BERZIET UNIVERSITY
FACULTY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER ENGINEERING

Computer Architecture-ENCS4370

Project #2

Prepared by

Lama Rimawi-1213515 **Amal Ziad-1192141**

Section#2

Instructor

Ayman Hroub

BIRZEIT

20– June – 2024

1. Abstract:

This project focuses on the designing and implementing a pipelined MIPS RISC processor in the Computer Architecture course. The processor follows the MIPS Reduced Instruction Set Computer (RISC) architecture and incorporates a subset of the MIPS instruction set. It divides the data path into five stages, including instruction fetch, decode, execute, memory access, and writeback, between each stage there’s a pipeline to keep the necessary values and registers. The RTL description provides the detailed RTL representation for different instruction types: R-Type, I-Type, J-Type, and S-Type. The control unit plays an important role in generating control signals based on the instruction type and opcode, which ensures proper coordination and efficient instruction execution. We also implemented Hazards detector, stall cycles handler, and data forwarding logic. In general, this project achieved an efficient pipelined MIPS design that supports many instructions of different types.

Contents

Table of figures

Table of tables

2. Design and implementation

1.1. Instruction fetch stage

1.1.1. PC register

The PC register is used to fetch instructions according to their place in instruction memory. In our implementation, it has the following inputs: clk, rst, PC next, and PC write. It depends on positive edge of clk to fetch each instruction, it also chooses the next instruction according to the source that will be explained in MUX4x1 to choose PC source. it outputs the address of the wanted instruction.

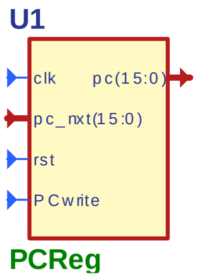


Figure 2-1 PC register

1.1.2. Instruction memory

The instruction memory contains the instructions that are stored in each entry as byte, which means each instruction is stored in two bytes or entries in the memory. After receiving PC which contains the wanted instruction's upper byte, this component concatenate the upper and lower bytes together to get the full instruction which is 16 bits.

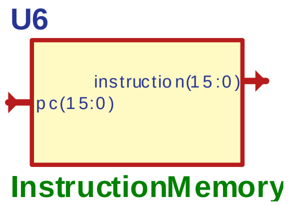


Figure 2-2 Instruction memory

1.1.3. Instruction decoder

The instruction decoder is used to split the received full instruction which is 16 bits, splitting it into their format and distinguish format type, operation code, mode (if any), registers, immediates, and also it passes current PC value to use it if necessary (it will be necessary in call and return operations). I used decoding or breaking the instruction in instruction fetch stage because by doing this way, each stage can independently handle its specific task, which improves overall efficiency and throughput. This modular approach allows the processor to execute multiple instructions simultaneously, which also increases instruction throughput and overall performance of the system.

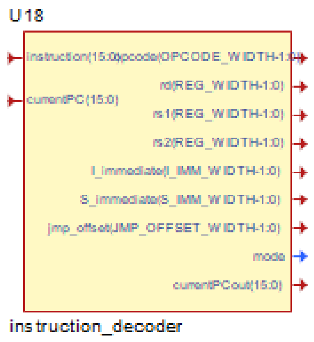


Figure 2-3 Instruction decoder

1.1.4. MUX4x1 to choose PC source

This 4x1 mux chooses the PC source of the next PC value, it depends on opcode to judge on which source, this is handled in signals controller unit according to the opcode. The first source is the default which is PC + 2. The second source is from branch operations it conditions matched. The third source is from jump logic which comes from call and jmp instructions. The forth source is from return instruction which takes the value of next instruction before call that is stored in R7.

1.1.5. Jump Logic

Jump logic concats the 6 most significant bits of PC with jump offset 10 bits, considering 1 left shift to maintain 16 bit length of the outputted PC target. This component is controlled by jump signal which is only active for J – type instructions such as jmp, call but not return.

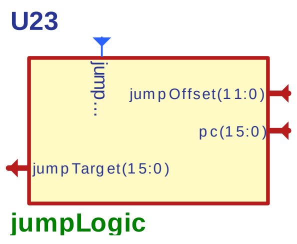


Figure 2-4 jump logic

1.1.6. Branch logic

If branch conditions matched, the current PC will be added to the extended immediate to branch.

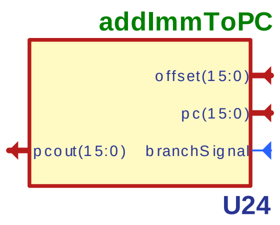


Figure 2-5 Branch logic

1.1.7. Return logic

If return instruction indicated, then the component will retrieve the stored PC from R7 in register file then add 2 to go to the next instruction after the one that called, then shift left by 1 to maintain the length of 16 bits.

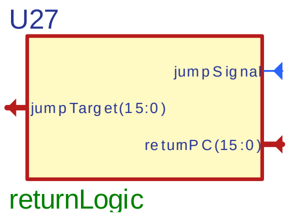


Figure 2-6 return logic

1.2. IF/ID pipeline

The IF/ID pipeline register is essential in a pipelined CPU, holding the fetched instruction and associated data as they move from the Instruction Fetch (IF) stage to the Instruction Decode (ID) stage. This register ensures smooth transitions and helps maintain the pipeline's efficiency.

Functionality

The IF/ID pipeline register captures and holds the instruction, register addresses, immediate values, and other data on each clock cycle, provided the write enable signal (`IFIDwr`) is active. It also has a reset (`rst`) signal to initialize its values.

Here is a simplified component of the IF/ID pipeline register:

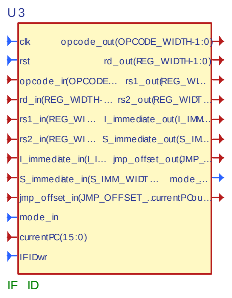


Figure 2-7 IF/ID PIPLINE

Inputs:

- `IFIDwr`: Write enable signal.
- `clk`: Clock signal.
- `rst`: Reset signal.
- `opcode_in`: Fetched opcode.

- `rd_in`, `rs1_in`, `rs2_in`: Register addresses.
- `I_immediate_in`, `S_immediate_in`, `jmp_offset_in`: Immediate values.
- `mode_in`: Mode signal.
- `currentPC`: Program counter value.

Outputs:

- `opcode_out`, `rd_out`, `rs1_out`, `rs2_out`: Outputs to decode stage.
- `I_immediate_out`, `S_immediate_out`, `jmp_offset_out`: Immediate values to decode stage.
- `mode_out`: Mode signal to decode stage.
- `currentPCout`: Program counter value to decode stage.

So the IF/ID pipeline register ensures the fetched instruction and associated data are correctly passed to the decode stage, maintaining pipeline efficiency and performance

1.3. Instruction decode stage

The Instruction Decode (ID) stage is a crucial part of a CPU's pipeline. This stage interprets the instruction fetched from memory, retrieves the necessary data from registers, and prepares control signals for the next stages. This process ensures the correct execution of instructions and smooth operation of the CPU.

Key Functions

Decoding the Instruction:

1. The instruction is split into its constituent fields: opcode, source registers, destination register, and immediate values.
2. The opcode determines the type of operation to be performed.

Reading Registers:

1. The source register values are read from the register file based on the decoded instruction.
2. These values are used in subsequent stages like Execute (EX) for operations.

Sign-Extending Immediate Values:

1. Immediate values in the instruction are sign-extended to match the processor's word size.
2. This is crucial for handling various instruction formats (I-type, S-type, etc.).

Generating Control Signals:

1. The control unit generates signals that direct the operations of the ALU, memory access, and write-back stages.
2. These signals include ALU operation codes, memory read/write controls, and register write enables.

Hazard Detection:

1. The ID stage also checks for data hazards (situations where instructions depend on the results of previous instructions).
2. It manages pipeline stalls or forwarding to ensure correct execution without conflicts.

So the ID stage is vital for interpreting instructions, preparing data, and setting up control signals. This stage ensures that the subsequent stages (like Execute) have all the information and data needed for correct operation. Proper functioning of the ID stage is crucial for the overall efficiency and performance of a pipelined CPU.

1.3.1. Register file

The register file is a crucial component in the Instruction Decode (ID) stage of a CPU. It stores the data for the registers that the CPU uses for operations. This module allows reading from and writing to the registers, ensuring that the correct data is available for the CPU's operations.

Functionality

The register file performs the following key functions:

1. **Reading Data:** Provides data from the source registers specified by the instruction.
2. **Writing Data:** Updates the destination register with the result of operations or other data.
3. **Handling Special Registers:** Manages special registers, like the program counter (PC), which may have unique update rules.
4.

Here is the Component for the register file:

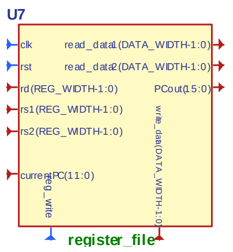


Figure 2-8 refile

Key Points Explained

Reading Data:

1. The module reads the values from the source registers specified by `rs1` and `rs2`.
2. These values are provided as `read_data1` and `read_data2` for use in subsequent stages of the pipeline.

Writing Data:

1. Data is written to the destination register specified by `rd` when the `reg_write` signal is active.
2. This allows the results of arithmetic operations, memory reads, or other instructions to be stored back in the register file.

Handling Special Registers:

1. The program counter (PC) is often treated as a special register.
2. In this module, if the `currentPC` signal is provided, it updates a specific register (assumed to be `regs[7]`).
3. This ensures that the current program counter value is always available for use.

So the register file is an integral part of the CPU, providing storage and access to register data required for instruction execution. It supports both reading and writing of register data and includes special handling for the program counter, ensuring smooth and efficient operation of the CPU pipeline.

1.3.2. MUX2x1 to choose destination register Rd

The `mux2x1Rd` module selects between two 3-bit input signals (`r1` and `r2`) using a single-bit select signal (`sel`). The chosen input is then sent to the output (`selectedR`).

Here is the Component for the MUX2x1 RD :

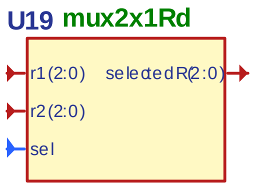


Figure 2-9 MUX2x1RD

Inputs and Outputs

Inputs:

- `r1`: A 3-bit input signal
- `r2`: Another 3-bit input signal
- `sel`: A control signal that determines which input to select

Output:

- `selectedR`: The selected 3-bit output signal

Functional Explanation

- **When** `sel` **is** `0`**:** The output `selectedR` takes the value of the input `r1`.
- **When** `sel` **is** `1`**:** The output `selectedR` takes the value of the input `r2`.

This behavior is controlled by an always block that continuously monitors the inputs and updates the output based on the value of `sel`.

1.3.3. 2 MUX4x1 for forwarding logic use

The 4-to-1 multiplexer module (`MUX4to1`) in Verilog, used for selecting one of four 16-bit input signals based on a 2-bit select signal.

Module Description

The `MUX4to1` module selects between four 16-bit input signals (`in0`, `in1`, `in2`, `in3`) using a 2-bit select signal (`sel`). The chosen input is sent to the output (`out`).

Here is the Component for the MUX4x1:

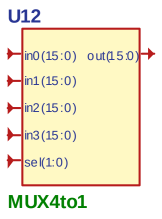


*Figure 2-10Mux4*1*

Inputs and Outputs

Inputs:

- `in0`: A 16-bit input signal
- `in1`: Another 16-bit input signal
- `in2`: Another 16-bit input signal
- `in3`: Another 16-bit input signal
- `sel`: A 2-bit control signal that determines which input to select

Output:

- `out`: The selected 16-bit output signal

1.4. ID/EX pipeline

ID/EX pipeline passes through it the following registers and values: extended immediate, register A which is chosen between 4 values before entering the pipeline, register B which is the same concept as A, and finally destination register Rd. this pipeline ensures keeping on data between instruction decode and execution stage. The required signals for next stages are also passed through this pipeline.

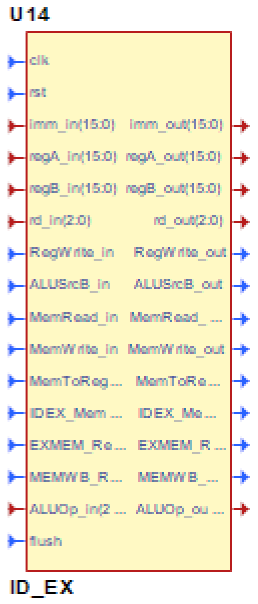


Figure 2-11 ID/EX pipeline

1.5. Execution stage

1.5.1. MUX2x1 to choose ALU operand B

The same concept of Muxes, this component is used here to choose the operand B of ALU between immediate and register B output from ID/EX pipeline. The mux is controlled with ALU source B signal which will be explained in Signals controller unit.

1.5.2. ALU unit

The ALU performs different arithmetic and logical operations based on the control signal ALU operation. The operations include ADD, SUB, AND, OR. The result of the operation is stored in the Result register.

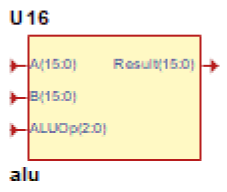


Figure 2-12 ALU

1.6. EX/MEM pipeline

The `EX_MEM` module captures and holds the outputs of the ALU, a register value, and a destination register identifier at each clock cycle, enabling the smooth flow of data between pipeline stages.

Here is the Component for the EX/MEM pipline :

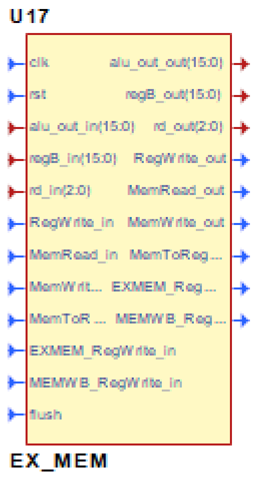


Figure 2-13 EX/MEM

Inputs and Outputs

Inputs:

- `clk`: Clock signal
- `rst`: Reset signal
- `alu_out_in`: 16-bit input from the ALU
- `regB_in`: 16-bit input from the B register
- `rd_in`: 3-bit input representing the destination register

Outputs:

- `alu_out_out`: 16-bit output to the MEM stage
- `regB_out`: 16-bit output to the MEM stage

    - `rd_out`: 3-bit output representing the destination register in the MEM stage

Functional Explanation

- **Reset (`rst`) is high:** The outputs (`alu_out_out`, `regB_out`, `rd_out`) are set to zero.
- **Reset (`rst`) is low:** On the rising edge of the clock (`clk`), the outputs capture the input values:
    - `alu_out_out` is set to `alu_out_in`.
    - `regB_out` is set to `regB_in`.
    - `rd_out` is set to `rd_in`.

This behavior ensures that the values from the Execute stage are correctly passed to the Memory stage on each clock cycle.

1.7. Memory access stage

The Memory Access (MEM) stage is a critical part of a pipelined processor architecture. It is responsible for reading from or writing to memory, depending on the type of instruction being executed.

Purpose

The MEM stage handles:

- **Reading data from memory** for load instructions.
- **Writing data to memory** for store instructions.

Inputs and Outputs

- **Inputs:**
    - Address from the ALU (calculated in the EX stage).
    - Data to be written (from the previous pipeline stage, usually EX or a forwarding unit).
    - Control signals indicating read or write operations.
- **Outputs:**
    - Data read from memory (for load instructions).
    - Address and data are forwarded for write operations (for store instructions).

1.7.1. Memory

In modern pipelined processor architectures, efficient memory access is crucial. The `ByteAddressableMemory` module is designed to facilitate byte-level memory operations, which are essential for various data manipulation tasks. This module handles 16-bit read and write operations across byte-addressable memory, ensuring precise and flexible memory management.

Purpose

The `ByteAddressableMemory` module enables:

- **16-bit Data Writes:** Writing 16-bit data across two consecutive memory addresses.
- **16-bit Data Reads:** Reading 16-bit data from two consecutive memory addresses.
- **Byte-Level Control:** Managing memory at the byte level for finer granularity and flexibility.

Here is the Component for the Memory:

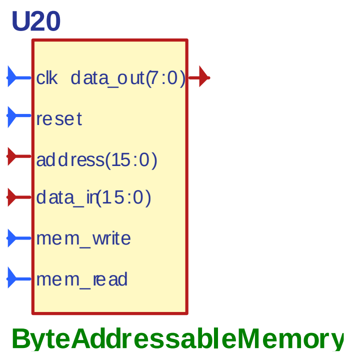


Figure 2-14 Memory

Inputs and Outputs

Inputs:

- `clk` (Clock): Synchronizes memory operations.
- `reset` (Reset): Initializes memory to a known state.
- `address` (16-bit): Specifies the memory address for read/write operations.
- `data_in` (16-bit): Data to be written to memory.
- `mem_write` (Memory Write): Control signal for writing data to memory.
- `mem_read` (Memory Read): Control signal for reading data from memory.

Output:

- `data_out` (16-bit): Data read from memory.

Functional Explanation

Reset Operation:

- When `reset` is high, all memory locations are initialized to `0`, ensuring a known starting state. This is achieved by iterating through the entire memory array and setting each byte to zero.

Memory Write Operation:

- If `mem_write` is high, the module performs a 16-bit write operation:
  - The lower byte of `data_in` is written to the specified `address`.
  - The upper byte of `data_in` is written to the next address (`address + 1`).
- This ensures that 16-bit data is correctly split and stored across two consecutive memory locations, maintaining data integrity.

Memory Read Operation:

- If `mem_read` is high, the module performs a 16-bit read operation:
  - It reads the byte at the specified `address` and the next byte at (`address + 1`).
  - These two bytes are combined to form a 16-bit value, which is then assigned to `data_out`.
- This allows 16-bit data to be reconstructed accurately from two consecutive memory locations.

So, the ByteAddressableMemory module is a fundamental component for managing memory operations in a pipelined processor. By supporting 16-bit read and write operations across byte-addressable memory, it ensures efficient and precise data handling, crucial for the overall performance and functionality of the processor.

1.7.2. MUX2x1 to choose which to write back on destination register Rd

In a pipelined processor, a multiplexer (MUX) is often used to select between different data sources for writing back to a destination register. The `MUX2x1` module described here is a 2-to-1 multiplexer that chooses between two 16-bit inputs based on a select signal.

Purpose

The `MUX2x1` module determines which of two possible data values should be written back to the destination register (`Rd`). This is critical in the write-back stage of the pipeline, where the final result of an instruction is stored.

So the MUX2x1 module is a simple yet essential component in the write-back stage of a pipelined processor. By selecting between two data inputs based on a control signal, it ensures that the correct data is written back to the destination register. This functionality is crucial for the proper operation and performance of the processor.

1.8. Signals controller unit

The signals control unit module generates control signals based on the 4-bit `opcode`, execution `mode`, and branch flag for branch instructions. It configures the ALU operation, memory access, register writes, branching, and jump operations. The unit uses the `opcode` to set specific control signals, which ensures the correct data path and operation flow through the pipeline stages, optimizing performance and handling instruction dependencies. All the signals are shown and explained in control signals.

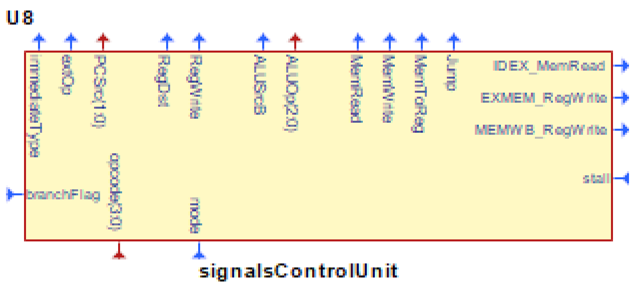


Figure 2-15 signals control unit

1.9. Hazards detector, stall cycle, and data forwarding Unit

In pipelined processors, instruction-level parallelism can lead to several challenges, including data hazards, control hazards, and structural hazards. To ensure the smooth execution of instructions, mechanisms like hazards detection, stall cycles, and data forwarding units are implemented. This section describes these mechanisms and their importance in maintaining the efficiency and correctness of a pipelined processor.

Here's a simplified Verilog module Component for a data hazards detector, stall, data Forwarding:

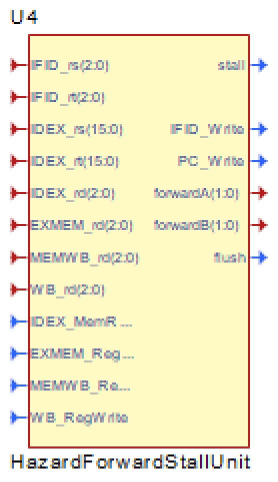


Figure 2-16 hazardComponent

1.9.1. Hazards Detector

A hazards detector identifies potential conflicts in the pipeline that can lead to incorrect data processing. The main types of hazards include:

1. **Data Hazards:** Occur when instructions depend on the results of previous instructions.
2. **Control Hazards:** Arise from the pipelining of branch instructions.
3. **Structural Hazards:** Occur when hardware resources are insufficient to handle the instructions.

1.9.2. Stall Cycle

A stall cycle (or pipeline stall) temporarily halts the pipeline to resolve hazards. When a hazards detector identifies a potential conflict, it signals a stall to prevent incorrect instruction execution.

Implementation

In the presence of a stall signal, the processor pauses the execution of certain stages while allowing others to proceed. This can be achieved by inserting no-operation (NOP) instructions or by holding the values in pipeline registers.

1.9.3. Data Forwarding Unit

The data forwarding unit resolves data hazards by forwarding the required data from later pipeline stages to earlier ones, bypassing the need to wait for data to be written back to the register file.

1.9.4. Functional Explanation

Hazards Detector

- **Data Hazards Detection:** Checks if the current instruction's source registers (`rs1`, `rs2`) match the destination registers (`rd`) of instructions in the EX or MEM stage that are reading from memory. If a match is found, a stall signal is issued to prevent data corruption.

Stall Cycle

- **Pipeline Control:** When a stall is detected, the pipeline control logic inserts a NOP instruction or holds the pipeline registers, effectively pausing the affected stages until the hazard is resolved.

Data Forwarding Unit

- **Data Forwarding:** The forwarding unit sends data directly from the EX or MEM stage to the ID stage if the current instruction depends on the result of a previous instruction that has not yet been written back to the register file. This bypasses the write-back stage, allowing the pipeline to continue without stalling.

The hazards detector, stall cycle, and data forwarding unit are essential components of a pipelined processor. They work together to detect and resolve hazards, ensure correct instruction execution, and maintain pipeline efficiency. By effectively managing data dependencies and control hazards, these units enhance the overall performance and reliability of the processor.

3. Control signals

We created these tables to help us understand the Datapath of each instruction, they are just an overview and will be manipulated throughout the states.

Table 1 control signals for each instruction: a

Instruction	extOp	PCsrc	RegDst	RegWrite	ALUSrcB	ALUOp	MemRead	MemWrite
AND	0	00	1	1	0	010	0	0
ADD	0	00	1	1	0	000	0	0
SUB	0	00	1	1	0	001	0	0
ADDI	0	00	0	1	1	010	0	0
ANDI	0	00	0	1	1	010	0	0
LW	0	00	0	1	1	000	1	0
LBu	0	00	0	1	1	000	1	0
LBs	1	00	0	1	1	000	1	0
SW	0	00	0	0	1	000	0	1
BGT	0	01	0	0	0	111	0	0
BGTZ	0	01	0	0	0	111	0	0
BLT	0	01	0	0	0	100	0	0
BLTZ	0	01	0	0	0	100	0	0
BEQ	0	01	0	0	0	001	0	0
BEQZ	0	01	0	0	0	001	0	0
BNE	0	01	0	0	0	001	0	0
BNEZ	0	01	0	0	0	001	0	0
JMP	0	10	0	0	0	101	0	0

CALL	0	10	0	1	0	101	0	0
RET	0	11	0	0	0	101	0	0
SV	0	0	0	0	1	000	0	1

Table 2 control signals for each instruction: b

Instruction	MemToReg	Jump	immediateType	IDEX_MemRead	EXMEM_RegWrite	MEMWB_RegWrite
AND	0	0	0	0	0	0
ADD	0	0	0	0	0	0
SUB	0	0	0	0	0	0
ADDI	0	0	0	0	0	0
ANDI	0	0	0	0	0	0
LW	1	0	0	1	0	1
LBu	1	0	0	1	0	1
LBs	1	0	0	1	0	1
SW	0	0	0	0	0	0
BGT	0	0	0	0	0	0
BGTZ	0	0	0	0	0	0
BLT	0	0	0	0	0	0
BLTZ	0	0	0	0	0	0
BEQ	0	0	0	0	0	0
BEQZ	0	0	0	0	0	0
BNE	0	0	0	0	0	0
BNEZ	0	0	0	0	0	0
JMP	0	1	0	0	0	0
CALL	0	1	0	0	1	0
RET	0	1	0	0	0	0
SV	0	0	1	0	0	0

And below is the table of each signal and its meaning and values to help understand the signals better in choosing them.

Table 3 signals description

Signal	Description	Values
extOp	Extension operation	Unsigned: 0

		Signed: 1
PCSrc	Source of next PC	PC +2: 00 Branch: 01 Jump: 10 Return: 11
RegDst	Destination register of instruction	Rs2: 0 Rd: 1
RegWrite	Enable writing to destination register in register file	Enable: 1 Disable: 0
ALUSrcB	Source of operand B before entering ALU	regB (output of ID): 0 Extended immediate: 1
ALUOp	ALU operation based on instruction	ADD: 000 SUB: 001 AND: 010 OR: 011 LT: 100 NOP: 101 GT: 111
MemRead	Enable reading from memory	Read: 1 No: 0
MemWrite	Writing to memory	Write: 1 No: 0
MemToReg	Write back data to destination register	Write back: 1 No: 0
Jump	Jump flag if instruction of J - Type	Jump: 1 No: 0
immediateType	Immediate type since length differs between I – type and S – type	S – type: 1 I - type: 0
IDEX_MemRead	For hazard detection use	Enabled: 1 Disabled: 0
EXMEM_RegWrite	For hazard detection use	Enabled: 1 Disabled: 0

MEMWB_RegWrite	For hazard detection use	Enabled: 1 Disabled: 0

4. RTL design

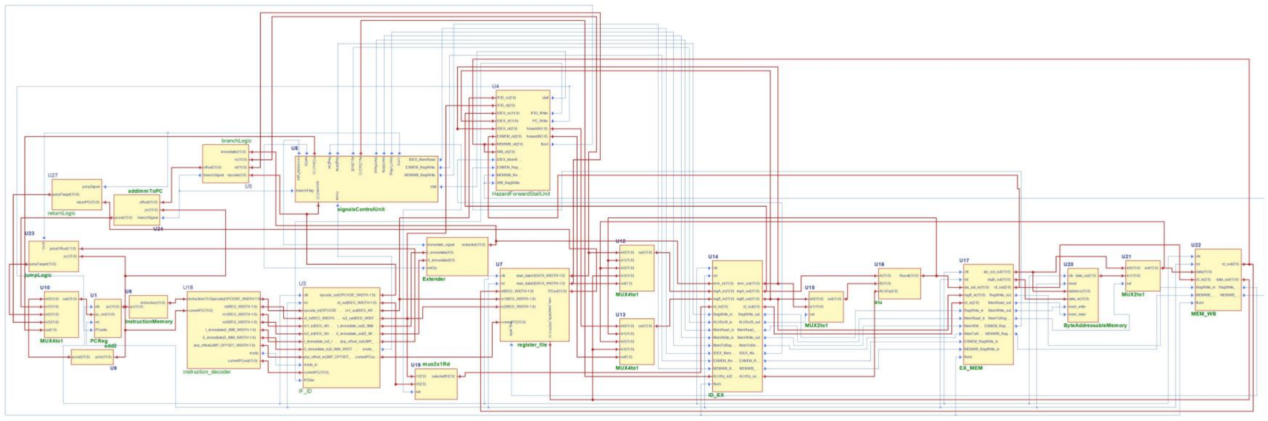


Figure 4-1 Full RTL design

RTL description

Table 4 RTL description

No.	Instr	Format	Meaning	Opcode Value	m
1	AND	R-Type	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	0000	
2	ADD	R-Type	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	0001	
3	SUB	R-Type	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	0010	
4	ADDI	I-Type	Reg(Rd) = Reg(Rs1) + Imm	0011	
5	ANDI	I-Type	Reg(Rd) = Reg(Rs1) + Imm	0100	
6	LW	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0101	
7	LBu	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0110	0
8	LBs	I-Type	Reg(Rd) = Mem(Reg(Rs1) + Imm)	0110	1
9	SW	I-Type	Mem(Reg(Rs1) + Imm) = Reg(Rd)	0111	

10	BGT	I-Type	if (Reg(Rd) > Reg(Rs1))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1000	0

11	BGTZ	I-Type	if (Reg(Rd) > Reg(0))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1000	1

12	BLT	I-Type	if (Reg(Rd) < Reg(Rs1))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1001	0

13	BLTZ	I-Type	if (Reg(Rd) < Reg(R0))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1001	1

14	BEQ	I-Type	if (Reg(Rd) == Reg(Rs1))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1010	0

15	BEQZ	I-Type	if (Reg(Rd) == Reg(R0))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1010	1

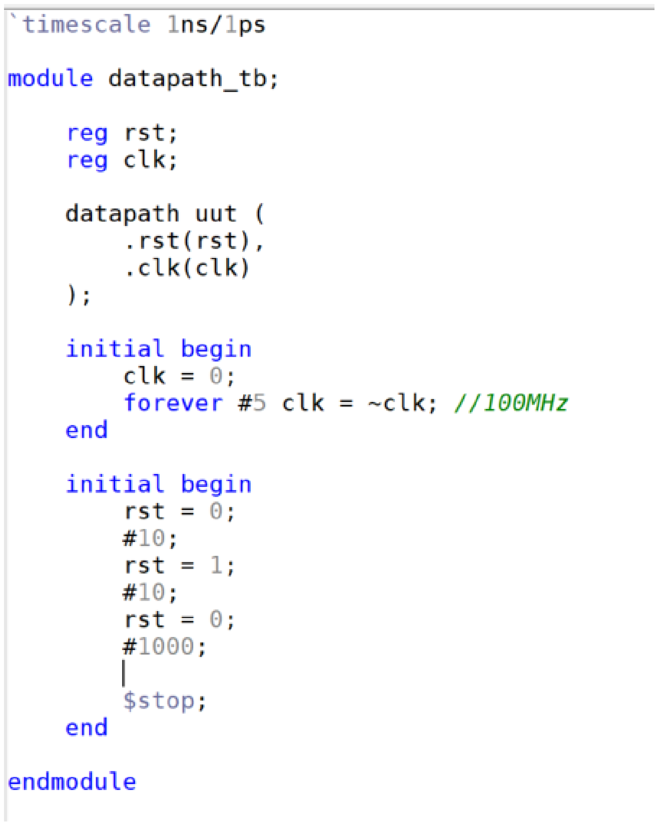
16	BNE	I-Type	if (Reg(Rd) != Reg(Rs1)) Next PC = PC + sign_extended (Imm)	1011	0
			else PC = PC + 2		

17	BNEZ	I-Type	if (Reg(Rd) != Reg(Rs1))
			Next PC = PC + sign_extended (Imm)
			else PC = PC + 2
				1011	1

18	JMP	J-Type	Next PC = {PC[15:10], Immediate}	1100	
19	CALL	J-Type	Next PC = {PC[15:10], Immediate} PC + 4 is saved on R7	1101	
20	RET	J-Type	Next PC = R7	1110	
21	Sv	S-Type	M[rs] = imm	1111	

5. Design verification

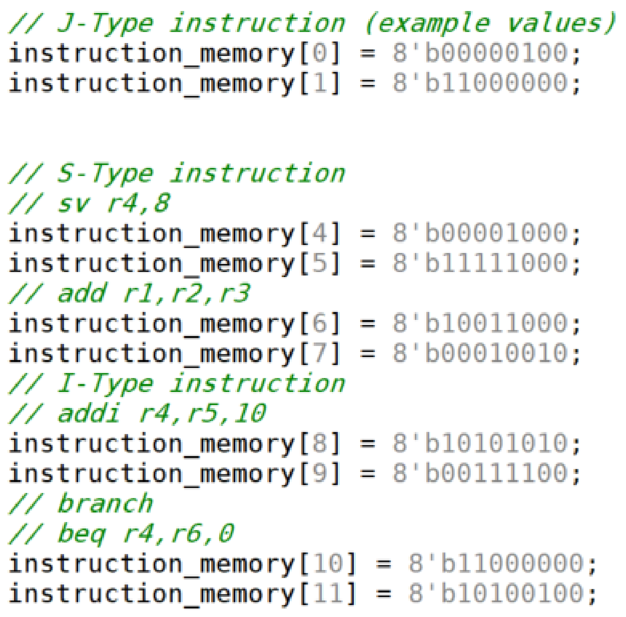
This testbench was created that generates a clock and turn off the machine after 1000 ns if the program took too long



```
`timescale 1ns/1ps

module datapath_tb;

    reg rst;
    reg clk;

    datapath uut (
        .rst(rst),
        .clk(clk)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk; //100MHz
    end

    initial begin
        rst = 0;
        #10;
        rst = 1;
        #10;
        rst = 0;
        #1000;

        $stop;
    end

endmodule
```

Figure 2 TestBench

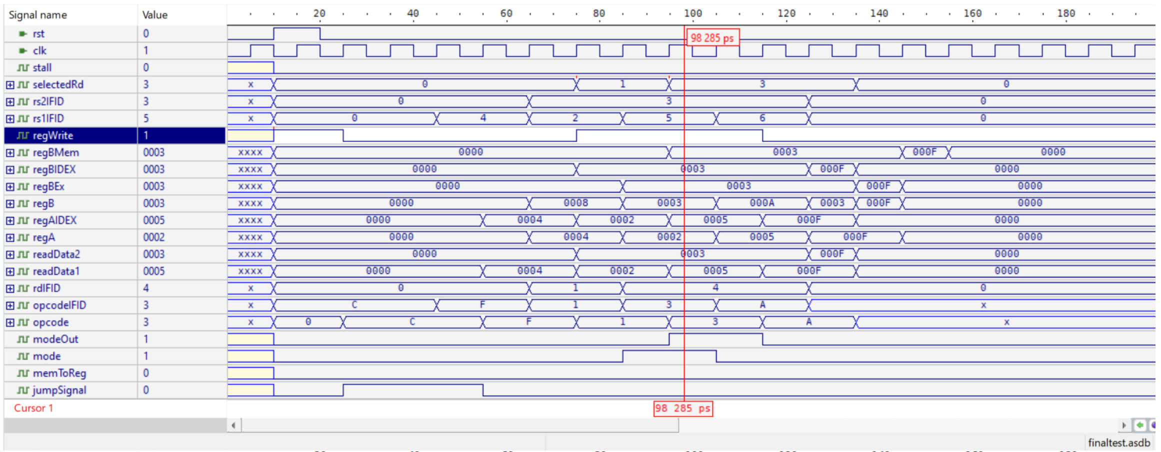
The instruction memory was initialized with this code to test and verify the results:



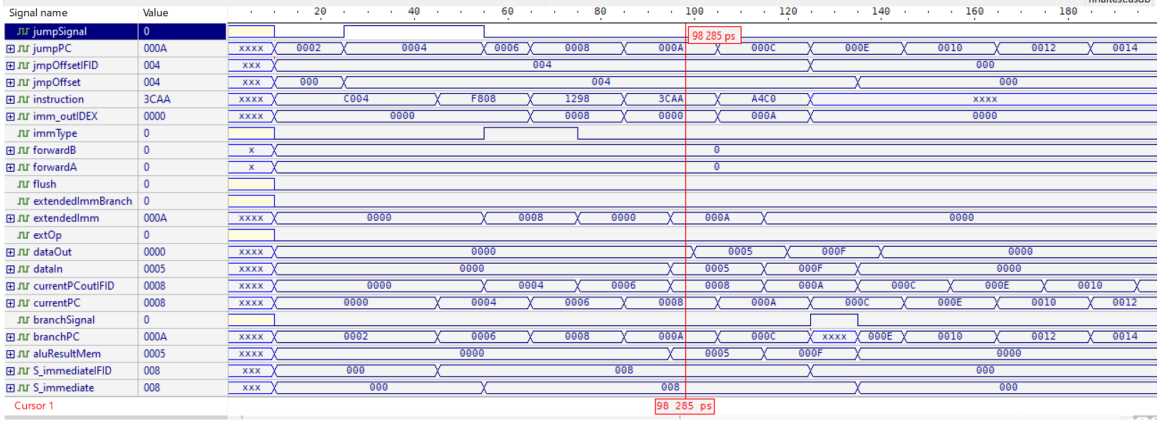
```
// J-Type instruction (example values)
instruction_memory[0] = 8'b00000100;
instruction_memory[1] = 8'b11000000;


// S-Type instruction
// sv r4,8
instruction_memory[4] = 8'b00001000;
instruction_memory[5] = 8'b11111000;
// add r1,r2,r3
instruction_memory[6] = 8'b10011000;
instruction_memory[7] = 8'b00010010;
// I-Type instruction
// addi r4,r5,10
instruction_memory[8] = 8'b10101010;
instruction_memory[9] = 8'b00111100;
// branch
// beq r4,r6,0
instruction_memory[10] = 8'b11000000;
instruction_memory[11] = 8'b10100100;
```

Figure 3 example of instructions

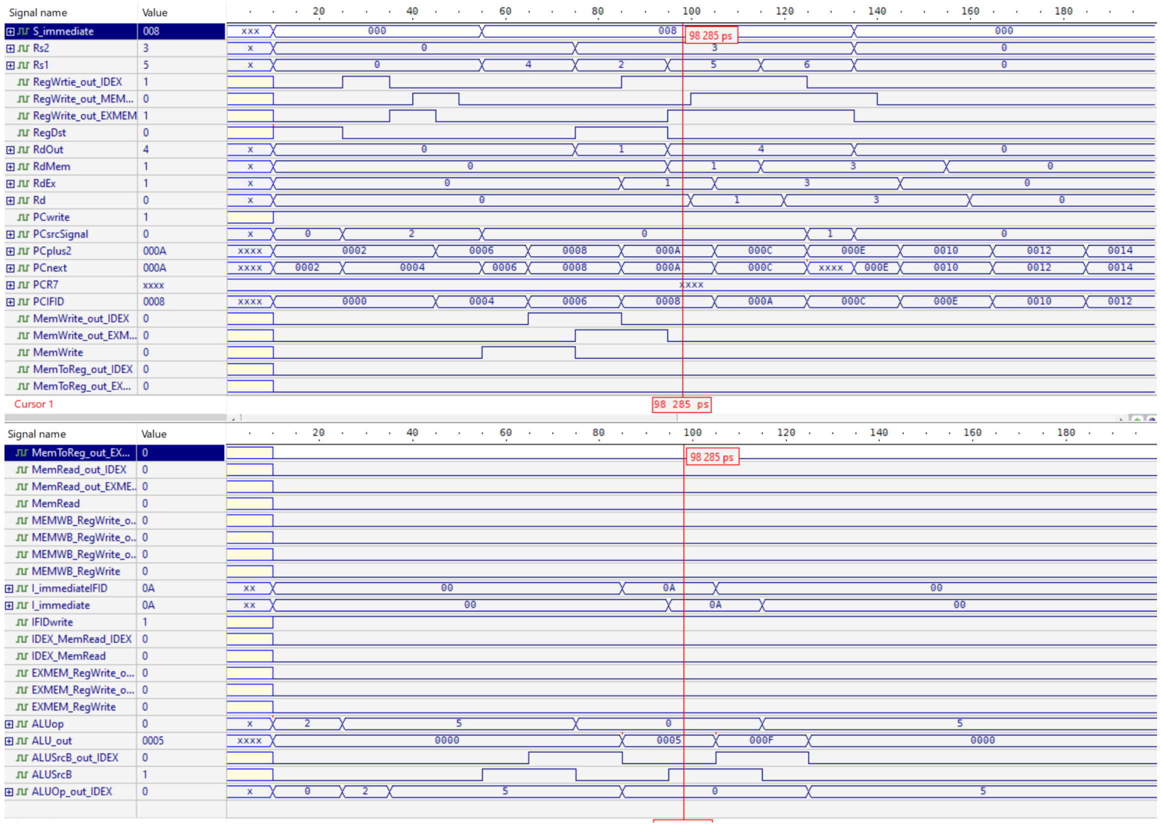
Testbench results:



Signal name	Value
rst	0
clk	1
stall	0
selectedRd	3
rs2IFID	3
rs1IFID	5
regWrite	1
regBMem	0003
regBIDEX	0003
regBEx	0003
regB	0003
regAIDEX	0005
regA	0002
readData2	0003
readData1	0005
rdIFID	4
opcodeIFID	3
opcode	3
modeOut	1
mode	1
memToReg	0
jumpSignal	0
Cursor 1	



Signal name	Value
jumpSignal	0
jumpPC	000A
jmpOffsetIFID	004
jmpOffset	004
instruction	3CAA
imm_outIDEX	0000
immType	0
forwardB	0
forwardA	0
flush	0
extendedImmBranch	0
extendedImm	000A
extOp	0
dataOut	0000
dataIn	0005
currentPCoutIFID	0008
currentPC	0008
branchSignal	0
branchPC	000A
aluResultMem	0005
S_immediateIFID	008
S_immediate	008
Cursor 1	



Signal name	Value
S_immediate	008
Rs2	3
Rs1	5
RegWrtie_out_IDEX	1
RegWrite_out_MEM...	0
RegWrite_out_EXMEM	1
RegDst	0
RdOut	4
RdMem	1
RdEx	1
Rd	0
PCwrite	1
PCsrcSignal	0
PCplus2	000A
PCnext	000A
PCR7	xxxx
PCIFID	0008
MemWrite_out_IDEX	0
MemWrite_out_EXM...	0
MemWrite	0
MemToReg_out_IDEX	0
MemToReg_out_EX...	0
Cursor 1	

Signal name	Value
MemToReg_out_EX...	0
MemRead_out_IDEX	0
MemRead_out_EXME..	0
MemRead	0
MEMWB_RegWrite_o..	0
MEMWB_RegWrite_o..	0
MEMWB_RegWrite_o..	0
MEMWB_RegWrite	0
I_immediateIFID	0A
I_immediate	0A
IFIDwrite	1
IDEX_MemRead_IDEX	0
IDEX_MemRead	0
EXMEM_RegWrite_o...	0
EXMEM_RegWrite_o...	0
EXMEM_RegWrite	0
ALUop	0
ALU_out	0005
ALUSrcB_out_IDEX	0
ALUSrcB	1
ALUOp_out_IDEX	0

Figure 4 test bench results

6. Team work

- Verilog implementations: both
- Hazards detector, stall, and data forwarding unit: Lama
- Signals controller: Amal
- RTL design: Amal
- Test benches: both
- Report: both

7. Conclusion

In summary, this report presented the successful design and implementation of a pipelined MIPS RISC processor for the ENCS4370 Computer Architecture project. Through rigorous testing and verification, each component of the processor, including the hazards detector, stall cycle, data forwarding, ALU, control unit, register file, and data memory, demonstrated accurate functionality and minimized errors. The RTL description, control state diagram, and testbench results confirmed the efficient operation and support for various instruction types and functions. This project's outcomes contribute to the advancement of computer engineering knowledge, providing a strong foundation for future developments in processor design and optimization.