Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Mini Project

Report on

"Design of B^2RUM Computer"

[Course Code: COMP 315]

Submitted By:

Rajeshwor Niroula (71)

Usta Adhikari (68)

Bikramaditya Prasad Subedi (75)

Milan Dhamala (73)

Bigen Aryal (77)

Submitted To:

Mr. Pankaj Dawadi

Submission Date:

12th July, 2021

Abstract

We have designed 32*20 memory size computer for our CAO project with total of 31 instructions, 11 MRI, 14 RRI and 6 IOI. These instructions are similar to those we have studied under CAO basic computer design but we have also added few extra instructions of our own design. To implement logical and arithmetic operations we have employed 20 stages of 20-bit ALU connecting to 20-bit AC and a single E flipflop. This design also features other registers namely PC, AR, IR, DR, OUTR, INPR and TR alone with S, R, IEN, FGI and FGO as flipflops whose functions are discussed below. For 20-bit data interaction among register we have designed 20-bit CBS and several timing and control lines using 4*16 decoders.

Acknowledgement

We would like to show our gratitude to Mr. Pankaj Raj Dawadi, Assistant professor, Kathmandu University for providing us with an opportunity for implementing the course syllabus into a basic computer model. We would also like to thank him heartily for the guidelines that made this design possible.

Table of Contents

List of Figures

List of Tables

Acronyms

AR: Address Register

PC: Program Counter

DR: Data Register

TR: Temporary Register

AC: Accumulator

FGI: Input Flag

FGO: Output Flag

MRI: Memory Reference Instruction

CPU: Central Processing Unit

CLR: Clear

RRI: Register Reference Instruction

NOP: No Operation

XOR: Exclusive OR

LDA: Load Accumulator

STA: Store Accumulator

BUN: Branch Unconditionally

CLA: Clear Accumulator

INTR: Increment TR

ION: Interrupt on

IOF: Interrupt off

Chapter 1: Introduction

Our basic computer design has a complex instruction set, it supports basic arithmetic and logic operations such as addition, subtraction, and, or etc. Accumulator is used to store the output of any ALU operation. Data register is used to store data from memory before performing any operations on them. Address register and program counter are both used to specify memory address. Data is transferred between registers using a common address bus. Program counter stores the next instruction to be executed and the computer interfaces with the user using input and output registers. All the instructions are executed by using the signals generated by the control unit consisting of a sequence counter and instruction decoders.

Von Neumann architecture is the design upon which many general purpose computers are based. It consists of three essential different entities: a processing unit, an I/O unit and a storage unit. The units are connected over a common bus. In Von Neumann data and instruction are both stored as binary digits in primary storage. The instructions are fetched from memory one at a time sequentially as controlled by a program counter. The processor decodes and executes an instruction, before cycling around to fetch the next instruction. This cycle continues until no more instructions are available.

A processor based on von Neumann architecture has five special registers which it uses for processing:

- the program counter (PC) holds the memory address of the next instruction to be fetched from primary storage
- the memory address register (MAR) holds the address of the current instruction that is to be fetched from memory, or the address in memory to which data is to be transferred
- the memory data register (MDR) holds the contents found at the address held in the MAR, or data which is to be transferred to primary storage
- the current instruction register (CIR) holds the instruction that is currently being decoded and executed
- the accumulator (ACC) is a special purpose register and is used by the arithmetic logic unit (ALU) to hold the data being processed and the results of calculations

Our basic computer design is based upon the principal of Von Neumann Architecture.

Chapter 2: Design Consideration

For our computer, we have chosen a memory of 32k *20 size.

32k refers to the number of address or slots or words and 20 represents the size of each word.

$$32k = 2^5 * 2^{10} = 2^{15}$$

To represent all this addresses we need 15 bits of data, hence our IR has 15 bits for addresses.

Since, Memory is loaded into IR we need 20 bit IR for 20-word size with 15 bits occupied for address. It leaves us with 5 bits remaining in IR which we used for op-code and addressing mode. Since, our computer has only have 2 types of addressing, direct and indirect, we have used only 1 bit for mode and remaining 4 bits for op-code.

We used total of 8 registers of which AR and PC has 15 bits size because they only store address. INPR and OUTR are of 10 bits because that's our input character and output character size as for all other register they are of 20 bits size.

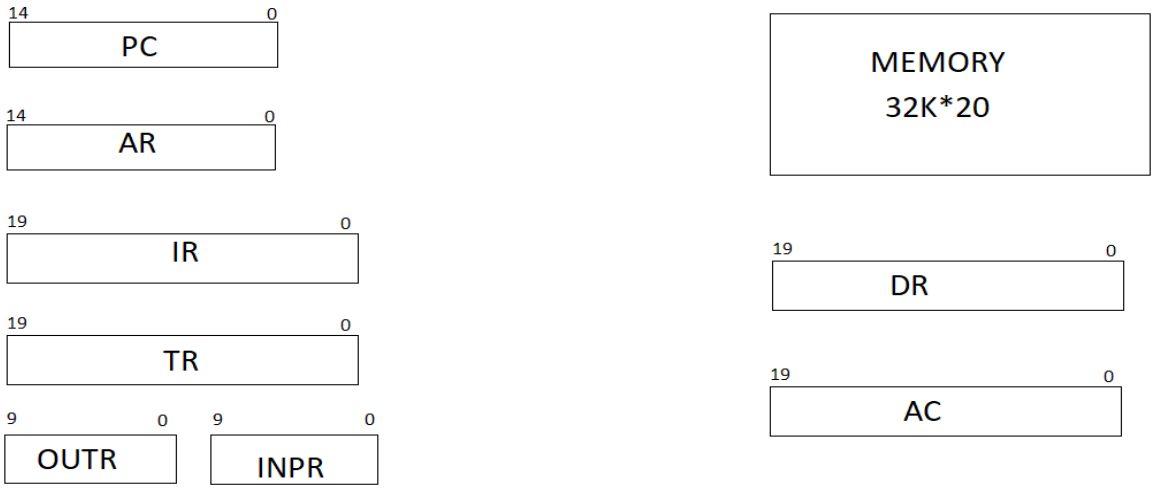

Figure 1: Structure of Basic Computer

- 2^5 =32 unique possible instructions
- 32K=2^5 * 2^{10}=2^{15}=32768 memory locations

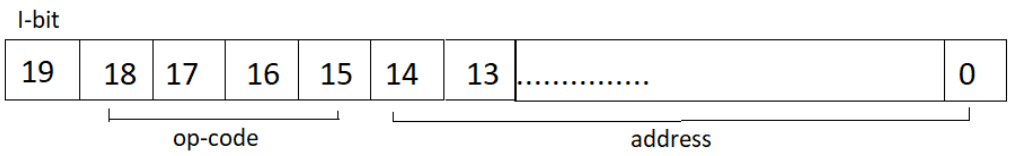

Figure 2: Design of IR

Registers	Bits
DR (Data Register)	20
AR (Address Register)	15
AC (Accumulator)	20
IR (Instruction Register)	20
PC (Program Counter)	15
TR (Temporary Register)	20
INPR (Input Character Register)	10
OUTR (Output Character Register)	10

Table 1: List of Register

Designing Instructions

We have three types of instruction as described below:

- **MRI (Memory Reference Instruction):** Like the name says these instructions deal with operations over memory stored data, for our computer we have designed 11 MRI instructions. These instructions can be identified by checking op-code bits. Anything but $(1111)_2$ for op-code is going to be an MRI instruction. These instructions are decoded from 18^{th}, 17^{th}, 16^{th} and 15^{th} bits of IR by a 4*16 decoder giving us (D_0-D_{15}) of which (D_0-D_{10}) represents MRI as tabulated below:

Decoder	Operation	I	0p-code				I	0p-code			
D_0	AND	0	0	0	0	0	1	0	0	0	0
D_1	ADD	0	0	0	0	1	1	0	0	0	1
D_2	LDA	0	0	0	1	0	1	0	0	1	0
D_3	STA	0	0	0	1	1	1	0	0	1	1
D_4	BUN	0	0	1	0	0	1	0	1	0	0
D_5	BSA	0	0	1	0	1	1	0	1	0	1
D_6	ISZ	0	0	1	1	0	1	0	1	1	0
D_7	OR	0	0	1	1	1	1	0	1	1	1
D_8	SUB	0	1	0	0	0	1	1	0	0	0
D_9	XOR	0	1	0	0	1	1	1	0	0	1
D_{10}	NAND	0	1	0	1	0	1	1	0	1	0

Table 2:MRI Code

- **RRI (Register Reference Instruction):** These instructions deal with operations over register stored data and doesn't involve memory. We have designed 14 RRI instructions. This instruction can be identified by checking op-code bits. In T_3 timing for I=0 if op-code= $(1111)_2$ its RRI since $(1111)_2$ activates decoder D_{15}, we can write above convention as:

$$I'.D_{15}.T_3=r$$

where r is common of all RRI instructions as for identifying individual instructions, we decode the address bits to give decoder output (B_0-B_{13}) which will represent 14 RRI as tabulated below:

INSTRUCTION	CODE	DECODER
CLA	78000	B_0
CLE	78001	B_1
CMA	78002	B_2
CME	78003	B_3
CIR	78004	B_4
CIL	78005	B_5
INC	78006	B_6
SPA	78007	B_7
SNA	78008	B_8
SZA	78009	B_9
SZE	78010	B_{10}
HLT	78011	B_{11}
TIRA	78012	B_{12}
TDRA	78013	B_{13}

Table 3: RRI Code

- **IOI(Input Output Instruction):** It deals with the input and output operations and can be identified in T_3 timing for I=1 if op-code = $(1111)_2$ its IOI. IOI uses same decoder as RRI and uses (B_0-B_5) to represent IOI instructions.

$$I.D_{15}.T_3=p$$

These instructions are tabulated below:

Instruction	Code	Decoder
INP	F8000	B_0
OUT	F8001	B_1
SKI	F8002	B_2
SKO	F8003	B_3
ION	F8004	B_4
IOF	F8005	B_5

Table 4: IOI Code

Instruction Cycle

Instruction cycle comprises of fetch, decode and execute cycle. The first two stage, fetch and decode are similar to all instructions, only execution will differ.

$R'T_0$: AR←PC

For timing T_0 address pointed by program counter is transferred to AR.

$R'T_1$: IR←M[AR], PC←PC+1

For timing T_1, memory of address in AR is stored in Instruction Register (IR) and program counter is incremented by 1 so that it can point to next instruction in the memory. This step is also known as fetch cycle.

$R'T_2$: I←IR(19), D_0-D_{15}←IR(18,17,16,15), AR←IR(14-0)

For timing T_2, 19th bit of IR is used as I bit, 18-15th bit selects a decoder output from (D_0-D_{15}).

As for remaining bits, they are stored in AR as address bits. This is also known as a decode cycle.

Once the decode is complete, process enters into timing T_3 where there are four possible outcomes.

1. $D_{15}'.I'.T_3$: NOP

For any decoder output except D_{15}, if the addressing mode is direct i.e I'(0), no operation will occur and computer transitions to T_4 cycle.

2. $D_{15}'.I.T_3$: AR←M[AR]

For any decoder output except D_{15}, if the addressing mode is indirect i.e I(1) memory stored in the address of AR is stored back into AR and the computer transitions to T_4 cycle.

3. $D_{15}.I'.T_3$: r

For D_{15} decoder output if the I bit equals to 0 it refers to RRI instructions.

4. $D_{15}.I.T_3$: p

For D_{15} decoder output if the I bit equals 1 it refers to IOI instruction.

After T_3 timing signal is complete, computer transitions to T_4 timing signal from where execution of instructions begins.

MRI

AND:

$D_0.T_4$: DR←M [AR]

$D_0.T_5$: AC← AC AND DR, SC←0

ADD to AC:

D_1T_4 : DR←M [AR]

D_1T_5 : AC←AC+DR, E←C_{out}, SC←0

LDA:

$D_2.T_4$: DR←M [AR]

$D_2.T_5$: AC←DR, SC←0

STA Store AC:

$D_3.T_4$: M [AR]←AC, SC←0

BUN:

$D_4.T_4$: PC←AR, SC←0

BSA:

$D_5.T_4$: M[AR]←PC, AR←AR+1

$D_5.T_5$: 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

OR:

D_7T_4 : DR←M [AR]

D_7T_5 : AC←AC v DR, SC←0

SUB:

D_8T_4 : DR←M [AR]

D_8T_5 : AC←AC - DR, E←C_{out}, SC←0

XOR:

D_9T_4 : DR←M [AR]

D_9T_5 : AC←AC⊕DR, SC←0

NAND:

$D_{10}T_4$: DR←M [AR]

$D_{10}T_5$: AC←AC NAND D R, SC←0

Indirect Addressing:

D_{15}'IT_3:AR←M [AR]

RRI:

$D_{10}I'.T_3$: r r:SC←0

CLA : Clear AC

rB_0 : AC←0

CLE : Clear E

rB_1 : E←0

CMA : Complement AC

rB_2 : AC←AC'

CME : Complement E

rB_3 : E←E'

CIR : Circulate Right

rB_4 : AC←shr (AC), AC(19)←E, E←AC(0)

CIL : Circulate Left

rB_5 : AC←shl (AC), AC(0)←E, E←AC(19)

INC : Increment AC

rB_6 : AC←AC+1

SPA : Skip if Positive

rB_7 : If (AC(19)=0), then PC←PC+1

SNA : Skip if Negative

rB_8 : If (AC(19)=1), then PC←PC+1

SZA : Skip if Zero AC

rB_9 : If AC=0 then, PC←PC+1

SZE : Skip if Zero E

rB_{10} : If (E=0), then PC←PC+1

HLT : Halt

rB_{11} : S←0 (S is start/stop flip flop)

TIRA : Transfer IR to ACC

rB_{12} : AC←IR

TDRA : Transfer DR complement to AC

rB_{13} : AC←DR'

IOI:

$D_{15}.I.T_3$: p

p:SC←0

INP: Input Character.

pB_0: AC(0-9) ← INPR, FGI ← 0

OUT: Output Character

pB_1: OUTR ← AC(0-9), FGO ← 0

SKI: Skip on Input flag.

pB_2: If (FGI=1) then PC ← PC +1

SKO: Skip on Output flag

pB_3: If (FGO=1) then PC ← PC+1

ION: Interrupt Enable On

pB_4: IEN ← 1

IOF: Interrupt Enable Off

pB_5: IEN ← 0

Interrupt Cycle

Interrupt is a disturbance caused by I/O for inputting data or outputting data. Since CPU is much faster than I/O devices there can be a speed mis-matching, to solve this we use flags to synchronize timing. The system keeps constantly checking for flag bits, R stored in a flip-flop. The IEN flip-flop can be set and cleared with two instructions. For as long as IEN is set to 0, the system cannot be interrupted. Once the IEN is set to 1 the system checks for either FGO or FGI flags. If any of the flags is set to 1 then the flag R is set to 1 and the interrupt cycle begins.

Interrupt can happen with any clock transition except when timing signals T_0, T_1 and T_2 are active.

IEN (FGI+FGO) . T_0' . T_1' . T_2' : 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

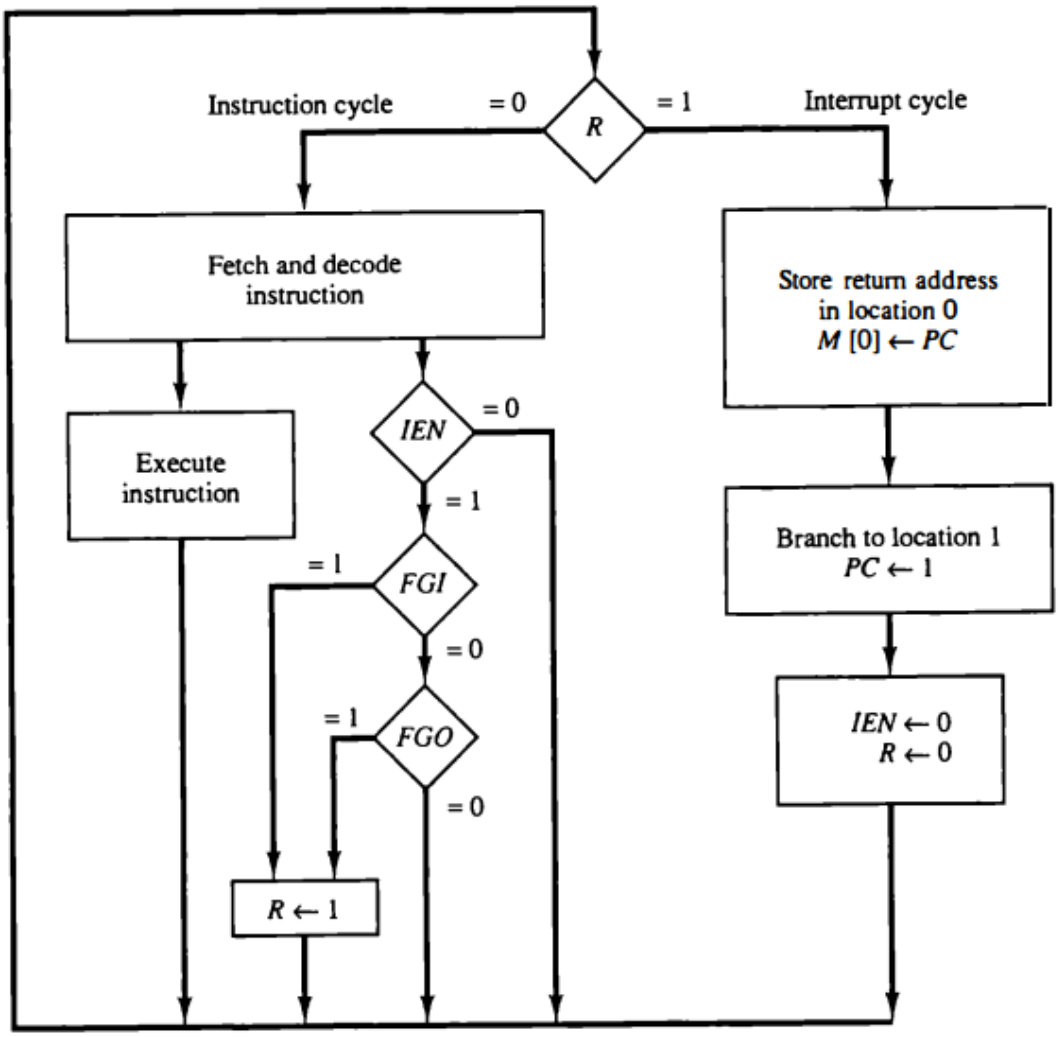

Figure 3: Interrupt Cycle Flowchart

Designing Control and Timing Unit:

Control unit comprises of IR, sequence counter and decoder 19th bit of IR is used as I bit, 18th -15th bit is decoded to D_0 - D_{15} and addresses bit will be decoded to B_0 - B_{13} all of which will be used for controlling registers and flip-flops throughout the computer.

As for timing signals a 4-bit sequence counter is connected to a decoder to give T_0 -T_{15} timing signal. SC is always incremented for every clock except when CLR is set to 1 through instruction cycles or if the timer reaches T_{15}.

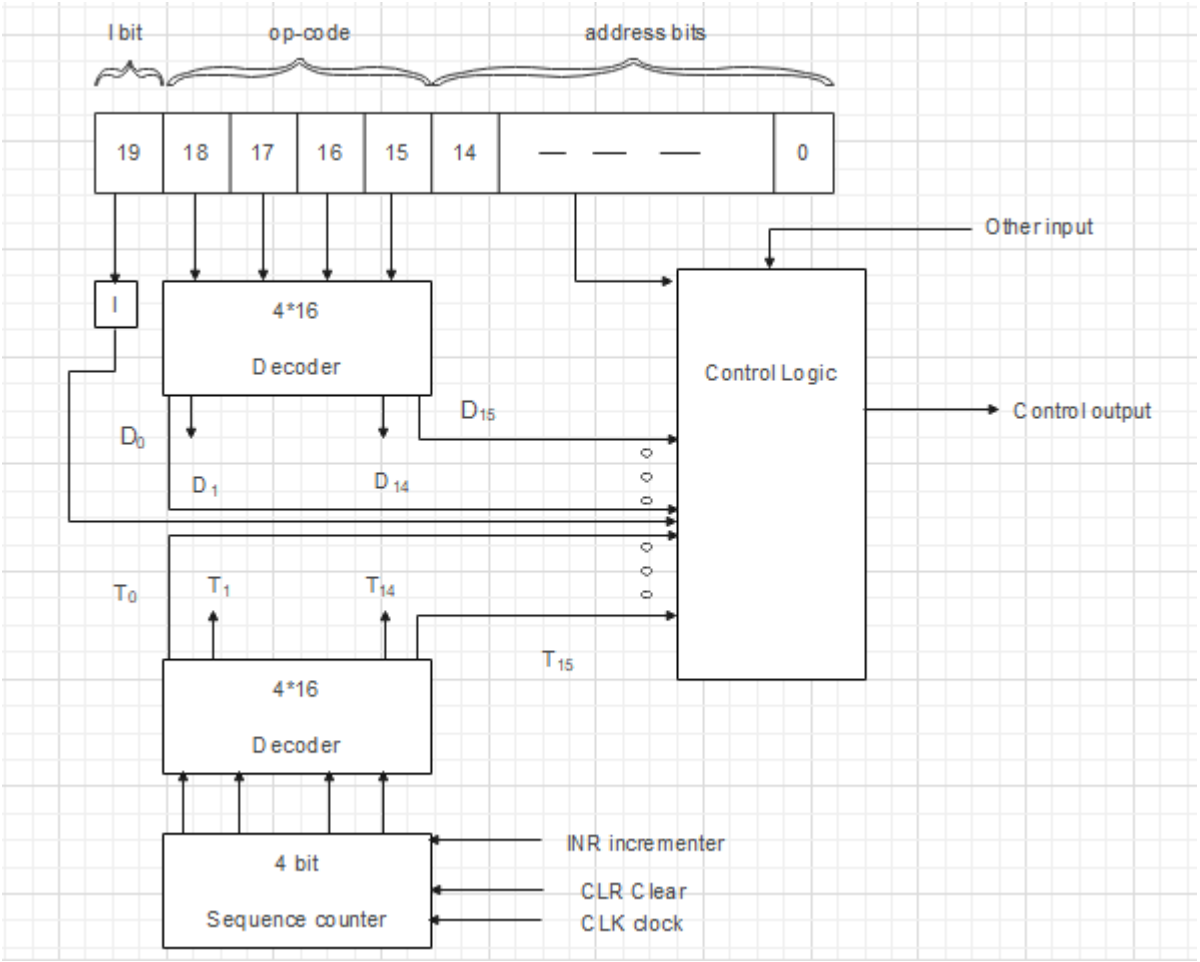

Figure 4: Timing and Control Unit

Chapter 3: Design of Individual Units

SC (Sequence Counter): Much about sequence counter has already been discussed in control and timing unit.

The logic for CLR (Clear) for SC can be obtained by OR-ing all of below conditions:

r:SC←0

$D_0.T_5$: AC← AC AND DR, SC←0

D_1T_5 : AC←AC+DR, E←C_{out}, SC←0

D_2T_5 : AC←DR, SC←0

D_3T_4 : M [AR]←AC, SC←0

D_4T_4 : PC←AR, SC←0

D_5T_5 : PC←AR, SC←0

D_6T_6 : M [AR]←DR, if (DR=0) then (PC←PC+1), SC←0

D_7T_5 : AC←AC v DR, SC←0

D_8T_5 : AC←AC - DR, E_S←C_{out}, SC←0

D_9T_5 : AC←AC⊕DR, SC←0

$D_{10}T_5$: AC←AC NAND D R, SC←0

p:SC←0

INR is set to be complement of CLR so that if CLR is not active then SC is constantly incrementing.

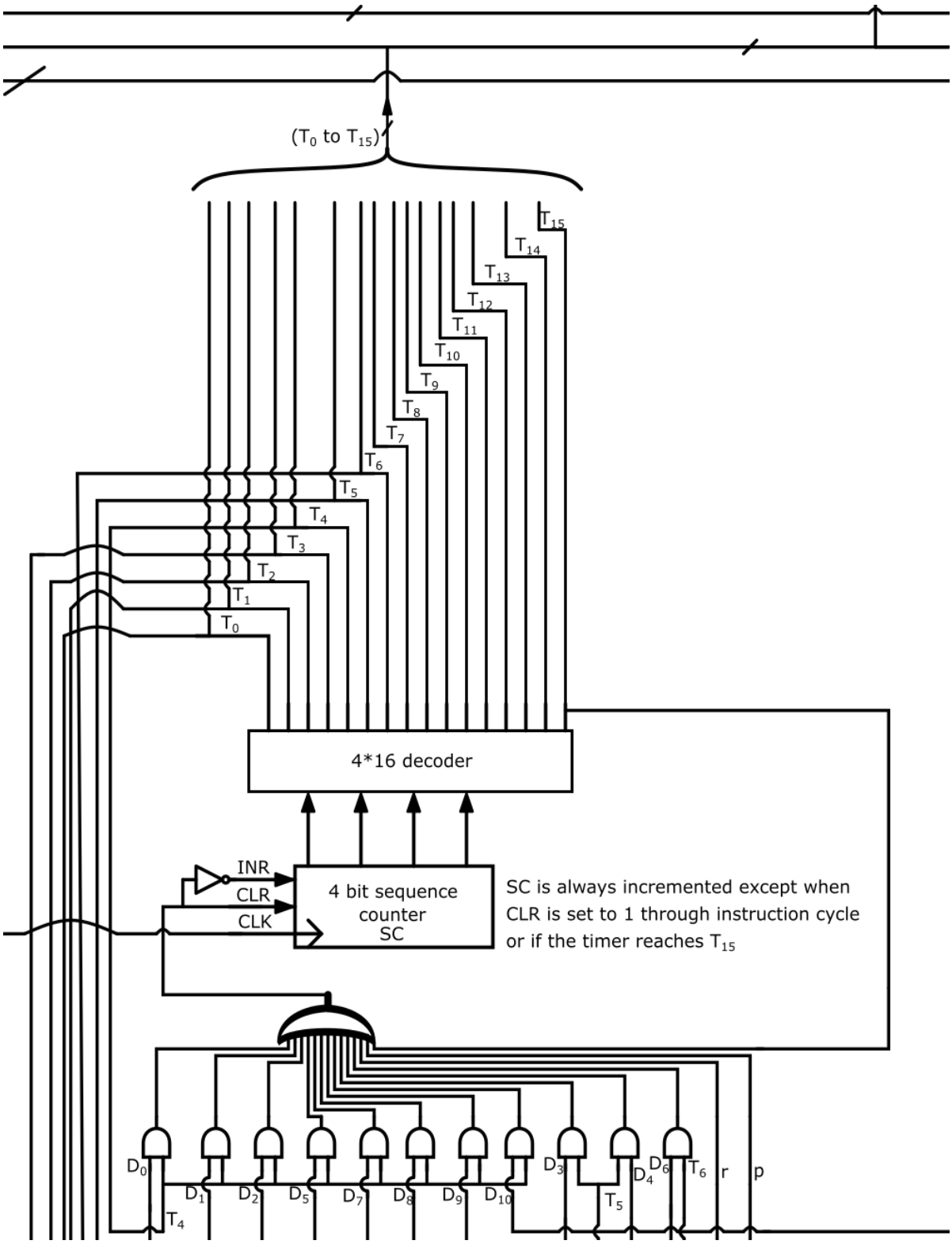

Figure 5: Sequence Counter

Program Counter (PC) : Program Counter register is used to point to the memory address to be loaded into IR. It is constantly incremented to processing list of instructions.

Logic for PC can be obtained by OR-ing below conditions:

LD:

$D_4.T_4$: PC←AR, SC←0

$D_5.T_5$: PC←AR, SC←0

CLR:

RT_1 : M[AR] ← TR, PC←0

INR:

$R'T_1$: IR←M [AR], PC ← PC+1

pB_2: If (FGI=1) then PC ← PC +1

pB_3: If (FGO=1) then PC ← PC+1

rB_7 : If (AC(19)=0), then PC←PC+1

rB_8 : If (AC(19)=1), then PC←PC+1

rB_9 : If AC=0 then, PC←PC+1

rB_{10} : If (E=0), then PC←PC+1

RT_2 : PC← PC + 1, IEN ← 0, R←0, SC←0

D_6T_6 : M [AR]←DR, if (DR=0) then (PC←PC+1), SC←0

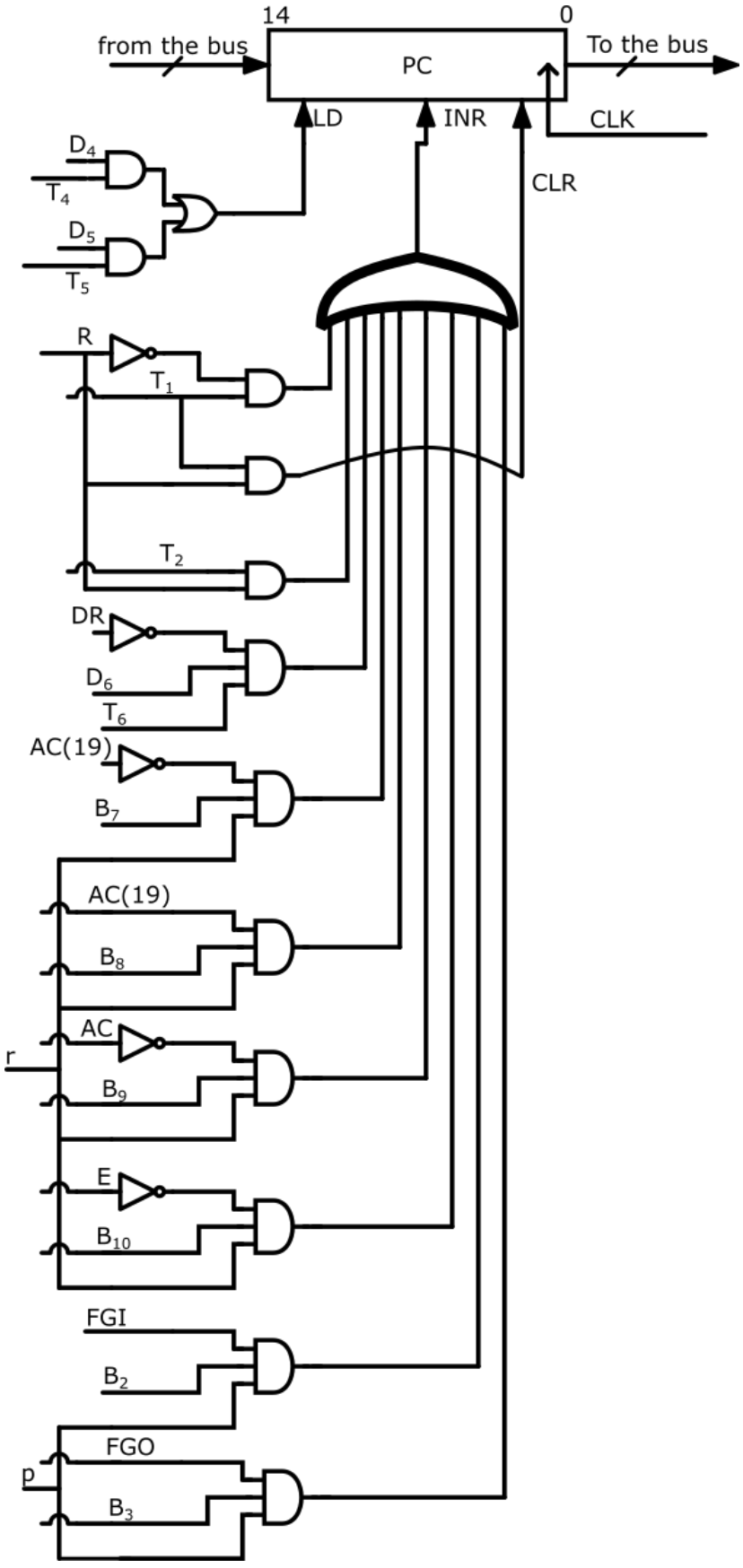

Figure 6: Program Counter

Address Register (AR): Address register is used to hold address stored in IR. We use only AR bits to access memory from memory unit.

Logic for AR can be obtained by OR-ing below conditions.

LD

$R'T_0$: AR←PC

$D_{15}'IT_3$:AR←M [AR]

$R'T_2$: I←IR [19], D_0...D_{15}←IR [18,17,16,15], AR←IR[14-0]

CLR

RT_0 : AR←0, TR←PC

INR

$D_5.T_4$: M[AR]←PC, AR←AR+1

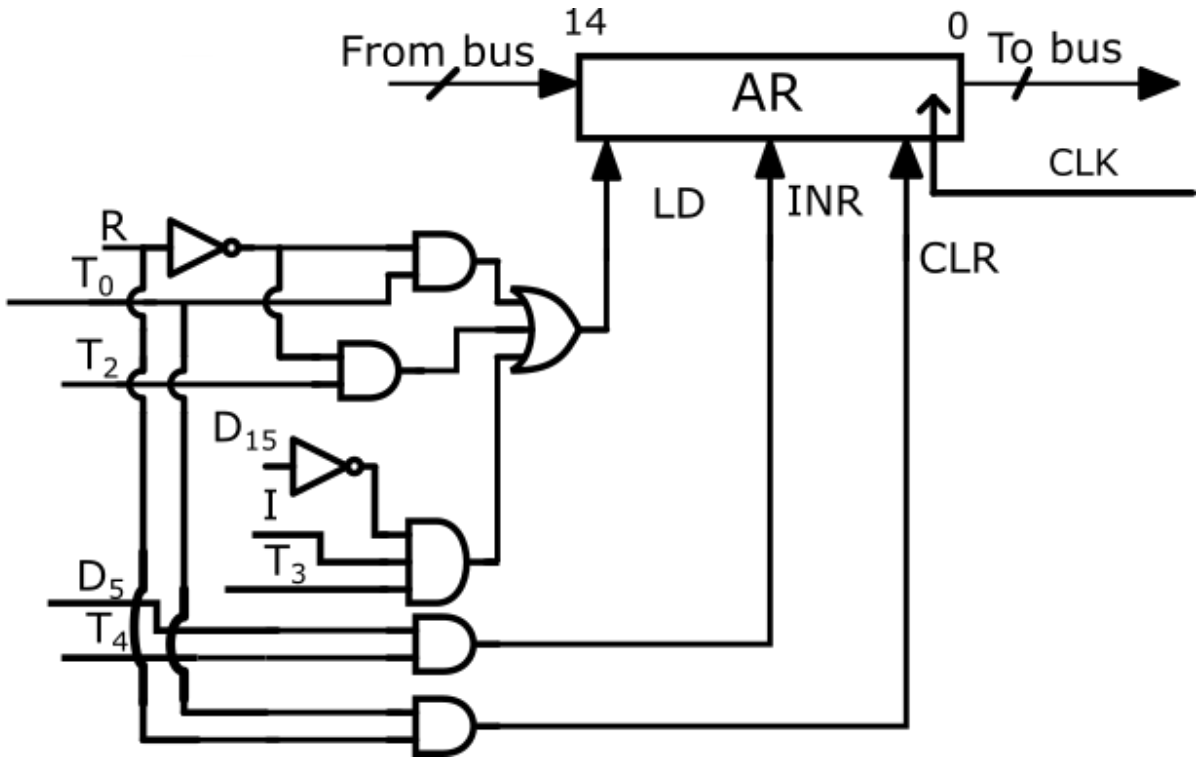

Figure 7: Address Register

Instruction Register (IR): Much about IR has already been discussed above. Its logic can be obtained by OR-ing below conditions.

LD

$R'T_1$: IR←M [AR], PC ← PC+1

The first bit of IR is used as an I bit and bit 18-15 are decoded using 4*16 decoder as shown in figure below along with design of decoder for RRI and IOI:

Designing decoder for RRI and IOI:

Similar to MRI instruction decoder RRI and IOI also needs decoding. D_{15} decoder output and I bit can identify RRI and IOI but further instruction needs to be decoded from the address bits i.e. 15 bits. But we don't have to decode all 15 bits because we only have total of 20 RRI and IOI altogether so we decode the changing bits.

Let's take a closer look at our bits layout in IR

0 1 1 1	1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
7	8	0	0	0

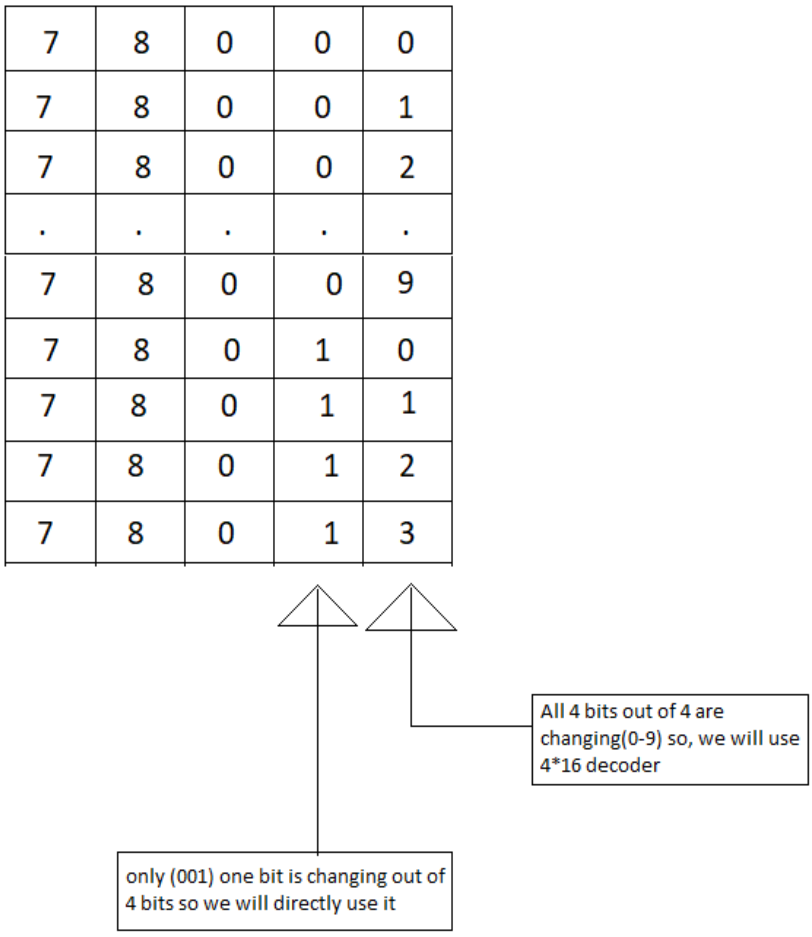

Figure 8: Bit layout in IR

4*16 decoder gives us 10 RRI instructions when D_{15} is active i.e. $B_0 \leftarrow B_9$. As for remaining RRI we AND 5^{th} bit with decoder output to get B_{10}, B_{11}, B_{12}, B_{13}. When D_{15} and I both are active same decoder output $B_0 - B_5$ gives 6 IOI instructions.

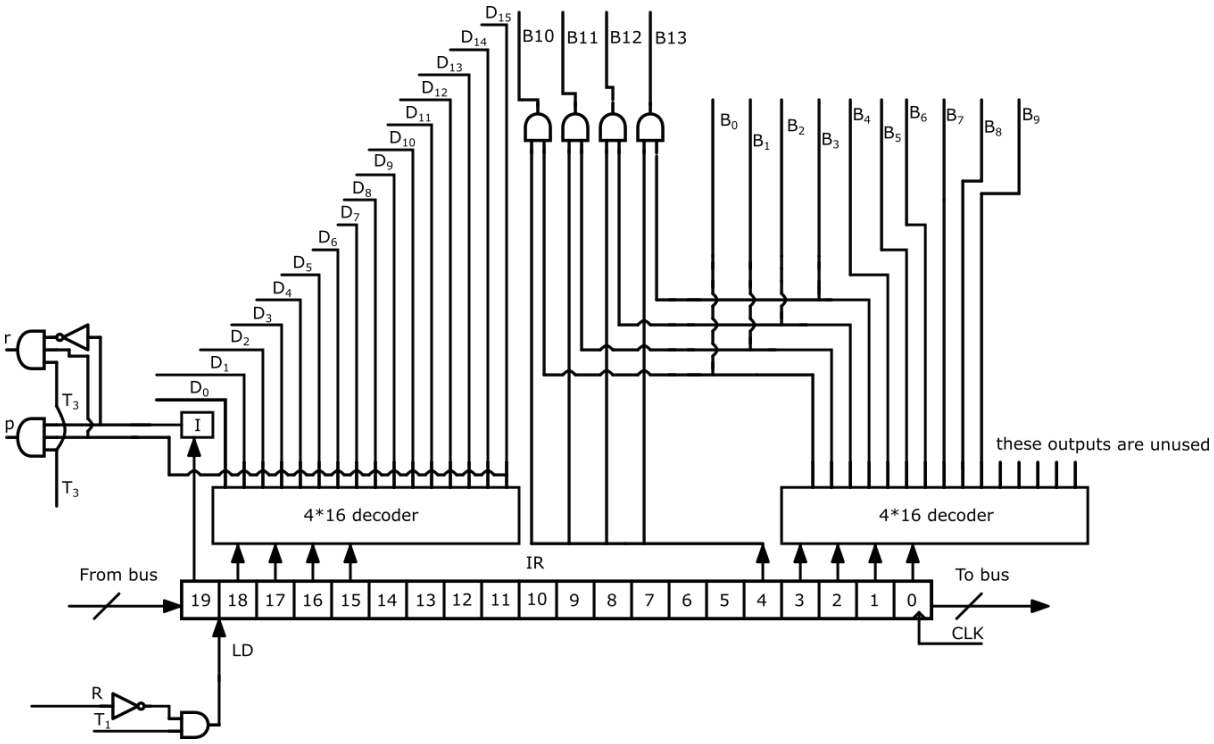

Figure 9: Instruction Register with Decoders

Data Register (DR): Data register holds data stored in memory identified by AR bits. Logic for DR can be obtained by OR-ing below conditions.

LD

D_6T_4 : DR←M [AR]

$D_0.T_4$: DR←M [AR]

D_1T_4 : DR←M [AR]

$D_2.T_4$: DR←M [AR]

D_7T_4 : DR←M [AR]

D_8T_4 : DR←M [AR]

D_9T_4 : DR←M [AR]

$D_{10}T_4$: DR←M [AR]

INR

D_6T_5 : DR←DR+1

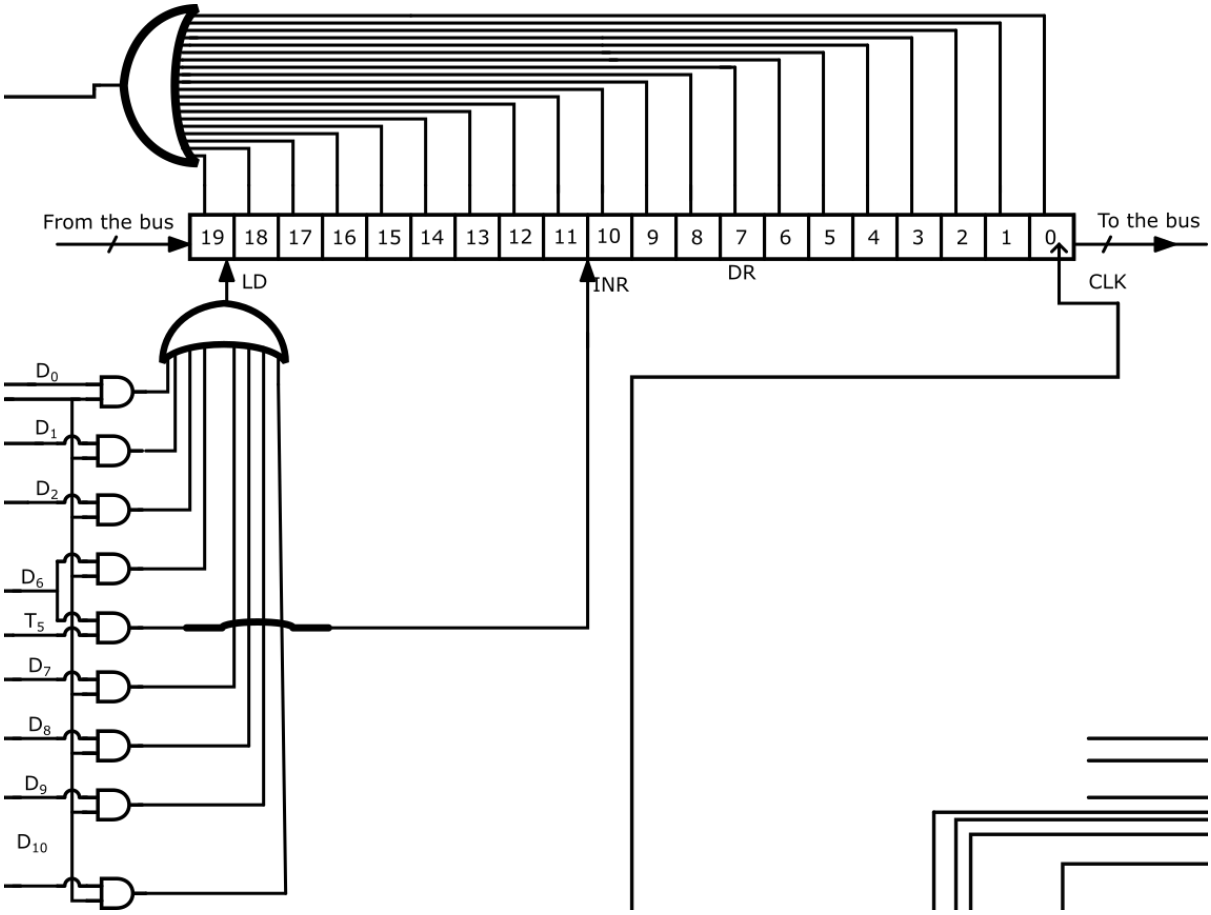

Figure 10: Data Register

Temporary register (TR): Temporary register like the name says is used for temporarily holding data.

LD

RT_0 : AR←0, TR←PC

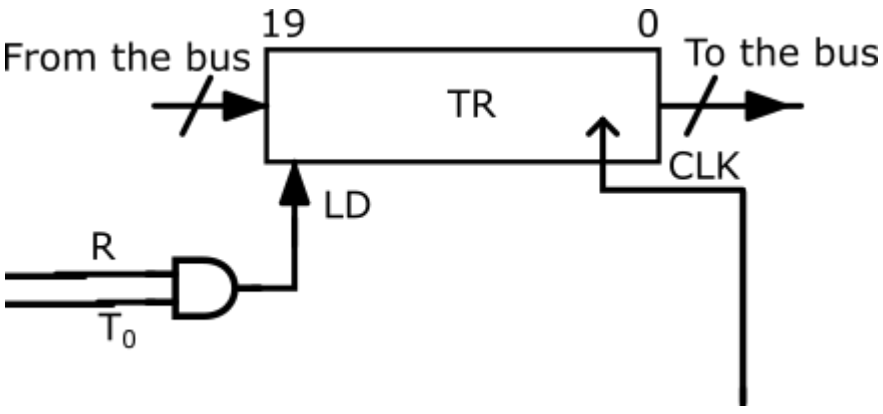

Figure 11: Temporary Register

OUTR: Output register holds data loaded by AC onto the bus for output.

LD:

pB_1: OUTR ← AC (0-9), FGO ← 0

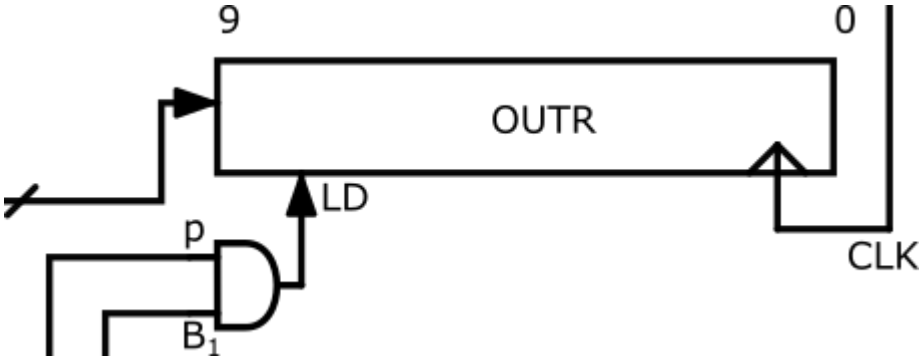

Figure 12: Output Register

INPR: Input register holds data input from the user. It only loads data directly into AC.

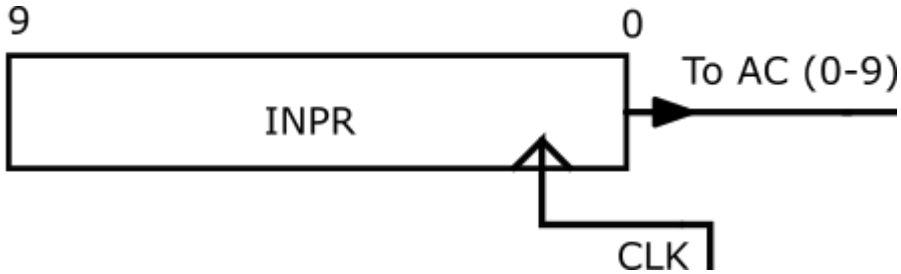

Figure 13: Input Register

Memory: Much about memory has already been discussed above. Unlike other registers memory is a stack of register. It has 2 input instructions i.e. Read and Write. Based on these instructions it either downloads or uploads a data from or to the bus.

Memory read:

$R'T_1$: IR←M [AR], PC ← PC+1

$D_0.T_4$: DR←M [AR]

D_1T_4 : DR←M [AR]

$D_2.T_4$: DR←M [AR]

D_6T_4 : DR←M [AR]

D_7T_4 : DR←M [AR]

D_8T_4 : DR←M [AR]

D_9T_4 : DR←M [AR]

$D_{10}T_4$: DR←M [AR]

D_{15}'IT_3:AR←M [AR]

Memory write:

$D_3.T_4$: M [AR]←AC, SC←0

$D_5.T_4$: M[AR]←PC, AR←AR+1

D_6T_6 : M [AR]←DR, if (DR=0) then (PC←PC+1), SC←0

RT_1 : M[AR] ← TR, PC←0

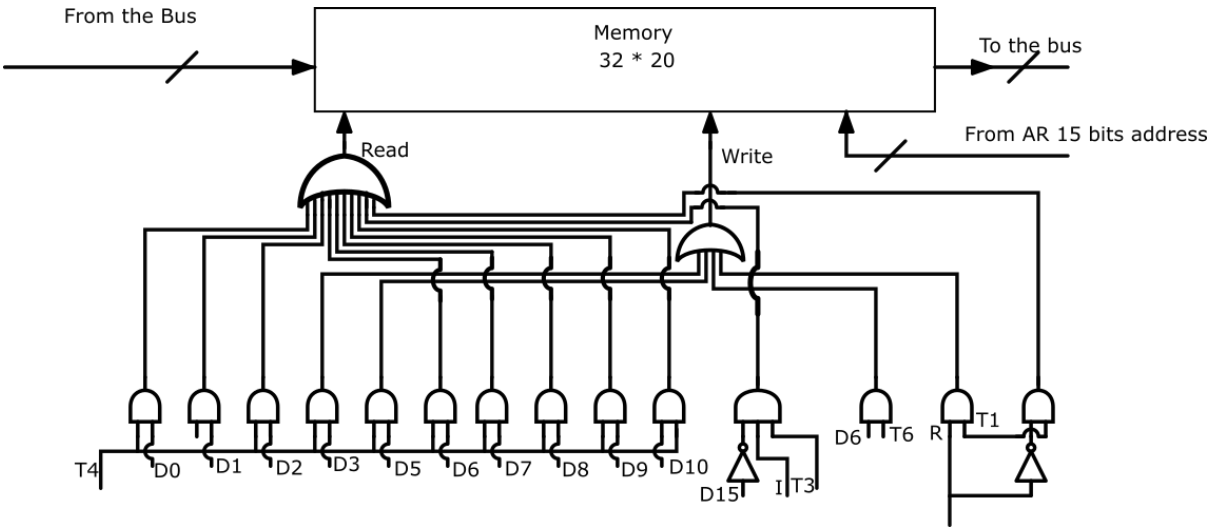

Figure : Memory

E-flip flop:

Design for E- flip flop is slightly more complicated than other flip flops. This is because it is used for storing data and not instructions or flags. But first let's take a look at all instructions storing data in E-flip flop.

D_1T_5 : E←C_{out}

D_8T_8 : E←C_{out}

rB_5 : E←AC(19)

rB_3 : E←E'

rB_4 : E←AC(0)

rB_1 : E←0

Normally only rB_1 would be connected to k and all the other would be connected to J as shown in the figure below:

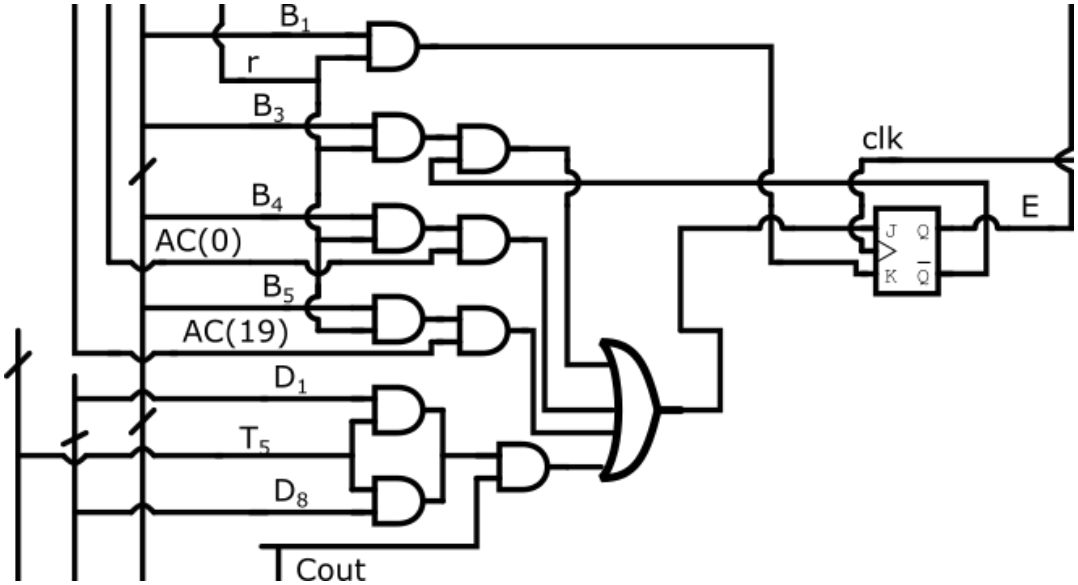

Figure : Wrong Implementation of Flip Flop

But this wouldn't work because C_{out}, AC(0), AC(19) and E' all can have 0 values. Since all instructions are mutually exclusive for $rB_1 = 0$ and any other of above instructions are true then for C_{out}, AC or E' = 0, J = 0 and k = 0.

Eg. for $rB_1 = 0$, $rB_5 = 1$ and AC(19) = 0.

In this case we won't be able to store zero value into E-flip flop instead the flip flop retains previous value because k = 0 and J also remains 0 since AC(19) is currently 0.

To solve this problem, we came up with a new Boolean expression. Let's make a solution for D_1T_5 and C_{out} as for all other expression we can mimic the same solution.

D_1T_5	C_{out}	J	K
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	0

Table 5: J and K for E flip flop

From above table we can see that for $D_1T_5 = 1$ if $C_{out} = 0$ we set k as 1 storing zero value and if $D_1T_5 = 1$ and $c_{out} = 1$ we set J = 1 storing 1 value of C_{out}.

We can deduce from above table,

$J = D_1.T_5.C_{out}$, $k = D_1T_5.C_{out}$'

Similarly,

$J = D_8T_5C_{out}, k = D_8T_5C_{out}'$

$J = rB_5.AC(19), k = rB_5.AC(19)'$

$J = rB_4.AC(0), k = rB_4.AC(0)'$

$J = rB_3.E, k = rB_3.E'$

As for:

rB_1: J=0, k = rB_1 which sets flip flop as 0.

Hence, the final logic can be obtained by OR-ing all J's and all k's as shown in the figure below:

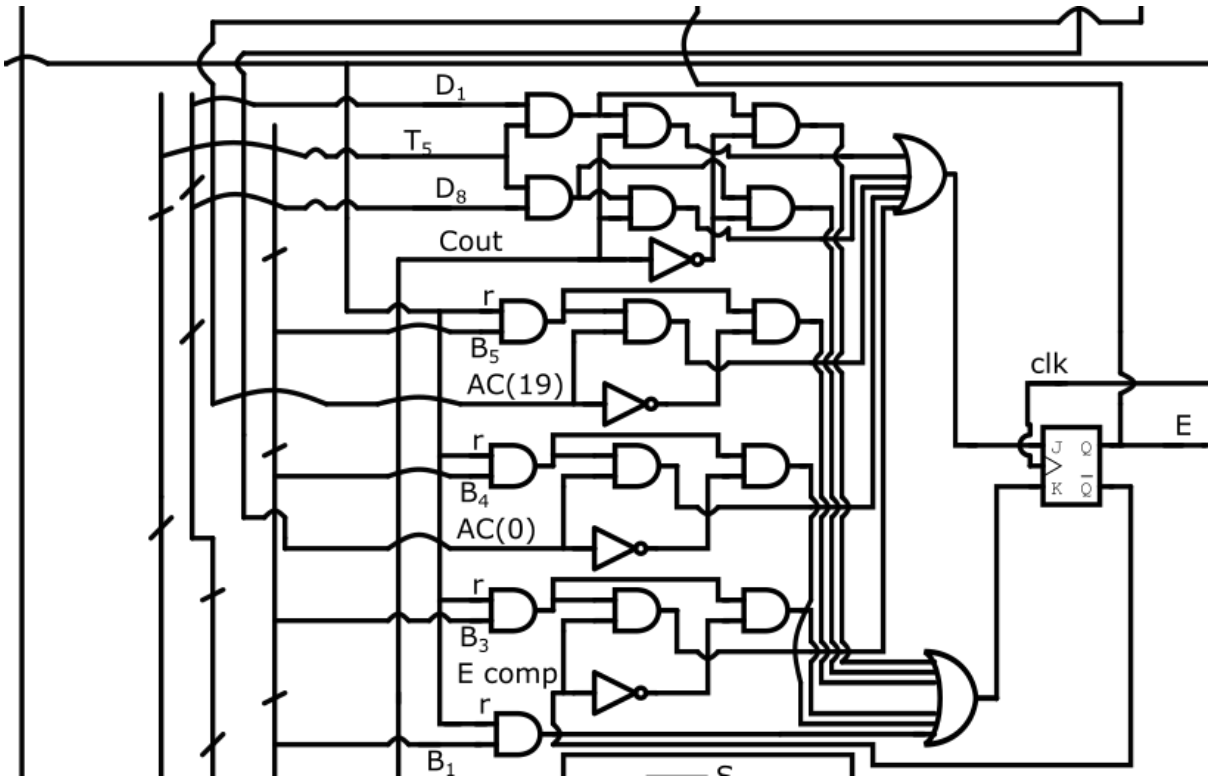

Figure 14: Right Implementation of E flip flop

Start-Stop flip-flop: It is a JK flip-flop that only resets for RB_{11} else it stays 1.

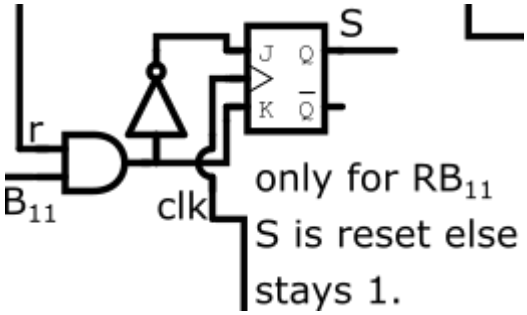

Figure 15:Start-stop flip flop

FGI flip flop: It is a JK flip flop that is set to one on input and it retains its value until PB_0=1 and the flip flop is reset.

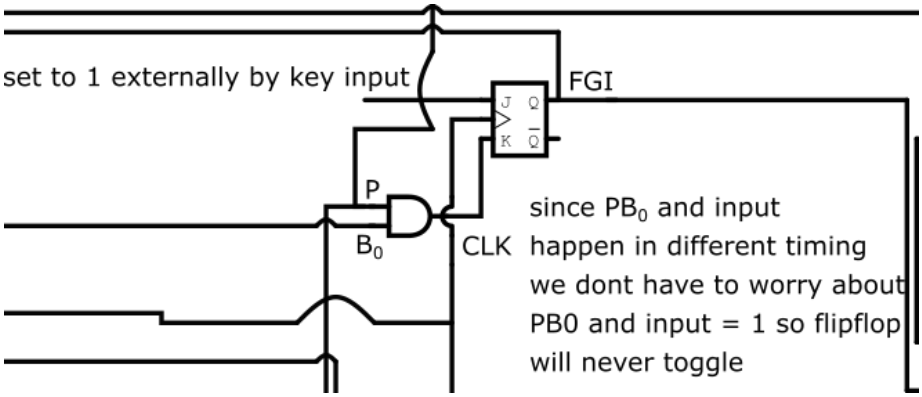

Figure 16: FGI Flip flop

FGO flip flop: It is a JK flip flop that is set to one on output request and it will retain it's value until PB_1=1.

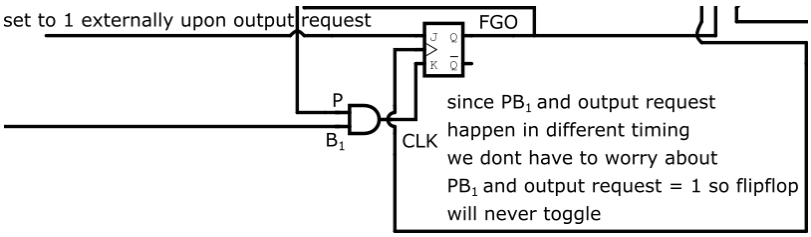

Figure 17: FGO Flip flop

IEN flip flop: Interrupt enable flag is stored in a JK flip flop whose expressions are listed below:

RT_2:IEN← 0

PB_4:IEN← 1

PB_5:IEN← 0

K=PB_5+RT_2

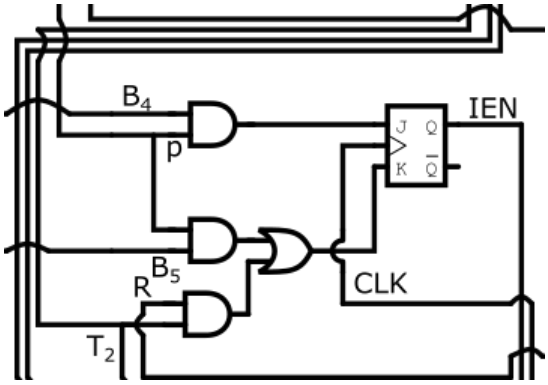

Figure 18: IEN Flip flop

R flip flop: It is a JK flip flop that sets to 1 whenever:

IEN(FGI+FGO) .T_0'.T_1'.T_2': R← 1

And resets when,

RT_2: R← 0

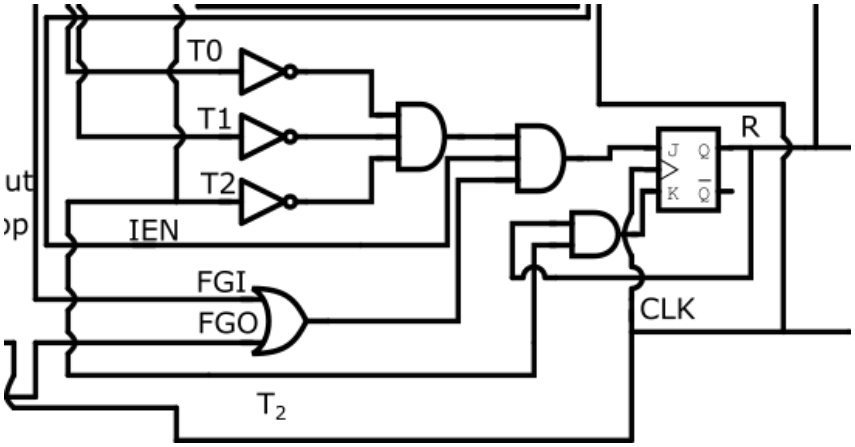

Figure 19: R Flip flop

Common Bus System (CBS): Peer to Peer connection among registers is too complex and requires too many connection lines so, we use a common bus to connect all these registers or at least those that need the bus. INPR and OUTR does not load anything to the bus, so aside of these two registers, we constructed a CBS that selects one of 7 registers i.e AR, PC, DR, AC, IR, TR and M[AR] (here M[AR] refers to one register from the stack of registers in the memory identified by the address stored in AR).

We used 8*1 MUX to select one of the 7 registers inputs and use one extra input for null value i.e empty input (no register selected) as shown below in the figure. Since, we have max 20 bit registers we will need 20 MUX from 8*1 MUX (0) to 8*1 MUX (19), along with three selection lines to choose one of 8 inputs among all MUX. 20 MUX all together will contribute to 20 bit CBS.

For MUX 15 to MUX 19 we have set AR and PC inputs to 0 since they only have 15 bits.

Input for 8*3 encoder							**Selection Lines for MUX**			**Register**
x_1	x_2	x_3	x_4	x_5	x_6	x_7	s_2	s_1	s_0	
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	Memory

Table 6: Encoder to selection line

$S_0 = x_1+x_3+x_5+x_7$

$S_1 = x_2+x_3+x_6+x_7$

$S_2 = x_4+x_5+x_6+x_7$

Each input for the encoder selects a unique register to be loaded in the bus. We only load the data to the bus however downloading it into registers is done by the register load or LD logic. So, while constructing "select" logic for a register we only considered micro-operation involving that register as the source like PC←AR etc. and the destination being another register through the bus like DR← M[AR].

In some cases, like in AC←DR bus is not involved in fact AC does not receive any data from the bus, it only receives data from either ALU or E Flip flop and only loads the data to the bus. So, any operation involving AC as a destination is not considered.

OUTR Register only downloads the data from the bus and doesn't load anything to it so it's not considered. INPR is not connected to the bus at all so it is also not considered so we are only concerned with 6 registers and 1 memory unit.

Logic for x_1 i.e, AR

D_4T_4 : PC← AR

D_5T_5 : PC← AR

Logic for x_2 i.e, PC

$R'T_0$: AR←PC

D_5T_4 : M[AR]←PC

Logic for x_3 i.e, DR

D_6T_6 : M[AR]← DR

Logic for x_4 i.e, AC

D_3T_4 : M[AR]←AC

PB_1 : OUTR←AC(0-9)

Logic for x_5 i.e, IR

$R'T_2$: AR← IR(14-0)

Logic for x_6 i.e, TR

RT_1 : M[AR]← TR

Logic for x_7 i.e Memory

R'T_1 : IR←M [AR], PC ← PC+1

$D_0.T_4$: DR←M [AR]

D_1T_4 : DR←M [AR]

$D_2.T_4$: DR←M [AR]

D_6T_4 : DR←M [AR]

D_7T_4 : DR←M [AR]

D_8T_4 : DR←M [AR]

D_9T_4 : DR←M [AR]

$D_{10}T_4$: DR←M [AR]

D_{15}'IT_3:AR←M [AR]

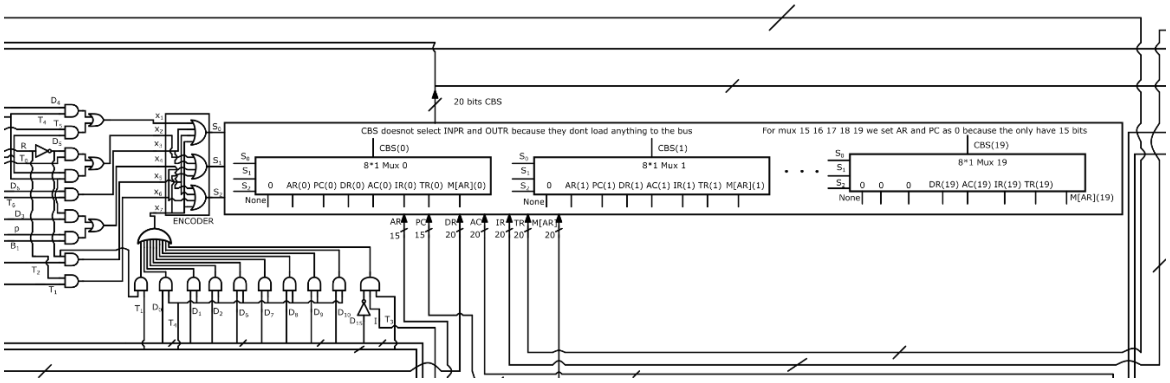

Figure 20: Common Bus System

ALU (Arithmetic Logic Shift Unit):

ADD and Subtraction: These two operations are performed by a single unit consisting of 2*1 MUX 1 inverter two AND gates and a single full adder. In this way not only can we simplify two operations but also make individual ALU units compact.

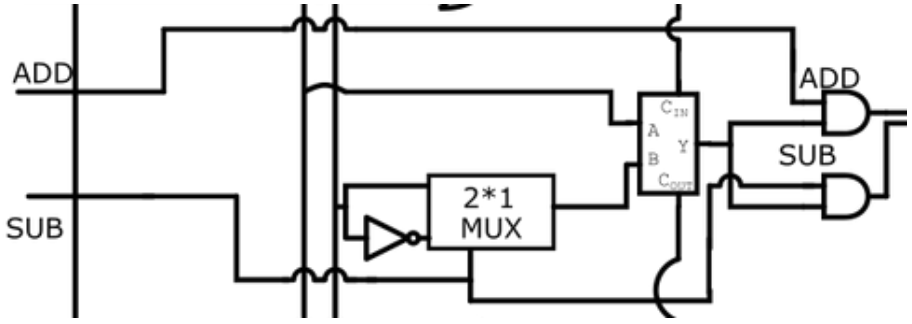

Figure 21: Add and subtract logic

Addition: For addition we will pass '0' as selection signal for MUX selecting AC(i) an DR(i) which will be added by a full adder under C_{in} 0 and releases a C_{out} which will be later stored in E flip flop. Under the control signal AND it will be passed into AC.

$AC(i) + DR(i) + C_{in}(0) = result + C_{out}$

$E \leftarrow C_{out}$

$AC(i) \leftarrow result$

Subtraction: A subtraction using 2's complement is AC(i)+DR(i)'+1

Hence, MUX selection signal becomes one. For C_{in} 1 will be passed.

$AC(i)+DR(i)'+C_{in}(1)= Result + C_{out}$

$E \leftarrow C_{out}$

$AC(i) \leftarrow result$

NAND: Performed using two inverters, one OR gate and the result is passed through AC when NAND control signal is 1.

$AC(i)' + DR(i)'$

DR or LDA: Under signal DR, DR(i) is loaded to the AC

$AC(i) \leftarrow DR(i)$

XOR: Under signal XOR

$AC(i) \leftarrow AC(i) \oplus DR(i)$

COMP: When COMP = 1

$AC(i) \leftarrow AC(i)'$

IR: When IR=1

$AC(i) \leftarrow IR(i)$

DRCOMP: When DRCOMP = 1

$AC(i) \leftarrow DR(i)'$

INPR: When INPR = 1

$AC(i) \leftarrow INPR(i)$

Since INPR is only 10 bits it will fill up AC(0-9) using ALU units (0-9) , all other bits will be zero because ALU units (10-19) will stay low.

SHR: When SHR = 1

$$AC(i) \leftarrow AC(i + 1)$$

i.e, for $AC(18) \leftarrow AC(19)$

But if i = 19 then i + 1 = 20, 20^{th} bit doesn't exist so the signal for AC(i + 1) will remain low hence ALU will store 0 in 19^{th} bit but immediately E flip flop will store its value in 19^{th} bit in the same clock cycle making it a circular right shift.

SHL: For SHL = 1

$$AC(i) \leftarrow AC(i - 1)$$

i.e, for $AC(17) \leftarrow AC(16)$

If i = 0 then AC(i - 1) = AC(-1) which does not exist so AC(i - 1) signal will stay low and ALU will store 0 in 0^{th} bit. But immediately E flip flop will store its value in 0^{th} bit in the same clock cycle making it a circular left shift.

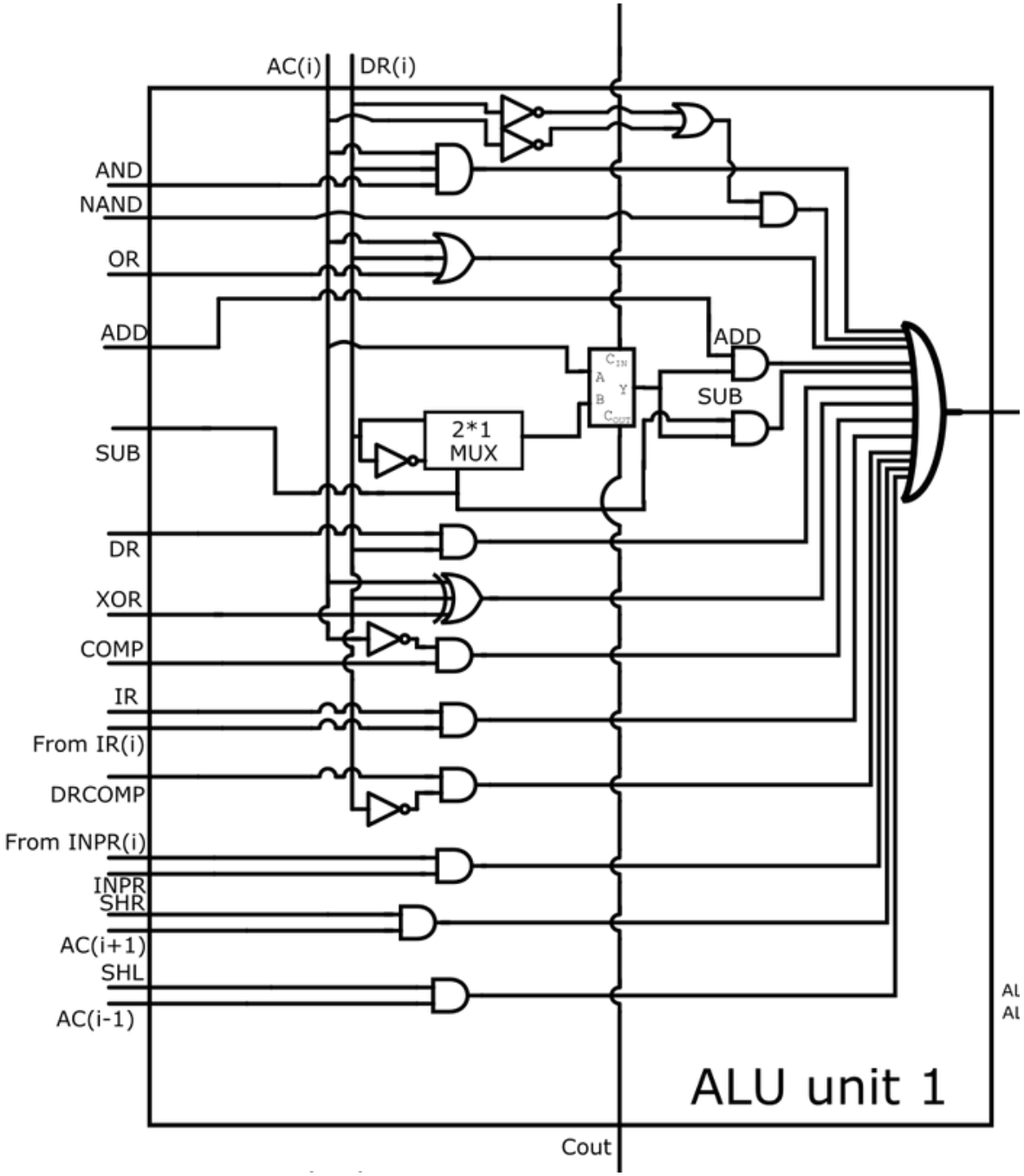

Figure 22: Single stage ALU

AC(ACCUMULATOR): It's a 20-bit register used for temporarily storing intermediate arithmetic and logical operation values. Unlike most other registers it doesn't receive any values from CBS, it only loads values to the CBS. Any value stored in AC comes from ALU. List of LD, INR and CLR logic for AC are listed below.

LD:

D_0T_5 : AC← AC AND DR, SC←0

D_1T_5 : AC←AC+DR, E←C_{out}, SC←0

$D_2.T_5$: AC←DR, SC←0

D_7T_5 : AC←AC v DR, SC←0

D_8T_5 : AC←AC - DR, E_S←C_{out}, SC←0

D_9T_5 : AC←AC⊕DR, SC←0

$D_{10}T_5$: AC←AC NAND DR, SC←0

rB_2 : AC←AC

rB_4 : AC←shr (AC), AC(19)←E, E←AC(0)

rB_5 : AC←shl (AC), AC(0)←E, E←AC(19)

PB_0 :AC(0-9)←INPR, FGI←0

rB_{12} : AC←IR

rB_{13} : AC←DR

CLR:

rB_0 : AC←0

INR:

rB_6 : AC←AC+1

These control signals are also shared by the ALU for its operations.

In addition to these signals AC (9) and AC (19) also receives bits from E flip flop when SHL and SHR signals are active.

AC (19) ←E.SHR

AC (0) ←E.SHL

For SHR or SHL =1, AC receives whatever is stored in E flipflop.

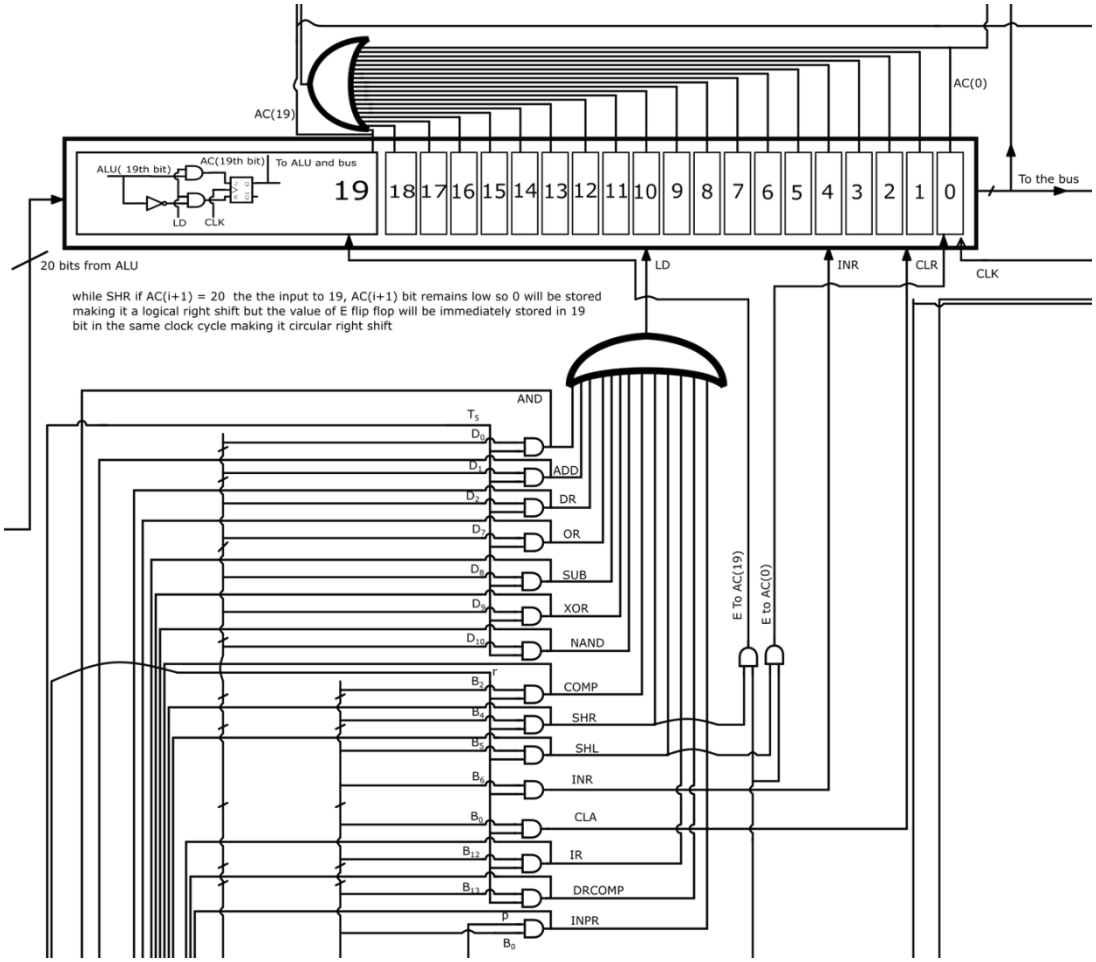

Figure 23: Accumulator

For 20-bit operation, the ALU needs to have 20th stages stacked on top of each other with carry in received by the first stage and carry out propagated down the rest of ALU stages until it is finally processed out of 19th stage and stored in E flipflop. Shared control signal between complete 20 stages of ALU with AC and E flip flop along with data transmission between these units are further made clear by following combined figure of AC, ALU and E flip flop.

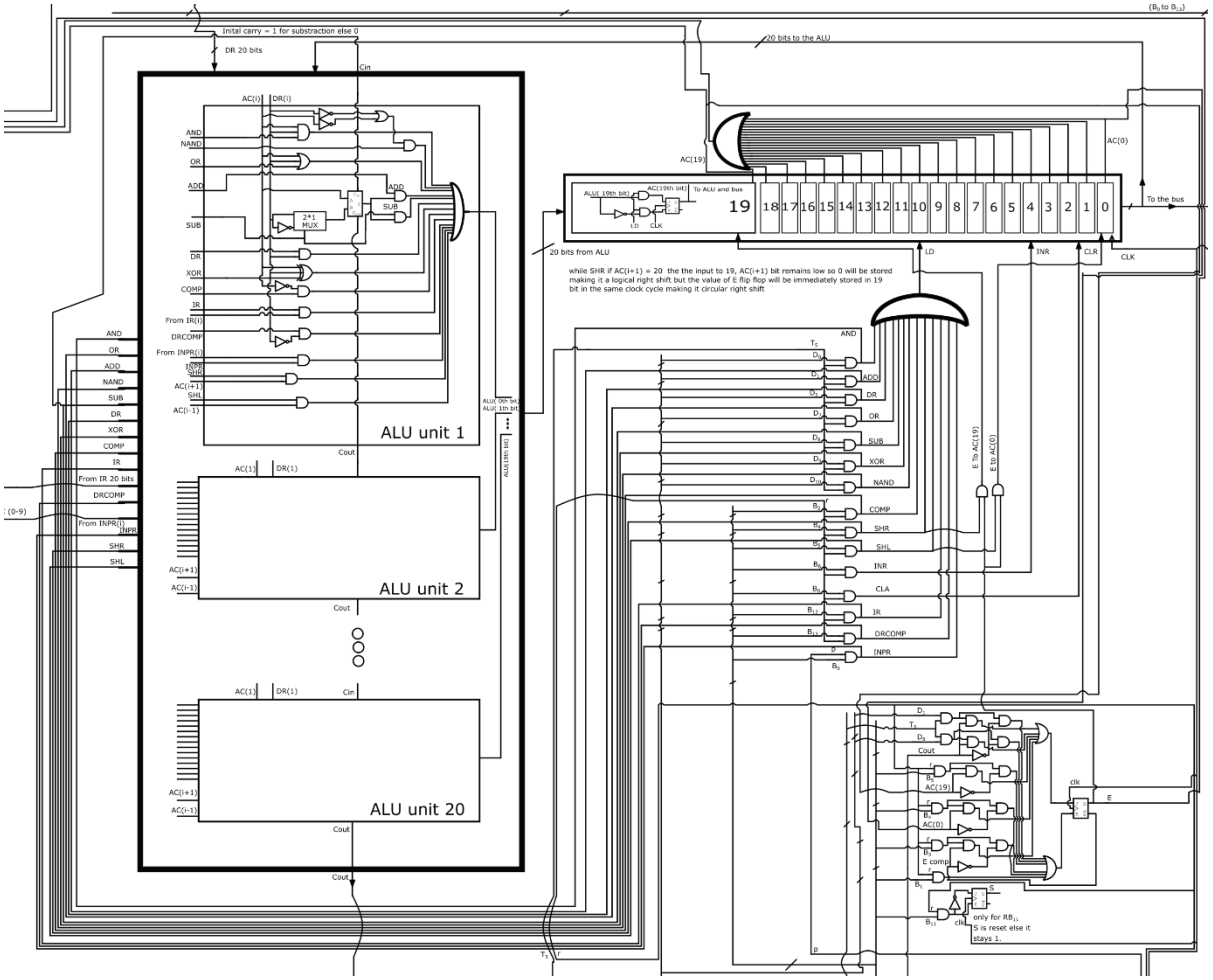

Figure 24: ALU with AC and E flip flop

Chapter 4: Conclusion

Using design consideration for 32k*20 Memory we designed a basic computer model with 8 registers. PC, AR, DR, TR, OUTR, INTR, IR, and AC connected by a CBS. We designed an ALU with 20 stages each performing 13 operations over data of DR, IR and AC and storing processed data over AC. We also designed an interrupt cycle for input and output of data to and from AC.

This basic model also has few extra instructions of our own design as oppose to the 4*16 design from our course.