Department of Electrical & Computer Engineering
AIR UNIVERSITY

PROJECT REPORT

Computer Organization & Assembly Language

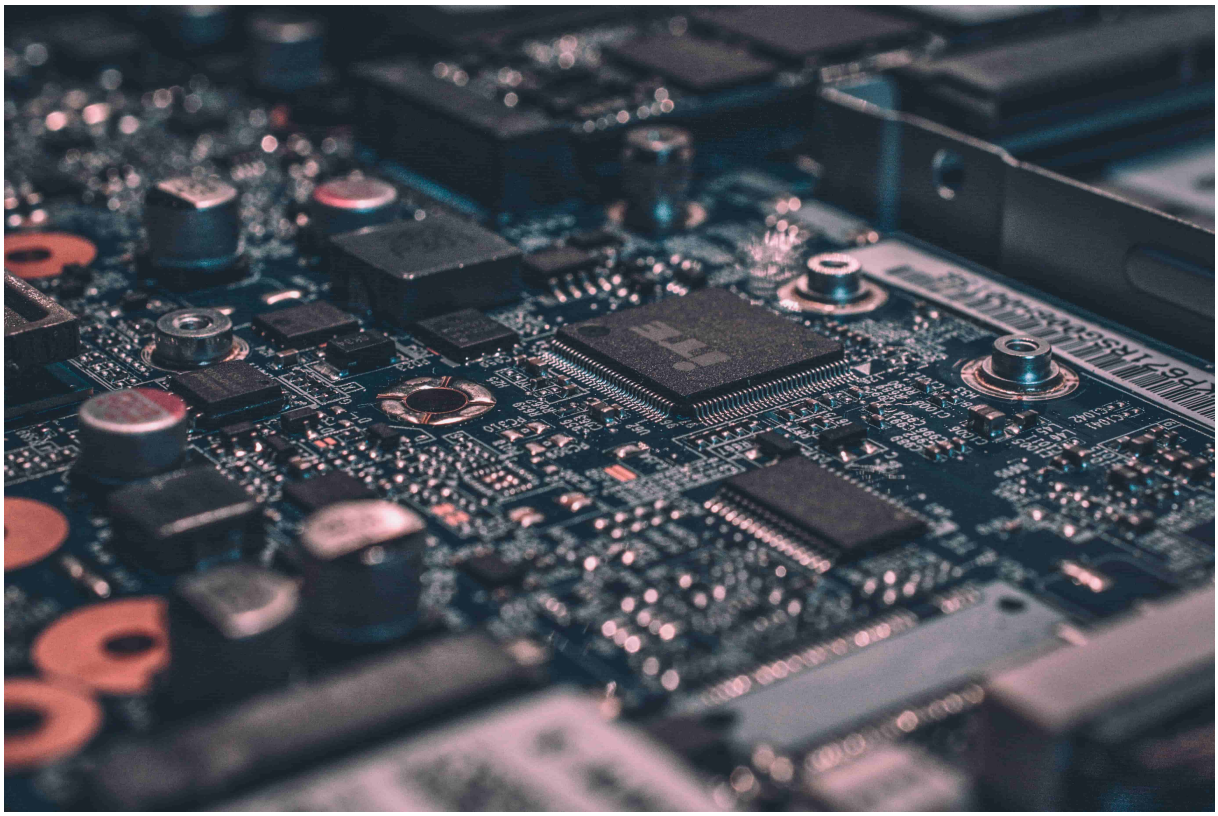
Instructor Name

Sir Najam Dar
Miss Ayesha Sadiq

Submitted by

Muhammad Burhan Ahmed
Agha Ammar Khan
Bilal Ijaz

05/14/2023



Implementation of 16-Bit RISC Processor

ABSTRACT

This project aims to design a RISC processor which is capable enough of executing basic instructions such as load/store, R-type and branch instructions. The design of this processor is a substantial contribution to the field of computer engineering and is based on a MIPS architecture(specific implementation of the RISC) with separate blocks for a smooth data flow. The datapath consists of five blocks which include the instruction fetch unit, the instruction decode, the execution unit, the memory unit, and the write unit. The processor is designed using a Verilog HDL and simulated using Xilinx software. The final implementation of the processor is tested using the MIPS test bench. The outcome of this project is a functional 16-bit data path that can execute a 16-bit instruction.

Contents

1 OBJECTIVES

- Implement a datapath using the concepts of MIPS.
- Hands-on experience in designing
- Understand the internal workings of a CPU.
- Familiarization with Verilog HDL.

1.0.1 Software Used :

- Xilinx

2 INTRODUCTION

MIPS (Microprocessor without interlocked pipeline stages)

MIPS is a reduced instruction set computer (RISC) architecture that allows for less cycles per instruction and is one of the earliest RISC Instruction set designs. Instead of a vast number of complex and specific instructions, such a computer has a limited amount of basic and generic commands. MIPS instructions are fixed at 32 bits in width.

Blocks of MIPS

MIPS datapath is composed of the following components:

- Instruction Memory(IM)
- Program Counter(PC)
- Instruction Register(IR)
- Register File(RF)
- ALU (Arithmetic Logic Unit)
- Data Memory(DM)
- Control Unit(CU)
- Multiplexers(MUXes)
- Sign Extender
- Shifters

2.1 Instruction Memory(IM)

Instruction Memory (IM) is a fundamental component of a computer system that stores the program instructions to be executed by the processor. It is a type of memory specifically designed to hold the instructions that make up a program. The IM is responsible for providing the instructions to the processor in the correct sequence and at the appropriate time during the execution process. The IM is typically organized as a sequential collection of memory locations, each containing a single instruction. These instructions are stored in a format that can be easily interpreted and executed by the processor. The size of the IM is determined by the number of instructions it can hold, and it is often specified in terms of the number of words or bytes. During the execution of a program, the processor fetches instructions from the IM and transfers them to the instruction register (IR) for decoding and execution. The IM is accessed using the program counter (PC), which keeps track of the memory address of the next instruction to be fetched. The IM plays a crucial role in the overall performance of a computer system. The speed and efficiency of accessing instructions from the IM can impact the execution time of a program. Therefore, optimizing the design and access methods of the IM is essential to enhance the overall system performance.

2.2 Program Counter(PC)

The Program Counter (PC) is a vital component in the execution of computer programs. It is a special register that holds the memory address of the next instruction to be fetched and executed by the processor. The PC keeps track of the program's current position in memory and determines the sequence in which instructions are executed. For a person with a technical understanding, the Program Counter is a hardware register that stores the address of the next instruction in memory. It is incremented after each instruction is fetched, indicating the location of the subsequent instruction. The PC plays a crucial role in the control flow of the program, ensuring that instructions are executed in the correct order. On the other hand, for a non-technical audience, the Program Counter can be explained as a "pointer" that guides the computer's execution through a program. It keeps track of the computer's progress through a series of steps or instructions, making sure that each step is completed before moving on to the next. Think of it as a roadmap that helps the computer know which instruction comes next and ensures that everything is done in the right order. In adapting the paragraphs for different audiences, I made the following writing decisions: Technical Version: Emphasized the technical details of the Program Counter, such as its role as a hardware register and its relationship with memory addresses. Non-Technical Version: Simplified the explanation by using analogies and relatable terms to help the non-technical audience understand the concept without getting overwhelmed by technical jargon.

2.3 Instruction Register(IR)

The Instruction Register (IR) is a register in a computer's CPU (Central Processing Unit) that holds the current instruction being executed. It is a part of the instruction cycle and plays a crucial role in fetching and decoding instructions. When the CPU fetches an instruction from memory, it is stored in the Instruction Register. The contents of the IR are then used by the CPU's control unit to decode the instruction and determine the appropriate actions to be taken. The decoding process involves extracting the opcode and any additional operands or addressing modes specified by the instruction. The IR typically has a fixed size that matches the length of instructions supported by the CPU architecture. It may consist of multiple fields

or bit positions that represent different parts of the instruction, such as the opcode, register identifiers, memory addresses, or immediate values. Once the instruction is decoded, the control unit uses the information from the IR to direct the execution of the instruction, which may involve accessing registers, performing arithmetic or logical operations, or accessing memory locations. The contents of the IR can change during the instruction cycle as the CPU progresses through the different stages of instruction execution.

2.4 Register File(RF)

The Register File (RF) is a component of a computer's CPU (Central Processing Unit) that stores a set of registers used for data storage and manipulation. It is a high-speed memory unit that provides fast access to the CPU's general-purpose registers. The RF typically consists of a collection of registers, each capable of holding a fixed-size binary value. The number of registers in the RF varies depending on the CPU architecture but is usually in the range of 8 to 32 registers or more. Each register is identified by a unique index or name. The purpose of the RF is to provide temporary storage space for data during program execution. It allows the CPU to quickly access and manipulate data without having to access the slower main memory. The registers in the RF are directly accessible by the CPU's arithmetic and logic unit (ALU) for performing arithmetic operations, logical operations, and data movement. The RF is often organized as a bank of registers, with each register having a specific purpose or role. Common types of registers found in the RF include general-purpose registers, which can store any type of data, and special-purpose registers, which have specific functions such as holding program counters, stack pointers, or status flags. RF is an essential component of the CPU, and its efficient utilization can greatly impact the performance of the computer system. By keeping frequently used data in registers, the CPU can minimize the need for memory accesses, improving the overall execution speed.

2.5 ALU (Arithmetic Logic Unit)

The Arithmetic Logic Unit (ALU) is a fundamental component of a computer's central processing unit (CPU). It is responsible for performing arithmetic operations (such as addition, subtraction, multiplication, and division) and logical operations (such as AND, OR, NOT) on binary data. The ALU takes input from the CPU's registers and performs the requested operations based on control signals. It operates on binary data, which is represented in the form of bits (0s and 1s). The ALU can perform arithmetic operations on integer values, bitwise operations on binary data, and logical operations to evaluate conditions and make decisions. The ALU consists of various subcomponents, including arithmetic circuits, logic gates, multiplexers, and control units. These components work together to execute the desired arithmetic or logical operation. The ALU is designed to process data in parallel, allowing for efficient and high-speed computations. The ALU is an integral part of the CPU's instruction execution process. It performs calculations and manipulations on data according to the instructions fetched from memory. The result of an ALU operation is typically stored in registers or sent to other components for further processing. In summary, the ALU is responsible for performing arithmetic and logical operations on binary data within a computer's CPU. It plays a critical role in executing instructions and carrying out computations necessary for the functioning of a computer system.

2.6 Data Memory(DM)

Data Memory (DM), also known as the main memory or RAM (Random Access Memory), is a component of a computer system where data is stored and retrieved during program execution. It is separate from the CPU and is used to hold both program instructions and data that are actively being processed by the CPU. The Data Memory is organized into addressable storage locations, each capable of holding a fixed number of bits or bytes of data. These storage locations are typically accessed using memory addresses, which are unique identifiers for each location. The Data Memory plays a crucial role in the execution of programs. It holds variables, arrays, structures, and other data types that are used by the CPU during program execution. When the CPU needs to read or write data, it sends the appropriate memory address to the Data Memory, and the requested data is transferred between the memory and the CPU registers. The size of the Data Memory determines the amount of data that can be stored and accessed by the CPU. It is typically measured in bytes or kilobytes, and the capacity can vary depending on the specific computer system. In modern computer architectures, Data Memory is usually implemented as dynamic random-access memory (DRAM) or static random-access memory (SRAM). DRAM provides higher storage capacity but slower access times, while SRAM offers faster access times but lower storage density. Overall, the Data Memory serves as a temporary storage space for data that is actively being processed by the CPU. It is an essential component of a computer system, enabling the manipulation and retrieval of data during program execution.

2.7 Control Unit(CU)

The Control Unit (CU) is a critical component of a computer's central processing unit (CPU). It is responsible for coordinating and controlling the operations of the CPU, ensuring that instructions are executed in the correct sequence and that data is processed and transferred accurately. The Control Unit performs several important functions: Instruction Decoding: The CU receives instructions from the memory and decodes them to determine the specific operations to be performed. It interprets the instruction's opcode (operation code) and identifies the required data operands. Instruction Sequencing: The CU ensures that instructions are executed in the correct order. It controls the flow of instructions, determining which instruction should be executed next based on the program counter and the current instruction being executed. Operand Fetching: The CU retrieves the data operands required by the instructions from the memory or registers. It fetches data from the appropriate memory locations or registers and makes it available for processing by the CPU. Execution Control: The CU controls the execution of arithmetic and logical operations within the CPU. It activates the appropriate functional units, such as the Arithmetic Logic Unit (ALU), to perform the required computations. Register Transfer: The CU manages the transfer of data between registers, memory, and other CPU components. It ensures that data is moved to the correct locations and that intermediate results are stored appropriately. Branch and Jump Handling: The CU handles branching and jumping instructions, determining the correct address to branch or jump to based on the conditions specified in the instructions. It updates the program counter accordingly. The Control Unit uses a combination of logic circuits, microcode, and control signals to carry out its functions. It interacts closely with other components of the CPU, such as the ALU, registers, and memory, to execute instructions and process data. Overall, the Control Unit plays a vital role in the execution of instructions and the control of data flow within the CPU. It ensures that instructions are properly decoded, executed, and coordinated, allowing for the orderly execution of programs and the processing of data.

2.8 Multiplexers(MUXes)

Multiplexers, often referred to as MUXes, are digital electronic devices that are used to select and route one of several input signals to a single output. They operate based on the control inputs that determine which input signal is passed through to the output. Multiplexers are commonly used in various digital systems and circuits to enable efficient data routing and selection. The primary function of a multiplexer is to combine multiple input signals into a single output signal based on the control inputs. A multiplexer has two main components: the data inputs and the control inputs. The number of data inputs depends on the specific multiplexer and its configuration. The control inputs determine which data input is selected and routed to the output. The control inputs are usually in binary format, and the number of control inputs determines the number of selectable inputs. For example, a 2-to-1 multiplexer has one control input, which can be either 0 or 1, and two data inputs. The control input selects one of the data inputs to be passed through to the output. Multiplexers can be implemented using various logic gates, such as AND gates, OR gates, and NOT gates. The selection of the input signal is based on the combination of logic levels applied to the control inputs. The output of a multiplexer follows the logic level of the selected input. Multiplexers have several applications in digital systems. They are often used in data routing, data selection, and signal switching. For example, multiplexers are commonly used in communication systems to select different data streams for transmission over a shared channel. They are also used in memory systems to enable the selection of specific memory locations. In summary, multiplexers are versatile digital devices used to route and select one of several input signals to a single output. They provide a compact and efficient way to manage data selection and routing in digital systems

2.9 Sign Extender

A sign extender, also known as a sign extension unit, is a component in computer systems that extends the sign bit of a binary number to fill additional bits. It is commonly used in arithmetic and logical operations, particularly when dealing with signed numbers. In computer systems, numbers are represented using binary notation, where the leftmost bit, also known as the sign bit, indicates the sign of the number. A sign bit of 0 represents a positive number, while a sign bit of 1 represents a negative number. The remaining bits represent the magnitude of the number. When performing operations that involve signed numbers, it is important to maintain the correct sign throughout the computation. This is where the sign extender comes into play. It takes the sign bit from the original binary number and extends it to fill additional bits, ensuring that the sign is preserved. The sign extender works by copying the sign bit and replicating it across the additional bits. For example, if the original binary number has a sign bit of 1, indicating a negative number, the sign extender will fill the additional bits with 1s to maintain the negative sign. The purpose of the sign extender is to ensure consistency and compatibility when performing arithmetic and logical operations on signed numbers. By extending the sign bit, the sign extender ensures that the sign is propagated correctly throughout the computation, allowing for accurate and reliable results. In computer architecture, the sign extender is typically part of the datapath or arithmetic logic unit (ALU), where it is used in conjunction with other components to perform signed arithmetic and logical operations. In summary, a sign extender is a component in computer systems that extends the sign bit of a binary number to fill additional bits, ensuring the correct sign is maintained during arithmetic and logical operations involving signed numbers.

2.10 Shifters

Shifters are digital circuits used in computer systems to shift the bits of a binary number to the left or right. They are fundamental components in various operations, including data manipulation, multiplication, division, and logical operations. A shifter typically consists of a series of flip-flops or multiplexers that allow for the movement of bits within a binary number. It takes an input binary number and shifts its bits by a specified number of positions, either to the left (left shift) or to the right (right shift). The two main types of shifters are arithmetic shifters and logical shifters. Arithmetic Shifters: These shifters preserve the sign of the number being shifted. In a left arithmetic shift, the bits are shifted to the left, and the vacant bit on the right is filled with a zero. In a right arithmetic shift, the bits are shifted to the right, and the vacant bit on the left is filled with the sign bit to preserve the sign of the number. Logical Shifters: These shifters do not consider the sign and treat the number as an unsigned binary value. In a left logical shift, the bits are shifted to the left, and the vacant bit on the right is filled with a zero. In a right logical shift, the bits are shifted to the right, and the vacant bit on the left is filled with a zero. Shifters can perform single-bit shifts or multi-bit shifts, depending on the requirements of the operation. They are widely used in various applications, such as data encoding and decoding, bitwise operations, data compression, and encryption algorithms. In computer architecture, shifters are often integrated into the arithmetic logic unit (ALU) or the datapath of a processor. They provide essential functionality for manipulating data and performing mathematical and logical operations on binary numbers. In summary, shifters are digital circuits used in computer systems to shift the bits of a binary number to the left or right. They can perform arithmetic or logical shifts and are integral to operations involving data manipulation and binary arithmetic.

3 Verilog Code

Following is the source code for Project

3.1 ALU Code

```
//********** ALU MODULES **********
// 16-bit MIPS ALU in Verilog using modified template of 4-bit ALU
module ALU (op,a,b,result,zero);
   input [15:0] a; //16 bit inputs a,b
   input [15:0] b;
   input [2:0] op; //3 bit alu opp code
   output [15:0] result; // produce 16 bit output
   output zero;
   wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16;

   //cascade 16 single bit alu's
   ALU1  alu0 (a[0], b[0], op[2], op[1:0],set,op[2],c1,result[0]);
   ALU1  alu1 (a[1], b[1], op[2], op[1:0],0, c1, c2,result[1]);
   ALU1  alu2 (a[2], b[2], op[2], op[1:0],0, c2, c3,result[2]);
   ALU1  alu3 (a[3], b[3], op[2], op[1:0],0, c3, c4,result[3]);
   ALU1  alu4 (a[4], b[4], op[2], op[1:0],0, c4, c5,result[4]);
   ALU1  alu5 (a[5], b[5], op[2], op[1:0],0, c5, c6,result[5]);
   ALU1  alu6 (a[6], b[6], op[2], op[1:0],0, c6, c7,result[6]);
   ALU1  alu7 (a[7], b[7], op[2], op[1:0],0, c7, c8,result[7]);
   ALU1  alu8 (a[8], b[8], op[2], op[1:0],0, c8, c9,result[8]);
   ALU1  alu9 (a[9], b[9], op[2], op[1:0],0, c9, c10,result[9]);
   ALU1  alu10 (a[10],b[10],op[2], op[1:0],0, c10, c11,result[10]);
   ALU1  alu11 (a[11],b[11],op[2], op[1:0],0, c11, c12,result[11]);
   ALU1  alu12 (a[12],b[12],op[2], op[1:0],0, c12, c13,result[12]);
   ALU1  alu13 (a[13],b[13],op[2], op[1:0],0, c13, c14,result[13]);
   ALU1  alu14 (a[14],b[14],op[2], op[1:0],0, c14, c15,result[14]);
   ALUmsb alu15 (a[15],b[15],op[2], op[1:0],0, c15, c16,result[15],set);

   or or1(or01, result[0],result[1]);
   or or2(or23, result[2],result[3]);
   nor nor1(zero,or01,or23);
endmodule
```

Explaination

The given code represents a 16-bit MIPS ALU (Arithmetic Logic Unit) module in Verilog. The ALU performs various arithmetic and logical operations on two 16-bit inputs a and b based on the 3-bit operation code op. The module outputs a 16-bit result and a zero signal. Here's a summary of the code: The module declaration begins with the line module ALU (op,a,b,result,zero);, specifying the input and output ports of the module. The inputs are defined as follows: a and b are 16-bit input signals. op is a 3-bit input signal representing the ALU operation code. The outputs are defined as follows: result is a 16-bit output signal representing the ALU result. zero is a single-bit output signal indicating whether the result is zero. The ALU module instantiates multiple instances of a 1-bit ALU component (ALU1) and one instance of a 1-bit most significant bit ALU component (ALUmsb) to perform the

operations on each bit of the input operands. Each ALU1 instance represents a 1-bit ALU, which takes two input bits (a[n] and b[n]), the ALU operation code (op[2], op[1:0]), and a carry-in (c[n-1]) as inputs. It produces a 1-bit output result (result[n]) and a carry-out (c[n]). The ALU1 instances are cascaded together, with the carry-out from each instance connected to the carry-in of the next instance. The output results of each ALU1 instance are connected to the corresponding bits of the result signal. The ALUmsb instance represents the most significant bit ALU, which is similar to ALU1 but also takes an additional set signal as input. The set signal determines whether the carry-in (c15) should be set to 1 or 0. The result[15] and set signals are connected to the corresponding bits of the result signal. Two or gates (or1 and or2) are used to combine adjacent pairs of bits from the result signal. A nor gate (nor1) is used to combine the outputs of the or gates and produce the zero signal. Overall, the module instantiates and connects multiple 1-bit ALUs to create a 16-bit ALU capable of performing various arithmetic and logical operations on 16-bit inputs. The result and zero signals are generated based on the specified operation code and input values.

3.2 ALU1 Code

```
// 1-bit ALU for bits 0-2
module ALU1 (a,b,binvert,op,less,carryin,carryout,result);
   input a,b,less,carryin,binvert;
   input [1:0] op;
   output carryout,result;
   wire sum, a_and_b, a_or_b, b_inv;

   not not1(b_inv, b);
   mux2x1 mux1(b,b_inv,binvert,b1);
   and and1(a_and_b, a, b);
   or or1(a_or_b, a, b);
   fulladder adder1(sum,carryout,a,b1,carryin);
   mux4x1 mux2(a_and_b,a_or_b,sum,less,op[1:0],result);
endmodule
```

Explaination

The module ALU1 represents a 1-bit Arithmetic Logic Unit (ALU) in Verilog. It performs arithmetic and logical operations on two input bits (a and b) based on the control signals (op and binvert). Here's a breakdown of the module: Inputs: a, b: Input bits to the ALU. binvert: Control signal indicating whether to invert the second input b. op: 2-bit control signal specifying the operation to be performed. less: Control signal indicating whether to perform a "less than" comparison. carryin: Carry-in input for the addition operation. Outputs: carryout: Carry-out output from the addition operation. result: Result of the ALU operation. Internal Wires: sum: Sum output from the full adder. a_and_b: Bitwise AND operation between a and b. a_or_b: Bitwise OR operation between a and b. b_inv: Inverted value of b. b1: Output of the 2-to-1 multiplexer (mux1) that selects between b and b_inv. The module consists of the following components: not1: Inverts the value of b to obtain b_inv. mux1: 2-to-1 multiplexer that selects between b and b_inv based on the binvert control signal. and1: Performs a bitwise AND operation between a and b. or1: Performs a bitwise OR operation between a and b. fulladder: Adds a, b1, and carryin inputs to produce sum and carryout. mux2: 4-to-1 multiplexer that selects the output based on the control signals op[1:0], less, and the values of a_and_b, a_or_b, and sum. The selected output is assigned to result. The ALU1 module is designed to perform

arithmetic operations (addition), logical operations (AND, OR), and comparison (less than) based on the control signals and input values.

3.3 ALUmsb Code

```
// 1-bit ALU for the most significant bit
module ALUmsb (a,b,binvert,op,less,carryin,carryout,result,sum);
   input a,b,less,carryin,binvert;
   input [1:0] op;
   output carryout,result,sum;
   wire sum, a_and_b, a_or_b, b_inv;

   not not1(b_inv, b);
   mux2x1 mux1(b,b_inv,binvert,b1);
   and and1(a_and_b, a, b);
   or or1(a_or_b, a, b);
   fulladder adder1(sum,carryout,a,b1,carryin);
   mux4x1 mux2(a_and_b,a_or_b,sum,less,op[1:0],result);
endmodule
```

Explaination:
The ALUmsb module represents the Most Significant Bit (MSB) of the Arithmetic Logic Unit (ALU) in Verilog. It is responsible for performing arithmetic and logical operations on the most significant bits of the input operands. Here's an explanation of the module: Inputs: a, b: Input bits representing the most significant bits of the operands. binvert: Control signal indicating whether to invert the second input b. op: 2-bit control signal specifying the operation to be performed. less: Control signal indicating whether to perform a "less than" comparison. carryin: Carry-in input for the addition operation. Outputs: carryout: Carry-out output from the addition operation. result: Result of the ALU operation. sum: Sum output from the addition operation. Internal Wires: a_and_b: Bitwise AND operation between a and b. a_or_b: Bitwise OR operation between a and b. b_inv: Inverted value of b. b1: Output of the 2-to-1 multiplexer (mux1) that selects between b and b_inv. The module consists of similar components to the previous ALU1 module, with the addition of the sum output: not1: Inverts the value of b to obtain b_inv. mux1: 2-to-1 multiplexer that selects between b and b_inv based on the binvert control signal. and1: Performs a bitwise AND operation between a and b. or1: Performs a bitwise OR operation between a and b. fulladder: Adds a, b1, and carryin inputs to produce sum and carryout. mux2: 4-to-1 multiplexer that selects the output based on the control signals op[1:0], less, and the values of a_and_b, a_or_b, and sum. The selected output is assigned to result. The ALUmsb module is specifically designed to handle the MSB of the ALU operation and generate the appropriate carry-out, result, and sum outputs based on the control signals and input values.

Purpose of Multiple small ALUs
Multiple small ALUs are used to build this single ALU in order to perform arithmetic and logical operations on each bit of the input operands simultaneously. This approach is known as bit-slice implementation. By breaking down the ALU into smaller ALU components (1-bit ALUs in this case), each component can independently perform operations on its respective bit of the input operands. This allows for parallel processing and efficient computation of the ALU result.

3.4 Branch Code

```
module BranchControl (BranchOp,Zero,BranchOut);

  input [1:0] BranchOp; // two bits to handle bne/beq
  input Zero;
  output BranchOut;
  wire ZeroInvert,i0,i1;

  not not1(ZeroInvert,Zero);
  and and1(i0,BranchOp[0],Zero);
  and and2(i1,BranchOp[1],ZeroInvert);
  or or1(BranchOut,i0,i1);
endmodule
```

3.5 CPU Code

```
//********** CPU **********
// CPU template modified from "mips-pipe3.vl"
module CPU (clock,PC,IFID_IR,IDEX_IR,EXMEM_IR,MEMWB_IR,WD);

  input clock;
  output [15:0] PC,IFID_IR,IDEX_IR,EXMEM_IR,MEMWB_IR,WD;
 reg  [15:0] PC, IMemory[0:1023], IFID_IR,IFID_PCplus2;
   reg [15:0] DMemory[0:1023],MEMWB_MemOut,MEMWB_ALUOut;
  initial begin
      //                          //from template //changed to work with our 4 regs
          //instruction format

    IMemory[0] = 16'b0001000100100000; // lw $8, 0($0)  -> lw $1, 0($0)  0101 00 01
       00000000
    IMemory[1] = 16'b0010000110010010; // lw $9, 4($0)  -> lw $2, 2($0)  0101 00 10
       00000010
    IMemory[2] = 16'b0011001010011100; // nop
    IMemory[3] = 16'b0000000000000000; // nop
    IMemory[4] = 16'b0000100111000000; // nop
   IMemory[5] = 16'b0000000000000000; // slt $10, $9, $8 -> slt $3, $2, $1 0111 10
      01 11 000000
   IMemory[6] = 16'b0100001101011001; // nop
   IMemory[7] = 16'b0000000000000000; // nop
   IMemory[8] = 16'b0000100111000000; // nop
   IMemory[9] = 16'b0000000000000000; // bne$10, $0, 28 -> bne $3, $0, 14 1001 11
      00 00000100
   IMemory[10] = 16'b0000000000000000; // nop
   IMemory[11] = 16'b0100001101011001; // nop
   IMemory[12] = 16'b0000000000000000; // nop
   IMemory[13] = 16'b0001011001000000; // sub $11, $11, $12 -> sub $1, $1, $2 0001
      01 10 01 000000
```

```
 IMemory[14] = 16'b0000000000000000; // nop
 IMemory[15] = 16'b0100001101011001; // nop
 IMemory[16] = 16'b0110010101011010; // nop
 IMemory[17] = 16'b0000100111000000; // add $10, $12, $11 -> add $3, $2, $1 0000
    10 01 11 000000 // should be branch target

  // Data
  DMemory [0] = 16'h5; // switch the cells and see how the simulation output
     changes
  DMemory [1] = 16'h7;
end

// Pipeline stages

//IF
wire [15:0] NextPC,PCplus2;

reg [1:0] EXMEM_Branch;
reg MEMWB_RegWrite,MEMWB_MemtoReg;
reg [15:0] EXMEM_Target,EXMEM_ALUOut,EXMEM_RD2;
reg EXMEM_Zero;
reg [1:0] MEMWB_rd;
ALU fetch (3'b010,PC,2,PCplus2,Unused); //play with timings here
BranchControl BranchCtrl (EXMEM_Branch,EXMEM_Zero,BranchConOut); // added branch
    control
mux2x1_16bit BranchMux (PCplus2,EXMEM_Target,BranchConOut,NextPC); // added mux
    for branch

//ID
reg  [15:0] IDEX_IR; // For monitoring the pipeline
wire [9:0] Control;
reg IDEX_RegWrite,IDEX_ALUSrc,IDEX_RegDst,IDEX_MemtoReg,IDEX_MemWrite;
reg  [1:0] IDEX_Branch; // because our bne and bqe are 2 bits
reg  [2:0] IDEX_ALUOp; // our aluOps are 3 bits
wire [15:0] RD1,RD2,SignExtend, WD;
reg  [15:0] IDEX_RD1,IDEX_RD2,IDEX_SignExt,IDEX_PCplus2,IDEXE_IR; // added
    IDEX_PCplus2,IDEXE_IR
reg  [1:0]  IDEX_rt,IDEX_rd; //should be 2 bit
reg_file rf
    (IFID_IR[11:10],IFID_IR[9:8],MEMWB_rd,WD,MEMWB_RegWrite,RD1,RD2,clock); //
    added MEMWB_rd & MEMWB_RegWrite
MainControl MainCtr (IFID_IR[15:12],Control);
assign SignExtend = {{8{IFID_IR[7]}},IFID_IR[7:0]};

//EXE
reg EXMEM_RegWrite,EXMEM_MemtoReg,EXMEM_MemWrite;
wire [15:0] Target;


reg [15:0] EXMEM_IR; // this is for monitoring the pipeline
reg [1:0] EXMEM_rd;
```

```
wire [15:0] B,ALUOut;
wire [1:0] WR;
ALU branch (3'b010,IDEX_SignExt<<1,IDEX_PCplus2,Target,Unused2);
ALU ex (IDEX_ALUOp, IDEX_RD1, B, ALUOut, Zero);
mux2bit2x1 RegDstMux (IDEX_rt, IDEX_rd, IDEX_RegDst, WR); // assign WR, Reg Dst
    mux
mux2x1_16bit ALUSrcMux (IDEX_RD2, IDEX_SignExt, IDEX_ALUSrc, B); // assign B, ALU
    src mux


//MEM


reg [15:0] MEMWB_IR; // For monitoring the pipeline
wire [15:0] MemOut;

assign MemOut = DMemory[EXMEM_ALUOut>>1];
always @(negedge clock) if (EXMEM_MemWrite) DMemory[EXMEM_ALUOut>>1] <= EXMEM_RD2;

//WB <--- this area is what I believe is causing trouble
//assign WD = ALUOut; //WD simply gets ALU output, FROM PR3
//assign WD = (MEMWB_MemtoReg) ? MEMWB_MemOut: MEMWB_ALUOut; // MemtoReg Mux,
    FROM PR4 TEMPLATE
mux2x1_16bit MemToReg (MEMWB_ALUOut, MEMWB_MemOut, MEMWB_MemtoReg, WD);

initial begin
  PC = 0;
 // Initialize pipeline registers
  IDEX_RegWrite=0;IDEX_MemtoReg=0;IDEX_Branch=0;IDEX_MemWrite=0;IDEX_ALUSrc=0;IDEX_RegDst=0;I
  IFID_IR=0;
  EXMEM_RegWrite=0;EXMEM_MemtoReg=0;EXMEM_Branch=0;EXMEM_MemWrite=0;
  EXMEM_Target=0;
  MEMWB_RegWrite=0;
  MEMWB_MemtoReg=0;
end

//RUNNING THE PIPELINE

always @(negedge clock) begin

  //STAGE 1 - IF
  PC <= NextPC;
 IFID_PCplus2 <= PCplus2;
  IFID_IR <= IMemory[PC>>1];

  //STAGE 2 - ID
  IDEX_IR <= IFID_IR; // For monitoring the pipeline
 // RegDst[1], AluSrc[1], MemtoReg[1], RegWrite[1], MemWrite1[1], BEQ[1], BNE[1],
     AluCtrl[3]
  {IDEX_RegDst,IDEX_ALUSrc,IDEX_MemtoReg,IDEX_RegWrite,IDEX_MemWrite,IDEX_Branch,IDEX_ALUOp}
      <= Control;
```

```
    IDEX_PCplus2 <= IFID_PCplus2;
    IDEX_RD1 <= RD1;
    IDEX_RD2 <= RD2;
    IDEX_SignExt <= SignExtend;
    IDEX_rt <= IFID_IR[9:8];
    IDEX_rd <= IFID_IR[7:6];

    //STAGE 3 - EX
    EXMEM_IR <= IDEX_IR; // For monitoring the pipeline
    EXMEM_RegWrite <= IDEX_RegWrite;
    EXMEM_MemtoReg <= IDEX_MemtoReg;
    EXMEM_Branch  <= IDEX_Branch;
    EXMEM_MemWrite <= IDEX_MemWrite;
    EXMEM_Target <= Target;
    EXMEM_Zero <= Zero;
    EXMEM_ALUOut <= ALUOut;
    EXMEM_RD2 <= IDEX_RD2;
    EXMEM_rd <= WR;

   //STAGE 4 - MEM
   MEMWB_IR <= EXMEM_IR; // For monitoring the pipeline
    MEMWB_RegWrite <= EXMEM_RegWrite;
    MEMWB_MemtoReg <= EXMEM_MemtoReg;
    MEMWB_MemOut <= MemOut;
    MEMWB_ALUOut <= EXMEM_ALUOut;
    MEMWB_rd <= EXMEM_rd;

   //STAGE 5 - WB
   // Register write happens on neg edge of the clock (if MEMWB_RegWrite is
      asserted)

  end
endmodule

//********** END OF CPU MODULE **********
```

Explaination

The provided code represents a simplified CPU module that implements a basic 5 stage pipeline. Lets break down the code and understand its working;

Input and output declarations: The module takes the clock signal as input and provides several outputs, including PC, IFID_IR, IDEX_IR, EXMEM_IR, MEMWB_IR, and WD. These signals represent various stages and data within the pipeline.

Memory Initialization: The initial block initializes the instruction memory (IMemory) and data memory (DMemory) with some sample values. These memories hold the instructions and data that the CPU will execute.

Pipeline stages: The code defines five pipeline stages: IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory Access), and WB (Write Back).

IF Stage: This stage is responsible for fetching the instruction from the instruction memory. The instruction memory is indexed using the program counter (PC), and the fetched instruction is stored in the IFID_IR register. The PC is incremented by 2, and the incremented value

(PCplus2) is stored in the IFID_PCplus2 register.
ID Stage: This stage decodes the instruction fetched in the previous stage. It involves several operations like extracting control signals (Control) based on the opcode, determining register destinations (IDEX_RegDst), ALU sources (IDEX_ALUSrc), memory outputs (IDEX_MemtoReg), etc. The relevant information is stored in various registers for the subsequent stages.
EX Stage: This stage performs the actual computation or execution based on the decoded instruction. It involves operations like ALU operations (ALUOp), register sources (RD1, RD2), sign extension (SignExtend), branch calculation (branch), etc. The computed results are stored in the EXMEM registers.
MEM Stage: This stage deals with memory-related operations. It includes memory write (EXMEM_MemWrite) and read (MemOut) operations. The memory operations are performed on the data memory (DMemory), and the results are stored in the MEMWB registers.
WB Stage: This stage handles the write-back operation. It determines whether the result should be written back to a register (MEMWB_RegWrite) and selects the appropriate data (MEMWB_MemOut or MEMWB_ALUOut) using a multiplexer. The result is stored in the WD register.
Pipeline Register Updates: At the end of each clock cycle (negedge of the clock), the pipeline registers are updated with the values from the previous stages, ensuring the flow of data through the pipeline.
Initial Block: The initial block initializes the PC and various pipeline registers to their initial values.
Simulation: The entire CPU module is designed to simulate the execution of instructions by updating the pipeline registers based on the clock signal. The instructions are fetched, decoded, executed, and their results are stored in the respective pipeline registers.
This code represents a simplified pipeline implementation and does not cover all aspects of a real-world CPU design. It serves as a starting point for understanding the basic concepts of pipelining and executing instructions in a sequential manner.

3.6 D Flip Flop

```
// Components
// all still used
module D_flip_flop(D,CLK,Q);
   input D,CLK;
   output Q;
   wire CLK1, Y;
   not  not1 (CLK1,CLK);
   D_latch D1(D,CLK, Y),
           D2(Y,CLK1,Q);
endmodule

module D_latch(D,C,Q);
   input D,C;
   output Q;
   wire x,y,D1,Q1;
   nand nand1 (x,D, C),
        nand2 (y,D1,C),
        nand3 (Q,x,Q1),
        nand4 (Q1,y,Q);
```

```
   not  not1  (D1,D);
endmodule
```

Explaination

The given code represents the implementation of a D flip-flop and a D latch in Verilog HDL. Let's break down the code and explain how these modules work.

The D_flip_flop module represents a D flip-flop. It has an input D for the data input, CLK for the clock signal, and an output Q for the output. Inside the module, there is a wire CLK1 and Y, which are intermediate signals used for internal connections.

The not gate with the label not1 is used to invert the CLK signal and assign it to CLK1. This is done because the D flip-flop uses the falling edge of the clock to latch the input.

The D_latch module represents a D latch. It has an input D for the data input, C for the latch control signal, and an output Q for the output. Inside the module, there are wires x, y, D1, and Q1, which are intermediate signals used for internal connections.

The nand gates are used to implement the logic of the D latch. nand1 takes inputs D and C and generates the inverted signal x. nand2 takes inputs D1 (which is the inverted form of D) and C and generates the inverted signal y. nand3 takes inputs x, Q1, and generates the output signal Q. nand4 takes inputs Q1, y, and generates the inverted output signal Q.

The not gate with the label not1 is used to invert the D signal and assign it to D1. This is done because the D latch holds the previous input state when the latch control signal is inactive.

In summary, the D_flip_flop module combines a D latch and an additional not gate to implement a D flip-flop. The D_latch module implements a simple D latch using nand gates and a not gate. These modules can be used to build more complex digital circuits and sequential logic systems.

3.7 Decoder Code

```
module decoder (S1,S0,D3,D2,D1,D0);
   input S0,S1;
   output D0,D1,D2,D3;

   not n1 (notS0,S0),
       n2 (notS1,S1);

   and a0 (D0,notS1,notS0),
       a1 (D1,notS1,  S0),
       a2 (D2,  S1,notS0),
       a3 (D3,  S1,  S0);
endmodule
```

3.8 Flip Flop Code

```
module flipFlop16(D,CLK,Q);
  input [15:0] D; // change input and outputs to 16 bits
  input CLK;     // clock pulse
  output [15:0] Q;
```

```
  //cascade single bit dflipflops
  D_flip_flop f0(D[0], CLK, Q[0]),
              f1(D[1], CLK, Q[1]),
              f2(D[2], CLK, Q[2]),
              f3(D[3], CLK, Q[3]),
              f4(D[4], CLK, Q[4]),
              f5(D[5], CLK, Q[5]),
              f6(D[6], CLK, Q[6]),
              f7(D[7], CLK, Q[7]),
              f8(D[8], CLK, Q[8]),
              f9(D[9], CLK, Q[9]),
             f10(D[10], CLK, Q[10]),
             f11(D[11], CLK, Q[11]),
             f12(D[12], CLK, Q[12]),
             f13(D[13], CLK, Q[13]),
             f14(D[14], CLK, Q[14]),
             f15(D[15], CLK, Q[15]);
endmodule
```

3.9 Full Adder Code

```
module fulladder (S,C,x,y,z);
   input x,y,z;
   output S,C;
   wire S1,D1,D2;

   halfadder HA1 (S1,D1,x,y),
             HA2 (S,D2,S1,z);
   or g1(C,D2,D1);
endmodule
```

3.10 Half Adder Code

```
module halfadder (S,C,x,y);
   input x,y;
   output S,C;

   xor (S,x,y);
   and (C,x,y);
endmodule
```

3.11 Main Control Unit Code

```
//********** CONTROL MODULES **********
module MainControl (Op, Control);
    input [3:0] Op;
   // RegDst[1], AluSrc[1], MemtoReg[1], RegWrite[1], MemWrite1[1], BEQ[1], BNE[1],
      AluCtrl[3]
    output reg [9:0] Control; // 10 bits from ^^^

    always @(Op) case (Op)
    4'b0000: Control <= 10'b1001000010; //add
    4'b0001: Control <= 10'b1001000110; //sub
    4'b0010: Control <= 10'b1001000000; //and
    4'b0011: Control <= 10'b1001000001; //or
    4'b0111: Control <= 10'b1001000111; //slt
    4'b0101: Control <= 10'b0111000010; //lw
    4'b0110: Control <= 10'b0100100010; //sw
    4'b1000: Control <= 10'b0000001110; //beq
    4'b1001: Control <= 10'b0000010110; //bne
    4'b0100: Control <= 10'b0101000010; //addi
    endcase
endmodule
```

Explaination:
The case statements inside the case statement handle specific values of "Op" and assign the corresponding binary value to "Control" using the "¡=" operator.
If "Op" is equal to 4'b0000 (binary 0), "Control" is assigned the value 10'b1001000010.
If "Op" is equal to 4'b0001 (binary 1), "Control" is assigned the value 10'b1001000110.
If "Op" is equal to 4'b0010 (binary 2), "Control" is assigned the value 10'b1001000000.
If "Op" is equal to 4'b0011 (binary 3), "Control" is assigned the value 10'b1001000001.
If "Op" is equal to 4'b0111 (binary 7), "Control" is assigned the value 10'b1001000111.
If "Op" is equal to 4'b0101 (binary 5), "Control" is assigned the value 10'b0111000010.
If "Op" is equal to 4'b0110 (binary 6), "Control" is assigned the value 10'b0100100010.
If "Op" is equal to 4'b1000 (binary 8), "Control" is assigned the value 10'b0000001110.
If "Op" is equal to 4'b1001 (binary 9), "Control" is assigned the value 10'b0000010110.
If "Op" is equal to 4'b0100 (binary 4), "Control" is assigned the value 10'b0101000010.
The case statement ends with the endcase keyword.

3.12 MUX Code

```
//used by WR
module mux2bit2x1(A,B,select,OUT);
   input [1:0] A,B;
    input select;
   output [1:0] OUT;
   //cascade 2 2x1 muxs
    mux2x1 mux1(A[0], B[0], select, OUT[0]),
           mux2(A[1], B[1], select, OUT[1]);
endmodule
```

```
//used by WD, B, Branch
module mux2x1_16bit(A, B, select, OUT);
   input [15:0] A,B;
    input select;
   output [15:0] OUT;
   //cascade 16 2x1 mux's
    mux2x1 mux1(A[0], B[0], select, OUT[0]),
           mux2(A[1],  B[1], select, OUT[1]),
           mux3(A[2],  B[2], select, OUT[2]),
           mux4(A[3],  B[3], select, OUT[3]),
           mux5(A[4],  B[4], select, OUT[4]),
           mux6(A[5],  B[5], select, OUT[5]),
           mux7(A[6],  B[6], select, OUT[6]),
           mux8(A[7],  B[7], select, OUT[7]),
           mux9(A[8],  B[8], select, OUT[8]),
           mux10(A[9], B[9], select, OUT[9]),
           mux11(A[10], B[10], select, OUT[10]),
           mux12(A[11], B[11], select, OUT[11]),
           mux13(A[12], B[12], select, OUT[12]),
           mux14(A[13], B[13], select, OUT[13]),
           mux15(A[14], B[14], select, OUT[14]),
           mux16(A[15], B[15], select, OUT[15]);
endmodule
```

```
module mux2x1(A,B,select,OUT);
  input A,B,select;
  output OUT;

  not not1(i0, select);
  and and1(i1, A, i0);
  and and2(i2, B, select);
  or or1(OUT, i1, i2);
endmodule
```

```
//used by alu components and some reg/mem stuff
module mux4x1(i0,i1,i2,i3,select,y);
  input i0,i1,i2,i3;
  input [1:0] select;
  output y;

  mux2x1 mux1(i0, i1, select[0], m1);
  mux2x1 mux2(i2, i3, select[0], m2);
  mux2x1 mux3(m1, m2, select[1], y);
endmodule
```

```
//********** REGFILE MODULES **********
```

```
module mux16bit(i0, i1, i2, i3, select, O);
  input [15:0] i0, i1, i2, i3; //change inputs to 16 bit
  input [1:0] select; //select kept 2 bits
  output [15:0] O; //output changed to 16 bits

  // cascade 4x1 mux's from template
  mux4x1 mux0(i0[0], i1[0], i2[0], i3[0], select, O[0]),
         mux1(i0[1], i1[1], i2[1], i3[1], select, O[1]),
         mux2(i0[2], i1[2], i2[2], i3[2], select, O[2]),
         mux3(i0[3], i1[3], i2[3], i3[3], select, O[3]),
         mux4(i0[4], i1[4], i2[4], i3[4], select, O[4]),
         mux5(i0[5], i1[5], i2[5], i3[5], select, O[5]),
         mux6(i0[6], i1[6], i2[6], i3[6], select, O[6]),
         mux7(i0[7], i1[7], i2[7], i3[7], select, O[7]),
         mux8(i0[8], i1[8], i2[8], i3[8], select, O[8]),
         mux9(i0[9], i1[9], i2[9], i3[9], select, O[9]),
        mux10(i0[10], i1[10], i2[10], i3[10], select, O[10]),
        mux11(i0[11], i1[11], i2[11], i3[11], select, O[11]),
        mux12(i0[12], i1[12], i2[12], i3[12], select, O[12]),
        mux13(i0[13], i1[13], i2[13], i3[13], select, O[13]),
        mux14(i0[14], i1[14], i2[14], i3[14], select, O[14]),
        mux15(i0[15], i1[15], i2[15], i3[15], select, O[15]);
endmodule
```

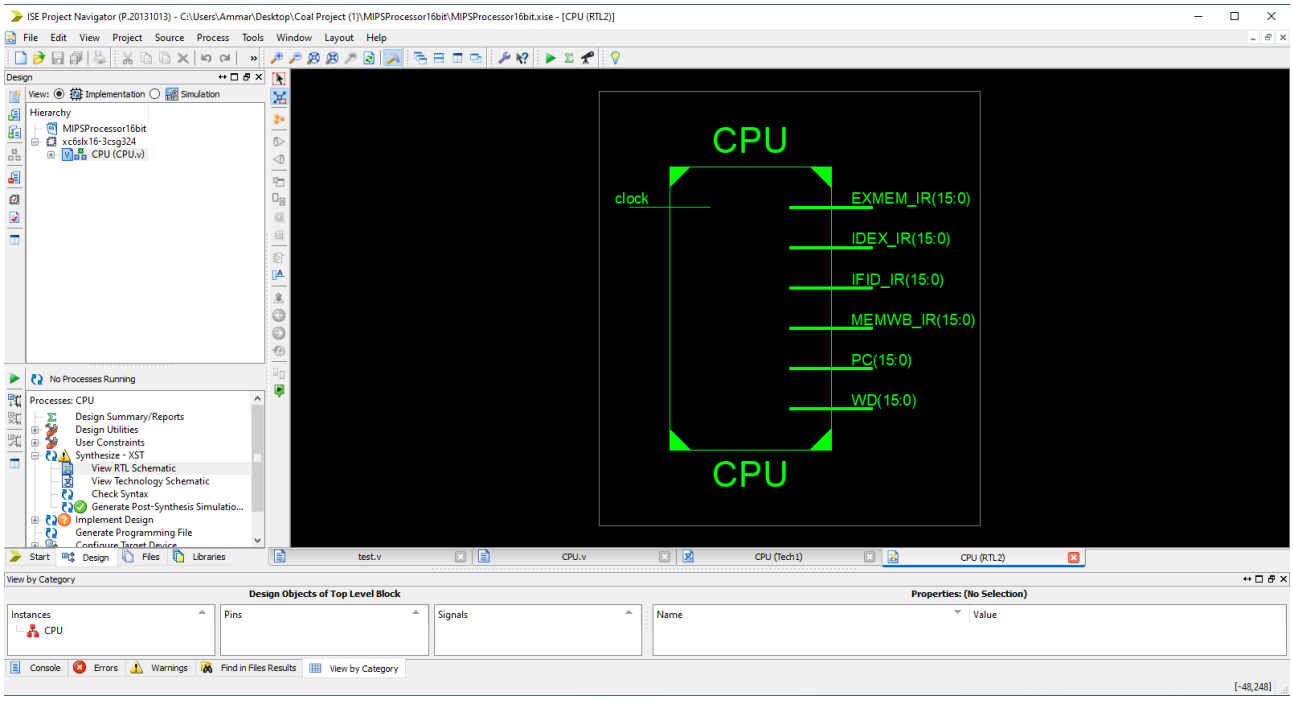
3.13 Test Code

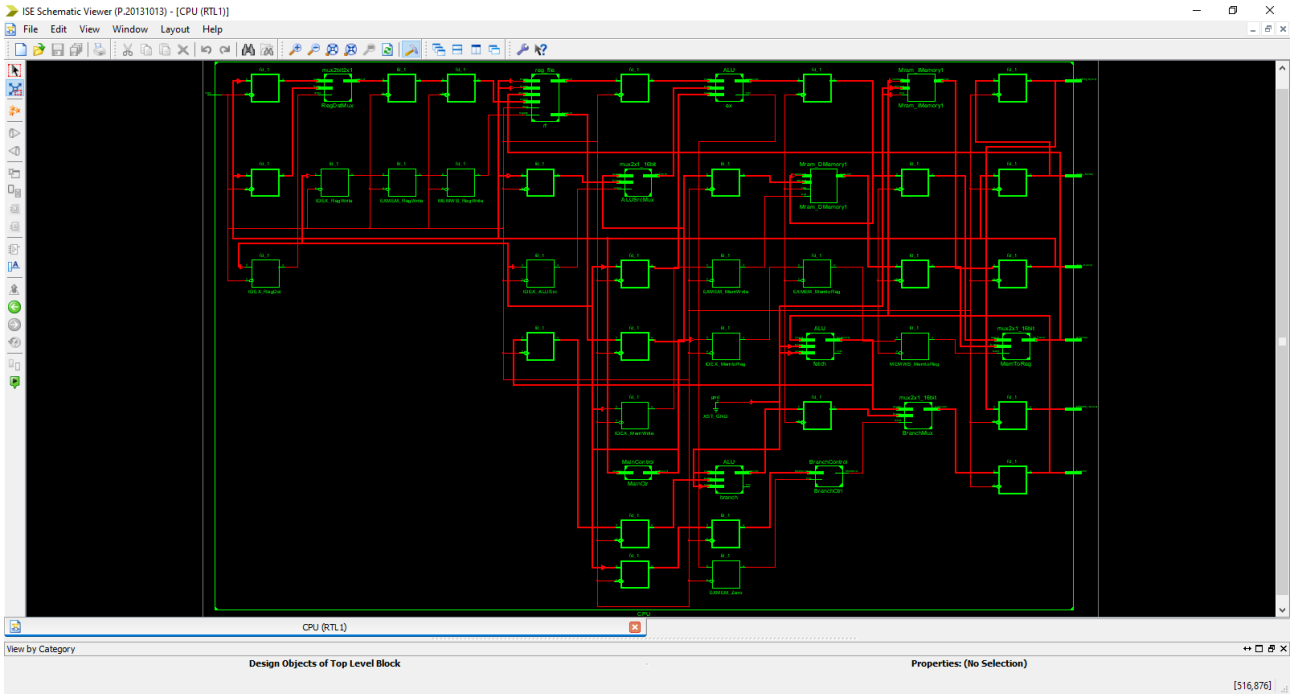
```
//********** TEST MODULE **********
module test ();

  reg clock;
  wire [15:0] PC,IFID_IR,IDEX_IR,EXMEM_IR,MEMWB_IR,WD;

  CPU test_cpu(clock,PC,IFID_IR,IDEX_IR,EXMEM_IR,MEMWB_IR,WD);

  always #1 clock = ~clock;

  initial begin
    $display ("time PC IFID_IR IDEX_IR EXMEM_IR MEMWB_IR WD");
    $monitor ("%2d %3d %h    %h    %h    %h    %h",
        $time,PC,IFID_IR,IDEX_IR,EXMEM_IR,MEMWB_IR,WD);
    clock = 1;
    #1000; //this number will have to be changed for the # of new instructions

  end
endmodule
```

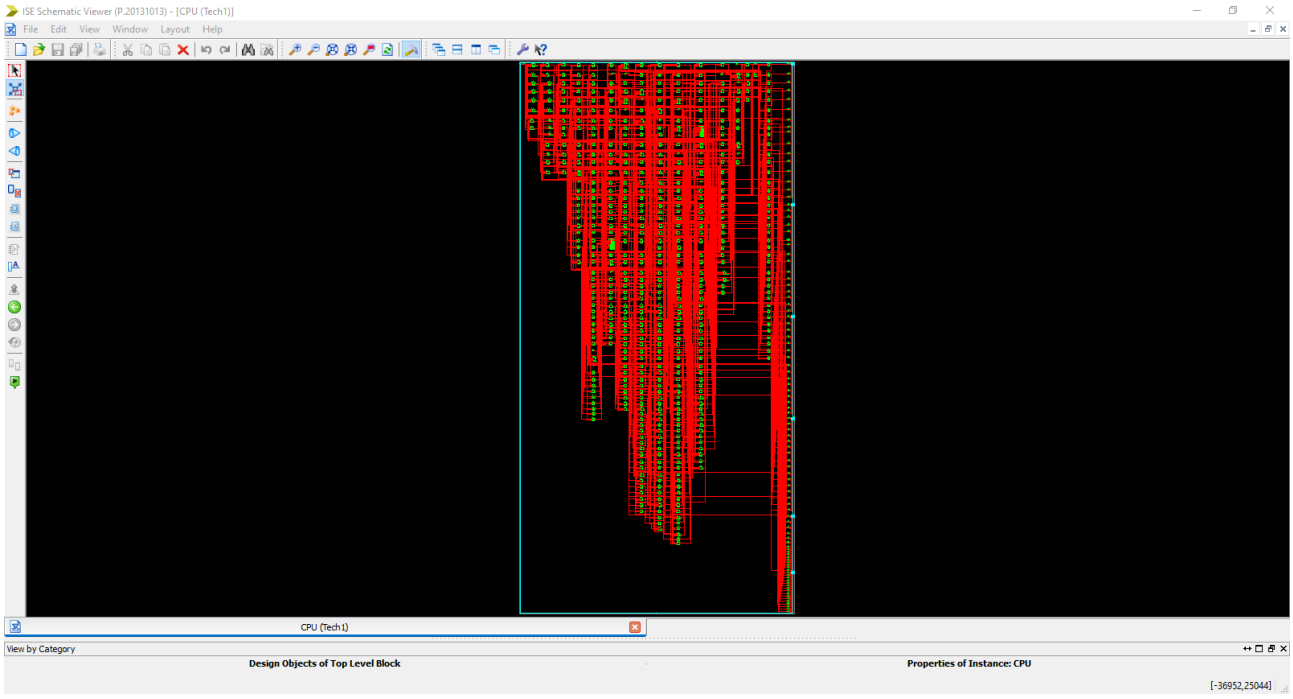
Explaination:
The purpose of the CPU module instance test_cpu is to simulate the behavior of a central

processing unit (CPU) design. It serves as the core component of the testbench and coordinates the execution of the different stages of the CPU pipeline.
The test_cpu instance takes the following inputs:
clock: A clock signal that controls the timing of the CPU operations.
It provides the following outputs:
PC: Represents the value of the program counter, which indicates the address of the current instruction being executed.
IFID_IR: Represents the instruction stored in the instruction fetch/decode stage of the pipeline.
IDEX_IR: Represents the instruction stored in the instruction decode/execution stage of the pipeline.
EXMEM_IR: Represents the instruction stored in the execution/memory stage of the pipeline.
MEMWB_IR: Represents the instruction stored in the memory/writeback stage of the pipeline.
WD: Represents the output of the CPU, which can be the result of an ALU operation or the value loaded from memory. By instantiating and connecting the test_cpu module in the testbench, it becomes possible to observe the values of key signals at each pipeline stage and monitor the execution of the CPU design during the simulation.

4 RTL View:

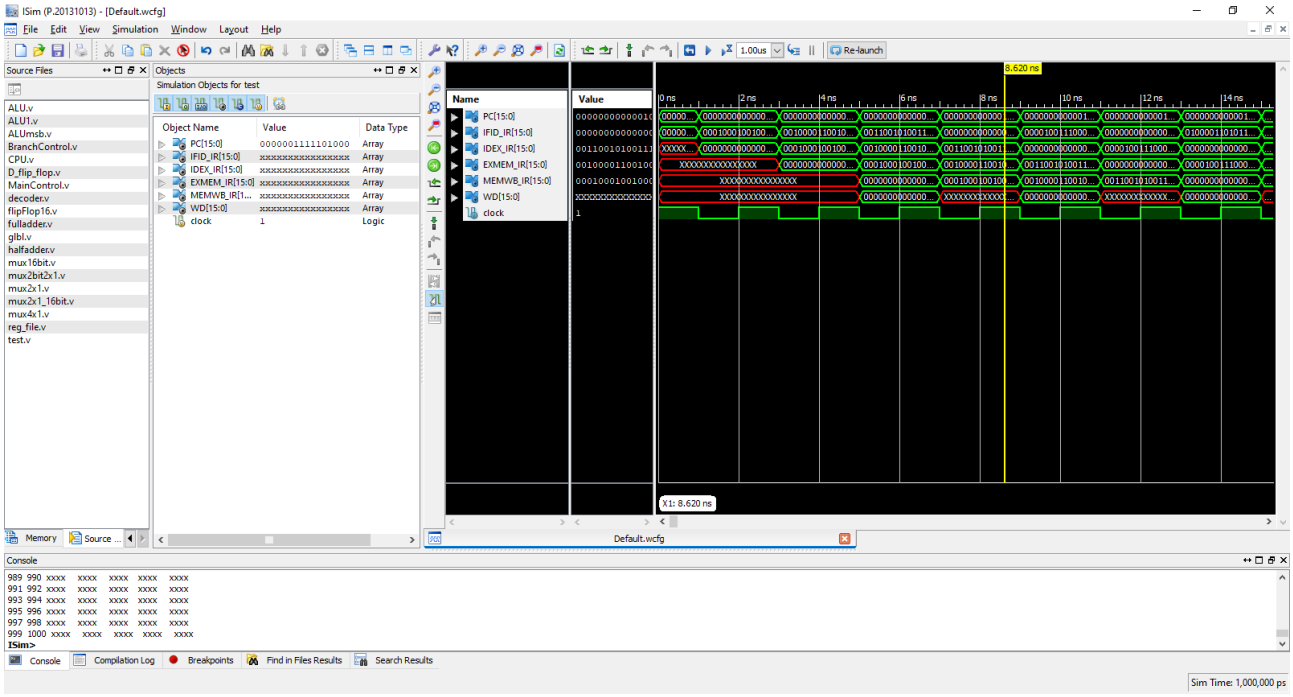




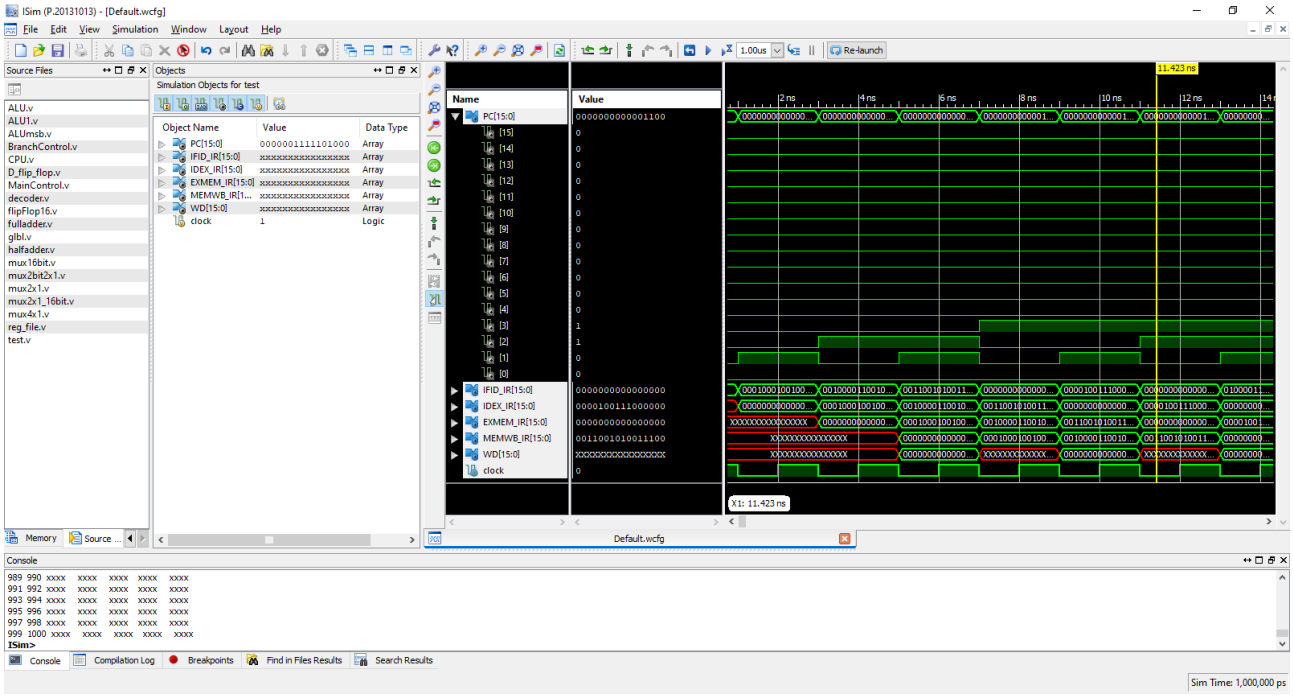


5 OUTPUT:

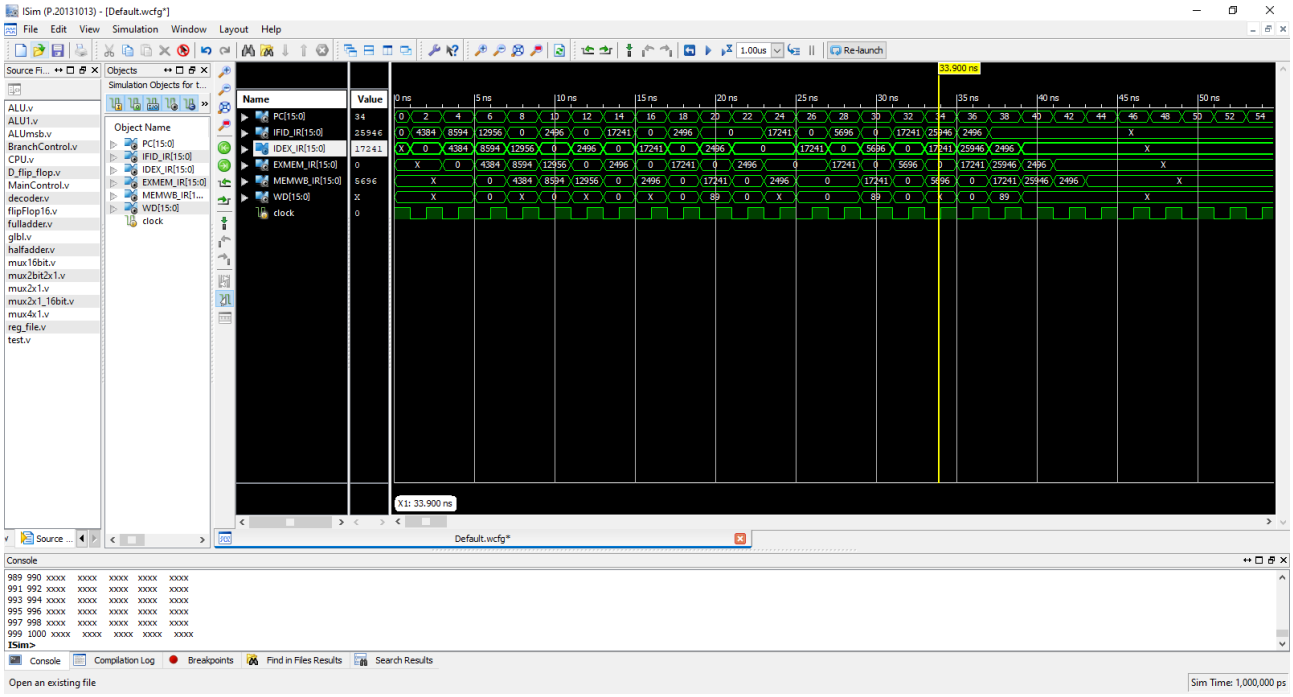
Following are the output Screenshots of Source Code;



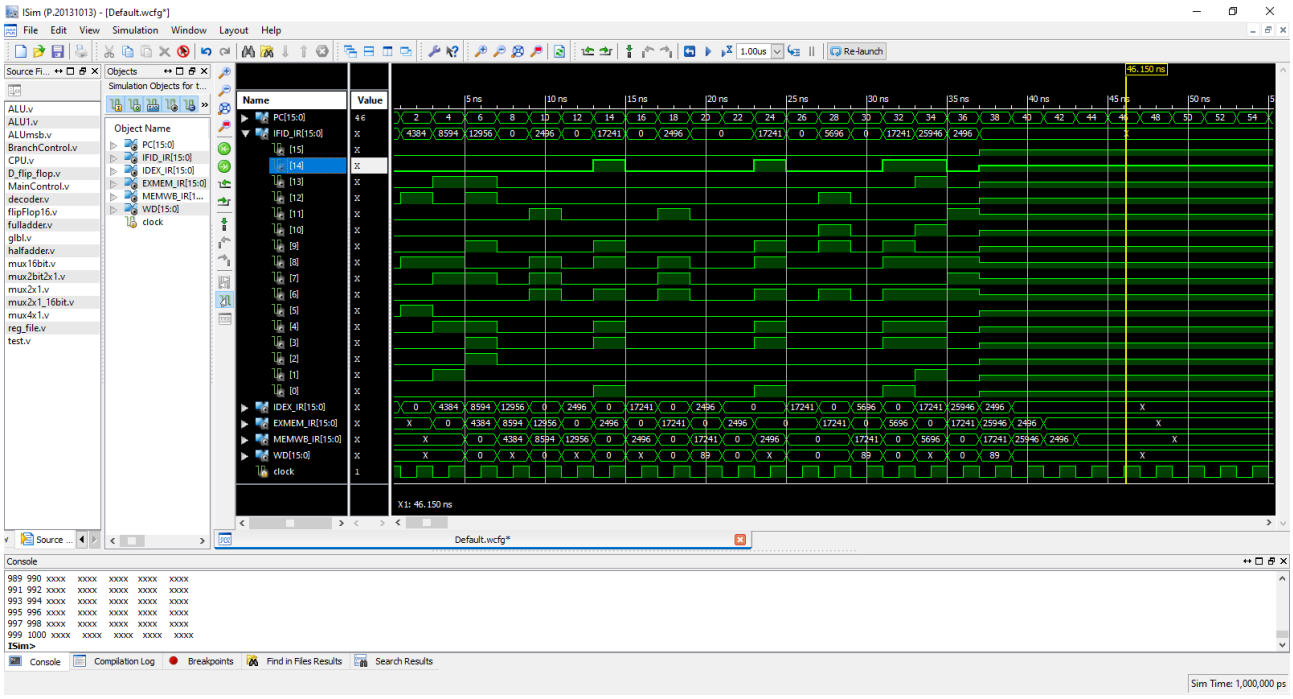
Screenshot 1



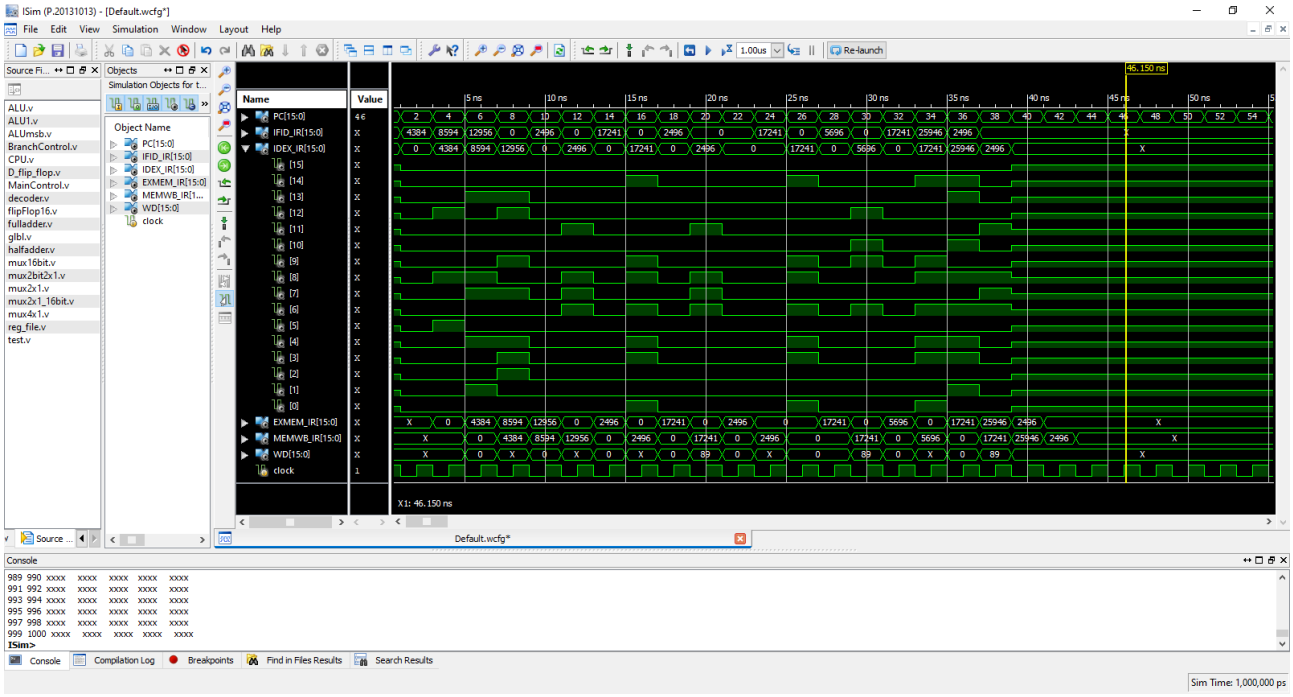
Screenshot 2



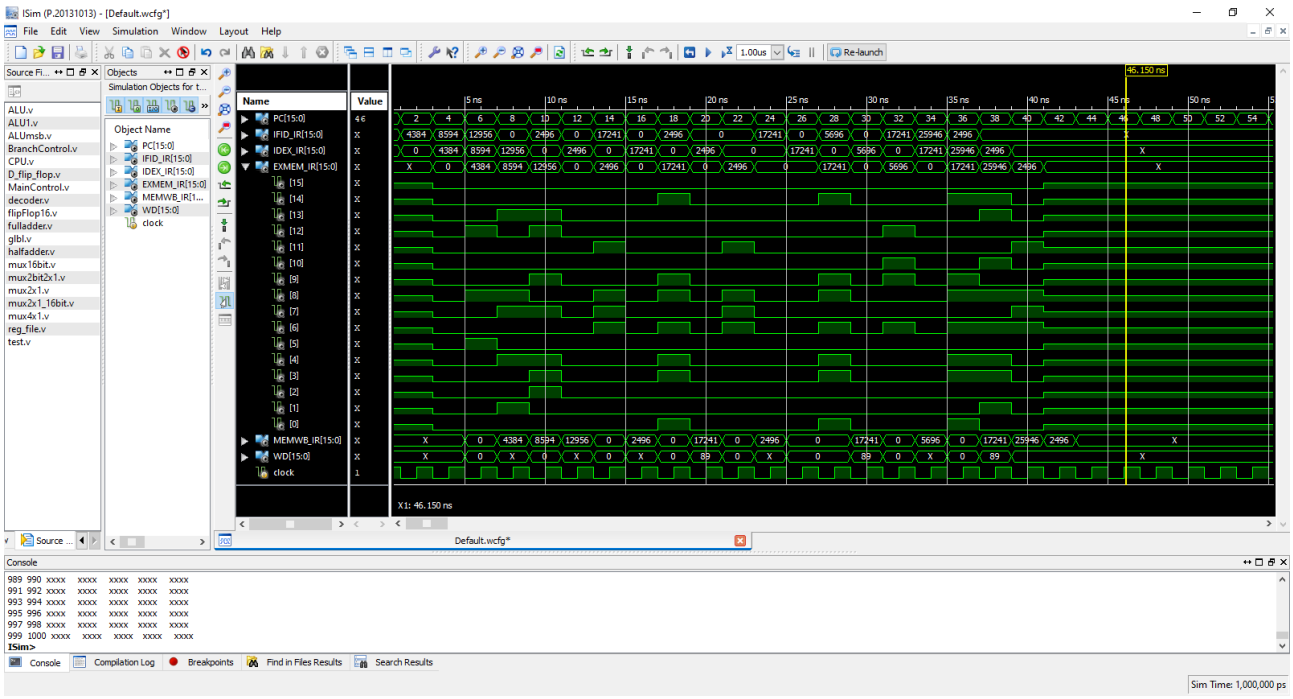
Screenshot 3



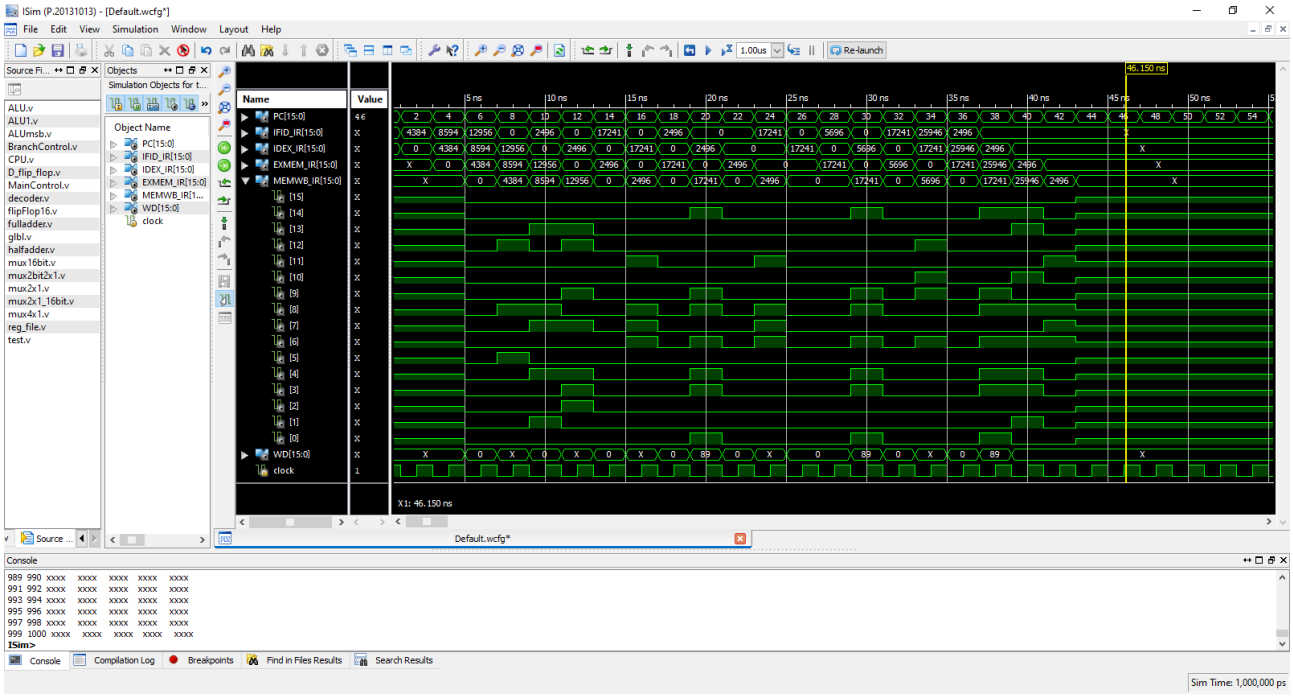
Screenshot 4



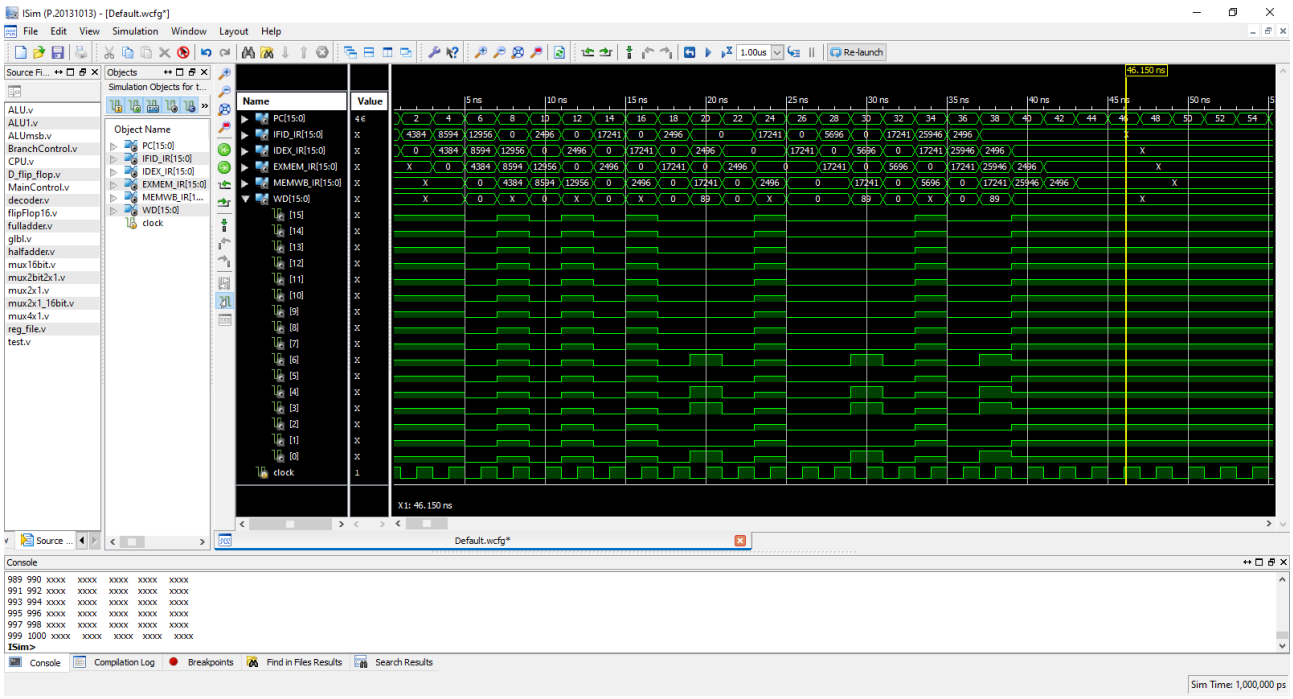
Screenshot 5



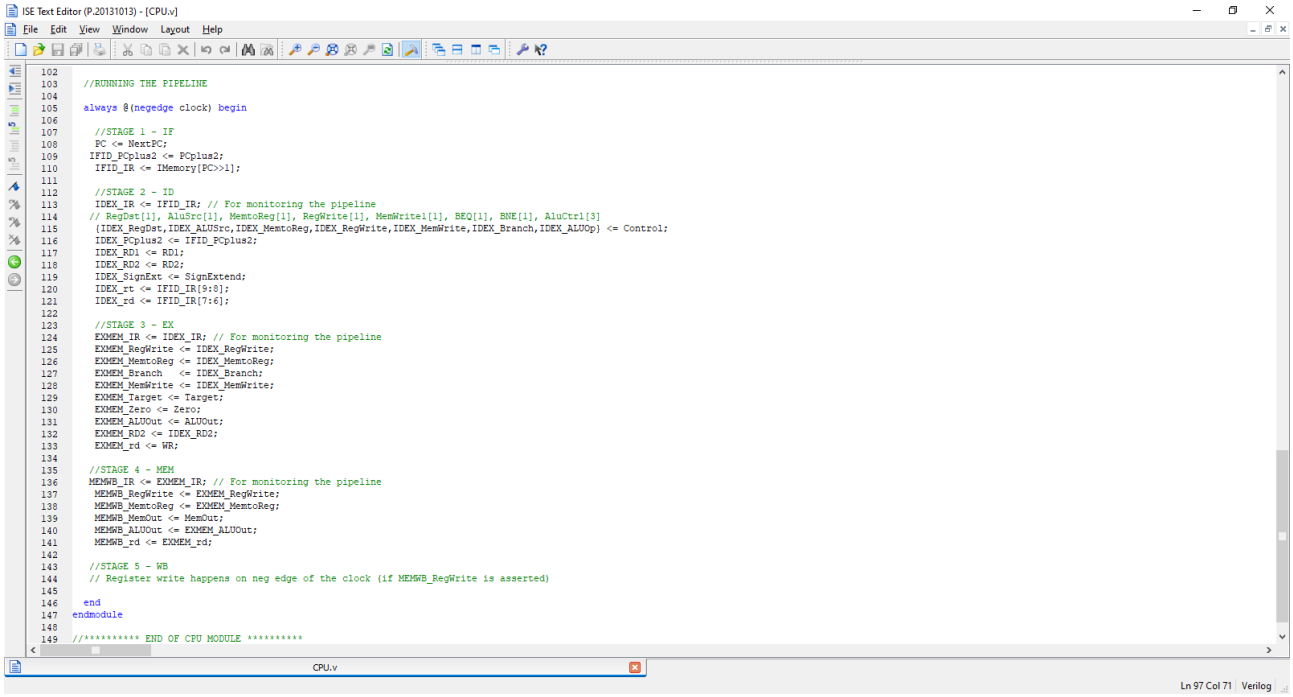
Screenshot 6



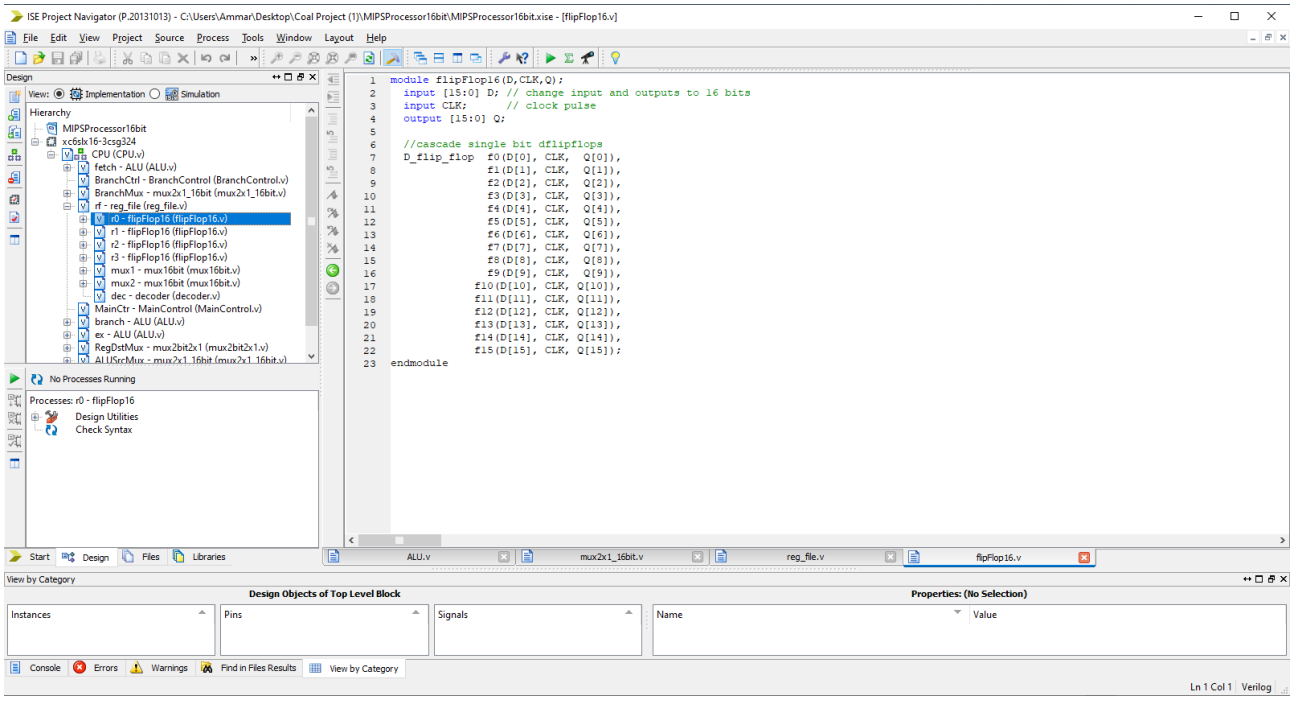
Screenshot 7



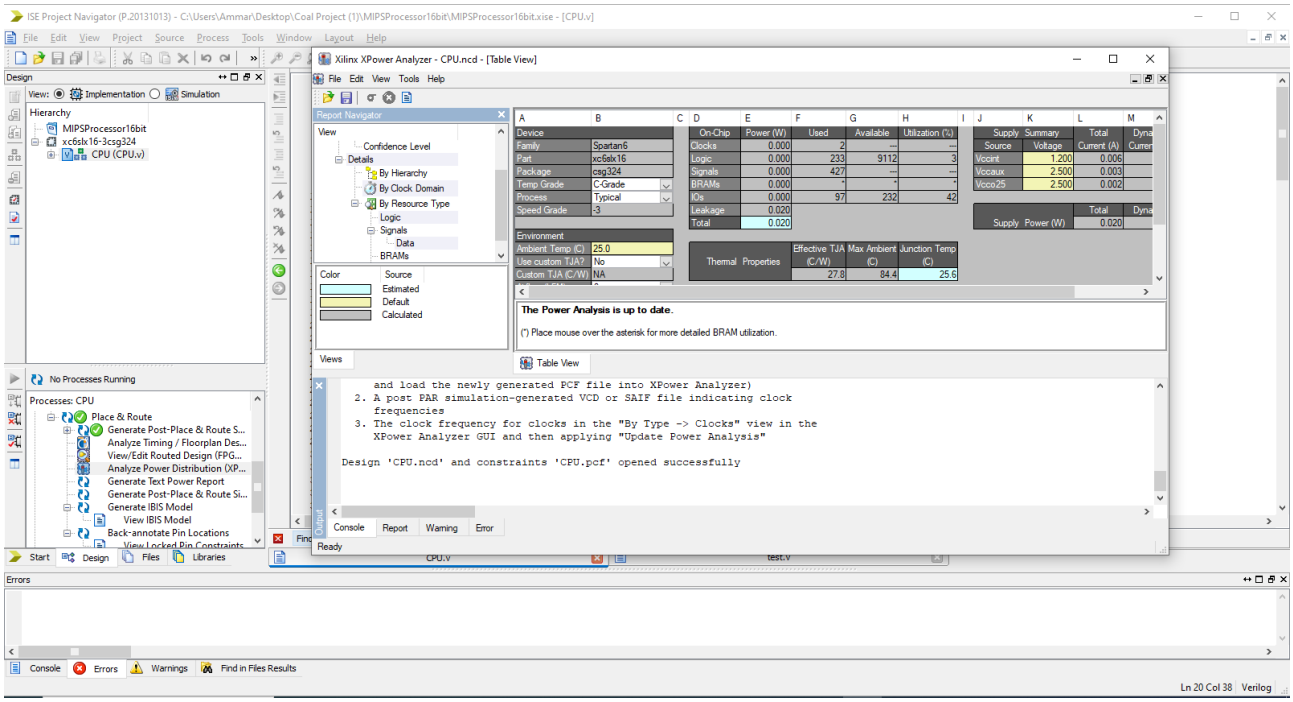
Screenshot 8



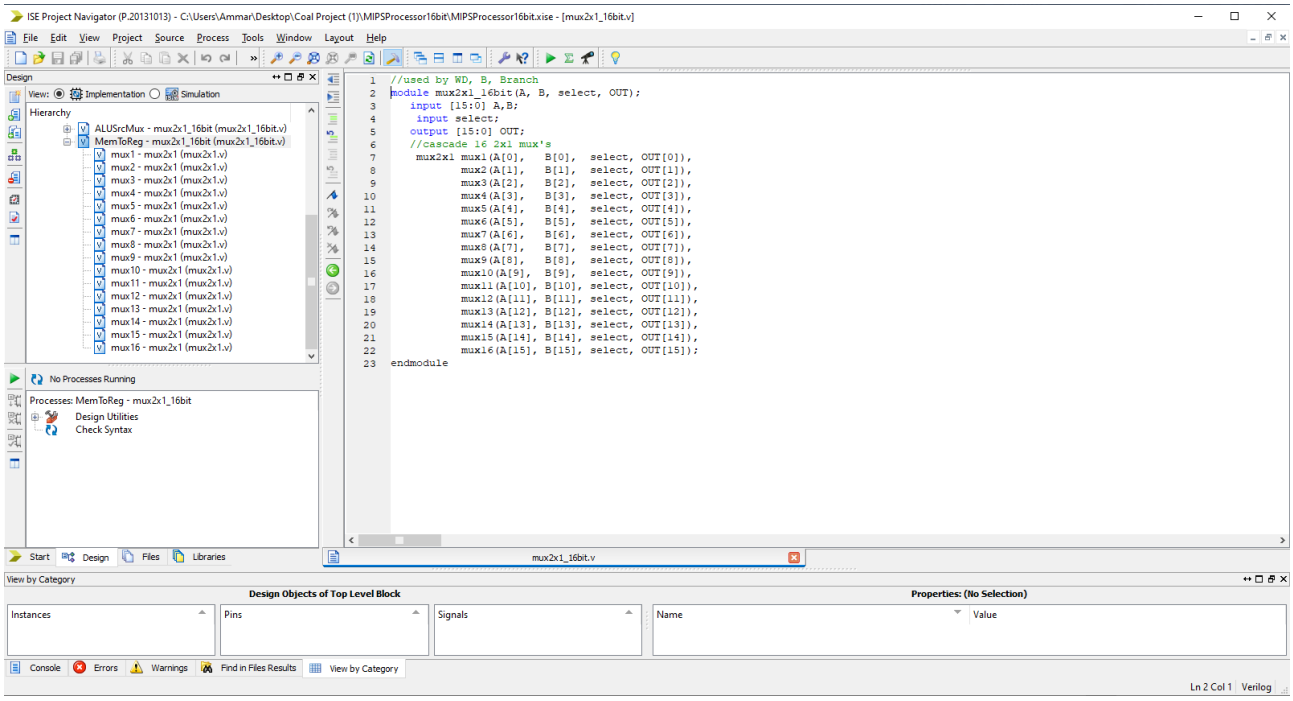
Screenshot 9



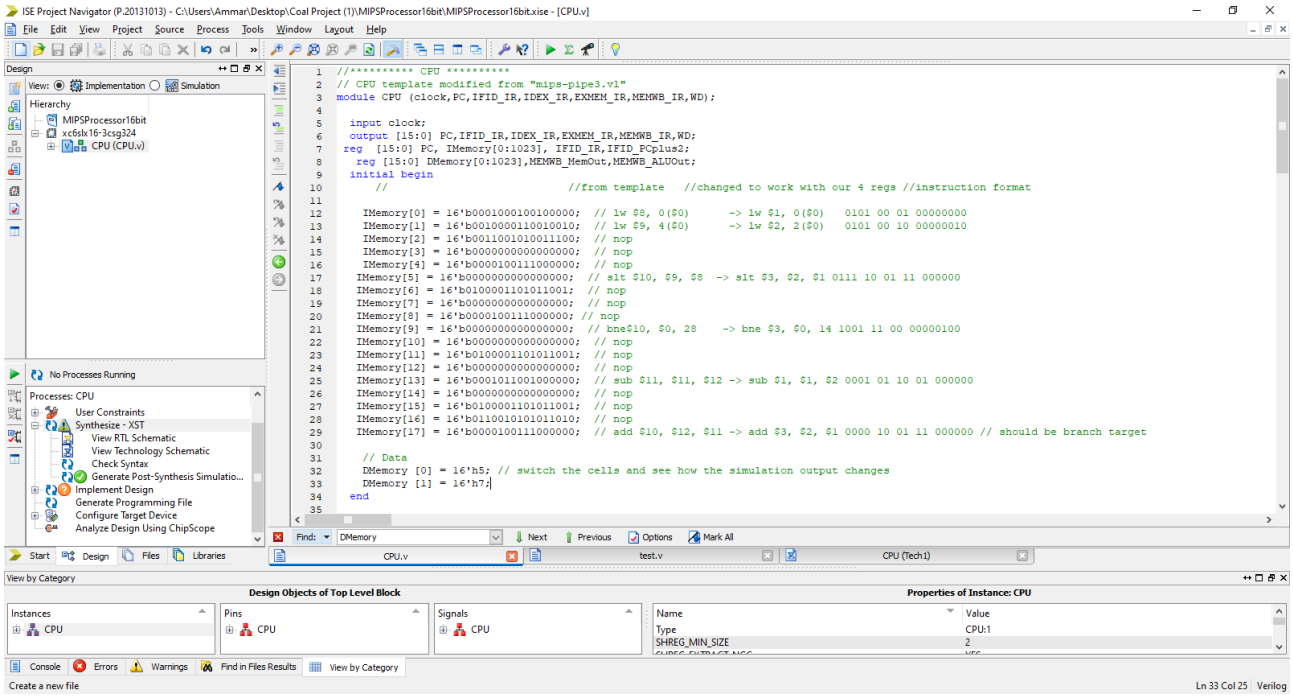
Screenshot 10



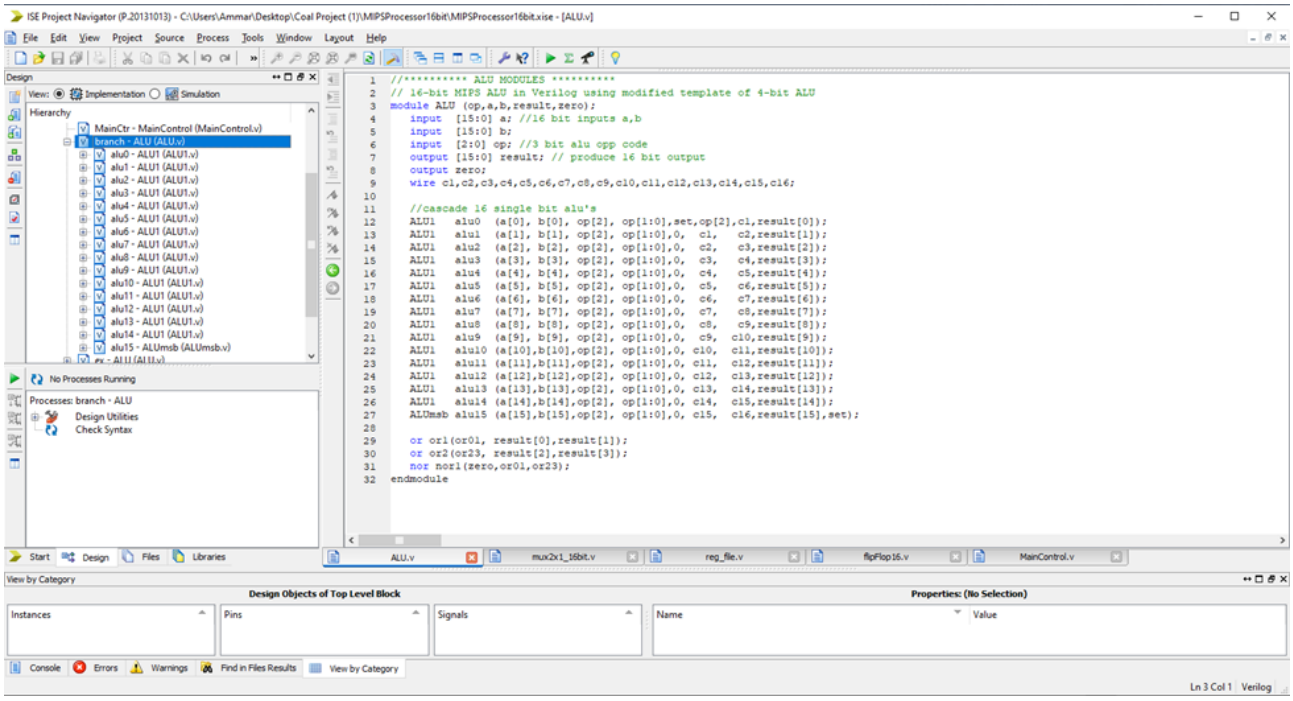
Screenshot 11



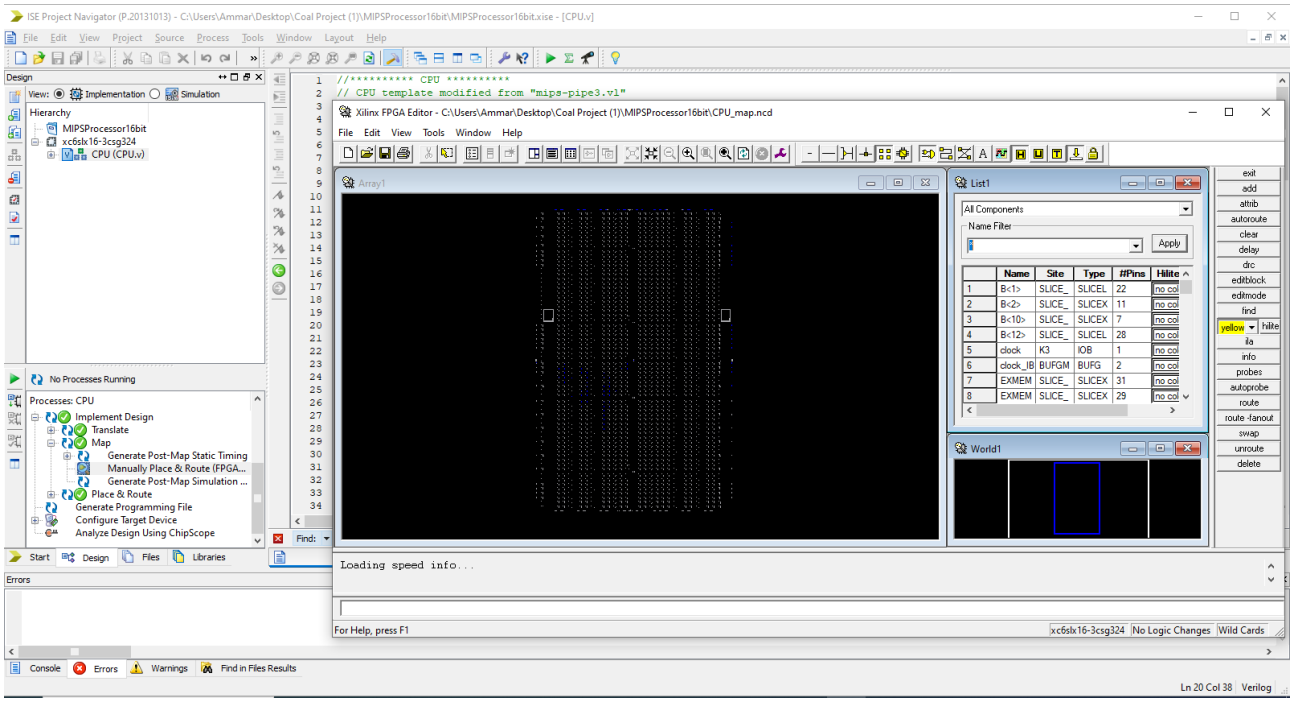
Screenshot 12



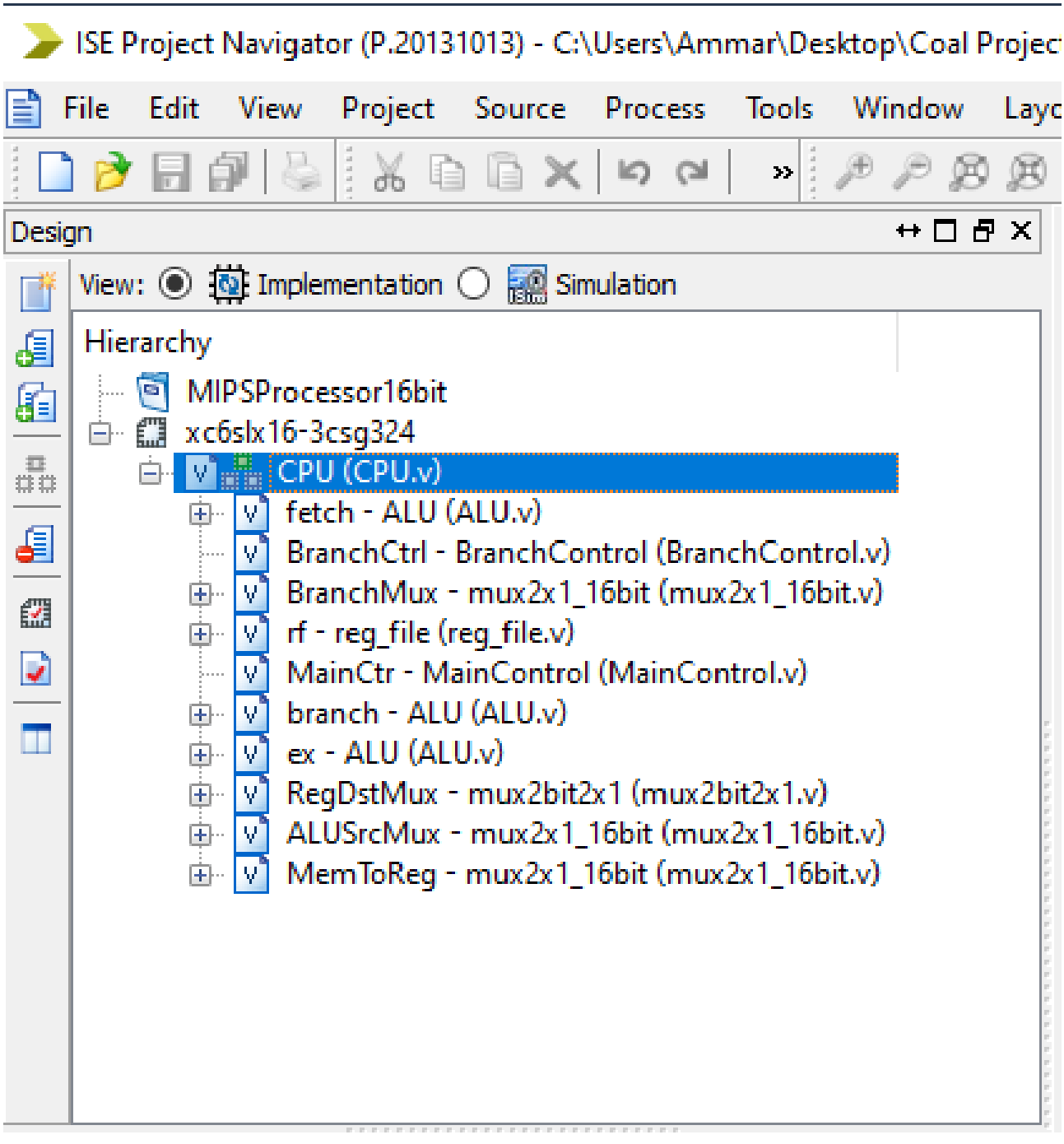
Screenshot 13



Screenshot 14



Screenshot 15



Screenshot 16

```
36  // Pipeline stages
37
38  //IF
39  wire [15:0] NextPC,PCplus2;
40
41  reg [1:0] EXMEM_Branch;
42  reg MEMWB_RegWrite,MEMWB_MemtoReg;
43  reg [15:0] EXMEM_Target,EXMEM_ALUOut,EXMEM_RD2;
44  reg EXMEM_Zero;
45  reg [1:0] MEMWB_rd;
46  ALU fetch (3'b010,PC,2,PCplus2,Unused); //play with timings here
47  BranchControl BranchCtrl (EXMEM_Branch,EXMEM_Zero,BranchConOut); // added branch control
48  mux2x1_16bit BranchMux (PCplus2,EXMEM_Target,BranchConOut,NextPC); // added mux for branch
49
50  //ID
51  reg  [15:0] IDEX_IR; // For monitoring the pipeline
52  wire [9:0] Control;
53  reg IDEX_RegWrite,IDEX_ALUSrc,IDEX_RegDst,IDEX_MemtoReg,IDEX_MemWrite;
54  reg  [1:0] IDEX_Branch; // because our bne and bqe are 2 bits
55  reg  [2:0] IDEX_ALUOp;  // our aluOps are 3 bits
56  wire [15:0] RD1,RD2,SignExtend, WD;
57  reg  [15:0] IDEX_RD1,IDEX_RD2,IDEX_SignExt,IDEX_PCplus2,IDEXE_IR; // added IDEX_PCplus2,IDEXE_IR
58  reg  [1:0]  IDEX_rt,IDEX_rd; //should be 2 bit
59  reg_file rf (IFID_IR[11:10],IFID_IR[9:8],MEMWB_rd,WD,MEMWB_RegWrite,RD1,RD2,clock); // added MEMWB_rd & MEMWB_RegWrite
60  MainControl MainCtr (IFID_IR[15:12],Control);
61  assign SignExtend = {{8{IFID_IR[7]}},IFID_IR[7:0]};
62
63  //EXE
64  reg EXMEM_RegWrite,EXMEM_MemtoReg,EXMEM_MemWrite;
65  wire [15:0] Target;
66
67
68  reg [15:0] EXMEM_IR; // this is for monitoring the pipeline
69  reg [1:0] EXMEM_rd;
70  wire [15:0] B,ALUOut;
```

Screenshot 17

ISE Text Editor (P.20131013) - [CPU.v]

File Edit View Window Layout Help

```
71   wire [1:0] WR;
72   ALU branch (3'b010,IDEX_SignExt<<1,IDEX_PCplus2,Target,Unused2);
73   ALU ex (IDEX_ALUOp, IDEX_RD1, B, ALUOut, Zero);
74   mux2bit2x1 RegDstMux (IDEX_rt, IDEX_rd, IDEX_RegDst, WR); // assign WR, Reg Dst mux
75   mux2x1_16bit ALUSrcMux (IDEX_RD2, IDEX_SignExt, IDEX_ALUSrc, B); // assign B, ALU src mux
76
77
78   //MEM
79
80
81   reg [15:0] MEMWB_IR; // For monitoring the pipeline
82   wire [15:0] MemOut;
83
84   assign MemOut = DMemory[EXMEM_ALUOut>>1];
85   always @(negedge clock) if (EXMEM_MemWrite) DMemory[EXMEM_ALUOut>>1] <= EXMEM_RD2;
86
87   //WB <--- this area is what I believe is causing trouble
88   //assign WD = ALUOut; //WD simply gets ALU output, FROM PR3
89   //assign WD = (MEMWB_MemtoReg) ? MEMWB_MemOut: MEMWB_ALUOut; // MemtoReg Mux, FROM PR4 TEMPLATE
90   mux2x1_16bit MemToReg (MEMWB_ALUOut, MEMWB_MemOut, MEMWB_MemtoReg, WD);
91
92   initial begin
93     PC = 0;
94    // Initialize pipeline registers
95     IDEX_RegWrite=0;IDEX_MemtoReg=0;IDEX_Branch=0;IDEX_MemWrite=0;IDEX_ALUSrc=0;IDEX_RegDst=0;IDEX_ALUOp=0;
96     IFID_IR=0;
97     EXMEM_RegWrite=0;EXMEM_MemtoReg=0;EXMEM_Branch=0;EXMEM_MemWrite=0;
98     EXMEM_Target=0;
99     MEMWB_RegWrite=0;
100    MEMWB_MemtoReg=0;
101  end
102
103  //RUNNING THE PIPELINE
104
105  always @(negedge clock) begin
106
107    //STAGE 1 - IF
108    PC <= NextPC;
109   IFID_PCplus2 <= PCplus2;
110    IFID_IR <= IMemory[PC>>1];
111
112    //STAGE 2 - ID
113    IDEX_IR <= IFID_IR; // For monitoring the pipeline
114   // RegDst[1], AluSrc[1], MemtoReg[1], RegWrite[1], MemWrite1[1], BEQ[1], BNE[1], AluCtrl[3]
115    {IDEX_RegDst,IDEX_ALUSrc,IDEX_MemtoReg,IDEX_RegWrite,IDEX_MemWrite,IDEX_Branch,IDEX_ALUOp} <= Control;
116    IDEX_PCplus2 <= IFID_PCplus2;
117    IDEX_RD1 <= RD1;
118    IDEX_RD2 <= RD2;
```

CPU.v

Ln 97 Col 71 Verilog

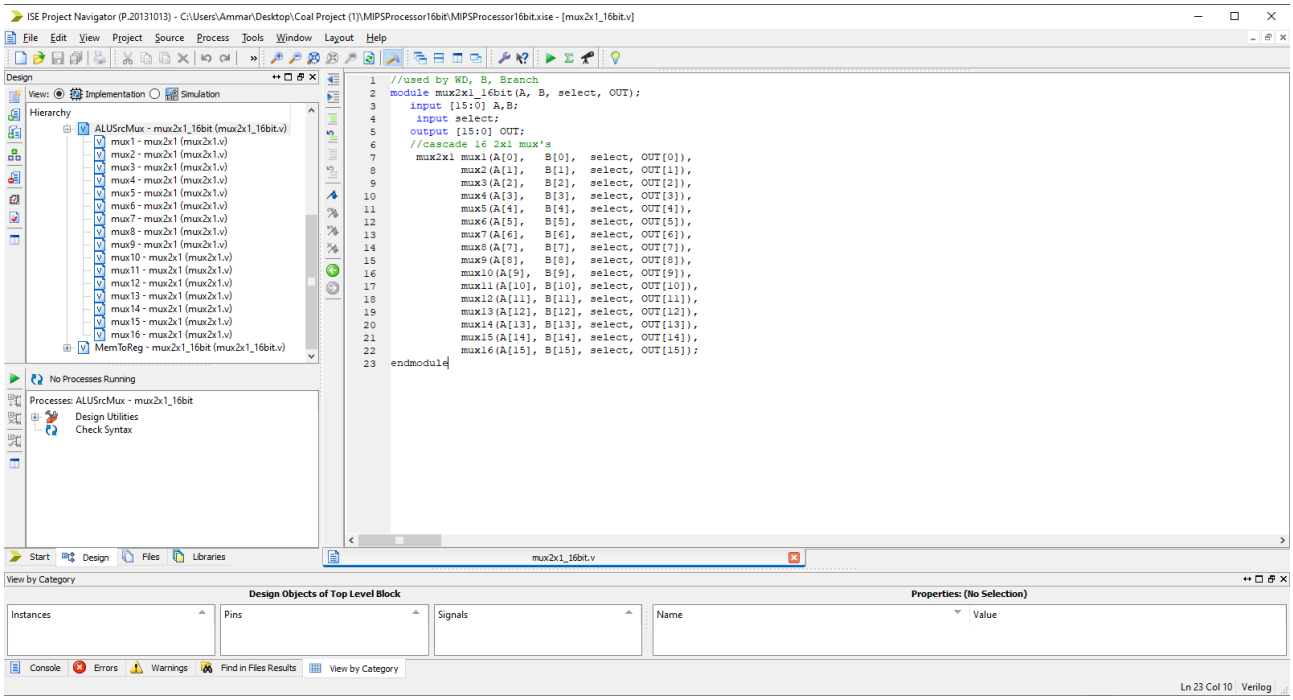
Screenshot 18

```
//used by WD, B, Branch
module mux2x1_16bit(A, B, select, OUT);
   input [15:0] A,B;
    input select;
   output [15:0] OUT;
   //cascade 16 2x1 mux's
    mux2x1 mux1(A[0],   B[0],  select, OUT[0]),
           mux2(A[1],   B[1],  select, OUT[1]),
           mux3(A[2],   B[2],  select, OUT[2]),
           mux4(A[3],   B[3],  select, OUT[3]),
           mux5(A[4],   B[4],  select, OUT[4]),
           mux6(A[5],   B[5],  select, OUT[5]),
           mux7(A[6],   B[6],  select, OUT[6]),
           mux8(A[7],   B[7],  select, OUT[7]),
           mux9(A[8],   B[8],  select, OUT[8]),
           mux10(A[9],  B[9],  select, OUT[9]),
           mux11(A[10], B[10], select, OUT[10]),
           mux12(A[11], B[11], select, OUT[11]),
           mux13(A[12], B[12], select, OUT[12]),
           mux14(A[13], B[13], select, OUT[13]),
           mux15(A[14], B[14], select, OUT[14]),
           mux16(A[15], B[15], select, OUT[15]);
endmodule
```

Screenshot 19

```
//used by WD, B, Branch
module mux2x1_16bit(A, B, select, OUT);
   input [15:0] A,B;
    input select;
   output [15:0] OUT;
   //cascade 16 2x1 mux's
    mux2x1 mux1(A[0],   B[0],  select, OUT[0]),
           mux2(A[1],   B[1],  select, OUT[1]),
           mux3(A[2],   B[2],  select, OUT[2]),
           mux4(A[3],   B[3],  select, OUT[3]),
           mux5(A[4],   B[4],  select, OUT[4]),
           mux6(A[5],   B[5],  select, OUT[5]),
           mux7(A[6],   B[6],  select, OUT[6]),
           mux8(A[7],   B[7],  select, OUT[7]),
           mux9(A[8],   B[8],  select, OUT[8]),
           mux10(A[9],  B[9],  select, OUT[9]),
           mux11(A[10], B[10], select, OUT[10]),
           mux12(A[11], B[11], select, OUT[11]),
           mux13(A[12], B[12], select, OUT[12]),
           mux14(A[13], B[13], select, OUT[13]),
           mux15(A[14], B[14], select, OUT[14]),
           mux16(A[15], B[15], select, OUT[15]);
endmodule
```

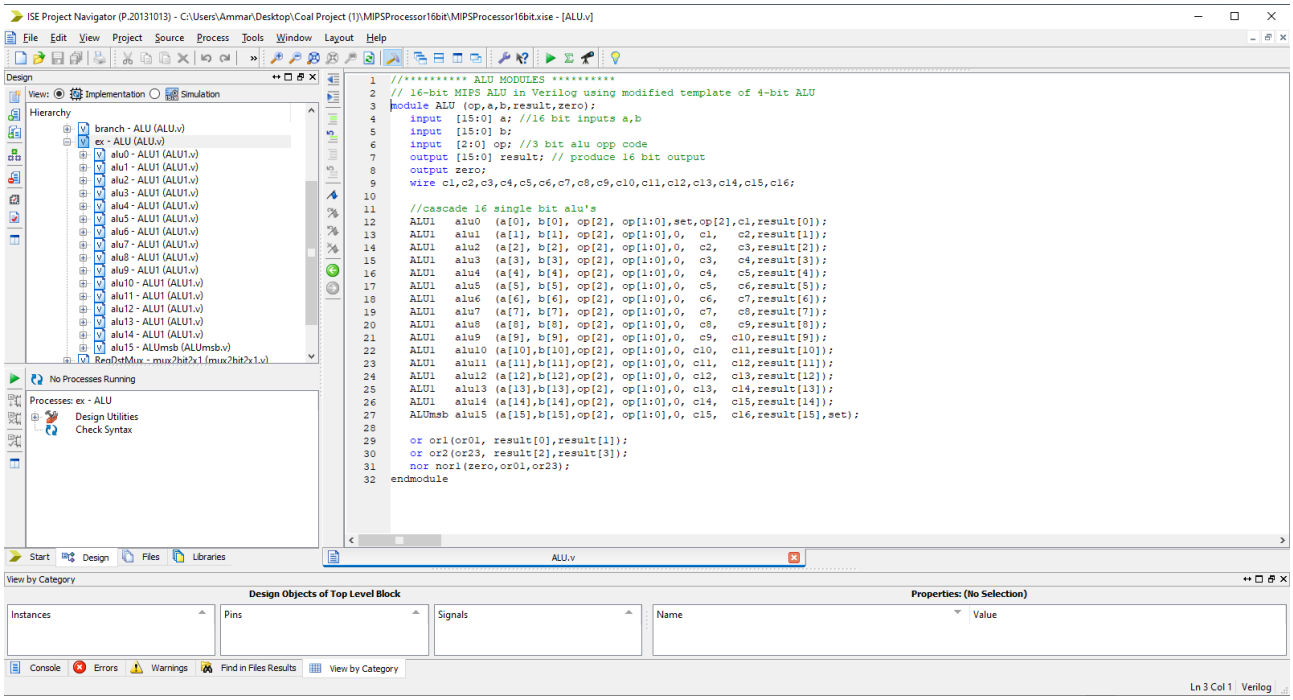
Screenshot 20



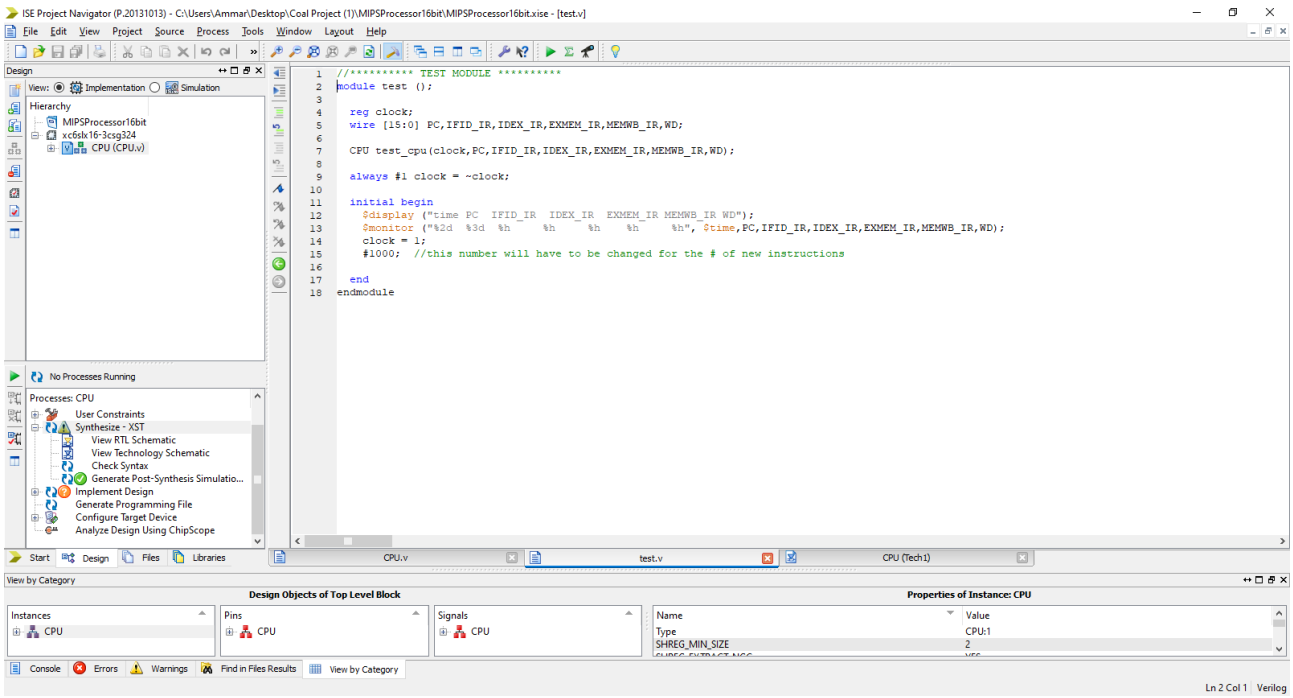
Screenshot 21

```
1  //********** ALU MODULES **********
2  // 16-bit MIPS ALU in Verilog using modified template of 4-bit ALU
3  module ALU (op,a,b,result,zero);
4     input  [15:0] a; //16 bit inputs a,b
5     input  [15:0] b;
6     input  [2:0] op; //3 bit alu opp code
7     output [15:0] result; // produce 16 bit output
8     output zero;
9     wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16;
10
11    //cascade 16 single bit alu's
12    ALU1   alu0  (a[0], b[0], op[2], op[1:0],set,op[2],c1,result[0]);
13    ALU1   alu1  (a[1], b[1], op[2], op[1:0],0,  c1,   c2,result[1]);
14    ALU1   alu2  (a[2], b[2], op[2], op[1:0],0,  c2,   c3,result[2]);
15    ALU1   alu3  (a[3], b[3], op[2], op[1:0],0,  c3,   c4,result[3]);
16    ALU1   alu4  (a[4], b[4], op[2], op[1:0],0,  c4,   c5,result[4]);
17    ALU1   alu5  (a[5], b[5], op[2], op[1:0],0,  c5,   c6,result[5]);
18    ALU1   alu6  (a[6], b[6], op[2], op[1:0],0,  c6,   c7,result[6]);
19    ALU1   alu7  (a[7], b[7], op[2], op[1:0],0,  c7,   c8,result[7]);
20    ALU1   alu8  (a[8], b[8], op[2], op[1:0],0,  c8,   c9,result[8]);
21    ALU1   alu9  (a[9], b[9], op[2], op[1:0],0,  c9,  c10,result[9]);
22    ALU1   alu10 (a[10],b[10],op[2], op[1:0],0, c10,  c11,result[10]);
23    ALU1   alu11 (a[11],b[11],op[2], op[1:0],0, c11,  c12,result[11]);
24    ALU1   alu12 (a[12],b[12],op[2], op[1:0],0, c12,  c13,result[12]);
25    ALU1   alu13 (a[13],b[13],op[2], op[1:0],0, c13,  c14,result[13]);
26    ALU1   alu14 (a[14],b[14],op[2], op[1:0],0, c14,  c15,result[14]);
27    ALUmsb alu15 (a[15],b[15],op[2], op[1:0],0, c15,  c16,result[15],set);
28
29    or or1(or01, result[0],result[1]);
30    or or2(or23, result[2],result[3]);
31    nor nor1(zero,or01,or23);
32 endmodule
```

Screenshot 22


ISE Project Navigator (P.20131013) - C:\Users\Ammar\Desktop\Coal Project (1)\MIPSProcessor16bit\MIPSProcessor16bit.xise - [ALU.v]
File Edit View Project Source Process Tools Window Layout Help
Design
View: Implementation Simulation
Hierarchy
branch - ALU (ALU.v)
ex - ALU (ALU.v)
alu0 - ALU1 (ALU1.v)
alu1 - ALU1 (ALU1.v)
alu2 - ALU1 (ALU1.v)
alu3 - ALU1 (ALU1.v)
alu4 - ALU1 (ALU1.v)
alu5 - ALU1 (ALU1.v)
alu6 - ALU1 (ALU1.v)
alu7 - ALU1 (ALU1.v)
alu8 - ALU1 (ALU1.v)
alu9 - ALU1 (ALU1.v)
alu10 - ALU1 (ALU1.v)
alu11 - ALU1 (ALU1.v)
alu12 - ALU1 (ALU1.v)
alu13 - ALU1 (ALU1.v)
alu14 - ALU1 (ALU1.v)
alu15 - ALUmsb (ALUmsb.v)
No Processes Running
Processes: ex - ALU
Design Utilities
Check Syntax
Start Design Files Libraries
//********** ALU MODULES **********
// 16-bit MIPS ALU in Verilog using modified template of 4-bit ALU
module ALU (op,a,b,result,zero);
input [15:0] a; //16 bit inputs a,b
input [15:0] b;
input [2:0] op; //3 bit alu opp code
output [15:0] result; // produce 16 bit output
output zero;
wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16;
//cascade 16 single bit alu's
ALU1 alu0 (a[0], b[0], op[2], op[1:0],set,op[2],c1,result[0]);
ALU1 alu1 (a[1], b[1], op[2], op[1:0],0, c1, c2,result[1]);
ALU1 alu2 (a[2], b[2], op[2], op[1:0],0, c2, c3,result[2]);
ALU1 alu3 (a[3], b[3], op[2], op[1:0],0, c3, c4,result[3]);
ALU1 alu4 (a[4], b[4], op[2], op[1:0],0, c4, c5,result[4]);
ALU1 alu5 (a[5], b[5], op[2], op[1:0],0, c5, c6,result[5]);
ALU1 alu6 (a[6], b[6], op[2], op[1:0],0, c6, c7,result[6]);
ALU1 alu7 (a[7], b[7], op[2], op[1:0],0, c7, c8,result[7]);
ALU1 alu8 (a[8], b[8], op[2], op[1:0],0, c8, c9,result[8]);
ALU1 alu9 (a[9], b[9], op[2], op[1:0],0, c9, c10,result[9]);
ALU1 alu10 (a[10],b[10],op[2], op[1:0],0, c10, c11,result[10]);
ALU1 alu11 (a[11],b[11],op[2], op[1:0],0, c11, c12,result[11]);
ALU1 alu12 (a[12],b[12],op[2], op[1:0],0, c12, c13,result[12]);
ALU1 alu13 (a[13],b[13],op[2], op[1:0],0, c13, c14,result[13]);
ALU1 alu14 (a[14],b[14],op[2], op[1:0],0, c14, c15,result[14]);
ALUmsb alu15 (a[15],b[15],op[2], op[1:0],0, c15, c16,result[15],set);
or or1(or01, result[0],result[1]);
or or2(or23, result[2],result[3]);
nor nor1(zero,or01,or23);
endmodule
ALU.v
View by Category
Design Objects of Top Level Block
Instances
Pins
Signals
Properties: (No Selection)
Name
Value
Console Errors Warnings Find in Files Results View by Category
Ln 3 Col 1 Verilog

6 Test Bench


ISE Project Navigator (P.20131013) - C:\Users\Ammar\Desktop\Coal Project (1)\MIPSProcessor16bit\MIPSProcessor16bit.xise - [test.v]
File Edit View Project Source Process Tools Window Layout Help
Design
View: Implementation Simulation
Hierarchy
MIPSProcessor16bit
xc6slx16-3csg324
CPU (CPU.v)
No Processes Running
Processes: CPU
User Constraints
Synthesize - XST
View RTL Schematic
View Technology Schematic
Check Syntax
Generate Post-Synthesis Simulatio...
Implement Design
Generate Programming File
Configure Target Device
Analyze Design Using ChipScope
Start Design Files Libraries
//********** TEST MODULE **********
module test ();
reg clock;
wire [15:0] PC,IFID_IR,IDEX_IR,EXMEM_IR,MEMWB_IR,WD;
CPU test_cpu(clock,PC,IFID_IR,IDEX_IR,EXMEM_IR,MEMWB_IR,WD);
always #1 clock = ~clock;
initial begin
$display ("time PC IFID_IR IDEX_IR EXMEM_IR MEMWB_IR WD");
$monitor ("%2d %3d %h %h %h %h %h", $time,PC,IFID_IR,IDEX_IR,EXMEM_IR,MEMWB_IR,WD);
clock = 1;
#1000; //this number will have to be changed for the # of new instructions
end
endmodule
CPU.v
test.v
CPU (Tech1)
View by Category
Design Objects of Top Level Block
Instances
CPU
Pins
CPU
Signals
CPU
Properties of Instance: CPU
Name
Value
Type
CPU:1
SHREG_MIN_SIZE
2
Console Errors Warnings Find in Files Results View by Category
Ln 2 Col 1 Verilog

7 Conclusion

Designing a 16-bit MIPS processor datapath using Xilinx involves several key steps and considerations. First, it is important to select the appropriate MIPS ISA variant that suits the project objectives. This will determine the instruction set and architecture of the processor. The MIPS datapath consists of various components, including the register file, ALU, control unit, multiplexers, and data memory. These components need to be designed and implemented to ensure smooth operation of the processor. The design process begins with implementing the instruction fetch stage, which involves fetching instructions from memory and updating the program counter (PC). The fetched instructions are then decoded, and the required registers are identified and accessed from the register file. Arithmetic and logical operations are performed using the ALU, which is an essential component of the datapath. The control unit is responsible for generating control signals based on the instruction being executed. To read from and write to memory, a data memory unit is included in the design. The register file is updated with the results of executed instructions. Thorough testing is crucial to ensure the correct functionality of the MIPS processor datapath. Various test cases should be executed to verify that the processor is executing instructions accurately. Once the design is tested and validated, it needs to be synthesized using Xilinx tools. This process generates a gate-level netlist, which can then be implemented on the target FPGA device. Performance analysis is important to evaluate the efficiency and effectiveness of the MIPS processor datapath. Any necessary optimizations can be applied to improve performance. It is also essential to consider system-level integration and interface the processor with peripherals and external memory as required by the project. Throughout the design process, it is important to follow good design practices, adhere to the MIPS architecture specifications, and leverage Xilinx design tools and resources for efficient implementation. Detailed guidance and support can be obtained from relevant textbooks, documentation, and experienced designers.

8 References

[1] Christine Connolly. “Pros from touchics”. In: *Industrial Robot: An International Journal* (2008).

[2] Michael H Dickinson. “insight into mechanical design”. In: *Proceedings of the National Academy of Sciences* 96.25 (1999), pp. 14208–14209.

[3] Robert Rosen. “revisited”. In: *The machine as metaphor and tool*. Springer, 1993, pp. 87–100.

[4] René Romain Roth. “The foundation of nics”. In: *Perspectives in bicine* 26.2 (1983), pp. 229–242.