COE 301 – Computer Organization

Term 222 – Spring 2023

Project: Pipelined Processor Design

Project Report

Team #29

Team Members:

NAME	ID
BASIL ALASHQAR	202045700
MUATH ALABEDI	202014940
SAMI ALSHURAIM	202042200

Table of Contents

- ## Objectives
  - **Gain proficiency in the process of designing a pipelined CPU.**
  - **Learn about the different kinds of pipeline hazards.**
  - **Apply Forwarding technique to manage data hazards in our design.**
  - **Validate the proper functioning of the pipelined CPU that we have created.**
  - **Implement pipeline stall as a solution to address RAW hazard that may occur after Load instruction.**
  - **Incorporate pipeline stall in our design to handle control hazards.**
- ## Introduction
  - This report presents the design and implementation of a pipelined 16-bit RISC processor with 16-bit instructions. The processor includes 7 general-purpose 16-bit registers and supports three instruction formats: R-type, I-type, and J-type. We used the Logisim simulator to model and test the processor and worked as a team to achieve our goals. In this report, we showcase our implementation approach. We also provide simulation results and discuss the performance and limitations of the processor.

- ## Design and Implementation
  - ### Single Cycle CPU
    - #### 2.1 Design Choices and Notable Features

      Our team's first step in creating our pipelined CPU was to design a single-cycle CPU. This CPU was comprised of several critical components, including a program counter (PC), an instruction memory, a register file, a main control unit, and an arithmetic logic unit (ALU). The PC served as a register that tracked the memory address of the next instruction to be fetched. The instruction memory is where the CPU obtained instructions from, while the register file stored data in 8 16-bit registers. The main control unit generated control signals that governed the flow of data between the CPU components. The ALU performed arithmetic and logical operations on data.
    - #### 2.2 Drawings of Components and Datapath

      To provide a visual representation of our pipelined CPU design, we created detailed drawings of the CPU's components and Datapath. The drawings include the x`registers, memory, ALU, multiplexers, and other essential components. The Datapath diagram shows the flow of data through the pipeline stages and the control signals that drive each stage.

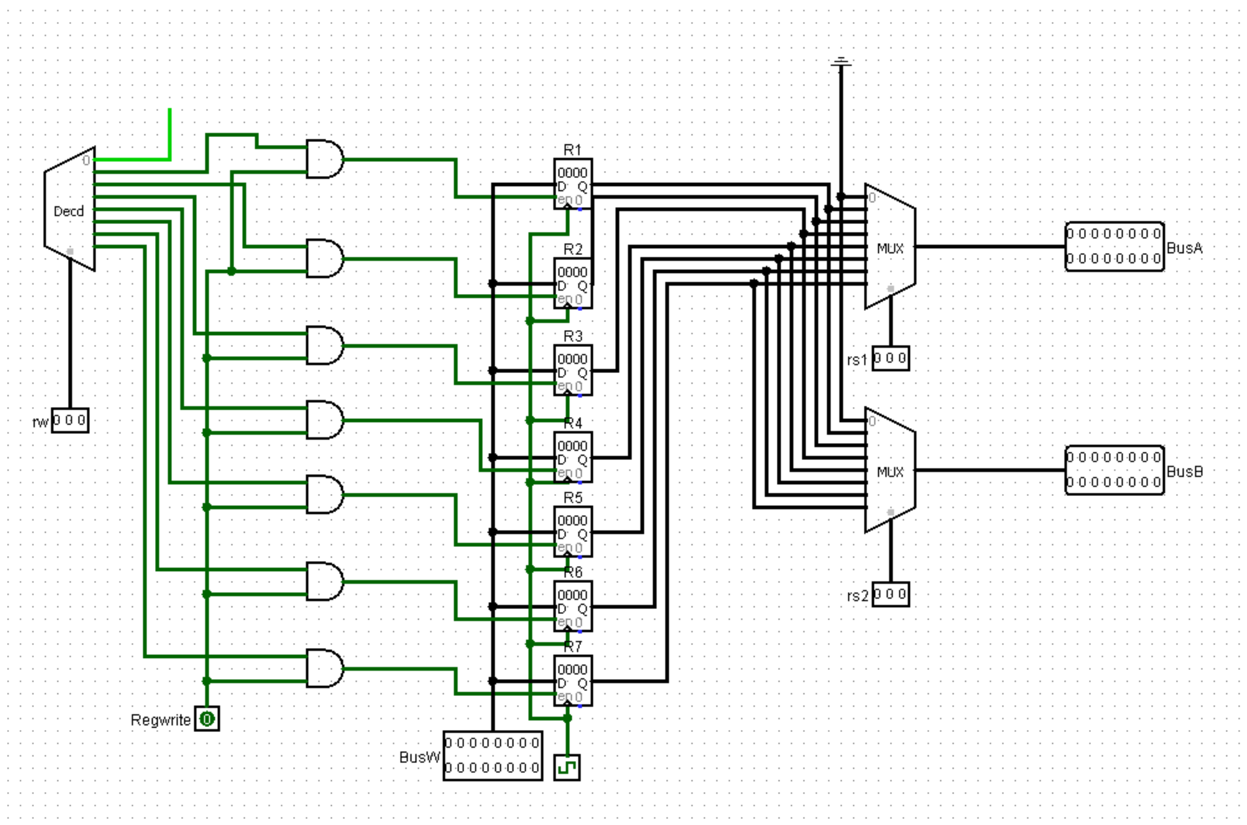


Figure 1. Register File

register file that stores data temporarily during instruction execution and consists of multiple registers that can be accessed in the CPU.
In terms of the design of the register file, we used decoder, and gates, registers, and multiplexers.

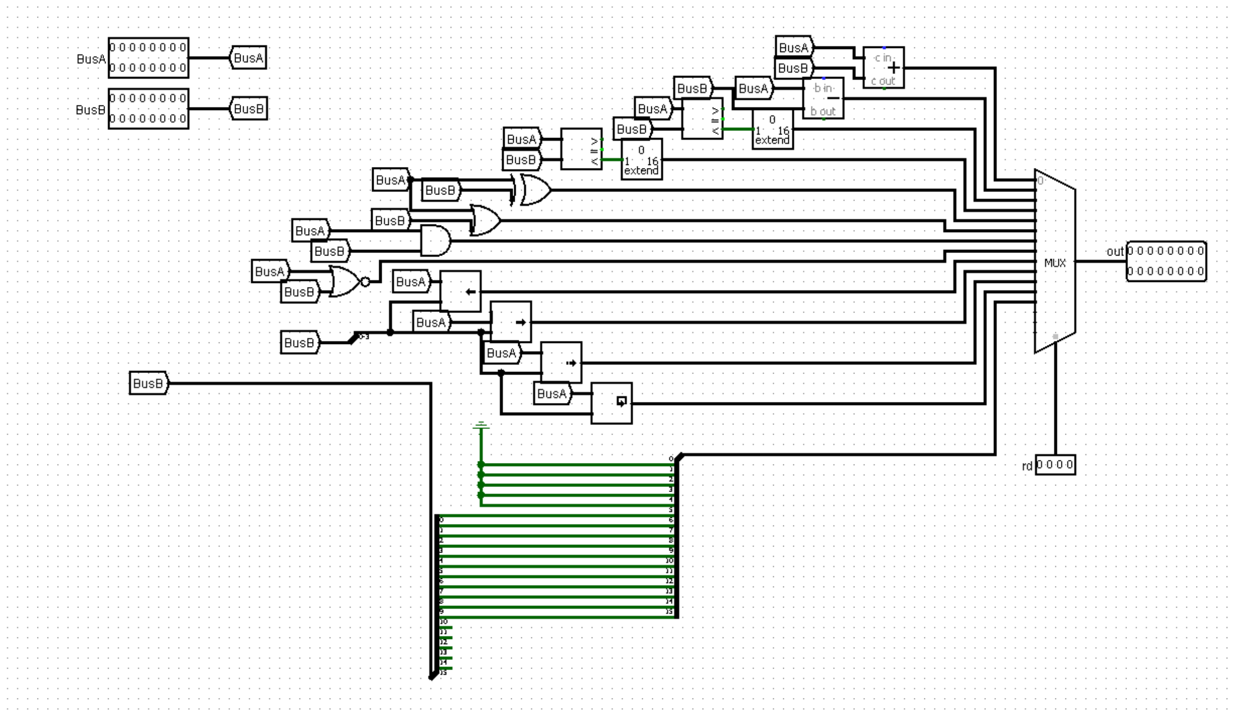


Figure 2. Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) performs arithmetic and logic operations on data stored in registers, such as addition, subtraction, AND, OR, and XOR, with the results stored back in registers. The ALU is a fundamental component of the CPU responsible for executing arithmetic and logic operations on data.
In terms of the design of the ALU we used multiplexer, and different logic gates(and,or,xor,nor) also we used shift registers and sign extenders components.

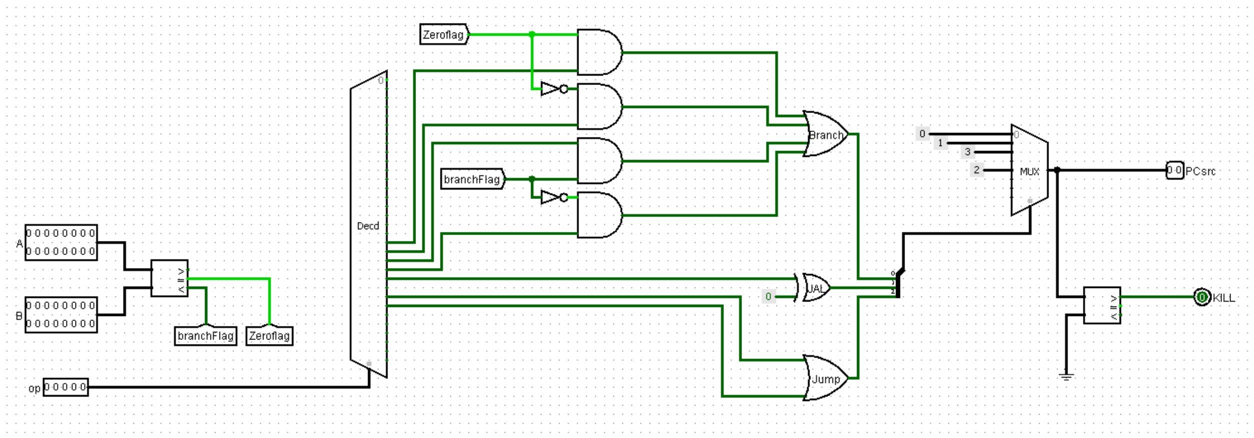


Figure 3. PC Control

The program counter (PC) control with branch flag and zero flag is used to execute conditional branch instructions such as BEQ (branch if equal), BNE (branch if not equal), BGE (branch if greater than or equal to), and BLT (branch if less than). The jump and link register (jalr) and jump and link (jal) instructions are used for unconditional branching, with jalr using a register as the target address and jal storing the return address in a register.

In terms of the design of the PC Control, we used a branch flag, zero flag, a decoder, multiple AND, OR gates, comparators and finally a multiplexer.

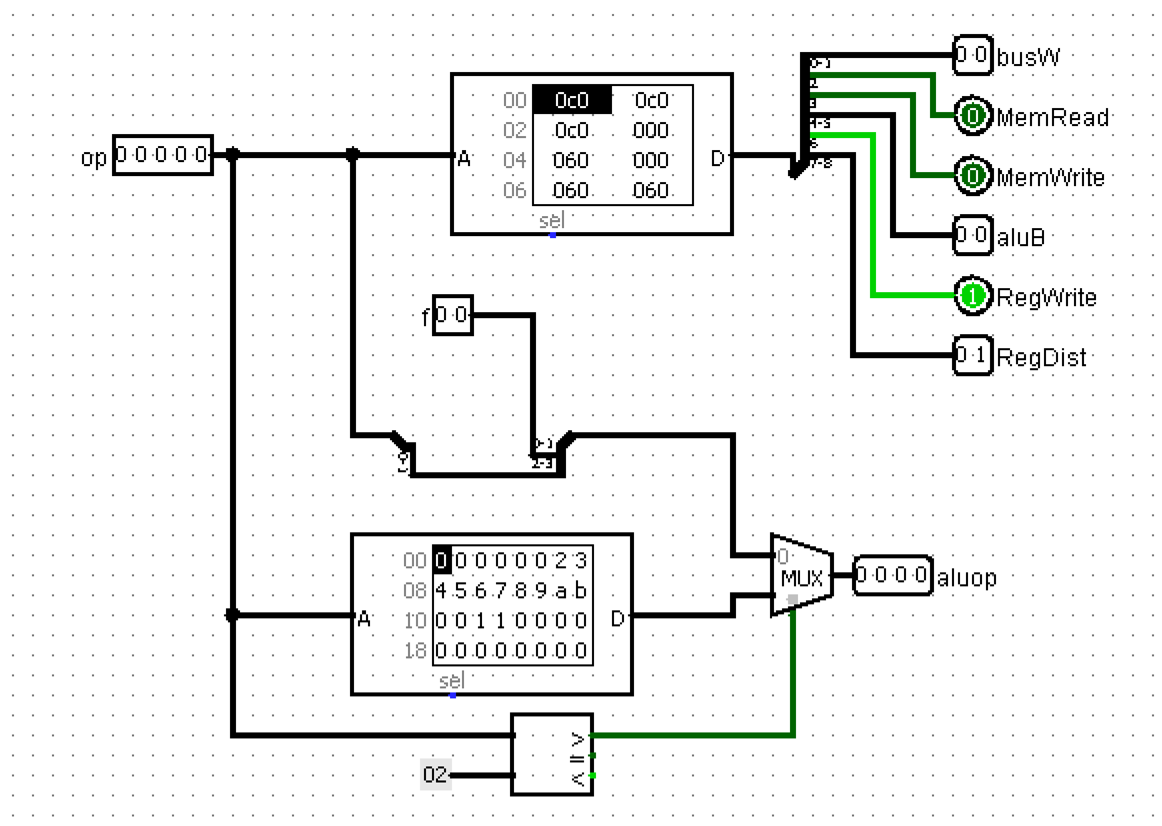


Figure 4. Main Control Unit

Main Control Unit in a CPU, showing how the instruction opcode is decoded to generate control signals that govern the operation of the CPU's Datapath. The main control unit is responsible for generating control signals, which is then used to carry on executing the instruction fetched.

- **Pipelined CPU**

  After creating the single-cycle CPU, we transformed the design to construct a pipelined CPU. The primary objective of pipelining is to optimize CPU performance by allowing several instructions to be executed simultaneously. In our pipelined CPU design, we included five stages: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and write back (WB). Each stage of the pipeline processed a distinct instruction simultaneously.

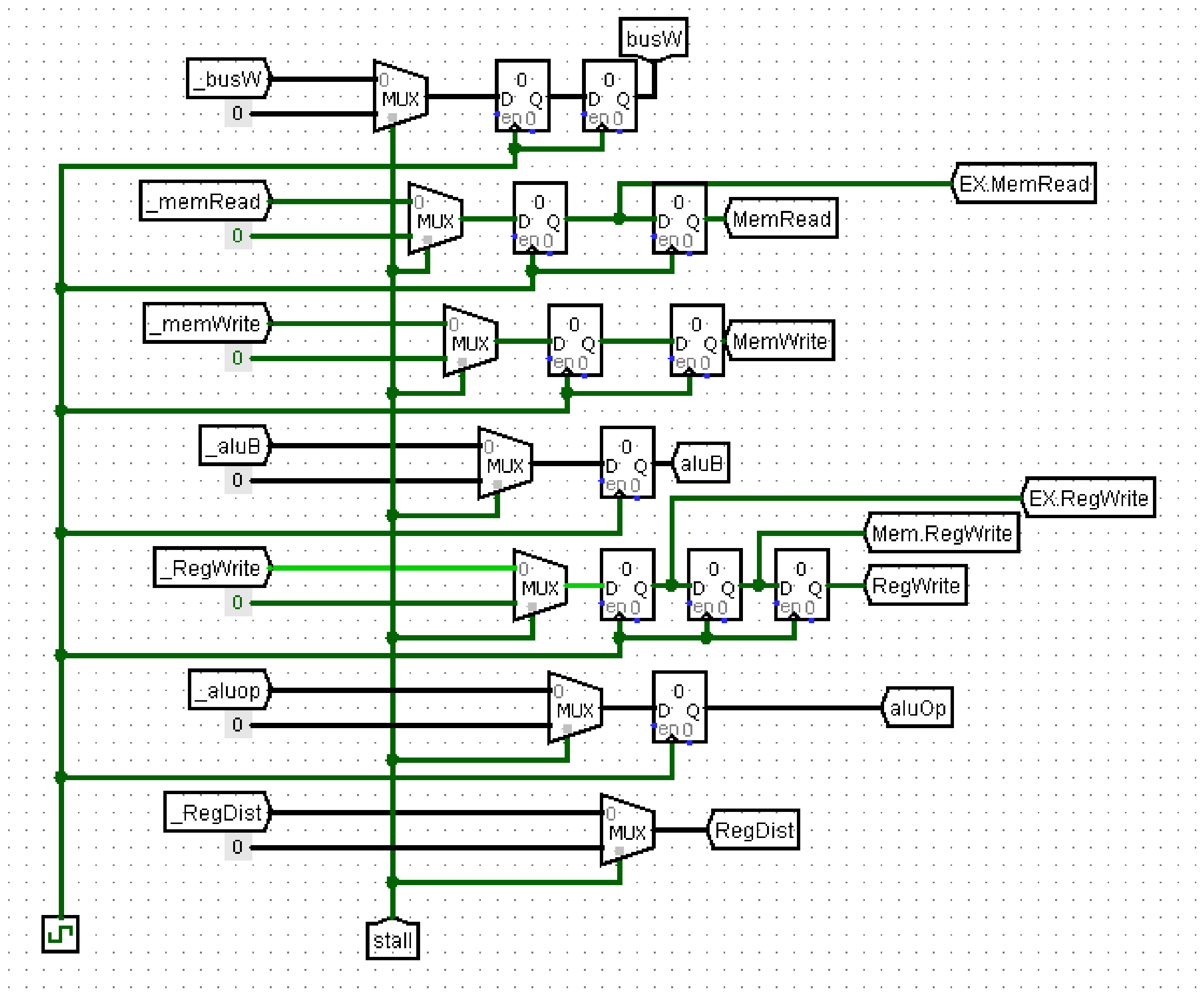


Figure 5. intermediate registers.

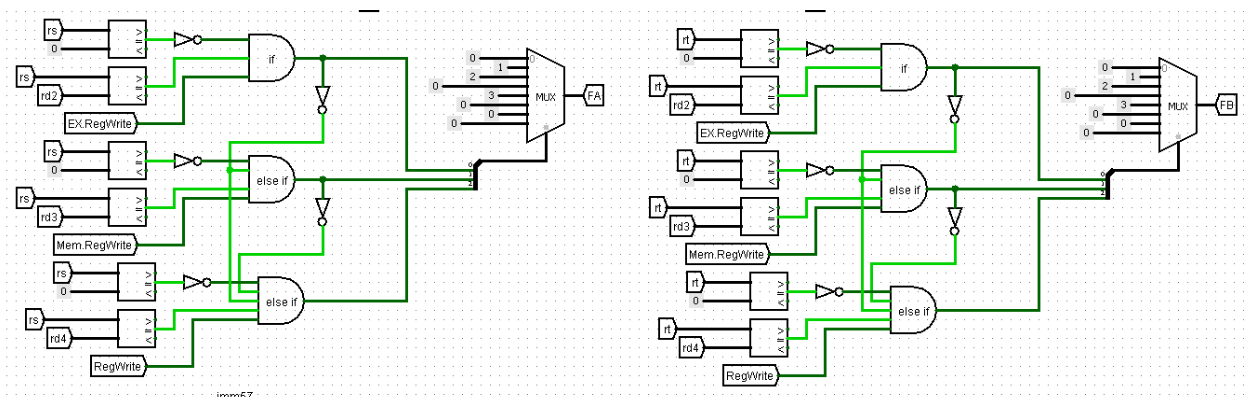


Figure 6. Forwarding Conditions

Forwarding Conditions defines the conditions for data forwarding in a pipelined CPU with data hazards. The conditions check if the required register values are available in the EX, MEM, or WB stages, and set the ForwardA or ForwardB signal accordingly, with values 1, 2, or 3 indicating forwarding from the corresponding stage, and 0 indicating that the operand value must be obtained from the register file.

In terms of the design of the Forwarding Conditions we used comparators, AND gates, inverters and multiplexers.

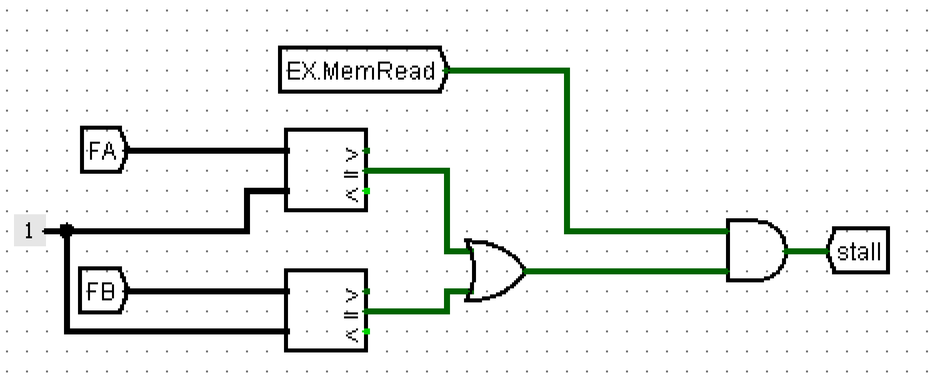


Figure 7. Stall

Stall shows a technique used to prevent data hazards in a pipelined CPU. When a data hazard is detected, a bubble, or stall, is inserted into the pipeline to delay the execution of the dependent instruction until the required data is available, ensuring correct execution of the program.
In terms of the design of the stall, we used comparators, or gate and AND gate.

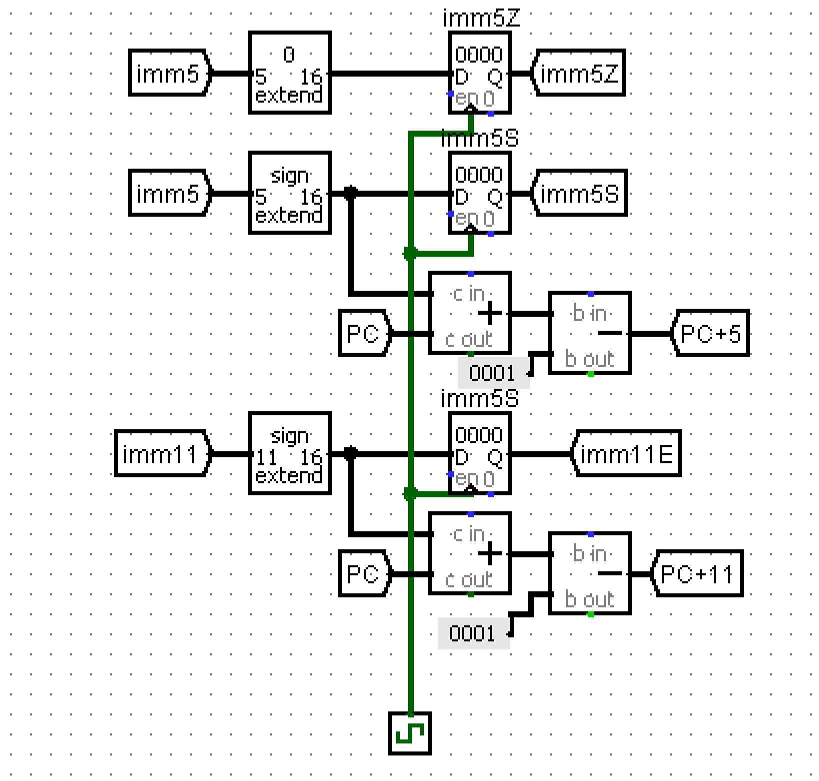


Figure 8. Immediate extensions and pc jump and branch calculations.

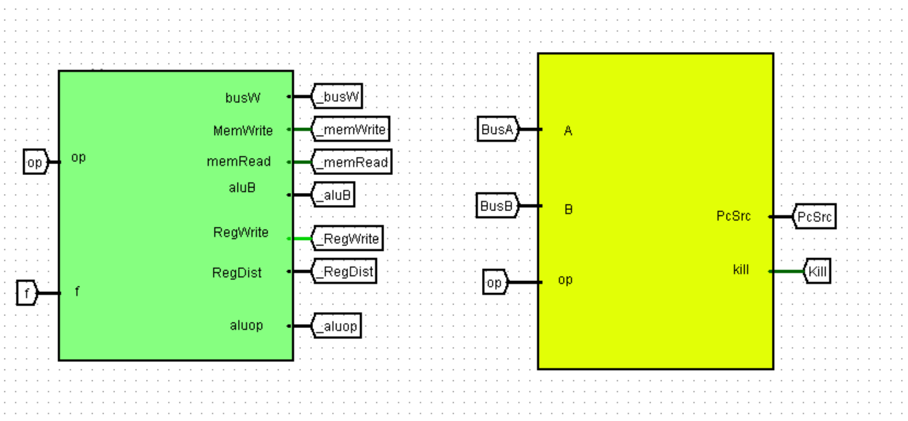


Figure 9. PC & Main Control connected Blocks.

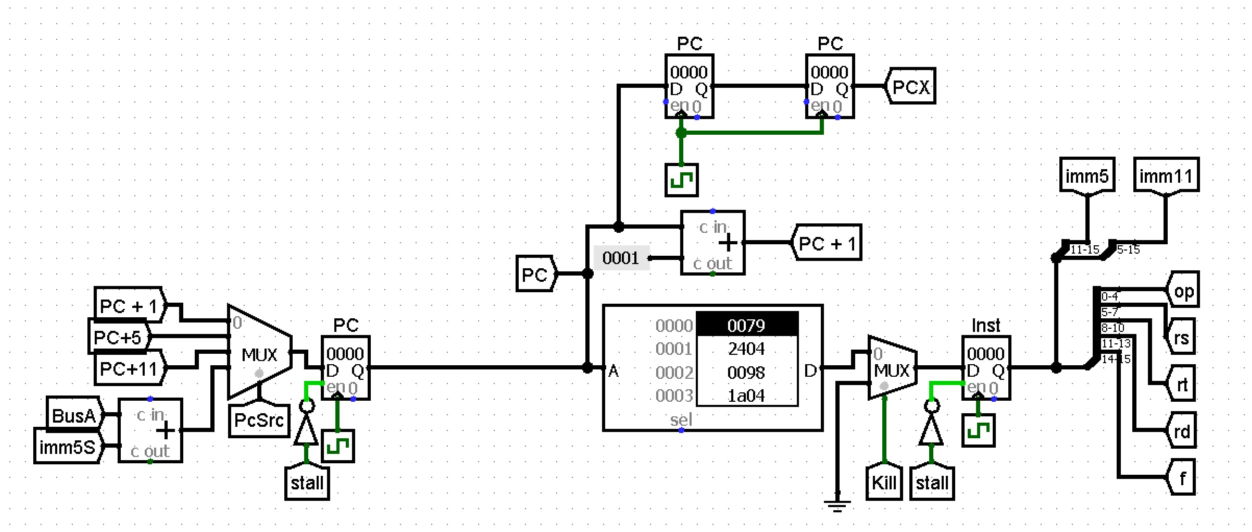


Figure 10. Datapath part1

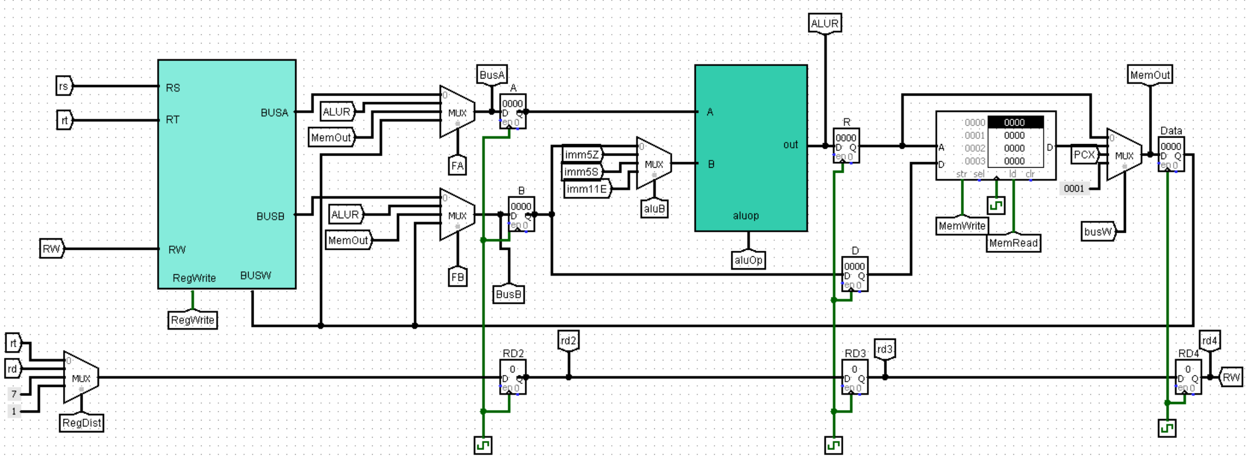


Figure 11. Datapath part2

- ## 2.3 Description of Control Logic and Signals
    - The control logic in a CPU is responsible for generating control signals that manage the flow of data between the different components of the CPU. In our pipelined CPU design, the control signals are generated by the main control unit and the forwarding logic unit.

      The main control unit generates control signals based on the instruction being executed. The control signals are used to enable or disable the different components of the CPU during each stage of the pipeline.

The Pipelined CPU is divided into 5 stages:
1 - Instruction Fetch (IF)
2 - Instruction Decode (ID)
3 - Execute (EX)
4 - Memory Access (MEM)
5 - Write Back (WB)

Program starts with pc register at 0000, which is then given to instruction memory to fetch the first instruction. this is done in the IF stage.

In the next clock pulse, the instruction comes out of the memory and is decoded into separate signals that specify the operation code, registers needed, immediate values, and other important signals. this is the ID stage.

Clock pulse after that is the Execution stage. It consists of the ALU, where we have two operands. One of them is the source register from the ID stage. The other is decided by a multiplexer with select signal aluB that either gives a value of a register or an immediate value.

After that we have the MEM stage where we access the data memory to either store or get a value. Whether we use this stage or not is based on the values of signals memRead and memWrite decoded in the ID stage.

Finally, we have the WB stage. we write the result obtained by the ALU or MEM back to the register file.

One notable fact to mention about the difference between single cycle and pipelined CPU design is that in the single cycle design, all of these stages are executed in one clock cycle. As opposed to the pipelined design, which takes at least 5 clock cycles to complete a single instruction.

Due to this difference, certain measures have to be taken when we want to execute two instructions where the second depends on the result of the first. Since the write back step is the last one and the result of an operation like add is finished after the execute stage, we can forward the result right after it comes out of the ALU to be used by the second instruction in its execute stage.

Additionally, a special case of the problem stated before is when we have a lw instruction with an instruction that depends on the value of its result right after it. We cant forward the result from the ALU because a lw instruction only gets its result after the MEM stage. Therefore, we need to delay the execution of the second instruction by one cycle. This is called stalling. Doing so allows the result of lw instruction to be ready right before the second instruction starts the execution stage.

- ## 2.4 Description of Forwarding Logic and Control Hazard Handling
  - Forwarding logic is a technique used in pipelined CPUs to handle RAW (Read After Write) data hazards by forwarding the data that is produced by an instruction in a later pipeline stage to an instruction that requires it in an earlier stage. This avoids pipeline stalls and improves performance.

    In the pipeline design with data forwarding, the data forwarding logic consists of two multiplexers at the inputs of the A and B registers, which are controlled by two signals, ForwardA and ForwardB. These signals determine whether the operands for the current instruction come from the register file or from a forwarded value from a previous instruction in a later pipeline stage.

    To handle RAW hazards, the forwarding logic checks whether the operands of the current instruction require a value that is not yet available in the register file. In that case, the forwarding logic checks the previous instructions in the pipeline and forwards the required value to the appropriate input of the A or

B register. The forwarding logic can forward data from three different stages: the ALU output stage, the MEM stage, or the WB stage.

The forwarding logic checks for the following conditions to determine if forwarding is necessary:

If the Rs field of the current instruction matches the destination register of the previous instruction in the EX stage and that instruction writes back to the register file, ForwardA is set to 1 to forward the value from the ALU output stage.

If the Rs field of the current instruction matches the destination register of the previous instruction in the MEM stage and that instruction writes back to the register file, ForwardA is set to 2 to forward the value from the MEM stage.

If the Rs field of the current instruction matches the destination register of the previous instruction in the WB stage and that instruction writes back to the register file, ForwardA is set to 3 to forward the value from the WB stage.

The same conditions apply to the Rt field of the current instruction for the ForwardB signal.

To handle control hazards, we have two things to keep in mind. For jump and branch instructions we always calculate the pc for taking the jump and branch instructions. We always take jump instruction therefore no further comparisons are needed, and the pc is set to pc+imm11 -1, the minus one serves as a filler for the wasted cycle in ID stage, and then the instruction that was fetched after the jump is flushed. The same is done with branch instructions except that we compare ra and rb first. Second, for JAL instruction we need to write back the return address then jump, so we need to wait for two cycles before jumping, which is the use for NPC1 and NPC2.

- **Simulation and Testing**

  The following is a brief description of each program's function and what it tests exactly. For more information about the function of any program, please check the readme of the specific program.

  - **3.1 Test program 1: Count Ones**

    This program takes an integer, checks if least significant bit is 1, increments a counter if so, then divides it by 2 until it becomes 0. It demonstrates that instructions like ori, addi, and some branch instructions work properly.

  - **3.2 Test program 2: Short**

    This program loops 15 times while storing the value of the current iteration in the first address in the memory.
    It demonstrates that looping and handling data hazards with forwarding are working.

  - **3.3 Test program 3: Bubble Sort**

    This program continues to loop through the array, swapping every two adjacent elements that are decreasing, until the array is sorted in increasing order.
    It shows that hazard detection works and that stalling and forwarding also work.

  - **3.4 Test program 4: All Instructions**

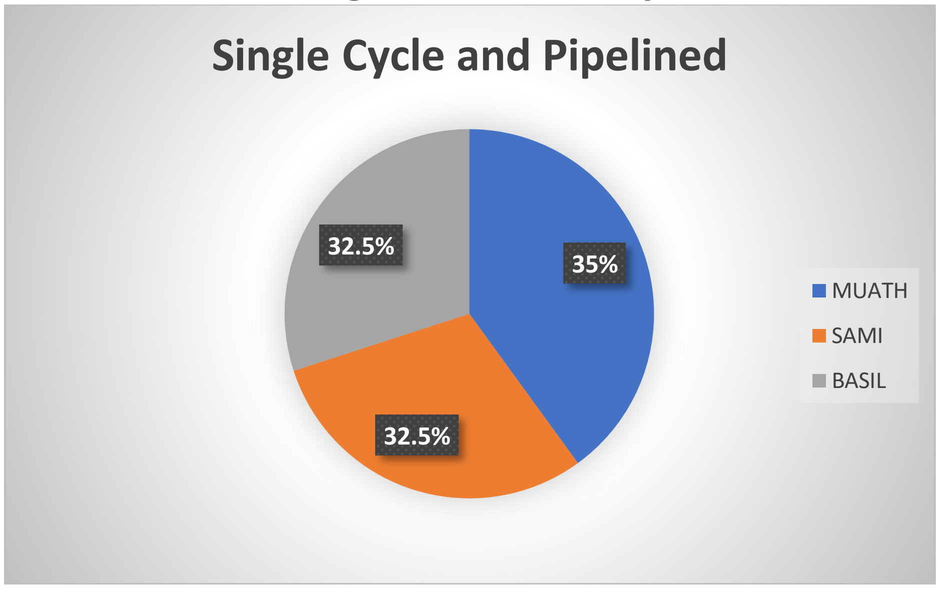
    This program does not do any specific task but rather contains all instructions in an order that ensures that each instruction's validity will be shown. Also, tests hazard handling using stall and forwarding.

- ## Teamwork

4.1 Coordination of Work Among Team Members

NAME	Worked On	
	Single Cycle	Pipelined
MUATH ALABEEDI	40%	30%
BASIL ALASHQAR	30%	35%
SAMI ALSHURAIM	30%	35%

4.2 Chart Showing Work Done by Each Team Member



- ## Conclusion

  - In conclusion, the pipeline project was a challenging yet rewarding experience for our team. We were able to design and implement a fully functional pipelined CPU. Starting with creating a single cycle CPU then converting it to a pipelined CPU. Throughout the project, we learned about the inner workings of pipelining, the different stages involved, and the various hazards that can arise. We also gained experience in implementing forwarding and branch prediction techniques to mitigate those hazards.