, PC = 0x00000010
[$s0] = 0x00000000, [$s1] = 0x00000000, [$s2] = 0x00000000
[$s3] = 0x00000000, [$s4] = 0x00000000, [$s5] = 0x00000000
[$s6] = 0x00000000, [$s7] = 0x00000000, [$t0] = 0x00000000
[$t1] = 0x00000000, [$t2] = 0x00000000, [$t3] = 0x00000000
[$t4] = 0x00000000, [$t5] = 0x00000000, [$t6] = 0x00000000
[$t7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
------------------------------------------------------------
Time:          4, CLK = 0, PC = 0x00000010
[$s0] = 0x00000000, [$s1] = 0x00000000, [$s2] = 0x00000000
[$s3] = 0x00000000, [$s4] = 0x00000000, [$s5] = 0x00000000
[$s6] = 0x00000000, [$s7] = 0x00000000, [$t0] = 0x00000020
[$t1] = 0x00000000, [$t2] = 0x00000000, [$t3] = 0x00000000
[$t4] = 0x00000000, [$t5] = 0x00000000, [$t6] = 0x00000000
[$t7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
------------------------------------------------------------
```

The following result shows the successful use of the lw and sw. We can see the $\$s7$ is changed from $0x57$ to $0x37$. This shows the previous **sw** and this **lw** is successful.

```
------------------------------------------------------------
Time:         15, CLK = 1, PC = 0x0000003c
[$s0] = 0x00000037, [$s1] = 0x00000057, [$s2] = 0xfffffff9
[$s3] = 0x00000000, [$s4] = 0x00000000, [$s5] = 0x00000000
[$s6] = 0x00000000, [$s7] = 0x00000000, [$t0] = 0x00000020
[$t1] = 0x00000037, [$t2] = 0x00000000, [$t3] = 0x00000000
[$t4] = 0x00000000, [$t5] = 0x00000000, [$t6] = 0x00000000
[$t7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
------------------------------------------------------------
Time:         16, CLK = 0, PC = 0x0000003c
[$s0] = 0x00000037, [$s1] = 0x00000037, [$s2] = 0xfffffff9
[$s3] = 0x00000000, [$s4] = 0x00000000, [$s5] = 0x00000000
[$s6] = 0x00000000, [$s7] = 0x00000000, [$t0] = 0x00000020
```

8.3 The Result of the Bonus Part of Pipeline Design

For the bonus part, we need to use stall and forwarding to detect whether **beq/bne** is taken, the following is an example when the program encounter the *beq* $\$s4$, \$0, $EXIT$ for the first time. The program stalled itself to find the answer and go on reading the next instruction later.

```
------------------------------------------------------------
Time:         20, CLK = 1, PC = 0x00000040
[$s0] = 0x00000037, [$s1] = 0x00000037, [$s2] = 0x00000000
[$s3] = 0x00000020, [$s4] = 0x00000000, [$s5] = 0x00000000
[$s6] = 0x00000000, [$s7] = 0x00000000, [$t0] = 0x00000020
[$t1] = 0x00000037, [$t2] = 0x00000000, [$t3] = 0x00000000
[$t4] = 0x00000000, [$t5] = 0x00000000, [$t6] = 0x00000000
[$t7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
------------------------------------------------------------
Time:         21, CLK = 0, PC = 0x00000040
[$s0] = 0x00000037, [$s1] = 0x00000037, [$s2] = 0x00000000
[$s3] = 0x00000020, [$s4] = 0x00000000, [$s5] = 0x00000000
[$s6] = 0x00000000, [$s7] = 0x00000000, [$t0] = 0x00000020
[$t1] = 0x00000037, [$t2] = 0x00000000, [$t3] = 0x00000000
[$t4] = 0x00000000, [$t5] = 0x00000000, [$t6] = 0x00000000
[$t7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
------------------------------------------------------------
Time:         21, CLK = 1, PC = 0x00000040
[$s0] = 0x00000037, [$s1] = 0x00000037, [$s2] = 0x00000000
[$s3] = 0x00000020, [$s4] = 0x00000000, [$s5] = 0x00000000
[$s6] = 0x00000000, [$s7] = 0x00000000, [$t0] = 0x00000020
[$t1] = 0x00000037, [$t2] = 0x00000000, [$t3] = 0x00000000
[$t4] = 0x00000000, [$t5] = 0x00000000, [$t6] = 0x00000000
[$t7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
------------------------------------------------------------
Time:         22, CLK = 0, PC = 0x00000040
[$s0] = 0x00000037, [$s1] = 0x00000037, [$s2] = 0x00000000
[$s3] = 0x00000020, [$s4] = 0x00000000, [$s5] = 0x00000000
[$s6] = 0x00000000, [$s7] = 0x00000000, [$t0] = 0x00000020
[$t1] = 0x00000037, [$t2] = 0x00000000, [$t3] = 0x00000000
[$t4] = 0x00000000, [$t5] = 0x00000000, [$t6] = 0x00000000
[$t7] = 0x00000000, [$t8] = 0x00000000, [$t9] = 0x00000000
------------------------------------------------------------
Time:         22, CLK = 1, PC = 0x00000044
```

9 Conclusion and discussion

In this project, we implements single cycle and pipeline version of MIPS processor in Verilog. The design supports almost most R-type, I-type and J-type MIPS instructions, such as **lw**(load word), **sw**(store word), **beq**(branch equal), **bne**(branch not equal) and **j**(jump). Through this project, we get a better knowledge about the single cycle and pipeline MIPS processor.

When working on the project, we find that both implementations have their advantages and disadvantages. For a single cycle processor, though it is easier to complement, it wastes lots of time since only one part of the processor will be working in one cycle. For a pipeline processor, it is harder to complement. Complex problems such as the various hazard should be handled with great care and effort. However, it is more time efficient since all of its parts are working together for most time.

10 Appendix

10.1 The Simulation Results we get

10.1.1 The Instruction Memory we use

ca65pipeline/InstructionMem$_f or_P 2_D emo_u pdated.txt$

10.1.2 The Result of Single Cycle Design

ca65textualresultsingle.txt

10.1.3 The Result of Pipeline Design

ca65textualresultpipeline.txt

10.1.4 The Instruction Memory we use for Bonus

ca65pipeline/InstructionMem$_f or_P 2_D emo_B onus.txt$

10.2 The Result of the Bonus Part of Pipeline Design

ca65pipeline/bonustextualresult.txt

10.3 Codes for Single Cycle Design

10.3.1 ALU

verilogsinglecycle/ALU.v

10.3.2 ALU Control

verilogsinglecycle/ALUControl.v

10.3.3 Control

verilogsinglecycle/Control.v

10.3.4 Data Memory

verilogsinglecycle/dataMemory.v

10.3.5 Instruction Memory

verilogsinglecycle/instructionMemory.v

10.3.6 PC

verilogsinglecycle/PC.v

10.3.7 Register

verilogsinglecycle/register.v

10.3.8 Signal Extend

verilogsinglecycle/signExtend.v

10.3.9 Single Cycle Processor

verilogsinglecycle/singleCycle.v

10.3.10 Test Bench we use

verilogsinglecycle/test$_b$*ench.v*

10.3.11 The Instruction Memory we use

ca65singlecycle/InstructionMem$_f$*or*$_P$2$_D$*emo*$_u$*pdated.txt*

10.4 Codes for Pipeline Design

10.4.1 ALU

verilogpipeline/ALU.v

10.4.2 ALU Control

verilogpipeline/ALUControl.v

10.4.3 Control

verilogpipeline/Control.v

10.4.4 Data Memory

verilogpipeline/DataMemory.v

10.4.5 Instruction Memory

verilogpipeline/InstructionMemory.v

10.4.6 IF/ID Register

verilogpipeline/IF$_I D.v$

10.4.7 ID/EX Register

verilogpipeline/ID$_E X.v$

10.4.8 EX/MEM Register

verilogpipeline/EX$_M EM.v$

10.4.9 MEM/WB Register

verilogpipeline/MEM$_W B.v$

10.4.10 Forwarding Unit

verilogpipeline/ForwardingUnit.v

10.4.11 Hazard Detection Unit

verilogpipeline/HazardDetectionUnit.v

10.4.12 PC

verilogpipeline/PC.v

10.4.13 Registers

verilogpipeline/Register.v

10.4.14 Signal Extend

verilogpipeline/SignExtend.v

10.4.15 32-bit 3 to 1 MUX

verilogpipeline/threetwonemux32.v

10.4.16 Pipeline Processor

verilogpipeline/PipelinedProcessor.v

10.5 Codes for SSD Display

10.5.1 Clock Divider

verilogpipeline/Clockdivider.v

10.5.2 Ring Counter

verilogpipeline/Ringcounter.v

10.5.3 SSD

verilogpipeline/SSDfour.v