BIRZEIT UNIVERSITY

Faculty of Engineering & Technology Electrical & Computer Engineering Department

ENCS4370

Computer Architecture

Project #2 Report

Prepared by:

Diaa Badaha	1210478	sec 3
Nasri Attari	1210810	sec 3
Abdulrahim Khader	1210427	sec 3

Instructor: Dr. Aziz Qaroush

Date: 20 – 6 – 2024

Abstract

This report outlines the design and implementation of a multicycle datapath processor using Verilog. The project aims to create a detailed and efficient processor architecture by breaking down each instruction into multiple cycles and meticulously analyzing each component and control signal necessary for its execution. This approach allows for optimized resource utilization and precise control over data flow, enhancing the overall performance and reliability of the processor. The report covers the design phases, including building the datapath for different instruction types, state diagram, control unit configuration, and thorough simulation and testing to validate the design.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1. CPU

A central processing unit (CPU) is a crucial hardware element serving as the core computational unit within a server. It plays a vital role in converting data into digital signals and executing mathematical operations on them, facilitating computing in servers and other smart devices. The CPU is the primary processor of these signals, making computation feasible and acting as the brain of any computing device. It retrieves instructions from memory, executes the necessary tasks, and returns the results to memory. The CPU manages all computational activities needed to operate the operating system and applications.[1]

1.2. Datapath

A data path is a group of functional units, including arithmetic logic units (ALUs) and multipliers, that carry out data processing operations, along with registers and buses. Together with the control unit, it forms the central processing unit (CPU). A larger data path can be created by connecting multiple data paths using multiplexers.[2]

1.3. Multicycle Datapath

Such data paths exhibit variable cycles per instruction (CPI), allowing instructions to be divided into a series of arbitrary steps. Unlike single-cycle data paths, which execute an entire instruction in one long cycle, multi-cycle data paths execute one instruction at a time but break it into multiple shorter cycles. This requires additional registers to store intermediate results for subsequent steps. While single-cycle data paths have a uniform cycle time dictated by the slowest instruction, multi-cycle data paths optimize each cycle for specific tasks, enhancing efficiency and reducing overall cycle time. However, unlike single-cycle data paths, clock cycle overlapping is not feasible in multi-cycle implementations, as each step is processed sequentially across multiple cycles, ensuring precise and orderly execution.

To elaborate, in a single-cycle data path, each instruction completes its entire execution process within a single clock cycle. This means that the cycle time must be long enough to accommodate the slowest instruction, resulting in potentially inefficient use of time for faster instructions. On the other hand, a multi-cycle data path breaks down the execution of an instruction into several shorter cycles, allowing each step of the instruction to be completed in a separate clock cycle. This approach can lead to more efficient utilization of time as each clock cycle is optimized for specific tasks, though it necessitates the use of registers to pass data between cycles. Thus, while single-cycle data paths emphasize simplicity and speed for individual instructions, multi-cycle data paths focus on optimizing overall efficiency and resource utilization.[3]

1.4. Components Associated with the Datapath.[3]

1.4.1. Arithmetic Logic Unit (ALU)

The ALU is a fundamental component of the data path, performing logical and arithmetic operations such as addition, subtraction, division, multiplication, comparisons, and bitwise operations. It primarily takes input data from the registers and processes it to generate the necessary output. Operating on binary data, the ALU manipulates bits based on control signals received from the control unit.

1.4.2. Registers

Registers are high-speed storage elements within the processor. They temporarily store data during processing, which can include intermediate results, operands, or program counters. Registers are crucial for fast data access, reducing the need to retrieve data from memory, which can slow down operations. There are two types of registers: general-purpose registers, which hold data during computations, and special-purpose registers, which store program status flags or addresses.

1.4.3. BUS

A BUS is a communication system that transfers data between different components of a computer or between two computers. It consists of hardware components like optical fibers and wires, as well as software components such as communication protocols. The BUS facilitates the movement of data and instructions between memory, registers, and other peripherals. Together, the ALU, registers, and BUS form the data paths.

1.4.4. Multiplexers

Multiplexers in a data path select data from multiple sources and route it to the appropriate destination. They are essential for the movement of data within a processor, helping to select from different inputs and directing them to the focal component. For example, a multiplexer might choose between two registers to source data for an ALU operation.

1.4.5. Control Unit

The data path interacts closely with the control unit, which generates control signals that coordinate the activities of the data path components. The control unit interprets instructions fetched from memory and produces the necessary signals to control data movement, ALU operations, and register manipulations. It ensures that instructions are executed in the correct sequence and that the data path functions in a synchronized manner.

2. Design and Implementation

For the datapath design, we meticulously analysed each instruction individually and developed the RTL (Register Transfer Level) code for each one. This detailed approach allowed us to identify the specific requirements for every instruction, ensuring that we considered all necessary elements such as the functional units, multiplexers, control signals, and other essential components. By examining the instructions one by one, we could precisely determine what each instruction needed to function correctly within the processor. This included specifying the exact logic for operations like arithmetic calculations, data movement, and control flow.

Additionally, we carefully designed and incorporated the control signals required to coordinate the activities of different components within the datapath. These control signals ensured that each part of the processor operated in harmony with the others, adhering to the overall timing and functional requirements. We paid close attention to potential hazards and conflicts, implementing appropriate measures such as forwarding and pipeline stalling where necessary to prevent data hazards and ensure smooth execution. Our design process included rigorous testing and validation at each step to verify that each instruction performed as intended without causing any violations or disruptions to the rest of the datapath. This iterative process of design, test, and refinement allowed us to create a cohesive and efficient pipelined processor where every instruction seamlessly integrates into the overall architecture, resulting in a robust design capable of executing a wide range of instructions efficiently while maintaining the integrity and performance of the datapath. The design was done as follows:

2.1. Building the Datapath

2.1.1. R-Type Instructions

We have 3 instructions in this type, which are:

- **AND:** Reg(Rd) = Reg(Rs1) & Reg(Rs2)
- **ADD:** Reg(Rd) = Reg(Rs1) + Reg(Rs2)
- **SUB:** Reg(Rd) = Reg(Rs1) - Reg(Rs2)

The RTL description for these instructions is as follows:

- **Instruction Fetch:** Instruction ← MEM[PC]
- **Instruction Decode:** data1 ← Reg(Rs1), data2 ← Reg(Rs2)
- **Execution:** ALU_result ← func(data1, data2)
- **Write Back:** Reg(Rd) ← ALU_result
- **Next PC:** PC = PC + 2

From the RTL, it is clear that we need several functional units, including a PC register, an instruction memory, a register file, and an ALU. We also require a control signal to enable writing to the register file. The PC register is connected to the instruction memory, from which the instruction is fetched. In the register file, the fields Rs1, Rs2, and Rd from the instruction are mapped to Ra, Rb, and Rw, respectively. The outputs Ba and Bb from the register file serve as inputs to the ALU. The result produced by the ALU is then connected back to Bw in the register file for write-back purposes. The next PC value is updated to PC + 2. This entire setup is illustrated in the next figure.

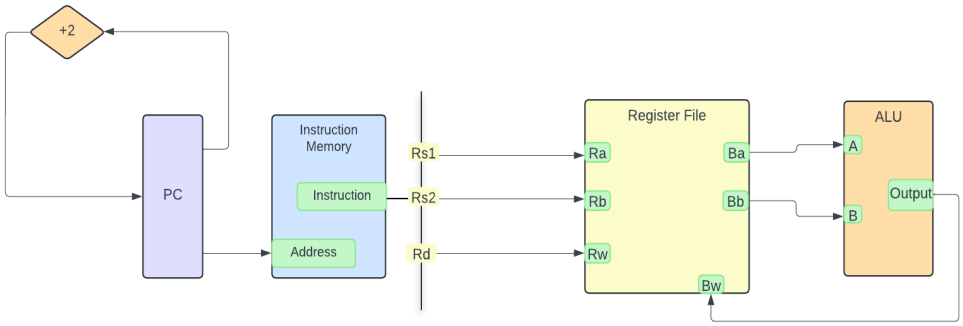

Figure 2-1: R-Type Instructions Datapath

2.1.2. ALU I-Type Instructions

Within this type, we have the following two instructions:

- **ADDI:** Reg(Rd) = Reg(Rs1) + Ext_16(Imm)
- **ANDI:** Reg(Rd) = Reg(Rs1) & Ext_16(Imm)

The corresponding RTL for these instructions is:

- **Instruction Fetch:** Instruction ← MEM[PC]
- **Instruction Decode:** data1 ← Reg(Rs1), data2 ← Ext_16(Imm)
- **Execution:** ALU_result ← func(data1, data2)
- **Write Back:** Reg(Rd) ← ALU_result
- **Next PC:** PC = PC + 2

These instructions are almost the same as R-Type instructions; the difference is that the second operand here is the 16-bit extended immediate value, whereas in R-Type instructions it is Rs2. To accommodate this, we need an extender to handle the immediate value and a control signal to specify whether the extension should be signed or unsigned. Additionally, a multiplexer is required to determine the second ALU operand, controlled by a signal that selects between the extended immediate and the value in Rs2. This setup ensures that the ALU can correctly process both I-Type and R-Type instructions, as illustrated in the following figure.

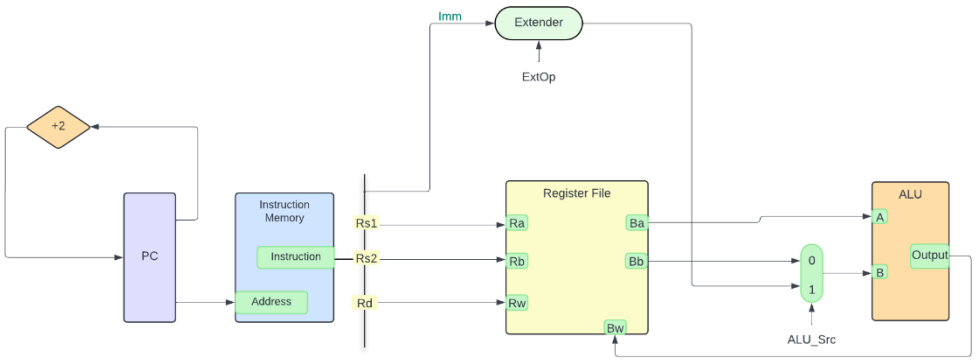

Figure 2-2: Datapath after Adding ALU I-Type Instructions

2.1.3. Load I-Type Instructions

The following three instructions fall under this type:

- **LW:** Reg(Rd) = Mem(Reg(Rs1) + Imm)
- **LBu:** Reg(Rd) = Mem(Reg(Rs1) + Imm)
- **LBs:** Reg(Rd) = Mem(Reg(Rs1) + Imm)

The RTL operations for these instructions are outlined below:

- **Instruction Fetch:** Instruction ← MEM[PC]
- **Instruction Decode:** data1 ← Reg(Rs1), data2 ← Ext_16(Imm)
- **Execute:** ALUout (address) ← data1 + data2
- **Memory Access:** MemData ← MEM[ALUout]
- **Write Back:** Reg(Rd) ← MemOut
- **Next PC:** PC ← PC + 2

From the RTL, it became evident that we need a Data Memory unit, with the ALU output serving as an input to it. A multiplexer has been added to select the data to be written back to the register file, choosing between the ALU output and the Memory output. Additionally, two more multiplexers have been included before the register file to select the Ra and Rw input bits, as their order differs between R-Type and I-Type instructions. These additions ensure the correct operation of both instruction types and are illustrated in Figure 2-3.

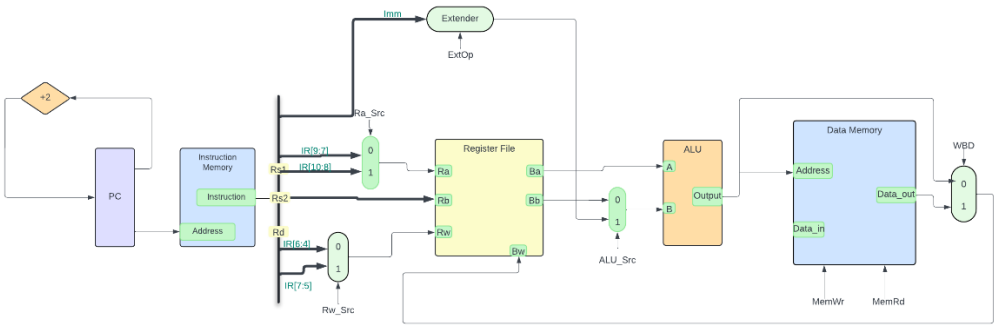

Figure 2-3: Datapath after Adding Load I-Type Instructions

2.1.4. Store I-Type Instruction

A single instruction is included in this type, which is:

- **SW:** Mem(Reg(Rs1) + Imm) = Reg(Rd)

With an RTL of:

- **Instruction Fetch:** Instruction ← MEM[PC]
- **Instruction Decode:** data1 ← Reg(Rs1), data2 ← Ext_16(Imm)
- **Execute:** ALUout (address) ← data1 + data2
- **Memory Access:** MEM[ALUout] ← Reg(Rd)
- **Next PC:** PC ← PC + 2

To implement this instruction, we added a multiplexer before the register file on Rb, allowing us to select between the outputs of the Rs2 and Rd multiplexers

as the input for Rb in the register file. Additionally, we needed to route data from the Rb the Data_in input of the data memory. These adjustments ensure proper data selection and routing for memory storage, as shown in Figure 2-4.

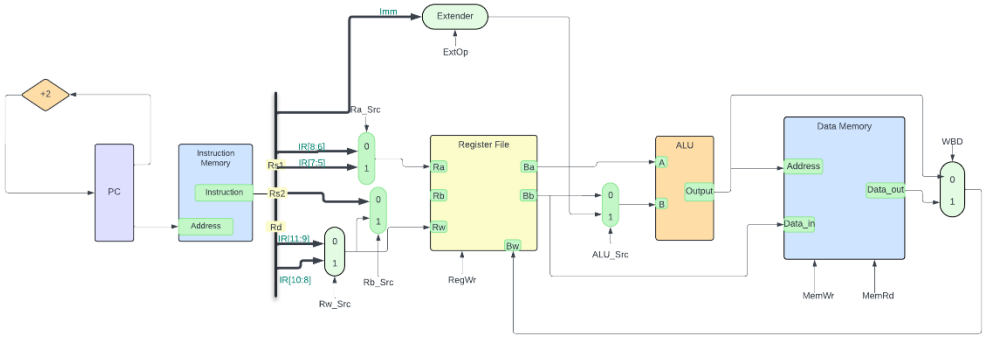

Figure 2-4: Datapath after Adding Store I-Type Instruction

2.1.5. Branch I-Type Instructions

This category includes the following instructions:

- **BGT:** if (Reg(Rd) > Reg(Rs1))
  Next PC = PC + sign_ext_16(Imm)
  else PC = PC + 2

- **BGTZ:** if (Reg(Rd) > Reg(R0))
  Next PC = PC + sign_ext_16(Imm)
  else PC = PC + 2

- **BLT:** if (Reg(Rd) < Reg(Rs1))
  Next PC = PC + sign_ext_16(Imm)
  else PC = PC + 2

- **BLTZ:** if (Reg(Rd) < Reg(R0))
  Next PC = PC + sign_ext_16(Imm)
  else PC = PC + 2

- **BEQ:** if (Reg(Rd) == Reg(Rs1))
  Next PC = PC + sign_ext_16(Imm)
  else PC = PC + 2

- **BEQZ:** if (Reg(Rd) == Reg(R0))
  Next PC = PC + sign_ext_16(Imm)
  else PC = PC + 2

- **BNE:** if (Reg(Rd) != Reg(Rs1))
  Next PC = PC + sign_ext_16(Imm)
  else PC = PC + 2

- **BNEZ:** if (Reg(Rd) != Reg(R0))
  Next PC = PC + sign_ext_16(Imm)
  else PC = PC + 2

The RTL for each instruction is detailed below:

- **BGT / BGTZ:**
  - ***Instruction Fetch:*** Instruction ← MEM[PC]
  - ***Instruction Decode:*** data1 ← Reg(Rs1 / R0), data2 ← Reg(Rd)
  - ***Execute:*** {ZF, NF} ← Sub(data1, data2)
  - ***Branch:*** if ({ZF, NF} == 2'bX1)
    PC = PC + 2 + sign_ext_16(offset)
    Else if ({ZF, NF} == 2'b00)
    PC = PC + 2
- **BLT / BLTZ:**
  - ***Instruction Fetch:*** Instruction ← MEM[PC]
  - ***Instruction Decode:*** data1 ← Reg(Rs1 / R0), data2 ← Reg(Rd)
  - ***Execute:*** {ZF, NF} ← Sub(data1, data2)
  - ***Branch:*** if ({ZF, NF} == 2'b00)
    PC = PC + 2 + sign_ext_16(offset)
    Else if ({ZF, NF} == 2'bX1)
    PC = PC + 2
- **BEQ / BEQZ:**
  - ***Instruction Fetch:*** Instruction ← MEM[PC]
  - ***Instruction Decode:*** data1 ← Reg(Rs1 / R0), data2 ← Reg(Rd)
  - ***Execute:*** ZF ← Sub(data1, data2)
  - ***Branch:*** if (ZF)
    PC = PC + 2 + sign_ext_16(offset)
    Else
    PC = PC + 2
- **BNE / BNEZ:**
  - ***Instruction Fetch:*** Instruction ← MEM[PC]
  - ***Instruction Decode:*** data1 ← Reg(Rs1 / R0), data2 ← Reg(Rd)
  - ***Execute:*** ZF ← Sub(data1, data2)
  - ***Branch:*** if (!ZF)
    PC = PC + 2 + sign_ext_16(offset)
    Else
    PC = PC + 2

From these RTL descriptions, we conclude that several modifications are necessary to ensure proper functionality:

- **Multiplexer for PC Register:** We need to add a multiplexer before the PC register to select the next PC address.
- **Multiplexer Inputs:** The first input of this multiplexer should be PC + 2, while the second input should be the output of an adder that computes PC + 2 + sign_ext_16(Imm).
- **Ra_Src Multiplexer:** We need to add a third option to the Ra_Src multiplexer, designated as 000, to allow R0 to be selected as the first operand (data1). Additionally, we need to introduce a control signal for this multiplexer to handle the selection.

These changes ensure that the processor can correctly handle the different branch conditions and update the PC accordingly. All edits are shown in Figure 2-5.

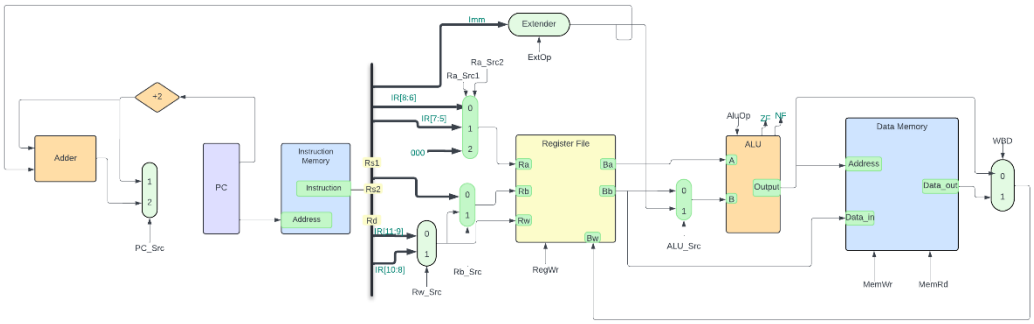

Figure 2-5: Datapath after Adding Branch Instructions

2.1.6. J-Type Instructions

The instructions in this type are:

- **JMP:** Next PC = {PC[15:10], Immediate}
- **CALL:** Next PC = {PC[15:10], Immediate}
  - PC + 2 is saved on R7
- **RET:** Next PC = R7

The RTL description for them is:

- **JMP:**
  - **Instruction Fetch:** Instruction ← MEM[PC]
  - **Target PC address:** target ← PC[15:10] || Imm
  - **Jump:** PC ← target

- **CALL:**
  - **Instruction Fetch:** Instruction ← MEM[PC]
  - **Target PC address:** target ← PC[15:10] || Imm
  - **Jump:** PC ← target
  - **Write Back:** Reg(R7) ← PC + 2

- **RET:**
  - **Instruction Fetch:** Instruction ← MEM[PC]
  - **Next PC address:** PC = Reg(R7)

From this RTL, the following modifications are necessary, as shown in Figure 2-6:

- **PC_Src Multiplexer:** We need to add two new options for the PC_Src multiplexer: one for the jump target address and another for the value in R7, used in the RET instruction. Additionally, a control signal must be added to manage this multiplexer.
- **Ra_Src Multiplexer:** A new selection option, designated as 111, should be added to the Ra_Src multiplexer to allow R7 to be chosen.
- **WB Data Multiplexer:** Another option should be added to the Write-Back (WB) data multiplexer to select PC + 2, enabling this value to be written to R7. An additional control signal is required for this multiplexer as well.

These enhancements ensure that the processor can handle jump and return instructions correctly.

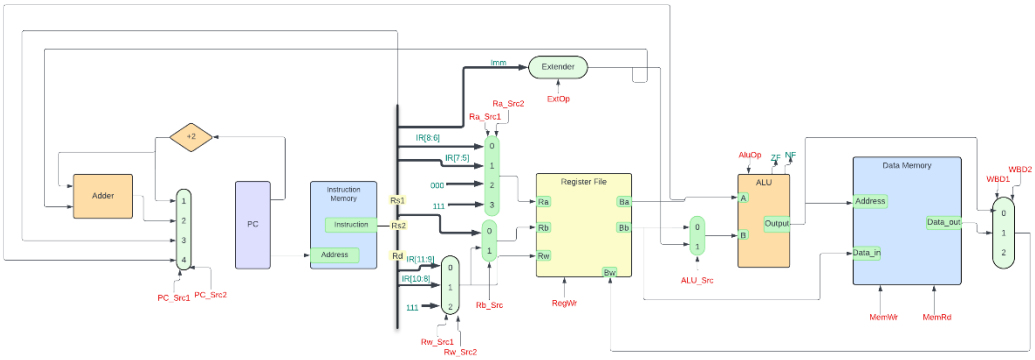

Figure 2-6: Datapath after Adding J-Type Instructions

2.1.7. S-Type Instructions

A single instruction is defined under this category:

- **Sv:** Mem[Rs] = Reg(R7)

And its RTL description is:

- **Instruction Fetch:** Instruction ← MEM[PC]
- **Instruction Decode:** Rs = Instruction[11:9]
- **Memory Access:** MEM[Rs] = Reg(R7)

According to that RTL, we need to implement the following changes, as shown in Figure 2-7:

- **Ra Multiplexer:** Add an additional option for Ra to account for the different placement of Rs in comparison to R-Type and I-Type instructions. This will require a new multiplexer with five choices.
- **Immediate Extender:** Add another extender to convert an 8-bit immediate to a 16-bit value.
- **Data Memory Input:** The output from the new extender will serve as an input to Data_in in the data memory.
- **Address Multiplexer:** Add a multiplexer to determine the address to be written, selecting between ALUout and Ra (where Ra corresponds to Rs in this case).

These modifications ensure the proper handling of the instruction set and maintain the processor's functionality.

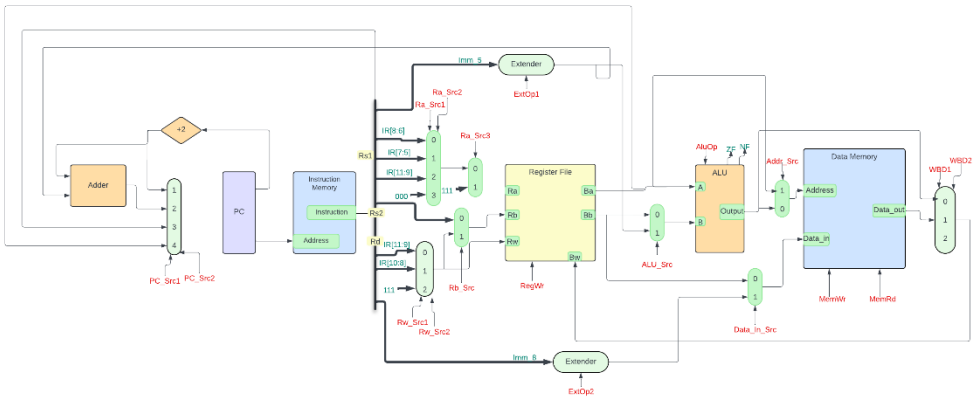

Figure 2-7: Datapath after Adding S-Type Instruction

After implementing all the necessary configurations based on each instruction's RTL, the final datapath, divided into stages, is illustrated in the following figure. This comprehensive setup ensures that all instructions are processed correctly and efficiently within the processor.You can view it clearly as a PDF file from the link: https://drive.google.com/file/d/1JpDpMJGJ8eZuW4PBRkB8Ps6pOvtvx89Y/view?usp=sharing

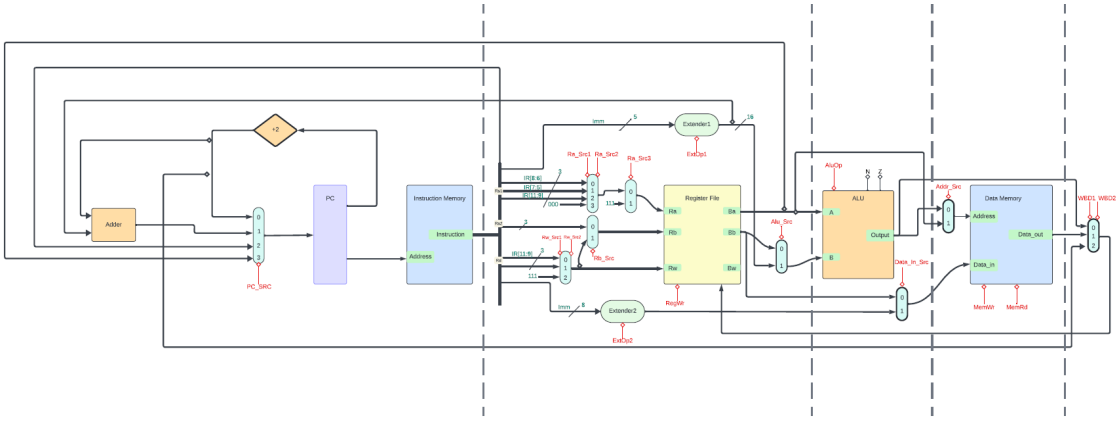

Figure 2-8: Final Datapath

2.2. State Diagram

Following a state diagram is essential for the datapath to function effectively in a multicycle processor. It ensures that each instruction is executed in a series of well-defined steps, allowing for optimal use of the processor's resources and precise control of data flow. This systematic approach helps in coordinating the complex interactions between various components, ensuring that each stage of instruction processing is completed before moving to the next.

The state diagram guides the processor through the necessary states for instruction fetch, decode, execution, memory access, and write-back stages. By following these steps, the processor can handle multiple instructions efficiently, reducing the chances of errors and improving overall performance. Additionally, it provides a clear framework for troubleshooting and optimizing the processor's operations.

The state diagram for our processor, illustrating these stages and transitions, is shown in the next figure. You can view it clearly by following this link: https://drive.google.com/file/d/1d-eMTWeiPmc2jlQuEaYdm5L7edjfYaFm/view?usp=sharing. This diagram is crucial for understanding how each part of the processor works together to execute instructions seamlessly.

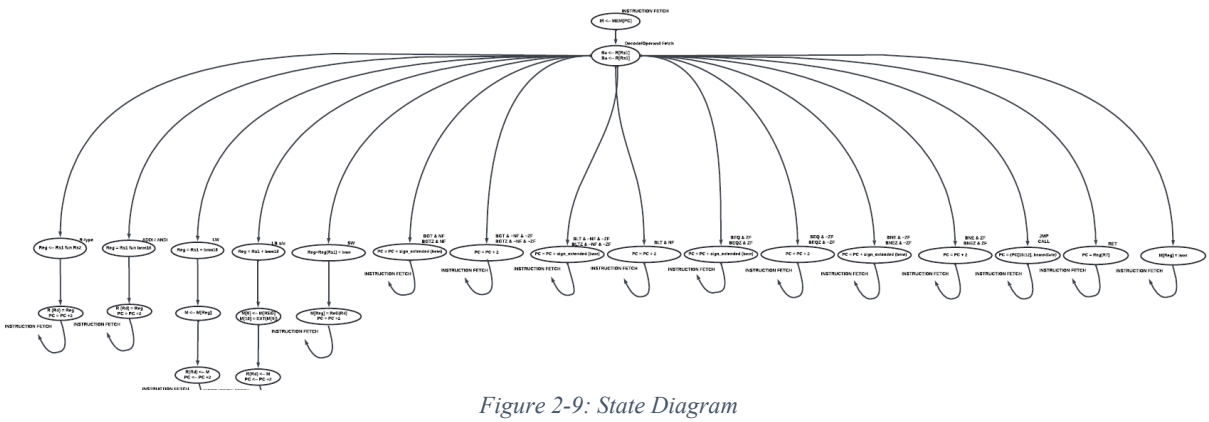

Figure 2-9: State Diagram

2.3. Control Unit

Ensuring that the control signals are assigned correctly for each specific instruction is one of the most essential and critical aspects of making the datapath function effectively. These control signals dictate the operation of various components within the processor, such as the ALU, multiplexers, and memory units, ensuring they perform the right operations at the right times. Without precise control signal assignments, the datapath would not be able to execute instructions correctly, leading to errors and inefficiencies. Therefore, meticulous attention to the control signal specifications for each instruction is vital for the overall performance and reliability of the processor.

In our processor's control unit, the control signals are meticulously designed and specified to match the requirements of each instruction type. These signals include settings for ALU operations, memory access, register file operations, and branch decisions, among others. The exact configuration of these control signals ensures that every component of the datapath is coordinated seamlessly, enabling accurate and efficient execution of instructions. By carefully assigning and managing these signals, we can optimize the processor's performance and minimize the risk of errors. The control signals used in our control unit are detailed in the following tables, which provide a comprehensive overview of how each signal is utilized to manage the processor's operations effectively. This detailed approach highlights the importance of precision and coordination in processor design, ensuring robust and reliable functionality.

Table 2-1: Control Signals - 1

	Ra SRC1	Ra SRC2	Ra SRC3	Rb SRC	Rw SRC1	Rw SRC2	EXT OP1	EXT OP2	REG WR	ALU SRC	Addr Src	DATA IN SRC	MEM WR	MEM RD	WB DATA1	WB DATA2
R-Type Instructions																
AND	0	0	0	0	0	0	X	X	1	0	X	X	0	0	0	0
ADD	0	0	0	0	0	0	X	X	1	0	X	X	0	0	0	0
SUB	0	0	0	0	0	0	X	X	1	0	X	X	0	0	0	0
I-Type Instructions																
ADDI	0	1	0	X	0	1	1	X	1	1	X	X	0	0	0	0
ANDI	0	1	0	X	0	1	0	X	1	1	X	X	0	0	0	0
LW	0	1	0	X	0	1	1	X	1	1	0	X	0	1	0	1
LBu	0	1	0	X	0	1	1	X	1	1	0	X	0	1	0	1
LBs	0	1	0	X	0	1	1	X	1	1	0	X	0	1	0	1
SW	0	1	0	1	0	1	1	X	0	1	0	0	1	0	X	X
BGT	0	1	0	1	0	1	1	X	0	0	X	X	0	0	X	X
BGTz	1	1	0	1	0	1	1	X	0	0	X	X	0	0	X	X
BLT	0	1	0	1	0	1	1	X	0	0	X	X	0	0	X	X
BLTz	1	1	0	1	0	1	1	X	0	0	X	X	0	0	X	X
BEQ	0	1	0	1	0	1	1	X	0	0	X	X	0	0	X	X
BEQz	1	1	0	1	0	1	1	X	0	0	X	X	0	0	X	X
BNE	0	1	0	1	0	1	1	X	0	0	X	X	0	0	X	X
BNEz	1	1	0	1	0	1	1	X	0	0	X	X	0	0	X	X
J-Type Instructions																
JMP	X	X	X	X	X	X	X	X	0	X	X	X	0	0	X	X
CALL	X	X	X	X	1	0	X	X	1	X	X	X	0	0	1	0
RET	X	X	1	X	X	X	X	X	0	X	X	X	0	0	X	X
S-Type Instructions																
SV	1	0	0	X	X	X	X	1	0	X	1	1	1	0	X	X

That table provides a detailed overview of the control signals required for different instruction types in our processor's control unit. It is divided into sections for R-Type, I-Type, J-Type, and S-Type instructions, each specifying the settings needed for various components.

- **Ra SRC1, Ra SRC2, Ra SRC3:** These columns define the source registers for the first operand in different contexts.
- **Rb SRC:** This column specifies the source register for the second operand.
- **Rw SRC1, Rw SRC2:** These columns indicate the destination registers.
- **EXT OP1, EXT OP2:** These columns manage the immediate value extension operations (signed / unsigned).
- **REG WR:** This column shows whether a write operation to the register file is enabled.
- **ALU SRC:** This column specifies the source for the ALU operation.
- **Addr Src:** This column indicates the source for the memory address.
- **DATA IN SRC:** This column specifies the source of data to be written into memory.
- **MEM WR, MEM RD:** These columns indicate whether a memory write or read operation is enabled.
- **WB DATA1, WB DATA2:** These columns specify the sources of data for the write-back stage.

For each instruction, the table uses binary values (0 or 1) and 'X' to denote specific settings. For example, the ADD instruction has the Ra SRC1, Ra SRC2, Rb SRC, Rw SRC1, and ALU SRC columns all set to 0, indicating specific register and ALU settings required for execution. The I-Type instructions like ADDI and ANDI have their own unique settings, such as enabling immediate value extension (EXT OP1 or EXT OP2). This table ensures that each instruction is executed with the appropriate control signals, facilitating precise and efficient operation of the processor.

Following the detailed control signals table, we derived specific logic equations for each control signal. These equations define the exact conditions under which each control signal is activated, ensuring that the datapath operates correctly for every instruction type. By using these equations, we can implement a control unit that generates the appropriate control signals based on the opcode and other relevant inputs.

- **RaSRC1** = BGTz + BLTz + BEQz + BNEZ + SV
- **RaSRC2** = ADDI + ANDI + LW + LBu + LBs + SW + BGT + BGTz + BLT + BLTz + BEQ + BEQz + BNE + BNEz
- **RaSRC3** = RET
- **RbSRC** = SW + BGT + BGTz + BLT + BLTz + BEQ + BEQz + BNE + BNEz
- **RwSRC1** = CALL
- **RwSRC2** = ADDI + ANDI + LW + LBu + LBs + SW + BGT + BGTz + BLT + BLTz + BEQ + BEQz + BNE + BNEz
- **EXTOP1** = ADDI + LW + LBu + LBs + SW + BGT + BGTz + BLT + BLTz + BEQ + BEQz + BNE + BNEZ
- **EXTOP2** = SV
- **REGWR** = AND + ADD + SUB + ADDI + ANDI + LW + LBu + LBs + CALL
- **ALUSRC** = ADDI + ANDI + LW + LBu + LBs + SW

- **AddrSRC** = SV
- **DATAINSRC** = SV
- **MEMWR** = SW + SV
- **MEMRD** = LW + LBu + LBs
- **WBDATA1** = CALL
- **WBDATA2** = LW + LBu + LBs

Table 2-2: Control Signals - 2

Opcode Value	Instr	ALUOp	2-bit Coding
0000	AND	AND	00
0001	ADD	ADD	01
0010	SUB	SUB	10
0011	ADDI	ADD	01
0100	ANDI	AND	00
0101	LW	ADD	01
0110	LBu	ADD	01
0110	LBs	ADD	01
0111	SW	ADD	01
1000	BGT	SUB	10
1000	BGTZ	SUB	10
1001	BLT	SUB	10
1001	BLTZ	SUB	10
1010	BEQ	SUB	10
1010	BEQZ	SUB	10
1011	BNE	SUB	10
1011	BNEZ	SUB	10
1100	JMP	X	X
1101	CALL	X	X
1110	RET	X	X
1111	Sv	X	X

The past table provides a detailed mapping of opcode values, instruction names, corresponding ALU operations, and their 2-bit coding for the instructions in our processor.

- **Opcode Value:** Binary values assigned to each instruction.
- **Instr:** Instruction names, such as AND, ADD, SUB, etc.
- **ALUOp:** Specifies the ALU operation for each instruction (e.g., AND, ADD, SUB).
- **2-bit Coding:** Binary coding for the ALU operations, with '00' for AND, '01' for ADD, '10' for SUB, and 'X' for operations like JMP, CALL, and RET that don't use the ALU.

This table ensures that each instruction is correctly encoded and processed by the ALU, supporting efficient execution within the processor.

Table 2-3: Control Signals - 3

OPCODE	ZF	NF	PCSRC
R-type	X	X	0
BGT | BGTZ	0	0	0
BGT | BGTZ	X	1	1
BLT | BLTZ	X	1	0
BLT | BLTZ	0	0	1
BEQ | BEQZ	0	X	0
BEQ | BEQZ	1	X	1
BNE | BNEZ	1	X	0
BNE | BNEZ	0	X	1
JMP | CALL	X	X	2
RET	X	X	3
OTHERS	X	X	0

This table is used for the PC source multiplexer (PCSRC) in our processor, which determines the next PC address based on the values of the Zero Flag (ZF) and Negative Flag (NF) after the execution stage. The table specifies how the opcode and the status of ZF and NF influence the selection of the next PC address.

- **R-type:** Independent of ZF and NF, the next PC is set to the current PC + 2.
- **BGT | BGTZ:** If ZF is 0 and NF is 0, the next PC is set to the branch address (PC + sign_extended(Imm)). If ZF is X and NF is 1, the next PC is also set to the branch address.
- **BLT | BLTZ:** If ZF is X and NF is 1, the next PC is set to the branch address. If ZF is 0 and NF is 0, the next PC is set to the current PC + 2.
- **BEQ | BEQZ:** If ZF is 0 and NF is X, the next PC is set to the current PC + 2. If ZF is 1 and NF is X, the next PC is set to the branch address.
- **BNE | BNEZ:** If ZF is 1 and NF is X, the next PC is set to the current PC + 2. If ZF is 0 and NF is X, the next PC is set to the branch address.
- **JMP | CALL:** The next PC is set to the jump or call target address.
- **RET:** The next PC is set to the address stored in R7.
- **OTHERS:** For any other instructions, the next PC is set to the current PC + 2.

This table ensures that the processor correctly determines the next PC address based on the outcomes of branch and jump instructions, thereby facilitating proper program flow control.

The equations derived from the control truth table are essential for determining the next program counter (PC) address based on branch and jump instructions. These equations consider various instruction types and their condition flags, such as the zero flag (ZF) and the negative flag (NF). By accurately generating branch and jump signals, the processor can correctly manage control flow, branching, jumping or returning to the appropriate address when specific conditions are met. The following equations detail the logic for branch, jump, and return signals based on the opcode and condition flags. That equations are as follows:

- **Branch** = ((BGT + BGTZ) . NF) + ((BLT + BLTZ) . ~ZF . ~NF) + ((BEQ + BEQZ) . ZF) + ((BNE + BNEZ) . ~ZF)
- **Jump** = JMP + CALL
- **Return** = RET

3. Simulation and Testing

3.1. PC Unit

This module, which its code is found in Appendix A, manages the Program Counter (PC) in a processor. It updates the PC based on the type of instruction being executed (whether it's a default sequential increment, a branch, a jump, or a return from a subroutine) using control signals and inputs for immediate values, jump offsets, and return addresses. The module ensures the correct next address is selected and loaded into the PC at each clock cycle when enabled.

```
1  `include "constants.v"
2  module pcModule_tb;
3
4      wire clock;
5      ClockGenerator clock_generator(clock);
6
7      reg [1:0] PCsrc = pcDefault;
8      reg signed [15:0] imm16;
9      reg [11:0] imm12;
10     reg [15:0] ret;
11     reg EN;
12     wire [15:0] PC;
13
14
15     pcModule Module(clock, PC, PCsrc, imm16, imm12, ret, EN);
16
17     initial begin
18         EN = HIGH;
19         #0
20
21         PCsrc <= pcDefault;
22         #10
23
24         PCsrc <= pcBRT;
25         imm16 <= 16'd10;
26         #10
27
28         PCsrc <= pcJMP;
29         imm12 <= 12'd10;
30         #10
31
32         PCsrc <= pcRET;
33         ret <= 16'd8;
34         #10
35
36         #10
37         $finish;
38     end
39
40 endmodule
```

Figure 3-1: PC Module Testbench

The testbench in Figure 3-1 ensures that the pcModule functions correctly by assigning a different value to the PC every 10ps. It systematically tests various control signals and immediate values, verifying that the module updates the PC as expected at each rising edge of the clock. This thorough testing confirmed that the pcModule operates accurately, adjusting the PC based on the provided instructions and control signals. The successful operation of the module is illustrated in Figure 3-2.

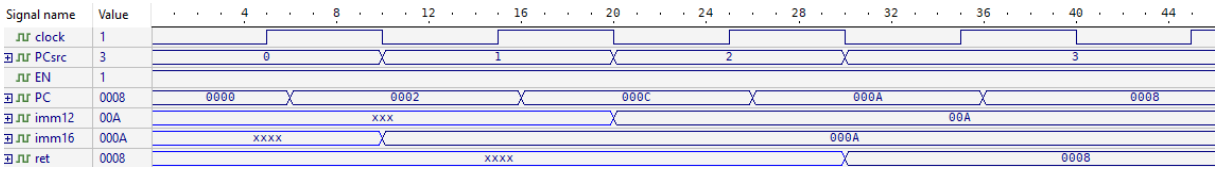

Figure 3-2: PC Testbench Waveform

3.2. ALU Module

This module, found in Appendix A, defines an Arithmetic Logic Unit (ALU) in Verilog. It performs basic arithmetic and logic operations based on the ALUOp control signal, such as AND, ADD, and SUB. The module takes two 16-bit operands (A and B), and outputs the result (Output). It also generates zero (zeroFlag) and negative (nFlag) flags based on the result. The ALU operations are enabled or disabled using the EN signal, ensuring correct execution and output.

```
1  `include "constants.v"
2  module ALU_tb;
3
4      wire EN;
5
6      ClockGenerator clock_generator(EN);
7
8      reg [15:0] A, B;
9      wire [15:0] Output;
10     wire zeroFlag;
11     wire nFlag;
12
13     // signals
14     reg [1:0] ALUop;
15
16
17     ALU alu(A, B, Output, zeroFlag, nFlag, ALUop, EN);
18
19     initial begin
20         #0
21         A <= 16'd10;
22         B <= 16'd20;
23         ALUop <= ALU_AND;
24         #10
25
26         A <= 16'd30;
27         B <= 16'd20;
28         ALUop <= ALU_ADD;
29         #10
30
31         A <= 16'h0FFF;
32         B <= 16'h0F0F;
33         ALUop <= ALU_SUB;
34         #10
35
36         #5 $finish;
37     end
38 endmodule
```

Figure 3-3: ALU Module Testbench

This testbench is designed to verify the functionality of the ALU module. Initially, it assigns A to 10 and B to 20. The first operation performed is the AND operation, which should result in 0 since the binary AND of 10100 and 01010 is 0 (Z flag in this case will be one). The second operation is the addition of these two values, which correctly results in 30. Next, the testbench assigns A to 4095 and B to 3850. The subsequent subtraction operation between these values results in 240, as shown in the waveform. All these results confirm the correct operation of the ALU as depicted in Figure 3-4.

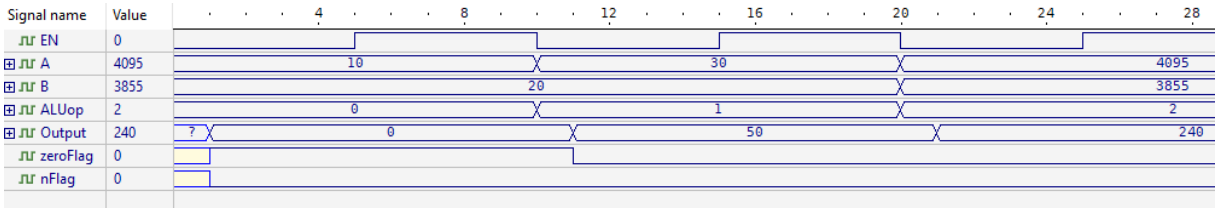

Figure 3-4: ALU Testbench Waveform

3.3. Clock Generator

This module, found in Appendix A, generates a clock signal with a 10ns period, which is crucial for synchronizing operations across various modules in our processor design. The clock starts low, which is important for the initial instruction fetch, and it toggles every 5ns, providing a consistent timing reference for the entire system. This clock function will be utilized in most of our modules to ensure proper timing and coordination.

```
1  module ClockGenerator_tb;
2
3      reg clock;
4
5      // Instantiate the ClockGenerator module
6      ClockGenerator uut (clock);
7
8      initial begin
9          $display("Simulation started at time %0t", $time);
10     end
11
12     // Monitor block to display clock changes
13     always @(posedge clock) begin
14         $display("(%0t) Clock toggled to %b", $time, clock);
15     end
16
17     // Stop simulation after some time
18     initial #100 $finish;
19
20 endmodule
```

Figure 3-5: Clock Generator Module Testbench

This testbench ensures the functionality of the clock module by simulating the clock signal and monitoring its output. The testbench instantiates the ClockGenerator module and uses a display block to show when the clock toggles. As shown in the console output in Figure 3-6, the clock starts at time 0, toggles every 10ns, and the state changes are displayed accordingly. This consistent toggling verifies that the clock module is operating correctly, generating a reliable clock signal for use in other modules. The simulation stops after 100ns, confirming the clock's periodic behavior throughout the test.

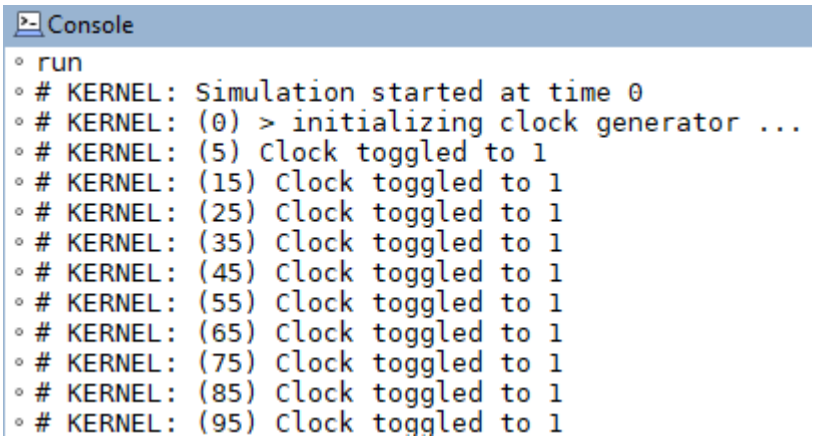

Figure 3-6: Clock Generator Testbench Output

3.4. Instruction Memory

Instruction memory module, named instructionMemory, is responsible for handling the instruction memory in the processor. It contains six instructions to test the functionality of the memory and instruction handling. The instructions included are:

1. ANDI R1, R2, 5
2. CALL 120
3. ANDI R1, R2, 5
4. ADDI R1, R2, 5
5. RET
6. ADDI R1, R2, 5

These instructions are stored in a byte-addressable memory array and are fetched based on the provided address. The module ensures that the correct instruction and opcode are outputted for execution, with the instructions stored in little-endian format to match the processor's requirements. This module can be found in Appendix A.

```
1  `include "constants.v"
2
3  module instruction_mem_tb;
4
5      reg [15:0] address;
6      reg clock;
7      wire [15:0] instruction;
8      wire [3:0] opcode;
9
10
11     instructionMemory uut(clock, address, instruction, opcode);
12
13     initial begin
14         // Initialize
15         clock = 0;
16         address = 0;
17         #50
18         #10 clock = 1; address <= 0;   // ANDI R1, R2, 5
19         #10 clock = 0;
20         #10 clock = 1; address <= 2;   // CALL 120
21         #10 clock = 0;
22         #10 clock = 1; address <= 4;   // ANDI R1, R2, 5
23         #10 clock = 0;
24         #10 clock = 1; address <= 120;   // ADDI R1, R2, 5
25         #10 clock = 0;
26         #10 clock = 1; address <= 122; // ADDI R1, R2, 5
27         #10 clock = 0;
28         #50 $finish;
29     end
30
31 endmodule
```

Figure 3-7: Instruction Memory Module Testbench

This testbench assigns different addresses as an input to the instructionMemory module to verify if the instructions are read correctly from memory. The testbench systematically sets the clock and address signals to check multiple instruction fetches, ensuring the memory outputs the correct instructions and opcodes. The results confirm the correct operation, as illustrated in Figure 3-8, which shows the waveform of the testbench, indicating accurate memory reads at each specified address.

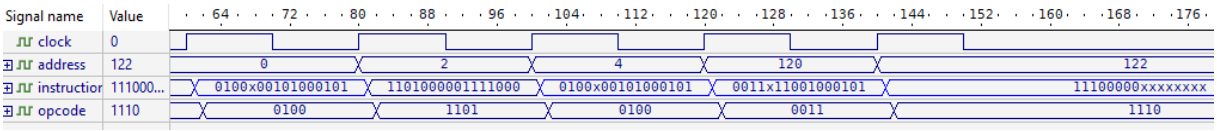

Figure 3-8: Instruction Memory Testbench Waveform

3.5. Data Memory

This module, located in Appendix A, defines the dataMemory unit. It manages memory read and write operations using various control signals. The module allows selection between different address and data input sources, such as the ALU output, Ba register, and a special store value. It supports byte-based load operations with either sign-extension or zero-extension, as well as 16-bit data reads and writes in little-endian format. The memory is preloaded with specific values for testing, ensuring that the module functions correctly across different scenarios.

```
28      // Initialize inputs
29      initial begin
30          #0
31          EN <= 0;
32
33          // Read Operation test
34          #1
35          ENW <= 0;
36          ENR <= 1;
37          EN <= 1;
38          address_src <= 0;
39          OpCode <= LB;
40          mode <= LOW;
41          ALUaddress <= 16'h0000;
42          #5
43          EN <= 0;
44          #5
45
46          // Write operation test
47          ENW <= 1;
48          ENR <= 0;
49          EN <= 1;
50          address_src <= 1;
51          data_in_src <= 0;
52          OpCode = SW;
53          mode = 0;
54          BaAddress <= 16'h0000;
55          regData <= 16'hABCD;
56          #5
57          EN <= 0;
58          #5
59
60          // Read operation test
61          ENW <= 0;
62          ENR <= 1;
63          EN <= 1;
64          address_src <= 1;
65          BaAddress <= 16'd15;
66          OpCode <= LW;
67          mode <= HIGH;
68          #10;
69
70          $finish;
71      end
```

Figure 3-9: Values for Testing in Data Memory Testbench

The Data Memory testbench code, fully detailed in Appendix A, is designed to rigorously validate the functionality of our data memory module through a series of specific memory operations. These operations ensure that both reading and writing processes are correctly implemented, adhering to the expected behavior.

1. **Read Test**: The initial test involves loading a byte from address 0. This operation checks the basic functionality of the memory read process, ensuring that data retrieval from a specified address is accurate.

2. **Write Test**: In this test, the value ABCD is written to address 0. Due to the little-endian encoding used in our design, the high-order byte (AB) is stored in address 1, while the low-order byte (CD) is stored in address 0. This test verifies the proper handling of data storage and confirms that the little-endian format is correctly implemented in our memory module.

3. **Read Test**: The final test reads a word from address 15. This operation is crucial for validating that our memory module can handle word-sized data correctly and ensures that data integrity is maintained when accessing larger data sizes.

These comprehensive tests were performed successfully, demonstrating that the data memory module operates as intended. The results, as shown in the waveform in the next figure, confirm the correctness and reliability of our data memory implementation. Each operation's successful execution reinforces the module's capability to handle both byte and word-sized data accurately, adhering to the specified memory addressing and data encoding formats.

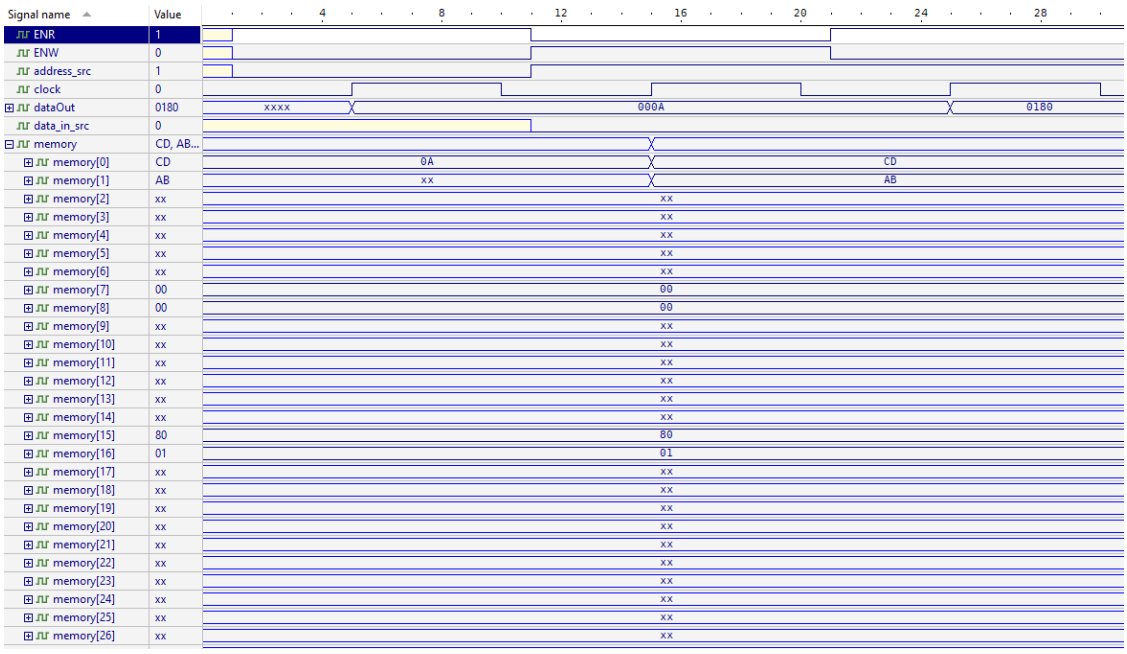

Figure 3-10: Data Memory Testbench Waveform

3.6. Register File

A module defines a registerFile is designed for this part, it is responsible for managing a set of 8 registers, each 16 bits wide. It allows reading from two registers (Ra and Rb) and writing to a register (Rw). The module uses the EN and ENW signals to control read and write operations, respectively. When enabled, it reads data from the specified registers into Ba and Bb on the positive edge of the clock and writes data from Bw to the specified register if the write enable signal is active. The initial block sets all registers to zero at the start. This module is included in Appendix A.

```
13      initial begin
14
15          #0
16          EN <= 0;
17          #1
18          RW <= 3'd1; // Write to register 1
19          BusW <= 16'd16; // write value 16
20          EnW <= 1; // Enable write
21          EN <= 1;
22          #5
23          EN <= 0;
24
25          #5
26          RW <= 3'd2; // Write to register 2
27          BusW <= 16'd32; // Write value 32
28          EnW <= 1; // Enable write
29          EN <=1;
30          #5
31          EN <= 0;
32
33          #5
34          RA <= 3'd1;
35          RB <= 3'd2;
36          EnW <= 0;
37          EN <= 1;
38          #5
39          EN <= 0;
40
41          #5
42          RW <= 3'd2; // Write to register 2
43          BusW <= 16'd64; // Write value 64
44          EnW <= 1; // Enable write
45          EN <= 1;
46          #5
47          EN <= 0;
48
49          #5
50          RA <= 3'd3;
51          RB <= 3'd0;
52          EnW <= 0;
53          EN <= 1;
54          #10
55
56          #10
57          $finish;
58      end
```

Figure 3-11: Values for Testing in Register File Testbench

The register file testbench, as shown in the previous code snippets, includes five specific tests to verify its functionality:

1. **Write Test**: Write the value 16 to register R1.
2. **Write Test**: Write the value 32 to register R2.
3. **Read Test**: Read the value from R1 to BusA and from R2 to BusB.
4. **Write Test**: Write the value 64 to register R1.
5. **Read Test**: Read the value from R3 to BusA and from R0 to BusB.

These tests were executed successfully, demonstrating the correct operation of the register file. The results are confirmed by the waveform shown in the next figure.

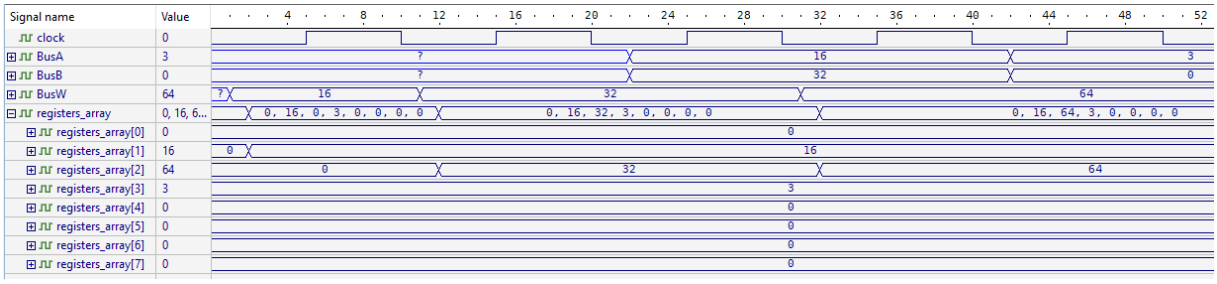

Figure 3-12: Register File Testbench Waveform

3.7. Control Unit

The Control Unit module orchestrates the various control signals and stages of the processor. It handles the execution of instructions by managing the control signals for different stages such as instruction fetch, decode, execute, memory access, and write-back. The module uses the opcode and flag inputs to determine the appropriate control signals and the next stage of execution. It includes logic for handling different instruction types, branching, and memory operations.

The complexity of the module is highlighted by its extensive logic and numerous control signals, making it over 300 lines in length. Due to its substantial size, including the full code within the main body of the report was impractical. Both the control unit module code and its testbench are found in the project submission file. The testbench of the CU includes all instructions of the given subset, showing all the signals that are to be outputted to the CPU.

The next 2 figures show the waveform of the testbench to test control signals. Compared with tables 2.1, 2.2, and 2.3, the waveform demonstrates the correctness of our implementation of the control unit module.

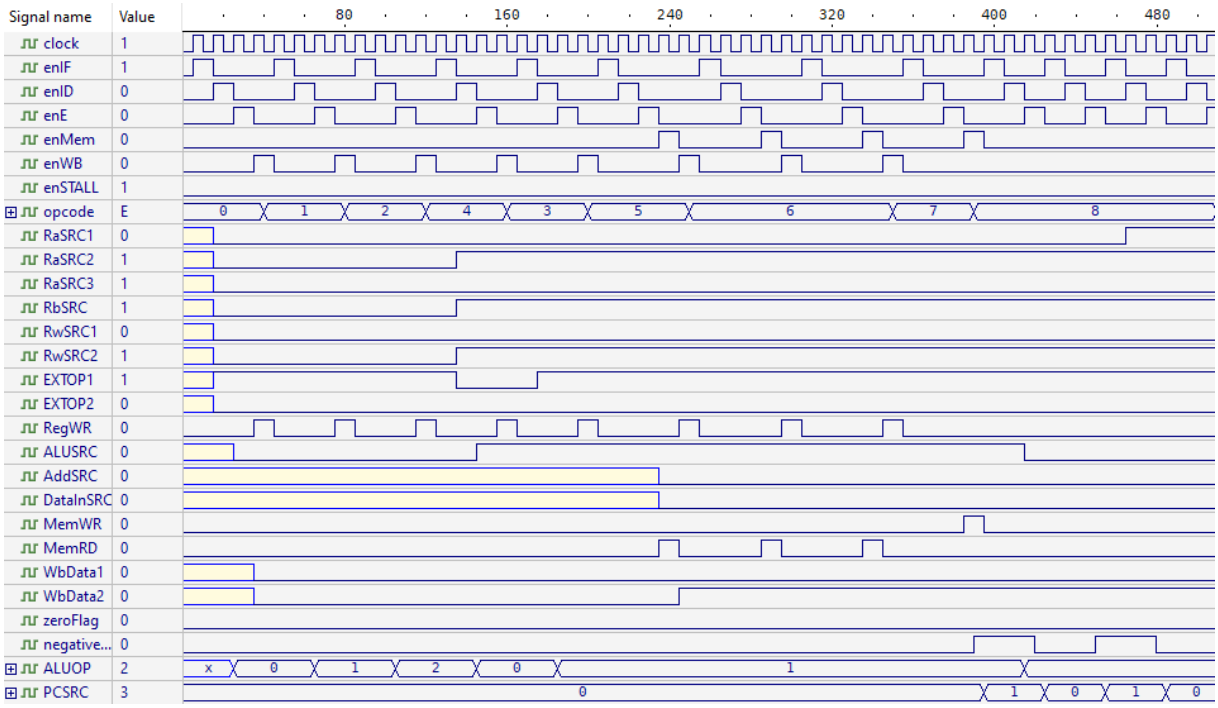

Figure 3-13: Control Unit Waveform – 1

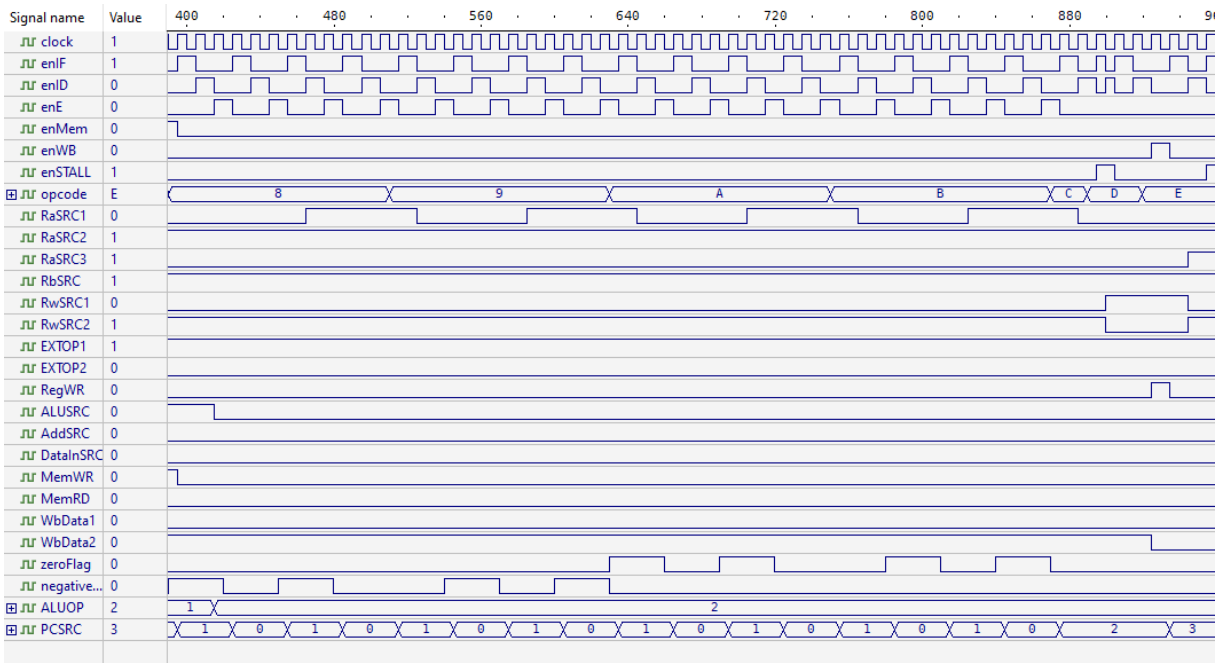

Figure 3-14: Control Unit Waveform – 2

3.8. Processor

The processor is designed to connect and coordinate all the different stages and modules necessary to execute the specified subset of operations. Constructed based on the carefully designed datapath, the processor's wiring is intricately aligned with the requirements of the instruction set. This meticulous wiring ensures that each component operates seamlessly within the overall architecture. The complete processor code is included in the project submission file. Test instructions were preloaded into the instruction memory to validate the processor's functionality, ensuring that it performs as expected across various scenarios.

In order to test the processor with various instructions, the instruction memory was edited and the following instructions were added:

- **ANDI:** R1 = R2 & 5
- **CALL:** Call subroutine at address 120, R7 = 4
- **ANDI:** R1 = R2 & 3
- **ADDI:** R6 = R2 + 5
- **RET:** Return from subroutine
- **ADDI:** R5 = R5 + 12
- **SUB:** R1 = R2 – R1
- **LW:** R1 = Mem[R0 + 15], Mem[15] = 8'd128 and Mem[16] = 8'd1
- **BGT:** If R0 > R1, PC = PC + 15 (branch not taken)
- **SW:** Mem[Reg[R0] + 15] = Reg(R4) = 8'd4
- **JMP:** Jump to address 120

These instructions are shown in Figure 3-15.

```
// ANDI R1 = R2 & 5
memory[0] = {R2, 5'd5};
memory[1] = {ANDI, X1, R1};
// CALL subroutine at address 120 ---- R7 = 4
memory[2] = {8'd120};
memory[3] = {CALL, 4'b0};
// ANDI R1 = R2 & 3
memory[4] = {R2, 5'd3};
memory[5] = {ANDI, X1, R1};
// ADDI R6 = R2 + 5
memory[120] = {R2, 5'd5};
memory[121] = {ADDI, X1, R6};
// RET return from subroutine
memory[122] = 8'bx;
memory[123] = {RET, 4'b0};
// ADDI R5 = R5 + 12
memory[6] = {R5, 5'd12};
memory[7] = {ADDI, X1, R5};
// SUB R1, R2, R1
memory[8] = {2'b10, R1, X3};
memory[9] = {SUB, R1, 1'b0};
// LW R1, R0, 15 ----- R1 = 8'd128
memory[10] = {R0, 5'd15};
memory[11] = {LW, X1, R1};
// BGT R0, R1, 15 ----- if R0 > R1 ---> PC = PC + 15, branch will not be taken at this case
memory[12] = {R1, 5'd15};
memory[13] = {BGT, 1'b0, R0};
// SW R4, R0, 15 ----- Mem(Reg[R0] + 15) = Reg(R4)
memory[14] = {R0, 5'd15};
memory[15] = {SW, X1, R4};
// JMP 120
memory[16] = {8'd120};
memory[17] = {JMP, 4'b0};
```

Figure 3-15: Instructions for Testing in the Instruction Memory

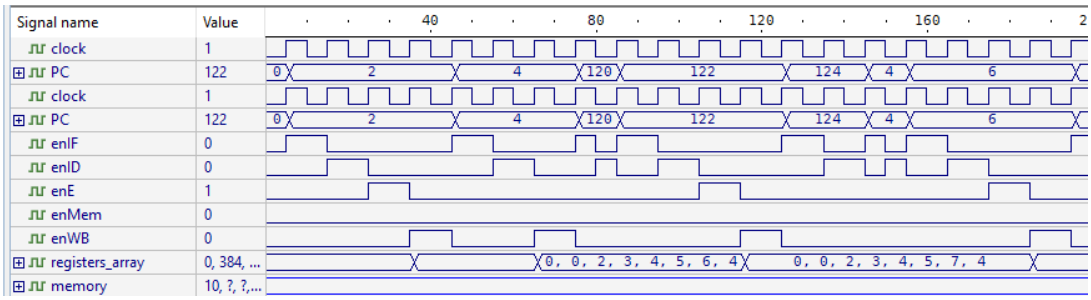

Figure 3-16: Processor Testing Waveform – 1

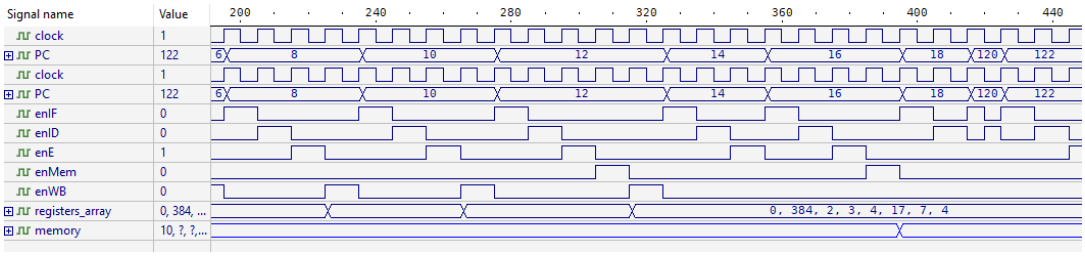

Figure 3-17: Processor Testing Waveform – 2

According to the figures above, all testing instructions have worked correctly, demonstrating the accurate operation of the processor. For further testing, we modified the instructions to focus on branch instructions as follows:

- BGTz R1, X, 4
  - If (Reg[R1] > 0) PC = PC + 2 + 4
  - Else PC = PC + 2
- BLT R1, R4, 2
  - If (Reg[R1] < Reg[R4]) PC = PC + 2 + 2
  - Else PC = PC + 2
- BNE R1, R2, -4
  - If (Reg[R1] != Reg[R2]) PC = PC + 2 – 4
  - Else PC = PC + 2
- BEQz R0, X, 8
  - If (Reg[R0] == 0) PC = PC + 2 + 8
  - Else PC = PC + 2

These additional tests ensure that the branching mechanisms function correctly under various conditions. Instructions are inserted to the instruction memory as shown in the next Figure:

```
// BGTz R1, 3'bX, 4 ---- Taken
memory[0] = {X3, 5'd4};
memory[1] = {BGT, 1'b1, R1};
// BLT R1, R4, 2    ---- Taken
memory[2] = {R4, 5'd6};
memory[3] = {BLT, 1'b0, R1};
// BNE R1, R2, -4   ---- Taken
memory[6] = {R2, 5'b11100};
memory[7] = {BNE, 1'b0, R1};
// ADDI R6 = R2 + 5
memory[20] = {R2, 5'd5};
memory[21] = {ADDI, X1, R6};
// BEQz R0, 3'bX, 8 ---- Taken
memory[12] = {X3, 5'd8};
memory[13] = {BEQ, 1'b1, R0};
```

Figure 3-18: Branch Instructions for Testing

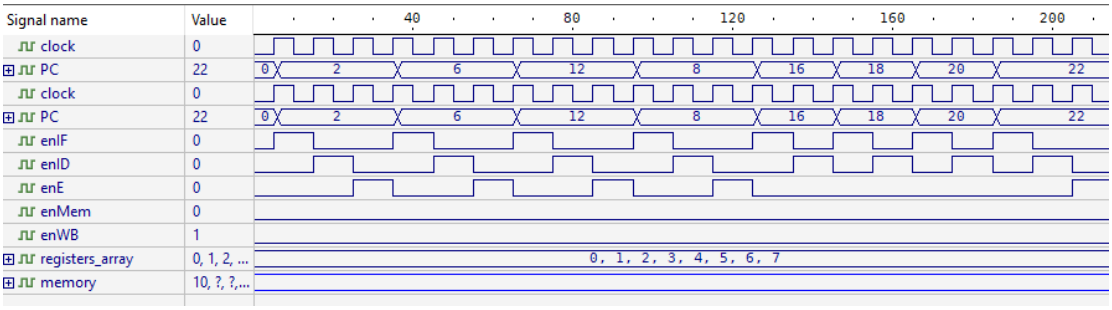

Figure 3-19: Branch Testing Waveform

As the past figure shows, all branch instructions work correctly. Additionally, as seen in Figures 3-16 and 3-17, all instructions operate correctly. The successful execution of these test instructions demonstrates that the processor handles branching and other operations as expected.

This comprehensive testing indicates that our processor is well-designed and effectively implemented as a multicycle processor. The correct operation of all instructions confirms the accuracy and efficiency of our design, ensuring reliable performance in various scenarios. By rigorously testing each instruction, including both arithmetic and branching operations, we have validated the robustness of our processor. The design and implementation process has proven to be successful, providing a solid foundation for future enhancements and applications. The results highlight the processor's capability to manage complex instruction sets, maintain precise control flow, and execute tasks with high reliability.

4. Group Members Contributions

The design of the datapath, control signal tables, equations, and the state diagram was a collaborative effort due to the intricate complexity and detailed requirements involved. This collaborative approach allowed us to leverage each member's strengths, share insights, and ensure the accuracy of the designs, resulting in a well-coordinated and robust final implementation.

For the coding part:

- **Nasri** worked on the data memory, ALU, register file modules and their respective testbenches, and the constants.v file.
- **Abed** developed the control unit code, including the main control, ALU control, and PC control signals along with its testbench.
- **Diaa** handled the clock generator, instruction memory, and PC modules, as well as their testbenches.

The processor code was a collaborative effort due to its complexity and the necessity for meticulous attention to detail. This teamwork ensured that every aspect was thoroughly addressed and accurately implemented.

For testing and simulation:

- Each partner tested the modules he implemented using their testbenches.
- The processor testing was a combined effort due to its complexity and the need for thorough verification across all modules to ensure seamless integration and functionality.

For the report writing:

- **Abed** covered sections from the beginning up to 2.1.4 Store I-Type instructions.
- **Nasri** handled sections from 2.1.5 Branch I-Type instructions to 3.3 Clock Generator Testing.
- **Diaa** worked on sections from 3.4 Instruction Memory Testing to the end.

5. Conclusion

In this project, we successfully designed and implemented a multicycle datapath processor using Verilog. The design process involved a comprehensive analysis of each instruction type, allowing us to identify the specific requirements and components needed for efficient execution. By dividing the instructions into multiple cycles, we optimized the processor's performance, ensuring accurate data handling and control signal coordination. The extensive testing and simulation confirmed the correctness of our design, demonstrating the processor's capability to execute a wide range of instructions efficiently. The control unit, despite its complexity, was meticulously developed to manage the control signals effectively, ensuring seamless operation across all stages of instruction processing. The successful implementation of this processor highlights the importance of detailed planning and verification in digital design projects.

6. References

1] https://www.cise.ufl.edu/~mssz/CompOrg/CDA-proc.html [Accessed on 18 June 2024 17:15]
2] https://en.wikipedia.org/wiki/Datapath [Accessed on 18 June 2024 17:40]
3] https://www.jaroeducation.com/blog/data-path-design-in-computer-architecture/ [Accessed on 18 June 2024 18:37]
4] ENCS4370 Course Slides.

7. Appendix A

```
`include "constants.v"


module pcModule(clock, PC, PCsrc, immediate16, offsetJump, ret_r7, EN);

    input wire clock;

    input wire [1:0] PCsrc;
    input wire signed [15:0] immediate16;
    input wire [11:0] offsetJump;
    input wire signed [15:0] ret_r7;
    input wire EN;

    // PC Output
    output reg [15:0] PC;

    // To store assignments
    wire [15:0] PC1;
    wire [15:0] branchPC;
    wire [15:0] jumpPC;
    wire [15:0] retPC;

    assign PC1 = PC + 16'd2;
    assign branchPC = PC + immediate16;
    assign jumpPC = {PC[15:12], offsetJump};
    initial begin
        PC <= 32'd0;
    end


    always @(clock) begin
        if (clock)begin
        #1ps
        if(EN) begin
            case (PCsrc)
                pcDefault: begin
                    PC = PC1;
                end
                pcBRT:  begin
                    PC = branchPC;
                end
                pcJMP: begin
                    PC = jumpPC;
                end
                pcRET: begin
                    PC = ret_r7;
                end
            endcase
        end
        end
    end
endmodule
```

Figure 6-1: PC Module

```v
`include "constants.v"
module ALU( A, B, Output, zeroFlag, nFlag, ALUop, EN);

    // Select to determine ALU operation
    input wire [1:0] ALUop;

    // ALU Operands
    input wire [15:0] A, B;
    input wire EN;

    output reg  [15:0]  Output; // Output result of ALU
    output reg zeroFlag;
    output reg nFlag;
    // ----------------- LOGIC -----------------

    assign zeroFlag = (0 == Output);
    assign nFlag = (Output[15] == 1); // if 2s complement number is negative, MSB is 1

    always @(*) begin
        #1ps // To wait for ALU source mux to select operands
        case (ALUop)
            ALU_AND:  Output <= A & B;
            ALU_ADD:  Output <= A + B;
            ALU_SUB:  Output <= A - B;
            default: Output <= 0;
        endcase
    end

    initial begin
        if (EN == LOW)begin
            Output = 32'd0;
        end
    end
endmodule
```

Figure 6-2: ALU Module

```v
// Generates clock square wave with 10ns period

module ClockGenerator (clock);
    initial begin
        $display("(%0t) > initializing clock generator ...", $time);
    end

    output reg clock = 0; // starting LOW is important for first instruction fetch

    always #5 begin
        clock = ~clock;
    end

endmodule
```

Figure 6-3: Clock Generator Module

```v
`include "constants.v"

module instructionMemory(clock, address, instruction, opcode);

    input wire clock;
    input wire [15:0] address;
    output reg [15:0] instruction;
    output reg [3:0] opcode;
    reg [7:0] memory [0:255]; // Byte addressable

    always @(clock) begin
        if(clock) begin
            #1ps
            instruction = {memory[address + 1], memory[address]}; // Assign instruction as little endian
            #1ps
            opcode = instruction[15:12];
        end
    end

    initial begin
        opcode = 4'b0000;
        instruction = 0;

        // ANDI R1 = R2 & 5
        memory[0] = {R2, 5'd5};
        memory[1] = {ANDI, X1, R1};
        // CALL subroutine at address 120
        memory[2] = {8'd120};
        memory[3] = {CALL, 4'b0};
        // ANDI R1 = R2 & 5
        memory[4] = {R2, 5'd5};
        memory[5] = {ANDI, X1, R1};
        // ADDI R1 = R2 & 5
        memory[120] = {R2, 5'd5};
        memory[121] = {ADDI, X1, R6};
        // RET return from subroutine
        memory[122] = 8'bx;
        memory[123] = {RET, 4'b0};
        // ADDI R1 = R2 & 5
        memory[6] = {R2, 5'd5};
        memory[7] = {ADDI, X1, R6};
    end

endmodule
```

Figure 6-4: Instruction Memory Module

```v
`include "constants.v"
module dataMemory(clock, ALUaddress, BaAddress, regData , svData , dataOut, ENW, ENR , address_src, data_in_src, EN, memory, OpCode, mode);

    input wire clock;
    input wire ENW;
    input wire ENR, EN, mode;
    input wire address_src, data_in_src;
    input wire [3:0] OpCode;
    input wire [15:0] ALUaddress;
    input wire [15:0] BaAddress;
    input wire [15:0] regData;
    input wire [15:0] svData;
    output reg [15:0] dataOut;
    output reg [7:0] memory [0:255];
    reg [15:0] AddressBus, dataIn;

    always  @(posedge EN) begin  // Determine where to fetch address from
        if (EN) begin
            if (address_src == Addr_ALU) begin
                AddressBus <= ALUaddress;
            end

           else if (address_src == Addr_Ba) begin
                AddressBus <= BaAddress;
            end
        end
    end

    always  @(posedge EN) begin  // Determine where to store data from
        if (EN) begin
            if (data_in_src == data_in_Bb) begin
                dataIn <= regData;
            end

            else if (data_in_src == data_in_SV) begin
                dataIn <= svData;
            end
        end
    end
```

Figure 6-5: Data Memory Module – 1

```
    always @(clock) begin
        if(clock) begin
     if (EN) begin
         if (ENR && OpCode == LB) begin
             if (mode == 1'b1) begin
                 dataOut <= {{8{memory[AddressBus][7]}}, memory[AddressBus]}; // Sign extend byte
             end else if (mode == 1'b0) begin
                 dataOut <= {{8{1'b0}}, memory[AddressBus]}; // Zero extend byte
             end
         end else if (ENR && OpCode != LB) begin
             // Read operation
             dataOut <= {memory[AddressBus + 1], memory[AddressBus]}; // Read 16-bit data in little-endian format
         end else if (ENW) begin
             // Write operation
             memory[AddressBus + 1] <= dataIn[15:8]; // Write upper 8 bits to next cell
             memory[AddressBus] <= dataIn[7:0]; // Write lower 8 bits to first cell
         end
     end
     end
endmodule

    initial begin
        // Store initial values in the memory for testing purposess
        memory[0]   = 8'd10;
        memory[7]   = 8'd00;
        memory[8] =   8'd00;
        memory[15]  = 8'd128;
        memory[16]  = 8'd1;
        memory[40]  = 8'd6;
        memory[60]  = 8'd5;
        memory[223] = 8'd15;
    end

endmodule
```

Figure 6-6: Data Memory Module – 2

```
`include "constants.v"
module dataMemory_tb;

    // Inputs
    wire clock;
    clockGenerator clock_generator(clock);
    reg ENW;
    reg ENR;
    reg EN;
    reg address_src;
    reg data_in_src;
    reg [15:0] ALUaddress;
    reg [15:0] BaAddress;
    reg [15:0] regData;
    reg [15:0] svData;
    reg [3:0] OpCode;
    reg mode;

    // Outputs
    reg [15:0] dataOut;

    // Internal signals
    reg [7:0] memory [0:255];

    // Instantiate the unit under test (UUT)
    dataMemory UUT (clock, ALUaddress, BaAddress, regData, svData, dataOut, ENW, ENR, address_src, data_in_src, EN, memory, OpCode, mode);

    // Initialize inputs
    initial begin
        #0
        EN <= 0;

        // Read Operation test
        #1
        ENW <= 0;
        ENR <= 1;
        EN <= 1;
        address_src <= 0;
        OpCode <= LB;
        mode <= LOW;
        ALUaddress <= 16'h0000;
        #5
        EN <= 0;
        #5
```

Figure 6-7: Data Memory Testbench – 1

```
45
46          // Write operation test
47          ENW <= 1;
48          ENR <= 0;
49          EN <= 1;
50          address_src <= 1;
51          data_in_src <= 0;
52          OpCode = SW;
53          mode = 0;
54          BaAddress <= 16'h0000;
55          regData <= 16'hABCD;
56          #5
57          EN <= 0;
58          #5
59
60          // Read operation test
61          ENW <= 0;
62          ENR <= 1;
63          EN <= 1;
64          address_src <= 1;
65          BaAddress <= 16'd15;
66          OpCode <= LW;
67          mode <= HIGH;
68          #10;
69
70          $finish;
71      end
72
73  endmodule
```

Figure 6-8: Data Memory Testbench - 2

```
1  `include "constants.v"
2  module registerFile(clock, Ra, Rb, Rw, EnW, Bw, Ba, Bb, EN, registers_array);
3
4      input wire clock;
5      input wire EnW, EN;
6      input wire [2:0] Ra, Rb, Rw;
7      output reg [15:0] Ba, Bb;
8      input wire [15:0] Bw;
9      output reg [15:0] registers_array [0:7];
10
11     always @(posedge EN) begin
12         #1ps
13
14         if(EN && !EnW) begin
15             Ba <= registers_array[Ra];
16             Bb <= registers_array[Rb];
17         end
18
19         if ( Rw != 3'b0 && EN && EnW) begin
20             registers_array[Rw] = Bw;
21         end
22
23     end
24
25     initial begin
26         // Initialize registers to be zeros
27         registers_array[0] <= 16'd0000;
28         registers_array[1] <= 16'd0000;
29         registers_array[2] <= 16'd0000;
30         registers_array[3] <= 16'd3;
31         registers_array[4] <= 16'd0000;
32         registers_array[5] <= 16'd0000;
33         registers_array[6] <= 16'd0000;
34         registers_array[7] <= 16'd0000;
35     end
36 endmodule
```

Figure 6-9: Register File Module

```
module registerFile_tb;

    wire clock;
    ClockGenerator clock_generator(clock);

    reg [2:0] RA, RB, RW;
    reg EnW, EN;
    reg [15:0] BusW;
    wire [15:0] BusA, BusB;
    wire [15:0] registers_array [0:7];
    registerFile regFile(clock, RA, RB, RW, EnW, BusW, BusA, BusB, EN, registers_array);

    initial begin

        #0
        EN <= 0;
        #1
        RW <= 3'd1; // Write to register 1
        BusW <= 16'd16; // write value 16
        EnW <= 1; // Enable write
        EN <= 1;
        #5
        EN <= 0;

        #5
        RW <= 3'd2; // Write to register 2
        BusW <= 16'd32; // Write value 32
        EnW <= 1; // Enable write
        EN <=1;
        #5
        EN <= 0;

        #5
        RA <= 3'd1;
        RB <= 3'd2;
        EnW <= 0;
        EN <= 1;
        #5
        EN <= 0;
```

Figure 6-10: Register File Testbench – 1

```
        #5
        RW <= 3'd2; // Write to register 2
        BusW <= 16'd64; // Write value 64
        EnW <= 1; // Enable write
        EN <= 1;
        #5
        EN <= 0;

        #5
        RA <= 3'd3;
        RB <= 3'd0;
        EnW <= 0;
        EN <= 1;
        #10

        #10
        $finish;
    end

endmodule
```

Figure 6-11: Register File Testbench – 2