Table of Contents

Control Unit

1. Types of control organization

The timing for all registers in the basic computer is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator. There are two major types of control organization: hardwired control and microprogrammed control :

1.1 hardwired control

In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. I n the microprogrammed organization, the control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations. A hardwired control, as the name implies, re quires changes in the wiring among the various components if the design has to be modified or changed

1.2 microprogrammed control

In the microprogrammed control, any required changes or modifications can be done by updating the microprogram in control memory.

Comparison of Hardwired and Microprogrammed Control Units

Feature	**Hardwired Control Unit**	**Microprogrammed Control Unit**
Design	Implemented using fixed combinational and sequential logic circuits (flip-flops, gates, etc.).	Uses a set of instructions (microinstructions) stored in memory to generate control signals.
Flexibility	Difficult to modify or update since changes require redesigning the hardware.	Easier to modify or update by rewriting the microprogram.
Complexity	Becomes complex for processors with a large instruction set.	Relatively simple to design and extend for complex instruction sets.
Speed	Faster, as control signals are generated directly through logic circuits.	Slower due to the need to fetch and decode microinstructions.
Cost	More expensive for complex designs because of hardware requirements.	Generally less expensive due to the use of programmable memory.
Reliability	High, as it relies on fixed hardware with fewer moving parts.	Slightly less reliable due to dependencies on memory.
Usage	Found in RISC (Reduced Instruction Set Computing) processors.	Commonly used in CISC (Complex Instruction Set Computing) processors.

2. Control Unit

The block diagram of the control unit is shown in **Fig 1.1** It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR). The instruction register is shown again in **Fig 1.1,** where it is divided into three parts: the I bit, the operation code, and bits 0 through 11. The operation code in bits 12 through 14 are decoded with a 3 x 8 decoder. The eight outputs of the decoder are designated by the symbols D0 through D7 The subscripted decimal number is equivalent to the binary value of the corresponding operation code. Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the counter are decoded into 16 timing signals T0 through T_{15} The internal logic of the control gates will be derived later when we consider the design of the computer in detail .

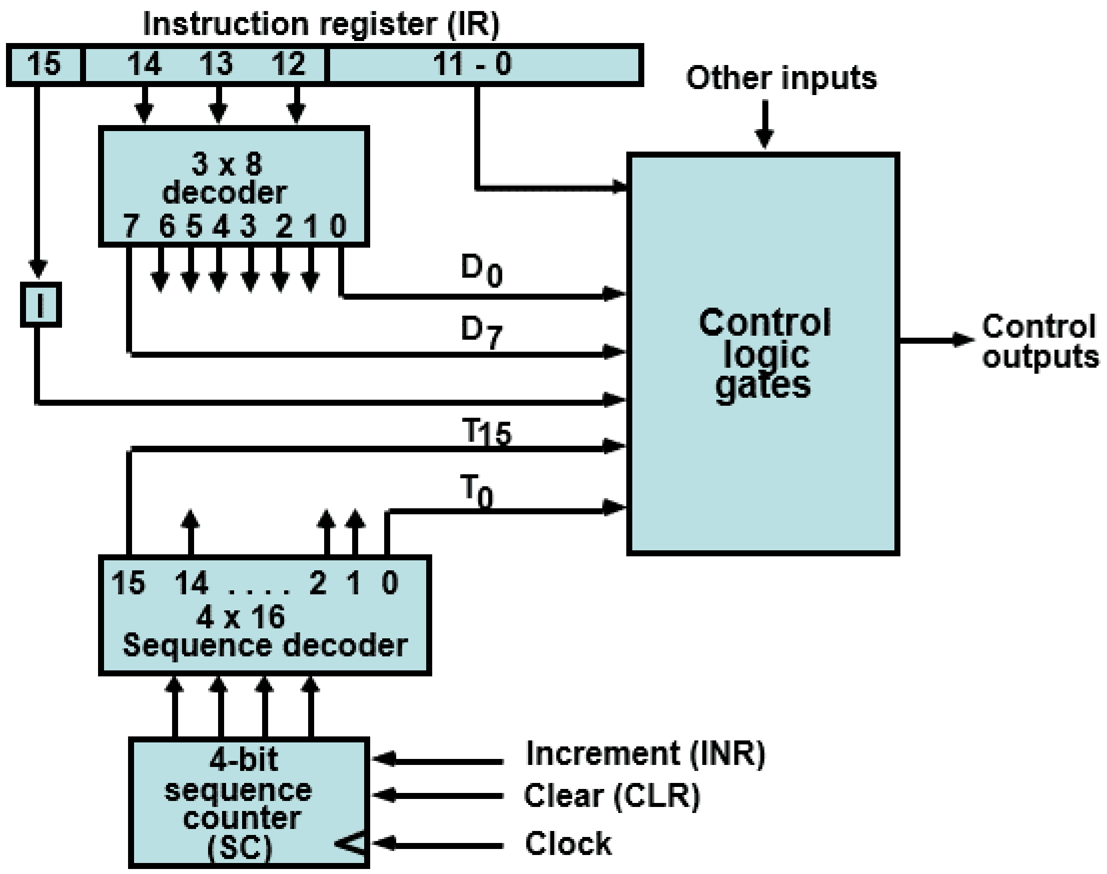


Figure 1.1

2.1 sequence counter SC

Counters employed in digital systems quite often require a parallel load capability for transferring an initial binary number prior to the count operation. **Figure 1.2** shows the logic diagram of a binary counter that has a parallel load capability and can also be cleared to 0 synchronous with the clock. When equal to D, the clear input sets all the K inputs to 1, thus clearing all flip-flops with the next clock transition. The input load control when equal to 1, disables the count operation and causes a transfer of data from the four parallel inputs into the four flip-flops (provided that the clear input is 0). If the clear and load inputs are both 0 and the increment input is 1, the circuit operates as a binary counter.

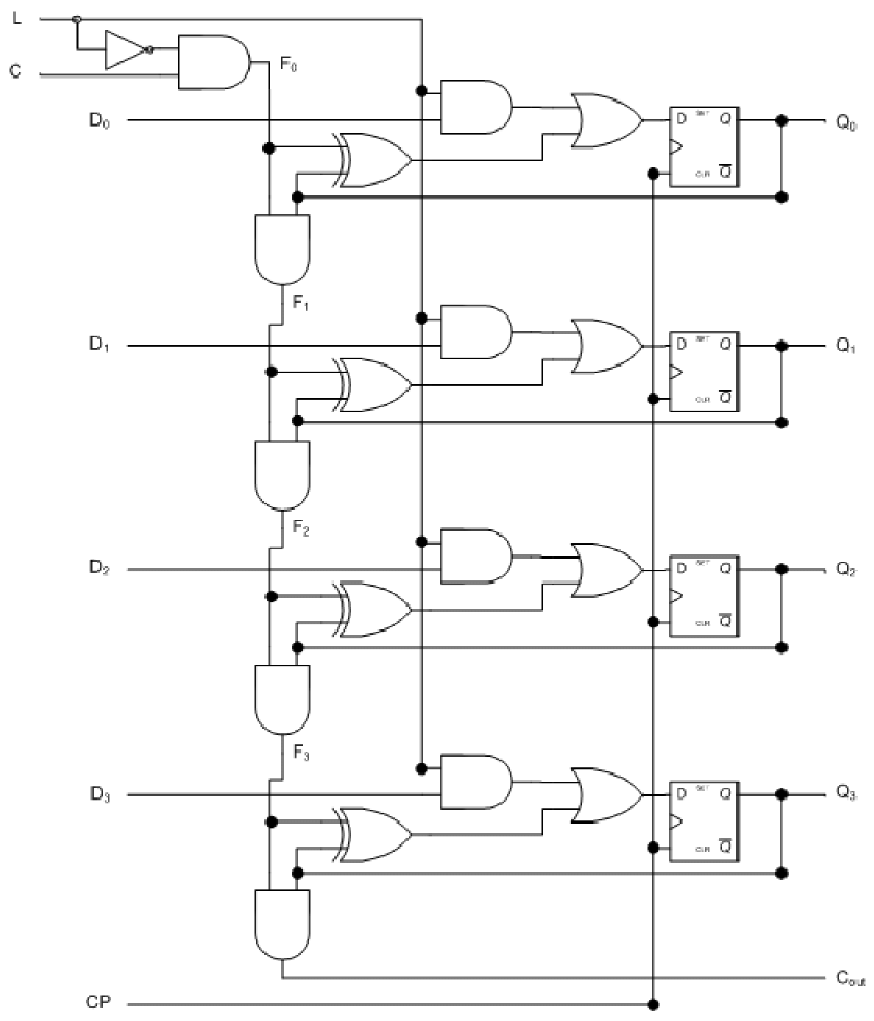


Figure 1.2 4-bit binary counter with parallel load and synchronous clear

The sequence counter SC can be incremented or cleared synchronously (see the counter of **Fig 1.2**). Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 x 16 decoder. Once in a while, the counter is cleared to 0, causing the next active timing signal to be To. As an example, consider the case where SC is incremented to provide timing signals T_0 , T_1 , T_2, T_3 , T_4 in sequence. At time T_4, SC is cleared to 0 if decoder output D_3 is active. This is expressed symbolically by the statement :

$$\boxed{\mathbf{D_3T_4 : SC \rightarrow 0}}$$

The timing diagram of **Fig 1.3** shows the time relationship of the control signals. The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition of the clock clears SC to 0, which in turn activates the timing signal T_0 out of the decoder. T_0 is active during one clock cycle. The positive clock transition labeled T_0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T_0. SC is incremented with every positive clock transition, unless its CLR input is active. This produces the sequence of timing signals T_0 , T_1 , T_2, T_3 , T_4 and so on, as shown in the diagram. (Note the relationship between the timing sign) and its corresponding positive clock transition. If SC is not cleared, the timing signals will continue with T_5 , T_6 , up to T_{15} and back to T_0

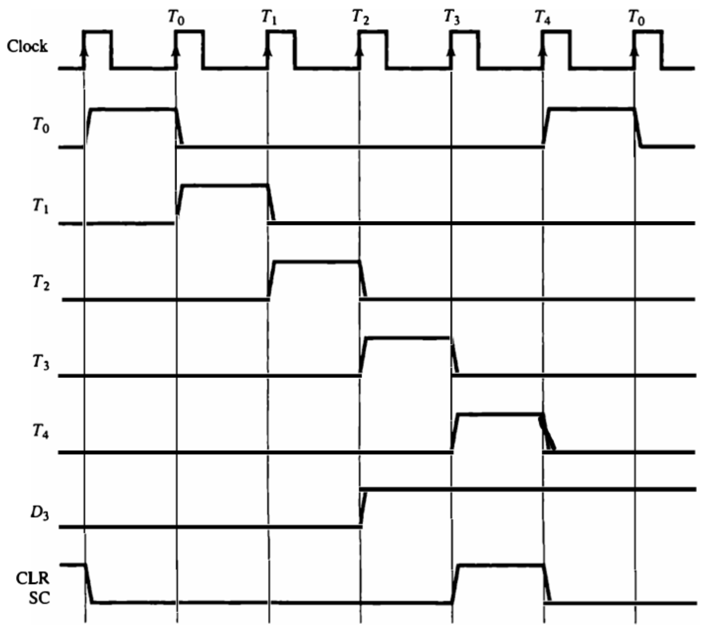


Figure 1.3 Example of control timing signals.

The last three waveforms in **Fig 1.3** show how SC is cleared when $D_3T_4 = 1$. Output D_3 from the operation decoder becomes active at the end of timing signal T_2. When timing signal T_4 becomes active, the output of the AND gate that implements the control function D_3T_4 becomes active. This signal is applied to the CLR input of SC. On the next positive clock transition (the one marked T, in the diagram) the counter is cleared to 0. This causes the timing signal T_0 to become active instead of T_5 that would have been active if SC were incremented instead of cleared. A memory read or write cycle will be initiated with the rising edge of a timing signal. It will be assumed that a memory cycle time is less than the clock cycle time. According to this assumption, a memory read or write cycle initiated by a timing signal will be completed by the time the next clock goes through its positive transition. The clock transition will then be used to load the memory word into a register. This timing relationship is not valid in many computers because the memory cycle time is usually longer than the processor clock cycle. In such a case it is necessary to provide wait cycles in the processor

until the memory word is available. T_0 facilitate the presentation, we will assume that a wait period is not necessary in the basic computer. T_0 fully comprehend the operation of the computer, it is crucial that one understands the timing relationship between the clock transition and the timing signals. For example, the register transfer statement

$$\boxed{\mathbf{T_0 : \ AR \leftarrow PC}}$$

specifies a transfer of the content of PC into AR if timing signal T_0 is active. T_0 is active during an entire clock cycle Interval During this time the content of PC is placed onto the bus (with $S_2S_1S_0 = 010$) and the LD (load) input of AR is enabled. The actual transfer does not occur until the end of the clock cycle when the clock goes through a positive transition. This same positive clock transition increments the sequence counter SC from 0000 to 0001. The next clock cycle has T_1 active and T_0 inactive.

2.2 Control Logic Gates

The block diagram of the control logic gates is shown in **Fig 1.2** The inputs to this circuit come from the two decoders, the I flip-flop, and bits 0 through 11 of IR. The other inputs to the control logic are: AC bits 0 through 15 to check if AC = 0 and to detect the sign bit in AC [15] , DR bits 0 through 15 to check if DR = 0; and the values of the seven flip-flops. The outputs of the control logic circuit are:

1. Signals to control the inputs of the nine registers

2. Signals to control the read and write inputs of memory

3 Signals to set, clear, or complement the flip-flops

4. Signals for S_2, S_1, and S_0 to select a register for the bus

5. Signals to control the AC adder and logic circuit

2.2.1 Control of Registers and Memory

The control inputs of the registers are LD (load), INR (increment), and CLR (clear). we want to derive the gate structure associated with the control inputs of AR. We have to find all the statements that change the content of AR according to the **Table 2.1**

$T_0 : AR \leftarrow PC$
$R'T_2 : AR \leftarrow IR(0\text{-}11)$
$D_7IT_3 : AR \leftarrow M[AR]$
$RT_0 : AR \leftarrow 0$
$D_5T_4 : AR \leftarrow AR + 1$

The first three statements specify transfer of information from a register or memory to AR. The content of the source register or memory is placed on the bus and the content of the bus is transferred into AR by enabling its LD control input. The fourth statement clears AR to 0. The last statement increments AR by 1 The control functions can be combined into three Boolean expressions as follows:

$LD(AR) = R'T_0 + R'T_2 + D7'IT_3$

$CLR(AR) = RT_0$

$INR(AR) = D_5T_4$

where LD(AR) is the load input of AR, CLR(AR) is the clear input of AR, and INR(AR) is the increment input of AR. The control gate logic associated with AR is shown in **Fig 1.4**

In a similar fashion we can derive the control gates for the other registers as well as the logic needed to control the read and write inputs of memory. The logic gates associated with the read input of memory is derived by finding the statements that specify a read operation. The read operation is recognized from the symbol $\leftarrow$ M[AR] .

$$\textbf{Read} = R'T_1 + D7'IT_3 + (D_0 + D_1 + D_2 + D_6)T_4$$

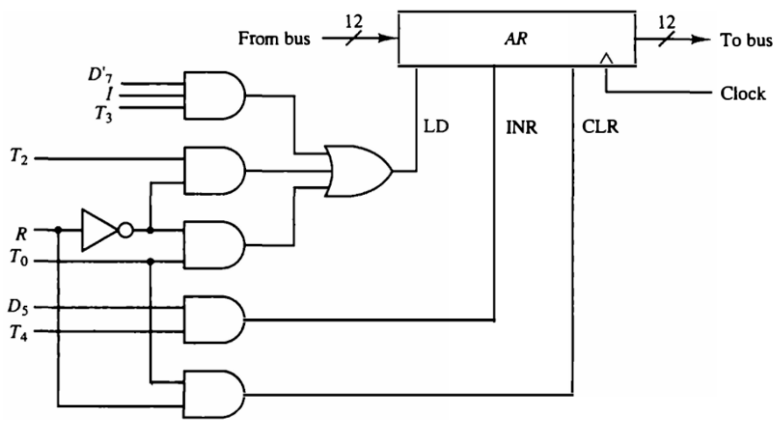


Figure 1.4 Control gates associated with AR.

2.2.2 Control of Single Flip-flops

The control gates for the seven flip-flops can be determined deriving the gate structure associated with the control inputs of flip flop. We have find all the statements that change the content of flip flop for example that IEN may change as a result of the two instructions ION and IOF according to the **Table 2.1**

$D_7I'T_3$ = r (common to all register-reference instructions)
D_7IT_3 = p (common to all input-output instructions)
IR(i) = B_i [bit in IR(0-11) that specifies the operation]
xawd

$pB_7 : IEN \leftarrow 1$

$pB_6 : IEN \leftarrow 0$

where p = D_7IT_3 and B_7 and B_6 are bits 7 and 6 of IR, respectively. Moreover, at the end of the interrupt cycle lEN is cleared to 0.

If a JK flip-flip for IEN is used the control gate logic will be as shown in **Fig 1.5**

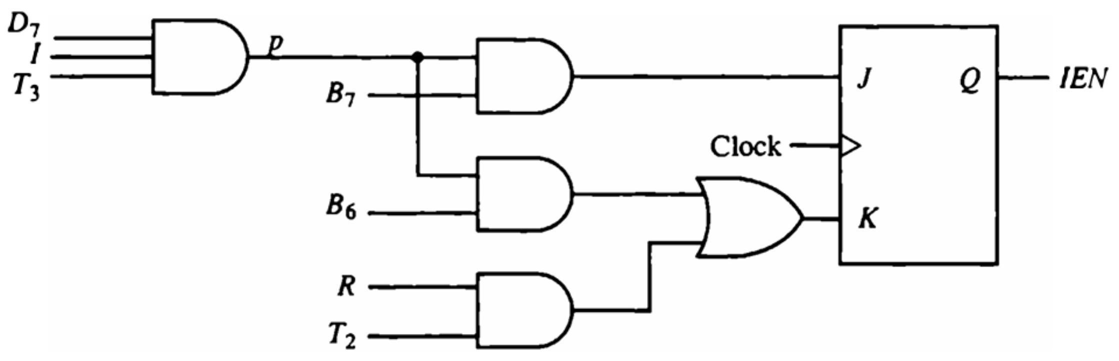


Fig 1.5 Control inputs for lEN

$\boxed{\begin{array}{l} \mathbf{J = pB_7} \\ \mathbf{K = (R\,T_2) + (p\,B_6)} \end{array}}$

J	K	Q (Next State)
0	0	No Change
0	1	Reset (Q=0)
1	0	Set (Q=1)
1	1	Toggle (Q=Q')

Table 1.1 Truth Table for J-K Flip-Flop

2.2.3 Control of Common Bus

The 16-bit common bus is controlled by the selection inputs S_2, S_1 and S_3 The decimal number shown with each bus input specifies the equivalent binary number that must be applied to the selection inputs in order to select the corresponding register. **Table 1.2** specifies the binary numbers for $S_2S_1S_0$ that select each register. Each binary number is associated with a Boolean variable x_1 through x_7, corresponding to the gate structure that must be active in order to select the register or memory for the bus. For example, when $x_1 = 1$, the value of $S_2S_1S_0$ must be 001 and the output of AR will be selected for the bus. **Table 1.2** is recognized as the truth table of a binary encoder. The placement of the encoder at the inputs of the bus selection logic is shown in **Fig. 1.6** The Boolean functions for the encoder are

Input							Output			Register selected for bus
x_1	x_2	x_3	x_4	x_5	x_6	x_7	S_0	S_1	S_2	______
0	0	0	0	0	0	0	0	0	0	None
1	0	0	0	0	0	0	0	0	1	AR
0	1	0	0	0	0	0	0	1	0	PC
0	0	1	0	0	0	0	0	1	1	DR
0	0	0	1	0	0	0	1	0	0	AC
0	0	0	0	1	0	0	1	0	1	IR
0	0	0	0	0	1	0	1	1	0	TR
0	0	0	0	0	0	1	1	1	1	MEM

Table 1.2 Encoder for Bus Selection Circuit

$$S_0 = x_1 + x_3 + x_5 + x_7$$
$$S_1 = x_2 + x_3 + x_6 + x_7$$
$$S_2 = x_4 + x_5 + x_6 + x_7$$

To determine the logic for each encoder input, it is necessary to find the control functions that place the corresponding register onto the bus. For example, to find the logic that makes $x_1 = 1$, we scan all register transfer statements and extract those statements that have AR as a source according to the **Table 2.1**

D_4T_4 : PC $\leftarrow$ AR

D_5T_5: PC $\leftarrow$ AR Therefore, the Boolean function for x_1 is $\boxed{x_1 = D_4T_4 + D_5T_5}$

The data output from memory are selected for the bus when $x_7 = 1$ and $S_2S_1S_0 = 111$. The gate logic that generates x_7 must also be applied to the read input of memory. Therefore, the Boolean function for x_7 is the same as the one derived previously for the read operation.

$x_7 = R'\,T_1 + D_7IT_3 + (D_0 + D_1 + D_2 + D_6)\,T_4$

In a similar manner we determined the gate logic for the other registers . the equations of x are :

$x_0 = 0$
$x_1 = D_4T_4+D_5T_5$
$x_2 = R'\,T_0+R'\,T_1 + RT_0 + RT_2 + D_5T_4 + D_6T_6$
$x_3 = D_0T_5 + D_1T_5 + D_2T_5 + D_6T_5 + D_6T_6$
$x_4 = D_0T_5 + D_1T_5 + D_3T_4+ pB[10]$
$x_5 = R'\,T_2$
$x_6 = RT_1$
$x_7 = R'\,T_1 + D_7IT_3 + (D_0 + D_1 + D_2 + D_6)\,T_4$

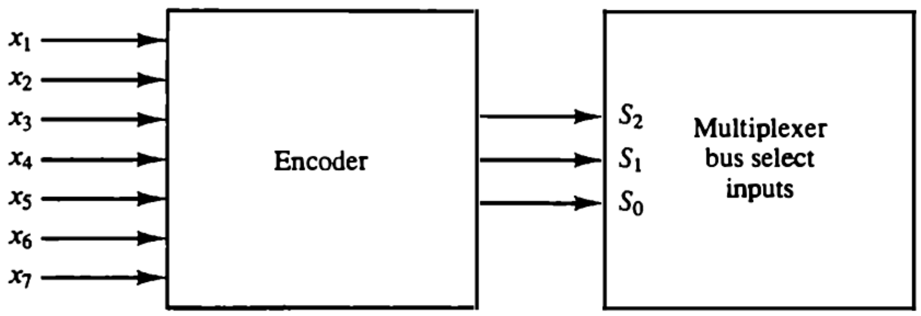


Figure 1.6 Encoder for bus selection inputs

2.2.4 Control of AC Register

The circuits associated with the AC register are shown in **Fig 1.7** The adder and logic circuit has three sets of inputs. One set of 16 inputs comes from the outputs of AC. Another set of 16 inputs comes from the data register DR. A third set of eight inputs comes from the input register INPR. The outputs of the adder and logic circuit provide the data inputs for the register. In addition, it is necessary to include logic gates for controlling the LD, INR, and CLR in the register and for controlling the operation of the adder and logic circuit. In order to design the logic associated with AC, it is necessary to go over the register transfer statements and extract all the statements that change the content of AC according to the **Table**

D_0T_5	**$AC \leftarrow AC \wedge DR$**	**AND with DR**
D_1T_5	**$AC \leftarrow AC + DR$**	**Add with DR**
D_2T_5	**$AC \leftarrow DR$**	**Transfer from DR**
pB_{11}	**$AC(0–7) \leftarrow INPR$**	**Transfer from INPR**
rB_9	**$AC \leftarrow AC$**	**Complement**
rB_7	**$AC \leftarrow shr\ AC, AC(15) \leftarrow E$**	
rB_6	**$AC \leftarrow shl\ AC, AC(0) \leftarrow E$**	
rB_{11}	**$AC \leftarrow 0$**	**Clear**
rB_5	**$AC \leftarrow AC + 1$**	**Increment**

From this list we can derive the control logic gates and the adder and logic circuit

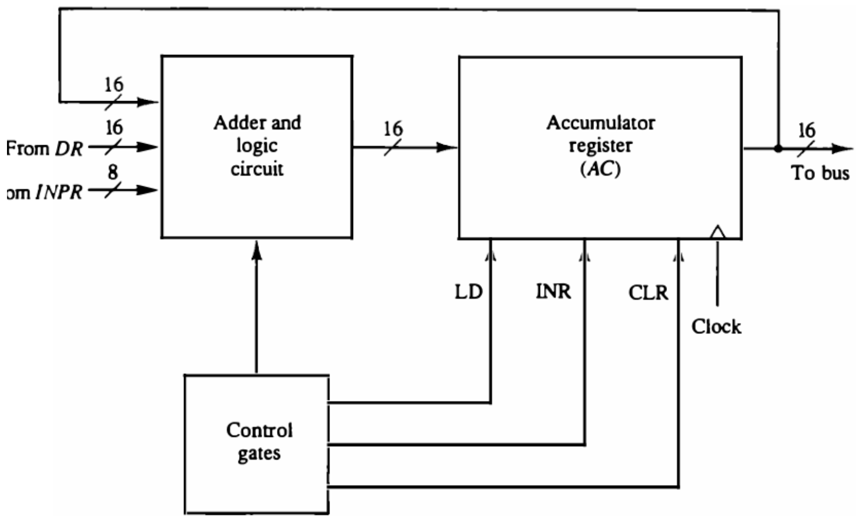


Figure 1.7 Circuits associated with AC

The gate structure that controls the LD, INR, and CLR inputs of AC is shown in **Fig 1.8** The gate configuration is derived from the control functions in the list above. The control function for the clear microoperation is rB_{11} , where $r = D_7I'T_3$ and $B_{11} = IR[11]$. The output of the AND gate that generates this control function is connected to the CLR input of the register. Similarly, the output of the gate that implements the increment microoperation is connected to the INR input of the register. The other seven microoperations are generated in the adder and logic circuit and are loaded into AC at the proper time. The outputs of the gates for each control function is marked with a symbolic name. These outputs are used in the design of the adder and logic circuit.

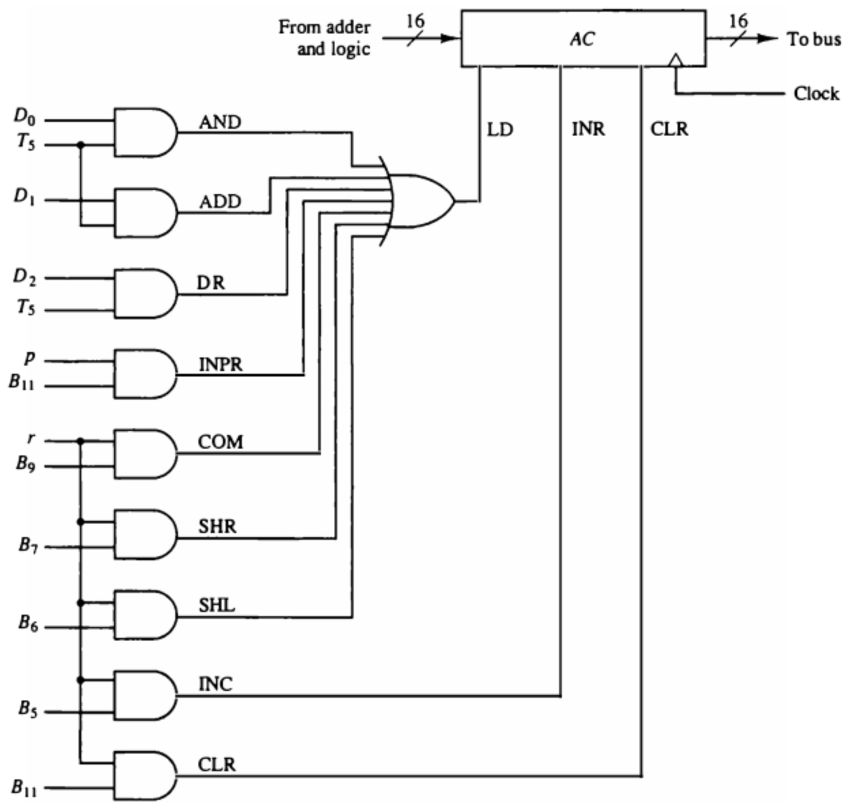


Figure 1.8 Gate structure for controlling the LD, lNR, and CLR of AC

2.3 control signals

An array is used to group multiple control signals for loading operations. each index of the array corresponds to a specific signal, simplifying the design and reducing the number of individually named signals. for example, INC [0] represents the load signal for the accumulator (AC), and load [1] represents the load signal for the data register (TR) , **<u>We add additional signals along with the signals from the previous phase</u>**

2.3.1 INC [5:0]

control signals to increment specific registers

- ✓**INC [0] :** increment accumulator **(AC)**
- ✓**INC [1] :** increment temporary register **(TR)**
- ✓**INC [2] :** increment data register **(DR)**
- ✓**INC [3] :** increment program counter **(PC)**
- ✓**INC [4] :** increment address register. **(AR)**
- ✓**INC [5] :** increment sequence counter **(SC)**

2.3.2 CLR [5:0]

control signals to clear (**reset to 0**) specific registers

- ✓**CLR [0]:** clear accumulator **(AC)**
- ✓**CLR [1]:** clear temporary register **(TR)**
- ✓**CLR [2]:** clear data register **(DR)**
- ✓**CLR [3]:** clear program counter **(PC)**
- ✓**CLR [4]:** clear a memory address register **(AR)**
- ✓**CLR [5]:** clear a sequence counter **(SC)**

2.3.3 RESET_FF [5:0]

control signal to force the output Flip Flop to a value of 0

- ✓**RESET_FF [0]:** clear flip flop **(IEN)**
- ✓**RESET_FF [1]:** clear flip flop **(R)**
- ✓**RESET_FF [2]:** clear flip flop **(START_STOP)**
- ✓**RESET_FF [3]:** clear flip flop **(FGI)**
- ✓**RESET_FF [4]:** clear flip flop **(FGO)**
- ✓**RESET_FF [5]:** clear flip flop **(I)**

2.3.4 Negative detection (NEG)

control signal to detect the negativity of AC , if AC [15] is zero it has **NEG** will carry 0 if AC[15] is one then **NEG** is one

2.3.5 SET [5:0]

control signal to force the Flip Flop to a value of 1

✓**SET[0]:** set flip flop **(IEN)**
✓**SET[1]:** set flip flop **(R)**
✓**SET[2]:** set flip flop **(FGI)**
✓**SET[3]:** set flip flop **(FGO)**
✓**SET[4]:** set flip flop **(I)**
✓**SET[5]:** set flip flop **(S)**

2.3.7 Enable [1:0]

control signal for load data into flip flop when it has a

✓**Enable [0]:** load flip flop **(E)**
✓**Enable [1]:** load flip flop **(I)**

2.3.8 ZERO [2:0]

To check if AC , DR and E has a value of zero

✓**ZERO [0]:** Checks whether the **AC** contains a value of zero
✓**ZERO [1] :** Checks whether the **DR** contains a value of zero
✓**ZERO [2] :** Checks whether the **E** contains a value of zero

---

2.3.9 Memory Signals

control signals for reading from and writing to memory

✓**MEM_Read :** enable reading from memory when it has a value of 1
✓**MEM_Write :** enable writing to the memory when it has a value of 1

2.3.10 Load [6:0]

control signals (**7 bits**) for loading data into specific registers

✓**LD [0]:** load accumulator (**AC**)
✓**LD [1]:** load output register **(OUTR)**
✓**LD [2]:** load temporary register **(TR)**
✓**LD [3]:** load instruction registers (**IR**)
✓**LD [4]:** load Data register **(DR)**
✓**LD [5]:** load program counter register **(PC)**
✓**LD [6]:** load address register **(AR)**

2.3.11 Clock and Reset

✓**ClK** : clock signal used to synchronize operations within the Datapath module
✓**RESET** : initializes or resets the Datapath’s internal states to their default values

---

2.4 Internal Design

2.4.1 Internal design of registers

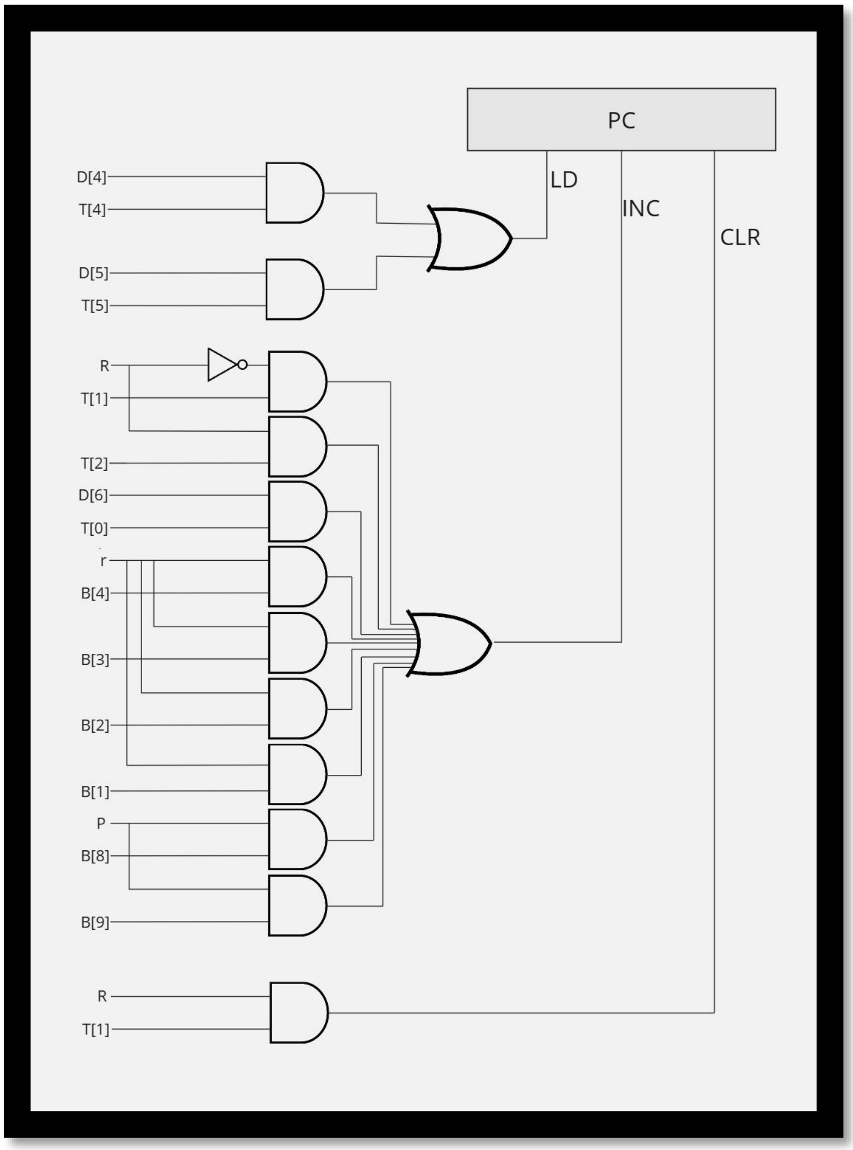


Figure 1.9 Internal design of PC register

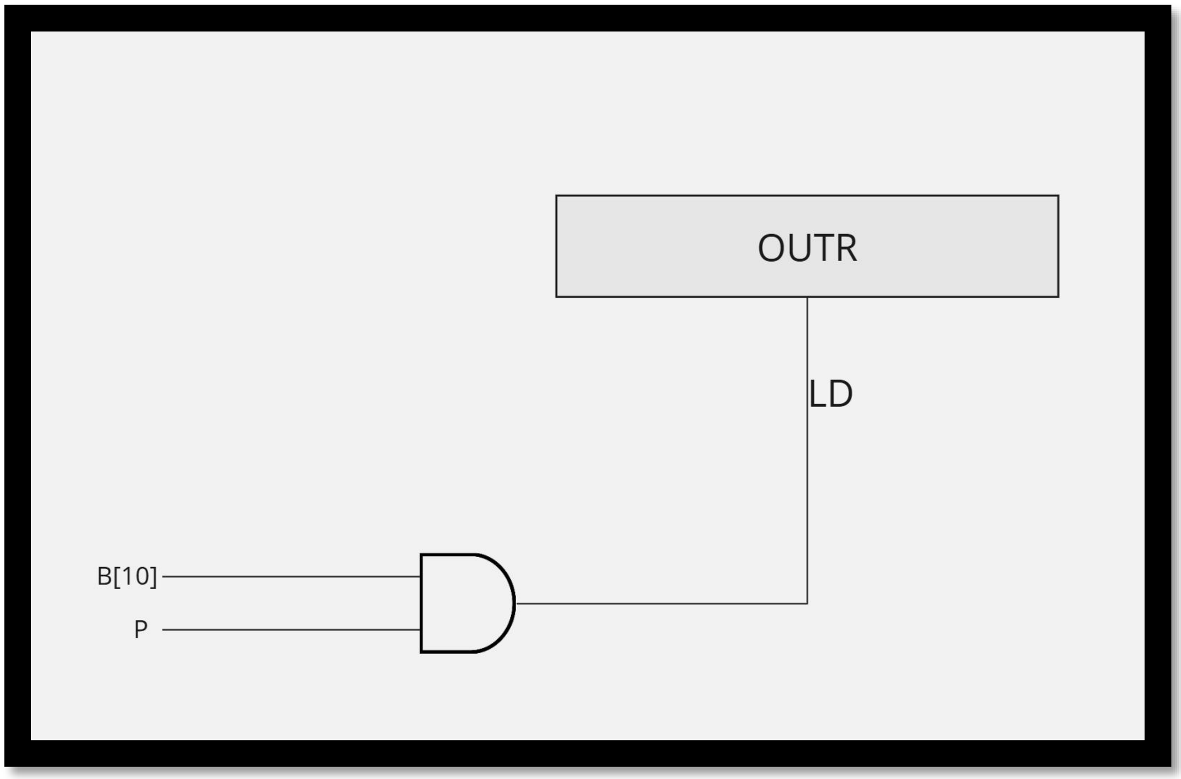


Figure 1.10 Internal design of OUTR register

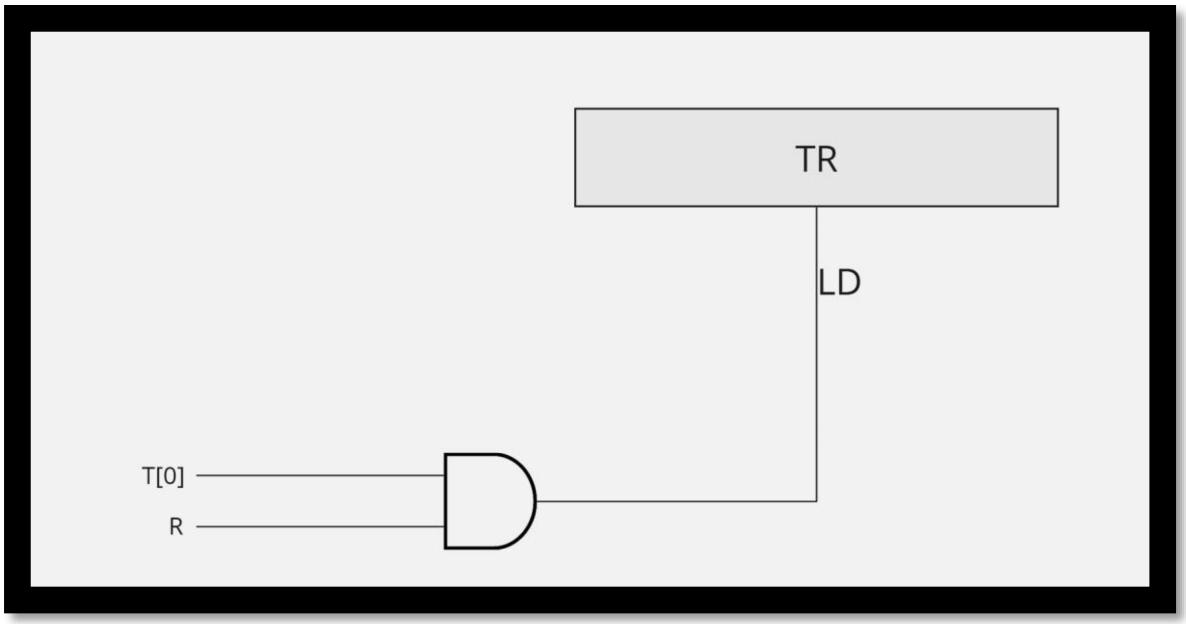


Figure 1.11 Internal design of TR register

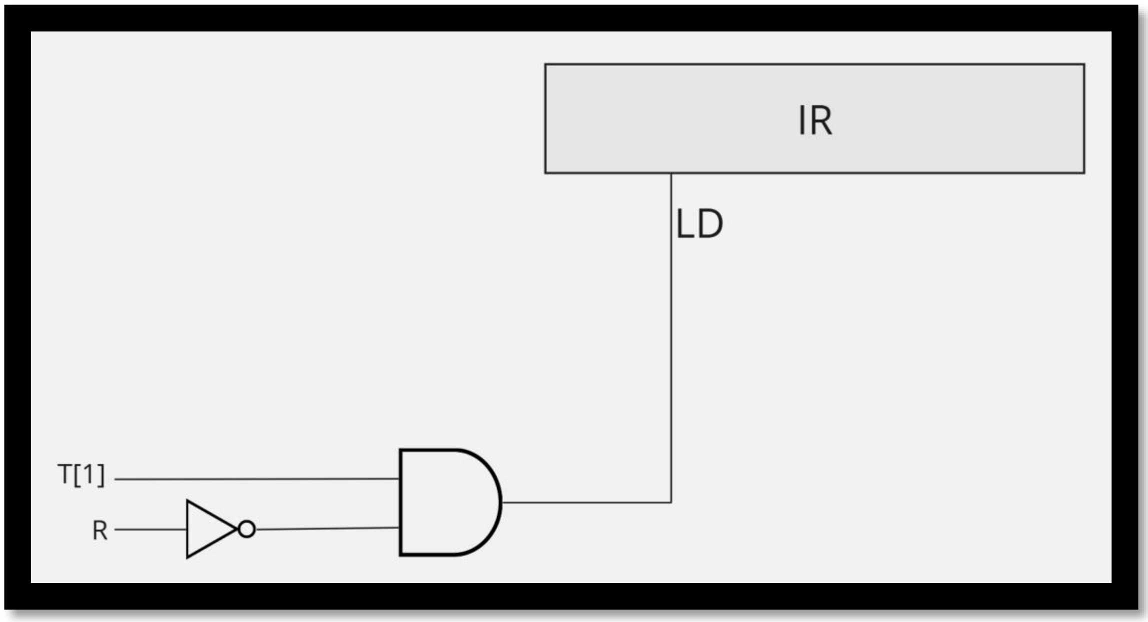


Figure 1.12 Internal design of IR register

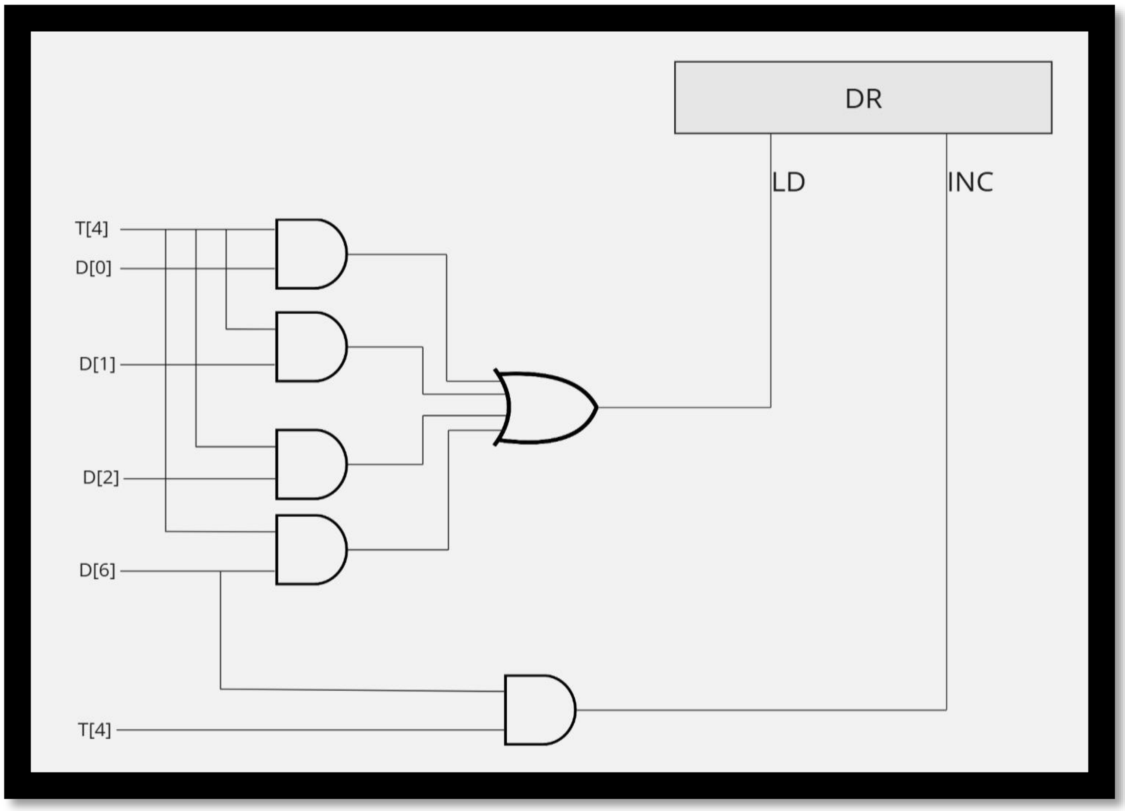


Figure 1.13 Internal design of DR register

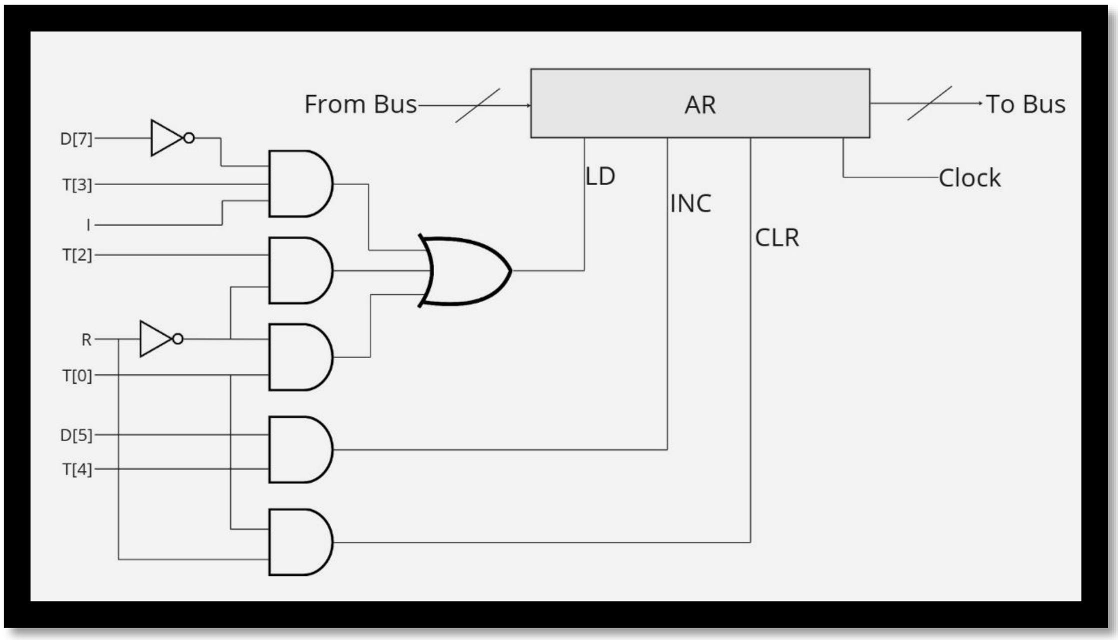


Figure 1.14 Internal design of AR register

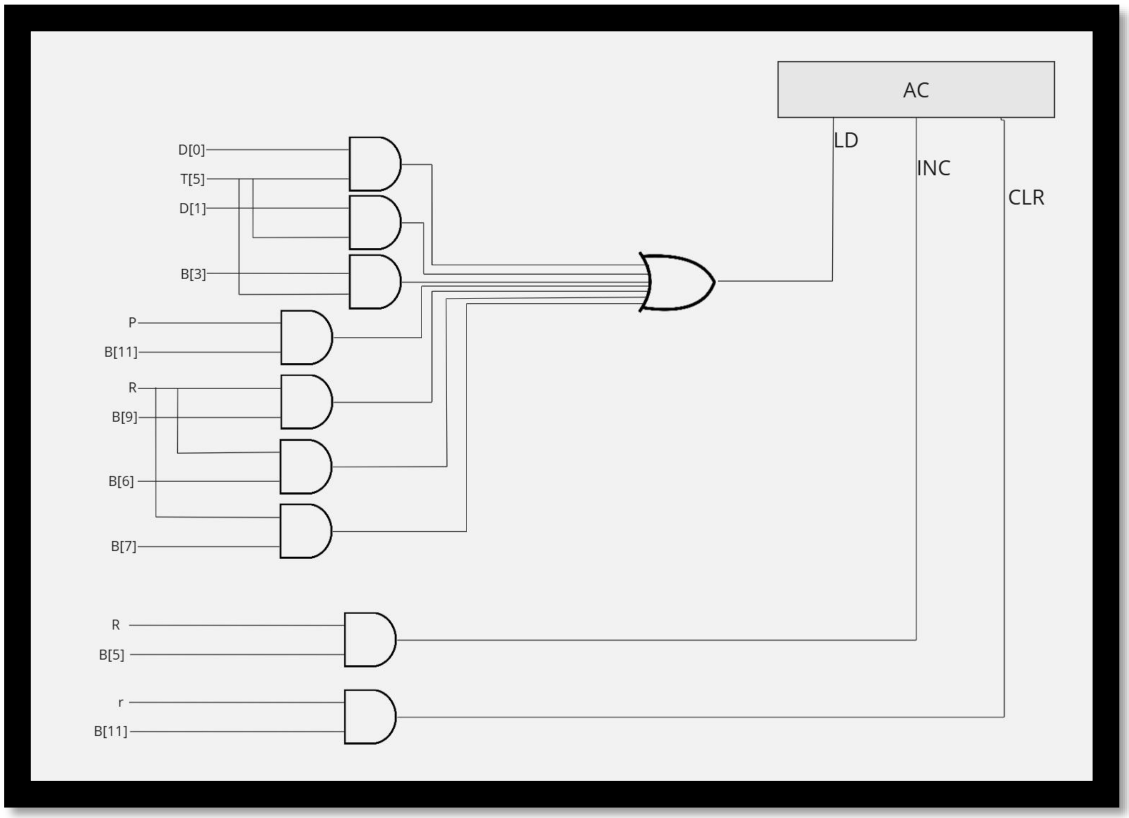


Figure 1.15 Internal design of AC register

2.4.1 Internal design of Flip Flop

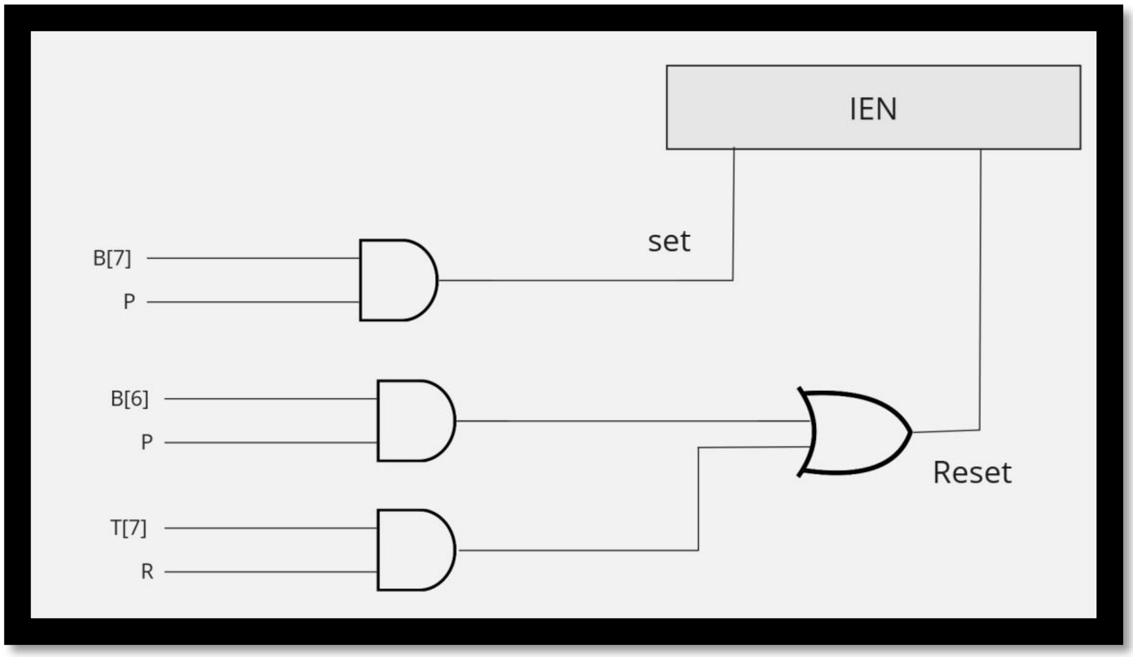


Figure 1.16 Internal design of IEN Flip Flop

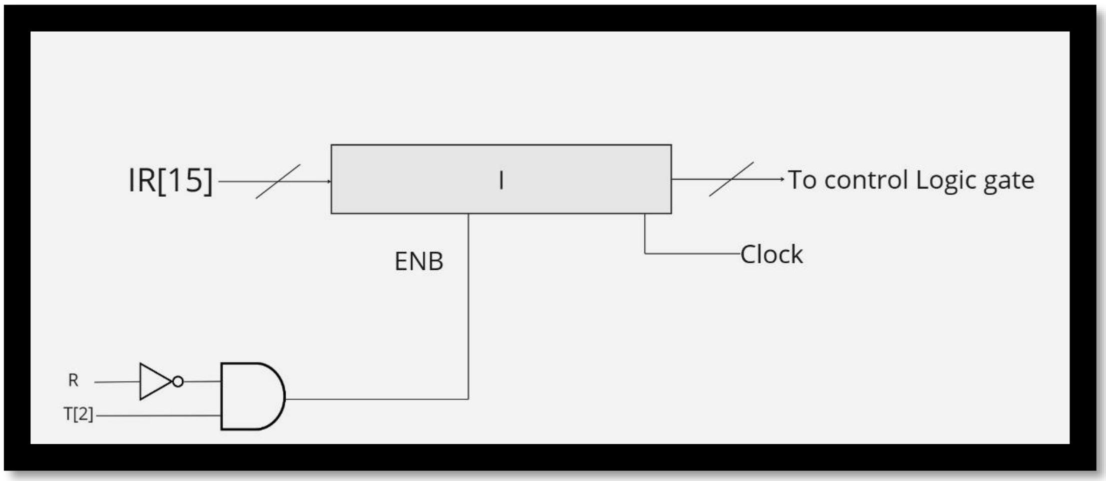


Figure 1.17 Internal design of I Flip Flop

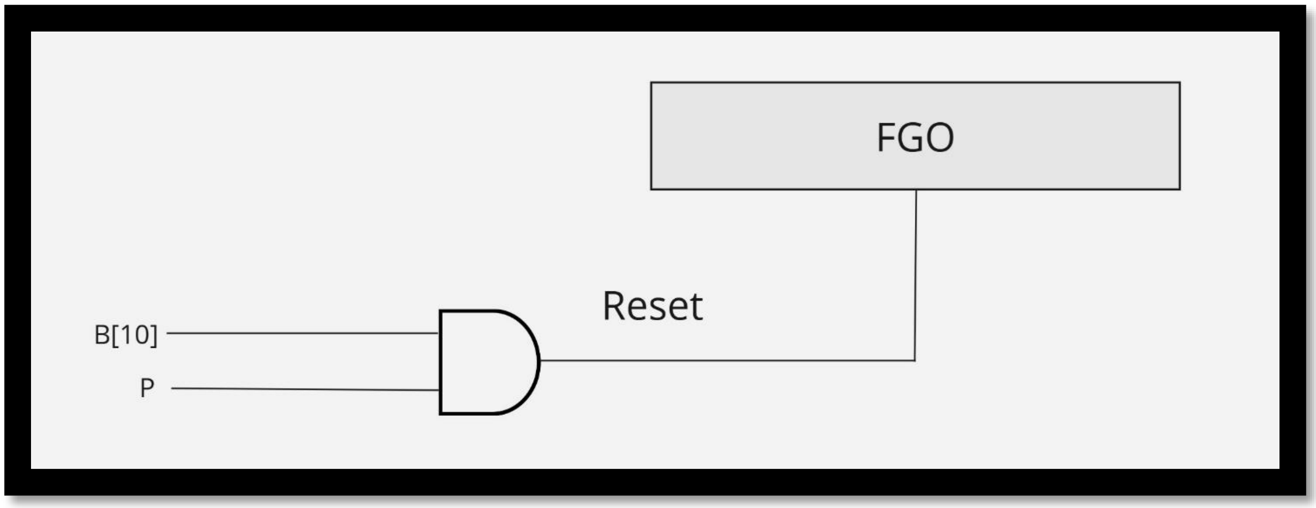


Figure 1.18 Internal design of FGO Flip Flop

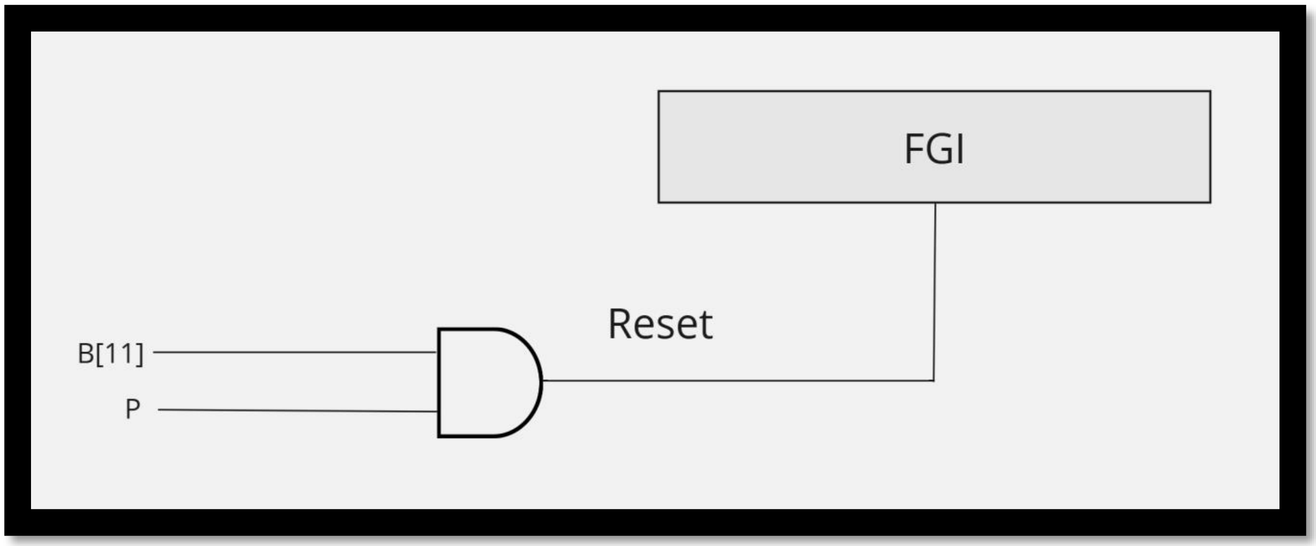


Figure 1.19 Internal design of FGI Flip Flop

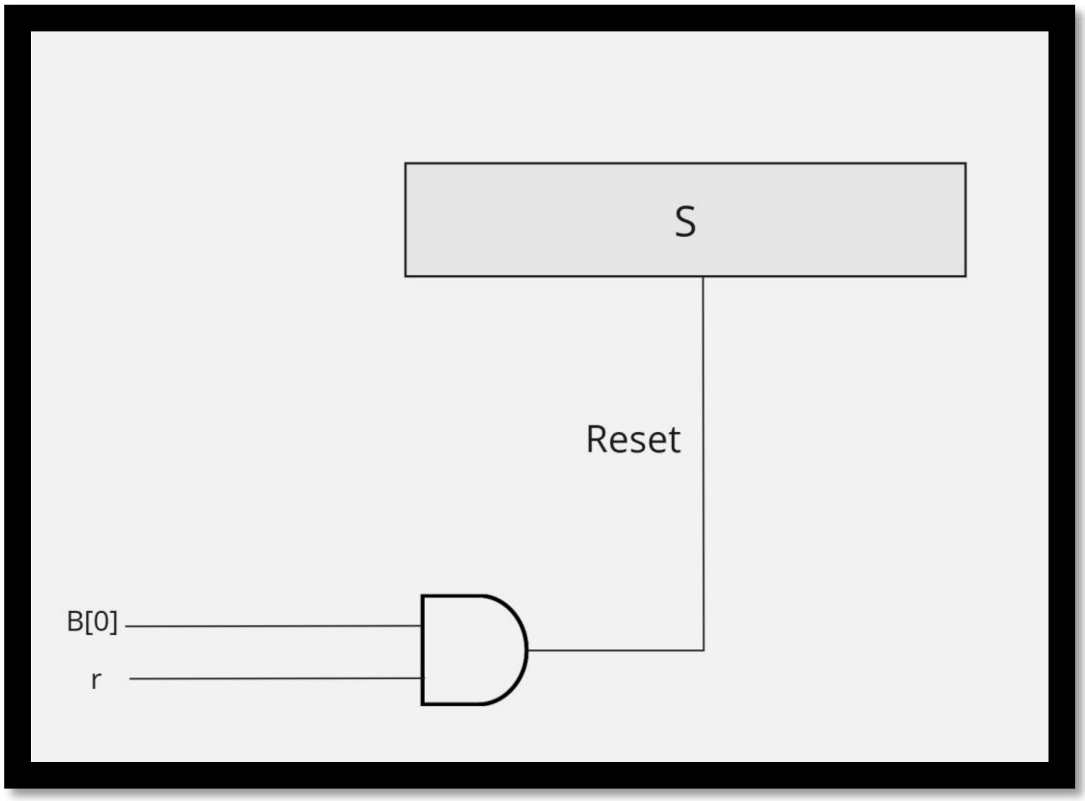


Figure 1.20 Internal design of Start Stop Flip Flop

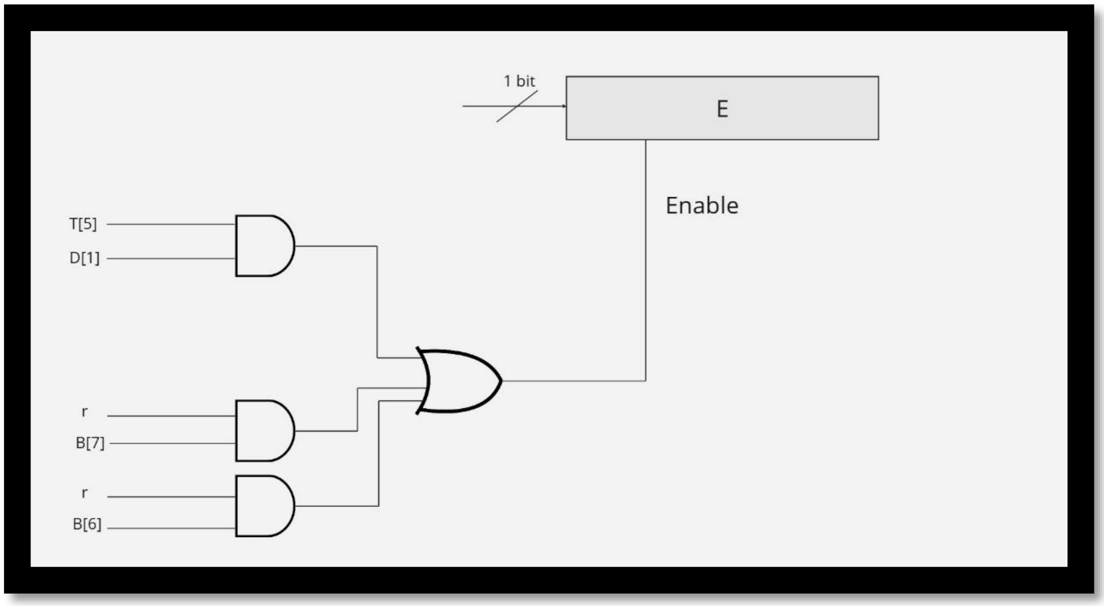


Figure 1.21 Internal design of E Flip Flop

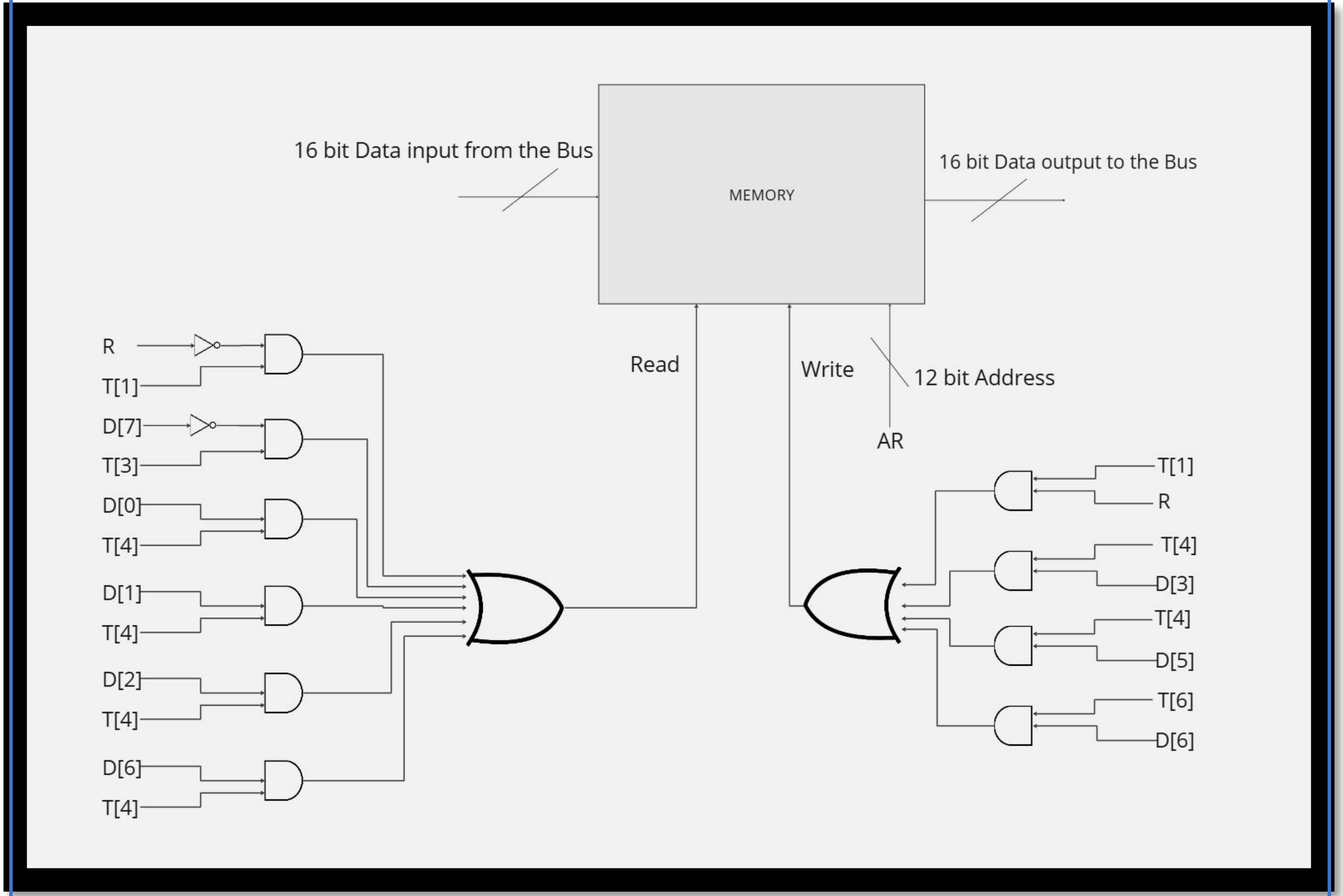


Figure 1.22 Internal design of Memory

Interrupt cycle

1. Input-Output Configuration

In this system, input and output operations are managed through a well-defined process. Input data consists of 8-bit alphanumeric codes entered by the programmer. This data is stored in the input register (INPR). Similarly, output data is stored in the output register (OUTR) and displayed as the result. The transmitter interface transfers the programmer's input data to INPR, while the receiver interface retrieves data from OUTR for display

1.1 Input Register

The input register (INPR) is an 8-bit register that holds the programmer's input. A 1-bit input flag (FGI) acts as a control flip-flop to synchronize the timing differences between the input source and the system. Initially, FGI is cleared to 0. When the programmer enters an 8-bit alphanumeric value, it is shifted into INPR, and FGI is set to 1.
The system continuously monitors FGI. If the flag is 1, it indicates that new data is available. The information in INPR is then transferred to the accumulator (AC), and FGI is cleared to 0. Once cleared, INPR is ready to accept new data

1.2 Output Register

The output register (OUTR) operates similarly but in reverse. Initially, the output flag (FGO) is set to 1, signaling that the system can load new data into OUTR. When the system transfers data from AC to OUTR, FGO is cleared to 0, indicating that the output operation is in progress. The output value is then processed and displayed as the result. Once the operation is complete, FGO is set to 1, allowing the system to load new data into OUTR. The system refrains from updating OUTR while FGO is 0, ensuring that the ongoing operation is not interrupted

---

2. Programmed Control Transfer and Interrupt Handling

The process of communication described is referred to as **programmed control transfer**. In this approach, the system continuously checks a flag bit, and when it finds the flag set, it initiates an information transfer

An alternative to programmed control is the use of **interrupt-driven transfer**. Here, the system allows external signals to notify it when data is ready for transfer, eliminating the need for constant flag checks. While running a program, the system does not actively monitor flags. Instead, when a flag is set, the system is momentarily interrupted to handle the transfer. After completing the transfer, the system resumes the original program from where it was interrupted

2.1 Interrupt Enable Mechanism

The interrupt enable flip-flop (IEN) can be controlled by two instructions. Clearing IEN to 0 **(using the IOF instruction)** disables interrupts, while setting IEN to 1 **(using the ION instruction)** enables them. This gives the programmer control over whether or not to use the interrupt mechanism

2.2 Interrupt Handling Process

The process of handling an interrupt is outlined as follows :

- During the execution of an instruction cycle, the system checks the IEN status
    - If IEN is 0 the system skips interrupt handling and continues with the next instruction cycle
    - If IEN is 1the system checks the flag bits
- If no flags are set, the system proceeds with the next instruction cycle.
- If a flag is set and IEN is 1, the interrupt flip-flop (R) is set to 1. At the end of the current instruction execution, the system enters an **interrupt cycle** instead of the next **Instruction cycle**

2.3 Interrupt Cycle

The interrupt cycle performs the following operations :

- The return address (stored in the program counter, PC) is saved in a predefined location (e.g., memory location 0)
- The PC is updated to point to the interrupt service routine (ISR) start address (e.g. address 1)
- Both IEN and R are cleared to prevent additional interrupts during the current ISR

The interrupt cycle begins after the last execute phase if the interrupt flip-flop (R) is set to 1. This occurs when IEN is 1 and either FGI or FGO is 1, but the timing signals T0, T1, and T2 should be not active. The condition for setting R to 1 can be expressed as:

$$\boxed{T_0' \, T_1' \, T_2' \, (IEN)\,(FGI + FGO) : R \leftarrow 1}$$

Here, the symbol + between FGI and FGO denotes a logical OR operation, and this is ANDed with IEN and the negated timing signals $T_0' \, T_1' \, T_2'$

2.4 Modified Fetch Phase

The fetch and decode phases of the **instruction cycle** are modified to account for the interrupt cycle Timing signals T0, T1, and T2 are now ANDed with R' (the complement of R) This ensures that the fetch and decode phases only proceed if R = 0 . If R = 1, the system enters the interrupt cycle instead

During the interrupt cycle:

1. The return address (stored in PC) is saved in memory location 0
2. The program branches to memory location 1
3. The IEN, R, and SC registers are cleared to 0

This is achieved through the following sequence of microoperations:

RT_0: $AR \leftarrow 0$, $TR \leftarrow PC$
RT_1: $M[AR] \leftarrow TR$, $PC \leftarrow 0$
RT_2: $PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$

- During RT_0 the address register (AR) is cleared to 0 and the content of PC is transferred to the temporary register (TR)
- During RT_1 the return address from TR is stored in memory at location 0, and PC is cleared to 0
- During RT_2 PC is incremented to 1 IEN and R are cleared to 0, and the sequence counter (SC) is cleared to 0

At the end of RT_2 the next instruction cycle begins with the condition $R'T_0$, The PC now holds the value 1 and the system fetches and executes the instruction located at address 1 This instruction typically branches to the service routine

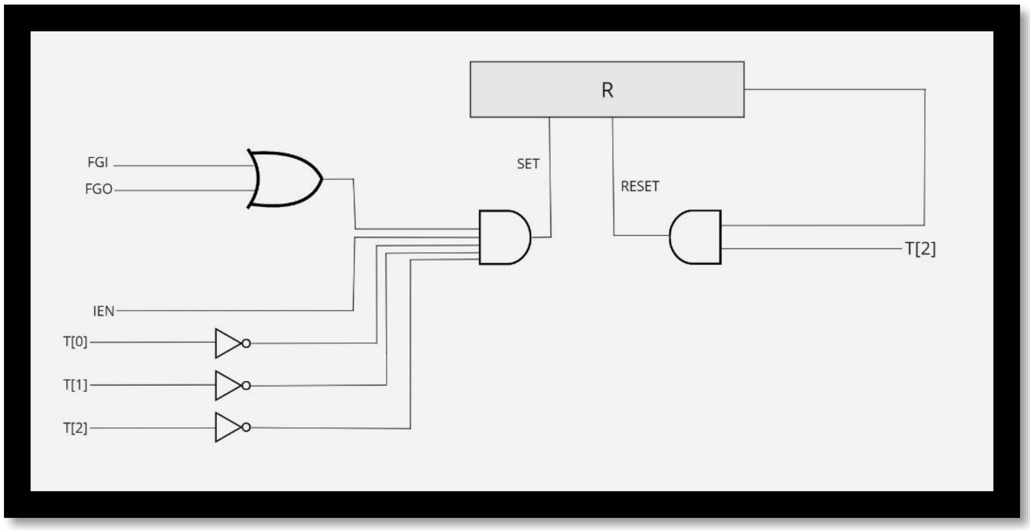


Figure 2.1 Internal design of R Flip Flop

Complete Computer Description

The final flowchart of the instruction cycle, including the interrupt cycle for the basic computer, is shown in **Fig 2.2** The interrupt flip-flop R may be set at any time during the indirect or execute phases. Control returns to timing signal T_0 after SC is cleared to 0 If R = 1, the computer goes through an interrupt cycle If R = 0 the computer goes through an instruction cycle. If the instruction is one of the memory-reference instructions, the computer first checks if there is an indirect address and then continues to execute the decoded instruction according to the **Fig 2.3** If the instruction is one of the register-reference instructions, it is executed with one of the microoperations listed in **Table 2.1** If it is an input-output instruction, it is executed with one of the microoperations listed in **Table 2.1**

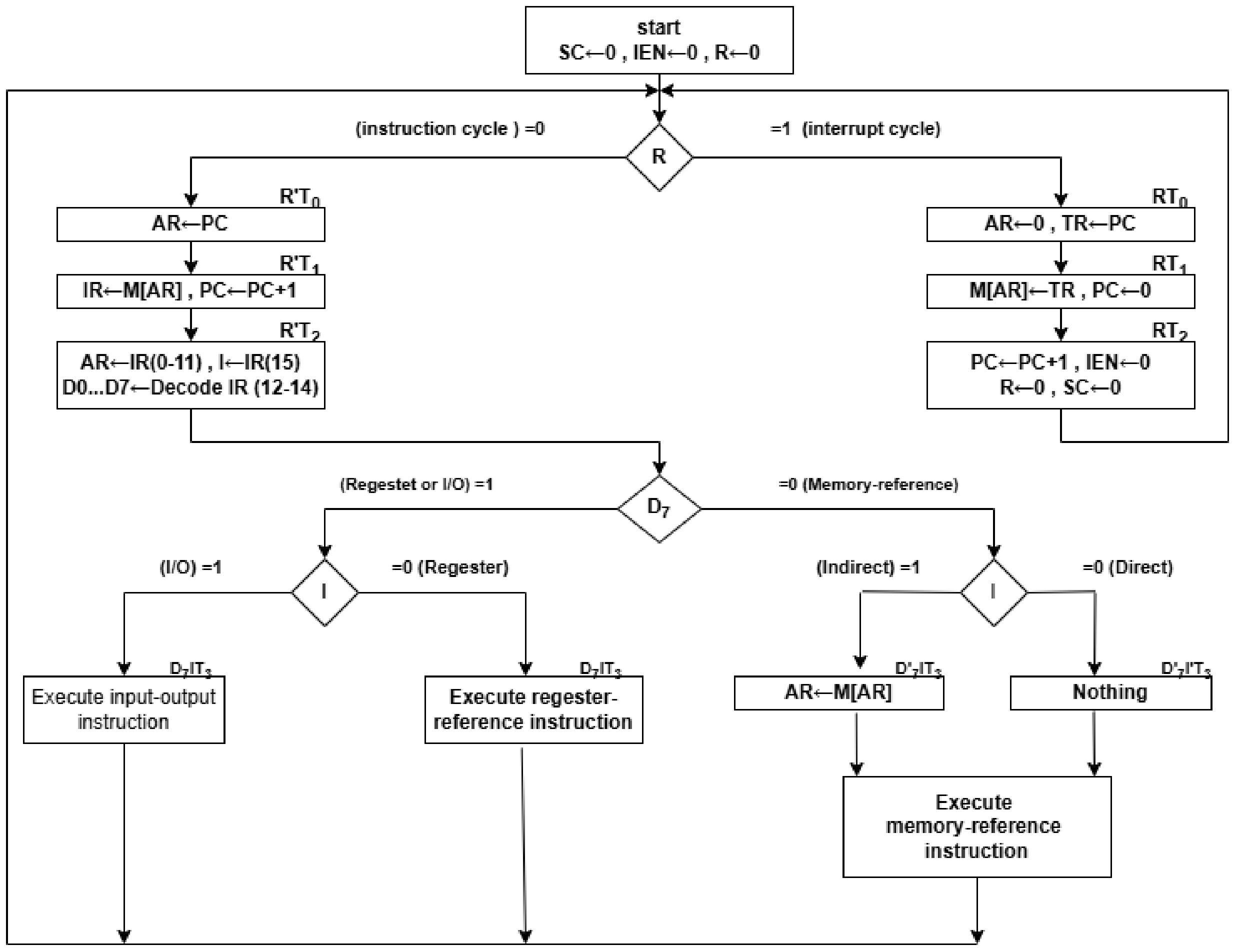


Figure 2.2 flowchart for computer operation.

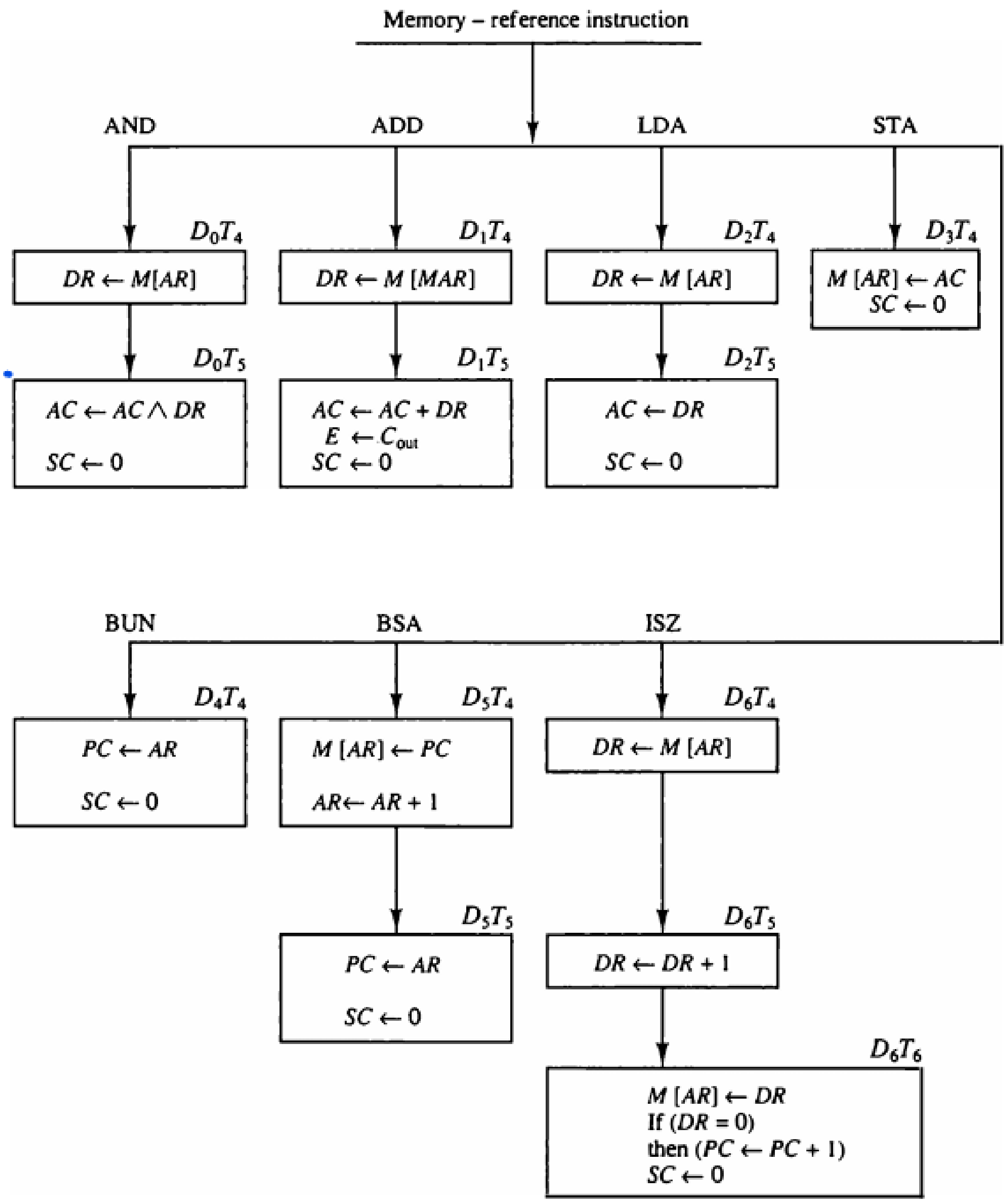


Figure 2.3 flowchart for memory reference instruction

Fetch	$R'T_1$	:	AR ←PC
	$R'T_1$	:	IR ←M[AR] , PC ←PC+1
Decode	$R'T_2$	:	D0…D7 ←Decode IR(12-14), AR←IR(0-11) , I ←I R(15)
Indirect	D'_7IT_2	:	AR← M[AR]
Interrupt			
$T'_0T'_1T'_2$(IEN)(FGI+FGO)		:	R←1
	RT_0	:	AR←0 , TR←PC
	RT_1	:	M[AR] ←TR , PC← 0
	RT_2	:	PC← PC+1 , IEN← 0 , R←0 , SC←0
Memory reference			
AND	D_0T_4	:	DR←M[AR]
	D_0T_5	:	AC←AC∧ DR , SC← 0
ADD	D_1T_4	:	DR← M[AR]
	D_1T_5	:	AC← AC+DR , E← C_{OUT} , SC←0
LDA	D_2T_4	:	DR← M[AR]
	D_2T_5	:	AC← DR , SC← 0
STA	D_3T_4	:	M[AR] ← AC , SC← 0
BUN	D_4T_4	:	PC← AR , SC← 0
BSA	D_5T_4	:	M[AR] ←PC , AR← AR+1
	D_5T_5	:	PC←AR , SC←0
ISZ	D_6T_4	:	DR← M[AR]
	D_6T_5	:	DR← DR+1
	D_6T_6	:	M[AR] ← DR , if (DR=0) then (PC←PC+1), SC←0
Register-reference	$D_7I'T_3$	:	r(common to all register-reference instructions)
	IR(i)	=	B_i (i=0,1,2,…,11)
	r	:	SC← 0
CLA	rB_{11}	:	AC← 0
CLE	rB_{10}	:	E←0
CMA	rB_9	:	AC←$\overline{AC}$
CME	rB_8	:	E ← $\overline{E}$
CIR	rB_7	:	AC← shr AC , AC(15)← E , E← AC(0)
CIL	rB_6	:	AC← shl AC , AC(0) ← E , E← AC(15)
INC	rB_5	:	AC← AC+1
SPA	rB_4	:	If (AC(15)=0) then (PC← PC+1)
SNA	rB_3	:	If (AC(15)=1) then (PC← PC+1)
SZA	rB_2	:	If (AC=0) then (PC ← PC+1)
HZE	rB_1	:	If (E=0) then (PC← PC+1)
HLT	rB_0	:	S← 0
Input_output	D_7IT_3	=	p (common to all input-output instructions)
	IR(i)	=	Bi (i= 6,7,8,9,10,11)
	p	:	SC← 0
INP	pB_{11}	:	AC(0-7) ← INPR , FGI ←0
OUT	pB_{10}	:	OUTR← AC(0-7) , FGO←0
SKI	pB_9	:	If (FGI=1) then (PC ← PC+1)
SKO	pB_8	:	If (FGO=1) then (PC← PC+1)
ION	pB_7	:	IEN← 1
IOF	pB_6	:	IEN← 0

Table 2.1 Control Functions and Microoperations for the Basic Computer

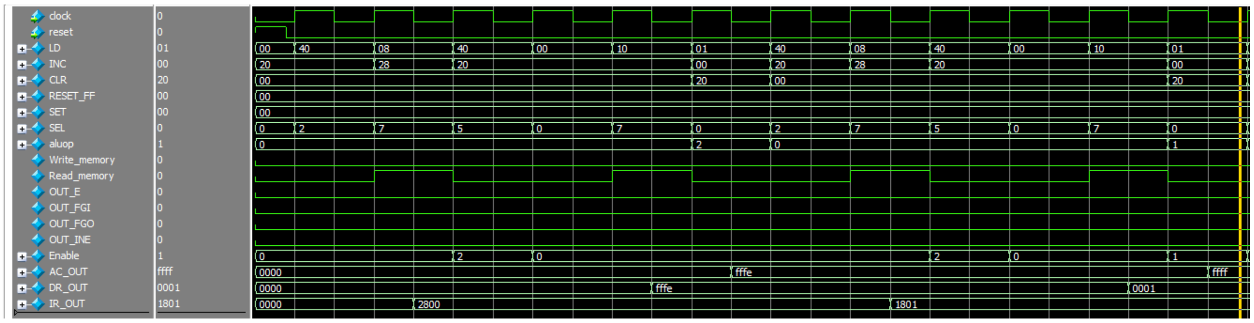
TESTING

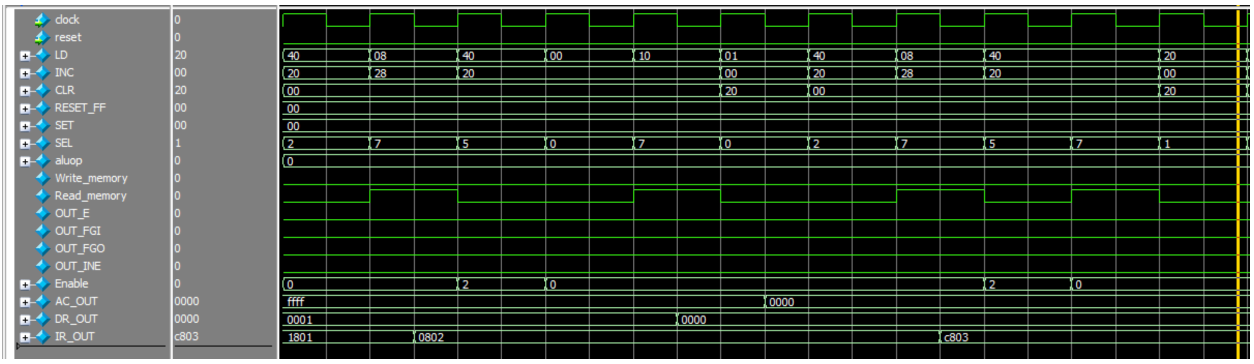
Test 1:

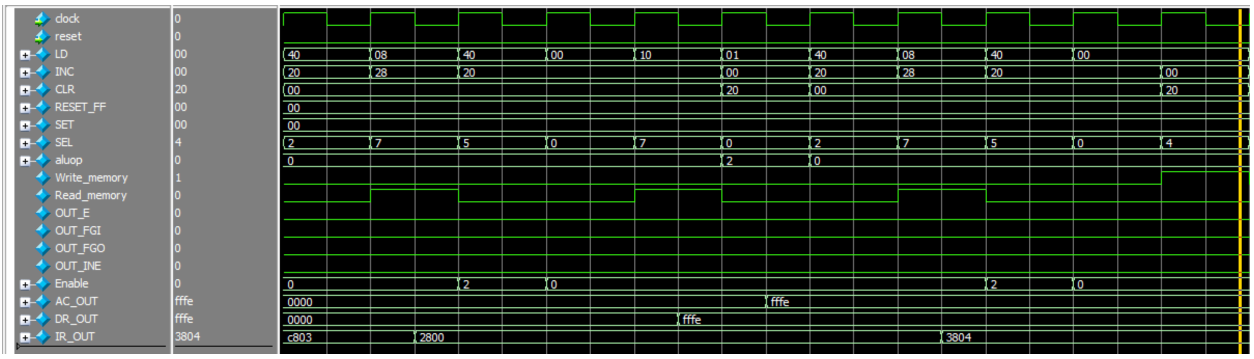
Instruction	Machine Code	operation
LDA [800]	2800	AC🡠 [800]
ADD [801]	1801	AC 🡠 AC + [801]
AND [802]	0802	AC 🡠 AC & [802]
BUN [803]	C004	PC🡠 [[803]]
LDA [800]	2800	AC🡠 [800]
STA [804]	3804	M[804]🡠 AC
BSA [805]	5805	PC 🡠 [805],M[805]🡠PC
ISZ [803]	6803	PC 🡠 PC+1 if [803]==0
CLA	7800	AC 🡠 0
CME	7100	E 🡠 ~E
CLE	7400	E 🡠 0
CMA	7200	AC 🡠 ~AC
CIR	7080	CIRCULATE AC AND E TO THE RIGHT
CIL	7040	CIRCULATE AC AND E TO THE LEFT
INC	7020	AC 🡠 AC +1
SPA	7010	Skip next instr. if AC is positive
NOP	xxxx	It will not be execute
SNA	7008	Skip next instr. if AC is negative
SZA	7004	Skip next instr. if AC is zero
NOP	xxxx	It will not be execute
SZE	7002	Skip next instr. if E is zero
NOP	xxxx	It will not be execute
INP	F800	AC 🡠 INPR
OUT	F400	OUTR 🡠 AC
SKI	F200	Skip on input flag
SKO	F100	Skip on output flag
ION	F080	IEN 🡠 1
IOF	F040	IEN 🡠 0

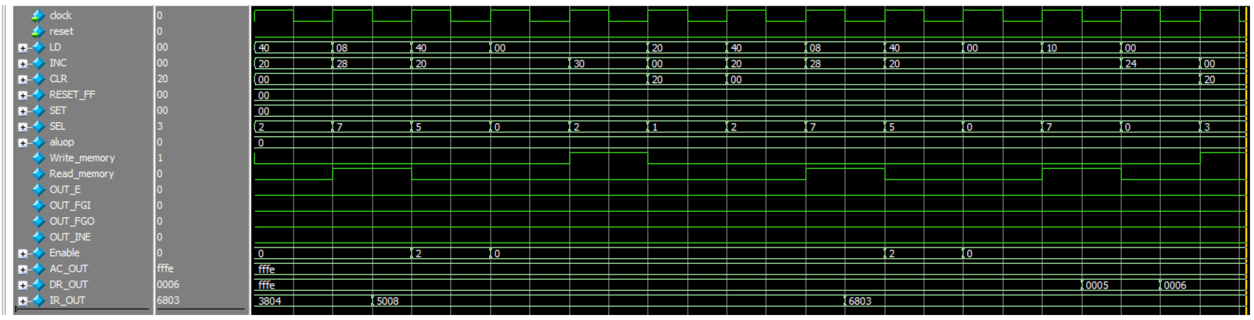
Data section:

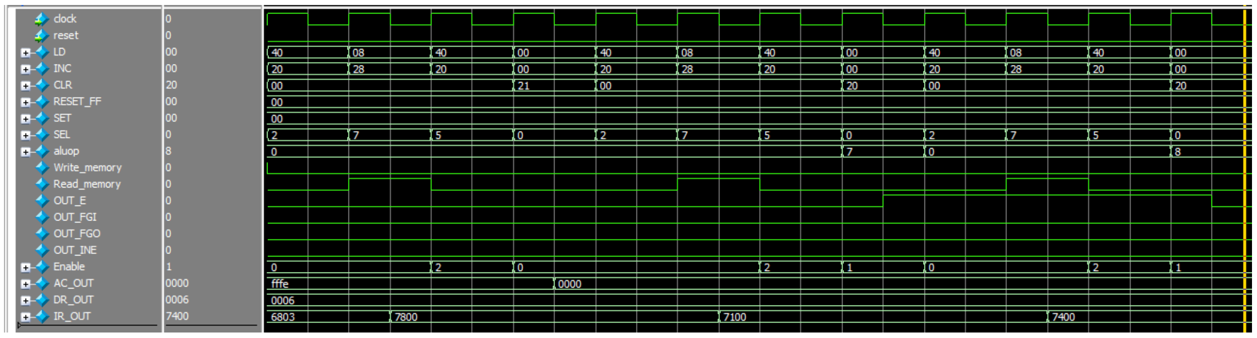
Location	data
800h	FFFE
801h	0001
802h	0000
803h	0004
804h	0000
805h	0008

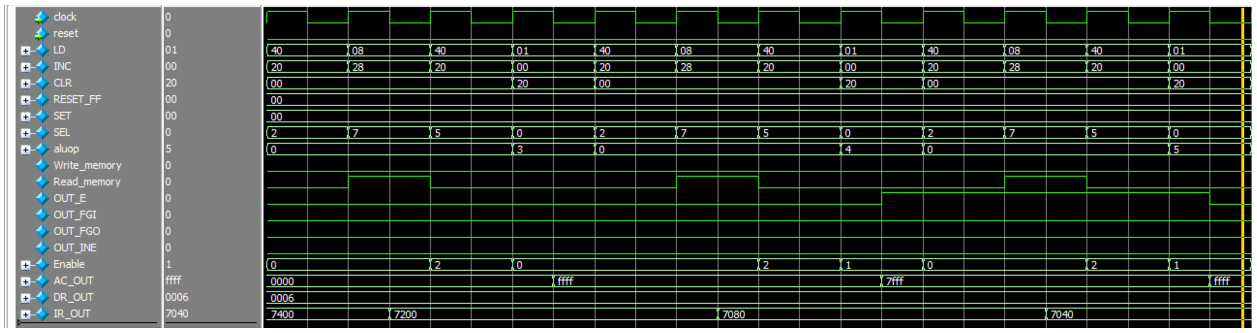

clock 0
reset 0
LD 01
INC 00
CLR 20
RESET_FF 00
SET 00
SEL 0
aluop 1
Write_memory 0
Read_memory 0
OUT_E 0
OUT_FGI 0
OUT_FGO 0
OUT_INE 0
Enable 1
AC_OUT ffff
DR_OUT 0001
IR_OUT 1801

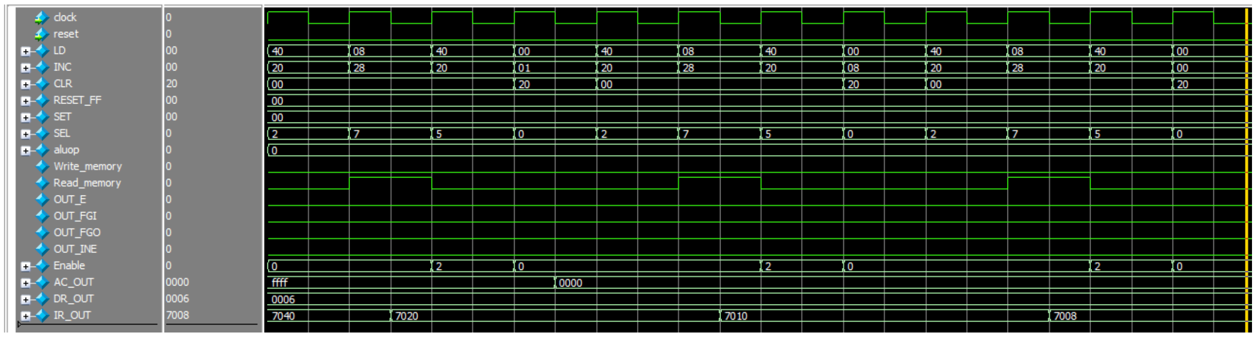

clock 0
reset 0
LD 20
INC 00
CLR 20
RESET_FF 00
SET 00
SEL 1
aluop 0
Write_memory 0
Read_memory 0
OUT_E 0
OUT_FGI 0
OUT_FGO 0
OUT_INE 0
Enable 0
AC_OUT 0000
DR_OUT 0000
IR_OUT c803

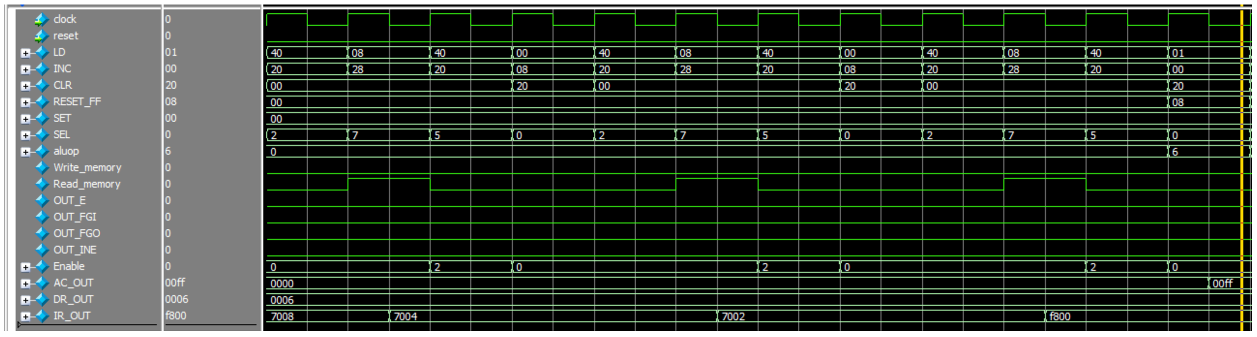

clock 0
reset 0
LD 00
INC 00
CLR 20
RESET_FF 00
SET 00
SEL 4
aluop 0
Write_memory 1
Read_memory 0
OUT_E 0
OUT_FGI 0
OUT_FGO 0
OUT_INE 0
Enable 0
AC_OUT fffe
DR_OUT fffe
IR_OUT 3804

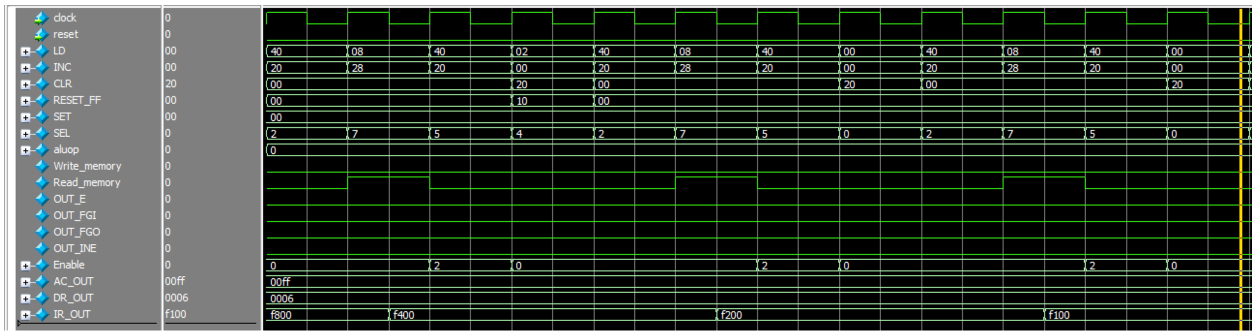

clock 0
reset 0
LD 00
INC 00
CLR 20
RESET_FF 00
SET 00
SEL 3
aluop 0
Write_memory 1
Read_memory 0
OUT_E 0
OUT_FGI 0
OUT_FGO 0
OUT_INE 0
Enable 0
AC_OUT fffe
DR_OUT 0006
IR_OUT 6803


clock 0
reset 0
LD 00
INC 00
CLR 20
RESET_FF 00
SET 00
SEL 0
aluop 8
Write_memory 0
Read_memory 0
OUT_E 0
OUT_FGI 0
OUT_FGO 0
OUT_INE 0
Enable 1
AC_OUT 0000
DR_OUT 0006
IR_OUT 7400


clock 0
reset 0
LD 01
INC 00
CLR 20
RESET_FF 00
SET 00
SEL 0
aluop 5
Write_memory 0
Read_memory 0
OUT_E 0
OUT_FGI 0
OUT_FGO 0
OUT_INE 0
Enable 1
AC_OUT ffff
DR_OUT 0006
IR_OUT 7040

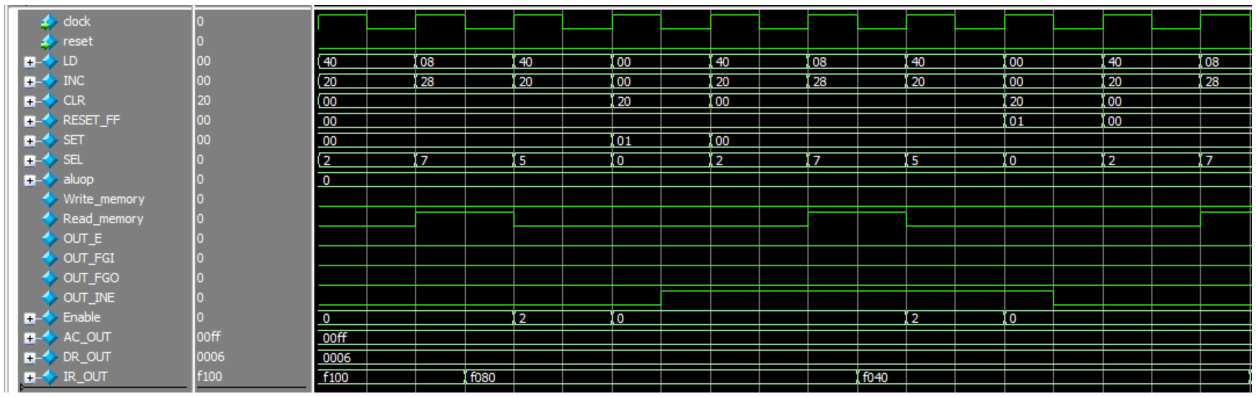

clock 0
reset 0
LD 00
INC 00
CLR 20
RESET_FF 00
SET 00
SEL 0
aluop 0
Write_memory 0
Read_memory 0
OUT_E 0
OUT_FGI 0
OUT_FGO 0
OUT_INE 0
Enable 0
AC_OUT 0000
DR_OUT 0006
IR_OUT 7008


clock 0
reset 0
LD 01
INC 00
CLR 20
RESET_FF 08
SET 00
SEL 0
aluop 6
Write_memory 0
Read_memory 0
OUT_E 0
OUT_FGI 0
OUT_FGO 0
OUT_INE 0
Enable 0
AC_OUT 00ff
DR_OUT 0006
IR_OUT f800



```
clock         0
reset         0
LD            00
INC           00
CLR           20
RESET_FF      00
SET           00
SEL           0
aluop         0
Write_memory  0
Read_memory   0
OUT_E         0
OUT_FGI       0
OUT_FGO       0
OUT_INE       0
Enable        0
AC_OUT        00ff
DR_OUT        0006
IR_OUT        f100
```

LD: 40 08 40 02 40 08 40 00 40 08 40 00
INC: 20 28 20 00 20 28 20 00 20 28 20 00
CLR: 00 20 00 20 00 20
RESET_FF: 00 10 00
SET: 00
SEL: 2 7 5 4 2 7 5 0 2 7 5 0
aluop: 0
Enable: 0 2 0 2 0 2 0
AC_OUT: 00ff
DR_OUT: 0006
IR_OUT: f800 f400 f200 f100



```
clock         0
reset         0
LD            00
INC           00
CLR           20
RESET_FF      00
SET           00
SEL           0
aluop         0
Write_memory  0
Read_memory   0
OUT_E         0
OUT_FGI       0
OUT_FGO       0
OUT_INE       0
Enable        0
AC_OUT        00ff
DR_OUT        0006
IR_OUT        f100
```

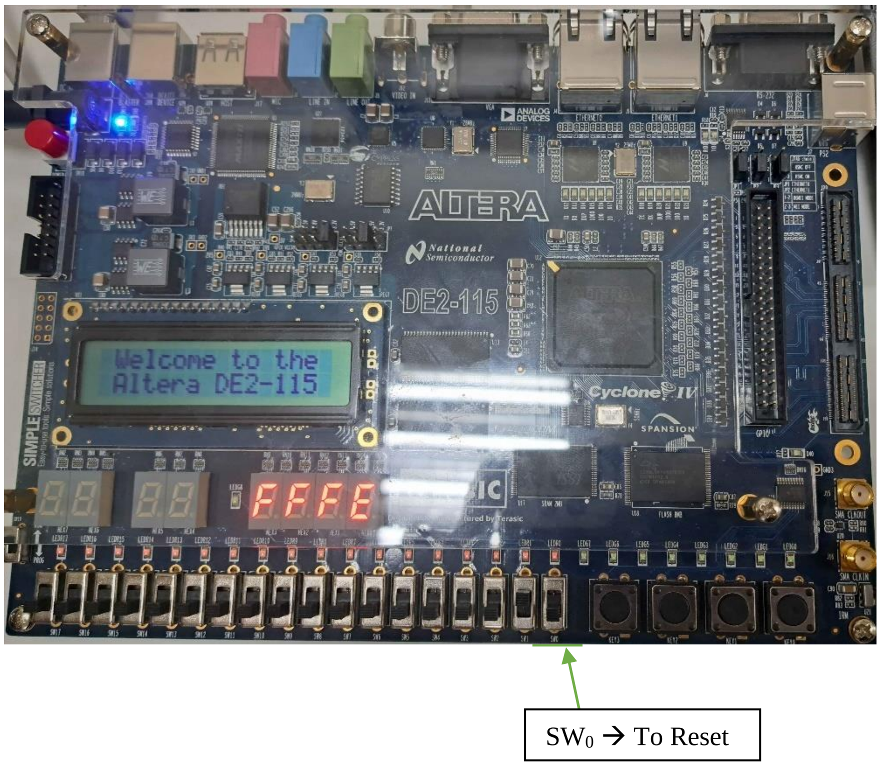
Test Design on FPGA

Decode is used to convert binary to hexadecimal, and decoding helps to display numbers on a 7-segment display.

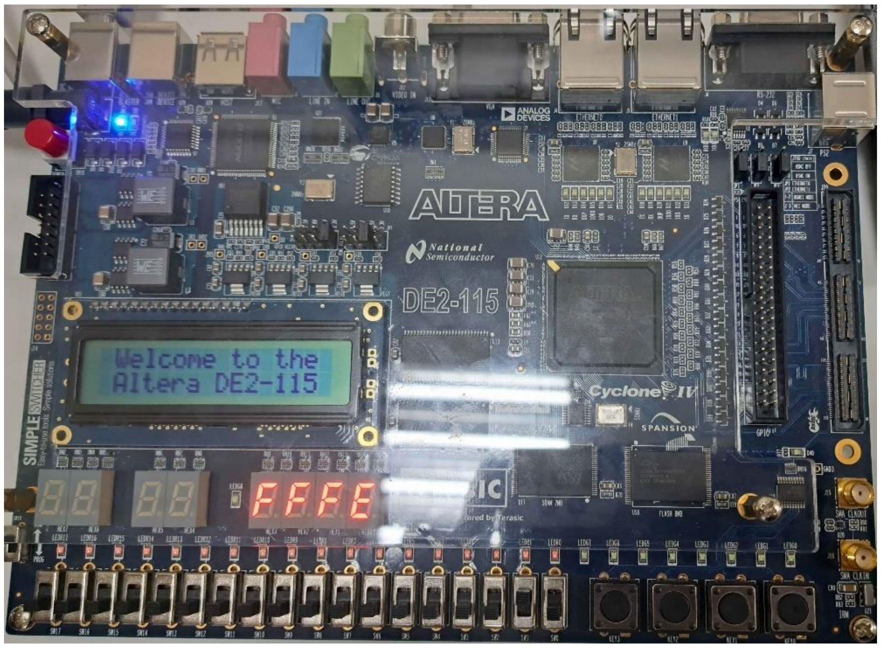
Decoder:

```
module decoder_4_16(in, out);
        input [3:0] in;
        output reg [6:0] out;
    always @ (*) begin
      case (in)
        4'h0: out = 7'b0000001;   // 0
        4'h1: out = 7'b1001111;   // 1
        4'h2: out = 7'b0010010;   // 2
        4'h3: out = 7'b0000110;   // 3
        4'h4: out = 7'b1001100;   // 4
        4'h5: out = 7'b0100100;   // 5
        4'h6: out = 7'b0100000;   // 6
        4'h7: out = 7'b0001111;   // 7
        4'h8: out = 7'b0000000;   // 8
        4'h9: out = 7'b0000100;   // 9
        4'hA: out = 7'b0001000;   // A
        4'hB: out = 7'b1100000;   // B
        4'hC: out = 7'b0110001;   // C
        4'hD: out = 7'b1000010;   // D
        4'hE: out = 7'b0110000;   // E
        4'hF: out = 7'b0111000;   // F
        default: out = 7'b1111111; // Off
     endcase
   end
end module
```

1. Reset:

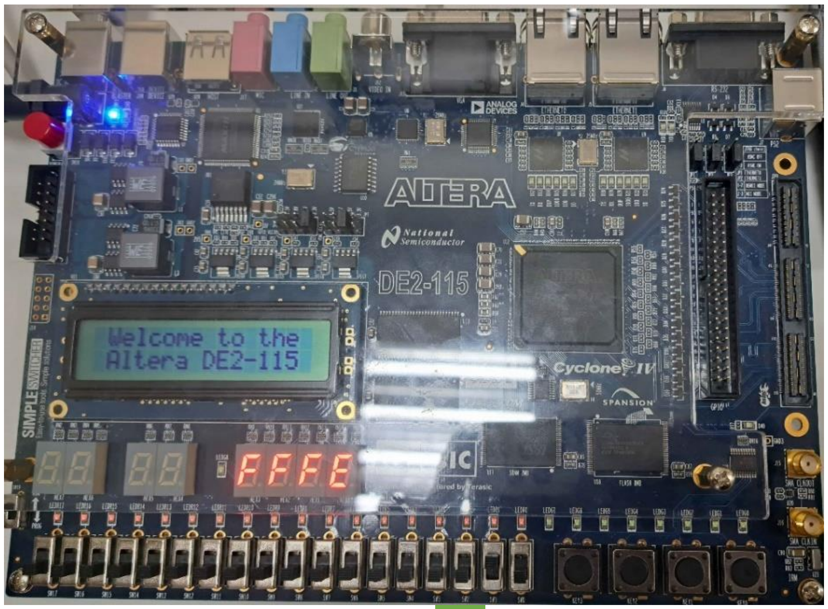


2. AC_OUT



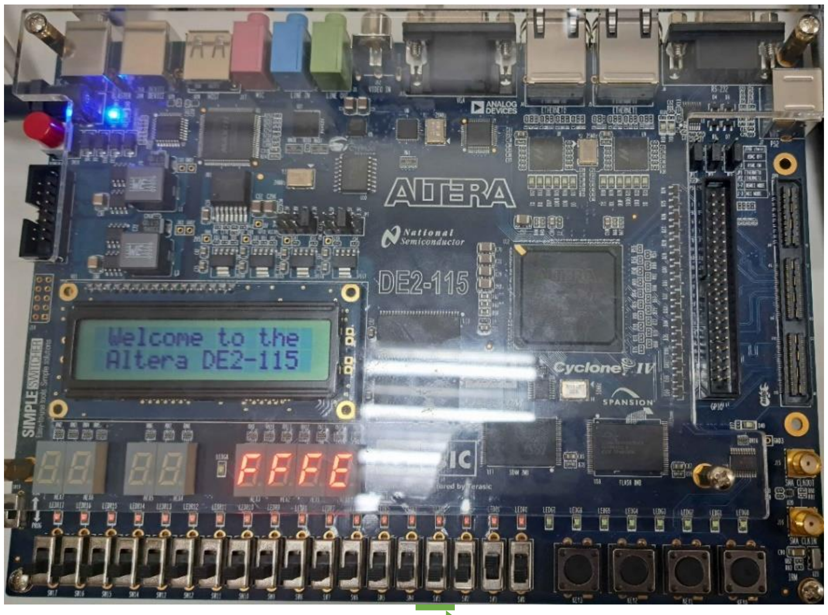
SW$_1$ → To Show AC

3. IR_OUT



SW$_2$ → To Show IR

4. DR_OUT



SW$_3$ → To Show DR