Kwame Nkrumah Univeristy of Science and Technology, Kumasi, Ghana.

COE 381: Microprocessors

Design and Simulation of a Multicycle 32 bit Beta Reduced Instruction Set CPU Using Logisim

Telecommunications Engineering 3

Group 1

October 2, 2022

Group Members

Name	**Index Number**	**Signature**
Njini Nathan Fofeyin	3574018	
Halida Abu Sufianu	3565218	
Fosu Dickson Marfo	3569318	
Oduro-Yeboah Yacoba	3571618	
Mensah Kwame Sompa	3570818	
Osei Bonsu Junior	3571918	
Asare Michael Ankomah	3567618	

Teamwork and Participation

The team met for 6 - 8 hours each week to complete the project. All members of the team were present during each session. The order of this document shows the chronology in which we incrementally built our processor. After discussing the general architecture, 2 members were assigned to build and debug a specific unit and debug. During group meetings, they explained the work to the entire group and improvements were made.

Meeting Days

Day	Period
Monday	15:00 - 21:00
Friday	10:00 - 15:30
Saturday	10:00 - 15:00 pm

Task Distribution for Implementation

Unit	Implementer
ALU	Osei Bonsu and Fosu Dickson
Shift Unit	Sompa Mensah
Multiplier Unit	Halida Abu Sufianu and Asare Michael
Register File	Yacoba and Njini Nathan
Datapath	Njini Nathan and Fosu Dickson
Control Unit	Njini Nathan and Asare Michael
Assembly and Test Run	Yacoba, Halida, Sompa Mensah

Contents

List of Tables

List of Figures

The Beta (β) Instruction Set Architecture

1 Introduction

An ISA describes the design of a Computer in terms of the basic operations it must support. The Beta is an example of a reduced-instruction-set computer (RISC) architecture. Reduced refers to the fact that in the Beta ISA, most instructions only access the internal registers for their operands and destination. Memory values are loaded and stored using separate memory-access instructions such as load and store. Therefore, RISC architectures lead to simple, high-performance hardware implementations. On the software side, BETA has a simple assembler since it is a RISC architecture.

The Beta is a general-purpose 32-bit architecture, that is designed to have separate registers and main memory units. All data path registers are 32 bits wide and when loaded with an address, can point to any location in the byte addressed memory. When read, register 31 is always 0, when written, the new value is discarded. The program counter is a special-purpose register, which holds the address of the memory location of the next instruction to be executed.

2 Beta and the RISC Architecture

Beta Instructions types

The Beta has three types of instructions:

- **Arithmetic and Logical**: operations compute instructions that perform operations on register values
- **Loads and Stores**: instructions that access values in main memory
- **Branches and Jumps**: instructions that conditionally change the value of the program counter

Beta Instruction Formats There are two instruction formats.

- **Without Literal** instructions specify an opcode, two source registers and a destination register.

- **With Literal** instructions specify 1 source register with a 32-bit constant, derived by sign-extending a 16-bit constant stored in the instruction itself.

Building Blocks

1 Arithmetic Logic Unit

We designed the Arithmetic and Logical Unit (ALU) to execute arithmetic and logical operations such as addition, subtraction and comparisons. Noteworthy is that other Logic and arithmetic operations such as multiplication, division, and shift operations are implemented using separate computation units. The logic unit on the other hand performs logical operations. We did not include division and multiplication operations in our ALU design for simplicity.

1.1 1-Bit ALU, MSB and LSB

Least Significant Bit ALU

This section performs, perhaps the most important computations in the ALU. For Addition and subtraction operations, the carry-in bit is received by the LSB Adder. For comparison instructions, the computations peformed by this unit as deplayed in Figure 2.1 are used to determine if the operands are less or equal to each other.

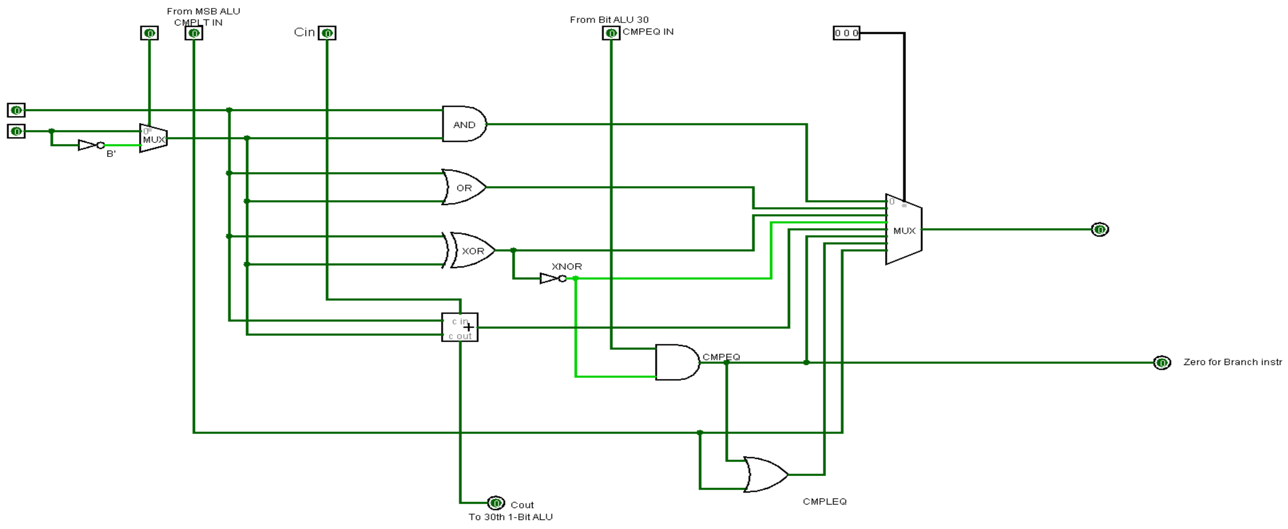


Figure 2.1: LSB ALU

Most Significant Bit ALU

The most significant bit performs the final computation to obtain the 32-bit output. The carry out from the adder which may be from add and subtract operations is discarded.

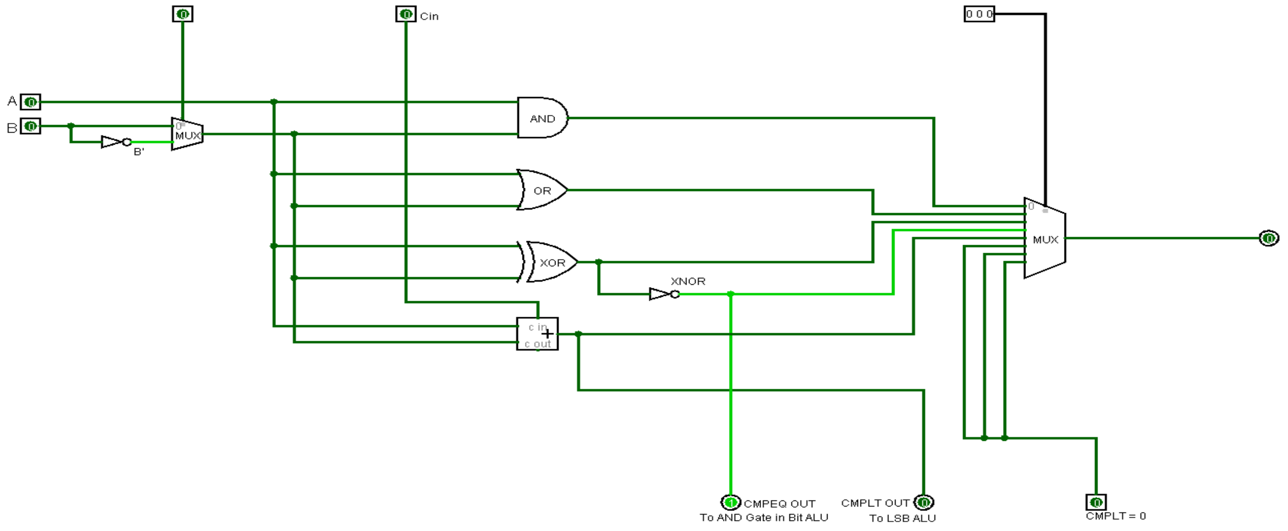


Figure 2.2: MSB ALU

1-Bit ALU

Each 1-bit ALU has two 1-bit inputs labelled A and B upon which it does arithmetic and logical computations, again, the comparison outputs for these bits are set to 0 as only the LSB result is relevant.

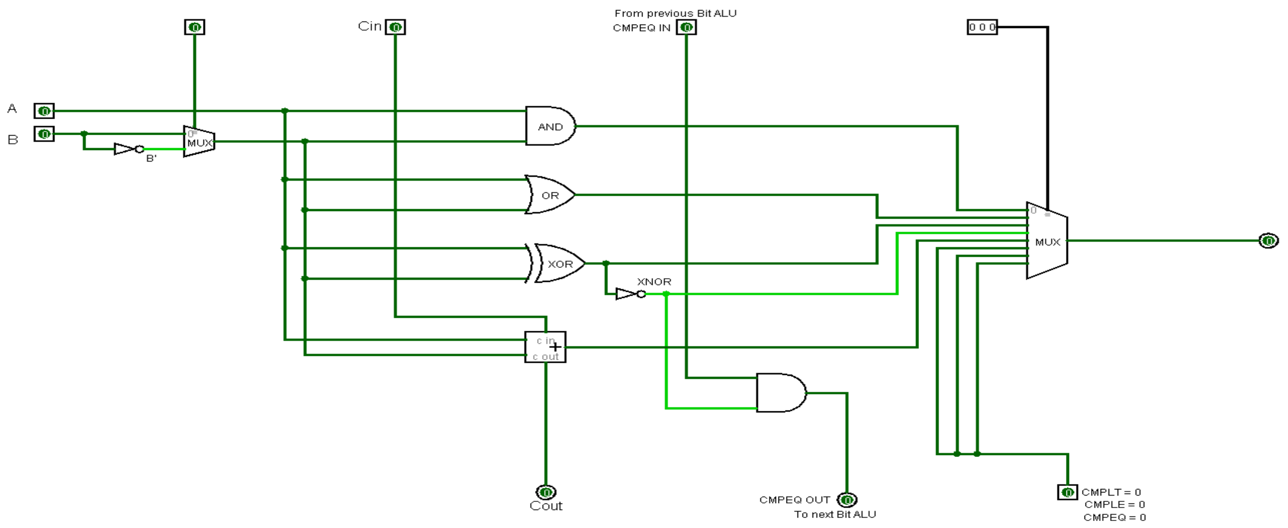
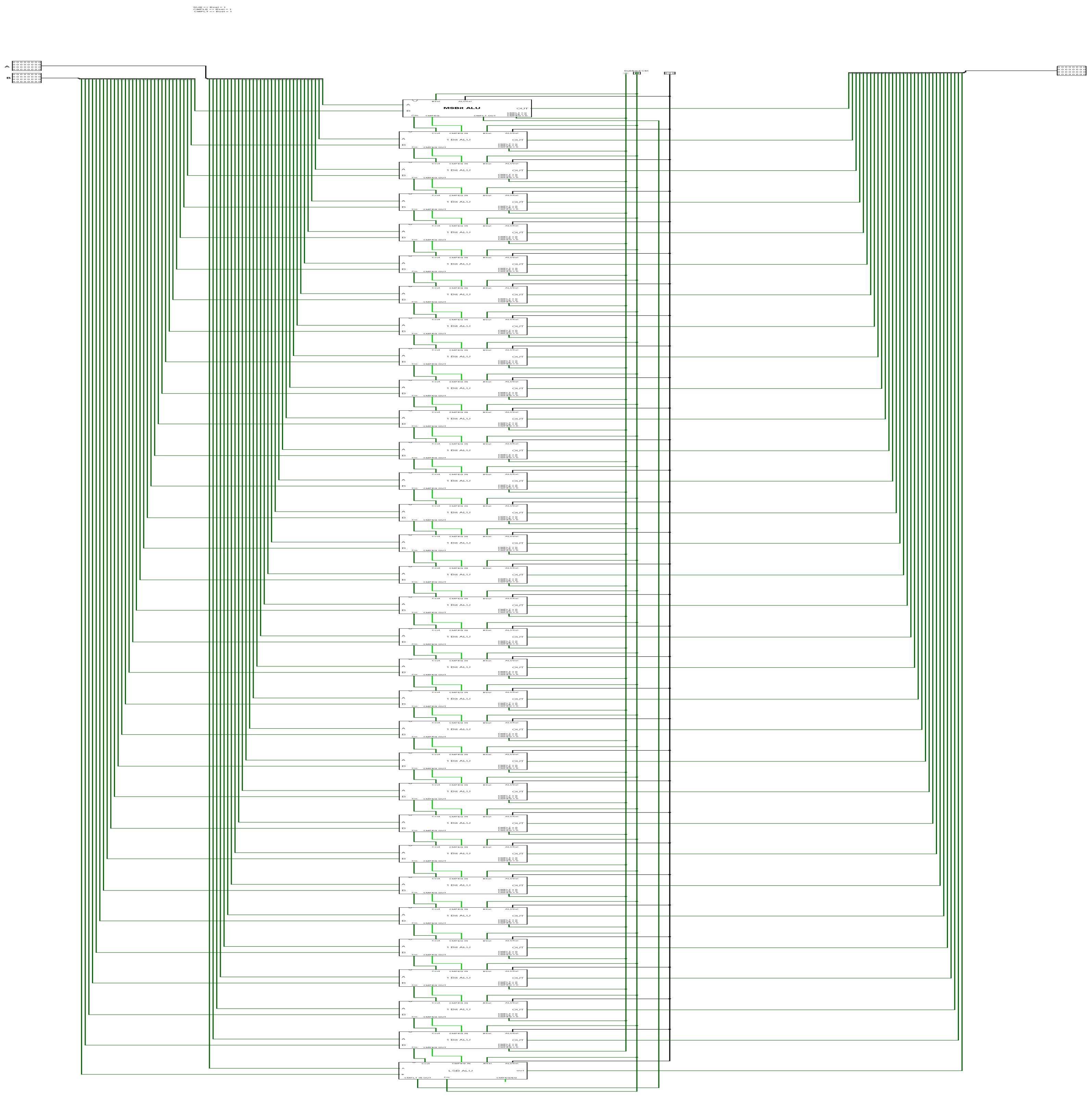


Figure 2.3: 1-Bit ALU

1.2 Final 32-Bit ALU

The full 32-bit ALU is created by connecting adjacent ALU blocks. For addition, subtraction and less than (or equal) instructions, the computation is performed from the LSB ALU. The carry-out from the LSB ripples through the 30, 1-bit adders and is used to obtain the 31st bit of the result in the MSB ALU. We implemented the ripple carry-out adder system by directly linking the carries of the various 1-bit adders. The complete working 32-bit ALU is shown in figure 2.4. [1] ALU operations such as ADD, SUB, AND, OR, XOR, XNOR can be done by assigning its corresponding ALU control. A more complete documentation of the ALU and other β instructions can be found on the Beta documentation site (computationstructures.org)

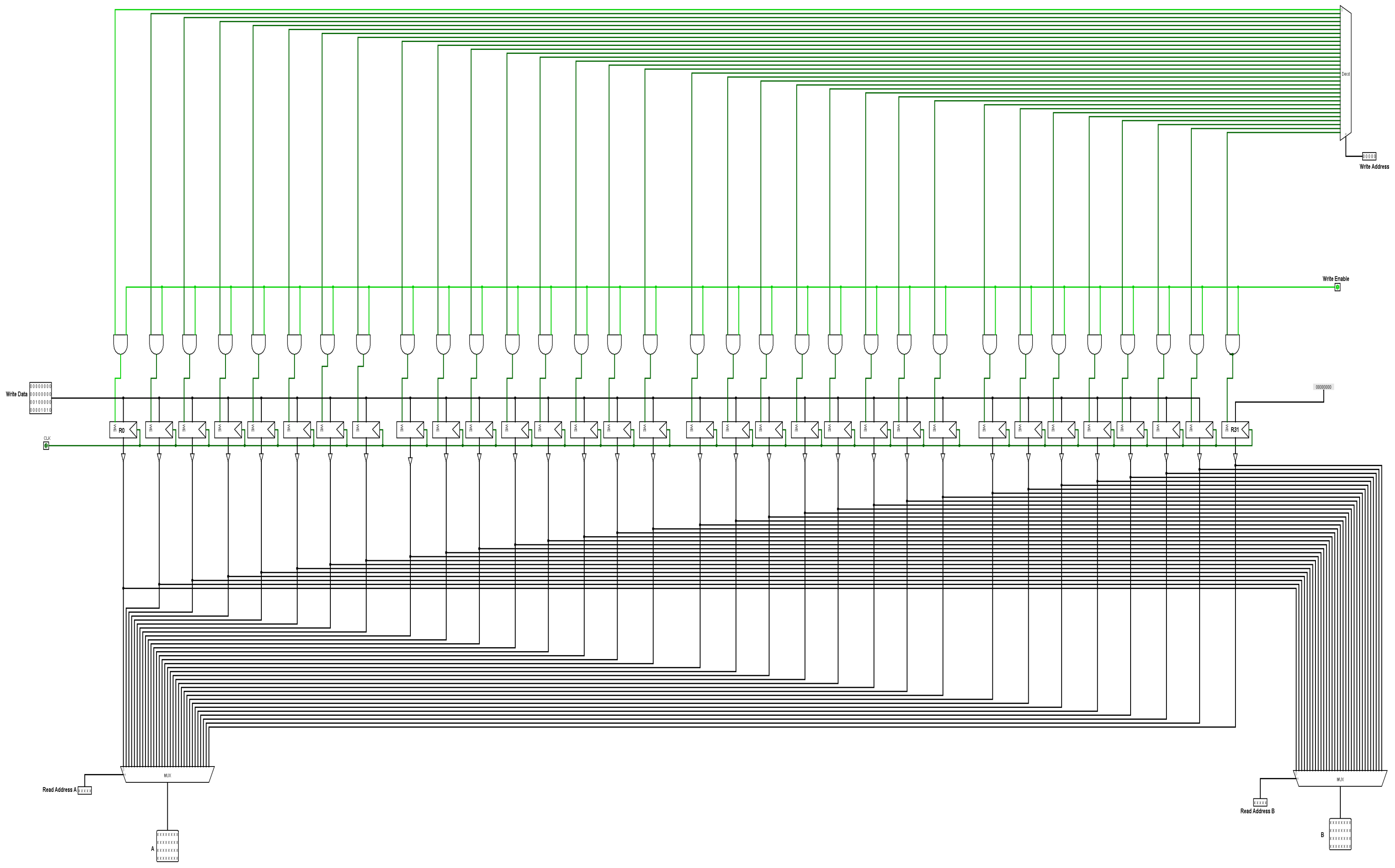
[1] you can zoom out to have a better visual of the attached 32-bit ALU and other attached images.


SUB => Bsel = 1
CMPLE => Bsel = 1
CMPLT => Bsel = 1
A
B
Subtract Ctrl
MSBit ALU
BSel
ALUSel
OUT
Cin
CMPEQ
CMPLT OUT
CMPLT = 0
CMPLE = 0
CMPEQ = 0
1 Bit ALU
Cout
CMPEQ IN
CMPEQ OUT
LSB ALU
out

2 Register File

Figure 2.5 displays our register file. The 5-bit decoder input (upper right) is the Write Address. The output of the decoder is a 1 bit enable signal for the register that the data is to be stored in. The 5-bit multiplexer select for A and B chooses the read address of the 2 registers. On a rising clock edge, the values of these registers are read as operands for the Computation units.

In addition, we use buffers to efficiently distribute the register values to the multiplexer gates. This design concept is espcially useful for the real-world where wiring may limit efficient voltage distribution.



3 Instruction and Data Memories

The instruction and Data memories are built from Logisim's inbuilt ROM and RAM models respectively. The Instruction memory (ROM) has 3 ports: the 24-bit address input, 32-bit Data Output and the constant device enable signal, sel. The address input identifies the location of the 32-bit word in memory. The β Instruction set defines 2^{32} byte memory. Due to Logisim's limitation to 24-bit addressing, our memory units are designed to be 2^{24} byte memory.

The Data memory unit is similar to the Instruction memory. However, Data has to be both stored and written to this unit. It is designed to have the Load and Write Control Signals that control the writing and reading data to the unit. We limit the abstraction of the memory design to inbuild Logisim memories since Memory Design is a separate design project on its own.

4 Shift Unit

The shift unit performs the Shift logical/arithmetic left, right and arithmetic right instructions. The instruction performed is determined by the SHIFTCtrl select input to the multiplexer.

The datapath uses the Shift unit in calculating the effective address of certain instructions such as LDR. It multiplies the Sign extended literal (SEXT(Literal)) by 4 through a Shift Left by 2 operation.

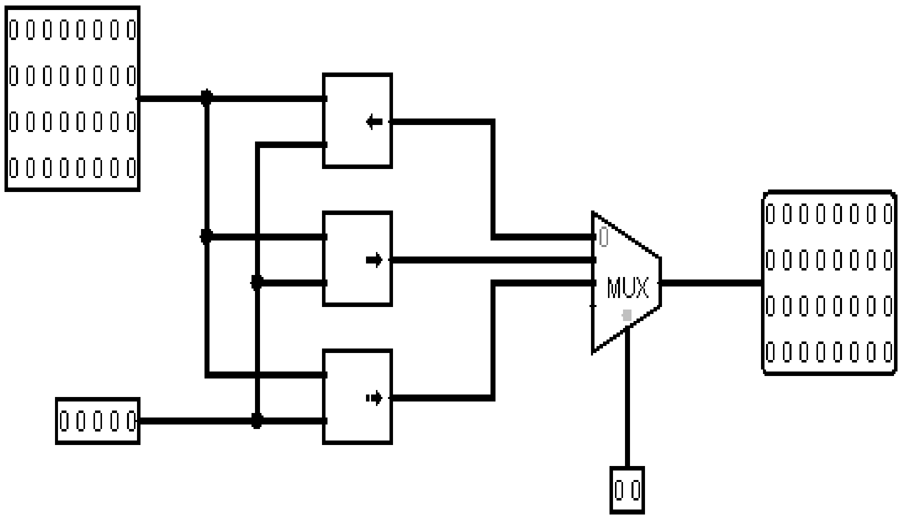


Figure 2.6: Multiplier/Divider Unit

5 Multiplier

Since multiplication and division are expensive operations, we take advantage of the 1-bit control signal to route input to only 1 of the 2 computation units. The inverter and control buffers are used to accomplish this task as seen in Figure 2.7. This prevents unnecessary computation before selecting either the multiplier or the divider output.

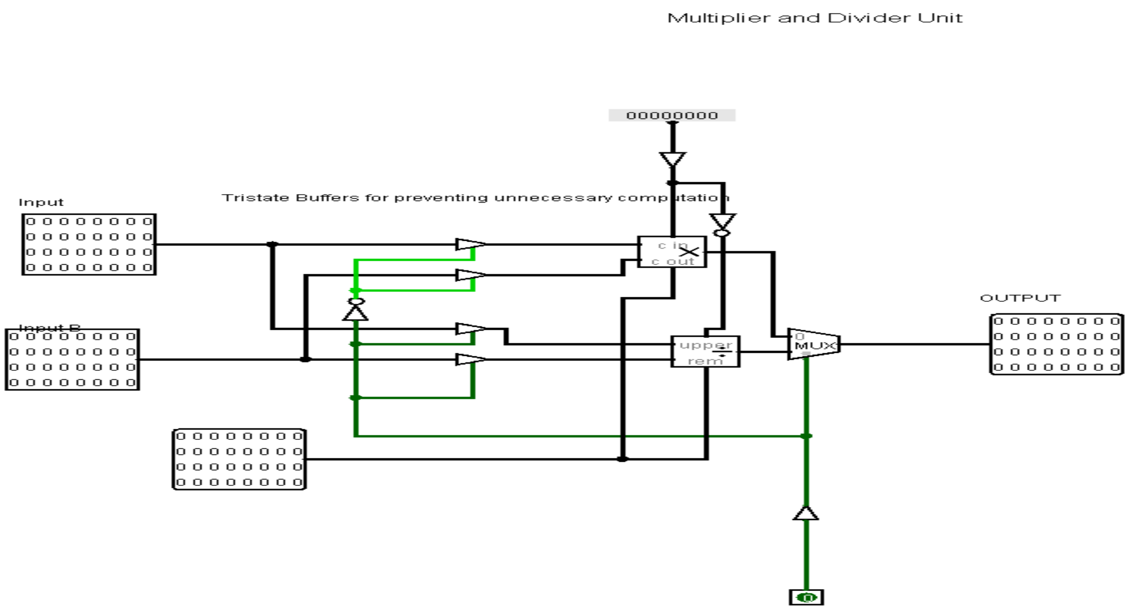

Multiplier and Divider Unit
00000000
Input
Tristate Buffers for preventing unnecessary computation
c in
c out
Input B
upper
rem
MUX
OUTPUT

Figure 2.7: Multiplier/Divider Unit

Multicycle Datapath

1 Introduction

Several building blocks as seen in the previous sections make up our design and are used in the Multicycle datapath. All these units and the datapath are based on the β instruction specifications. In this section, we limit our description of the instructions and focus more on the datapath design and processes. To get a complete understanding of the instruction set, please refer the β documentation for more details.

The Program Counter

The program counter contains the address of the instruction being executed at the current time. The PC+4 value at the start of every instruction is performed by the ALU. The datapath as seen in figure 3.8 has a 1-bit select multiplexer and 3-bit multiplexers which select the PC and the 4 value respectively. The ALU Addition operation is activated during the 1st clock cycle of instruction execution.This enables the calculation of the PC+4 value possible – which is routed through another multiplexer to the Program Counter to be used for the next instruction.

During normal execution, the PC value will be incremented by 4 on the 1st clock cycle of execution. This normal PC increment is altered for the Branch (BEQ and BNE) and Jump (JMP) instructions. Here, the multiplexers select the Sign extended literal, shift it left by 2 (multiplication by 4) in the 1st clock cycle. In the second clock cycle, the obtained 4*SEXT(Literal) is added to the computed PC+4 to obtain a new effective address (EA) which is stored in a non-architectural register. Please note for the JMP instruction, EA is calculated differently. In the next clock cycle, this value is written into the PC to obtain the address of the next instruction. value for the using the for incrementing the PC value.

ALU, and Memory Address Datapath

The β has 32 bit registers that hold values for use by the datapath. There are branches and jumps whose execution may change the program counter and hence the address of the next instruction to be executed. The first class of ALU, Shift and Multiplier Unit instructions perform an operation on two register operands Ra and Rb. Result is stored in a specified destination register Rc, which supplies the address for the write port. The outputs of the read data ports are directed to the inputs of the ALU to serve as its two input operands. The ALUCtrl (determined by the 6-bit opcode in the control unit) then determines which operation is to be performed. The output of the ALU is then conveyed through another non-architectural register to the Write Data port of the register File through a multiplexer. The shift and multiplication/division instructions are performed in a similar manner but using their respective units. We can therefore highlight the instruction setps as shown below:

• **Fetch** – Instruction is read from the instruction memory using program counter and stored in a non-architectural register.

• **Decode** – The opcode field is used to determine the values for the datapath control signals. The opcode is decoded by the control unit while the Register Fields of the instructions are read and decoded into the register file. During the decode phase, the ALU is used to compute the PC+4 value.

• **Read** – Contents of the register in the Ra and Rb fields are read from the register file and also stored in a non-architectural register to be read-out in the next clock cycle.

• **Execute** – the required computation is performed on the two operand values using either the ALU, Shift Unit or Multiplier depending on the instruction.

• In the **write-back** stage, the result of the operation is written to the Rc field of the register file. ST, LD and LDR are the instructions that access both memory and the register file while every other instruction access the register file.

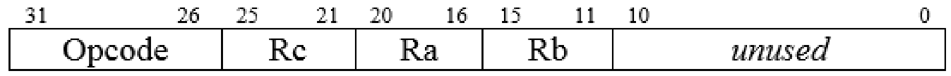
2 Datapath for Without Literal Instructions

2.1 Logic and Arithmetic Instructions

Only arithmetic and logical instructions have instruction formats without literal. After instruction fetch, Ra and Rb are conveyed from the read ports of the Register file and fed into the ALU as inputs through the non-architectural registers and multiplexers. The control logic then works on the opcode to select the appropriate ALU, Shift or Multiplier Control Signals. The result is then sent back to the Register File to be written into the Rc register. Write EN via control unit is set to 1 to enable the option to write.

The β ALU instructions(without literal) have 4 instruction fields. There's a 6-bit field specifying the ALU operation to be performed. This field is known as the opcode. The two source operands come from registers whose numbers are specified by the 5-bit "Ra" and "Rb" fields. So we can specify any register from R0 to R31 as a source operand. However, software limits the use of Registers 27 to 30 for stack pointing, exception pointing and more. We will not concern ourselves with these as our focus is on hardware. The destination register is specified by the 5-bit "Rc" field and the remaining 10 bits are unused. Table 3.3 specifies the field use as explained.

Table 3.3: Without Literal Instructions

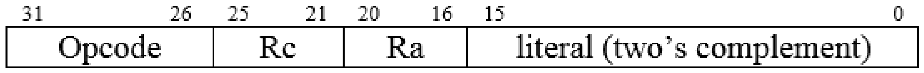


31 26	25 21	20 16	15 11	10 0
Opcode	Rc	Ra	Rb	*unused*

3 Datapath for Literal Instructions

Some β instructions may a 16-bit 2's complement constant (literal) in the 15:0 bits. This constant may be used as a second operand in Effective Address (EA) calculation or in regular ALU, Shift and multiplication computations. Table 3.4 specifies the instruction field utilization. Instructions such as Branch (BEQ, BNE), Jump (JMP), Load (LD and LDR) all follow this specification. Jump has a specified literal (0×0 in Hex)

Table 3.4: With Literal Instructions



31 26	25 21	20 16	15 0
Opcode	Rc	Ra	literal (two's complement)

3.1 Logic and Arithmetic Instructions with constants

The ALU, Shift and Multiplier instructions may also use a constant as the second operand. This is done by adding a multiplexer to the ALU datapath. When its ALUOpa control signal is set to 0, ALU takes its input operand from the register file. When set to 1, a sign extended constant (32-bit operand formed by sign-extending the 16-bit 2's complement constant stored in the literal field of the ALU instruction) is selected as the input operand.

3.2 JMP, LD and ST Instructions

Load and Store

The Load and Store instructions access the main memory. The main memory has 3 ports; 2 read ports for instruction fetch and 1 write port, used by the store instruction to write data into the main memory. For address calculation, the contents of the Ra are added to the sign-extended 16-bit literal.

For load instructions, the output of the ALU is conveyed to the main memory as the address we wish to access. The contents of the addressed location are returned by the memory and conveyed back to the register file, but the Rc instruction field is not connected to a register file read address. A multiplexer is used to enable the Rc field to be selected as the address for the register file's second read port

The memory has 2 control signals; LE (LoadDataOutMemEN) and WE (Write-DataInMemEN). LE is set to 1 when a value is to be read from the memory. WE is set to 1 when main memory needs to store the value on its WD (Write Data) port into the addressed memory location. A multiplexer is used to select which value to write back to the Register File: the output of the ALU or the data returning from the main memory.

To access memory, the CPU has to generate an address. Load and Store compute the address by adding the sign-extended constant to the contents of register Ra.

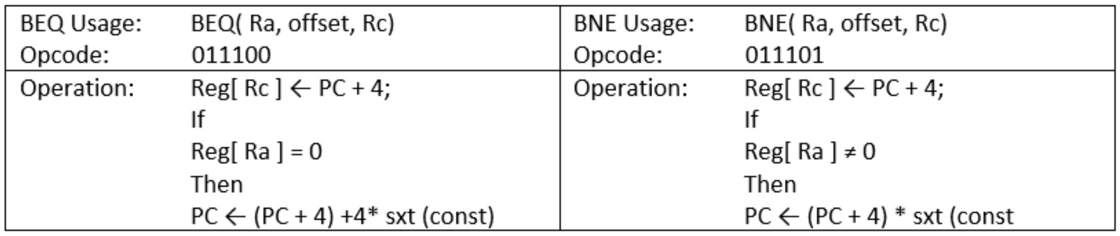
Jump

The jump instruction uses the Logial AND operation on the Ra and 0×FFFFFFFC values to calculate the effective address, EA. The multiplexers select these inputs to be loaded in the 2nd cycle of instruction execution. The value in the Ra register is taken to the next PC value. When PCEN is set to 0, the incremented PC value is chosen and read into the Program counter. When set to 1, the value of the Ra register is chosen as is the case with this JMP instruction.

3.3 BNE and BEQ Instructions

Branch instructions use the instruction format with the 16-bit 2's complement constant. The operations can be summarized with the Table 3.5:

Table 3.5: BEQ and BNE Instructions



BEQ Usage: Opcode:	BEQ(Ra, offset, Rc) 011100	BNE Usage: Opcode:	BNE(Ra, offset, Rc) 011101
Operation:	Reg[Rc] $\leftarrow$ PC + 4; If Reg[Ra] = 0 Then PC $\leftarrow$ (PC + 4) +4* sxt (const)	Operation:	Reg[Rc] $\leftarrow$ PC + 4; If Reg[Ra] $\neq$ 0 Then PC $\leftarrow$ (PC + 4) * sxt (const

BEQ

To perform the BEQ instruction, the PC+4 value is first calculated in the 1st clock cycle. In the second clock cycle, the Effective address is calculated since the literal has to be sign extended and added to PC. This gives us the address of the instruction following the BEQ. The value PC value is written to the 'Rc' register whether the branch is taken or not. On the other hand, if the value in Ra is zero, the EA is written to the program counter (Branching).

BEQ tests the value of the Ra register to see if it's equal to 0. If it is equal to 0, the branch is taken and the PC is incremented by the amount specified in the constant field of the instruction. The constant (called an 'offset' since we are using it to offset the PC) is treated as a word offset and is multiplied by 4 to convert it to a byte offset since the PC uses byte addressing. If the contents of the Ra = 0, the EA value is written to the Program Counter.

BNE

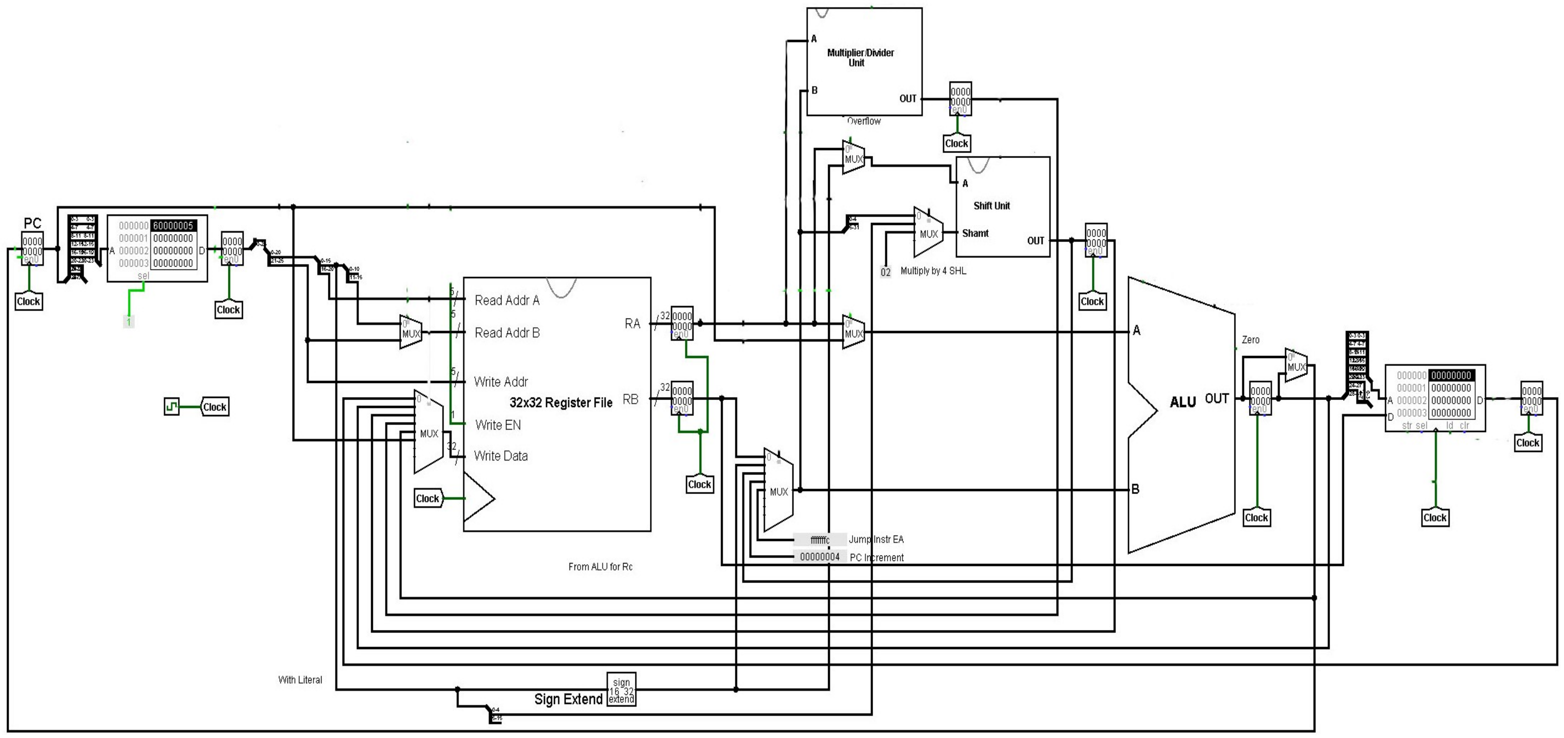
BNE instruction is similar to BEQ in its operation; the condition for a branch is just reversed in this case. Here, the branch is taken if the value in register Ra $\neq 0$.

3.4 LD and LDR Instructions

The Load-relative (LDR) instruction behaves like a normal LD instruction, except that the memory address is taken from the branch. Some addresses are too large to fit into the 16-bit literal field. These large addresses are stored in the main memory and directly accessed for use by the LD and ST. We included the LDR instruction in order to access large constants that have to be stored in main memory because they are too large to fit into the 16-bit literal field of an instruction.

3.5 Complete Datapath

Figure 3.8 describes the complete datapath for all the instructions described in this chapter. The describe



Fig

4 Non-architectural registers

Our multicycle datapath is designed to complete instructions within 3-4 clock cycles. Registers are used in the Fetch, Decode, execute and write-back stages steps to temporarily store data and address values till the next rising edge of the clock. Non-architectural registers are normal 32 bit registers that are crucial to the multicycle execution.

Multicycle Control

1 Introduction

To understand this section, the reader should be versed in the required datapath signals highlighted in the previous section. Multicycle datapath uses non-architectural registers, ALU, Multiplier and Shift, Multiplexers, the Register File and Memory units. These building blocks require: enable, select, load and store signals which control instruction execution on each clock cycle. Due to several clock cycles, the control unit for our CPU is a Finite State Machine (FSM) which requires both sequential and combinational logic. The control signals are fully described in Figure 4.9. The signals and buses are labelled to be self-explanatory. Working of the control and its interaction with the datapath can be referenced in Figure 5.11. Please note that the branch is an input to the PLA. Its value is asserted using the NOT Gate for the BEQ instruction.

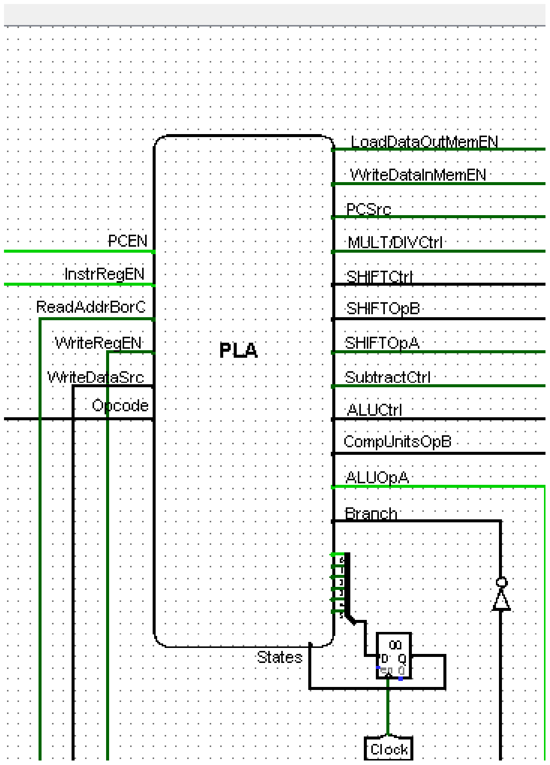


Figure 4.9: Control Unit

The sequential logic encodes the state of execution which in turn activates the correct control signals for that state. There are 39 states (0 to 38). The number of states were obtained by labelling the Finite State Machine. For this reason, we use 6 bits to encode the states (000000 for State0 to 100110 for State38). As shown in Figure 4.9, a 6-bit register is used to store these state values for the next execution step. The register uses the system clock and on every rising clock edge, it gives the control's combinational cicruit the state needed to execute the next step of the instruction.

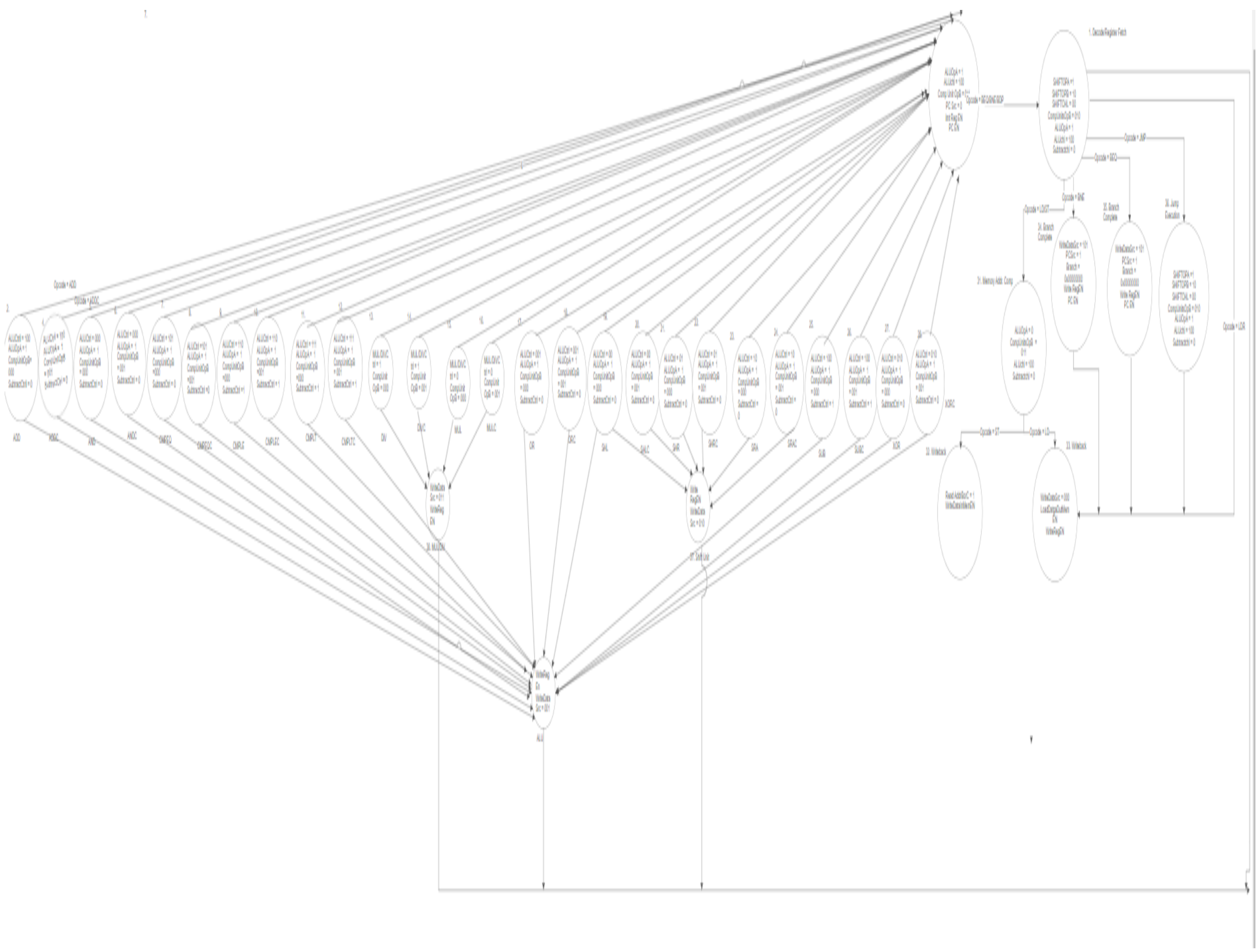
Instructions execute within 3-5 states. Branch, Jump and Load Relative Instructions are executed in 3 clock cycles, they use only 3 states. The ALU, Shift and Multiplication/Division instructions execute in 4 clock cycles – making use of 4 states. Similarly, the Load and Store Instructions execute in 4 states.

For the combinational Logic, we use a Programmable Logic Array (PLA) designed to specifically support the instruction execution by the datapath. The inputs of the PLA are the Instruction Opcode and the State Encodings, each 6 bits.

2 The Programmable Logic Array (PLA)

We first obtain the FSM graph for the Programmable Logic Array as shown in Figure 2. From the FSM, we create the Logic equations table by using active states for the output. [2]

[2]Due to the compact nature of the Graph, legibility of the state outputs is hard. Please zoom out on a omputer with a good GPU to get a good visual. This goes same for the other large images attached in this report.



It is directly intuitive from Table 4.6 to design the PLA using an Array of AND gates to decode the States and Opcode; and to use OR Gates to perform the logical add of the States and Opcode to obtain the Control signals and the next state (of instruction execution) outputs. During the design, we minimized logic by combining repeated logical sums from other outputs to obtain outputs with same logic.

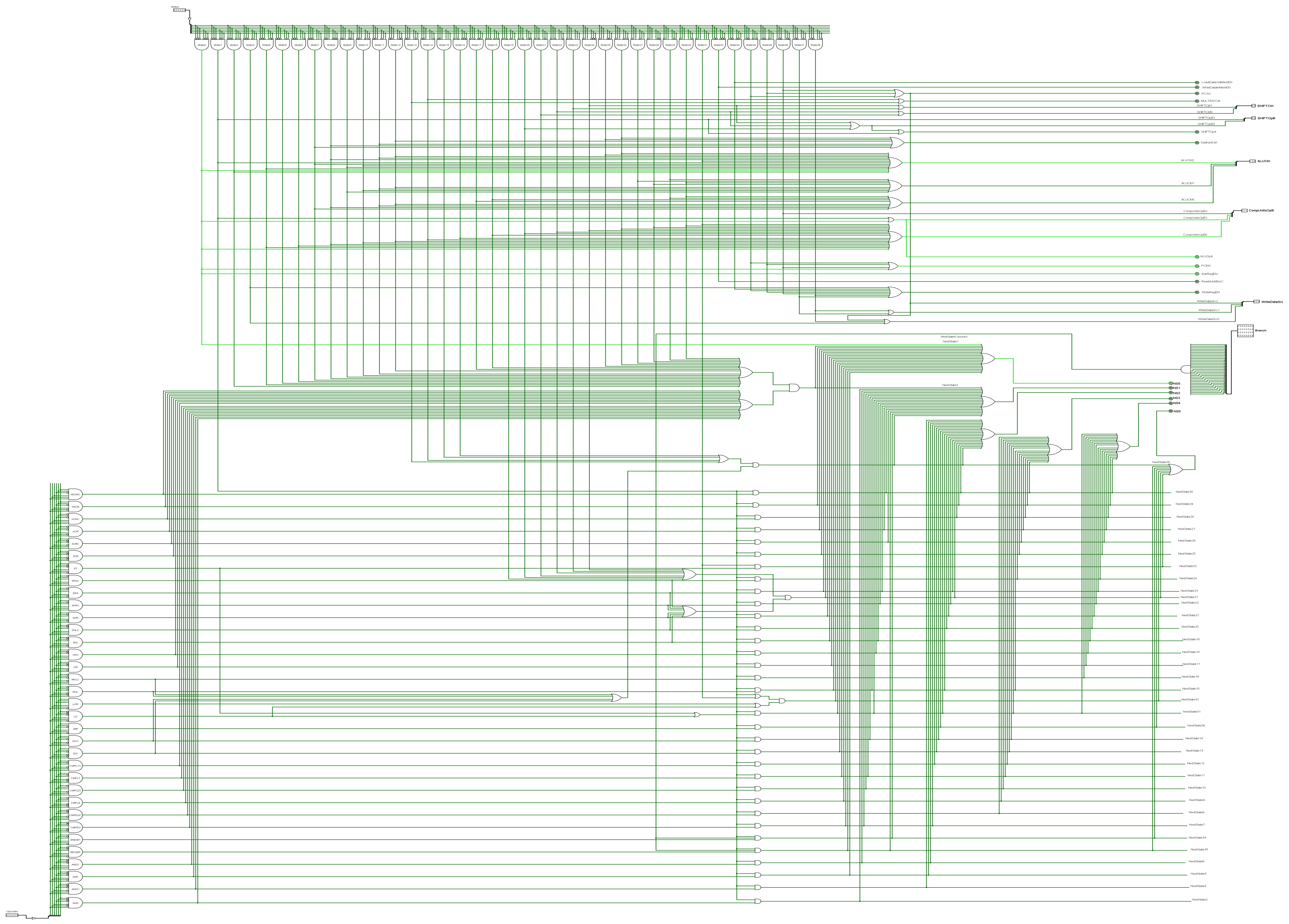
Table 4.6: Logic Equations Table

Output	**Current states**	**Opcode**
LoadDataOutMemEN	State33	
WriteDataInMemEN	State32	
PCSrc	State34 + State35 + State36	
MULT/DIVCtrl	State13 + State14	
SHIFTCtrl1	State23 + State24	
SHIFTCtrl0	State21 + State22	

SHIFTOpB1	State1	
SHIFTOpB0	State20 + State22 + State24	
SHIFTOpA	State1 + State20 + State22 + State24	
SubtractCtrl	State9 + State10 + State11 + State12 + State25 + State26	
ALUCtrl2	State0 + State1 + State2 + State4 + State7 + State8 + State9 + State10 + State11 + State12 + State25 + State26 + State31	
ALUCtrl1	State9 + State10 + State11 + State12 + State27 + State28 + State29 + State30	
ALUCtrl0	State17 + State18 + State29 + State30 + State7 + State8 + State11 + State12	
CompUnitsOpB2	State36	
CompUnitsOpB1	State0 + State1	
CompUnitsOpB0	State0+ State4 + State6 + State8 + State10 + State12 + State14 + State16 + State18 + State20 + State22 + State24 + State26 + State28 + State30 + State31	
ALUOpA	State0 + State1	
PCEN	State0 + State34 + State35 + State36	
InstrRegEN	State0	
ReadAddrBorC	State32	
WriteRegEN	State3 + State33 + State34 + State35 + State36 + State37 + State38	
WriteDataSrc2	State34 + State35 + State36	
WriteDataSrc1	State37 + State38	
WriteDataSrc0	State3 + State34 + State35 + State36 + State38	
NextState0	State37 + State3 + State38 + State32 + State33 + State34 + State35 + State36	
NextState1	State0	
NextState2	State1	ADD
NextState3	State2 + State4 + State5 + State6 + State7 + State8 + State9 + State10 + State11 + State12 + State17 + State18 + State25 + State26 + State27 + State28 + State29 + State30	ADD + ADDC + AND + ANDC + CMPEQ + CMPEQC + CMPLE + CMPLEC + CMPLT + CMPLTC + OR + ORC + SUB + SUBC + XOR + XORC + XNOR + XNORC

NextState4	State1	ADDC
NextState5	State1	AND
NextState6	State1	ANDC
NextState7	State1	CMPEQ
NextState8	State1	CMPEQ
NextState9	State1	CMPLE
NextState10	State1	CMPLEC
NextState11	State1	CMPLT
NextState12	State1	CMPLTC
NextState13	State1	DIV
NextState14	State1	DIVC
NextState15	State1	MUL
NextState16	State1	MULC
NextState17	State1	OR
NextState18	State1	ORC
NextState19	State1	SHL
NextState20	State1	SHLC
NextState21	State1	SHR
NextState22	State1	SHRC
NextState23	State1	SRA
NextState24	State1	SRAC
NextState25	State1	SUB
NextState26	State1	SUBC
NextState27	State1	XOR
NextState28	State1	XORC
NextState29	State1	XNOR
NextState30	State1	XNORC
NextState31	State1	LD or ST
NextState32	State31	ST
NextState33	State1 + State31	LD or LDR
NextState34	State1	BNE and Branch
NextState35	State1	BEQ and Branch
NextState36	State1	JMP
NextState37	State19 + State20 + State21 + State22 + State23 + State24	SHL + SHLC + SHR + SHRC + SRA + SRAC
NextState38	State13 + State14 + State15 + State16	DIV + DIVC + MUL + MULC

From the obtained logic equations table, we design the Programmable Logic Controller as show in Figure 4.10



Figu

Final 32 Bit CPU

1 Beta(β) Assembly and CPU Testing

In our design, we used the online BSim tool to assemble the instructions implemented. The assembly instructions were loaded into our Instruction Memory in Logisim. On simulation, the expected results were obtained. Similarly, the input data was also loaded into the Data Memory.

Beta Assembly Code

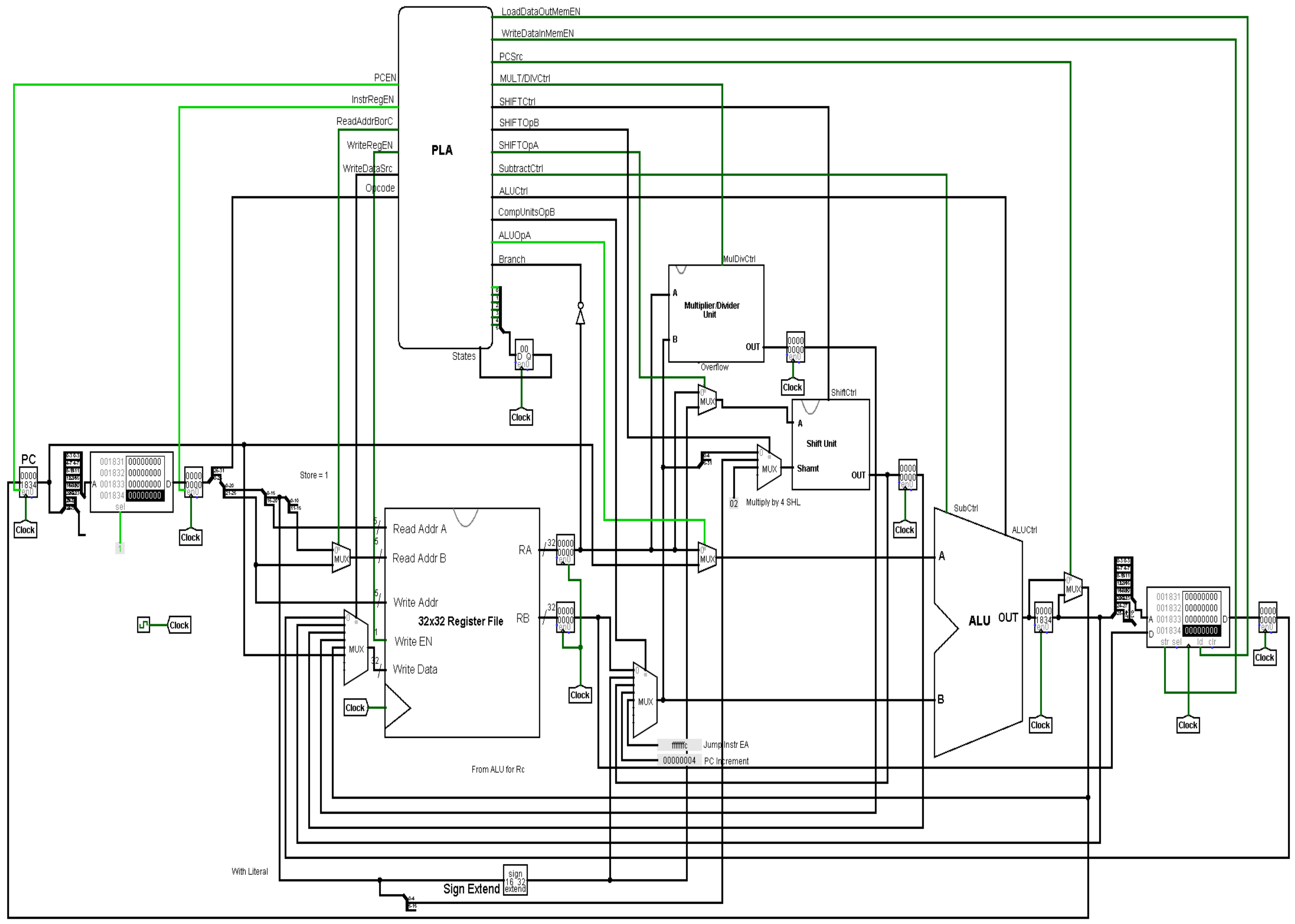
```
SUB(R0,R0,R0); //set Reg[R0] ← 0, use as base
LD(R0,5,R1); //Reg[R1] ← Mem[0] (= 1)
LD(R0,4,R2); //Reg[R2] ← Mem[4] (= A)
LD(R0,8,R3); //Reg[R3] ← Mem[8] (= B)
SUB(R4,R4,R4); //Reg[R4] ← 0, running total
ADD(R4,R2,R4); //Reg[R4] += A
CMPLT(R2,R3,R5); //Reg[R5] ← A<B
BEQ(R5,2,R0); // if Reg[R5] = 0, go forward 2 instructions
ADD(R2,R1,R2); //A++
BEQ(R0,-5,R0); // go back 5 instructions
ST(R4,0,R0); //Mem[0] ← Reg[R4]
BEQ(R0,-1,R0); //program is over, keep looping back to here
```

Machine Language representation

```
84000000 60200000 60400004 60600008 84842000 80841000 94a21800 7005fff8
80420800 7000fff4 64800000 7000fff3
```

Test Input

```
00000001 00000001 0000000a
```



LoadDataOutMemEN
WriteDataInMemEN
PCSrc
MULT/DIVCtrl
SHIFTCtrl
SHIFTOpB
SHIFTOpA
SubtractCtrl
ALUCtrl
CompUnitsOpB
ALUOpA
Branch
PCEN
InstrRegEN
ReadAddrBorC
WriteRegEN
WriteDataSrc
Opcode
PLA
States
PC
Store = 1
Read Addr A
Read Addr B
Write Addr
Write EN
Write Data
32x32 Register File
RA
RB
Multiplier/Divider Unit
MulDivCtrl
Overflow
Shift Unit
ShiftCtrl
Shamt
Multiply by 4 SHL
SubCtrl
ALUCtrl
ALU
OUT
Jump Instr EA
PC Increment
From ALU for Rc
With Literal
Sign Extend
Clock
MUX

2 Performance Measurements

In the design process, we minimized additional hardware by utilizing the ALU for both address and instruction computations in different cycles without using any additional hardware. This makes the processor faster and more energy-efficient in the real-world.

A major advantage of our CPU is that it can use very high frequency clock speeds and does not limit smaller and faster instructions to execute in the time limitations set by the slowest instructions - this is the case with single cycel CPUs.

In addition, the design of our datapath and separation of the Memory Units (Harvard Architecture) removes the Instruction/Data bottleneck making the Processor faster.

Conclusion

This was a very challenging project and the team was dedicated to designing every unit, every step of the way. We would be delighted to demonstrate our work during the presentation. In further work, we can work on pipelining the processor and adding more instructions to support Digital Signal Processing. Initially, our plan was to later optimize the datapath to a digital signal processor by adopting the Motorola DSP Architecture to perform multiply-accumulate and rounding instructions. This explains the structure of our datapath with separate Multiplier, Shift and ALU Units. It is our sincere hope that the effort and hardwork of the team is returned with good scores.