COEN 210: Computer Architecture
Fall 2023

GROUP 08 - Project Report

TEAM MEMBERS

Dhanasekar Ravi Jayanthi (SCU ID : 07700005982)
Manasa Madiraju (SCU ID : 07700009942)
Pujitha Kallu (SCU ID : W1653660)
Rohit Roy Chowdhury (SCU ID :7700000162)

Guided by: Prof. Yan Cui

School of Engineering
Santa Clara University
Santa Clara, CA

TABLE OF CONTENTS

In this project, you will design a 32-bit pipelined CPU for the given SCU Instruction Set Architecture (SCU ISA). The SCU ISA is described below.

- Register file size: 64 registers, each register has 32 bits.
- PC: 32 bits
- 2's complement is assumed.
- Instruction format: Each instruction is 32-bit wide, and consists of five fields: `opcode`, `rd`, `rs`, `rt`, and `unused`. The format is as follows.

Opcode (4 bits)	rd (6 bits)	rs (6 bits)	rt (6 bits)	unused(10 bits)

The 13 instructions are defined below.

Instruction	Symbol	Opcode	rd	rs	rt	Function
No operation	**NOP**	**0000**	×	×	×	**No operation**
Save PC	**SVPC rd, y**	**1111**	**rd**	**y**		**xrd ← PC+y**
Load	**LD rd, rs**	**1110**	**rd**	**rs**	×	**xrd ← M[xrs]**
Store	**ST rt, rs**	**0011**	×	**rs**	**rt**	**M[xrs] ← xrt**
Add	**ADD rd, rs, rt**	**0100**	**rd**	**rs**	**rt**	**xrd ← xrs + x rt**
Increment	**INC rd, rs, y**	**0101**	**rd**	**rs**	**y**	**xrd ← xrs + y**
Negate	**NEG rd, rs**	**0110**	**rd**	**rs**	×	**x rd ← - xrs**
Subtract	**SUB rd, rs, rt**	**0111**	**rd**	**rs**	**rt**	**x rd ← xrs – x rt**
Jump	**J rs**	**1000**	×	**rs**	×	**PC ← xrs**
Branch if zero	**BRZ rs**	**1001**	×	**rs**	×	**PC ← xrs, if Z = 1**
Jump memory	**JM rs**	**1010**	×	**rs**	×	**PC ← M[xrs]**
Branch if negative	**BRN rs**	**1011**	×	**rs**	×	**PC ← xrs, if N = 1**
MAX	**MAX, rd, rs, rt**	**0001**	**rd**	**rs**	**rt**	**See ***

*xrd = Max{memory[xrs], memory[xrs + 1], …, memory[xrs + xrt -1]},
×: don't care

Question :-

Use the instructions in the SCU ISA to write two versions of the assembly program of finding the maximum in *n* numbers described below.
(1) MAX =max {a1, a2, a3,....an}
a.The first version (software loop) does not use the MAX instruction and
b. The second one (hardware loop) uses the MAX instruction.

Register Assumptions:

x0 - Constant 0 'Zero' value
x5 - Address of the 1st element index in array
x7 - First element of an array or current index [i]
x8 - Length of an array [n]
x10 - Next element in the iteration
x11 - Maximum number in the array
x15 - Current maximum
x14 - Address of the current element in an array
x15, x16, x17, x18, x19 - Temporary registers to store values

Version 1 : (Software Loop)

Software Loop (without using Max Instruction) and by using instructions provided.

ASSEMBLY CODE:

```
1.ADD x7, x7, x0        // Add x7 and x0 ,store result in x7 (Assigning value i)
2.INC x8, x8, -1        // Decrement the value in x8 by 1 and store in x13 (n-1)
3.BRZ x0                // Branch to exit if n==0;
4.ADD x14, x5, x0       // Add x5 and x0 and store the result in x14
5.LD x15, x14           // Load the value from memory location pointed by x14 into x15
6.SVPC x16, 52          // Save the program counter (PC) value plus 52 into x16
7.SVPC x17, 40          // Save the PC value plus 40 into x17
8.SVPC x18, 4           // Save the PC value plus 4 into x18
9.SUB x19, x7, x8       // Subtract x8 from x7 and store the result in x19
10.BRZ x16              // Branch to the address in x16 if value in x19 is zero
11.INC x14, x14, 4      // Increment the value in x14 by 4
12.INC x7, x7, 1        // Increment the value in x7 by 1
13.LD x10, x14          // Load the value from the memory location x14 into x10
14.SUB x11, x15, x10    // Subtract x10 from x15 and store the result in x11
15.BRN x17              // Branch to the address in x17 if value in x11 is negative
16.J x18                // Jump to the address stored in x18
17.ADD x15, x10, x0     // Add x10 to x0 and store the result in x15
18.J x18                // Unconditional jump to the address stored in x18
19.NOP                  // No-Operation, EXIT
```

Example:

M = [5,6,9,8]
Addresses = [1000,1004,1008,1012] , n=4

Explanation :

In summary, the code iterates through an array element [n], maintaining a maximum value by comparing it with each element in the array. It updates the maximum value when it encounters a value greater than the current maximum. The code uses conditional branches to control the loop and update the maximum value as needed.

Version 2 : (Hardware Loop)

Hardware Loop (Using Max Instruction)

Register Description:

x1 - stores the current max value

x3 - Size of the input array, counter

x4 - Address of the size of the array

```
1.LD x3, x4              // load the size of array from x4 to x3
2.INC x4, x4, 0x4        // move current position to the first element of an array
3.MAX x1, x4, x3         // find maximum element of the array using MAX
4.NOP                    // Exit
```

Explanation:

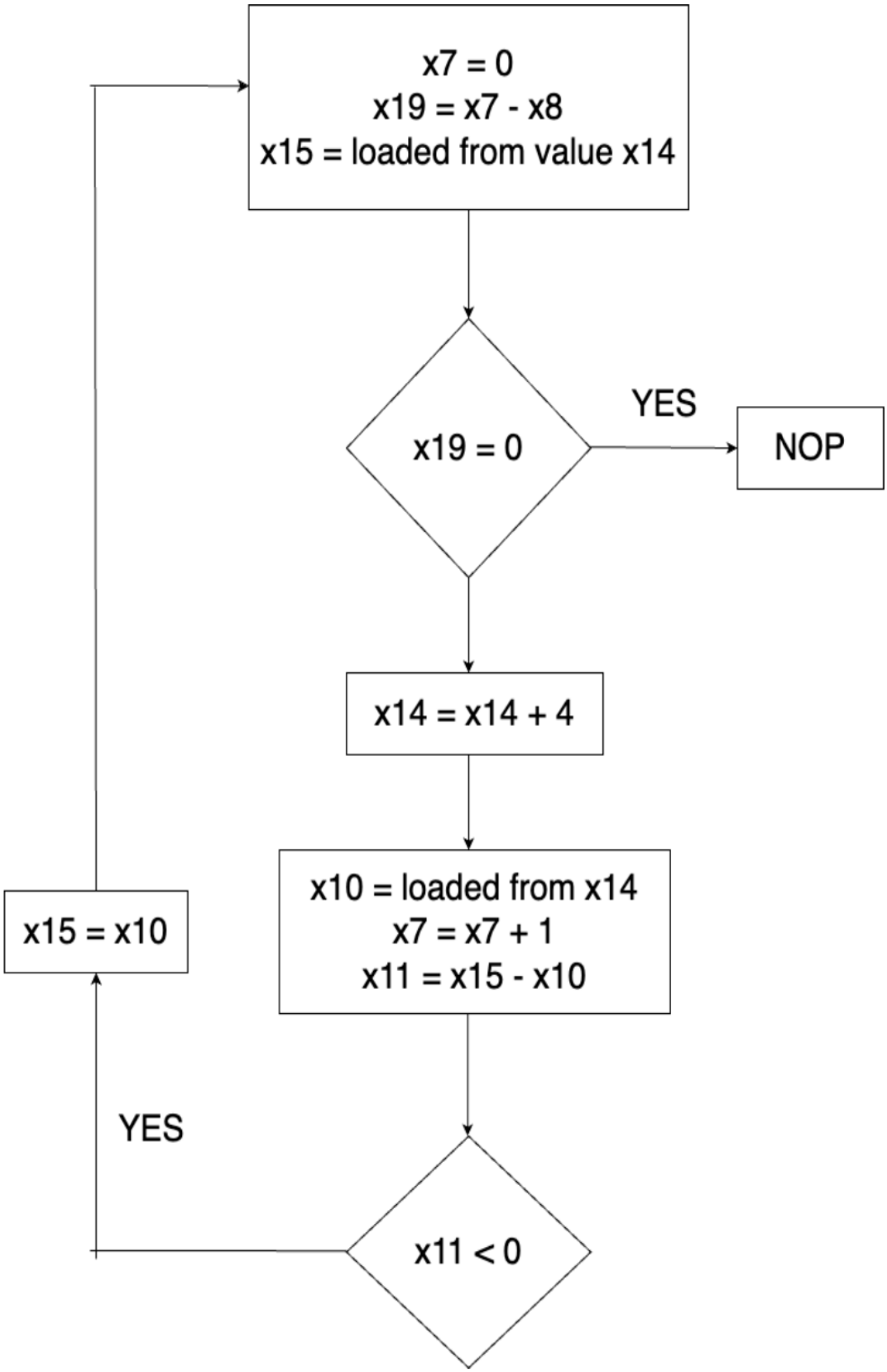
In summary, the code is to load the size of an array, set the current position to the first element of the array, and then potentially find the maximum element within the array using custom "MAX" instruction.

Equivalent python code:

```
def largest(arr1, n):

 # Initialize maximum element
      max = arr1[0]
 # Traverse array elements from second and
#   compare every element with current max by calculating difference
    for i in range(1, n):
        delta = max-arr1[i]
       if delta<0:
            max = arr1[i]
        return max
# Driver Code
   arr1 = [5,6,9,8]
   n = len(arr1)
  Ans = largest(arr1, n)
  print("Largest in given array ", Ans)
```

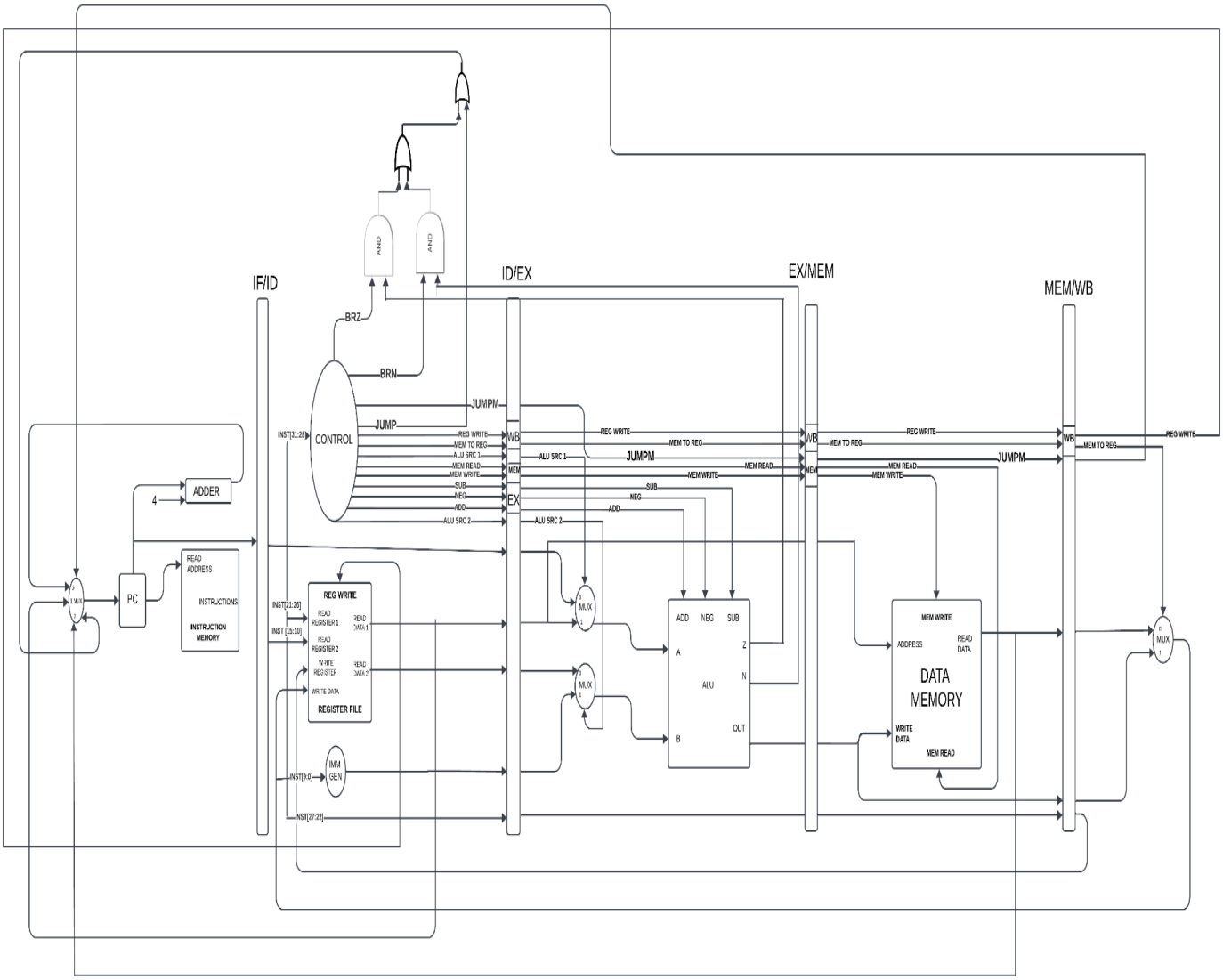
Flow Diagram :



Datapath And Control

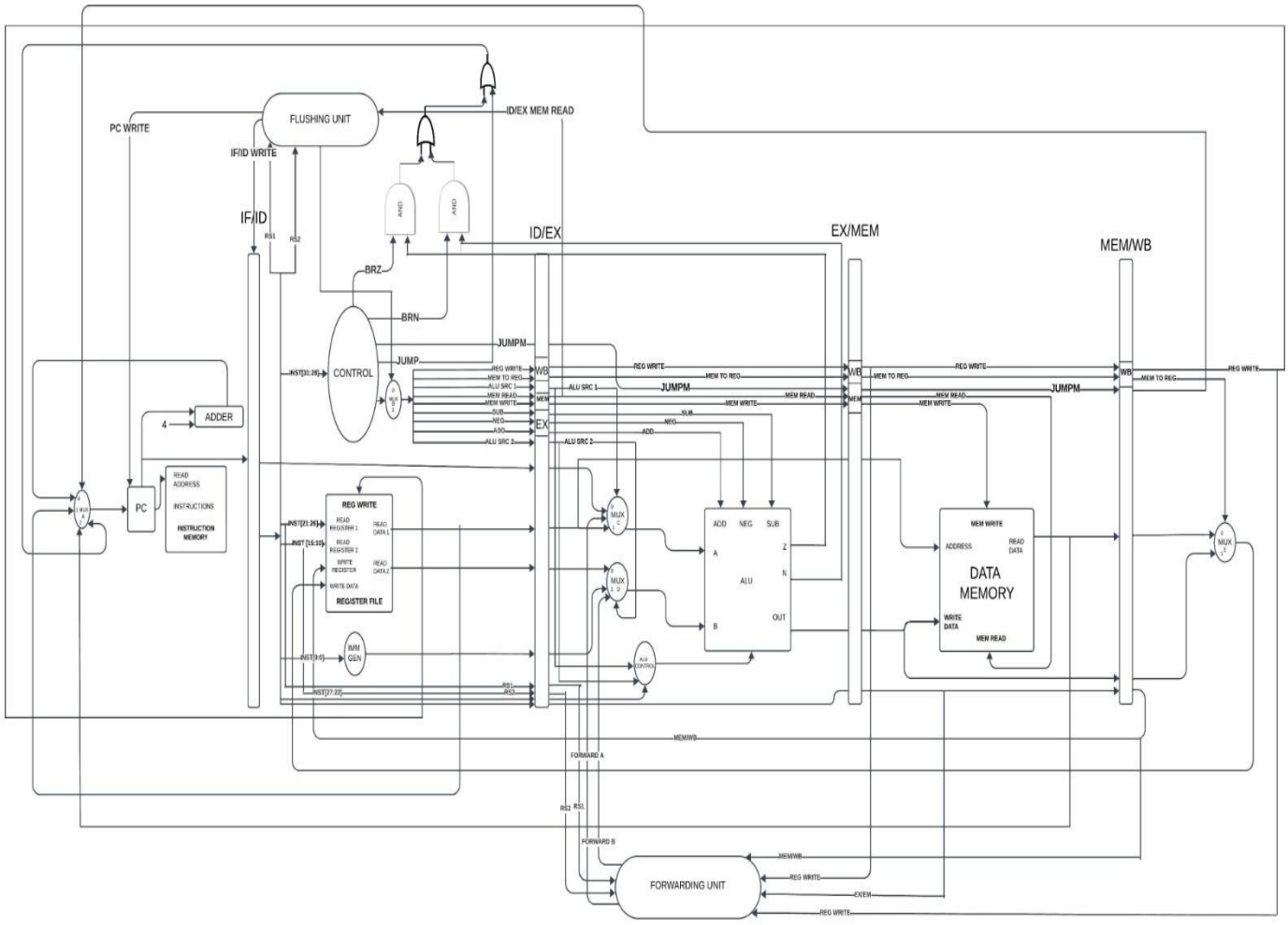
Datapath Figure 1:

A datapath with no hazards is an idealized design that ensures smooth and uninterrupted execution of instructions without encountering any delays or conflicts. Hazards in a datapath typically arise from dependencies between instructions, including data hazards, control hazards, and structural hazards.
The below datapath is the abstract version of the multicycle datapath without any hazards.



Datapath Figure 2:

Data hazards emerge due to the reliance of one instruction on a preceding one that is still in the pipeline. For instance, consider the case from our code where an Load instruction is immediately followed by a Subtract instruction both involving the use of register **x15**. In contrast, a datapath with hazards requires additional mechanisms to handle conflicts and dependencies to ensure correct and efficient execution of instructions. The primary solution revolves around the realization that there's no necessity to wait for the instruction to finish before attempting to address the data hazard. Therefore, the datapath is enhanced by incorporating additional components, namely the **Forwarding Unit** and **Flushing Unit** to avoid potential hazards.



Explanation:

There are in total 5 stages in data pipelining:

a) Information fetch
b) Information decode
c) Execution (ALU unit)
d) Memory Access
e) Write Back

→ **Instruction fetch**:

This stage contains the Program Counter (PC), responsible for storing the memory address of the next instruction to be fetched, and the Instruction Memory, which holds the instructions to be executed. The role of this stage is to fetch the instruction located at the memory address specified by the Program Counter. Fetching the instruction from the corresponding memory location. Incrementing the PC is achieved through an adder, and alternative methods involve instructions like BRN, BRZ, and JUMP, as demonstrated in lines 5, 6, 7. The current address is stored in registers x16, x17, x18 during specific instructions, such as line 9, where a branch at zero sets the PC to the value stored in register x16. Multiple PC incrementing approaches are managed through a Multiplexer (MUX) controlled by specific signals. This strategy ensures systematic traversal of the entire array, facilitating comparisons for the identification of the maximum array element using the output.

→ **Instruction decode:**

Here, the Control Unit and the Registers are placed. The Control Unit interprets the fetched instruction to determine the necessary actions and signals to execute it. Meanwhile, the Registers store data temporarily and facilitate operations within the CPU.Upon fetching an instruction, it undergoes decoding into various types, including R, D, and I. The values extracted from these instructions are then read and stored in distinct registers with the assistance of register files. In the case of I-type instructions, when an immediate value is present, an immediate converter becomes essential to extend the bit representation from 16 bits to 32 bits.

→ **Execution**:

This stage involves various components, including the Arithmetic Logic Unit (ALU) that performs arithmetic and logic operations. Two Multiplexers (MUX) provide inputs to the ALU, allowing it to receive and process the required data or instructions. The execution unit manages all necessary arithmetic operations for determining the maximum value, relying on control signals such as Add, Neg, Sub, etc. Control signals like ALU Src 1 and 2 govern whether the second input should be derived from read data or an immediate value. Within the ALU, the Z signal indicates when the current pointer reaches the array's length ($Z = 1$). Conversely, the N signal handles situations where the maximum array element needs updating. For instance, if the current maximum is 2 and the current element is 1 ($2 - 1 > 0$), indicating a positive difference, the current element becomes the new maximum. In such cases, where $N = 0$, the maximum value is updated. The output of this process is directed to the data memory.

→ Data Memory & Write back:

This stage handles memory access for read or write operations and finalizes the results of the executed instruction, writing back the computed values to the appropriate registers or memory locations.

Regarding the specific use of AND and OR gates for JUMP, BRZ, and BRN instructions, these gates likely play a role in determining the conditions under which the Program Counter should be updated after executing these instructions. For instance, the gates might evaluate certain flags or conditions (such as zero or negative results) to decide whether to alter the Program Counter's value.

This pipelined design aims to enhance CPU performance by dividing the instruction execution process into discrete stages, allowing multiple instructions to be processed simultaneously.

If N= 0, then the maximum value needs to be updated. So, we need to perform a memory write operation to update the current maximum. Last MUX, to select if we want to write back to the register for further execution or save it to the memory.

There could be a scenario when data hazards can occur where the current instruction depends on the o/p of the previous instruction. Solutions to overcome data hazards are:

1) *NOP operation (Bubble/ Stall)* - While executing the NOP operation, if one of the instructions needs to be executed first then the NOP operation stalls the pipeline for some clock cycles before executing the second instruction affecting the performance of the whole data path.

2) *Forwarding* - Forwarding helps to mitigate this performance degradation by placing the intermediate buffer in the pipeline stages so that even if 1st instruction is not completed, its o/p can be passed to the next one to maintain data consistency.

Control

The truth table of control lines is shown below:

Instruction	Opcode	Reg Wrt	Mem Wrt	Mem Read	ALUSrc1	ALUSrc2	BRN	BRZ	JUMP	JUMP M	ADD	SUB	NEG	Memto Reg
Add	0100	1	0	0	1	0	0	0	0	0	1	0	0	1
BRZ	1001	0	0	0	x	x	0	1	0	0	0	0	0	x
BRN	1011	0	0	0	x	x	1	0	0	0	0	0	0	x
Increment	0101	1	0	0	1	1	0	0	0	0	1	0	0	1
Jump	1000	0	0	0	x	x	0	0	1	0	0	0	0	x
JM	1010	0	0	1	x	x	1	0	0	1	0	0	0	x
Load	1100	1	0	1	x	x	0	0	0	0	0	0	0	0
Negate	0110	1	0	0	1	x	0	0	0	0	0	0	1	1
NOP	0000	0	0	0	x	x	0	0	0	0	0	0	0	x
Save PC	1111	1	0	0	0	1	0	0	0	0	1	0	0	1
Store	0011	0	1	0	x	x	0	0	0	0	0	0	0	x
Subtract	0111	1	0	0	1	0	0	0	0	0	0	1	0	1

According to the MAX function stated in the Project MAX document, we believe that the above control signals are valid.

1. **The truth table of 4 to 1 MUX:**

PC CONTROL 1	PC CONTROL 2	OUTPUT
0	0	PC+4
0	1	BRZ
1	0	BRN
1	1	J

2. **The truth table of MUX A:**

ALU Src	Forward A	OUTPUT
0	0	RT
0	1	WB DATA
1	0	IMM
1	1	WB DATA

3. **The truth table of MUX B:**

Save PC	Forward B	OUTPUT
0	0	RS
0	1	WB DATA
1	0	PC
1	1	WB DATA

Emulation

The SCU ISA instructions are able to realize all the basic instructions in RISC-V ISA. The emulation relationship is shown as follows :-

RISC -V	**SCU -ISA**
ADD rd, rs1, rs2	ADD rd, rs1,rs2
SUB rd, rs1, rs2	SUB rs, rs1,rs2
ADDI rd, ,rs1, imm	INC rd, rs1, imm
LW rd, (base_register) rs1	INC rs1, rs1, base_register LD rd, rs1
SW rs2, (base_register)) rs1	INC rs1, rs1, base_register ST rs2, rs1

BEQ rs1, rs2, label
(e.g., beq x3, x4, 100)
	SUB x2, x2, x2
	INC x2, x2, 100
	SUB x3, x3, x4
	BRZ x2

JAL rd, label (e.g jal x4, 100)	SVPC x4, 1 SUB x3, x3, x3 INC x3, x3, 100 J x3
JALR rd, offset(rs1) (e.g., jalr x3, 100(x4))	SVPC x3, 1 INC x4, 100 J x4

AND rd, rs1, rs2
(e.g., and x2, x3,x4)
	SUB x2, x2, x2
	SVPC x10, 16
	ADD x3, x3, x4
	BRZ x10
	INC x2, x3, -1
	NOP

OR rd, rs1, rs2
(e.g., or x2, x3,x4)
	SUB x2, x2, x2
	SVPC x10, 16
	ADD x3, x3, x4
	BRZ x10
	INC x2, x2, 1
	NOP

NOR rd, rs1, rs2
(e.g., nor x2, x3, x4)
	SUB x2, x2, x2
	INC x2, x2, 1
	SVPC x10, 16
	ADD x3, x3, x4
	BRZ x10
	INC x2, x2, -1
	NOP

ANDI rd, rs1, rs2 (e.g., andi x2, x3, imm)	SUB x2, x2, x2 SVPC x10, 16 INC x3, x3, imm BRZ x10 INC x2, x2, -1 NOP

Performance Analysis

The Datapath for our 5-stage pipelined CPU is shown below. We put the ALU and data memory in the same stage to boost performance even though instructions relating to the data memory in our SCU-ISA do not use an ALU and all data is moved straight from register files to the data memory.

Instruction Count:

To prevent hazards in the assembly code, we employ forwarding and flushing units. Thus, it is not necessary to include the NOP instruction in the code. After examining the assembly code we can say that 17 instructions are included in the loop. However, we are not considering the NOP instruction while calculating the number of instructions as NOP is written to terminate the assembly code.

Hence, the total instruction count will be

Instruction Count = 17n + 4

Cycle Time:

There are four stages in the pipeline, and we know that the delay times for the instruction memory, data memory, and ALU are each 3ns. Therefore, the **cycle time** should be set at **3ns** based on the pipeline.

CPI:

To eliminate the control hazard, we are using a flushing unit. Thus, it will consume 1 more cycle because of the BRN instruction.

CPI = Total number of clock cycles/ Instruction

count Now, CPI = (K + M + 1 - 1) / M

Where, K = number of stages

M = number of instructions

Hence,

CPI = (4 + 17n + 4 + 1 – 1) / (17n + 4)

= (8 + 17n) / (17n + 4)

For e.g., if we consider an array of 4 elements and we want to calculate the maximum number, then the loop will run for 4 times i.e. n = 4.

Therefore, CPI = (8 + 17*4 + 1 – 1) / (17*4 + 4)

CPI = 1.0555

Execution Time

Formula for calculating Execution time is,

Execution Time = (Instruction count) * (CPI) * (Clock Cycle Time)

= (4 + 17n) * [(8 + 17n)/(17n + 4)] * (3ns)

Now, for the example which we have considered earlier,

Execution Time = (4 + 17*4) * [(8 + 17*4)/(17*4 + 4)] * (3ns)

= 279ns