

**COEN 210: COMPUTER ARCHITECTURE**

PROJECT REPORT

**SEARCHING FOR AN ELEMENT IN AN ARRAY**

**Team Members**

CHIRAG RADHAKRISHNA & ATHARVA ABHAY BAL

**CONTENTS**

|  |  |
| --- | --- |
| **TOPIC** | **PAGE** |
| 1. Introduction… | **3** |
| 2. Objective/Benchmark… | **4** |
| 3. Instruction Set Architecture (ISA)…  3.1 Assumptions in the design  3.2Instruction set  3.3 Register File  3.4 Instruction Formats  3.5 Instructions  3.6 Instruction List | **5-12**  5  5-6  6-7  7-9  9-11  12 |
| 4.Assembly Code | **13** |
| 5. Datapath and Control | **14-20** |
| 6. Emulation of MIPS Instructions | **21-22** |
| 7. Performance Analysis | **23-25** |

**CHAPTER 1**

**INTRODUCTION**

We use many powerful programming languages in the present world. However, the computers are only capable of understanding machine-level instructions (machine language). The code we write must be compiled into assembly-level code. A simple function in C programming requires multiple lines of code in ARM or MIPS to perform the equivalent operation in the hardware.

The multi-cycle datapath (pipeline) allows the execution of multiple instructions in an overlapped fashion. At a given clock cycle, multiple instructions are executing in the datapath, with each instruction at a different stage. Over the years, multiple initiatives have been undertaken to improve pipeline performance and speed-up. Different solutions involve different pipeline stages, different control signals, and hardware components.

In our project, we have attempted to design an Instruction Set Architecture (ISA) and a pipeline implementation of the architecture to search an array for the presence of a particular value.

**CHAPTER 2**

**OBJECTIVE/BENCHMARK**

The aim of this project is to define an Instruction Set Architecture (ISA) and a pipeline implementation of this architecture to perform the following operation:

**INPUT:**

An unsorted array A = [a1, a2, …... an] and a value v.

**OUTPUT:**

Output = **k** if ak = v;

= **0**, otherwise.

**CHAPTER 3**

**INSTRUCTION SET ARCHITECTURE (ISA)**

**3.1 ASSUMPTIONS IN THE DESIGN:**

There are a few assumptions in our design. The assumptions are considered both for the architecture and the L-Type instruction.

The general assumptions are as follows:

* Word Length is exactly **four** bytes.
* There are a total of **64** registers and each register is **32** bits.
* 2’s complement is assumed.
* Time delays for individual components are **fixed** beforehand.
* The register file is divided into 2 halves of 32 registers each.

**3.2 INSTRUCTION SET:**

**Q.** What is an Instruction Set Architecture (ISA)?

**Ans.** The Instruction Set Architecture (ISA) is an abstraction layer that facilitates as the boundary between the hardware and the software.

Instructions in our Instruction Set Architecture (ISA) are **32-bit** each. The ISA includes the following instruction formats/types:

1. R-Type
2. I-Type
3. J-Type
4. L-Type

**R-Type:**

Instructions under this category are mainly involved in arithmetic and logical operations.

**I-Type:**

Instructions under this category are used to access the data memory, either to read data or write data into memory. Branch instructions are also included under this category to evaluate conditions.

**J-Type:**

The J-Type instructions are used to jump to a particular address. They cause the program control flow to change and direct execution to commence/continue at a different part of the program.

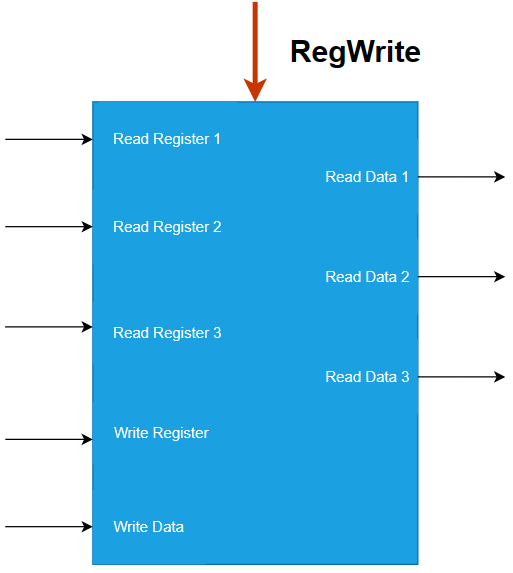
**L-Type:**

There is only one instruction in this category. The instruction is used to search for the presence of a value in an array of n elements.

**3.3 REGISTER FILE:**

The register file has a total of **64** registers. Each register is **32**-bit. We divide the register file into two halves of 32 registers each. The register file has **3 read** ports and **1 write** port. The R, I and J-Type instructions in our ISA read only two registers. Only the L-Type instruction reads 3 registers.

The **RegWrite** control signal helps determine if the specific instruction will be writing into a particular register.



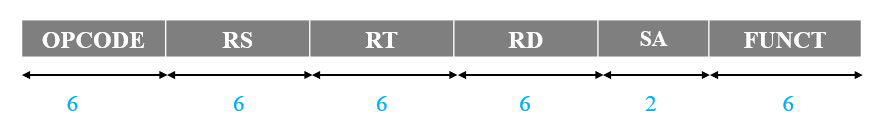
**Figure:** Register File Design

**3.4 INSTRUCTION FORMATS:**

**R-Type Format:**

*Format for the R-Type Instruction is as follows:*

1. **Opcode:** 6 Bits.
2. Register Source Operand 1[**RS**]: 6 Bits.
3. Register Source Operand 2 [**RT**]: 6 Bits.
4. Register Destination [**RD**]: 6 Bits.
5. **Shift Amount:** 2 Bits.
6. **Funct Field:** 6 Bits.

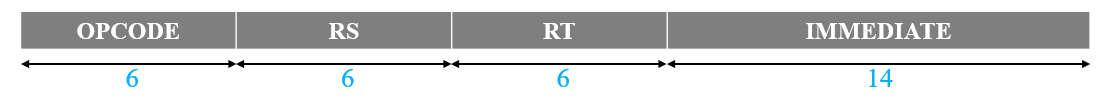
****

**Figure:** R-Type Instruction Format

**I-Type Format:**

*Format for the I-Type Instruction is as follows:*

1. **Opcode:** 6 Bits.
2. Register Source Operand [**RS**]: 6 Bits.
3. Register Destination [**RT**]: 6 Bits.
4. **Immediate:** 14 Bits.

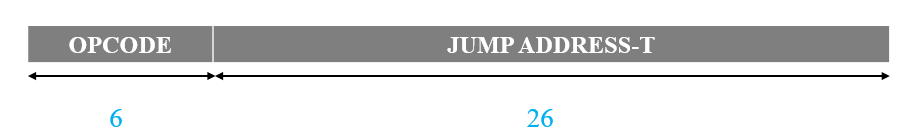


**Figure:** I-Type Instruction Format

**J-Type Format:**

*Format for the J-Type Instruction is as follows:*

1. **Opcode:** 6 Bits.
2. Address to Jump (**Target**): 26 Bits.

****

**Figure:** J-Type Instruction Format

**L-Type Format:** The L-Type Instruction is the only instruction type that reads data from 3 registers and accesses the second half of the register file to read data.

*Format for the L-Type Instruction is as follows:*

1. **Opcode:** 6 Bits.
2. Register Source Operand 1[**RS**]: 6 Bits.
3. Register Source Operand 2 [**RT**]: 6 Bits.
4. Register Destination [**RD**]: 6 Bits.
5. Register Source Operand 3 [**RU**]: 6 Bits.
6. **Unused:** 2 Bits.

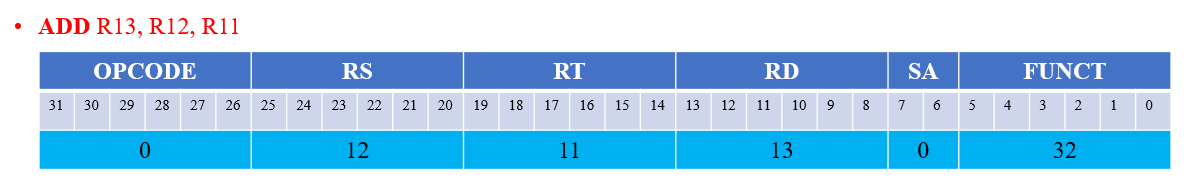


**Figure:** L-Type Instruction Format

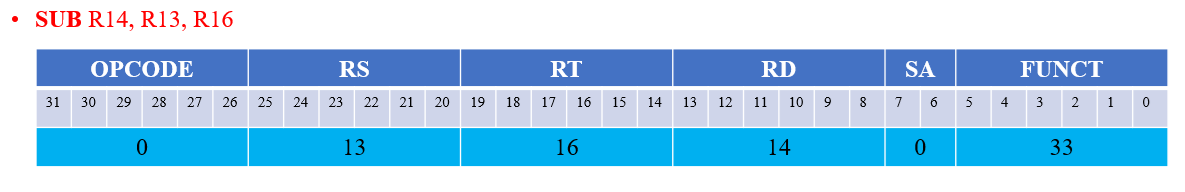
**3.5 INSTRUCTIONS:**

**R-Type Instructions:**

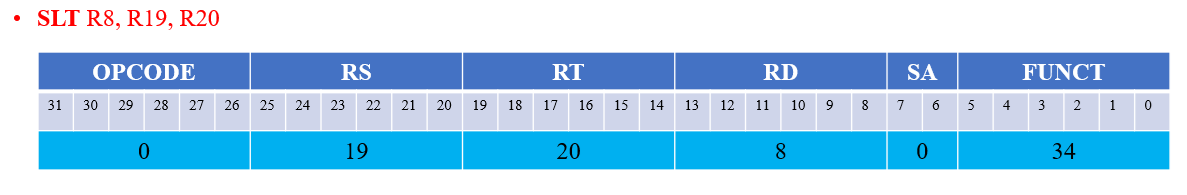
1. **Add:** Adds two register operands.

****

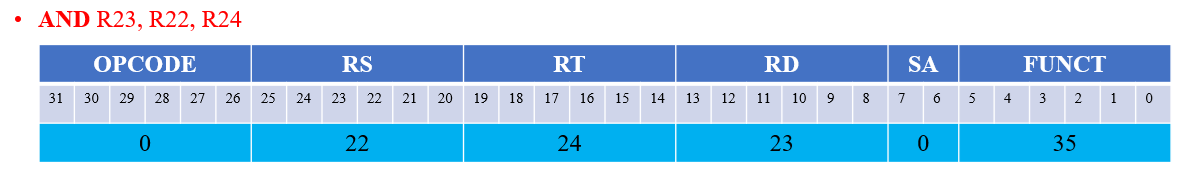
1. **Sub:** This instruction performs subtraction between the two register values.



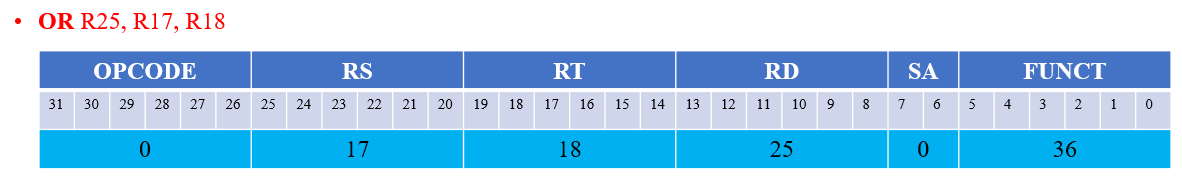
1. **Slt:** Compares two registers and sets the destination register value to 1 if the value of the first register is less than that of the value of the second register. Else, the output is 0.

****

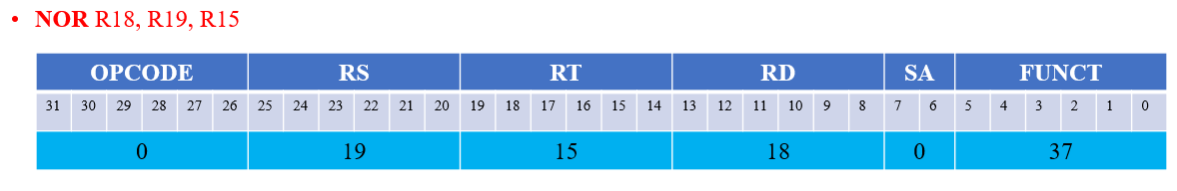
1. **And:** Performs bit by bit and (&) on the data present in the two input/source registers.

****

1. **Or:** Performs bit by bit or (|) on the data present in the two input/source registers.

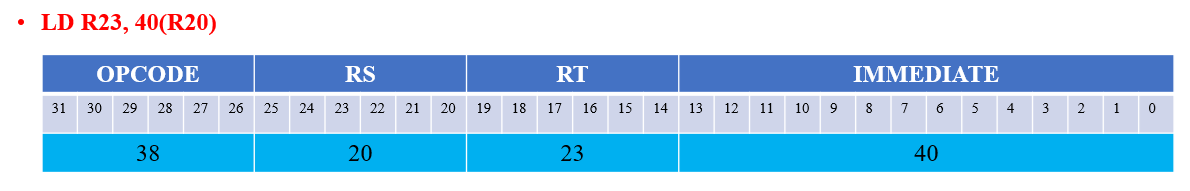
****

1. **Nor:** Performs bit by bit nor (~|) on the data present in the two input/source registers.

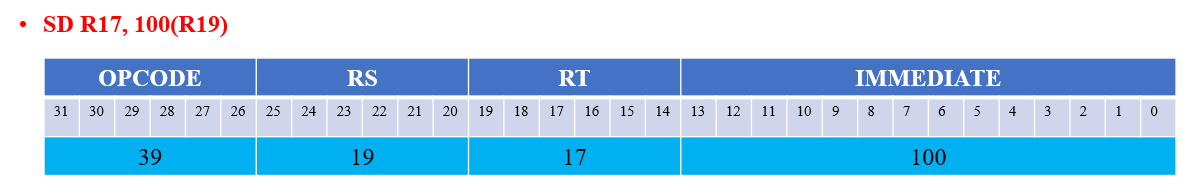
****

**I-Type Instructions:**

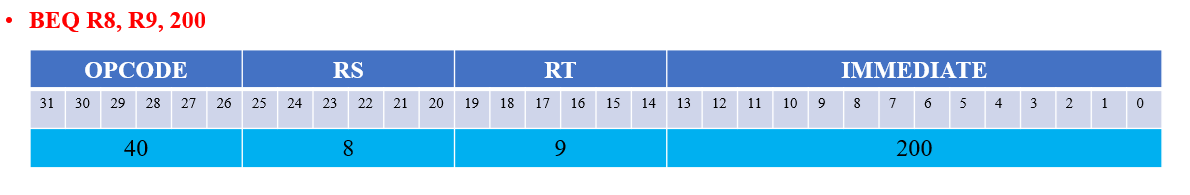
1. **Ld:** Loads data from memory to the destination register.

****

1. **Sd:** This instruction helps write data into the data memory.

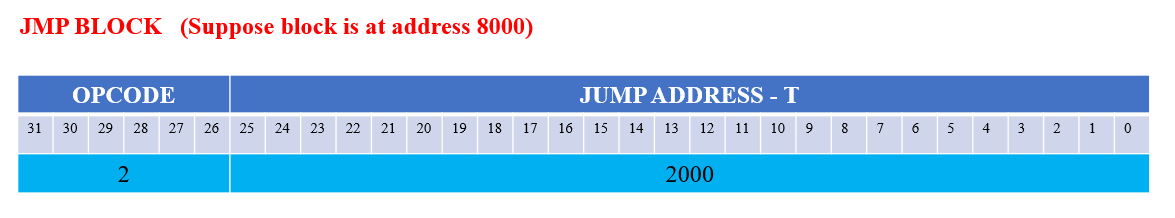
****

1. **Beq:** Compares the values of two registers for equality. If equal, execution will continue at the target address.

****

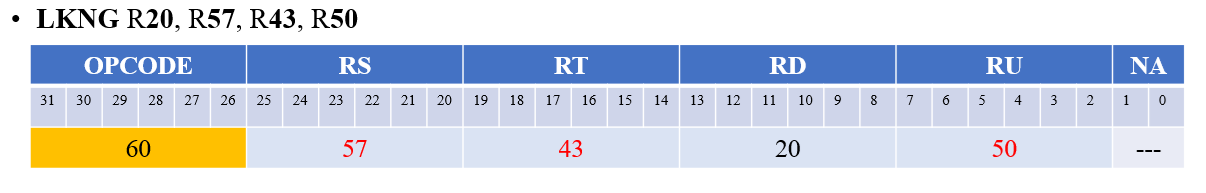
**J-Type Instructions:**

1. **Jmp:** Helps jump to a particular address/location.

****

**L-Type Instructions:**

1. **Lkng:** This instruction searches for the presence of an element in an array and returns the index at which the element is present (if found).

****

**Assumptions for the LKNG Instruction:**

* The LKNG instruction reads 3 registers.
* It is the only instruction in our ISA that reads from the second half of the register file.
* **R50** is the register reserved for storing the n value.
* **R43** contains the starting address of the array element (first element).
* **R57** comprises the value to be searched for.

**3.6 INSTRUCTION LIST:**

a. Add

b. Sub

c. Slt

d. And

e. Or

f. Nor

g. Ld

h. Sd

i. Beq

j. Jmp

k. Lkng

**CHAPTER 4**

**ASSEMBLY CODE**

Equivalent code using C programming:

int array [] = [a0, a1, ..., an];

int value = v;

for (int k = 0; k<n; ++k)

{

if (v == a[k])

printf (“%d”, k);

}

printf(“%d”,0);

To perform this operation of searching for an element in the array in the hardware, the LKNG can be used to perform this operation.

Hence the assembly code is as follows:

**LKNG** Destination, value, start\_address, n (Instruction Format)

**LKNG** R**20**, R**57**, R**43**, R**50**

We require **one** instruction to perform the required operation.

**CHAPTER 5**

**DATAPATH AND CONTROL**

**PIPELINE:**

Our datapath uses a pipeline implementation. The pipeline allows us to execute multiple instructions simultaneously. There are a total of 5 pipeline stages. They are as follows:

**IF:** Instruction fetch

**ID:** Instruction decode and register fetch

**EX:** Execution and effective address calculation

**MEM:** Accessing the data memory

**WB:** Write back to the register.

**COMPONENTS:**

In our datapath design, we have added **three** new components to assist in achieving the execution of the operation. The components are explained as follows:

1. **Compare Box:**

This component is based on the XOR gate. The compare box takes two values as input and tests them for equality. Suppose it takes two values a and b, it works and provides the output as follows:

**1**, if a == b

**0**, otherwise.



**Figure:** Compare Box

1. **Counter:**

The counter performs the equivalent action of a for-loop counter. It initially takes the value 0. For subsequent iterations, it is fed the latest value of the counter variable. It simply increments the counter variable by one.



**Figure:** Counter

1. **Loop Unit:**

Helps in incrementing the loop counter and keep track of the address of the array element under examination.

**Inputs to the loop unit:**

* Loop control signal.
* Output from OR gate (loop-end) which helps determine if the loop has ended.
* Present counter value.
* Current array element address.

**Outputs of the loop unit:**

* Forwarded value of loop counter.
* Forwarded value of the array element address

The loop unit also has a reset bit. Based on the value of the loop control signal and the output from the or gate (loop-end), the counter and address values are reset to begin a set of new iterations in the hardware.

Reset bit value = loop-end & loop

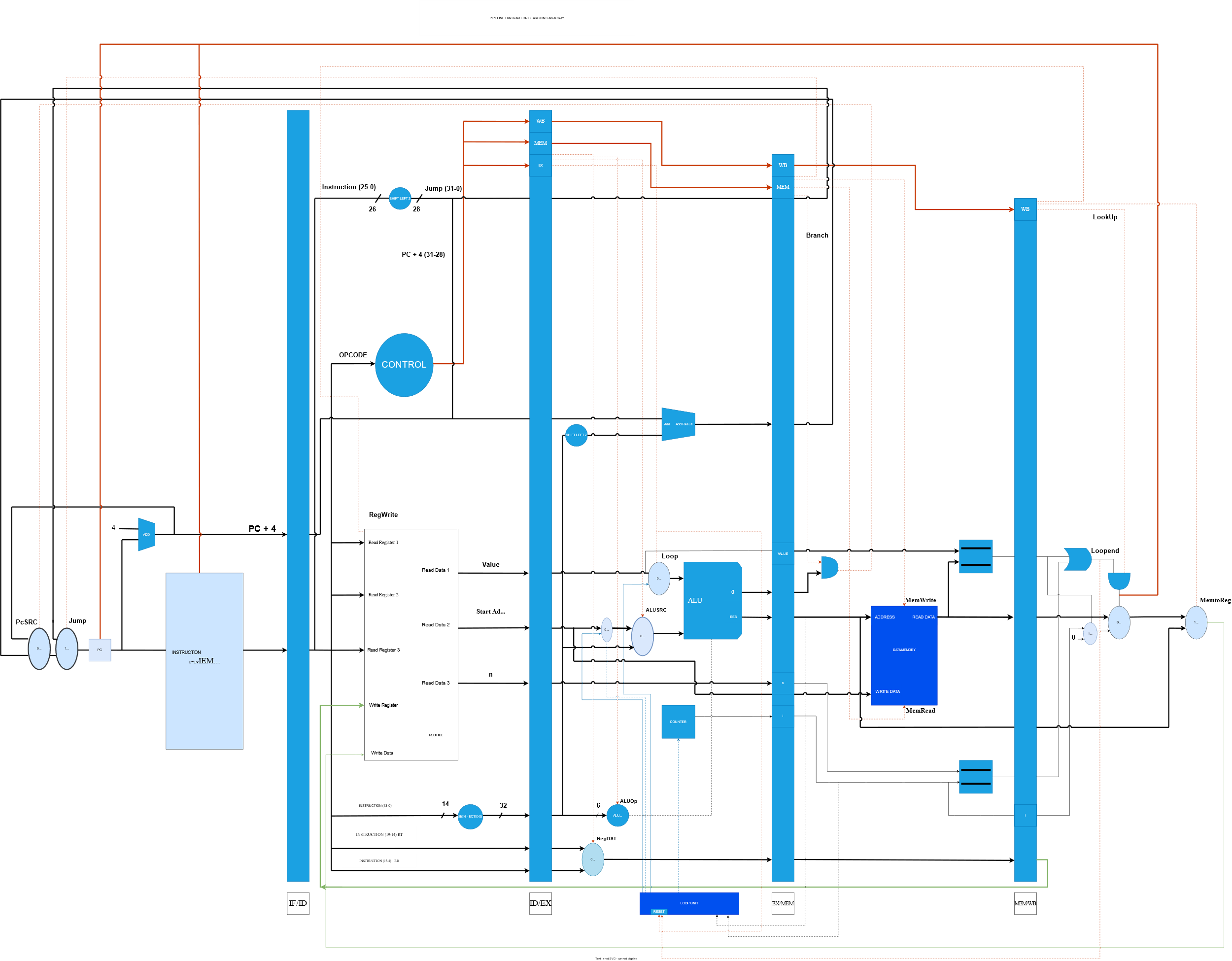
If reset = 1, reset counter to default value. Read address from register.

If reset = 0, forward the latest address and counter values.



**Figure:** Loop Unit

**DATAPATH:**



**Figure:** Datapath implementing the operation

**CONTROL SIGNALS:**

There are a total of **11** control signals generated by the control component/controller based on the opcode of the instruction. The PcSRC control signal is generated in the MEM stage. Most control signals are same as the ones used in the 5 stage MIPS Pipeline. For our datapath design, we have added two new control signals. They are the **loop** control and the **lookup** control respectively.

The control signals are passed through the pipeline registers in a manner like the data flow.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Execution and Effective Address Calculation Stage Control Lines | | | | Mem Stage Control Lines | | | | WB Stage Control Lines | | |
| **RegDST** | **AluOp**  **(2 Bit)** | **ALUSrc** | **Loop** | **Branch** | **Jump** | **MemRead** | **MemWrite** | **LookUp** | **RegWrite** | **MemToReg** |

**Table:** Control Signals

The above table shows the control signals generated by the controller based on the opcode of the instruction. Also, we have mentioned the stage in the pipeline where they will be used. The functionality of each control signal is explained below:

**EX Stage:**

1. RegDST: Used to select the destination register rd or rt.
2. ALUSrc: This control signal determines if the second ALU input is from the register or the immediate value that is obtained from the ID Stage.
3. AluOp: This 2-bit code helps determine the operation to be performed in the ALU.
4. Loop: This control signal is set to 1 when the LKNG instruction is being executed. It helps activate the loop unit and commence the execution of the loop in the datapath (hardware).

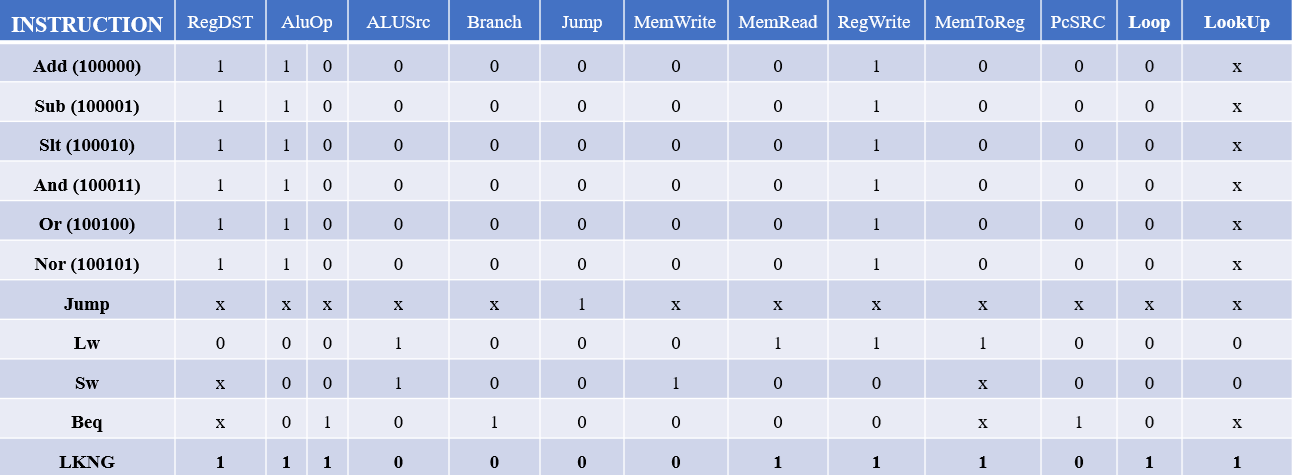
**MEM Stage:**

1. Branch: Indicates that the current instruction is a branch and facilitates the calculation of the PcSRC control signal.
2. MemRead: This control signal helps read data from the data memory at the specified location.
3. MemWrite: Enables an instruction to write data into the data memory at the specified location.
4. Jump: Set to 1 for a jump instruction. It helps jump to the target address and continue the execution from that point.

**WB Stage:**

1. RegWrite: Determines if data must be written into the destination register in the register file.
2. MemToReg: MemToReg determines if the data to be written back to the register is from data memory or the present ALU result.
3. LookUp: This control signal is set to 1 for the LKNG instruction. This indicates that we are looking up the values in the respective array. Further, it helps determine if the loop execution has finished in the hardware and helps write the index value to the destination register.

Control Signals by instruction:



**Table:** Control Signal values by instruction

**CLOCK CYCLES:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **INSTRUCTION** | **CLOCK CYCLES** | | | | |
| 1 | 2 | 3 | 4 | 5 |
| LKNG R20, R19, R18, R50 | **IF** | **ID** | **EX** | **MEM** | **WB** |

**WORKING/FLOW OF LKNG INSTRUCTION IN THE DATAPATH:**

To implement the operation for searching an array for the presence of the key value, we have implemented the **LKNG** instruction. This instruction belongs to the L-Type category and reads 3 register values. It is the only instruction in our ISA that reads from the second half of the register file. The working through the 5 stages of the pipeline are as follows:

**IF:** The LKNG instruction is fetched from the instruction memory.

**ID**: 3 registers are read to obtain the starting address of the array element, the number of elements in the array (n) and the value to be searched. The control signals are also generated for the instruction.

**EX:** Here, the loop control signal is directed to the loop unit. The loop unit starts the loop counter. The ALU is used to calculate the effective address (word-aligned) and the value to be searched is simply forwarded to the next stage.

**MEM:** The array value is read. The compare box compares the value (key) with the array element. At the same time, the other checks if the loop counter has equalled the n value. The results from this compare boxes are forwarded to the WB stage. The present values of the array element address and loop counter are passed to the loop unit which forwards these values to run the next iteration.

**WB:** The results from the compare box are passed through an OR gate. Essentially if the value equals the present array element or the loop counter equals n, we must exit the loop execution. Therefore, the OR gate output is fed along with the lookup control signal (exerted for LKNG) to an AND gate. The AND gate output helps determine if we need to select data from memory or the present loop counter value to be written to the destination register. The AND gate takes the lookup control signal as input. The lookup signal indicates that we have looked up a value from memory. The loop counter value to be written is influenced by the output of the compare box which compares the value and the array element. If they are equal, the current loop counter value will be written. Else, 0 will be written to the destination register.

After selecting the loop counter value, this value passes through MemToReg and is written to the register.

**CHAPTER 6**

**EMULATION OF MIPS INSTRUCTIONS**

For a few instructions in MIPS, we have formulated equivalent instructions in our ISA. The following table provides equivalent instructions for the following MIPS instructions:

|  |  |
| --- | --- |
| **MIPS Instruction** | **Equivalent Instruction in our ISA** |
| Add $s1, $s2, $s3 | Add R13, R12, R11 |
| Sub $s1, $s2, $s3 | Sub R14, R13, R16 |
| Slt $t0, $s3, $s4 | Slt R8, R19, R20 |
| And $t0, $t1, $t2 | And R23, R22, R24 |
| Or $s1, $s2, $s3 | Or R25, R17, R18 |
| Nor $s1, $s2, $s3 | Nor R18, R19, R15 |
| Lw $t0, 32($s3) | Ld R23, 40(R20) |
| Sw $s1, 100($s3) | Sd R17, 100(R19) |
| Beq $t1, $t2, 100 | Beq R8, R9, 200 |
| J Exit | JMP Block |

|  |  |
| --- | --- |
| **Our ISA Instruction** | **Meaning** |
| Add R13, R12, R11 | R13 = R12+R11 |
| Sub R14, R13, R16 | R14 = R13-R16 |
| Slt R8, R19, R20 | R8 = R19<R20? 1:0 |
| And R23, R22, R24 | R23 = R22&R24 |
| Or R25, R17, R18 | R25 = R17 | R18 |
| Nor R18, R19, R15 | R18 = R19~|R15 |
| Ld R23, 40(R20) | R23 = Mem [R20 + 40] |
| Sd R17, 100(R19) | Mem [100 + R19] = R17 |
| Beq R8, R9, 200 | If (R8==R9) goto (PC + 4 + (200\*4)) |
| JMP Block | Goto PC + 4 + (4\*Block) |
| LKNG R20, R57, R43, R50 | R20 = k if R57 in array, else 0 |

**CHAPTER 7**

**PERFORMANCE ANALYSIS**

In this chapter, we analyse the performance of our datapath.

The delays for the components in our datapath are as follows:

1. ALU: 2ns
2. Adder: 2ns
3. Register File: 2ns
4. Instruction Memory: 2ns
5. Data Memory: 2ns
6. Counter, Loop-unit: 1 ns
7. Mux, Buffers, Or Gate, And Gate & Compare Box = 0 ns

**Cycle Time** = 2ns

The cycle time is 2 ns. The critical path for the LKNG instruction includes the instruction fetch, reading from the register file, calculation of the branch address in the ALU, reading the array element from data memory, and comparing the value with the element, which is lastly followed by the write-back. The delay caused due to the components present along this critical path is 2 ns each. The other paths involved during execution of the LKNG instruction include, calculation of the new PC value, the loop counter increment and forwarding of the address and counter values. These are not considered to be part of the critical path and hence the cycle time is 2ns.

**Number of Cycles** = 5 + (N-1) = N + 4

**CPI** = (N+4)/1 = N+4

**Execution Time** = [2 \* (N+4)] ns

**Performance** = 1/ (Execution Time)

**JUSTIFICATION OF OUR ISA AND DATAPATH**:

We can successfully implement the necessary operation using our ISA and datapath. We have added new components that help simplify the process of our search for the value. Further, there is a clear understanding of the flow of the data and how the output is obtained.

**PROS:**

1. The equivalent MIPS code to execute the operation/benchmark is as follows:

Loop: #Load current array element into register t2.

Lw $t2, 0($t0)

#If array element equals target value, we exit the Loop.

Beq $t0, $t2, Exit

#Obtain the address of the next array element

Addi $t0, $t0, 4

#Increment the loop counter

Addi $t1, $t1, 1

#Execute the loop again if value is not found during the current iteration

J Loop

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instruction | Clock Cycles | | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Lw $t2, 0($t0) | IF | ID | EX | MEM | WB |  |  |  |  |
| Beq $t0,$t2, Exit |  | IF | ID | EX | MEM | WB |  |  |  |
| Addi $t0, $t0, 4 |  |  | IF | ID | EX | MEM | WB |  |  |
| Addi $t1, $t1, 1 |  |  |  | IF | ID | EX | MEM | WB |  |
| J Loop |  |  |  |  | IF | ID | EX | MEM | WB |

If we examine the clock cycles taken by the equivalent MIPS code to implement the loop to find the element, we require a minimum of 9 cycles. This loop can run at most ‘n’ times. More cycles are required for instructions that load the target value, initialise the loop counter and load the final output into a register. We can reduce the number of cycles taken by using our ISA and datapath. Further, we have achieved the execution of the operation using a single instruction.

1. The new hardware components that we have added to the datapath help us clearly specify how we achieve the execution of the specified operation (benchmark).
2. Given the assumptions we have made for our LKNG instruction beforehand, a clear understanding of the instruction, its execution and performance has been established.

**CONS:**

1. For our new instruction, a few bits have not been utilized and are unused.
2. By adding new components and while their purpose is justified, we have not used many registers in implementing the specified operation (benchmark).

**Therefore, by examining the performance analysis, the advantages and disadvantages of our instruction set and datapath, we believe our design suits the purpose and offers reasonably good performance.**