MANISH PUNDIR
2022PVL0063

28 March 2023

Minor Project 1

Review Report On **”MIPS Based Single Cycle Processor Implementation”**

---

1 Problem Statement

A MIPS 32 Architecture based Single Cycle Processor has to be Implemented on the Xilinx Vivado Platform using Verilog HDL. The Processor must able to execute the R Type(ADD, SUB, AND, OR, SLT), I Type(SW, LW) and J Type(JUMP, BEQ) Instructions.

2 Solution

2.1 Introduction

A single cycle processor is a type of microprocessor architecture that executes one instruction per clock cycle. In contrast to a multi-cycle processor, which takes several clock cycles to complete a single instruction, a single cycle processor can complete an instruction in a single clock cycle.
The MIPS 32 architecture is a popular 32-bit RISC (Reduced Instruction Set Computer) architecture commonly used in embedded systems, networking devices, and consumer electronics. It is known for its simplicity, efficiency, and high performance.
In a single cycle processor based on the MIPS 32 architecture, each instruction is fetched, decoded, and executed in a single clock cycle. This means that the processor must be able to execute the most complex instruction in a single cycle, which can limit the clock speed and increase the hardware complexity.
The processor consists of several components, including the instruction memory, data memory, ALU (Arithmetic Logic Unit), control unit, and registers. The instruction memory holds the program instructions, while the data memory holds the data that the program operates on. The ALU performs arithmetic and logical operations, while the control unit manages the flow of instructions and data between the different components. The registers are high-speed memory locations used to hold data for processing.
The single cycle processor based on MIPS 32 architecture follows a simple pipeline with five stages: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and write-back (WB). In each clock cycle, one instruction goes through all the five stages, and the next instruction is fetched in the IF stage.
Despite its simplicity, the single cycle processor based on MIPS 32 architecture has some limitations, including a limited clock speed due to the need to execute complex instructions in a single cycle and high hardware complexity to support all instructions. Nonetheless, the single cycle processor based on MIPS 32 architecture is widely used in embedded systems, and it provides an efficient and reliable way to execute simple programs with low power consumption.

2.2 Types of Instruction

2.2.1 R Type Instructions

R-type instructions are a category of instructions in the MIPS assembly language that exclusively work on register operands. They are responsible for performing arithmetic and logical operations on register values, with the result stored in a destination register.
The name ”R-type” is derived from the opcode of these instructions, which specifies the operation type, and three

register fields that specify the source registers and the destination register.
Some examples of R-type instructions include ADD, SUB, AND, OR, and XOR, which perform operations such as addition, subtraction, bitwise AND, bitwise OR, and bitwise XOR on two source registers and store the result in a destination register.
As R-type instructions only work on register values and do not access memory, they are widely considered to be the fastest and most efficient type of instruction in the MIPS architecture. As such, they are commonly used in high-performance computing applications, such as scientific computing and digital signal processing

2.2.2 I Type Instructions

-type instructions are a type of instruction in the MIPS assembly language that utilize an immediate operand, a constant value included in the instruction itself, in addition to register operands.
These instructions perform operations on register values and the immediate operand and store the result in a destination register or memory location. I-type instructions are used for a variety of tasks, including arithmetic and logical operations, data transfer operations, and conditional branching.
The name "I-type" comes from the opcode of these instructions, which specifies the operation type, two register fields for specifying the source and destination registers, and an immediate field that holds the immediate operand.
Examples of I-type instructions in MIPS assembly language include ADDI, ORI, LW, and BEQ, which perform operations such as addition with an immediate value, bitwise OR with an immediate value, loading a word from memory into a register, and performing a conditional branch if two registers are equal.
Although I-type instructions are slower than R-type instructions, as they require an extra cycle to fetch the immediate operand from memory, they are still widely used in many MIPS-based applications due to their versatility and usefulness.

2.2.3 J Type Instructions

J-type instructions are a type of instruction in MIPS assembly language that perform unconditional jumps, transferring control of the program to a specific memory address.
These instructions have a single field, the jump target address, which is a 26-bit address that specifies the target memory address for the jump operation. The opcode for J-type instructions is implied since the instruction format only contains this field.
J-type instructions are primarily used in loops and function calls, as they enable a program to jump to a specific location in memory without executing the instructions between the current instruction and the jump target. This feature can be used to skip over a section of code or to jump to a specific subroutine or function.
Two examples of J-type instructions in MIPS assembly language are J and JAL. J jumps to the specified target address while JAL stores the return address in the ra register before jumping.
J-type instructions are faster than I-type instructions since they do not require memory access. However, they are less flexible as they only perform unconditional jumps. Nonetheless, they are a critical component of the MIPS instruction set and find broad usage in MIPS-based applications.

2.3 DataPath

The datapath of a single-cycle MIPS processor is a set of connected components that execute the instructions. The datapath is divided into five stages: instruction fetch (IF), instruction decode (ID), execute (EX), memory (MEM), and write back (WB).
During the IF stage, the processor retrieves the instruction from memory utilizing the program counter (PC) as the address. The instruction is then sent to the ID stage where it is decoded and the required registers are read.
The EX stage performs the instruction execution. The operation that is performed depends on the type of instruc-

tion, such as arithmetic, logical, or comparison.
The MEM stage performs any memory operations, like loading or storing data in memory.
In the WB stage, the results of the instruction execution are written back to the register file.
The datapath includes several components, including the PC, instruction memory, register file, ALU, and data memory, which collaborate to execute the instructions.
In summary, the datapath of a single-cycle MIPS processor comprises five stages that process instructions in a single cycle. The datapath includes multiple connected components that work in harmony to execute instructions efficiently.

2.4 Different DataPath Elements

Program Counter (PC):

- The PC is a register that holds the address of the next instruction to be fetched from memory.
- At the start of program execution, the PC is initialized to the address of the first instruction in memory.
- The processor uses the PC to generate a memory address for the next instruction to fetch.
- The PC is incremented after each instruction fetch to point to the next instruction in memory.
- If an instruction is a jump or branch instruction, the PC is updated to a new address specified by the instruction.
- The PC can be modified by the operating system or hardware in response to interrupts, exceptions, or other events. The control unit generates control signals to manage the operation of the datapath, including updating the PC.
- The PC is critical to the processor's ability to fetch instructions and execute programs correctly.

Instruction Memory:

- The processor generates the memory address of the instruction it wants to fetch by using the Program Counter (PC) register.
- The address is then sent to the Instruction Memory as a read request.
- The Instruction Memory retrieves the instruction at the specified memory address and sends it back to the processor.
- The fetched instruction is stored in an Instruction Register within the processor, where it waits to be decoded and executed.
- The PC is then updated to point to the next instruction in memory, so that the processor can fetch the next instruction in the program.
- This process repeats for every instruction in the program until the program has completed execution.

Register File:

- The Register File is a component of the processor that consists of a set of registers, each of which can store a value.

- The register file is used to hold the processor's working data, including temporary variables, intermediate results, and values loaded from memory.
- The register file is accessed by the processor to read or write data to and from the registers.
- The registers are identified by their register numbers, which are encoded in instruction fields.
- The register file has two input ports and one output port. The two input ports are used to read data from two registers simultaneously, while the output port is used to write data to a register.
- The register file uses multiplexers to select the two input registers and a write enable signal to control whether data is written to a register or not.
- The control unit generates control signals to manage the operation of the datapath, including reading and writing data to the register file.
- The register file is a critical component of the processor's datapath, enabling it to store and manipulate data efficiently.

Arithmetic Logic Unit (ALU):

- R-Type Instructions:
  - The ALU takes two input operands from the register file, based on the instruction's register specification.
  - The control unit selects the operation to be performed based on the instruction's function code.
  - The ALU performs the selected operation on the two operands, generating a result.
  - The result is written back to the register file, based on the instruction's destination register specification.
- I-Type Instructions:
  - The ALU takes two input operands from the register file, based on the instruction's register specification and immediate value.
  - The control unit selects the operation to be performed based on the instruction's opcode.
  - The ALU performs the selected operation on the two operands, generating a result.
  - The result is either written back to the register file, based on the instruction's destination register specification, or used as an address for data memory access.
- J-Type Instructions:
  - The ALU takes two input operands: the current program counter (PC) and the jump address extracted from the instruction.
  - The control unit sets the PC to the jump address, which is the concatenation of the upper four bits of the PC and the jump target extracted from the instruction.
  - The PC is incremented by 4 in the next clock cycle to fetch the instruction at the new address.

Data Memory:

- Data Memory is a component of the processor that stores data values in memory locations. It can be accessed through Load and Store instructions.
- The address of the data memory location to be accessed is determined by the contents of a register specified in the instruction.

- If the instruction is a Load instruction, the data memory sends the contents of the specified memory location to the register file for use in subsequent operations.
- If the instruction is a Store instruction, the data memory receives data from a register specified in the instruction and stores it in the memory location specified by the address.
- The size of the data memory is determined by the number of addressable memory locations and the number of bits in each memory location.
- The speed of data memory access is a critical factor in the overall performance of the processor and is affected by factors such as memory latency and cache size.
- The processor control unit manages the flow of data between the data memory and other components of the datapath, ensuring that data is correctly loaded and stored at the appropriate memory locations.

Control Unit:

- The Control Unit is responsible for generating control signals that direct the operations of the processor's various components, such as the ALU, register file, data memory, and instruction memory.
- The Control Unit receives the instruction from the Instruction Memory and uses the opcode and function code to determine the sequence of operations required to execute the instruction.
- The Control Unit generates control signals that set the appropriate values on the processor's various buses and registers, including the ALU control signals, register file write enable signal, and data memory read/write signals.
- The Control Unit also manages the processor's pipeline stages and ensures that instructions are executed in the correct order.
- The Control Unit generates branch control signals to handle branch instructions and jump control signals to handle jump instructions.
- The Control Unit may also generate exceptions and interrupts in response to external events, such as invalid memory accesses or hardware errors.
- The design of the Control Unit is critical to the performance and functionality of the processor, and it is typically implemented using a combination of digital circuits such as decoders, multiplexers, and logic gates.

Multiplexers (MUX):

- Digital circuits that select between two or more inputs and output one of them.

Adders:

- Digital circuits that perform addition operations.

Shifters:

- Digital circuits that shift bits left or right.

Sign Extenders:

- A digital circuit that extends a sign bit to fill an entire register or memory location.

PC Incrementer:

- A circuit that increments the program counter to fetch the next instruction.

Instruction Decoder:

- A circuit that decodes the instruction fetched from memory and generates control signals for the datapath.

2.5 DataPath Diagram

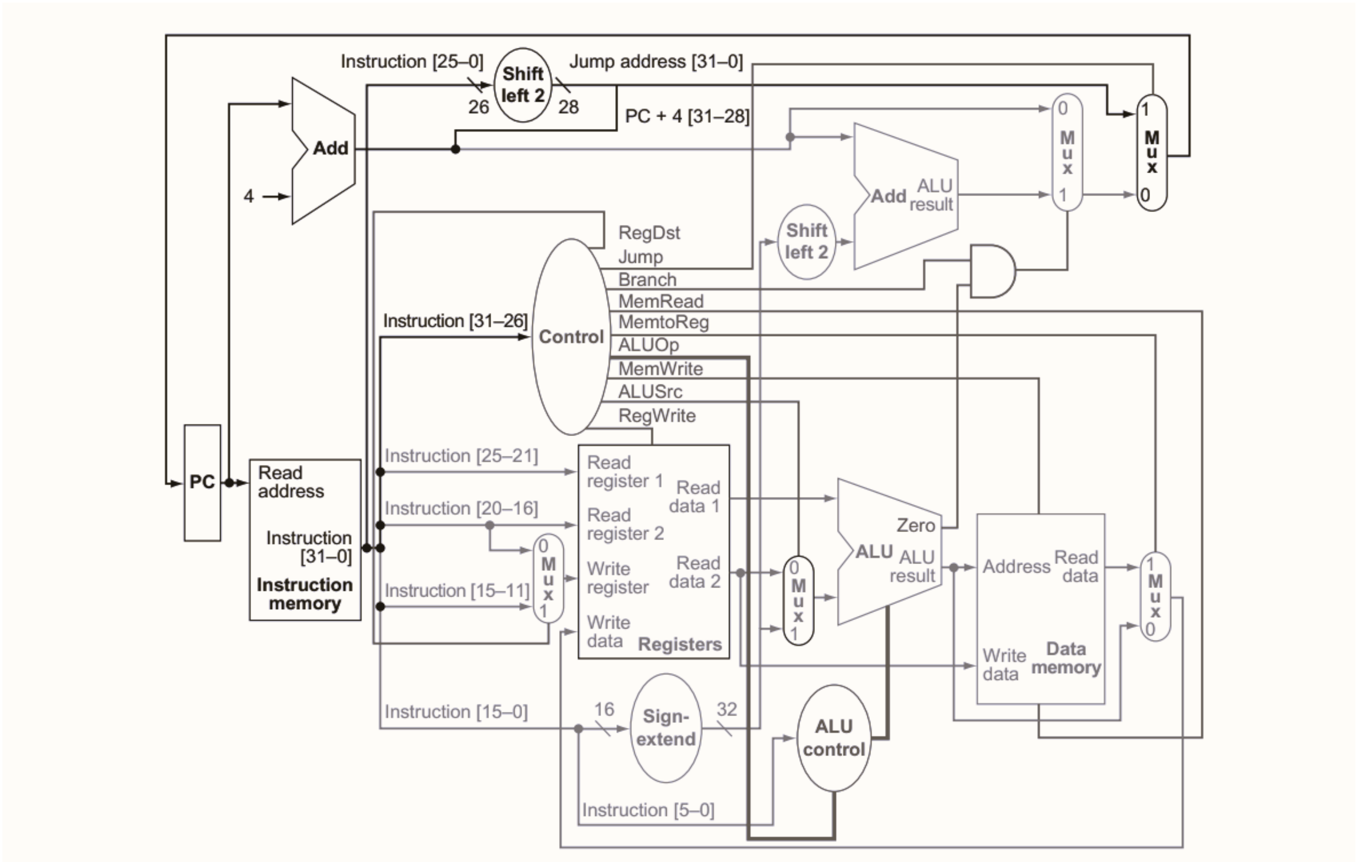


Figure 1: DataPath of MIPS Single Cycle Implementation

2.6 Working and Code of Modules

2.6.1 Instruction Memory

```
module Instruction_Memory(PC_Read_address, Instruction_Out);
input [31:0] PC_Read_address;
output reg [31:0] Instruction_Out;
reg [7:0] Inst_Mem[0:4095];
initial
begin
Inst_Mem[0] = 8'h8C;
```

```
Inst_Mem[1] = 8'h09;
Inst_Mem[2] = 8'h00;
Inst_Mem[3] = 8'h00;

Inst_Mem[4] = 8'h8c;
Inst_Mem[5] = 8'h0a;
Inst_Mem[6] = 8'h00;
Inst_Mem[7] = 8'h04;

Inst_Mem[8] = 8'h11;
Inst_Mem[9] = 8'h49;
Inst_Mem[10] = 8'h00;
Inst_Mem[11] = 8'h04;

Inst_Mem[12] = 8'h01;
Inst_Mem[13] = 8'h2a;
Inst_Mem[14] = 8'h58;
Inst_Mem[15] = 8'h22;

Inst_Mem[16] = 8'h08;
Inst_Mem[17] = 8'h00;
Inst_Mem[18] = 8'h00;
Inst_Mem[19] = 8'h08;

Inst_Mem[28] = 8'h01;
Inst_Mem[29] = 8'h2a;
Inst_Mem[30] = 8'h58;
Inst_Mem[31] = 8'h20;

Inst_Mem[32] = 8'h01;
Inst_Mem[33] = 8'h2a;
Inst_Mem[34] = 8'h58;
Inst_Mem[35] = 8'h22;

end
always @(PC_Read_address)
begin
Instruction_Out = {Inst_Mem[PC_Read_address], Inst_Mem[PC_Read_address+32'd1],
Inst_Mem[PC_Read_address+32'd2], Inst_Mem[PC_Read_address+32'd3]};
end
endmodule
```

2.7 Explanation of the Instruction Memory Code

- The module takes in a 32-bit input PCReadaddress, which specifies the address in the instruction memory that needs to be read.
- It outputs a 32-bit InstructionOut, which represents the instruction read from the specified memory address.
- The module uses a register array InstMem of size 4096 (0 to 4095) to store the instructions.

- In the initial block, the instruction memory is initialized with some sample instructions. Each instruction is 32 bits wide and is stored in 4 consecutive bytes of the memory.
- In the always block, the output InstructionOut is computed based on the input address PCReadaddress. It concatenates the 4 consecutive bytes starting from the specified memory address to form the complete instruction.
- The curly braces used in the concatenation operation combine the 4 bytes into a single 32-bit instruction output.

2.8 PC Incrementor

```
module PC_Adder(Current_Ins,Next_Ins);
input[31:0]Current_Ins;
output[31:0]Next_Ins;
assign Next_Ins=Current_Ins+32'd4;
endmodule
```

2.9 Explanation of PC Incrementor

The ”PC_Adder” module takes a 32-bit input signal ”Current_Ins” as the current instruction address and outputs a 32-bit signal ”Next_Ins” which represents the next instruction address.

The ”Next_Ins” signal is calculated by adding the value of ”Current_Ins” to a 32-bit constant value of 4, which is done using the ”assign” statement.

Therefore, the ”PC_Adder” module serves as a simple adder unit to increment the current instruction address by 4, which is the size of a single instruction in a MIPS 32-bit processor.

2.10 Shift Left by 2 Module

```
module Shift_Left_2_26b(
input [25:0] data_in,
output reg [27:0] data_out );
always @(*)
begin
data_out= data_in << 2;
end
endmodule
```

2.11 Explanation of Shift Left by 2 Module

This Verilog module performs the operation of left-shifting the input data by 2 bits and outputs the result in 28 bits. The module takes in a 26-bit input signal named ”data_in” and outputs a 28-bit signal named ”data_out”.

2.12 Sign_Extension

```
module Sign_Extension(input [15:0] In,output reg [31:0] Out);
always@(In)
begin
if (In[15]==1)
begin
Out = {16'hffff , In};
end
else
begin
Out = {16'h0000, In};
end
end
endmodule
```

2.13 Explanation of Sign_Extension Code

The module takes a 16-bit input signal ”In” and outputs a 32-bit signal ”Out” after sign extension. The module consists of an always block which is triggered by any change in the input signal ”In”. If the most significant bit (MSB) of the input signal is 1, it means that the number is negative and requires sign extension. In this case, the output signal is formed by concatenating 16'bFFFF (all ones) and the input signal ”In”. If the MSB of the input signal is 0, the number is positive and the output signal is formed by concatenating 16'b0000 (all zeros) and the input signal ”In”. The ”Out” signal is declared as a reg type output signal, which means that the output value can be stored and retained until it is updated by the next value of ”In”.

2.14 Register_File

```
module Register_File(clk,Read_Register1,Read_Register2,Write_Register,
Write_data,Read_data1,Read_data2,Reg_write,PC_in,PC_out,Reset);
input[4:0]Read_Register1,Read_Register2,Write_Register;
input[31:0]Write_data,PC_in;
output[31:0]Read_data1,Read_data2,PC_out;
input clk,Reg_write,Reset;
reg[31:0]Register_Mem[0:31];
integer i;
initial
begin
for(i=0;i<32;i=i+1)
begin
Register_Mem[i]<=32'b0;
end
end
assign PC_out = Register_Mem[29];
always@(posedge clk)
begin
Register_Mem[29]=PC_in;
if(Reg_write)
begin
```

```
Register_Mem[Write_Register]=Write_data;
end
end
assign Read_data1 = Register_Mem[Read_Register1];
assign  Read_data2 = Register_Mem[Read_Register2];
always@(posedge clk)
begin
if (Reset)
begin
Register_Mem[29] = 0;
end
end
endmodule
```

2.15 Explanation of Register Code

This Register File stores 32-bit data in 32 registers. It has five inputs: clk for clock signal, Read_Register1 and Read_Register2 for selecting the registers to be read, Write_Register for selecting the register to be written, Write_data for the data to be written, and PC_in for updating the value of register 29 (the program counter). It has three outputs: Read_data1 and Read_data2 for the data read from the selected registers, and PC_out for the value of register 29. The module also has two control inputs: Reg_write for enabling the register write operation, and Reset for resetting the contents of the registers to zero. The module uses a synchronous positive-edge clock and an integer i for looping through the register memory during initialization. The Register_Mem array stores the 32 registers, and is initialized to all zeros in the initial block. The PC_out is assigned the value of register 29 (Register_Mem[29]). During a positiveedge clock cycle, if the Reg_write signal is high, the data in Write_data is written to the selected register (Register_Mem[Write_Register]). Similarly, during a positiveedge clock cycle, if the Reset signal is high, the contents of register 29 are set to zero. The Read_data1 and Read_data2 outputs are assigned the data stored in the selected registers Register_Mem[Read_Register1]and Register_Mem[Read_Register2], respectively.

2.16 Control Unit

```
    module Control_Main(
    input [5:0] opcode,
    output reg RegDst,
    output reg Jump,
    output reg Branch,
    output reg MemRead,
    output reg MemtoReg,
    output reg ALUop1,ALUop2,
    output reg MemWrite,
    output reg ALUSrc,
    output reg RegWrite
);

always @(*) begin
    case (opcode)
        // r type
        6'b000000: begin
```

```
            RegDst = 1;
            Jump = 0;
            Branch = 0;
            MemRead = 0;
            MemtoReg = 0;
            ALUop1 = 1'b1;
            ALUop2=1'b0;
            MemWrite = 0;
            ALUSrc = 0;
            RegWrite = 1;
        end
        // beq
        6'b000100: begin
            RegDst = 1'bx;
            Jump = 0;
            Branch = 1;
            MemRead = 0;
            MemtoReg = 1'bx;
            ALUop1 = 1'b0;
            ALUop2=1'b1;
            MemWrite = 0;
            ALUSrc = 0;
            RegWrite = 0;
        end
        // lw
        6'b100011: begin
            RegDst = 0;
            Jump = 0;
            Branch = 0;
            MemRead = 1;
            MemtoReg = 1;
             ALUop1 = 1'b0;
            ALUop2=1'b0;
            MemWrite = 0;
            ALUSrc = 1;
            RegWrite = 1;
        end
        // sw
        6'b101011: begin
            RegDst = 0;
            Jump = 0;
            Branch = 0;
            MemRead = 0;
            MemtoReg = 1'bx;
             ALUop1 = 1'b0;
            ALUop2=1'b0;
            MemWrite = 1;
            ALUSrc = 1;
            RegWrite = 0;
        end
        // jump
        6'b000010: begin
            RegDst = 1'bx;
            Jump = 1;
            Branch = 0;
            MemRead = 0;
            MemtoReg = 1'bx;
             ALUop1 = 1'b0;
            ALUop2=1'b0;
            MemWrite = 0;
            ALUSrc = 0;
            RegWrite = 0;
        end
    endcase
end

endmodule
```

2.17 Explanation Of Control unit Code

This Verilog code implements a combinational logic circuit that takes a 6-bit opcode as input and produces control signals as outputs for a MIPS processor.

The control signals are used to control various components of the processor such as the register file, ALU, and memory. The signals are assigned based on the opcode using a case statement.

Here are the control signals and their meanings:

- RegDst: Selects the destination register for the result of an instruction (0 for rt, 1 for rd).
- Jump: Indicates whether the instruction is a jump or not.
- Branch: Indicates whether the instruction is a branch or not.
- MemRead: Indicates whether the instruction is a load word or not.
- MemtoReg: Selects the source of data to be written to a register (0 for ALU result, 1 for memory data).
- ALUop1 and ALUop2: Control the operation of the ALU.
- MemWrite: Indicates whether the instruction is a store word or not.
- ALUSrc: Selects the second input to the ALU (0 for register data, 1 for immediate value). RegWrite: Indicates whether the instruction writes to a register or not.
- Note that some signals are assigned to ”x” to indicate that they are don’t cares for certain instructions.

2.18 ALU

```
    module ALU_Main( input [31:0] R_data1,
  input [31:0] R_data2,
  input [2:0] ALU_control,
  output reg [31:0] ALU_Result,
  output reg zero
);
     always @(*)
  begin
    case(ALU_control)
      3'b000: ALU_Result = R_data1 & R_data2;
      3'b001: ALU_Result = R_data1 | R_data2;
      3'b010: ALU_Result = R_data1 + R_data2;
      3'b110: ALU_Result = R_data1 - R_data2;
      3'b111: ALU_Result = (R_data1 < R_data2) ? 32'd1 : 32'd0;
      default: ALU_Result = R_data1 + R_data2;
    endcase

    zero = (ALU_Result == 0)? 1'b1: 1'b0;
  end
endmodule
```

2.19 Explanation of ALU Code

This is a Verilog module for an Arithmetic Logic Unit (ALU). ALU takes two 32-bit inputs R_data1 and R_data2 and performs an operation specified by the 3-bit ALU_control input. The module outputs the 32-bit result of the operation in ALU_Result and a zero flag in the zero output indicating if the result is zero. The module uses a case statement to select the operation to perform based on the value of ALU_control. The following operations are supported:

- **Bitwise AND (000):** Performs a bitwise AND operation on the inputs.
- **Bitwise OR (001):** Performs a bitwise OR operation on the inputs.
- **Addition (010):** Adds the inputs.
- **Subtraction (110):** Subtracts R_data2 from R_data1.
- **Set on less than (111):** If R_data1 is less than R_data2, set ALU_Result to 1, otherwise set it to 0.
- **Default:** Performs an addition operation as the default operation.
- The zero flag is set to 1 if the result in ALU_Result is zero and 0 otherwise.

Overall, this module is used as a building block in a CPU to perform arithmetic and logical operations in the instruction execution stage.

2.20 Data Memory

```
    module Data_Memory(Address,Write_data,Read_data,Mem_Write,Mem_Read,clk);
input[31:0]Address,Write_data;
input Mem_Write,Mem_Read;
input clk;
output reg [31:0] Read_data;
```

```
reg[7:0] Data_Memory[0:4095];
initial
begin
Data_Memory[0] = 8'h00;
Data_Memory[1] = 8'h00;
Data_Memory[2] = 8'h00;
Data_Memory[3] = 8'h06;
Data_Memory[4] = 8'h00;
Data_Memory[5] = 8'h00;
Data_Memory[6] = 8'h00;
Data_Memory[7] = 8'h05;
Data_Memory[8] = 8'h8f;
Data_Memory[9] = 8'h09;
Data_Memory[10] = 8'h00;
Data_Memory[11] = 8'h08;
Data_Memory[12] = 8'haf;
Data_Memory[13] = 8'h08;
Data_Memory[14] = 8'h00;
Data_Memory[15] = 8'h10;
Data_Memory[16] = 8'h10;
Data_Memory[17] = 8'h09;
Data_Memory[18] = 8'h09;
Data_Memory[19] = 8'h03;
Data_Memory[20] = 8'h00;
Data_Memory[21] = 8'h29;
Data_Memory[22] = 8'h49;
Data_Memory[23] = 8'h20;

end
always@(*)
begin
if(Mem_Write&&~Mem_Read)
begin
Data_Memory[Address]= Write_data[31:24];
Data_Memory[Address+32'd1]=Write_data[23:16];
Data_Memory[Address+32'd2]=Write_data[15:8];
Data_Memory[Address+32'd3]=Write_data[7:0];
end
else if(Mem_Read&&~Mem_Write)
begin
Read_data[31:24]=Data_Memory[Address];
Read_data[23:16]=Data_Memory[Address+32'd1];
Read_data[15:8]=Data_Memory[Address+32'd2];
Read_data[7:0]=Data_Memory[Address+32'd3];
end
end
endmodule
```

2.21 Explanation Of Data Memory Code

- Address: a 32-bit input port for specifying the address of the memory location to be read/written.
- Write_data: a 32-bit input port for specifying the data to be written to the memory.
- Read_data: a 32-bit output port for outputting the data read from the memory.
- Mem_Write: an input port used to enable memory write operation.
- Mem_Read: an input port used to enable memory read operation.
- clk: the clock input port.
- The memory is implemented as an array of 8-bit elements (Data_Memory), with 4096 elements (i.e., 0 to 4095). It is initialized with some values using an initial block.

The always block is used to perform the read/write operation based on the input signals. When Mem_Write is high and Mem_Read is low, it means that a write operation should be performed. The Write_data is written to the memory array using the Address as the starting address. Since each element in the array is 8 bits, the 32-bit Write_data is broken down into four 8-bit values and stored in consecutive memory locations starting from the Address.

When Mem_Read is high and Mem_Write is low, it means that a read operation should be performed. The data read from the memory is stored in Read_data. Similar to the write operation, the data is read from the four consecutive memory locations starting from the Address and combined to form a 32-bit value, which is stored in Read_data.

2.22 Concatenation

```
module Jump_Concatination(
input [27:0] Ins_28b,
input [3:0] PC_plus4,
output reg [31:0] Jump_Address);
always @(*)
begin
Jump_Address= {PC_plus4, Ins_28b};
end
endmodule
```

2.23 MUX

```
module Mux2x1_32b(
input [31:0] Input0,
input [31:0] Input1,
input sel,
output reg [31:0] out);
always @ (*)
begin
if(sel)
out= Input1;
else
```

```
out= Input0;
end
endmodule
```

2.24 Adder ALU

```
module Mux2x1_32b(
input [31:0] Input0,
input [31:0] Input1,
input sel,
output reg [31:0] out);
always @ (*)
begin
if(sel)
out= Input1;
else
out= Input0;
end
endmodule
```

2.25 ALU Control

```
    module ALU_Control(
    input ALU_Op1, ALU_Op2,
    input [5:0] Funct,
    output reg [2:0] ALU_Control
);
always @(*)
    begin
        case ({ALU_Op1, ALU_Op2})
            2'b00: ALU_Control = 3'b010; // add
            2'b01: ALU_Control = 3'b110; // add
            2'b10:
                case (Funct)
                    6'b100100: ALU_Control = 3'b000; // and
                    6'b100101: ALU_Control = 3'b001; // or
                    6'b100000: ALU_Control = 3'b010; // add
                    6'b100010: ALU_Control = 3'b110; // sub
                    6'b101010: ALU_Control = 3'b111; // slt
                    default: ALU_Control = 3'b000;
                endcase
            default: ALU_Control = 3'b000;
        endcase
    end

endmodule
```

2.26 Explanation of ALU Control

It takes in two inputs, ALU_Op1 and ALU_Op2, and a 6-bit Funct signal. It outputs a 3-bit signal ALU_Control that determines the operation to be performed by the ALU based on the input signals.

The module uses a case statement to determine the ALU_Control output signal based on the values of ALU_Op1, ALU_Op2, and Funct. The case statement checks the two-bit concatenation of ALU_Op1 and ALU_Op2 to determine which operation is being requested. If ALU_Op1 and ALU_Op2 are both 0, then the output is 010, which corresponds to an add operation. If ALU_Op1 and ALU_Op2 are both 1, then the output is 110, which also corresponds to an add operation. If ALU_Op1 is 1 and ALU_Op2 is 0, then the case statement checks the value of Funct to determine the operation. If Funct matches one of the predefined values (100100 for and, 100101 for or, 100000 for add, 100010 for sub, or 101010 for slt), then the corresponding output signal is set. If none of the predefined values match, the output is set to 000, which corresponds to an and operation. If neither ALU_Op1 nor ALU_Op2 is 1, the output is also set to 000.

3 Test Cases

```
if(a == b) {
c = a + b;}
else
{c = a - b;}
```

3.1 MIPS Assembly Code for the Given Snippet

```
lw $t0, 0($s0)
lw $t1, 4($s0)
beq $t0, $t1, add
sub $t2, $t0, $s1
j end
add:
add $t2, $t0, $t1
end:sw $t2, 8($s0)
```

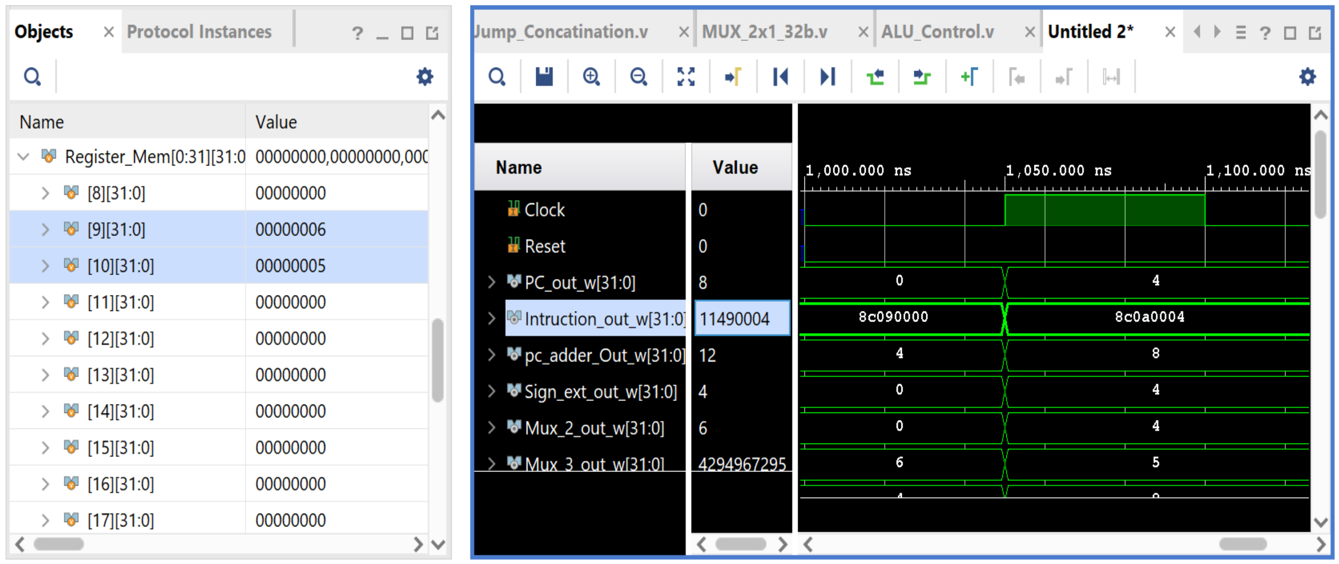
3.2 HexaDecimal Instructions for the Above MIPS Assembly Code

```
8c090000 # Load the Register_Mem[9]($t0) with Value of data_Memory[3:0] = 6
8c0a0004 # Load the Register_Mem[10]($t1) with Value of data_Memory[7:4] = 5
11490004 # Beq on the $t0 , $t1  Jump to Label add(ac0b0008) if $t0=$t1
otherwise continues to 012a5822.
012a5822 #Subtraction Instruction  will subtract the  Register_Mem[10] from
Register_Mem[9] and will save the result in  Register_Mem[11]
08000008 # jump to 32 will make PC 16 to 32
012a5820 #Add Register_Mem[9] ,Register_Mem[10] and store in Register_Mem[11]
ac0b0008 # store the value of Register_Mem[11] to Data_Mem[11:8]
```

We have to save the following hexadecimal Instructions in the Instruction Memory.
Data_Memory[3:0] = 6
Data_Memory[7:4] = 5

3.3 Test Case for 8c090000 and 8c0a0004

Both Instructions will load the register file from the data memory.

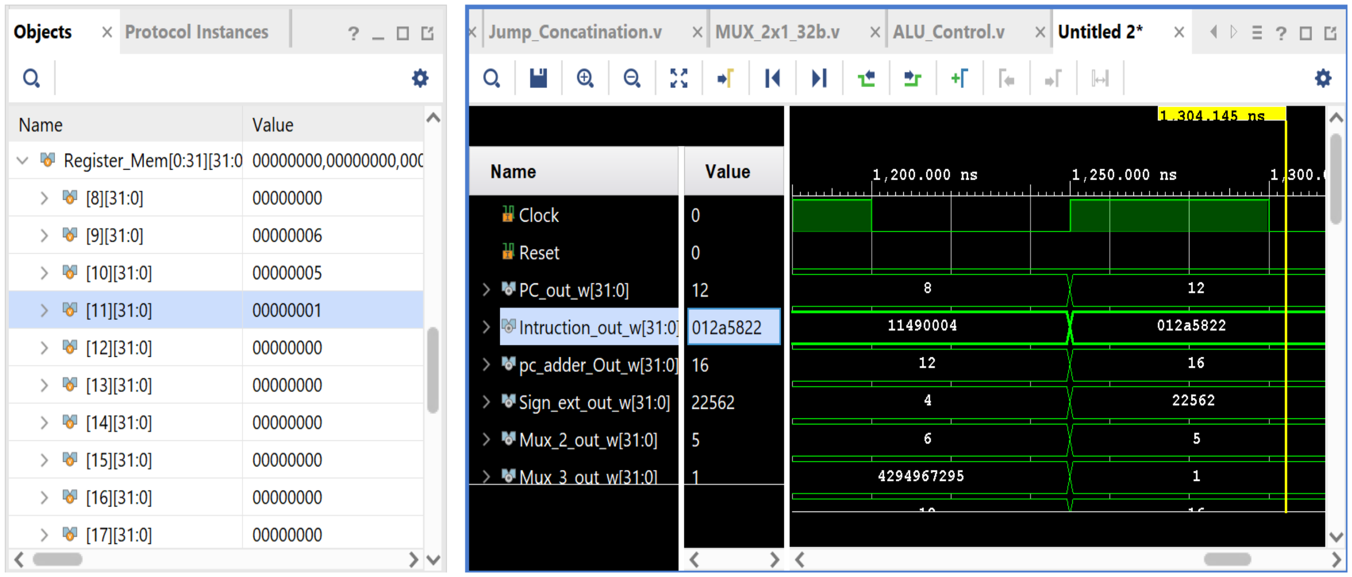


Name	Value
Register_Mem[0:31][31:0	00000000,00000000,00(
[8][31:0]	00000000
[9][31:0]	00000006
[10][31:0]	00000005
[11][31:0]	00000000
[12][31:0]	00000000
[13][31:0]	00000000
[14][31:0]	00000000
[15][31:0]	00000000
[16][31:0]	00000000
[17][31:0]	00000000

Name	Value		
Clock	0		
Reset	0		
PC_out_w[31:0]	8	0	4
Intruction_out_w[31:0]	11490004	8c090000	8c0a0004
pc_adder_Out_w[31:0]	12	4	8
Sign_ext_out_w[31:0]	4	0	4
Mux_2_out_w[31:0]	6	0	4
Mux_3_out_w[31:0]	4294967295	6	5

Figure 2: Test Case for 8c090000 and 8c0a0004

3.4 Test Case for 11490004 and 012a5822



Name	Value
Register_Mem[0:31][31:0	00000000,00000000,00(
[8][31:0]	00000000
[9][31:0]	00000006
[10][31:0]	00000005
[11][31:0]	00000001
[12][31:0]	00000000
[13][31:0]	00000000
[14][31:0]	00000000
[15][31:0]	00000000
[16][31:0]	00000000
[17][31:0]	00000000

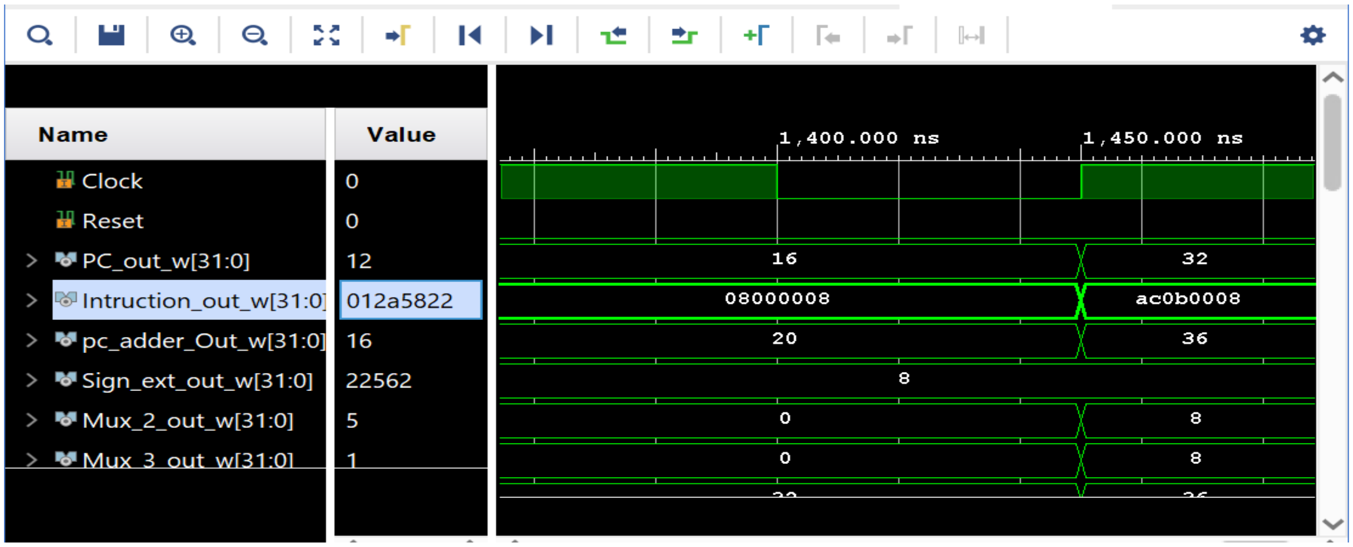
Name	Value		
Clock	0		
Reset	0		
PC_out_w[31:0]	12	8	12
Intruction_out_w[31:0]	012a5822	11490004	012a5822
pc_adder_Out_w[31:0]	16	12	16
Sign_ext_out_w[31:0]	22562	4	22562
Mux_2_out_w[31:0]	5	6	5
Mux_3_out_w[31:0]	1	4294967295	1

Figure 3: Beq Instruction 11490004 and 012a5822

As the Loaded register are not equal so PC moves to next Instruction that is 012a5822. 012a5822 is the Subtraction Inatruction and will subtract the Register_Mem[10] from Register_Mem[9] and will save the result in Register_Mem[11].

3.5 Test Case For 08000008

This Instruction is the Jump Type Instruction it will make the PC_out jump from current address 16 to next PC_out address 32 and it will jump to the next instruction ac0b0008.

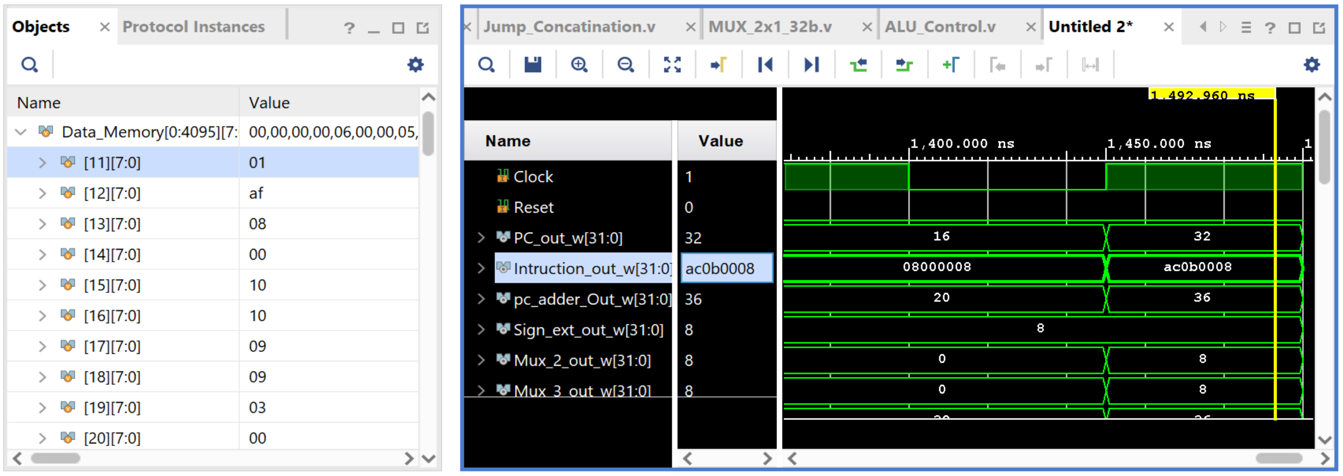


Name	Value
Clock	0
Reset	0
PC_out_w[31:0]	12
Intruction_out_w[31:0]	012a5822
pc_adder_Out_w[31:0]	16
Sign_ext_out_w[31:0]	22562
Mux_2_out_w[31:0]	5
Mux_3_out_w[31:0]	1

Figure 4: 08000008 Instruction Execution

3.6 Test Case For ac0b0008

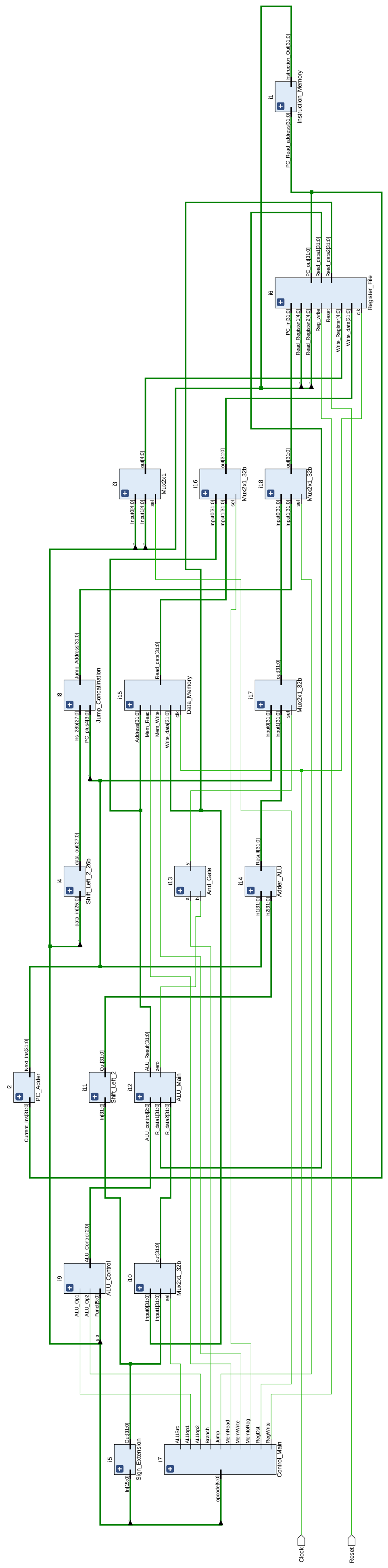
This is the store Word Instruction and will store the value of Register_Mem[11] to Data_Mem[11:8]. so it will store the value of Register_Mem[11] = 1 to Data_Mem[11].



Name	Value
Data_Memory[0:4095][7:	00,00,00,00,06,00,00,05,
[11][7:0]	01
[12][7:0]	af
[13][7:0]	08
[14][7:0]	00
[15][7:0]	10
[16][7:0]	10
[17][7:0]	09
[18][7:0]	09
[19][7:0]	03
[20][7:0]	00

Name	Value
Clock	1
Reset	0
PC_out_w[31:0]	32
Intruction_out_w[31:0]	ac0b0008
pc_adder_Out_w[31:0]	36
Sign_ext_out_w[31:0]	8
Mux_2_out_w[31:0]	8
Mux_3_out_w[31:0]	8

Figure 5: 08000008 Instruction Execution



i1 Instruction_Memory
PC_Read_address[31:0]
Instruction_Out[31:0]
i6 Register_File
PC_in[31:0]
Read_Register1[4:0]
Read_Register2[4:0]
Reg_write
Reset
Write_Register[4:0]
Write_data[31:0]
clk
PC_out[31:0]
Read_data1[31:0]
Read_data2[31:0]
i3 Mux2x1
Input0[4:0]
Input1[4:0]
sel
out[4:0]
i16 Mux2x1_32b
i18 Mux2x1_32b
i17 Mux2x1_32b
Input0[31:0]
Input1[31:0]
out[31:0]
i8 Jump_Concatination
Ins_28b[27:0]
PC_plus4[3:0]
Jump_Address[31:0]
i15 Data_Memory
Address[31:0]
Mem_Read
Mem_Write
Write_data[31:0]
Read_data[31:0]
i4 Shift_Left_2_26b
data_in[25:0]
data_out[27:0]
i13 And_Gate
a
b
y
i14 Adder_ALU
in1[31:0]
in2[31:0]
Result[31:0]
i2 PC_Adder
Current_Ins[31:0]
Next_Ins[31:0]
i11 Shift_Left_2
In[31:0]
Out[31:0]
i12 ALU_Main
ALU_control[3:0]
R_data1[31:0]
R_data2[31:0]
ALU_Result[31:0]
zero
i9 ALU_Control
ALU_Op1
ALU_Op2
Func[5:0]
ALU_Control[3:0]
i10 Mux2x1_32b
i5 Sign_Extension
In[15:0]
Out[31:0]
i7 Control_Main
opcode[5:0]
ALUSrc
ALUop1
ALUop2
Branch
Jump
MemRead
MemWrite
MemtoReg
RegDst
RegWrite
Clock
Reset