I was assigned to do a 16 bits Project.

All the following components are designed accordingly.

Instructions format

My Instruction formats are of R (Register), I (Immediate), and J (Jump) type.

R-Type instructions are used for arithmetic and logical operations. The fields are defined as follows:

Op (4 bit)	Rs (3 bit)	Rt (3 bit)	Rd (3 bit)	Shamt (3 bit)

Op: 4-bit opcode specifying the operation (e.g., add, sub, etc.).

Rs: 3-bit source register specifier.

Rt: 3-bit target register specifier.

Rd: 3-bit destination register specifier.

Shamt: 3-bit shift amount (used for shift operations).

I-Type instructions are used for operations that involve immediate values and for data transfer instructions. The fields are:

Op (4 bit)	Rs (3 bit)	Rt (3 bit)	Immediate (6 bit)

Op: 4-bit opcode specifying the operation (e.g., addi, lw, sw, etc.).

Rs: 3-bit base register specifier.

Rt: 3-bit target register specifier.

Immediate: 4-bit immediate operand.

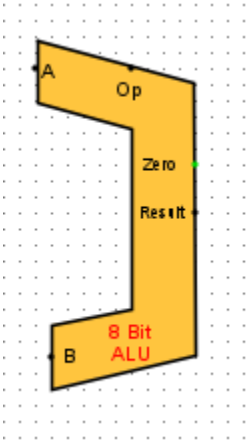
J-Type instructions are used for jump operations. The fields are:

Op (4 bit)	Target (12 bit)

Op: 4-bit opcode specifying the jump operation (e.g., j, jal, etc.).

Target: 12-bit address to jump to.

ALU



An arithmetic-logic unit is the part of a central processing unit that carries out arithmetic and logic operations on the operands in computer instruction words. My Project's ALU have ADD, ADDI, SUB, INC(A), SHIFTLEFT, SHIFTRIGHT and XOR operations.

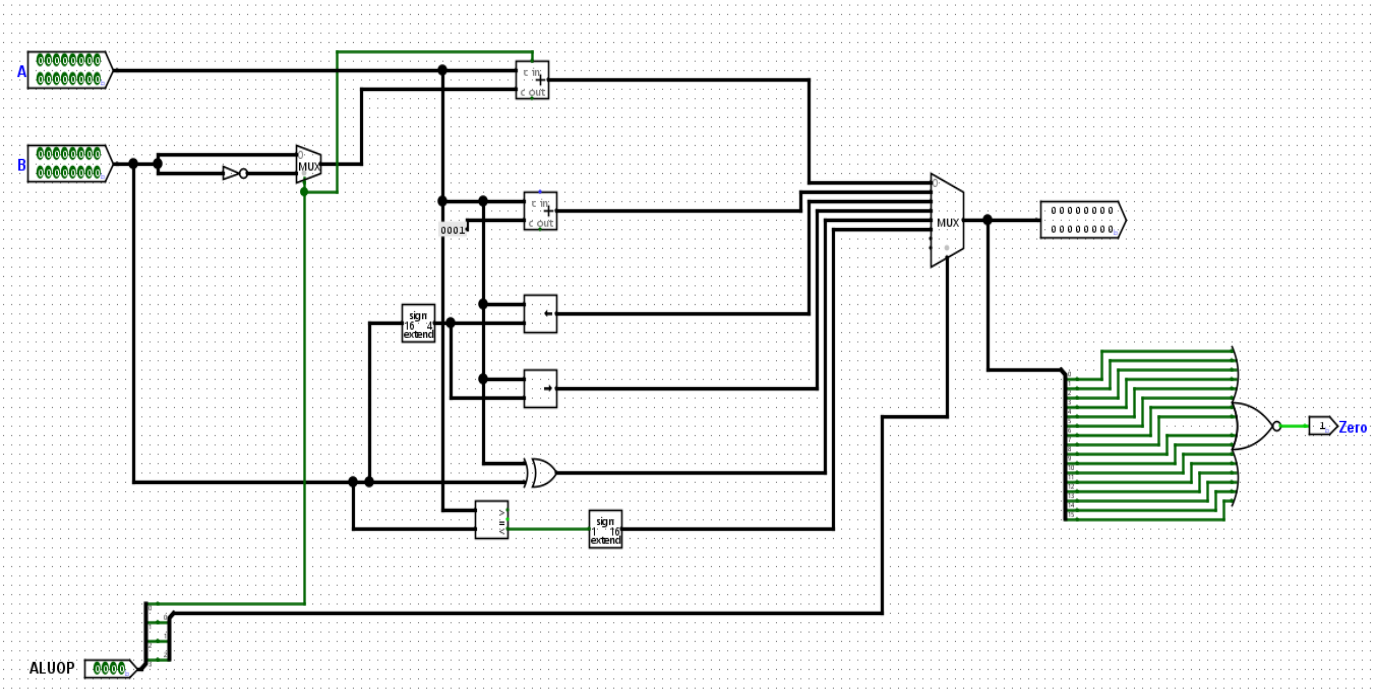
My Input and Output bits are 16.

ALUOP	Operation
0 0 0 0	ADD
0 0 0 1	SUB
0 0 1 0	INC_A
0 1 0 0	SHIFTLEFT_A
0 1 1 0	SHIFTRIGHT_A
1 0 0 0	XOR
1 0 1 0	SLT

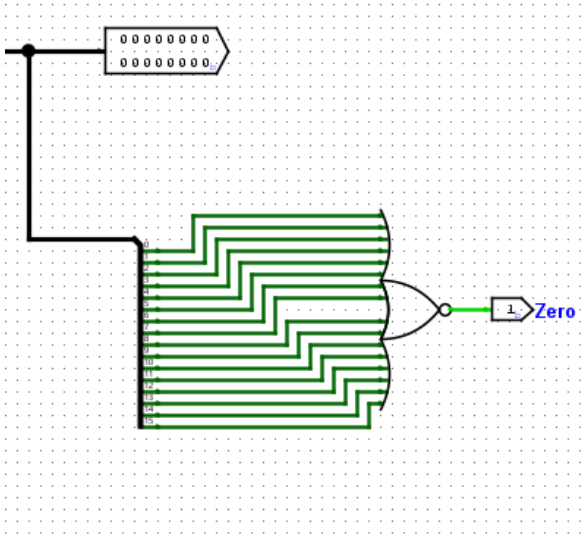
For the SUB part. The Cin to adder will be 1 and B will be inverted with the help of a MUX.

So, the output will be A + ¬B + 1. B is complemented and added with A and 1 is added. So overall operation becomes A – B.

ALUOP has 4 bits. The LSB is for choosing whether to add or subtract. The rest of the 3 bits are for MUX selection which chooses what operation to perform.



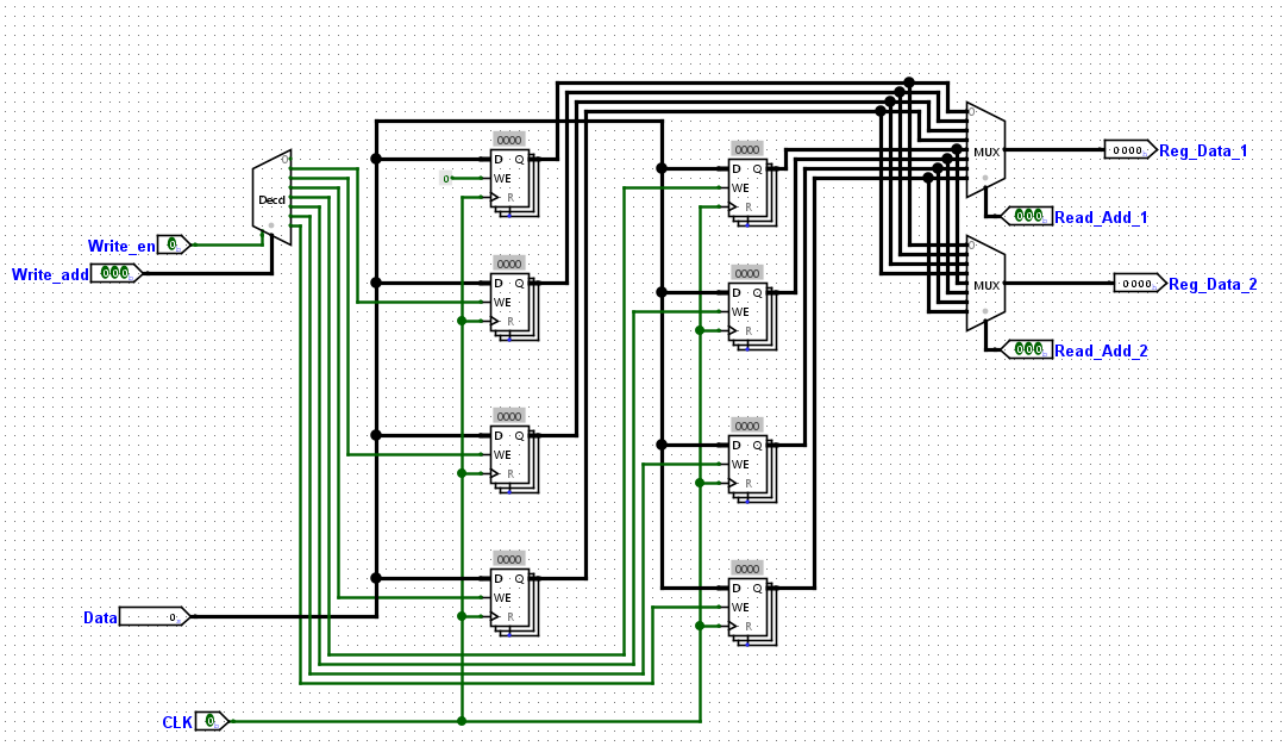
I also implemented a **Zero** Logic, which basically takes the **NOR** of all the bits of the output. If all bits are zero, output is one, which means output is **zero.**



Register File

A register file is a small set of high-speed storage cells inside the CPU. There are special-purpose registers such as the IR and PC, and also general-purpose registers for storing operands of instructions such as add, sub etc. Since there are very few registers compared to memory cells, registers also require far fewer bits to specify which register to use.

I used 8-bit Register File for my project. As my RS RT and RD are 3 bits. $2^3 = 8$ bits register file would suffice.



Register File

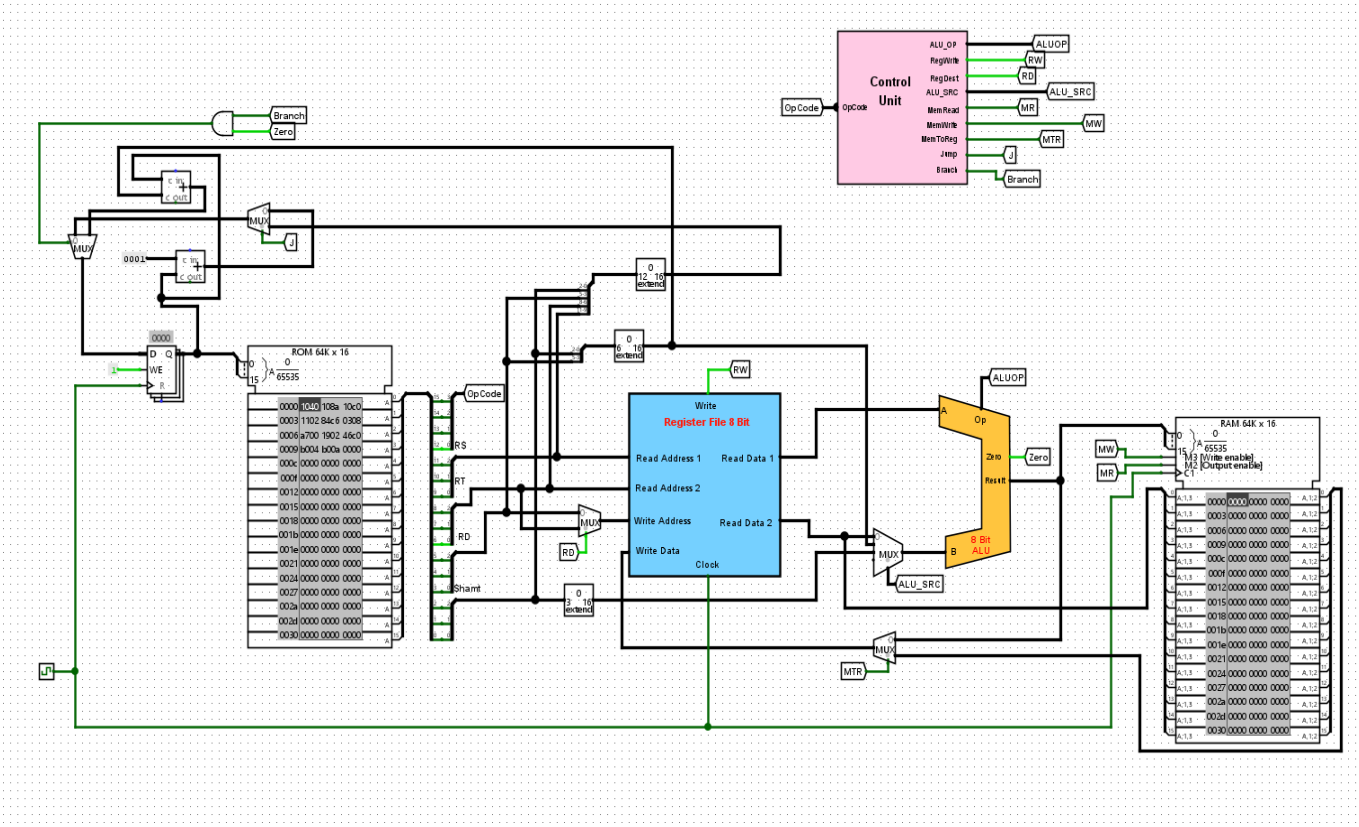
Datapath

The Datapath is a crucial component of a CPU, playing a key role in executing the fetch-decode-execute cycle. The process for designing a Datapath typically involves the following steps:

1. Identify the different types and formats of instructions specified.
2. Create components of the Datapath and establish connections between them to accommodate each type or format of instruction.
3. Combine the Datapath segments from Step 2 to form a comprehensive Datapath that can handle multiple instructions.

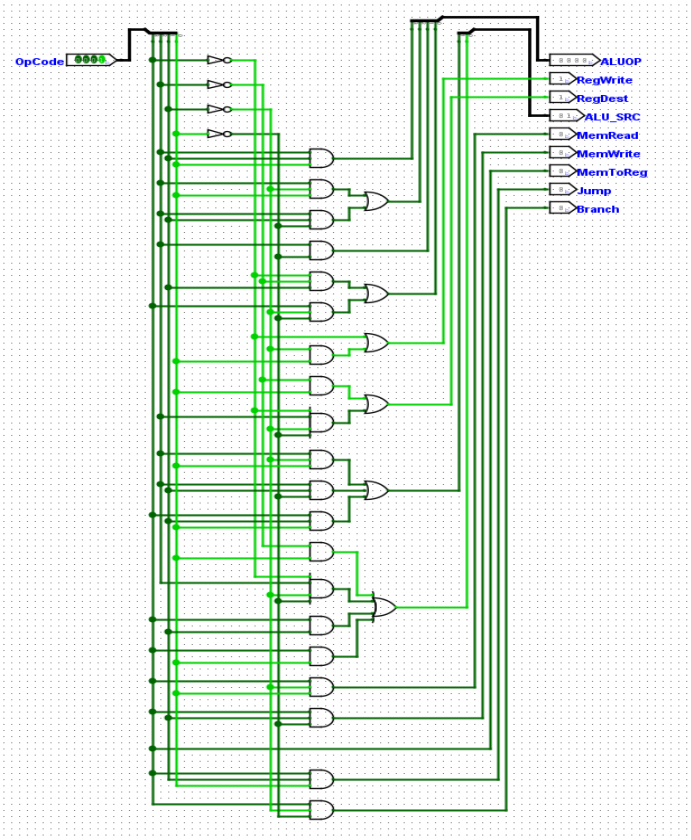
Some basic components of a simple Datapath include memory, which holds the current instruction; the program counter (PC), which keeps track of the address of the ongoing instruction; and the Arithmetic Logic Unit (ALU), which carries out the instruction. Connecting these fundamental components together establishes a rudimentary Datapath.

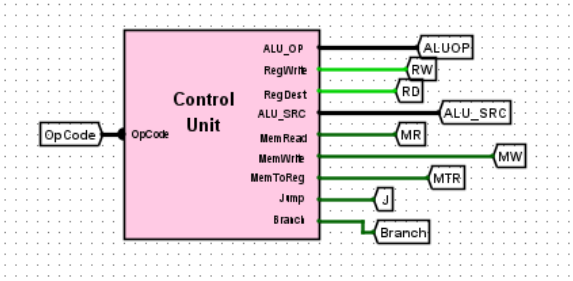
My Datapath can handle I type, R type and J type instructions. I designed it in such a way that it can carry out instructions such as add, sub, jump, load, store branch etc. I utilized the Register File and ALU that I described earlier to design my Datapath and overall circuit.



Control unit

Operation	Op Code	ALU Operation	ALU-OP	Reg Write	Reg Dest	ALU SRC	Mem Read	Mem Write	Mem To Reg	Jump	Branch
ADD	0000	Add	0000	1	0	00	0	0	0	0	0
ADDI	0001	Add	0000	1	1	01	0	0	0	0	0
SUB	0010	Sub	0001	1	0	00	0	0	0	0	0
SUBI	0011	Sub	0001	1	1	01	0	0	0	0	0
INC_A	0100	INC_A	0010	1	1	01	0	0	0	0	0
SL	0101	ShiftLeft	0100	1	0	10	0	0	0	0	0
SR	0110	ShiftRight	0110	1	0	10	0	0	0	0	0
XOR	0111	XOR	1000	1	0	00	0	0	0	0	0
BEQ	1000	Sub	0001	0	x	00	0	0	x	0	1
LW	1001	Add	0000	1	1	01	1	0	1	0	0
SW	1010	Add	0000	0	0	01	0	1	x	0	0
Jump	1011	xxx	xxxx	0	x	xx	0	0	x	1	0
SLT	1100	SLT	1010	1	0	00	0	0	0	0	0





Block Diagram

Logic Diagram

Given problems C version then assembly

Q. Store first 10 Even numbers in memory and their sum on $R1 using loop.

C Code

```c
#include <stdio.h>

int main()
{
    int sum = 0;
    int n = 10;
    int i = 0;
    int val = 2;
    int arr[10];
    while (i != n)
    {
        sum += val;
        arr[i] = val;
        val += 2;
        i++;
    }
    printf("%d\n", sum);
}
```

Assembly

#		Instruction	Instruction Type	Binary	Hexa
0	sum - > r1 = 0	addi r0 r1 0	**I**	0001 000 001 000 000	1040
1	n - > r2 = 10	addi r0 r2 10	**I**	0001 000 010 001 010	108a
2	i - > r3 = 0	addi r0 r3 0	**I**	0001 000 011 000 000	10c0
3	val - > r4 = 2	addi r0 r4 2	**I**	0001 000 100 000 010	1102
4	while (i ! = n)	beq r2 r3 10	**I**	1000 010 011 000 110	84c6
5	sum += val;	add r1 r4 r1	**R**	0000 001 100 001 000	0308
6	mem[r3 + 0] = r4	sw r3 r4 0	**I**	1010 011 100 000 000	a700
7	val = val + 2;	addi r4 r4 2	**I**	0001 100 100 000 010	1902

8	i++;	inc r3	**I**	0100 011 011 000 000	46c0
9		jmp 4	**J**	1011 000 000 000 100	b004
10		jmp 10	**J**	1011 000 000 001 010	b00a

Conclusion

In conclusion, this project provided an invaluable opportunity to dive deeply into the world of computer architecture and digital design. By designing a 16-bit processor, I gained comprehensive understanding of the core operations and functionalities within a processor's instruction set. Moreover, the task of creating compiled assembly codes enabled me to test the processor's capabilities thoroughly and assured me of the effectiveness of my design. This was a good project to implement all that I've learned in this course.