# CSE 224 – Introduction to Digital Systems

# Term Project

1- Based on the VSCPU instruction set given at cpu.tc, complete all the instructions by extending the VSCPU Verilog code (with interrupt handling) that I started in the class.

2- Write a VSCPU assembly code that implements floor division. Hint: Division can be implemented with a series of subtraction. For subtraction use two's complete representation.

3- Use the simulator at cpu.tc to generate machine code. Then test your VSCPU Verilog code with the machine code for floor division.

4- Prepare a report and give details of the steps of the project.

5- Synthesize your VSCPU. Note the timing and the area synthesis results.

6- Submit a 1- report with synthesis results, 2- your VSCPU design and 3- testbench Verilog codes, 4- floor division VSCPU assembly code, and 5- an edaplayground project link.


## Name: Elif

## Surname: Çolak

## Number: 20200702023

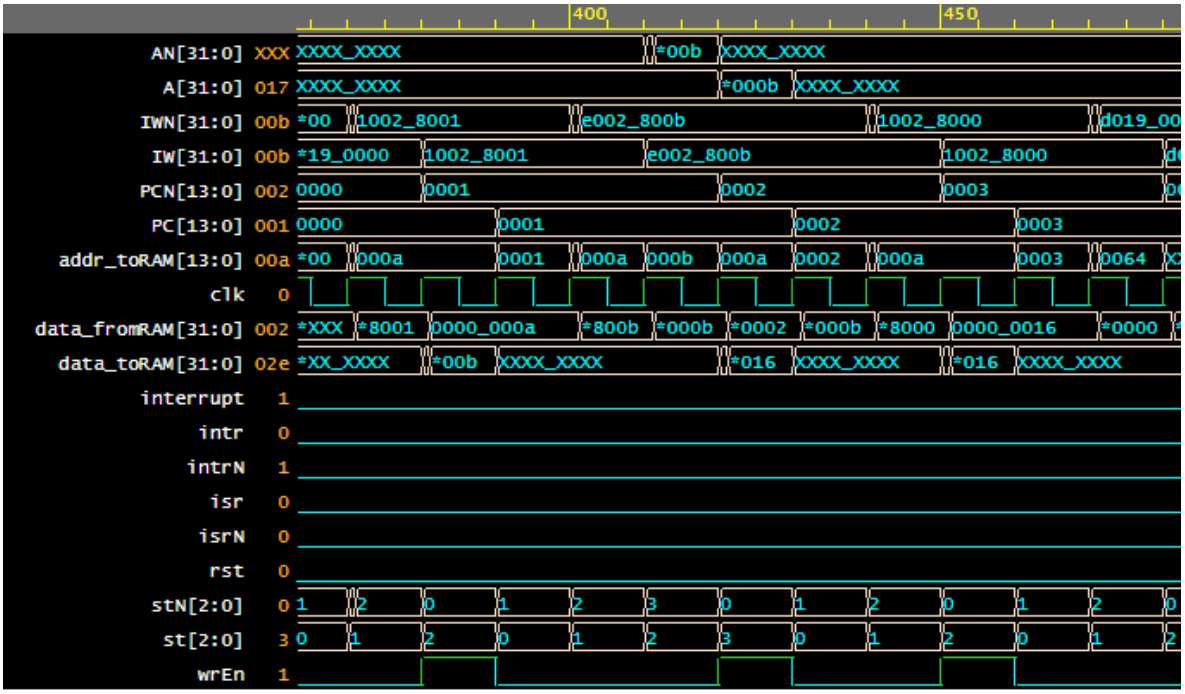
# VSCPU DESIGN CODE

In addition to the instructions written in the lesson, I added other instructions and their interrupt handling to the VSCPU Verilog Code. By writing the instructions in the VSCPU instruction set in the design file section, I have given the VSCPU some features such as addition, multiplication, shifting and copying. In this design, I also wrote what to do in each instruction in case of an interrupt to the CPU. To give an example of how I wrote the code from one VSCPU instruction, after I understood what the instruction was doing at first, I decided on the number of states I should use. First, the 3'b000 state is a fixed state for all constructitons. Because that state exists to understand which instruction to do. By giving the value in the program counter to addr_toRAM, we are in a position to get the instruciton from the data_fromRAM in the next state, and then we go to 3'b001 state. This state is sending the instruction via data_fromRAM. The purpose of the IW is to keep and retain the instruction. Now we need to divide the codes to be written according to the instruction to be made in this state with ifs. Because the purpose of each instruction is different. data_fromRAM[31:28] == 4'b0000 gives the ADD instruction so by checking 31:28, 31:29 oe 31:20 ve can divide the instruction with ifs. For example, if we are doing the ADD instruciton addr_toRAM = data_fromRAM[27:14]; can be done to read the value of A in the next state. Then we go the 3'b0010 state. Since we gave the address of a to the addr_toRAM in this state, the value of a now comes from the data_fromRAM. AN = data_fromRAM; Keeps the A's value. For the addition operation, we also need to obtain the B's value, so we write this: addr_toRAM = IW[13:0]; . Then we go the 3'b0100 state. Here we now have obtained B's value from the data. By doing addr_toRAM = IW[27:14]; We are writing the additions result to A's adress. data_toRAM = A + data_fromRAM; does the addition. After that we make write enable and increment the program counter We interrupt the last states of the istructions. If there is no ingterrupt we go back to 3'b000 state and if there is first we go to 3'b100 state and then 3'b101. These states are added for the interrupt situation. 3'b100 gets the ISR address and 3'b101 stores the next instruction to be executed before the interrupt. It stores it because after the interrupt handled program shoul continue where it is left. And also there are immediate instructions such as ADDi. Unlike the Add it adds a memory adresses value with a given value not the adress. Its im is 1'b1.

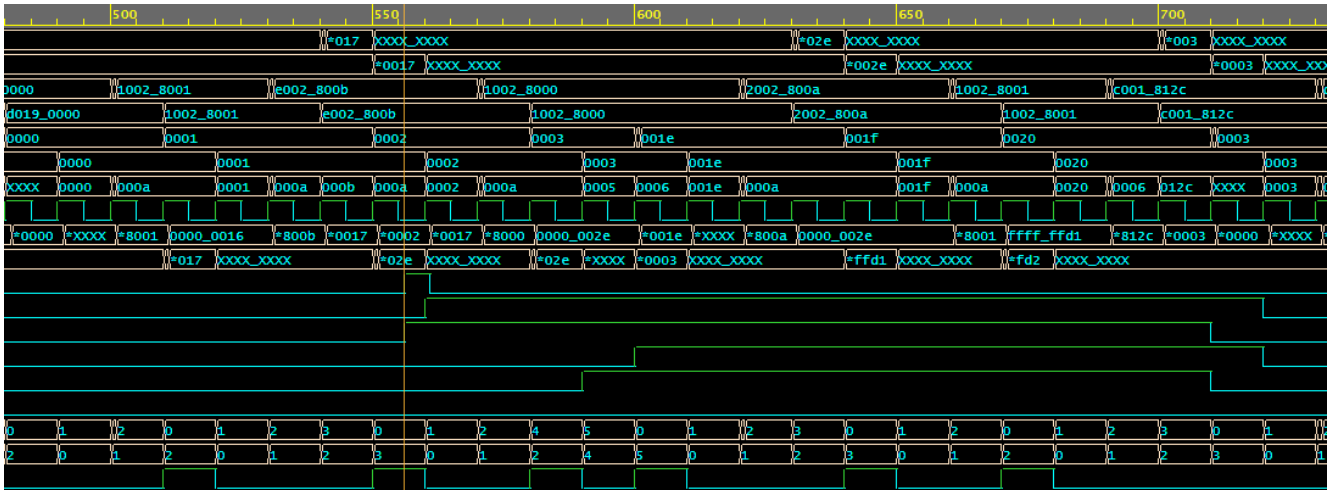
After I wrote the design file. I tested all my intructions by writing the assembly codes in the Turkish CPU simulator then getting their machine codes and appending it to testbench. As a result i saw that all my instruction codes are working correctly by checking the EPWave in edaplayground.

# EXAMPLE OF MUL and ADDi INSTRUCTION

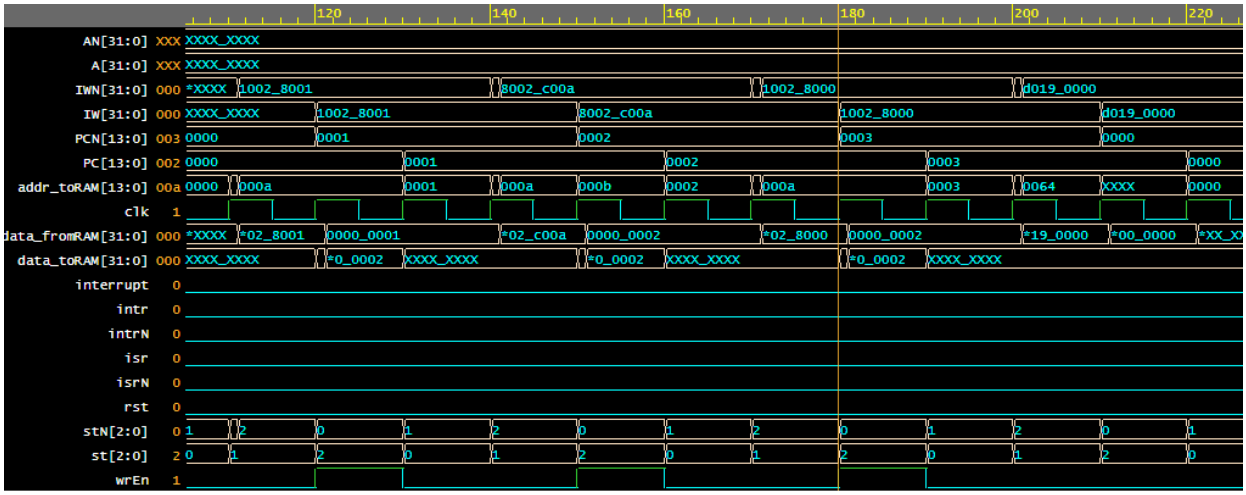
```
;
   initial begin
     $dumpfile("dump.vcd"); $dumpvars;
     blram.memory[0] = 32'h10028001; // ADDi 10 1
     blram.memory[1] = 32'he002800b; // MUL 10 11
     blram.memory[2] = 32'h10028000;
     blram.memory[3] = 32'hd0190000;
     blram.memory[5] = 32'd30; //ISR begin address @30 end address @32
     blram.memory[10] = 32'h1;
     blram.memory[11] = 32'h2;
     blram.memory[30] = 32'h2002800a;
     blram.memory[31] = 32'h10028001;
     blram.memory[32] = 32'hc001812c; //BZJ 6, 300
     blram.memory[100] = 32'h0;
     blram.memory[300] = 32'h0;
```



First  1 is added to value of the 10th address. Then 10th addressed value is multiplied by 11th addresses value and stored in 10th address. This operation is done infinite times and if there is an interrupt program handles it and then go back to doing addi and mul again and again. In the EPWave we can see that 0000 adds 1. Then in 0001 multipication by 2 happens. (*016 + 1) = * 017 then ( *017 x 2) = *02e. In 0003 program goes back 0000 if there is no interrupt. If there is after doing the current instruction program counter goes to 001e -> 001f->0020 then goes back to actual program.



# EXAMPLE OF CP and ADDi INSTRUCTION




First 1 is added to value of the 10$^{th}$ address. Then 10$^{th}$ addressed value is copied to 11$^{th}$ addresses. This operation is done infinite times and if there is an interrupt program handles it and then go back to doing addi and cp again and again. In the EPWave we can see that 0000 adds 1. Then in 0001 value of 10$^{th}$ address is added to 11$^{th}$ adress . (*0001+ 1) = * 0002 then inside 11$^{th}$ address-> *0002 the program goes back 0000 if there is no interrupt. If there is after doing the current instruction program counter goes to 001e -> 001f->0020 then goes back to actual program.

# FLOOR DIVISION

Input ASM Code

```
0: CP 15 9 // Copy 9th addresses value to 15th address
1: NAND 15 9  // 1's complement
2: ADDi 15 1  // 2's complement
3: ADD  10 15 //A Added negative number to 10th addresses value to make subtraction
4: CP 12 10
5: ADDi 11 1 // Keeping track of how many times the substraction occured in 11th adress
6: LT 12 9 //If it is more than we keep substracting, else we got out of the loop
7: BZJ 13 12
9: 5
10: 11
11: 0 // Keeps floor divisions result
12: 0
13: 0
15: 0
```

Memory address 11  sucessfully keeps the floor division result whih is 2. (11/5 = 2 in integer)

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
    current_instruction:  ADDi 11 1
    program counter     :  5
    Memory content before executing instruction
    mem[ 11 ]           :  1
    Memory content after executing instruction
    mem[ 11 ]           :  2
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Floor division can be done with the given instructions in the VSCPU. I thought I could NAND(nands the value at two addresses) a number with itself which gives its 1's complement then by doing ADDi(adds 1 value at address and 1 immediate value) I add this number 1 which it gives 2's complement of itself. These operations gave me the negative form of a number. Then I thought I could ADD(takes the value at two addresses) this negative number with the number I'm going to do floor division on until I get 0. Then I kept counting the number of subtractions I made until the result was zero at another address, and that addresses value at the end gave me the result of the floor division. I made the situation of continuing to subtract until zero by creating a loop. LT checks if the number is less than or greater than the subtracted number.If the result from LT(If the value in the 1st address is less than the value in the 2nd address, it writes 1 to the 1st address, otherwise it writes 0) is zero, the BZJ(If the value at the second address is less than the value at the 1st address, it goes to the address with the value of the 1st address, if it is larger, the program counter increases by 1.) instruction continues the loop, if it is 1, it exits the loop and finishes subtraction. Since I will be doing operations on the number I will substract, I copied it to an address with CP (copies the value at two addresses) and used the value at that address in order not to lose its value. In the second CP(copies the value at two addresses), since the result of each loop will be used in LT, I wrote its value to another address and used the value at that address in order not to lose its value.