CPU Design Lab Report

K V P Sai Teja
CS21B037

This report contains the information about the CPU design implementation using verilog. I have used a control unit (CU) and an arithmetic and logic unit (ALU) for this implementation.

The Control Unit and the Arithmetic and Logic Unit

The CU takes the 19-bit input and separates the first 3 bits as the operation code, which tells the ALU what operation to perform, and the next 8 bits as the first operand and the remaining 8 bits as the second operand. Now all the registers are sent to the ALU from where the result is sent back to the ALU.

- Now let's assume that the addition, subtraction, increment and the other operations have been computed, whose working will be explained later in this report. Firstly, I precompute the results of all the operations and store them in addresult, subresult and so on. Using the select lines, I have computed all the possible combinations that could arise using three select lines (except $\sim s_0 . \sim s_1 . \sim s_2$) and stored as different variables, let's say ww1,ww2,... and so on.
- Now I want to convert these unit bits into 8 bit binary strings, like if it's equal to 0, then I convert it into 00000000, and if it's 1 then I convert it into 11111111. This can be achieved by first converting ww1,ww2,... into 8 bit numbers whose first 7 bits are 0s and the last bit is a 1 or a 0. Let's call them www1,www2,....
- Now by taking the 2s complement of this converted ww1, ww2, ww3, we get 00000000 for 0-case and 11111111 for 1-case. Now I just performed bitwise-and operation for these www1,www2 with corresponding addresult, subresult… . Now when I compute the or of all these and operations, I finally get the required result.

Working of various functions

1) *Full Adder* :- This is the basic module for the CPU design, it takes 2 unitary bits and a carry as input and gives a carry and the sum as the output. Here the sum is a^b^cin and cout=a.b+b.cin+a.cin

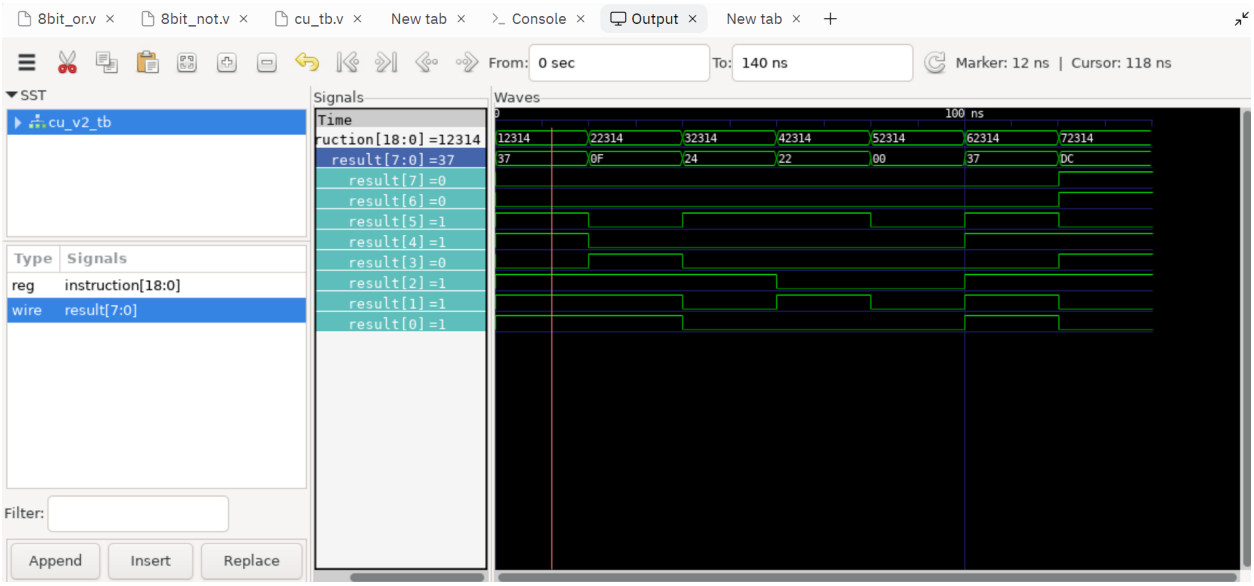
2) *Addition* :- The 8-bit ripple carry adder is constructed from 8 full adders where the carry is carried throughout the full adders and finally gives the required sum as output. This takes 2 8-bit registers and a carry initial as input and outputs the 8-bit sum.
3) *Increment* :- This can be easily computed by passing the given 8-bit register , 8-bit 0 and an initial carry of 1 as inputs to the 8-bit ripple carry adder. The output of the ripple carry adder is the required result.
4) *Bitwise-not* :- This can be computed by just taking the complement of each bit using not gates.
5) *2s-complement* :- This can be computed by taking the bitwise-not of the input 8-bit register and then incrementing the obtained result.
6) *Subtraction* :- This is computed by taking the 2s-complement of the second operand and adding it to the first operand using the 8-bit ripple carry adder already constructed.
7) *Decrement* :-This is computed by adding the 2s-complement of 1 to the given input i.e; the 2s-complement of 1 is 8'b11111111. So we use the 8-bit adder again and give the given input , 8'b11111111 and initial carry 0 as inputs and the sum obtained is the required result.
8) *Bitwise-and*:- This can be easily computed by taking the bitwise-and of the given two 8-bit inputs using and gates.
9) *Bitwise-or*:- This can be easily computed by taking the bitwise-or of the given two 8-bit inputs using or gates.

Working and Testing

✷ A snapshot of the testbench for the entire CPU :-

```
4   module cu_v2_tb;
5
6       reg [18:0] instruction;
7       wire [7:0] result;
8
9       cu_v2 uut (result, instruction);
10
11      initial begin
12
13          $dumpfile ("cu_v2_tb.vcd");
14          $dumpvars (0, cu_v2_tb);
15
16          instruction = 19'b0010010001100010100; #20; //Addition
17          instruction = 19'b0100010001100010100; #20; //Subtraction
18          instruction = 19'b0110010001100010100; #20; //Bitwise And
19          instruction = 19'b1000010001100010100; #20; //Bitwise Or
20          instruction = 19'b1010010001100010100; #20; //Bitwise Not
21          instruction = 19'b1100010001100010100; #20; //Increment
22          instruction = 19'b1110010001100010100; #20; //Decrement
23
24          $display ("Test Completed");
25      end
26
27  endmodule
```

GTKWAVE output for the test bench: The results are in hexadecimal format.


8bit_or.v
8bit_not.v
cu_tb.v
New tab
Console
Output
New tab
From: 0 sec
To: 140 ns
Marker: 12 ns | Cursor: 118 ns
SST
cu_v2_tb
Type Signals
reg instruction[18:0]
wire result[7:0]
Filter:
Append
Insert
Replace
Signals
Time
ruction[18:0] =12314
result[7:0] =37
result[7] =0
result[6] =0
result[5] =1
result[4] =1
result[3] =0
result[2] =1
result[1] =1
result[0] =1
Waves
100 ns
12314
22314
32314
42314
52314
62314
72314
37
0F
24
22
00
37
DC