CO224 – Computer Architecture

Lab 5 – Building a Simple Processor

Part 5 – Extended ISA – Report

Group 40 – ADEESHA W.L.T.(E/21/017) EPITAKADUWA E.K.G.D(E/21/126)

In this, the CPU was enhanced by introducing **five new instructions** to extend its functionality. These additions required both **hardware modifications** to the CPU and **updates to the assembler** to ensure correct instruction encoding and execution. The added instructions provide support for multiplication, bitwise shifting, rotation, and conditional branching. The details of the newly supported instructions are as follows:

1.bne <offset><r1><r2>
Compares the contents of r1 and r2. If they are **not equal**, the program counter is updated to branch by the given offset.

2.**mult <r1> <r2> <r3>**
Performs multiplication of the values in registers r2 and r3, storing the result in register r1.

3.**sll <r1> <r2> <offset>**
Performs a logical left shift of the value in r2 by the given offset, storing the result in r1.

4.**srl <r1> <r2> <offset>**
Performs a logical right shift on the value in r2 by the offset amount, and stores the result in r1.

5.**sra <r1> <r2> <offset>**
Performs an arithmetic right shift on r2 (preserving the sign bit) by the offset, storing the result in r1.

6.**ror <r1> <r2> <offset>**
Rotates the bits in r2 to the right by the given offset and stores the rotated value in r1.

Hardware Changes

To support these new operations:

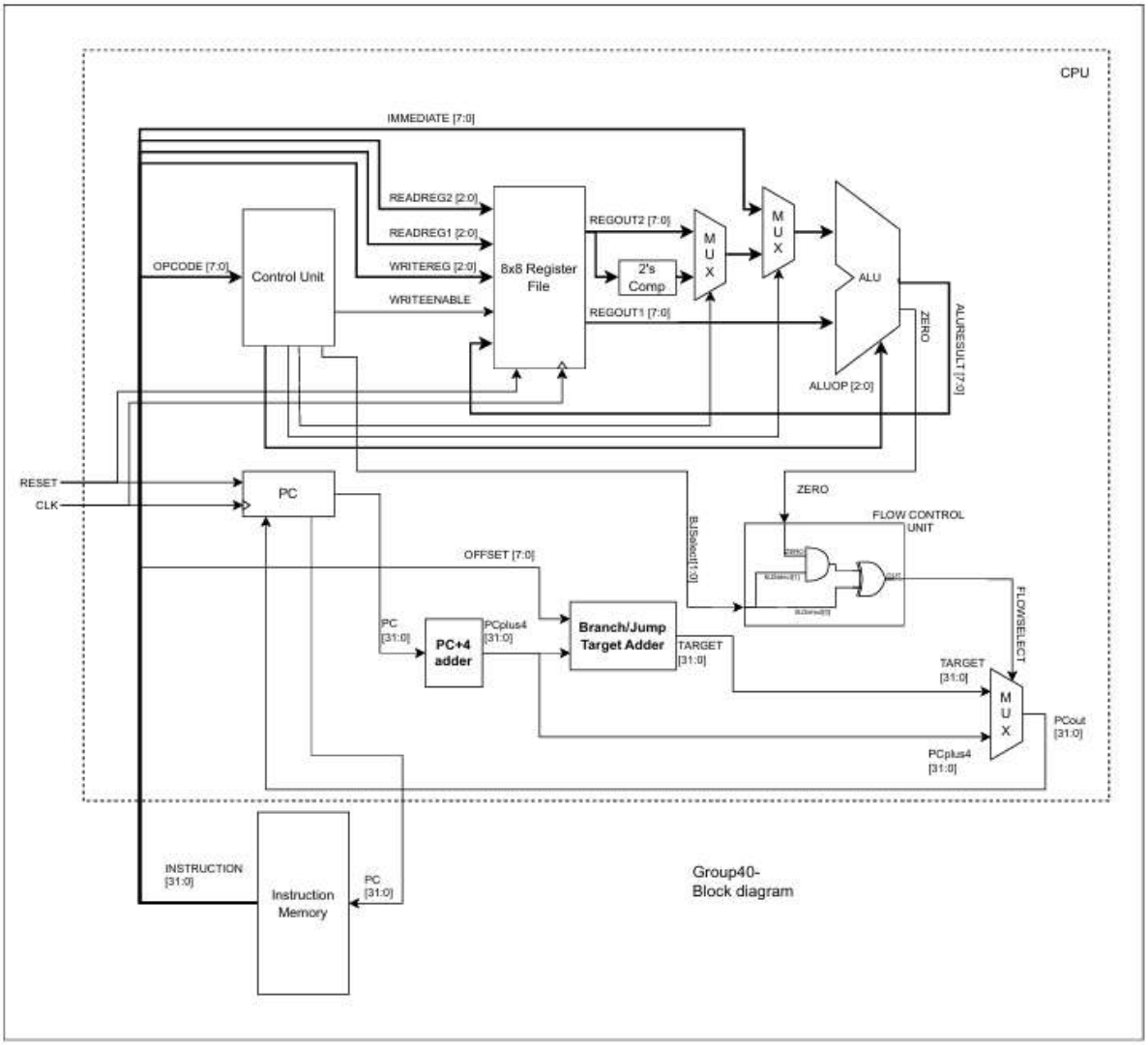
- The **ALU** was expanded with new functional units capable of performing multiplication, shifts (logical and arithmetic), and bit rotations.
- The **Branch/Jump logic** was slightly modified to incorporate the behavior of the bne (branch if not equal) instruction.

ALU Structure

The **Arithmetic Logic Unit (ALU)** is a core component of the CPU responsible for carrying out arithmetic and logical operations. It performs essential tasks such as addition, subtraction, bitwise operations (AND, OR), shifting, multiplication, and more. In this CPU design, the ALU was extended to support new functions like multiplication, logical and arithmetic shifts, and bit rotation, enabling more complex and efficient instruction execution. Each ALU operation is selected using a control signal (SELECT), and the execution time varies depending on the complexity of the operation.

SELECT	**Function**	**Description**	**Supported Instructions**	**Unit's Delay**
000	FORWARD	DATA2→RESULT	loadi, mov	#1
001	ADD	DATA1+DATA2→RESULT	add, sub, beq, bne	#2
010	AND	DATA1&DATA2→RESULT	and	#1
011	OR	DATA1|DATA2→RESULT	or	#1
100	LSHIFT	DATA1<<DATA2→RESULT	sll	#2
101	RSHIFT	(Logical Shift) DATA1>>DATA2→RESULT **Or** (Arithmetic Shift) DATA1>>>DATA2→RESULT **Or** Rotate Right DATA1 by DATA2 times (Operation chosen depends on first two MSB bits of DATA2)	srl, sra, ror	#2
110	MULT	DATA1*DATA2→RESULT	mult	#3

CPU Block diagram



This block diagram shows the main parts of the CPU and how they work together to execute instructions. It includes components like the **Instruction Memory**, **Program Counter (PC)**, **Control Unit**, **Register File**, **ALU**, and **Flow Control Unit**. The Control Unit sends signals to control the data flow, while the ALU performs operations like add, shift, and multiply. The updated diagram also supports new instructions such as mult, sll, srl, sra, ror, and bne, by adding new control logic and ALU functions.

OPCODEs for Decoding Instructions

Instruction	OPCODE
loadi	00000000
mov	00000001
add	00000010
sub	00000011
and	00000100
or	00000101
j	00000110
beq	00000111
bne	00001000
mult	00001001
sll	00001010
srl	00001011
sra	00001100
ror	00001101

BNE Instruction Implementation

To support the bne (branch if not equal) instruction, a small but effective change was made to the **Flow Control Unit** in the CPU. The **OR gate** that was previously used was replaced with an **XOR gate**, and the control signals BRANCH and JUMP were combined into a new 2-bit signal called **BJSelect**.

```
module flowControl(
    input [1:0] BJSELECT,
    input ZERO,
    output OUT
);

// XOR-based flow control logic:
// BEQ: BJSELECT = 2'b10, branch if ZERO == 1
// BNE: BJSELECT = 2'b11, branch if ZERO == 0
assign OUT = BJSELECT[0] ^ (BJSELECT[1] & ZERO);

endmodule
```

This change allows the CPU to perform a branch if the values in two registers are **not equal**, by checking if the **ZERO signal from the ALU is low**. For the bne instruction, **BJSelect is set to 11**, which causes a jump only when the comparison result is not zero.

The **timing and performance** of the bne instruction also remains the same as the existing beq instruction, keeping the design simple and efficient.

This table shows the behaviour of the flow control unit.

BJSelect	FLOWSELECT
00	0
01	1
10	1 if ZERO is HIGH
11	1 if ZERO is LOW

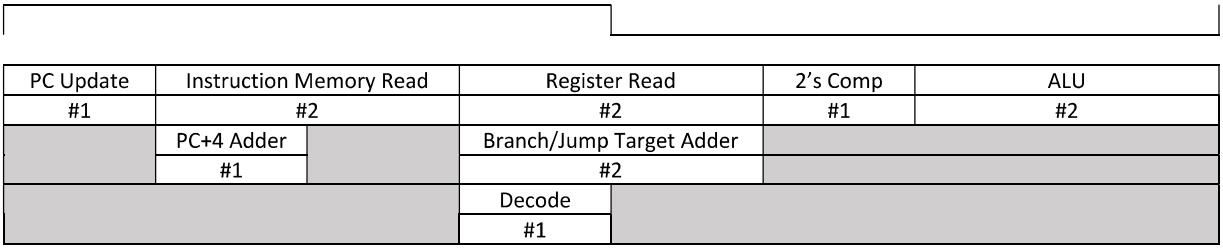


Figure 1:Timing Details of BNE instruction.

MULT Unit Implementation

To support the MULT instruction in the CPU architecture, a dedicated multiplication unit was designed and integrated into the ALU. This unit is capable of performing 8-bit unsigned binary multiplication and operates in a fully combinational manner to ensure fast computation without relying on sequential control.

Design Overview

The multiplier is based on a combinational array multiplier architecture, which mimics the manual binary multiplication process. The multiplication is performed by generating and summing partial products using Full Adders. The design consists of the following key components:

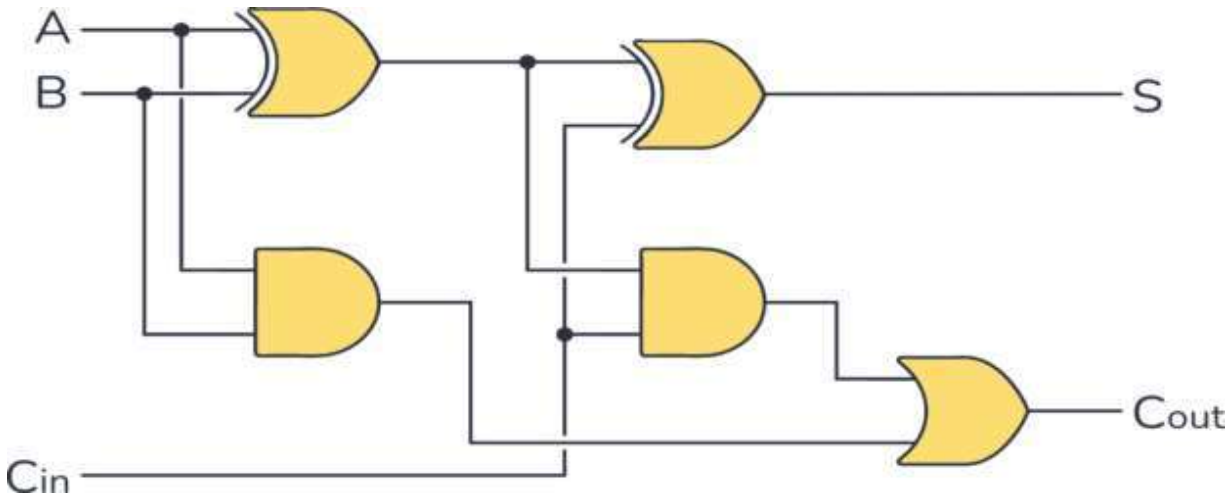
- Partial Product Generation:
  Each bit of the multiplier is ANDed with all bits of the multiplicand, producing rows of partial products. This forms the basic multiplication matrix.
- Adder Tree (Carry-Save Addition):
  The partial products are summed using a tree of Full Adders (FAs) arranged in layers. Each layer reduces the number of rows by summing three rows into two until a final result is obtained. This structure is similar to a Wallace Tree or Dadda Tree, which optimizes the speed of multiplication.
- Full Adder Module:
  A custom Full Adder module was implemented using basic combinational logic (XOR,

AND, OR gates). It takes three input bits (two operand bits and a carry-in) and produces a sum and a carry-out bit. These adders are connected to propagate carries correctly through the structure.

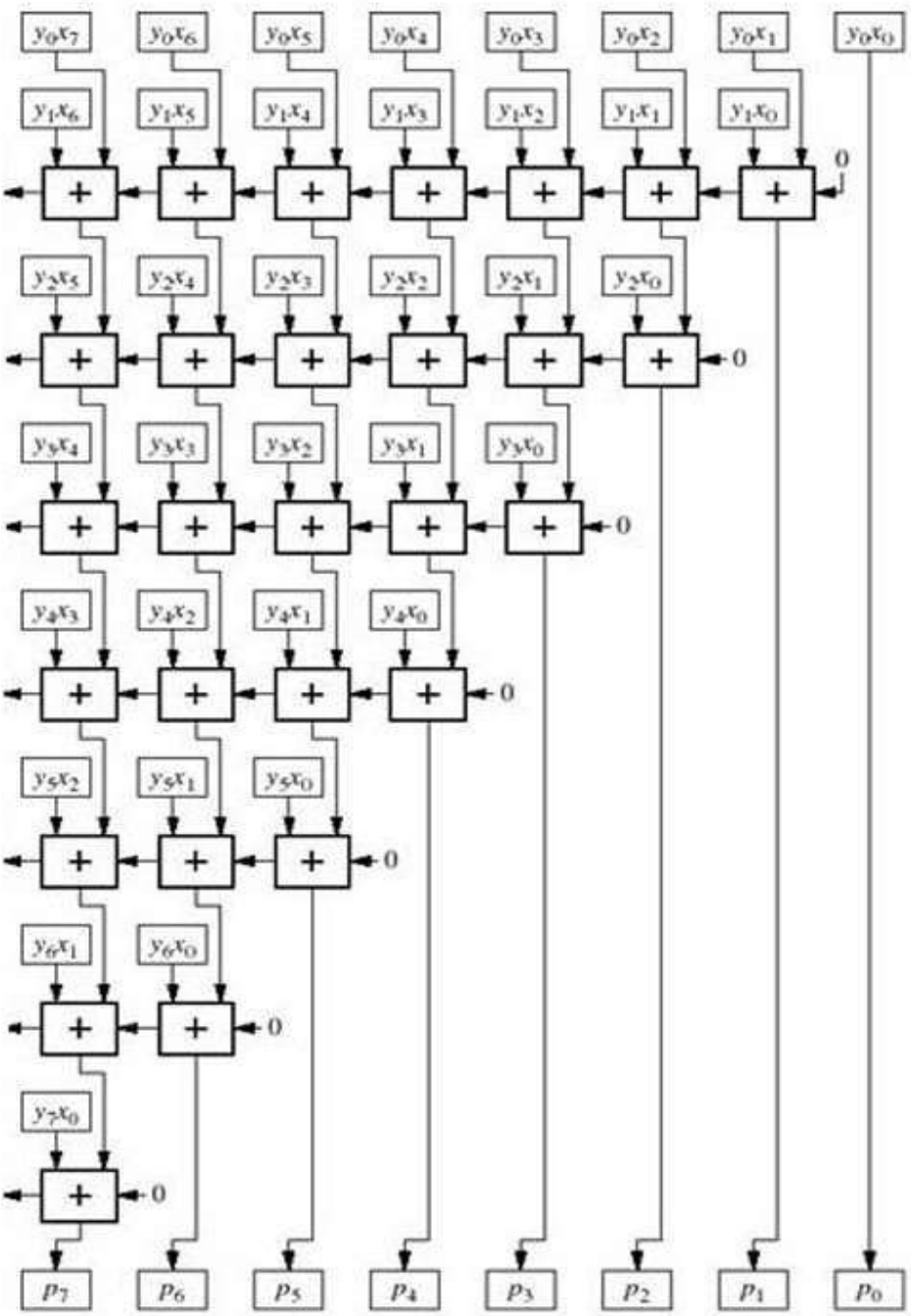
Implementation of the full adder

```
//Full Adder
module FullAdder(A, B, C, SUM, CARRY);

    //Input and output port declaration
    input A, B, C;
    output SUM, CARRY;

    //Combinational logic for SUM and CARRY bit outputs
    assign SUM = (A ^ B ^ C);
    assign CARRY = (A & B) + (C & (A ^ B));

endmodule
```

Using basic combinational logic.



Unit block diagram



This block diagram represents an 8x8 array multiplier used for binary multiplication of two 8-bit numbers, labeled as y7y6…y0y_7 y_6 \ldots y_0y7y6…y0 and x7x6…x0x_7 x_6 \ldots x_0x7x6 …x0. Each small block with a plus sign (+) represents an adder that sums partial products generated by multiplying individual bits of the two numbers. The partial products are aligned

diagonally and added through multiple stages to produce the final multiplication result, which is an output PPP ranging from P0P_0P0 to P7P_7P7. The zeros (0) represent the addition of zero values for alignment in the addition process. This array structure efficiently calculates the full product by summing all partial bit multiplications in parallel.

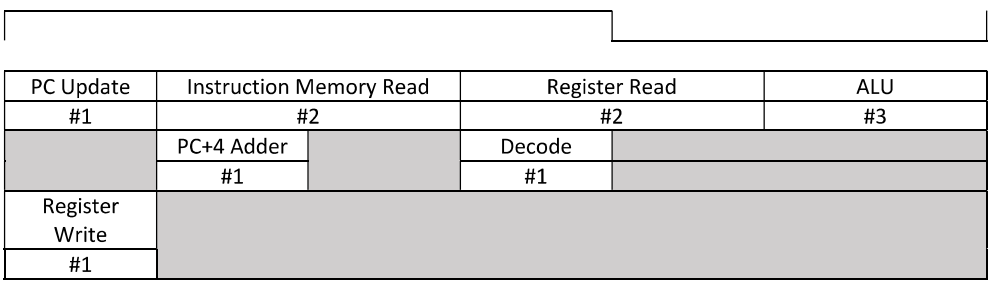


Figure 2:Timing Details for the MULT instruction

Right Shift Unit

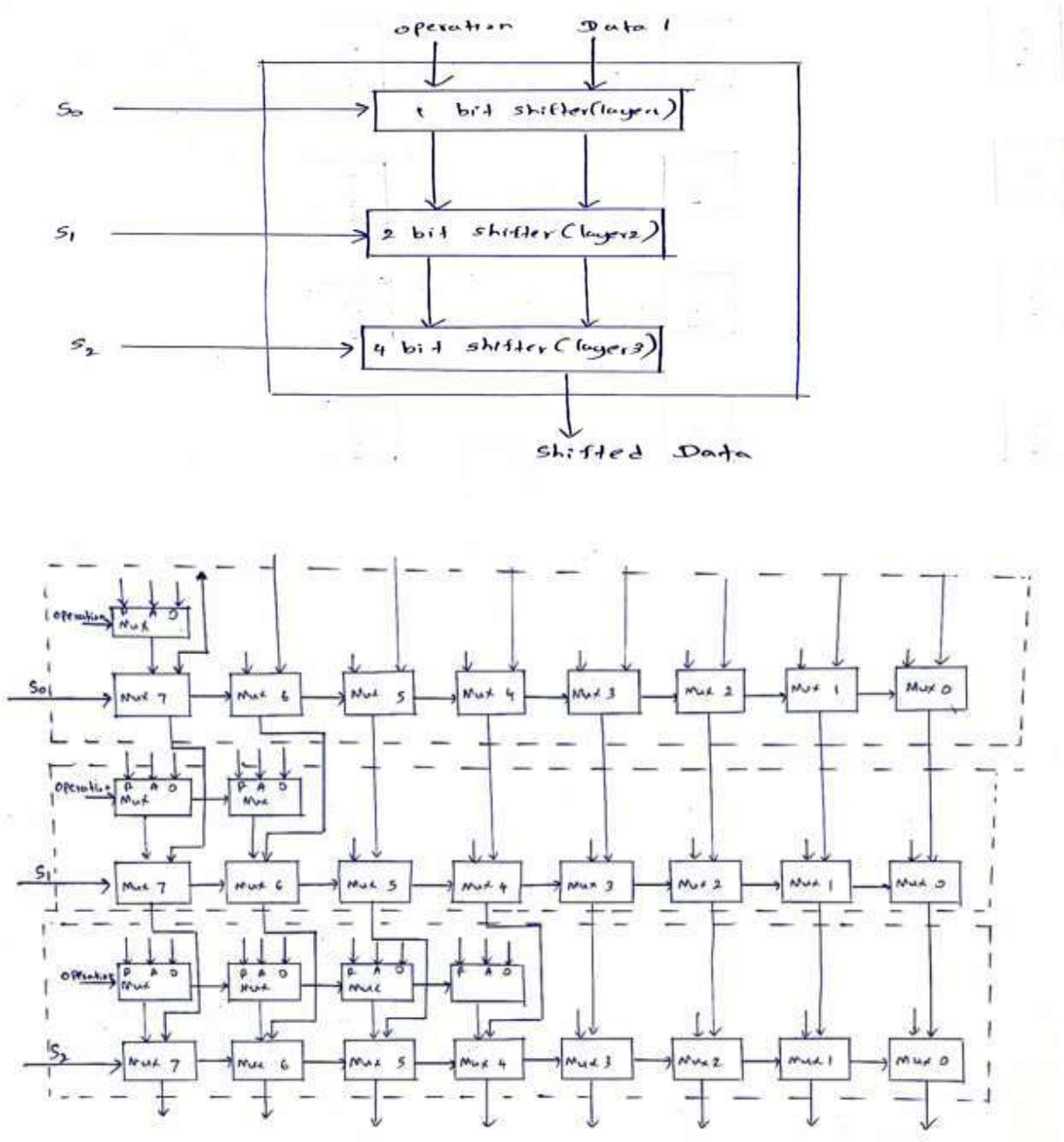
All three right shift instructions are executed using a dedicated functional unit called the RSHIFT unit, which is responsible for performing all types of right shift and rotate operations. The RSHIFT unit is based on a barrel shifter architecture, enabling efficient bit manipulation. In addition to the standard shifting capability, the RSHIFT unit incorporates extra hardware to support both arithmetic and circular (rotate) shift operations. The following block diagram illustrates the internal structure and components of the RSHIFT unit.

Module for 2x1 multiplexer

```
//2x1 MUX Module
module mux2x1(out, in0,in1,s);
    //Declaration of input and output ports
    input in0, in1,s;
    output out ;
    wire orIn0 ,orIn1;

    //MUX implementation
    and (orIn0,in0,!s);
    and (orIn1,in1,s);
    or (out , orIn0,orIn1);

endmodule
```

Module for 3x1 Multiplexer

```
//3x1 MUX Module
module mux3x1 (out , in2 , in1 , in0 , s);
    //Declaration of input and output ports
    input in0,in1,in2;
    output out ;
    input [1:0] s ;
    wire orIn [2:0] ;
    wire andOut[2:0];

    //MUX implementation
    or (out , orIn[0], orIn[1] , orIn[2]);

    and(andOut[0] , !s[0],!s[1]);
    and (orIn[0] , in0 , andOut[0]);


    and(andOut[1] , s[0],!s[1]);
    and (orIn[1] , in1 , andOut[1]);

    and(andOut[2] , !s[0],s[1]);
    and (orIn[2] , in2 , andOut[2]);

endmodule
```



Right Shift Unit Block diagram

Given that the data word size is 8 bits, any shift operation by a value greater than 8 yields the same result as shifting by exactly 8 bits. To reduce hardware complexity, the shifter is designed to support shift operations only up to a maximum of 8 positions. For shift values exceeding 8,

the assembler has been enhanced to automatically detect these cases and substitute the shift amount with 8, thereby ensuring correct operation while simplifying the hardware design.

DATA[7:6]	Operation Performed
00	Logical Right Shift
01	Arithmetic Right Shift
10	Rotate Right

Behaviour of Right shift unit

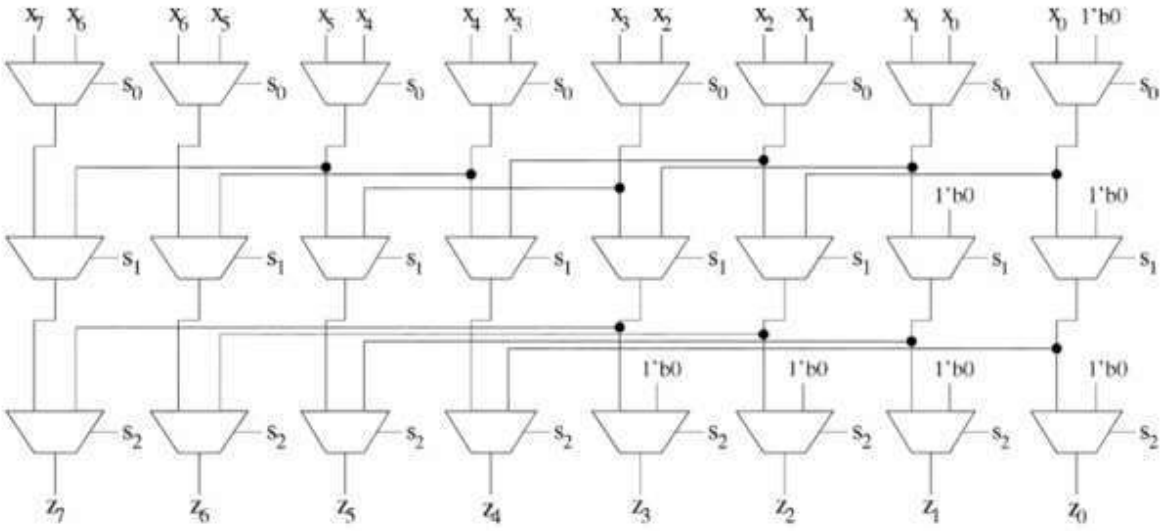
For the srl/sra/ror

PC Update	Instruction Memory Read		Register Read	ALU	
#1	#2		#2	#2	
	PC+4 Adder		Decode		
	#1		#1		
Register Write					
#1					

Timing Diagram of Right shift instruction

LSHIFT Unit

To enable bitwise left shift operations, an additional functional unit was integrated into the ALU. This LSHIFT unit utilizes a barrel shifter architecture, employing multiple layers of multiplexers (MUXes) to produce the desired shifted output. Each MUX within the unit is implemented as an independent module, enhancing modularity and clarity in design. The following block diagram illustrates the structure of the LSHIFT functional unit.



Because the data word is 8 bits wide, performing a shift by any value greater than 8 yields the same result as shifting by 8. To keep the hardware design straightforward, the shifter is configured to support shift operations for up to 8 positions only. For shift amounts exceeding 8, the assembler has been updated to recognize these cases and automatically substitute the shift value with 8. This approach ensures correct functionality while maintaining a simple hardware implementation.

The LSHIFT unit is specifically designed to execute the SLL (Shift Logical Left) instruction. For this operation, a simulation delay of #2 time units has been assigned. The timing details for this instruction are provided below.

PC Update	Instruction Memory Read		Register Read		ALU	
#1	#2		#2		#2	
	PC+4 Adder		Decode			

Timing deays fot SLL instruction datapath