Project 3
ECE 485: Computer Organization and Design

Alan Palayil and Zepei Cen
Professor: Won-Jae Yi

Acknowledgement: I acknowledge all works including figures, codes and writings belong to me and/or persons who are referenced. I understand if any similarity in the code, comments, customized program behavior, report writings and/or figures are found, both the helper (original work) and the requestor (duplicated/modified work) will be called for academic disciplinary action.

Electronic Signature: ***Alan Palayil*** ***A20447935*** (12/04/2022)

Zepei Cen ***A20437716*** (12/04/2022)

Contents

Abstract:

In this project, we were asked to create a custom 32-bit RISC processor which is a stripped-down MIPS processor. The implementation first starts by implementing datapath first without memory and control interface. The design needs to be able to process a 5-stage MIPS pipeline with each stage being separated in equal amounts of clock cycle time. The design also needs to have Instruction Fetch from memory (IF); Instruction Decode and read register (ID); Execute Operation/ calculate address (EX); Memory Access operand (MEM); and Write Back result to register (WB) in each stage. Referring to the minimum subset of the ISA to create our own clock cycle per stage, while equally assigning them between each stage.

We also tried to implement the Forward Unit and Hazard Detection Unit to our 32-bit RISC processor design. This implementation overcomes hazards with the sample data inputs within a test bench.

Introduction:

The purpose of this Project 3 is to combine the knowledge of datapaths as well as the knowledge from the previous 2 projects to build a custom 32-bit RISC processor, a stripped-down MIPS processor. The goal of this project is to have a hands-on practical approach to computer architecture design, and understanding of how to implement a single cycle data path and as well as the problems that arise with them.

In this project, we have to design and implement the required 32-bit MIPS processor with a minimal instruction set from the MIPS ISA with datapath (single cycle) pipelining utilizing VHDL. After following the requirements and designing the model, we test the design using our own test bench.

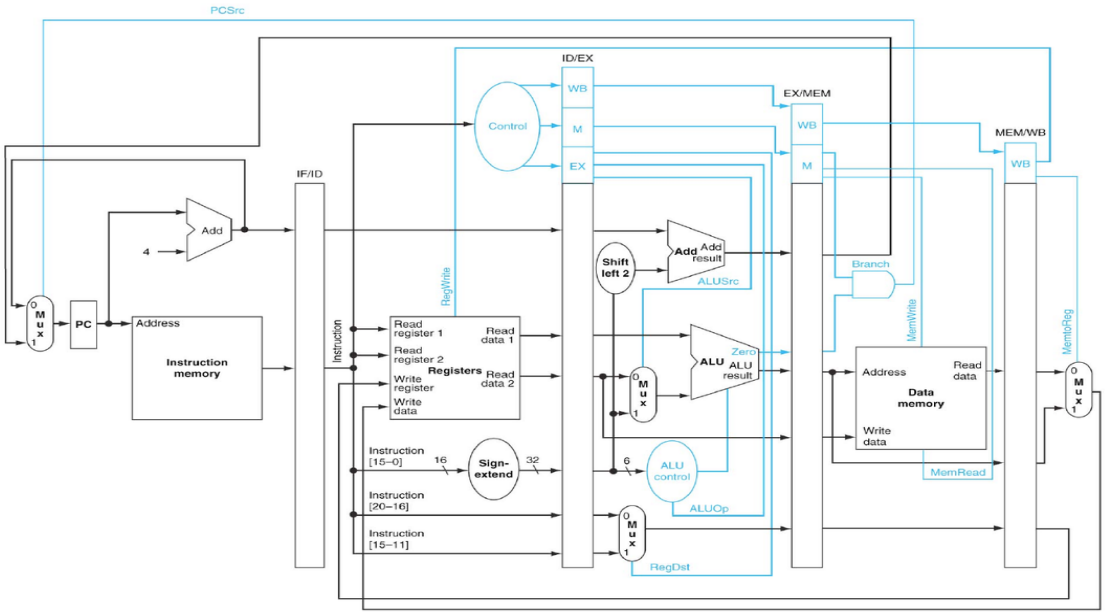
System Design:

i. Description of the Design

The idea was built over a 16-bit MIPS processor and worked our way up from there. We knew that the project would be to design a 32-bit RISC processor, which is a stripped-down MIPS processor, knowing that we did research first on a MIPS processor, and looked at a design for a 16-bit one and scaled it up.

When looking at this we determined all the various parts needed to create said processor. Looking at the provided example from the project outline we knew that we would need a VHDL design for the following:

- MUXs
- Adders
- Control Unit
- Sign Extend
- ALU
- Data Memory
- Program Counter
- Instruction Memory
- Registers
- Shift Left 2 Unit
- ALU Control
- Datapath



Using the sample design of the architecture from the project

Table I. Required MIPS Instruction Set

Code Sequence	OpCode [31:26]	Function Field [5:0]	Instruction	Operation
1	000000	100010	sub	sub $s2, $s1, $s3
2	000000	100100	and	and $t2, $s2, $t5
3	000000	100101	or	or $t2, $s0, $t2
4	000000	100000	add1 (RCA)	add1 $s3, $t1, $t0
5	100011	-	lw	lw $t3, 100($s2)
6	001000	-	addi	addi $s4, $t3, 200
7	101011	-	sw	sw $t1, 100($t2)
8	000000	100111	nor	nor $t1, $s1, $t0
9	000000	101010	slt	slt $t1, $s2, $s2
10	000010	-	j	j $2500

For the functionality and the handling of the instructions we based it off the table

ii. Description of the Datapath

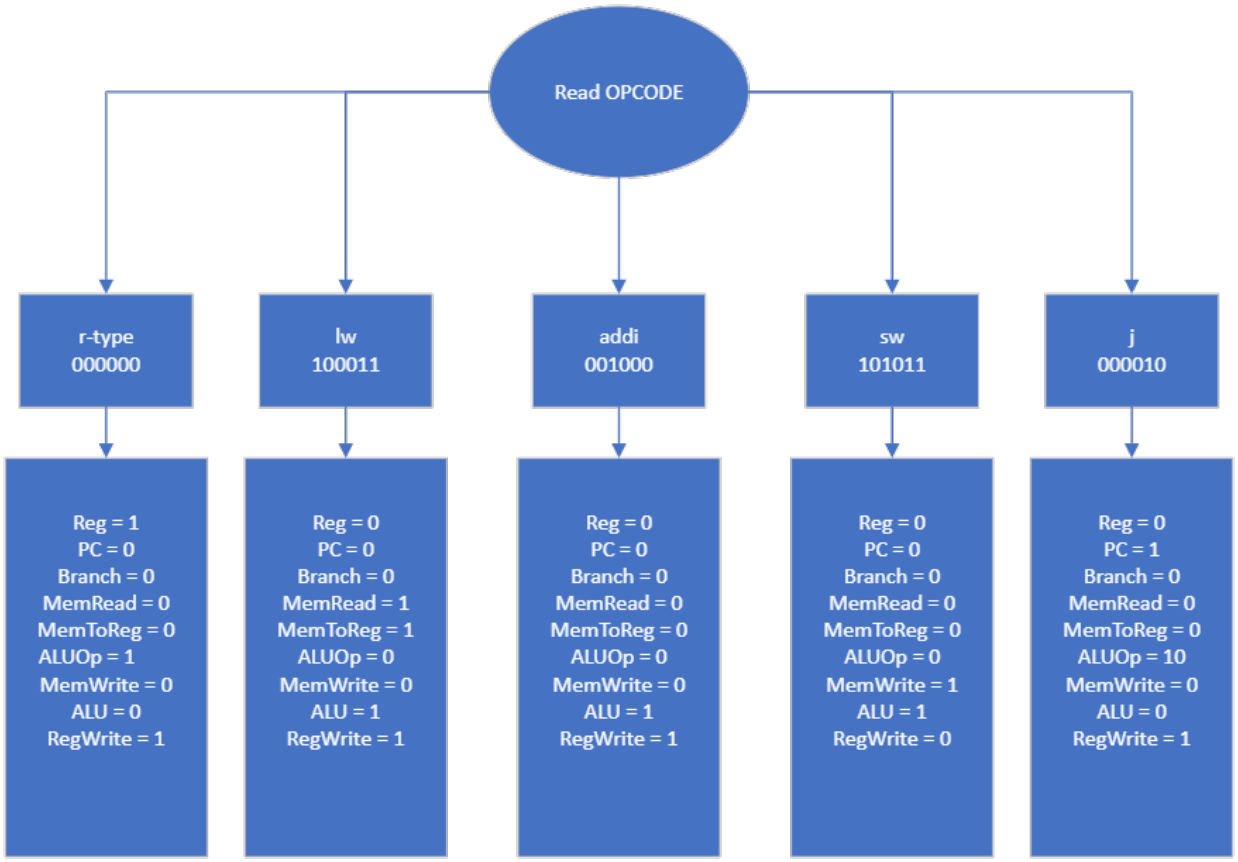
The Datapath that we built around was a single cycle datapath. A single cycle datapath implementation is executed in a one clock cycle where each component of the processor can only be used once. There are advantages and disadvantages for single cycle datapaths, the main advantage being single cycle data paths are easier to implement , but this comes at the cost of speed being determined by the longest memory access which tends to be slower than registers. We create signals for Shift Left-2, Sign Extend, MUX, Data Memory, Registers, Program Counter, ALU, and Adder. The signals are all port mapped into three categories: input, control, and output signals.

iii. Description of the Control Logic design

The Control Unit is part of our MIPS processor, its main function is to direct the operations of the processor. Our design for the control logic is completely in reference to the standard MIPS32 instruction. The control unit is given two input signals: one for the instruction and the other for ZeroCarry. The output signals are all used to set the behavior for the control unit. We set the data for each operation such as AND, OR, Add, Subtract, Shift Left/Right, Jump, etc. with a specific instruction with an if-else condition for each of the operations in a behavioral model.

iv. Overview of the Block Diagram (System Flowchart)

The System Flowchart



v. Other features of the Design

- We have added a top-level to the design of our MIPS processor which consists of the memory block communicating through a bidirectional Data bus, an Address bus, and a few control lines. This feature fetches instructions from the external memory and executes these instructions.
- We did implement Forward Unit which is used for data hazards for the optimization in pipelined MIPS processors. Forward Unit deficits the limit performance which usually occur due to pipeline stalls. Along with that we implemented the Data Hazard Detection Unit controls the PC and IF/ID registers along with MUX that chooses between real control values and deasserts the control fields if the load-use hazard test.

Simulation Results and Discussion:

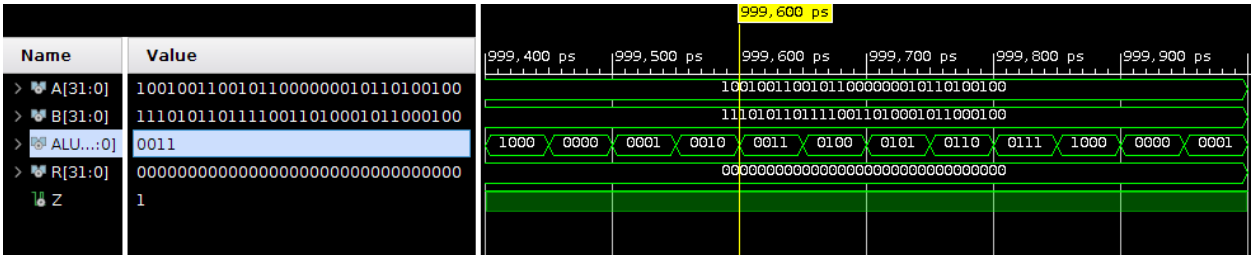
i. Description of Test cases for our design

- For our design due to time constraints, we only created three testbenches: 32-bit ALU, Sign Extend, and Word Align. In the 32-ALU testbench, we set up two input signals A and B and check at the operation used every 50ps.
- For the Sign Extend testbench, we set up signals A and B after which we set two set cases in order to check the validity of the testbench.
- In the Word Align testbench, we set up signal A and test the case A or B. All the testbenches were cross-checked using the waveform generator.

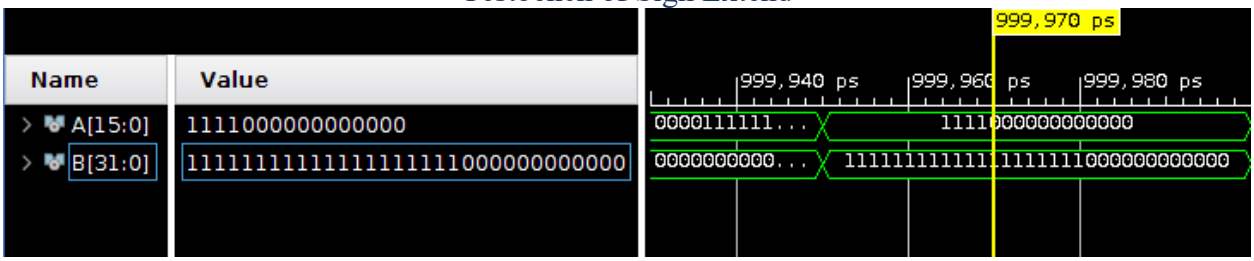
ii. Descriptions of each screenshot

The screenshots below are generated of the testbenches and show how they fluctuate between 0 and 1. All the screenshots are taken using Vivado.

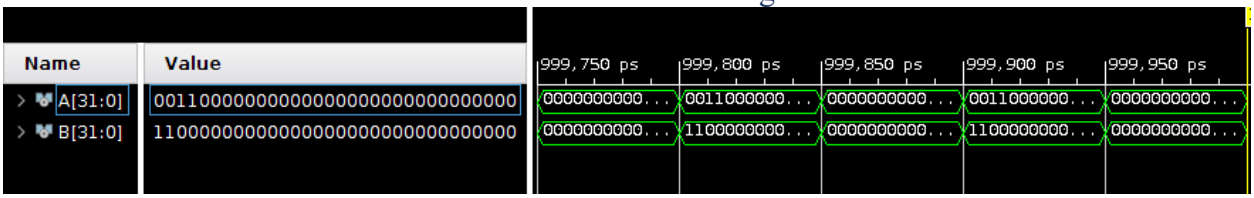
Testbench of 32-bit ALU



Testbench of Sign Extend



Testbench of Word Align



iii. Discussion of any improvements or additional features

- Optimization is definitely an improvement that can be made. The code can be shrunken and, in some cases, can be reduced and some parts can be removed with better pipeline, but the goal was to produce something that would work.
- Currently the datapath is a single cycle datapath which is simpler , but slow compared to a multi cycle datapath . So, a possible improvement could be a shift from a single cycle datapath to a multi cycle datapath.

iv. Discussion of the optimization of our design

- The goal for this project was not to optimize but to create a functional processor , and because of that our focus was not to optimize the anything but to actually have something to show for but there are a few things that can be improved on as listed above.

v. Discussion of any issues faced and improvements that could be made

- We ran into an issue with library compatibility , initially when developing the ALU we ran into issues with the code giving an area of not knowing how to handling unsigned values , after looking into this we realized to solve this we needed to import the following
    1. use IEEE.STD_LOGIC_arith.ALL.
    2. use IEEE.STD_LOGIC_UNSIGNED.ALL.
- Considering the practical approach of building our Control Unit there might be better ways to create a model with the possibility of dynamic instruction for each operation instead of hard coding them.

Conclusion:

The project gave us an insight on the practical approach to set up a 32-bit MIPS using all the components using various behavioral, structural, and testbench models to run under the Waveform Generator. While working on this, we learned how difficult it is to implement a full

MIPS processor. While building all the testbenches we were able to make sure to compare our results with the expected results and achieved the values that we wanted. Another thing that we learned from this project was seeing how the implementation of the previous project can be transferred over and brought over and used for future projects as for project 3 we used the example of the adders from project 2 where it was split into smaller chunks, and we followed the same process for our ALU in this project. This allowed us to unit evaluate our ALU and know that our ALU was never the component that was causing the issue. As this is our final project it served as a perfect capstone of all the skills learned in previous projects and helped us further our knowledge of VHDL and helped us understand the process of creating a processor.

Distribution of Work:

The following are the components worked and designed by us:

Alan Palayil	• Add • Control Unit • MUX • Program Counter • Register • Sign Extend • Top Level • Testbench Sign Extend • Testbench Word Align • Word Align
Zepei Cen	• 1-bit ALU • 4-bit ALU • 32-bit ALU • Data Path • Data Memory • Forward Unit • Hazard Detection Unit • Instruction Memory • RCA • Testbench 32-bit ALU

List of References:

- VHDL Tutorial by van der Spiegel, VHDL Tutorial by Ashenden, ModelSim Tutorial, and Vivado Tutorial
    - Taught me how to set up port maps. This is important for taking in the correct inputs to get the desired outputs.
- https://allaboutfpga.com/4-bit-ripple-carry-adder-vhdl-code/
    - Taught me how to reference array indices in port maps. This was important because using arrays simplified the overall code.
- https://www.ics.uci.edu/~jmoorkan/vhdlref/vhdl.html
    - Gave me multiple examples on how to set up signals and port maps. This was important so I could figure out what exactly was needed for my code.
- https://www.ics.uci.edu/~jmoorkan/vhdlref/arrays.html
    - This taught me how to assign values to an array in different ways. This was important for initializing my arrays.
- https://stackoverflow.com/questions/40723529/assign-values-to-an-array-partially-in-vhdl
    - Taught me how to partially choose a specific range of indices from an array. This was important for when doing port maps to only choose the indices required.
- https://stackoverflow.com/questions/42544675/can-i-access-2-indices-of-an-array-at-the-same-time-vhdl/42549118
    - Taught me another way to choose different indices from an array. This was also important for port mapping the specific indices required.
- https://www.nandland.com/vhdl/examples/example-array-type-vhdl.html
    - Taught me how initialize signal arrays. This was important for setting up the signals in the test benches.
- https://www.youtube.com/watch?v=mqqW_EM34cU
    - Taught me how the waveform generator for 32-bit MIPS testbenches to set up and how each component can be evaluated.

Appendix:

i. Entire Source Code of project with comments

Behavioral ADD

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ADD is
    generic(N : integer := 32);
    port(   A : in  std_logic_vector (N-1 downto 0);
            B : in  std_logic_vector (N-1 downto 0);
            S : out std_logic_vector (N-1 downto 0)
        );
end entity;

architecture Behavioral of ADD is
begin
    S <= std_logic_vector(signed(A) + signed(B));
end architecture;
```

Structural 1-bit ALU

```
library ieee;
use ieee.std_logic_1164.all;

entity ALU_1 is
    port(   X : in std_logic;
            Y : in std_logic;
            Ci : in std_logic;
            ALUOp: in STD_LOGIC_VECTOR(3 downto 0);
            Co : out std_logic;
            R : out std_logic);
end ALU_1;

architecture structural of ALU_1 is

    component RCA
        port (  a, b, Ci : in std_logic;
            Co, Sum : out std_logic);
    end component;

    signal sig_a, sig_b, sig_Ci, sig_clock, sig_less: std_logic;
    signal sig_adder, sig_and, sig_or, sig_Co, sig_slt: std_logic;

begin
```

```
    process(ALUOp, a, b) is
    begin
        if ALUOp(3) = '0' then
            sig_a <= a;
        else
            sig_a <= not a;
        end if;
        if ALUOp(2) = '0' then
            sig_b <= b;
        else
            sig_b <= not b;
        end if;
    end process;

    --set signals for mux
    adder: RCA port map (a => sig_a, b => sig_b, Ci => Ci, Co => sig_Co, Sum =>
adder_op);
    and_op <= a AND b;
    or_op: <= a or b;

    process(sig_adder, sig_and, sig_or, ALUOp) is
    begin
    --set mux
        case ALUOp(1 downto 0) is
            when "00" => R <= and_op; Co <= '0';
            when "01" => R <= or_op; Co <= '0';
            when "10" => R <= adder_op; Co <= sig_Co;
            when "11" => R <= adder_op; Co <= sig_Co;
            when others =>  R <= '0'; Co <= '0';
        end case;
    end process;
end structural;
```

Structural 4-bit ALU

```
library ieee;
use ieee.std_logic_1164.all;

entity ALU_4 is
  port( a, b, ALU_control: in STD_LOGIC_VECTOR (3 downto 0);
    Ci: in STD_LOGIC;
    Co: out STD_LOGIC;
    output: out STD_LOGIC_VECTOR (3 downto 0));
end ALU_4;

architecture structural of ALU_4 is
```

```
    --define 1 bit ALU
    component ALU_4
      port( a, b, Ci: in STD_LOGIC;
        ALU_control: in STD_LOGIC_VECTOR(3 downto 0);
        Co, output: out STD_LOGIC);
    end component;

    signal carry: STD_LOGIC_VECTOR(2 downto 0);

    --use 1-bit ALU to complete 4-bit ALU
    begin
    U1: ALU_4 port map(a => a(0), b => b(0), Ci => Ci => ALU_control =>
ALU_control, Co => carry(0), output => output(0));
    U2: ALU_4 port map(a => a(1), b => b(1), Ci => carry(0) => ALU_control =>
ALU_control, Co => carry(1), output => output(1));
    U3: ALU_4 port map(a => a(2), b => b(2), Ci => carry(1) => ALU_control =>
ALU_control, Co => carry(2), output => output(2));
    U4: ALU_4 port map(a => a(3), b => b(3), Ci => carry(2) => ALU_control =>
ALU_control, Co => Co, output => output(3));

end structural;
```

Structural 32-bit ALU

```
library ieee;
use ieee.std_logic_1164.all;

entity ALU_32 is
  port( a, b: in STD_LOGIC_VECTOR (31 downto 0);
    ALU_control: in STD_LOGIC_VECTOR (3 downto 0);
    Co, zero: out STD_LOGIC;
    output: out STD_LOGIC_VECTOR (31 downto 0));
end ALU_32;

architecture structural of ALU_32 is

    --define 4-bit ALU
    component ALU_4
      port( a, b, ALU_control: in STD_LOGIC_VECTOR (3 downto 0);
        Ci: in STD_LOGIC;
        Co: out STD_LOGIC;
        output: out STD_LOGIC_VECTOR (3 downto 0));
    end component;

    signal carry: STD_LOGIC_VECTOR(8 downto 0);
    signal sig_Co, sig_zero: STD_LOGIC;
    signal sig_out, sig_slt, sig_out1: STD_LOGIC_VECTOR(31 downto 0);
```

```
    begin
        process(ALU_control, carry, sig_Co) is
        begin
            carry(0) <= '0';
            sig_Co <= carry(8);
            if(ALU_control = x"6") or (ALU_control = x"7") then
                carry(0) <= '1';
                sig_Co <= not carry(8);
            end if;
        end process;

            process(sig_out1, sig_zero) is
        begin
            sig_zero <= '0';
            if (sig_out1 = x"00000000") then
                sig_zero <= '1';
            end if;
            end process;

        --use 4-bit ALU to complete 32-bit ALU
        U1: ALU_4 port map(a => a(3 downto 0), b => b(3 downto 0), ALU_control =>
ALU_control, Ci => carry(0) =Co => carry(1), output => sig_out(3 downto 0));
        U2: ALU_4 port map(a => a(7 downto 4), b => b(7 downto 4), ALU_control =>
ALU_control, Ci => carry(1) =Co => carry(2), output => sig_out(7 downto 4));
        U3: ALU_4 port map(a => a(11 downto 8), b => b(11 downto 8), ALU_control
=> ALU_control, Ci => carry(2) =Co => carry(3), output => sig_out(11 downto 8));
        U4: ALU_4 port map(a => a(15 downto 12), b => b(15 downto 12),
ALU_control => ALU_control, Ci => carry(3) =Co => carry(4), output => sig_out(15
downto 12));
        U5: ALU_4 port map(a => a(19 downto 16), b => b(19 downto 16),
ALU_control => ALU_control, Ci => carry(4) =Co => carry(5), output => sig_out(19
downto 16));
        U6: ALU_4 port map(a => a(23 downto 20), b => b(23 downto 20),
ALU_control => ALU_control, Ci => carry(5) =Co => carry(6), output => sig_out(23
downto 20));
        U7: ALU_4 port map(a => a(27 downto 24), b => b(27 downto 24),
ALU_control => ALU_control, Ci => carry(6) =Co => carry(7), output => sig_out(27
downto 24));
        U8: ALU_4 port map(a => a(31 downto 28), b => b(31 downto 28),
ALU_control => ALU_control, Ci => carry(7) =Co => carry(8), output => sig_out(31
downto 28));

        --set output for slt
        process(sig_Co, sig_slt, ALU_control, sig_out)
```

```
        begin
            sig_out1 <= sig_out;
            sig_slt(31 downto 1) <= "00000000000000000000000000000000";
            sig_slt(0) <= sig_Co;
            if ALU_control = x"7" then
                sig_out1 <= sig_slt;
            end if;
        end process;

        Co <= sig_Co;
        output <= sig_out1;
        zero <= sig_zero;

end structural;
```

Behavioral ALU

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ALU is
    generic(N      : integer := 32);
    port(   A      : in std_logic_vector(N-1 downto 0);
            B      : in std_logic_vector(N-1 downto 0);
            ALUOp  : in std_logic_vector(3   downto 0); --9 Operations Used

            R      : out std_logic_vector(N-1 downto 0);
            Z      : out std_logic
        );
end entity;

architecture Behavioral of ALU is
    signal T : std_logic_vector(N-1 downto 0);
begin
    T <=
        std_logic_vector(unsigned(A) +
unsigned(B))                                    after 1 ns when ALUOp = "0000"
else --ADD
        std_logic_vector(unsigned(A) -
unsigned(B))                                    after 1 ns when ALUOp = "0001"
else --SUB
        A
AND  B                                                                  af
ter 1 ns when ALUOp = "0010" else --AND
```

```
        A
OR   B                                                                  af
ter 1 ns when ALUOp = "0011" else --OR
        A
NOR  B                                                                  af
ter 1 ns when ALUOp = "0100" else --NOR
        A NAND
B                                                                   after 1
ns when ALUOp = "0101" else --NAND
        A
XOR  B                                                                  af
ter 1 ns when ALUOp = "0110" else --XOR
        std_logic_vector(shift_left( unsigned(A), to_integer(unsigned(B(10 downto
6))))) after 1 ns when ALUOp = "0111" else --SLL
        std_logic_vector(shift_right(unsigned(A), to_integer(unsigned(B(10 downto
6))))) after 1 ns when ALUOp = "1000" else --SRL
        (others => '0');
    Z <= '1' when (T <= "00000000000000000000000000000000") else '0';
    R <= T;
end architecture;
```

Behavioral Control Unit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ControlUnit is
    port(   I        : in std_logic_vector(31 downto 0);
            ZC       : in std_logic;

            ALUOp    : out std_logic_vector(3 downto 0); --9 Ops
            ALUSrc   : out std_logic;
            Branch   : out std_logic;
            Jump     : out std_logic;
            MemRead  : out std_logic;
            MemToReg : out std_logic;
            MemWrite : out std_logic;
            RegDst   : out std_logic;
            RegWrite : out std_logic
        );
end entity;

architecture Behavioral of ControlUnit is
    signal D : std_logic_vector(11 downto 0);
begin
    D <=
```

```
        "100000000001" when (I(31 downto 26) = "000000" AND I(10 downto 0)  =
"00000100000") else --ADD

        "100000001001" when (I(31 downto 26) = "000000" and I(10 downto 0)  =
"00000100010") else --SUB

        "100000010001" when (I(31 downto 26) = "000000" and I(10 downto 0)  =
"00000100100") else --AND

        "100000011001" when (I(31 downto 26) = "000000" and I(10 downto 0)  =
"00000100101") else --OR

        "100000100001" when (I(31 downto 26) = "000000" and I(10 downto 0)  =
"00000100111") else --NOR

        "100000110001" when (I(31 downto 26) = "000000" and I(5 downto 0)   =
"100110") else -- XOR

        "100000111011" when (I(31 downto 26) = "000000" and I(5 downto 0)   =
"000000") else --SLL

        "100001000011" when (I(31 downto 26) = "000000" and I(5 downto 0)   =
"000010") else --SRL

        "100001001001" when (I(31 downto 26) = "000000" and I(10 downto 0)  =
"00000101010") else --SLT

        "000000000011" when I(31 downto 26) = "001000" else --ADDI

        "000110000011" when I(31 downto 26) = "100011" else --LW

        "000000000110" when I(31 downto 26) = "101011" else --SW

        "000000010011" when I(31 downto 26) = "001100" else --ANDI

        "000000011011" when I(31 downto 26) = "001101" else --ORI

        "001000001000" when I(31 downto 26) = "000100" else --BEQ

        "001000110010" when I(31 downto 26) = "000101" else --BNQ

        "000001001011" when I(31 downto 26) = "001010" else --SLTI

        "010000000000" when I(31 downto 26) = "000010" else --JMP
```

```
        (others =>'0');



    ALUOp    <= D(6 downto 3);  -- 9 Ops
    ALUSrc   <= D(1);
    Branch   <= D(9) AND (ZC XOR I(26));
    Jump     <= D(10);
    MemRead  <= D(8);
    MemToReg <= D(7);
    MemWrite <= D(2);
    RegDst   <= D(11);
    RegWrite <= D(0);
end Behavioral;
```

Behavioral Data Memory

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity DataMemory is
    generic(N        : integer := 32);
    port(   CLK      : in  std_logic;
            NegReset : in  std_logic;
            MemAddr  : in  std_logic_vector(N-1 downto 0);
            MemWrite : in  std_logic;
            MemRead  : in  std_logic;
            DataIn   : in  std_logic_vector(N-1 downto 0);

            DataOut  : out std_logic_vector(N-1 downto 0)
        );
end entity;

architecture Behavioral of DataMemory is
    type mem_type is array(127 downto 0) of std_logic_vector(N-1 downto 0);
    signal mem : mem_type;
begin
    process(CLK,NegReset)
    begin
        if NegReset = '0' then
            mem <= (others => (others => '0'));
        elsif rising_edge(CLK) and MemWrite = '1' then
            mem(to_integer(unsigned(MemAddr))) <= DataIn;
        end if;
    end process;
```

```
    DataOut <= (mem(to_integer(unsigned(MemAddr)))) when MemRead = '1' else
(others => '0');
end architecture;
```

Behavioral Data Path

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity DataPath is
    generic(N : integer := 32);
    port(   CLK      : in  std_logic;
            CTRL     : in  std_logic;
            I        : in  std_logic_vector(31 downto 0);

            ALUOp    : in  std_logic_vector(3 downto 0); --9 Ops
            ALUSrc   : in  std_logic;
            Branch   : in  std_logic;
            Jump     : in  std_logic;
            MemRead  : in  std_logic;
            MemToReg : in  std_logic;
            MemWrite : in  std_logic;
            RegDst   : in  std_logic;
            RegWrite : in  std_logic;

            NI       : out std_logic_vector(31 downto 0);
            ZC       : out std_logic
        );
end entity;

architecture Behavioral of DataPath is
    component WordAlign is
        port(   A   : in  std_logic_vector(31 downto 0);

                B   : out std_logic_vector(31 downto 0)
        );
    end component;

    component SignExtend is
        port(   A   : in  std_logic_vector(15 downto 0);

                B   : out std_logic_vector(31 downto 0)
        );
    end component;

    component MUX is
```

```
    --generic(N : integer);
    port(   X : in  std_logic_vector(N-1 downto 0);
            Y : in  std_logic_vector(N-1 downto 0);
            S : in  std_logic;

            Z : out std_logic_vector(N-1 downto 0)
        );
end component;

component DataMemory is
    port(   CLK      : in std_logic;
            NegReset : in std_logic;
            MemAddr  : in std_logic_vector(N-1 downto 0);
            MemWrite : in std_logic;
            MemRead  : in std_logic;
            DataIn   : in std_logic_vector(N-1 downto 0);

            DataOut  : in std_logic_vector(N-1 downto 0)
        );
end component;

component Registers is
    port(   CLK       : in std_logic;
            NegReset  : in std_logic;
            Addr1     : in std_logic_vector(4 downto 0);
            Addr2     : in std_logic_vector(4 downto 0);
            AddrOut   : in std_logic_vector(4 downto 0);
            DataWrite : in std_logic_vector(N-1 downto 0);
            RegWrite  : in std_logic;

            Reg1      : out std_logic_vector(N-1 downto 0);
            Reg2      : out std_logic_vector(N-1 downto 0)
        );
end component;

component ProgramCounter is
    port(   A    : in  std_logic_vector(31 downto 0);
            CTRL : in  std_logic;
            CLK  : in  std_logic;

            B    : out std_logic_vector(31 downto 0)
        );
end component;

component ADD is
```

```
        port(   A : in  std_logic_vector (N-1 downto 0);
                B : in  std_logic_vector (N-1 downto 0);
                S : out std_logic_vector (N-1 downto 0)
            );
    end component;

    component ALU is
        port(   A      : in std_logic_vector(N-1 downto 0);
                B      : in std_logic_vector(N-1 downto 0);
                ALUOp  : in std_logic_vector(3   downto 0); --9 Operations Used

                R      : out std_logic_vector(N-1 downto 0);
                Z      : out std_logic
            );
    end component;


    constant PCInc          : std_logic_vector(31  downto 0) :=
"00000000000000000000000000000100";

    signal   PCOut          : std_logic_vector(31  downto 0);
    signal   MUX_RegWrite   : std_logic_vector(4   downto 0);
    signal   SignExtend_SLL : std_logic_vector(31  downto 0);
    signal   SLL_ADD        : std_logic_vector(31  downto 0);
    signal   DataRead1_ALU  : std_logic_vector(N-1 downto 0);
    signal   DataRead2_MUX  : std_logic_vector(N-1 downto 0);
    signal   MUX_ALU        : std_logic_vector(N-1 downto 0);
    signal   ALU_DataMemory : std_logic_vector(N-1 downto 0);
    signal   DataMemory_MUX : std_logic_vector(N-1 downto 0);
    signal   MUX_DataWrite  : std_logic_vector(N-1 downto 0);
    signal   ADD0_MUX        : std_logic_vector(31  downto 0);
    signal   MUX_MUX        : std_logic_vector(31  downto 0);
    signal   MUX_PC         : std_logic_vector(31  downto 0);
    signal   ADD1_MUX        : std_logic_vector(31  downto 0);
    signal   SLL_MUX        : std_logic_vector(31  downto 0);
    signal   SLLOut         : std_logic_vector(31  downto 0);
    signal   JumpShift      : std_logic_vector(31  downto 0);

begin

    SLL_MUX <= ADD0_MUX(31 downto 28) & SLLOut(27 downto 0);
    JumpShift <= "000000" & I(25 downto 0);

    MEM         : DataMemory     port map(CLK, CTRL, ALU_DataMemory, MemWrite,
MemRead, DataRead2_MUX, DataMemory_MUX);
```

```
    ALUnit      : ALU            port map(DataRead1_ALU, MUX_ALU, ALUOp,
ALU_DataMemory, ZC);

    ADDBranch   : ADD            port map(ADD1_MUX, SLL_ADD, ADD0_MUX);
    ADDPC       : ADD            port map(PCOut, PCInc, ADD1_MUX);

    MUXALU      : MUX            generic map(32) port map(DataRead2_MUX,
SignExtend_SLL, ALUSrc, MUX_ALU);
    MUXBranch   : MUX            generic map(32) port map(ADD1_MUX, ADD0_MUX,
Branch, MUX_MUX);
    MUXJump     : MUX            generic map(32) port map(MUX_MUX, SLL_MUX, Jump,
MUX_PC);
    MUXMem      : MUX            generic map(32) port map(ALU_DataMemory,
DataMemory_MUX, MemToReg, MUX_DataWrite);
    MUXReg      : MUX            generic map(5)  port map(I(20 downto 16), I(15
downto 11), RegDst, MUX_RegWrite);

    ShiftBranch : WordAlign      port map(SignExtend_SLL, SLL_ADD);
    ShiftJump   : WordAlign      port map(JumpShift, SLLOut);

    ShiftExtend : SignExtend     port map(I(15 downto 0), SignExtend_SLL);
    Reg         : Registers      port map(CLK, CTRL, I(25 downto 21), I(20 downto
16), MUX_RegWrite, MUX_DataWrite, RegWrite, DataRead1_ALU, DataRead2_MUX);

    PC          : ProgramCounter port map(MUX_PC, CTRL, CLK, PCOut);
    NI <= PCOut;
end architecture;
```

Behavioral Instruction Memory

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity InstructionMemory is
    port(   RegAddr : in  std_logic_vector(31 downto 0);
            I       : out std_logic_vector(31 downto 0)
        );
end entity;

architecture Behavioral of InstructionMemory is
    type R is array (0 to 1500) of std_logic_vector(7 downto 0);
    signal IMem : R := (
        --# => "8#",
        others => "00000000"
    );
begin
```

```
    I <= IMem(to_integer(unsigned(RegAddr)))   &
         IMem(to_integer(unsigned(RegAddr)+1)) &
         IMem(to_integer(unsigned(RegAddr)+2)) &
         IMem(to_integer(unsigned(RegAddr)+3));
end architecture;
```

Behavioral MUX

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity MUX is
    generic(N : integer := 32);
    port(   X : in  std_logic_vector(N-1 downto 0);
            Y : in  std_logic_vector(N-1 downto 0);
            S : in  std_logic;

            Z : out std_logic_vector(N-1 downto 0)
        );
end entity;

architecture Behavioral of MUX is
begin
    process(X,Y,S)
    begin
        if    (S = '0') then
                Z <= X;
        elsif (S = '1') then
            Z <= Y;
        end if;
    end process;
end architecture;
```

Behavioral Program Counter

```
--This file defines usages for the program counter

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ProgramCounter is
--define ports used, both input and output use 32 bits
    port(   A    : in  std_logic_vector(31 downto 0);
            CTRL : in  std_logic;
            CLK  : in  std_logic;
```

```
            B    : out std_logic_vector(31 downto 0)
        );
end entity;

architecture Behavioral of ProgramCounter is
begin
    --if set enabled then change B
    process(CLK)
    begin
        if CTRL = '0' then
            B <= (others => '0');
        elsif rising_edge(CLK) then
            B <= A;
        end if;
    end process;
end architecture;
```

Behavioral 1-bit RCA

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


--Create entity for behavioral model initializing inputs and outputs
entity RCA_1b is
    port(   A,B,Cin  : in  std_logic; --Data inputs
            S,Cout   : out std_logic  --Data outputs
        );
end entity;

architecture Behavioral of RCA_1b is
begin
    process (A,B,Cin) is
    begin
        --Define logic for each output
        if    (A = '0' and B = '0' and Cin = '0') then
                S <= '0'; Cout <= '0';
        elsif (A = '0' and B = '0' and Cin = '1') then
                S <= '1'; Cout <= '0';
        elsif (A = '0' and B = '1' and Cin = '0') then
                S <= '1'; Cout <= '0';
        elsif (A = '0' and B = '1' and Cin = '1') then
                S <= '0'; Cout <= '1';
        elsif (A = '1' and B = '0' and Cin = '0') then
                S <= '1'; Cout <= '0';
        elsif (A = '1' and B = '0' and Cin = '1') then
```

```
                S <= '0'; Cout <= '1';
        elsif (A = '1' and B = '1' and Cin = '0') then
                S <= '0'; Cout <= '1';
        elsif (A = '1' and B = '1' and Cin = '1') then
                S <= '1'; Cout <= '1';
        end if;
    end process;
end architecture;
```

Behavioral Registers

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Registers is
    generic(N : integer := 32);
    port(   CLK       : in std_logic;
            NegReset  : in std_logic;
            Addr1     : in std_logic_vector(4 downto 0);
            Addr2     : in std_logic_vector(4 downto 0);
            AddrOut   : in std_logic_vector(4 downto 0);
            DataWrite : in std_logic_vector(N-1 downto 0);
            RegWrite  : in std_logic;

            Reg1      : out std_logic_vector(N-1 downto 0);
            Reg2      : out std_logic_vector(N-1 downto 0)
        );
end entity;

architecture Behavioral of Registers is
    type registers_type is array(0 to 31) of std_logic_vector(N-1 downto 0);
    signal R : registers_type := ((others => (others => '0')));
begin
    process(CLK)
    begin
        if NegReset = '0' then
            R(to_integer(unsigned(AddrOut))) <= (others => '0');
        elsif rising_edge(CLK) and RegWrite = '1' then
            R(to_integer(unsigned(AddrOut))) <= DataWrite;
        end if;
    end process;
    Reg1 <= R(to_integer(unsigned(Addr1)));
    Reg2 <= R(to_integer(unsigned(Addr2)));
end Behavioral;
```

Behavioral Top Level

```
library ieee;
```

```vhdl
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity TopLevel is
    generic(N : integer := 32);
    port(   CLK  : in std_logic;
            CTRL : in std_logic
        );
end entity;

architecture Behavioral of TopLevel is
    component ControlUnit is
        port(   I        : in std_logic_vector(31 downto 0);
                ZC       : in std_logic;

                ALUOp    : out std_logic_vector(3 downto 0); --9 Ops
                ALUSrc   : out std_logic;
                Branch   : out std_logic;
                Jump     : out std_logic;
                MemRead  : out std_logic;
                MemToReg : out std_logic;
                MemWrite : out std_logic;
                RegDst   : out std_logic;
                RegWrite : out std_logic
            );
    end component;

    component DataPath is
        port(   CLK      : in  std_logic;
                CTRL     : in  std_logic;
                I        : in  std_logic_vector(31 downto 0);

                ALUOp    : in  std_logic_vector(3 downto 0); --9 Ops
                ALUSrc   : in  std_logic;
                Branch   : in  std_logic;
                Jump     : in  std_logic;
                MemRead  : in  std_logic;
                MemToReg : in  std_logic;
                MemWrite : in  std_logic;
                RegDst   : in  std_logic;
                RegWrite : in  std_logic;

                NI       : out std_logic_vector(31 downto 0);
                ZC       : out std_logic
            );
```

```
    end component;

    component InstructionMemory is
        port(   RegAddr : in  std_logic_vector(31 downto 0);
                I       : out std_logic_vector(31 downto 0)
            );
    end component;

    signal TL_RegDst, TL_Jump, TL_Branch, TL_MemRead, TL_MemToReg : std_logic;
    signal TL_MemWrite, TL_ALUSrc, TL_RegWrite , TL_ZC     : std_logic;
    signal TL_ALUOp                                        :
std_logic_vector(3 downto 0);
    signal TL_NI, TL_I                                     :
std_logic_vector(31 downto 0);

begin
    CU : ControlUnit port map(
        TL_I,
        TL_ZC,

        TL_ALUOp,
        TL_ALUSrc,
        TL_Branch,
        TL_Jump,
        TL_MemRead,
        TL_MemToReg,
        TL_MemWrite,
        TL_RegDst,
        TL_RegWrite
    );

    DP : DataPath port map(
        CLK,
        CTRL,
        TL_I,

        TL_ALUOp,
        TL_ALUSrc,
        TL_Branch,
        TL_Jump,
        TL_MemRead,
        TL_MemToReg,
        TL_MemWrite,
        TL_RegDst,
        TL_RegWrite,
```

```
        TL_NI,
        TL_ZC
    );

    I : InstructionMemory port map(TL_NI, TL_I);
end architecture;
```

Behavioral Sign Extend

```
--This file defines the sign extend operation
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity SignExtend is
--define ports
    port(   A   : in  std_logic_vector(15 downto 0);

            B   : out std_logic_vector(31 downto 0)
        );
end entity;

architecture Behavioral of SignExtend is
--the most significant bit needs to be copied to the upper 16
begin
    B <=   "0000000000000000" & A when (A(15) = '0') else
           "1111111111111111" & A;
end architecture;
```

Behavioral Word Align

```
--The file defines the shift left 2 operation
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity WordAlign is
    port(   A   : in  std_logic_vector(31 downto 0);
            B   : out std_logic_vector(31 downto 0)
        );
end entity;

architecture Behavioral of WordAlign is
begin
    B <= std_logic_vector(unsigned(A) sll 2);
end architecture;
```

Behavioral Hazard Detection Unit

```
library ieee;
```

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.mips32.all;
entity hazard is
  port (IF_ID_IR      : in  m32_word;
        ID_EX_IR      : in  m32_word;
        ID_EX_Control : in  m32_vector(11 downto 0);
        Nop           : out m32_1bit);
end hazard;

architecture Behavioral of Hazard is

  -- EX Stage signals
  signal ID_EX_Mem_Read       : m32_1bit;
  signal ID_EX_Rt             : m32_5bits;

  -- ID Stage signals
  signal IF_ID_Rt             : m32_5bits;
  signal IF_ID_Rs             : m32_5bits;

  -- Intermediate comparison signals
  signal IF_ID_Rs_Compare     : m32_5bits;
  signal IF_ID_Rs_Is_ID_EX_Rt : m32_1bit;

  signal IF_ID_Rt_Compare     : m32_5bits;
  signal IF_ID_Rt_Is_ID_EX_Rt : m32_1bit;

begin

  -- Check if instruction is a load
  ID_EX_Mem_Read <= ID_EX_Control(7);

  -- Check if next instruction uses the register the load is writing to
  IF_ID_Rs <= IF_ID_IR(25 downto 21);
  IF_ID_Rt <= IF_ID_IR(20 downto 16);

  ID_EX_Rt <= ID_EX_IR(20 downto 16);

  -- Compare ID stage Rs to EX stage Rt
  IF_ID_Rs_Compare <= ID_EX_Rt XOR IF_ID_Rs;
  IF_ID_Rs_Is_ID_EX_Rt <= '1' when (IF_ID_Rs_Compare = (IF_ID_Rs_Compare'range =>
'0')) else '0';

  -- Compare ID stage Rt to EX stage Rt
  IF_ID_Rt_Compare <= ID_EX_Rt XOR IF_ID_Rt;
```

```
  IF_ID_Rt_Is_ID_EX_Rt <= '1' when (IF_ID_Rt_Compare = (IF_ID_Rt_Compare'range =>
'0')) else '0';

  -- Determine if a stall is necessary
  Nop <= ID_EX_Mem_Read AND (IF_ID_Rt_Is_ID_EX_Rt OR IF_ID_Rs_Is_ID_EX_Rt);

end Behavioral;
```

Behavioral Forward Unit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.mips32.all;
entity forward is
  port (ID_EX_IR       : in  m32_word;
        EX_MEM_IR      : in  m32_word;
        EX_MEM_Rd      : in  m32_5bits;
        MEM_WB_Rd      : in  m32_5bits;
        MEM_WB_Jmp_Src : in  m32_1bit;
        EX_MEM_Control : in  m32_vector(11 downto 0);
        MEM_WB_Control : in  m32_vector(11 downto 0);
        Fwd_A_Code     : out m32_2bits;
        Fwd_B_Code     : out m32_2bits);
end forward;
architecture Behavioral of Forward is

  -- EX Stage Signals
  signal EX_MEM_Reg_Write_En   : m32_1bit;
  signal EX_MEM_Func           : m32_6bits;
  signal EX_MEM_Reg_Write      : m32_1bit;
  signal EX_MEM_Jmp_Src        : m32_1bit;
  signal EX_MEM_ALU_Op         : m32_2bits;
  signal EX_MEM_ALU_Full_Code  : m32_byte;
  signal EX_MEM_Rd_Non_Zero    : m32_1bit;
  signal ID_EX_Rs_Is_EX_MEM_Rd : m32_1bit;
  signal ID_EX_Rt_Is_EX_MEM_Rd : m32_1bit;
  signal ALU_A_EX_Compare      : m32_5bits;
  signal ALU_B_EX_Compare      : m32_5bits;

  -- MEM Stage Signals
  signal MEM_WB_Reg_Write_En   : m32_1bit;
  signal MEM_WB_Reg_Write      : m32_1bit;
  signal MEM_WB_Rd_Non_Zero    : m32_1bit;
  signal ID_EX_Rs_Is_MEM_WB_Rd : m32_1bit;
  signal ID_EX_Rt_Is_MEM_WB_Rd : m32_1bit;
  signal ALU_A_MEM_Compare     : m32_5bits;
```

```
  signal ALU_B_MEM_Compare     : m32_5bits;

  -- Others
  signal EX_A_Hazard           : m32_1bit;
  signal EX_B_Hazard           : m32_1bit;
  signal ID_EX_Rt              : m32_5bits;
  signal ID_EX_Rs              : m32_5bits;

begin

  -- Check if instruction in EX stage will write to a register
  EX_MEM_Reg_Write_En <= EX_MEM_Reg_Write AND (NOT EX_MEM_Jmp_Src);

  -- Determine if instruction in EX stage is a jr instruction
  EX_MEM_Func <= EX_MEM_IR( 5 downto  0);

  EX_MEM_Reg_Write <= EX_MEM_Control(8);
  EX_MEM_ALU_Op    <= EX_MEM_Control( 1 downto 0);

  EX_MEM_ALU_Full_Code <= EX_MEM_ALU_Op & EX_MEM_Func;

  with EX_MEM_ALU_Full_Code select
    EX_MEM_Jmp_Src <= '1' when "10000100", -- JR
                      '0' when others;   -- No JR


  -- Check if instruction in MEM stage will write to a register
  MEM_WB_Reg_Write <= MEM_WB_Control(8);
  MEM_WB_Reg_Write_En <= MEM_WB_Reg_Write AND (NOT MEM_WB_Jmp_Src);

  -- Check if destination register in EX stage is non-zero
  EX_MEM_Rd_Non_Zero <= '0' when (EX_MEM_Rd = (EX_MEM_Rd'range => '0')) else '1';


  -- Check if destination register in MEM stage is non-zero
  MEM_WB_Rd_Non_Zero <= '0' when (MEM_WB_Rd = (MEM_WB_Rd'range => '0')) else '1';


  -- Check if ALU A source register in ID stage matches destination register in
EX stage
  ID_EX_Rs <= ID_EX_IR(25 downto 21);

  ALU_A_EX_Compare <= ID_EX_Rs XOR EX_MEM_Rd;
  ID_EX_Rs_Is_EX_MEM_Rd <= '1' when (ALU_A_EX_Compare = (ALU_A_EX_Compare'range
=> '0')) else '0';
```

```
  -- Check if ALU A source register in ID stage matches destination register in
MEM stage
  ALU_A_MEM_Compare <= ID_EX_Rs XOR MEM_WB_Rd;
  ID_EX_Rs_Is_MEM_WB_Rd <= '1' when (ALU_A_MEM_Compare = (ALU_A_MEM_Compare'range
=> '0')) else '0';


  -- Check if ALU B source register in ID stage matches destination register in
EX stage
  ID_EX_Rt <= ID_EX_IR(20 downto 16);

  ALU_B_EX_Compare <= ID_EX_Rt XOR EX_MEM_Rd;
  ID_EX_Rt_Is_EX_MEM_Rd <= '1' when (ALU_B_EX_Compare = (ALU_B_EX_Compare'range
=> '0')) else '0';

  -- Check if ALU B source register in ID stage matches destination register in
MEM stage
  ALU_B_MEM_Compare <= ID_EX_Rt XOR MEM_WB_Rd;
  ID_EX_Rt_Is_MEM_WB_Rd <= '1' when (ALU_B_MEM_Compare = (ALU_B_MEM_Compare'range
=> '0')) else '0';


  -- Determine if ALU A data hazard exists in EX stage or MEM stages
  EX_A_Hazard   <= EX_MEM_Reg_Write_En AND EX_MEM_Rd_Non_Zero AND
ID_EX_Rs_Is_EX_MEM_Rd;
  Fwd_A_Code(1) <= EX_A_Hazard;
  Fwd_A_Code(0) <= MEM_WB_Reg_Write_En AND MEM_WB_Rd_Non_Zero AND
ID_EX_Rs_Is_MEM_WB_Rd AND (NOT EX_A_Hazard);

  -- Determine if ALU B data hazard exists in EX stage or MEM stages
  EX_B_Hazard   <= EX_MEM_Reg_Write_En AND EX_MEM_Rd_Non_Zero AND
ID_EX_Rt_Is_EX_MEM_Rd;
  Fwd_B_Code(1) <= EX_B_Hazard;
  Fwd_B_Code(0) <= MEM_WB_Reg_Write_En AND MEM_WB_Rd_Non_Zero AND
ID_EX_Rt_Is_MEM_WB_Rd AND (NOT EX_B_Hazard);

end Behavioral;
```

ii. Entire Testbench Code with comments used to verify design

32-bit ALU Testbench

```
library ieee;
use ieee.std_logic_1164.all;

entity TB_ALU_32 is
```

```
end entity;

architecture testbench of TB_ALU_32 is
    component ALU
        generic(N      : integer := 32);
        port(   A      : in std_logic_vector(N-1 downto 0);
                B      : in std_logic_vector(N-1 downto 0);
                ALUOp  : in std_logic_vector(3   downto 0); --9 Operations Used

                R      : out std_logic_vector(N-1 downto 0);
                Z      : out std_logic
            );
    end component;
    --set up signals
    signal A     : std_logic_vector(31 downto 0);
    signal B     : std_logic_vector(31 downto 0);
    signal ALUOp : std_logic_vector(3 downto 0);
    signal R     : std_logic_vector(31 downto 0);
    signal Z     : std_logic;
begin
    dut: ALU
        port map (
            A     => A,
            B     => B,
            ALUOp => ALUOp,
            R     => R,
            Z     => Z
        );


    test : process
    begin

        A   <= "10010011001011000000010110100100"; B   <=
"11101011011110011010001011000100";

        ALUOp <= "0000"; wait for 50 ps; -- ADD
        ALUOp <= "0001"; wait for 50 ps; -- SUB
        ALUOp <= "0010"; wait for 50 ps; -- AND
        ALUOp <= "0011"; wait for 50 ps; -- OR
        ALUOp <= "0100"; wait for 50 ps; -- NOR
        ALUOp <= "0101"; wait for 50 ps; -- NAND
        ALUOp <= "0110"; wait for 50 ps; -- XOR
        ALUOp <= "0111"; wait for 50 ps; -- SLL
        ALUOp <= "1000"; wait for 50 ps; -- SRL
```

```
    end process;
end architecture;
```

Sign Extend Testbench

```
library ieee;
use ieee.std_logic_1164.all;

entity TB_SignExtend is
end entity;

architecture Testbench of TB_SignExtend is
    component SignExtend
        port(   A   : in  std_logic_vector(15 downto 0);

                B   : out std_logic_vector(31 downto 0)
        );
    end component;
    --set up signals
    signal A     : std_logic_vector(15 downto 0);
    signal B     : std_logic_vector(31 downto 0);
begin
    dut: SignExtend
        port map (
            A     => A,
            B     => B
        );
    test : process
    begin
        A <= x"0FFF"; wait for 50 ps; -- Case 0
        A <= x"F000"; wait for 50 ps; -- Case 1
    end process;
end architecture;
```

Word Align Testbench

```
library ieee;
use ieee.std_logic_1164.all;

entity TB_WordAlign is
end entity;

architecture Testbench of TB_WordAlign is
    component WordAlign
        port(   A   : in  std_logic_vector(31 downto 0);

                B   : out std_logic_vector(31 downto 0)
        );
    end component;
```

```
    --set up signals
    signal A     : std_logic_vector(31 downto 0);
    signal B     : std_logic_vector(31 downto 0);
begin
    dut: WordAlign
        port map (
            A     => A,
            B     => B
        );
    test : process
    begin
        A <= x"30000000"; wait for 50 ps; -- Case A
        A <= x"00000003"; wait for 50 ps; -- Case B
    end process;
end architecture;
```