IITB-RISC-PIPELINE

EE309 - 2023
PROJECT-REPORT

Ojas P. Karanjkar-210070040
Omkar M. Nitsure-210070057
Sanket Kothawade-210070044
Kushal C. Gajbe-210070048

Contents

1 Introduction

The project aims to design and develop a Central Processing Unit (CPU) using Reduced Instruction Set Computing (RISC) and a 6-stage Pipeline Architecture. The project involves the design and implementation of various components of the CPU, such as the Instruction Decoder, Control Unit, Arithmetic and Logic Unit (ALU), and Memory Management Unit (MMU), among others. The use of RISC architecture simplifies the design of the CPU, making it easier to optimize its performance and efficiency.

The 6-stage Pipeline Architecture is an essential component of the project, which allows the CPU to execute multiple instructions simultaneously, thus enhancing its processing speed. The IITB-RISC-23 is a 16-bit computer system with 8 registers. The pipeline stages include Instruction Fetch, Instruction Decode, Register Read, Execute, Memory Access, and Write Back.

2 LITERATURE REVIEW

2.1 RISC

The success of RISC computer architecture, which uses a smaller set of instructions to process instructions more quickly, can be attributed to this feature. Since the 1980s, when the RISC architecture was originally put forth, it has been widely used in the development of microprocessors, embedded systems, and other computing hardware.

Using a limited set of quick-to-execute basic instructions is one of the distinguishing characteristics of RISC architecture. In contrast to complicated Instruction Set Computing (CISC) designs, which employ a significant amount of complicated instructions, this enables quicker execution times and enhanced performance. Additionally, RISC design minimizes the hardware required for decoding and executing instructions, leading to smaller, more power-efficient CPUs.

2.2 PIPELINE

The use of pipeline architecture in CPU design enhances performance by enabling the simultaneous execution of many instructions. The pipeline is a collection of steps through which instructions move while being executed, and each stage gives the instruction a specific task to be completed.
The fundamental concept behind a pipeline design is to divide an instruction's execution into smaller phases, each of which is carried out by a distinct CPU component. Multiple instructions are in various stages of execution at once thanks to the pipeline architecture. The execution of multiple instructions can overlap when one instruction enters a stage as it is finishing up, and another can exit that

stage. When compared to running instructions in sequential order, this overlap leads to quicker execution times and better performance.
However, the pipeline architecture can also introduce some challenges, such as hazards that can occur when instructions interfere with each other's execution. For example, a data hazard can occur when an instruction needs to wait for the result of a previous instruction before it can proceed. Similarly, a control hazard can occur when an instruction needs to wait for the outcome of a conditional branch before it can proceed.

To address these challenges, various techniques have been developed to optimize the pipeline architecture, such as forwarding and stalling. Forwarding involves passing the result of instruction directly to the next instruction that needs it while stalling involves pausing the pipeline to wait for the data or control dependencies to be resolved.

3 Methodology

The IITB-RISC-23 is a 16-bit computer system with 8 registers. It should follow the standard 6-stage pipelines.

- Instruction fetch,
- instruction decode,
- register read,
- execute,
- memory access,
- and write back.

3.1 Instruction Fetch

The first stage in the pipeline architecture of the IITB CPU project is the Instruction Fetch stage. In this stage, the CPU fetches the next instruction from memory and prepares it for execution.
The CPU accesses the memory location indicated by the program counter and gets the instruction stored there during the instruction fetch stage. The instruction is subsequently placed in the instruction register (IR), a temporary register that is accessible by the pipeline's further steps.
Now, the important point to note here is that we need to increment the PC in this same stage so that the next stage(s) never remains empty/idle.

3.2 Instruction Decode

The second stage of the 6-stage pipeline architecture in the IITB CPU project plays a critical role in the overall functioning of the processor. As mentioned earlier, this stage is responsible for decoding the instruction fetched in the previous stage and preparing it for the subsequent stages of the pipeline.

After the instruction has been fetched and stored in the instruction register, it travels through the pipeline registers and enters the controller. The controller is often referred to as the "brain" of the processor, as it is responsible for coordinating the activities of all the other components.

The first task of the controller in the instruction decode stage is to recognize the type of instruction that has been received. This is important because different types of instructions require different operations to be performed, and different operands to be used. For example, an arithmetic instruction such as add or subtract will require different operations and operands than a jump instruction.
Using its knowledge of the instruction type, the controller then sends control signals to all the other components of the processor. It is important that these control signals are sent through the pipeline registers, to ensure that each stage of the pipeline has the appropriate controls for the current instruction being processed.

Use of PRIORITY ENCODER(PEN):-
Load Multiple and Store Multiple instructions had to be dealt with separately from others. In LM/SM we made used of the priority encoder. Priority encoder scans through the the immediate 8 bits to identify the register to be modified in the register file. In LM/SM instructions, the pipeline register 1 needed to be stalled and the contents of the PC were reloaded in the PC again. At the start of LM/SM instruction, we ensured that all the components used in LM/SM instruction are enabled in the first iteration. In the subsequent instructions, we disabled pipeline register 1, loaded the PC with its previous value and made changes to mux 14, 15, 16 and 17. These control bits were modified in the Priority Encoder itself and the control word was modified subsequently.The end inst bit marks the end of Load Multiple and Store Multiple instructions and we load the next PC in the program counter and enable all the pipeline registers.

3.3 Register Read

The Register Read stage is the third step of the IITB CPU project's 6-stage pipeline architecture. The operands for the current instruction are read from the register file at this step, which comes after the Instruction Decode stage.
The operands' register addresses are taken from the instruction at this point and given to the register file. A register file is a group of registers that may be read from or written to and store data values. It is possible to access the contents of each register using its specific address.

The register file reads the data from the designated registers after obtaining the register addresses and passes it to the pipeline's next step. This information can be used as the address for a memory access operation, as an input to arithmetic or logical operations, or both.

The resolution of register dependencies between instructions is one of the main difficulties in the Register Read stage. A risk may arise if two instructions use the same register as an operand, but only one of the instructions has written its result to the register file.

3.4 Execute

The Execution stage is the fourth stage in the 6-stage pipeline architecture of the IITB CPU project. This stage is responsible for performing the actual calculations and operations specified by the instruction.

To accomplish this, we have implemented several components in this stage, including an Arithmetic Logic Unit (ALU) and two adders. The ALU is responsible for performing the versatile operations required by the instructions. Adder 1(numbered as 2 in the datapath), on the other hand, is solely used to calculate addresses. It can calculate various types of addresses, such as PC+imm2, Ra+imm2, PC+imm, and more.

Adder 2(numbered as component 4) is used to aid Adder 1. Its output is fed into Adder 1 as an input. This arrangement allows for more complex calculations that require the addition of multiple values. Additionally, there are sign extenders for the immediate bits that are of the type J and I.

There are only two main outputs from this stage that are sent to the next stage: the output of Adder 1 and the output of the ALU. Each input port to the Adder and ALU has muxes to it. These muxes can select from different sources of input, such as the output of the previous stage or a constant value. The controls for these muxes come from the controller, as we mentioned earlier.

3.5 Memory Access

The IITB CPU project's 6-stage pipeline architecture includes the Memory Access stage as the fifth level. This phase is in charge of going to memory and getting the information the instruction needs.

In the Memory Access stage, the address calculated in the previous stage using Adder 1 is used to determine the memory location or address needed for the operation. This address is then sent to the memory unit to retrieve the data stored at that location.

Once the data has been obtained, it is sent to the next pipeline register where it will be temporarily kept until the Write Back stage.

3.6 Write Back

The sixth stage of the IITB CPU project's six-stage pipeline architecture is called Write Back. The instruction results are written back to the register file during this phase. The instruction's result, which had been momentarily stored in the pipeline register, is now written back to the destination register that was indicated in the instruction. The pipeline is prepared to receive the next command after the data has been written back to the register file.

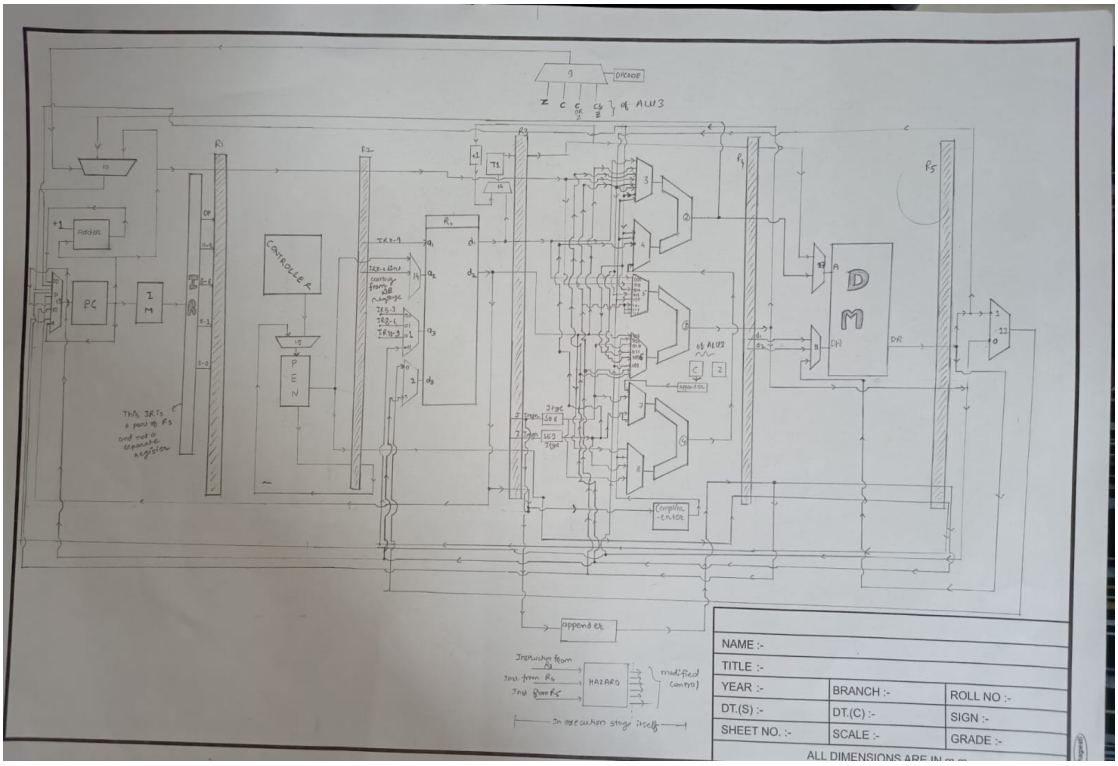


Figure 1: CPU

4 DEALING WITH HAZARDS

The issue of data dependencies in a CPU architecture is a common problem that can lead to incorrect results or even system failure. In our project, we have encountered data dependencies in the form of load instructions that affect subsequent instructions, as well as first and second-level data dependencies that require careful handling. To address these issues, we have implemented a hazard unit in stage 4 of the pipeline architecture, which modifies the already-decided control signal in stage 2. The modified control signal is then used to control the muxes in stages 3,

4, 5, and 6, ensuring that the desired data is used in the execution of instructions.

For branch-type instructions, we have also implemented muxes in stages 1 and 4 to address data dependencies. By carefully controlling these muxes, we can ensure that the correct data is used in the execution of these instructions, avoiding incorrect results or system failure.

Additionally, we have implemented a stalling mechanism for instructions that follow load instructions and use the same data in the register file. In this scenario, we stall for one cycle by disabling write in the pipeline register, ensuring that the correct data is used in the execution of subsequent instructions.
Overall, we checked if there are any hazards i.e. instructions with immediate or 2nd order dependency in the execution stage of our 6-stage pipeline. We defined a hazard unit that checks for hazards and modifies the control signals coming from the controller in the Instruction decode stage. To deal with immediate or 2nd order dependency involving a common register as the common operand, we took its newly computed or loaded value from the pipeline registers and fed them back to the inputs of muxes connected to ALUs. So the controls of the muxes were then modified accordingly so that these lines are selected for the execution of the next instruction.
A few data hazard condition solutions(data forwarding) are described below:-

4.0.1 ADD ADD:-

In this case, the ALU output which is now in the next stage is sent back to the previous stage i.e. the execution stage. This ensures that the next add instruction can directly access the result without waiting to retrieve it from the register file.

4.0.2 LW ADD:-

For this, the data from the data memory which is now in the write back stage is sent back to the execution stage. Here the data is the one which will be loaded in the register which will be used for the ADD instruction. Here we are stalling for one cycle and then we are back forwarding data from the data memory line.

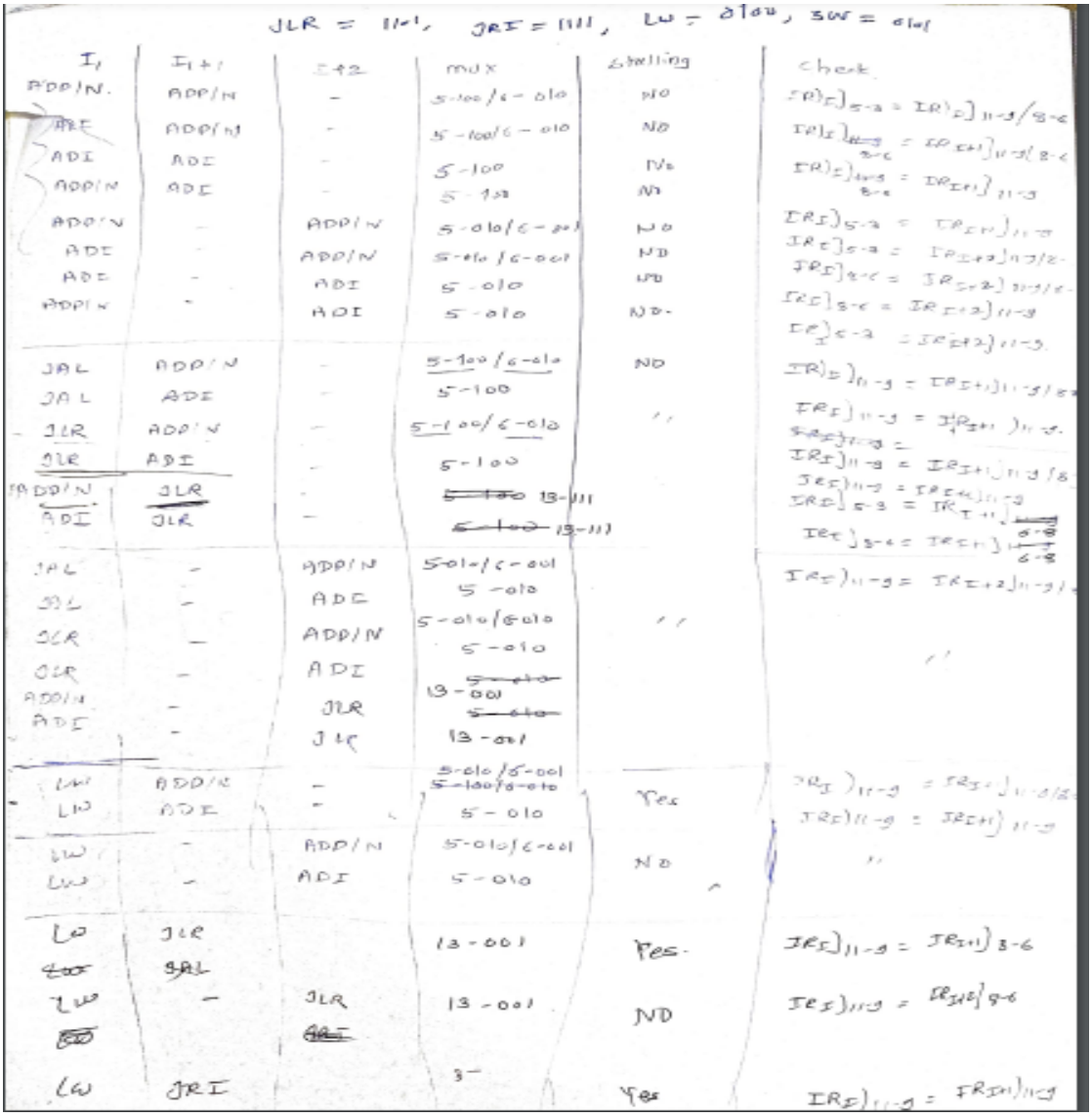
4.0.3 LW SW:-

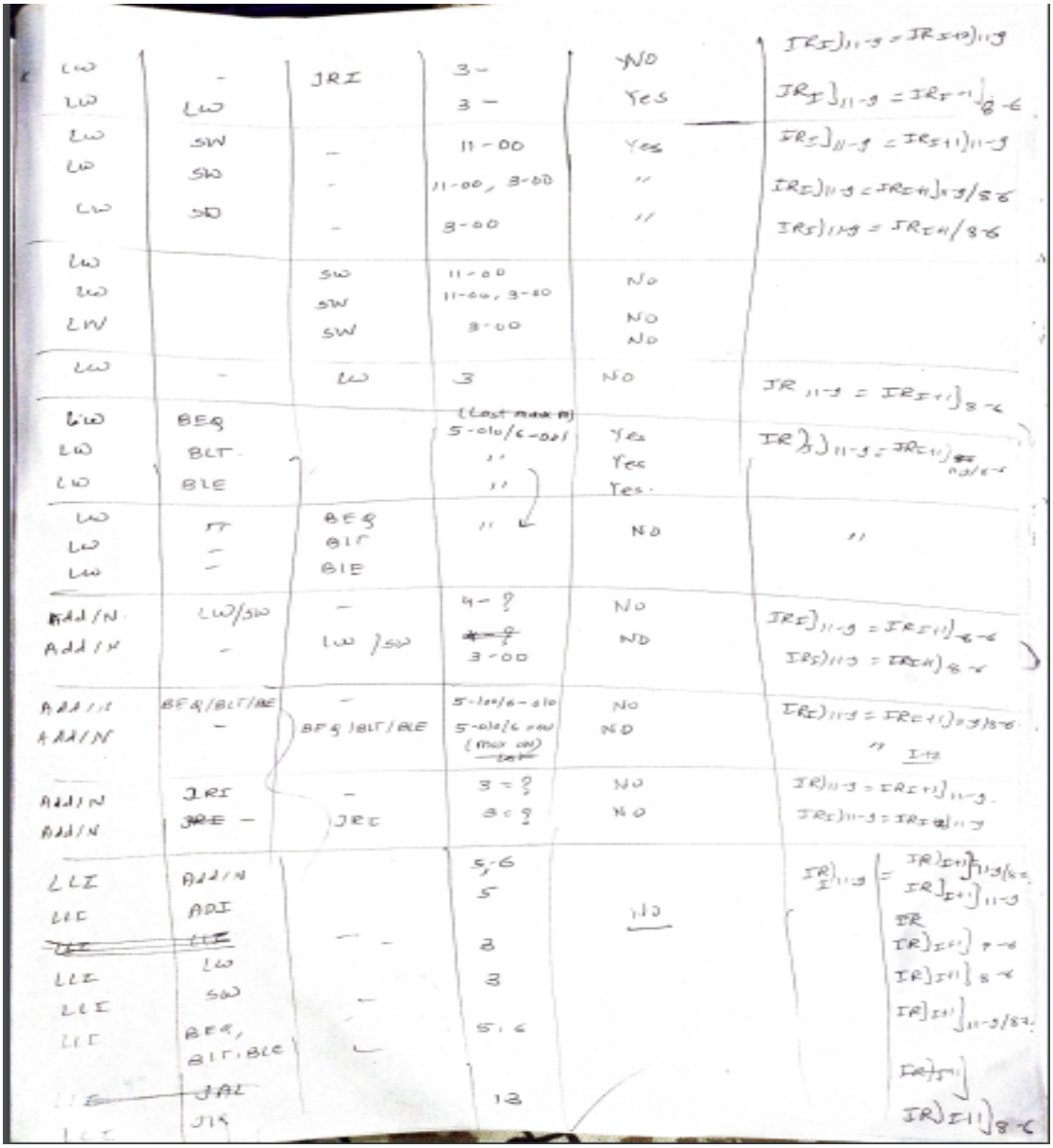
Here the output data due to the load instruction is sent back again to the data memory input as this is the exact same data that will be stored in the data memory.

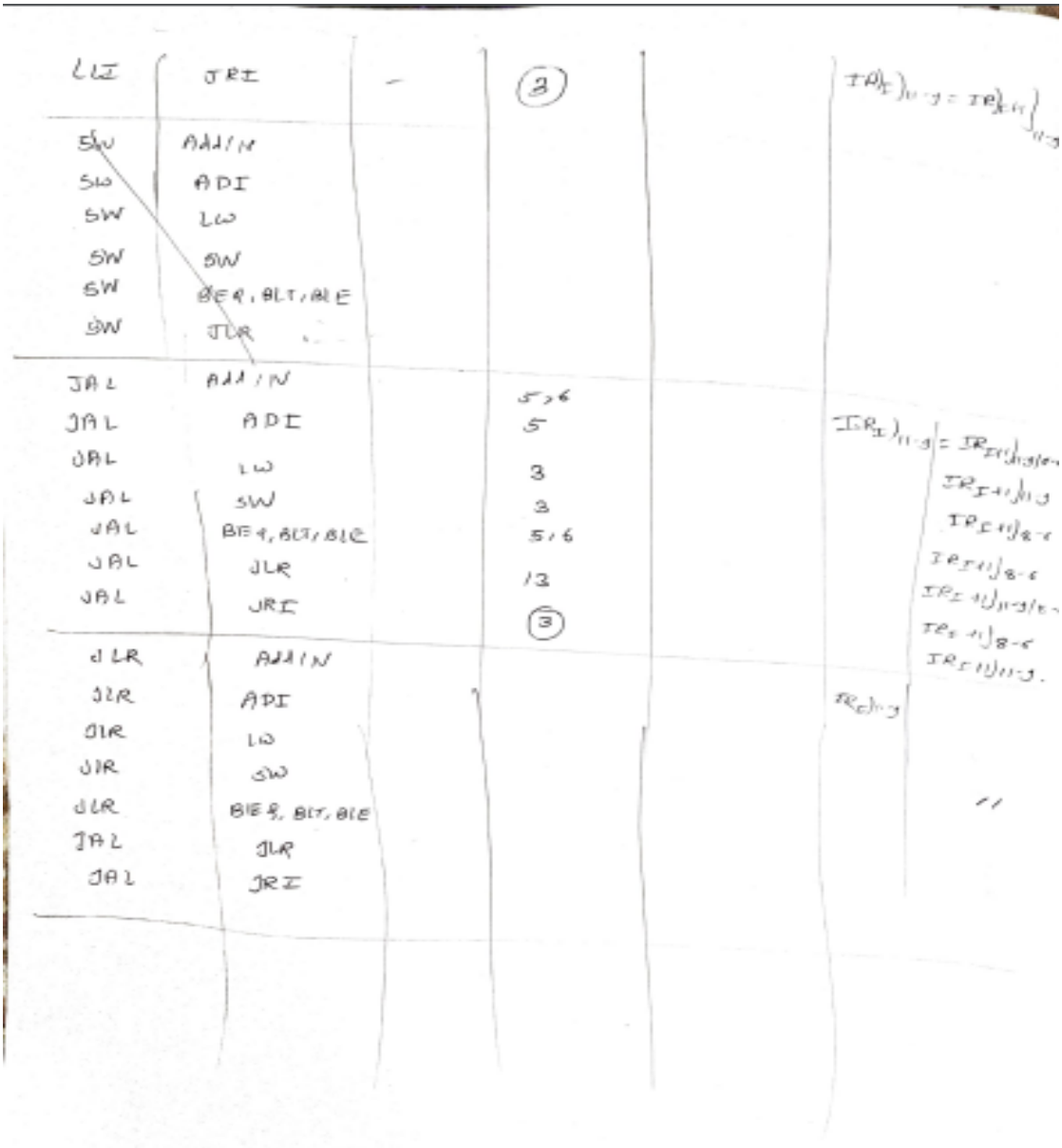
4.0.4 ADD SW:-

To deal with this data hazard condition, we send the ALU output(result) from the write-back stage back to the memory access stage as this result(calculated address) is the memory location that instruction SW needs.
Likewise, there are a lot of conditions to be checked for controlling data hazards.

Here we provide our approach to deal with the same.






LLI
JRI
3
SW
ADI
LW
SW
BEQ, BLT, BLE
JLR
ADI
JAL
JLR
5>6
5
3
3
5, 6
13
3