BIRZEIT UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Computer Architecture

ENCS4370

Project No. 2

Multi-Cycle RISC processor

Prepared by

Sara Issa – 1190673

Raneem Daqa – 1202093

Leen Abu Omar – 1190113

Supervised by

Dr. Aziz Qaroush

BIRZEIT

January – 2024

Abstract

The design and development of a multi-cycle processor built around a particular RISC instruction set is the main topic of this study. To determine the primary components needed, a detailed examination of every instruction was part of the design process. After that, these parts were constructed and included into the processor's architecture. The process of implementation included determining the control signals for each component separately. This required figuring out the right signal timing and order to guarantee correct instruction execution. Extensive testing was done to confirm the processor's operation.

To ensure that each instruction operated as intended, it was put to the test separately. Furthermore, the processor was subjected to comprehensive scenarios in order to assess its performance and guarantee precise outcomes.

During the phases of design, implementation, and testing, the Multi-Cycle Processor's accuracy and efficiency in executing the specified RISC instruction set was assessed. The testing process's outcomes validated the processor's design and implementation decisions by confirming that it operates as expected.

Table of Contents

Acronyms and Abbreviations

RTN	Register Transfer Notation
R-type	Register Type
I-type	Immediate Type
J-type	Jump Type
S-type	Stack Type
CPU	Central Processing Unit
ALU	Arithmetic Logic Unit
PC	Program Counter
IF	Instruction Fetch
ID	Instruction Decode
EX	Execute
MEM	Memory
WB	Write Back

1. Introduction

A major advancement in CPU design is the multi-cycle RISC processor architecture, or RISC-V, the modular instruction set architecture (ISA) of RISC-V, which was first created at UC Berkeley, is well-known for being open source and permitting a great level of customisation and flexibility in processor design. Because of its versatility, RISC-V is becoming more and more popular, particularly for applications in the automotive, 5G mobile, artificial intelligence, and data center domains.[1][2]

A RISC-V processor's execution pipeline consists of multiple phases, including Memory Access (MEM), Write Back (WB), Instruction Fetch (IF), Instruction Decode (ID), and Execute (EX), every step is essential to the effective processing of instructions. While the ID stage decodes and ascertains the controls required for the instruction, the IF stage is in charge of reading the instruction from the program counter. The outcome of several different kinds of instructions, such as ALU, DIV, MUL, Store, Load, and branch/jump instructions, is calculated in the EX-stage. The findings are written back into the register file at the WB stage after the MEM stage manages memory access.[3]

A branch predictor unit, which optimizes decision-making in branch instructions, and data and instruction caches, which expedite memory access, are other features of the RISC-V architecture. These elements support the processor's overall effectiveness and performance.

Modularity is one of the features that set RISC-V apart from other architectures. Multiple base ISAs and optional extensions are included in the RISC-V ISA for particular functionality like vector instructions and single- and double-precision floating-point calculations. Hardware makers can customize the processor to meet specific requirements thanks to this modular approach, which is especially advantageous for embedded or edge devices where power conservation is a crucial consideration.[4]

Due to its modular design and open-source nature, RISC-V offers flexibility, efficiency, and a multitude of application options, making it a significant breakthrough in RISC architecture overall.

The aim in this project is to use the VHDL language to design and test a basic multi-cycle RISC processor. A restricted set of instructions covering memory operations, logical operations, arithmetic calculations, and stack are supported by the processor's design. The suggested processor architecture's use of the multi-cycle approach makes it possible to implement it simply and with a smaller instruction set. Our goal in creating this simple multicycle RISC processor is to obtain a thorough understanding of processor design and computer architecture principles.

2. Design Specification and Implementation

2.1. Motivation

In this project, we present the design of a straightforward multi-cycle RISC processor, driven by the following specifications:

- Efficient Instruction Handling: The processor operates with a 32-bit instruction size, ensuring streamlined instruction processing.
- Abundant General-Purpose Registers: Featuring 32 32-bit general-purpose registers (R0 to R31), our design facilitates versatile data manipulation and storage.
- Program Control Management: A dedicated program counter (PC) and a specialized control stack for return addresses enhance program control and execution flow.
- Stack Pointer Efficiency: The stack pointer (SP) is a special-purpose register that efficiently points to the top of the control stack, streamlining stack management. The initial SP value is zero.
- Versatile Instruction Types: The processor supports four instruction types: ‘R-type, I-type, J-type, and S-type’ providing flexibility for diverse programming needs.
- ALU Output Signal: The processor's ALU includes a valuable "zero" signal, indicating when the result of the last ALU operation is zero, and a "negative" signal, indicating when the result of the last ALU operation is negative. These signals facilitate efficient conditional branching in programming.
- Memory Separation: Separate data and instruction memories optimize memory access and execution efficiency.

2.2. Overall design

The multi-cycle design method is used to build a processor that can carry out several types of instructions. The data flow in this system is segmented into steps, each of which is associated with a distinct clock cycle. A distinct portion of the instruction is carried out by each clock cycle, and the outcomes are saved in the relevant register or memory address.

At the end of each clock cycle, data needed for upcoming clock cycles is saved in state elements to reduce redundancy. The register file, program counter (PC), memory, and stack are examples of parts of the programmer-visible state that store data required for instructions in subsequent clock cycles. Nevertheless, extra registers hold information that the same instruction needs in subsequent cycles.

Rather of trying to complete every task in a single cycle, the data route for this multi-cycle RISC processor was developed using a technique that focused on segmenting instruction execution into manageable, sequential phases. These actions are grouped together and carried out in a particular order.

The key groups in this design include:

- Fetching the instruction from memory.
- Decoding the instruction and reading the relevant registers.
- Executing ALU calculations or accessing memory.
- Writing the results back to the register file.
- Determining the next program counter (PC) and updating it.

Several components, including registers, an ALU, multiplexers, memory, and a control unit, are included in the overall data path design. In order to maximize hardware utilization and facilitate functional unit sharing during the execution of a single instruction, these components were chosen and connected. The arrangement of additional registers is determined by the fitting of combination units into a single clock cycle and the information needed for the correct execution of instructions in following cycles.

Each clock cycle in this multi-cycle design can support a maximum of one operation, such as an ALU operation, Stack operation (push and pop operations), register file access, instruction memory access, or data memory access. The data produced by these functional units is stored in temporary registers for use in later cycles in order to prevent timing conflicts and guarantee accuracy. This reduces the possibility of utilizing wrong values and lessens the requirement for duplicate data copying.

2.3. Instruction Types and Formats

There are four types of instructions in the ISA: R-type, I-type, J-type, and S-type. Each of these types of instruction has an opcode field of 6 bits, which determines its specific operation.

2.3.1. Register Type (R-type)

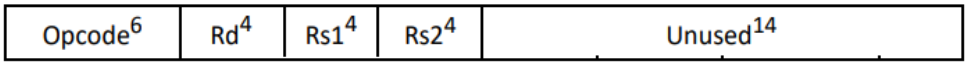


Figure 2-1: R-type Instruction Format

There are the following fields in the R-type format:

- 4-bit Rd: destination register
- 4-bit Rs1: first source register
- 4-bit Rs2: second source register
- 14-bit unused

2.3.2. Immediate Type (I-type)

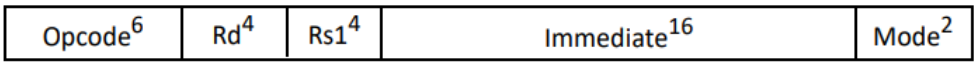


Figure 2-2: I-type Instruction Format

There are the following fields in the I-type format:

- 4-bit Rd: destination register
- 4-bit Rs1: first source register
- 16-bit immediate: unsigned for logic instructions, and signed otherwise.
- 2-bit mode: It is used only with load/store instructions, such that
  - 00: no increment/decrement of the base register
  - 01: post increment the base register
  - 10-11: unused

2.3.3. Jump Type (J-type)

Instruction formats of J-type include the following. To distinguish between instructions, the opcode is used.

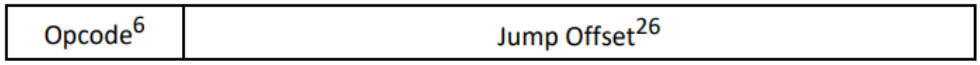


Figure 2-3: J-type Instruction Format 1

- JMP L # Unconditional jump to the target L.
- CALL F # Call the function F. F is a label.

  # The return address is pushed on the stack

- 26-bit Jump Offset: By concatenating the 16-bit offset with the 6 bits most significant in the current PC, the target address is calculated.

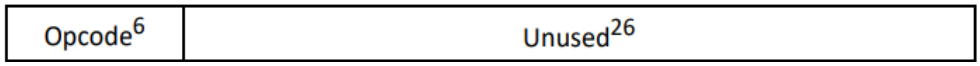


Figure 2-4: J-type Instruction Format 2

- RET # return from a function.

  # The next PC will be the top element of the stack

- 26-bit Unused.

2.3.4. Stack Type (S-type)

- PUSH Rd # Push the value of Rd on the top of the stack
- POP Rd # Pop the stack and store the topmost element in Rd

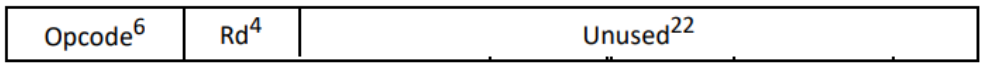


Figure 2-5: S-type Instruction Format

There are the following fields in the S-type format:

- 4-bit Rd.

Below is a table showing the instructions supported by this instruction set, with their opcode values, and their RTN (Register Transfer Notation) meanings.

No.	Instr	Meaning	Opcode Value
R-Type Instructions			
1	AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	000000
2	ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	000001
3	SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	000010
I-Type Instructions			
4	ANDI	Reg(Rd) = Reg(Rs1) & Imm^{16}	000011
5	ADDI	Reg(Rd) = Reg(Rs1) + Imm^{16}	000100
6	LW	Reg(Rd) = Mem(Reg(Rs1) + Imm^{16})	000101
7	LW.POI	Reg(Rd) = Mem(Reg(Rs1) + Imm^{16}) Reg[Rs1] = Reg[Rs1] + 1	000110
8	SW	Mem(Reg(Rs1) + Imm^{16}) = Reg(Rd)	000111
9	BGT	if (Reg(Rd) > Reg(Rs1)) Next PC = PC + sign_extended (Imm^{16}) else PC = PC + 1	001000
10	BLT	if (Reg(Rd) < Reg(Rs1)) Next PC = PC + sign_extended (Imm^{16}) else PC = PC + 1	001001
11	BEQ	if (Reg(Rd) == Reg(Rs1)) Next PC = PC + sign_extended (Imm^{16}) else PC = PC + 1	001010
12	BNE	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm^{16}) else PC = PC + 1	001011
J-Type Instructions			
13	JMP	Next PC = {PC[31:26], $Immediate^{26}$ }	001100
14	CALL	Next PC = {PC[31:26], $Immediate^{26}$ } PC + 1 is pushed on the stack	001101
15	RET	Next PC = top of the stack	001110
S-Type Instructions			
16	PUSH	Rd is pushed on the top of the stack	001111
17	POP	The top element of the stack is popped, and it is stored in the Rd register	010000

Figure 2-6: Instructions' Encoding

2.4. Components

The following elements are required, as we discovered after examining the instruction set.

2.4.1. Instruction Memory

A multi-cycle processor's instruction memory functions as a read-only component, providing instructions to the processor based on the address input that is supplied. Combinational logic can be used to implement it because of its simple architecture and write access limitation. Read control signals are not required with this configuration.

The instruction memory module in our project, called InstMem, has a 32-bit address input signal (Address). It also has a 32-bit signal (Din). In addition, an output wire (Instruction) is present to send the instruction that was retrieved from the designated memory address. The effective storage and retrieval of instructions by the InstMem module is made possible by these input and output components, which in turn plays a crucial role in the execution of program instructions and makes a substantial contribution to the general functioning of the computer system.

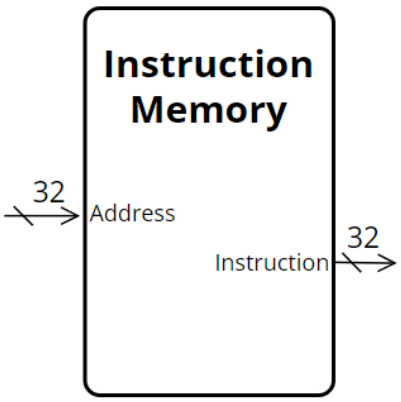


Figure 2-7: Instruction Memory

2.4.2. Register File

A CPU register file is a type of memory storage that holds and retrieves data. It is made up of several registers that are used for mapping and have distinct addresses. Inputs such as a data bus, read/write enabled, and a clock signal for synchronization can be used to access and modify these registers, which store data. An essential component of the CPU's effective data storing and retrieval system is the register file. register file is a type of memory storage that holds and retrieves data. It is made up of several registers that are used for mapping and have distinct addresses. Inputs such as a data bus, read/write enabled, and a clock signal for synchronization can be used to access and modify these registers, which store data. An essential component of the CPU effective data storing and retrieval system is the register file.

The Register File module in our project is designed to manage registers within a computer's CPU. It has inputs including the clock signal (Clk), two register write signals (RegRw)stands for Register Read/Write, and (Rs1Rw) specific control line for reading or writing from the first source register, and 4-bit address signals (RA, RB, RW). Additionally, there is a 32-bit input signals (BusW), (BusW1) for data to be written into the registers. The module provides two 32-bit output wires (BusA, BusB) for reading data from the specified registers. By efficiently handling these inputs and outputs, the RegFile module facilitates effective storage and retrieval of data within the CPU registers, contributing to the overall functionality and performance of the computer system.

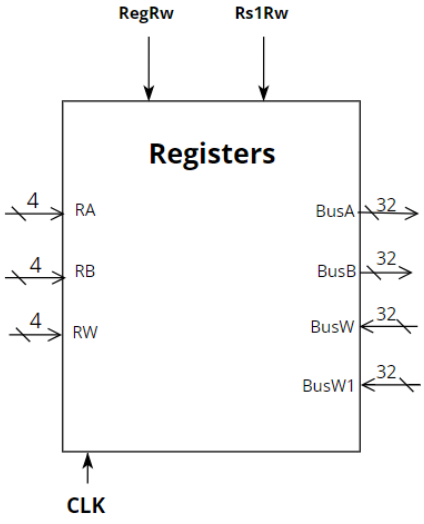


Figure 2-8: Register File

2.4.3. Extender

A key component of a computer's CPU that changes a data value size is the extender. In this implementation, the 16-bit is extended immediately. It can expand the supplied input by passing an instant to the extender. The extender's function is to make the immediate value bigger so that it can be used in actions that require bigger amounts of data. The extender is essential to the CPU overall functionality since it gives the CPU the required size extension, ensuring that the CPU can access the right amount of data for a variety of functions.

Our project extender module uses the (ExtOp) inputs to extend a 32-bit input signal (Imm,16), with extended bits, it generates an output signal with 32 bits. The (ExtOp) input determines whether signing or not to do the extension. The module sets the remaining bits in accordance with the kind of extension and duplicates the corresponding bits from the input signal to the output. Signed extension is carried out by setting the remaining bits to the sign bit if ExtOp is true. To create an unsigned extension, set the remaining bits to zero if ExtOp is false. The sign extender module successfully extends the input signal to the required length by using this logic.

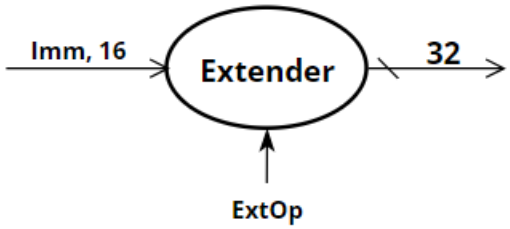


Figure 2-9: Extender

2.4.4. Arithmetic Logic Unit (ALU)

As a crucial part of a computer's Central Processing Unit (CPU), the Arithmetic Logic Unit (ALU) performs basic arithmetic and logical operations that are necessary for computing activities. Managing logical operations like AND, OR, NOT, and XOR in addition to addition, subtraction, multiplication, and division is all part of its complex function. Only binary data is processed by the ALU, which produces outputs that are either used as memory addresses or kept in registers.

The ALU is outfitted with logic gates and specific circuits to provide the best speed and efficiency. Together, these parts expedite computations and guarantee the ALU's contribution to system performance. Input and output buses enable smooth data transfer for information retrieval, processing, and writing back by facilitating communication with other CPU components.

The particular instruction opcode that is being executed controls the ALU's components' ongoing activity. The opcode determines the type of operation—logical or arithmetic—and affects the output that is transmitted to the result bus. The ALU also has two flags: zero and negative, which indicate whether or not zero exists and which indicates the direction of the outcome, respectively.

Moreover, the ALU includes carry and overflow flags, which are important signals in arithmetic operations. Whether a carry-out or borrow occurred during addition or subtraction is indicated by the carry flag. Concurrently, the overflow indicator indicates whether the outcome of a signed arithmetic operation surpasses the representable range. In order to ensure that carry and overflow conditions are handled correctly, these flags provide vital status information that supports conditional branching or following instructions. This painstaking attention to detail improves the arithmetic and logical computations' accuracy and dependability.

The ALU module in our project uses a 6-bit opcode to process the 32-bit inputs A and B. A 32-bit result is the result, along with carry, zero, negative, and overflow flags. These flags offer useful information about the occurrence of overflow, carry-out, zero result, and negative result circumstances, respectively, depending on which 6-bit opcodes are used to execute the instructions. The ALU computes results and status flags in an efficient manner based on the enlarged opcode space, enabling a wide variety of arithmetic and logical operations in the CPU. This wider range of opcodes makes it possible to control processes more precisely, which improves the computing system's overall effectiveness and functionality.

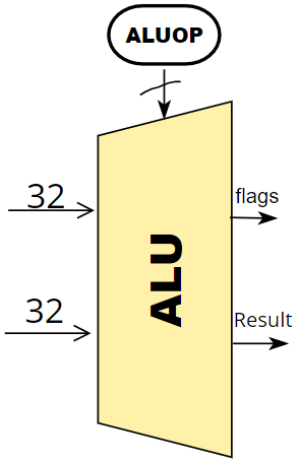


Figure 2-10: Arithmetic Logic Unit

2.4.5. Data Memory

The Data Memory component in a processor is a critical module responsible for storing and retrieving data during program execution. In general, a comprehensive data memory unit typically features specifications such as memory width, address width, data input/output, clock signal, and control signals for read and write operations. In the context of a 32-bit data memory, the module can store and process data values with a width of 32 bits. The address input, typically 32 bits wide, enables the module to access a vast memory space, allowing for the potential addressing of over four billion memory locations. Control signals, including those for memory read (MemR) and memory write (MemW), along with a clock signal (CLK), govern the timing and execution of read and write operations. Data_in and Data_out, both 32 bits wide, represent the data to be written into and read from memory, respectively.

In our project, the Data Memory module plays a pivotal role in managing data read and write operations. This module adheres to the specifications mentioned earlier, with a memory width of 32 bits, a 32-bit address input, and 32-bit data input and output. The clock signal (CLK) ensures proper synchronization, controlling the timing of memory operations. Two crucial control signals, MemR and MemW, dictate the nature of the operation – whether it is a read or a write. During a read operation, the Data Memory module retrieves data from the memory location specified by the 32-bit address input and updates the Data_out value. Conversely, in a write operation, the module writes the 32-bit data provided in Data_in to the memory location specified by the address. This project-specific Data Memory component provides an efficient and flexible means of handling data storage and retrieval within the processor, contributing to the overall functionality and performance of the system. The careful integration of clock synchronization and control signals ensures that data is accurately read from and written to memory, meeting the demands of diverse computational tasks.

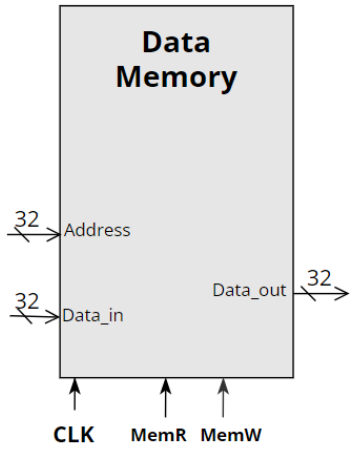


Figure 2-11: Data Memory

2.5. State diagram

As a first step in implementing the desired design, it is essential to define the flow of the state changes throughout the execution of the processer. The figure below shows a state diagram.

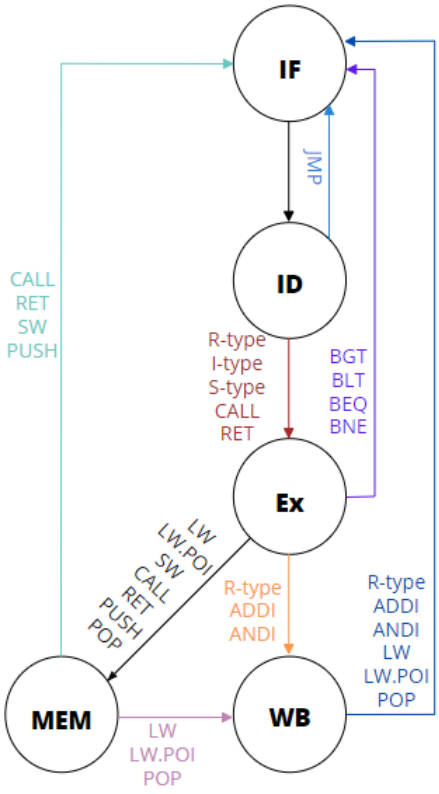


Figure 2-12: State diagram

2.6. Datapath

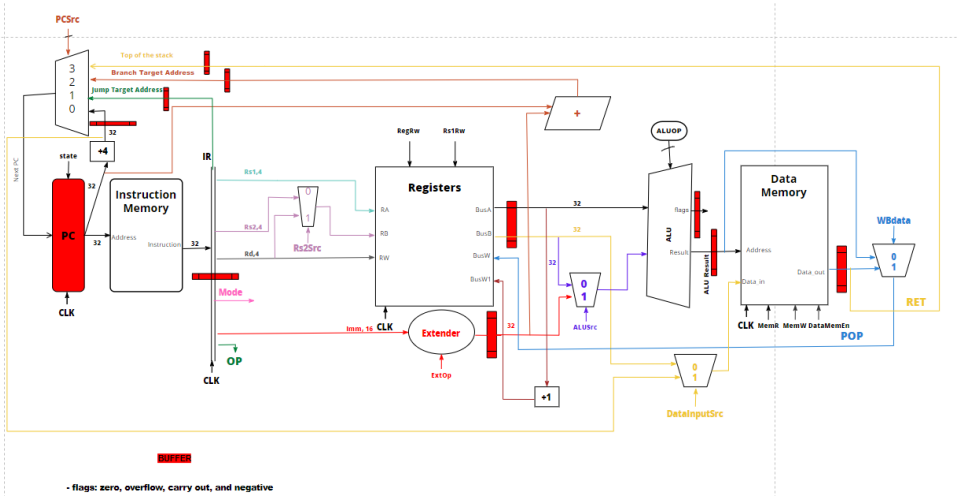


Figure 2-13: Datapath

The previous figure shows the Datapath, which consists of CPU stages:

2.6.1. Instruction Fetch (IF)

During the process of fetching instructions, a program counter (PC) is used to provide the address to which the instruction memory should be accessed based on the content of the instruction memory. When the current state is IF (Instruction Fetch), the PC address is updated at the positive clock edge. As a result of the instruction being executed, it can be updated to one of four different values:

- When the PC address is 00, instructions are executed sequentially. By adding 1 to its current value, the program counter (PC) is incremented to the next instruction address. The program continues to the following instruction in memory as a result.
- When the PC address is 01, this indicates that the instruction is a jump instruction. Program counter (PC) is updated with the jump target address as a result. In general, JMP and CALL instructions are used for handling exceptions and subroutines.
- When the PC address is 10, then it is an indication that the instruction is a branch instruction. Whenever the branch condition is true, the program counter is updated with the branch target address. The branch instructions allow for decisions and loops to be made in the execution of the code based on the conditional alteration of the program flow.
- When the PC address is 11, the instruction is a RET jump. Consequently, the program counter (PC) is updated with the top of the stack.

Finally, the instruction fetching mechanism allows the processor to correctly fetch and execute instructions based on the control flow requirements of the program by updating the program counter to these different values. By implementing branching, looping, and sequential execution, this program ensures that it runs in a controlled, structured manner.

2.6.2. Instruction Decode (ID)

The instruction decode stage decodes fetched instructions and extracts their components. Usually, the instruction is decoded into its OP code, first source register (Rs1), second source register (Rs2), destination register (Rd), mode, immediate values, jump offset, and other related fields.

Following the decoding of an instruction, the Control Unit generates control signals based on the type and function of the instruction. The decisions made in following processor stages are based on these control signals.

Reading the registers from the Register File is an important step in the instruction decode process. The decoded instruction is used to identify the source registers (Rs1 and Rs2), and the register file is accessed to retrieve the values associated with each. The operands required for following stages are made available through the use of this base register values retrieval.

The instruction decode step is finished once the register values and immediate value are obtained. After that, the immediate value and register values are transmitted to the execution stage for additional processing. These values are used as inputs when the instruction requires the execution of arithmetic, logic, or control operations.

2.6.3. Execute (EX)

The CPU performs the operation specified by the instruction, using the ALU (Arithmetic Logic Unit). The ALU executes computations or operations using input values from previous stages, such as register values and immediate values, during the execution stage. Arithmetic operations like addition and subtraction may be included, as well as logical operations like AND.

The resulting value or status is sent to the following stage once the ALU has finished its work. The outcome of the operation and a flag indicating the operation's status (zero, negative, overflow, or carry) are possible outputs from the execution stage. Control units receive the generated flags to determine whether to take a branch.

2.6.4. Memory (MEM)

The control signals from previous stages of a processor determine whether a read operation, write operation, or no operation (also called a "no-op") will take place at the memory stage. If the instruction involves data transfer to or from memory, the CPU accesses the data in memory. A summary of each possibility is provided below:

- Read Operation (when MemR = 1 & MemW = 0): The processor must retrieve data from memory if the control signals point to a read operation. The instruction being executed usually provides the memory address. The control signals cause the memory system's circuits and signals to become active, enabling the reading of the data kept at the designated memory address. The data that has been retrieved is subsequently sent to the processor's next stages for additional processing. NOTE: Used in RET, POP, LW, and LW.POI instructions.

- Write Operation (when MemR = 0 & MemW = 1): The processor must store data in memory when the control signals indicate a write operation. As with a read operation, the instruction being executed usually provides the memory address. The control signals allow the memory system's required circuits and signals to write the data to the assigned memory address. NOTE: Used in SW, PUSH, and CALL instructions.
- No Operation "No-Op" (when MemR = 0 & MemW = 0): The control signals from previous stages may sometimes suggest that there is no need for a memory operation. This may occur when an instruction such as an arithmetic or logic instruction that depends only on register values does not require accessing or changing data stored in memory. In these situations, the processor can move on to the next stage without performing any memory-related operations because the control signals essentially bypass the memory stage.
- Unused case (when MemR = 1 & MemW = 1): cannot read and write from/to memory at the same time.

The instruction being executed and the particular design and implementation of the processor's control unit are what usually determine whether a read, write, or no-op occurs. In order to ensure correct data and instruction flow during program execution, the control signals from previous stages assist in coordinating the interactions between the processor and the memory system.

DataMemEn: The processor must Read/Write data from/to data memory when this control signal equal 1. NOTE: used in LW, LW.POI, and SW instructions. Else the processor must Read/Write data from/to stack when this control signal equal 0. NOTE: Used in Push & POP instructions

2.6.5. Write Back (WB)

One of the most important phases in a processor's instruction execution process is the write-back stage, or WB stage. When the CPU has completed the necessary calculations or has retrieved data from memory, it writes the execution results back to the register file. Data bus and register file are the key components.

Depending on the type of instruction being executed, specific information needs to be written back according to the value of WBdata:

ALU Results (when WBdata = 0): The Arithmetic Logic Unit, or ALU, is used in many arithmetic and logical instructions to perform operations like AND, addition, and subtraction. Usually, the result of these calculations is written back to a register file.

Memory Data (when WBdata = 1): Information is retrieved from memory and subsequently written back to the register file in load instructions (such as LW, LW.POI). Memory data could consist of the value stored at a particular memory address or a portion of it.

2.7. Control Unit

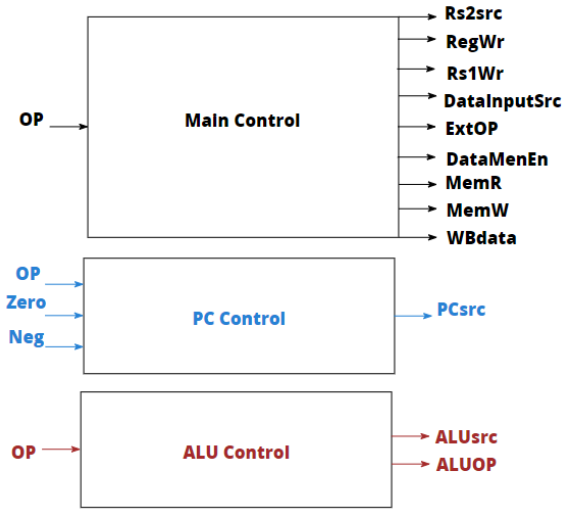


Figure 2-14: Main, ALU, and PC Control

2.7.1. Main Control Signals

The following table show the main control signals.

Table 2-1: Main Control Signals

Signal	Effect when '0'	Effect when '1'
Rs2Src	source register = Rs2	source register = Rd
DataInputSrc	Data_in is the value of register Rs2 that appears on BusB	Data_in is the value of PC + 1
ALUSrc	Second ALU operand is the value of register Rs2 that appears on BusB	Second ALU operand is the value of the extended 16-bit immediate
ExtOp	16-bit immediate is zero-extended	16-bit immediate is sign-extended
WBdata	BusW = ALU result and BusW1 = Reg(Rs1)	BusW = Data_out from Memory
MemRd	Data memory is NOT read	Data memory is read Data_out ← Memory[address]
MemWr	Data Memory is NOT written	Data memory is written Memory[address] ← Data_in
RegWr	No register is written	Destination register is written with the data on BusW (with/without) BusW1
Rs1Wr	No Rs1 register is written	Rs1 register is written with the data BusW1
DataMemEn	Read/Write from/to Stack	Read/Write from/to Data Memory

2.7.2. Truth Tables

The control signals were derived from the Datapath after the Datapath had been built. Tables below show how each control signal changes depending on the instruction type, function, and flags.

Table 2-2: Main Control Truth Table

OP	Rs2Src	RegWr	DataInputSrc	ALUSrc	ExtOp	WBdata	MemR	MemW	Rs1Wr	DataMemEn
R-type	0 = Rs2	1	X	0 = BusB	X	0 = ALU	0	0	0	X
ADDI	X	1	X	1 = Imm	1 = sign	0 = ALU	0	0	0	X
ANDI	X	1	X	1 = Imm	0 = zero	0 = ALU	0	0	0	X
LW	X	1	X	1 = Imm	1 = sign	1 = Mem	1	0	0	1
LW.POI	X	1	X	1 = Imm	1 = sign	1 = Mem	1	0	1	1
SW	X	0	0 = BusB	1 = Imm	1 = sign	X	0	1	X	1
BGT	1 = Rd	0	X	0 = BusB	X	X	0	0	X	X
BLT	1 = Rd	0	X	0 = BusB	X	X	0	0	X	X
BEQ	1 = Rd	0	X	0 = BusB	X	X	0	0	X	X
BNE	1 = Rd	0	X	0 = BusB	X	X	0	0	X	X
JMP	X	0	X	X	X	X	0	0	X	X
CALL	X	0	1 = PC + 1	X	X	X	0	1	X	0
RET	X	0	X	X	X	1 = Mem	1	0	X	0
PUSH	1 = Rd	0	0 = BusB	X	X	X	0	1	X	0
POP	X	1	X	X	X	1 = Mem	1	0	0	0

NOTE: X is a don’t care (can be 0 or 1), used to minimize logic.

Table 2-3: PC Control Truth Table

OP	Zero flag	Neg Flag	PcSrc
R-type	X	X	0 = PC+1
J-type	X	X	1 = jump target address
BEQ	0	X	0 = PC+1
BEQ	1	X	2 = Branch Target Address
BNE	0	X	2 = Branch Target Address
BNE	1	X	0 = PC+1

BGT	X	0	2 = Branch Target Address
BGT	X	1	0 = PC+1
BLT	X	0	0 = PC+1
BLT	X	1	2 = Branch Target Address
RET	X	X	3 = Top of the stack
Other than Jump or Branch	X	X	0 = PC + 1

Table 2-4: ALU Control Truth Table

OP	ALUOp	2-bit Coding
R-type	AND	00
R-type	ADD	01
R-type	SUB	10
ANDI	AND	00
ADDI	ADD	01
LW	ADD	01
LW.POI	ADD	01
SW	ADD	01
BGT	SUB	10
BLT	SUB	10
BEQ	SUB	10
BNE	SUB	10
JUMP	X	X
CALL	X	X
RET	X	X
PUSH	X	X
POP	X	X

2.7.3. Boolean equations

2.7.3.1. Logic Equations for Main Control Signals

- Rs2Src = (R-type)`
- RegWr = R-type + ADDI + ANDI + LW + LW.POI + POP
- Rs1Wr = LW.POI
- DataInputSrc = CALL
- ALUSrc = ADDI + ANDI + LW + LW.POI + SW
- ExtOp = (ANDI)`
- WBdata = (R-type + ADDI + ANDI)`
- MemRd = (LW + LW.POI + RET + POP)
- MemWr = SW + CALL + Push
- DataMemEn = LW + LW.POI + SW

2.7.3.2. PC Control Logic

- if (Op == J) PCSrc = 1;
- else if ((Op == BEQ && Zero == 1) || (Op == BNE && Zero == 0) ||
  (Op == BGT && Neg == 0) || (Op == BLT && Neg == 1)) PCSrc = 2;
- else if (Op == RET) PCSrc = 3;
- else PCSrc = 0;

2.8. Temporary registers

In a multi-cycle processor - where instructions are executed in stages - the usage of temporary registers is essential. These registers act as intermediary storage units, allowing data to move smoothly between stages and guaranteeing that instructions are carried out as efficiently as possible.

2.8.1. Instruction Register (IR)

In the multi-cycle processor architecture, the Instruction Register (IR) plays a pivotal role in executing instructions and facilitating data flow across various stages. It serves as the repository for the instruction currently in execution, storing critical information such as source registers (RS1, RS2), destination register (RD), and immediate values. The IR's centralized storage enables easy access to instruction data throughout the processor's different stages.

2.8.2. Data Registers for Read RS1 and RS2

To support the execution of instructions, the processor incorporates Data Registers dedicated to reading values from source registers (RS1 and RS2). These registers store essential data for stages like execution and memory access, ensuring the processor can effectively operate on the required operands.

2.8.3. ALU Result and Flags Registers

Temporary ALU Result and Flags Registers are instrumental in arithmetic or logical operations executed by the Arithmetic Logic Unit (ALU). These registers not only capture the results of operations but also store flags such as the zero flag, carry flag, overflow flag, and negative flag. The flags provide crucial information about the status of the operation, frequently utilized in subsequent stages for branching or condition checking.

2.8.4. Memory Data Register

The Memory Data Register serves as a temporary repository for data fetched from memory during memory access operations. It holds the value retrieved from memory, contributing to the write-back stage or supporting further processing in subsequent instructions.

2.8.5. Extender Value Register

In various stages of instruction execution, the Extender Value Register retains values obtained from the extender unit. It preserves extended values, including sign-extension or zero-extension, ensuring compatibility with the diverse requirements of different instruction stages.

3. Simulation and Testing

To validate the functionality of our multi-cycle processor, we conducted test cases focusing on scenarios where multiple instructions are executed consecutively. These tests go beyond unit testing and examine the processor's capability to handle a sequence of instructions in a real-world scenario.

In order to simulate the execution of a program, we kept the proper binary values of the instructions in memory for each test instance. A variety of operations on integer variables are covered by these test cases, such as additions, comparisons, and function calls. The test program keeps the final findings in memory and computes results based on variable values.

We want to evaluate the processor's capacity to accurately carry out a sequence of instructions in a multi-cycle scenario by creating these test cases. This method offers a comprehensive assessment of the processor's performance, guaranteeing that it will not only operate correctly at the level of individual instructions but also retain integrity when processing a series of instructions in an actual application. These test cases' outcomes were examined to verify that the multi-cycle processor handled a variety of instruction sequences in the manner that was anticipated.

3.1. Extender

Extender was tested by generating random inputs and observing the outputs, as can be shown in the waveform in the figure bellow:

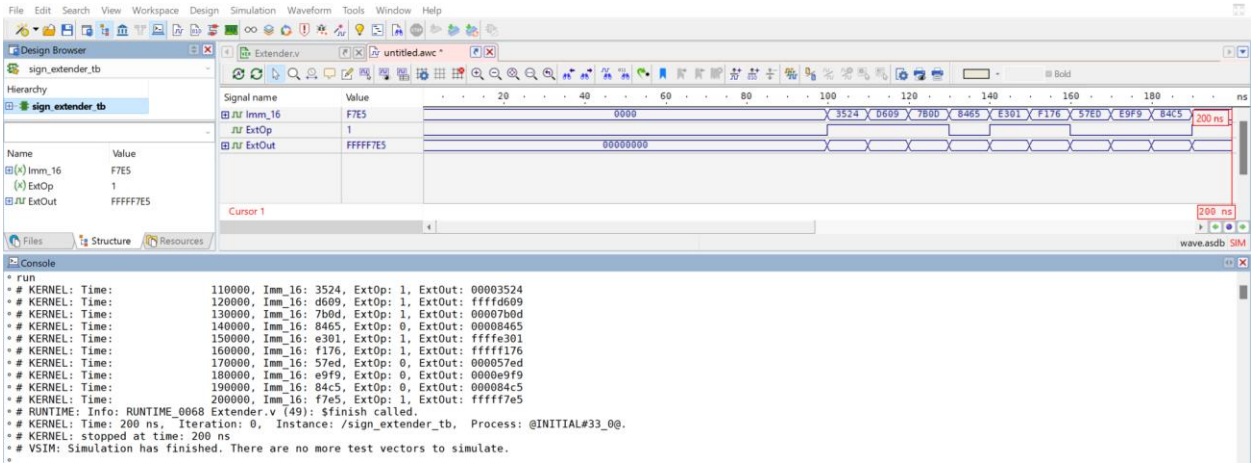


Figure 3-3-1: Extender Waveform

3.2. Data memory

Data memory was tested by generating random inputs and observing the outputs, as can be shown in the waveform in the figure bellow:

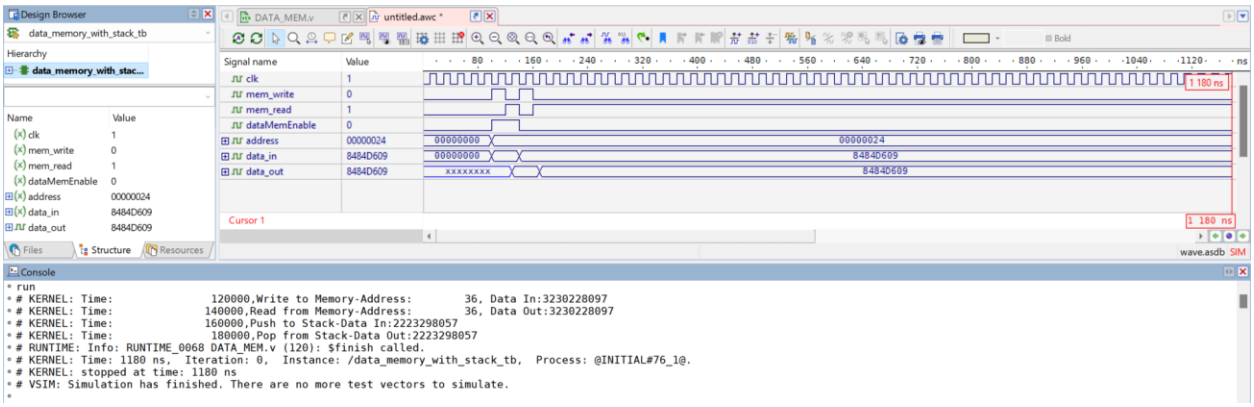


Figure 2-3-2: Data Memory Waveform

3.3. Adder

Adder was tested by generating random inputs and observing the outputs, as can be shown in the waveform in the figure bellow:

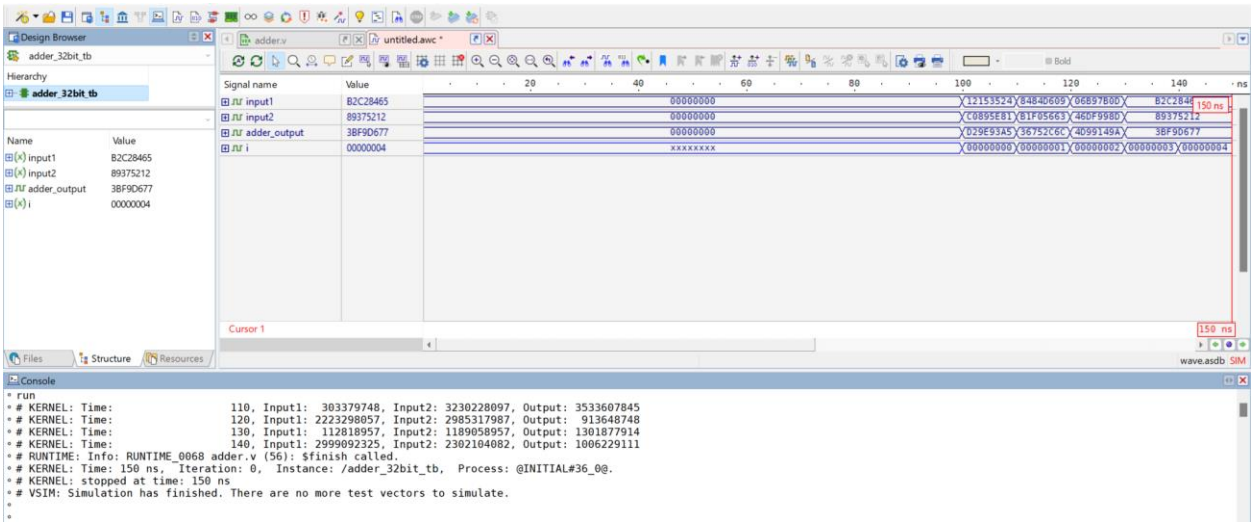


Figure 2-3-3: Adder Waveform

3.4. Mux 2x1

Mux2x1 was tested by generating inputs and observing the outputs, as can be shown in the waveform in the figure bellow:

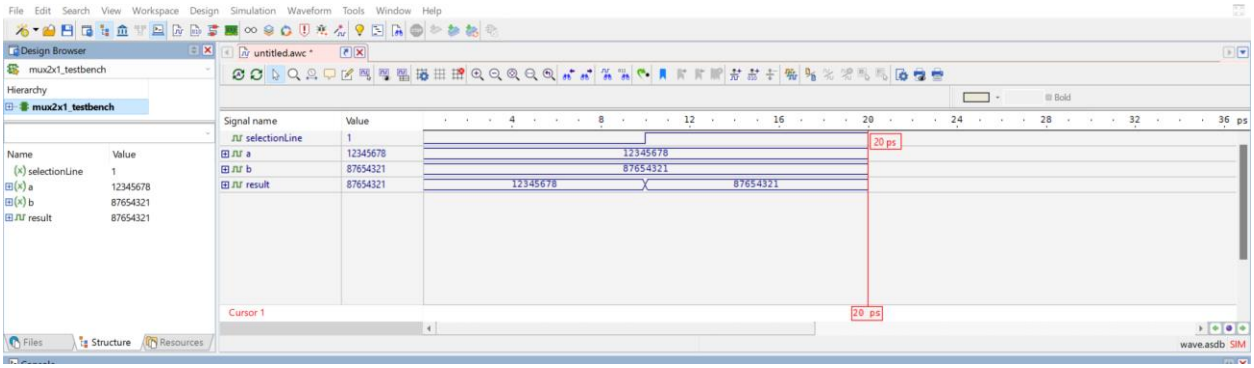


Figure 2-3-4:Mux2x1 Waveform

3.5. Mux 4x1

Mux4x1 was tested by generating inputs and observing the outputs, as can be shown in the waveform in the figure bellow:

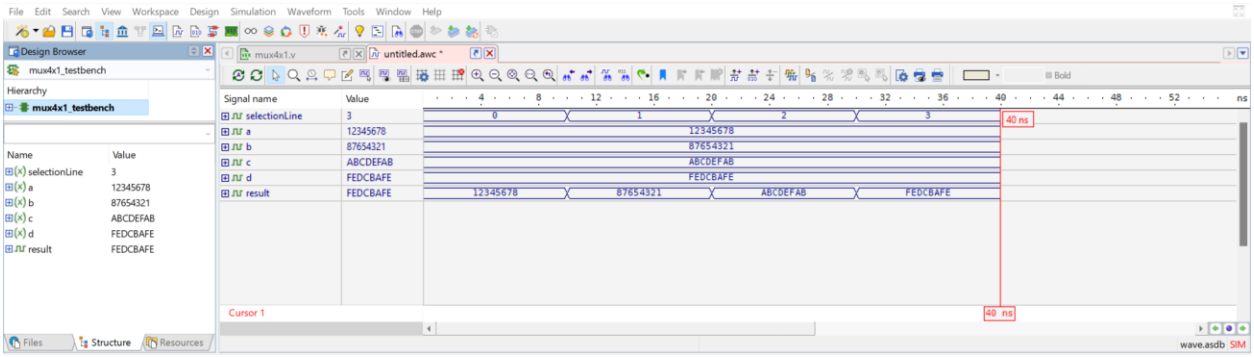


Figure 2-3-5: Mux4x1 Waveform

3.6. D - Flip Flop

D - Flip Flop was tested by generating random inputs and observing the outputs, as can be shown in the waveform in the figure bellow:

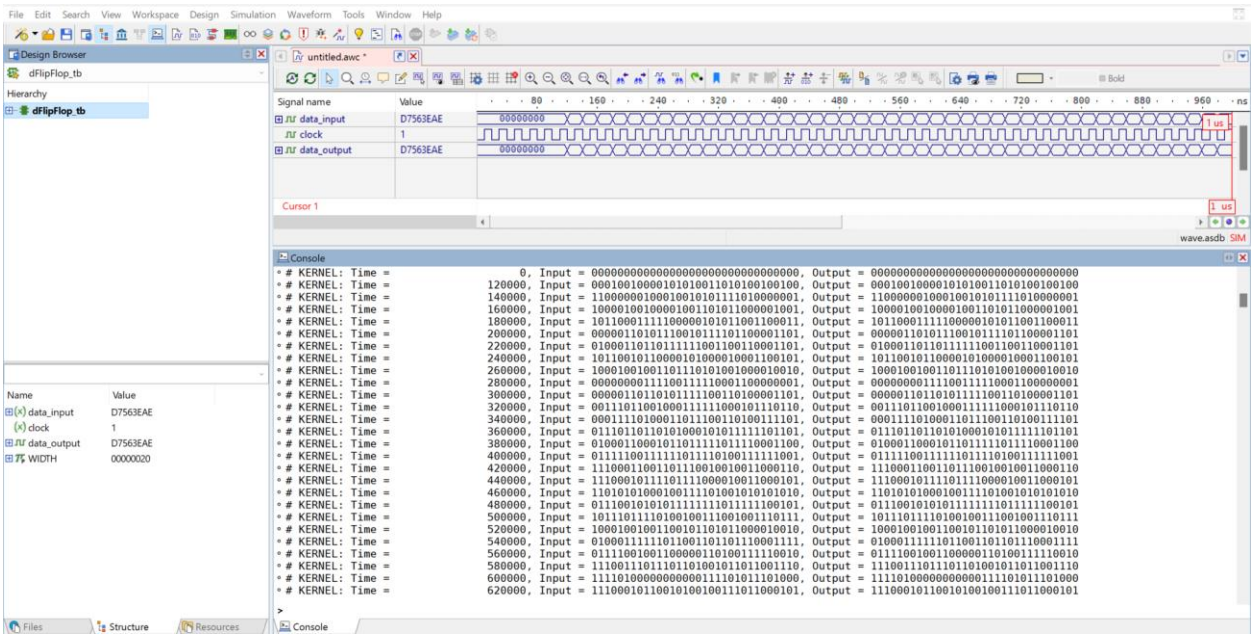


Figure 2-3-6: DFlipFlop Waveform

3.7. PC

PC was tested by generating random inputs and observing the outputs, as can be shown in the waveform in the figure bellow:

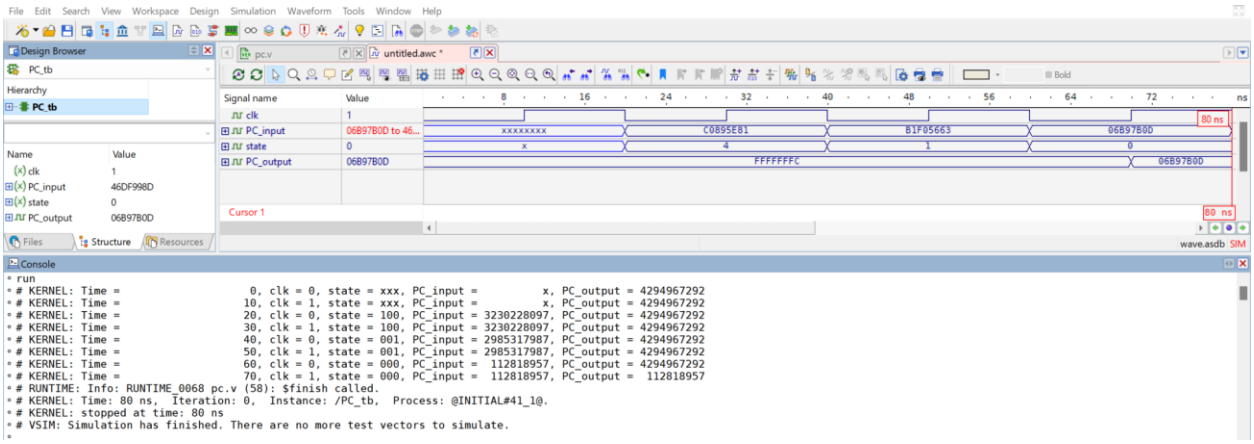


Figure 2-3-7: PC Waveform

3.8. Instruction Memory

Instruction Memory was tested by generating random inputs and observing the outputs, as can be shown in the waveform in the figure bellow:

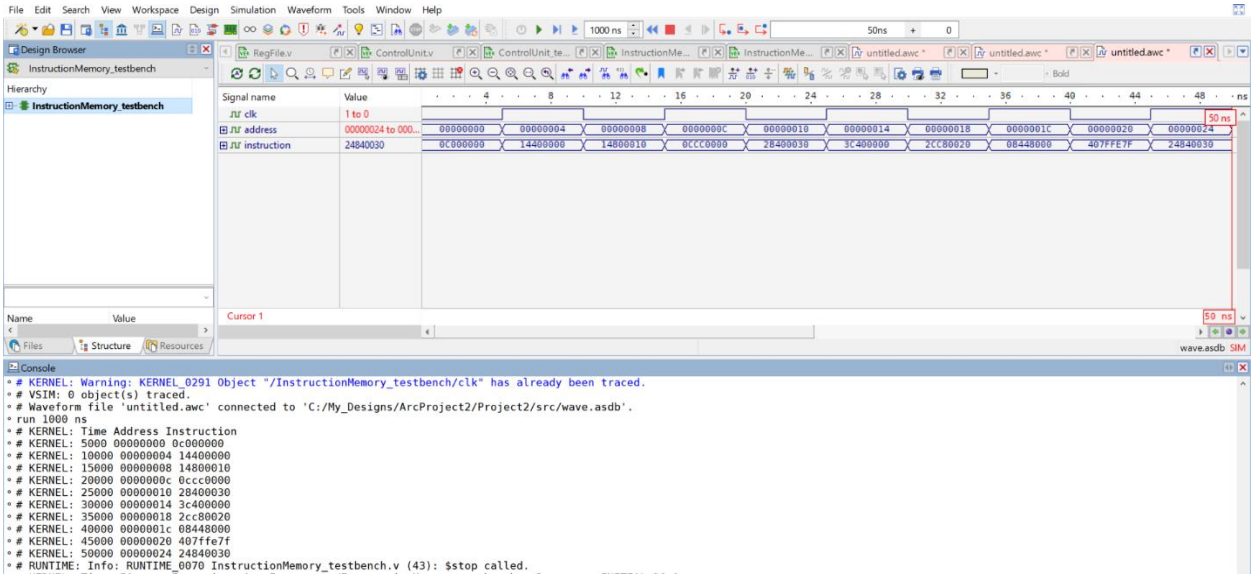


Figure 2-3-8:Instruction Memory Waveform

3.9. IR

IR was tested by generating random inputs and observing the outputs, as can be shown in the waveform in the figure bellow:

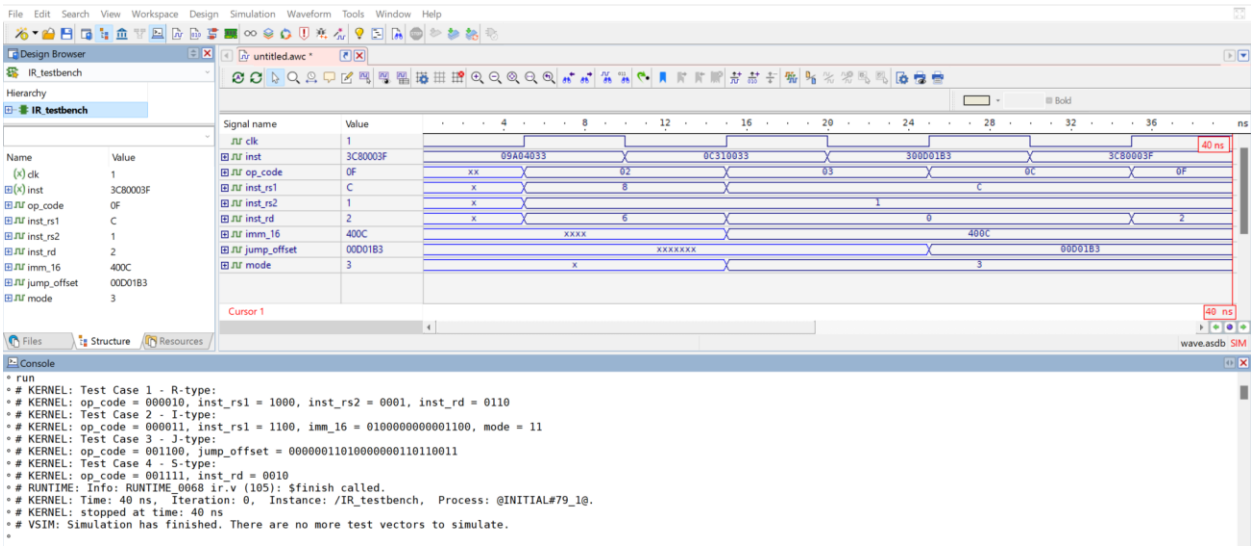


Figure 2-3-9: IR Waveform

3.10. Register File

Register File was tested by generating random inputs and observing the outputs, as can be shown in the waveform in the figure bellow:

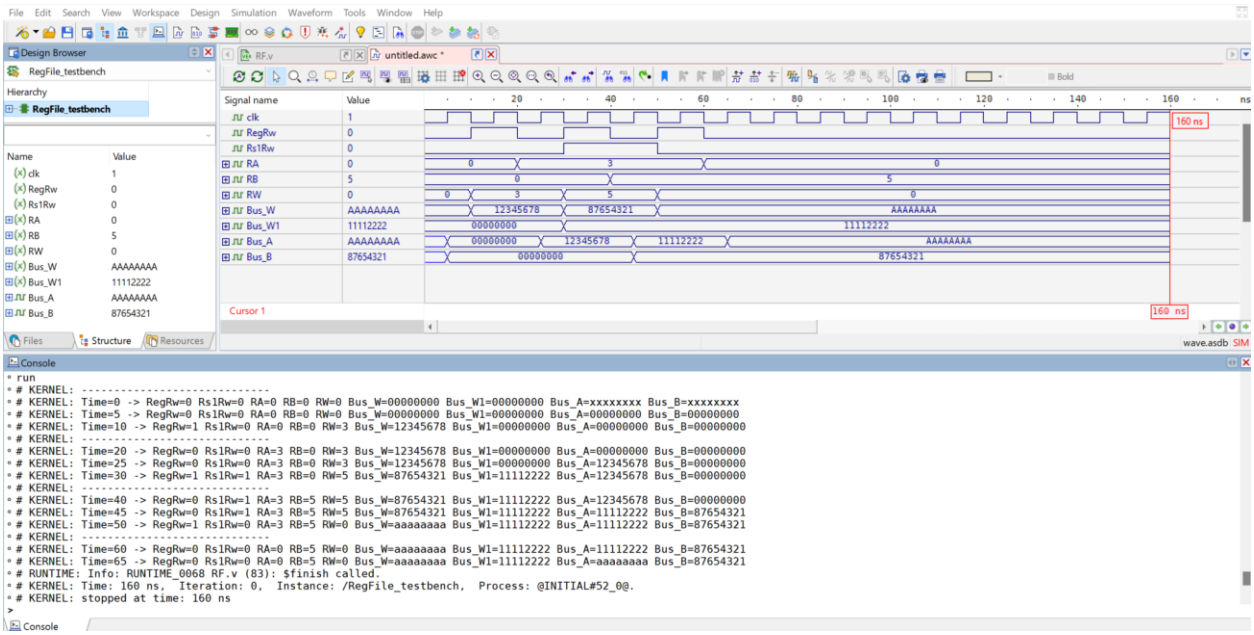
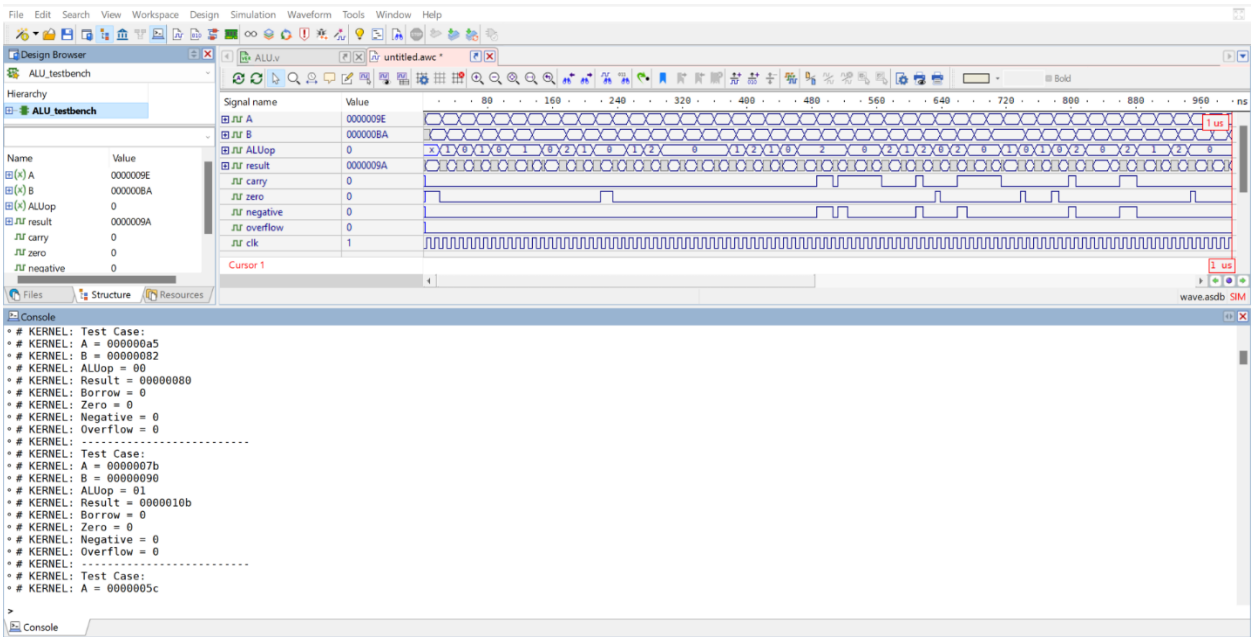


Figure 2-3-10: Register File Waveform

3.11. ALU

Alu was tested by generating random inputs and observing the outputs, as can be shown in the waveform in the figure bellow:



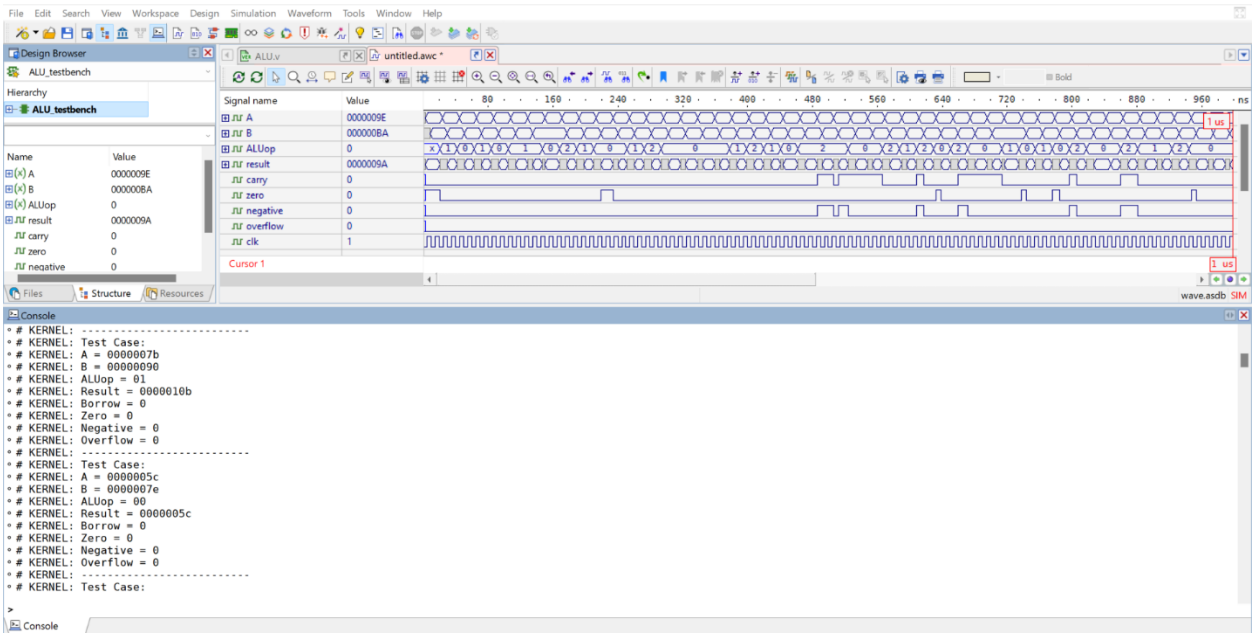


Figure 3-11: ALU Waveform

3.12. Control Unit

Control Unit was tested by generating random inputs and observing the outputs, as can be shown in the waveform in the figure bellow:

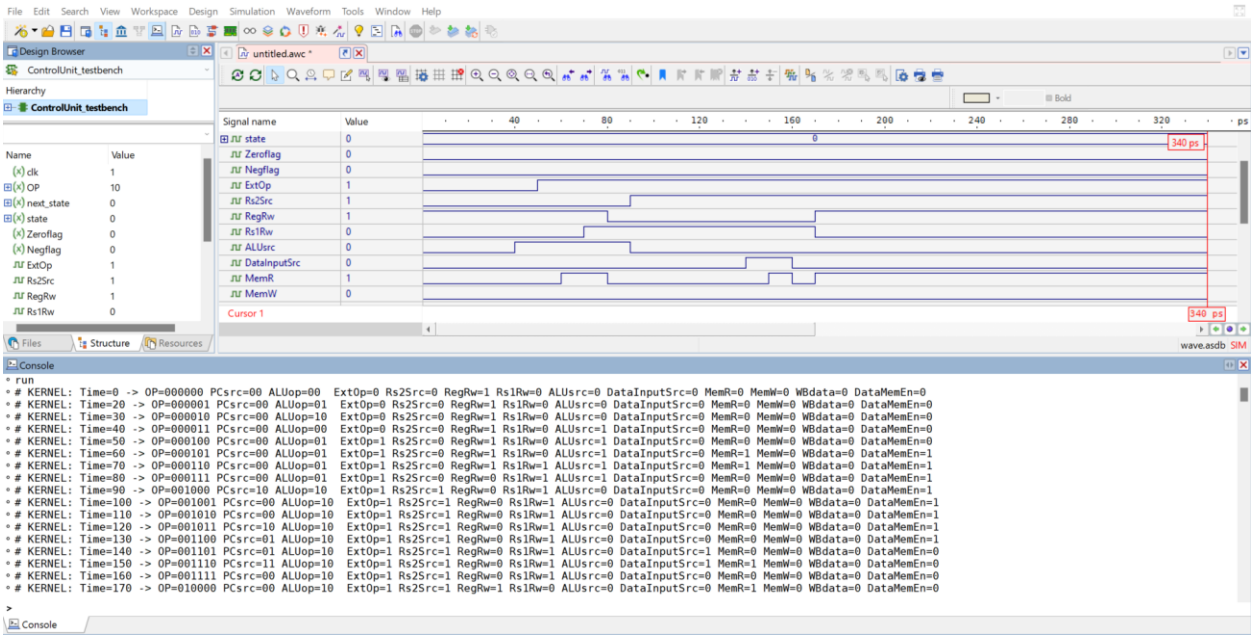


Figure 3-12: Control Unit Waveform

3.13. Concatenation

Concatenation was tested by generating random inputs and observing the outputs, as can be shown in the waveform in the figure bellow:

Memory allocation was performed for three variables, with the initial two integers, 2 and 1, stored at memory addresses 0x00000000 and 0x00000004, respectively. In adherence to the word addressable design of our computer, the third integer, representing the result of the code, is stored at memory position 0x00000008.

- memory [0] = 0x02
- memory [4] = 0x04
- memory [8] = 0x00

The table below displays the tested sequence of instructions (test code), including the instruction address, as well as the binary and hexadecimal representations of each instruction.

Table 3-1: Testbench Instructions

Instruction Address	Assembly Instruction	Binary Representation	Hex Representation
0	ANDI $t0, $t0, 0	00001100000000000000000000000000	0x0C000000
4	LW $t1, 0($t0)	00010100010000000000000000000000	0x14200000
8	LW $t2, 4($t0)	00010100100000000000000000010000	0x15200010
12	ANDI $t3, $t3, 0	00001100110011000000000000000000	0x06600000
16	BEQ $t1, $t0, Label_1	00101000010000000000000000110000	0x28400030
20	PUSH $t1	00111100010000000000000000000000	0x3C200000
24	BNE $t3, $t2, Label_2	00101100110010000000000000100000	0x36A00020
28: Label_1	SUB $t1, $t1, $t2	00000100001000100100000000000000	0x08248000
32: Label_2	POP $t4	01000000011111111111111001111111	0x407FFFFF
36	BLT $t2, $t1, Label_3	00100100100001000000000000110000	0x12480030
40	SW $t4, 8($t0)	00011101000000000000000000100000	0x1D000020
44: End	J End	00110000000000000000000000101100	0x3000002C
48: Label_3	AND $t3, $t1, $t2	00000000110001001000000000000000	0x00048000

The result of CPU test bench:

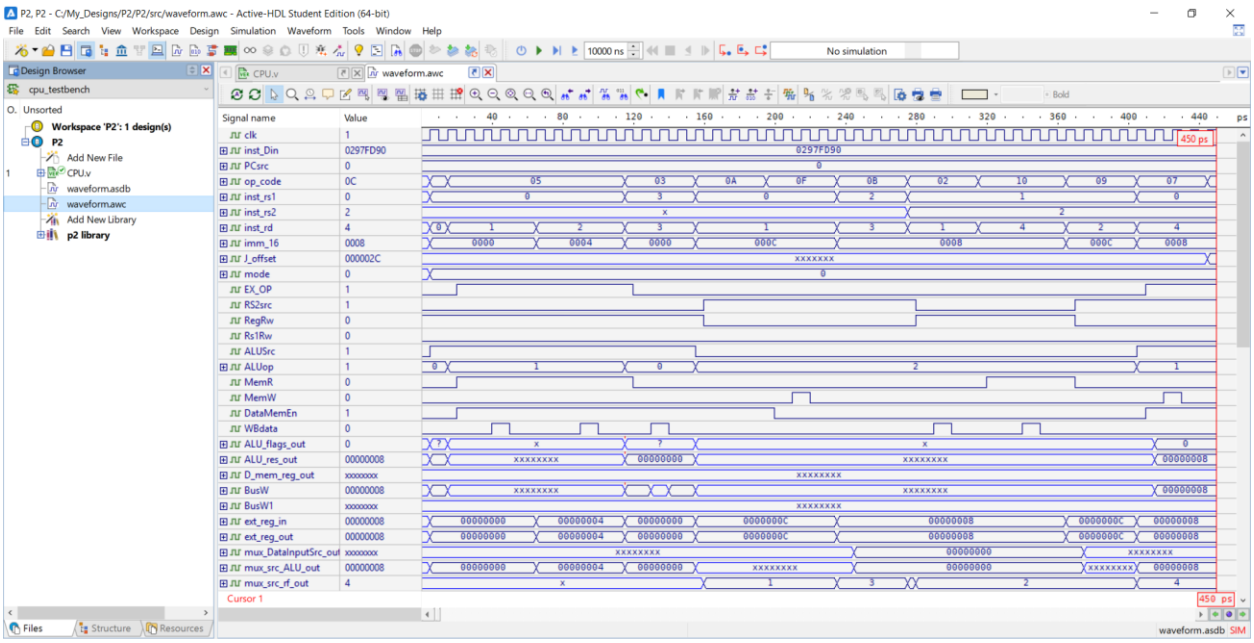
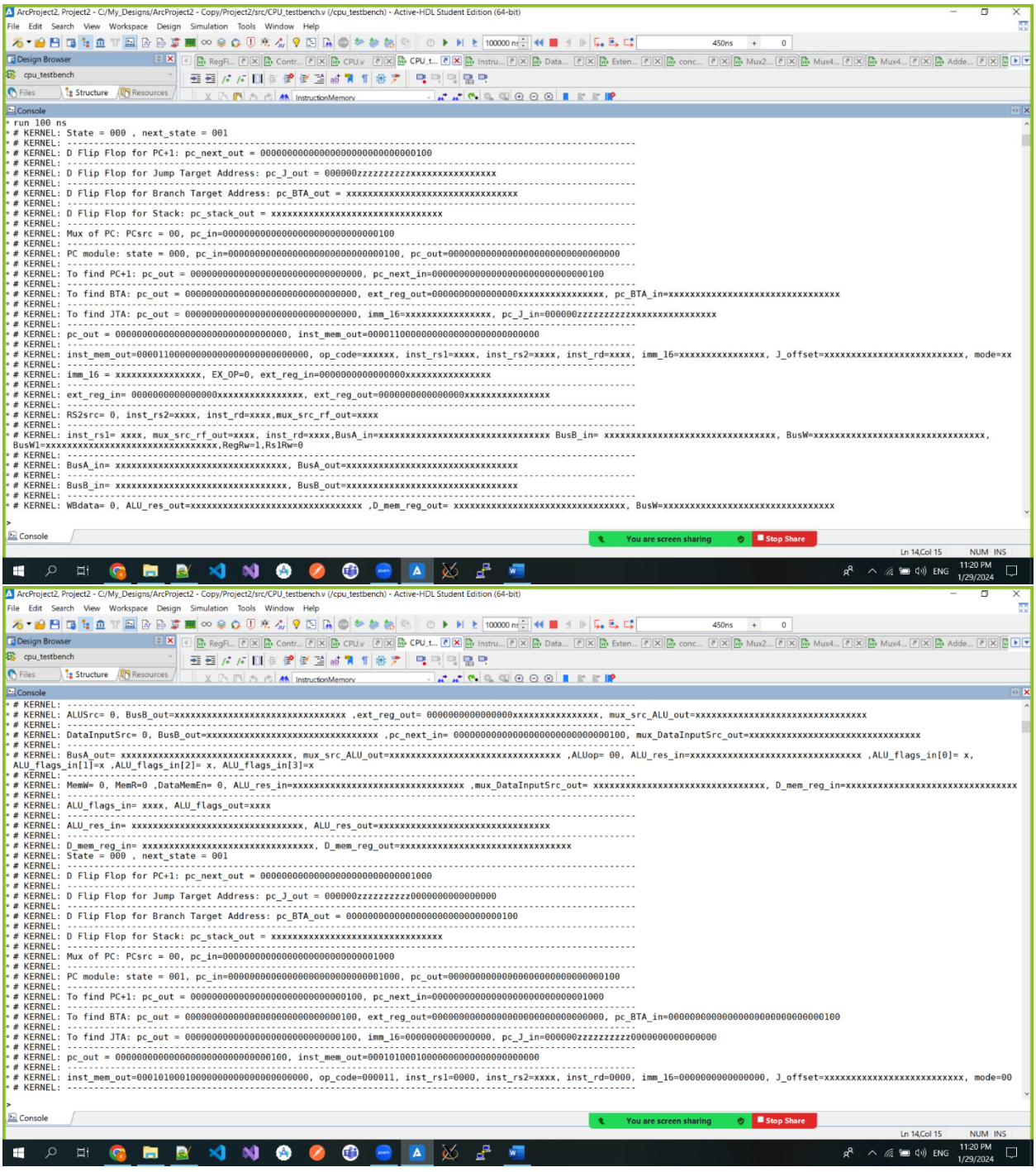
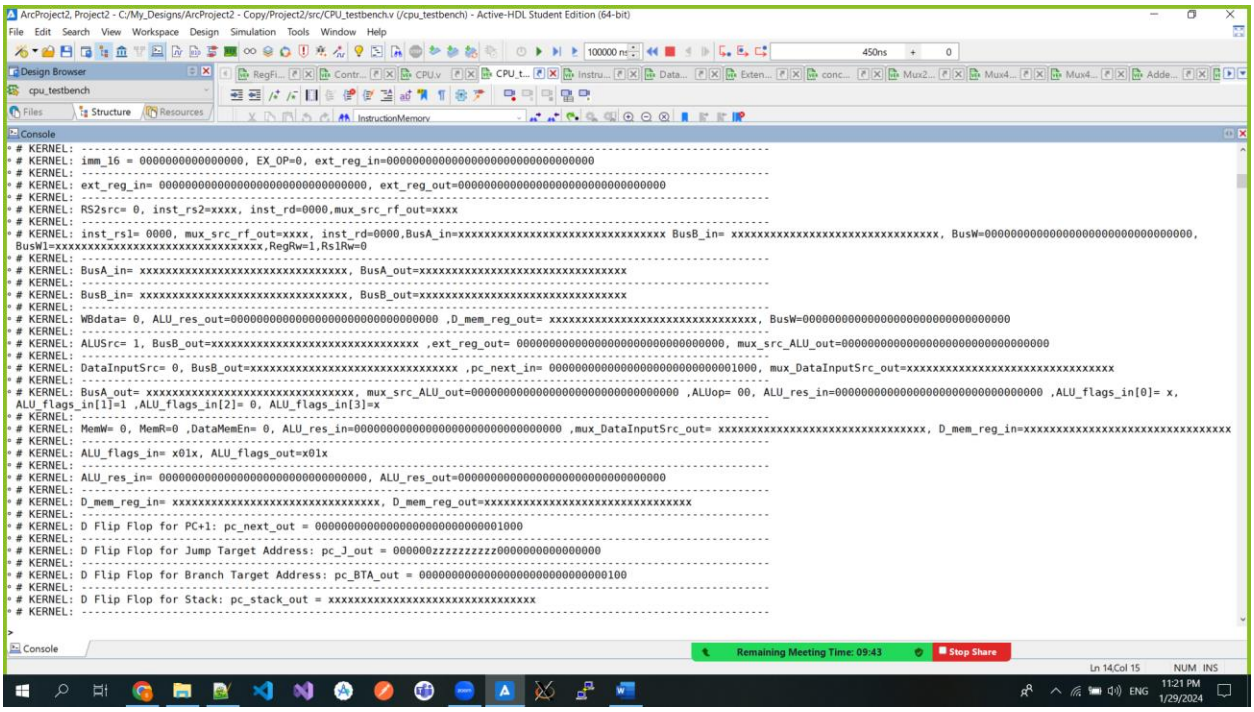
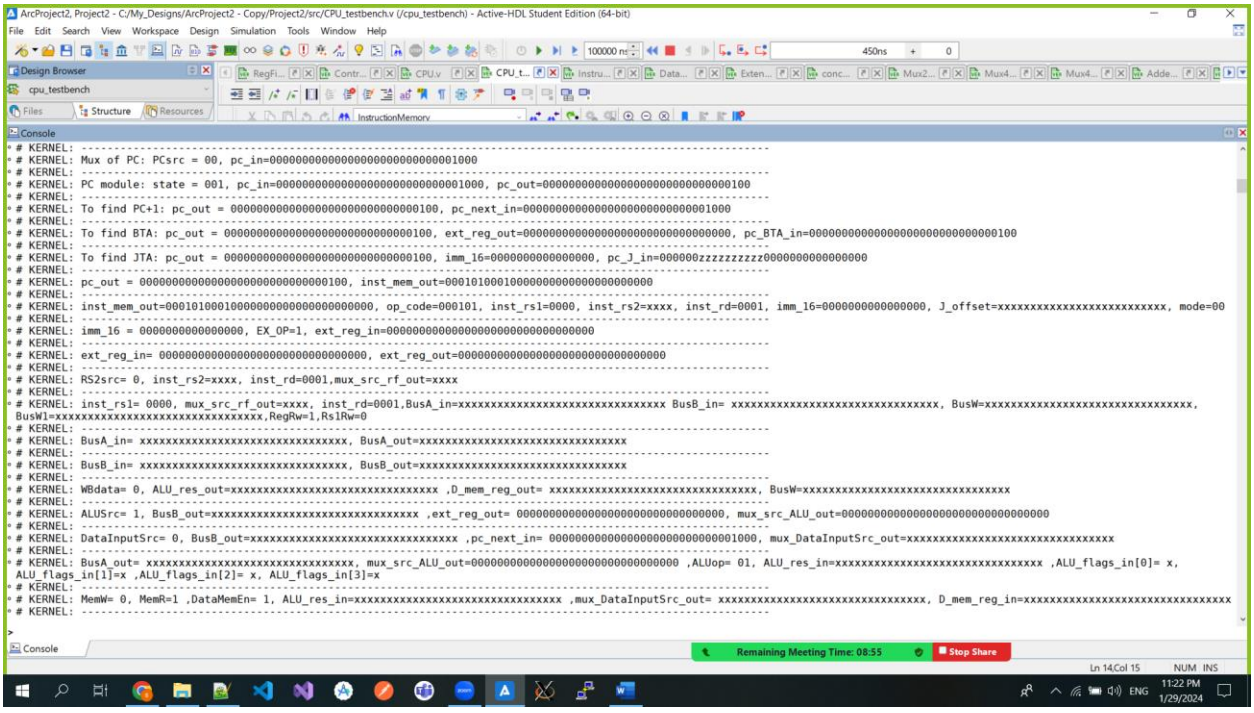
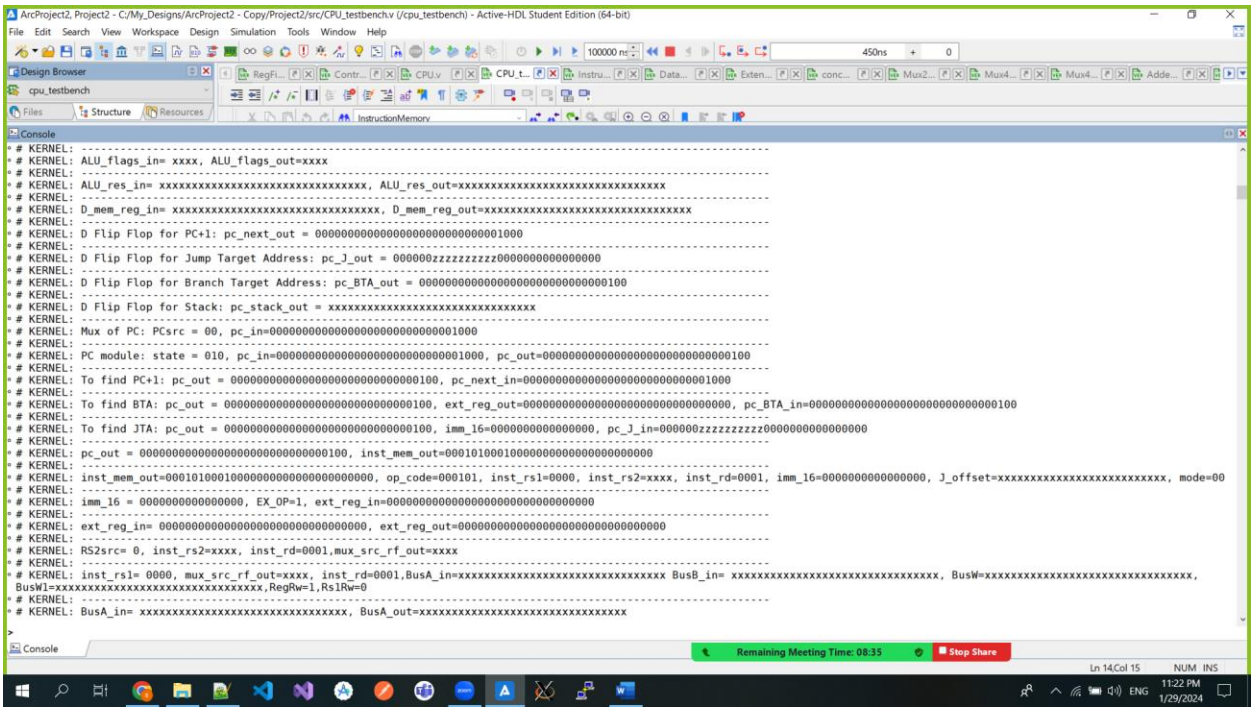


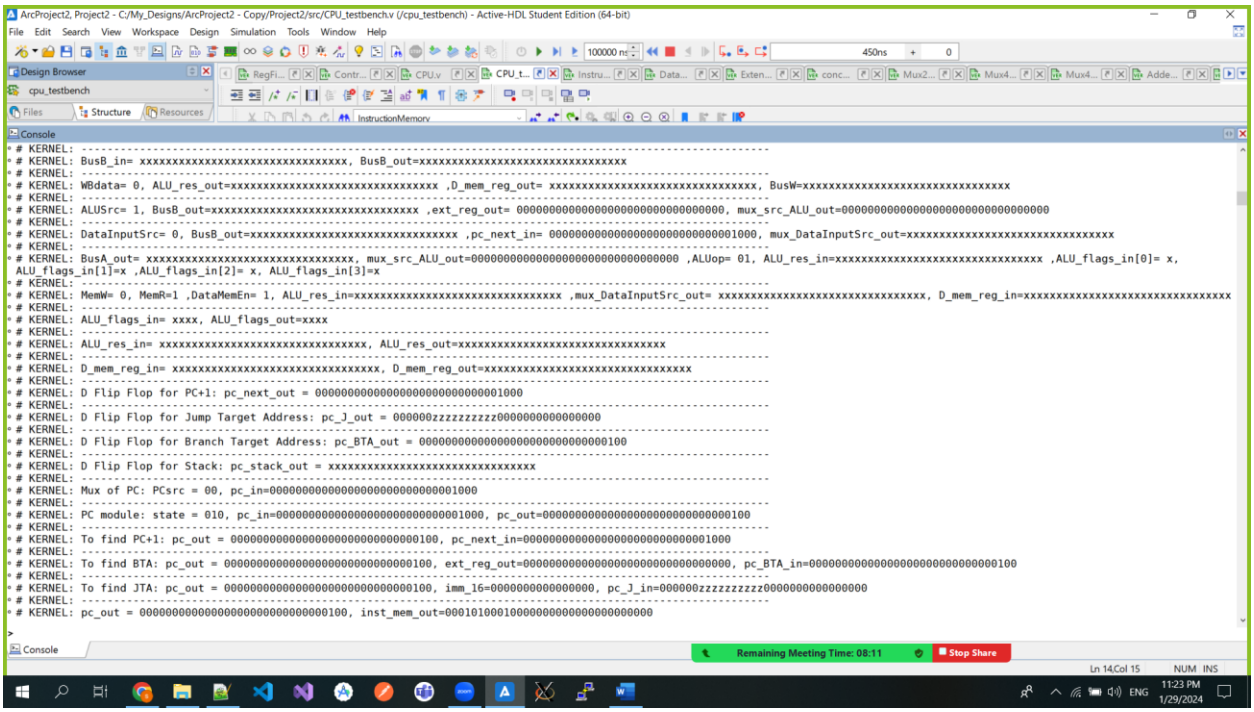
Figure 3-13: CPU testbench

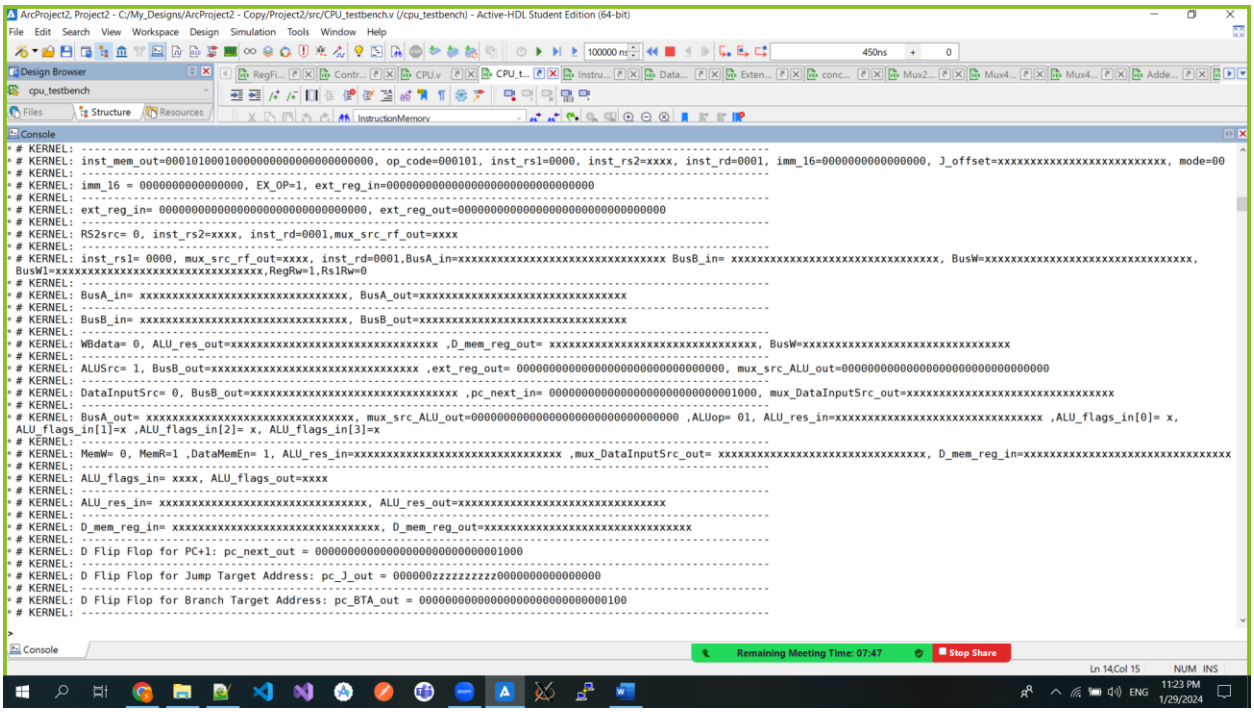

ArcProject2, Project2 - C:/My_Designs/ArcProject2 - Copy/Project2/src/CPU_testbench.v (/cpu_testbench) - Active-HDL Student Edition (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
cpu_testbench
Files Structure Resources
Console
run 100 ns
# KERNEL: State = 000 , next_state = 001
# KERNEL: D Flip Flop for PC+1: pc_next_out = 00000000000000000000000000000100
# KERNEL: D Flip Flop for Jump Target Address: pc_J_out = 000000zzzzzzzzzzxxxxxxxxxxxxxxxx
# KERNEL: D Flip Flop for Branch Target Address: pc_BTA_out = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: D Flip Flop for Stack: pc_stack_out = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: Mux of PC: PCsrc = 00, pc_in=00000000000000000000000000000100
# KERNEL: PC module: state = 000, pc_in=00000000000000000000000000000100, pc_out=00000000000000000000000000000000
# KERNEL: To find PC+1: pc_out = 00000000000000000000000000000000, pc_next_in=00000000000000000000000000000100
# KERNEL: To find BTA: pc_out = 00000000000000000000000000000000, ext_reg_out=0000000000000000xxxxxxxxxxxxxxxx, pc_BTA_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: To find JTA: pc_out = 00000000000000000000000000000000, imm_16=xxxxxxxxxxxxxxxx, pc_J_in=000000zzzzzzzzzzxxxxxxxxxxxxxxxx
# KERNEL: pc_out = 00000000000000000000000000000000, inst_mem_out=00001100000000000000000000000000
# KERNEL: inst_mem_out=00001100000000000000000000000000, op_code=xxxxxx, inst_rs1=xxxx, inst_rs2=xxxx, inst_rd=xxxx, imm_16=xxxxxxxxxxxxxxxx, J_offset=xxxxxxxxxxxxxxxxxxxxxxxxxx, mode=xx
# KERNEL: imm_16 = xxxxxxxxxxxxxxxx, EX_OP=0, ext_reg_in=0000000000000000xxxxxxxxxxxxxxxx
# KERNEL: ext_reg_in= 0000000000000000xxxxxxxxxxxxxxxx, ext_reg_out=0000000000000000xxxxxxxxxxxxxxxx
# KERNEL: RS2src= 0, inst_rs2=xxxx, inst_rd=xxxx,mux_src_rf_out=xxxx
# KERNEL: inst_rs1= xxxx, mux_src_rf_out=xxxx, inst_rd=xxxx,BusA_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,
BusW1=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,RegRw=1,Rs1Rw=0
# KERNEL: BusA_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusA_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: WBdata= 0, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,D_mem_reg_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
You are screen sharing
Stop Share
Ln 14,Col 15
NUM INS
11:20 PM
1/29/2024
ArcProject2, Project2 - C:/My_Designs/ArcProject2 - Copy/Project2/src/CPU_testbench.v (/cpu_testbench) - Active-HDL Student Edition (64-bit)
# KERNEL: ALUSrc= 0, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ext_reg_out= 0000000000000000xxxxxxxxxxxxxxxx, mux_src_ALU_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: DataInputSrc= 0, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,pc_next_in= 00000000000000000000000000000100, mux_DataInputSrc_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: BusA_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, mux_src_ALU_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ALUop= 00, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ALU_flags_in[0]= x,
ALU_flags_in[1]=x ,ALU_flags_in[2]= x, ALU_flags_in[3]=x
# KERNEL: MemW= 0, MemR=0 ,DataMemEn= 0, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,mux_DataInputSrc_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ALU_flags_in= xxxx, ALU_flags_out=xxxx
# KERNEL: ALU_res_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: D_mem_reg_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: State = 000 , next_state = 001
# KERNEL: D Flip Flop for PC+1: pc_next_out = 00000000000000000000000000001000
# KERNEL: D Flip Flop for Jump Target Address: pc_J_out = 000000zzzzzzzzzz0000000000000000
# KERNEL: D Flip Flop for Branch Target Address: pc_BTA_out = 00000000000000000000000000000100
# KERNEL: D Flip Flop for Stack: pc_stack_out = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: Mux of PC: PCsrc = 00, pc_in=00000000000000000000000000001000
# KERNEL: PC module: state = 001, pc_in=00000000000000000000000000001000, pc_out=00000000000000000000000000000100
# KERNEL: To find PC+1: pc_out = 00000000000000000000000000000100, pc_next_in=00000000000000000000000000001000
# KERNEL: To find BTA: pc_out = 00000000000000000000000000000100, ext_reg_out=00000000000000000000000000000000, pc_BTA_in=00000000000000000000000000000100
# KERNEL: To find JTA: pc_out = 00000000000000000000000000000100, imm_16=0000000000000000, pc_J_in=000000zzzzzzzzzz0000000000000000
# KERNEL: pc_out = 00000000000000000000000000000100, inst_mem_out=00010100010000000000000000000000
# KERNEL: inst_mem_out=00010100010000000000000000000000, op_code=000011, inst_rs1=0000, inst_rs2=xxxx, inst_rd=0000, imm_16=0000000000000000, J_offset=xxxxxxxxxxxxxxxxxxxxxxxxxx, mode=00
You are screen sharing
Stop Share
Ln 14,Col 15
NUM INS
11:20 PM
1/29/2024

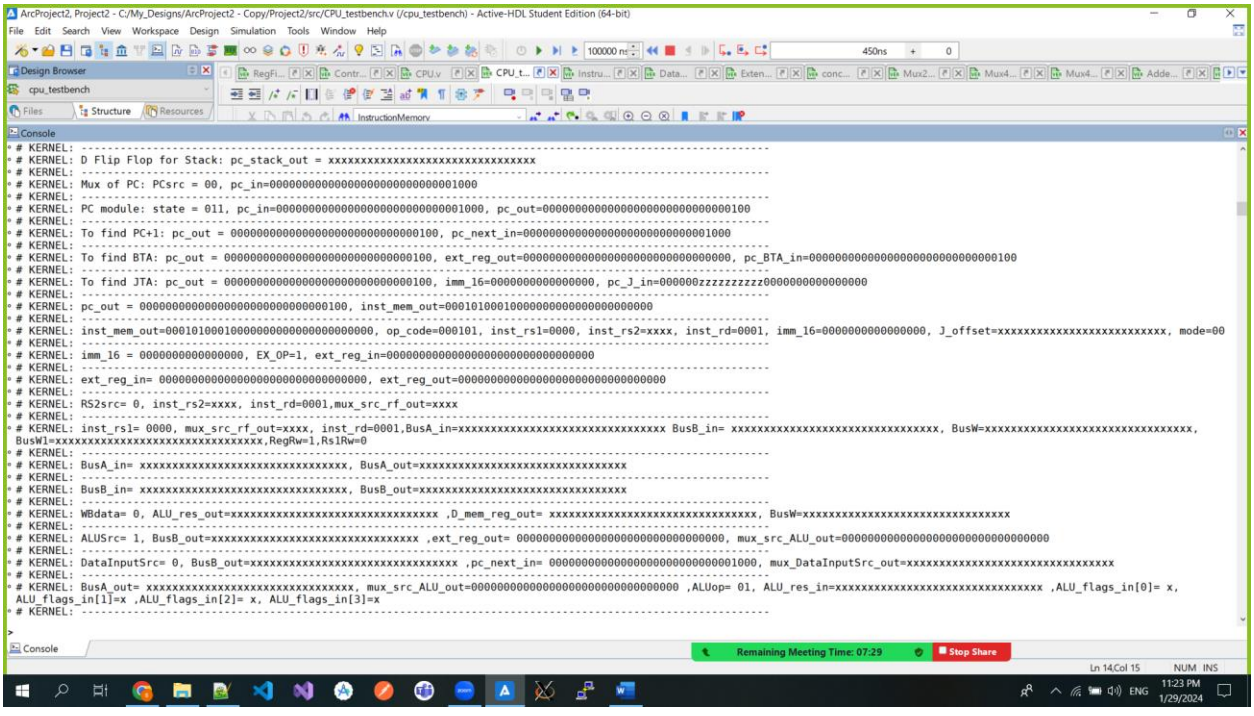

ArcProject2, Project2 - C:/My_Designs/ArcProject2 - Copy/Project2/src/CPU_testbench.v (/cpu_testbench) - Active-HDL Student Edition (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
cpu_testbench
Files Structure Resources
Console
# KERNEL: ----------
# KERNEL: imm_16 = 0000000000000000, EX_OP=0, ext_reg_in=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: ext_reg_in= 00000000000000000000000000000000, ext_reg_out=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: RS2src= 0, inst_rs2=xxxx, inst_rd=0000,mux_src_rf_out=xxxx
# KERNEL: ----------
# KERNEL: inst_rs1= 0000, mux_src_rf_out=xxxx, inst_rd=0000,BusA_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=00000000000000000000000000000000,
BusW1=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,RegRw=1,Rs1Rw=0
# KERNEL: ----------
# KERNEL: BusA_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusA_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: WBdata= 0, ALU_res_out=00000000000000000000000000000000 ,D_mem_reg_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: ALUSrc= 1, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ext_reg_out= 00000000000000000000000000000000, mux_src_ALU_out=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: DataInputSrc= 0, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,pc_next_in= 00000000000000000000000000001000, mux_DataInputSrc_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: BusA_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, mux_src_ALU_out=00000000000000000000000000000000 ,ALUop= 00, ALU_res_in=00000000000000000000000000000000 ,ALU_flags_in[0]= x,
ALU_flags_in[1]=1 ,ALU_flags_in[2]= 0, ALU_flags_in[3]=x
# KERNEL: ----------
# KERNEL: MemW= 0, MemR=0 ,DataMemEn= 0, ALU_res_in=00000000000000000000000000000000 ,mux_DataInputSrc_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: ALU_flags_in= x01x, ALU_flags_out=x01x
# KERNEL: ----------
# KERNEL: ALU_res_in= 00000000000000000000000000000000, ALU_res_out=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: D_mem_reg_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: D Flip Flop for PC+1: pc_next_out = 00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: D Flip Flop for Jump Target Address: pc_J_out = 000000zzzzzzzzzz00000000000000
# KERNEL: ----------
# KERNEL: D Flip Flop for Branch Target Address: pc_BTA_out = 00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: D Flip Flop for Stack: pc_stack_out = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
Remaining Meeting Time: 09:43
Stop Share
Ln 14,Col 15
NUM INS
ENG
11:21 PM
1/29/2024

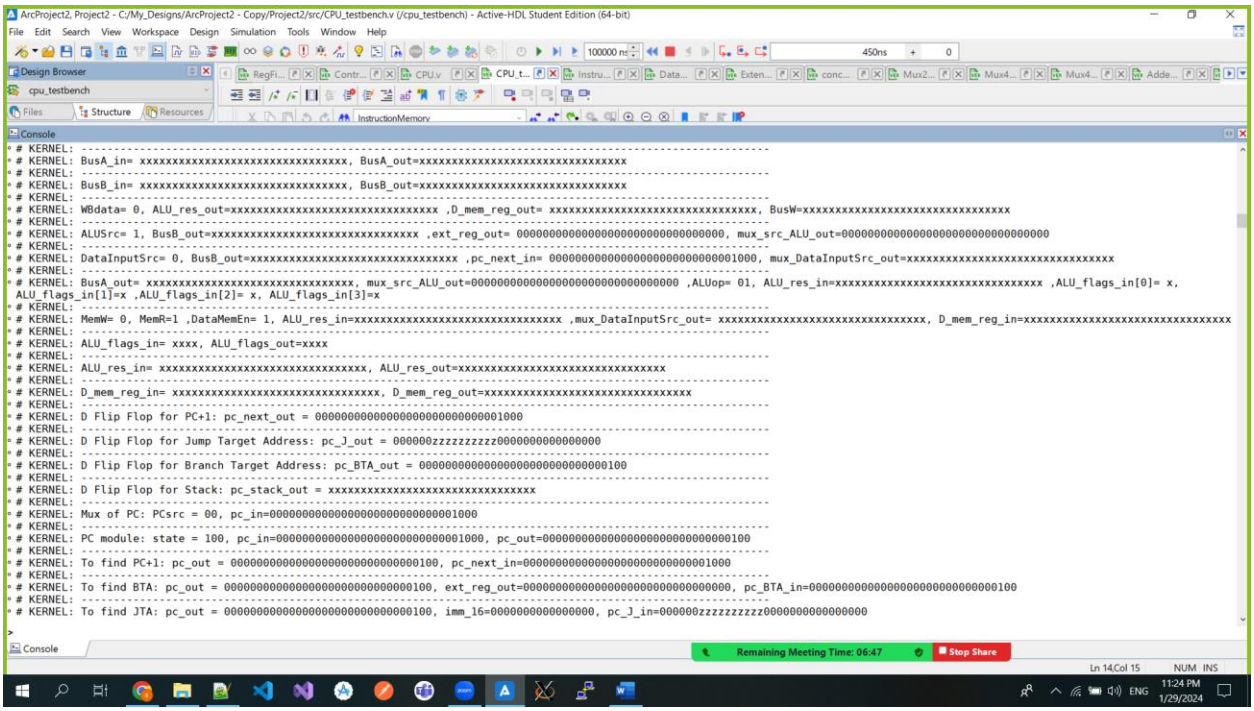

ArcProject2, Project2 - C:/My_Designs/ArcProject2 - Copy/Project2/src/CPU_testbench.v (/cpu_testbench) - Active-HDL Student Edition (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
cpu_testbench
Files Structure Resources
Console
# KERNEL: ----------
# KERNEL: Mux of PC: PCsrc = 00, pc_in=00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: PC module: state = 001, pc_in=00000000000000000000000000001000, pc_out=00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: To find PC+1: pc_out = 00000000000000000000000000000100, pc_next_in=00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: To find BTA: pc_out = 00000000000000000000000000000100, ext_reg_out=00000000000000000000000000000000, pc_BTA_in=00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: To find JTA: pc_out = 00000000000000000000000000000100, imm_16=0000000000000000, pc_J_in=000000zzzzzzzzzz00000000000000
# KERNEL: ----------
# KERNEL: pc_out = 00000000000000000000000000000100, inst_mem_out=00010100010000000000000000000000
# KERNEL: ----------
# KERNEL: inst_mem_out=00010100010000000000000000000000, op_code=000101, inst_rs1=0000, inst_rs2=xxxx, inst_rd=0001, imm_16=0000000000000000, J_offset=xxxxxxxxxxxxxxxxxxxxxxxxxx, mode=00
# KERNEL: ----------
# KERNEL: imm_16 = 0000000000000000, EX_OP=1, ext_reg_in=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: ext_reg_in= 00000000000000000000000000000000, ext_reg_out=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: RS2src= 0, inst_rs2=xxxx, inst_rd=0001,mux_src_rf_out=xxxx
# KERNEL: ----------
# KERNEL: inst_rs1= 0000, mux_src_rf_out=xxxx, inst_rd=0001,BusA_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,
BusW1=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,RegRw=1,Rs1Rw=0
# KERNEL: ----------
# KERNEL: BusA_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusA_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: WBdata= 0, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,D_mem_reg_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: ALUSrc= 1, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ext_reg_out= 00000000000000000000000000000000, mux_src_ALU_out=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: DataInputSrc= 0, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,pc_next_in= 00000000000000000000000000001000, mux_DataInputSrc_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: BusA_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, mux_src_ALU_out=00000000000000000000000000000000 ,ALUop= 01, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ALU_flags_in[0]= x,
ALU_flags_in[1]=x ,ALU_flags_in[2]= x, ALU_flags_in[3]=x
# KERNEL: ----------
# KERNEL: MemW= 0, MemR=1 ,DataMemEn= 1, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,mux_DataInputSrc_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
Remaining Meeting Time: 08:55
Stop Share
Ln 14,Col 15
NUM INS
ENG
11:22 PM
1/29/2024

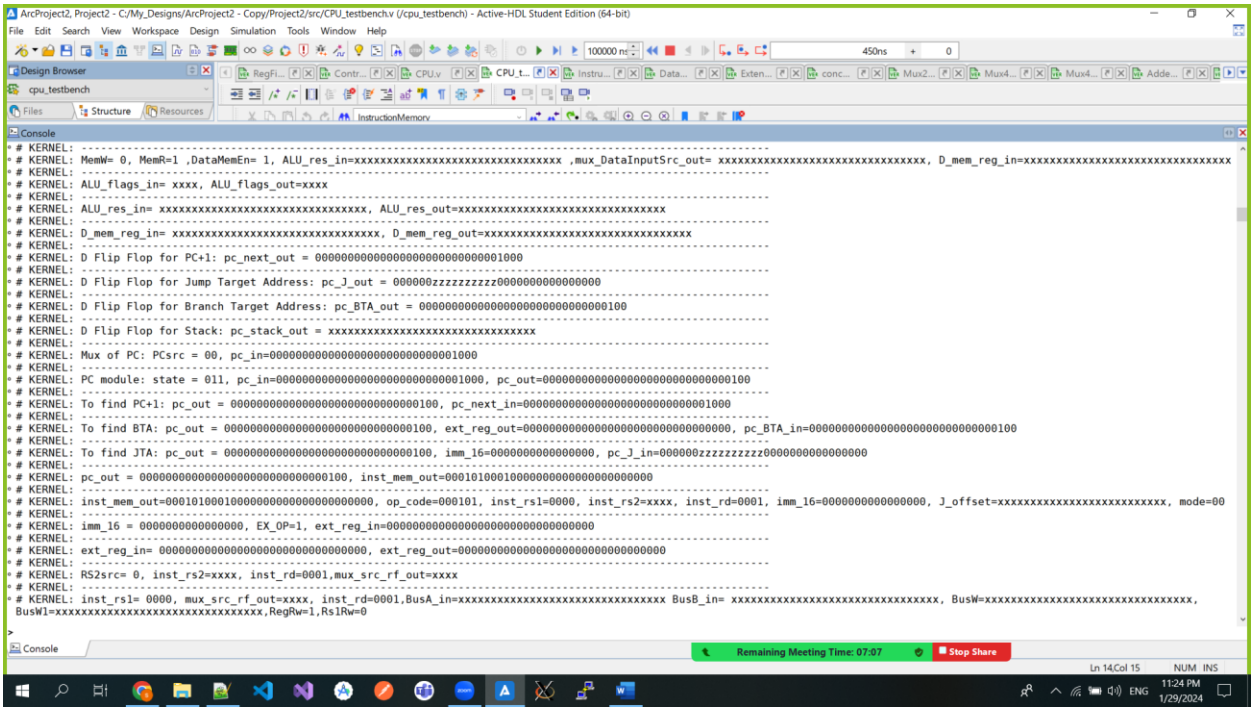

ArcProject2, Project2 - C:/My_Designs/ArcProject2 - Copy/Project2/src/CPU_testbench.v (/cpu_testbench) - Active-HDL Student Edition (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
cpu_testbench
Files Structure Resources
Console
# KERNEL: ALU_flags_in= xxxx, ALU_flags_out=xxxx
# KERNEL: ALU_res_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: D_mem_reg_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: D Flip Flop for PC+1: pc_next_out = 00000000000000000000000000001000
# KERNEL: D Flip Flop for Jump Target Address: pc_J_out = 000000zzzzzzzzzz0000000000000000
# KERNEL: D Flip Flop for Branch Target Address: pc_BTA_out = 00000000000000000000000000000100
# KERNEL: D Flip Flop for Stack: pc_stack_out = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: Mux of PC: PCsrc = 00, pc_in=00000000000000000000000000001000
# KERNEL: PC module: state = 010, pc_in=00000000000000000000000000001000, pc_out=00000000000000000000000000000100
# KERNEL: To find PC+1: pc_out = 00000000000000000000000000000100, pc_next_in=00000000000000000000000000001000
# KERNEL: To find BTA: pc_out = 00000000000000000000000000000100, ext_reg_out=00000000000000000000000000000000, pc_BTA_in=00000000000000000000000000000100
# KERNEL: To find JTA: pc_out = 00000000000000000000000000000100, imm_16=0000000000000000, pc_J_in=000000zzzzzzzzzz0000000000000000
# KERNEL: pc_out = 00000000000000000000000000000100, inst_mem_out=00010100010000000000000000000000
# KERNEL: inst_mem_out=00010100010000000000000000000000, op_code=000101, inst_rs1=0000, inst_rs2=xxxx, inst_rd=0001, imm_16=0000000000000000, J_offset=xxxxxxxxxxxxxxxxxxxxxxxxxx, mode=00
# KERNEL: imm_16 = 0000000000000000, EX_OP=1, ext_reg_in=00000000000000000000000000000000
# KERNEL: ext_reg_in= 00000000000000000000000000000000, ext_reg_out=00000000000000000000000000000000
# KERNEL: RS2src= 0, inst_rs2=xxxx, inst_rd=0001,mux_src_rf_out=xxxx
# KERNEL: inst_rs1= 0000, mux_src_rf_out=xxxx, inst_rd=0001,BusA_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,
BusWl=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,RegRw=1,Rs1Rw=0
# KERNEL: BusA_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusA_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Remaining Meeting Time: 08:35
Stop Share
Ln 14,Col 15
NUM INS
11:22 PM
1/29/2024

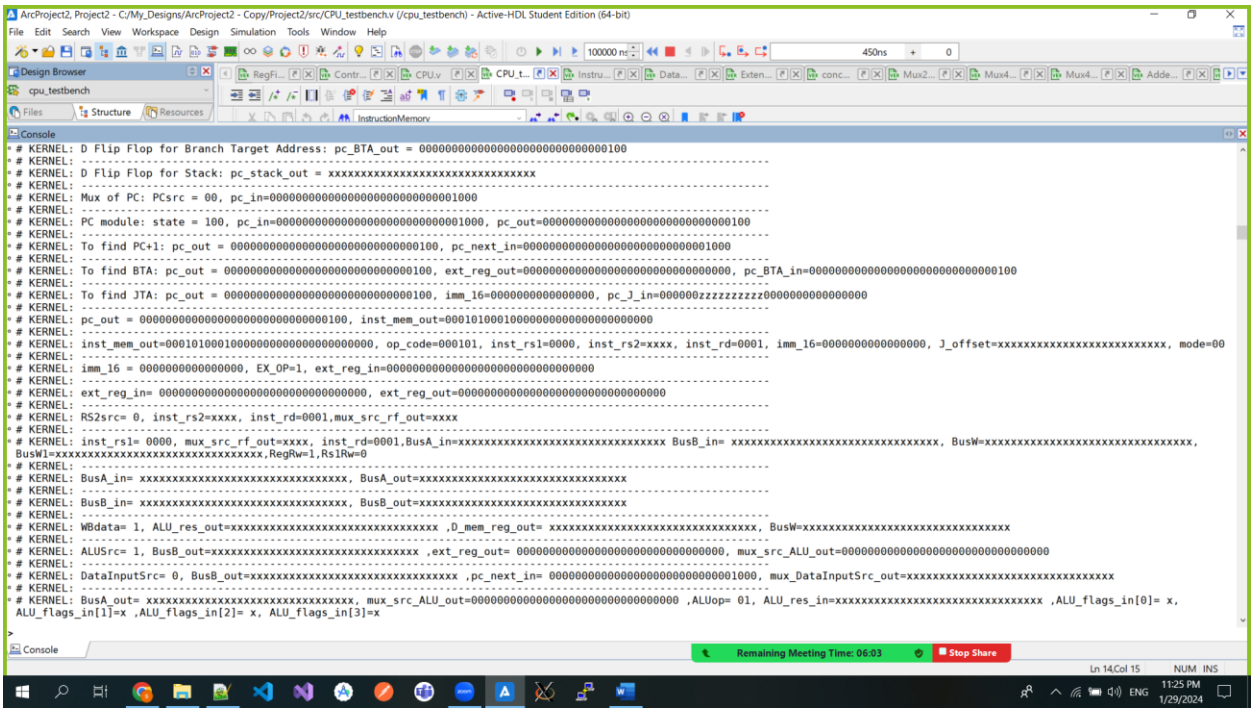

ArcProject2, Project2 - C:/My_Designs/ArcProject2 - Copy/Project2/src/CPU_testbench.v (/cpu_testbench) - Active-HDL Student Edition (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
cpu_testbench
Files Structure Resources
Console
# KERNEL: BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: WBdata= 0, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,D_mem_reg_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ALUSrc= 1, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ext_reg_out= 00000000000000000000000000000000, mux_src_ALU_out=00000000000000000000000000000000
# KERNEL: DataInputSrc= 0, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,pc_next_in= 00000000000000000000000000001000, mux_DataInputSrc_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: BusA_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, mux_src_ALU_out=00000000000000000000000000000000 ,ALUop= 01, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ALU_flags_in[0]= x,
ALU_flags_in[1]=x ,ALU_flags_in[2]= x, ALU_flags_in[3]=x
# KERNEL: MemW= 0, MemR=1 ,DataMemEn= 1, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,mux_DataInputSrc_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ALU_flags_in= xxxx, ALU_flags_out=xxxx
# KERNEL: ALU_res_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: D_mem_reg_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: D Flip Flop for PC+1: pc_next_out = 00000000000000000000000000001000
# KERNEL: D Flip Flop for Jump Target Address: pc_J_out = 000000zzzzzzzzzz0000000000000000
# KERNEL: D Flip Flop for Branch Target Address: pc_BTA_out = 00000000000000000000000000000100
# KERNEL: D Flip Flop for Stack: pc_stack_out = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: Mux of PC: PCsrc = 00, pc_in=00000000000000000000000000001000
# KERNEL: PC module: state = 010, pc_in=00000000000000000000000000001000, pc_out=00000000000000000000000000000100
# KERNEL: To find PC+1: pc_out = 00000000000000000000000000000100, pc_next_in=00000000000000000000000000001000
# KERNEL: To find BTA: pc_out = 00000000000000000000000000000100, ext_reg_out=00000000000000000000000000000000, pc_BTA_in=00000000000000000000000000000100
# KERNEL: To find JTA: pc_out = 00000000000000000000000000000100, imm_16=0000000000000000, pc_J_in=000000zzzzzzzzzz0000000000000000
# KERNEL: pc_out = 00000000000000000000000000000100, inst_mem_out=00010100010000000000000000000000
Remaining Meeting Time: 08:11
Stop Share
Ln 14,Col 15
NUM INS
11:23 PM
1/29/2024

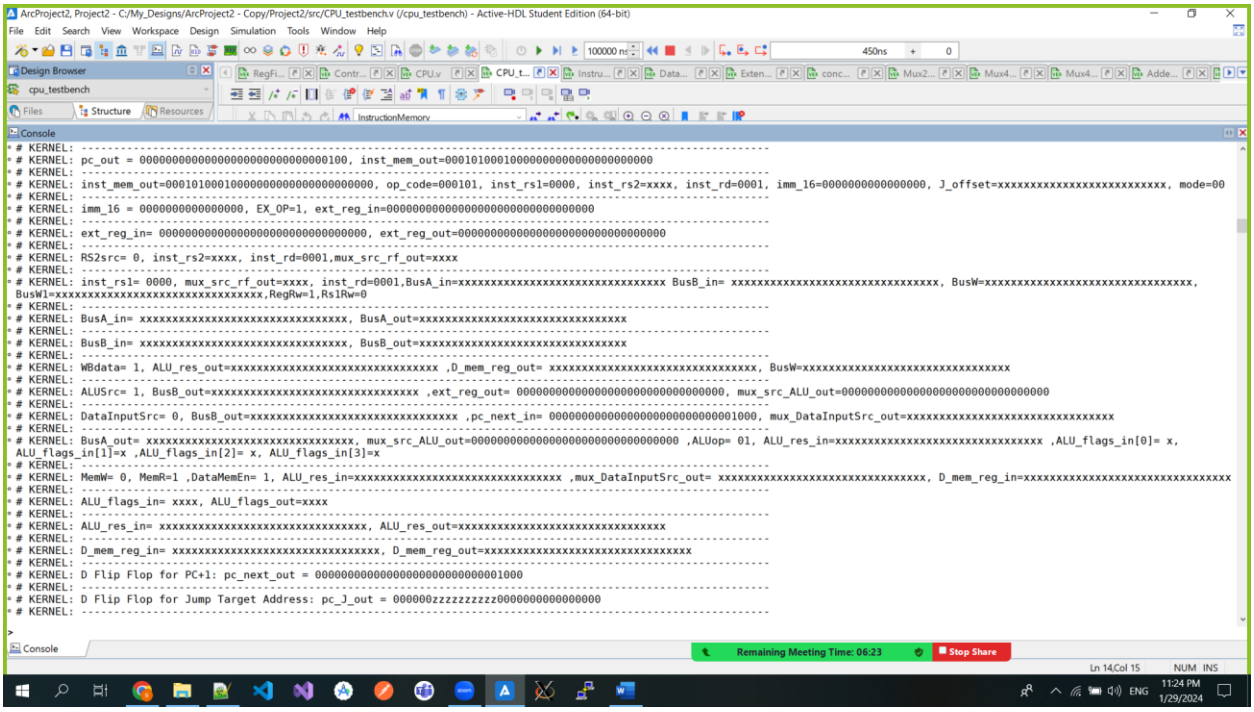

ArcProject2, Project2 - C:/My_Designs/ArcProject2 - Copy/Project2/src/CPU_testbench.v (/cpu_testbench) - Active-HDL Student Edition (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
cpu_testbench
Files Structure Resources
Console
# KERNEL: ----------
# KERNEL: inst_mem_out=00010100010000000000000000000000, op_code=000101, inst_rs1=0000, inst_rs2=xxxx, inst_rd=0001, imm_16=0000000000000000, J_offset=xxxxxxxxxxxxxxxxxxxxxxxxxx, mode=00
# KERNEL: ----------
# KERNEL: imm_16 = 0000000000000000, EX_OP=1, ext_reg_in=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: ext_reg_in= 00000000000000000000000000000000, ext_reg_out=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: RS2src= 0, inst_rs2=xxxx, inst_rd=0001,mux_src_rf_out=xxxx
# KERNEL: ----------
# KERNEL: inst_rs1= 0000, mux_src_rf_out=xxxx, inst_rd=0001,BusA_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,
BusW1=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,RegRw=1,Rs1Rw=0
# KERNEL: ----------
# KERNEL: BusA_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusA_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: WBdata= 0, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,D_mem_reg_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: ALUSrc= 1, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ext_reg_out= 00000000000000000000000000000000, mux_src_ALU_out=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: DataInputSrc= 0, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,pc_next_in= 00000000000000000000000000001000, mux_DataInputSrc_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: BusA_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, mux_src_ALU_out=00000000000000000000000000000000 ,ALUop= 01, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ALU_flags_in[0]= x,
ALU_flags_in[1]=x ,ALU_flags_in[2]= x, ALU_flags_in[3]=x
# KERNEL: ----------
# KERNEL: MemW= 0, MemR=1 ,DataMemEn= 1, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,mux_DataInputSrc_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: ALU_flags_in= xxxx, ALU_flags_out=xxxx
# KERNEL: ----------
# KERNEL: ALU_res_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: D_mem_reg_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: D Flip Flop for PC+1: pc_next_out = 00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: D Flip Flop for Jump Target Address: pc_J_out = 000000zzzzzzzzzz0000000000000000
# KERNEL: ----------
# KERNEL: D Flip Flop for Branch Target Address: pc_BTA_out = 00000000000000000000000000000100
# KERNEL: ----------
Console
Remaining Meeting Time: 07:47
Stop Share
Ln 14,Col 15
NUM INS
ENG
11:23 PM
1/29/2024

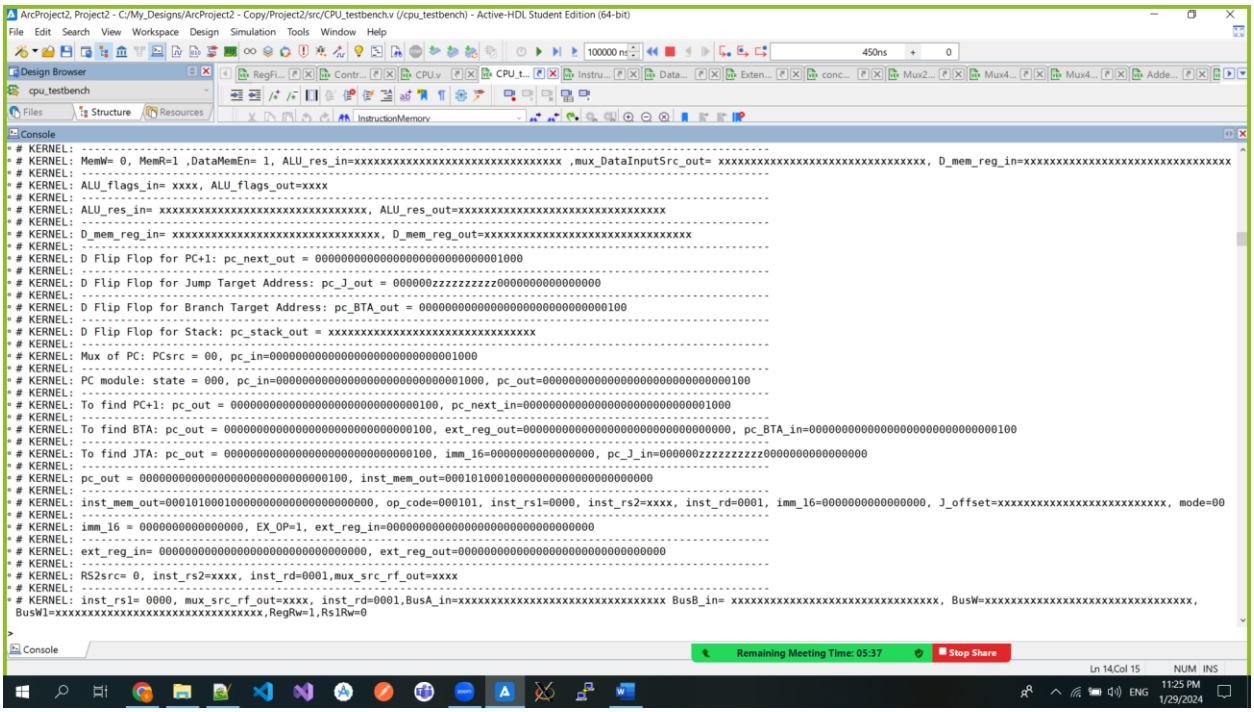

ArcProject2, Project2 - C:/My_Designs/ArcProject2 - Copy/Project2/src/CPU_testbench.v (/cpu_testbench) - Active-HDL Student Edition (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
cpu_testbench
Files Structure Resources
Console
# KERNEL: ----------
# KERNEL: D Flip Flop for Stack: pc_stack_out = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: Mux of PC: PCsrc = 00, pc_in=00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: PC module: state = 011, pc_in=00000000000000000000000000001000, pc_out=00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: To find PC+1: pc_out = 00000000000000000000000000000100, pc_next_in=00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: To find BTA: pc_out = 00000000000000000000000000000100, ext_reg_out=00000000000000000000000000000000, pc_BTA_in=00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: To find JTA: pc_out = 00000000000000000000000000000100, imm_16=0000000000000000, pc_J_in=000000zzzzzzzzzz0000000000000000
# KERNEL: ----------
# KERNEL: pc_out = 00000000000000000000000000000100, inst_mem_out=00010100010000000000000000000000
# KERNEL: ----------
# KERNEL: inst_mem_out=00010100010000000000000000000000, op_code=000101, inst_rs1=0000, inst_rs2=xxxx, inst_rd=0001, imm_16=0000000000000000, J_offset=xxxxxxxxxxxxxxxxxxxxxxxxxx, mode=00
# KERNEL: ----------
# KERNEL: imm_16 = 0000000000000000, EX_OP=1, ext_reg_in=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: ext_reg_in= 00000000000000000000000000000000, ext_reg_out=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: RS2src= 0, inst_rs2=xxxx, inst_rd=0001,mux_src_rf_out=xxxx
# KERNEL: ----------
# KERNEL: inst_rs1= 0000, mux_src_rf_out=xxxx, inst_rd=0001,BusA_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,
BusW1=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,RegRw=1,Rs1Rw=0
# KERNEL: ----------
# KERNEL: BusA_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusA_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: WBdata= 0, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,D_mem_reg_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: ALUSrc= 1, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ext_reg_out= 00000000000000000000000000000000, mux_src_ALU_out=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: DataInputSrc= 0, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,pc_next_in= 00000000000000000000000000001000, mux_DataInputSrc_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: BusA_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, mux_src_ALU_out=00000000000000000000000000000000 ,ALUop= 01, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ALU_flags_in[0]= x,
ALU_flags_in[1]=x ,ALU_flags_in[2]= x, ALU_flags_in[3]=x
# KERNEL: ----------
Console
Remaining Meeting Time: 07:29
Stop Share
Ln 14,Col 15
NUM INS
ENG
11:23 PM
1/29/2024


ArcProject2, Project2 - C:/My_Designs/ArcProject2 - Copy/Project2/src/CPU_testbench.v (/cpu_testbench) - Active-HDL Student Edition (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
cpu_testbench
Files Structure Resources
Console
# KERNEL: ----------
# KERNEL: BusA_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusA_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: WBdata= 0, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,D_mem_reg_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: ALUSrc= 1, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ext_reg_out= 00000000000000000000000000000000, mux_src_ALU_out=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: DataInputSrc= 0, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,pc_next_in= 00000000000000000000000000001000, mux_DataInputSrc_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: BusA_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, mux_src_ALU_out=00000000000000000000000000000000 ,ALUop= 01, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ALU_flags_in[0]= x,
ALU_flags_in[1]=x ,ALU_flags_in[2]= x, ALU_flags_in[3]=x
# KERNEL: ----------
# KERNEL: MemW= 0, MemR=1 ,DataMemEn= 1, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,mux_DataInputSrc_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: ALU_flags_in= xxxx, ALU_flags_out=xxxx
# KERNEL: ----------
# KERNEL: ALU_res_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: D_mem_reg_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: D Flip Flop for PC+1: pc_next_out = 00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: D Flip Flop for Jump Target Address: pc_J_out = 000000zzzzzzzzzz0000000000000000
# KERNEL: ----------
# KERNEL: D Flip Flop for Branch Target Address: pc_BTA_out = 00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: D Flip Flop for Stack: pc_stack_out = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: Mux of PC: PCsrc = 00, pc_in=00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: PC module: state = 100, pc_in=00000000000000000000000000001000, pc_out=00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: To find PC+1: pc_out = 00000000000000000000000000000100, pc_next_in=00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: To find BTA: pc_out = 00000000000000000000000000000100, ext_reg_out=00000000000000000000000000000000, pc_BTA_in=00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: To find JTA: pc_out = 00000000000000000000000000000100, imm_16=0000000000000000, pc_J_in=000000zzzzzzzzzz0000000000000000
Console
Remaining Meeting Time: 06:47
Stop Share
Ln 14,Col 15
NUM INS
ENG
11:24 PM
1/29/2024


ArcProject2, Project2 - C:/My_Designs/ArcProject2 - Copy/Project2/src/CPU_testbench.v (/cpu_testbench) - Active-HDL Student Edition (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
cpu_testbench
Files Structure Resources
Console
KERNEL: ----------
KERNEL: MemW= 0, MemR=1 ,DataMemEn= 1, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,mux_DataInputSrc_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
KERNEL: ----------
KERNEL: ALU_flags_in= xxxx, ALU_flags_out=xxxx
KERNEL: ----------
KERNEL: ALU_res_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
KERNEL: ----------
KERNEL: D_mem_reg_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: D Flip Flop for PC+1: pc_next_out = 00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: D Flip Flop for Jump Target Address: pc_J_out = 000000zzzzzzzzzz0000000000000000
# KERNEL: ----------
# KERNEL: D Flip Flop for Branch Target Address: pc_BTA_out = 00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: D Flip Flop for Stack: pc_stack_out = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: Mux of PC: PCsrc = 00, pc_in=00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: PC module: state = 011, pc_in=00000000000000000000000000001000, pc_out=00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: To find PC+1: pc_out = 00000000000000000000000000000100, pc_next_in=00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: To find BTA: pc_out = 00000000000000000000000000000100, ext_reg_out=00000000000000000000000000000000, pc_BTA_in=00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: To find JTA: pc_out = 00000000000000000000000000000100, imm_16=0000000000000000, pc_J_in=000000zzzzzzzzzz0000000000000000
# KERNEL: ----------
# KERNEL: pc_out = 00000000000000000000000000000100, inst_mem_out=00010100010000000000000000000000
# KERNEL: ----------
# KERNEL: inst_mem_out=00010100010000000000000000000000, op_code=000101, inst_rs1=0000, inst_rs2=xxxx, inst_rd=0001, imm_16=0000000000000000, J_offset=xxxxxxxxxxxxxxxxxxxxxxxxxx, mode=00
# KERNEL: ----------
# KERNEL: imm_16 = 0000000000000000, EX_OP=1, ext_reg_in=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: ext_reg_in= 00000000000000000000000000000000, ext_reg_out=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: RS2src= 0, inst_rs2=xxxx, inst_rd=0001,mux_src_rf_out=xxxx
# KERNEL: ----------
# KERNEL: inst_rs1= 0000, mux_src_rf_out=xxxx, inst_rd=0001,BusA_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,
BusW1=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,RegRw=1,Rs1Rw=0
Console
Remaining Meeting Time: 07:07
Stop Share
Ln 14,Col 15
NUM INS
ENG
11:24 PM
1/29/2024


ArcProject2, Project2 - C:/My_Designs/ArcProject2 - Copy/Project2/src/CPU_testbench.v (/cpu_testbench) - Active-HDL Student Edition (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
cpu_testbench
Files Structure Resources
Console
# KERNEL: D Flip Flop for Branch Target Address: pc_BTA_out = 00000000000000000000000000000100
# KERNEL: D Flip Flop for Stack: pc_stack_out = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: Mux of PC: PCsrc = 00, pc_in=00000000000000000000000000001000
# KERNEL: PC module: state = 100, pc_in=00000000000000000000000000001000, pc_out=00000000000000000000000000000100
# KERNEL: To find PC+1: pc_out = 00000000000000000000000000000100, pc_next_in=00000000000000000000000000001000
# KERNEL: To find BTA: pc_out = 00000000000000000000000000000100, ext_reg_out=00000000000000000000000000000000, pc_BTA_in=00000000000000000000000000000100
# KERNEL: To find JTA: pc_out = 00000000000000000000000000000100, imm_16=0000000000000000, pc_J_in=000000zzzzzzzzzz0000000000000000
# KERNEL: pc_out = 00000000000000000000000000000100, inst_mem_out=00010100010000000000000000000000
# KERNEL: inst_mem_out=00010100010000000000000000000000, op_code=000101, inst_rs1=0000, inst_rs2=xxxx, inst_rd=0001, imm_16=0000000000000000, J_offset=xxxxxxxxxxxxxxxxxxxxxxxxxx, mode=00
# KERNEL: imm_16 = 0000000000000000, EX_OP=1, ext_reg_in=00000000000000000000000000000000
# KERNEL: ext_reg_in= 00000000000000000000000000000000, ext_reg_out=00000000000000000000000000000000
# KERNEL: RS2src= 0, inst_rs2=xxxx, inst_rd=0001,mux_src_rf_out=xxxx
# KERNEL: inst_rs1= 0000, mux_src_rf_out=xxxx, inst_rd=0001,BusA_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,
BusW1=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,RegRw=1,Rs1Rw=0
# KERNEL: BusA_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusA_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: WBdata= 1, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,D_mem_reg_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ALUSrc= 1, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ext_reg_out= 00000000000000000000000000000000, mux_src_ALU_out=00000000000000000000000000000000
# KERNEL: DataInputSrc= 0, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,pc_next_in= 00000000000000000000000000001000, mux_DataInputSrc_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: BusA_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, mux_src_ALU_out=00000000000000000000000000000000 ,ALUop= 01, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ALU_flags_in[0]= x,
ALU_flags_in[1]=x ,ALU_flags_in[2]= x, ALU_flags_in[3]=x
Remaining Meeting Time: 06:03
Stop Share
Ln 14,Col 15
NUM INS
11:25 PM
1/29/2024


ArcProject2, Project2 - C:/My_Designs/ArcProject2 - Copy/Project2/src/CPU_testbench.v (/cpu_testbench) - Active-HDL Student Edition (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
cpu_testbench
Files Structure Resources
Console
# KERNEL: pc_out = 00000000000000000000000000000100, inst_mem_out=00010100010000000000000000000000
# KERNEL: inst_mem_out=00010100010000000000000000000000, op_code=000101, inst_rs1=0000, inst_rs2=xxxx, inst_rd=0001, imm_16=0000000000000000, J_offset=xxxxxxxxxxxxxxxxxxxxxxxxxx, mode=00
# KERNEL: imm_16 = 0000000000000000, EX_OP=1, ext_reg_in=00000000000000000000000000000000
# KERNEL: ext_reg_in= 00000000000000000000000000000000, ext_reg_out=00000000000000000000000000000000
# KERNEL: RS2src= 0, inst_rs2=xxxx, inst_rd=0001,mux_src_rf_out=xxxx
# KERNEL: inst_rs1= 0000, mux_src_rf_out=xxxx, inst_rd=0001,BusA_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,
BusW1=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,RegRw=1,Rs1Rw=0
# KERNEL: BusA_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusA_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: WBdata= 1, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,D_mem_reg_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ALUSrc= 1, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ext_reg_out= 00000000000000000000000000000000, mux_src_ALU_out=00000000000000000000000000000000
# KERNEL: DataInputSrc= 0, BusB_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,pc_next_in= 00000000000000000000000000001000, mux_DataInputSrc_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: BusA_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, mux_src_ALU_out=00000000000000000000000000000000 ,ALUop= 01, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,ALU_flags_in[0]= x,
ALU_flags_in[1]=x ,ALU_flags_in[2]= x, ALU_flags_in[3]=x
# KERNEL: MemW= 0, MemR=1 ,DataMemEn= 1, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,mux_DataInputSrc_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ALU_flags_in= xxxx, ALU_flags_out=xxxx
# KERNEL: ALU_res_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: D_mem_reg_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: D Flip Flop for PC+1: pc_next_out = 00000000000000000000000000001000
# KERNEL: D Flip Flop for Jump Target Address: pc_J_out = 000000zzzzzzzzzz0000000000000000
Remaining Meeting Time: 06:23
Stop Share
Ln 14,Col 15
NUM INS
11:24 PM
1/29/2024


ArcProject2, Project2 - C:/My_Designs/ArcProject2 - Copy/Project2/src/CPU_testbench.v (/cpu_testbench) - Active-HDL Student Edition (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
cpu_testbench
Files Structure Resources
Console
KERNEL: ----------
KERNEL: MemW= 0, MemR=1 ,DataMemEn= 1, ALU_res_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ,mux_DataInputSrc_out= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
KERNEL: ----------
KERNEL: ALU_flags_in= xxxx, ALU_flags_out=xxxx
KERNEL: ----------
KERNEL: ALU_res_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, ALU_res_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
KERNEL: ----------
KERNEL: D_mem_reg_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, D_mem_reg_out=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: D Flip Flop for PC+1: pc_next_out = 00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: D Flip Flop for Jump Target Address: pc_J_out = 000000zzzzzzzzzz0000000000000000
# KERNEL: ----------
# KERNEL: D Flip Flop for Branch Target Address: pc_BTA_out = 00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: D Flip Flop for Stack: pc_stack_out = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
# KERNEL: ----------
# KERNEL: Mux of PC: PCsrc = 00, pc_in=00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: PC module: state = 000, pc_in=00000000000000000000000000001000, pc_out=00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: To find PC+1: pc_out = 00000000000000000000000000000100, pc_next_in=00000000000000000000000000001000
# KERNEL: ----------
# KERNEL: To find BTA: pc_out = 00000000000000000000000000000100, ext_reg_out=00000000000000000000000000000000, pc_BTA_in=00000000000000000000000000000100
# KERNEL: ----------
# KERNEL: To find JTA: pc_out = 00000000000000000000000000000100, imm_16=0000000000000000, pc_J_in=000000zzzzzzzzzz0000000000000000
# KERNEL: ----------
# KERNEL: pc_out = 00000000000000000000000000000100, inst_mem_out=00010100010000000000000000000000
# KERNEL: ----------
# KERNEL: inst_mem_out=00010100010000000000000000000000, op_code=000101, inst_rs1=0000, inst_rs2=xxxx, inst_rd=0001, imm_16=0000000000000000, J_offset=xxxxxxxxxxxxxxxxxxxxxxxxxx, mode=00
# KERNEL: ----------
# KERNEL: imm_16 = 0000000000000000, EX_OP=1, ext_reg_in=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: ext_reg_in= 00000000000000000000000000000000, ext_reg_out=00000000000000000000000000000000
# KERNEL: ----------
# KERNEL: RS2src= 0, inst_rs2=xxxx, inst_rd=0001,mux_src_rf_out=xxxx
# KERNEL: ----------
# KERNEL: inst_rs1= 0000, mux_src_rf_out=xxxx, inst_rd=0001,BusA_in=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx BusB_in= xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, BusW=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,
BusW1=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx,RegRw=1,Rs1Rw=0
Remaining Meeting Time: 05:37
Stop Share
Ln 14,Col 15
NUM INS
ENG
11:25 PM
1/29/2024

4. Challenges

While developing the multi-cycle RISC processor using Verilog, several challenges were encountered, each requiring careful consideration and resolution to ensure the successful completion of the project.

- RISC-V Architecture's Complexity: Despite being modular and versatile, the RISC-V architecture is nevertheless complex. A deep understanding of the architecture's complexities was necessary to comprehend and implement its varied instruction set, which included R-type, I-type, J-type, and S-type instructions along with different formats.

- Problems with Timing and Synchronization: Timing issues arise when a multi-cycle design is implemented. For timing conflicts to be avoided and accurate instruction execution to be ensured, data must be properly transported and synchronized across stages of the processor.3

- Component Integration: It was difficult to cogently integrate disparate parts like the instruction memory, register file, extender, ALU, and data memory. Careful attention to detail was needed to link these components in a way that would allow for effective data flow and maintain synchronization.

- Control Signal Generation: Complex decision-making logic was required in order to generate control signals for the processor's numerous components. It took careful analysis and debugging to make sure the control signals were appropriately derived depending on the instruction type, function, and flags.

- Testing and Validation: Creating thorough test cases to verify the processor's operation was difficult. Careful testing and debugging were necessary to make sure the processor could handle a range of instructions and sequences correctly.

5. Conclusion

In conclusion, this project has successfully demonstrated the design and implementation of a multi-cycle RISC processor using Verilog. The RISC-V architecture was introduced at the outset of the project, emphasizing its modular and open-source nature, which make it appropriate for a variety of applications. After that, attention turned to the design requirements, which included effective instruction handling, a large number of general-purpose registers, program control management, the effectiveness of stack pointers, a variety of instruction kinds, and ALU output signals.

The general architecture of the processor was described, with a focus on the multi-cycle approach - which divides the execution of instructions into manageable chunks. The functioning and interactions of the processor's main parts, including the instruction memory, register file, extender, ALU, and data memory, were described in depth. The decision-making process during instruction execution was demonstrated by the state diagram and truth tables for main control signals, PC control, and ALU control.

Additionally, the importance of temporary registers - such as the Instruction Register (IR), Data Registers for reading RS1 and RS2, ALU Result and Flags Registers, Memory Data Register, and Extender Value Register - in promoting efficient data flow between stages was underlined.

To confirm the multi-cycle processor's operation, simulation and testing were also incorporated into the project. Test cases were created to evaluate the processor's capacity to manage a series of instructions, offering a thorough assessment of its performance in practical situations.

In the end, the project's objective of developing and evaluating a simple, multi-cycle RISC processor with Verilog was accomplished. The multi-cycle design method and modular, open-source RISC-V architecture enhance the processor's flexibility, efficiency, and versatility, thereby establishing RISC-V as a noteworthy advancement in RISC architecture. The project's results advance our knowledge of computer architecture and processor design concepts.

6. References

[1]: https://www.synopsys.com/blogs/chip-design/risc-v-architecture-explained.html

[2]: https://www.epcc.ed.ac.uk/whats-happening/articles/evolution-risc-architecture

[3]: https://www.elprocus.com/risc-v-processor/

[4]: https://www.epcc.ed.ac.uk/whats-happening/articles/evolution-risc-architecture

[5]: https://online.visual-paradigm.com/w/bpflqtqc/app/diagrams/#diagram:workspace=bpflqtqc&proj=0&id=3&type=Flowchart

7. Appendix

Appendix 1: Mux 2*1

```
module mux2x1 (
  input wire selectionLine, // selection line -> 1 bit
  input wire [31:0] a,
  input wire [31:0] b,
  output reg [31:0] result
);
  always @* begin
    if (selectionLine == 1'b0)
      result <= a;
    else
      result <= b;
  end
endmodule
```

Appendix 2: Mux 2*1 Testbench

```
module mux2x1_testbench;
  // Signals
  reg selectionLine;
  reg [31:0] a, b;
  wire [31:0] result;

  // Instantiate the mux2x1 module
  mux2x1 uut (
    .selectionLine(selectionLine),
    .a(a),
    .b(b),
    .result(result)
  );

  // Initial block: Initialize testbench signals
  initial begin
```

```
    // Test case 1: selectionLine is 0
    selectionLine = 0;
    a = 32'h12345678;
    b = 32'h87654321;
    #10; // Wait for a while
    if (result !== a)
                    $display("Test case 1 failed");

    // Test case 2: selectionLine is 1
    selectionLine = 1;
    a = 32'h12345678;
    b = 32'h87654321;
    #10; // Wait for a while
    if (result !== b)
                    $display("Test case 2 failed");
    // Finish simulation
    $stop;
  end
endmodule
```

Appendix 3: Mux 4*1

```
module mux4x1 (
  input wire [1:0] selectionLine, // selection line -> 2 bit
     input wire [31:0] a,
     input wire [31:0] b,
     input wire [31:0] c,
           input wire [31:0] d,
  output reg [31:0] result
   );

  always @*
       begin
          case(selectionLine)
          2'b00 : result = a;
          2'b01 : result = b;
```

```
            2'b10 : result = c;
            2'b11 : result = d;
            endcase
        end
    endmodule
```

Appendix 4: Mux 4*1 Testbench

```
`timescale 1ns/1ns
module mux4x1_testbench;
  // Inputs
  reg [1:0] selectionLine;
  reg [31:0] a, b, c, d;
  // Outputs
  wire [31:0] result;

  // Instantiate the mux4x1 module
  mux4x1 uut (
    .selectionLine(selectionLine),
    .a(a),
    .b(b),
    .c(c),
    .d(d),
    .result(result)
  );

  // Initial block for test stimulus
  initial begin
    // Test Case 1
    selectionLine = 2'b00;
    a = 32'h12345678;
    b = 32'h87654321;
    c = 32'hABCDEFAB;
    d = 32'hFEDCBAFE;

    #10; // Wait for 10 time units
```

```
// Check the result
if (result !== a) $display("Test Case 1 Failed! Expected: %h, Got: %h", a, result);
else $display("Test Case 1 Passed!");

// Test Case 2
selectionLine = 2'b01;
a = 32'h12345678;
b = 32'h87654321;
c = 32'hABCDEFAB;
d = 32'hFEDCBAFE;

#10; // Wait for 10 time units

// Check the result
if (result !== b) $display("Test Case 2 Failed! Expected: %h, Got: %h", b, result);
else $display("Test Case 2 Passed!");

// Test Case 3
selectionLine = 2'b10;
a = 32'h12345678;
b = 32'h87654321;
c = 32'hABCDEFAB;
d = 32'hFEDCBAFE;

#10; // Wait for 10 time units

// Check the result
if (result !== b) $display("Test Case 3 Failed! Expected: %h, Got: %h", b, result);
else $display("Test Case 3 Passed!");

    // Test Case 4
selectionLine = 2'b11;
a = 32'h12345678;
```

```
    b = 32'h87654321;
    c = 32'hABCDEFAB;
    d = 32'hFEDCBAFE;

    #10; // Wait for 10 time units

    // Check the result
    if (result !== b) $display("Test Case 4 Failed! Expected: %h, Got: %h", b, result);
    else $display("Test Case 4 Passed!");

    // Finish simulation
    $finish;
  end
endmodule
```

Appendix 5: Adder

```
module adder_32bit (
    input  [31:0] input1,  // Input from the extender
    input  [31:0] input2,       // Input from the PC
    output [31:0] adder_output  // Output to the mux
);

    // Assuming a ripple-carry adder for this example
    wire [31:0] sum;
    assign sum = input1 + input2;
    assign adder_output = sum;  // Pass the sum as the branch target address
endmodule
```

Appendix 6: Adder Testbench

```
`timescale 1ns / 1ps
module adder_32bit_tb;
    // Test Inputs
    reg [31:0] input1;
    reg [31:0] input2;
```

```
    // Output
    wire [31:0] adder_output;

    // Instantiate the Unit Under Test (UUT)
    adder_32bit uut (
        .input1(input1),
        .input2(input2),
        .adder_output(adder_output)
    );

    integer i;

    initial begin
        // Initialize Inputs
        input1 = 0;
        input2 = 0;

        // Wait for 100 ns for global reset to finish
        #100;

        // Add stimulus here
        for (i = 0; i < 4; i = i + 1) begin
            input1 = $random;
            input2 = $random;
            #10; // Wait 10 ns between input changes

            // Display the inputs and output
            $display("Time: %d, Input1: %d, Input2: %d, Output: %d",
                    $time, input1, input2, adder_output);
        end
        // Finish the simulation
        #10 $finish;
    end
endmodule
```

Appendix 7: PC

```
module PC(clk, state, PC_input, PC_output);
        input wire clk;
        input wire [31:0] PC_input;
        input [2:0]state;
        output reg [31:0] PC_output;

  initial begin
   PC_output = 0;
  end
  always @(posedge clk ) begin
    if (state == 3'b000)
      // output equals input
      PC_output<= PC_input;
  end
endmodule
```

Appendix 8: PC Testbench

```
`timescale 1ns / 1ps

module PC_testbench;

  reg clk;
  reg [31:0] PC_input;
  reg [2:0] state;
  wire [31:0] PC_output;

  // Instantiate the Unit Under Test (UUT)
  PC uut (
    .clk(clk),
    .state(state),
    .PC_input(PC_input),
    .PC_output(PC_output)
  );
```

```
 // Clock generation
 initial begin
   clk = 0;
   forever #10 clk = ~clk;  // Toggle clock every 10 ns
 end
   initial begin

 #20;
state = $random % 8; // Set state to 0
PC_input = $random; // Generate a random 32-bit number

#20;
state = $random % 8; // Set state to 0
PC_input = $random; // Generate a random 32-bit number

#20;
state = 0; // Set state to 0
PC_input = $random; // Generate a random 32-bit number

 #20;
 state = 0; // Set state to 0
 PC_input = $random; // Generate a random 32-bit number
$finish;
                    end
  // Monitor changes
  initial begin
    $monitor("Time = %d, clk = %b, state = %b, PC_input = %d, PC_output = %d",
          $time, clk, state, PC_input, PC_output);
  end
endmodule
```

Appendix 9 D Flip Flop

```
module dFlipFlop #(parameter WIDTH = 32)(
   input [WIDTH-1:0] data_input,
```

```
    input clock,
    output reg [WIDTH-1:0] data_output
);

    always @ (*) begin
        data_output <= data_input;
    end
endmodule
```

Appendix 10: D Flip Flop Testbench

```
`timescale 1ns / 1ps

module dFlipFlop_testbench;

    // Parameters
    parameter WIDTH = 32;

    // Inputs
    reg [WIDTH-1:0] data_input;
    reg clock;

    // Outputs
    wire [WIDTH-1:0] data_output;

    // Instantiate the Unit Under Test (UUT)
    dFlipFlop #(.WIDTH(WIDTH)) uut (
        .data_input(data_input),
        .clock(clock),
        .data_output(data_output)
    );

    // Clock generation
    initial begin
        clock = 0;
        forever #10 clock = ~clock; // Clock with period of 20ns
```

```
    end

    // Test case generation
    initial begin
        // Initialize Inputs
        data_input = 0;

        // Wait for global reset
        #100;

        // Apply random inputs
        forever begin
            #20 data_input = $random; // Change input every 20ns
        end
    end

    // Monitor changes and display them
    initial begin
        $monitor("Time = %t, Input = %b, Output = %b", $time, data_input, data_output);
    end

    // End simulation after a certain time
    initial begin
        #1000 $finish; // Stop simulation after 1000ns
    end
endmodule
```

Appendix 11: Control Unit

```
module controlUnit(clk, next_state, state, PCsrc, OP, ExtOp, Rs2Src, RegRw, Rs1Rw, ALUsrc, DataInputSrc, ALUop, MemR,
                MemW, Zeroflag, Negflag, WBdata, DataMemEn);

  input clk;
  input Zeroflag, Negflag;
  input [5:0] OP;
```

```
 output reg ExtOp = 0, Rs2Src = 0, RegRw = 0, Rs1Rw = 0, ALUsrc = 0, DataInputSrc = 0, MemR = 0, MemW = 0, WBdata = 0, DataMemEn = 0;
 output reg [1:0] PCsrc = 2'b00;
 output reg [1:0] ALUop = 2'b00;
 input [2:0] next_state;
 input [2:0] state;

 parameter IF_STAGE = 3'b000;
 parameter ID_STAGE = 3'b001;
 parameter EX_STAGE = 3'b010;
 parameter MEM_STAGE = 3'b011;
 parameter WB_STAGE = 3'b100;

 parameter AND = 0;
 parameter ADD = 1;
 parameter SUB = 2;

        // main control signals
 always@ (negedge clk) begin
  casex({OP})
                        6'b00000?: begin // R-type: AND + ADD
                                Rs2Src <= 0;
                                MemR <= 0;
                                MemW <= 0;
           RegRw <= 1;
                Rs1Rw <= 0;

          if(next_state == WB_STAGE) begin
                                        WBdata <= 1;
                                end
                else begin
             WBdata <= 0;
           end
                        end
```

```
                6'b000010: begin // SUB
                        Rs2Src <= 0;
                        MemR <= 0;
                        MemW <= 0;
  RegRw <= 1;
        Rs1Rw <= 0;

 if(next_state == WB_STAGE) begin
                                WBdata <= 1;
                        end
        else begin
   WBdata <= 0;
  end
                end


                6'b000011: begin // I-type: ANDI
  RegRw <= 1;
  ExtOp <= 0;
        MemR <= 0;
                        MemW <= 0;
  Rs1Rw <=0;
 if(next_state == WB_STAGE) begin
                                WBdata <= 1;
                        end
        else begin
   WBdata <= 0;
  end
end

 6'b000100: begin // I-type: ADDI
  RegRw <= 1;
  ExtOp <= 1;
        MemR <= 0;
```

```
                        MemW <= 0;
  Rs1Rw <=0;
 if(next_state == WB_STAGE) begin
                                WBdata <= 1;
                        end
        else begin
    WBdata <= 0;
  end
end

                6'b000101: begin // I-type: LW
  RegRw <= 1;
  ExtOp <= 1;
        MemR <= 1;
                        MemW <= 0;
  Rs1Rw <=0;
                        DataMemEn <= 1;

 if(next_state == WB_STAGE) begin
                                WBdata <= 1;
                        end
        else begin
    WBdata <= 0;
  end
                end


        6'b000110: begin // I-type: LW.POI
  RegRw <= 1;
  ExtOp <= 1;
        MemR <= 1;
                        MemW <= 0;
  Rs1Rw <=1;
                        DataMemEn <= 1;
```

```
  if(next_state == WB_STAGE) begin
                                        WBdata <= 1;
                                end
          else begin
    WBdata <= 0;
   end
                        end


          6'b000111: begin // I-type: STORE
   RegRw <= 0;
   ExtOp <= 1;
   DataInputSrc <= 0;
          MemR <= 0;
                                DataMemEn <= 1;
  if(next_state == MEM_STAGE) begin
                                        MemW = 1;
                                end
          else begin
    MemW <= 0;
   end
                        end


6'b0010??: begin //Branch
   Rs2Src <= 1 ;
   RegRw <= 0;
   MemR <= 0;
                                MemW <= 0;
end
```

```
                6'b001100:  begin // J-type: jump
    RegRw <= 0;
    MemR <= 0;
    MemW <= 0;
end


                6'b001101:  begin // J-type: CALL
                        RegRw <= 0;
   DataInputSrc <= 1;
   MemR <= 0;
                        DataMemEn <= 0;
  if(next_state == MEM_STAGE) begin
                                MemW = 1;
                        end
          else begin
     MemW <= 0;
   end
 end

 6'b001110:  begin // J-type: RET
   RegRw <= 0;
   MemR <= 1;
   MemW <= 0;
                        DataMemEn <= 0;
   if(next_state == WB_STAGE) begin
                                WBdata <= 1;
                        end
          else begin
     WBdata <= 0;
   end
  end

6'b001111:  begin // PUSH
```

```
        Rs2Src <= 1 ;
        RegRw <= 0;
        DataInputSrc <= 0;
        MemR <= 0;
                        DataMemEn <= 0;
        if(next_state == MEM_STAGE) begin
                                        MemW = 1;
                                end
                else begin
            MemW <= 0;
           end
       end


        6'b010000:  begin // POP
        RegRw <= 1;
        MemR <= 1;
        MemW <= 0;
                        DataMemEn <= 0;
                        Rs1Rw <=0;
        if(next_state == WB_STAGE) begin
                                        WBdata <= 1;
                                end
                else begin
            WBdata <= 0;
           end
       end
                endcase
         end

// PC control
always @(negedge clk) begin // TODO: posedge
                if(next_state == IF_STAGE) begin
       if (OP == 6'b001100 || OP == 6'b001101 || OP == 6'b001110)begin
```

```
          if (OP == 6'b001110)begin
           PCsrc <= 3; // RET
           end
          else begin
            PCsrc <= 1; // Jump Target Address
          end
        end
                    else if ((OP == 6'b001010 && Zeroflag == 1) || (OP == 6'b001011 && Zeroflag == 0) ||
(OP == 6'b001000 && Negflag == 0) || (OP == 6'b001001 && Negflag == 1))begin // Branch
          PCsrc <= 2; // Branch Target Address
        end
        else begin
          PCsrc <= 0; // Other instructions
        end
                end
   end


 // ALU control signal
 always @(posedge clk, OP) begin
   casex({OP})
                        6'b000000: begin // AND
                                ALUsrc <= 0;
                                ALUop <= AND;
                        end

                        6'b000001: begin // ADD
                                ALUsrc <= 0;
                                ALUop <= ADD;
                        end

                        8'b00000010: begin // SUB
                                ALUsrc <= 0;
                                ALUop <= SUB;
                        end
```

```
                6'b000011: begin // ANDI
                        ALUsrc <= 1;
                        ALUop <= AND;
                end

                6'b000100: begin // ADDI
                        ALUsrc <= 1;
                        ALUop <= ADD;
                end

        6'b000101: begin // LW
                        ALUsrc <= 1;
                        ALUop <= ADD;
                end

                6'b00011?: begin // LW.POI + SW
                        ALUsrc <= 1;
                        ALUop <= ADD;
                end

                6'b0010??: begin // Branch
                        ALUsrc <= 0;
                        ALUop <= SUB;
                end
        endcase
    end
endmodule
```

Appendix 12: Control Unit Testbench

```
module ControlUnit_testbench();

  reg clk;
  reg [5:0] OP;
  reg [2:0] next_state;
```

```
reg [2:0] state;
reg Zeroflag, Negflag;
wire ExtOp, Rs2Src, RegRw, Rs1Rw, ALUsrc, DataInputSrc, MemR, MemW, WBdata, DataMemEn;
wire [1:0] PCsrc;
wire [1:0] ALUop;

controlUnit uut (
  .clk(clk),
  .next_state(next_state),
  .state(state),
  .PCsrc(PCsrc),
  .OP(OP),
  .ExtOp(ExtOp),
  .Rs2Src(Rs2Src),
  .RegRw(RegRw),
  .Rs1Rw(Rs1Rw),
  .ALUsrc(ALUsrc),
  .DataInputSrc(DataInputSrc),
  .ALUop(ALUop),
  .MemR(MemR),
  .MemW(MemW),
  .Zeroflag(Zeroflag),
  .Negflag(Negflag),
  .WBdata(WBdata),
  .DataMemEn(DataMemEn)
);

initial begin
  // Initialize inputs
  clk = 0;
  OP = 6'b000000;
  next_state = 3'b000;
  state = 3'b000;
  Zeroflag = 0;
```

```
    Negflag = 0;

    // Apply stimulus
        #10 OP = 6'b000000; // Triggering AND case
        #10 OP = 6'b000001; // Triggering ADD case
        #10 OP = 6'b000010; // Triggering SUB case
    #10 OP = 6'b000011; // Triggering ANDI case
        #10 OP = 6'b000100; // Triggering ADDI case
        #10 OP = 6'b000101; // Triggering LW case
        #10 OP = 6'b000110; // Triggering LW.POI case
        #10 OP = 6'b000111; // Triggering SW case
    #10 OP = 6'b001000; // Triggering Branch case: BGT
    #10 OP = 6'b001001; // Triggering Branch case: BLT
    #10 OP = 6'b001010; // Triggering Branch case: BEQ
    #10 OP = 6'b001011; // Triggering Branch case: BNE
        #10 OP = 6'b001100; // Triggering JMP case
        #10 OP = 6'b001101; // Triggering CALL case
        #10 OP = 6'b001110; // Triggering RET case
        #10 OP = 6'b001111; // Triggering PUSH case
        #10 OP = 6'b010000; // Triggering POP case

    #170 $finish;
  end

  always #5 clk = ~clk; // Clock generation

  // Print statements
  initial begin
    $monitor("Time=%0t ->       state=%b, next_state=%b, OP=%b PCsrc=%b ALUop=%b ExtOp=%b Rs2Src=%b RegRw=%b Rs1Rw=%b ALUsrc=%b DataInputSrc=%b MemR=%b MemW=%b WBdata=%b DataMemEn=%b",
      $time, state, next_state, OP, PCsrc, ALUop, ExtOp, Rs2Src, RegRw, Rs1Rw, ALUsrc, DataInputSrc, MemR, MemW, WBdata, DataMemEn);
  end
```

```
endmodule
```

Appendix 13: ALU

```
/* ALU Arithmetic and Logic Operations
-----------------------------------------------------------------------
|ALUop|   ALU Operation
-----------------------------------------------------------------------
| 00  |   result = A and B;
-----------------------------------------------------------------------
| 01  |   result = A + B;
-----------------------------------------------------------------------
| 10  |   result = A - B;
-----------------------------------------------------------------------*/
module ALU (
  input [31:0] A,                  // ALU 32-bit Input A
  input [31:0] B,                  // ALU 32-bit Input B
  input [1:0] ALUop,               // 2-bit ALU Selection
  output reg [31:0] result, // ALU 32-bit Output
  output reg carry,                // Carry Out Flag
  output reg zero,                         // Zero Flag
  output reg negative,             // Negative Flag
  output reg overflow              // Overflow Flag
);

always @(A or B or ALUop) begin
   case (ALUop)
      2'b00: begin // AND
        result = A & B;
      end

      2'b01: begin // ADD
        result = A + B;
        carry = A[31] & B[31] | A[31] & ~result[31] | ~result[31] & B[31]; // Carry out
        overflow = A[31] & B[31] & ~result[31] | ~A[31] & ~B[31] & result[31]; // Overflow for addition
      end
```

```
        2'b10: begin // SUB
          result = A - B;
          carry = B > A; // Borrow in subtraction is the carry here
          overflow = ~A[31] & B[31] & result[31] | A[31] & ~B[31] & ~result[31]; // Overflow for subtraction
        end

        default: begin
          result = 32'b0;
          carry = 1'b0;
          zero = 1'b0;
          negative = 1'b0;
          overflow = 1'b0;
        end
      endcase
      zero = (result == 32'b0); // checks if the result is equal to zero
      negative = result[31]; // MSB as sign bit
    end
    endmodule
```

Appendix 14: ALU Testbench

```
    `timescale 1ns/1ps

    module ALU_testbench;

      reg [31:0] A, B;
      reg [1:0] ALUop;
      wire [31:0] result;
      wire carry, zero, negative, overflow;

      ALU uut (
        .A(A),
        .B(B),
        .ALUop(ALUop),
        .result(result),
```

```
  .carry(carry),
  .zero(zero),
  .negative(negative),
  .overflow(overflow)
);

// Clock generation
reg clk = 0;
always #5 clk = ~clk;

initial begin
        forever begin
    #3 A <= $urandom_range(0,255);
    #6 B <= $urandom_range(0,255);
    #12 ALUop <= $urandom_range(0,2) ;

            $display("Test Case:");
        $display("A = %h", A);
        $display("B = %h", B);
        $display("ALUop = %b", ALUop);
        $display("Result = %h", result);
        $display("Borrow = %b", carry);
            $display("Zero = %b", zero);
        $display("Negative = %b", negative);
        $display("Overflow = %b", overflow);
            $display("--------------------------");
   end
 end
 initial begin
  #1000;
  $finish ;
 end
endmodule
```

Appendix 15: Register File

```
module RegFile(RA, RB, RW, Bus_A, Bus_B, Bus_W, Bus_W1, clk, RegRw, Rs1Rw);
    input clk, RegRw, Rs1Rw;
    input [3:0] RA, RB, RW;
    input [31:0] Bus_W, Bus_W1;
    output reg [31:0] Bus_A, Bus_B;

    integer i = 0;
    reg [31:0] register_file [15:0]; // 16 32-bit general-purpose registers: from R0 to R15.

    initial begin
        for (i = 0; i < 16; i = i + 1)
            register_file[i] = 0;
    end

    always @(posedge clk) begin
                        // Write
        if (RegRw == 1 && Rs1Rw == 0) begin
            register_file[RW] <= Bus_W;
        end else if (RegRw == 1 && Rs1Rw == 1) begin
            register_file[RW] <= Bus_W;
            register_file[RA] <= Bus_W1;
        end else begin
                                // Read
            Bus_A <= register_file[RA];
            Bus_B <= register_file[RB];
                        end
    end
endmodule
```

Appendix 16: Register File Testbench

```
module RegFile_testbench;

  reg clk, RegRw, Rs1Rw;
```

```
reg [3:0] RA, RB, RW;
reg [31:0] Bus_W, Bus_W1;
wire [31:0] Bus_A, Bus_B;

// Instantiate the RegFile module
RegFile uut (
  .RA(RA),
  .RB(RB),
  .RW(RW),
  .Bus_A(Bus_A),
  .Bus_B(Bus_B),
  .Bus_W(Bus_W),
  .Bus_W1(Bus_W1),
  .clk(clk),
  .RegRw(RegRw),
  .Rs1Rw(Rs1Rw)
);

initial begin
  // Initialize inputs
  clk = 0;
  RegRw = 0;
  Rs1Rw = 0;
  RA = 0;
  RB = 0;
  RW = 0;
  Bus_W = 32'h0000_0000;
  Bus_W1 = 32'h0000_0000;

        $display("------------------------------");

  // Test Case 1: Write to register 3, Read from register 3
  #10 RegRw = 1; Rs1Rw = 0; RW = 3; Bus_W = 32'h1234_5678; // Write to register 3
  #10 RegRw = 0; RA = 3; // Read from register 3
```

```
        $display("-----------------------------");

    // Test Case 2: Write to registers 5 and 3, Read from registers 5 and 3
    #10 RegRw = 1; Rs1Rw = 1; RW = 5; Bus_W = 32'h8765_4321; Bus_W1 = 32'h1111_2222; // Write to registers 5 and 3
    #10 RegRw = 0; RA = 3; RB = 5; // Read from registers 3 and 5

        $display("-----------------------------");

    // Test Case 3: Write to register 0, Read from register 0
    #10 RegRw = 1; Rs1Rw = 0; RW = 0; Bus_W = 32'hAAAA_AAAA; // Write to register 0
    #10 RegRw = 0; RA = 0; // Read from register 0

        $display("-----------------------------");

    #100 $finish;
  end

  always #5 clk = ~clk; // Clock generation
  // Print statements
  initial begin
    $monitor("Time=%0t -> RegRw=%0d Rs1Rw=%0d RA=%0d RB=%0d RW=%0d Bus_W=%h Bus_W1=%h Bus_A=%h Bus_B=%h",
        $time, RegRw, Rs1Rw, RA, RB, RW, Bus_W, Bus_W1, Bus_A, Bus_B);
  end
endmodule
```

Appendix 17: IR

```
module IR(clk, inst, op_code, inst_rs1, inst_rs2, inst_rd, imm_16, jump_offset, Mode);
    input clk ;
    input [31:0] inst;
        output reg [5:0] op_code;
        output reg [3:0] inst_rs1 , inst_rs2 , inst_rd;
        output reg [15:0] imm_16;
```

```
        output reg [25:0] jump_offset;
        output reg [1:0] Mode;

 always @(posedge clk)
        begin
                op_code = inst[31:26];

                // R-type: AND, ADD, and SUB
                if (op_code == 6'b000000 || op_code == 6'b000001 || op_code == 6'b000010)begin
                        inst_rd = inst[25:22];
                        inst_rs1 = inst[21:18];
                        inst_rs2 = inst[17:14];
                end
                // I-type
                else if (op_code == 6'b000011 || op_code == 6'b000100 || op_code == 6'b000101 || op_code
== 6'b000110
                || op_code == 6'b000111 || op_code == 6'b001000 || op_code == 6'b001001 || op_code ==
6'b001010 || op_code == 6'b001011 )begin
                        inst_rd = inst[25:22];
                        inst_rs1 = inst[21:18];
                        imm_16 = inst[17:2];
                        Mode = inst[1:0];

                end
                // J-type: JMP & CALL
                else if (op_code == 6'b001100 || op_code == 6'b001101)begin
                        jump_offset = inst[25:0];

                end
                // S-type: PUSH & POP
                else if (op_code == 6'b001111 || op_code == 6'b010000)begin
                        inst_rd = inst[25:22];

                end
           else begin
```

```
                    // J-type: RET "26-bit unused"
                end
            end
        endmodule
```

Appendix 18: IR Testbench

```
`timescale 1ns/1ns
module IR_testbench;
  // Inputs
  reg clk;
  reg [31:0] inst;

  // Outputs
  wire [5:0] op_code;
  wire [3:0] inst_rs1, inst_rs2, inst_rd;
  wire [15:0] imm_16;
  wire [25:0] jump_offset;
  wire [1:0] mode;

  // Instantiate the module under test
  IR uut (
    .clk(clk),
    .inst(inst),
    .op_code(op_code),
    .inst_rs1(inst_rs1),
    .inst_rs2(inst_rs2),
    .inst_rd(inst_rd),
    .imm_16(imm_16),
    .jump_offset(jump_offset),
    .Mode(mode)
  );

  // Clock generation
  initial begin
    clk = 0;
```

```
    forever #5 clk = ~clk;
  end

  // Test cases
  initial begin
    // Test Case 1: R-type instruction -> SUB
    inst = 32'b00001001101000000100000000110011;
    #10;
    $display("Test Case 1 - R-type:");
    $display("op_code = %b, inst_rs1 = %b, inst_rs2 = %b, inst_rd = %b", op_code, inst_rs1, inst_rs2,
inst_rd);

    // Test Case 2: I-type instruction -> ANDI
    inst = 32'b00001100001100010000000000110011;
    #10;
    $display("Test Case 2 - I-type:");
    $display("op_code = %b, inst_rs1 = %b, imm_16 = %b, mode = %b", op_code, inst_rs1, imm_16, mode);

    // Test Case 3: J-type instruction -> JMP
    inst = 32'b00110000000011010000000110110011;
    #10;
    $display("Test Case 3 - J-type:");
    $display("op_code = %b, jump_offset = %b", op_code, jump_offset);

    // Test Case 4: S-type instruction
    inst = 32'b00111100100000000000000000111111;
    #10;
    $display("Test Case 4 - S-type:");
    $display("op_code = %b, inst_rd = %b", op_code, inst_rd);

    // End simulation
    $finish;
  end
endmodule
```

Appendix 19: Instruction Memory

```
module InstructionMemory(
    input [31:0] address, // Input address from the processor (word-addressed)
    output reg [31:0] instruction // Output instruction at the given address
);

        // Define the word size
        parameter WORD_SIZE = 32;

        // Define the size of the instruction memory in words
        parameter MEMORY_SIZE = 1024;

        // Declare the instruction memory as an array of words
        reg [WORD_SIZE-1:0] memory [0:MEMORY_SIZE-1];

    // Initialize the memory with the instructions
    initial begin
        for (int i = 0; i < 1024; i ++) begin
            memory[i] = 32'h00000000;
        end

        memory[0] = 32'b000011_0000_0000_0000000000000000_00; // ANDI $t0, $t0, 0

        memory[4] = 32'b000101_0001_0000_0000000000000000_00; // LW $t1, 0($t0)

        memory[8] = 32'b000101_0010_0000_0000000000000100_00; // LW $t2, 4($t0)

        memory[12] = 32'b000011_0011_0011_0000000000000000_00; // ANDI $t3, $t3, 0

        memory[16] = 32'b001010_0001_0000_0000000000001100_00; // BEQ $t1, $t0, Label_1

        memory[20] = 32'b001111_0001_00000000000000000000_00; // PUSH $t1

        memory[24] = 32'b001011_0011_0010_0000000000001000_00; // BNE $t3, $t2, Label_2
```

```
        memory[28] = 32'b000010_0001_0001_0010_00000000000000; // Label_1: SUB $t1, $t1, $t2

        memory[32] = 32'b010000_0001_11111111111001111111; // Label_2: POP $t4

        memory[36] = 32'b001001_0010_0001_0000000000001100_00; // BLT $t2, $t1, Label_3

        memory[40] = 32'b000111_0100_0000_0000000000001000_00; // SW $t4, 8($t0)

                memory[44] = 32'b001100_00000000000000000000101100; // End: JMP End

                memory[48] = 32'b000000_0011_0001_0010_00000000000000; // Label_3: AND $t3, $t1, $t2
    end

    // Fetch instruction based on address
    always @(address) begin
                // Check if the address is word-aligned
        if (address % (WORD_SIZE/8) == 0) begin
                // Read the instruction from memory
                instruction <= memory[address]; // Word-aligned address
        end else begin
                        instruction <= 32'h00000000;
        end
    end
endmodule
```

Appendix 20: Instruction Memory Testbench

```
`timescale 1ns / 1ns
module InstructionMemory_testbench;
  // Inputs
  reg [31:0] address;

  // Outputs
  reg [31:0] instruction;
```

```
// Clock generation
reg clk;

// Instantiate the module
InstructionMemory IM (
  .address(address),
  .instruction(instruction)
);

// Clock generation process
always begin
  #5 clk = ~clk;
end

// Initial block for testbench setup
initial begin
  // Initialize inputs
  address = 0;
  clk = 0;

  // Display header
  $display("Time\tAddress\tInstruction");

  // Run simulation for multiple addresses
  // You can add more test cases as needed
  repeat (10) begin
    #5; // Wait for a few time units
    $display("%0t\t%h\t%h", $time, address, instruction);
          address = address + 4; // Increment address by 4 for the next test
  end

  // End simulation
  $stop;
```

```
  end
endmodule
```

Appendix 21: Data Memory

```
module data_memory_with_stack (
    input wire clk,
    input wire mem_write,
    input wire mem_read,
    input wire dataMemEnable,       // Push to stack
    input wire [31:0] address,      // Address for memory operations
    input wire [31:0] data_in,      // Data to write to memory or push to stack
    output reg [31:0] data_out      // Data read from memory or stack
);

    parameter DATA_MEM_SIZE = 256;
    parameter STACK_START = 200;     // Define start address of stack in memory
    parameter STACK_SIZE = 56;       // Define size of the stack

    reg [31:0] memory [0:DATA_MEM_SIZE-1];
    reg [31:0] sp = STACK_START;

            initial    begin
    // save 0x0002 at address = 0
      memory[0] = 'h02;

            // save 0x0001 at address = 4
      memory[4] = 'h01;

            // save 0x0000 at address = 8
      memory[8] = 'h00;
            end

    always @(posedge clk) begin
       // Handle data memory operations
```

```
        if (mem_write == 1 && mem_read == 0 && sp < STACK_START + STACK_SIZE &&
dataMemEnable == 0) begin
            // Push case: The stack has available location (not full), and we want to push into the stack
            memory[sp] <= data_in;
            sp = sp + 4;
        end

        else if (mem_write == 0 && mem_read == 1 && sp > STACK_START && dataMemEnable == 0)
begin
            // Pop case: The stack is not empty, and we want to pop from the stack
            sp = sp - 4;
            data_out <= memory[sp];
        end

        else if (mem_write == 1 && mem_read == 0 && dataMemEnable == 1) begin
            // Store into the static memory
            memory[address[31:2]] <= data_in;    // Shift left by 2 (division by 4) for word-addressable memory
        end

        else if (mem_write == 0 && mem_read == 1 && dataMemEnable == 1) begin
            // Load from the static memory
            data_out <= memory[address[31:2]];   // Shift left by 2 (division by 4) for word-addressable memory
        end
    end
endmodule
```

Appendix 22: Data Memory Testbench

```
`timescale 1ns / 1ps
module data_memory_with_stack_tb;
    reg clk;
    reg mem_write;
    reg mem_read;
    reg dataMemEnable;
    reg [31:0] address;
    reg [31:0] data_in;
```

```
wire [31:0] data_out;

// Instantiate the Unit Under Test (UUT)
data_memory_with_stack uut (
    .clk(clk),
    .mem_write(mem_write),
    .mem_read(mem_read),
    .dataMemEnable(dataMemEnable),
    .address(address),
    .data_in(data_in),
    .data_out(data_out)
);

// Clock generation
initial begin
    clk = 0;
    forever #10 clk = ~clk; // Toggle clock every 10 ns
end

// Test case generation
initial begin
    // Initialize Inputs
    mem_write = 0;
    mem_read = 0;
    dataMemEnable = 0;
    address = 0;
    data_in = 0;

    // Wait for global reset
    #100;

    // Test 1: Write to memory
    mem_write = 1;
    mem_read = 0;
```

```
        dataMemEnable = 1; // Enable static memory
        address = $random % 256;
        data_in = $random;
        #20;
        $display("Time: %t,Write to Memory-Address:%d, Data In:%d",$time, address, data_in);

        // Test 2: Read from memory
        mem_write = 0;
        mem_read = 1;
        #20;
        $display("Time:%t,Read from Memory-Address:%d, Data Out:%d", $time, address, data_out);

        // Test 3: Push to stack
        mem_write = 1;
        mem_read = 0;
        dataMemEnable = 0; // Enable stack operation
        data_in = $random;
        #20;
        $display("Time:%t,Push to Stack-Data In:%d", $time, data_in);

        // Test 4: Pop from stack
        mem_write = 0;
        mem_read = 1;
        #20;
        $display("Time: %t,Pop from Stack-Data Out:%d", $time, data_out);

        // Add more tests as needed for edge cases (e.g., stack overflow, stack underflow)

        // Finish simulation
        #1000;
        $finish;
    end
endmodule
```

Appendix 23: Extender

```
module sign_extender (Imm_16, ExtOp, ExtOut);
        input wire [15:0] Imm_16;
    input wire ExtOp;
    output reg [31:0] ExtOut ;
        always @(*) begin
    if (ExtOp) begin
        // Sign-extend Imm_16
        ExtOut = {{16{Imm_16[15]}}, Imm_16};
    end else begin
        // Zero-extend Imm_16
        ExtOut = {16'b0, Imm_16};
    end
end
endmodule
```

Appendix 24: Extender Testbench

```
`timescale 1ns / 1ps
module sign_extender_testbench;
reg [15:0] Imm_16;
reg ExtOp;
wire [31:0] ExtOut;

// Instantiate the Unit Under Test (UUT)
sign_extender uut (
    .Imm_16(Imm_16),
    .ExtOp(ExtOp),
    .ExtOut(ExtOut)
);

initial begin
    // Initialize Inputs
    Imm_16 = 0;
    ExtOp = 0;
```

```
    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here
    repeat (10) begin
        Imm_16 = $random;
        ExtOp = $random % 2; // Randomly generate 0 or 1
        #10; // Wait 10 ns between changes
        $display("Time: %t, Imm_16: %h, ExtOp: %b, ExtOut: %h", $time, Imm_16, ExtOp, ExtOut);
    end
    $finish;
end
endmodule
```

Appendix 25: Concatenation

```
module concatenation(
  // Module ports and parameters
  input [31:0] PC,
  input [25:0] Immediate26,
  output [31:0] ConcatenatedResult
);
  assign ConcatenatedResult = {PC[31:26], Immediate26};
endmodule
```

Appendix 26: Concatenation Testbench

```
module concatenation_testbench;
  // Parameters
  parameter DELAY = 1;

  // Signals
  reg [31:0] PC;
  reg [25:0] Immediate26;
  wire [31:0] ConcatenatedResult;
```

```
// Instantiate the module
concatenation uut (
  .PC(PC),
  .Immediate26(Immediate26),
  .ConcatenatedResult(ConcatenatedResult)
);

// Initial block for testbench stimulus
initial begin
  // Test case 1
  PC = 32'hAABBCCDD;
  Immediate26 = 26'h123456;

  // Display input values
  $display("Test Case 1:");
  $display("PC = 0x%h", PC);
  $display("Immediate26 = 0x%h", Immediate26);

  // Wait for some time
  #DELAY;

  // Display output value
  $display("ConcatenatedResult = 0x%h", ConcatenatedResult);

  // Test case 2
  PC = 32'h11223344;
  Immediate26 = 26'hABCDEF;

  // Display input values
  $display("Test Case 2:");
  $display("PC = 0x%h", PC);
  $display("Immediate26 = 0x%h", Immediate26);

  // Wait for some time
```

```
    #DELAY;

    // Display output value
    $display("ConcatenatedResult = 0x%h", ConcatenatedResult);

    // Add more test cases as needed

    // Stop simulation
    $stop;
  end
endmodule
```

Appendix 27: CPU

```
module cpu( clk , inst_Din);
    input clk ;
    input [31:0] inst_Din ;

    parameter IF_STAGE = 3'b000;
    parameter ID_STAGE = 3'b001;
    parameter EX_STAGE = 3'b010;
    parameter MEM_STAGE = 3'b011;
    parameter WB_STAGE = 3'b100;

   //PC
  reg [31:0] pc_next_out , pc_next_in ;
  reg [31:0] pc_J_out , pc_J_in;
  reg [31:0] pc_BTA_out , pc_BTA_in;
  reg [31:0] pc_stack_in , pc_stack_out ;
  reg [1:0] PCsrc;
  reg [31:0]pc_in , pc_out;

  //Instruction memory
  reg [31:0] inst_mem_out;

  //IR
```

```
reg [5:0] op_code;
reg [3:0] inst_rs1 , inst_rs2 , inst_rd;
reg [15:0] imm_16;
reg [25:0] J_offset;
reg [1:0] mode;

//extender
reg [31:0] ext_reg_in , ext_reg_out ;
reg EX_OP;

//Mux of register file
reg [3:0] mux_src_rf_out;
reg RS2src;

//register file
reg [31:0] BusA_in, BusA_out;
reg [31:0] BusB_in, BusB_out;
reg [31:0] BusW,BusW1;
reg RegRw,Rs1Rw;

//Mux of ALU
reg ALUSrc;
reg [31:0] mux_src_ALU_out;

//ALU
reg [31:0] ALU_res_in, ALU_res_out;
reg [3:0] ALU_flags_in, ALU_flags_out; // 0:carry  1:zero  2:negative  3:overflow
reg [1:0] ALUop;

//Mux of Data Memory
reg DataInputSrc;
reg [32-1:0] mux_DataInputSrc_out;

//Data Memory
```

```
 reg [31:0] D_mem_reg_in, D_mem_reg_out;
 reg MemR, MemW,DataMemEn;

 //Mux of WBdata
 reg WBdata;

 reg [2:0] state = 3'b000, next_state = 3'b000;

 int i = 0;
 always @(negedge clk) begin
  if (i < 1) begin
     i++;
     state <= 3'b000; // Set the initial state
  end else begin
     state <= next_state;
     // $display("State = %0d, next_state = %0d, op_code = %b", state, next_state, op_code);
  end
end

always @(negedge clk) begin
     case (state)
       3'b000: begin
          // Define FETCH behavior here...
          next_state = 3'b001;
                              $display("State = %b , next_state = %b" , state , next_state);
       end

       3'b001: begin
                    if(op_code==6'b001100)begin // JMP
                                             next_state = 3'b000;
                                  end
                                  else begin
                                             next_state = 3'b010; //other instructions
                                  end
```

```
        end
        3'b010: begin
          if (op_code==6'b0010??) begin     //BGT + BLT + BEQ + BNE
                                    next_state = 3'b000;
                            end
         else if (op_code==6'b00000? || op_code==6'b000010 || op_code==6'b000011 || op_code==6'b000100) begin       //R_type + ANDI + ADDI
                                    next_state = 3'b100;
                            end
                            else     //LW + LW.POI + SW + CALL + RET + PUSH + POP
                                    next_state =3'b011;
                            end

        3'b100: begin         //R-type + ADDI + ANDI + LW + LW.POI +POP
           next_state = 3'b000;
        end

        3'b011 : begin
                                  if(op_code==6'b000101 || op_code==6'b000110 || op_code==6'b010000 )begin //LW + LW.POI + POP
                                    next_state = 3'b100;
                                  end
                        else //CALL + RET + SW +PUSH
                                  begin
                                    next_state = 3'b000;
                           end
                        end
        default: begin
           next_state = 3'b000;
        end
      endcase

    end
```

```
controlUnit CU (
   .clk(clk),
   .next_state(next_state),
   .state(state),
   .PCsrc(PCsrc),
   .OP(op_code),
   .ExtOp(EX_OP),
   .Rs2Src(RS2src),
   .RegRw(RegRw),
   .Rs1Rw(Rs1Rw),
   .ALUsrc(ALUSrc),
   .DataInputSrc(DataInputSrc),
   .ALUop(ALUop),
   .MemR(MemR),
   .MemW(MemW),
   .Zeroflag(ALU_flags_out[1]),
   .Negflag(ALU_flags_out[2]),
   .WBdata(WBdata),
   .DataMemEn(DataMemEn)
 );



  //dff
 dFlipFlop #(32) next_pc (.data_input(pc_next_in) , .clock(clk) , .data_output(pc_next_out));
 always @(posedge clk) begin
            $display("----------------------------------------------------------------------------------------------------
--");
            $display("D Flip Flop for PC+1: pc_next_out = %b" , pc_next_out);
 end

 dFlipFlop #(32)next_pc_JTA (.data_input(pc_J_in) , .clock(clk) , .data_output(pc_J_out) );
 always @(posedge clk) begin
```

```
        $display("------------------------------------------------------------------------------------------------------
--");
        $display("D Flip Flop for Jump Target Address: pc_J_out = %b" , pc_J_out);
 end

 dFlipFlop #(32)next_pc_BTA (.data_input(pc_BTA_in) , .clock(clk) , .data_output(pc_BTA_out));
 always @(posedge clk) begin
        $display("------------------------------------------------------------------------------------------------------
--");
        $display("D Flip Flop for Branch Target Address: pc_BTA_out = %b" , pc_BTA_out);
 end

 dFlipFlop #(32)next_pc_stack (.data_input(pc_stack_in) , .clock(clk) , .data_output(pc_stack_out));
 always @(posedge clk) begin
        $display("------------------------------------------------------------------------------------------------------
--");
        $display("D Flip Flop for Stack: pc_stack_out = %b" , pc_stack_out);
 end

 //mux_pc
 mux4x1                                                                                  mux_pc
(.selectionLine(PCsrc),.a(pc_next_out),.b(pc_J_out),.c(pc_BTA_out),.d(pc_stack_out),.result(pc_in));
 always @(posedge clk) begin
        $display("------------------------------------------------------------------------------------------------------
--");
        $display("Mux of PC: PCsrc = %b, pc_in=%b" , PCsrc, pc_in);
 end

 //pc
 PC pc_u (.clk(clk),.state(state),.PC_input(pc_in),.PC_output(pc_out));
 always @(posedge clk) begin
        $display("------------------------------------------------------------------------------------------------------
--");
        $display("PC module: state = %b, pc_in=%b, pc_out=%b" , state, pc_in, pc_out);
 end
```

```
  adder_32bit adder_next_pc (.input1(pc_out),.input2(4), .adder_output(pc_next_in));
  always @(posedge clk) begin
     $display("-----------------------------------------------------------------------------------------------------");
          $display("To find PC+1: pc_out = %b, pc_next_in=%b" , pc_out, pc_next_in);
  end


  adder_32bit Adder (.input1(pc_out),.input2(ext_reg_out), .adder_output(pc_BTA_in));
  always @(posedge clk) begin
          $display("-----------------------------------------------------------------------------------------------------
--");
          $display("To find BTA: pc_out = %b, ext_reg_out=%b, pc_BTA_in=%b" , pc_out, ext_reg_out,
pc_BTA_in);
  end


  concatenation uut (.PC(pc_out), .Immediate26(imm_16), .ConcatenatedResult(pc_J_in));
  always @(posedge clk) begin
          $display("-----------------------------------------------------------------------------------------------------
--");
          $display("To find JTA: pc_out = %b, imm_16=%b, pc_J_in=%b" , pc_out, imm_16, pc_J_in);
  end


  InstructionMemory  IM(.address(pc_out),.instruction(inst_mem_out));
  always @(posedge clk) begin
         $display("-----------------------------------------------------------------------------------------------------
-");
    $display("pc_out = %b, inst_mem_out=%b" , pc_out, inst_mem_out);
  end



  IR  ir  (.clk(clk),.inst(inst_mem_out),  .op_code(op_code),  .inst_rs1(inst_rs1),  .inst_rs2(inst_rs2),
.inst_rd(inst_rd),.imm_16(imm_16),.jump_offset(J_offset), .Mode(mode));
  always @(posedge clk) begin
          $display("-----------------------------------------------------------------------------------------------------
-");
    $display("inst_mem_out=%b,  op_code=%b,  inst_rs1=%b,  inst_rs2=%b,  inst_rd=%b,  imm_16=%b,
J_offset=%b, mode=%b", inst_mem_out, op_code, inst_rs1, inst_rs2, inst_rd, imm_16, J_offset, mode);
```

```
  end

  sign_extender extender (.Imm_16(imm_16),  .ExtOp(EX_OP),.ExtOut(ext_reg_in));
  always @(posedge clk) begin
    $display("-----------------------------------------------------------------------------------------------------------");
    $display("imm_16 = %b, EX_OP=%b, ext_reg_in=%b" , imm_16, EX_OP, ext_reg_in);
  end

  dFlipFlop #(32)reg_extender(.data_input(ext_reg_in) , .clock(clk) , .data_output(ext_reg_out));
  always @(posedge clk) begin
        $display("-----------------------------------------------------------------------------------------------------------
-");
    $display("ext_reg_in= %b, ext_reg_out=%b" ,ext_reg_in,ext_reg_out);
  end

  mux2x1 Mux_RF (.selectionLine(RS2src),.a(inst_rs2),.b(inst_rd),.result(mux_src_rf_out));
  always @(posedge clk) begin
        $display("-----------------------------------------------------------------------------------------------------------
-");
    $display("RS2src= %b, inst_rs2=%b, inst_rd=%b,mux_src_rf_out=%b"
,RS2src,inst_rs2,inst_rd,mux_src_rf_out);
  end

  RegFile RF
(.RA(inst_rs1),.RB(mux_src_rf_out),.RW(inst_rd),.Bus_A(BusA_in),.Bus_B(BusB_in),.Bus_W(BusW),.
Bus_W1(BusW1),.clk(clk),.RegRw(RegRw), .Rs1Rw(Rs1Rw));
  always @(posedge clk) begin
        $display("-----------------------------------------------------------------------------------------------------------
-");
    $display("inst_rs1= %b, mux_src_rf_out=%b, inst_rd=%b,BusA_in=%b BusB_in= %b, BusW=%b,
BusW1=%b,RegRw=%b,Rs1Rw=%b"
,inst_rs1,mux_src_rf_out,inst_rd,BusA_in,BusB_in,BusW,BusW1,RegRw,Rs1Rw);
  end

  dFlipFlop #(32)reg_BusA(.data_input(BusA_in) , .clock(clk) , .data_output(BusA_out));
  always @(posedge clk) begin
```

```
        $display("-----------------------------------------------------------------------------------------------------
-");
   $display("BusA_in= %b, BusA_out=%b" ,BusA_in,BusA_out);
  end

  dFlipFlop #(32)reg_BusB(.data_input(BusB_in) , .clock(clk) , .data_output(BusB_out));
  always @(posedge clk) begin
        $display("-----------------------------------------------------------------------------------------------------
-");
   $display("BusB_in= %b, BusB_out=%b" ,BusB_in,BusB_out);
  end

  mux2x1 Mux_WBdata (.selectionLine(WBdata),.a(ALU_res_out),.b(D_mem_reg_out),.result(BusW));
   always @(posedge clk) begin
        $display("-----------------------------------------------------------------------------------------------------
-");
   $display("WBdata= %b, ALU_res_out=%b ,D_mem_reg_out= %b, BusW=%b"
,WBdata,ALU_res_out,D_mem_reg_out,BusW);
  end

  mux2x1 Mux_ALU (.selectionLine(ALUSrc),.a(BusB_out),.b(ext_reg_out),.result(mux_src_ALU_out));
   always @(posedge clk) begin
        $display("-----------------------------------------------------------------------------------------------------
-");
   $display("ALUSrc= %b, BusB_out=%b ,ext_reg_out= %b, mux_src_ALU_out=%b"
,ALUSrc,BusB_out,ext_reg_out,mux_src_ALU_out);
  end

  mux2x1 Mux_DataInputSrc
(.selectionLine(DataInputSrc),.a(BusB_out),.b(pc_next_in),.result(mux_DataInputSrc_out));
   always @(posedge clk) begin
        $display("-----------------------------------------------------------------------------------------------------
-");
   $display("DataInputSrc= %b, BusB_out=%b ,pc_next_in= %b, mux_DataInputSrc_out=%b"
,DataInputSrc,BusB_out,pc_next_in,mux_DataInputSrc_out);
  end
```

```
 ALU alu ( .A(BusA_out),.B(mux_src_ALU_out),.ALUop(ALUop),.result(ALU_res_in),
.carry(ALU_flags_in[0]),.zero(ALU_flags_in[1]),.negative(ALU_flags_in[2]),.overflow(ALU_flags_in[3
]));

  always @(posedge clk) begin

        $display("-----------------------------------------------------------------------------------------------
-");

   $display("BusA_out= %b, mux_src_ALU_out=%b ,ALUop= %b, ALU_res_in=%b ,ALU_flags_in[0]=
%b, ALU_flags_in[1]=%b ,ALU_flags_in[2]= %b, ALU_flags_in[3]=%b"
,BusA_out,mux_src_ALU_out,ALUop,ALU_res_in,ALU_flags_in[0],ALU_flags_in[1],ALU_flags_in[2]
,ALU_flags_in[3]);

 end


 data_memory_with_stack Data_Mem
(.clk(clk),.mem_write(MemW),.mem_read(MemR),.dataMemEnable(DataMemEn),.address(ALU_res_in
),.data_in(mux_DataInputSrc_out),.data_out(D_mem_reg_in));

   always @(posedge clk) begin

        $display("-----------------------------------------------------------------------------------------------
-");

   $display("MemW= %b, MemR=%b ,DataMemEn= %b, ALU_res_in=%b ,mux_DataInputSrc_out= %b,
D_mem_reg_in=%b" ,MemW,MemR,DataMemEn,ALU_res_in,mux_DataInputSrc_out,D_mem_reg_in);

 end


 dFlipFlop #(4)reg_AluFlags (.data_input(ALU_flags_in), .clock(clk) , .data_output(ALU_flags_out) );

   always @(posedge clk) begin

        $display("-----------------------------------------------------------------------------------------------
-");

   $display("ALU_flags_in= %b, ALU_flags_out=%b" ,ALU_flags_in,ALU_flags_out);

 end


 dFlipFlop #(32)reg_AluResult(.data_input(ALU_res_in), .clock(clk), .data_output(ALU_res_out) );

   always @(posedge clk) begin

        $display("-----------------------------------------------------------------------------------------------
-");

   $display("ALU_res_in= %b, ALU_res_out=%b" ,ALU_res_in,ALU_res_out);

 end


 dFlipFlop #(32)reg_DataMem(.data_input(D_mem_reg_in), .clock(clk) , .data_output(D_mem_reg_out))
;
```

```
    always @(posedge clk) begin
        $display("-----------------------------------------------------------------------------------------------------
-");
    $display("D_mem_reg_in= %b, D_mem_reg_out=%b" ,D_mem_reg_in,D_mem_reg_out);
  end
endmodule
```

Appendix 28: CPU Testbench

```
module cpu_testbench;
  reg clk;
  reg [31:0] inst_Din;
  // Instantiate the CPU module
  cpu dut (
    .clk(clk),
    .inst_Din(inst_Din)
  );

  // Clock generation
  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

  // Test vector generation
  initial begin
    // Test case 1
    inst_Din = 32'b000000_1010_0101_1111_11110110010000; // Replace this with your instruction
    #450; // Wait for a clock cycle or two for the system to start processing
    $display("Test Case 1 Results:");
    $display("Current State: %b", dut.state);
    $display("Next State: %b", dut.next_state);
    $display("op_code: %b", dut.op_code);
    $display("inst_rs1: %b", dut.inst_rs1);
    $display("inst_rs2: %b", dut.inst_rs2);
    $display("inst_rd: %b", dut.inst_rd);
```

```
    // Finish simulation after all test cases
    $stop;
  end
endmodule
```