Ghulam Ishaq Khan Institute of Engineering Sciences and Technology

**Von 2.0**



Faculty of Computer Science

BS Cybersecurity

Computer Organization and Assembly Language

CE222-E

Muhammad Bin Waseem 2023403

Muhammad Daniyal 2023406

Mahad Aqeel 2023286

Date of submission 27/04/2025

# **Table of Contents**

1. **Abstract .......................................................................................................................................... 3**
2. **Theory and Design ........................................................................................................................ 3**
3. **Microoperations ............................................................................................................................ 4**
4. **Instruction Encoding .................................................................................................................... 7**
5. **Innovation and Optimization ..................................................................................................... 10**
6. **Instruction Formats and Addressing Modes ............................................................................ 13**
7. **Details of Registers ...................................................................................................................... 13**
8. **Complete CPU Design ................................................................................................................ 14**
9. **System Bus - Control of Registers, Flags, and Memory .......................................................... 16**
10. **Interrupt Initiated and Program Controlled I/O ..................................................................... 23**
11. **Complete Computer Description ............................................................................................... 26**
12. **Bibliography ................................................................................................................................ 27**

**Abstract**

This work describes an expansion and modernization of the Instruction Set Architecture (ISA) of a Basic Computer that was modeled upon the Von Neumann architecture. Using Stallings (2022)’ foundational instructions, we have added new instructions like SUB, DMT, DSZ, and custom encryption decryption operations (REC and RDC). We show how running at a more advanced computation and data processing level may be enabled on top of a minimalistic system. It is backed with optimised IO handling techniques to enhance system performance and reliability and contains every micro operation for each instruction cycle that is clearly defined. In addition to this, this initiative helps to emphasize the underlying mechanisms of a basic stored program computer and it also serves as an educational prototype to understand low level processor architecture and instruction execution. We strive to bridge abstract theoretical concepts of computer architectures with practical enhancements based on computers tasks of today, through implementation and simulation of these operations.

**Theory and design**

**Introduction to the Basic Computer and Instruction Set Architecture:**One of the simplest forms of a stored program computer is known as the (Basic) Computer (or the Von Neumann machine for short.). This includes the Central Processing Unit (CPU), memory, and input/output devices as its integral component. In the computer, the instruction set is a set of predefined instructions that are part of Instruction Set Architecture (ISA) and the computer follows those on a given set of instructions.

As William Stallings (2022) puts it, the ISA describes the set of operations performed by the processor, the addressing modes and data encoding it allows the machine to use. ADD, LDA and STA types of these instructions allow the CPU to do something with memory and perform some arithmetic operation and control the flow of execution by branching operation (Stalling 2022).

While a basic instruction set such as that of the early computers might seem minimal, Stallings stresses that in fact, such an instruction set is incredibly useful for dynamically building a functional system. Essential operations for memory manipulation, register control, and I/O processing are provided by the ISA of the Basic Computer and taken together they constitute the basis for more complex computing systems (Stallings, 2022).

A great change has evolved the basic computer into a far more powerful computer. While the concepts remain constant, innovation and improvement have allowed significant improvement in efficiency, huge generation, and multiple business operations. In order to make this system more versatile and capable of handling a greater class of tasks, we have added several instructions to the ISA in our report. The advanced arithmetic, logical operations and optimized I/O handling are these additions, which are a great step forward to creating a more stable and high performance computing environment. Introducing these novel computer instructions is meant to enhance the well proven functionality of the simple computer and expand its function to satisfy modern application requirements.

**Microoperations:**

**SUB - Memory Reference Instruction**

T0 AR ← PC Load PC into AR

T1 IR ← M[AR], PC ← PC + 1 Fetch instruction

T2 D0, . . . , D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15)

T3 direct/indirect

T4 DR ← M[AR] Load memory operand

T5 AC ← AC + ~DR + 1

**SWP - Memory Reference Instruction**

T0 AR ← PC

T1 IR ← M[AR], PC ← PC + 1 Fetch instruction

T2 D0, . . . , D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15)

T3 direct/indirect

T4 DR ← M[AR] Fetch operand

T5 M[AR] ← AC

T6 AC←DR

**CLM - Memory Reference Instruction**

T0 AR ← PC

T1 IR ← M[AR], PC ← PC + 1 Fetch instruction

T2 D0, . . . , D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15)

T3 direct/indirect

T4 AC ← 0

T5 M[AR] ← AC

**DSZ - Memory Reference Instruction**

T0 AR ← PC

T1 IR ← M[AR], PC ← PC + 1 Fetch instruction

T2 D0, . . . , D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15)

T3 Direct/Indirect

T4 DR ← M[AR]

T5 DR←DR-1

T6 M[AR]←DR , if DR = 0 then PC←PC+1

**DMT - Register Reference Instruction**

T0 AR ← PC

T1 IR ← M[AR], PC ← PC + 1 Fetch instruction

T2 D0, . . . , D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15)

T3 AC ← AC - 1

**XOR - Memory Reference Instruction**

T0 AR ← PC

T1 IR ← M[AR], PC ← PC + 1 Fetch instruction

T2 D0, . . . , D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15)

T3 Direct/Indirect

T4 DR ← M[AR] Load memory operand

T5 AC←DR **⊕** AC

**REC**- REVERSE ENCRYPTION  **- Memory Reference Instruction**

T0 AR ← PC Fetch the address of the instruction

T1 IR ← M[AR], PC ← PC + 1 Fetch the instruction and increment PC

T2 D0, . . . , D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15)

T3 DR ← M[AR]

T4 AC ← DR ⊕ K Perform XOR between DR and K

T5 AC ← SHL(AC) Left shift AC

T6 DR(15) ← AC(0) , DR(14) ← AC(1) , …….. DR(0)←AC(15) Mirror the bits of AC and store in DR and then in memory

T7 M[AR] ← DR

**RDC** – REVERSE DECRYPT **- Memory Reference Instruction**

T0 AR ← PC Fetch the address of the instruction

T1 IR ← M[AR], PC ← PC + 1 Fetch the instruction and increment PC

T2 D0...D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15) Decode instruction

T3 DR ← M[AR] Load encrypted data into DR

T4 AC(15) ← DR(0) , AC(14) ← DR(1) , …….. AC(0)←DR(15) Reverse the bits of AC

T5 AC ← SHR(AC) Shift DR right (undo SHL)

T6 AC ← AC ⊕ K XOR AC with key to get original data

Note: Decrypted value can be stored manually by applying STA from the user. It is not mentioned in the cycle to ensure Human Error Handling.

**SIN - Register Reference Instruction**

T0 AR ← PC Fetch the address of the instruction

T1 IR ← M[AR], PC ← PC + 1 Fetch the instruction and increment PC

T2 D0...D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15) Decode instruction

T3 PC ← SHRI , AR ←SHRI , PGI ← 1

**SON - Register Reference Instruction**

T0 AR ← PC Fetch the address of the instruction

T1 IR ← M[AR], PC ← PC + 1 Fetch the instruction and increment PC

T2 D0...D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15) Decode instruction

T3 PC← SHRO , AR ←SHRO

**SHR - Register Reference Instruction**

T0 AR ← PC Fetch the address of the instruction

T1 IR ← M[AR], PC ← PC + 1 Fetch the instruction and increment PC

T2 D0...D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15) Decode instruction

T3 AC ← shr AC

**SHL - Register Reference Instruction**

T0 AR ← PC Fetch the address of the instruction

T1 IR ← M[AR], PC ← PC + 1 Fetch the instruction and increment PC

T2 D0...D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15) Decode instruction

T3 AC ← shl AC

**Optimized IIN - I/O instruction**

T0 AR ← PC Fetch the address of the instruction

T1 IR ← M[AR], PC ← PC + 1 Fetch the instruction and increment PC

T2 D0...D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15) Decode instruction

T3 If (FGI = 1) then ( AC(0–7) ← INPR, FGI ← 0 ,PC←PC+1)

**Optimized IOT - I/O instruction**

T0 AR ← PC Fetch the address of the instruction

T1 IR ← M[AR], PC ← PC + 1 Fetch the instruction and increment PC

T2 D0...D15 ← Decode IR(11-14), AR ← IR(0-10), I ← IR(15) Decode instruction

T3 If (FGO = 1) then ( OUTR ← AC(0–7), FGO ← 0 ,PC ← PC + 1)

**Key Register,SRI,SRO will be loaded through hardware.**

**Interrupt Cycle**

~T0~T1~T2(PGI)(IEN)(FGI+FGO):

* FRT0: SHRI ← AR , PC← SHI , PGI←0,IEN← 0, R←0,SC←0
* (~F)RT0: SHRO ← AR, PC← SHO , IEN← 0, R←0,SC←0

**Instruction Encoding:**

| **Type of Instruction** | **Opcode (4 bits)** | **Binary** | **Mnemonic** | **p/r/D** | **Description** |
| --- | --- | --- | --- | --- | --- |
| **Memory Reference** | **0000** | **x 0000 xxxxxxxxxxx** | **AND** | **D0** | AND memory word to AC |
| **0001** | **x 0001 xxxxxxxxxxx** | **ADD** | **D1** | Add memory word to AC |
| **0010** | **x 0010 xxxxxxxxxxx** | **LDA** | **D2** | Load AC from memory |
| **0011** | **x 0011 xxxxxxxxxxx** | **STA** | **D3** | Store content of AC into memory |
| **0100** | **x 0100 xxxxxxxxxxx** | **BUN** | **D4** | Branch Unconditionally |
| **0101** | **x 0101 xxxxxxxxxxx** | **BSA** | **D5** | Branch and save return address |
| **0110** | **x 0110 xxxxxxxxxxx** | **ISZ** | **D6** | Increment and skip if zero |
| **0111** | **x 0111 xxxxxxxxxxx** | **SUB** | **D7** | Subtract memory word to AC |
| **1000** | **x 1000 xxxxxxxxxxx** | **REC** | **D8** | Reverse Encryption |
| **1001** | **x 1001 xxxxxxxxxxx** | **DSZ** | **D9** | Decrement and skip if zero |
| **1010** | **x 1010 xxxxxxxxxxx** | **XOR** | **D10** | XOR AC with memory |
|  |  |  |  |  |
| **1011** | **x 1011 xxxxxxxxxxx** | **SWP** | **D11** | Swap AC with memory |
| **1100** | **x 1100 xxxxxxxxxxx** | **CLM** | **D12** | Clear memory address |
| **1101** | **x 1101 xxxxxxxxxxx** | **RDC** | **D13** | Reverse Decryption |
| **Register Reference** | **1110** | **0 1110 10000000000** | **CLA** | **rB10** | Clear AC |
| **1110** | **0 1110 01000000000** | **CLE** | **rB9** | Clear E |
| **1110** | **0 1110 00100000000** | **CMA** | **rB8** | Complement AC |
| **1110** | **0 1110 00010000000** | **CME** | **rB7** | Complement E |
| **1110** | **0 1110 00001000000** | **CIR** | **rB6** | Circular Shift Right AC and E |
| **1110** | **0 1110 00000100000** | **CIL** | **rB5** | Circular Shift Left AC and E |
| **1110** | **0 1110 00000010000** | **INC** | **rB4** | Increment AC |
| **1110** | **0 1110 00000001000** | **SPA** | **rB3** | Skip next instruction if AC is positive |
| **1110** | **0 1110 00000000100** | **SNA** | **rB2** | Skip next instruction if AC is negative |
| **1110** | **0 1110 00000000010** | **SZA** | **rB1** | Skip next instruction if AC is zero |
| **1110** | **0 1110 00000000001** | **SZE** | **rB0** | Skip next instruction if E is zero |
| **1111** | **0 1111 10000000000** | **HLT** | **uB10** | Halt Computer |
| **1111** | **0 1111 01000000000** | **SHR** | **uB9** | Shift right AC |
| **1111** | **0 1111 00100000000** | **SHL** | **uB8** | Shift left AC |
| **1111** | **0 1111 00010000000** | **SIN** | **uB7** | Jump to Interrupt Location for Input |
| **1111** | **0 1111 00001000000** | **SON** | **uB6** | Jump to Interrupt location for Output |
| **1111** | **0 1111 00000100000** | **DMT** | **uB5** | Decrement AC |
| **Input/Output** | **1110** | **1 1110 10000000000** | **INP** | **pB10** | Input character to AC |
| **1110** | **1 1110 01000000000** | **OUT** | **pB9** | Output character to AC |
| **1110** | **1 1110 00100000000** | **SKI** | **pB8** | Skip on input flag |
| **1110** | **1 1110 00010000000** | **SKO** | **pB7** | Skip on output flag |
| **1110** | **1 1110 00001000000** | **ION** | **pB6** | Interrupt ON |
| **1110** | **1 1110 00000100000** | **IOF** | **pB5** | Interrupt OFF |
| **1110** | **1 1110 00000010000** | **IIN** | **pB4** | Optimized Input |
| **1110** | **1 1110 00000001000** | **IOT** | **pB3** | Optimized Output |

**Innovation and Optimization:**

This project introduces key innovations in both I/O operations and advanced arithmetic/logical instructions, transforming the traditional Von Neumann architecture into a more capable and efficient system. Below, we highlight the major advancements:

1. Advanced Arithmetic & Logical Operations

* Reverse Encryption/Decryption (REC/RDC)

A standout innovation is the REC (Reverse Encrypt) and RDC (Reverse Decrypt) instructions, which provide lightweight data security using:

XOR with a 16-bit key (K register) for basic encryption.

Bit reversal (mirroring) to obfuscate data further.

Shift operations (SHL/SHR) to diffuse patterns.

Microcode Execution:

REC: XOR → SHL → Bit-Reverse → Store

RDC: Bit-Reverse → SHR → XOR → Restore

*Use Case: Secure communication between memory and I/O devices.*

* Enhanced Arithmetic Instructions

SUB: Implements 2’s complement subtraction via microcode (negation + addition).

DMT (Decrement- Register Reference) : Decrements the TEMP register: TEMP ← TEMP - 1

DSZ (Decrement and Skip if Zero) : M[addr] ← M[addr] - 1; if M[addr] == 0 → skip next instruction

SWP (Swap Memory with AC) : Swaps contents of memory at addr and the accumulator.

XOR (Logical XOR with AC): XOR AC with Memory Address

CLM (Clear Memory at Address)

*Optimization:*

* *Contains reduction of complex logics to single cycle or loop based operations.*
* *It enforces reusability of micro-ops for memories (such as SHR, XOR).*
* *Prevents redundant sequences of instruction for operations such as SWP.*
* *It offers basic cryptographic support without further hardware.*
* *Therefore these instructions reduce clock cycles.*

2. Optimized I/O Operations

* Interrupt-Driven I/O

The optimized IIN (Input) and IOT (Output) instructions now:

Check FGI (Input Flag) and FGO (Output Flag) to avoid busy-wait polling.

Auto-increment PC if the device is ready, reducing CPU stalls.

Microcode Example (IIN):

T3: If (FGI = 1) then (AC[0-7] ← INPR, FGI ← 0, PC ← PC + 1)

*Impact: Faster I/O handling by eliminating manual status checks.*

* Memory-Mapped I/O

Unified addressing: I/O devices share memory space (e.g., INPR/OUTR are memory-mapped).

*Simplifies programming: Use LDA/STA for I/O, avoiding dedicated instructions.*

3. Hardware Innovations

* Dedicated Registers

16-bit K register: Stores encryption keys securely.

Temporary Register (TR): Enables multi-step operations .

* Compact ISA Design

11-bit PC, 16-bit AC/DR: Balances addressability and data width.

4-bit opcodes: Efficient encoding for 16+ instructions.

4. Theoretical vs. Practical Impact

| **Feature** | **Traditional Approach** | **Our Innovation** |
| --- | --- | --- |
| Subtraction | Requires ADD with negated operand | Native SUB microcode |
| I/O handling | Polling loops | Interrupt-ready flags |
| Encryption | Software-based (slow) | Hardware Accelerated |
| Decrement | Manually complement plus increment | Auto- Subtraction of 1 |
| Swap | Manually load and store in locations multiple times | Reduces clock cycles significantly by loading and storing in one flow |

**Instruction Formats and Addressing modes**

* Memory Reference Instruction
  + [Opcode (4 bits)] [Addressing Mode (1 bit)] [Memory Address (11 bits)]
* Register Reference Instruction
  + [Opcode (4 bits)] [Addressing Mode (1 bit)] [Register Operation (11 bits)]
* Input/Output Instruction
  + [Opcode (4 bits)] [Addressing Mode (1 bit)] [Input/Output Operation (11 bits)]

The Size of Memory in the processor would be : **2048 words x 16 bits**

**Details of Registers**

The following table consists of a comprehensive list of all the registers used,the no.of bits that can be stored in those registers, and the function of those registers

| **Register Symbol** | **No. Of Bits** | **Register Name** | **Function** |
| --- | --- | --- | --- |
| DR | 16 | Data Register | Holds fetched data |
| AR | 11 | Address Register | Holds memory address |
| IR | 16 | Instruction Register | Holds current instruction |
| AC | 16 | Accumulator | Performs ALU operations |
| PC | 11 | Program Counter | Points to next instruction |
| TR | 16 | Temporary Register | Holds temporary data |
| K | 16 | Key Register | Stores key for REC/RDC |
| INPR | 8 | Input Register | Holds input character |
| OUTR | 8 | Output Register | Holds output character |
| SC | 3 | Sequence Counter | Counter for clock cycles that aids in microoperations of an instruction |
| SHRI | 11 | Shadow Register input | Stores return address |
| SHRO | 11 | Shadow Register output | Stores return address |
| SHI | 11 | Backup Shadow Input Register | Stores interrupt address |
| SHO | 11 | Backup Shadow Output Register | Stores interrupt address |

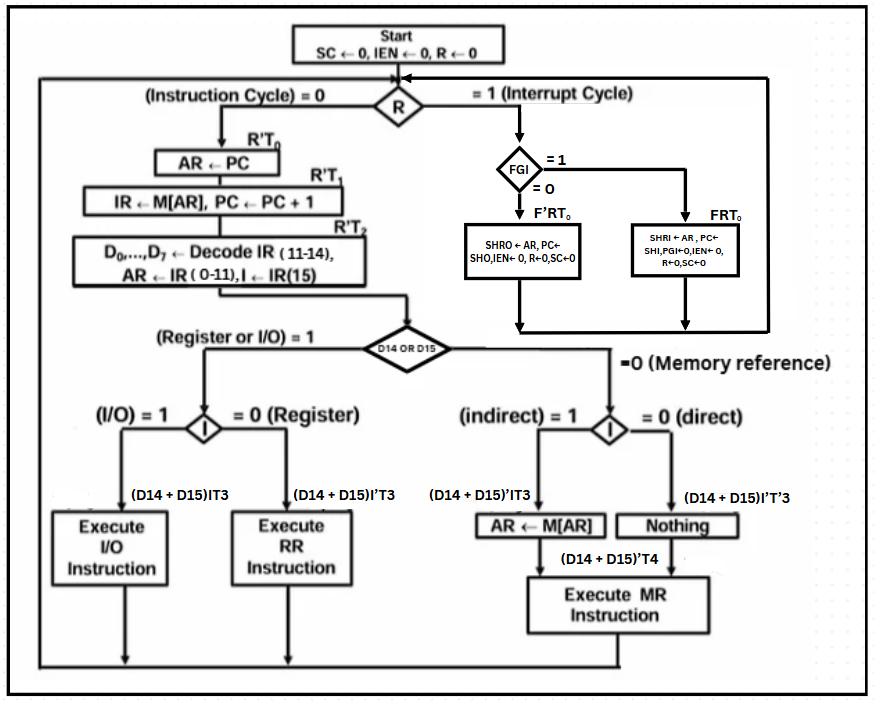
**Complete CPU design**

The Von 2.0 will consist of the following hardware components:

1. Memory unit with *2048 words*, *16 bits each*
2. Fourteen registers : AR, PC, DR, AC, IR, TR, OUTR, INPR, SC, SHRO, SHRI,SHI,SHO, K
3. 8 Flip-flops : I, E, R, FGI, FGO, PGI , IEN, S
4. Two decoders : 4 x 16 operation decoder and a (4 x 16 ) timing decoder
5. 16 bit common bus
6. Control Logic Gates
7. Adder and logic circuit connected to the input of AC

The memory unit is a standard component that can be obtained readily from a commercial source.

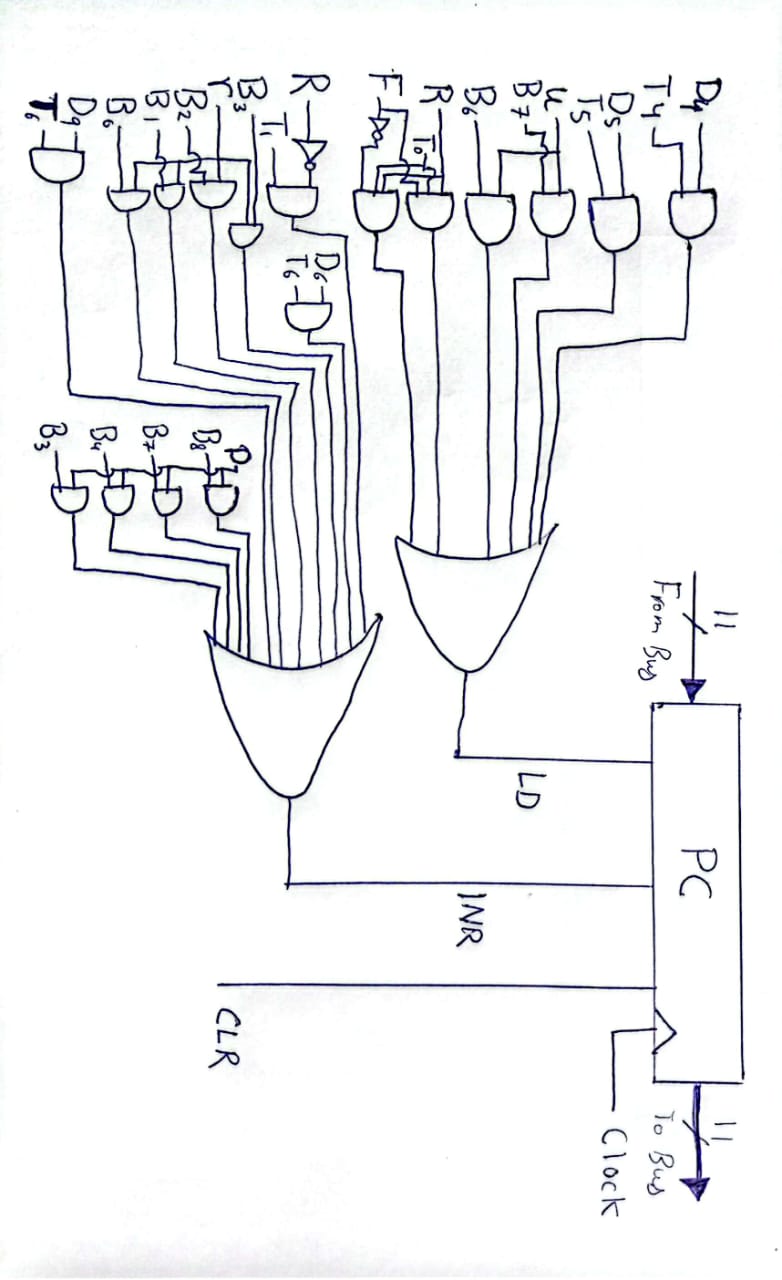
**Flowchart for Computer Description**



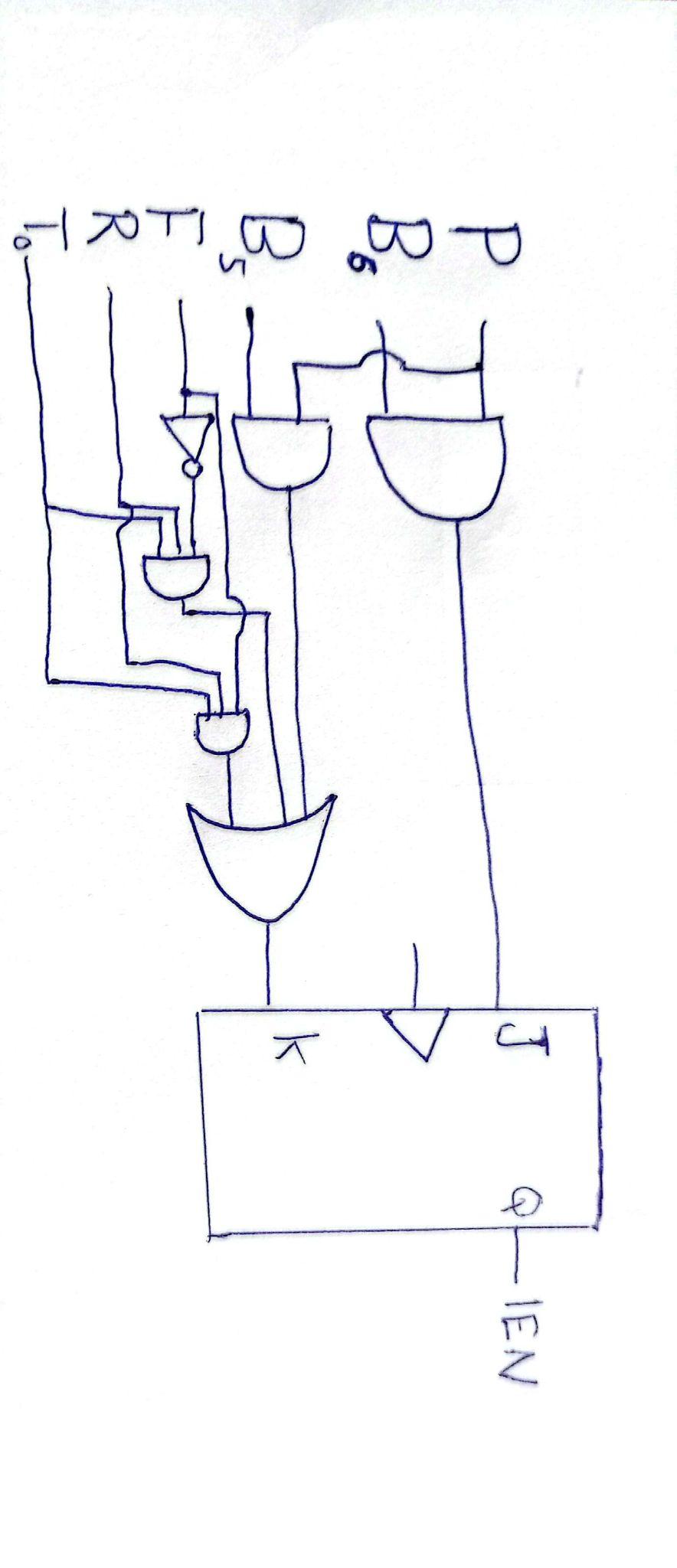
**System Bus - Control of Registers, Flags and Memory–**

| **S3** | **S2** | **S1** | **S0** | **Register** |
| --- | --- | --- | --- | --- |
| **0** | **0** | **0** | **0** | **None** |
| **0** | **0** | **0** | **1** | **AR** |
| **0** | **0** | **1** | **0** | **SHRI** |
| **0** | **0** | **1** | **1** | **SHRO** |
| **0** | **1** | **0** | **0** | **SRI** |
| **0** | **1** | **0** | **1** | **SRO** |
| **0** | **1** | **1** | **0** | **PC** |
| **0** | **1** | **1** | **1** | **DR** |
| **1** | **0** | **0** | **0** | **AC** |
| **1** | **0** | **0** | **1** | **IR** |
| **1** | **0** | **1** | **0** | **TR** |
| **1** | **0** | **1** | **1** | **K** |
| **1** | **1** | **0** | **0** | **Memory** |

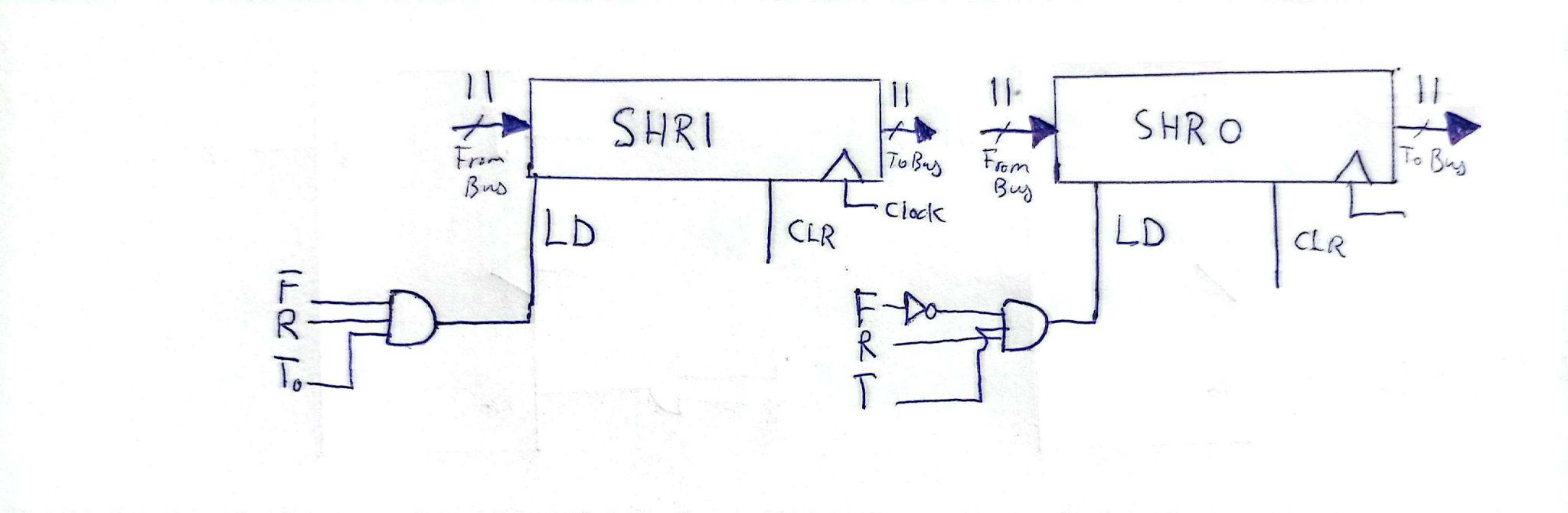
**Control Of PC**

****

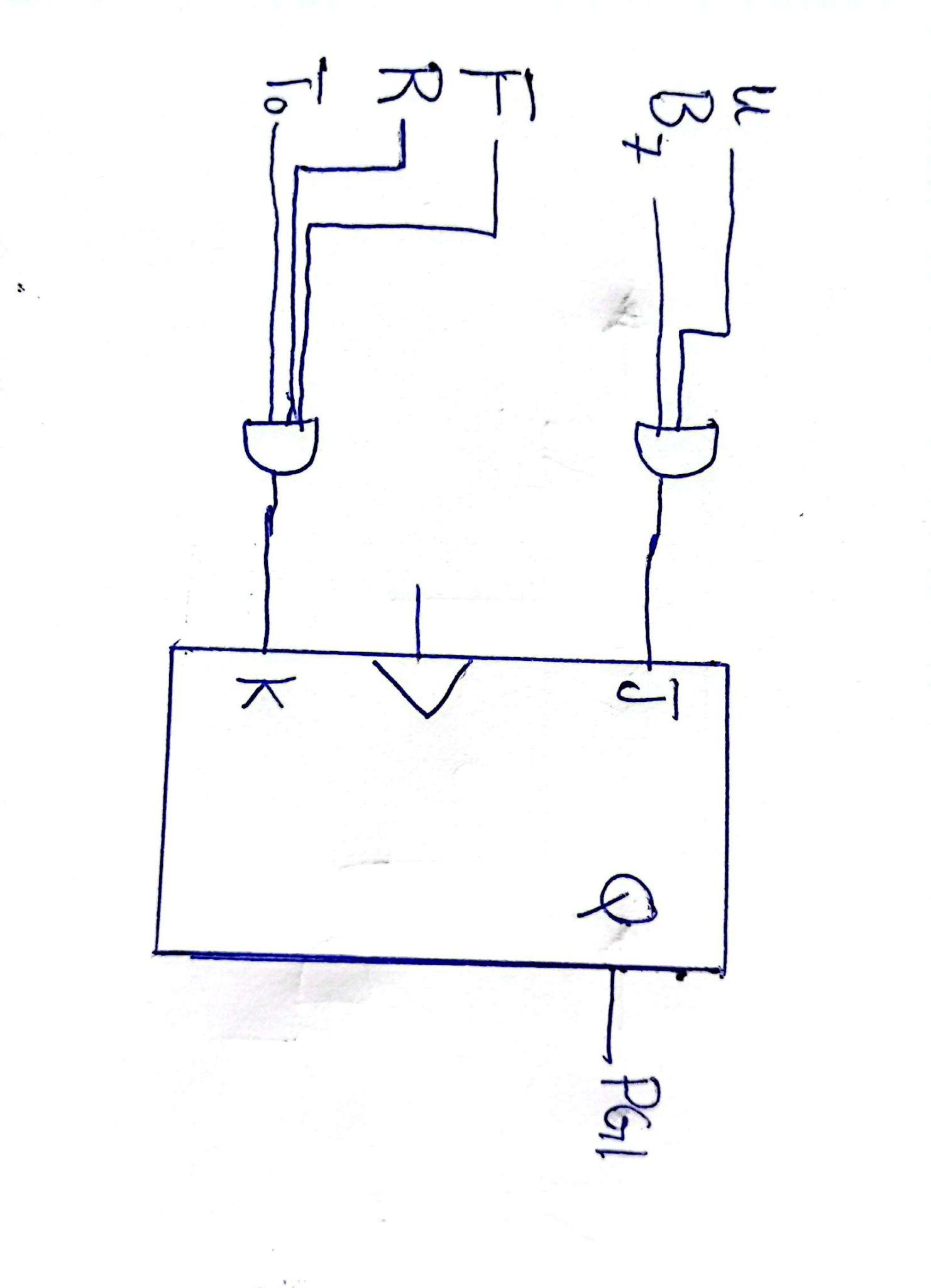
**Control Of IEN**

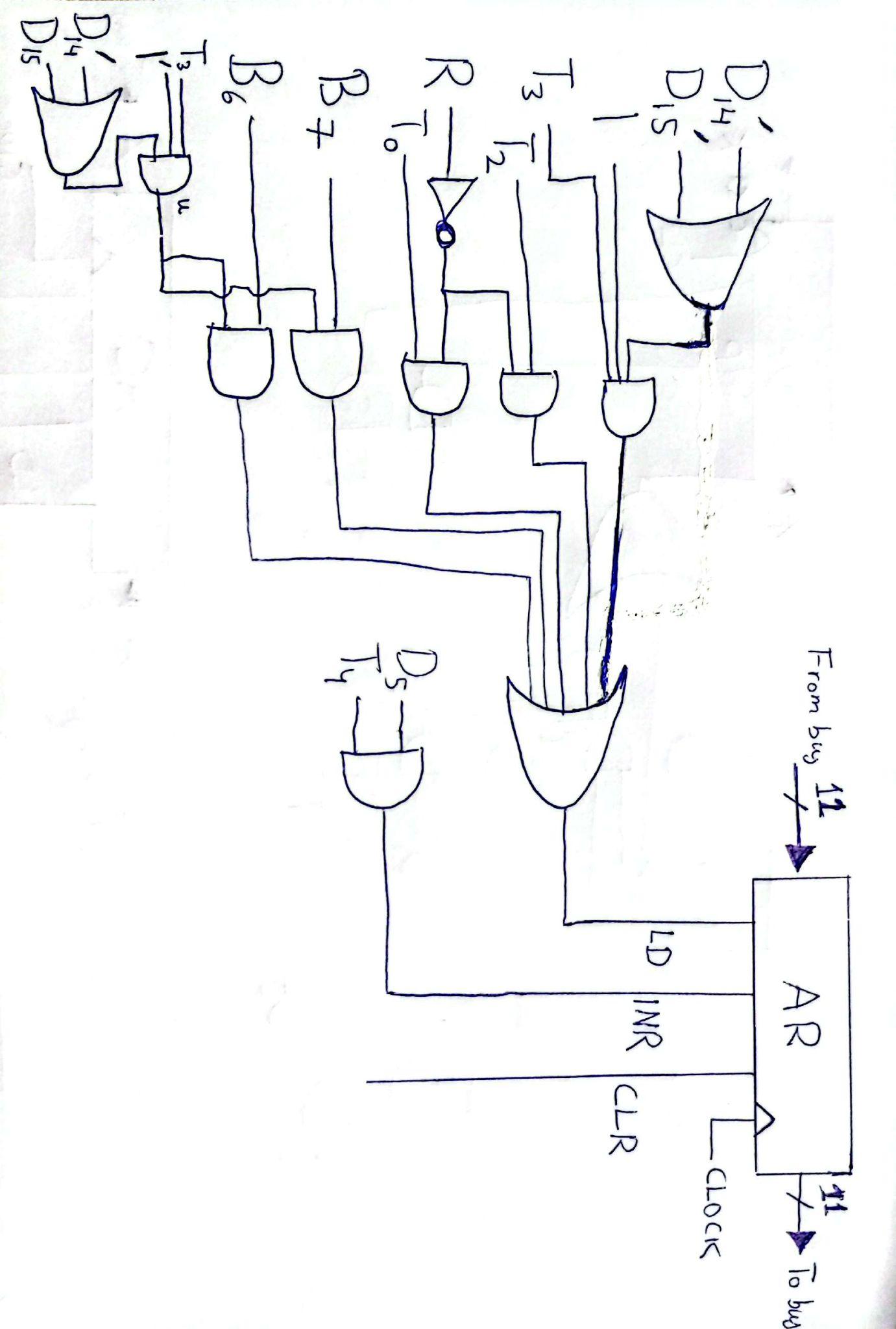
****

**Control of SHRI & SHR**

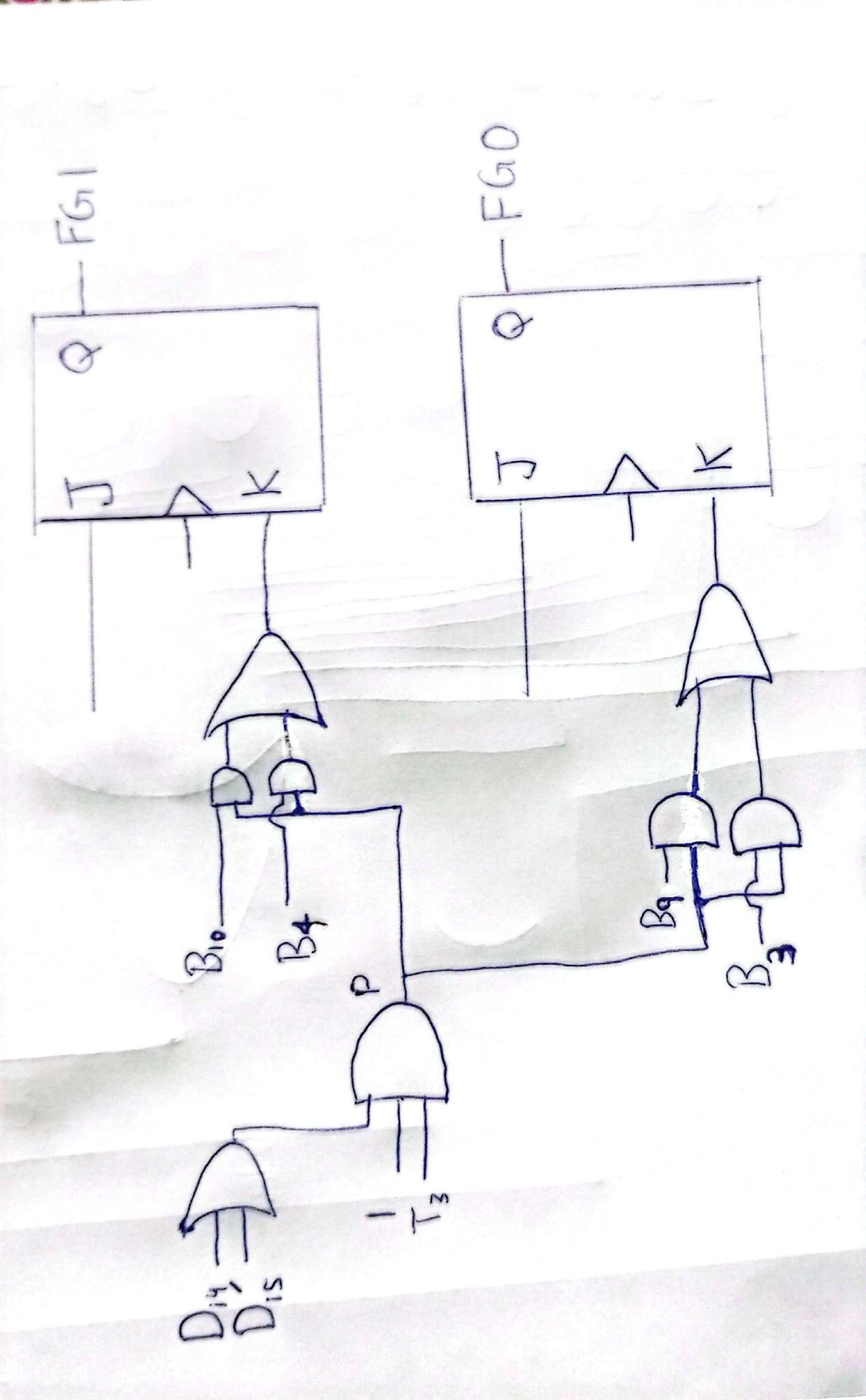
****

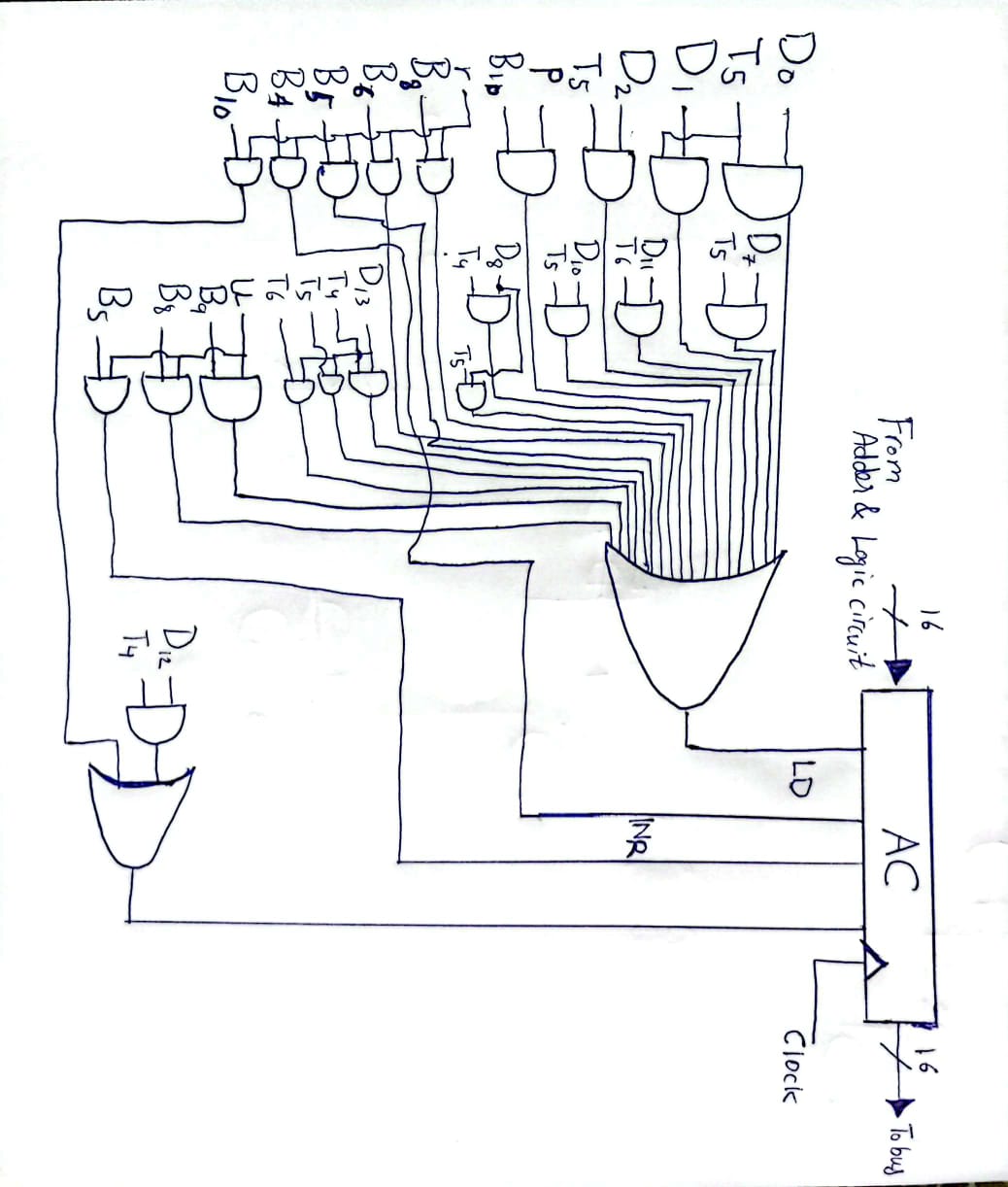
**Control of PGI**

****

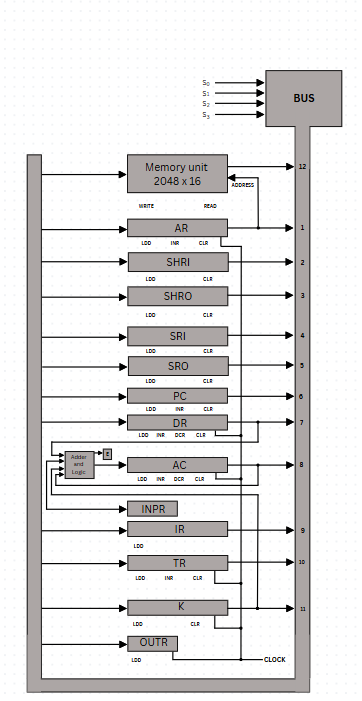
**Control of AR,FGI,FGO**

**Control Of FGI & FGO**

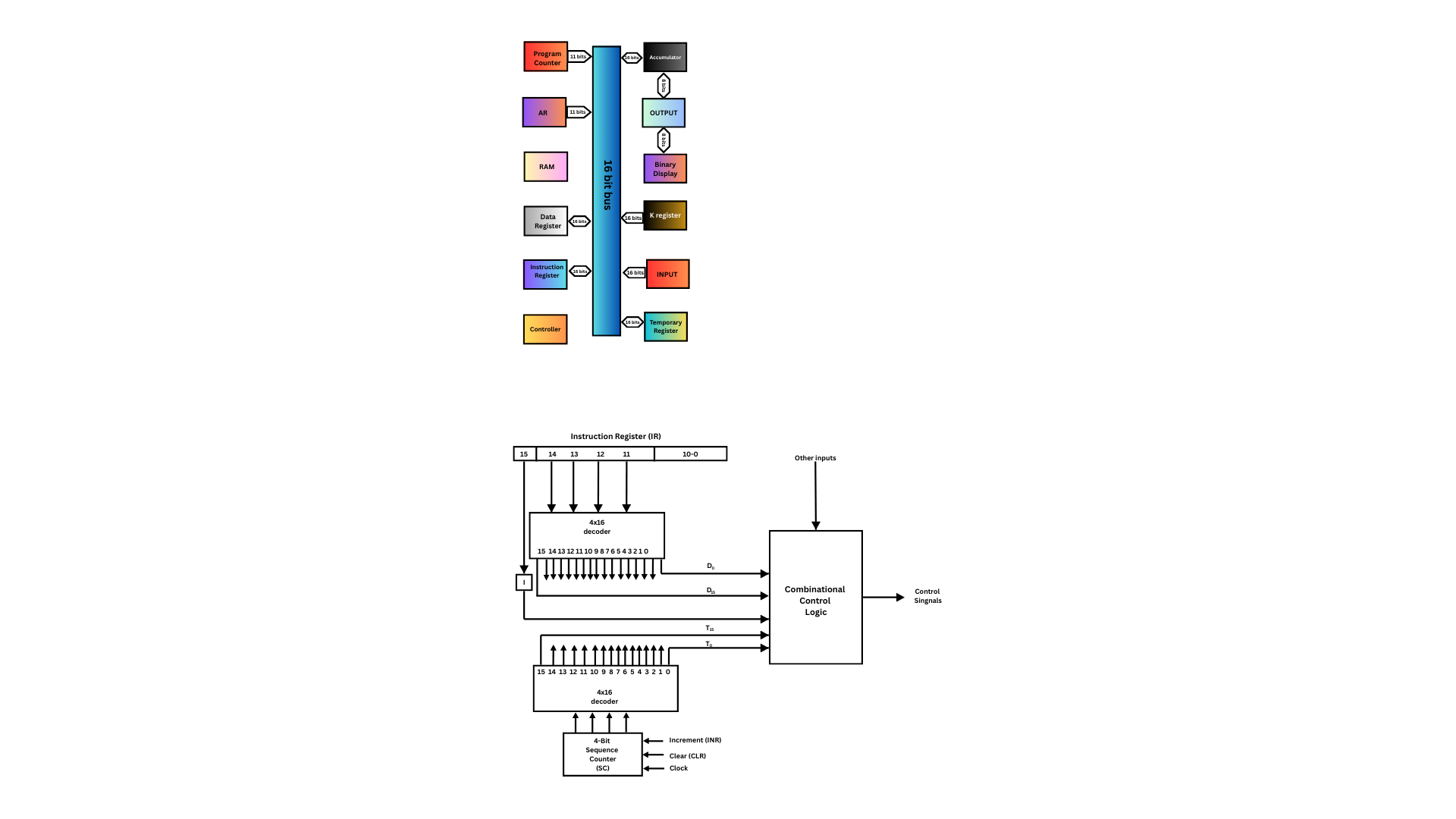


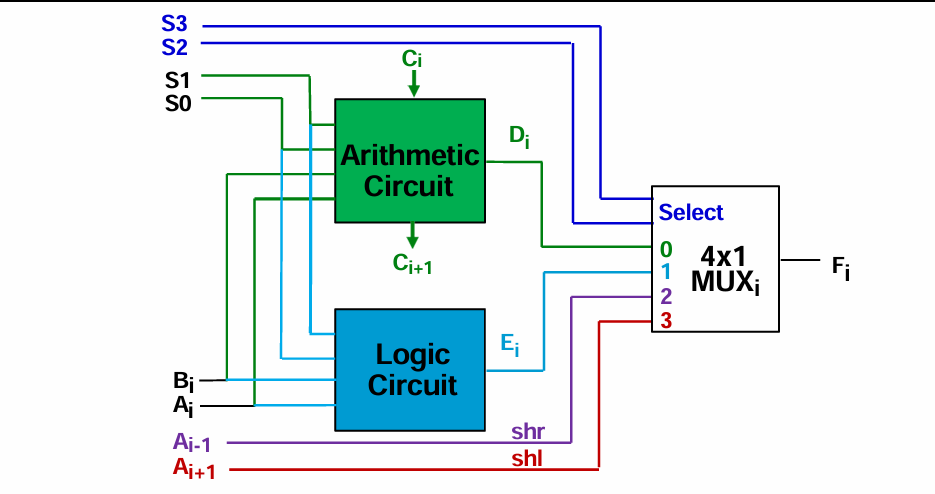
**Control of AC**

**BUS**

****

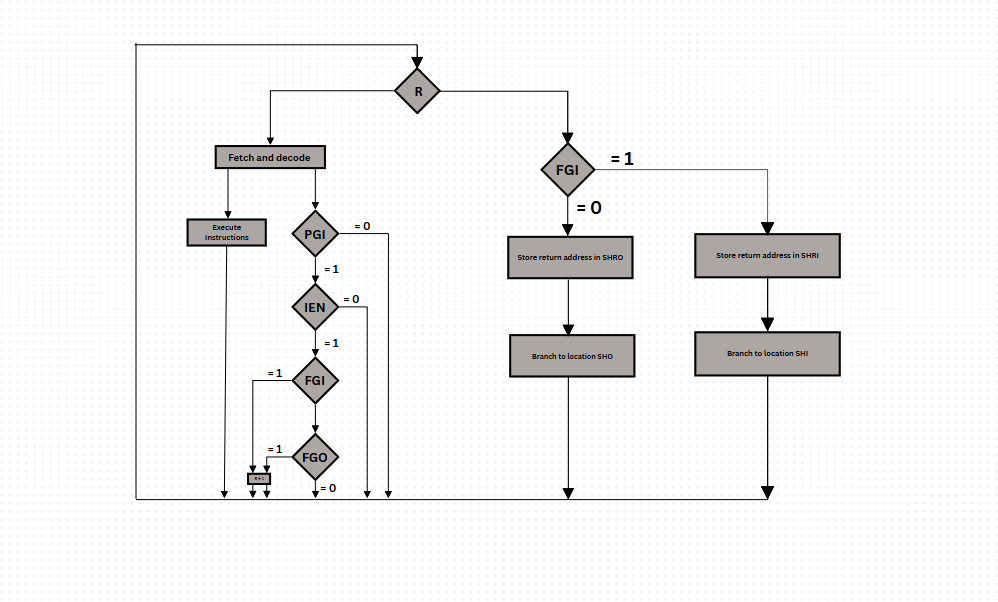
**Control Unit**





**ALU**

**Interrupt initiated and Program controlled I/O**

****

In computer systems, I/O operations can be done via two ways, program controlled I/O, and interrupt initiated I/O. There are advantages and trade offs for both mechanisms and implementing one versus the other directly affects system efficiency, CPU utilization and responsiveness to external devices.

**Program-Controlled I/O**

In an example of the program controlled I/O, also referred to as the polling, the processor monitors the status of the I / O device for the available data transfer. To this purpose, the flag registers are used, such as FGI (Input Flag) and FGO (Output Flag). Once such an instruction as IN or OUT is executed, the processor makes a loop to check these flags again and again until the device lets the processor know that it is ready.

This is a simple implementation without loss of mechanism, but it still is inefficient since it wastes lots of CPU cycles in the polling pages as described in Morris Mano’s Computer System Architecture. Waiting for the I/O device to become ready results in the processor on the CPU remaining idle, a particularly bad thing when I/O devices are slow in relation to the speed of the CPU.

### 

### **Interrupt-Initiated I/O**

In traditional computer systems, Input/Output (I/O) operations are either handled through **program-controlled polling** or **interrupt-initiated mechanisms**. While polling is simple, it is inefficient as it forces the CPU to waste valuable cycles checking device readiness.

In contrast, **interrupt-initiated I/O**, as discussed in Morris Mano’s *Computer System Architecture*, allows the CPU to respond only when an external device signals its readiness. The CPU temporarily suspends the current execution, saves the program state, and jumps to an Interrupt Service Routine (ISR). After servicing the I/O device, the CPU resumes its previous operation, leading to much greater efficiency.

**Key Features of Interrupt-Initiated I/O (from Mano):**

* Utilizes an **Interrupt Enable (IEN)** flip-flop to control whether interrupts are allowed.
* Adds an **Interrupt Cycle** phase after the Execute Cycle.
* Automatically **saves the return address** and redirects execution to the appropriate ISR.

## **Prioritized Nested Interrupt System (Our Enhancement)**

In the *Von 2.0* processor, we introduce a **prioritized interrupt mechanism** to further optimize I/O handling, ensuring that **input operations are given precedence over output operations**.

This is achieved using a special control flag called **PGI (Prioritized Flag Input)**. The logic is structured as follows:

**Interrupt Condition**:  
Interrupt occurs if (PGI ⋅ IEN ⋅ (FGI + FGO)) = 1

* **Prioritization Behavior**:  
  + The system can **interrupt the main program for output**.
  + It can **further interrupt the output service routine for input**.
  + However, **input service routines cannot be interrupted by output requests** (input has higher priority).
* Upon an **input interrupt**, the **PGI flag is cleared to 0**, and **IEN** is also disabled, **preventing any further nesting** until the input is fully serviced.

This ensures that **critical input data is never delayed** due to lower-priority output tasks, enhancing real-time responsiveness.

## **Shadow Registers for Context Preservation**

To properly manage nested interrupts without corrupting program flow, four dedicated shadow registers have been added:

| **Register** | **Purpose** |
| --- | --- |
| **SHRI** | Stores the return address for input interrupts |
| **SHRO** | Stores the return address for output interrupts |
| **SHI** | Stores the address of the input ISR |
| **SHO** | Stores the address of the output ISR |

These registers allow the CPU to store both return addresses and interrupt handler addresses, enabling fast and conflict-free context switching during interrupt servicing.

## **New Micro-Operations for Interrupt Management**

We introduce two specialized micro-operations to facilitate returning from interrupts:

* **SIN (Service Input Return)**:  
   Restores the PC and AR from the **SHRI** register after an input interrupt, re-enabling nested interrupts by resetting PGI.
* **SON (Service Output Return)**:  
   Restores the PC and AR from the **SHRO** register after servicing an output interrupt.

These instructions ensure that after an interrupt is handled, the system can cleanly and efficiently resume the previously running program or lower-priority ISR.

## 

## **Our Optimized I/O Model Summary**

Our final I/O handling model in *Von 2.0* combines:

* **Microcoded IIN/IOT instructions**:  
   Checking flags (FGI/FGO) at runtime to conditionally skip waiting.
* **Memory-mapped I/O**:  
   Treating devices as memory addresses, enabling simpler LDA/STA operations.
* **Prioritized Nested Interrupts**:  
   Ensuring critical input is serviced immediately without being blocked.
* **Dedicated Shadow Registers**:  
   Efficiently saving and restoring execution contexts during interrupts.

This hybrid design **maximizes CPU efficiency** without introducing the full complexity of modern interrupt vectoring, making it ideal for both **educational learning** and **practical embedded system design**.

**Complete Computer Description**

It is a Von 2.0 16 bit stored program memory 2,048 word machine. Its single address format contains 14 registers: AR, PC, DR, AC, IR, TR, K, INPR, OUTR, SC, SHRI, SHRO, SHI, SHO, and 8 control flip flops: I, E, R, FGI, FGO, PGI, IEN, S. REC/RDC has a common 16-bit bus for data transfer to an ALU with arithmetic, logic, shift, and cryptographic capabilities. It has memory mapped I/O and supports prioritised interrupts through shadow registers (SHRI/SHRO). Operations are sequentially executed from the control unit by a 4-bit sequence counter (SC) and hardwired logic.

This design extends Mano’s basic computer with:

* Enhanced ALU: Adds XOR, SQR, and bit-reversal.
* Secure I/O: Hardware-accelerated REC/RDC instructions.
* Optimized Control: Hybrid interrupt/polling via PGI flag.

The *Von 2.0* system extends the classical Basic Computer by incorporating modern design principles:

* A richer instruction setPrioritized interrupt nesting
* Efficient cryptographic routines
* Memory-mapped I/O and microcoded control

It serves as both a practical processor model and an educational architecture that bridges textbook principles with hands-on enhancements.

**Bibliography**

1. Mano, M. M. (2007). Computer System Architecture (3rd ed.). PearsonEducation.
2. Stallings, W. (2022). Computer Organization and Architecture: Designing for Performance (11th ed.). Pearson Education.
3. GeeksforGeeks. (2024, January 15). Introduction of the control unit and its design. https://www.geeksforgeeks.org/introduction-of-control-unit-and-its-design/
4. Limeup. (2023). ALU (Arithmetic Logic Unit). <https://limeup.io/glossary/alu/>
5. Wikipedia contributors. (2023, November 12). System bus. In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/System\_bus

**NIGGA MAHAD**