Lab 5 Part 5 Report – Extended Instruction Set Implementation

Group 06
E/21/087-Dewagedara D.M.E.S.
E/21/302 Perera W.S.S.

1. OVERVIEW

In this bonus part, we extended our processor to support six new instructions:

1. `mult` (multiply)
2. `sll` (logical shift left)
3. `srl` (logical shift right)
4. `sra` (arithmetic shift right)
5. `ror` (rotate right)
6. `bne` (branch if not equal)

Each of these instructions was added without increasing the ALU control signal width (3-bit ALUOP), requiring efficient reuse of existing functional units and minimal additional control logic.

1.1. Instruction Encoding

Each instruction uses the following format:

OPCODE	RD/IMM	RT	RS/IMM

1.2 ALU Functions

INSTRUCTION	FORMAT	EXAMPLE	OPCODE(3-bit ALUOP)
`mult`	`mult rd rt rs`	`mult 4 1 2`	100
`sll`	`sll rd rt imm`	`sll 4 1 0x02`	101
`srl`	`srl rd rs imm`	`srl 4 1 0x02`	101 (same as sll)
`sra`	`sra rd rt imm`	`sra 4 1 0x02`	110
`ror`	`ror rd rt imm`	`ror 4 1 0x02`	111
`bne`	`bne rd rt rs`	`bne 4 1 2`	001 (same as sub)

We reused opcode 101 and 001 and reassigned ALUOPs to avoid needing more than 3 bits.

1.3 ALU Functional Unit Modifications

To implement all six instructions using only 8 ALUOP values, we structured shared functional units as follows

ALUOP	FUNCTION	UNIT DELAY
000	FORWARD	#1
001	ADD/SUBTRACT	#2
010	BITWISE AND	#1
011	BITWISE OR	#1
100	MULTIPLICATION	#2
101	SHIFT LEFT LOGICAL/SHIFT RIGHT LOGICAL	#2
110	SHIFT RIGHT ARITHMETIC	#2
111	ROTATE RIGHT	#2

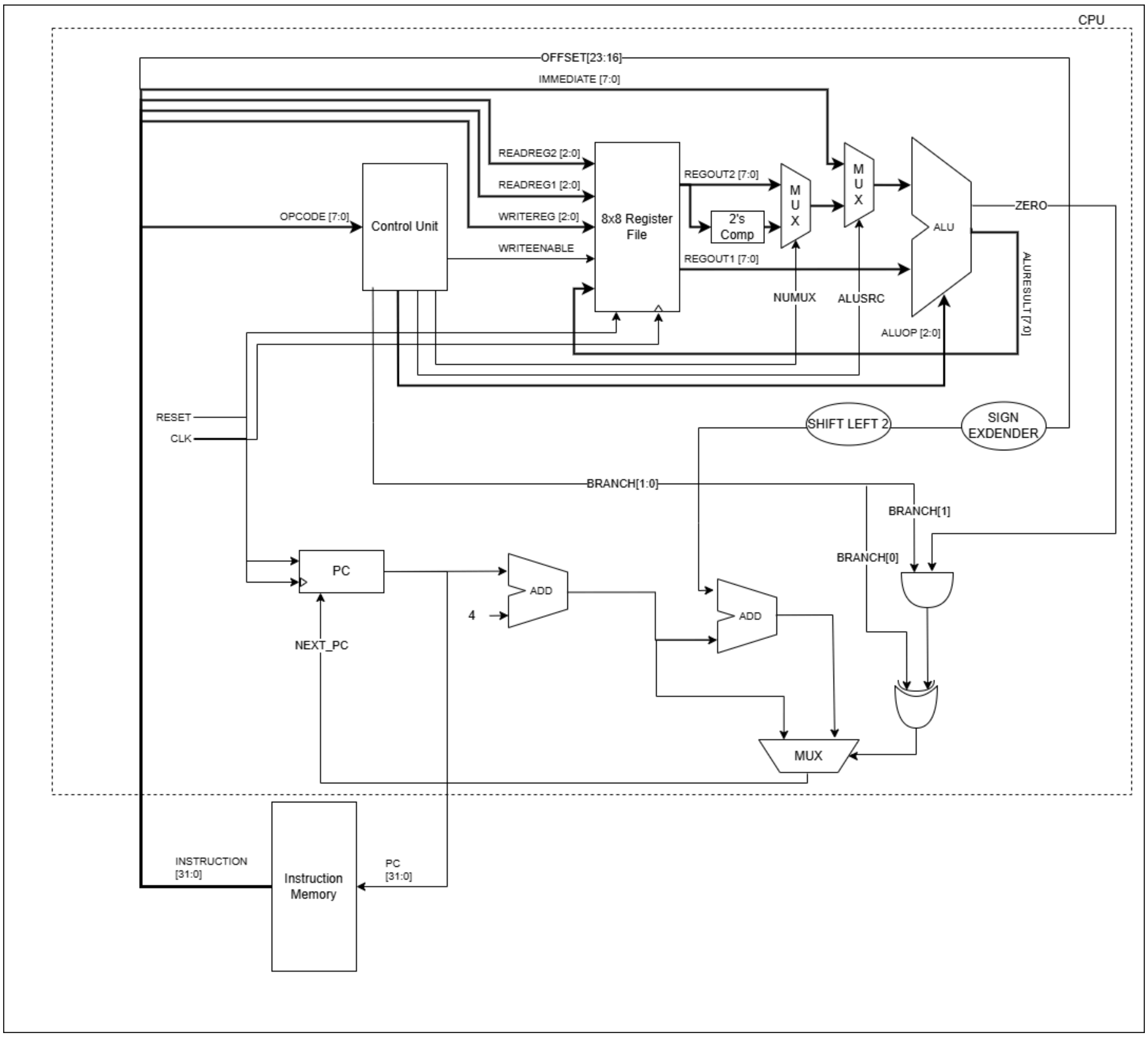
1.4 Instruction Encodings, Assigned Opcodes, and Control Signals

INSTRUCTION	INSTRUCTION OPCODE	WRITE ENABLE	ALUOP	ALUSRC	NEMUX	BRANCH
`add`	00000000	1	001	1	0	00
`sub`	00000001	1	001	1	1	00
`and`	00000010	1	010	1	0	00
`or`	00000011	1	011	1	0	00
`mov`	00000100	1	000	1	0	00
`loadi`	00000101	1	000	0	0	00
`j`	00000110	0	000	0	0	01
`beq`	00000111	0	001	1	1	10
`bne`	00001000	0	001	1	1	11
`mult`	00001001	1	100	1	0	00
`sll`	00001010	1	101	0	0	00
`srl`	00001011	1	101	0	0	00
`sra`	00001100	1	110	0	0	00
`ror`	00001101	1	111	0	0	00

Branch Encoding,

BRANCH	FLOWSELECT
00	Normal sequential execution
01	Unconditional jump
10	Branch if equal (BEQ)
11	Branch if not equal (BNE)

1.5 CPU BLOCK DIGRAM



2 .New Instructions

1. multiplication -`mult`

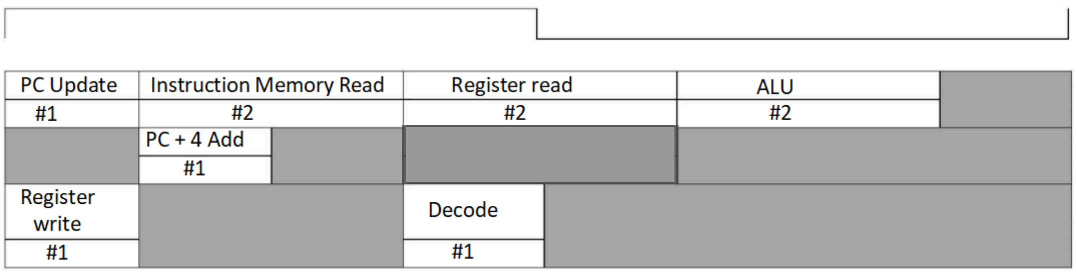
The `mult` instruction is implemented using a dedicated 8×8 array multiplier built into the ALU. It generates partial products by ANDing each bit of the multiplicand with each bit of the multiplier. These partial products are then combined using layers of Full Adders, which are constructed from basic logic gates.

The addition of partial products is organized in a layered, triangular structure: the early layers handle more bits, while later layers deal with fewer. This structure helps optimize hardware usage and ensures correct carry propagation through dedicated carry chains between Full Adders.

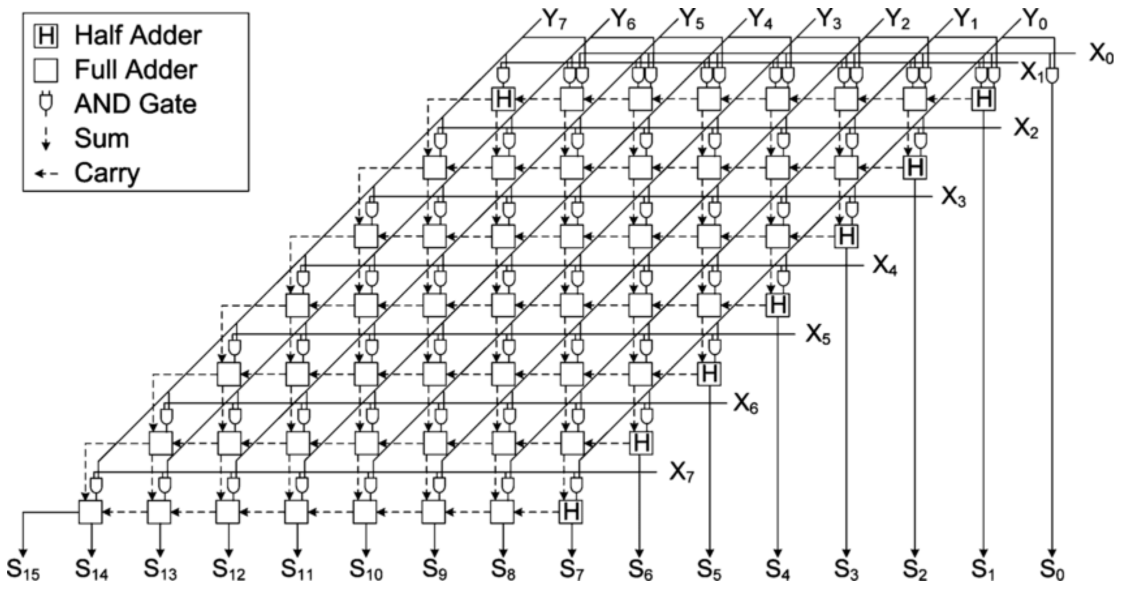
All operations take place within an 8-bit system. If the multiplication result exceeds 255, the extra bits are truncated, leading to overflow and possibly incorrect results. This trade-off keeps the multiplier hardware simple and efficient while maintaining basic multiplication functionality.

The unit delay is taken as 2units to keep instruction complete within a single clock cycle. (even with 3 units delay instruction complete within a single clock cycle)

TIMING DIAGRAM,



DESIGN



H Half Adder
Full Adder
AND Gate
Sum
Carry
Y_7 Y_6 Y_5 Y_4 Y_3 Y_2 Y_1 Y_0
X_0 X_1 X_2 X_3 X_4 X_5 X_6 X_7
S_{15} S_{14} S_{13} S_{12} S_{11} S_{10} S_9 S_8 S_7 S_6 S_5 S_4 S_3 S_2 S_1 S_0

Design diagram shows a full 16×16 array multiplier, which produces a 64-bit output. Since our processor is 8-bit , we only implemented the lower 8×8 half of this design. This generates a 16-bit result , from the result we only preserve the lower 8 bits of the final output. The upper 8 bits are discarded, which can lead to overflow if the result exceeds 255.

2. Logical Shift Left and Logical Shift Right – `sll` and `srl` instructions

Shift Unit Integration in ALU

A dedicated functional unit was added to the ALU to support both left and right bitwise shift operations. This was achieved using a unified hardware structure built as a barrel shifter, which uses layers of multiplexers to generate the correct shifted output based on the direction. To ensure modular design and easier verification, each multiplexer component was implemented as a separate module.

For our design, we estimate a delay of 2 time units.(even with 3 units delay instruction complete within a single clock cycle)

Shift Amount Handling and Overflow Behavior

Given the processor’s 8-bit data word size, shifting by more than 8 bits effectively removes all data bits from the representable range. As a result, shifting by any value greater than 8 produces the same output as shifting by 8—completely zero. To reduce hardware complexity while preserving correctness, the shifter only supports shift values up to 8. If the shift amount exceeds this limit, the assembler detects it and automatically replaces it with 8. This preserves the expected zero-fill behavior without additional hardware.

Direction Encoding Using Shift Amount

The shift unit does not rely on extra control signals to separate between left and right shifts. Instead, the most significant bit (MSB) of the shift amount is used as a direction flag:

- **MSB = 0** → Left shift
- **MSB = 1** → Right shift

This MSB is embedded during compilation, so the datapath doesn't require additional wiring for shift direction control.

Internal Routing in Barrel Shifter

Inside the barrel shifter, the multiplexers use the MSB to determine the routing of bits:

- For **left shifts (MSB = 0)**, bits are routed to higher positions with zero-fill at the least significant bits.
- For **right shifts (MSB = 1)**, bits are routed to lower positions with zero-fill at the most significant bits.

This allows a single, compact architecture to support both shift directions effectively.

Consistent Overflow Handling

Both shift operations—**SLL (shift left logical)** and **SRL (shift right logical)**—share a consistent overflow behavior. Any shift amount greater than 8 results in an output of all zeros. This makes the unit's behavior predictable and simplifies testing and verification, since shift direction doesn't affect overflow results.

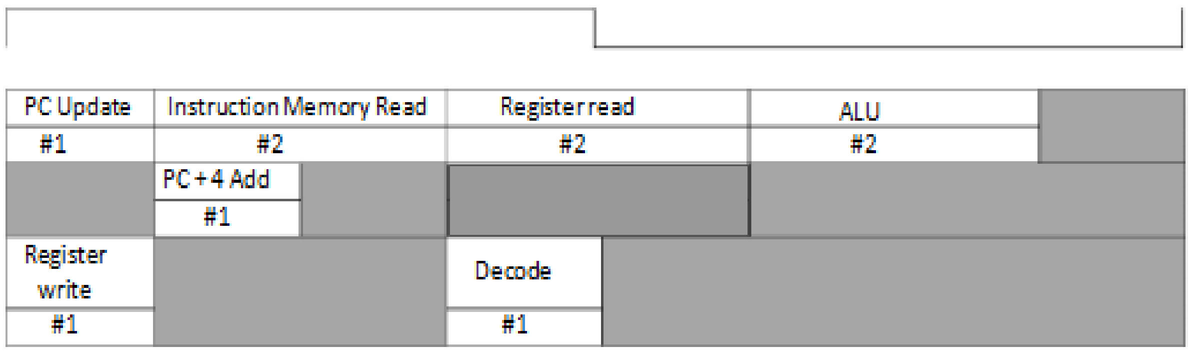
Corner Case: Misinterpreted Shift Direction

A known limitation of this approach is a specific edge case. If a programmer writes a command like `sll r3 r4 128`, the binary of 128 is `10000000`. Here, the MSB is 1, so the processor interprets it as an SRL with a shift amount of 0 (based on the remaining 7 bits). This causes a right shift by 0—returning the original value—where the programmer likely intended a left shift by 128 (which should produce all zeros). While this is a rare case, it highlights a trade-off in the direction encoding scheme.

instruction encoding scheme, all other shift operations within the normal range function correctly according to the intended directional semantics.

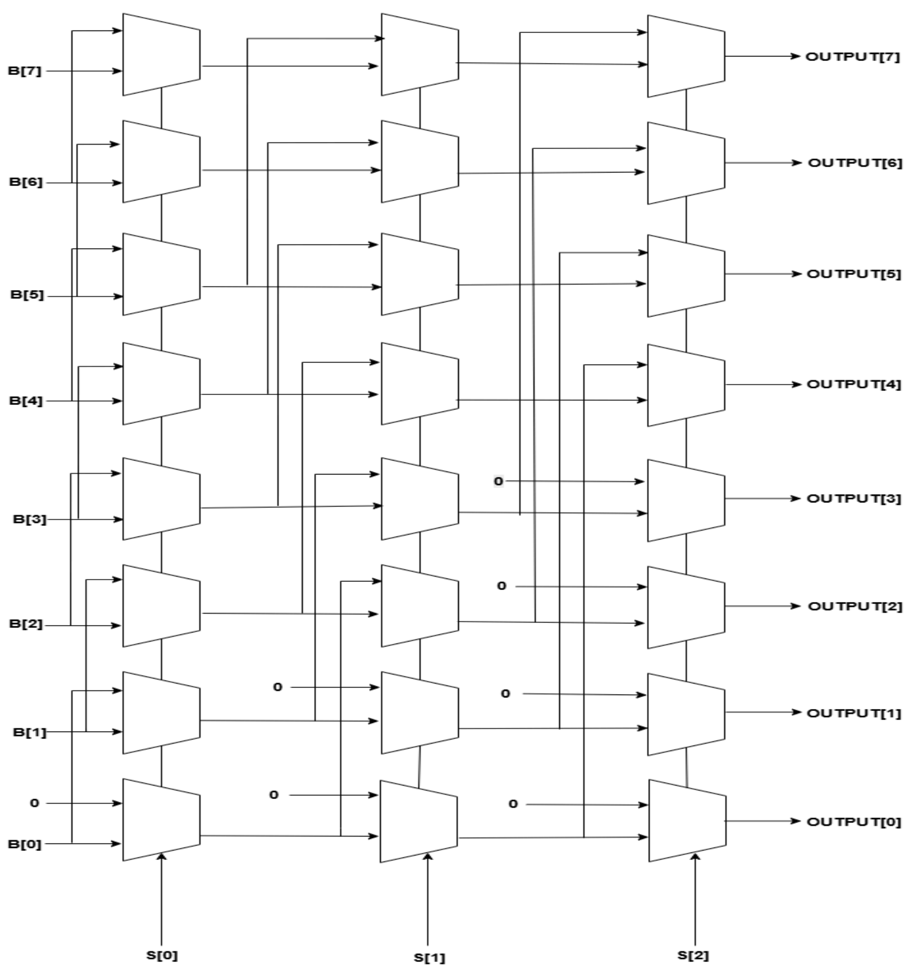
MOST SIGNIFICANT BIT IN IMMEDIATE	**OPERATION PERFORMED**
0 / shift amount remaining seven bits	**`sll`**
1 / shift amount remaining seven bits	**`srl`**

TIMING DIAGRAM,



DESIGN

LOGICAL LEFT SHIFT



3. **BNE**

BNE Instruction Implementation

The `BNE` (Branch if Not Equal) instruction reuses the existing **2-bit branch control signal** and doesn't require any new control lines. The branch control uses a consistent encoding scheme:

- `00` → Normal sequential execution
- `01` → Unconditional jump
- `10` and `11` → BNE variants

This encoding helps make full use of the available bits while staying compatible with the existing branch logic.

Flow Control Logic

The `BNE` logic is handled in the Flow Control Unit using a combinational circuit that takes three inputs:

- `branch[1]`
- `branch[0]`
- `zero` flag from the ALU

The behavior of the circuit is based on a truth table using these inputs:

- When `branch = 00`: output is always `0`, meaning **no branch** (normal execution).
- When `branch = 01`: output is always `1`, enabling **unconditional jumps**.

BNE-Specific Behavior

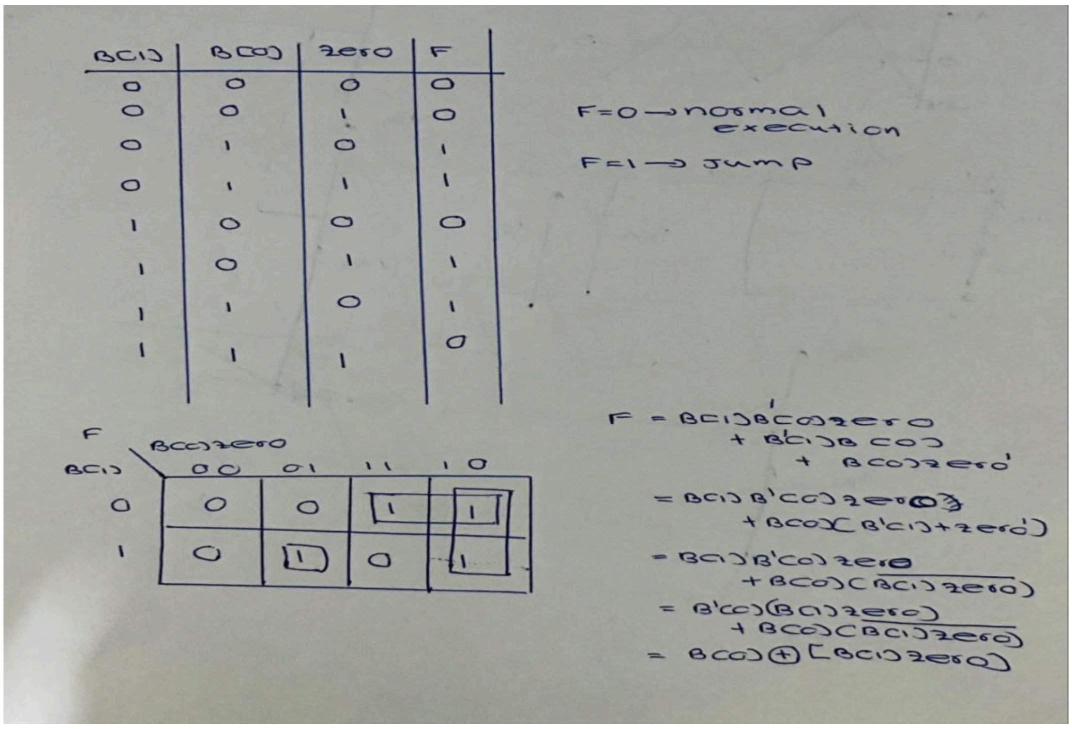
The key behavior for `BNE` happens when `branch = 10` or `11`:

- If the **zero flag is LOW** (operands are **not equal**), the output is `1`, so the branch is taken.

- If the **zero flag is HIGH** (operands are **equal**), the output is `0`, and the program continues normally.

This is the **opposite logic** of a branch-equal operation, which branches when the zero flag is HIGH.

This behavior is achieved through a combinational circuit derived from Karnaugh map optimization.



B[1]	B[0]	zero	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

F=0 → normal execution

F=1 → Jump

F: B[1] \ B[0]zero	00	01	11	10
0	0	0	1	1
1	0	1	0	1

$$F = B[1]B'[0]zero + B'[1]B[0] + B[0]zero'$$

$$= B[1]B'[0]zero + B[0](B'[1]+zero')$$

$$= B[1]B'[0]zero + B[0](\overline{B[1]zero})$$

$$= B'[0](B[1]zero) + B[0](\overline{B[1]zero})$$

$$= B[0] \oplus [B[1]zero]$$

TIMING DIAGRAM,

PC Update	Instruction Memory Read		Register read		2's Comp	ALU
#1	#2		#2		#1	#2
	PC + 4 Add		Branch/jump Target			
	#1		#2			
Register write			Decode			
#1			#1			

4. ROR , SRA , SRL

The ROR (Rotate Right), SRA (Arithmetic Right Shift), and SRL (Logical Right Shift) instructions are all implemented using a single, unified barrel shifter circuit. This shared design improves hardware efficiency by avoiding the need for separate units for each operation. It also simplifies the implementation in code, as the same hardware block can handle all three operations using only a few additional multiplexers for selecting the correct bit-fill behavior.

Three-Layer Barrel Shifter Design

The barrel shifter consists of three layers, each responsible for a specific shift amount:

- Layer 1: Shift by 1 bit
- Layer 2: Shift by 2 bits
- Layer 3: Shift by 4 bits

These layers follow a logarithmic structure, allowing any shift from 0 to 7 bits by activating a combination of layers. For example, a shift by 5 is achieved by enabling the 1-bit and 4-bit layers. This setup covers all possible shift values for an 8-bit word efficiently.

Operation-Specific Bit Fill Logic (Layer 1)

The **first layer** is critical because it handles how the newly created MSB is filled, which differs for each operation:

- **`sra`**(Arithmetic Right Shift): MSB is filled with the original MSB to preserve the sign.
- **`srl`** (Logical Right Shift): MSB is filled with 0.
- **`ror`** (Rotate Right): MSB is filled with the original LSB to rotate bits.

This behavior is managed using 3x1 multiplexers at each bit position, with selection lines controlled by operation type signals (`SRA, SRL, or ROR`).

Second and Third Layer Functionality

The second and third layers perform 2-bit and 4-bit shifts respectively. They take input from the previous layer and apply the shift only if indicated by control signals. These layers use the same logic as the first layer to determine how to fill in the new bits (sign extension, zero-fill, or rotation), ensuring consistent behavior across all layers.

Shift Control and Consistent Timing

The control signals are based on the **shift amount**:

- Bit 0 ⟶ controls 1-bit shift (Layer 1)
- Bit 1 ⟶ controls 2-bit shift (Layer 2)
- Bit 2 ⟶ controls 4-bit shift (Layer 3)

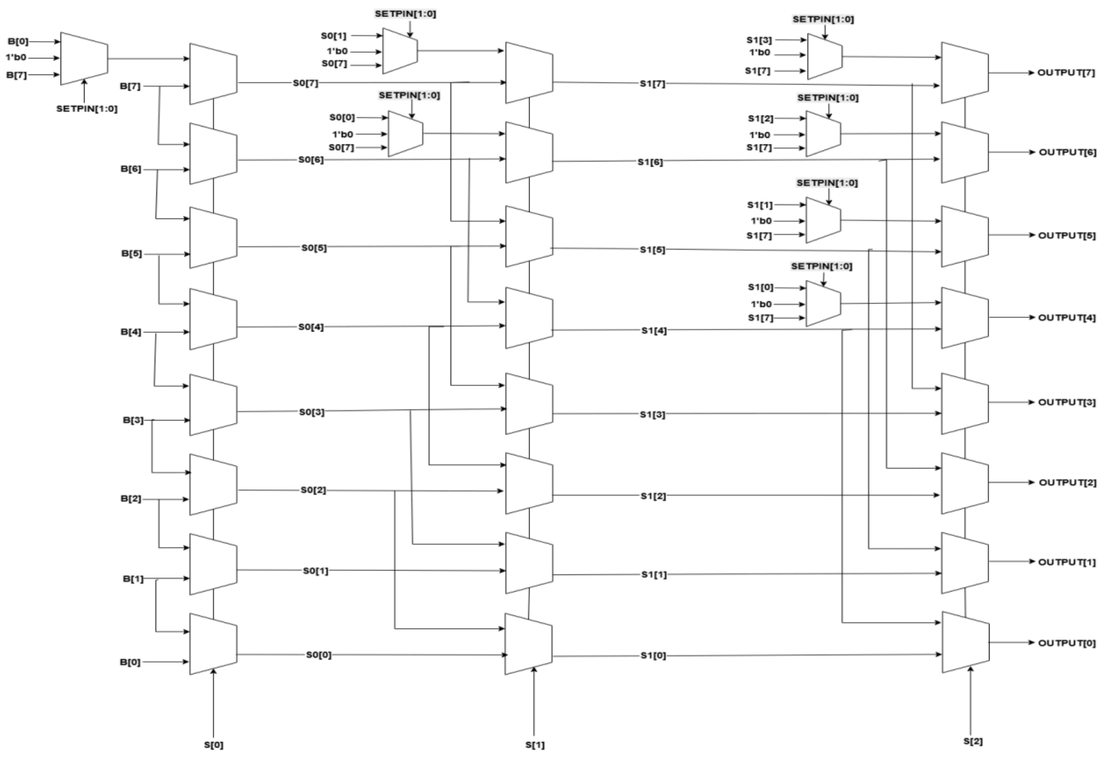
Every input always passes through all three layers, so the timing remains constant, regardless of the shift amount. Only the MUX selection paths change, based on the control inputs.

Hardware Efficiency and Simpler Implementation

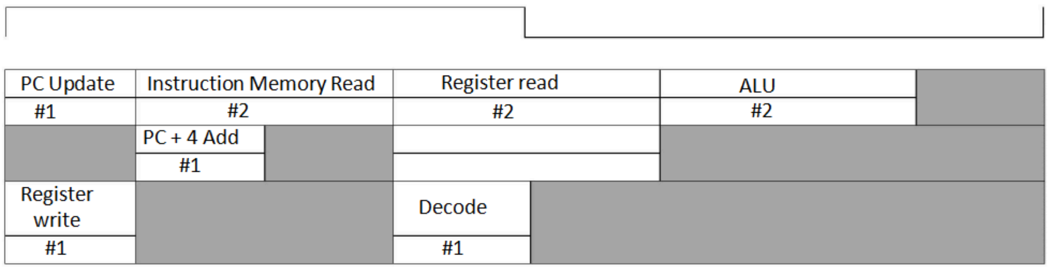
By combining `ROR, SRA, SRL` into a single shift unit, the design avoids redundant circuitry, leading to better hardware utilization. Instead of creating three separate shifters, this design handles all operations with shared logic and a few extra multiplexers. This also makes the Verilog implementation simpler, as the shift unit can be controlled using a small set of operation-type signals, reducing complexity in the datapath and control unit.

DIAGRAM,

LOGICAL RIGHT SHIFT,ROTATE RIGHT,ARTHEMATIC RIGHT SHIFT



TIMING DIGARM



PC Update	Instruction Memory Read	Register read	ALU	▨
#1	#2	#2	#2	▨
▨	PC + 4 Add		▨	▨
▨	#1		▨	▨
Register write	▨	Decode	▨	▨
#1	▨	#1	▨	▨