Manish Pundir : 2022PVL0063
Ninad Kamble : 2022PVL0064

Dated 27 April 2023

Embedded System Design Major Project
”Design and Implementation of a MIPS 32 Pipelined Processor”

1 Problem Statement

The aim of this major project is to design and implement a 5 Stage MIPS 32 pipelined processor capable of executing R-type instructions, store word and load word instructions, as well as beq and jump instructions. The processor should also effectively handle data and control hazards, ensuring efficient and accurate execution of instructions.

1.1 Description

In the field of embedded system design, a crucial aspect is the development of high-performance processors capable of executing a wide range of instructions. This project focuses on the design and implementation of a MIPS 32 pipelined processor that can effectively handle various instruction types while addressing data and control hazards.It’s important to note that in a pipelined processor, multiple instructions are in different stages of execution simultaneously. Each stage operates on a different instruction, and as one instruction progresses to the next stage, the subsequent instruction enters the previous stage. This allows for parallel execution of multiple instructions and improves the overall throughput of the processor. The processor should support the following instruction types:

- **R-type Instructions:** The processor should be able to execute R-type instructions, which involve arithmetic and logical operations on registers.
- **Store Word and Load Word Instructions:** The processor should support store word and load word instructions, enabling the transfer of data between the memory and registers.
- **Branch Equal (beq) Instructions:** The processor should be capable of executing beq instructions, allowing conditional branching based on the equality of two registers.
- **Jump Instructions:** The processor should support jump instructions, enabling unconditional branching to specified memory addresses.

Additionally, the designed processor should effectively handle data and control hazards to ensure accurate execution of instructions. The hazards may include, but are not limited to, the following:

- **Data Hazards:** These hazards occur when the data required for the execution of an instruction is not yet available due to previous instructions still being processed.
- **Control Hazards:** These hazards occur when conditional branching or jumping instructions introduce uncertainties in the instruction fetch stage, potentially leading to incorrect or delayed instruction execution.

The project should address these hazards by implementing appropriate mechanisms, such as data forwarding, pipeline stalls, branch prediction, and control unit modifications, to minimize the impact of hazards on the processor’s performance.

The project should encompass the following key aspects:

- Designing the MIPS 32 pipelined processor architecture, including the instruction fetch, decode, execute, memory, and writeback stages.
- Implementing the core components of the processor, such as registers, arithmetic logic unit (ALU), control unit, memory unit, and instruction and data caches.
- Ensuring proper pipelining of the processor, considering the interplay of different stages and their dependencies.
- Incorporating hazard detection and mitigation techniques, such as data forwarding, stalls, branch prediction, and control unit modifications, to handle data and control hazards efficiently.
- Verifying the functionality and correctness of the implemented processor through extensive testing and simulation, ensuring it produces accurate results for a variety of test cases.
- Documenting the design, implementation details, testing methodology, and results in a comprehensive report, along with appropriate diagrams, figures, and code snippets.

Successful completion of this project will demonstrate a deep understanding of processor architecture, pipelining, hazard handling techniques, and embedded system design principles. It will also showcase the ability to analyze and solve complex problems in the field of computer engineering.

2 Instruction Execution Steps

2.1 R Type Instructions

In a 5-stage pipelined processor, the execution of R-type instructions follows the steps outlined below:

- **Stage 1: Instruction Fetch (IF):** The instruction is fetched from the instruction memory using the program counter (PC). The PC is incremented to point to the next instruction.
- **Stage 2: Instruction Decode (ID):** The fetched instruction is decoded to determine the operation to be performed and the operands involved. The register values corresponding to the source operands are read from the register file.
- **Stage 3: Execute (EX):** The ALU (Arithmetic Logic Unit) performs the required arithmetic or logical operation specified by the instruction. The result of the operation is calculated.
- **Stage 4: Memory Access (MEM):** This stage is not applicable to R-type instructions as they do not involve memory accesses. The stage is used for memory-related operations in load/store instructions.
- **Stage 5: Write Back (WB):** The result of the operation is written back to the destination register in the register file.

2.2 Load Instruction

In a 5-stage pipelined processor, the execution of load type instructions (such as load word - lw) follows the steps outlined below:

- **Stage 1: Instruction Fetch (IF):**

  The instruction is fetched from the instruction memory using the program counter (PC). The PC is incremented to point to the next instruction.

- **Stage 2: Instruction Decode (ID):**

  The fetched instruction is decoded to determine the operation to be performed and the operands involved. The register values corresponding to the source operands are read from the register file.

- **Stage 3: Execute (EX):**

  The effective memory address for the load operation is calculated by adding the base address (stored in a register) with the immediate value (offset) specified in the instruction. The memory address is sent to the data memory.

- **Stage 4: Memory Access (MEM):**

  The data memory retrieves the content stored at the calculated memory address. The fetched data is made available for the subsequent stages.

- **Stage 5: Write Back (WB):**

  The fetched data from the memory is written back to the destination register in the register file.

2.3 Store Instructions

In a 5-stage pipelined processor, the execution of store type instructions (such as store word - sw) follows the steps outlined below:

- **Stage 1: Instruction Fetch (IF):**

  The instruction is fetched from the instruction memory using the program counter (PC). The PC is incremented to point to the next instruction.

- **Stage 2: Instruction Decode (ID):**

  The fetched instruction is decoded to determine the operation to be performed and the operands involved. The register values corresponding to the source operands are read from the register file.

- **Stage 3: Execute (EX):**

  The effective memory address for the store operation is calculated by adding the base address (stored in a register) with the immediate value (offset) specified in the instruction. The data to be stored (from a source register) is passed to the next stage for memory access.

- **Stage 4: Memory Access (MEM):**

  The memory unit uses the calculated memory address from the EX stage to store the data in the data memory.

- **Stage 5: Write Back (WB):**

  There is no write back stage for store type instructions, as they do not produce any result that needs to be written back to a register.

2.4 BEQ Instructions

In a 5-stage pipelined processor, the execution of branch equal (beq) type instructions follows the steps outlined below:

- **Stage 1: Instruction Fetch (IF):**

  The instruction is fetched from the instruction memory using the program counter (PC). The PC is incremented to point to the next instruction.

- **Stage 2: Instruction Decode (ID):**

  The fetched instruction is decoded to determine the operation to be performed and the operands involved. The register values corresponding to the source operands are read from the register file.

- **Stage 3: Execute (EX):**

  The values of the two source registers are compared to check if they are equal. If the comparison is true (registers are equal), the branch target address is calculated by adding the immediate value (offset) specified in the instruction to the incremented PC. If the comparison is false (registers are not equal), the PC is incremented to point to the next instruction without branching.

- **Stage 4: Memory Access (MEM):**

  This stage is not applicable to beq instructions as they do not involve memory accesses. The stage is used for memory-related operations in load/store instructions.

- **Stage 5: Write Back (WB):** This stage is not applicable to beq instructions as they do not produce any result that needs to be written back to a register.

3 5 Stages of Pipeline

3.1 Instruction Fetch Stage (IF):

The instruction fetch stage follows these steps:

- The PC value is used as the address to fetch the instruction from the instruction memory.
- The fetched instruction is stored in the instruction register (IR).
- The PC is incremented to point to the next instruction (PC+4) for sequential instruction execution.
- The fetched instruction in the IR is then passed to the next stage (instruction decode) for further processing.

3.1.1 Components in IF Stage

- **Program Counter** The PC is a special-purpose register that holds the address of the next instruction to be fetched.It keeps track of the current program location and is updated after each instruction fetch. PC is the part of Register File. we have assign register number 31 of register file to the Program Counter.
- **Instruction Memory** The instruction memory stores the program instructions. It is typically implemented as a separate memory unit optimized for instruction fetch operations. The PC value is used as the address to fetch the instruction from the instruction
- **Verilog Code for Instruction Memory**

```
module Instruction_memory(PC_Read_address,
Instruction_Outt);
input [31:0] PC_Read_address; // byte address of current instruction
output reg [31:0] Instruction_Outt; // current Instruction_Outt
reg [7:0] I_M[0:4095]; // Instruction_Outt memory of size 4096 bytes
initial
begin
//sub $t3(11) $t1(9) $t2(10)
```

```
I_M[0] = 8'h01;
I_M[1] = 8'h2a;
I_M[2] = 8'h58;
I_M[3] = 8'h22;
//and $t4(12) $t1(9) $t2(10)
I_M[4] = 8'h01;
I_M[5] = 8'h2a;
I_M[6] = 8'h60;
I_M[7] = 8'h24;
//or $t5(13) $t1(9) $t2(10)
I_M[8] = 8'h01;
I_M[9] = 8'h2a;
I_M[10] = 8'h68;
I_M[11] = 8'h25;
//add $t6(14) $t1(9) $t2(10)
I_M[12] = 8'h01;
I_M[13] = 8'h2a;
I_M[14] = 8'h70;
I_M[15] = 8'h20;
//sub $t7(15) $t6(14) $t1(9)
I_M[16] = 8'h01;
I_M[17] = 8'hc9;
I_M[18] = 8'h78;
I_M[19] = 8'h22;
//or $t8(24) $t6(14) $t7(15)
I_M[20] = 8'h01;
I_M[21] = 8'hcf;
I_M[22] = 8'hc0;
I_M[23] = 8'h25;
//and $t9(25) $t7(15) $t8(24)
I_M[24] = 8'h01;
I_M[25] = 8'hf8;
I_M[26] = 8'hc8;
I_M[27] = 8'h24;
//lw $s5(21) 0x0001($s1(17))
I_M[28] = 8'h8e;
I_M[29] = 8'h35;
I_M[30] = 8'h00;
I_M[31] = 8'h01;
//stall
//add $s6(22) $s5(21) $s2(18)
I_M[32] = 8'h02;
I_M[33] = 8'hb2;
I_M[34] = 8'hb0;
I_M[35] = 8'h20;
// beq $t0(8) $t1(9) 0x0020
I_M[36] = 8'h11;
I_M[37] = 8'h09;
I_M[38] = 8'h00;
I_M[39] = 8'h20;
```

```
//dummy instr2
I_M[40] = 8'h01;
I_M[41] = 8'h51;
I_M[42] = 8'h98;
I_M[43] = 8'h20;
//sub $s7(23) $t1(9) $t2(10)
I_M[168] = 8'h01;
I_M[169] = 8'h2a;
I_M[170] = 8'hb8;
I_M[171] = 8'h22;
end
always @(PC_Read_address)
begin
Instruction_Outt = {I_M[PC_Read_address],
I_M[PC_Read_address+32'd1], I_M[PC_Read_address+32'd2],
I_M[PC_Read_address+32'd3]};
end
endmodule
```

- **PC Incrementer** PC Incrementer Increments the value of Current PC by adding 4 in it. This logic is simply implemented by an adder.
- **PC Incrementer Verilog Code**

```
module Adder_pc4(Current_PC,Next_PC);
input[31:0]Current_PC; //current program counter address
output  [31:0]Next_PC; //next program counter address
assign Next_PC=Current_PC+32'd4;
endmodule
```

- **MUX in IF Stage** In the instruction fetch (IF) stage of a pipelined processor, the multiplexer (mux) is a component used to select the appropriate input for the program counter (PC). It is responsible for determining the next value of the PC based on various conditions or control signals.
  - **Sequential Execution**: In the absence of any branching or jumping instructions, the mux selects the incremented PC value (PC + 4) to fetch the next instruction sequentially.
  - **Branch Instruction:** When a branch instruction is encountered, the mux selects the branch target address if the branch condition is satisfied. The branch target address is typically calculated in the execute (EX) stage, based on the condition and the immediate offset specified in the instruction. If the branch condition is not met, the mux selects the incremented PC value (PC + 4) to fetch the next instruction sequentially.
  - **Jump Instruction:** For jump instructions, the mux selects the target address specified in the instruction to fetch the next instruction from that address. Jump instructions typically involve changing the PC to an absolute address or using a register value to determine the target address.
- **Verilog Code for MUX**

```
module Multiplexer_2x1_32bit(Input1,Input2,Sel,Mux_out);
input[31:0]Input1,Input2;
input Sel;
output reg [31:0]Mux_out;
```

```
always@(Input1 or Input2 or Sel)
begin
if(Sel)
Mux_out<=Input2;
else
Mux_out<= Input1;
end
endmodule
```

3.2 Instruction Decode Stage(ID)

In the instruction decode (ID) stage of a pipelined processor, the fetched instruction from the instruction fetch (IF) stage is decoded to determine the operation to be performed and the operands involved. The instruction decode stage typically involves the following steps:

1 **Instruction Decoding:** The fetched instruction is examined and decoded to determine the operation it represents. The opcode field of the instruction is analyzed to identify the instruction type (e.g., arithmetic, load, store, branch).

2 **Operand Fetch:** The decoded instruction specifies the operands required for the operation. The register values corresponding to the source operands are read from the register file. The register file is typically a set of registers that hold data values.

3 **Immediate Value Extraction (if applicable):** Some instructions, such as immediate arithmetic instructions, load immediate, or branch instructions, may involve an immediate value (constant) as part of the instruction. The immediate value is extracted from the instruction and made available for subsequent stages.

4 **Control Signal Generation:** Based on the instruction type and decoded fields, control signals are generated to coordinate the operations in subsequent pipeline stages. These control signals control the behavior of the execution stage (EX), memory access stage (MEM), and write-back stage (WB) for the current instruction.

5 **Operand Forwarding (if applicable):** In cases where the result of a previous instruction is needed as an operand for the current instruction, forwarding logic determines if the required data is available in the pipeline. If the required data is available, it is forwarded from the appropriate pipeline register to the current stage, bypassing the need to access the register file.

3.2.1 Components in ID stage

- **Control Unit** The control unit in the ID stage performs the following tasks:
  - **Instruction Type Determination:**
    The control unit examines the opcode field of the fetched instruction to identify the instruction type (e.g., arithmetic, load, store, branch). This information is used to determine the specific control signals needed for the instruction's execution.
  - **Control Signal Generation:**
    Based on the identified instruction type and other decoded fields, the control unit generates the necessary control signals for subsequent stages of the pipeline. These control signals dictate the operation of the execution stage (EX), memory access stage (MEM), and write-back stage (WB) for the current instruction.

– **Data Path Configuration:**
  The control unit configures the data path within the processor based on the instruction's requirements. This includes enabling or disabling specific components, setting multiplexer (mux) select lines, and controlling the flow of data and instructions between different stages.

3.2.2 Verilog Code for Control Unit

```
module Control_Unit(Opcode,RegDst,Jump,Branch,MemRead,
MemtoReg,ALUOP1,ALUOP2,MemWrite,ALUSrc,RegWrite,Flush);
input [5:0]Opcode;
output reg RegDst;
output reg Jump;
output reg Branch;
output reg MemRead;
output reg MemtoReg;
output reg ALUOP1;
output reg ALUOP2;
output reg MemWrite;
output reg ALUSrc;
output reg RegWrite;
output reg Flush;
parameter R_type=6'b000000,LW=6'b100011,SW=6'b101011,
BEQ=6'b000100,JumP=6'b000010;// i&R types instructions
initial
begin
RegDst<=1'b0; //make RegDst the destination register
Jump=1'b0;
Branch=1'b0;
MemRead=1'b0;
MemtoReg=1'b0;
ALUOP1=1'b0;
ALUOP2=1'b0;
MemWrite=1'b0;
ALUSrc=1'b0;
RegWrite=1'b0;
Flush = 1'b0;
end
always@(Opcode)
case(Opcode)
R_type: begin    //in case of R-type instruction
RegDst<=1'b1; //make RegDst the destination register
Jump<=1'b0;
Branch<=1'b0;
MemRead<=1'b0;
MemtoReg<=1'b0;
ALUOP1<=1'b1;
ALUOP2<=1'b0;
MemWrite<=1'b0;
ALUSrc<=1'b0;
```

```
RegWrite<=1'b1; //write in registers files
Flush = 1'b0;
end
LW: begin  //in case of load woRegDst instruction
RegDst<=1'b0; //make RegDst the destination register
Jump<=1'b0;
Branch<=1'b0;
MemRead<=1'b1;
MemtoReg<=1'b1;
ALUOP1<=1'b0;
ALUOP2<=1'b0;
MemWrite<=1'b0;
ALUSrc<=1'b1;
RegWrite<=1'b1; //write in registers files
Flush = 1'b0;
end
SW: begin  //in case of store woRegDst instruction
RegDst<=1'bx; //make RegDst the destination register
Jump<=1'b0;
Branch<=1'b0;
MemRead<=1'b0;
MemtoReg<=1'bx;
ALUOP1<=1'b0;
ALUOP2<=1'b0;
MemWrite<=1'b1;
ALUSrc<=1'b1;
RegWrite<=1'b0; //write in registers files
Flush = 1'b0;
end
BEQ: begin //in case of Branch if equal instruction
RegDst<=1'bx; //make RegDst the destination register
Jump<=1'b0;
Branch<=1'b1;
MemRead<=1'b0;
MemtoReg<=1'bx;
ALUOP1<=1'b0 ;
ALUOP2<=1'b1 ;
MemWrite<=1'b0;
ALUSrc<=1'b0;
RegWrite<=1'b0; //write in registers files
Flush <= 1'b1;
end
JumP: begin
RegDst<=1'bx; //make RegDst the destination register
Jump<=1'b1;
Branch<=1'b0;
MemRead<=1'b0;
MemtoReg<=1'bx;
ALUOP1<=1'b0 ;
ALUOP2<=1'b0 ;
```

```
MemWrite<=1'b0;
ALUSrc<=1'b0;
RegWrite<=1'b0; //write in registers files
Flush = 1'b0;
end
endcase
endmodule
```

- **Left Shifter by 2** The left shifter by 2 is typically used in the ID stage for certain types of instructions that involve immediate values or offsets. The purpose of the left shift operation is to scale the immediate value by a factor of 4, adjusting it to represent the appropriate memory address or operand.

  For example, let's consider a load or store instruction with an immediate offset. In such cases, the immediate value may represent the offset or displacement from a base address. By left-shifting the immediate value by 2, the offset is multiplied by 4, aligning it with the byte addressing of the memory system.

- **Verilog Code for Left Shifter by 2**

```
module Shift_left2(input [31:0] Sign_Immex,
output reg [31:0] sll2_out);
always @(Sign_Immex)
begin
sll2_out= Sign_Immex << 2;
end
endmodule
```

- **Sign Extension** The steps involved in sign extension in the ID stage of MIPS are as follows:
  - Retrieve the 16-bit immediate value from the instruction.
  - Check the most significant bit (MSB) of the 16-bit value, which represents the sign bit.
  - If the MSB is 0, the immediate value is positive, and no sign extension is needed.
  - If the MSB is 1, indicating a negative value, sign extension is performed.
  - To extend the sign, the 16-bit value is replicated to fill the remaining 16 bits of the 32-bit value, ensuring that the sign bit is copied to all the additional bits.

- **Verilog code for Sign Extension**

```
module Sign_extended(Immediate,Sign_Immex);
input[15:0]Immediate;
output reg [31:0]Sign_Immex;
always@(Immediate)
if(Immediate[15]==1)
begin
Sign_Immex<={{16{Immediate[15]}},Immediate};
end
else if (Immediate[15]==0)
begin
Sign_Immex<={{16{Immediate[15]}},Immediate};
end
endmodule
```

- **Register File** The register file is a key component used to store and retrieve register values. It serves as a central storage unit for the processor's general-purpose registers and plays a crucial role in instruction execution. The register file present in the ID stage typically consists of multiple registers, each capable of holding a fixed-size value (such as 32 bits in a MIPS architecture). The number of registers can vary depending on the processor's design and instruction set architecture (ISA). In the ID stage, the register file is primarily used for the following purposes:
  - **Operand Retrieval:** The decoded instruction identifies specific source registers required for the instruction's execution. The register file is accessed to retrieve the values stored in these source registers. The retrieved register values are then used as operands in subsequent stages for arithmetic, logic, or memory operations.
  - **Register Value Update:** In some cases, the ID stage may also involve updating the register file with new values. For example, in instructions that write results to registers, the ID stage may decode the destination register and prepare it to receive the computed value. The register file in the ID stage is typically implemented as a set of flip-flops or storage elements organized in an array. Each register has a unique identifier (register number or name) associated with it, allowing easy access and retrieval of register values based on the instruction's requirements.
- **Verilog Code for Register File**

```
module Reg_File_module(clk,rs,rt,rd,write_data,Read_data1,
Read_data2,Reg_write,PC_in,PC_Write,PC_out);
input[4:0]rs,rt,rd;
input[31:0]write_data,PC_in;
output reg [31:0]Read_data1,Read_data2;
output  [31:0]PC_out;
input Reg_write;
input PC_Write;
input clk;
reg[31:0] Reg_File[0:31];
integer i;
initial
begin
for(i=0;i<32;i=i+1)
begin
Reg_File[i]<=32'b0;
end
end
initial
begin
Reg_File[8]<=32'd8;
Reg_File[9]<=32'd8;
Reg_File[10]<=32'd5;
Reg_File[17]<=32'd8;
Reg_File[18]<=32'd6;
Reg_File[19]<=32'd5;
end
assign PC_out = Reg_File[31];
always@(posedge clk)
begin
if(PC_Write)
```

```
Reg_File[31]=PC_in;
if(Reg_write)
begin
Reg_File[rd]<=write_data;
end
end
always@(negedge clk)
begin
Read_data1 = Reg_File[rs];
Read_data2 = Reg_File[rt];
end
endmodule
```

- **Hazard Detection Unit Hazard detection:** The ID stage checks for any potential hazards that could cause problems in the pipeline. Hazards occur when an instruction in the pipeline depends on the result of a previous instruction that has not yet completed its execution. The ID stage detects these hazards and takes corrective action to prevent pipeline stalls or data hazards.

  The hazard detection unit in the ID stage checks for three types of hazards:

  - **Data hazards:** Data hazards occur when an instruction depends on the result of a previous instruction that has not yet completed its execution. The hazard detection unit checks for such dependencies and stalls the pipeline if necessary.
  - **Control hazards:** Control hazards occur when the outcome of a branch instruction is not yet known. The hazard detection unit detects control hazards and flushes the pipeline if necessary.

- **Verilog code for Hazard Detection Unit**

```
module Hazard_Detection_Unit(IF_ID_Reg_Rt,IF_ID_Reg_Rs,
ID_EX_MemRead,ID_EX_Reg_Rt,PC_write,
IF_ID_write,Mux_HDU_out);
input ID_EX_MemRead;
input [4:0] IF_ID_Reg_Rt,IF_ID_Reg_Rs,ID_EX_Reg_Rt;
output reg PC_write,IF_ID_write,Mux_HDU_out;
initial
begin
IF_ID_write = 1'b1;
PC_write = 1'b1;
end
always@(*)
begin
if((ID_EX_MemRead)&&((ID_EX_Reg_Rt==IF_ID_Reg_Rs)
 || (ID_EX_Reg_Rt==IF_ID_Reg_Rt)))
begin
PC_write=0;
IF_ID_write=0;
Mux_HDU_out=1;
end
else
begin
PC_write=1;
```

```
IF_ID_write=1;
Mux_HDU_out=0;
end
end
endmodule
```

- **Adder in ID stage** By adding the PC+4 and the shifted left output, the adder in the ID stage calculates the target address for the branch instruction. The resulting address is then used to update the program counter (PC) to fetch the next instruction from the correct location, either the next sequential instruction or the branch target instruction.

  The adder in the ID stage enables the processor to dynamically compute the branch target address, facilitating conditional branching and the correct flow of program execution based on the branch condition.

- **Verilog code for Adder in ID stage**

```
module Adder_branch( input [31:0]PC_PLUS_4,Sign_Ex_sll_2,
output [31:0] add_out);
assign add_out = PC_PLUS_4 + Sign_Ex_sll_2;
endmodule
```

- **MUX In ID Stage** The mux (multiplexer) in the ID stage, which takes input from the control unit and another input of 0, plays a significant role in controlling the flow of data and instructions within the pipeline processor.The significance of this mux is

  - **Control Signal Selection:**
    The control unit generates various control signals based on the decoded instruction in the ID stage. These control signals determine the behavior of subsequent pipeline stages, such as the execution stage (EX), memory access stage (MEM), and write-back stage (WB). The mux in the ID stage selects between the control unit's generated control signals and the 0 input, based on specific conditions or instructions.

- **Verilog Code for MUX In ID Stage**

```
module Multiplexer_2x1_9bit(Input1,Input2,Sel,Mux_out);
input[8:0]Input1,Input2;
input Sel;
output reg [8:0]Mux_out;
always@(Input1 or Input2 or Sel)
begin
if(Sel)
Mux_out<=Input2;
else
Mux_out<= Input1;
end
endmodule
```

- **Comparator**
  Branch Instruction Decision: Based on the result of the comparison, the comparator generates a control signal indicating whether the branch condition is satisfied or not. If the branch condition is satisfied, the control signal indicates that the branch instruction should be taken, and the program flow should be altered accordingly. If the branch condition is not satisfied, the control signal indicates that the branch instruction should not be taken, and the program continues executing the next sequential instruction.

- **Verilog Code for Comparator**

```
module comparator( input [31:0] in1, in2, output out);
assign out = ~(in1^in2);
endmodule
```

3.3 EX Stage

The execution (EX) stage in a 5-stage pipelined processor is responsible for performing arithmetic, logical, and data manipulation operations on the operands obtained from the instruction decode (ID) stage. The EX stage carries out the actual computation or operation specified by the instruction being executed. Here's an overview of the working of the EX stage:

- **Operand Fetch:**

  The EX stage receives the source operands, which may include values from the register file, immediate values from the instruction, or data forwarded from the previous pipeline stages. These operands are necessary for performing arithmetic, logical, or data manipulation operations.

- **ALU Operation:**

  The EX stage utilizes the Arithmetic Logic Unit (ALU) to perform the specified operation on the operands. The ALU is a digital circuit capable of executing various arithmetic (addition, subtraction, etc.) and logical (AND, OR, etc.) operations. The specific ALU operation to be performed is determined by the control signals generated in the ID stage based on the instruction being executed.

- , **Result Calculation:**

  The ALU computes the result of the operation based on the selected operation and operands. For example, in an addition operation, the ALU adds the two operands together and produces the sum as the result.

- **Data Forwarding and Hazard Handling:**

  The EX stage also handles data hazards by forwarding the result of the ALU operation to subsequent pipeline stages if needed. Data forwarding ensures that dependent instructions receive the correct and updated values promptly, avoiding stalls or pipeline bubbles.

3.4 Components in EX Stage

- **MUX in EX Stage** In a typical 5-stage pipelined processor, the execution (EX) stage involves several multiplexers (muxes) that control the flow of data and select between different inputs. The number of muxes in the EX stage can vary depending on the specific processor design and implementation. However, I will provide a common example of muxes found in the EX stage of a 5-stage pipelined processor:

  - **ALU Operand Mux:**

    This mux selects the operands for the Arithmetic Logic Unit (ALU) operation. It chooses between the immediate value from the instruction or the value from a register, based on the control signals generated in the ID stage.

  - **ALU Control Mux:**

    This mux selects the ALU operation to be performed. It chooses between different arithmetic or logical operations based on the control signals generated in the ID stage.

  - **Register File Read Mux:**
    This mux selects the source registers for the EX stage operation. It chooses between the values read from the register file based on the register numbers extracted from the instruction.
  - **Data Forwarding Muxes:**
    These muxes are responsible for forwarding data from the previous pipeline stages (MEM and WB) to resolve data hazards. They select between the ALU result, memory read data, or register write-back data based on the hazard detection and control signals.

- **Verilog Code for Mux In EX Stage**

```
module Multiplexer_4x1(a,b,c,d,Sel_2,ALU_MUX1_out);
input[31:0] a,b,c,d;
input [1:0]Sel_2;
output reg[31:0] ALU_MUX1_out;
always @(*) begin
 case (Sel_2)
2'b00: ALU_MUX1_out = a;
2'b01: ALU_MUX1_out = b;
2'b10: ALU_MUX1_out = c;
2'b11: ALU_MUX1_out = d;
endcase
end
endmodule
```

- **ALU Control in EX stage** In the execution (EX) stage of a pipelined processor, the ALU control unit plays a critical role in determining the specific operation to be performed by the Arithmetic Logic Unit (ALU). It generates control signals that select the appropriate ALU operation based on the instruction being executed. Here's what the ALU control unit does in the EX stage:
  - **Instruction Decoding:**
    The ALU control unit receives the opcode or instruction code from the instruction decode (ID) stage. It examines the opcode to identify the specific instruction being executed and the operation required. ALU Operation Selection:
    Based on the opcode and other control signals, the ALU control unit generates control signals that select the appropriate ALU operation. These control signals determine whether the ALU should perform addition, subtraction, logical AND, logical OR, shift operations, or any other arithmetic or logical operation supported by the processor's instruction set architecture.
  - **Operand Selection:**
    The ALU control unit also generates control signals that determine which operands are provided to the ALU. It selects the operands from the inputs available in the EX stage, such as register values, immediate values, or data forwarded from the previous pipeline stages. The specific operands required for the selected ALU operation are chosen by the ALU control unit and sent to the ALU.
  - **Control Signal Transmission:**
    The control signals generated by the ALU control unit are passed to the ALU and other relevant components in the EX stage. These control signals guide the ALU in performing the desired operation and assist in managing data forwarding, hazard detection, and control flow within the pipeline.

- **Verilog Code for ALU Control in EX Stage**

```
module ALU_control (ALU_Op1,ALU_Op2,Funct,ALU_Control);
input ALU_Op1,ALU_Op2;
input [5:0]Funct;
output  [2:0]  ALU_Control;
reg [2:0]ALU_Ctrl;
assign ALU_Control=ALU_Ctrl;
always@(ALU_Op1 or ALU_Op2 or Funct)
begin
if (ALU_Op1==0 && ALU_Op2 == 0)
begin
ALU_Ctrl = 3'b010;
end
else if(ALU_Op1==0 && ALU_Op2 == 1)
begin
ALU_Ctrl = 3'b110;
end
else if(ALU_Op1==1 && ALU_Op2 == 0)
begin
case (Funct)
6'b100100:ALU_Ctrl = 3'b000; // and
6'b100101:ALU_Ctrl = 3'b001; // or
6'b100000:ALU_Ctrl = 3'b010; // add
6'b100010:ALU_Ctrl = 3'b110; // sub
6'b101010:ALU_Ctrl = 3'b111; // slt
default:ALU_Ctrl = 3'b000;
endcase
end
end
endmodule
```

- **Verilog Code for ALU In EX Stage**

```
module ALU(R_data1,R_data2,ALU_control,ALU_Result,zero);
parameter
add=3'b010,
And_op=3'b000,
Or_op=3'b001,
sub=3'b110,
slt=3'b111;
input[31:0]R_data1,R_data2; //2 source operands rs ,rt
input[2:0]ALU_control;
output reg[31:0]ALU_Result;
output  zero;
always@(*)
begin
case(ALU_control)
 And_op: ALU_Result = R_data1 & R_data2;
 Or_op: ALU_Result = R_data1 | R_data2;
 add: ALU_Result = R_data1 + R_data2;
 sub: ALU_Result = R_data1 - R_data2;
 slt: begin
if(R_data1<R_data2)
ALU_Result<=32'd1;
else
ALU_Result=32'd0;
end
default:  ALU_Result=R_data1 + R_data2;
endcase
end
assign zero =(ALU_Result==0)?1:0;
endmodule
```

- **Forwarding Unit in EX Stage** The Forwarding Unit in the execution (EX) stage of a pipelined processor plays a crucial role in resolving data hazards and ensuring the correct flow of data within the pipeline. It detects situations where a subsequent instruction depends on the result of a previous instruction that is not yet available in the pipeline. Here's what the Forwarding Unit does in the EX stage:

  - **Hazard Detection:**
    The Forwarding Unit examines the dependencies between instructions in the pipeline to identify potential data hazards. It checks if the current instruction being executed in the EX stage requires a data value that will be produced by a previous instruction.
  - **Data Forwarding:**
    If a data hazard is detected, meaning the required data is not yet available in the pipeline, the Forwarding Unit facilitates data forwarding. Data forwarding allows the result of a previous instruction, which is still in the pipeline, to be forwarded directly to the current instruction in the EX stage that requires it. By bypassing the usual flow of data through the pipeline registers, the Forwarding Unit provides the necessary data to the current instruction without stalling the pipeline.
  - **Operand Selection:**
    The Forwarding Unit determines which data sources to forward to the current instruction. It selects the appropriate forwarding paths, such as forwarding from the ALU result, memory read data, or register write-back data, based on the dependencies and availability of data.
  - **Control Signal Generation:**
    Once the Forwarding Unit determines the forwarding paths and sources, it generates control signals that direct the multiplexers (muxes) in the EX stage to select the correct data source for the current instruction. These control signals ensure that the forwarded data is chosen over the regular data flow to resolve data hazards. By enabling data forwarding, the Forwarding Unit minimizes pipeline stalls and allows dependent instructions to proceed smoothly without waiting for the data to be written back to registers or memory. It ensures that the correct and updated data is available when needed, improving the overall efficiency and performance of the pipelined processor.

- **Verilog Code for Forwarding Unit in EX Stage**

```
module Forwarding_Unit(ID_EX_Reg_Rs,ID_EX_Reg_Rt,EX_MEM_Reg_Rd,EX_MEM_RegWrite,ME
MEM_WB_Reg_Rd,ForwardA,ForwardB);
input [4:0] ID_EX_Reg_Rs,ID_EX_Reg_Rt,EX_MEM_Reg_Rd,MEM_WB_Reg_Rd;
input EX_MEM_RegWrite,MEM_WB_RegWrite;
output reg[1:0]ForwardA,ForwardB;
always@(*)
begin
if((EX_MEM_RegWrite) && ~(EX_MEM_Reg_Rd ==0) && (EX_MEM_Reg_Rd ==ID_EX_Reg_Rs ) )
```

```
ForwardA=2'b10;
else if ((MEM_WB_RegWrite) && ~(MEM_WB_Reg_Rd ==0) && (MEM_WB_Reg_Rd ==ID_EX_Reg_
ForwardA=2'b01;
else
ForwardA=2'b00;
end
always@(*)
begin
if((EX_MEM_RegWrite) && ~(EX_MEM_Reg_Rd ==0) && (EX_MEM_Reg_Rd ==ID_EX_Reg_Rt ) )
ForwardB=2'b10;
else if ((MEM_WB_RegWrite) && ~(MEM_WB_Reg_Rd ==0) && (MEM_WB_Reg_Rd ==ID_EX_Reg_
ForwardB=2'b01;
else
ForwardB=2'b00;
end
endmodule
```

3.5 MEM Stage

In a 5-stage pipelined processor, the MEM (memory access) stage is the third stage of the pipeline. It primarily deals with memory-related operations, such as accessing data from memory or writing data back to memory. Here's an overview of the working of the MEM stage:

- **Load/Store Instruction Execution:** The MEM stage primarily handles load and store instructions. For a load instruction, the MEM stage performs the memory read operation to fetch data from memory. For a store instruction, the MEM stage performs the memory write operation to store data into memory.

- **Memory Address Calculation:** The MEM stage calculates the memory address required for the load or store operation. For a load instruction, the memory address is typically obtained by adding an immediate offset to the base register value. For a store instruction, the memory address is calculated in a similar manner.

- **Memory Access:** In the MEM stage, the calculated memory address is used to access the memory hierarchy. If it is a load instruction, the MEM stage retrieves the data from the memory location addressed. If it is a store instruction, the MEM stage writes the data from the pipeline registers to the memory location addressed.

- **Data Forwarding:** The MEM stage may also perform data forwarding to resolve data hazards. If the data produced by the previous instruction (EX stage) is required by a subsequent instruction, it may be forwarded directly to the necessary stage in the pipeline.

- **Result Preparation:** After the memory access operation is completed, the MEM stage prepares the result or data to be passed on to the next stage (WB - write-back stage) of the pipeline. For a load instruction, the data retrieved from memory is passed on as the result. For a store instruction, there is no specific result to pass on, so the MEM stage may pass control signals or other relevant information to the WB stage.

- **Control Signal Generation:** The MEM stage generates control signals for the subsequent pipeline stages based on the instruction being executed. These control signals dictate the behavior of the following stages, such as data forwarding, branch prediction, or control flow adjustments.

  The MEM stage ensures proper memory access, data retrieval, and storage operations in a pipelined processor. It handles load/store instructions, calculates memory addresses, performs memory accesses, forwards data when necessary, prepares results, and generates control signals for the subsequent pipeline stages. The efficient functioning of the MEM stage contributes to the overall performance and functionality of the processor.

3.5.1 Components in MEM Stage

- **verilog Code for Data Memory**

```
module Data_memory(input Mem_Write,Mem_Read,clk,input[31:0]Address
,Write_data,output reg [31:0] Mem_Data);
reg[7:0] Data_Memory[0:1023];
initial
begin
Data_Memory[9] = 8'h00;
Data_Memory[10] = 8'h00;
Data_Memory[11] = 8'h04;
Data_Memory[12] = 8'h01;
end
always@(*)
begin
if(Mem_Read)
begin
Mem_Data[31:24]=Data_Memory[Address];
Mem_Data[23:16]=Data_Memory[Address+32'd1];
Mem_Data[15:8]=Data_Memory[Address+32'd2];
Mem_Data[7:0]=Data_Memory[Address+32'd3];
end
end
always@(posedge clk)
begin
if(Mem_Write)
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
end
endmodule
```

3.6 WB Stage

In a 5-stage pipelined processor, the WRITE BACK (WB) stage is the final stage of the pipeline. It is responsible for updating the register file or other appropriate destination with the results of the executed instruction. Here's an overview of the working of the WB stage:

- **Result Availability:** The WB stage receives the result or output data produced by the previous stage, which is typically the MEM (memory access) stage. This result can be the data retrieved from memory in the case of a load instruction or the result of an ALU operation in other instructions.

- **Register File Update:** The WB stage updates the register file with the computed result. If the instruction being executed is a register-register instruction, the WB stage writes the result back to the appropriate register location in the register file. The register file is a component that stores the register values used by the processor for data manipulation and control flow.
- **Data Forwarding:** If subsequent instructions in the pipeline are dependent on the result being written back to the register file, data forwarding may occur. Data forwarding allows the result to be forwarded directly to the subsequent instruction, bypassing the usual flow through the register file. This ensures that dependent instructions receive the correct and updated values promptly, avoiding stalls or pipeline bubbles.
- **Control Signal Generation:** The WB stage generates control signals that communicate the completion of the instruction execution and update to the register file. These control signals may include signals to clear or update internal flags or status bits related to the executed instruction.
- **Program Counter (PC) Update:** In some cases, the WB stage may update the program counter (PC) value to determine the address of the next instruction to be fetched. For example, in branch or jump instructions, the WB stage may modify the PC based on the branch or jump target address calculated in the previous stages.

  The primary role of the WB stage is to write back the result of the executed instruction to the appropriate destination, typically the register file. It ensures that the correct and updated data is stored in the register file for subsequent instructions to access. Additionally, the WB stage may update the PC and generate control signals to reflect the completion of instruction execution and any necessary changes to the program flow.

  Overall, the proper functioning of the WB stage ensures the correctness and efficiency of instruction execution in the pipelined processor.

3.7 Pipe Registers

3.7.1 IF/ID Register

The IF/ID register, also known as the Instruction Fetch/Decode register, is a pipeline register that exists between the Instruction Fetch (IF) stage and the Instruction Decode (ID) stage in a pipelined processor. It serves as a temporary storage location for the fetched instruction and related control signals. Here's an overview of the IF/ID register:

- **Instruction Storage:** The IF/ID register stores the fetched instruction from the Instruction Cache or Instruction Memory in the IF stage. It holds the binary representation of the instruction, which includes the opcode, operands, and any immediate values.
- **Control Signal Storage:** Along with the instruction, the IF/ID register also stores control signals related to the fetched instruction. These control signals can include signals such as branch prediction results, branch target addresses, or flags indicating the type of instruction (e.g., load, store, ALU operation).
- **Transfer of Data and Control Signals:** The IF/ID register facilitates the transfer of the fetched instruction and control signals from the IF stage to the ID stage. It ensures a synchronized and controlled movement of data between the two stages of the pipeline.
- **Data Synchronization:** The IF/ID register helps in synchronizing the flow of instructions through the pipeline. It acts as a buffer that allows the IF stage to fetch the next instruction while the ID stage processes the current instruction. This helps maintain a steady flow of instructions and enables parallel processing.
- **Stalling and Hazard Detection:** The IF/ID register is also involved in hazard detection and pipeline stalling. It allows the pipeline to detect hazards, such as data dependencies or control hazards, by providing access to

the fetched instruction and associated control signals. If a hazard is detected, the pipeline may introduce stall cycles to resolve the hazard before proceeding with the execution.

The IF/ID register plays a crucial role in the proper functioning of a pipelined processor by facilitating the transfer of instructions and control signals between the IF and ID stages. It helps maintain the continuity of instruction flow, enables synchronization, and assists in hazard detection and pipeline control.

3.7.2 Verilog Code for IF/ID Register

```verilog
module Register_IF_ID(clk,IF_ID_in1,IF_ID_Ctrl,IF_ID_out1,Flush);
input clk;
input IF_ID_Ctrl,Flush;
input [63:0]IF_ID_in1;
output reg [63:0]IF_ID_out1;
always@(posedge clk)
begin
if (Flush)
IF_ID_out1 = 64'b0;
else if (~Flush)
begin
if(IF_ID_Ctrl)
IF_ID_out1<=IF_ID_in1;
end
end
endmodule
```

3.7.3 ID/EX Register

The ID/EX register, also known as the Instruction Decode/Execute register, is a pipeline register that resides between the Instruction Decode (ID) stage and the Execute (EX) stage in a pipelined processor. It serves as a temporary storage location for the decoded instruction and related control signals. Here's an overview of the ID/EX register:

- **Instruction Storage:** The ID/EX register stores the decoded instruction received from the ID stage. It holds the information extracted from the instruction, such as the opcode, source and destination registers, immediate values, and other relevant data.
- **Control Signal Storage:** Along with the instruction, the ID/EX register also stores control signals related to the decoded instruction. These control signals include signals such as ALU operation codes, data forwarding control signals, branch prediction results, or flags indicating instruction type and behavior.
- **Data Transfer:** The ID/EX register facilitates the transfer of the decoded instruction and control signals from the ID stage to the EX stage. It ensures a synchronized and controlled movement of data between these two stages of the pipeline.
- **Data Forwarding and Hazard Detection:** The ID/EX register assists in forwarding data from previous pipeline stages to resolve data hazards. It provides access to the decoded instruction's source registers, allowing subsequent instructions to access the necessary data without stalling the pipeline. The register may also play a role in hazard detection, identifying dependencies and initiating hazard resolution mechanisms if required.

- **Storing Intermediate Results:** The ID/EX register can store intermediate results or calculations performed in the ID stage. This can be useful when multiple pipeline stages require access to the same data or when calculations need to be carried out over multiple clock cycles.

- **Pipeline Control:** The ID/EX register may contain control signals that help manage the pipeline's behavior. This can include signals for enabling or disabling specific pipeline features, branch prediction information, or branch target addresses.

  The ID/EX register acts as a bridge between the ID and EX stages, ensuring the smooth transfer of decoded instructions, control signals, and intermediate results. It enables data forwarding, assists in hazard detection, and supports the coordination and control of pipeline operations. By efficiently managing the flow of information between these pipeline stages, the ID/EX register contributes to the overall performance and functionality of the pipelined processor.

3.7.4 Verilog Code For ID/EX Stage Regsiter

```
module Register_ID_EX(clk,ID_EX_in1,ID_EX_out1);
input clk;
input [124:0]ID_EX_in1;
output reg [124:0]ID_EX_out1;
always@(posedge clk)
begin
ID_EX_out1<=ID_EX_in1;
end
endmodule
```

3.7.5 EX/MEM Stage Register

The EX/MEM register, also known as the Execute/Memory Access register, is a pipeline register that resides between the Execute (EX) stage and the Memory Access (MEM) stage in a pipelined processor. It serves as a temporary storage location for the results and control signals generated by the execution stage. Here's an overview of the EX/MEM register:

- **Result Storage:** The EX/MEM register stores the results generated by the execution stage, which typically involve the ALU (Arithmetic Logic Unit) operations or other calculations. It holds the computed result of the executed instruction, such as the arithmetic result, logical result, or memory address.

- **Control Signal Storage:** Along with the results, the EX/MEM register also stores control signals related to the executed instruction. These control signals can include flags indicating the status of the executed instruction (e.g., overflow, zero result), signals for memory-related operations (e.g., load or store), and other relevant control information.

- **Data Transfer:** The EX/MEM register facilitates the transfer of the results and control signals from the EX stage to the MEM stage. It ensures a synchronized and controlled movement of data between these two stages of the pipeline.

- **Data Forwarding and Hazard Detection:** The EX/MEM register assists in forwarding data from previous pipeline stages to resolve data hazards. It provides access to the results generated by the execution stage, allowing subsequent instructions to access the necessary data without stalling the pipeline. The register may also play a role in hazard detection, identifying dependencies and initiating hazard resolution mechanisms if required.

- **Memory Access:** In some cases, the EX/MEM register may contain the memory address for load or store instructions. It holds the address information to be used in the subsequent MEM stage for accessing data in memory.
- **Pipeline Control:** - The EX/MEM register may contain control signals that help manage the pipeline's behavior. - This can include signals for enabling or disabling specific pipeline features, flags indicating the need for further pipeline stages (e.g., branch prediction), or branch target addresses.

  The EX/MEM register acts as a temporary storage location for the results and control signals generated by the execution stage. It enables the smooth transfer of data between the EX and MEM stages, facilitates data forwarding, assists in hazard detection, and supports the coordination and control of pipeline operations. By efficiently managing the flow of information between these pipeline stages, the EX/MEM register contributes to the overall performance and functionality of the pipelined processor.

3.7.6 Verilog Code for EX/MEM Register

```verilog
module Register_EX_MEM(clk,EX_MEM_in1,EX_MEM_out1);
input clk;
input [74:0]EX_MEM_in1;
output reg [74:0]EX_MEM_out1;
always@(posedge clk)
begin
EX_MEM_out1<=EX_MEM_in1;
end
endmodule
```

3.7.7 MEM/WB Stage Register

The EX/MEM register, also known as the Execute/Memory Access register, is a pipeline register that resides between the Execute (EX) stage and the Memory Access (MEM) stage in a pipelined processor. It serves as a temporary storage location for the results and control signals generated by the execution stage. Here's an overview of the EX/MEM register:

- **Result Storage:** The EX/MEM register stores the results generated by the execution stage, which typically involve the ALU (Arithmetic Logic Unit) operations or other calculations. It holds the computed result of the executed instruction, such as the arithmetic result, logical result, or memory address.
- **Control Signal Storage:** Along with the results, the EX/MEM register also stores control signals related to the executed instruction. These control signals can include flags indicating the status of the executed instruction (e.g., overflow, zero result), signals for memory-related operations (e.g., load or store), and other relevant control information.
- **Data Transfer:** The EX/MEM register facilitates the transfer of the results and control signals from the EX stage to the MEM stage. It ensures a synchronized and controlled movement of data between these two stages of the pipeline.
- **Data Forwarding and Hazard Detection:** The EX/MEM register assists in forwarding data from previous pipeline stages to resolve data hazards. It provides access to the results generated by the execution stage, allowing subsequent instructions to access the necessary data without stalling the pipeline. The register may also play a role in hazard detection, identifying dependencies and initiating hazard resolution mechanisms if required.

- **Memory Access:** In some cases, the EX/MEM register may contain the memory address for load or store instructions. It holds the address information to be used in the subsequent MEM stage for accessing data in memory.
- **Pipeline Control:** The EX/MEM register may contain control signals that help manage the pipeline's behavior. This can include signals for enabling or disabling specific pipeline features, flags indicating the need for further pipeline stages (e.g., branch prediction), or branch target addresses.

  The EX/MEM register acts as a temporary storage location for the results and control signals generated by the execution stage. It enables the smooth transfer of data between the EX and MEM stages, facilitates data forwarding, assists in hazard detection, and supports the coordination and control of pipeline operations. By efficiently managing the flow of information between these pipeline stages, the EX/MEM register contributes to the overall performance and functionality of the pipelined processor.

3.7.8 Verilog Code for MEM/WB Register

```verilog
module Register_MEM_WB(clk,MEM_WB_in1,MEM_WB_out1);
input clk;
input [70:0]MEM_WB_in1;
output reg [70:0]MEM_WB_out1;
always@(posedge clk)
begin
MEM_WB_out1<=MEM_WB_in1;
end
endmodule
```

4 Datapath of Pipeine Stages

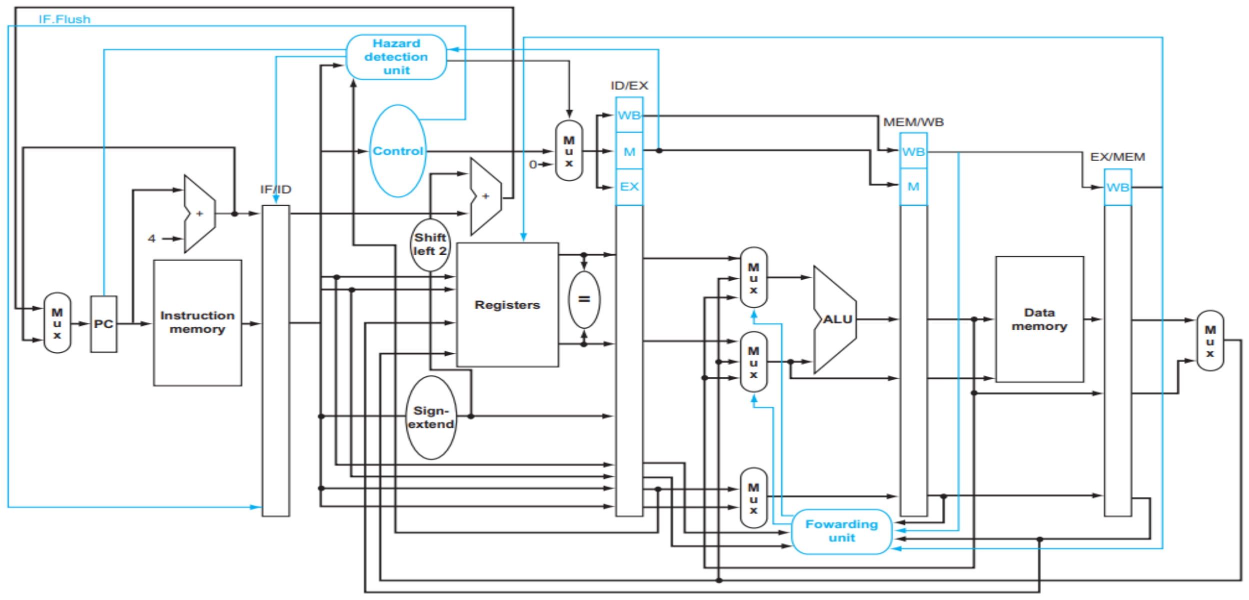


Figure 1: Datapath of Pipeine Stages

5 Testcase

5.1 Non Dependence Instructions

1.SUB $t3,$t1,$t2

In a MIPS 32-bit pipelined processor, the instruction SUB $t3,$t1,$t2 subtracts the value in register$t2 from the value in register $t1 and stores the result in register $t3.
Here subtraction operation will perform and value will goes to register file[11]=3.

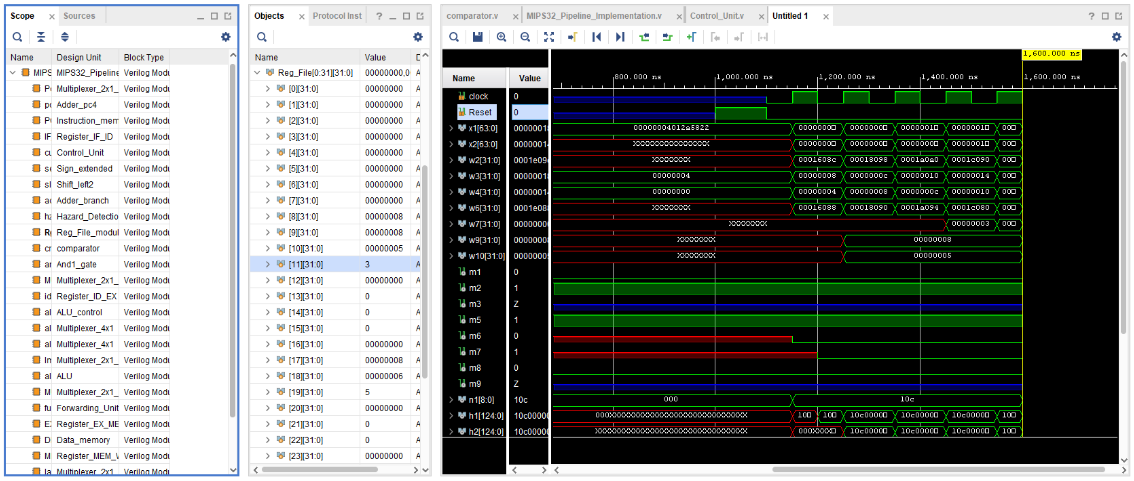


Figure 2: Subtraction Operation

2.and $t4,$t1,$t2

In a MIPS 32-bit pipelined processor, the instruction and $t4, $t1, $t2 performs a bitwise logical AND operation between the values in registers $t1 and $t2, and stores the result in register $t4
Here and operation will perform and value will goes to register file[12]=0.

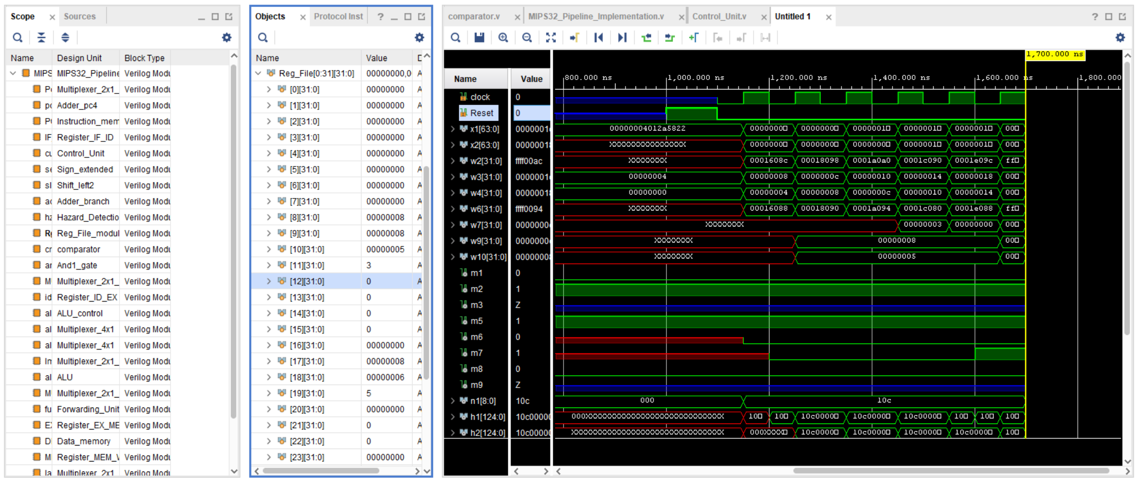


Figure 3: AND Operation

3.OR $t5,$t1,$t2

In a MIPS 32-bit pipelined processor, the instruction or \$t5, \$t1, \$t2 performs a bitwise logical OR operation between the values in registers \$t1 and \$t2, and stores the result in register \$t5. Here or operation will perform and value will goes to register file[13]=13.

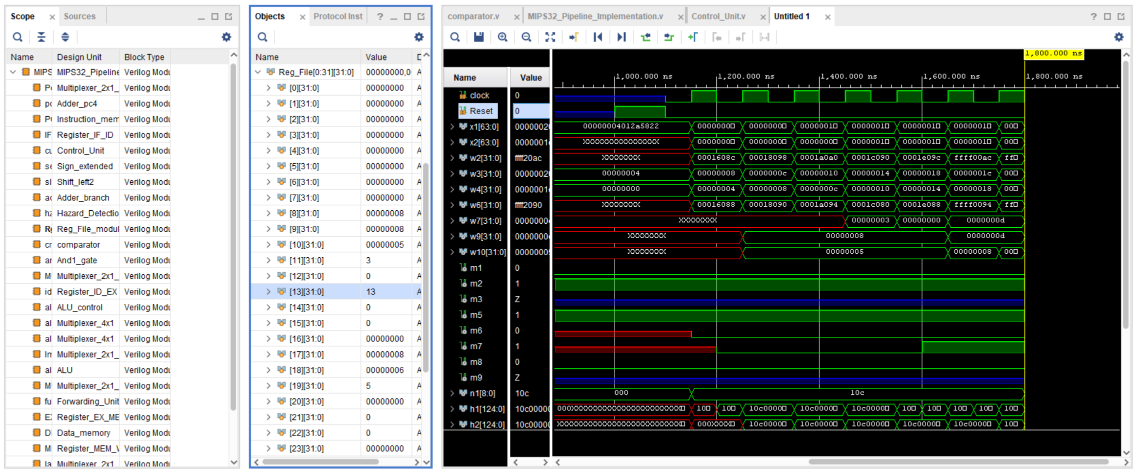


Figure 4: OR Operation

4.ADD \$t6,\$t1,\$t2

In a MIPS 32-bit pipelined processor, the instruction add \$t6, \$t1, \$t2 performs an addition operation between the values in registers \$t1 and \$t2, and stores the result in register \$t6. Here ADD operation will perform and value will goes to register file[14]=13.

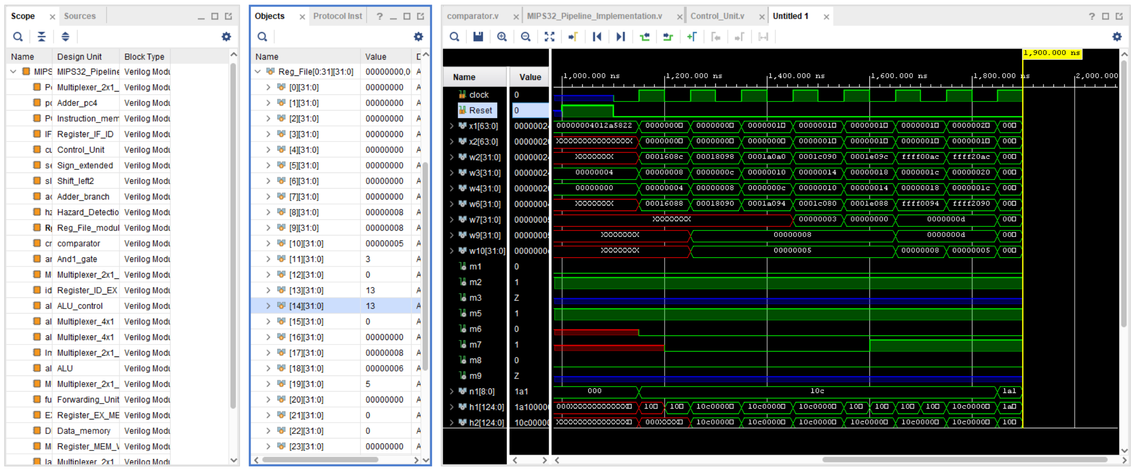
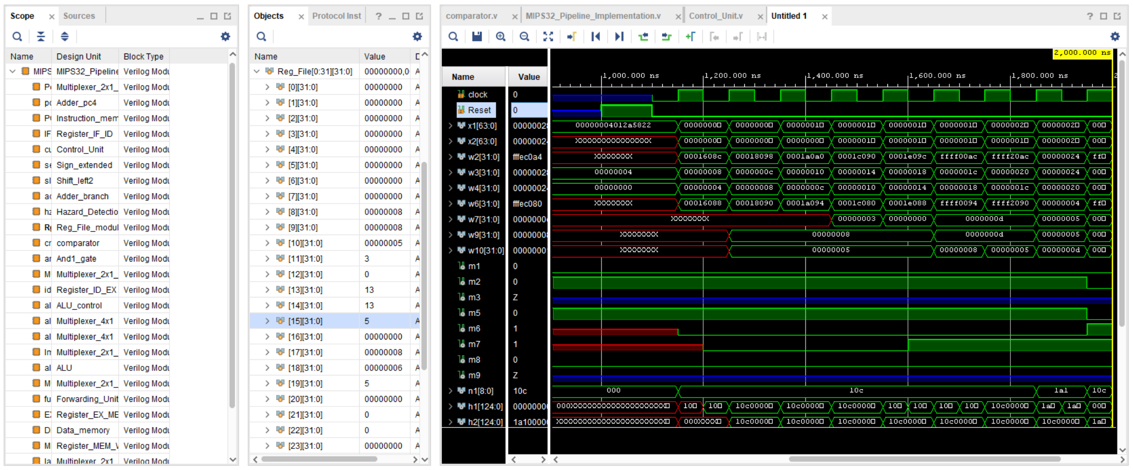


Figure 5: ADD Operation

5.2 Dependence Instructions/Data Hazard

5.SUB $t7,$t6,$t1

There is a dependency between the instructions ADD(previous instructions) and SUB.
The result of the ADD instruction is stored in $t6, and then $t6 is used as an operand in the SUB instruction.
This creates a data dependency known as a Data hazard.This hazard can be avoided by the Forwarding the path.
Here SUB operation will perform and value will goes to register file[15]=5.

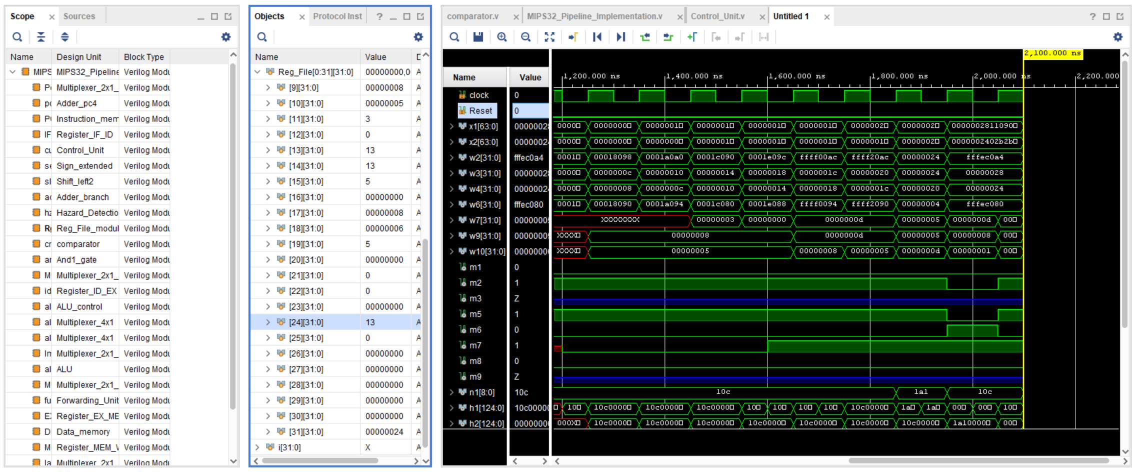


Name	Value
Reg_File[0:31][31:0]	00000000,0
[0][31:0]	00000000
[1][31:0]	00000000
[2][31:0]	00000000
[3][31:0]	00000000
[4][31:0]	00000000
[5][31:0]	00000000
[6][31:0]	00000000
[7][31:0]	00000000
[8][31:0]	00000008
[9][31:0]	00000008
[10][31:0]	00000005
[11][31:0]	3
[12][31:0]	0
[13][31:0]	13
[14][31:0]	13
[15][31:0]	5
[16][31:0]	00000000
[17][31:0]	00000008
[18][31:0]	00000006
[19][31:0]	5
[20][31:0]	00000000
[21][31:0]	0
[22][31:0]	0
[23][31:0]	00000000

Figure 6: SUB Operation

6.OR $t8,$t6,$t7

There is a dependency between the ADD instructions and OR instructions.
The result of the ADD instruction is stored in $t6, and then $t6 is used as an operand in the OR instruction.
This creates a data dependency known as a Data hazard. This hazard can be avoided by the Forwarding the path.
Here OR operation will perform and value will goes to register file[24]=13.

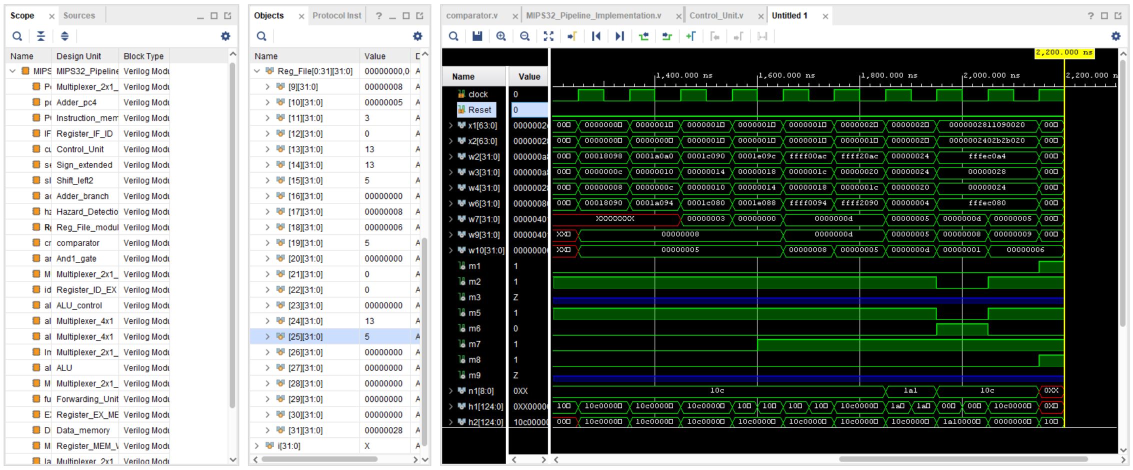


Name	Value
Reg_File[0:31][31:0]	00000000,0
[9][31:0]	00000008
[10][31:0]	00000005
[11][31:0]	3
[12][31:0]	0
[13][31:0]	13
[14][31:0]	13
[15][31:0]	5
[16][31:0]	00000000
[17][31:0]	00000008
[18][31:0]	00000006
[19][31:0]	5
[20][31:0]	00000000
[21][31:0]	0
[22][31:0]	0
[23][31:0]	00000000
[24][31:0]	13
[25][31:0]	0
[26][31:0]	00000000
[27][31:0]	00000000
[28][31:0]	00000000
[29][31:0]	00000000
[30][31:0]	00000000
[31][31:0]	00000024
i[31:0]	X

Figure 7: OR Operation

7.and $t9,$t7,$t8

There is a dependency between the OR (Previous instructions) and and instructions.
The result of the OR instruction is stored in $t8, and then $t* is used as an operand in the and instruction.
This creates a data dependency known as a Data hazard. This hazard can be avoided by the Forwarding the path.
Here and operation will perform and value will goes to register file[25]=5.



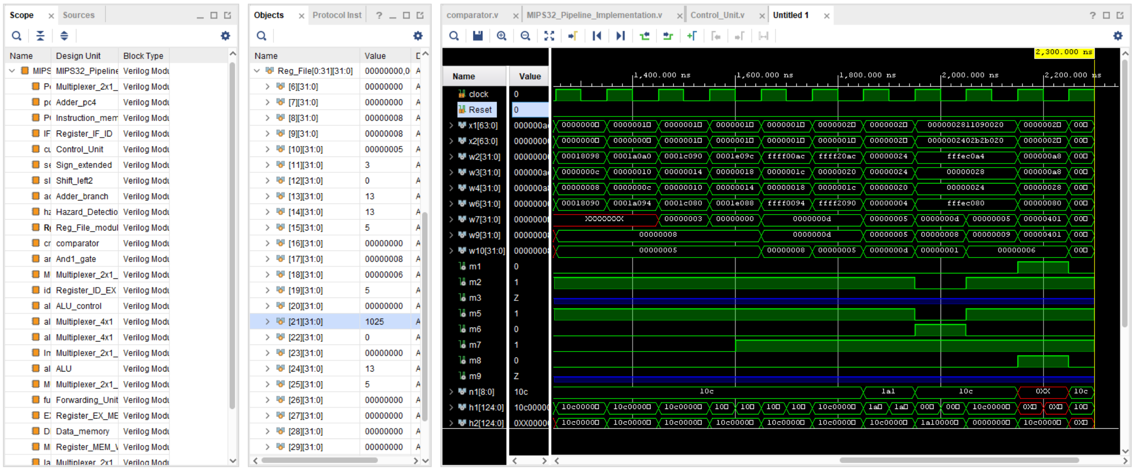
Name	Value
Reg_File[0:31][31:0]	00000000,0
[9][31:0]	00000008
[10][31:0]	00000005
[11][31:0]	3
[12][31:0]	0
[13][31:0]	13
[14][31:0]	13
[15][31:0]	5
[16][31:0]	00000000
[17][31:0]	00000008
[18][31:0]	00000006
[19][31:0]	5
[20][31:0]	00000000
[21][31:0]	0
[22][31:0]	0
[23][31:0]	00000000
[24][31:0]	13
[25][31:0]	5
[26][31:0]	00000000
[27][31:0]	00000000
[28][31:0]	00000000
[29][31:0]	00000000
[30][31:0]	00000000
[31][31:0]	00000028
i[31:0]	X

Figure 8: And Operation

5.3 Hazard Detection Unit

8.lw $s5(21) 0x0001($s1(17))

The instruction loads a 32-bit word from memory into register $s5. The effective memory address is calculated by adding the immediate value 0x0001 to the value in register $s1. The memory access stage retrieves the word data from memory at the calculated address, and the write back stage stores the retrieved data into register $s5.The vaule that load in the register file[21]=1025.

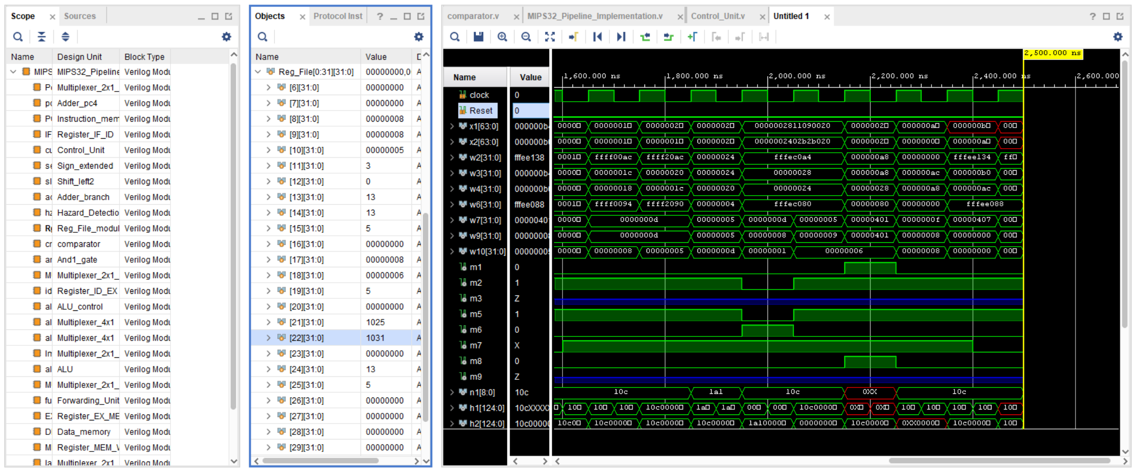


Name	Value
Reg_File[0:31][31:0]	00000000,0
[6][31:0]	00000000
[7][31:0]	00000000
[8][31:0]	00000008
[9][31:0]	00000008
[10][31:0]	00000005
[11][31:0]	3
[12][31:0]	0
[13][31:0]	13
[14][31:0]	13
[15][31:0]	5
[16][31:0]	00000000
[17][31:0]	00000008
[18][31:0]	00000006
[19][31:0]	5
[20][31:0]	00000000
[21][31:0]	1025
[22][31:0]	0
[23][31:0]	00000000
[24][31:0]	13
[25][31:0]	5
[26][31:0]	00000000
[27][31:0]	00000000
[28][31:0]	00000000
[29][31:0]	00000000

Figure 9: Load

9.ADD $s6,$s5,$s2

These two instructions load a word from memory into register $s5, and then add the value in register $s2 to the value in $s5 and store the result in register $s6. There is a data dependency between these two instructions, as the add instruction depends on the value that is loaded into $s5 by the lw instruction. Specifically, the result of the lw instruction must be written into $s5 before it can be used by the add instruction. Therefore, the add instruction cannot begin execution until the lw instruction has completed its write back stage and written the result into $s5. Therefore we need to stall the pipeined to avoid this type of hazard. The value after the execution of this instruction the Register file[22]=1031.



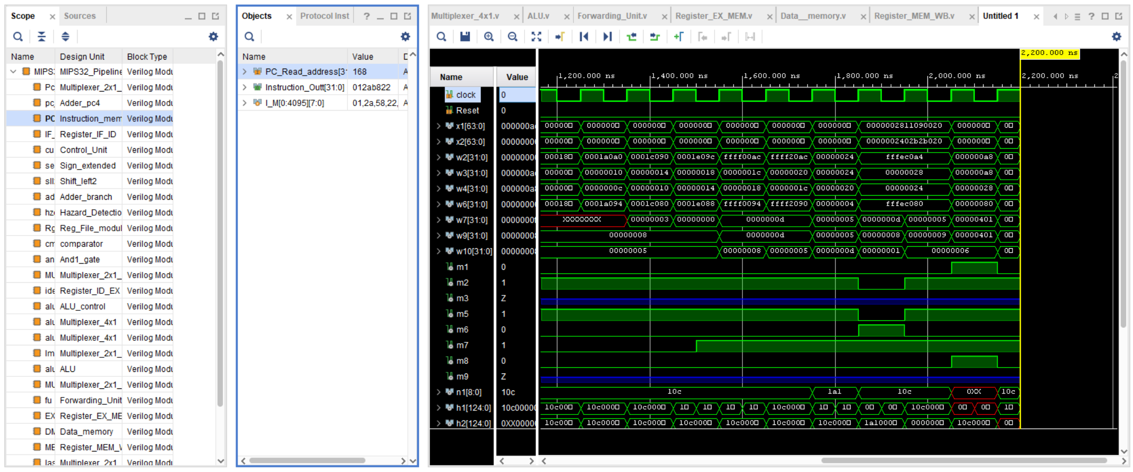
Name	Value
Reg_File[0:31][31:0]	00000000,0
[6][31:0]	00000000
[7][31:0]	00000000
[8][31:0]	00000008
[9][31:0]	00000008
[10][31:0]	00000005
[11][31:0]	3
[12][31:0]	0
[13][31:0]	13
[14][31:0]	13
[15][31:0]	5
[16][31:0]	00000000
[17][31:0]	00000008
[18][31:0]	00000006
[19][31:0]	5
[20][31:0]	00000000
[21][31:0]	1025
[22][31:0]	1031
[23][31:0]	00000000
[24][31:0]	13
[25][31:0]	5
[26][31:0]	00000000
[27][31:0]	00000000
[28][31:0]	00000000
[29][31:0]	00000000

Figure 10: add

5.4 Control Hazard

10. beq $t0(8) $t1(9) 0x0020

The instruction performs a comparison between the values in registers *t0andt*1. If they are equal, the PC is updated to the target address 0x0020. This instruction is typically used for conditional branching and altering the control flow of the program based on the result of the comparison.



Name	Value
PC_Read_address[3	168
Instruction_Out[31:0]	012ab822
I_M[0:4095][7:0]	01,2a,58,22,

Figure 11: Beq Instruction

11.ADD $s7,$t1,$t2

There is a control hazard between the branch instruction ”beq $t0, $t1, 0x0020” and the subsequent instruction ”ADD $s7, $t1,$t2”. A control hazard, also known as a branch hazard, occurs when the execution of instructions is affected by a branch instruction’s outcome.This hazard is removed by flushing the instructions.

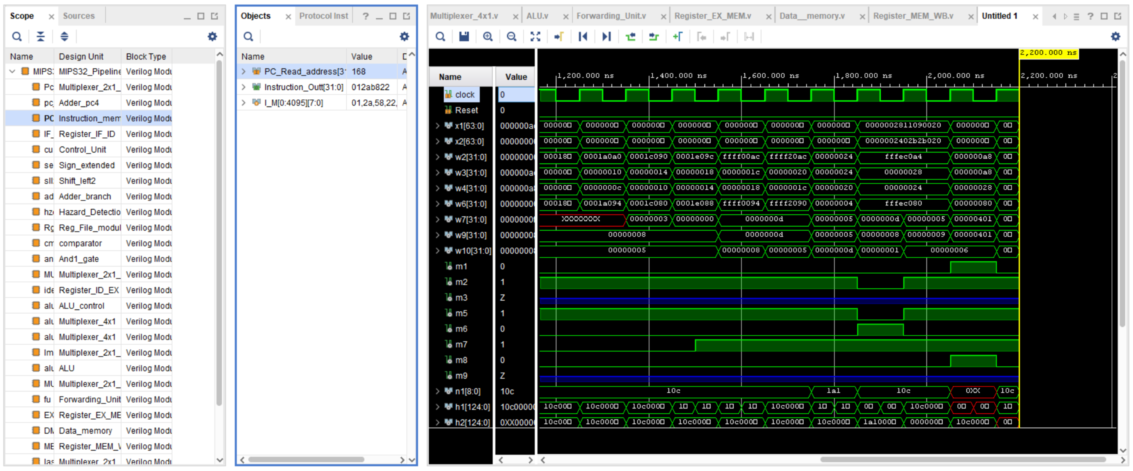


Figure 12: Add Instruction