DESIGN AND IMPLEMENTATION OF HACK CPU AND SYNCHRONOUS COUNTER THAT COUNTS DOWN FROM DECIMAL DIGIT 9 TO 0

A PROJECT REPORT

Submitted by

CHAITANYA VARMA	**-**	**CB.SC.U4AIE24017**
HARSHITH REDDY	**-**	**CB.SC.U4AIE24018**
JISHNU TEJA DANDAMUDI	**-**	**CB.SC.U4AIE24019**
JIVITES DAMODAR	**-**	**CB.SC.U4AIE24020**

CENTER FOR COMPUTATIONAL ENGINEERING AND NETWORKING

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

COIMBATORE-641112

2024

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE
AMRITA VISHWA VIDYAPEETHAM
COIMBATORE-641112

DECLARATION

We, **Chaitanya Varma – CB.SC.U4AIE24017**, **Harshith Reddy - CB.SC.U4AIE24018**, **Jishnu Teja Dandamudi - CB.SC.U4AIE24019** and **Jivites Damodar - CB.SC.U4AIE24020**, hereby declare that the project work entitled "**DESIGNING AND IMPLEMENTATION OF HACK CPU AND SYNCHRONOUS COUNTER THAT COUNTS DOWN FROM DECIMAL DIGIT 9 TO 0**" is the record of the original work done by us under the guidance of **Dr. Lekshmi C. R.**, Assistant Professor, Department of Artificial Intelligence, Amrita Vishwa Vidyapeetham, Coimbatore.

Chaitanya Varma - CB.SC.U4AIE24017 **Harshith Reddy - CB.SC.U4AIE24018**

Jishnu Teja Dandamudi - CB.SC.U4AIE24019 **Jivites Damodar - CB.SC.U4AIE24020**

Place: Coimbatore – 641112
Date:

COUNTERSIGNED

Dr. Lekshmi C R,
Assistant Professor,
Department of Artificial Intelligence

TABLE OF CONTENTS

INTRODUCTION

1. **HARDWARE DESCRIPTION LANGUAGE(HDL):**
   Today, hardware designers no longer build anything with their bare hands. Instead, they plan and optimize the chip architecture on a computer workstation, using structured modelling formalisms like Hardware Description Language, or HDL (also known as VHDL, where V stands for Virtual). The designer specifies the chip structure by writing an HDL program, which is then subjected to a rigorous battery of tests. These tests are carried out virtually, using computer simulation: A special software tool, called a hardware simulator, takes the HDL program as input and builds an image of the modelled chip in memory. Next, the designer can instruct the simulator to test the virtual chip on various sets of inputs, generating simulated chip outputs. The outputs can then be compared to the desired results, as mandated by the client who ordered the chip built. In addition to testing the chip's correctness, the hardware designer will typically be interested in a variety of parameters such as speed of computation, energy consumption, and the overall cost implied by the chip design. All these parameters can be simulated and quantified by the hardware simulator, helping the designer optimize the design until the simulated chip delivers desired cost/performance levels. Thus, using HDL, one can completely plan, debug, and optimize the entire chip before a single penny is spent on actual production. When the HDL program is deemed complete, that is, when the performance of the simulated chip satisfies the client who ordered it, the HDL program can become the blueprint from which many copies of the physical chip can be stamped in silicon. This final step in the chip life cycle - from an optimized HDL program to mass production - is typically out-sourced to companies that specialize in chip fabrication, using one switching technology or another.

2. **HARDWARE SIMULATION:**
   Since HDL is a hardware construction language, the process of writing and debugging HDL programs is quite similar to software development. The main difference is that instead of writing code in a language like Java, we write it in HDL, and instead of using a compiler to translate and test the code, we use a hardware simulator. The hardware simulator is a computer program that knows how to parse and interpret HDL code, turn it into an executable representation, and test it according to the specifications of a given test script. There exist many commercial hardware simulators on the market, and these vary greatly in terms of cost, complexity, and ease of use. Together with this book we provide a simple (and free!) hardware simulator that is sufficiently powerful to support sophisticated hardware design projects. In particular, the simulator provides all the necessary tools for building, testing, and integrating all the chips presented in the book, leading to the construction of a general-purpose computer.

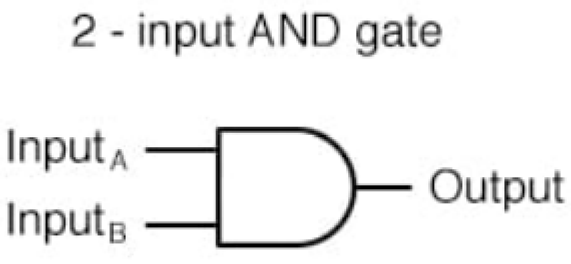
LIST OF GATES USED IN THE OVERALL PROJECT

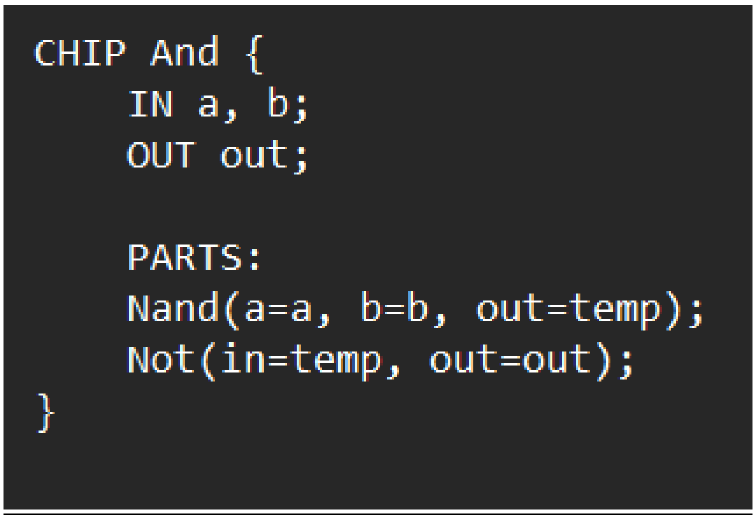
1. And
2. Or
3. Not
4. And16
5. Or16
6. Not16
7. Or8Way
8. Mux
9. Mux16
10. Xor
11. Or16Way
12. Half Adder
13. Full Adder
14. Add16
15. Inc16

CHIPS USED

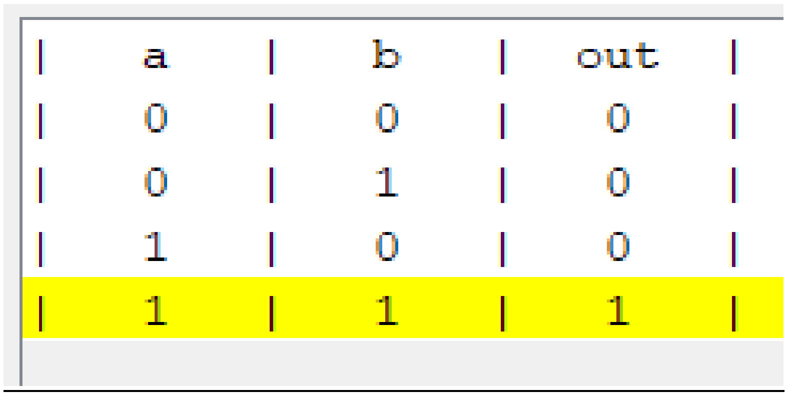
1. AND

The And function returns 1 (true) only when both of the inputs are 1 (true).





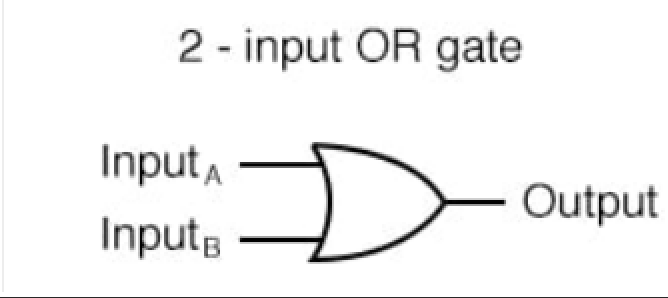
```
CHIP And {
    IN a, b;
    OUT out;

    PARTS:
    Nand(a=a, b=b, out=temp);
    Not(in=temp, out=out);
}
```

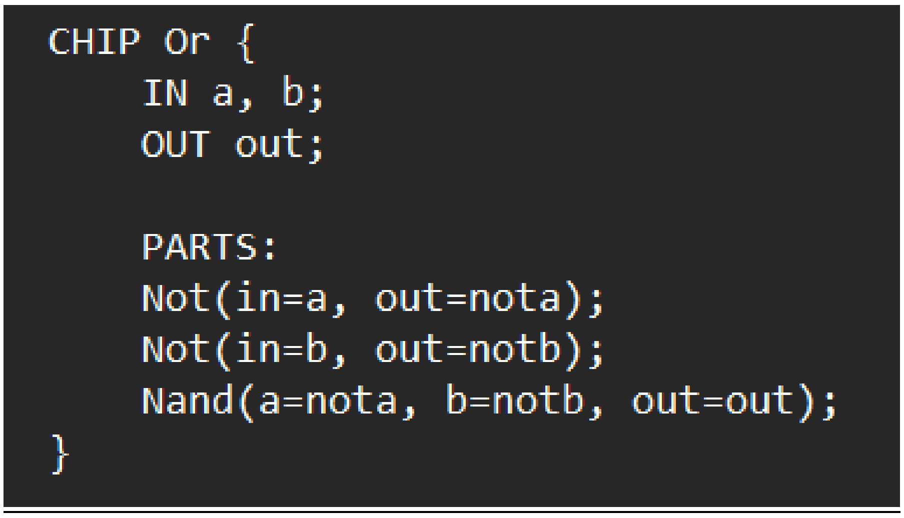


```
|   a   |   b   |  out  |
|   0   |   0   |   0   |
|   0   |   1   |   0   |
|   1   |   0   |   0   |
|   1   |   1   |   1   |
```

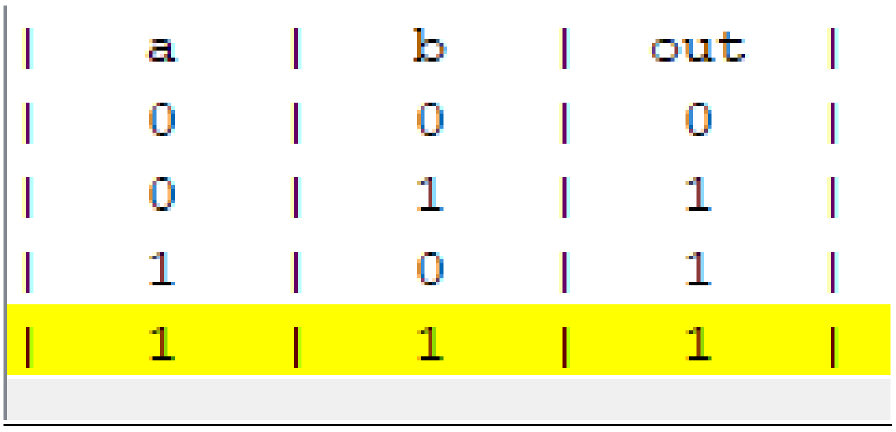
2. OR

The Or function returns 1 (true) when either of the inputs are 1 (true).





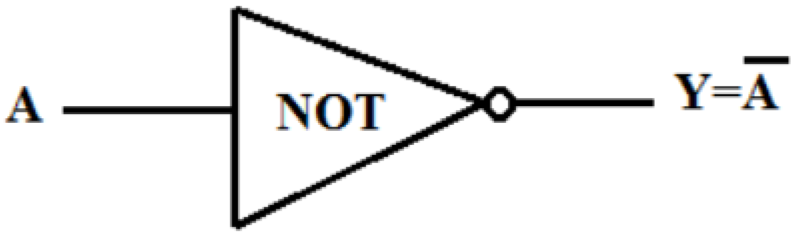
```
CHIP Or {
    IN a, b;
    OUT out;

    PARTS:
    Not(in=a, out=nota);
    Not(in=b, out=notb);
    Nand(a=nota, b=notb, out=out);
}
```

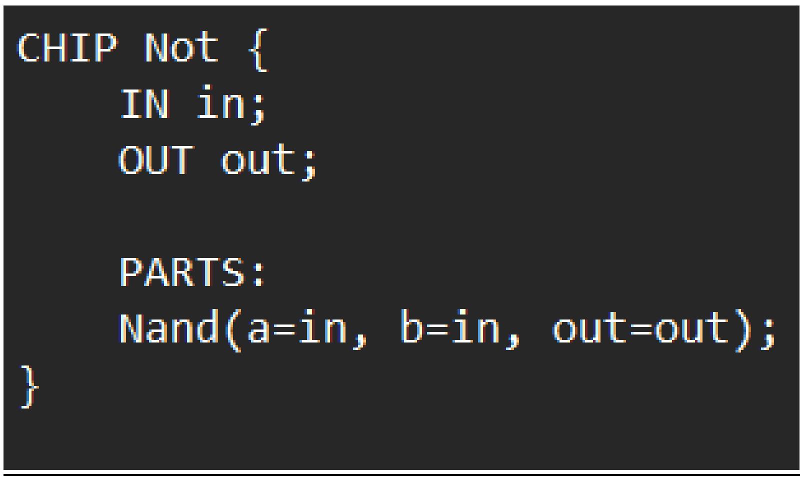


```
|   a   |   b   |  out  |
|   0   |   0   |   0   |
|   0   |   1   |   1   |
|   1   |   0   |   1   |
|   1   |   1   |   1   |
```

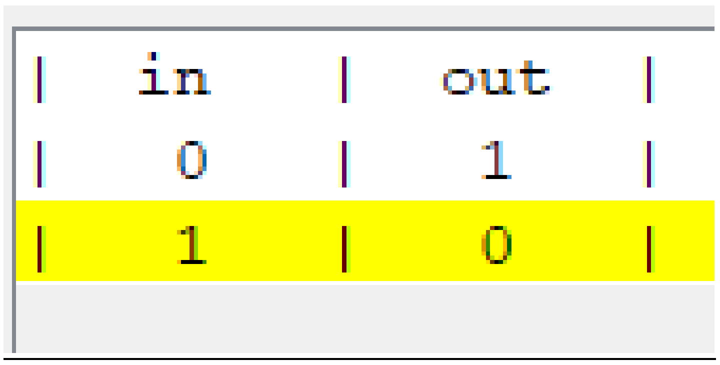
3. NOT

The single-input Not Gate, also referred as Inverter, inverts the value of input.





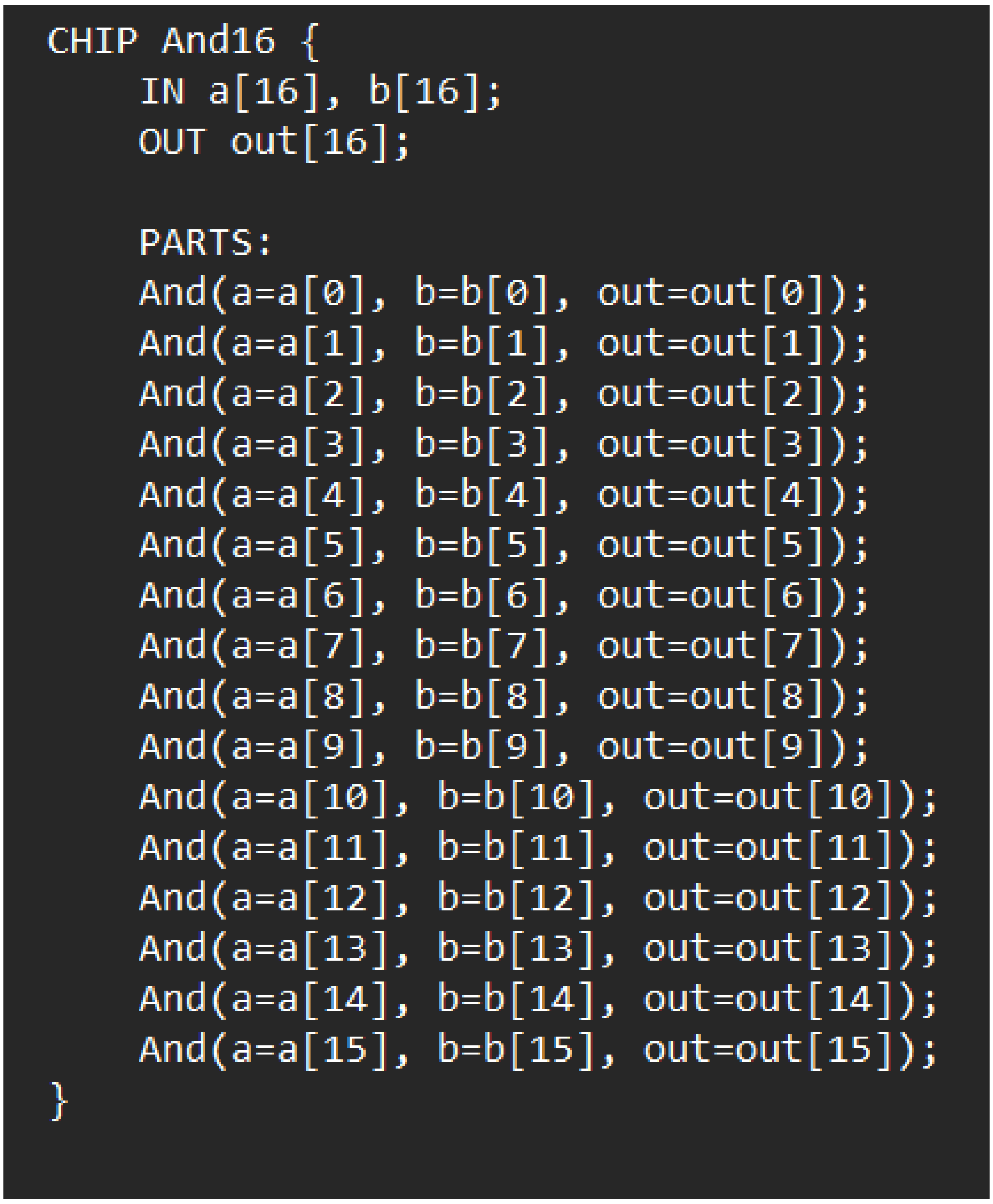
```
CHIP Not {
    IN in;
    OUT out;

    PARTS:
    Nand(a=in, b=in, out=out);
}
```



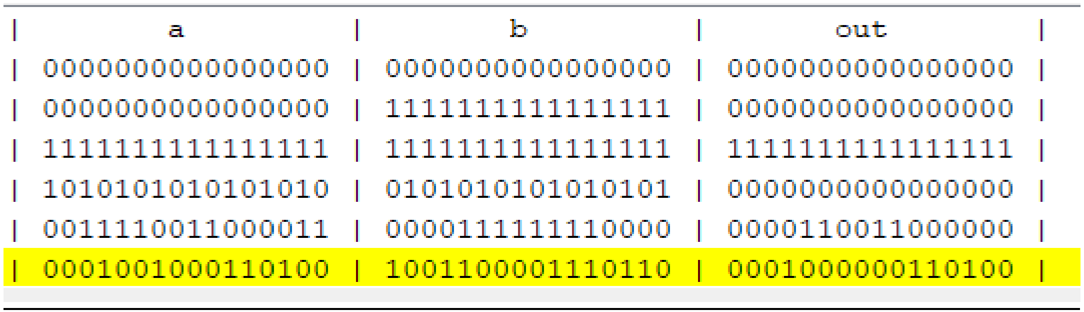
```
|   in    |   out   |
|    0    |    1    |
|    1    |    0    |
```

4. <u>AND16</u>

An 16-bit And Chip applies the Boolean operation And to every one of the bits in its 16-bit input bus.



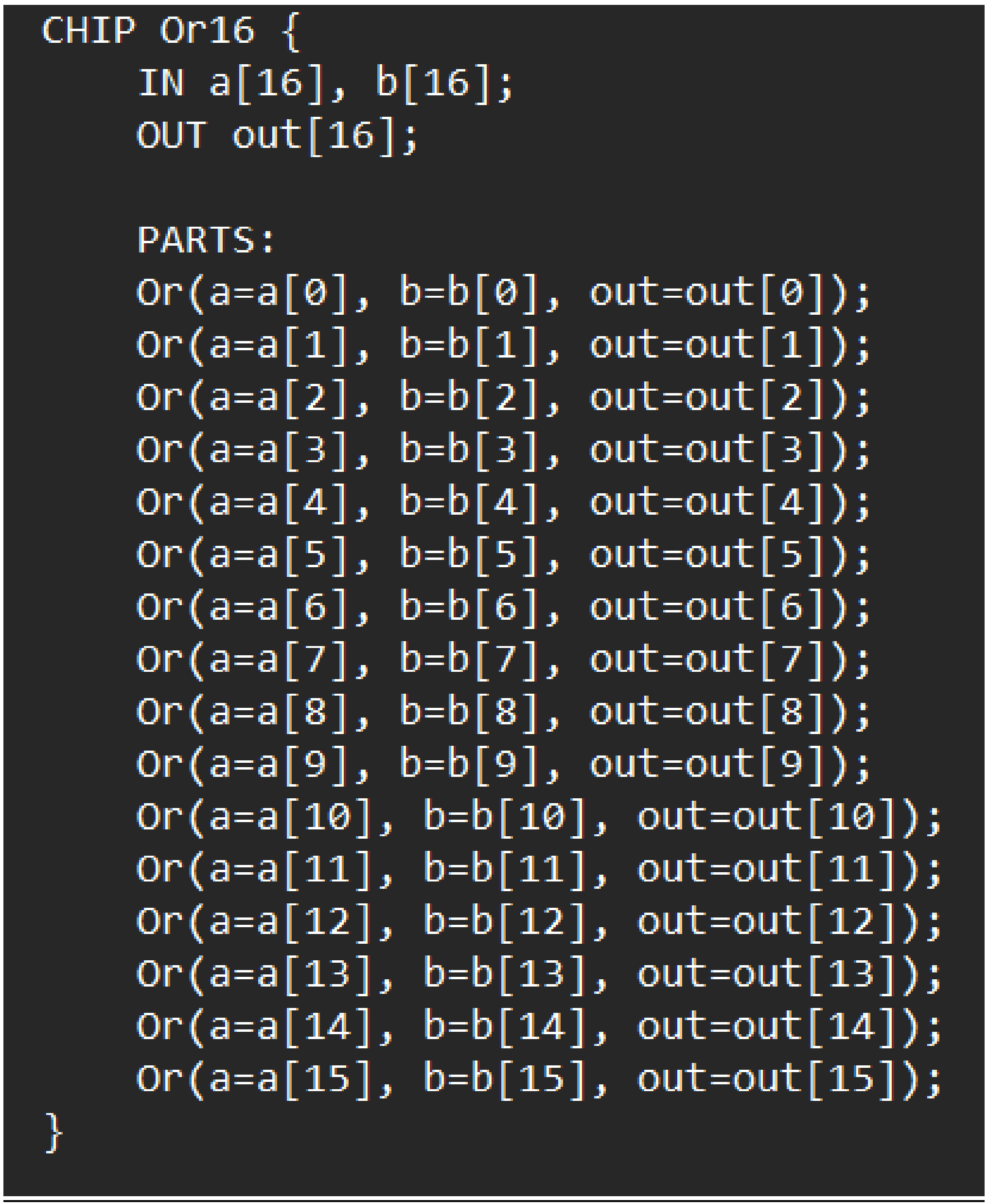
```
CHIP And16 {
    IN a[16], b[16];
    OUT out[16];

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



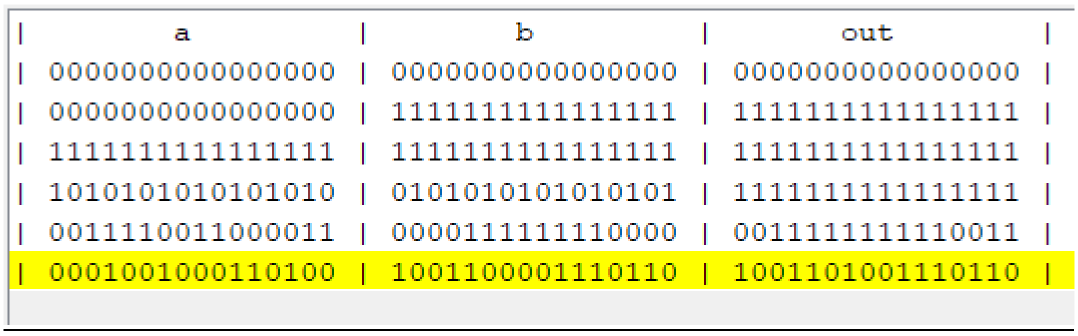
```
|        a         |        b         |       out        |
| 0000000000000000 | 0000000000000000 | 0000000000000000 |
| 0000000000000000 | 1111111111111111 | 0000000000000000 |
| 1111111111111111 | 1111111111111111 | 1111111111111111 |
| 1010101010101010 | 0101010101010101 | 0000000000000000 |
| 0011110011000011 | 0000111111110000 | 0000110011000000 |
| 0001001000110100 | 1001100001110110 | 0001000000110100 |
```

5. OR16

An 16-bit Or Chip applies the Boolean operation Or to every one of the bits in its 16-bit input bus.



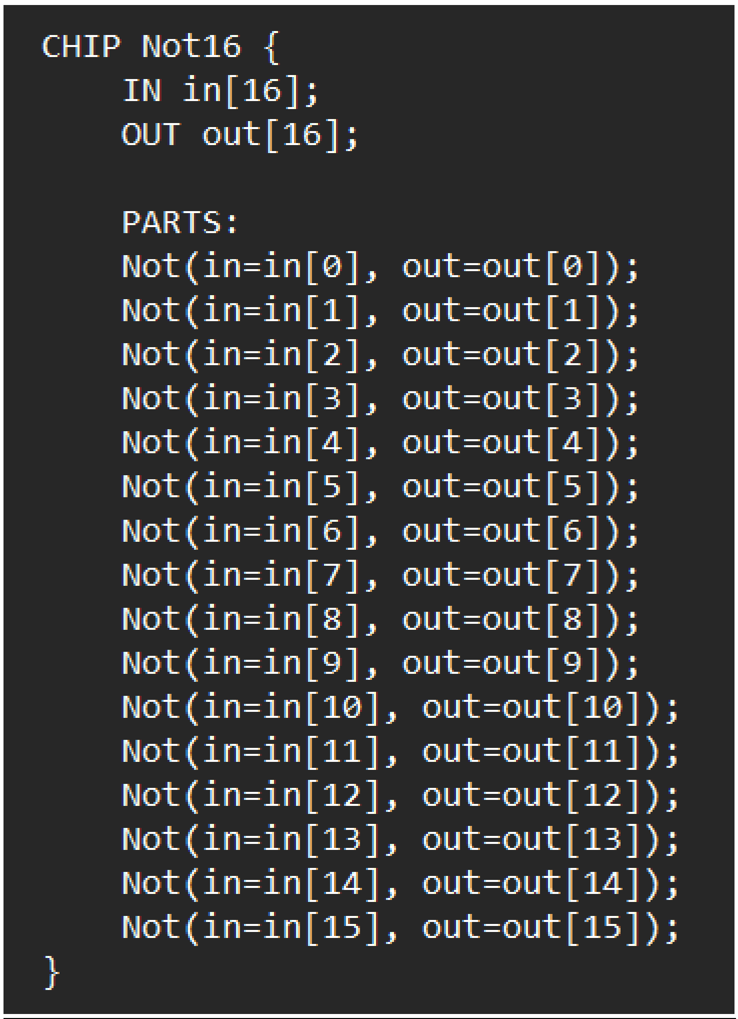
```
CHIP Or16 {
    IN a[16], b[16];
    OUT out[16];

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



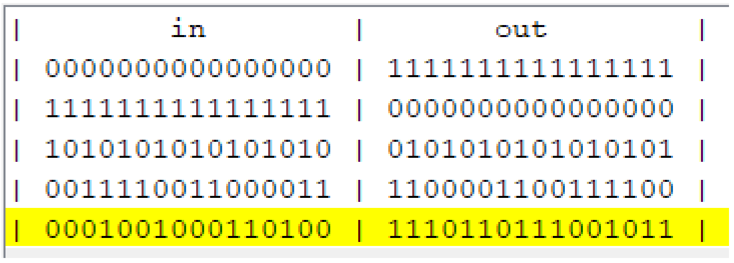
```
|        a         |        b         |       out        |
| 0000000000000000 | 0000000000000000 | 0000000000000000 |
| 0000000000000000 | 1111111111111111 | 1111111111111111 |
| 1111111111111111 | 1111111111111111 | 1111111111111111 |
| 1010101010101010 | 0101010101010101 | 1111111111111111 |
| 0011110011000011 | 0000111111110000 | 0011111111110011 |
| 0001001000110100 | 1001100001110110 | 1001101001110110 |
```

6. NOT16

Unlike Not gate, Not16 takes 16-bit input, inverts each bit and produces output.
In other words, An 16-bit Not Chip applies the Boolean operation Not to every one of the bits in its 16-bit input bus.



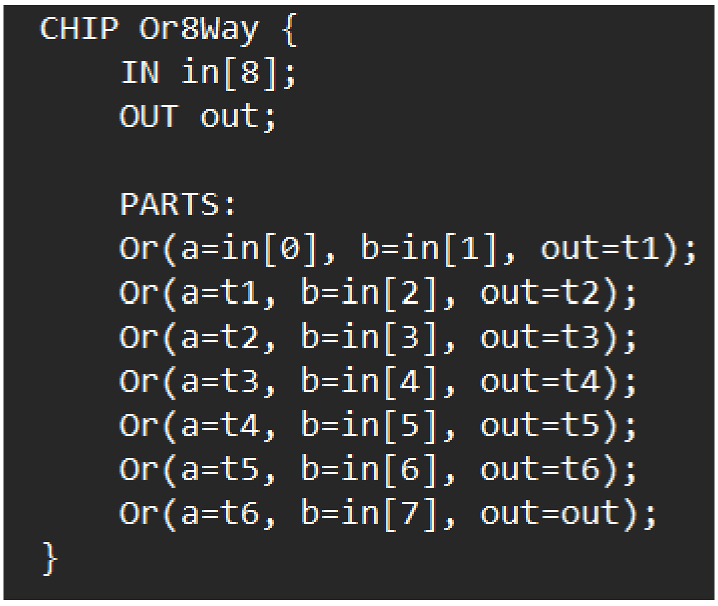
```
CHIP Not16 {
    IN in[16];
    OUT out[16];

    PARTS:
    Not(in=in[0], out=out[0]);
    Not(in=in[1], out=out[1]);
    Not(in=in[2], out=out[2]);
    Not(in=in[3], out=out[3]);
    Not(in=in[4], out=out[4]);
    Not(in=in[5], out=out[5]);
    Not(in=in[6], out=out[6]);
    Not(in=in[7], out=out[7]);
    Not(in=in[8], out=out[8]);
    Not(in=in[9], out=out[9]);
    Not(in=in[10], out=out[10]);
    Not(in=in[11], out=out[11]);
    Not(in=in[12], out=out[12]);
    Not(in=in[13], out=out[13]);
    Not(in=in[14], out=out[14]);
    Not(in=in[15], out=out[15]);
}
```



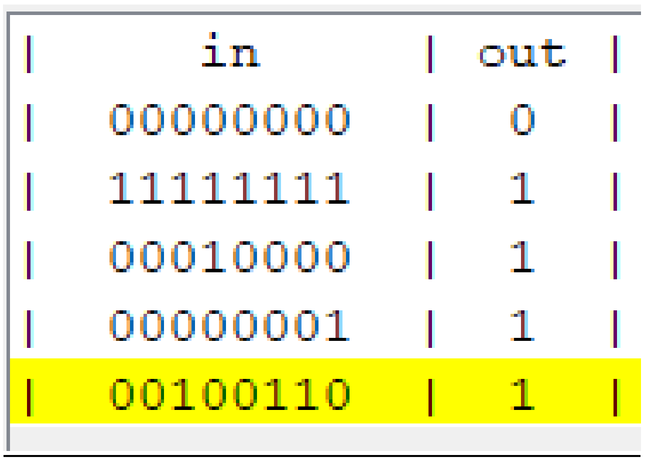
```
|        in        |       out        |
| 0000000000000000 | 1111111111111111 |
| 1111111111111111 | 0000000000000000 |
| 1010101010101010 | 0101010101010101 |
| 0011110011000011 | 1100001100111100 |
| 0001001000110100 | 1110110111001011 |
```

7. OR8WAY

An 8-way or gate outputs 1 when at least one bit of its 16-bit input is 1.
Unlike n-bit input logic gates, n-way logic gates use the same output iteratively over the Boolean operation, which means, it uses the previous output as input for the next similar Boolean operation.



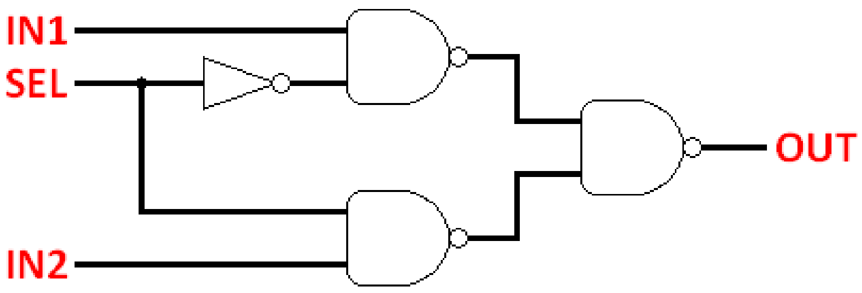
```
CHIP Or8Way {
    IN in[8];
    OUT out;

    PARTS:
    Or(a=in[0], b=in[1], out=t1);
    Or(a=t1, b=in[2], out=t2);
    Or(a=t2, b=in[3], out=t3);
    Or(a=t3, b=in[4], out=t4);
    Or(a=t4, b=in[5], out=t5);
    Or(a=t5, b=in[6], out=t6);
    Or(a=t6, b=in[7], out=out);
}
```

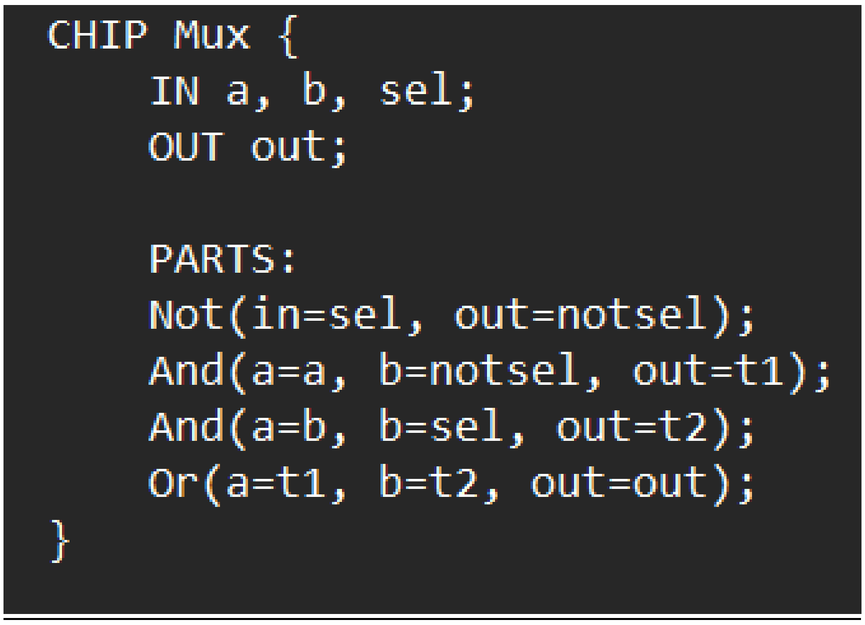


```
|     in     | out |
|  00000000  |  0  |
|  11111111  |  1  |
|  00010000  |  1  |
|  00000001  |  1  |
|  00100110  |  1  |
```

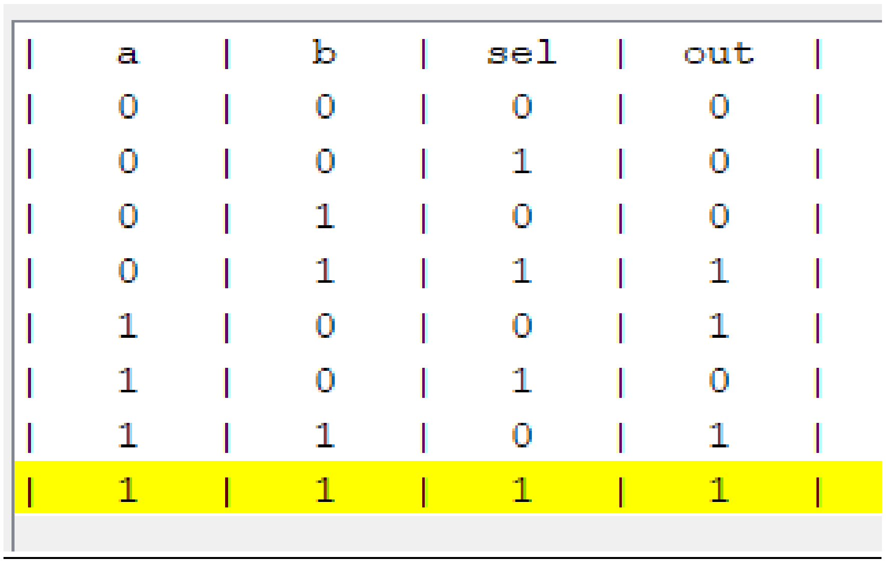
8. MUX

A Multiplexor, also known as Selector, is a three-input bit gate that uses one of the inputs called "selection bit" to select one of the given two inputs for outputting them.
The name multiplexor was adopted from communications systems, where similar devices are used to serialize (multiplex) several input signals over a single output wire.





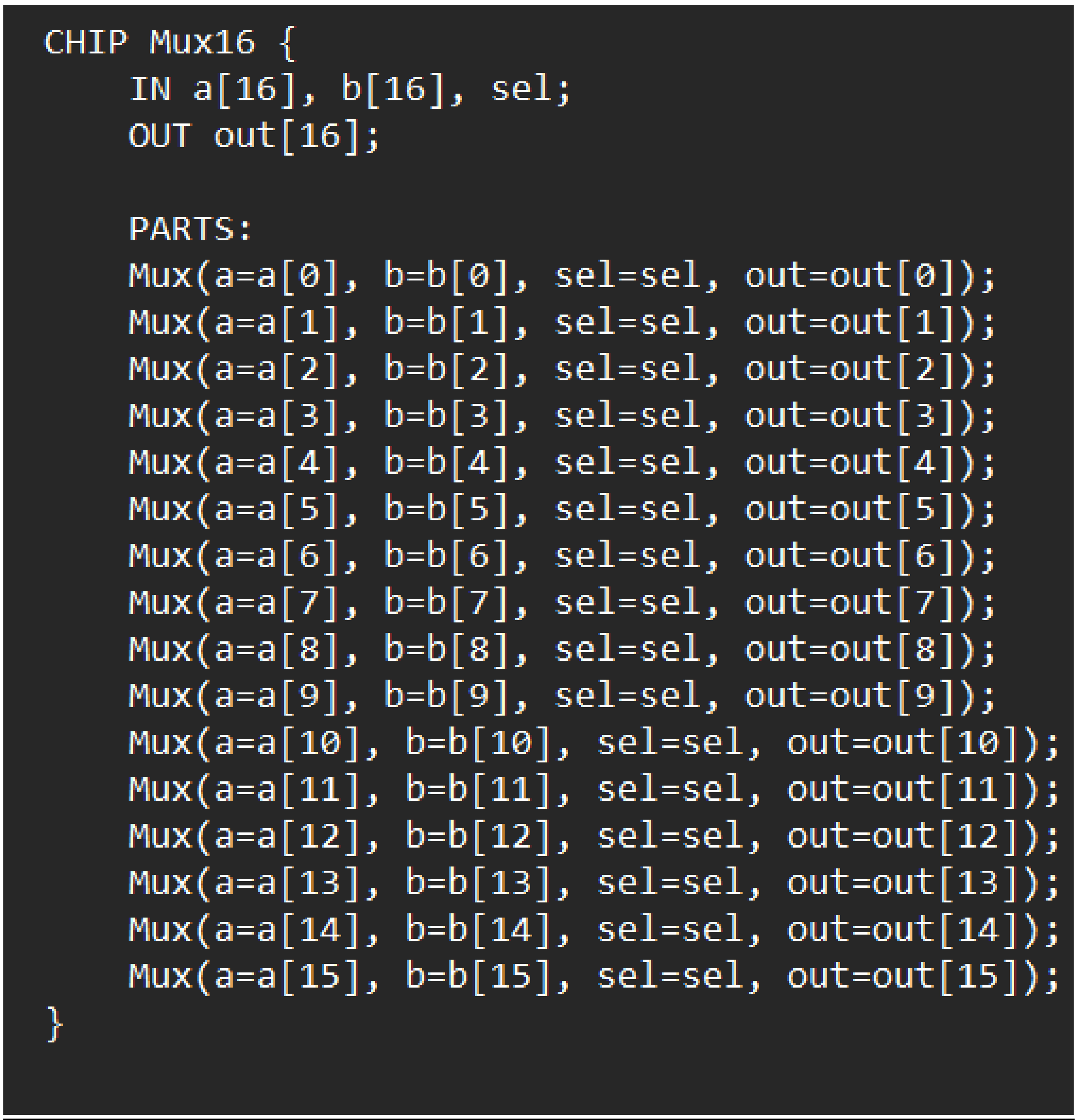
```
CHIP Mux {
    IN a, b, sel;
    OUT out;

    PARTS:
    Not(in=sel, out=notsel);
    And(a=a, b=notsel, out=t1);
    And(a=b, b=sel, out=t2);
    Or(a=t1, b=t2, out=out);
}
```



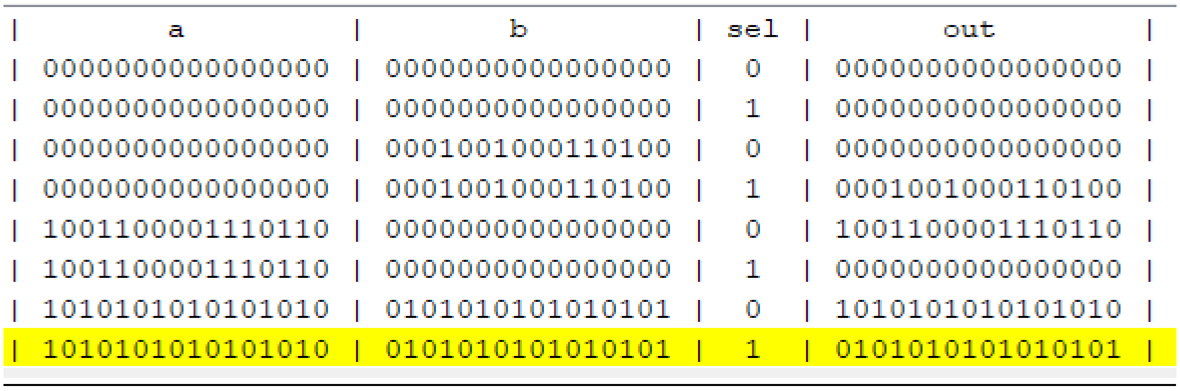
```
|   a   |   b   |  sel  |  out  |
|   0   |   0   |   0   |   0   |
|   0   |   0   |   1   |   0   |
|   0   |   1   |   0   |   0   |
|   0   |   1   |   1   |   1   |
|   1   |   0   |   0   |   1   |
|   1   |   0   |   1   |   0   |
|   1   |   1   |   0   |   1   |
|   1   |   1   |   1   |   1   |
```

9. MUX16

An 16-bit multiplexor is almost similar to the Multiplexor Chip described previously, except that both inputs are each 16-bit wide, while the selector is a single bit.



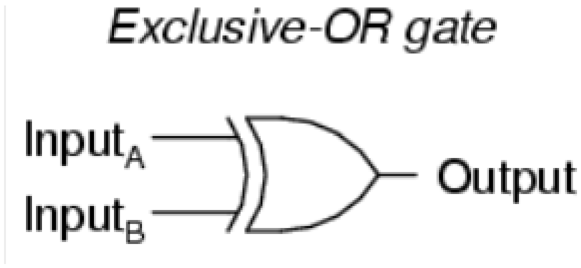
```
CHIP Mux16 {
    IN a[16], b[16], sel;
    OUT out[16];

    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]);
}
```

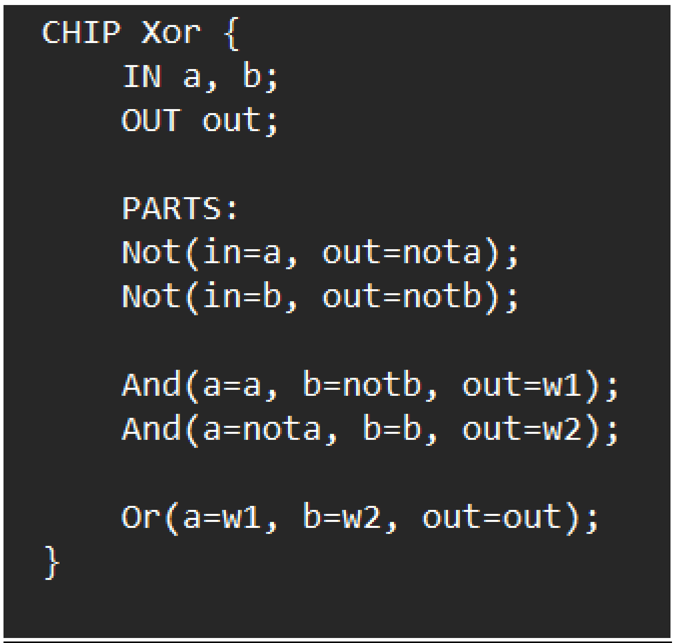


```
|        a         |        b         | sel |       out        |
| 0000000000000000 | 0000000000000000 |  0  | 0000000000000000 |
| 0000000000000000 | 0000000000000000 |  1  | 0000000000000000 |
| 0000000000000000 | 0001001000110100 |  0  | 0000000000000000 |
| 0000000000000000 | 0001001000110100 |  1  | 0001001000110100 |
| 1001100001110110 | 0000000000000000 |  0  | 1001100001110110 |
| 1001100001110110 | 0000000000000000 |  1  | 0000000000000000 |
| 1010101010101010 | 0101010101010101 |  0  | 1010101010101010 |
| 1010101010101010 | 0101010101010101 |  1  | 0101010101010101 |
```

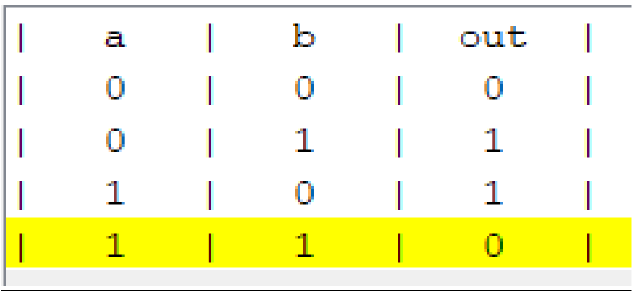
10. XOR

The Xor function, also known as Exclusive-OR, returns 1 (true) when either of the inputs are same (both are 0 or both are 1 simultaneously).



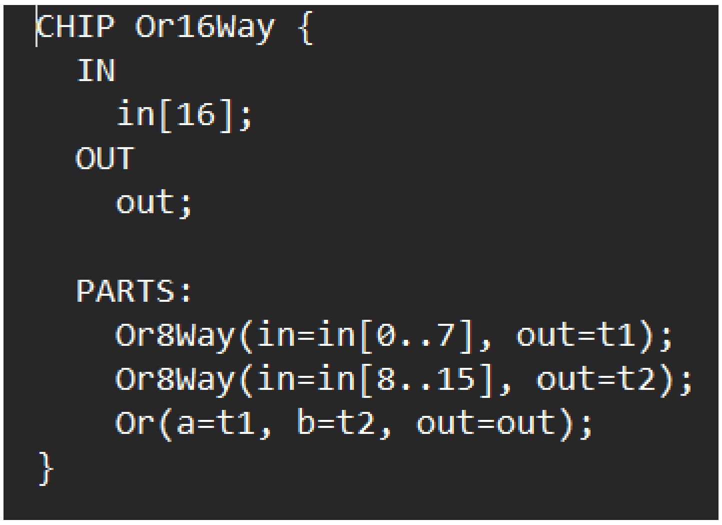


```
CHIP Xor {
    IN a, b;
    OUT out;

    PARTS:
    Not(in=a, out=nota);
    Not(in=b, out=notb);

    And(a=a, b=notb, out=w1);
    And(a=nota, b=b, out=w2);

    Or(a=w1, b=w2, out=out);
}
```

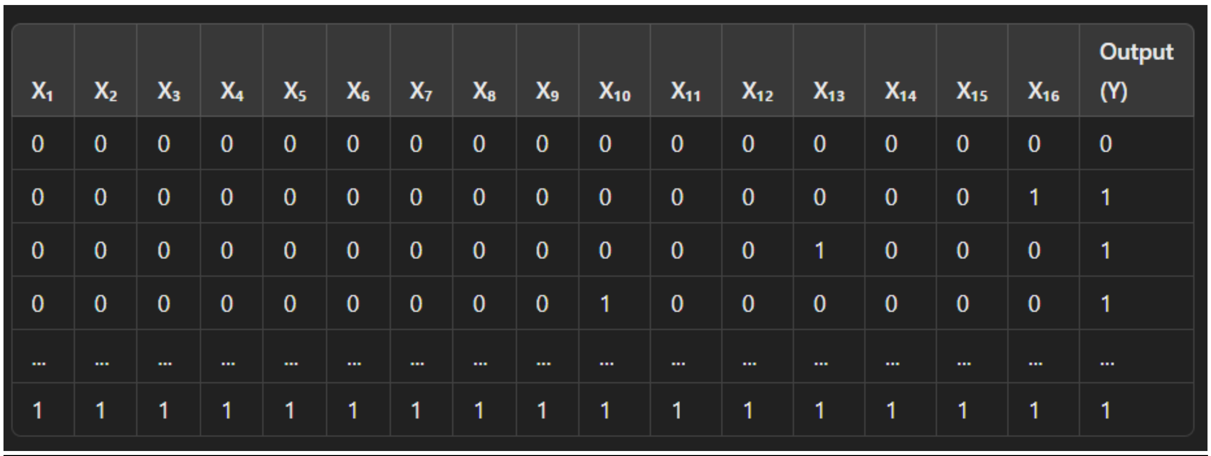


```
|   a   |   b   |  out  |
|   0   |   0   |   0   |
|   0   |   1   |   1   |
|   1   |   0   |   1   |
|   1   |   1   |   0   |
```

11. OR16WAY



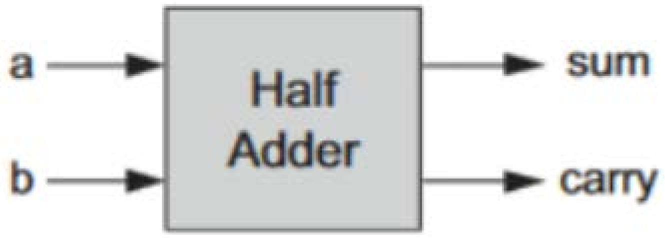
```
CHIP Or16Way {
   IN
     in[16];
   OUT
     out;

   PARTS:
     Or8Way(in=in[0..7], out=t1);
     Or8Way(in=in[8..15], out=t2);
     Or(a=t1, b=t2, out=out);
}
```

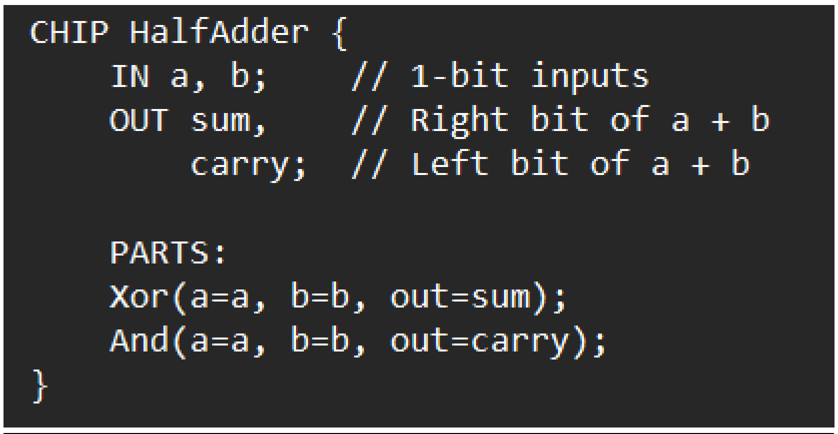


X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}	X_{12}	X_{13}	X_{14}	X_{15}	X_{16}	Output (Y)
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

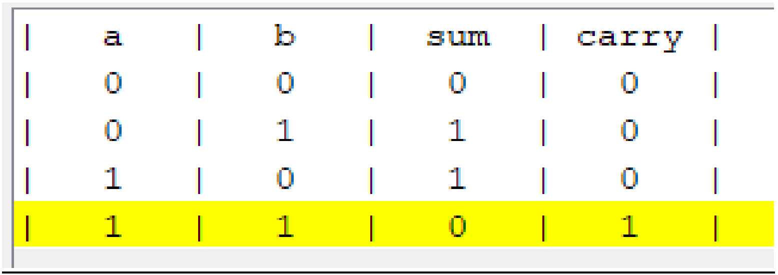
12. HALF ADDER

The chip used to add two n-bit numbers is known as Adder, also known as n-bit Adder. Half Adder chip is used to add 2-bits.





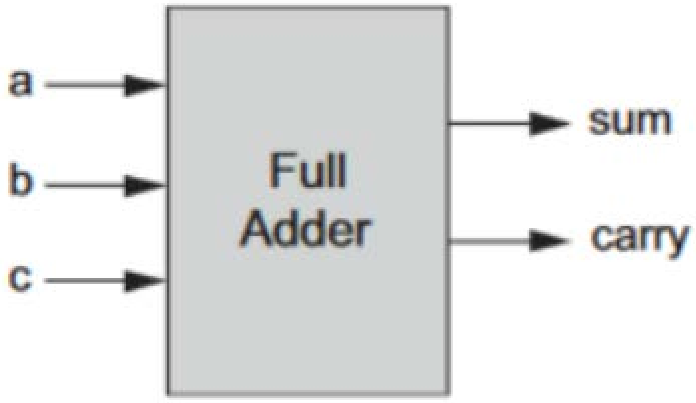
```
CHIP HalfAdder {
    IN a, b;     // 1-bit inputs
    OUT sum,     // Right bit of a + b
        carry;   // Left bit of a + b

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

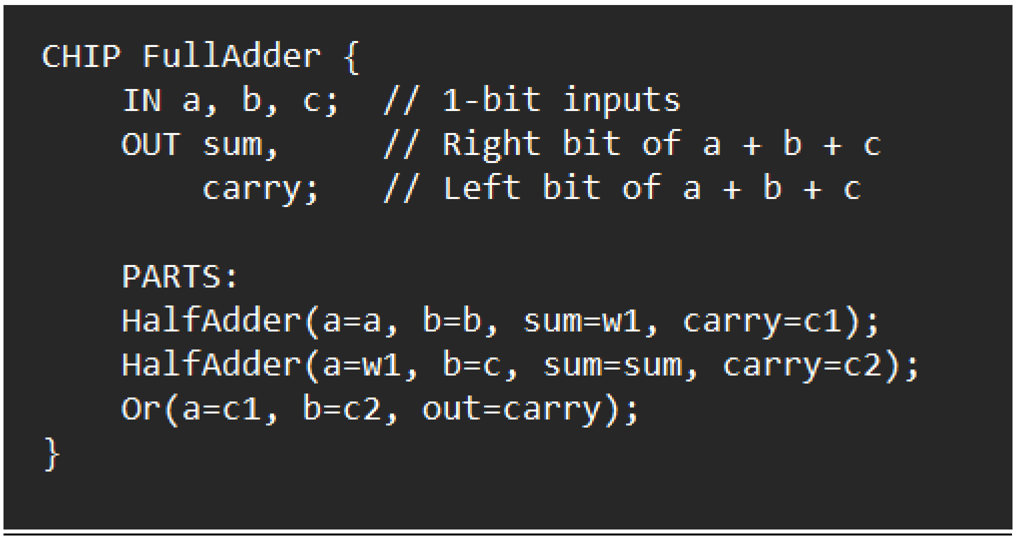


a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

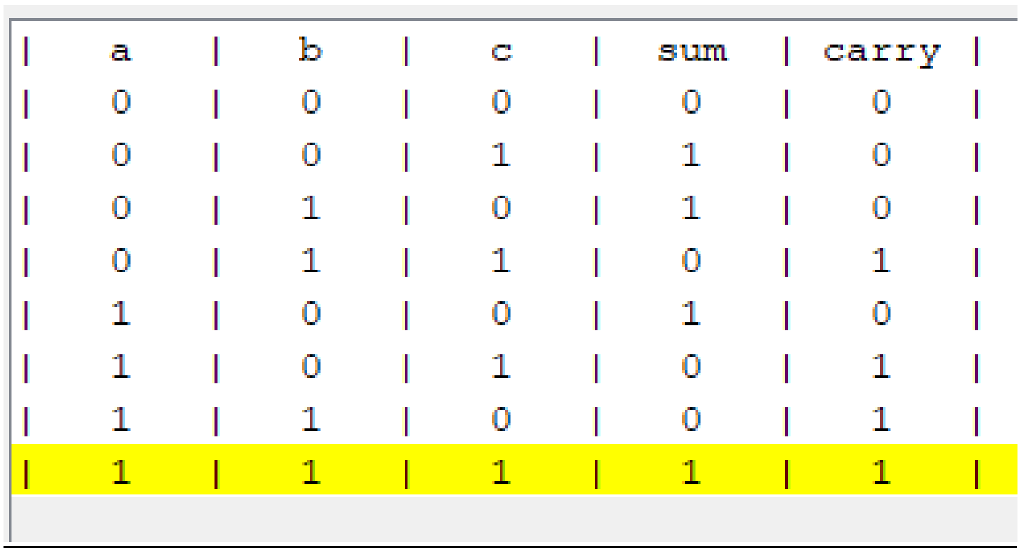
13. FULL ADDER

Full Adder chip is used to add 3-bits.





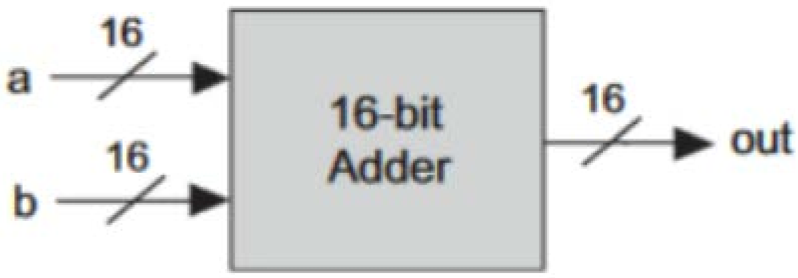
```
CHIP FullAdder {
    IN a, b, c;  // 1-bit inputs
    OUT sum,     // Right bit of a + b + c
        carry;   // Left bit of a + b + c

    PARTS:
    HalfAdder(a=a, b=b, sum=w1, carry=c1);
    HalfAdder(a=w1, b=c, sum=sum, carry=c2);
    Or(a=c1, b=c2, out=carry);
}
```

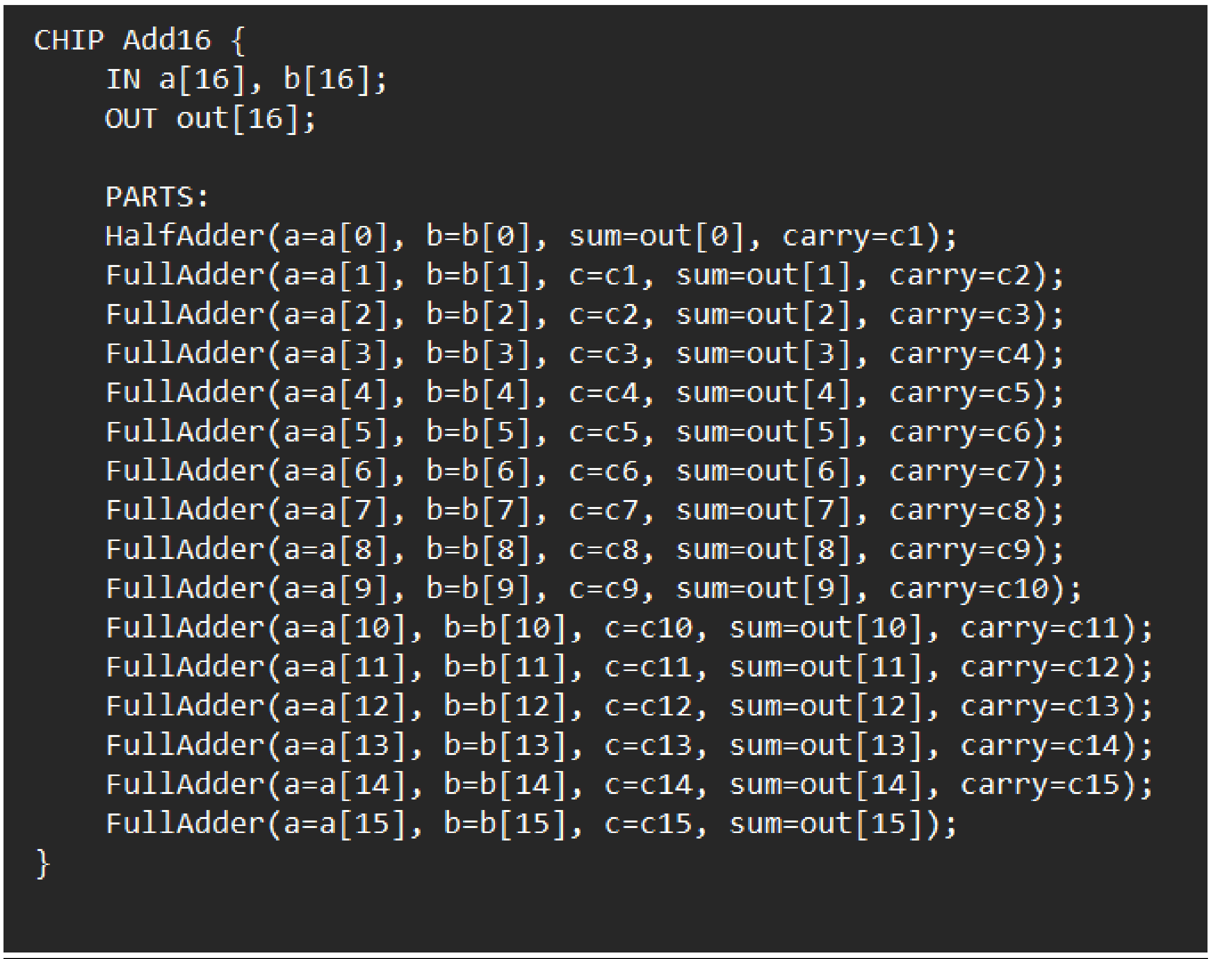


```
|   a   |   b   |   c   |  sum  | carry |
|   0   |   0   |   0   |   0   |   0   |
|   0   |   0   |   1   |   1   |   0   |
|   0   |   1   |   0   |   1   |   0   |
|   0   |   1   |   1   |   0   |   1   |
|   1   |   0   |   0   |   1   |   0   |
|   1   |   0   |   1   |   0   |   1   |
|   1   |   1   |   0   |   0   |   1   |
|   1   |   1   |   1   |   1   |   1   |
```

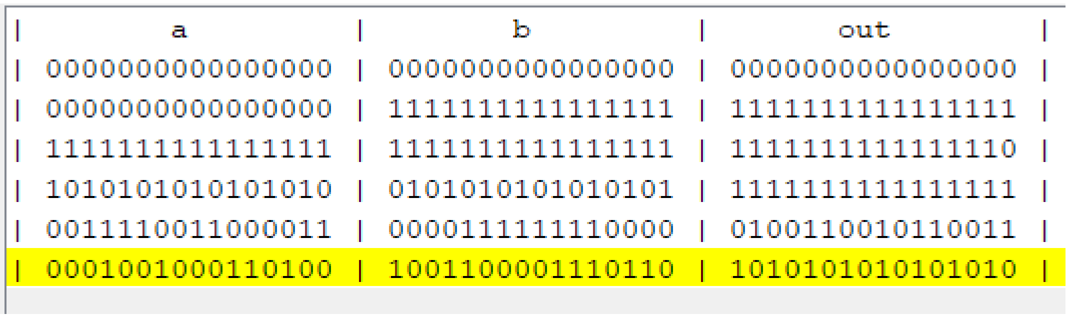
14. ADD16

16-bit Adder chip is used to add two 16-bit numbers.





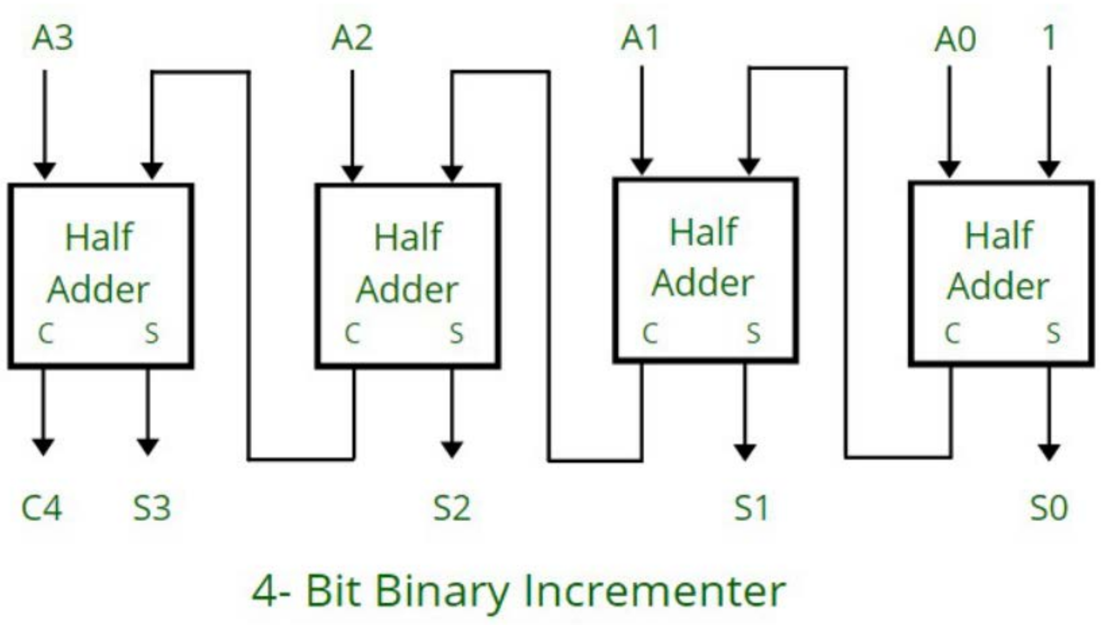
```
CHIP Add16 {
    IN a[16], b[16];
    OUT out[16];

    PARTS:
    HalfAdder(a=a[0], b=b[0], sum=out[0], carry=c1);
    FullAdder(a=a[1], b=b[1], c=c1, sum=out[1], carry=c2);
    FullAdder(a=a[2], b=b[2], c=c2, sum=out[2], carry=c3);
    FullAdder(a=a[3], b=b[3], c=c3, sum=out[3], carry=c4);
    FullAdder(a=a[4], b=b[4], c=c4, sum=out[4], carry=c5);
    FullAdder(a=a[5], b=b[5], c=c5, sum=out[5], carry=c6);
    FullAdder(a=a[6], b=b[6], c=c6, sum=out[6], carry=c7);
    FullAdder(a=a[7], b=b[7], c=c7, sum=out[7], carry=c8);
    FullAdder(a=a[8], b=b[8], c=c8, sum=out[8], carry=c9);
    FullAdder(a=a[9], b=b[9], c=c9, sum=out[9], carry=c10);
    FullAdder(a=a[10], b=b[10], c=c10, sum=out[10], carry=c11);
    FullAdder(a=a[11], b=b[11], c=c11, sum=out[11], carry=c12);
    FullAdder(a=a[12], b=b[12], c=c12, sum=out[12], carry=c13);
    FullAdder(a=a[13], b=b[13], c=c13, sum=out[13], carry=c14);
    FullAdder(a=a[14], b=b[14], c=c14, sum=out[14], carry=c15);
    FullAdder(a=a[15], b=b[15], c=c15, sum=out[15]);
}
```



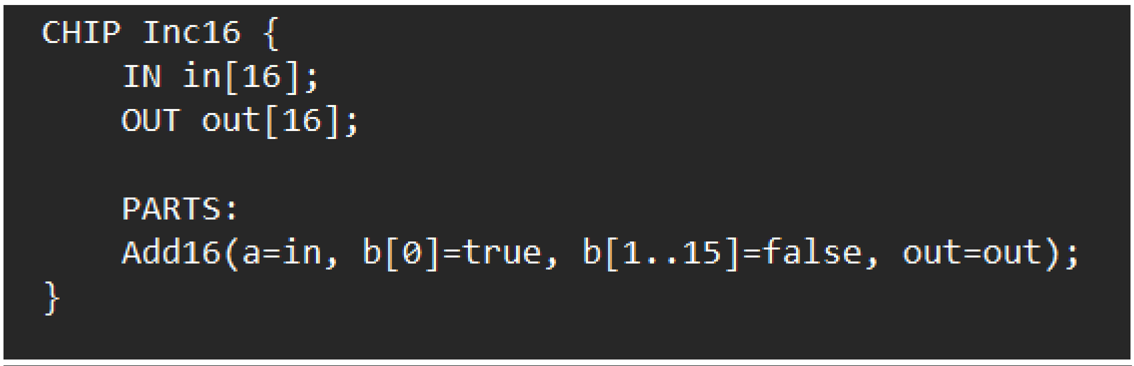
```
|        a         |        b         |       out        |
| 0000000000000000 | 0000000000000000 | 0000000000000000 |
| 0000000000000000 | 1111111111111111 | 1111111111111111 |
| 1111111111111111 | 1111111111111111 | 1111111111111110 |
| 1010101010101010 | 0101010101010101 | 1111111111111111 |
| 0011110011000011 | 0000111111110000 | 0100110010110011 |
| 0001001000110100 | 1001100001110110 | 1010101010101010 |
```

15. INC16

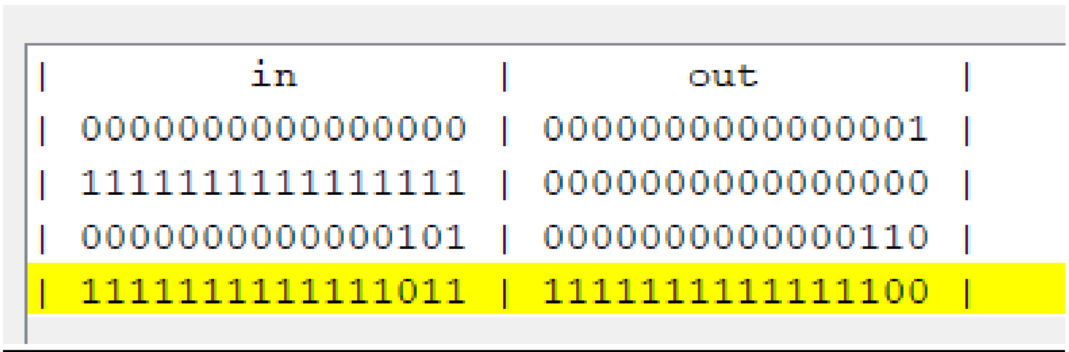
16-bit Incrementor chip is a special kind of Adder, used to increment a 16-bit input by 1.



4- Bit Binary Incrementer



```
CHIP Inc16 {
    IN in[16];
    OUT out[16];

    PARTS:
    Add16(a=in, b[0]=true, b[1..15]=false, out=out);
}
```



```
|        in        |       out        |
| 0000000000000000 | 0000000000000001 |
| 1111111111111111 | 0000000000000000 |
| 0000000000000101 | 0000000000000110 |
| 1111111111111011 | 1111111111111100 |
```

DESIGNING AND IMPLEMENTING A 16 BIT HACK CPU

1. **HACK CPU:**

   The Hack CPU consists of the ALU specified in chapter 2 and three registers called data register (D), address register (A), and program counter (PC). D and A are general-purpose 16-bit registers that can be manipulated by arithmetic and logical instructions like A=D-1, D=D|A, and so on, following the Hack machine language specified in chapter 4. While the D-register is used solely to store data values, the contents of the A-register can be interpreted in three different ways, depending on the instruction’s context: as a data value, as a RAM address, or as a ROM address. The Hack machine language is based on two 16-bit command types. The address instruction has the format 0vvvvvvvvvvvvvvv, each v being 0 or 1. This instruction causes the computer to load the 15-bit constant vvv...v into the A-register. The compute instruction has the format 111accccccdddjjj. The a- and c-bits instruct the ALU which function to compute, the d-bits instruct where to store the ALU output, and the j-bits specify an optional jump condition, all according to the Hack machine language specification. The computer architecture is wired in such a way that the output of the program counter (PC) chip is connected to the address input of the ROM chip. This way, the ROM chip always emits the word ROM[PC], namely, the contents of the instruction memory location whose address is “pointed at” by the PC. This value is called the current instruction.

2. **Contents of HACK CPU:**

   The HACK CPU has the following key features:

   - 16-bit word size: The CPU processes data in 16-bit chunks.
   - Registers: It includes a program counter (PC), instruction register (IR), and a set of general-purpose registers (R0 to R7).
   - ALU: An Arithmetic Logic Unit to perform arithmetic and logical operations.
   - Memory: Support for RAM and ROM.
   - Input/Output: Basic I/O operations through the RAM.

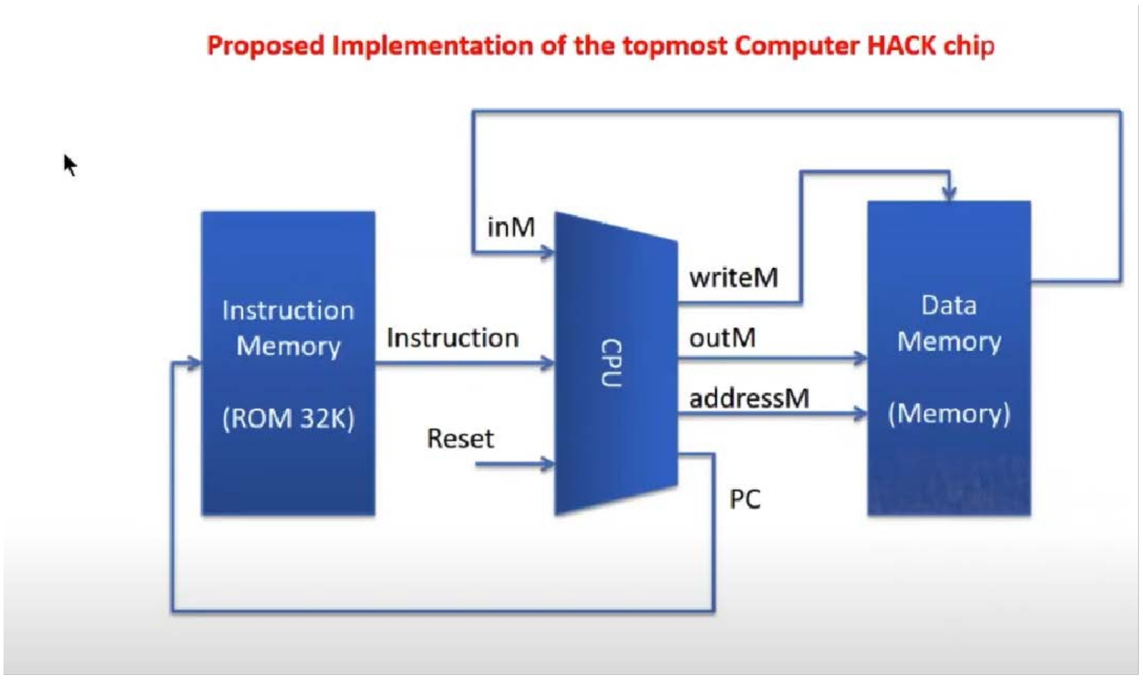
   a. Registers
      - Program Counter (PC): Points to the address of the next instruction.
      - Instruction Register (IR): Holds the current instruction being executed.
      - General-purpose registers (R0 to R7): For temporary data storage.

   b. ALU Operations

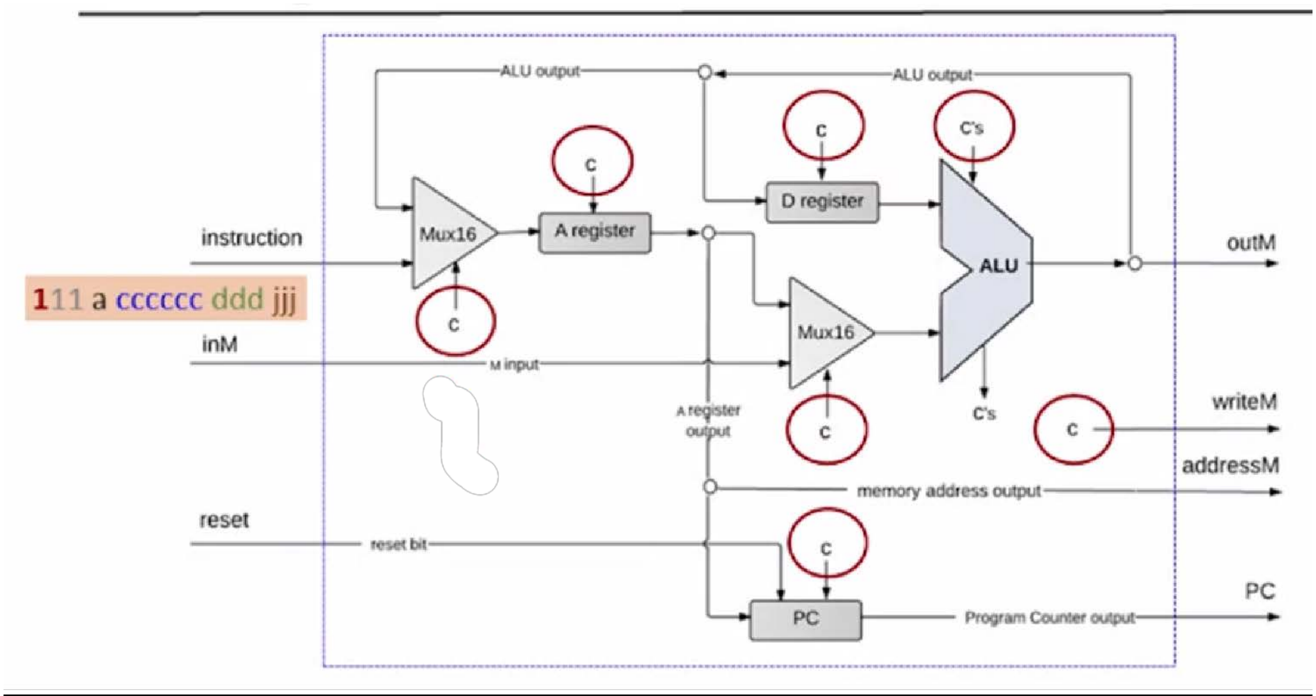
      The ALU supports several operations, such as:
      - Addition
      - Subtraction
      - Bitwise AND, OR, NOT

   c. Control Unit
      - Decodes instructions and controls the execution of operations.

   d. Memory
      - RAM: Stores data and instructions.
      - ROM: Stores the program to be executed.

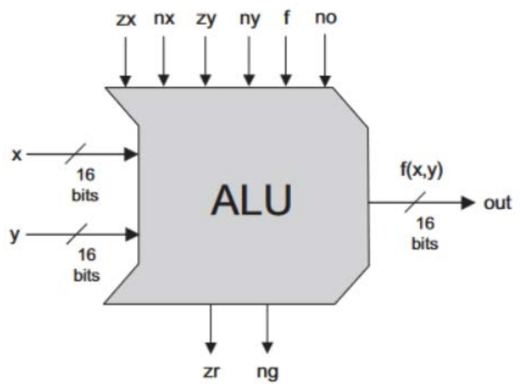


CENTRAL PROCESSING UNIT



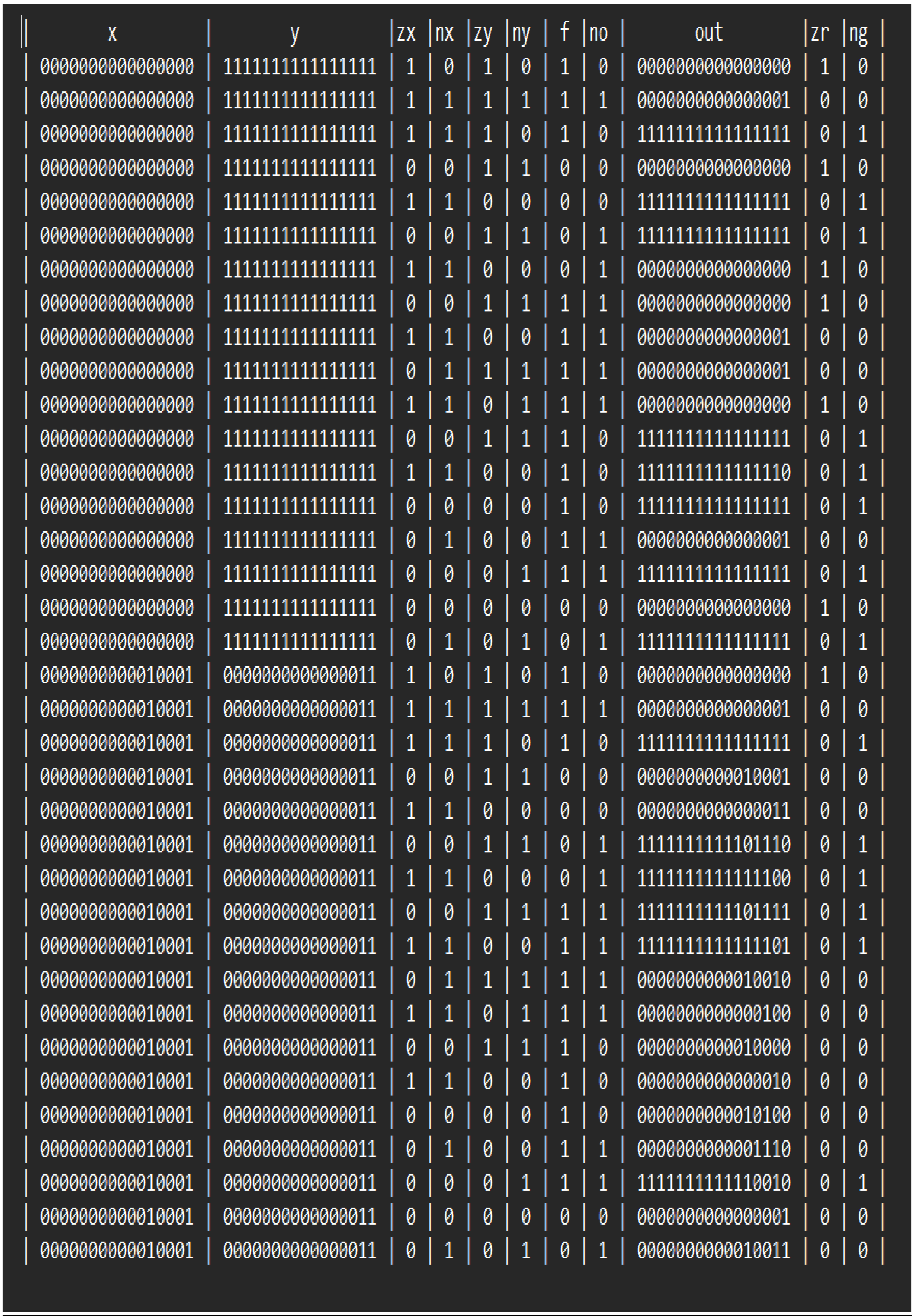
ALU

The Hack ALU computes a fixed set of functions on given two 16-bit inputs, out of which the function can be one of the possible eighteen functions.
We instruct the ALU which function to compute using six input bits, called control bits to the selected binary values.



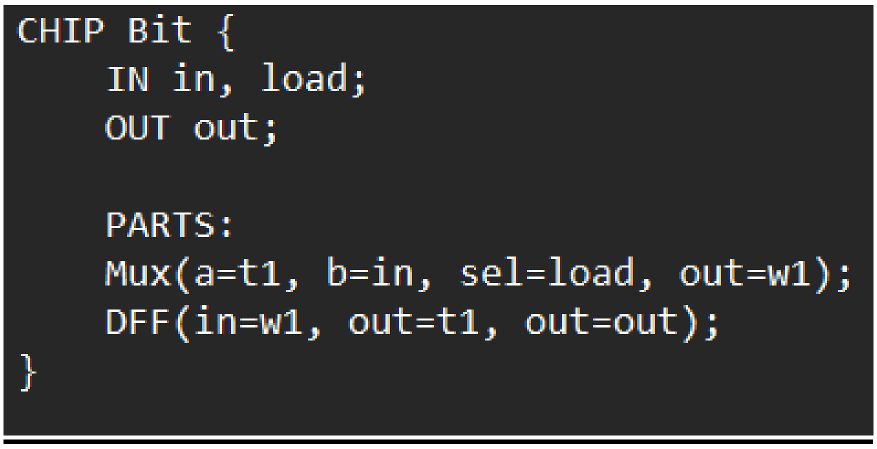
```
CHIP ALU {
    IN
        x[16], y[16],  // 16-bit inputs
        zx, // zero the x input?
        nx, // negate the x input?
        zy, // zero the y input?
        ny, // negate the y input?
        f,  // compute  out = x + y (if f == 1) or out = x & y (if == 0)
        no; // negate the out 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);
        Not16(in=x1, out=notx1);
        Mux16(a=x1, b=notx1, sel=nx, out=tx);

        Mux16(a=y, b=false, sel=zy, out=y1);
        Not16(in=y1, out=noty1);
        Mux16(a=y1, b=noty1, sel=ny, out=ty);

        And16(a=tx, b=ty, out=tand);
        Add16(a=tx, b=ty, out=tadd);
        Mux16(a=tand, b=tadd, sel=f, out=t);

        Not16(in=t, out=nott);
        Mux16(a=t, b=nott, sel=no, out=out, out[15]=ng, out=o1);

        Or16Way(in=o1, out=notzr);
        Not(in=notzr, out=zr);
}
```

OUTPUT



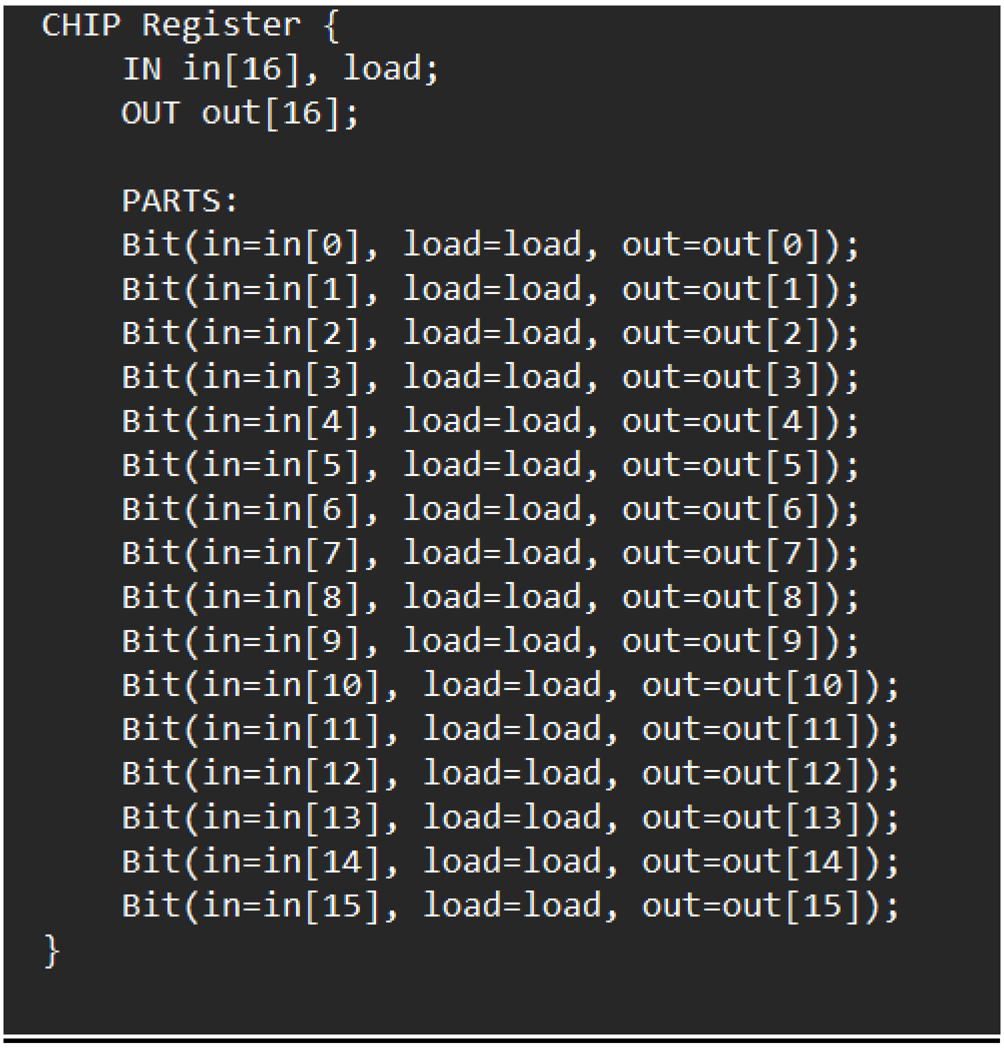
```
||        x         |        y         |zx |nx |zy |ny | f |no |       out        |zr |ng |
| 0000000000000000 | 1111111111111111 | 1 | 0 | 1 | 0 | 1 | 0 | 0000000000000000 | 1 | 0 |
| 0000000000000000 | 1111111111111111 | 1 | 1 | 1 | 1 | 1 | 1 | 0000000000000001 | 0 | 0 |
| 0000000000000000 | 1111111111111111 | 1 | 1 | 1 | 0 | 1 | 0 | 1111111111111111 | 0 | 1 |
| 0000000000000000 | 1111111111111111 | 0 | 0 | 1 | 1 | 0 | 0 | 0000000000000000 | 1 | 0 |
| 0000000000000000 | 1111111111111111 | 1 | 1 | 0 | 0 | 0 | 0 | 1111111111111111 | 0 | 1 |
| 0000000000000000 | 1111111111111111 | 0 | 0 | 1 | 1 | 0 | 1 | 1111111111111111 | 0 | 1 |
| 0000000000000000 | 1111111111111111 | 1 | 1 | 0 | 0 | 0 | 1 | 0000000000000000 | 1 | 0 |
| 0000000000000000 | 1111111111111111 | 0 | 0 | 1 | 1 | 1 | 1 | 0000000000000000 | 1 | 0 |
| 0000000000000000 | 1111111111111111 | 1 | 1 | 0 | 0 | 1 | 1 | 0000000000000001 | 0 | 0 |
| 0000000000000000 | 1111111111111111 | 0 | 1 | 1 | 1 | 1 | 1 | 0000000000000001 | 0 | 0 |
| 0000000000000000 | 1111111111111111 | 1 | 1 | 0 | 1 | 1 | 1 | 0000000000000000 | 1 | 0 |
| 0000000000000000 | 1111111111111111 | 0 | 0 | 1 | 1 | 1 | 0 | 1111111111111111 | 0 | 1 |
| 0000000000000000 | 1111111111111111 | 1 | 1 | 0 | 0 | 1 | 0 | 1111111111111110 | 0 | 1 |
| 0000000000000000 | 1111111111111111 | 0 | 0 | 0 | 0 | 1 | 0 | 1111111111111111 | 0 | 1 |
| 0000000000000000 | 1111111111111111 | 0 | 1 | 0 | 0 | 1 | 1 | 0000000000000001 | 0 | 0 |
| 0000000000000000 | 1111111111111111 | 0 | 0 | 0 | 1 | 1 | 1 | 1111111111111111 | 0 | 1 |
| 0000000000000000 | 1111111111111111 | 0 | 0 | 0 | 0 | 0 | 0 | 0000000000000000 | 1 | 0 |
| 0000000000000000 | 1111111111111111 | 0 | 1 | 0 | 1 | 0 | 1 | 1111111111111111 | 0 | 1 |
| 0000000000010001 | 0000000000000011 | 1 | 0 | 1 | 0 | 1 | 0 | 0000000000000000 | 1 | 0 |
| 0000000000010001 | 0000000000000011 | 1 | 1 | 1 | 1 | 1 | 1 | 0000000000000001 | 0 | 0 |
| 0000000000010001 | 0000000000000011 | 1 | 1 | 1 | 0 | 1 | 0 | 1111111111111111 | 0 | 1 |
| 0000000000010001 | 0000000000000011 | 0 | 0 | 1 | 1 | 0 | 0 | 0000000000010001 | 0 | 0 |
| 0000000000010001 | 0000000000000011 | 1 | 1 | 0 | 0 | 0 | 0 | 0000000000000011 | 0 | 0 |
| 0000000000010001 | 0000000000000011 | 0 | 0 | 1 | 1 | 0 | 1 | 1111111111101110 | 0 | 1 |
| 0000000000010001 | 0000000000000011 | 1 | 1 | 0 | 0 | 0 | 1 | 1111111111111100 | 0 | 1 |
| 0000000000010001 | 0000000000000011 | 0 | 0 | 1 | 1 | 1 | 1 | 1111111111101111 | 0 | 1 |
| 0000000000010001 | 0000000000000011 | 1 | 1 | 0 | 0 | 1 | 1 | 1111111111111101 | 0 | 1 |
| 0000000000010001 | 0000000000000011 | 0 | 1 | 1 | 1 | 1 | 1 | 0000000000010010 | 0 | 0 |
| 0000000000010001 | 0000000000000011 | 1 | 1 | 0 | 1 | 1 | 1 | 0000000000000100 | 0 | 0 |
| 0000000000010001 | 0000000000000011 | 0 | 0 | 1 | 1 | 1 | 0 | 0000000000010000 | 0 | 0 |
| 0000000000010001 | 0000000000000011 | 1 | 1 | 0 | 0 | 1 | 0 | 0000000000000010 | 0 | 0 |
| 0000000000010001 | 0000000000000011 | 0 | 0 | 0 | 0 | 1 | 0 | 0000000000010100 | 0 | 0 |
| 0000000000010001 | 0000000000000011 | 0 | 1 | 0 | 0 | 1 | 1 | 0000000000001110 | 0 | 0 |
| 0000000000010001 | 0000000000000011 | 0 | 0 | 0 | 1 | 1 | 1 | 1111111111110010 | 0 | 1 |
| 0000000000010001 | 0000000000000011 | 0 | 0 | 0 | 0 | 0 | 0 | 0000000000000001 | 0 | 0 |
| 0000000000010001 | 0000000000000011 | 0 | 1 | 0 | 1 | 0 | 1 | 0000000000010011 | 0 | 0 |
```

BIT



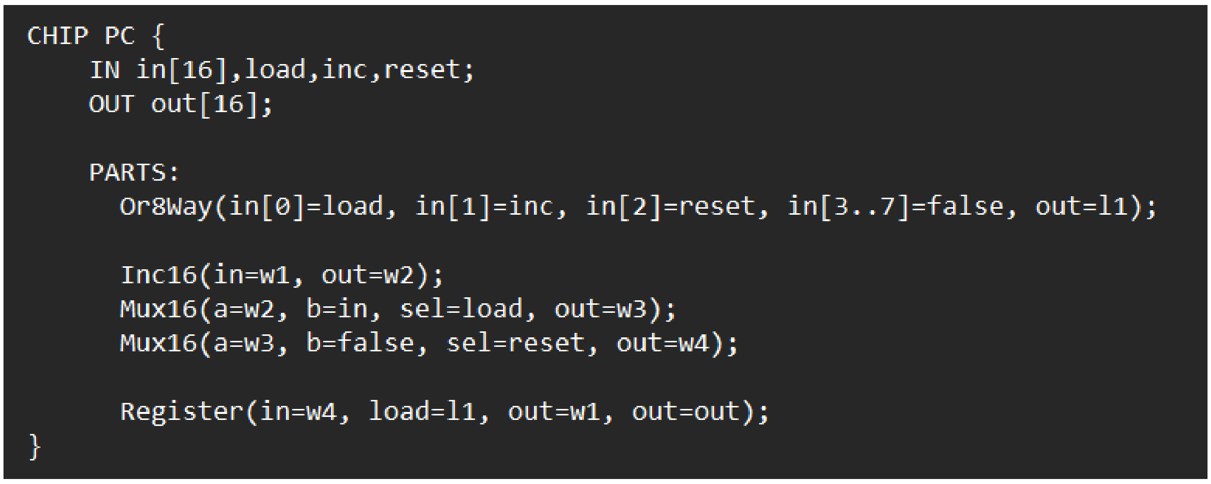
```
CHIP Bit {
    IN in, load;
    OUT out;

    PARTS:
    Mux(a=t1, b=in, sel=load, out=w1);
    DFF(in=w1, out=t1, out=out);
}
```

REGISTER



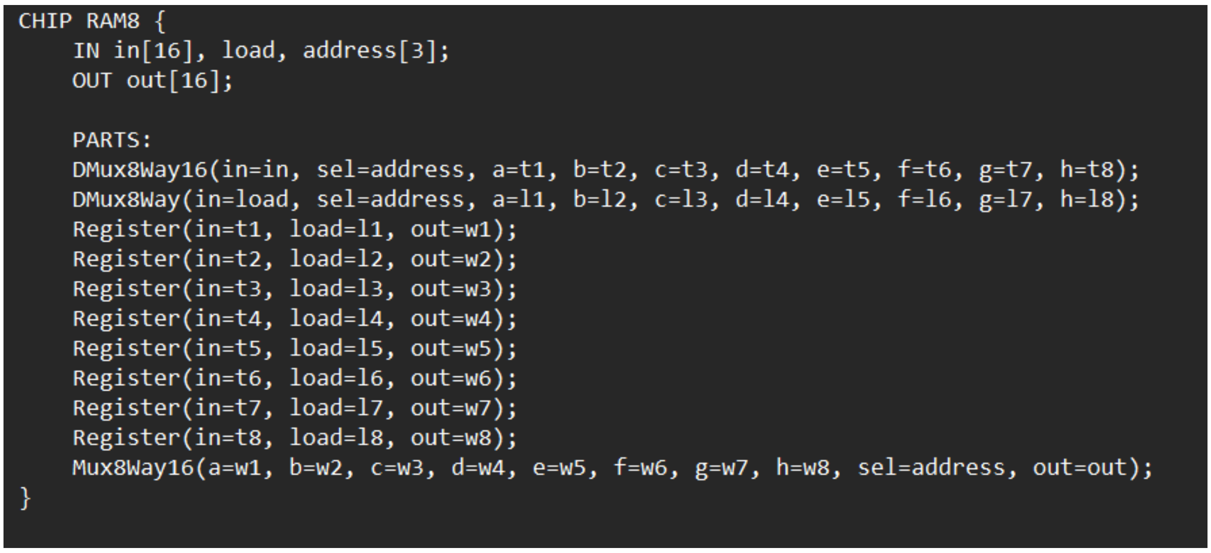
```
CHIP Register {
    IN in[16], load;
    OUT out[16];

    PARTS:
    Bit(in=in[0], load=load, out=out[0]);
    Bit(in=in[1], load=load, out=out[1]);
    Bit(in=in[2], load=load, out=out[2]);
    Bit(in=in[3], load=load, out=out[3]);
    Bit(in=in[4], load=load, out=out[4]);
    Bit(in=in[5], load=load, out=out[5]);
    Bit(in=in[6], load=load, out=out[6]);
    Bit(in=in[7], load=load, out=out[7]);
    Bit(in=in[8], load=load, out=out[8]);
    Bit(in=in[9], load=load, out=out[9]);
    Bit(in=in[10], load=load, out=out[10]);
    Bit(in=in[11], load=load, out=out[11]);
    Bit(in=in[12], load=load, out=out[12]);
    Bit(in=in[13], load=load, out=out[13]);
    Bit(in=in[14], load=load, out=out[14]);
    Bit(in=in[15], load=load, out=out[15]);
}
```

PC



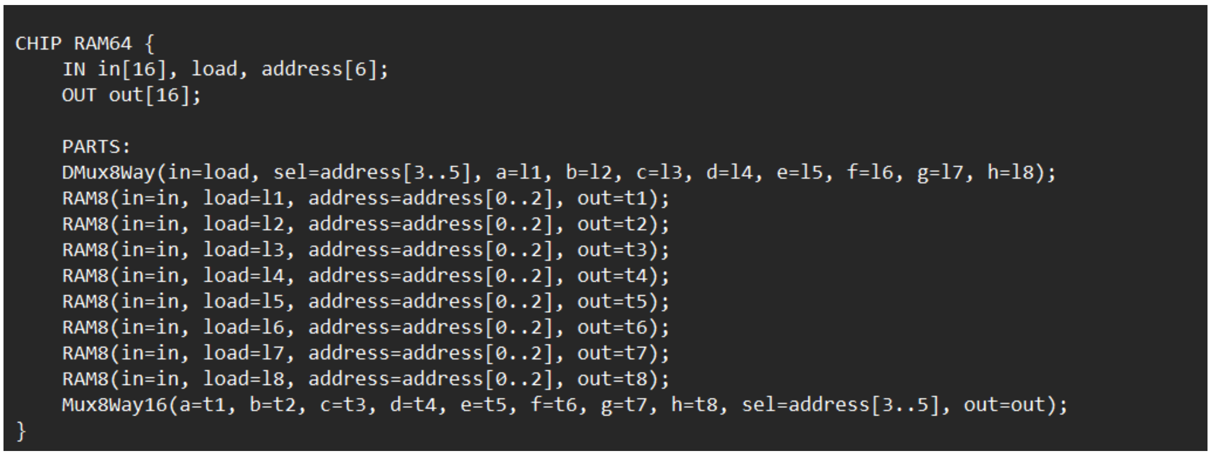
```
CHIP PC {
    IN in[16],load,inc,reset;
    OUT out[16];

    PARTS:
      Or8Way(in[0]=load, in[1]=inc, in[2]=reset, in[3..7]=false, out=l1);

      Inc16(in=w1, out=w2);
      Mux16(a=w2, b=in, sel=load, out=w3);
      Mux16(a=w3, b=false, sel=reset, out=w4);

      Register(in=w4, load=l1, out=w1, out=out);
}
```

RAM8



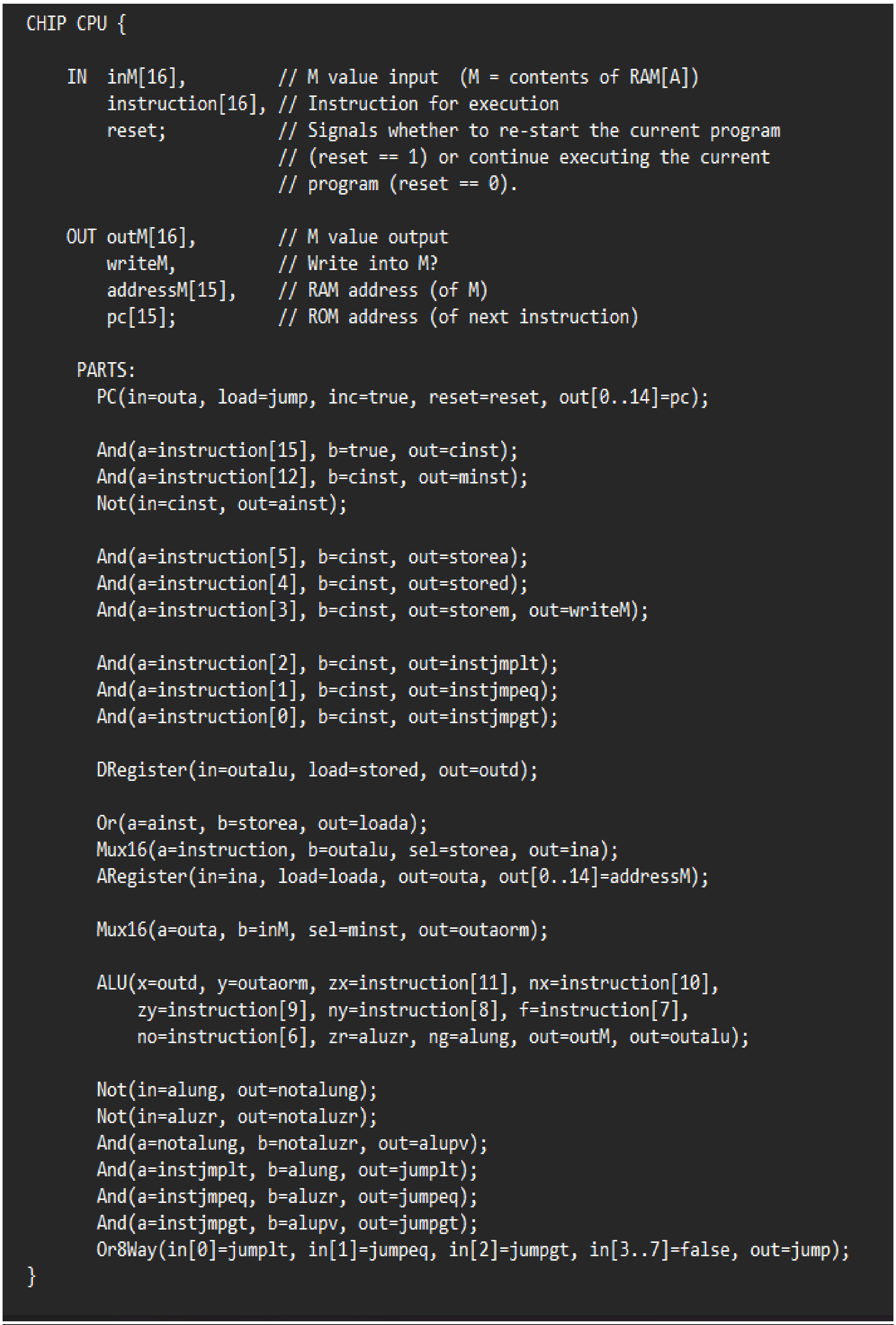
```
CHIP RAM8 {
    IN in[16], load, address[3];
    OUT out[16];

    PARTS:
    DMux8Way16(in=in, sel=address, a=t1, b=t2, c=t3, d=t4, e=t5, f=t6, g=t7, h=t8);
    DMux8Way(in=load, sel=address, a=l1, b=l2, c=l3, d=l4, e=l5, f=l6, g=l7, h=l8);
    Register(in=t1, load=l1, out=w1);
    Register(in=t2, load=l2, out=w2);
    Register(in=t3, load=l3, out=w3);
    Register(in=t4, load=l4, out=w4);
    Register(in=t5, load=l5, out=w5);
    Register(in=t6, load=l6, out=w6);
    Register(in=t7, load=l7, out=w7);
    Register(in=t8, load=l8, out=w8);
    Mux8Way16(a=w1, b=w2, c=w3, d=w4, e=w5, f=w6, g=w7, h=w8, sel=address, out=out);
}
```

RAM64



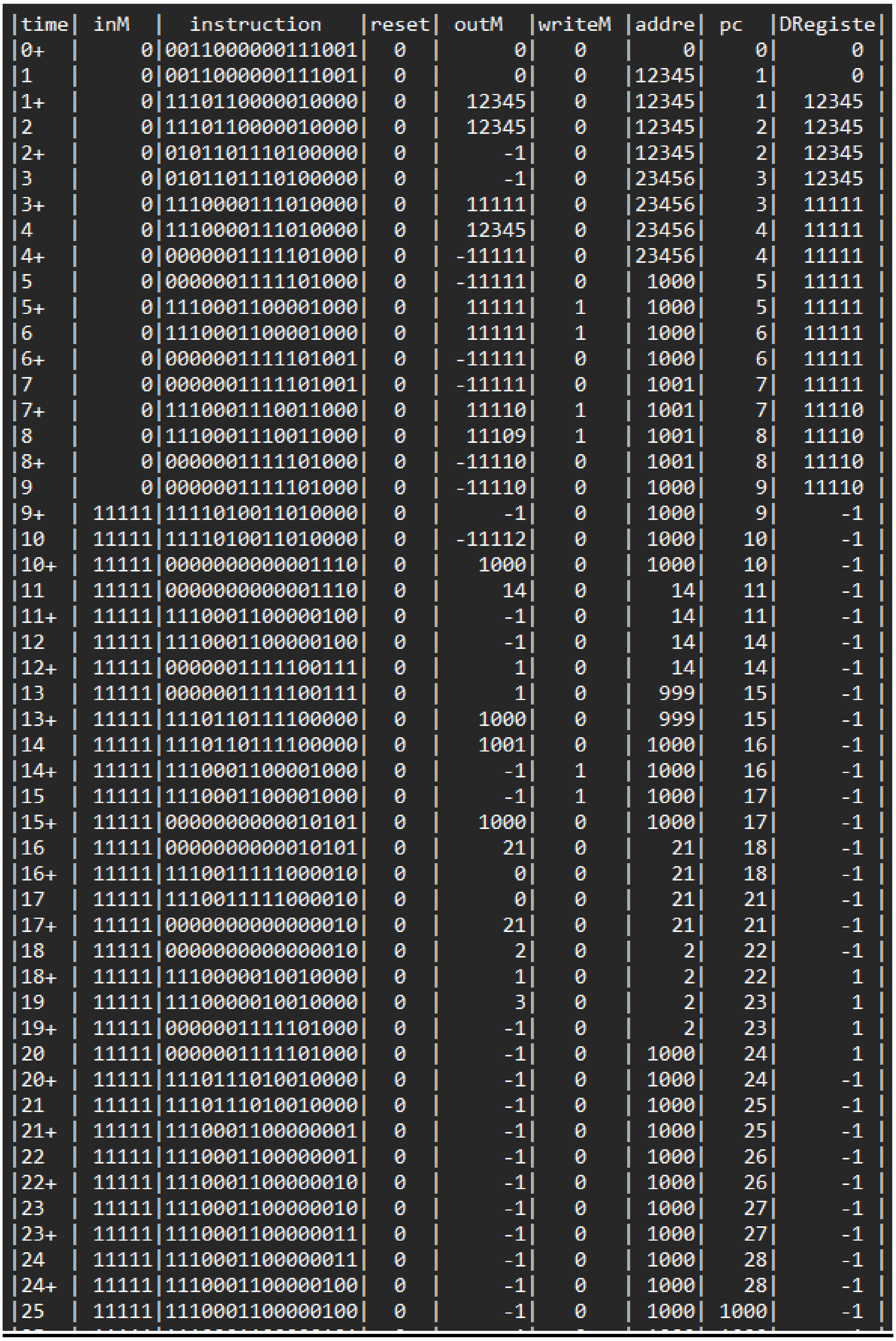
```
CHIP RAM64 {
    IN in[16], load, address[6];
    OUT out[16];

    PARTS:
    DMux8Way(in=load, sel=address[3..5], a=l1, b=l2, c=l3, d=l4, e=l5, f=l6, g=l7, h=l8);
    RAM8(in=in, load=l1, address=address[0..2], out=t1);
    RAM8(in=in, load=l2, address=address[0..2], out=t2);
    RAM8(in=in, load=l3, address=address[0..2], out=t3);
    RAM8(in=in, load=l4, address=address[0..2], out=t4);
    RAM8(in=in, load=l5, address=address[0..2], out=t5);
    RAM8(in=in, load=l6, address=address[0..2], out=t6);
    RAM8(in=in, load=l7, address=address[0..2], out=t7);
    RAM8(in=in, load=l8, address=address[0..2], out=t8);
    Mux8Way16(a=t1, b=t2, c=t3, d=t4, e=t5, f=t6, g=t7, h=t8, sel=address[3..5], out=out);
}
```

HACK CPU

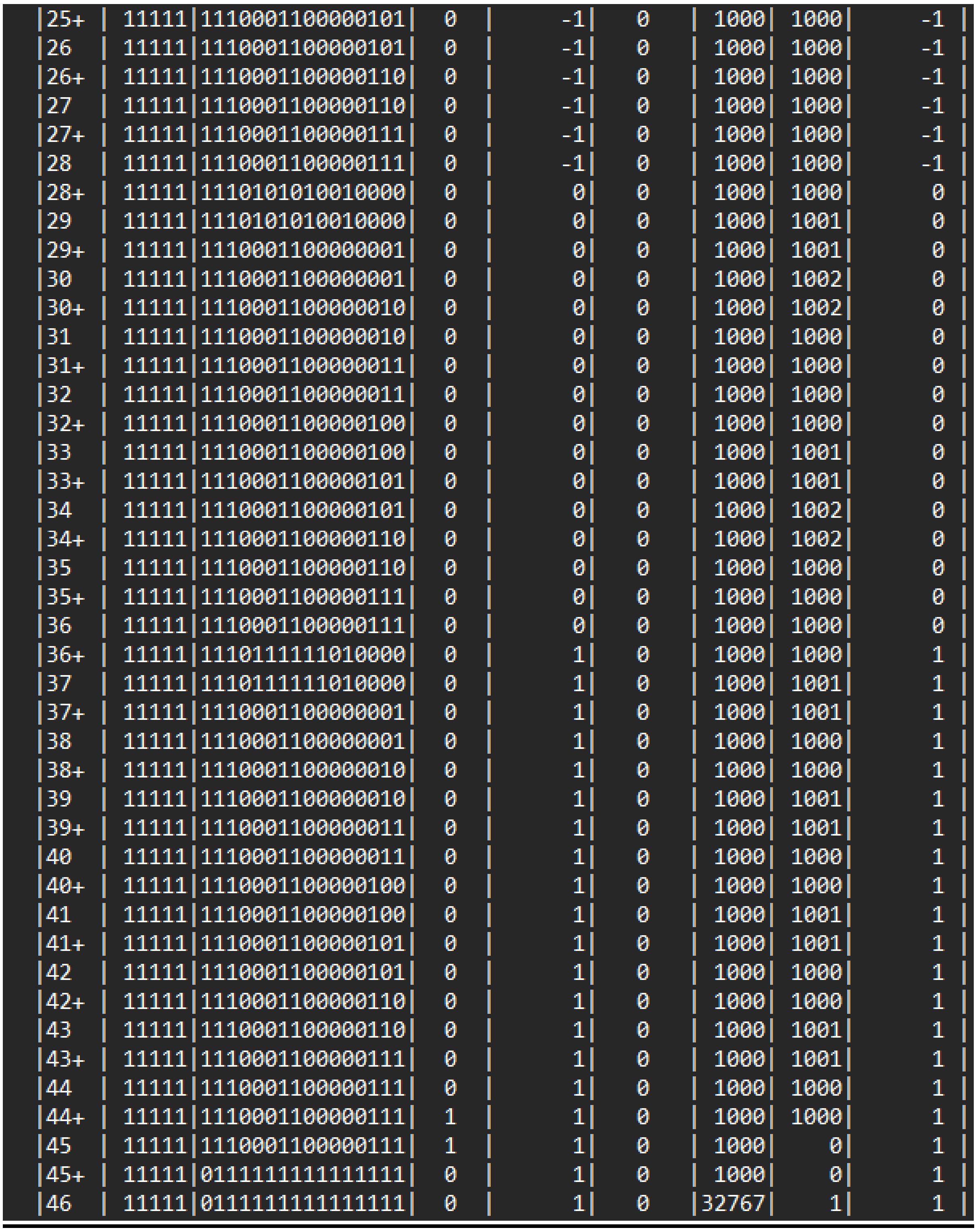


```
CHIP CPU {

    IN  inM[16],         // M value input  (M = contents of RAM[A])
        instruction[16], // Instruction for execution
        reset;           // Signals whether to re-start the current program
                         // (reset == 1) or continue executing the current
                         // program (reset == 0).

    OUT outM[16],        // M value output
        writeM,          // Write into M?
        addressM[15],    // RAM address (of M)
        pc[15];          // ROM address (of next instruction)

     PARTS:
       PC(in=outa, load=jump, inc=true, reset=reset, out[0..14]=pc);

       And(a=instruction[15], b=true, out=cinst);
       And(a=instruction[12], b=cinst, out=minst);
       Not(in=cinst, out=ainst);

       And(a=instruction[5], b=cinst, out=storea);
       And(a=instruction[4], b=cinst, out=stored);
       And(a=instruction[3], b=cinst, out=storem, out=writeM);

       And(a=instruction[2], b=cinst, out=instjmplt);
       And(a=instruction[1], b=cinst, out=instjmpeq);
       And(a=instruction[0], b=cinst, out=instjmpgt);

       DRegister(in=outalu, load=stored, out=outd);

       Or(a=ainst, b=storea, out=loada);
       Mux16(a=instruction, b=outalu, sel=storea, out=ina);
       ARegister(in=ina, load=loada, out=outa, out[0..14]=addressM);

       Mux16(a=outa, b=inM, sel=minst, out=outaorm);

       ALU(x=outd, y=outaorm, zx=instruction[11], nx=instruction[10],
           zy=instruction[9], ny=instruction[8], f=instruction[7],
           no=instruction[6], zr=aluzr, ng=alung, out=outM, out=outalu);

       Not(in=alung, out=notalung);
       Not(in=aluzr, out=notaluzr);
       And(a=notalung, b=notaluzr, out=alupv);
       And(a=instjmplt, b=alung, out=jumplt);
       And(a=instjmpeq, b=aluzr, out=jumpeq);
       And(a=instjmpgt, b=alupv, out=jumpgt);
       Or8Way(in[0]=jumplt, in[1]=jumpeq, in[2]=jumpgt, in[3..7]=false, out=jump);
}
```

OUTPUT



```
|time| inM  |  instruction   |reset| outM  |writeM |addre|  pc  |DRegiste|
|0+  |     0|0011000000111001|  0  |      0|   0   |    0|     0|     0  |
|1   |     0|0011000000111001|  0  |      0|   0   |12345|     1|     0  |
|1+  |     0|1110110000010000|  0  |  12345|   0   |12345|     1| 12345  |
|2   |     0|1110110000010000|  0  |  12345|   0   |12345|     2| 12345  |
|2+  |     0|0101101110100000|  0  |     -1|   0   |12345|     2| 12345  |
|3   |     0|0101101110100000|  0  |     -1|   0   |23456|     3| 12345  |
|3+  |     0|1110000111010000|  0  |  11111|   0   |23456|     3| 11111  |
|4   |     0|1110000111010000|  0  |  12345|   0   |23456|     4| 11111  |
|4+  |     0|0000001111101000|  0  | -11111|   0   |23456|     4| 11111  |
|5   |     0|0000001111101000|  0  | -11111|   0   | 1000|     5| 11111  |
|5+  |     0|1110001100001000|  0  |  11111|   1   | 1000|     5| 11111  |
|6   |     0|1110001100001000|  0  |  11111|   1   | 1000|     6| 11111  |
|6+  |     0|0000001111101001|  0  | -11111|   0   | 1000|     6| 11111  |
|7   |     0|0000001111101001|  0  | -11111|   0   | 1001|     7| 11111  |
|7+  |     0|1110001110011000|  0  |  11110|   1   | 1001|     7| 11110  |
|8   |     0|1110001110011000|  0  |  11109|   1   | 1001|     8| 11110  |
|8+  |     0|0000001111101000|  0  | -11110|   0   | 1001|     8| 11110  |
|9   |     0|0000001111101000|  0  | -11110|   0   | 1000|     9| 11110  |
|9+  | 11111|1111010011010000|  0  |     -1|   0   | 1000|     9|    -1  |
|10  | 11111|1111010011010000|  0  | -11112|   0   | 1000|    10|    -1  |
|10+ | 11111|0000000000001110|  0  |   1000|   0   | 1000|    10|    -1  |
|11  | 11111|0000000000001110|  0  |     14|   0   |   14|    11|    -1  |
|11+ | 11111|1110001100000100|  0  |     -1|   0   |   14|    11|    -1  |
|12  | 11111|1110001100000100|  0  |     -1|   0   |   14|    14|    -1  |
|12+ | 11111|0000001111100111|  0  |      1|   0   |   14|    14|    -1  |
|13  | 11111|0000001111100111|  0  |      1|   0   |  999|    15|    -1  |
|13+ | 11111|1110110111100000|  0  |   1000|   0   |  999|    15|    -1  |
|14  | 11111|1110110111100000|  0  |   1001|   0   | 1000|    16|    -1  |
|14+ | 11111|1110001100001000|  0  |     -1|   1   | 1000|    16|    -1  |
|15  | 11111|1110001100001000|  0  |     -1|   1   | 1000|    17|    -1  |
|15+ | 11111|0000000000010101|  0  |   1000|   0   | 1000|    17|    -1  |
|16  | 11111|0000000000010101|  0  |     21|   0   |   21|    18|    -1  |
|16+ | 11111|1110011111000010|  0  |      0|   0   |   21|    18|    -1  |
|17  | 11111|1110011111000010|  0  |      0|   0   |   21|    21|    -1  |
|17+ | 11111|0000000000000010|  0  |     21|   0   |   21|    21|    -1  |
|18  | 11111|0000000000000010|  0  |      2|   0   |    2|    22|    -1  |
|18+ | 11111|1110000010010000|  0  |      1|   0   |    2|    22|     1  |
|19  | 11111|1110000010010000|  0  |      3|   0   |    2|    23|     1  |
|19+ | 11111|0000001111101000|  0  |     -1|   0   |    2|    23|     1  |
|20  | 11111|0000001111101000|  0  |     -1|   0   | 1000|    24|     1  |
|20+ | 11111|1110111010010000|  0  |     -1|   0   | 1000|    24|    -1  |
|21  | 11111|1110111010010000|  0  |     -1|   0   | 1000|    25|    -1  |
|21+ | 11111|1110001100000001|  0  |     -1|   0   | 1000|    25|    -1  |
|22  | 11111|1110001100000001|  0  |     -1|   0   | 1000|    26|    -1  |
|22+ | 11111|1110001100000010|  0  |     -1|   0   | 1000|    26|    -1  |
|23  | 11111|1110001100000010|  0  |     -1|   0   | 1000|    27|    -1  |
|23+ | 11111|1110001100000011|  0  |     -1|   0   | 1000|    27|    -1  |
|24  | 11111|1110001100000011|  0  |     -1|   0   | 1000|    28|    -1  |
|24+ | 11111|1110001100000100|  0  |     -1|   0   | 1000|    28|    -1  |
|25  | 11111|1110001100000100|  0  |     -1|   0   | 1000|  1000|    -1  |
```



```
|25+ | 11111|1110001100000101| 0  |       -1| 0   | 1000| 1000|      -1 |
|26  | 11111|1110001100000101| 0  |       -1| 0   | 1000| 1000|      -1 |
|26+ | 11111|1110001100000110| 0  |       -1| 0   | 1000| 1000|      -1 |
|27  | 11111|1110001100000110| 0  |       -1| 0   | 1000| 1000|      -1 |
|27+ | 11111|1110001100000111| 0  |       -1| 0   | 1000| 1000|      -1 |
|28  | 11111|1110001100000111| 0  |       -1| 0   | 1000| 1000|      -1 |
|28+ | 11111|1110101010010000| 0  |        0| 0   | 1000| 1000|       0 |
|29  | 11111|1110101010010000| 0  |        0| 0   | 1000| 1001|       0 |
|29+ | 11111|1110001100000001| 0  |        0| 0   | 1000| 1001|       0 |
|30  | 11111|1110001100000001| 0  |        0| 0   | 1000| 1002|       0 |
|30+ | 11111|1110001100000010| 0  |        0| 0   | 1000| 1002|       0 |
|31  | 11111|1110001100000010| 0  |        0| 0   | 1000| 1000|       0 |
|31+ | 11111|1110001100000011| 0  |        0| 0   | 1000| 1000|       0 |
|32  | 11111|1110001100000011| 0  |        0| 0   | 1000| 1000|       0 |
|32+ | 11111|1110001100000100| 0  |        0| 0   | 1000| 1000|       0 |
|33  | 11111|1110001100000100| 0  |        0| 0   | 1000| 1001|       0 |
|33+ | 11111|1110001100000101| 0  |        0| 0   | 1000| 1001|       0 |
|34  | 11111|1110001100000101| 0  |        0| 0   | 1000| 1002|       0 |
|34+ | 11111|1110001100000110| 0  |        0| 0   | 1000| 1002|       0 |
|35  | 11111|1110001100000110| 0  |        0| 0   | 1000| 1000|       0 |
|35+ | 11111|1110001100000111| 0  |        0| 0   | 1000| 1000|       0 |
|36  | 11111|1110001100000111| 0  |        0| 0   | 1000| 1000|       0 |
|36+ | 11111|1110111111010000| 0  |        1| 0   | 1000| 1000|       1 |
|37  | 11111|1110111111010000| 0  |        1| 0   | 1000| 1001|       1 |
|37+ | 11111|1110001100000001| 0  |        1| 0   | 1000| 1001|       1 |
|38  | 11111|1110001100000001| 0  |        1| 0   | 1000| 1000|       1 |
|38+ | 11111|1110001100000010| 0  |        1| 0   | 1000| 1000|       1 |
|39  | 11111|1110001100000010| 0  |        1| 0   | 1000| 1001|       1 |
|39+ | 11111|1110001100000011| 0  |        1| 0   | 1000| 1001|       1 |
|40  | 11111|1110001100000011| 0  |        1| 0   | 1000| 1000|       1 |
|40+ | 11111|1110001100000100| 0  |        1| 0   | 1000| 1000|       1 |
|41  | 11111|1110001100000100| 0  |        1| 0   | 1000| 1001|       1 |
|41+ | 11111|1110001100000101| 0  |        1| 0   | 1000| 1001|       1 |
|42  | 11111|1110001100000101| 0  |        1| 0   | 1000| 1000|       1 |
|42+ | 11111|1110001100000110| 0  |        1| 0   | 1000| 1000|       1 |
|43  | 11111|1110001100000110| 0  |        1| 0   | 1000| 1001|       1 |
|43+ | 11111|1110001100000111| 0  |        1| 0   | 1000| 1001|       1 |
|44  | 11111|1110001100000111| 0  |        1| 0   | 1000| 1000|       1 |
|44+ | 11111|1110001100000111| 1  |        1| 0   | 1000| 1000|       1 |
|45  | 11111|1110001100000111| 1  |        1| 0   | 1000|    0|       1 |
|45+ | 11111|0111111111111111| 0  |        1| 0   | 1000|    0|       1 |
|46  | 11111|0111111111111111| 0  |        1| 0   |32767|    1|       1 |
```

DESIGN AND IMPLEMENT A SYNCHRONOUS COUNTER THAT COUNTS DOWN FROM DECIMAL DIGIT 9 TO 0

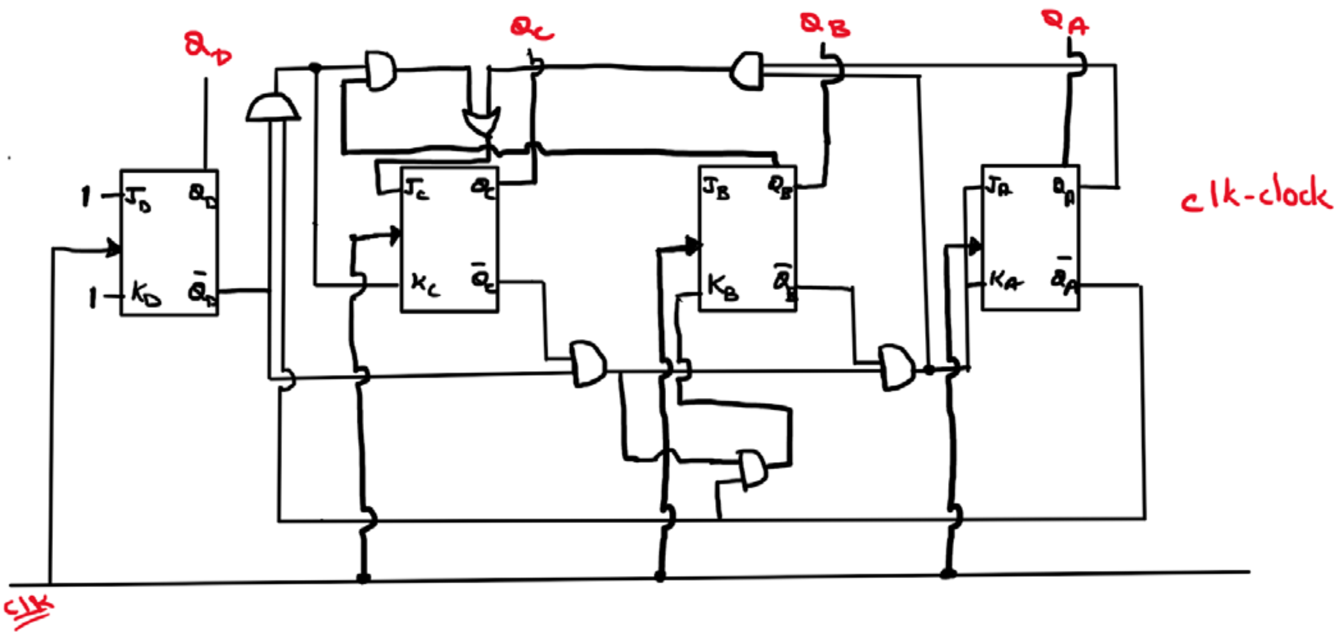
Introduction:
A 4-bit 9-0 Synchronous Counter is a digital circuit that counts in a binary sequence from 9 to 0 and then resets to 9, essentially counting decimal digits from 9 to 0. It's built using flip-flops, which change states in synchronization with the clock signal, ensuring all bits update at the same time. Synchronous counters have several advantages over asynchronous counters, including reduced propagation delays and increased reliability at higher clock frequencies. These counters find applications in digital clocks, timers, and other devices requiring precise counting.

This project aims to design and implement a 4-bit synchronous counter that limits the counting to a range from 9 to 0 (binary sequence 1001 to 0000). Such a counter is also known as a decade counter because it counts ten states. Beyond the zeroth count (0000), the counter resets to zero on the next clock pulse.

Objectives:
Understand the fundamentals of synchronous counters - Gain insight into how synchronous counters differ from asynchronous counters, focusing on the use of a clock pulse to synchronize state changes.
Design a 4-bit binary counter circuit - Develop a binary counter using JK flip-flops to count sequentially from 9 to 0.
Implement a reset mechanism - Ensure the counter resets back to 0 after reaching the binary equivalent of 0 (0000).

Analyse applications - Explore the practical applications of synchronous counters in devices requiring controlled counting mechanisms, such as digital clocks, frequency dividers, and electronic timers.

BLOCK DIAGRAM FOR MOD-10 SYNCHRONOUS DOWN COUNTER

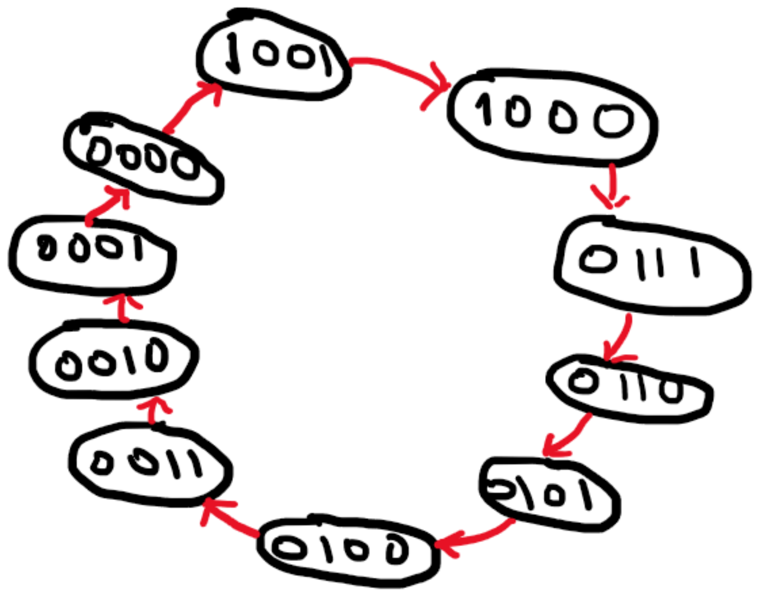


TRUTH TABLE

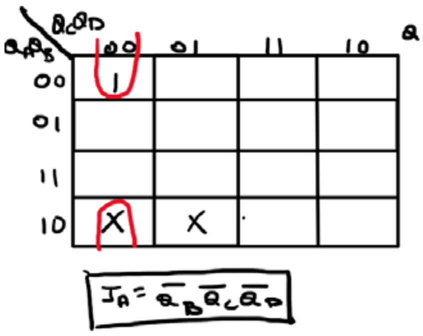
Present State				Next State				J_A	K_A	J_B	K_B	J_C	K_C	J_D	K_D
Q_A	Q_B	Q_C	Q_D	Q_{A+1}	Q_{B+1}	Q_{C+1}	Q_{D+1}								
1	0	0	1	1	0	0	1	x	0	0	x	0	x	x	1
1	0	0	0	0	1	1	1	x	1	1	x	1	x	1	x
0	1	1	1	0	1	1	0	0	x	x	0	x	0	x	1
0	1	1	0	0	1	0	1	0	x	x	0	x	1	1	x
0	1	0	1	0	1	0	0	0	x	x	0	0	x	x	1
0	1	0	0	0	0	1	1	0	x	x	1	1	x	1	x
0	0	1	1	0	0	1	0	0	x	0	x	x	0	x	1
0	0	1	0	0	0	0	1	0	x	0	x	x	1	1	x
0	0	0	1	0	0	0	0	0	x	0	x	0	x	x	1
0	0	0	0	1	0	0	1	1	x	0	x	0	x	1	x

x → Don't care

STATE DIAGRAM

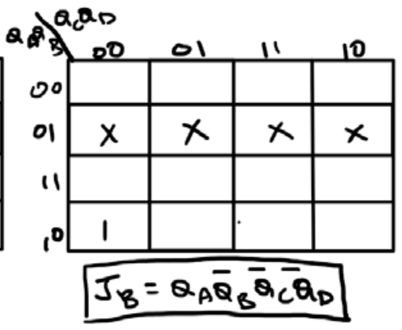


K-MAPS



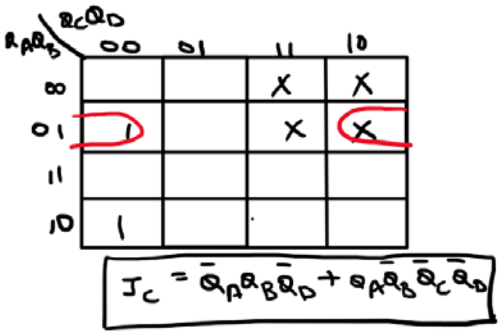
Q_AQ_B \ Q_CQ_D	00	01	11	10
00	1			
01				
11				
10	X	X		

$$\boxed{J_A = \bar{Q}_B \bar{Q}_C \bar{Q}_D}$$



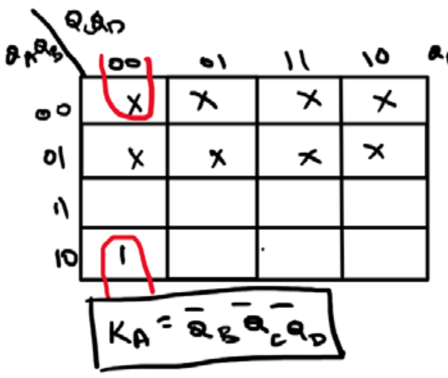
Q_AQ_B \ Q_CQ_D	00	01	11	10
00				
01	X	X	X	X
11				
10	1			

$$\boxed{J_B = Q_A \bar{Q}_B \bar{Q}_C \bar{Q}_D}$$



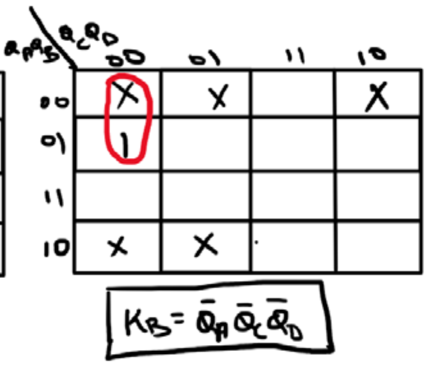
Q_AQ_B \ Q_CQ_D	00	01	11	10
00			X	X
01	1		X	X
11				
10	1			

$$\boxed{J_C = \bar{Q}_A Q_B \bar{Q}_D + Q_A \bar{Q}_B \bar{Q}_C \bar{Q}_D}$$



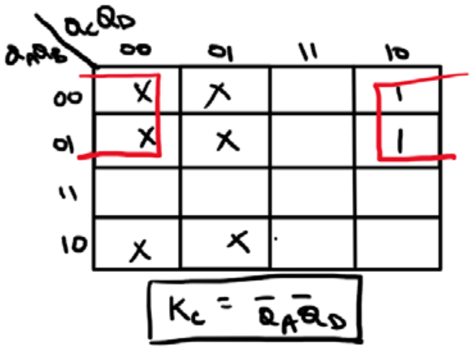
Q_AQ_B \ Q_CQ_D	00	01	11	10
00	X	X	X	X
01	X	X	X	X
11				
10	1			

$$\boxed{K_A = \bar{Q}_B \bar{Q}_C \bar{Q}_D}$$



Q_AQ_B \ Q_CQ_D	00	01	11	10
00	X	X		X
01	1			
11				
10	X	X		

$$\boxed{K_B = \bar{Q}_A \bar{Q}_C \bar{Q}_D}$$

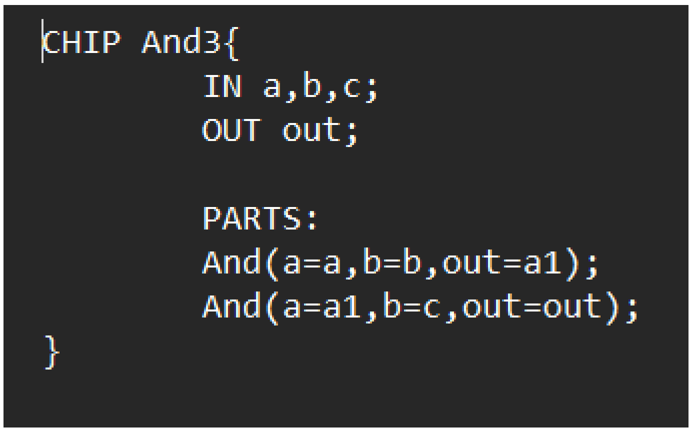


Q_AQ_B \ Q_CQ_D	00	01	11	10
00	X	X		1
01	X	X		1
11				
10	X	X		

$$\boxed{K_C = \bar{Q}_A \bar{Q}_D}$$

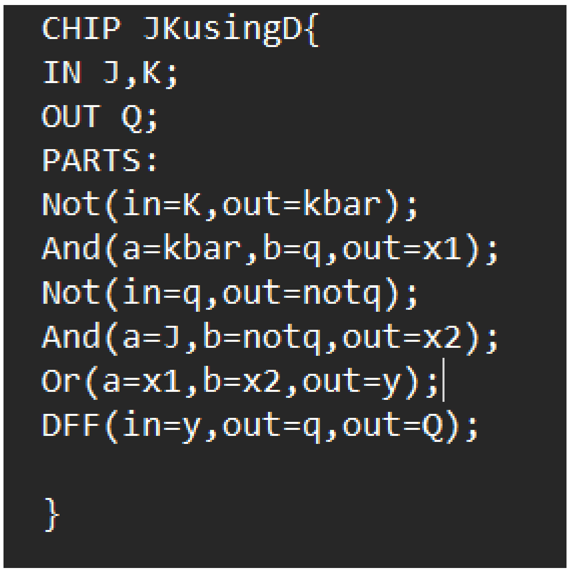
GATES USED

AND3



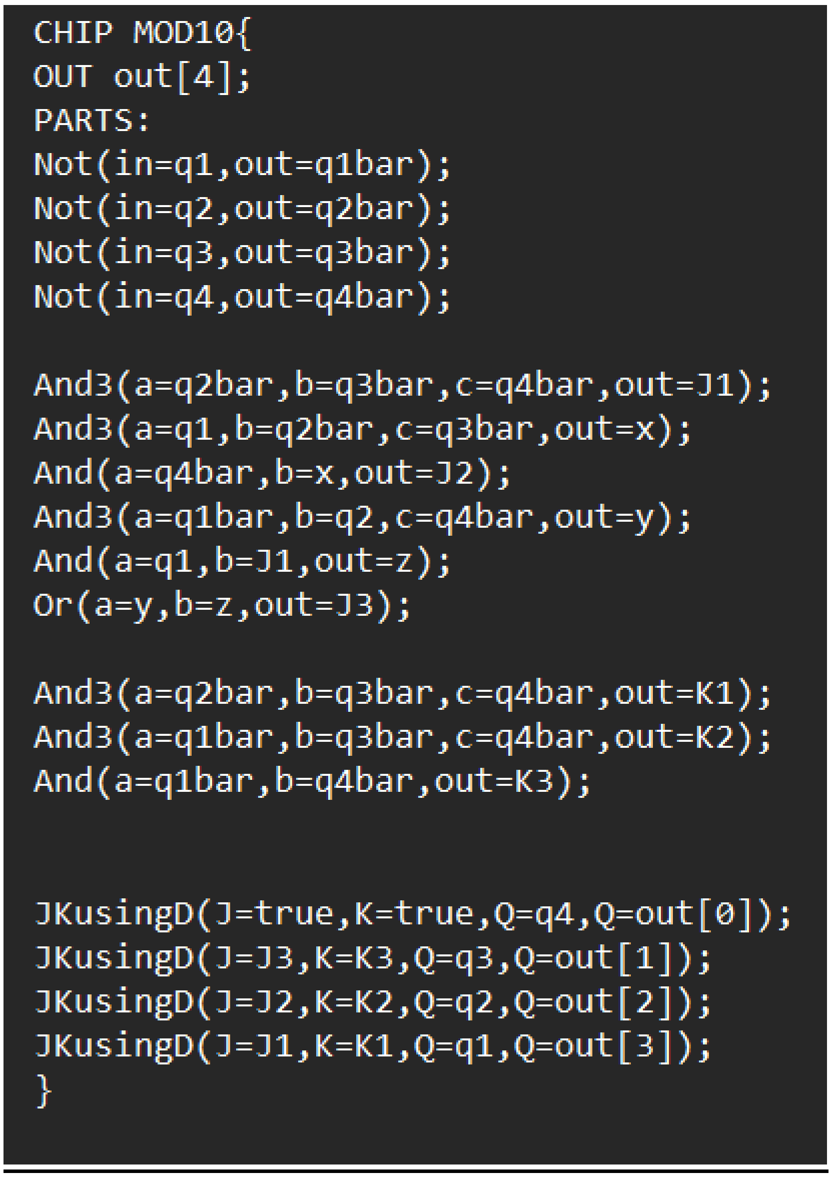
```
CHIP And3{
        IN a,b,c;
        OUT out;

        PARTS:
        And(a=a,b=b,out=a1);
        And(a=a1,b=c,out=out);
}
```

JK FLIP FLOP USING D FLIP FLOP



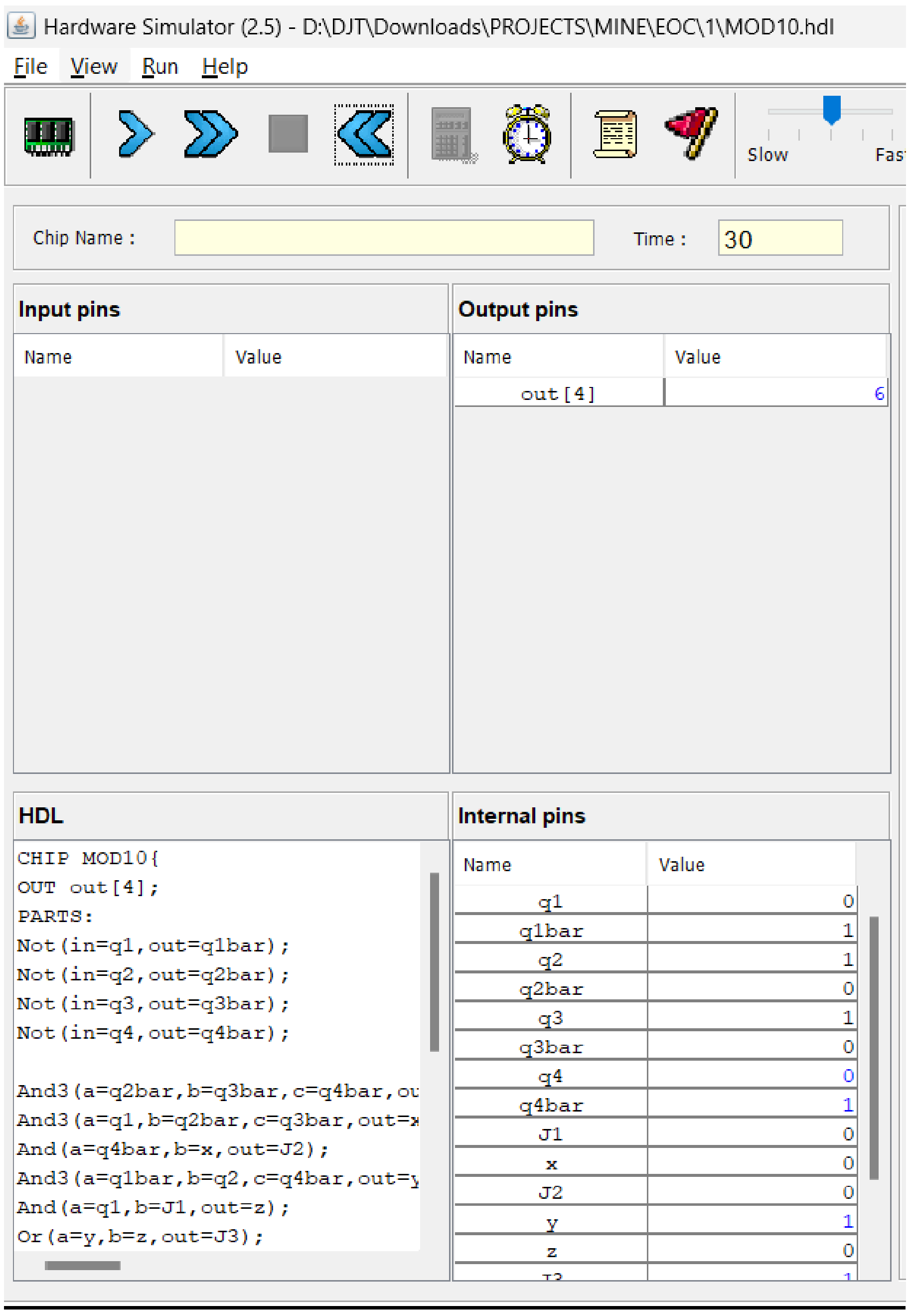
```
CHIP JKusingD{
IN J,K;
OUT Q;
PARTS:
Not(in=K,out=kbar);
And(a=kbar,b=q,out=x1);
Not(in=q,out=notq);
And(a=J,b=notq,out=x2);
Or(a=x1,b=x2,out=y);
DFF(in=y,out=q,out=Q);

}
```

MOD10



```
CHIP MOD10{
OUT out[4];
PARTS:
Not(in=q1,out=q1bar);
Not(in=q2,out=q2bar);
Not(in=q3,out=q3bar);
Not(in=q4,out=q4bar);

And3(a=q2bar,b=q3bar,c=q4bar,out=J1);
And3(a=q1,b=q2bar,c=q3bar,out=x);
And(a=q4bar,b=x,out=J2);
And3(a=q1bar,b=q2,c=q4bar,out=y);
And(a=q1,b=J1,out=z);
Or(a=y,b=z,out=J3);

And3(a=q2bar,b=q3bar,c=q4bar,out=K1);
And3(a=q1bar,b=q3bar,c=q4bar,out=K2);
And(a=q1bar,b=q4bar,out=K3);


JKusingD(J=true,K=true,Q=q4,Q=out[0]);
JKusingD(J=J3,K=K3,Q=q3,Q=out[1]);
JKusingD(J=J2,K=K2,Q=q2,Q=out[2]);
JKusingD(J=J1,K=K1,Q=q1,Q=out[3]);
}
```

OUTPUT


Hardware Simulator (2.5) - D:\DJT\Downloads\PROJECTS\MINE\EOC\1\MOD10.hdl
File
View
Run
Help
Slow
Fas

Chip Name :

Time : 30

Input pins

Name	Value

Output pins

Name	Value
out[4]	6

HDL

```
CHIP MOD10{
OUT out[4];
PARTS:
Not(in=q1,out=q1bar);
Not(in=q2,out=q2bar);
Not(in=q3,out=q3bar);
Not(in=q4,out=q4bar);

And3(a=q2bar,b=q3bar,c=q4bar,ou
And3(a=q1,b=q2bar,c=q3bar,out=x
And(a=q4bar,b=x,out=J2);
And3(a=q1bar,b=q2,c=q4bar,out=y
And(a=q1,b=J1,out=z);
Or(a=y,b=z,out=J3);
```

Internal pins

Name	Value
q1	0
q1bar	1
q2	1
q2bar	0
q3	1
q3bar	0
q4	0
q4bar	1
J1	0
x	0
J2	0
y	1
z	0
J3	1

CONCLUSION

We have designed and implemented a HACK CPU. The Hack CPU is the target for the Jack language compiler later in the course. By compiling Jack code to Hack machine language, students see how high-level code translates to machine-level operations, solidifying their understanding of compilation and CPU operation.

We have also designed and implemented a SYNCHRONOUS COUNTER that counts down from DECIMAL DIGIT 9 TO 0. This type of counter is useful for visual countdowns, such as those seen on display boards at events, sports arenas, or conferences, where it counts down each second or minute. It is often coupled with a display to show the countdown, providing a visual cue that counts down from a set time until an event, like the start of a race or a break in a program. Many household and consumer devices use down counters in settings like microwave timers, washing machine cycles, or cooking timers, decrementing through a sequence until the task completes. Since all flip-flops in a synchronous counter are triggered simultaneously, they offer a predictable and stable output, ideal for precise applications. Synchronous counters are faster than asynchronous counters because they do not suffer from propagation delay between stages, making them suitable for high-speed digital applications.