Department of Artificial Intelligence

22AIE102: Elements of Computing Systems 1

Project Report

Project A – Hack CPU (Central Processing Unit)

Project B – Serial binary adder using SR flip flop

UNDER THE GUIDANCE OF:
Dr. JYOTHISH LAL G,

Centre for Computational Engineering and Networking,

School of AI,

Amrita Vishwa Vidyapeetham.

Group 6:

Jayan S – 23033

KS Jagan – 23034

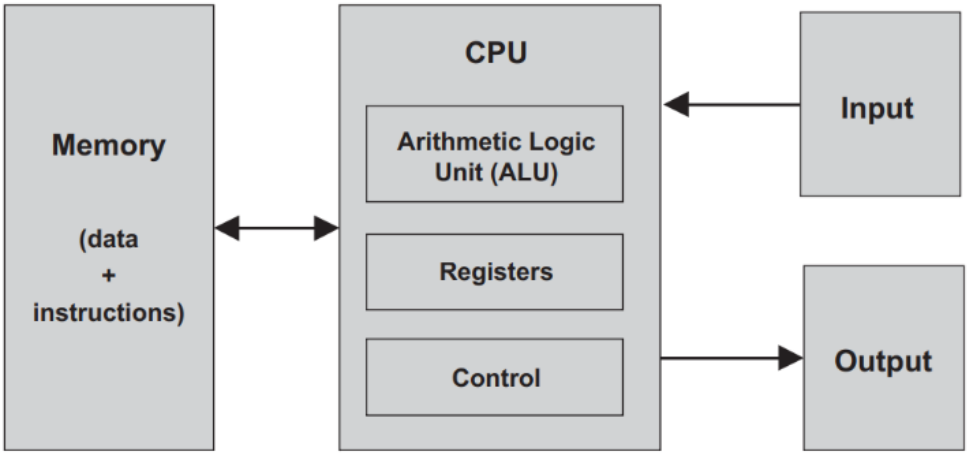
Keerthivasan S V – 23037

Krish S - 23040

Table of Contents

Part A: The HACK CPU

A.1. Introduction



The Central Processing Unit (CPU) serves as the nerve center and computational powerhouse of a computer system, playing a pivotal role in the execution of instructions and the overall functionality of the machine. Often referred to as the brain of the computer, the CPU interprets and carries out the instructions of a computer program, enabling the system to perform a myriad of tasks.

At its core, the CPU consists of several key components that work in tandem to execute instructions efficiently. The Arithmetic-Logic Unit (ALU) performs mathematical and logical operations, such as addition, subtraction, and comparisons. A set of registers, which are high-speed storage locations, temporarily hold data that the CPU is actively processing. Meanwhile, the Control Unit coordinates the execution of instructions, managing the flow of data and ensuring the proper sequencing of operations.

The CPU's ability to fetch, decode, and execute instructions stored in memory is fundamental to its operation. This process involves retrieving instructions, interpreting them, and directing the ALU and other components to carry out the specified tasks. The CPU's speed and efficiency profoundly impact the overall performance of the computer, making it a critical component in modern computing systems.

The CPU's significance lies in its role as the central hub for processing and managing data, making it a cornerstone of computer architecture and functionality. As technology advances, CPUs (Central Processing Units) continue to evolve, providing increased speed, efficiency, and capabilities to meet the demands of modern computing applications.

Memory

INSTRUCTION MEMORY

- Stores functions (e.g., addition, subtraction).
- Acts as a read-only memory (ROM) from the CPU's perspective.
- Each address holds a 16-bit instruction.

DATA MEMORY

- Stores variables for execution.
- Functions as read-write memory (RAM).
- Holds 16-bit values at each address.

Arithmetic Logic Unit (ALU)

The ALU chip is built to perform all the low-level arithmetic and logical operations featured by the computer. For example, a typical ALU can add two numbers, compute a bitwise and function on two numbers, compare two numbers, and so on. How much functionality an ALU should have been a matter of need, budget, energy, and similar cost-effectiveness considerations. Any function not supported by the ALU as a primitive hardware operation can be later realized by the computer’s system software (yielding a slower implementation, of course).

Address Registers

Many machine language instructions involve memory access: reading data, writing data, and fetching instructions. In any one of these operations, we must specify which memory register we wish to operate on. This is done by supplying an address. In some cases, the address is coded as part of the instruction, while in other cases the address is specified, or computed, by some previous instruction. In the latter case, the address must be stored somewhere. This is done using a CPU-resident chip called address register. Unlike regular registers, the output of an address register is typically connected to the address input of a memory device.

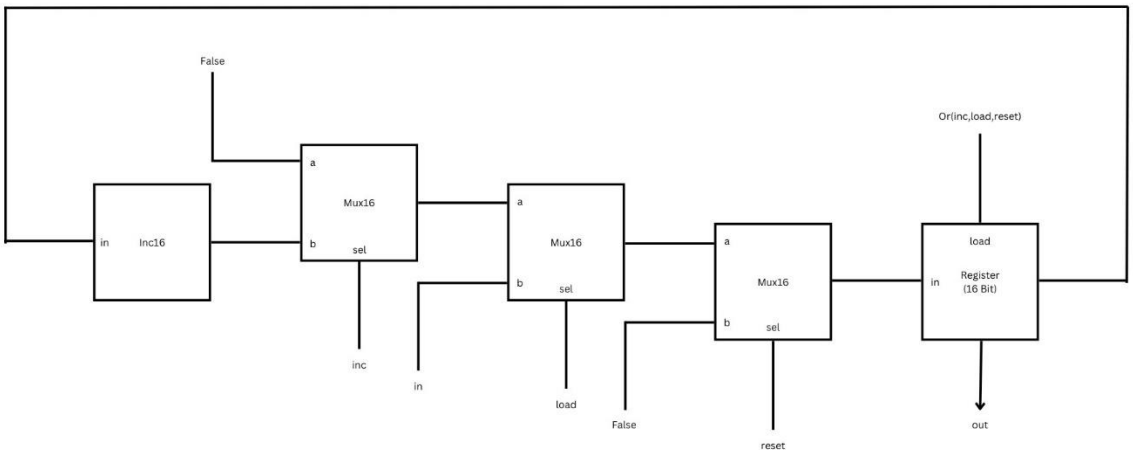
Therefore, placing a value in the address register has the side effect of selecting a particular memory register, and this register makes itself available to subsequent instructions designed to manipulate it. For example, suppose we wish to set Memory [17] to 1. In the Hack language, this can be done using the pair of instructions @17 (which sets A=17 and makes the M mnemonic stand for Memory [17]), followed by M=1 (which sets the selected memory register to 1).

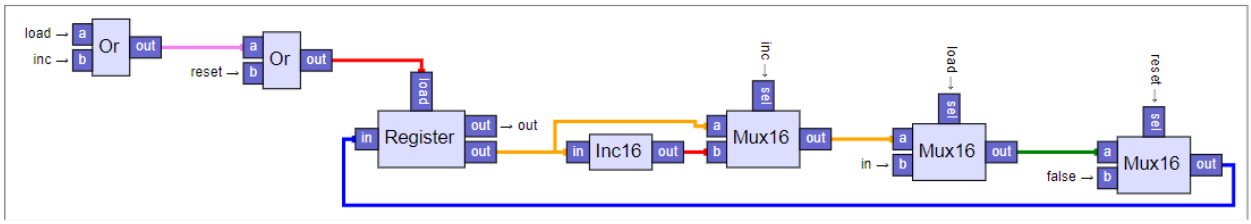
In addition to supporting this fundamental addressing operation, an address register is, well, a register. Therefore, if needed, it can be used as another data register. For example, suppose we wish to set the D register to 17. This can be done using the pair of instructions @17, followed by D=A. Here we use A not as an address register, but rather as a data register. The fact that Memory [17] was selected as a side effect of @17 is completely ignored.

Data Registers

These registers give the CPU short-term memory services. For example, if a program wants to calculate $(a - b) \cdot c$, we must first compute and remember the value of $(a - b)$. In principle, this temporary result can be stored in some memory register. A much more sensible solution is to store it locally inside the CPU, using a data register.

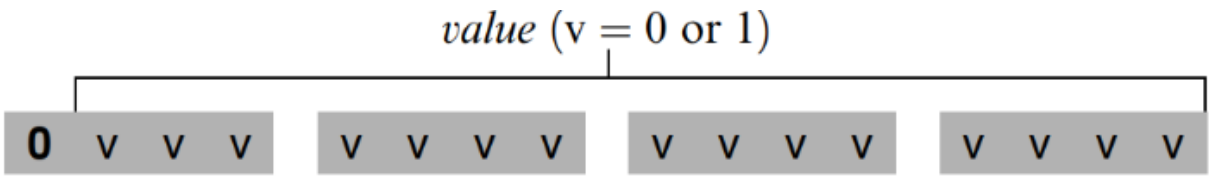
Program Counter





When executing a program, the CPU must always keep track of the address of the instruction that must be fetched and executed next. This address is kept in a special register called program counter, or PC. The contents of the PC are computed and updated as a side effect of executing the current instruction.

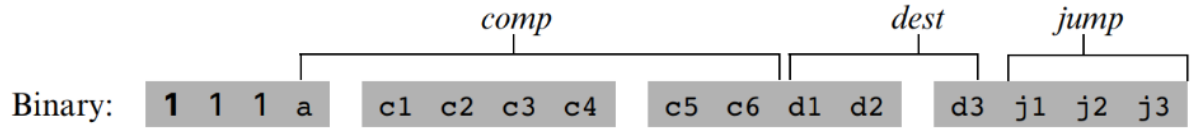
A Instruction Design



The A-instruction is used to set the A register to a 15-bit value. This instruction causes the computer to store the specified value in the A register. For example, the instruction @5, which is equivalent to 0000000000000101, causes the computer to store the binary

representation of 5 in the A register. The A-instruction is used for three different purposes. First, it provides the only way to enter a constant into the computer under program control. Second, it sets the stage for a subsequent C-instruction designed to manipulate a certain data memory location, by first setting A to the address of that location. Third, it sets the stage for a subsequent C-instruction that specifies a jump, by first loading the address of the jump destination to the A register.

C Instruction Design



The C-instruction is the programming workhorse of the Hack platform—the instruction that gets everything done. The instruction code is a specification that answers three questions: (a) what to compute, (b) where to store the computed value, and (c) what to do next? Along with the A-instruction, these specifications determine all the possible operations of the computer.

d1	d2	d3	*Mnemonic*	*Destination (where to store the computed value)*
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

j1 ($out < 0$)	j2 ($out = 0$)	j3 ($out > 0$)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If $out > 0$ jump
0	1	0	JEQ	If $out = 0$ jump
0	1	1	JGE	If $out \geq 0$ jump
1	0	0	JLT	If $out < 0$ jump
1	0	1	JNE	If $out \neq 0$ jump
1	1	0	JLE	If $out \leq 0$ jump
1	1	1	JMP	Jump

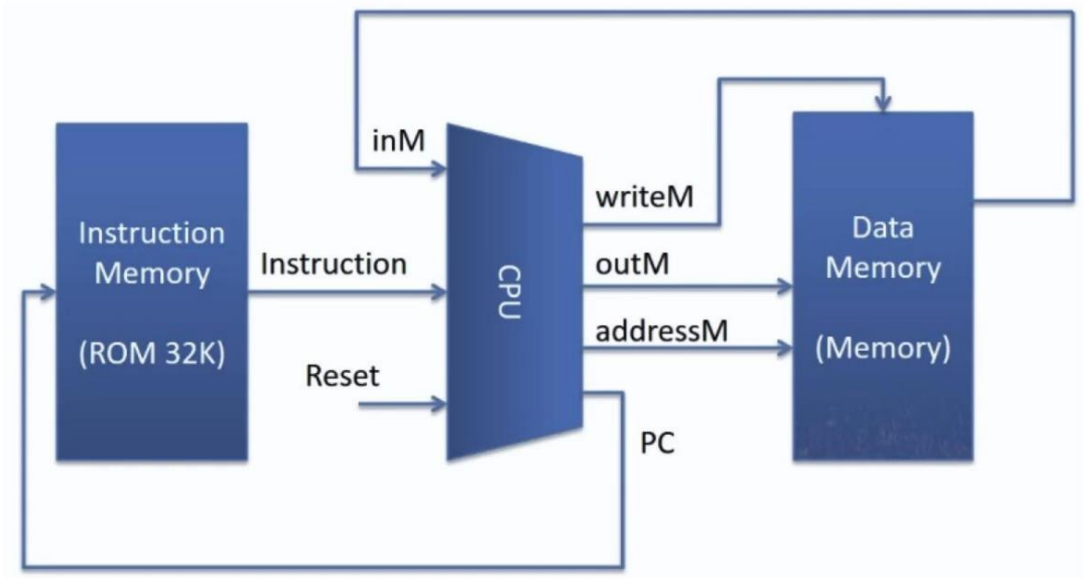
(when a=0) *comp* mnemonic	c1	c2	c3	c4	c5	c6	(when a=1) *comp* mnemonic
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M

A.2. Objective

The objective of project part a is to study the architecture of a general-purpose central processing unit and how it can be programmed/implemented in hardware description language.

A.3. Design

General Hack Computer structure:

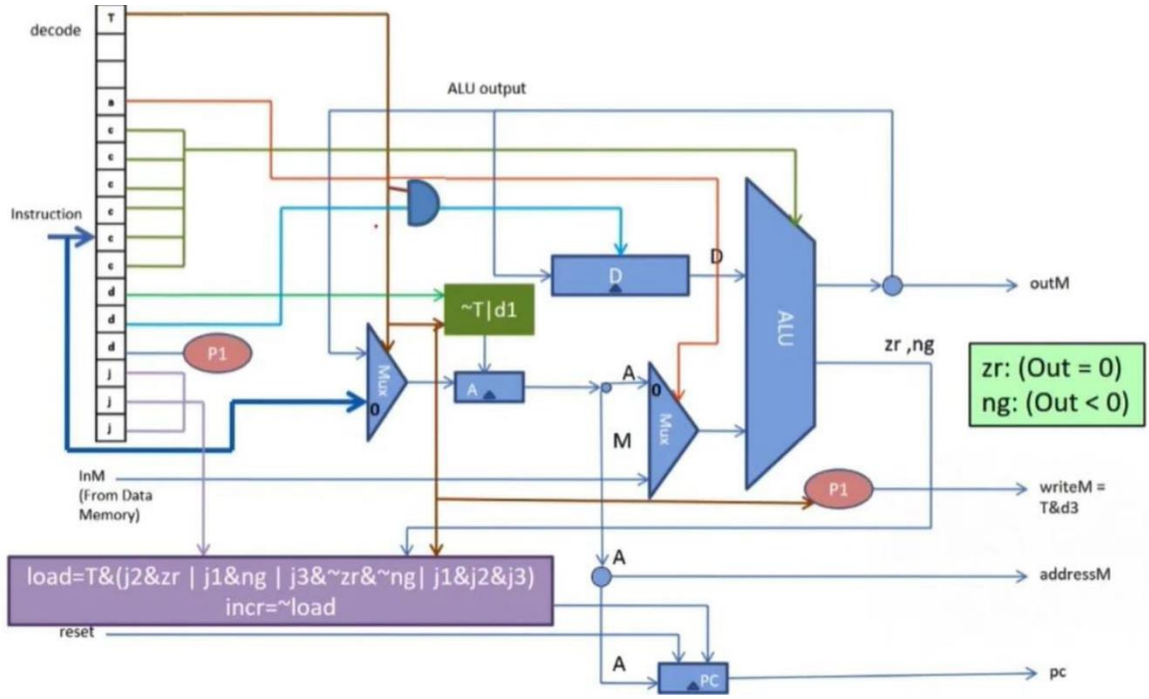


The HACK CPU has a set of registers, including the A register for general-purpose computation, the D register for data manipulation, and the program counter (PC) for keeping track of the address of the next instruction.

There are two types of instructions that it receives: A - instruction and C- instruction. According to the specified instruction, it should store the data in A - register or D - register and manipulate it in the Arithmetic Logic Unit (ALU).

There are three inputs for the CPU namely, input from Data Memory (RAM), instruction from Instruction Memory (ROM) and a reset option. Now let's look at its gate architecture. The output for the CPU is writeM (load input for RAM), outM (output of ALU) and addressM (location to get the next instruction from).

CPU structure:



Explanation

Loading into A - register

For a given set of instructions from instruction memory we must determine if it is A - instruction or C-instruction based on the opcode (instruction[15]). If instruction[15] = 0, it is an A-instruction, and its value should be stored in A - register. As for C - instruction, it stores the value in A - register depending on the destination (dest) bits.

So, a MUX16 is used to select between the instruction and the output from Arithmetic Logic Unit (ALU) with the opcode as a selection line.To load the output of the result into A - register based on the dest bits, we use ~T + d1 as load for the register, where d1 is the first dest bit.

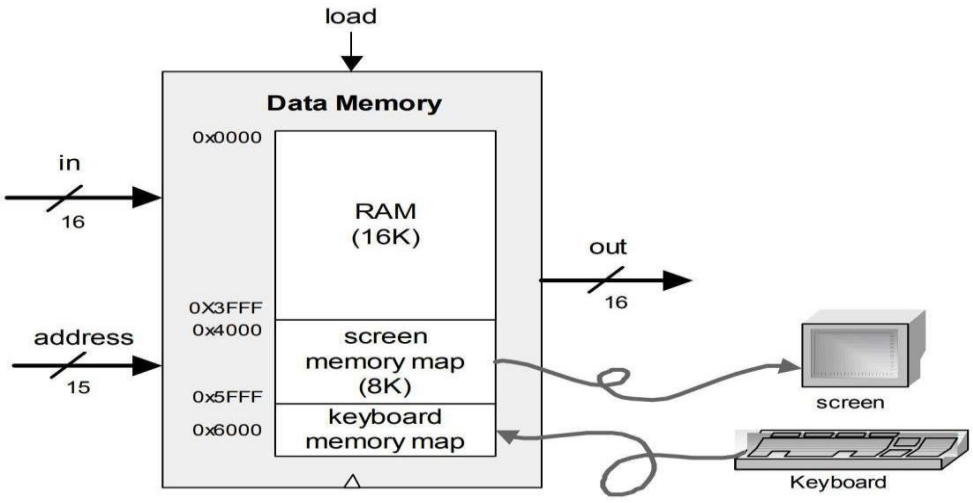
Loading into D - register

The ALU output is loaded into the D - register only if it is a C - instruction and the user specified it to store it in D - register (so the dest bit d2 should be 1). So, an And gate is used to check if it is a C - instruction and d2 is 1 and connected to the register as load.

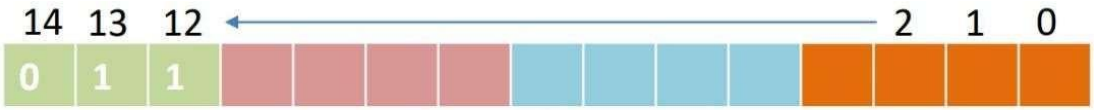
Computations in ALU

The ALU should perform the respective operation based on the given combination of computation bits c1,c2,c3,c4,c5,c6. So, these bits are used as selection lines to the ALU. The inputs to the ALU are from D - register and M/A - register. The output from the ALU is one of the required output from HACK CPU, outM.

Memory

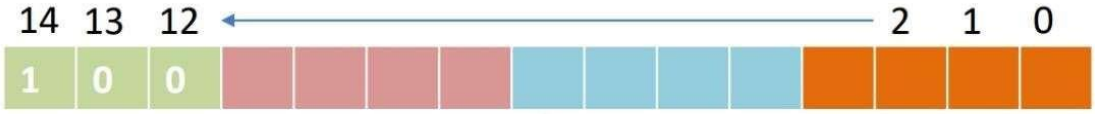


Writing into RAM:



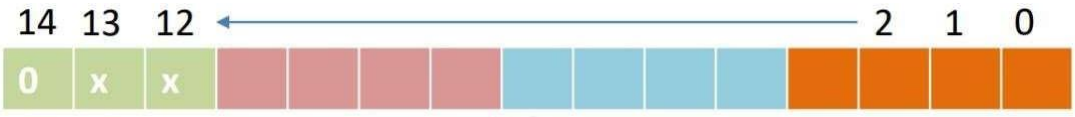
If the writeM is 1 and the 15th bit (which is addressM(14)) is 0, we should write the outM from CPU into the RAM. Where to write in memory? It will be specified by the bits from addressM(13) to addressM(0)

Writing onto screen:



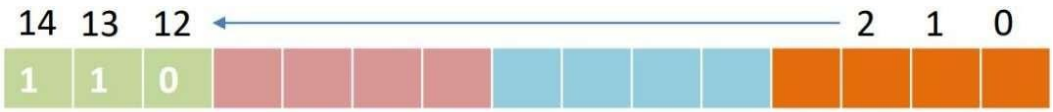
If the writeM is 1 and the 15th bit [which is addressM(14)] is 1, we should write the outM from CPU into the screen Where to write in the screen? It will be specified by the bits from addressM(12) to addressM(0)

Reading from ram:



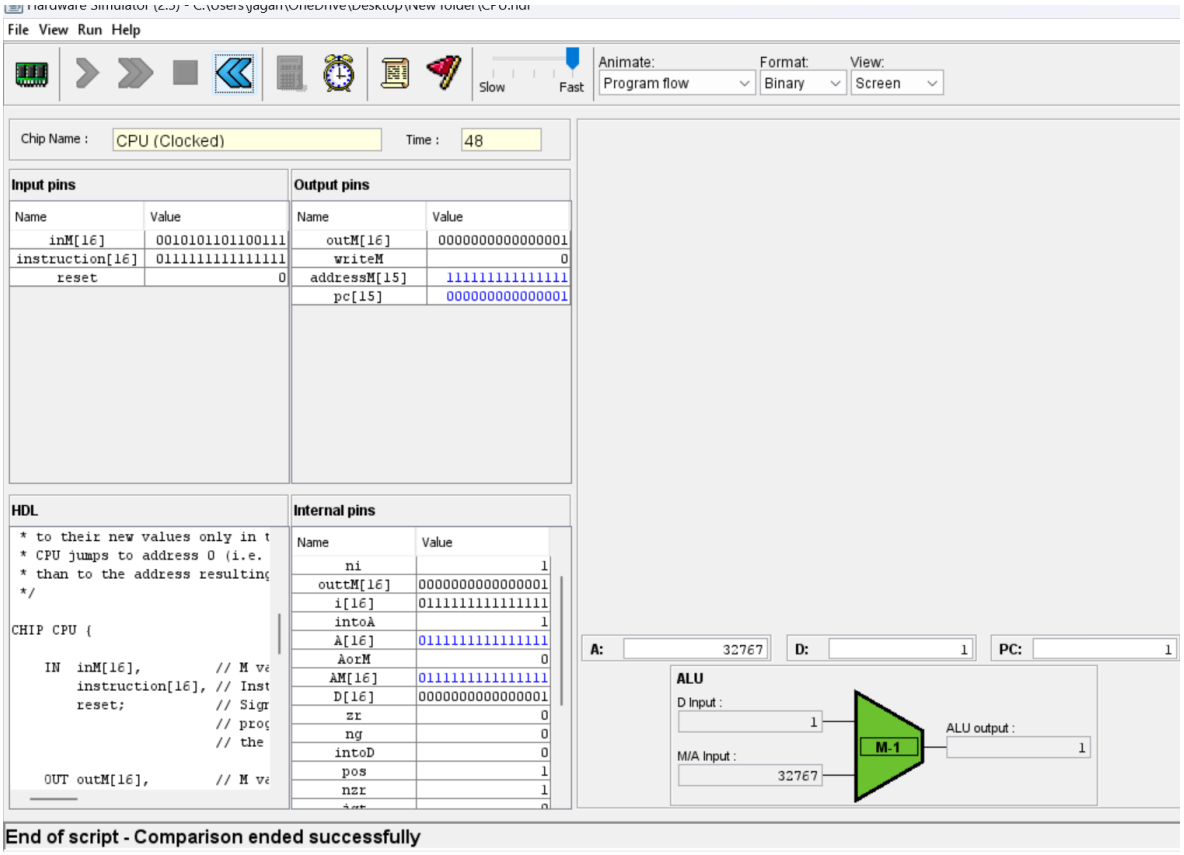
If the writeM is 0 and the 15th bit [which is addressM(14)] is 0, we should read the inM from RAM specified by bits from addressM(13) to addressM(0)

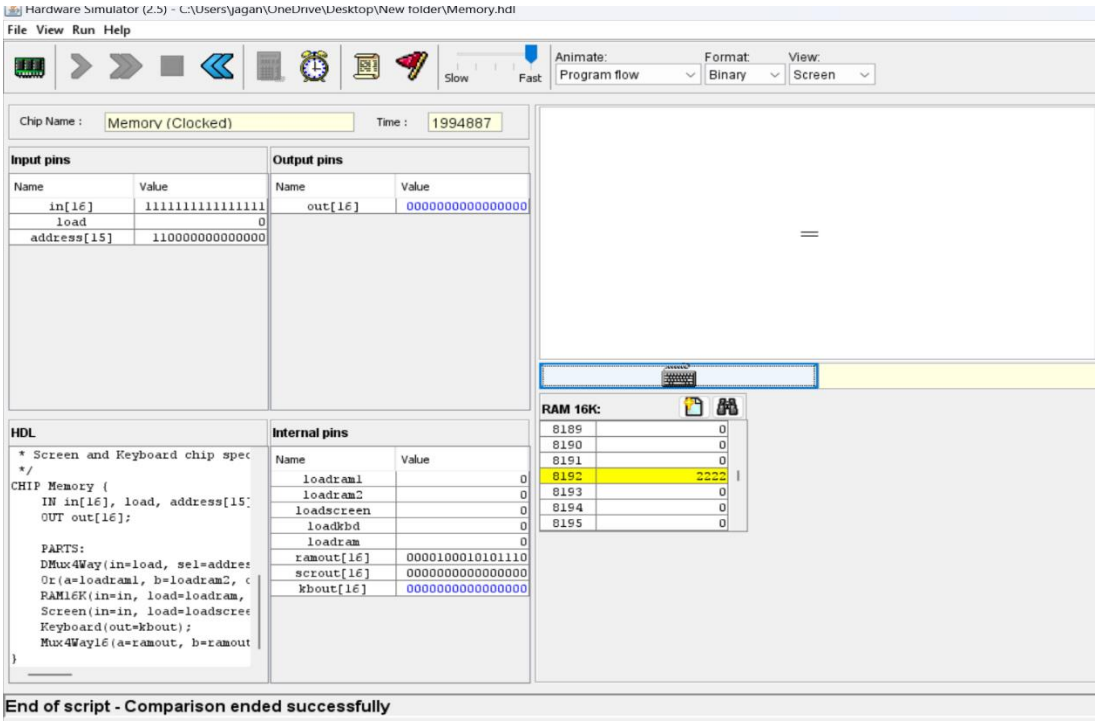
Reading from keyboard:



If the writeM is 0 and the 15th bit and 14th bit [which is addressM(14) and addressM(13)] is 1, we should read the keyboard

A.4. Results and Discussion





A.5. Applications/Scope

- Computer Architecture Courses: Universities and colleges, such as ours, widely adopt the Hack computer as a cornerstone for introductory courses. Its clear instruction set, readily visualizable data paths, and emphasis on control flow provide a hands-on grasp of these fundamental concepts.
- Online Learning Platforms: Massive Open Online Courses (MOOCs) like "Build a Modern Computer from First Principles: From Nand to Tetris" leverage the Hack computer's power. Students can build a virtual computer step-by-step, gaining practical experience in assembly language programming and hardware design.
- Research Playground: Due to its simplicity and flexibility, the Hack computer serves as a valuable platform for researchers exploring innovative CPU design ideas. Experimenting with new instructions, memory architectures, and control mechanisms becomes readily accessible.

A.6. Conclusion

In conclusion, the Hack CPU project has not only provided a practical application of theoretical knowledge but also instilled a profound appreciation for the elements of computing. The

experience gained from this project extends far beyond the technical aspects, offering insights into problem-solving, attention to detail, and the iterative nature of engineering. It leaves behind a solid foundation of knowledge and experience.

A.7. References

The elements of computing systems building a modern computer from first principles by Noam Nisan and Shimon Schocken

A.8. Appendix

```
CHIP CPU {

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

    OUT outM[16],        // M value output
        writeM,          // Write to M?
        addressM[15],    // Address in data memory (of M)
        pc[15];          // Address of next instruction

    PARTS:
    // get type of instruction
    Not(in=instruction[15], out=Ainstruction);
    Not(in=Ainstruction, out=Cinstruction);

    And(a=Cinstruction, b=instruction[5], out=ALUtoA);    // C-inst and dest to A-reg?
    Mux16(a=instruction, b=ALUout, sel=ALUtoA, out=Aregin);

    Or(a=Ainstruction, b=ALUtoA, out=loadA);    // load A if A-inst or C-inst&dest to A-reg
    ARegister(in=Aregin, load=loadA, out=Aout);

    Mux16(a=Aout, b=inM, sel=instruction[12], out=AMout);   // select A or M based on a-bit

    And(a=Cinstruction, b=instruction[4], out=loadD);
    DRegister(in=ALUout, load=loadD, out=Dout);    // load the D register from ALU

    ALU(x=Dout, y=AMout, zx=instruction[11], nx=instruction[10],
        zy=instruction[9], ny=instruction[8], f=instruction[7],
        no=instruction[6], out=ALUout, zr=ZRout, ng=NGout); // calculate

    // Set outputs for writing memory
    Or16(a=false, b=Aout, out[0..14]=addressM);
    Or16(a=false, b=ALUout, out=outM);
    And(a=Cinstruction, b=instruction[3], out=writeM);

    // calc PCload & PCinc - whether to load PC with A reg
    And(a=ZRout, b=instruction[1], out=jeq);    // is zero and jump if zero
    And(a=NGout, b=instruction[2], out=jlt);    // is neg and jump if neg
    Or(a=ZRout, b=NGout, out=zeroOrNeg);
    Not(in=zeroOrNeg, out=positive);            // is positive (not zero and not neg)
    And(a=positive, b=instruction[0], out=jgt); // is pos and jump if pos
    Or(a=jeq, b=jlt, out=jle);
    Or(a=jle, b=jgt, out=jumpToA);              // load PC if cond met and jump if cond
    And(a=Cinstruction, b=jumpToA, out=PCload); // Only jump if C instruction
    Not(in=PCload, out=PCinc);                  // only inc if not load
    PC(in=Aout, inc=PCinc, load=PCload, reset=reset, out[0..14]=pc);
}
```

```
CHIP Computer {

    IN reset;

    PARTS:
    ROM32K(address=nIns, out=cIns);
    CPU(inM=oM, instruction=cIns, reset=reset, outM=outM,writeM=wM, addressM=addM,pc=nIns);
    Memory(in=outM, load=wM, address=addM, out=oM);
}
```

```
CHIP Memory {
    IN in[16], load, address[15];
    OUT out[16];

    PARTS:
    DMux(in=load,sel=address[14],a=ramload,b=skload);
    DMux(in=skload,sel=address[13],a=sload,b=nothing);

    RAM16K(in=in,load=ramload,address=address[0..13],out=ramout);
    Screen(in=in,load=sload,address=address[0..12],out=screenout);

    Keyboard(out=kbd);
    Or8Way(in=address[0..7], out=notkbd1);
    Or8Way(in[0..4]=address[8..12], in[5..7]=false, out=notkbd2);
    Or(a=notkbd1,b=notkbd2,out=notkbd);
    Mux16(a=kbd,b=false,sel=notkbd,out=kbdout);

    Mux16(a=ramout,b=outsk,sel=address[14],out=out);
    Mux16(a=screenout,b=kbdout,sel=address[13],out=outsk);

}
```

```
CHIP ALU {
    IN
        x[16], y[16], zx, nx, zy, ny, f, no;

    OUT
        out[16], zr, ng;

    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);
}
```

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

    PARTS:
    Inc16(in=o,out=o1);
    Mux16(a=o,b=o1,sel=inc,out=o2);
    Mux16(a=o2,b=in,sel=load,out=o3);
    Mux16(a=o3,b=false,sel=reset,out=o4);
    Or(a=load,b=inc,out=or1);
    Or(a=or1,b=reset,out=or3);
    Register(in=o4,load=or3,out=out,out=o);
}
```

```
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]);
}
```

Part B: The 2 Bit Serial Binary Adder

B.1. Introduction

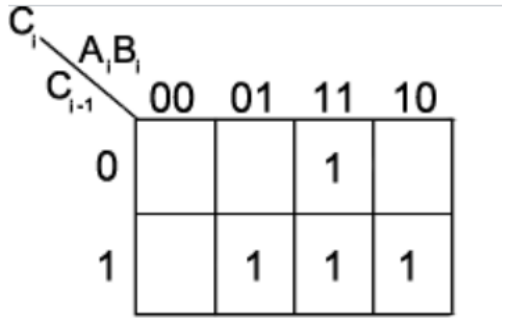
Serial binary adder is a sequential logic circuit that performs the addition of two binary numbers in serial form. Serial binary adder performs bit by bit addition. A single full adder is used to add one pair of bits at a time along with the carry. The carry output from the full adder is applied to a flip-flop. After that output is used as carry for the next significant bits.

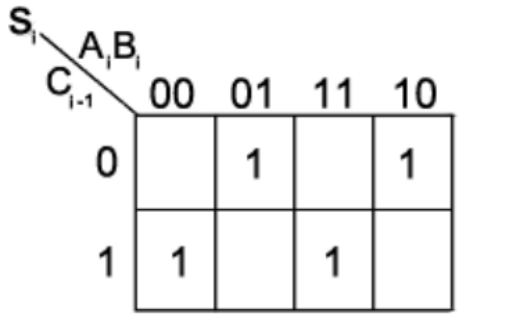
B.2. Project Objectives

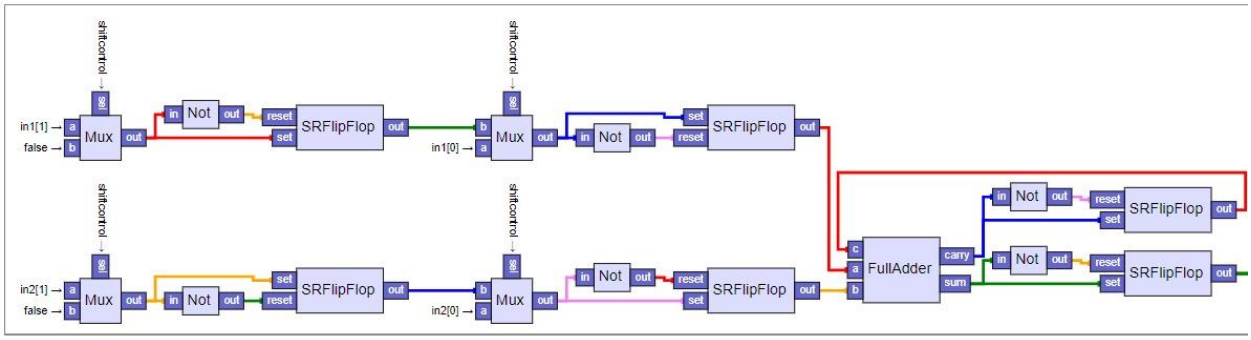
The objective of Project Part B is to study the design and structure of a 2-bit serial binary adder and how it can be programmed/implemented in hardware description language.

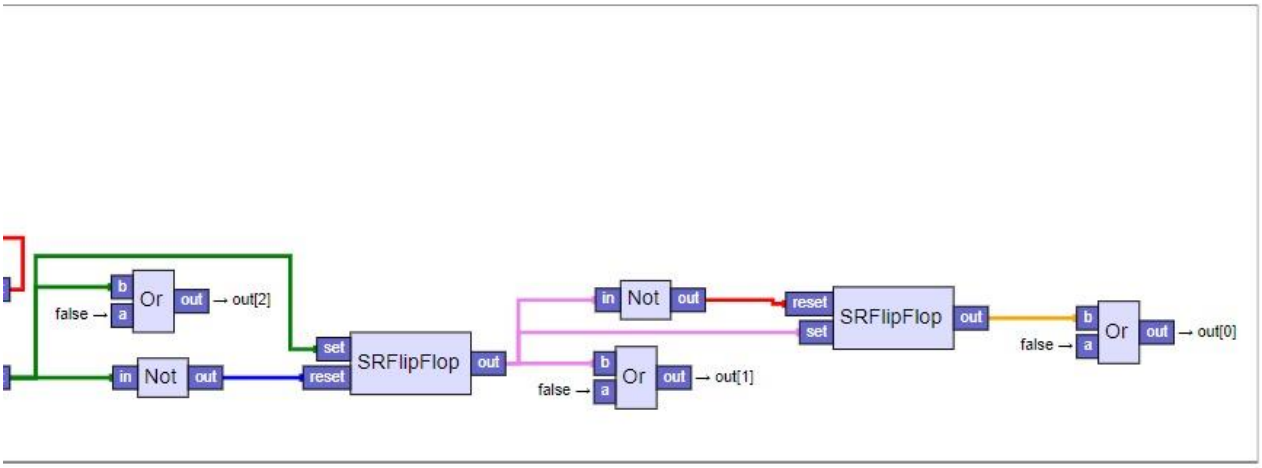
B.3. Design/Methodology

K Map for full adder









Following our project guidelines, we have used only SR flip flops for data storage. This was done by conversion of D flip flops, the standard data storage device, into SR flip flops, by connecting the input to the S pin of the flip flop and the negation of the input to the R pin of the flip flop, by using a NOT gate in between the two.

The first step is to store the second bits of the two inputs inside the SR flip flops so that they can be used in the future as the inputs for operations, such as addition for sum. This is done by connecting the second bits to 2 to 1 multiplexer, with the data lines being the bits and a false value, and load as the shift control. When the shift control is zero in the first clock cycle, the bits are selected, and then stored in their respective SR flip flops.

The next step is to decide whether to shift the bits or not. In this step, like the previous step, 2 to 1 multiplexers are used to decide between storing the first bits or the second bits which are currently stored in the previous set of SR flip flops, in the next set of SR flip flops. Since, initially, the shift control is zero, and it is the load for the 2 to 1 multiplexer, the first bits of the input are selected and thus stored in the SR flip flops. These two bits are then inputted into a full adder, as well as a carry bit, which is initially zero for the first clock cycle, and will be different for the next clock cycles. The carry that will be inputted for the next clock cycles will be discussed in the next step.

After the full adder performs its operations, the output sum is stored in an SR flip flop, and the carry is directed to an SR flip flop, where it will be stored, and then used as the input carry for the next clock cycle.

After the sum is stored in the SR flip flop, it is also sent to an OR gate, where it is in disjunction with a false value, whose equivalent is the sum itself, and is then used as the last bit of the chip output, which is three bits long to account for overflow.

The sum, after being stored in the first SR flip flop, while being sent to the OR gate, is sent to another SR flip flop, in which the operation of the previous step is repeated, except it is used as the middle bit of the chip output.

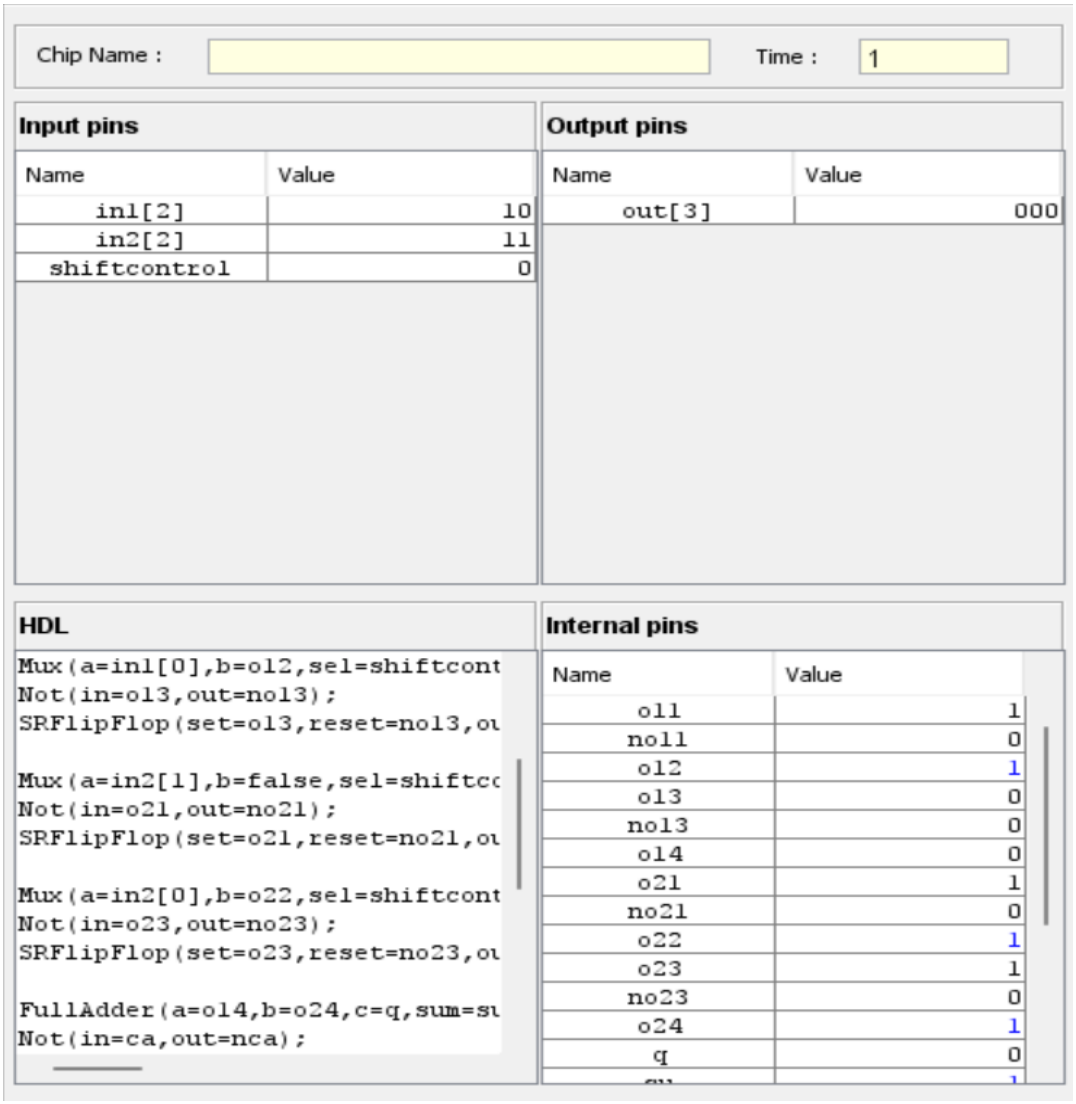
Lastly, the sum bit, after passing through and being stored in two SR flip flops, is directed to an OR gate which is used for the same purpose, except for the first bit of the chip output.

B.4. Results and Discussion

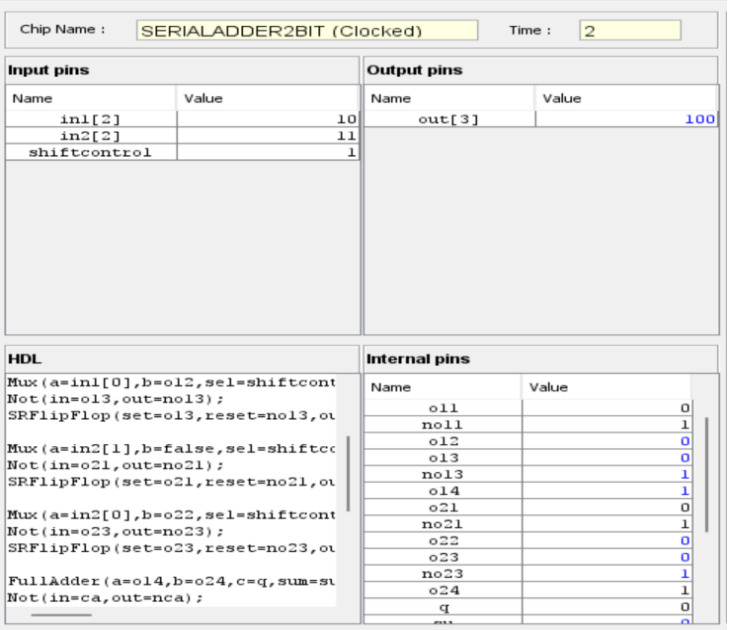
Output

Two inputs-value of two 2-bit numbers for which the sum is to be calculated.

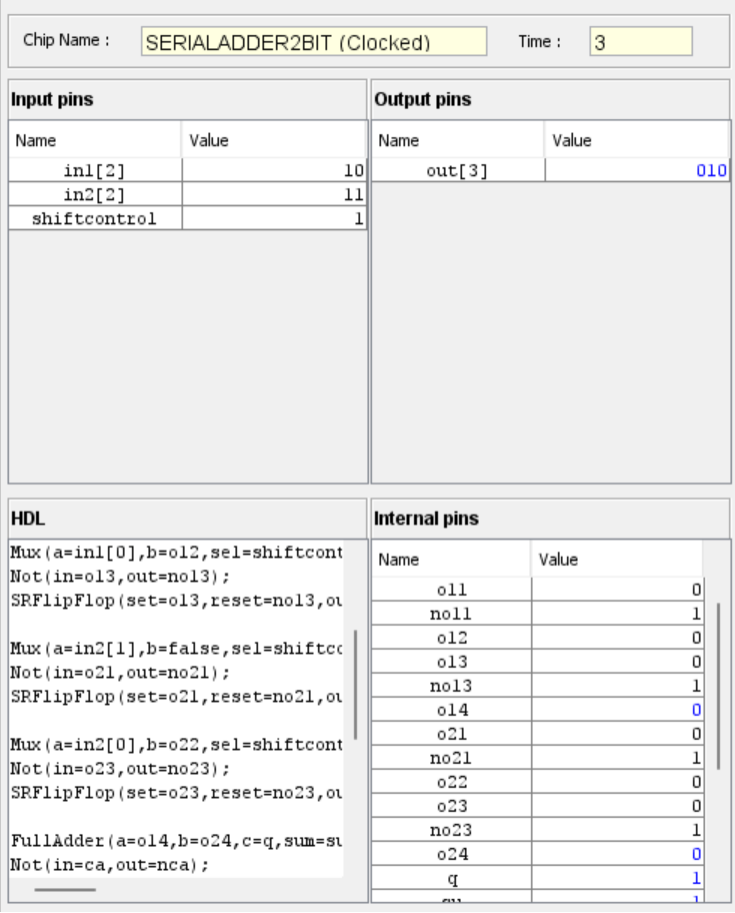
3-bit output which includes the carry as the most significant bit

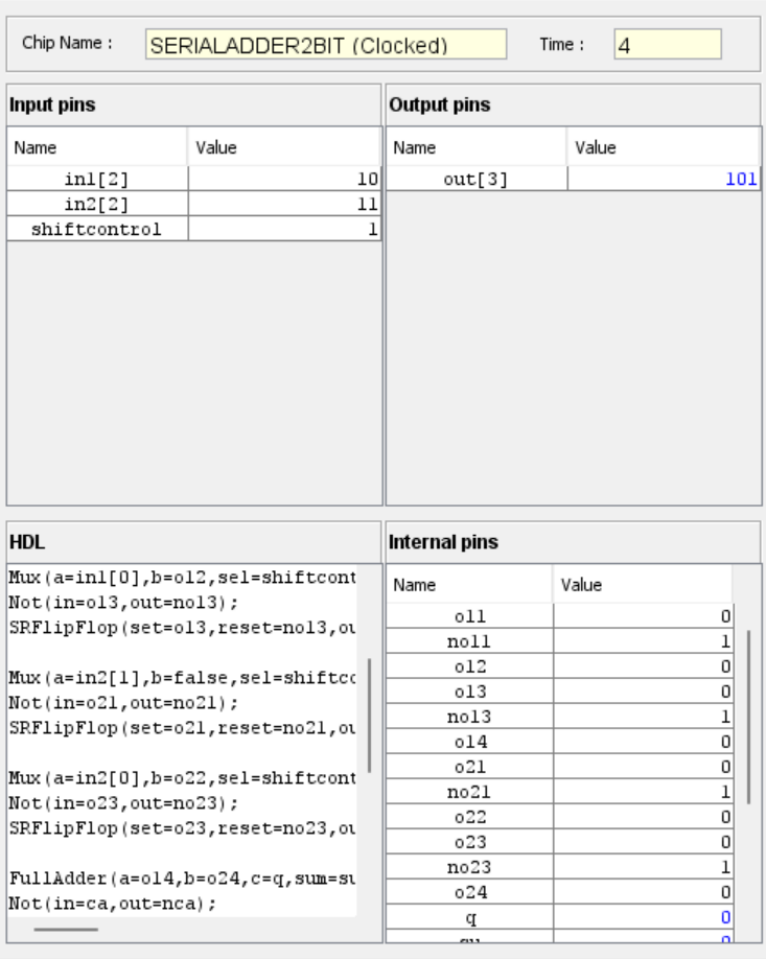


We enter the value (2 bit) of in 1 and in2 for which we calculate the sum and carry, we also set the shift control to 0 for the first clock for loading the in1 and in2 bits parallelly.



After the 1st clock cycle, we change the shift control to 1 to make the input values to traverse the circuit serially.





We notice the 3-bit output updates every clock cycle.

It takes totally 4 clock cycles to get the output-

1st clock the in value is loaded

2nd clock to 4th clocks the input value traverses into the circuit and finally 3-bit output is obtained.

B.5. Applications/Scope

Arithmetic Operations in Circuits with Limited Hardware:

- Calculators: Basic calculators often use serial adders to perform addition, subtraction, multiplication, and division due to their lower hardware requirements.
- Microcontrollers and Embedded Systems: Serial adders are common in low-power devices with constrained chip space, as they occupy fewer logic gates compared to parallel adders.

Communication Systems:

- Error Correction Codes (ECC): Serial adders are employed in calculating checksums and cyclic redundancy checks (CRC) for error detection and correction in data transmission.
- Modems and Data Transmission: They play a role in modems and other communication devices to process serial data streams.

B.6. Conclusion

B.7. References

https://www.geeksforgeeks.org/sr-flip-flop/

https://www.geeksforgeeks.org/siso-shift-register/

https://www.geeksforgeeks.org/serial-binary-adder-in-digital-logic/

The elements of computing systems building a modern computer from first principles by Nisan, Noam and Schocken, Shimon

B.8. Appendix

```
CHIP SRFlipFlop{
    IN set, reset;
    OUT out;

    PARTS:
    Not(in=reset,out=nR);
    And(a=nR,b=dffOut,out=a1);
    Or(a=set,b=a1,out=o1);
    DFF(in=o1,out=dffOut,out=out);
}
```

```
CHIP SerialAdder2Bit {
    IN in1[2],in2[2],shiftcontrol;
    OUT out[3];
PARTS:
    Mux(a=in1[1],b=false,sel=shiftcontrol,out=o11);
    Not(in=o11,out=no11);
    SRFlipFlop(set=o11,reset=no11,out=o12);

    Mux(a=in1[0],b=o12,sel=shiftcontrol,out=o13);
    Not(in=o13,out=no13);
    SRFlipFlop(set=o13,reset=no13,out=o14);

    Mux(a=in2[1],b=false,sel=shiftcontrol,out=o21);
    Not(in=o21,out=no21);
    SRFlipFlop(set=o21,reset=no21,out=o22);

    Mux(a=in2[0],b=o22,sel=shiftcontrol,out=o23);
    Not(in=o23,out=no23);
    SRFlipFlop(set=o23,reset=no23,out=o24);

    FullAdder(a=o14,b=o24,c=q,sum=su,carry=ca);
    Not(in=ca,out=nca);
    SRFlipFlop(set=ca,reset=nca,out=q);

    Not(in=su,out=nsu);
    SRFlipFlop(set=su,reset=nsu,out=o2);
    Or(a=false,b=o2,out=out[2]);

    Not(in=o2,out=no2);
    SRFlipFlop(set=o2,reset=no2,out=o1);
    Or(a=false,b=o1,out=out[1]);

    Not(in=o1,out=no1);
    SRFlipFlop(set=o1,reset=no1,out=o0);
    Or(a=false,b=o0,out=out[0]);
}
```