Faculty of Engineering and Technology
Electrical and Computer Engineering Department
ENCS4370 - Computer Architecture

Project No. 2: Multi-Cycle RISC processor

Prepared by

Nour Naji 1190270
Aseel Sabri 1190144
Zakiya AbuMurra 1191636

Supervised by
Dr. Aziz Qaroush

BIRZEIT
july-2023

Abstract

This report focuses on the design and implementation of a Multi-Cycle Processor based on a specific RISC instruction set. The design process involved a thorough analysis of each instruction to identify the main components required. These components were then built and integrated into the processor design. The implementation phase encompassed the derivation of control signals for the individual components. This involved determining the appropriate timing and sequencing of signals to ensure proper execution of instructions.

To validate the functionality of the processor, extensive testing was conducted. Each instruction was tested individually to verify its correct operation. Additionally, complete scenarios were applied to the processor to evaluate its performance and ensure accurate results.

Through the design, implementation, and testing phases, the Multi-Cycle Processor was evaluated for its ability to execute the given RISC instruction set accurately and efficiently. The results of the testing process confirmed the proper functioning of the processor, validating its design and implementation choices.

Table of Contents

List of Figures

List of Tables

1 Introduction

A Multi-Cycle RISC (Reduced Instruction Set Computer) processor is a type of central processing unit (CPU) design that follows the principles of the RISC architecture. RISC processors aim to simplify the execution of instructions by employing a reduced set of simple and uniform instructions, allowing for faster and more efficient processing. [1]

The term "multi-cycle" refers to the fact that the execution of instructions in a multi-cycle RISC processor occurs over multiple clock cycles. Each clock cycle is dedicated to a specific stage of the instruction execution process, such as instruction fetch, instruction decode, execution, memory access, and write-back. This division of the instruction execution process into multiple cycles allows for more flexibility and modularity in the design of the processor. [1]

The multi-cycle RISC processor architecture offers several advantages over other processor designs. Firstly, it simplifies the control logic of the processor by dividing the instruction execution process into smaller stages, which are easier to manage. This simplification helps in reducing the complexity of the processor's design and facilitates higher clock speeds.

Multi-cycle approach enables the use of more complex instructions without sacrificing performance. While RISC processors traditionally rely on a small set of simple instructions, multi-cycle designs allow for the execution of more complex instructions by dividing their execution into multiple cycles. This flexibility improves the overall functionality and versatility of the processor. [2]

In this project, our objective is to develop and test a simple multi-cycle RISC processor using the VHDL language. The processor is designed to support a limited set of instructions encompassing memory operations, logical operations, arithmetic computations, and Stack. The utilization of the multi-cycle technique in the proposed processor architecture enables a straightforward implementation with a condensed instruction set. By designing this basic multi-cycle RISC processor, we aim to gain a comprehensive understanding of computer architecture principles and processor design.

2 Design and Implementation

2.1 Overall design

The multi-cycle design approach is employed to create a processor capable of executing various instructions. In this design, the data path is divided into stages, each corresponding to a specific clock cycle. Each clock cycle executes a different part of the instruction, and the results are stored in the appropriate register or memory location.

To minimize redundancy, data required for subsequent clock cycles is stored in state elements at the end of each clock cycle. Programmer-visible state elements like the register file, program counter (PC), memory, and Stack hold data needed for instructions in future clock cycles. However, data required by the same instruction in later cycles is stored in additional registers.

The methodology used in developing the data path for this multi-cycle RISC processor focuses on breaking down instruction execution into small, sequential steps rather than attempting to perform all actions within a single cycle. These steps are organized into groups that are executed in a specific 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.
- Writing the next PC to the Stack based on a stop bit.

The overall data path design incorporates various components, such as registers, an ALU, multiplexers, memory, a control unit, and a Stack. These components are selected and interconnected to optimize hardware utilization and enable functional unit sharing within the execution of a single instruction. Additional registers are strategically placed based on the fitting of combination units within one clock cycle and the data required for proper instruction execution in subsequent cycles.

In this multi-cycle design, each clock cycle is designed to accommodate at most one operation, such as instruction memory access, data memory access, register file access (two reads or one write), ALU operation, or Stack operation. To avoid timing conflicts and ensure accuracy, data generated by these functional units is saved in temporary registers for use in subsequent cycles. This minimizes the need for redundant data copying and mitigates the risk of using incorrect values.

2.2 Instruction Set

Table below shows the instructions supported by this instruction set, with their meaning and Function Value.

No.	Instr	Meaning	Function Value
R-Type Instructions			
1	AND	Reg(Rd) = Reg(Rs1) & Reg(Rs2)	00000
2	ADD	Reg(Rd) = Reg(Rs1) + Reg(Rs2)	00001
3	SUB	Reg(Rd) = Reg(Rs1) - Reg(Rs2)	00010
4	CMP	zero-signal = Reg(Rs) < Reg(Rs2)	00011
I-Type Instructions			
5	ANDI	Reg(Rd) = Reg(Rs1) & Immediate^{14}	00000
6	ADDI	Reg(Rd) = Reg(Rs1) + Immediate^{14}	00001
7	LW	Reg(Rd) = Mem(Reg(Rs1) + Imm^{14})	00010
8	SW	Mem(Reg(Rs1) + Imm^{14}) = Reg(Rd)	00011
9	BEQ	Branch if (Reg(Rs1) == Reg(Rd))	00100
J-Type Instructions			
10	J	PC = PC + Immediate^{24}	00000
11	JAL	PC = PC + Immediate^{24} Stack.Push (PC + 4)	00001
S-Type Instructions			
12	SLL	Reg(Rd) = Reg(Rs1) << SA^{5}	00000
13	SLR	Reg(Rd) = Reg(Rs1) >> SA^{5}	00001
14	SLLV	Reg(Rd) = Reg(Rs1) << Reg(Rs2)	00010
15	SLRV	Reg(Rd) = Reg(Rs1) >> Reg(Rs2)	00011

Figure 2.1:Instruction Set

The format of each type is as follows:

R-type:

Function^{5}	Rs1^{5}	Rd^{5}	Rs2^{5}	Unused^{9}	Type^{2}	Stop^{1}

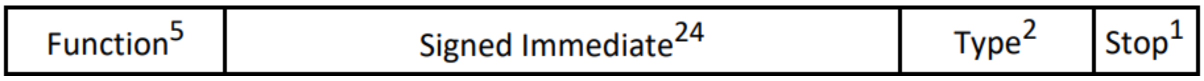
Figure 2.2: R-Type Instruction Format

I-type:

Function^{5}	Rs1^{5}	Rd^{5}	Immediate^{14}	Type^{2}	Stop^{1}

Figure 2.3: I-Type Instruction Format

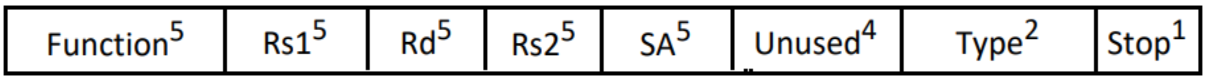
J-type:



Function^{5}	Signed Immediate^{24}	Type^{2}	Stop^{1}

Figure 2.4: J-Type Instruction Format

S-type:



Function^{5}	Rs1^{5}	Rd^{5}	Rs2^{5}	SA^{5}	Unused^{4}	Type^{2}	Stop^{1}

Figure 2.5: S-Type Instruction Format

2.3 Components

After analyzing the instruction set, we found that the following components are needed.

2.3.1 Instruction Memory

the Instruction Memory in a multi-cycle processor function as a read-only memory unit, providing instructions to the processor based on the address input. Its simplicity and lack of write access make it suitable for implementation as combinational logic, without the need for read control signals.

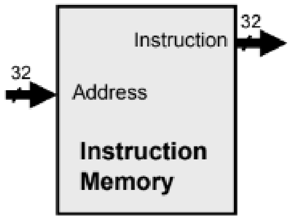


Figure 2.6:Instruction Memory

In our project, The Instruction Memory has a 32-bit address signal (address) as an input, in addition to memory write signal (memW), and a 32-bit signal (Din) used for testing purposes and for inserting data into the memory. The module also features an output wire (Dout) that carries the instruction read from the specified memory location. By utilizing these inputs and outputs, the InstMem module enables the system to store and retrieve instructions, playing a crucial role in the execution of program instructions and the overall functionality of the computer system.

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

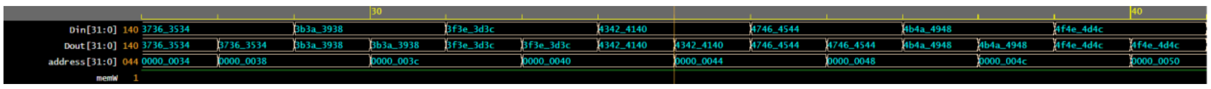

30
40
Din[31:0] 140 3736_3534 3b3a_3938 3f3e_3d3c 4342_4140 4746_4544 4b4a_4948 4f4e_4d4c
Dout[31:0] 140 3736_3534 3736_3534 3b3a_3938 3b3a_3938 3f3e_3d3c 3f3e_3d3c 4342_4140 4342_4140 4746_4544 4746_4544 4b4a_4948 4b4a_4948 4f4e_4d4c 4f4e_4d4c
address[31:0] 044 0000_0034 0000_0038 0000_003c 0000_0040 0000_0044 0000_0048 0000_004c 0000_0050
memW 1

Figure 2.7:Instruction Memory waveform-1

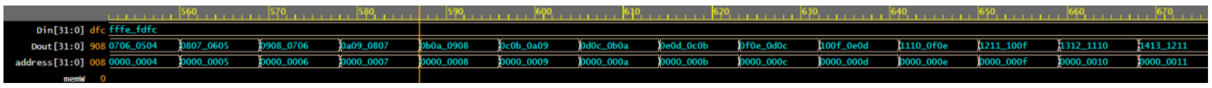

560 570 580 590 600 610 620 630 640 650 660 670
Din[31:0] dfc fffe_fdfc
Dout[31:0] 908 0706_0504 0807_0605 0908_0706 0a09_0807 0b0a_0908 0c0b_0a09 0d0c_0b0a 0e0d_0c0b 0f0e_0d0c 100f_0e0d 1110_0f0e 1211_100f 1312_1110 1413_1211
address[31:0] 008 0000_0004 0000_0005 0000_0006 0000_0007 0000_0008 0000_0009 0000_000a 0000_000b 0000_000c 0000_000d 0000_000e 0000_000f 0000_0010 0000_0011
memW 0

Figure 2.8:Instruction Memory waveform-2

2.3.2 Data Memory

The Memory Data Register (MDR) in a processor act as a temporary storage register for data during memory read and write operations. When reading from memory, the MDR holds the data retrieved from the specified memory location, allowing it to be processed or stored in other registers. During memory write operations, the MDR holds the data to be written into memory, facilitating the transfer of data from the processor to the specified memory location.

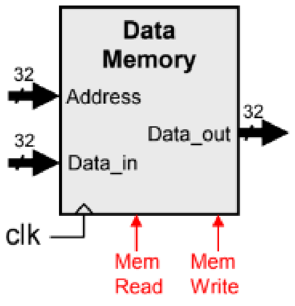


Figure 2.9:Data Memory

The Data Memory module in our project handles data memory operations, including reading and writing data. It takes inputs including the memory address, data input, read and write control signals, and clock. The module outputs the data read from the memory. It performs memory read and write operations based on the control signals and updates the memory contents accordingly. The clock signal ensures proper synchronization of the module.

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

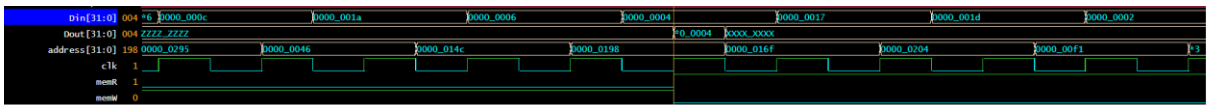

Din[31:0] 004 *6 0000_000c 0000_001a 0000_0006 0000_0004 0000_0017 0000_001d 0000_0002
Dout[31:0] 004 zzzz_zzzz *0_0004 xxxx_xxxx
address[31:0] 198 0000_0295 0000_0046 0000_014c 0000_0198 0000_016f 0000_0204 0000_00f1 *3
clk 1
memR 1
memW 0

Figure 2.10: Data Memory waveform-1

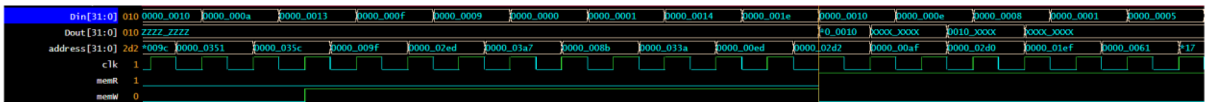

Din[31:0] 010 0000_0010 0000_000a 0000_0013 0000_000f 0000_0009 0000_0000 0000_0001 0000_0014 0000_001e 0000_0010 0000_000e 0000_0008 0000_0001 0000_0005
Dout[31:0] 010 zzzz_zzzz *0_0010 xxxx_xxxx 0010_xxxx xxxx_xxxx
address[31:0] 2d2 *009c 0000_0351 0000_035c 0000_009f 0000_02ed 0000_03a7 0000_008b 0000_033a 0000_00ed 0000_02d2 0000_00af 0000_02d0 0000_01ef 0000_0061 *17
clk 1
memR 1
memW 0

Figure 2.11: Data Memory waveform-2

2.3.3 Register File

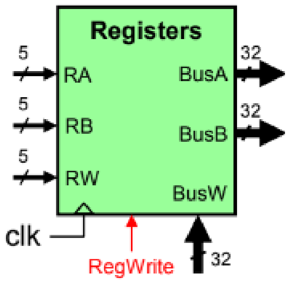


Figure 2.12:register file

A register file is a memory storage unit within the CPU that stores and retrieves data. It consists of multiple registers with unique addresses used for mapping. These registers hold information and can be accessed and manipulated using addresses, along with inputs such as a data bus, read/write enables, and a clock signal for synchronization. The register file plays a vital role in efficient data storage and retrieval within the CPU.

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), register write signal (reg_write), and 5-bit address signals (RA, RB, RW). Additionally, there is a 32-bit input signal (Bus_W) for data to be written into the registers. The module provides two 32-bit output wires (Bus_A, Bus_B) 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's registers, contributing to the overall functionality and performance of the computer system.

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

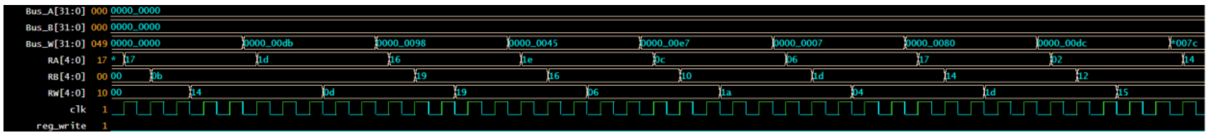


Figure 2.13:register file waveform-1

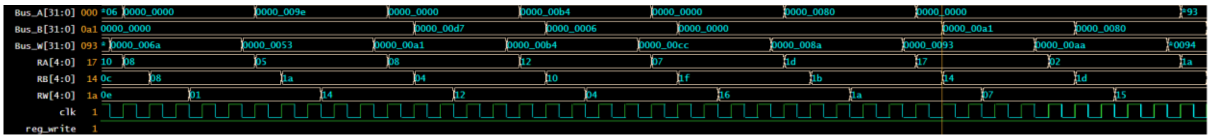


Figure 2.14:register file waveform-2

2.3.4 Extender

The extender is a crucial component in a computer's CPU that modifies the size of a data value. In this implementation, the extender is divided into three separate components: one for extending the 5-bit shift amount, and the other two are for extending the 14-bit and 24-bit immediate. By passing a parameter to the extender, it can effectively extend the specified input, depending on the specific operation's requirements. The extender's purpose is to increase the size of the immediate value so that it can be utilized in operations that demand larger data sizes. By providing the necessary size extension, the extender ensures that the CPU has access to the appropriate data size for performing various operations, thus playing a vital role in the overall functionality of the CPU.

The extender module in our project extends a 32-bit input signal (in) based on the signop and EXsrc inputs. It produces a 32-bit output signal (out) with extended bits. The EXsrc input determines the number of bits to extend, while the signop input controls whether signed or unsigned extension is performed. The module copies the corresponding bits from the input signal to the output and sets the remaining bits based on the extension type. If signop is true, signed extension is performed by setting the remaining bits to the sign bit. If signop is false, unsigned extension is done by setting the remaining bits to zero. By applying this logic, the sign_extender module effectively extends the input signal to the desired length.

The Extender testing-waveform is shown in the figure bellow.

When EXsrc = 0 , 1

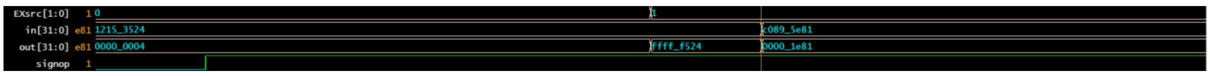

EXsrc[1:0] 1 0 1
in[31:0] e81 1215_3524 c089_5e81
out[31:0] e81 0000_0004 ffff_f524 0000_1e81
signop 1

Figure 2.15:Extender waveform-1

EXsrc = 2

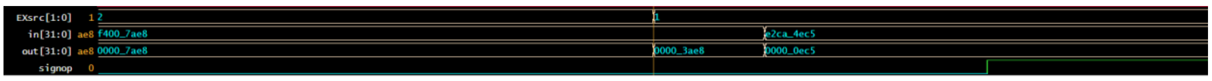

EXsrc[1:0] 1 2 1
in[31:0] ae8 f400_7ae8 e2ca_4ec5
out[31:0] ae8 0000_7ae8 0000_3ae8 0000_0ec5
signop 0

Figure 2.16:Extender waveform-2

2.3.5 Arithmetic Logic Unit

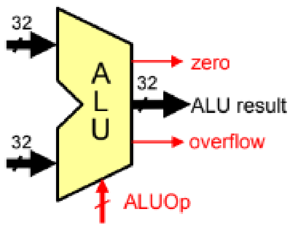


Figure 2.17:Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is a critical component of a computer's CPU responsible for performing arithmetic and logical operations. It handles operations like addition, subtraction, multiplication, division, as well as logical operations such as AND, OR, NOT, and XOR. The ALU operates on binary data and produces a result that is stored in a register or used as memory location. It is optimized for high-speed operation through specialized circuits and logic gates. The ALU communicates with other CPU components through input and output buses, allowing data to be fetched, processed, and written back. In this implementation, the ALU's components work continuously, but the specific output sent to the result bus depends on the instruction opcode being executed. The ALU also features zero and negative flags, which are set or cleared based on the zeroness and sign of the result, respectively. Additionally, the ALU includes carry and overflow flags, which indicate if there was a carry-out or borrow during arithmetic operations, and if the result of a signed arithmetic operation exceeds the representable range, respectively. These flags provide important status information for subsequent instructions or conditional branching, ensuring proper handling of carry and overflow conditions and enhancing the accuracy and reliability of arithmetic and logical computations.

The ALU module in our project performs arithmetic and logical operations on 32-bit inputs A and B using a 3-bit opcode. It produces a 32-bit result along with carry, zero, negative, and overflow flags. These flags provide information about carry-out, zero result, negative result, and overflow conditions. The ALU efficiently computes results and status flags based on the input values and opcode, enabling various arithmetic and logical operations in a processor.

The testing results of our ALU unit are shown in the waveform bellow.

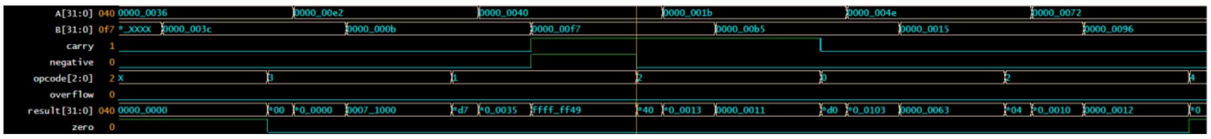


Figure 2.18:ALU waveform-1

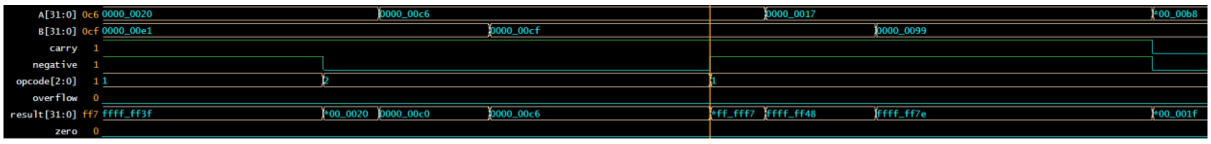


Figure 2.19:ALU waveform-2

2.3.6 Stack

The stack, implemented using a separate on-chip memory, is a crucial data structure in computer systems. It follows the Last-In-First-Out (LIFO) principle and is utilized for managing temporary data, local variables, and return addresses during program execution. The stack pointer (SP) is a special-purpose register that holds the address of the empty element on the top of the stack. It facilitates subroutine calls by storing return addresses, allowing nested function calls, and ensuring proper program flow. The stack also enables efficient saving and restoration of register values during subroutine execution. Additionally, it supports memory management by dynamically allocating and deallocating memory for local variables and function calls. Overall, the stack plays a vital role in managing data and control flow, enhancing program execution and facilitating organized memory utilization.

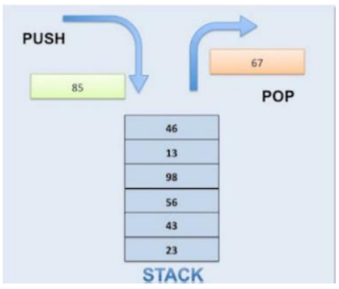
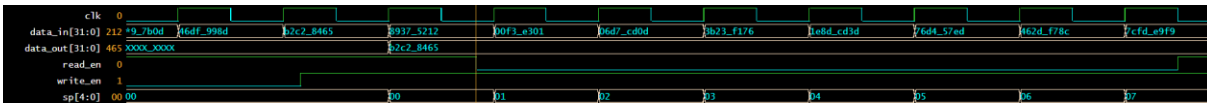
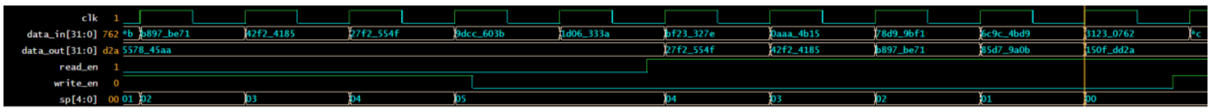


Figure 2.20:stack

The stack module in our project implements a stack data structure using a clock signal (clk), read enable signal (read_en), write enable signal (write_en), data input (data_in), and data output (data_out). It allows for storing and retrieving data in a last-in-first-out (LIFO) fashion. When the read enable signal is asserted, the module outputs and pops the top element from the stack. When the write enable signal is asserted, the module stores and pushes the data_in value at the top of the stack.

The testing results of our stack unit are shown in the waveform bellow.


clk 0
data_in[31:0] 212 *9_7b0d 46df_998d b2c2_8465 8937_5212 00f3_e301 06d7_cd0d 3b23_f176 1e8d_cd3d 76d4_57ed 462d_f78c 7cfd_e9f9
data_out[31:0] 465 XXXX_XXXX b2c2_8465
read_en 0
write_en 1
sp[4:0] 00 00 00 01 02 03 04 05 06 07
Figure 2.21:stack waveform-1


clk 1
data_in[31:0] 762 *b b897_be71 42f2_4185 27f2_554f 9dcc_603b 1d06_333a bf23_327e 0aaa_4b15 78d9_9bf1 6c9c_4bd9 3123_0762 *c
data_out[31:0] d2a 5578_45aa 27f2_554f 42f2_4185 b897_be71 85d7_9a0b 150f_dd2a
read_en 1
write_en 0
sp[4:0] 00 01 02 03 04 05 04 03 02 01 00
Figure 2.22:stack waveform-2

2.4 State diagram

In order to implement the desired design, the flow of the processer states throughout its execution should be determined. The state diagram of our processer is shown in the figure bellow. Each of the states is to be explained in the next section.

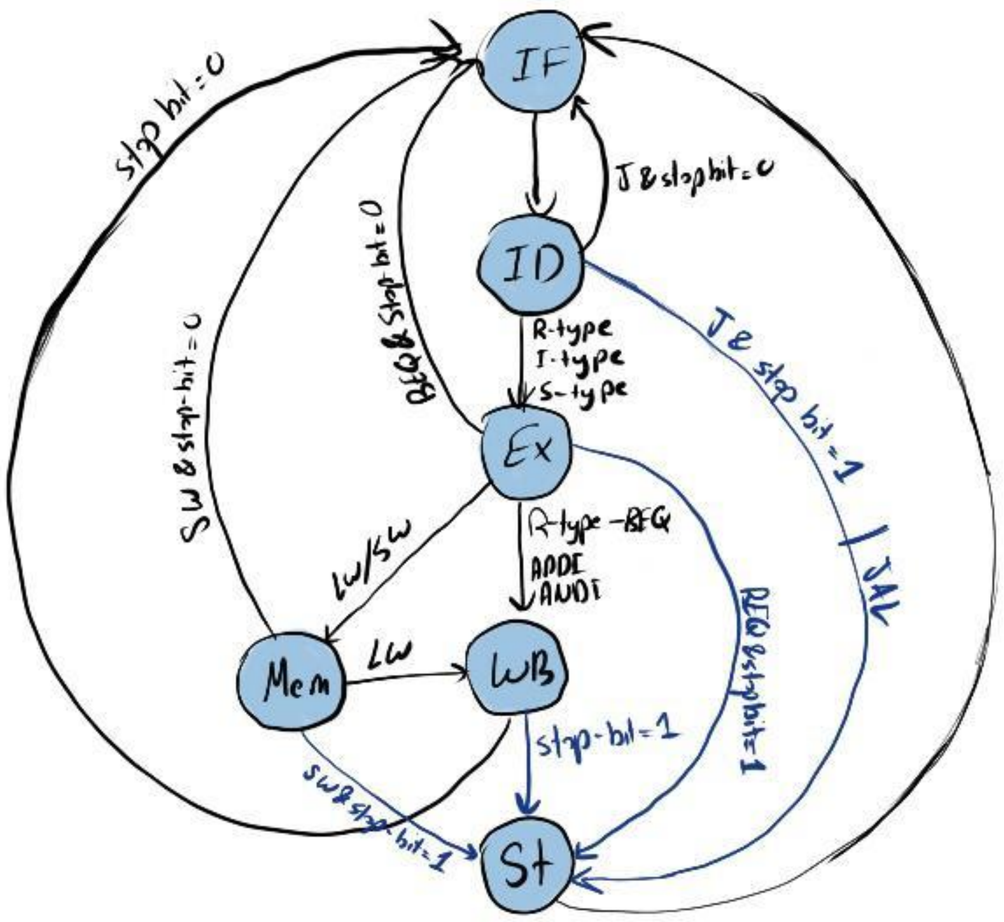


Figure 2.23:State diagram

2.5 Datapath

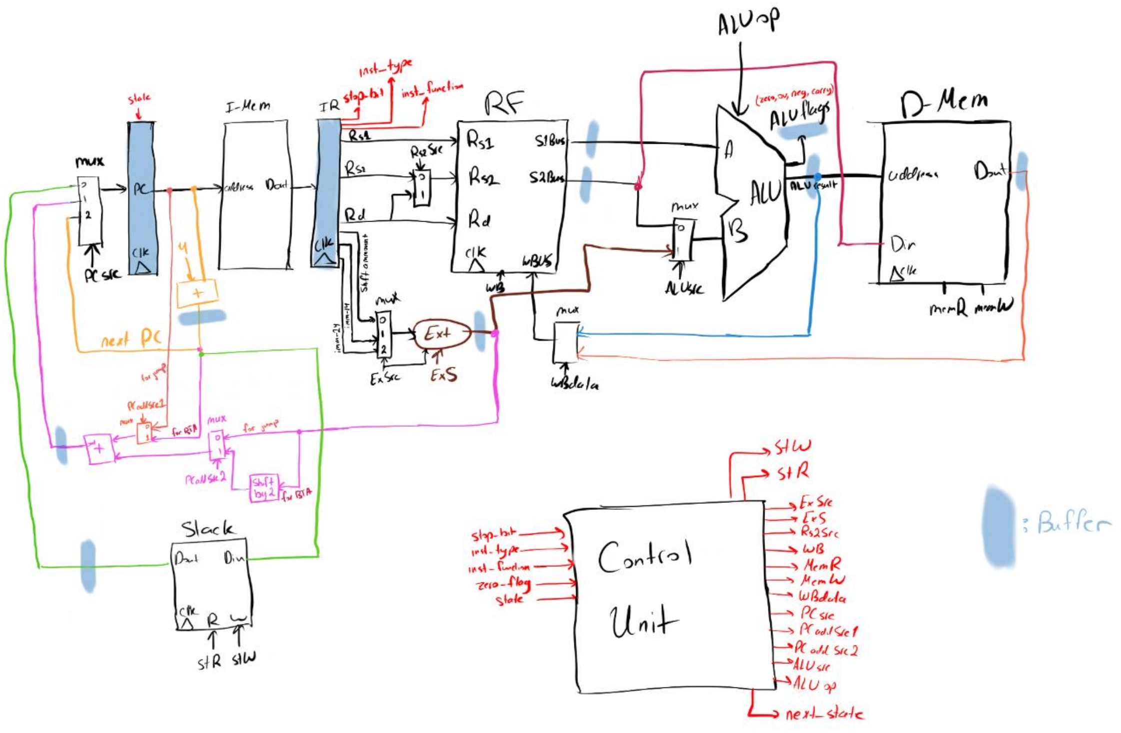


Figure 2.24:Datapath

Figure above shows the Datapath, which can be divided into 6 stages.

2.5.1 CPU-Stage:

2.5.1.1 Instruction Fetch

The instruction fetching process involves accessing the instruction memory based on the address provided by the program counter (PC). The PC address is updated at the positive clock edge when the state is IF (Instruction Fetch). It can be updated to three different values, depending on the instruction being executed:

If the PC address is 00, it means that the stop bit is 1 and requires the program counter to be updated with the value from the stack PC (Program Counter). This allows the program to jump to a different location in the code, usually to return from a subroutine or an interrupt.

If the PC address is 01, it signifies that the instruction is a branch or jump instruction. In this case, the program counter is updated with the branch target address or the jump distenation, respectively. Branch instructions are used to conditionally alter the program flow, allowing for decisions and loops in the code execution. On the other hand, Jump and Jump-And-Link instructions are usually used to handle subroutines and expceptions.

If the PC address is 10, it implies a normal sequential execution of instructions. The program counter is updated by adding 4 to its current value, incrementing it to the next instruction address. This ensures the program proceeds to the subsequent instruction in memory.

By updating the program counter to these different values, the instruction fetching mechanism enables the processor to correctly fetch and execute instructions based on the control flow requirements of the program. It allows for branching, looping, and sequential execution, ensuring the program runs in a controlled and organized manner.

2.5.1.2 Instruction Decode

The instruction decode stage is responsible for decoding the fetched instruction and extracting its different components. The instruction is typically decoded into its type, function code, source register (Rs), target register (Rt), destination register (Rd), shift amount (SA), immediate values (Immediate14, Immediate24), stop bit, and other relevant fields.

Once the instruction is decoded, the Control Unit starts generating control signals depending on the available instruction type and function. These control signals determine the actions to be taken in subsequent stages of the processor.

One important task during the instruction decode stage is reading the registers from the Register File. The source registers (Rs and Rt) are identified based on the decoded instruction, and their corresponding values are fetched from the register file. This retrieval of register values ensures that the necessary operands are available for subsequent stages.

After obtaining the register values and immediate data, the instruction decode stage is completed. The values of the registers and immediate data are then passed on to the execution stage for further processing. These values serve as inputs for the execution of arithmetic, logical, or control operations specified by the instruction.

2.5.1.3 Execute

The execution stage is where the actual operations specified by the instruction are performed using the Arithmetic Logic Unit (ALU) unit.

During the execution stage, the ALU takes the input values from the previous stages, such as register values and immediate data, and performs the necessary calculations or operations. This may include arithmetic operations like addition, subtraction, multiplication, or logical operations such as AND, OR, or comparison operations.

Once the ALU completes the operation, the resulting value or status is passed on to the next stage. The output from the execution stage may include the result of the operation, a flag indicating the status of the operation (e.g., overflow or carry), or any other relevant information. The produced flags are fed to the control unit to determine, for example, if a branch will be taken.

2.5.1.4 Memory

In the memory stage of a processor, the control signals derived from previous stages determine whether a read operation, a write operation, or no operation (also known as a "no-op") will occur. These control signals govern the interaction between the processor and the memory system. Here's a breakdown of each possibility:

- Read Operation: If the control signals indicate a read operation, it means that the processor needs to fetch data from memory. The memory address is typically provided by the instruction being executed. The control signals activate the necessary circuits and signals within the memory system to read the data stored at the specified memory address. The retrieved data is then passed along to the subsequent stages of the processor for further processing.
- Write Operation: When the control signals indicate a write operation, it means that the processor needs to store data in memory. Similar to a read operation, the memory address is typically provided by the instruction being executed. The control signals enable the necessary circuits and signals within the memory system to write the data to the specified memory address.
- No Operation (No-Op): In certain cases, the control signals derived from previous stages may indicate that no memory operation is necessary. This could happen when an instruction does not involve accessing or modifying data in memory, such as arithmetic or logic instructions that operate solely on register values. In such cases, the control

signals effectively bypass the memory stage, allowing the processor to proceed to the next stage without any memory-related operations.

The determination of whether a read, write, or no-op occurs is typically based on the instruction being executed and the specific design and implementation of the processor's control unit. The control signals derived from earlier stages help coordinate the interactions between the processor and the memory system, ensuring the correct flow of data and instructions during the execution of a program.

2.5.1.5 Write Back

The write-back stage, also known as the WB stage, is a crucial step in the instruction execution process within a processor. Its primary responsibility is to write data back to the appropriate destination, typically the register file, after performing necessary computations or retrieving data from memory.

The specific data to be written back depends on the type of instruction being executed:

ALU Result: In many arithmetic and logical instructions, the ALU (Arithmetic Logic Unit) performs computations, such as addition, subtraction, bitwise operations, etc. The result of these computations is typically written back to the register file.

Memory Data: In load instructions (e.g., LW - Load Word), data is fetched from memory and then written back to the register file. The memory data could be the value stored at a particular memory address or a portion of it.

2.5.1.6 Stack

The stack in the processor is a data structure used to manage execution flow and store return addresses. It is a Last-In-First-Out (LIFO) structure. The stack pointer (SP) is a special-purpose register that keeps track of the top of the stack. When a JAL instruction is executed, the SP is incremented, and the return address (PC + 4) is stored in memory at the SP location. The PC is then updated to the target address. When it's time to return (stop-bit =1), the return address is loaded from the stack, the SP is decremented, and execution continues from the loaded address.

2.6 Temporary registers

In multi-cycle processor, several temporary registers are introduced to facilitate the execution of instructions and the flow of data between different stages:

- Instruction Register (IR): The IR holds the instruction being executed and stores relevant information such as the source registers (RS1, RS2), destination register (RD), shift amount (SA), and immediate value. It enables easy access to instruction data throughout various stages.
- Data Registers for Read RS1 and RS2: These registers hold the values of the source registers (RS1 and RS2) of the instruction. They store the necessary data for stages like execution and memory access, allowing the processor to operate on the required operands.
- ALU Result and Flags Registers: These registers hold the result of arithmetic or logical operations performed by the Arithmetic Logic Unit (ALU). Additionally, they store flags such as the zero flag, carry flag, overflow flag and negative flag, indicating the status of the operation. These flags are often used in subsequent stages for branching or condition checking.
- Memory Data Register: This register holds the data fetched from memory. It stores the value returned from memory access and is used in the write-back stage or for further processing in subsequent instructions.
- Extender Value Register: This register holds the value obtained from the extender unit. It retains the extended value, such as sign-extension or zero-extension, which may be required in various stages of instruction execution.
- Stack Output Buffer: This buffer holds the output of the stack operation, allowing it to be used in the Fetch stage. It assists in the smooth flow of control and data by providing the necessary information for fetching the next instruction.

By utilizing these temporary registers and buffers, the multi-cycle processor ensures efficient data handling, proper sequencing of instructions, and seamless transfer of information between different stages of execution.

2.7 Control Unit

After building the Datapath, the control signals were derived. The tables below show how each control signal changes depending on the instruction type, function, and flags.

Note that the WB and MemW signals, are set to 1 only at the write back stage and memory stage, respectively.

Table 1: Main Control Signals

Type	Function	ExSrc	ExS	RS2Src	WB	MemR	MemW	WBdata
R-type	And, add, sub	X	X	0	1	0	0	0
R- type	CMP	X	X	0	0	0	0	X
I-type	ADDI	1	1	X	1	0	0	0
I-type	ANDI	1	0	X	1	0	0	0
I-type	LW	1	1	X	1	1	0	1
I-type	SW	1	1	1	0	0	1	X
I-type	BEQ	1	1	1	0	0	0	X
J-type	×	2	1	X	0	0	0	X
S-type	SLL, SLR	0	0	X	1	0	0	0
S-type	SLLV, SLRV	X	X	0	1	0	0	0

Table 2: PC Control Signals

Type	Function	Stop bit	Zero flag	PCSrc	PCaddSrc1	PCaddSrc2
R-type	X	1	X	0	X	X
R-type	X	0	X	2	X	X
I-type	All-BEQ	1	X	0	X	X
I-type	All-BEQ	0	X	2	X	X
I-type	BEQ	1	X	0	X	X
I-type	BEQ	0	1	1	1	1
J-type	J	1	X	0	X	X
J-type	J	0	X	1	0	0
J-type	JAL	X	X	1	0	0
S-type	X	1	X	0	X	X
S-type	X	0	X	2	X	X

Table 3: ALU Control Signals

Type	Function	ALUsrc	ALUop
R-type	AND	0	AND
R-type	ADD	0	ADD
R-type	SUB	0	SUB
R-type	CMP	0	SUB
I-type	ADDI	1	ADD

I-type	SUBI	1	SUB
I-type	LW	1	ADD
I-type	SW	1	ADD
I-type	BEQ	0	SUB
J-type	J	X	X
J-type	JAL	X	X
S-type	SLL	1	SLL
S-type	SLR	1	SLR
S-type	SLLV	0	SLL
S-type	SLRV	0	SLR

Table 4: Stack Control Signals

Type	**Function**	**Stop bit**	**StR**	**StW**
R-type, I-type, S-type	X	0	0	0
R-type, I-type, S-type	X	1	1	0
J-type	J	0	0	0
J-type	J	1	1	0
J-type	JAL	X	0	1

3 Processer Simulation and Testing

Here is the revised description of the test cases for scenarios of executing multiple instructions one after another:

Unit testing has been performed for each component and instruction, as shown previously. The following test cases represent scenarios where multiple instructions are executed consecutively on the multicycle path. We have placed the binary values of the instructions in memory and followed their execution accordingly

We have created a code that performs a series of operations on integer variables, including logical shifts, comparisons, additions, and function calls. performs calculations based on their values, and stores the final result in memory:

3.1 Data Memory

Three variables are to be reserved in memory. Two previously stored integers: 2 and 1, which are stored in memory locations 0x00000000 and 0x00000004, respectively. The third integer is to be stored, i.e., the result of the code, at memory location 0x00000008. Note that our machine is a little Indian machine.

```
Mem[0]  = 0x02
Mem[1]  = 0x00
Mem[2]  = 0x00
Mem[3]  = 0x00
Mem[4]  = 0x01
Mem[5]  = 0x00
Mem[6]  = 0x00
Mem[7]  = 0x00
Mem[8]  = 0x00
Mem[9]  = 0x00
Mem[10] = 0x00
Mem[11] = 0x00
```

The tested sequence of instruction (test code) is shown in the table bellow, alongside with the instruction address, in addition to the binary and hexadecimal instruction representation.

Table 5: Testbench Instructions

Instruction Address	Assembly Instruction	Binary Representation	Hex Repre-sentation
0	ANDI $t0, $t0, 0	00000 00000 00000 00000000000000 10 0	0x00000004
4	LW $t1, 0($t0)	00010 00000 00001 00000000000000 10 0	0x10020004
8	LW $t2, 4($t0)	00010 00000 00010 00000000000100 10 0	0x10040024

12	ANDI $t3, $t3, 0	00000 00011 00011 00000000000000 10 0	0x00C60004
16	BEQ $t1, $t0, not_reached	00100 00000 00001 00000000000010 10 0	0x20020014
20	SLR $t3, $t2, 1	00001 00010 00011 XXXXX 00001 XXXX 11 0	0x088??0??
24	BEQ $t3, $t3, BTA	00100 00011 00011 00000000000001 10 0	0x20C6000C
28: not_reached	ADDI $t3, $t3, 10	00000 00011 00011 00000000001010 10 0	0x00C60054
32: BTA	SLLV $t3, $t1, $t2	00011 00001 00011 00010 XXXXX XXXX 11 0	0x18462???
36	JAL func	00001 000000000000000000001100 01 0	0x08000062
40	SW $t3, 8($t0)	00011 00000 00011 00000000001000 10 0	0x18060044
44: End	J End	00000 000000000000000000000000 01 0	0x00000002
48: func	sADD $t3, $t1, $t2	00001 00001 00011 00010 XXXXXXXXX 00 1	0x08462???

These instructions and data, were inserted to instruction and data memory manually, then a testbench with oscillating clock was run. The resulting waveform is shown in the figure bellow.

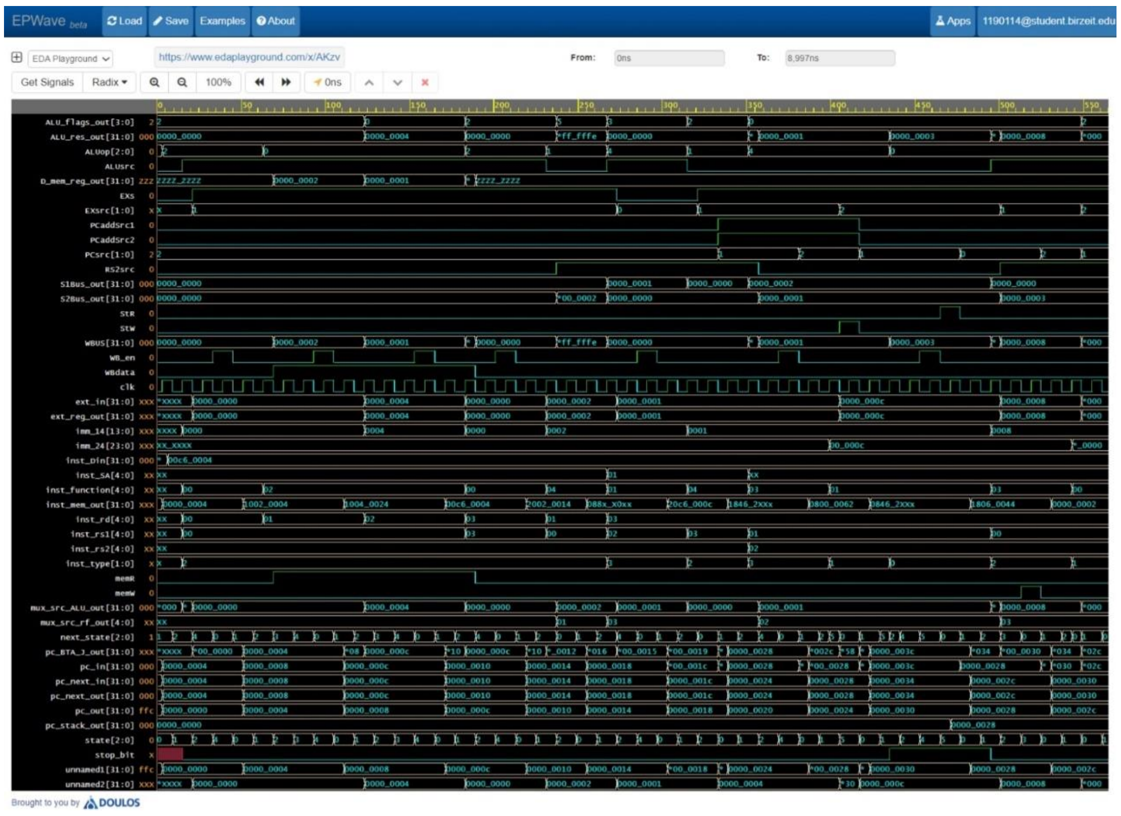


Figure 3.1:test CPU

It can be seen from the waveform that the flow of the instructions is executed as expected.

4 Conclusion

In conclusion, the design of a processor involves precise steps and thorough testing to ensure its correctness and performance. Testing includes evaluating all components and instructions, conducting scenarios to manipulate data and test branching, and verifying the design's functionality and accuracy. By implementing rigorous testing procedures, designers can have confidence in the processor's reliability and effectiveness in handling complex tasks.

5 References

[1] "Wikibooks," [Online]. Available: https://en.wikibooks.org/wiki/Microprocessor_Design/Multi_Cycle_Processors. [Accessed 2023 July 8].

[2] "USC Viterbi- School of Engineering," [Online]. Available: https://ee.usc.edu/~redekopp/ee357/slides/EE357Unit16_Multi_Cycle_CPU.pdf. [Accessed 2023 July 8].

6 Appendix

[1] CPU test(1) - EDA Playground