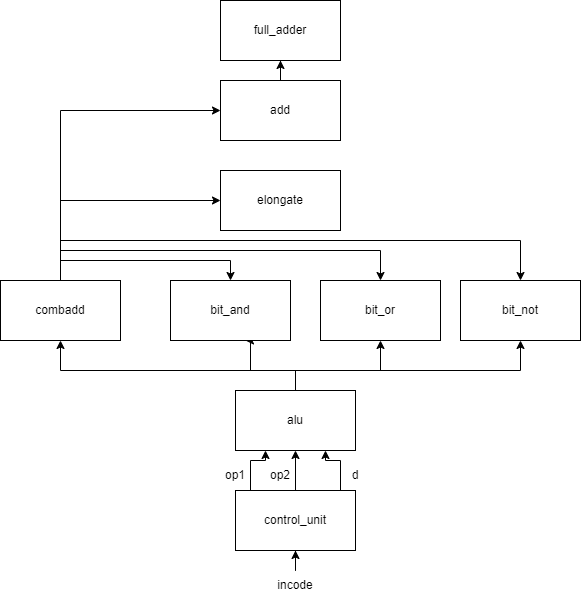
19-bit CPU Design Report

Name: **Saran.V.Balachandar**

Connections/dependencies between the modules used.



CLU

* The control unit (CU) takes in a 19-bit instruction code as the input, and its output is an 8-bit binary number, our required answer.
* The instruction code is split into 3 parts, the 3-bit opcode and two 8-bit operands, op1, and op2.
* To determine what opcode is given in the input, I created a 3x8 decoder and set the active line in the wire d[7:0] to 1.
* Now I have the two operands and the deciphered opcode, so I pass these parameters to the ALU module, which performs a particular operation on these two operands based on the opcode given in the input.

ALU

* The ALU receives the two operands and the output line ‘d’ from the 3x8 decoder as input.
* Since we are supposed to use only structural design, I store the answers for all 7 operations.

Storing the answers for the 7 operations

* I compressed the first four operations – ADD, SUBTRACT, INCREMENT and DECREMENT into a single module called ‘combadd’ (combined addition). This is more efficient, as I create only one instance of the ‘add’ module for all four operations combined.
* The naive solution would be to create four instances of the ‘add’ module, one per operation, which would have been inefficient.
* Inside the ‘combadd’ module:
  + I passed the operands op1 and op2 and the output lines from the 3x8 decoder as input to this module.
  + I then created a 4x1 multiplexer to determine op2, based on which line is active in 'd'.
    - op2: op2 if opcode = ‘001’ – ADD.
    - op2: 2’s complement of op2, if opcode = ‘010’ – SUBTRACT (Since subtraction of op2 from op1 is equivalent to adding op1 to 2’s complement of op2).
    - op2: 8'b00000001, if opcode = ‘011’ – INCREMENT (Since increment is defined as adding 1 to op1).
    - op2: 8'b11111111, if opcode = ‘100’ – DECREMENT (Since decrement is defined as subtracting 1 from op1, which is the same as adding 2’s complement of 1 = 8’b11111111 to op1).
  + I do one-hot encoding of each bit in the output line 'd' to an 8-bit bus. It acts as an enable to determine which of the 4 operations is to be done.
  + Once I find the required op2 from the multiplexer, I add it with op1 using the ‘add’ module and return the 8-bit binary number as output.
* Similarly, I stored the answers for the operations AND, OR, and NOT.
  + AND – I created the ‘bit\_and’ module, which does the bitwise AND operation on each respective bit of op1 and op2.
  + OR - I created the ‘bit\_or’ module, which does the bitwise OR operation on each respective bit of op1 and op2.
  + NOT – I created the ‘bit\_not’ module, which does the bitwise NOT operation on each bit of op1.
* Now that I have all 4 stored answers (1st four operations are combined + the last 3) in wires in the ALU module, I need to construct the final 8-bit binary number answer based on the input opcode.

Constructing the final 8-bit binary number answer from these 4 stored answers

* I created 8 4x1 multiplexers to determine each bit of the output using the 4 stored answers in the wires.
* I return the 8-bit binary number obtained as the required final answer.

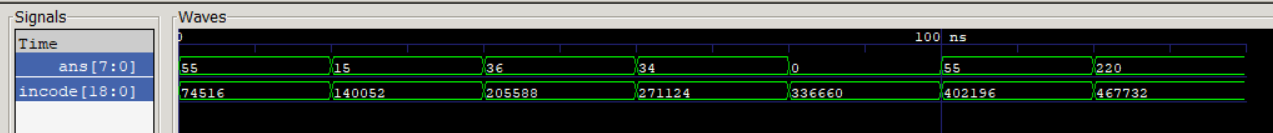
Testing using a test bench

Test bench used:



Output

Screenshot of the output waveform (values are in decimal format)



* We easily verify that all the outputs are correct.

Modules used, their functions, and their waveforms

1. control\_unit.v

Input: [18:0] incode

Output: [7:0] ans

Function: Takes in a 19-bit instruction code and produces the final answer, an 8-bit binary.

Modules used inside: alu

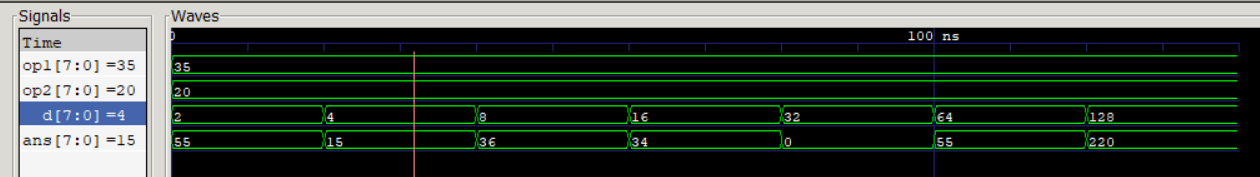
1. alu.v

Input: [7:0] op1, [7:0] op2, [7:0] d

Output: [7:0] ans

Function: Returns the final 8-bit binary answer after operating on op1 and op2 based on the input opcode.

Modules used inside: combadd, bit\_and, bit\_or, bit\_not



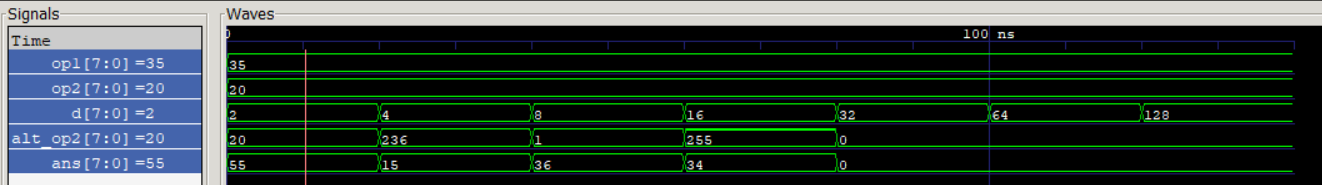
1. combadd.v

Input: [7:0] op1, [7:0] op2, [7:0] d

Output: [7:0] ans

Function: Combined module for the first 4 operations (ADD, SUBTRACT, INCREMENT, DECREMENT)

Modules used inside: bit\_not, add, elongate, bit\_and, bit\_or

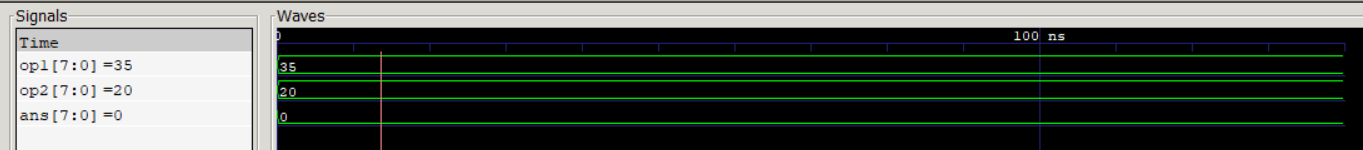


1. bit\_and.v

Input: [7:0] op1, [7:0] op2

Output: [7:0] ans

Function: Bitwise AND of two 8-bit binary numbers

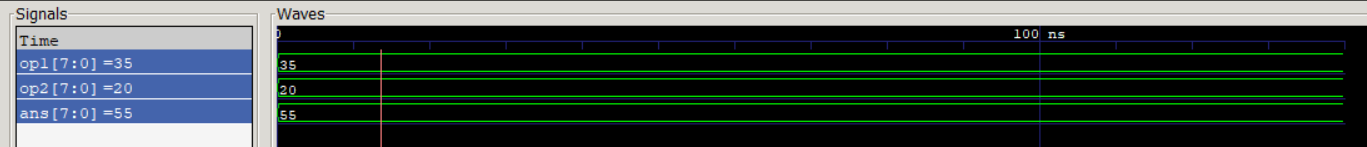


1. bit\_or.v

Input: [7:0] op1, [7:0] op2

Output: [7:0] ans

Function: Bitwise OR of two 8-bit binary numbers

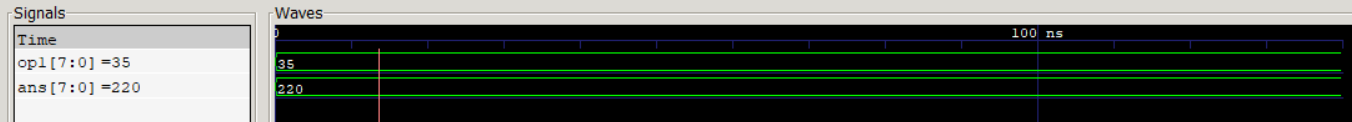


1. bit\_not.v

Input: [7:0] op1

Output: [7:0] ans

Function: Bitwise NOT of an 8-bit binary number

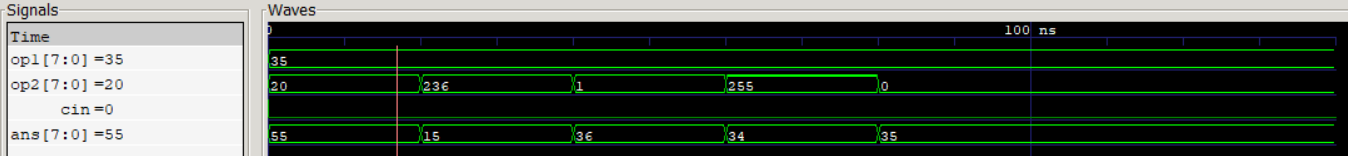


1. add.v

Input: [7:0] op1, [7:0] op2, cin

Output: [7:0] ans

Function: It is an 8-bit Ripple Carry Adder

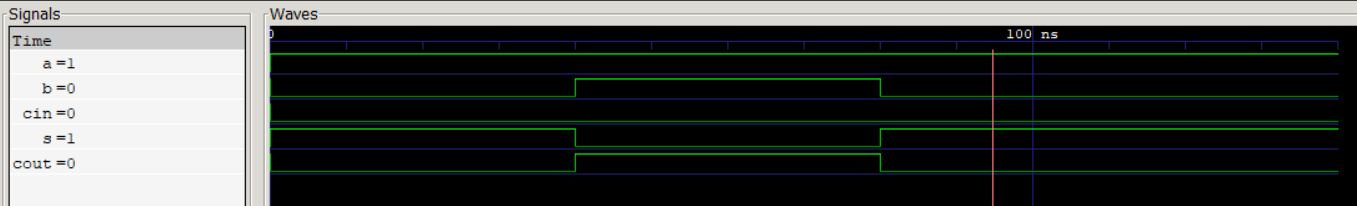
Modules used inside: full\_adder

1. full\_adder.v

Input: a, b, cin

Output: s, cout

Function: It is a 1-bit Full Adder

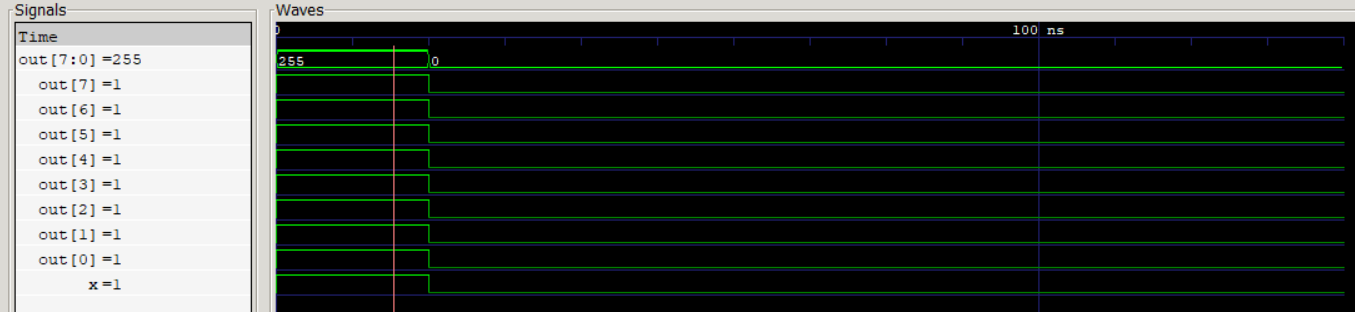


1. elongate.v

Input: x

Output: [7:0] out

Function: Creates an 8-bit binary number xxxxxxxx from a single bit input x.



1. test\_bench.v

Function: Test bench for the program.