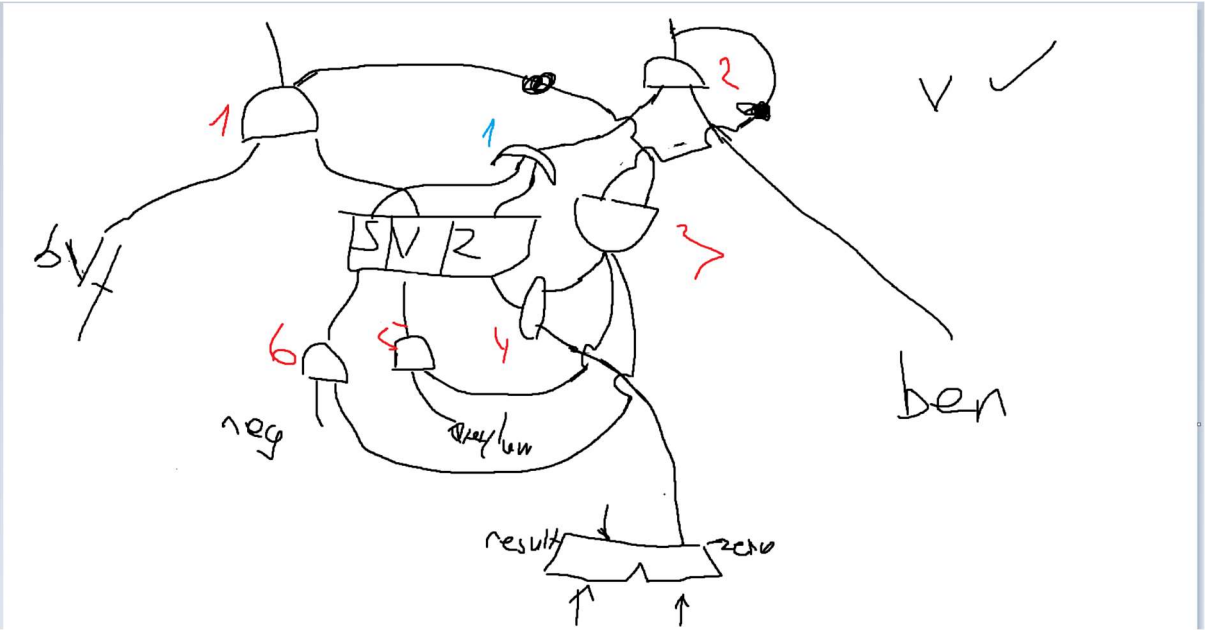
Implementation:

I first designed the CPSR register as it was wished.

```
You, 24 hours ago | 1 author (You)
module CPSR(negative_flag_and, overflow_and, zero_flag_and, svz, clk);

input negative_flag_and, overflow_and, zero_flag_and, clk;

output reg [2:0] svz;

always @(posedge clk)
begin
    svz[0] = zero_flag_and;
    svz[1] = overflow_and;
    svz[2] = negative_flag_and;

    // case(add_sub_bit)
    //  2'b00: begin
    //          if(~(a_most_significant) & ~(b_most_significant) & alu_
    //          else if ((a_most_significant) & (b_most_significant) &
    //      end
    //  2'b01: begin
    //          if((a_most_significant) & ~(b_most_significant) & ~alu_
    //          else if (~(a_most_significant) & (b_most_significant) &
    //      end
    //  2'b11: svz[1] = 0 // no overflow
    // endcase
end
initial
begin
    svz[0]=0;
    svz[1]=0;
    svz[2]=0;
end      You, 2 days ago • first real commit …
endmodule
```

Since we have to change the CPSR register according to the output of itself, I thought this calls for a flip flop and designed my own for the job. I also first wrote always block to execute for every change in the AND gates but it was problematic, the solution was to use a clk as an argument and execute the always block using the positive edge of the clock rather than when the changes in the AND gates happen.



The red numbered ones are AND gates and the single blue numbered one is an OR gate. First and second and gate outputs are NOT'ed and connected to third AND gate. This will set all the registers to zero if bvf or ben are taken.

```
3'b000: alu_out = a & b; // bitwise and
3'b001: alu_out = a | b; // bitwise or
3'b011: alu_out = a ^ b; //bitwise XOR
3'b100: alu_out = ~(a|b); // NOR
```

I modified the alu32 to execute NOR as well for the given bit sequence.

```
if (~(f3|f2|f1|f0))
    gout=3'b010; // addition
if (f1 & ~f2 & f3 & ~f0)
    gout=3'b111; // set less than
if (f1 &~(f3) & ~f0 & ~f2)        You, 2 days ago • first real commit …
    gout=3'b110; // subtract 001000
if (f2 & f0 & ~f1 & ~f3)
    gout=3'b001; // or
if (f2 &~(f0) & ~f1 & ~f3)
    gout=3'b000; // and
if (f2 & f1 & f3 & f0) // XOR
    gout = 3'b011;
if (~f3 & f2 & f1 & f0) // it was not taken, so I will use it for nor 000111
    gout = 3'b100;
```

I modified the ALU control to send this bit sequence to ALU to execute NOR operation, can be seen at the end.

```
You, yesterday | 1 author (You)
1  module AND(a,b,out, clk);
2  input a,b, clk;
3  output reg out;
4  always @(a or b)
5  begin
6  out = a & b;
7  end
8  initial
9  begin
10     out=0;
11 end
12 endmodule        You, 2 days
```

```
1  module OR(a,b,out,clk);
2  input a,b,clk;
3  output reg out;
4  always @(a or b)
5  begin
6  out = a | b;
7  end
8  initial
9  begin
10     out=0;
11 end
12 endmodule
13
```

I also created AND and OR gates for my flip flop inside Verilog. I first tried to use clk for these as well besides CPSR register but later I found out they are not needed in the gates to perform correctly so I removed them from the equation totally and only left a clock inside CPSR.

```
You, 2 days ago | 1 author (You)
1  module shift_for_jump_operation(shout, shin);
2  output [27:0] shout;
3  input [25:0] shin;
4  assign shout=shin<<2;
5  endmodule
6
```

Added a shifter specifically for the jump instruction here.

```
module control_modified(in, regdest, alusrc, memtoreg, regwrite, memread, memwrite, branch, aluop1, aluop2,
input [5:0] in;
output regdest, alusrc, memtoreg, regwrite, memread, memwrite, branch, aluop1, aluop2, ben, bvf, jump; //

wire rformat,lw,sw,beq;

assign rformat =~| in;

assign lw = in[5]& (~in[4])&(~in[3])&(~in[2])&in[1]&in[0]; // 100011

assign sw = in[5]& (~in[4])&in[3]&(~in[2])&in[1]&in[0]; // 101011

assign beq = ~in[5]& (~in[4])&(~in[3])&in[2]&(~in[1])&(~in[0]); // 000100

assign ben = (~in[5])& (~in[4])&(~in[3])&in[2]&(in[1])&(~in[0]); // 000110

assign bvf = (~in[5])& (~in[4])&(~in[3])&(in[2])&(~in[1])&(in[0]); // 000101

assign j = ~in[5]& (~in[4])& ~in[3] & (~in[2])&in[1]& ~in[0]; // 000010

assign addi = ~in[5]& (~in[4])& in[3] & (~in[2])& ~in[1]& ~in[0]; // 001000, alusrc = 1, regwrite = 1, aluc

// these are all one bit
assign regdest = rformat;
assign alusrc = lw|sw|addi;
assign memtoreg = lw;
assign regwrite = rformat|lw|addi;
assign memread = lw;
assign memwrite = sw;
assign branch = beq;       You, 2 days ago • first real commit …
assign aluop1 = rformat|(~addi); // two bits but shown seperately
assign aluop2 = beq|(~addi); // since when aluop1 and aluop2 are zero we do addition (lw and sw) I want to
assign jump = j;

endmodule
```

As it can be seen, I made changes and additions in control. I first added control wires as it was asked in the homework, 6 for ben and 5 for bvf specifically. Did the same for jump and addi instructions as well. As it was mentioned in the lab I figured it is enough to set the writing operations to zero when they are not needed (for jump, ben and bvf) so I didn't add anything about them at the end, but for addi, since it is I type, alusrc must be 1 when it is executed, as well as regwrite. Hence, I added addi to them.

Next implementations take place in the updated processor:

Implementation for jump:

```
// Added for jump (ref page 19 09 processor slide)

shift_for_jump_operation shift_for_jump_operation1(sign_extended_jump, inst25_0);
// assign pc_and_sign_extended_jump_concatenated = {adder1out[31:28], sign_extended_jump}; // adder1 out is PC + 4
mult2_to_1_32 multiplexer_to_choose_whether_jump(from_jump_to_bvf_multiplexer, from_branch_to_jump_multiplexer,
{adder1out[31:28], sign_extended_jump}, jump_signal_coming_from_controller);

// Added for jump
```

Implementation for bvf and ben:

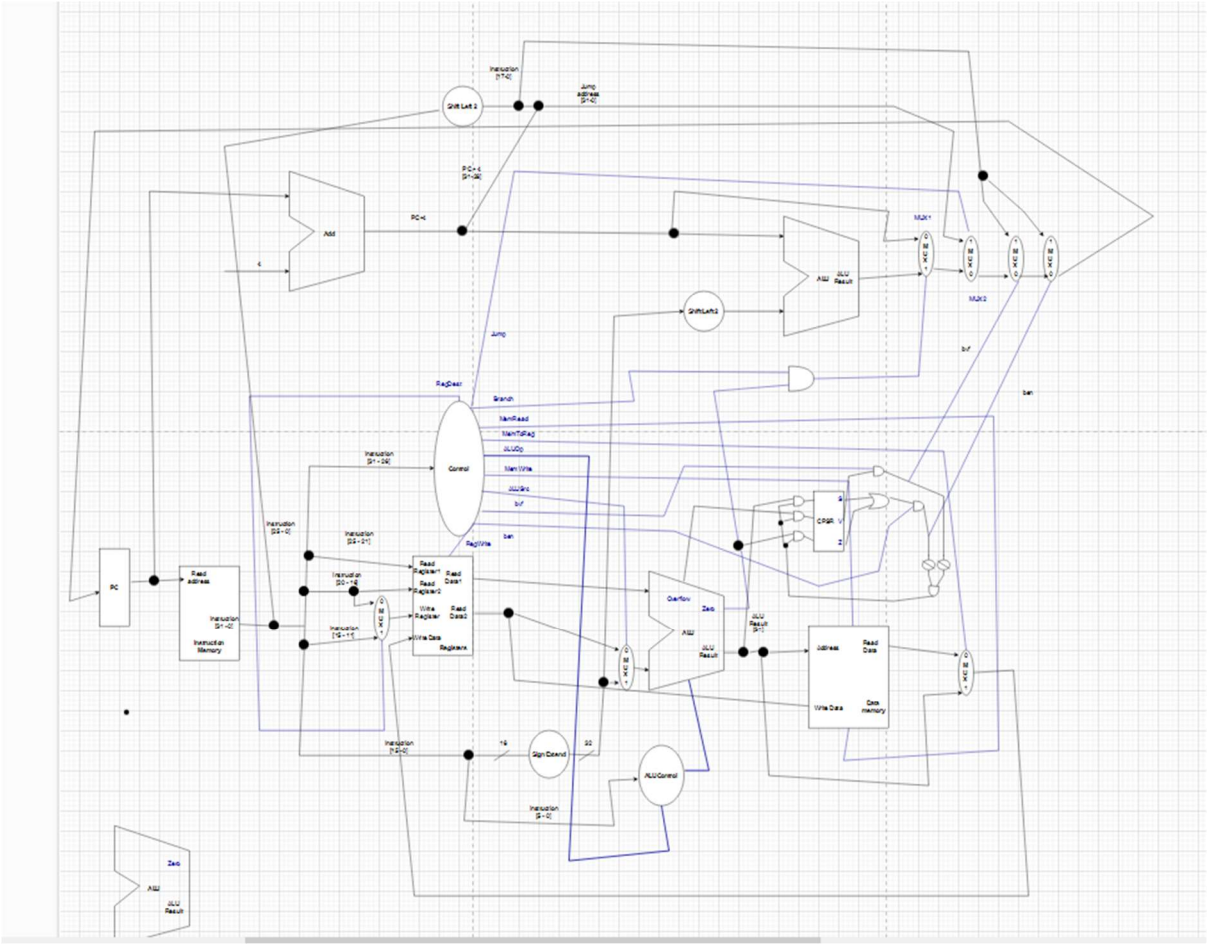
```
// added multiplexers for ben and bvf jump

signext signext_bvf_ben(inst15_0, sign_extended_bvf_ben);
mult2_to_1_32 multiplexer_to_choose_whether_bvf(from_bvf_to_ben_multiplexer, from_jump_to_bvf_multiplexer, sign_extended_
mult2_to_1_32 multiplexer_to_choose_whether_ben(out4, from_bvf_to_ben_multiplexer, sign_extended_bvf_ben, and_2_out);

// added for ben and bvf jump
```

Implementation of flip flop and CPSR:

```
// added for cpsr
AND cpsr_and1(bvf_signal_coming_from_controller, svz[1], and_1_out, clk);
AND cpsr_and2(ben_signal_coming_from_controller, or_1_out, and_2_out, clk);
AND cpsr_and3(~and_1_out, ~and_2_out, and_3_out, clk);
AND cpsr_and4(zout, and_3_out, and_4_out, clk);
AND cpsr_and5(overflow_out_alu, and_3_out, and_5_out, clk);
AND cpsr_and6(sum[31], and_3_out, and_6_out, clk);
OR cpsr_or1(svz[2], svz[0], or_1_out, clk);
CPSR cpsr1(and_6_out, and_5_out, and_4_out, svz, clk);
// added for cpsr
```

As I mentioned I tried clock with everything to solve my problem, I didn't remove them from the wires except CPSR (last arguments) to mention my steps during the coding.

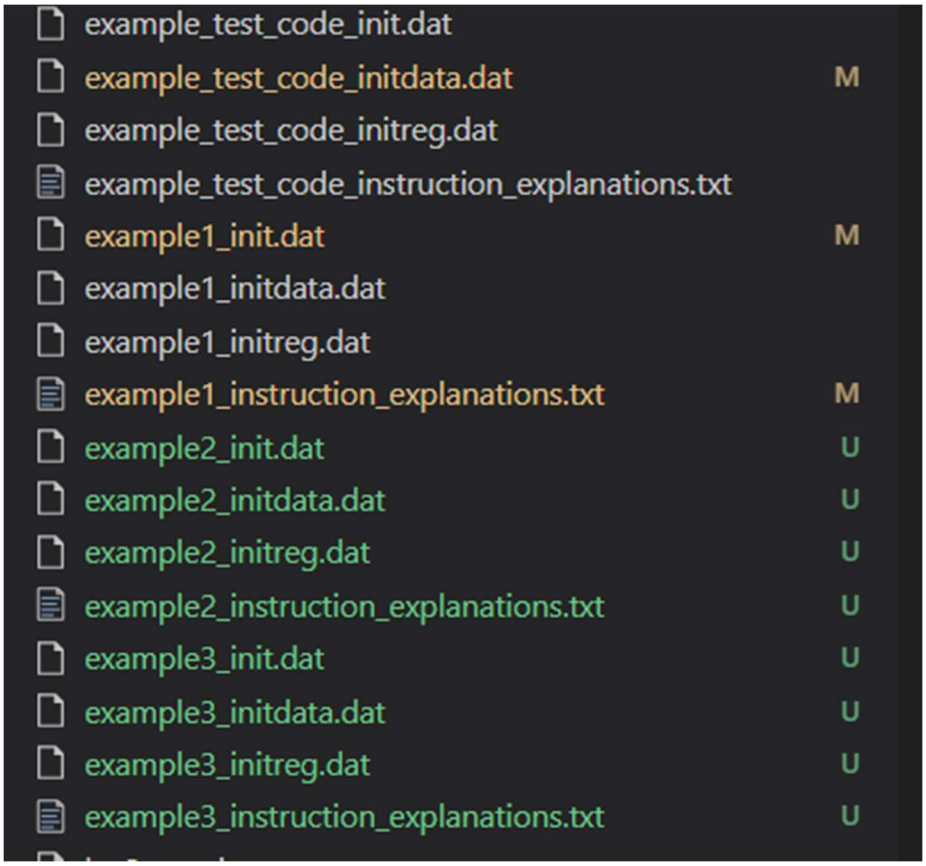
My revised datapath is something like so:



I will also add it as a separate picture. On top, I added three more multiplexers to choose from while setting PC, namely for bvf, ben, and jump instructions which are newly added. New control wires for jump, bvf and ben. New shifter on top for jump instruction. Also CPSR and necessary AND and OR gates for the flip flop which is used for CPSR.

Demonstration:

Below I will demonstrate the three examples given in the PDF file as well as the example test code which was at the end of the file.



For every example I have created 4 different documents like the example it was given to us. Bit and hex sequences can be found in instruction explanations.

For the instructions where PC changes occur, I will omit PC incrementation of 4 every cycle and only mention it when a jump occurs.

In the example below, I first made a slight change to the example 1 demonstrated in the PDF file and made the second instruction (ben) jump to the beginning of the instruction memory. As it can be seen, on the ModelSim terminal, it causes a loop. (I also changed the jump instruction which is on the third line for every three example to make it jump to the start so it is obvious if jump is taken, in the PDF file it seemed like jumping to the next instruction that is to come.)

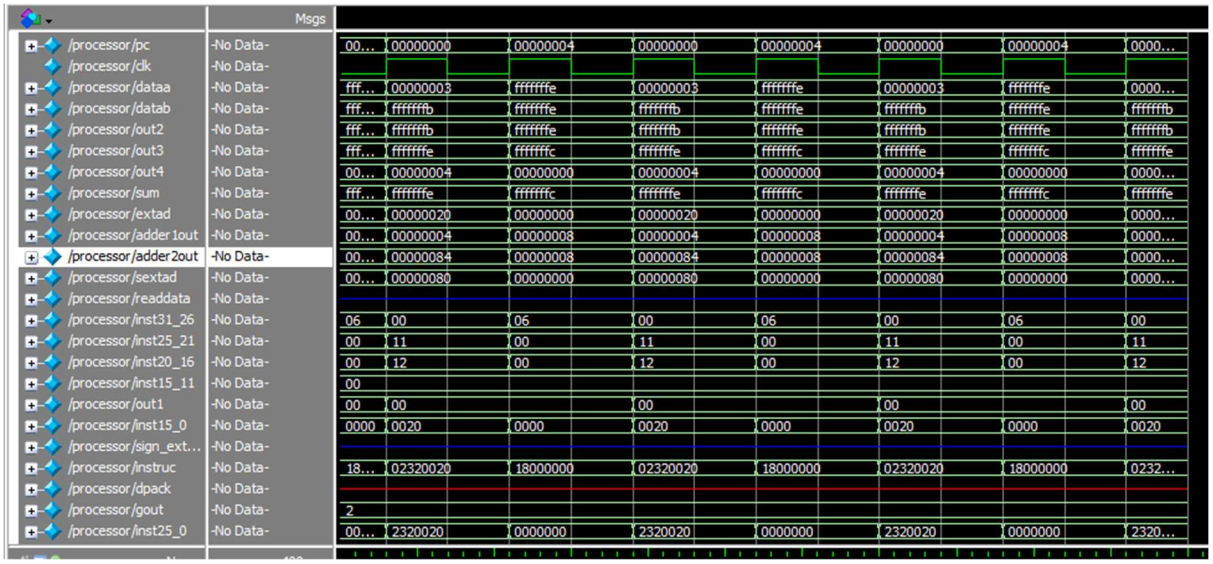


Figure 1: Modified example one, the program is in an infinite loop (see pc changing between 0 and 4 continuously.)

```
...
add $t0, $s1, $s2 -> 000000 10001 10010 00000 00000 100000 - > 02320020 s1 is 17.
ben label -> 000110 0000 000000 0000 0000 0000 0000 -> 18000000
j start -> 000010 0000 0000 0000 0000 0000 0000 00 -> 08000000
add $t0, $t1, $t2 -> 000000 00001 00010 00000 00000 100000 - > 00220020
nor $t3, $t4, $t5 -> 000000 00100 00101 00011 00000 000111 -> 00851807
```

Figure 2: Modified example 1 instructions

```
# Instruction Memory[30]= xx  Data Memory[30]= xx   Register[30]= xxxxxxxx
#                    0PC 00000000  SUM fffffffe   INST 02320020   REGISTER 00000030 00000032 00000042 00000014
#                   20PC 00000004  SUM fffffffc   INST 18000000   REGISTER 00000030 00000032 00000042 00000014
#                   60PC 00000000  SUM fffffffe   INST 02320020   REGISTER 00000030 00000032 00000042 00000014
run
#                  100PC 00000004  SUM fffffffc   INST 18000000   REGISTER 00000030 00000032 00000042 00000014
#                  140PC 00000000  SUM fffffffe   INST 02320020   REGISTER 00000030 00000032 00000042 00000014
#                  180PC 00000004  SUM fffffffc   INST 18000000   REGISTER 00000030 00000032 00000042 00000014
run
#                  220PC 00000000  SUM fffffffe   INST 02320020   REGISTER 00000030 00000032 00000042 00000014
#                  260PC 00000004  SUM fffffffc   INST 18000000   REGISTER 00000030 00000032 00000042 00000014
VSIM 114> run
#                  300PC 00000000  SUM fffffffe   INST 02320020   REGISTER 00000030 00000032 00000042 00000014
#                  340PC 00000004  SUM fffffffc   INST 18000000   REGISTER 00000030 00000032 00000042 00000014
#                  380PC 00000000  SUM fffffffe   INST 02320020   REGISTER 00000030 00000032 00000042 00000014
```

Figure 3: Terminal view after the execution of modified example 1

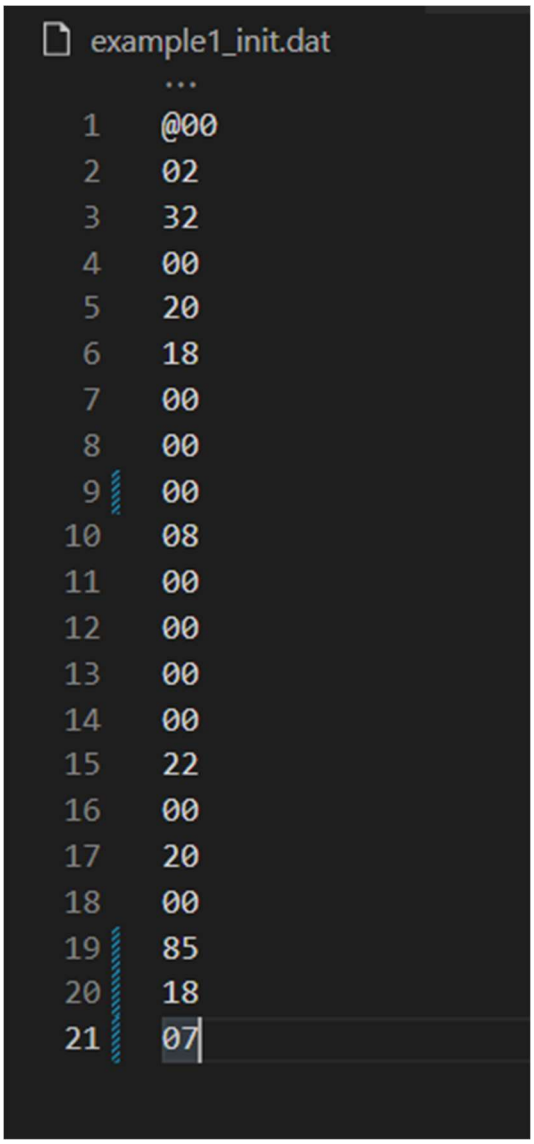


Figure 4: Instruction memory for modified example 1

Example 1:

Then I write the instructions as given in the PDF file so that ben jumps to the PC address of 16. Demonstrated below:

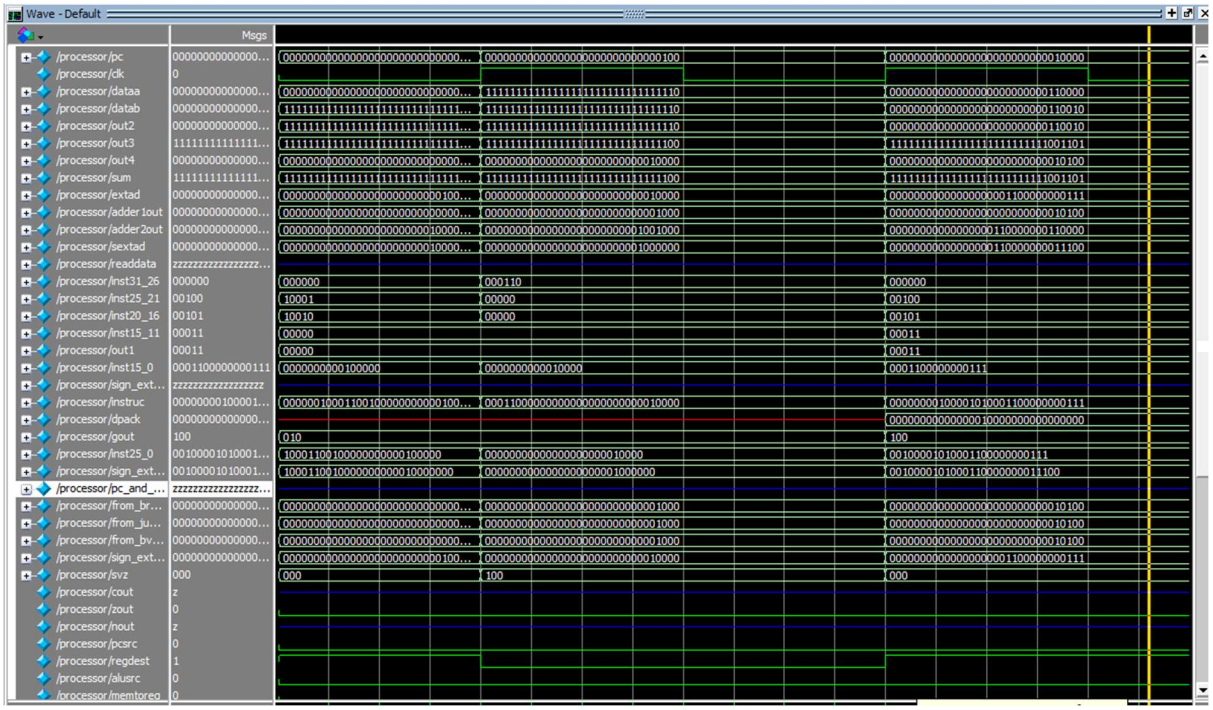


Figure 5: Wave view of example 1

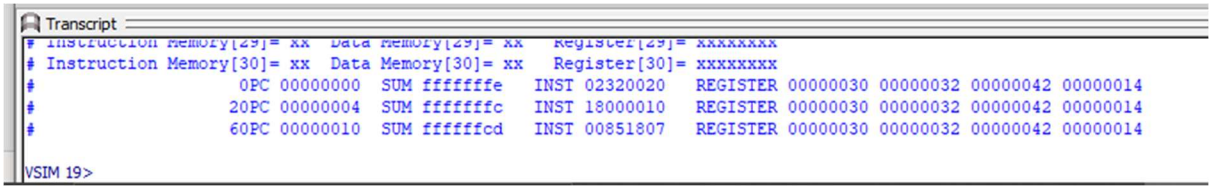


Figure 6: Terminal view of example 1

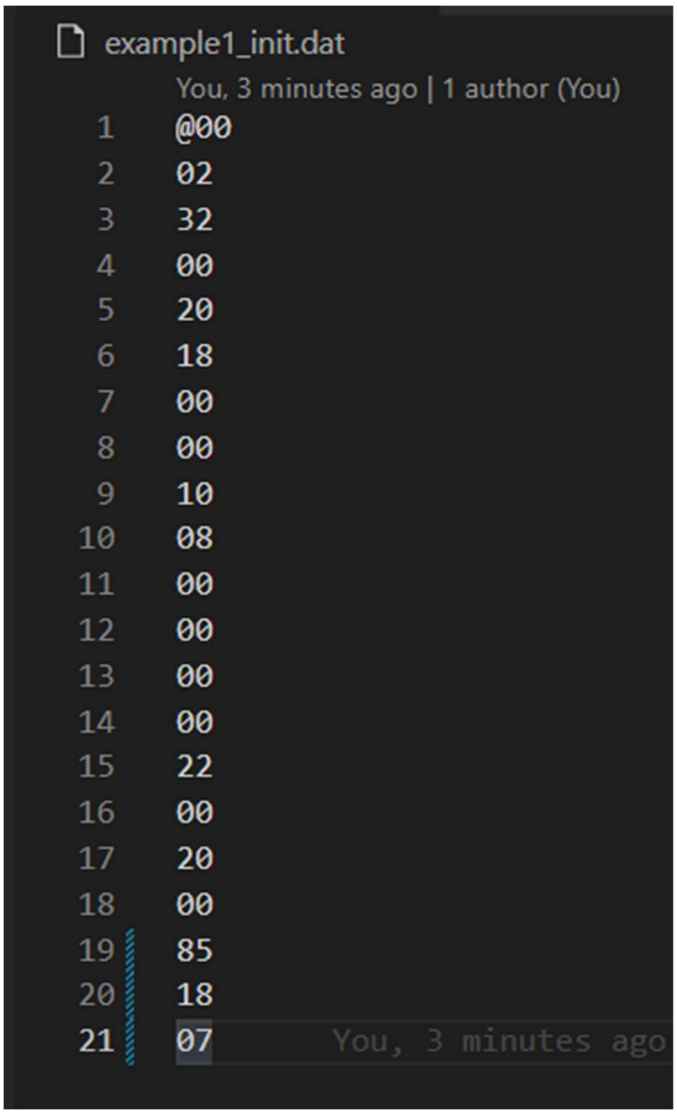


Figure 7: Instruction memory for example 1

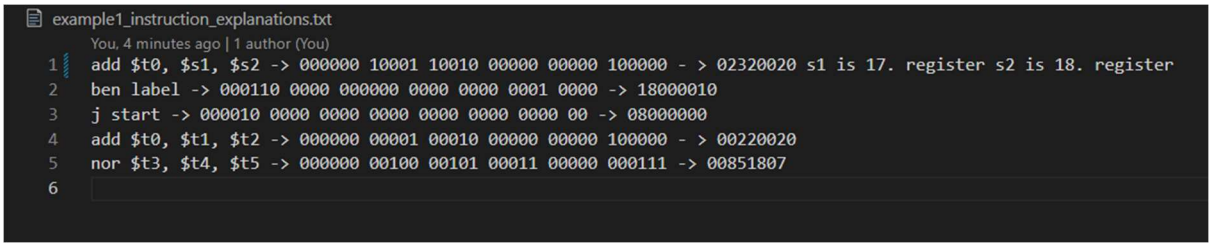


Figure 8: Explanations of the instructions for example 1

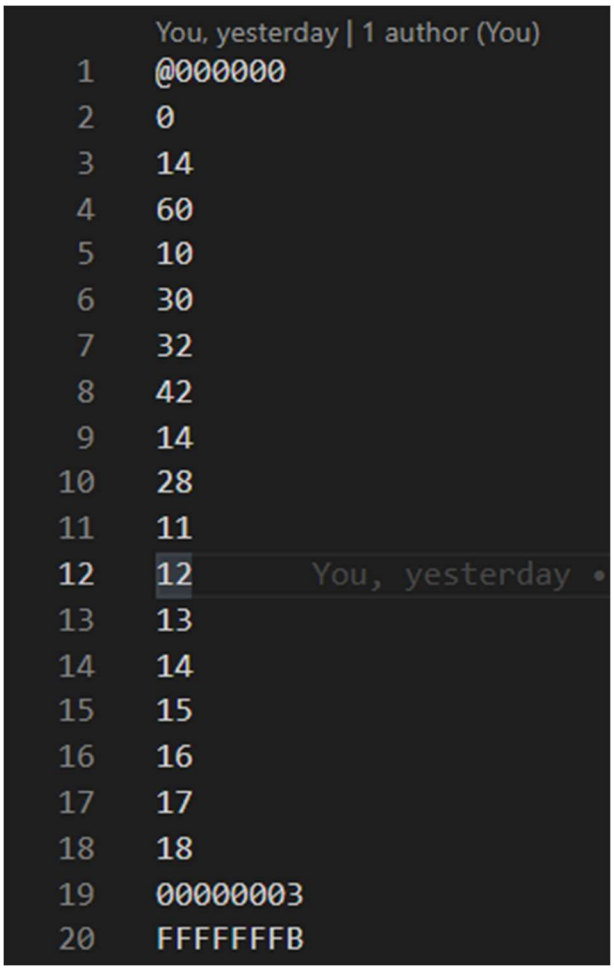


Figure 9: Register values for example 1

How it works:

1) First instruction to be executed is addition of s1 and s2 registers. The values of the registers and the result of the addition is given in the PDF file. In the picture below it can be seen that the execution checks out.

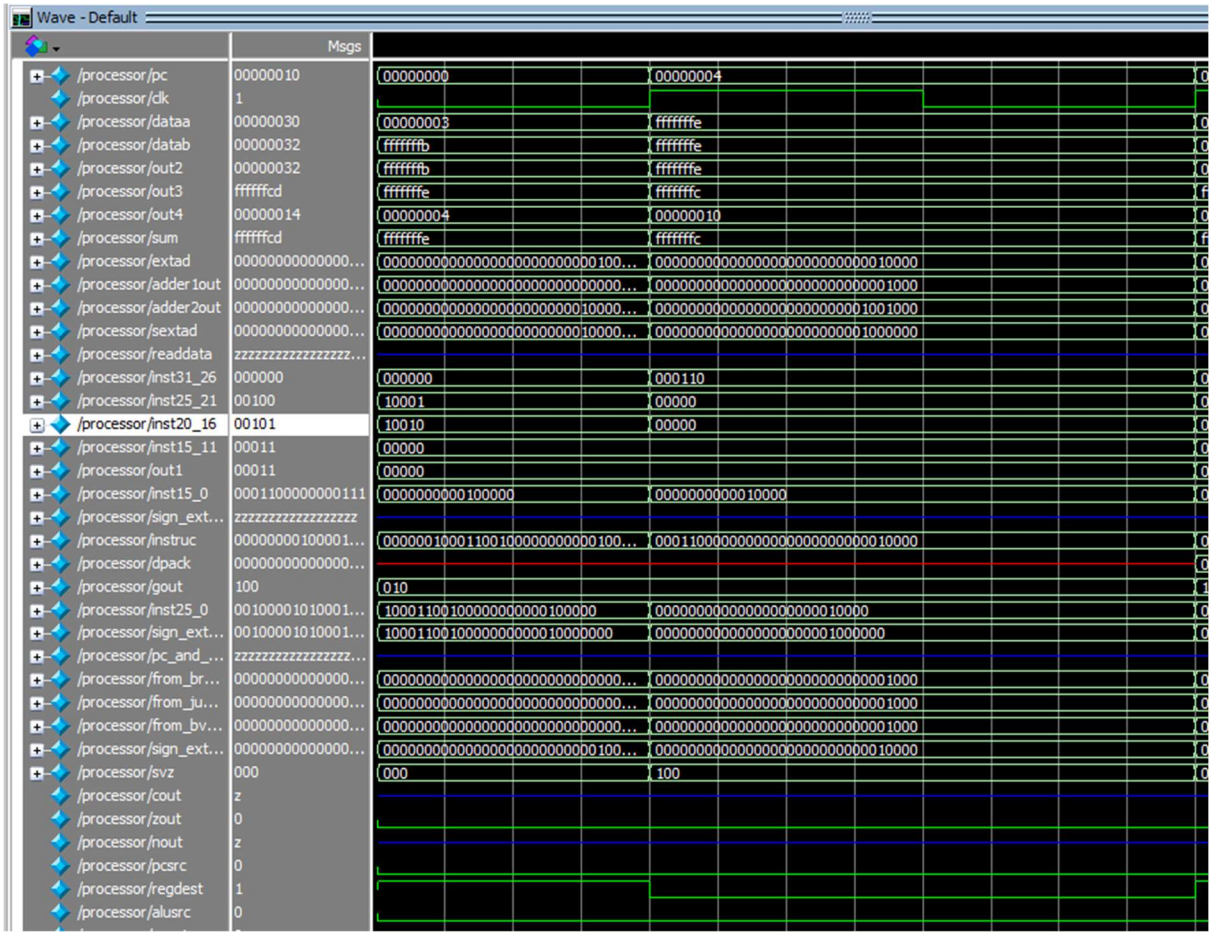


Figure 10: Addition is done correctly, dataa, datab and sum can be observed. SVZ is set to 100 in the second cycle

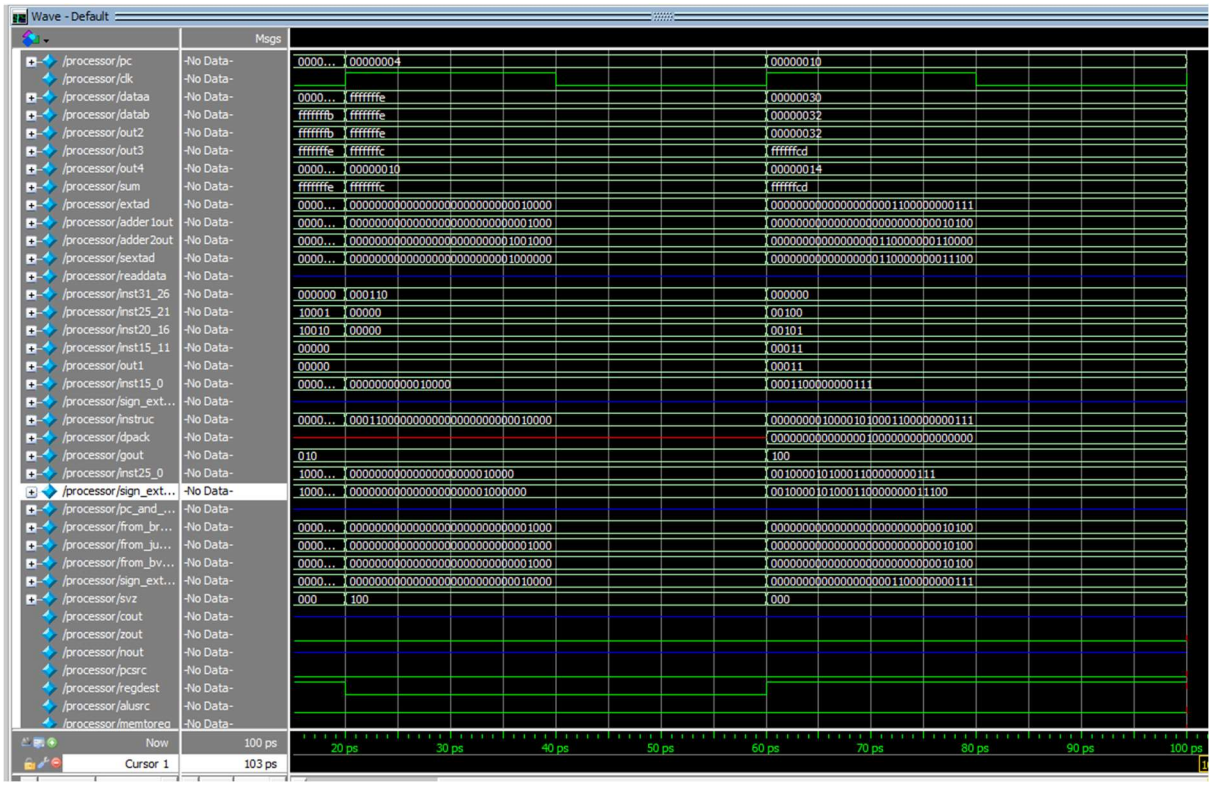


Figure 11: Branch is taken correctly and PC is set to 0x10 accordingly

Sum is ffffffe, and dataa and datab are consistent (Greensheet tells me that s1 is 17. register and s2 is 18. register so I did the rest of the initreg to fill until I reached s1 and s2.) On the next cycle, this addition instruction will set the SVZ to 100 from 000 (can be seen near bottom.)

2) Since now S register is set to 1, ben branch will be taken, on the third cycle, PC is set to 0x10 (which checks out with the second cycle's out4 which sets PC. I added an additional nor at the end where ben should go to in order to show that instruction executes correctly, the bit sequence of dataa and datab can also be checked on the second image to see that nor operation is correct whose result is showed in sum) Notice in the picture below how ben is set to 1 and then is reset after taking the branch. SVZ registers are also reset after taking the branch.

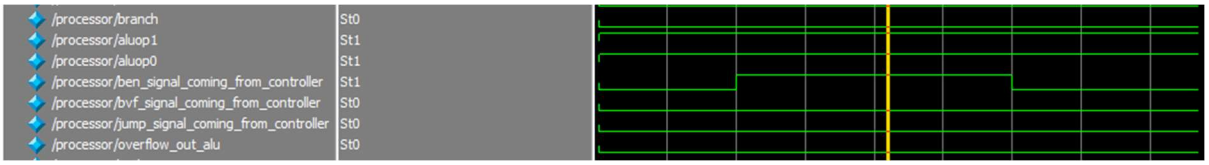


Figure 12: Ben signal is active for the second cycle and resets after taking the branch

Example 2:

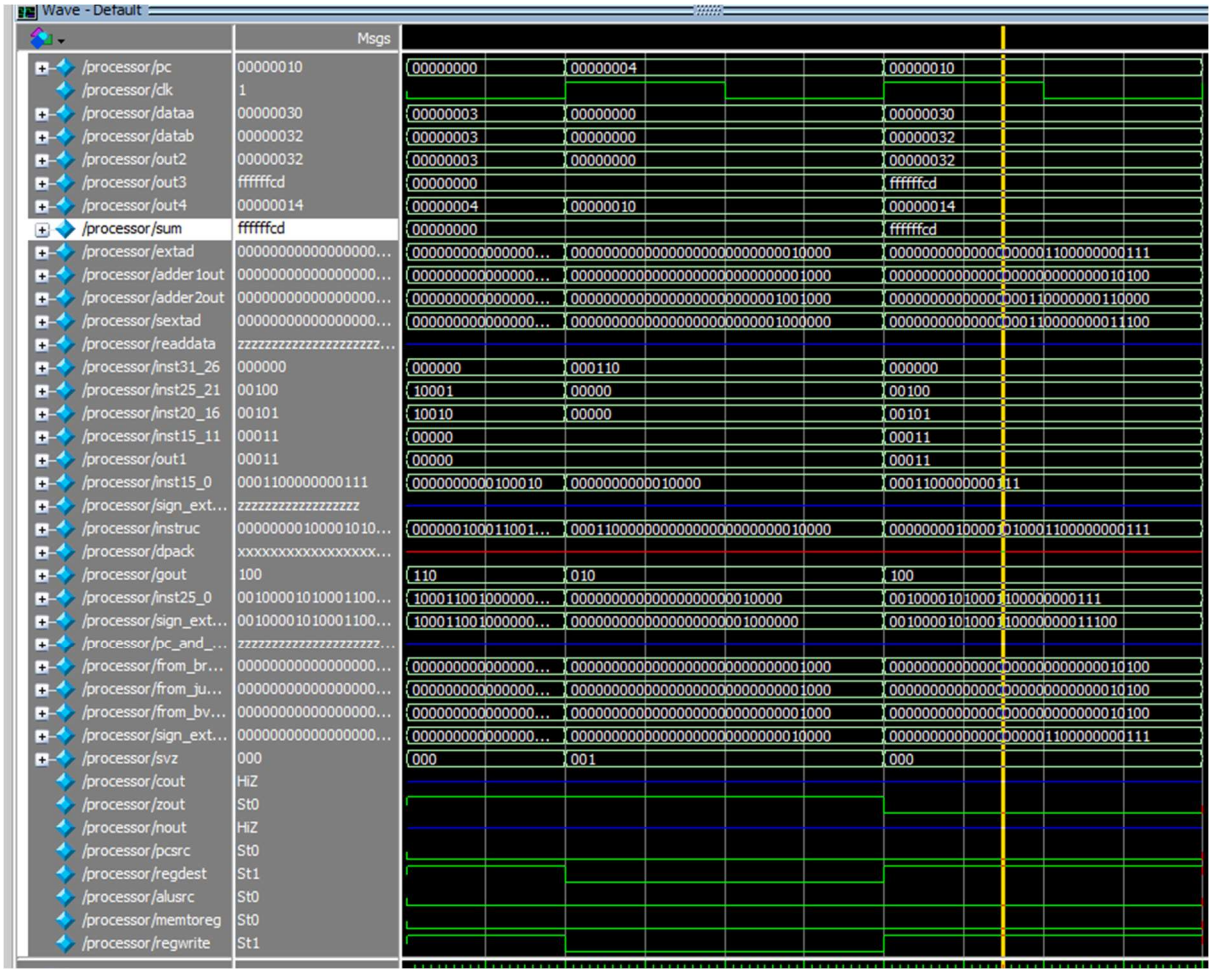


Figure 13: Example 2 wave view

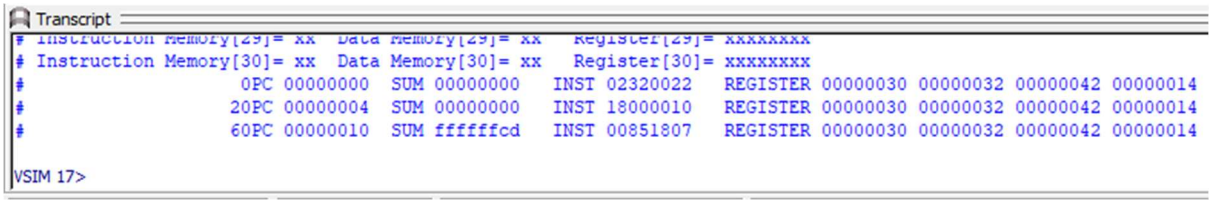


Figure 14: Example 2 terminal view

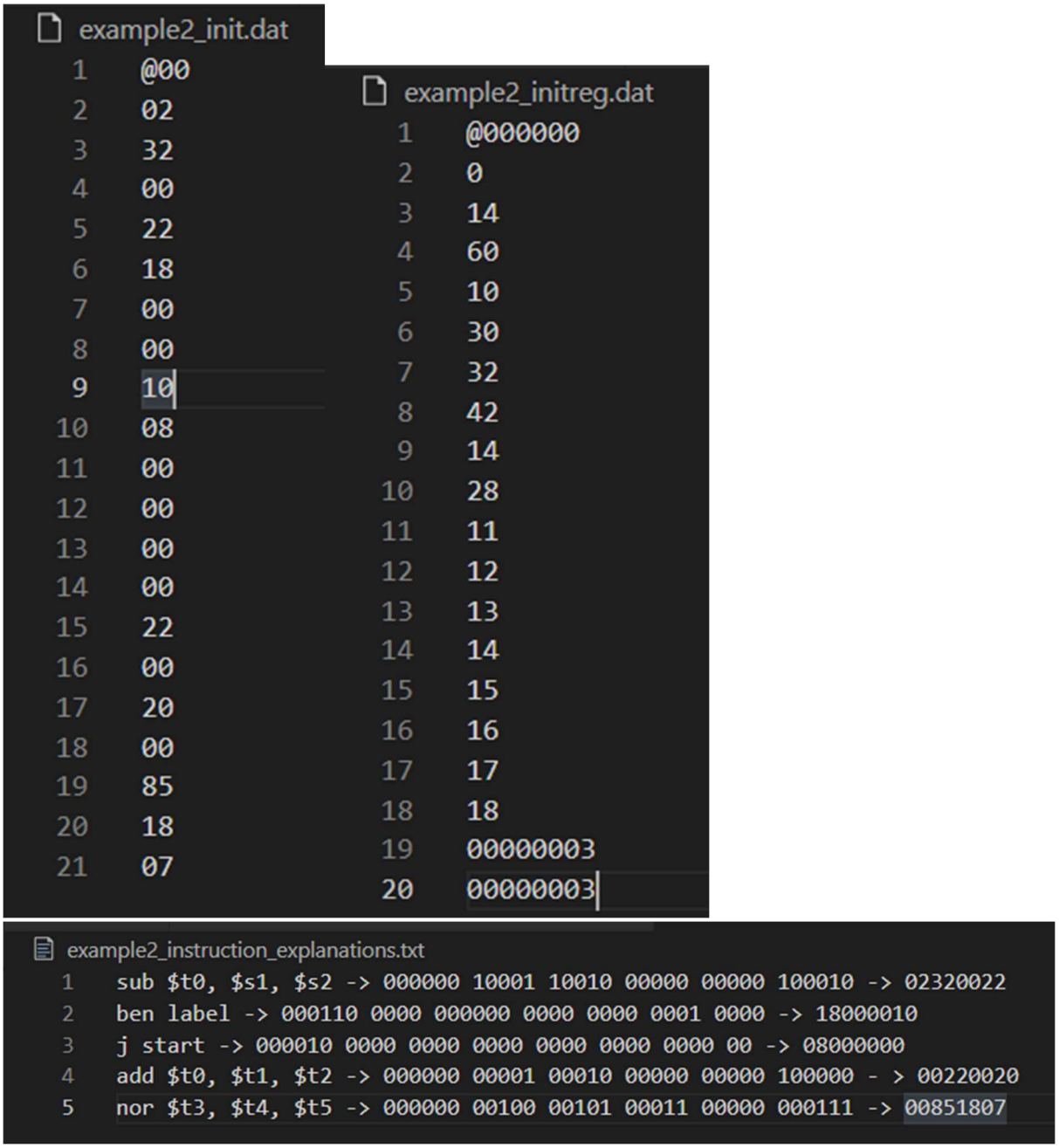


Figure 15, 16, 17: Instruction memory, register contents and explanations of instructions

How it works:

1) This second example is not much different from the first example. Only the first instruction is sub instead of add, which leads to a zero flag this time and ben will still take the branch, but this time not because of the negative flag but zero flag. Notice

how dataa and datab are equal and their subtraction will lead to zero (seen in sum in wave figure.) This will then set Z register of SVZ to 1 and branch will be taken. Rest is the same with first example.

Example 3:

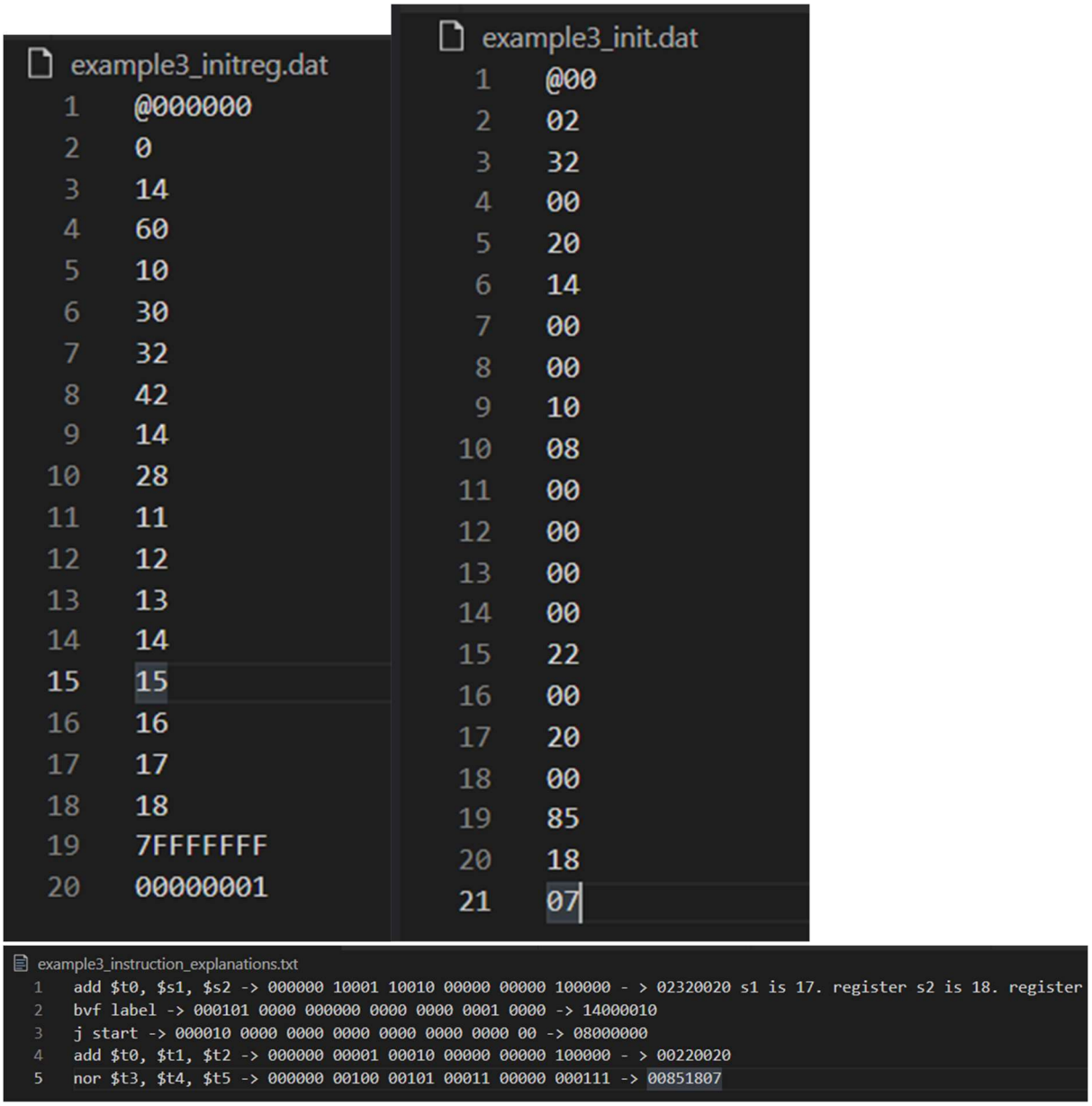


Figure 18, 19, 20: Register content, instruction memory and explanations of the instructions

How it works:

1) Like the first example, we have an addition instruction at the beginning and rest is the same. However, this time, the numbers that are being summed are different, they will set both negative flag and overflow so the bvf can take the branch using overflow flag.

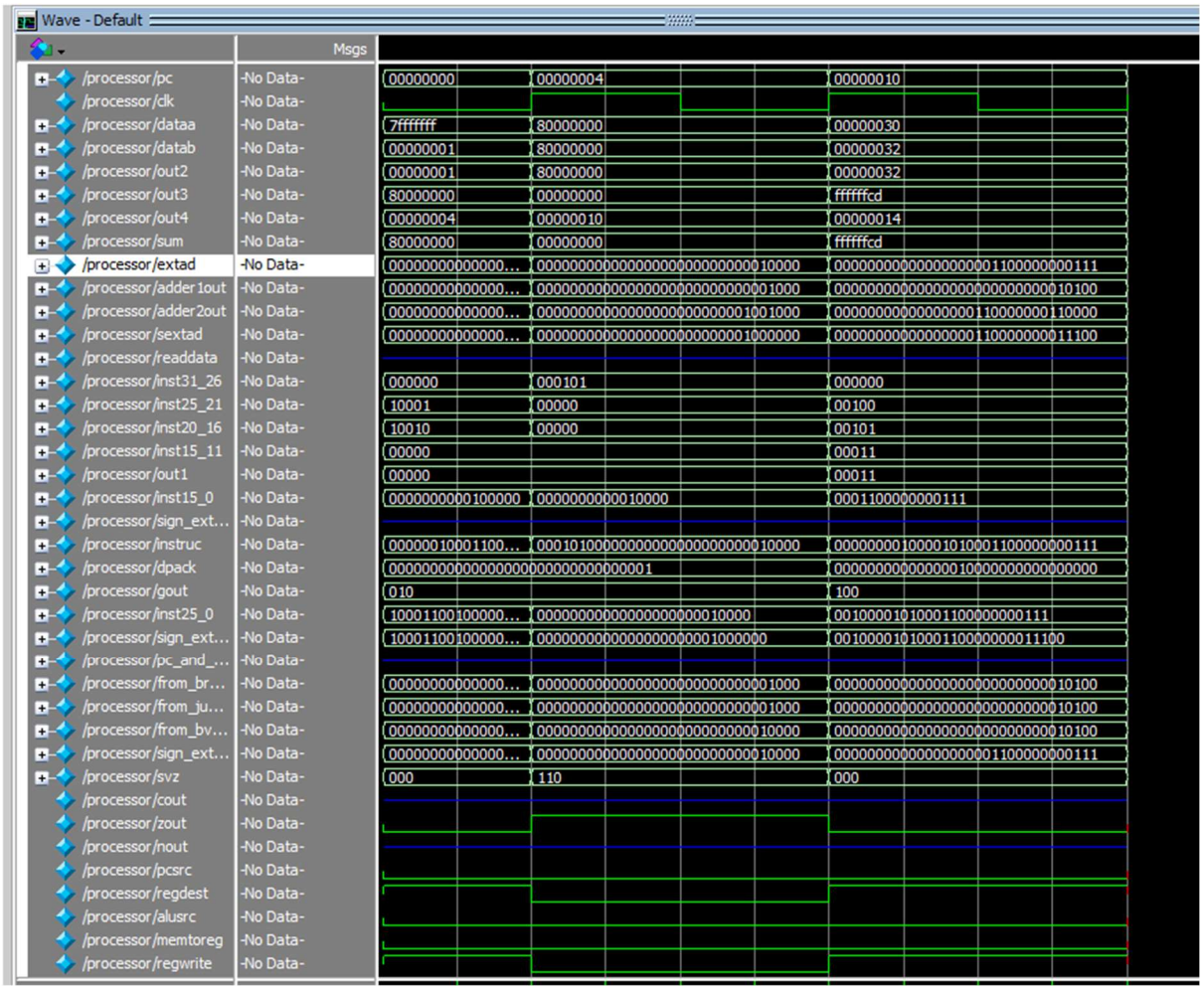


Figure 21: Example 3 wave view, see SVZ is set to 110 from 100 and after taking the branch it is reset to 100. Result of 110 is because the result is both an overflow and negative.

```
# Instruction Memory[29]= xx  Data Memory[29]= xx   Register[29]= xxxxxxxx
# Instruction Memory[30]= xx  Data Memory[30]= xx   Register[30]= xxxxxxxx
#                    0PC 00000000  SUM 80000000   INST 02320020   REGISTER 00000030 00000032 00000042 00000014
#                   20PC 00000004  SUM 00000000   INST 14000010   REGISTER 00000030 00000032 00000042 00000014
#                   60PC 00000010  SUM fffffffcd  INST 00851807   REGISTER 00000030 00000032 00000042 00000014

VSIM 20>
```

Figure 22: Terminal view of Example 3

EXAMPLE TEST CODE

Below is the implementation of the example test code which was given at the end of the PDF file:

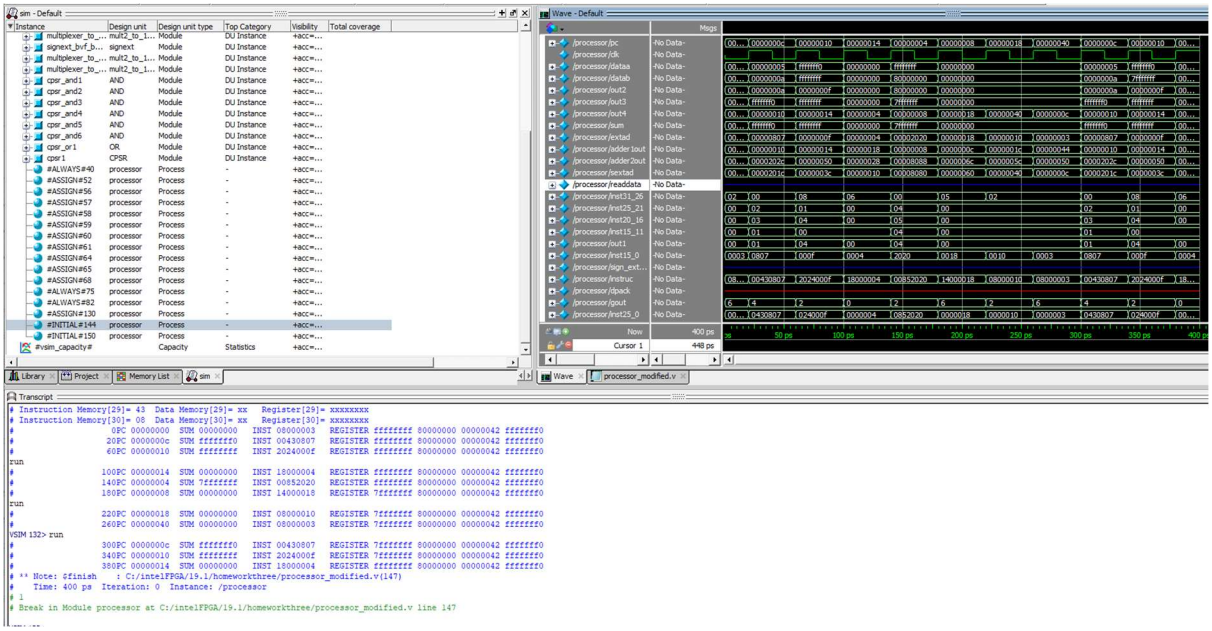


Figure 22: Example test code, terminal and wave view together

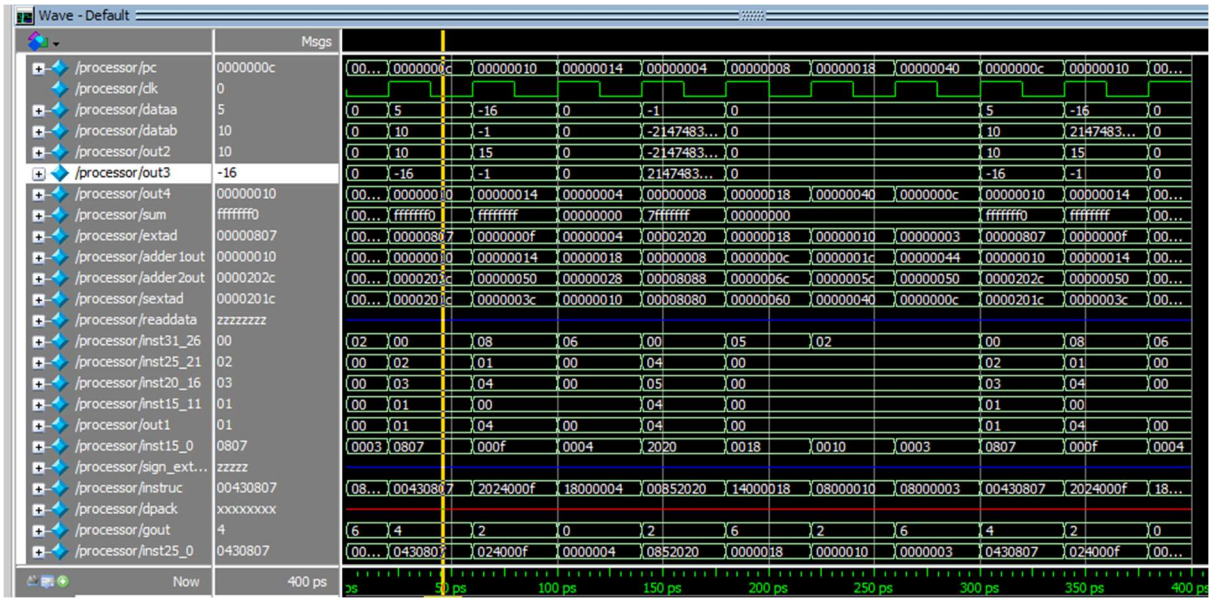
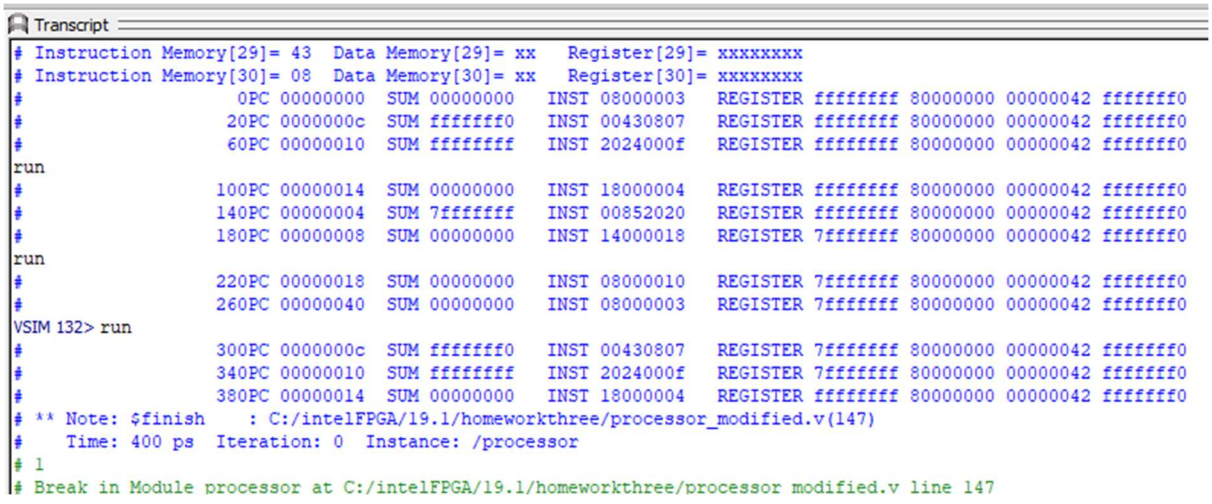


Figure 23: Example test code, wave view



```
Transcript
# Instruction Memory[29]= 43  Data Memory[29]= xx   Register[29]= xxxxxxxx
# Instruction Memory[30]= 08  Data Memory[30]= xx   Register[30]= xxxxxxxx
#                    0PC 00000000  SUM 00000000   INST 08000003   REGISTER ffffffff 80000000 00000042 fffffff0
#                   20PC 0000000c  SUM fffffff0   INST 00430807   REGISTER ffffffff 80000000 00000042 fffffff0
#                   60PC 00000010  SUM ffffffff   INST 2024000f   REGISTER ffffffff 80000000 00000042 fffffff0
run
#                  100PC 00000014  SUM 00000000   INST 18000004   REGISTER ffffffff 80000000 00000042 fffffff0
#                  140PC 00000004  SUM 7fffffff   INST 00852020   REGISTER ffffffff 80000000 00000042 fffffff0
#                  180PC 00000008  SUM 00000000   INST 14000018   REGISTER 7fffffff 80000000 00000042 fffffff0
run
#                  220PC 00000018  SUM 00000000   INST 08000010   REGISTER 7fffffff 80000000 00000042 fffffff0
#                  260PC 00000040  SUM 00000000   INST 08000003   REGISTER 7fffffff 80000000 00000042 fffffff0
VSIM 132> run
#                  300PC 0000000c  SUM fffffff0   INST 00430807   REGISTER 7fffffff 80000000 00000042 fffffff0
#                  340PC 00000010  SUM ffffffff   INST 2024000f   REGISTER 7fffffff 80000000 00000042 fffffff0
#                  380PC 00000014  SUM 00000000   INST 18000004   REGISTER ffffffff 80000000 00000042 fffffff0
# ** Note: $finish    : C:/intelFPGA/19.1/homeworkthree/processor_modified.v(147)
#    Time: 400 ps  Iteration: 0  Instance: /processor
# 1
# Break in Module processor at C:/intelFPGA/19.1/homeworkthree/processor_modified.v line 147
```

Figure 24: Example test code, terminal view

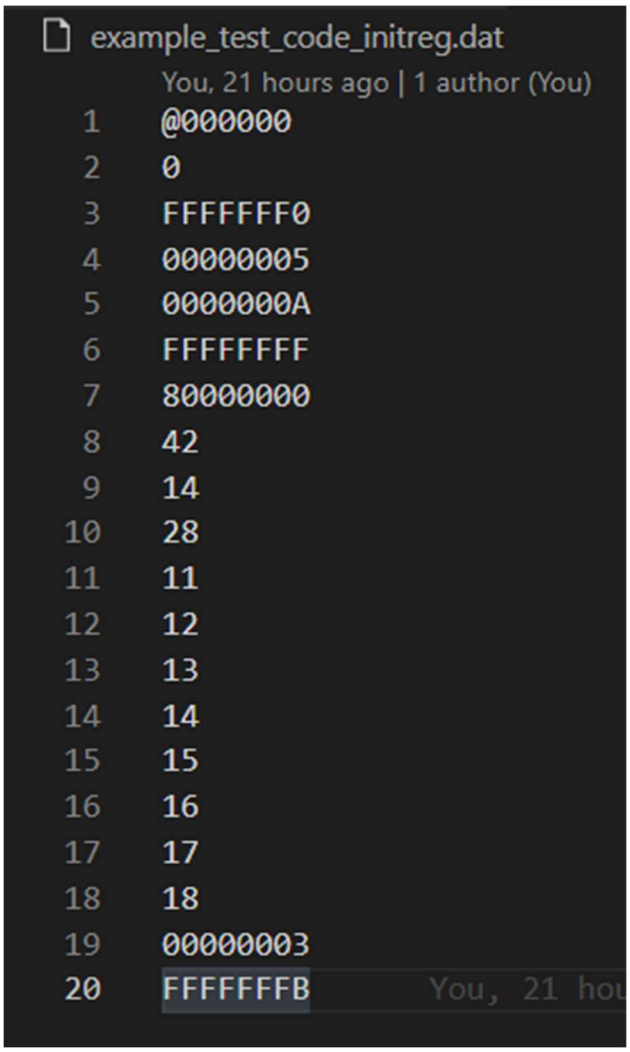
Instruction order can be traced via console above and the hexadecimal conversions below. After coming to an end, I think ModelSim starts again, that's why we have the 08000003 instruction after jump to the end (08000010) and it executes the commands from the start. I added filling sub instructions to show that the jump instruction executes correctly.

```
j 0x0C -> 000010 0000 0000 0000 0000 0000 0000 11 -> 08000003
add $r4, $r4, $r5 -> 000000 00100 00101 00100 00000 100000 - > 00852020
bvf 0x18 -> 000101 0000000000 0000000000011000 -> 14000018
nor $r1, $r2, $r3 -> 000000 00010 00011 00001 00000 000111 -> 00430807
addi $r4, $r1, 0x0F -> 001000 00001 00100 0000 0000 0000 1111 -> 2024000F
ben 0x04 -> 000110 0000000000 0000000000000100 -> 18000004
j 0x40 -> 000010 000000000000000000 0001 0000 -> 08000010
sub $r1, $r2, $r3 -> 000000 00010 00011 00001 00000 001000 -> 00430808
sub $r1, $r2, $r3 -> 000000 00010 00011 00001 00000 001000 -> 00430808
sub $r1, $r2, $r3 -> 000000 00010 00011 00001 00000 001000 -> 00430808
sub $r1, $r2, $r3 -> 000000 00010 00011 00001 00000 001000 -> 00430808
sub $r1, $r2, $r3 -> 000000 00010 00011 00001 00000 001000 -> 00430808
sub $r1, $r2, $r3 -> 000000 00010 00011 00001 00000 001000 -> 00430808
sub $r1, $r2, $r3 -> 000000 00010 00011 00001 00000 001000 -> 00430808
sub $r1, $r2, $r3 -> 000000 00010 00011 00001 00000 001000 -> 00430808

execution order must be -> jump, nor, addi, ben, add, bvf, jump
execution order must be -> 08000003, 00430807, 2024000F, 18000004, 00852020, 14000018, 08000010
```

Here is the init file used for this example:

```
@00
08
00
00
03
00
85
20
20
14
00
00
18
00
43
08
07
20
24
00
0F
18
00
00
04
08
00
00
10
00
43
08
08
00
43
08
08
00
43
08
08
00
43
08
08
00
43
08
08
00
43
08
08
00
43
08
08
00
43
08
08
```

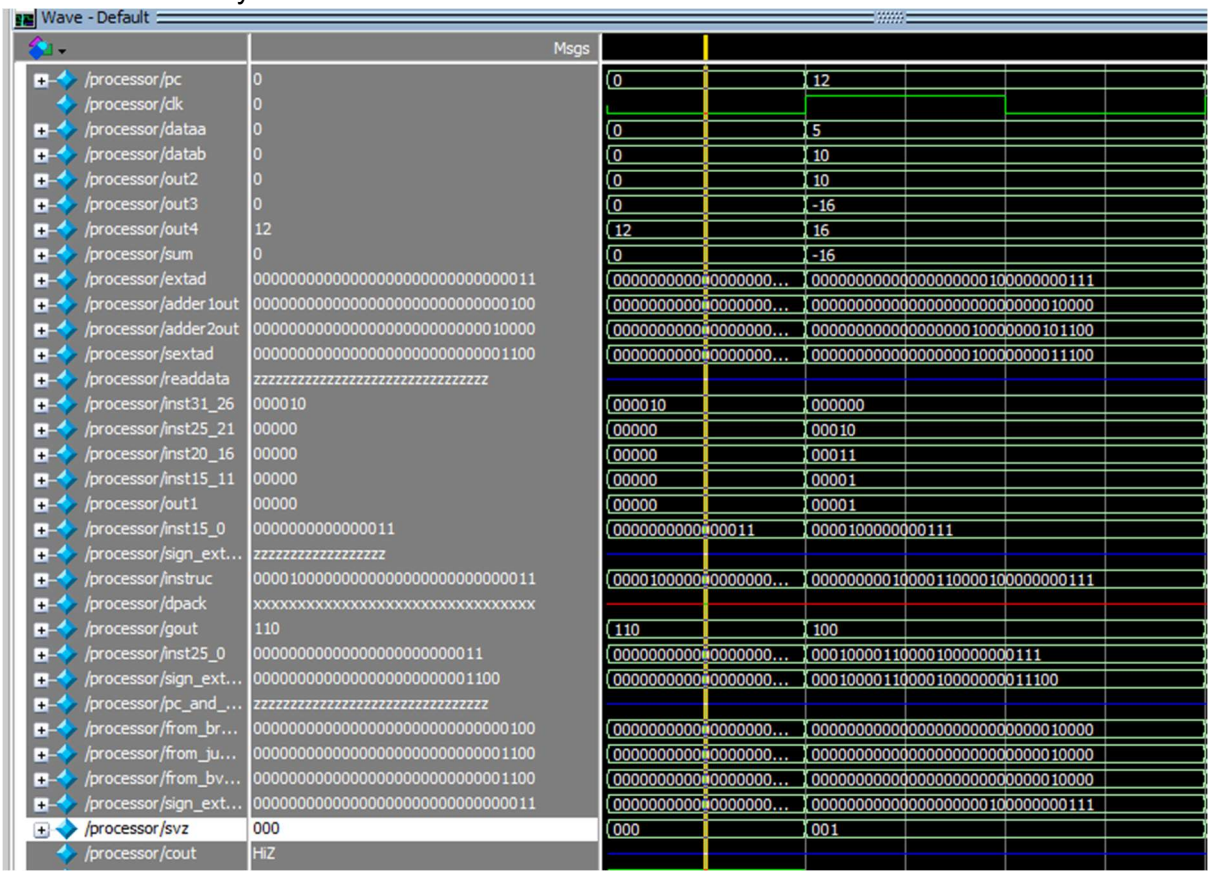
Content of the registers:



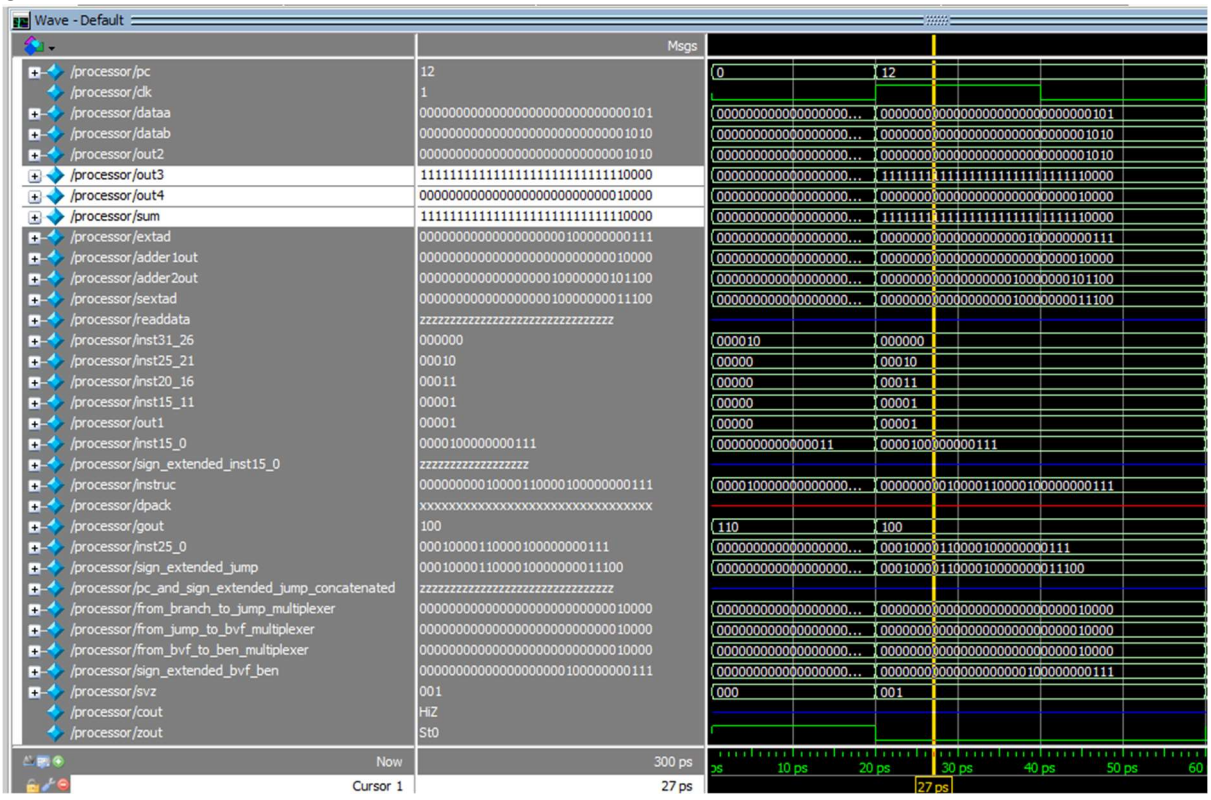
How it works:

1) The first instruction is a jump instruction to the 0x0C, Since jump instructions are shifted left by two, to represent a jump to the location 0C I have written 3 in hexadecimal (this will be 12 after multiplication by four). As it can be seen PC changes to 12 after the first instruction here, svz changes from 000 to 001 because dataa and datab are 0 and the ALU result will be zero (even if we didn't use ALU gout can be seen to be 110 on the first cycle which signals for subtraction, that's why zero flag is set). out4 is what sets PC and it can be seen to be 12 before the PC changes.

in the next clock cycle:



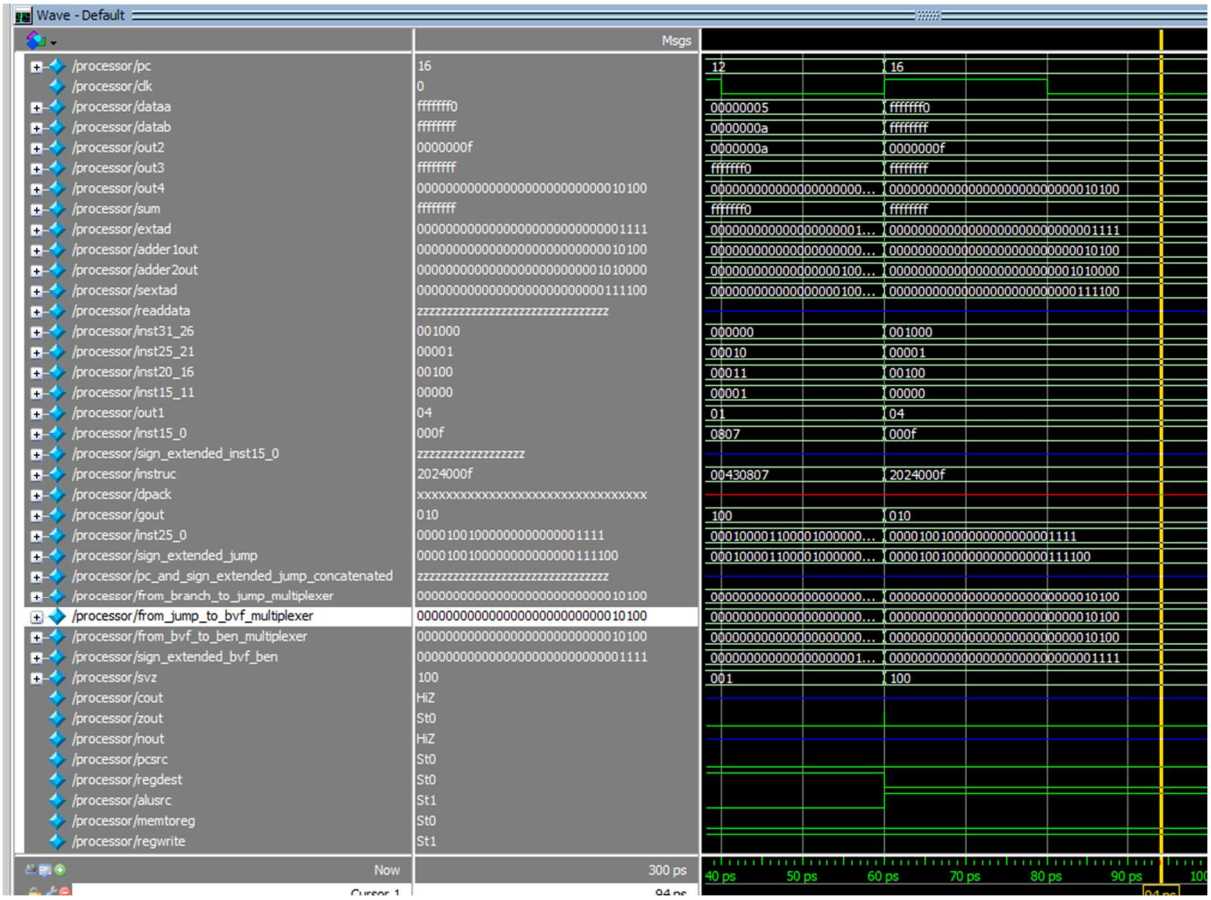
2) Second instruction is nor'ing instruction, contents of r2 and r3 registers are exactly as given in the PDF file.



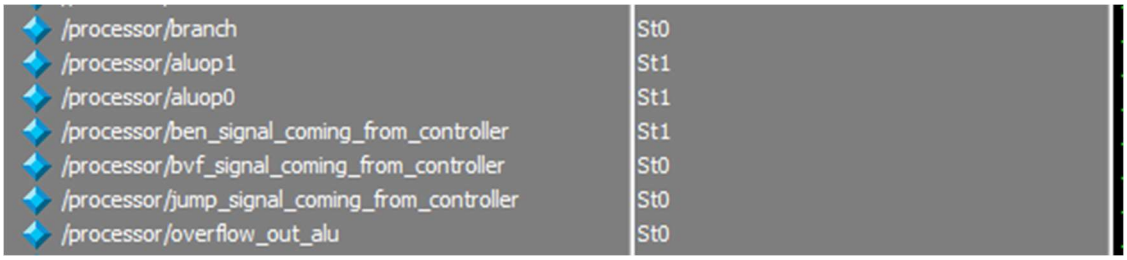
dataa and datab will have their last four bits 1 after or'ing but they will be later the only ones which are 0 because nor operation is equivalent to first or'in then taking not

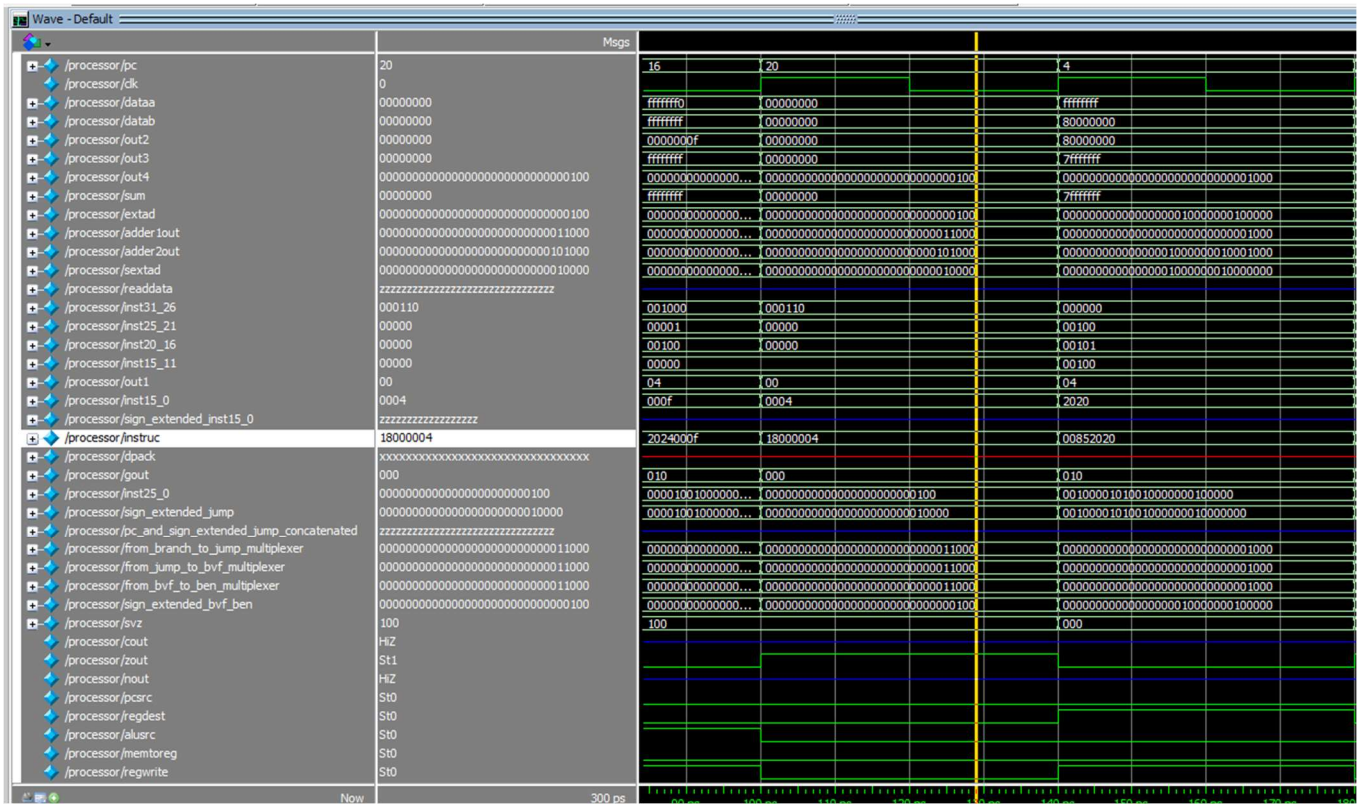
of every bit. So the result on sum wire checks out. Since the most significant bit after nor operation is 1, S of SVZ register is set to 1.

3) Next instruction is addi, inside r1 we had 0xFFFFFFF0 coming from last instruction and we add this with a constant 0x000F, the result is 0xFFFFFFFF (can be seen in sum) which is assigned to r4. dataa can be seen to demonstrate r1, out2 represents the sign extended instruction[15-0] which was the constant. This constant is taken to ALU because ALUSrc is set to 1.

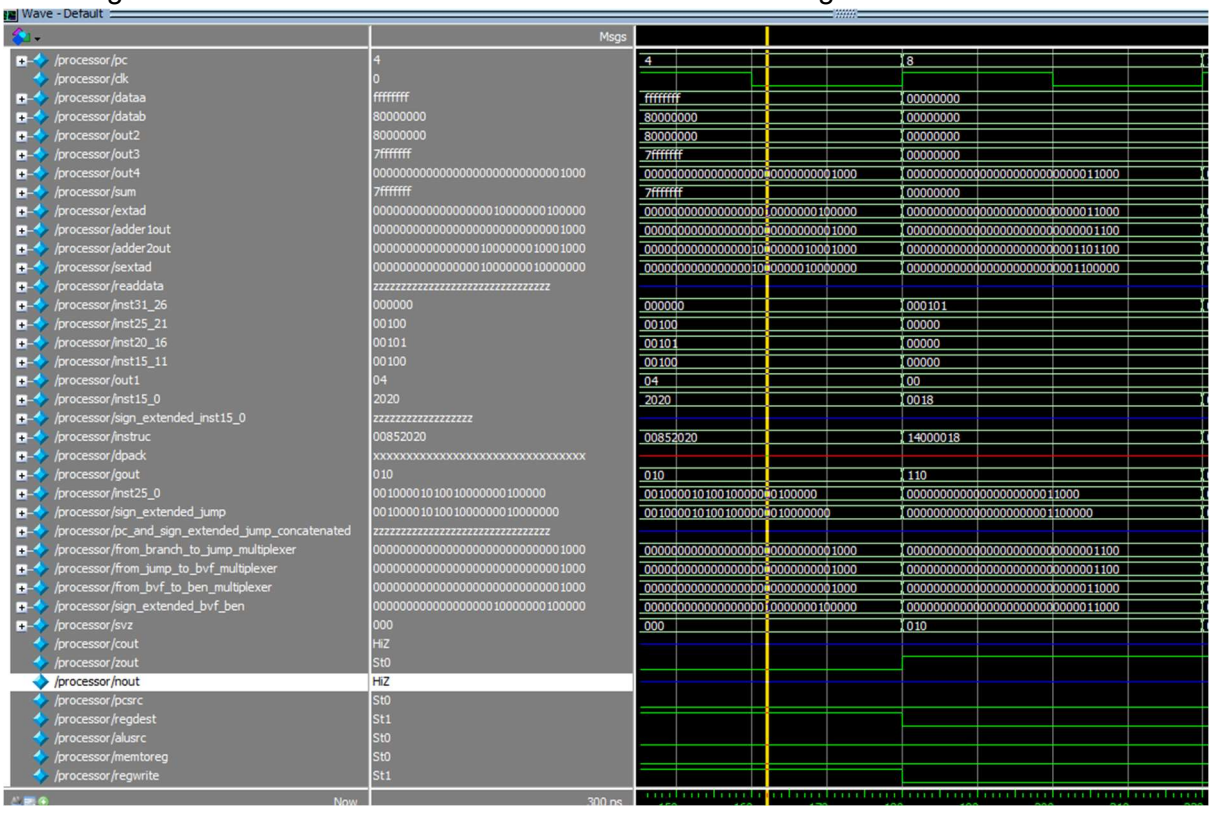


4) Next instruction is ben 0x04, since S register of SVZ was set to 1 in the previous instruction, ben branch will be taken. Notice how PC is set to 4 after branch is taken and SVZ is reset. Also the multiplexer to choose ben is activated and the other multiplexers are set to zero on the design which are on top (jump, branch, bvf). So out4 will be the constant used in the ben instruction (see out4 on the next image which sets the pc.)

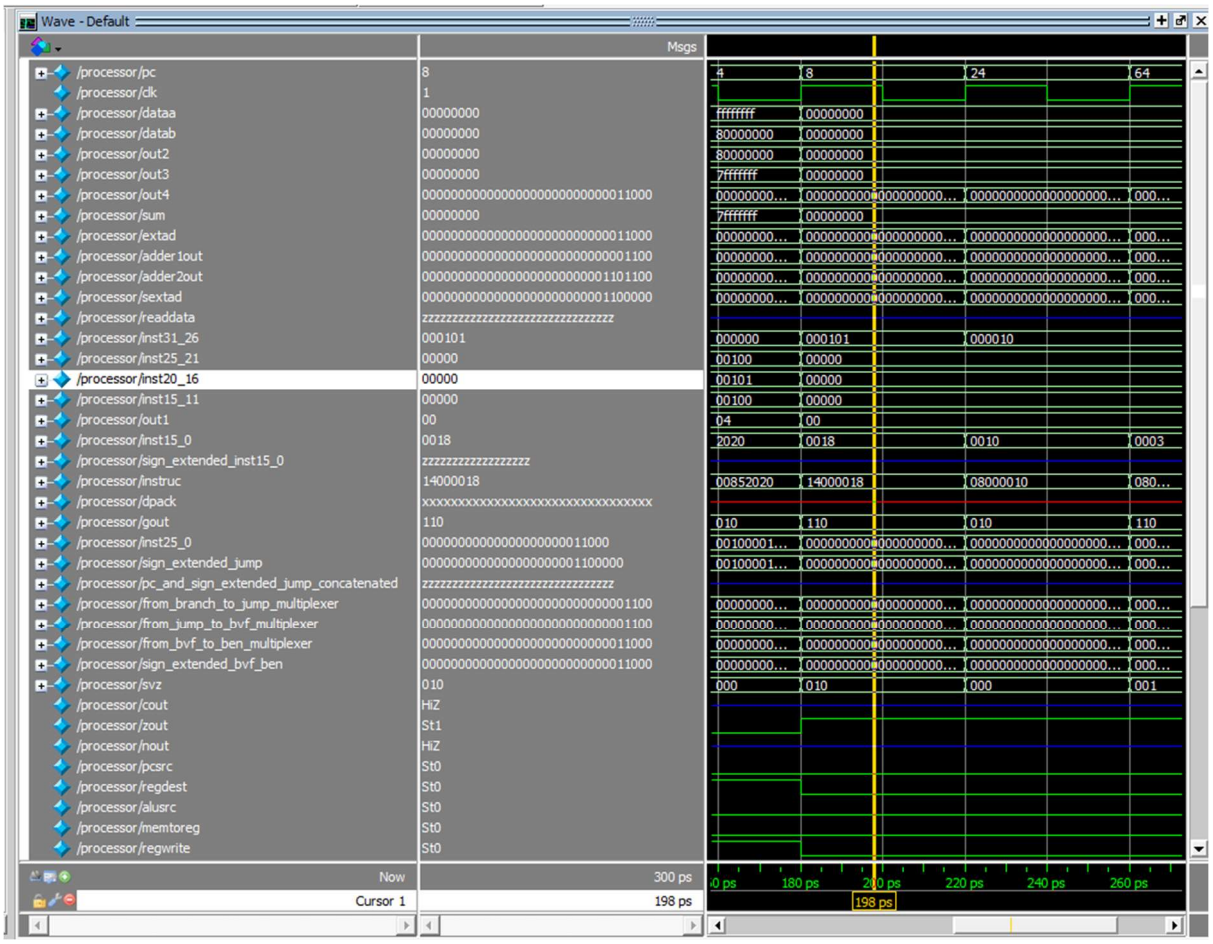




5) Next instruction is add, looking at dataa and datab we are adding 0xFFFFFFFF 0x80000000. Result is seen to be 0x7FFFFFFF. This causes an overflow because the numbers used for the addition were both negative and result is positive now. So the V register of the SVZ will be set to 1 as it is seen in the image below.



6) Next instruction is bvf and we are jumping to 0x18 because V was set to 1 from the previous instruction. Notice how PC is set to 24 from 8 and SVZ is reset from 010 to 000



7) Last instruction is jump which is found at 0x18. See how only the multiplexer for jump is active for this instruction for the multiplexers which are found on the top of the design. PC change can be seen on the image above. It is correctly set to 64 (see hexadecimal 40 in the below picture in out4, which sets PC). There are no instructions to execute there (I filled the sub instructions to show that program correctly skips them) and the program terminates.