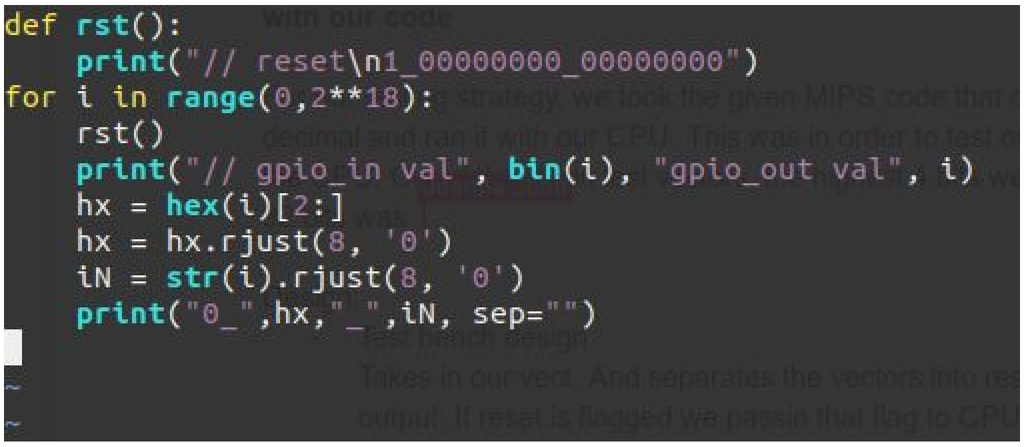
TESTING STRATEGY
For our testing strategy, we took the given MIPS code that converts numbers from binary to decimal and ran it with our CPU. This was in order to test our implemented MIPS instructions in the CPU. Our test vectors are 68 bits long, the highest 4 bits were used to flag the reset; the lowest 32 bits was used for the expected output from our CPU; and the rest of the 32 bits in the middle are the GPIO in. Then the input test vectors were generated from the python code shown below in the Justification section. With this strategy, our program can have any number [0,2^18) as input. We tested all possible numbers in this range.

When we designed the test bench, we did it to take in our vectors and separate them into parts: reset, GPIO in, and GPIO expected output. If reset is flagged we pass in that flag to the CPU and wait 10 nanoseconds. This 10 nanosecond wait is to give the CPU time to process the flag and complete the reset command. If reset is not flagged then we move on to the next 32 bits, the GPIO in. We pass that in and wait 700 nanoseconds. This 700 nanosecond wait is because that is the amount of time our MIPS code takes to convert the numbers from binary to decimal. We found this run time by running one test vector and recording the time. Since each test vector is running the same code we determined that this time frame can be used as the limit for all test vectors that we run . Then the test bench checks if the CPU output is equivalent to the GPIO in expected output. If the CPU output and expected output were not equal we print out the CPU output, expected output and that there was an error resulting in an unexpected output. Along with this we print which vector resulted in giving the error.

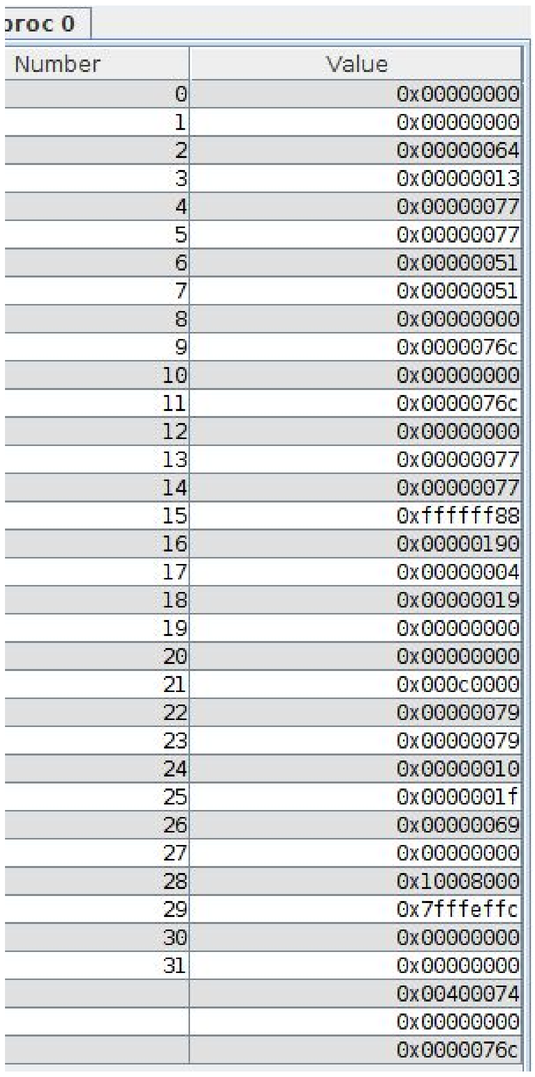
For the CPU design, it takes in the clock and reset signals as well as the GPIO in and returns GPIO out. The CPU then initializes the ALU module as well as a register file module. We implemented the control unit inside the CPU. IT then loads the given MIPS code into the CPU memory to be compared to the CPU output later. Starting at the top, the MIPS code is executed one instruction at a time. The total of these instructions are run for 700 nanoseconds and the resulting output is compared with the expected output.

JUSTIFICATION
Python3 code used to generate test vectors, file name gen218.py

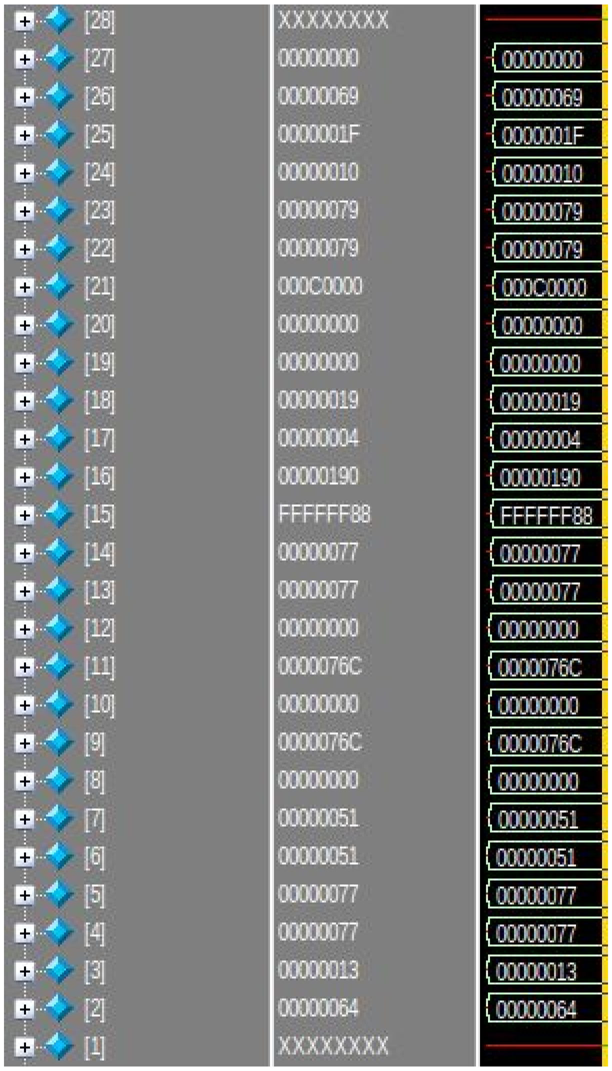


```
def rst():
    print("// reset\n1_00000000_00000000")
for i in range(0,2**18):
    rst()
    print("// gpio_in val", bin(i), "gpio_out val", i)
    hx = hex(i)[2:]
    hx = hx.rjust(8, '0')
    iN = str(i).rjust(8, '0')
    print("0_",hx,"_",iN, sep="")

~
~
```


roc 0

Number	Value
0	0x00000000
1	0x00000000
2	0x00000064
3	0x00000013
4	0x00000077
5	0x00000077
6	0x00000051
7	0x00000051
8	0x00000000
9	0x0000076c
10	0x00000000
11	0x0000076c
12	0x00000000
13	0x00000077
14	0x00000077
15	0xffffff88
16	0x00000190
17	0x00000004
18	0x00000019
19	0x00000000
20	0x00000000
21	0x000c0000
22	0x00000079
23	0x00000079
24	0x00000010
25	0x0000001f
26	0x00000069
27	0x00000000
28	0x10008000
29	0x7fffeffc
30	0x00000000
31	0x00000000
	0x00400074
	0x00000000
	0x0000076c



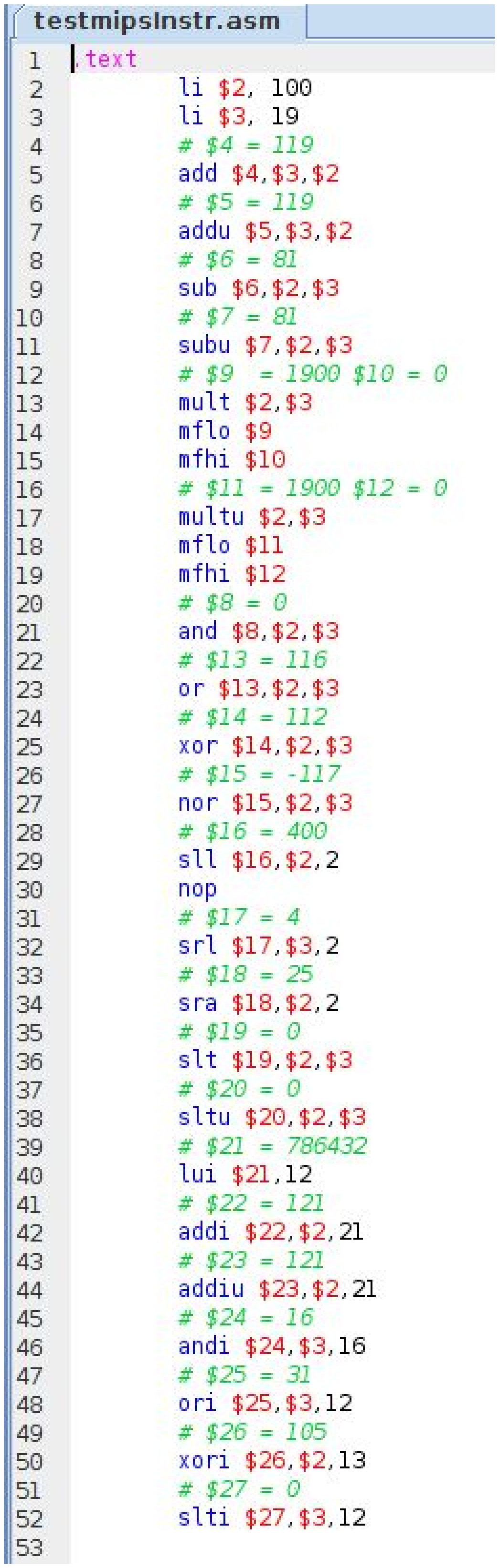
[28]	XXXXXXXX	
[27]	00000000	00000000
[26]	00000069	00000069
[25]	0000001F	0000001F
[24]	00000010	00000010
[23]	00000079	00000079
[22]	00000079	00000079
[21]	000C0000	000C0000
[20]	00000000	00000000
[19]	00000000	00000000
[18]	00000019	00000019
[17]	00000004	00000004
[16]	00000190	00000190
[15]	FFFFFF88	FFFFFF88
[14]	00000077	00000077
[13]	00000077	00000077
[12]	00000000	00000000
[11]	0000076C	0000076C
[10]	00000000	00000000
[9]	0000076C	0000076C
[8]	00000000	00000000
[7]	00000051	00000051
[6]	00000051	00000051
[5]	00000077	00000077
[4]	00000077	00000077
[3]	00000013	00000013
[2]	00000064	00000064
[1]	XXXXXXXX	

ANALYSIS

Our implemented design included testing our binary to decimal MIPS program (from the CPU) with all the possible inputs. This refers to all the numerical values within the range of [0-2^18). To test the accuracy of the CPU output, we then went through and tested each of the instructions in the given MIPS code(in Justification section above) and compared the two outputs. Since all of these test cases give the expected output our design works.

Below Listed is the mips code used to test mips instructions. And some of test vectors used to test the cpu running mips bin2dec program. The rest of the test vectors are in the vectors.dat file in the packed source.

testmipsInstr.asm turned into hexcode file named testmipsInstr.dat



```
.text
        li $2, 100
        li $3, 19
        # $4 = 119
        add $4,$3,$2
        # $5 = 119
        addu $5,$3,$2
        # $6 = 81
        sub $6,$2,$3
        # $7 = 81
        subu $7,$2,$3
        # $9  = 1900 $10 = 0
        mult $2,$3
        mflo $9
        mfhi $10
        # $11 = 1900 $12 = 0
        multu $2,$3
        mflo $11
        mfhi $12
        # $8 = 0
        and $8,$2,$3
        # $13 = 116
        or $13,$2,$3
        # $14 = 112
        xor $14,$2,$3
        # $15 = -117
        nor $15,$2,$3
        # $16 = 400
        sll $16,$2,2
        nop
        # $17 = 4
        srl $17,$3,2
        # $18 = 25
        sra $18,$2,2
        # $19 = 0
        slt $19,$2,$3
        # $20 = 0
        sltu $20,$2,$3
        # $21 = 786432
        lui $21,12
        # $22 = 121
        addi $22,$2,21
        # $23 = 121
        addiu $23,$2,21
        # $24 = 16
        andi $24,$3,16
        # $25 = 31
        ori $25,$3,12
        # $26 = 105
        xori $26,$2,13
        # $27 = 0
        slti $27,$3,12

```

```
// reset
1_00000000_00000000
// gpio_in val 0b0 gpio_out val 0
0_00000000_00000000
// reset
1_00000000_00000000
// gpio_in val 0b1 gpio_out val 1
0_00000001_00000001
// reset
1_00000000_00000000
// gpio_in val 0b10 gpio_out val 2
0_00000002_00000002
// reset
1_00000000_00000000
// gpio_in val 0b11 gpio_out val 3
0_00000003_00000003
// reset
1_00000000_00000000
// gpio_in val 0b100 gpio_out val 4
0_00000004_00000004
// reset
1_00000000_00000000
// gpio_in val 0b101 gpio_out val 5
0_00000005_00000005
// reset
1_00000000_00000000
// gpio_in val 0b110 gpio_out val 6
0_00000006_00000006
// reset
1_00000000_00000000
// gpio_in val 0b111 gpio_out val 7
0_00000007_00000007
// reset
1_00000000_00000000
// gpio_in val 0b1000 gpio_out val 8
0_00000008_00000008
// reset
1_00000000_00000000
// gpio_in val 0b1001 gpio_out val 9
0_00000009_00000009
```

```
// reset
1_00000000_00000000
// gpio_in val 0b1010 gpio_out val 10
0_0000000a_00000010
```