



PROJECT REPORT

B. Tech (CSE(AI)) (2021 Batch)

Declaration

We hereby declare that this project submitted to Centre is a record of original work done under the guidance of

Dr. Jyothish Lal G, Faculty Associate of the department of elements of computing systems (EOC), and Dr. Sachin Kumar S, Faculty Associate of the department of object – oriented programming (OOP), Amrita University Coimbatore.

Date: 14.02.2022

Place: Ettimadai

ACKNOWLEDGMENT

The completion of this project has been made possible by the efforts of many. We would like to thank Dr. Sasangan Ramanathan, Dean- Engineering, our faculty for the course 19AIE101, Dr. Jyothish Lal G, and our faculty for the course 21AIE105, Dr. Sachin Kumar S for their unstinting support. We would also like to extend our gratitude to our friends and family who have aided us throughout the project.

Elements of Computing Systems & Object Oriented Programming

TEAM MEMBERS

VIKHYAT BANSAL [CB.EN. U4AIE21076]

VIGNESH M. [CB.EN. U4AIE21076]

MARREDDY MOHIT SASANK REDDY [CB.EN. U4AIE21076]

YARRAM SRI SATHWIK REDDY [CB.EN. U4AIE21076]

TOPIC

- **Implementation of CPU in HACK COMPUTER and JAVA**
- **Implementation of Synchronous BCD Counter using JK Flip Flop in HDL and JAVA**

Abstract

A central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control and input/output (I/O) operations specified by the instructions.

In this we will be creating a synchronous BCD Counter that will be synchronized with the clock to an extent that each time pulse will generate equivalent 4-bit output in output pins and will repeat itself after every 10th pulse.

Introduction

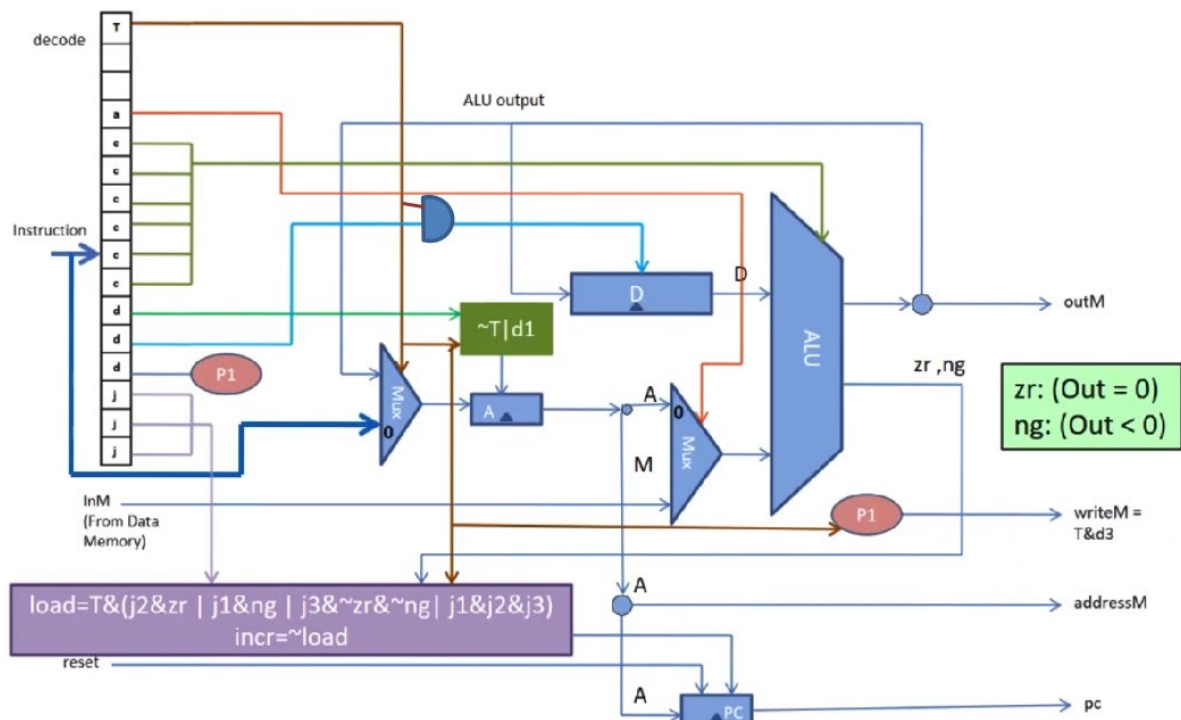
Consists of an ALU and a set of registers, designed to fetch and execute instructions written in the Hack machine language.

In particular, functions as follows:

Executes the input instruction according to the Hack machine language specification. The D and A in the computer language refers to CPU-main registers, while M refers to the external memory location addressed by A, i.e., to Memory[A] ($M = \text{RAM}[A]$). The inM input holds the value of that location. If the current instruction needs to write a value to M, the value is taken in outM, the address of the target location is taken in the addressM output, and the writeM control bit is asserted. (When writeM=0, any value might appear in outM). The outM and writeM outputs are combinational: they are affected instantaneously by the execution of the current instruction. The addressM and PC outputs are clocked: and they are affected by the execution of the instruction, they commit to their new values only in the next time pulse. If reset=1 then the CPU jumps to address 0 (i.e., sets pc=0 in the next time unit) rather than to the address resulting from executing the current instruction.

JK flip – flop in some ways is a modified version of SR flip – flop. Logic diagram consists of four input NAND gates and the inputs are replaced with J and K from S and R. A binary coded decimal decade counter exhibits a binary sequence and goes from 0000 to the 1001 state. Rather than going from the 1001 state to the 1010 state and so on, instead it recycles to the 0000 state.

CPU Chip Design



Components (Chips) Used in CPU Design

- Logic Gates (NAND, AND, OR, NOT,)
- ALU
- Add16, And16
- Bit
- Built-In Chips (A-Register, D-Register)
- MUX 16
- Program Counter

```
CHIP Not {  
  IN in;  
  OUT out;  
  
  PARTS:  
    Nand (a=in, b=in, out=out);  
}
```

```
CHIP And {  
  IN a, b;  
  OUT out;  
  
  PARTS:  
    Nand (a=a, b=b, out=nandOut);  
    Not (in=nandOut, out=out);  
}
```

```
CHIP Or {  
  IN a, b;  
  OUT out;  
  
  PARTS:  
    Not (in=a, out=nota);  
    Not (in=b, out=notb);  
    Nand (a=nota, b=notb, out=out);  
}
```

```
CHIP Xor {  
  IN a, b;  
  OUT out;  
  
  PARTS:  
    Not (in=a, out=nota);  
    Not (in=b, out=notb);  
    And (a=a, b=notb, out=x);  
    And (a=nota, b=b, out=y);  
    Or (a=x, b=y, out=out);  
}
```

```

CHIP Mux16 {
    IN a[16], b[16], sel;
    OUT out[16];

    PARTS:
    Mux(a=a[0], b=b[0], sel=sel, out=out[0]);
    Mux(a=a[1], b=b[1], sel=sel, out=out[1]);
    Mux(a=a[2], b=b[2], sel=sel, out=out[2]);
    Mux(a=a[3], b=b[3], sel=sel, out=out[3]);
    Mux(a=a[4], b=b[4], sel=sel, out=out[4]);
    Mux(a=a[5], b=b[5], sel=sel, out=out[5]);
    Mux(a=a[6], b=b[6], sel=sel, out=out[6]);
    Mux(a=a[7], b=b[7], sel=sel, out=out[7]);
    Mux(a=a[8], b=b[8], sel=sel, out=out[8]);
    Mux(a=a[9], b=b[9], sel=sel, out=out[9]);
    Mux(a=a[10], b=b[10], sel=sel, out=out[10]);
    Mux(a=a[11], b=b[11], sel=sel, out=out[11]);
    Mux(a=a[12], b=b[12], sel=sel, out=out[12]);
    Mux(a=a[13], b=b[13], sel=sel, out=out[13]);
    Mux(a=a[14], b=b[14], sel=sel, out=out[14]);
    Mux(a=a[15], b=b[15], sel=sel, out=out[15]);
}

```

FullAdder - Notepad

```

File Edit Format View Help
CHIP FullAdder {
    IN a, b, c;
    OUT sum, carry;

    PARTS:
    Xor (a=a, b=b, out=o1);
    Xor (a=o1, b=c, out=sum);
    And (a=a, b=b, out=t1);
    And (a=b, b=c, out=t2);
    And (a=a, b=c, out=t3);
    Or (a=t1, b=t2, out=t4);
    Or (a=t4, b=t3, out=carry);
}

```

```

CHIP HalfAdder {
    IN a, b;
    OUT sum, carry;

    PARTS:
    Xor(a=a, b=b, out=sum);
    And(a=a, b=b, out=carry);
}

```

```
Add16.hdl - Notepad
File Edit Format View Help
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/02/Adder16.hdl

/**
 * Adds two 16-bit values.
 * The most significant carry bit is ignored.
 */

CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
        HalfAdder (a=a[0], b=b[0], sum=out[0], carry=carry1);
        FullAdder (a=a[1], b=b[1], c=carry1, sum=out[1], carry=carry2);
        FullAdder (a=a[2], b=b[2], c=carry2, sum=out[2], carry=carry3);
        FullAdder (a=a[3], b=b[3], c=carry3, sum=out[3], carry=carry4);
        FullAdder (a=a[4], b=b[4], c=carry4, sum=out[4], carry=carry5);
        FullAdder (a=a[5], b=b[5], c=carry5, sum=out[5], carry=carry6);
        FullAdder (a=a[6], b=b[6], c=carry6, sum=out[6], carry=carry7);
        FullAdder (a=a[7], b=b[7], c=carry7, sum=out[7], carry=carry8);
        FullAdder (a=a[8], b=b[8], c=carry8, sum=out[8], carry=carry9);
        FullAdder (a=a[9], b=b[9], c=carry9, sum=out[9], carry=carry10);
        FullAdder (a=a[10], b=b[10], c=carry10, sum=out[10], carry=carry11);
        FullAdder (a=a[11], b=b[11], c=carry11, sum=out[11], carry=carry12);
        FullAdder (a=a[12], b=b[12], c=carry12, sum=out[12], carry=carry13);
        FullAdder (a=a[13], b=b[13], c=carry13, sum=out[13], carry=carry14);
        FullAdder (a=a[14], b=b[14], c=carry14, sum=out[14], carry=carry15);
        FullAdder (a=a[15], b=b[15], c=carry15, sum=out[15], carry=carry16);
}
```

```
CHIP Or8Way {
    IN in[8];
    OUT out;

    PARTS:
        Or(a=in[0], b=in[1], out=or01);
        Or(a=in[2], b=in[3], out=or23);
        Or(a=in[4], b=in[5], out=or45);
        Or(a=in[6], b=in[7], out=or67);
        Or(a=or01, b=or23, out=or0123);
        Or(a=or45, b=or67, out=or4567);
        Or(a=or0123, b=or4567, out=out);
}
```


In JAVA:

```
public class Nand {  
    public static int nand (int a, int b){  
        if(a==1 && b==1){  
            return 0;  
        }  
        else {  
            return 1;  
        }  
    }  
}
```

```
public class Not {  
    public static int not(int a){  
        return Nand.nand(a,a);  
    }  
}
```

```
public class Not16 {  
    public static int [] not16(int [] in){  
        // It performs the same function as a not gate, however it works with 16-bit arrays.  
        |  
        int [] a = new int[16];  
        System.arraycopy(in,  srcPos: 0, a,  destPos: 0, a.length);  
        int [] not = new int[16];  
        for (int i = 0; i < a.length; i++){  
            not[i] = Not.not(a[i]);  
        }  
        return not;  
    }  
}
```

```
public class And {  
    public static int and(int a, int b){  
        // return a&b;  
        return Not.not(Nand.nand(a,b));  
    }  
}
```

```

public class And16 {
    public static int [] and16(int [] a, int [] b){

        int [] and = new int[16];

        for (int i = 0; i < a.length; i++){
            and[i] = And.and(a[i], b[i]);
        }
        return and;
    }
}

```

```

public class Or {
    public static int or(int a, int b){

        // return a|b;
        // Nand of (a complement + b complement)

        return Nand.nand(Not.not(a), Not.not(b));
    }
}

```

```

public class Or16 {
    public static int [] or16(int [] a, int [] b){

        int [] or = new int[16];

        for (int i = 0; i < a.length; i++){
            or[i] = Or.or(a[i], b[i]);
        }
        return or;
    }
}

```

```

package com.TEAM_A11;

public class Or16way {
    public static int or16Way(int [] in) {
        int y = Basic_Gates.Or(in[0],in[1]);
        //y=1
        for (int i=2;i<=14;i++){
            y = Basic_Gates.Or(y,in[i]);
        }
        int out = Basic_Gates.Or(y,in[15]);
        return out;
    }
}

```

```

public class Multiplexer {
    public static int mux(int a, int b, int sel){
        int notSel, aAndNotSel, bAndSel;
        notSel= Not.not(sel);
        aAndNotSel= And.and(a,notSel);
        bAndSel=And.and(b,sel);
        return Or.or(aAndNotSel,bAndSel);
    }
}

```

```

public class Mux16 {
    public static int [] mux16(int [] a, int [] b, int sel){

        int [] mux = new int[16];
        for (int i = 0; i < a.length; i++){
            mux[i] = Multiplexer.mux(a[i], b[i], sel);
        }
        return mux;
    }
}

```

```

4
5     public class Halfadder {
6         public static int [] halfAdder(int a, int b){
7             int [] sumCarry = new int[2];
8             sumCarry[0] = Xor.xor(a,b);
9             sumCarry[1] = And.and(a,b);
10            return sumCarry;
11        }
12    }

```

```

package academy.learnprogramming;

public class Xor {
    public static int xor(int a, int b){
        int aAndNotb, bAndNota;
        aAndNotb= And.and(a,Not.not(b));
        bAndNota=And.and(b,Not.not(a));
        return Or.or(aAndNotb,bAndNota);
    }
}

```

```

public class Fulladder {
    public static int [] fullAdder(int a, int b, int c){

        int [] sumCarry = new int[2];
        int [] hA1 = new int[2];
        int [] hA2 = new int[2];

        hA1 = Halfadder.halfAdder(a,b);
        hA2 = Halfadder.halfAdder(hA1[0],c);

        sumCarry [0] = hA2[0];
        sumCarry [1] = Or.or(hA1[1],hA2[1]);

        return sumCarry;
    }
}

```

```

package com.TEAM_A11;

public class Add16 {
    public static int[] adder(int[] a,int[] b){

        int [] tmp0 = Full_adder.fullAdder(a[15],b[15], carry: 0);
        int[] tmp1 = Full_adder.fullAdder(a[14], b[14], tmp0[1]);
        int[] tmp2 = Full_adder.fullAdder(a[13], b[13], tmp1[1]);
        int[] tmp3 = Full_adder.fullAdder(a[12], b[12], tmp2[1]);
        int[] tmp4 = Full_adder.fullAdder(a[11], b[11], tmp3[1]);
        int[] tmp5 = Full_adder.fullAdder(a[10], b[10], tmp4[1]);
        int[] tmp6 = Full_adder.fullAdder(a[9], b[9], tmp5[1]);
        int[] tmp7 = Full_adder.fullAdder(a[8], b[8], tmp6[1]);
        int[] tmp8 = Full_adder.fullAdder(a[7], b[7], tmp7[1]);
        int[] tmp9 = Full_adder.fullAdder(a[6], b[6], tmp8[1]);
        int[] tmp10 = Full_adder.fullAdder(a[5], b[5], tmp9[1]);
        int[] tmp11 = Full_adder.fullAdder(a[4], b[4], tmp10[1]);
        int[] tmp12 = Full_adder.fullAdder(a[3], b[3], tmp11[1]);
        int[] tmp13 = Full_adder.fullAdder(a[2], b[2], tmp12[1]);
        int[] tmp14 = Full_adder.fullAdder(a[1], b[1], tmp13[1]);
        int[] tmp15 = Full_adder.fullAdder(a[0], b[0], tmp14[1]);

        int[] tmp16 = { tmp15[0], tmp14[0],tmp13[0],tmp12[0],tmp11[0],tmp10[0],tmp9[0],tmp8[0],
            tmp7[0],tmp6[0],tmp5[0],tmp4[0],tmp3[0],tmp2[0],tmp1[0],tmp0[0] };

        return tmp16;
    }
}

```

The Arithmetic Logical Unit

The ALU computes a function on the two inputs, and outputs the result
f: one out of a family of pre-defined arithmetic and logical functions
 Logical functions: And, Or, Not, Xor.

IMPLEMENTATION

It operates on two 16-bit, two's complement values, Outputs a 16-bit, two's complement value.

```

CHIP ALU {
  IN
    x[16], y[16], // 16-bit inputs
    zx, // zero the x input?
    nx, // negate the x input?
    zy, // zero the y input?
    ny, // negate the y input?
    f, // compute out = x + y (if 1) or x & y (if 0)
    no; // negate the out output?

  OUT
    out[16], // 16-bit output
    zr, // 1 if (out == 0), 0 otherwise
    ng; // 1 if (out < 0), 0 otherwise

  PARTS:

  Mux16(a=x,b=false,sel=zx,out=x1);
  Mux16(a=y,b=false,sel=zy,out=y1);

  Not16(in=x1,out=notx1);
  Mux16(a=x1,b=notx1,sel=nx,out=x2);

  Not16(in=y1,out=noty1);
  Mux16(a=y1,b=noty1,sel=ny,out=y2);

  Add16(a=x2,b=y2,out=addout);
  And16(a=x2,b=y2,out=andout);

  Mux16(a=andout,b=addout,sel=f,out=fout);

  Not16(in=fout,out=notfout);
  Mux16(a=fout,b=notfout,sel=no,out=out, out[0..7]=outfirst, out[8..15]=outsecond,out[15]=ng);

  Or8Way(in=outfirst,out=zr0);
  Or8Way(in=outsecond,out=zr1);
  Or(a=zr0,b=zr1,out=zr2);
  Not(in=zr2,out=zr);
}

```

```

package com.TEAM_A11;

import java.util.Arrays;

public class ALU {
    public static int [] alu(int zx, int nx, int zy,
        int ny, int f, int no, int [] x, int [] y){

        int [] temp1 = new int[16];
        int[] temp2 = new int[16];
        Arrays.fill(temp2, val: 1);

        int [] zmx,notx,znx;
        int [] zmy ,noty,zny;
        int [] xplusy, xandy,fxy;

        zmx = Mux.mux16(x,temp1,zx);
        notx = Basic_Gates.Not16(zmx);
        znx = Mux.mux16(zmx,notx,nx);

        zmy = Mux.mux16(y,temp1,zy);
        noty = Basic_Gates.Not16(zmy);
        zny = Mux.mux16(zmy,noty,ny);

        xplusy = Add16.adder(znx,zny);
        xandy = Basic_Gates.and16(znx,zny);
        fxy = Mux.mux16(xandy,xplusy,f);

        int [] nfxy ,abc;
        nfxy = Basic_Gates.Not16(fxy);
        abc = Mux.mux16(fxy,nfxy,no);

        int orabc,zr, ng;
        orabc = Or16way.or16Way(abc);
        zr = Basic_Gates.Not(orabc);

        int [] temp;
        temp = Basic_Gates.and16(temp2, abc);
        ng = temp[15];

        int [] out;
        out = Basic_Gates.or16(abc,temp1);

        return out;
    }
}

```

Registers

A 1-bit register that can hold a 1-bit value. But we need multi-bit registers to hold more bits. So, we have to design 16-bit register. As we already have 1-bit register, we combine 16 of those we get a 16 Bit Register. And register should be a load pin too.

Two Registers used in HACK CPU are A-Register and D-Register.

Hack machine language:

- • 16-bit A-instruction
- • 16-bit C-instruction

Hack program = sequence of instructions written in the Hack machine language.

Program Counter

A program counter is a register in a computer processor that contains the address(location) of the instruction being executed at the current time

The computer must keep track of which instruction should be fetched and executed next.

HACK CPU:

The CPU of the Hack platform is designed to execute 16-bit instructions according to the Hack machine language specification presented in chapter 4. The Hack CPU expects to be connected to two separate memory modules: an instruction memory, from which it fetches instructions for execution, and a data memory, from which it can read, and into which it can write, data values.

Working of CPU (Reference CPU Design Diagram)

We will work our way from T to j1, j2, j3 to see how hack CPU is working

$\sim T | d1$ ($\sim T + T.d1$) = | is a symbol for OR

~ is a symbol for complement

Instruction bit count starts from below and move upwards from 0 to 15 i.e. j1 = 0 and T = 15

Starting from ALU, it has two input bits x and y and 6 control bits (from 6 to 11) (represented by green line) and output is ALU output and status bit of zr and ng depending on the output, x input comes from D-Register and y input comes either from A register or Memory[M].

Bit-12(marked as a) (selection line for MUX 2) specifies whether y input comes from memory or A-Register (Address Register).

Remember, when a==1 (bit-12 is 1) inM is taken, when a==0 address register is taken as y input.

T bit is given as selection line to MUX-1 so to decide whether A-instruction or C-Instruction is going to be used.

When T==0, input to the mux-1 is going to be bit from 0 to 14 which are 15-bit binary constant.

When T==1, input to the mux-1 will be nothing but ALU output

Let us talk about how A-Register is loaded,

The load of A-Register that is ($\sim T + T.d1$), we need high load(must be 1) to work with A-Register, now there are two possible cases i.e. T==0 and T==1, so when T==0 load will be high but when T==1 then in that case d1 = 1 is a necessary condition so that load comes out to be high and this can be confirmed after seeing the table given in C-Instruction slide.

Let us talk about how D-Register is loaded,

The load of D-register will be high when d2 line is high and T must be 1 that is why both T and d2 are passed through an AND gate .

Same can be noticed when the diagram is seen carefully .

When $d3 == 1$ and $T == 1$ using AND GATE, output is given on writeM ($M = RAM[A]$).

Another important component is PC(program counter)

Firstly we must remember the priority order in which PC works i.e. reset , load, inc.

Load is a situation where we have some C-Instruction as for loading to a particular address we need to encounter some jump instruction which is possible only in C-Instruction. Load will only be enabled when there is some kind of jump instruction.

If you don't require to jump to any instruction and want to move ahead one by one we need to remember that $inc = \sim load$ (load is low) meaning that increment is only possible when $load == 0$ or load is low, and increment comes last in priority order of program counter.

NOW for load to be high we will see 4 situation for JUMP i.e

Unconditional jump, Output = 0 ,>0 ,<0

$j3 == 1$ (when output > 0)

$j2 == 1$ (when output = 0)

$j1 == 1$ (when output < 0)

$j1, j2, j3 == 1$ (unconditional jump)

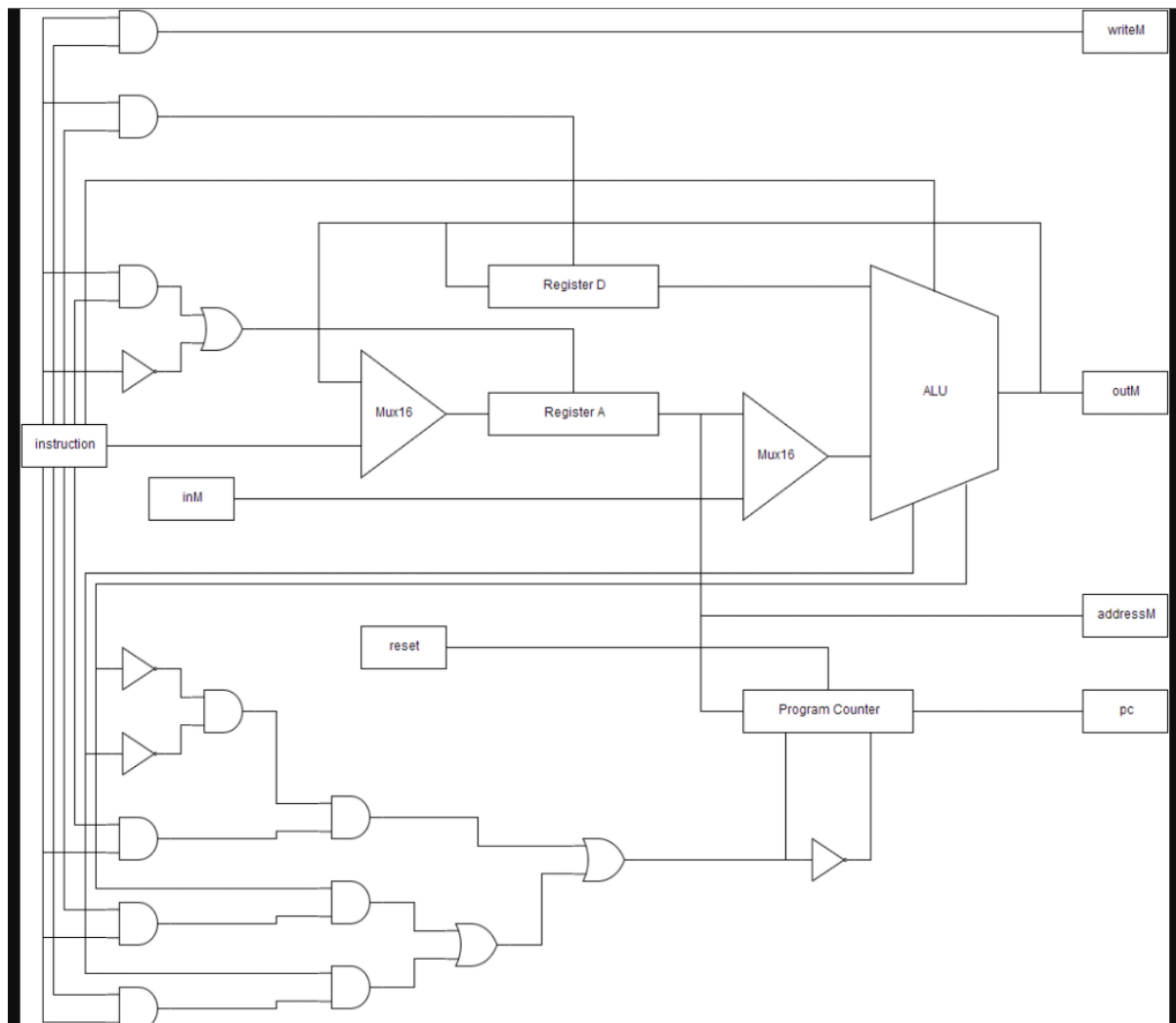
Now when we see load equation carefully we will see that T must be high or $== 1$ and 1 out of any 4 conditions in brackets must be high or $== 1$ to set load as high or $== 1$.

Now for increment there are two possibilities:

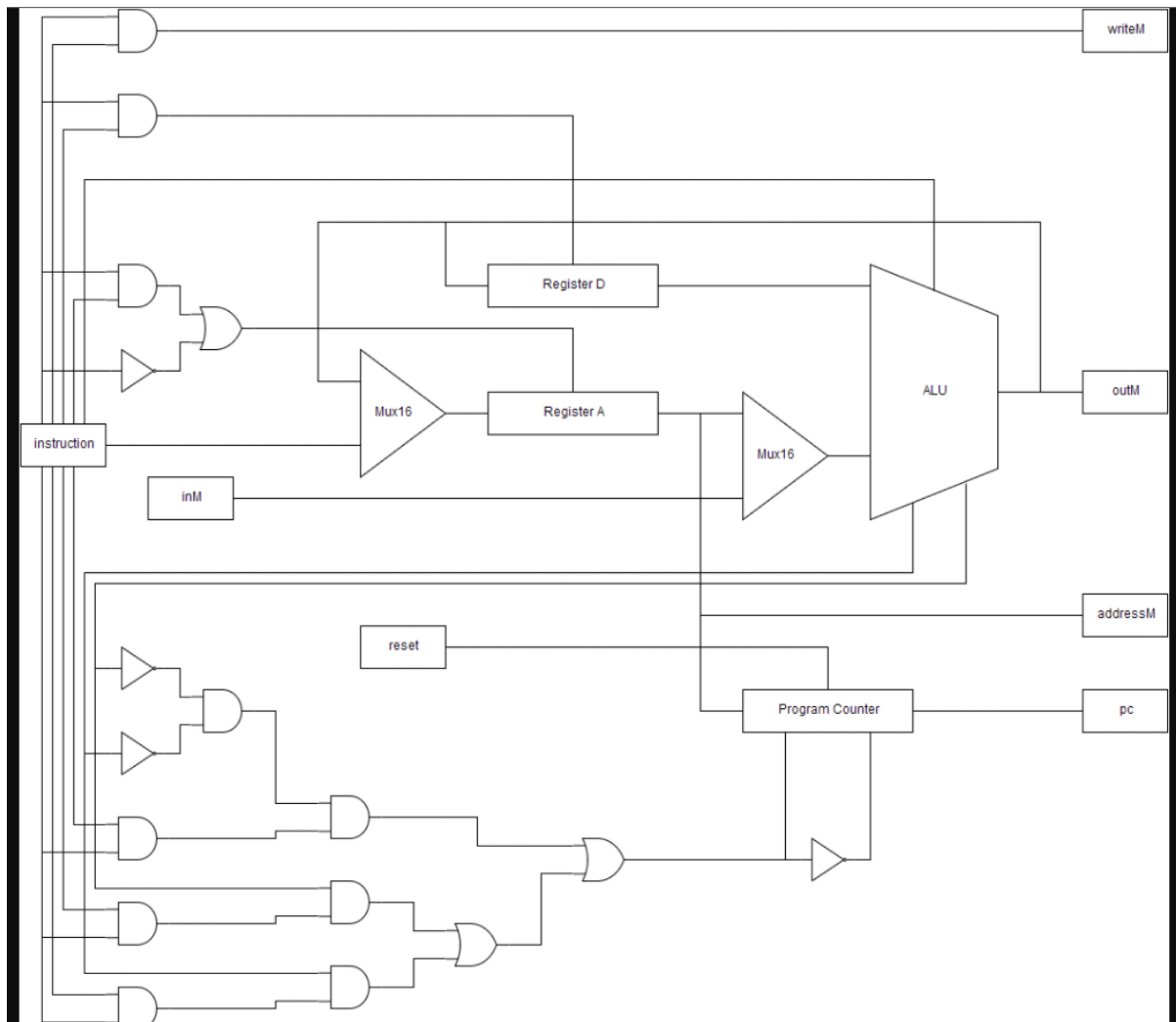
When $T == 0$, it will be an A-Instruction and $load == 0$ which will lead to taking of all 15 bits and it will come in program counter and increment will happen one by one.

When $T == 1$, it will be a C-Instruction but all jump condition are equal to zero which makes $load == 0$ and leads to increment function in program counter.

Logic (Implementation) Diagram for Hack CPU :



Logic (Implementation) Diagram for Hack CPU :



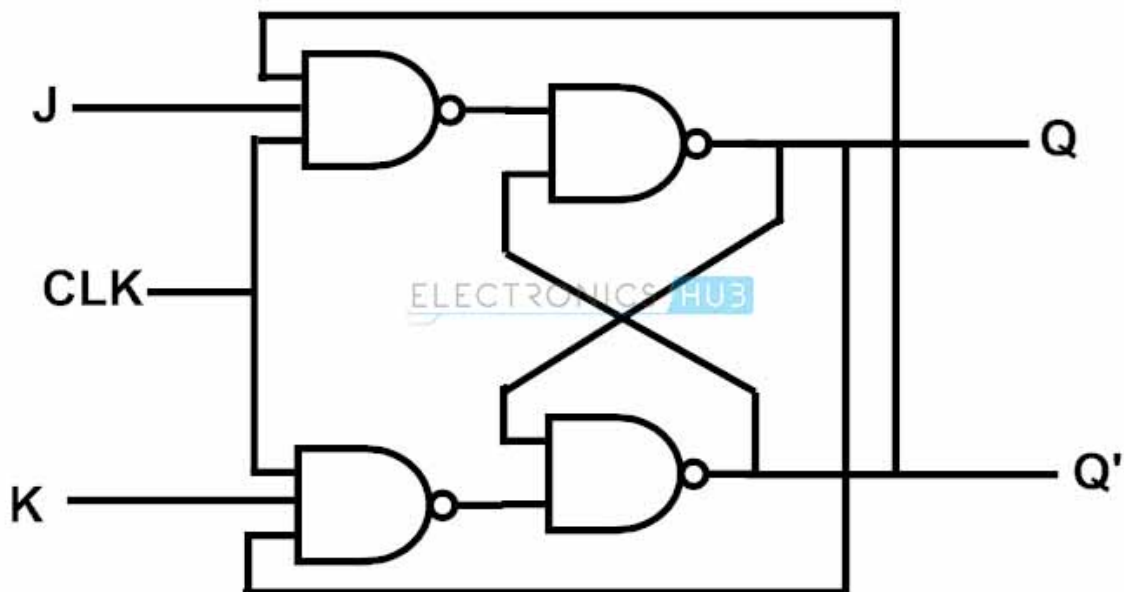
Design and implement a synchronous BCD counter using JK-Flip flop

JK Flip Flop

A JK flip – flop is called a Universal Programmable flip – flop because, using its inputs J, K Preset and Clear, function of any other flip – flop can be imitated.

A JK flip – flop is the modification of SR flip – flop with no illegal state or values.

WORKING



- **Case 1: When both the inputs J and K are LOW, then Q returns its previous state value i.e. it holds the previous data.**

When we apply a clock pulse to the J K flip flop and the J input is low then irrespective of the other NAND gates, the NAND gate-1 output becomes HIGH. In the same manner, if the K input is low then output of NAND gate-2 is also HIGH. So thus, the output remains in the same state i.e., no change in the state of flip flop.

- **Case 2: When J is LOW and K is HIGH, then flip flop will be in Reset state i.e., $Q = 0$, $Q' = 1$.**

When we apply a clock pulse to the J K flip flop and the inputs are J is low and K is high the output of the NAND gate connected to J input becomes 1. Then Q becomes 0. This will reset the flip flop again to its previous state. So, the Flip flop will be in RESET state.

- **Case 3: When J is HIGH and K is LOW, then flip – flop will be in Set state i.e., $Q = 1$, $Q' = 0$**

When we apply a clock pulse to the J K flip flop and the inputs are J is high and K is low the output of the NAND gate connected to K input becomes 1. Then Q' becomes 0. This will set the flip flop with the high clock input. So, the Flip flop will be in SET state.

- **Case 4: When both the inputs J and K are HIGH, then flip – flop is in Toggle state. This means that the output will complement of the previous state.**

K-Map with Characteristic Equation & Gate level implementation

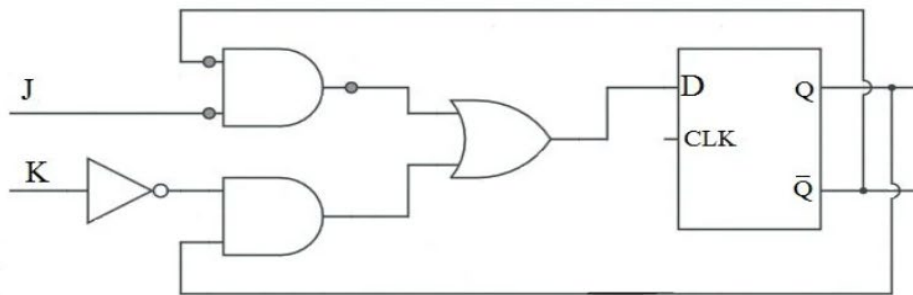
The K-Map for the required input-output relation is:

		KQ _n			
		00	01	11	10
J	0	0	1	0	0
	1	1	1	0	1

$$D = J\bar{Q}_n + \bar{K}Q_n$$

*K-Map Solution for D – JK Flip Flop using
D Flip Flop*

So, a logic diagram can be developed on the basis of these relations as:



JK Flip Flop using D Flip Flop – Logic Diagram

Synchronous BCD Counter

Synchronous Counters are so called because the clock input of all the individual flip-flops within the counter are all clocked together at the same time by an external clock signal.

With the **Synchronous Counter**, the external clock signal is connected to the clock input of EVERY individual flip-flop within the counter so that all of the flip-flops are clocked together simultaneously (in parallel) at the same time giving a fixed time relationship. In other words, changes in the output occur in “synchronization” with the clock signal.

A BCD counter is one of the 4-bit binary counters, which counts from 0 to a pre-determined count with an applied clock signal. When the count reaches the predetermined count value, it resets all the flip-flops and starts to count again from 0. This type of counter is designed by using 4 JK flip flops and counts from 0 to 9, and the result is represented in digital form. After reaching the count of 9 (1001), it resets and starts again.

BCD or decade counter circuit is designed by using JK flip flops and NAND gate. The BCD counter design is very simple, and it requires 4 JK flip flops because it is a 4-bit binary counter.

Working of Synchronous BCD Counter

First, notice that FF0 (Q0) toggles on each clock pulse, so the logic equation for its J0 and K0 inputs is

$$J_0=K_0=1.$$

This equation is implemented by connecting J0 and K0 to a constant HIGH level.

TABLE 9-5				
States of a BCD decade counter.				
Clock Pulse	Q ₃	Q ₂	Q ₁	Q ₀
Initially	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10 (recycles)	0	0	0	0

Next, notice in Table 9-5 that FF1 (Q1) changes on the next clock pulse each time when Q0=1 and Q3=0, so the logic equation for the J1 and K1 inputs is

$$J_1=K_1=Q_0Q_3'$$

$$\begin{array}{lll}
 Q_3 = Q_4 & J_3 = J_4 & K_3 = K_4 \\
 Q_2 = Q_3 & J_2 = J_3 & K_2 = K_3 \\
 Q_1 = Q_2 & J_1 = J_2 & K_1 = K_2 \\
 Q_0 = Q_1 & J_0 = J_1 & K_0 = K_1
 \end{array}$$

State Table and K-Map

TABLE 8.12 State table for Example 8.5

Present State				Next State				Flip Flop Inputs							
Q ₄	Q ₃	Q ₂	Q ₁	Q ₄	Q ₃	Q ₂	Q ₁	J ₄	K ₄	J ₃	K ₃	J ₂	K ₂	J ₁	K ₁
0	0	0	0	0	0	0	1	0	X	0	X	0	X	1	X
0	0	0	1	0	0	1	0	0	X	0	X	1	X	X	1
0	0	1	0	0	0	1	1	0	X	0	X	X	0	1	X
0	0	1	1	0	1	0	0	0	X	1	X	X	1	X	1
0	1	0	0	0	1	0	1	0	X	X	0	0	X	1	X
0	1	0	1	0	1	1	0	0	X	X	0	1	X	X	1
0	1	1	0	0	1	1	1	0	X	X	0	X	0	1	X
0	1	1	1	1	0	0	0	1	X	X	1	X	1	X	1
1	0	0	0	1	0	0	1	X	0	0	X	0	X	1	X
1	0	0	1	0	0	0	0	X	1	0	X	0	X	X	1
1	0	1	0	X	X	X	X	X	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X	X	X	X	X	X

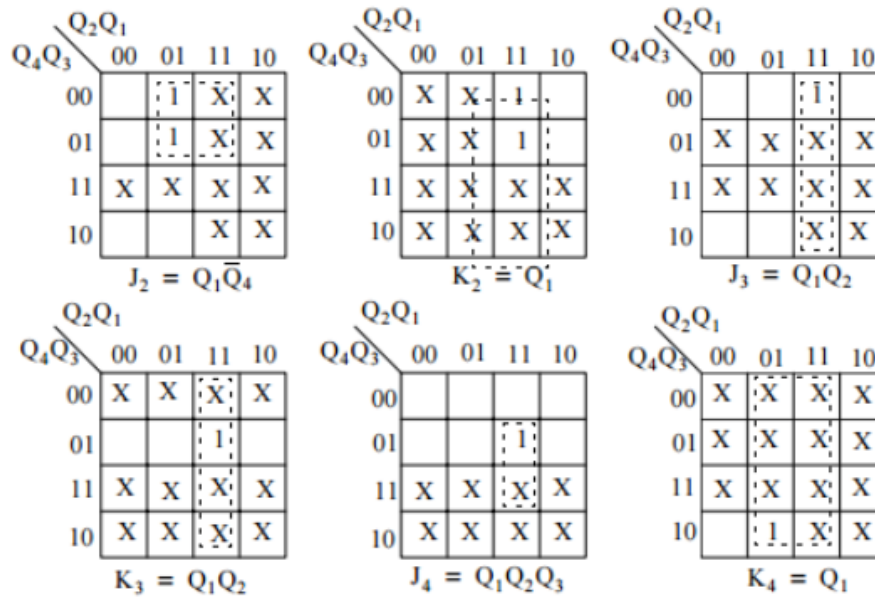


Figure 8.30. K-maps for Table 8.12

We summarize these logic expressions for the flip flop inputs as follows:

$$\begin{aligned}
 J_1 &= K_1 = 1 \\
 J_2 &= Q_1 \bar{Q}_4 \\
 K_2 &= Q_1 \\
 J_3 &= K_3 = Q_1 Q_2 \\
 J_4 &= Q_1 Q_2 Q_3 \\
 K_4 &= 1
 \end{aligned}$$

Logic Diagram for HDL CODE

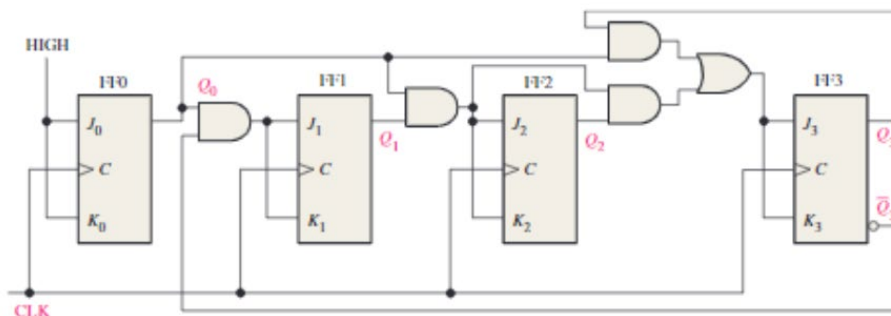



FIGURE 9-18 A synchronous BCD decade counter. Open file F09-18 to verify operation.

Multisim

 JKFF - Notepad

File Edit Format View Help

```
CHIP JKFF {  
  IN J, K;  
  OUT Qt1, nQt1;  
  
  PARTS:  
    Not(in=K, out=notK);  
    And (a=J, b=nQt, out=in1);  
    And (a=notK, b=Qt, out=in2);  
    Or (a=in1, b=in2, out=orout);  
    DFF(in=orout,out=Qt1,out=Qt);  
    Not(in=Qt,out=nQt1,out=nQt);  
}
```

 syncBCD - Notepad

File Edit Format View Help

```
CHIP syncBCD {  
  IN ;  
  OUT q3,q2,q1,q0 ;  
  
  PARTS:  
    JKFF(J=true,K=true,Qt1=q0,Qt1=q01);  
    And(a=q01,b=nq3,out=in1);  
    JKFF(J=in1,K=in1,Qt1=q1,Qt1=q11);  
    And(a=q01,b=q11,out=in2);  
    JKFF(J=in2,K=in2,Qt1=q2,Qt1=q21);  
    And(a=in2,b=q21,out=or1);  
    And(a=q01,b=q31,out=or2);  
    Or(a=or1,b=or2,out=in3);  
    JKFF(J=in3,K=in3,Qt1=q3,nQt1=nq3,Qt1=q31);  
}
```

In JAVA:

```
public class JK {  
  
    public static int jk (int j, int k) {  
        int not_k= NOT.not(k);  
        int and_in1= AND.and(j,NOT.not(dff_q1));  
        int and_in2= AND.and(not_k,dff_q1);  
        int or_orout= OR.or(and_in1,and_in2);  
        int dff_q1= run(or_orout);  
        int out = NOT.not(or_orout);  
        return out;  
    }  
}
```

```
package academy.learnprogramming;  
  
public class Main {  
    public static void main(String[] args) {  
  
    }  
    public static Object BCD(int b) {  
        if(b % 10 == 0){  
            int l= 0;  
            String ll=String.format("%04d",l);  
            return ll;  
        }else if(b==1 || b%10==1){  
            int k= 1;  
            String kk=String.format("%04d",k);  
            return kk;  
        }else if(b==2 || b%10==2){  
            int j=10;  
            String jj=String.format("%04d",j);  
            return jj;  
        }else if(b==3 || b%10==3){  
            int i=11;  
            String ii=String.format("%04d",i);  
            return ii;  
        }else if(b==4 || b%10==4){  
            int h=100;  
            String hh=String.format("%04d",h);  
            return hh;  
        }  
    }  
}
```

```
}else if(b==5 || b%10==5){
    int g=101;
    String gg=String.format("%04d",g);
    return gg;
}else if(b==6 || b%10==6){
    int f=110;
    String ff=String.format("%04d",f);
    return ff;
}else if(b==7 || b%10==7){
    int e=111;
    String ee=String.format("%04d",e);
    return ee;
}else if(b==8 || b%10==8){
    int d=1000;
    return d;
}else if(b==9 || b%10==9){
    int c=1001;
    return c;
}else
    return -1;
}
```


REFERENCES

<https://www.watelectronics.com/bcd-counter-design-operation>

https://www.electronics-tutorials.ws/counter/count_3.html

<https://www.electronicshub.org/jk-flipflop>

<https://electrosome.com/d-flip-flop-conversion-techniques>

<https://www.coursehero.com/file/72960705/DLF-Lab-no-12docx>

Book – Digital Fundamentals Thomas L. Floyd

THANK YOU !