New Instructions

For new R type operations, I have added 1 more output to alu which is **n**. This output becomes 1,2 or 3 depending on alu result.

New Registers for New Instructions

Since new R type instructions needs to write to 2 registers in 1 cycle, I have added 2 more inputs to register block, which are **write_register_2** and **write_data_2**.

I noticed that the **write_register_2** is always **rs** and **write_data_2** is always alu result and designed according to that.

JR Case

Since jr is also R type, it kinda makes the design uglier. To handle JR I have added 1 more signal **jrsig** to **alu control** since, jr is R type and all of the R type instructions opcode are same. I needed to check the funct code and figure out it is **jr**.

R types can write both of the registers, so all of the R types has write signals, but JR is also an R type and it should not write to any register. To fix that, i have **and** gated the regwrite signals with **!jrsig**.

Lui Case

I forgot to add 16 bit shifter to this image but it is there in quartus project. The **luisig** mux **1** input is actually **16bit left shifted zero extended immediate**.

Ori Case

Since ori needs **zero extended immediate** for operation, I have added 1 more input to **alu input mux**

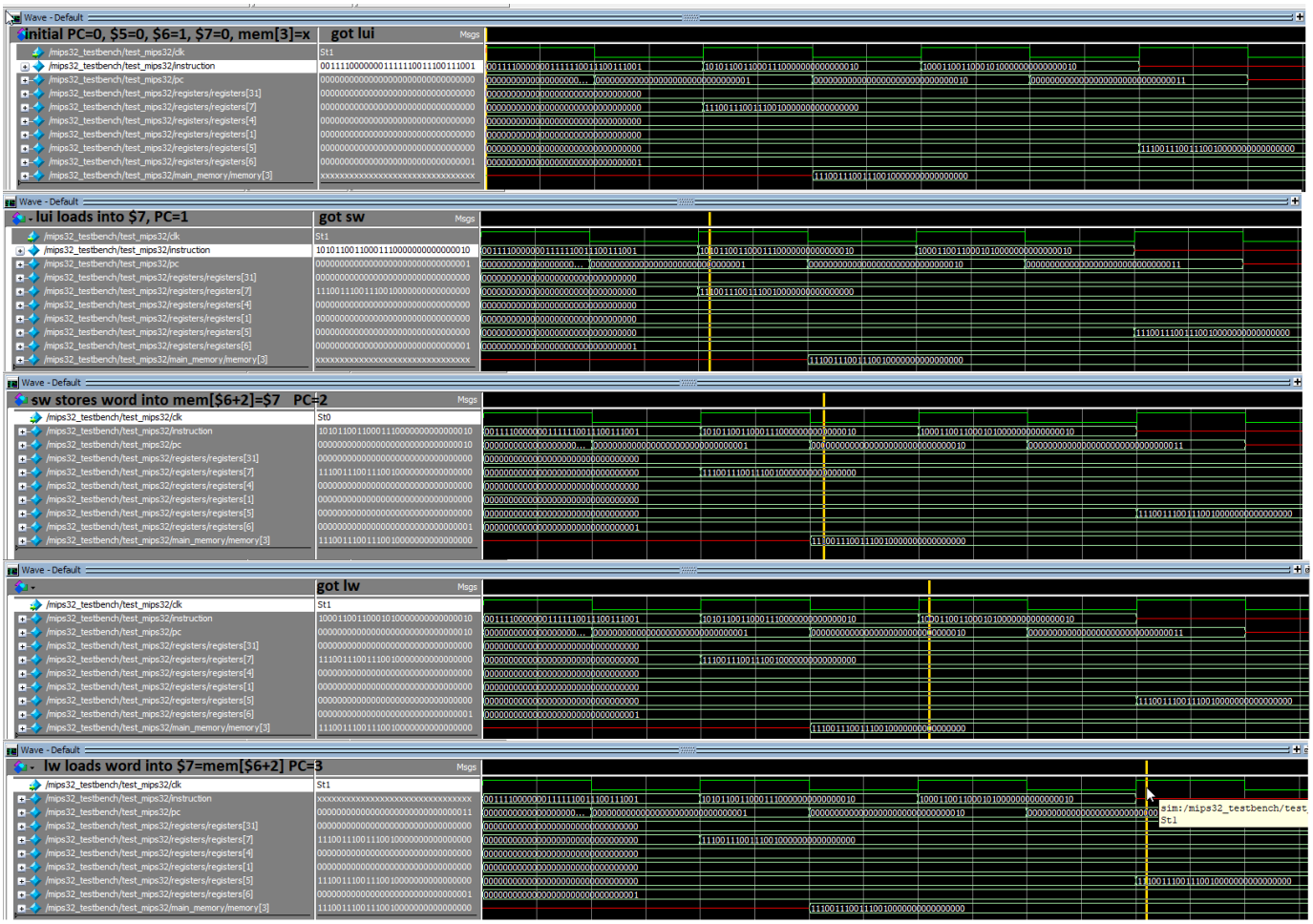
Tests

- I have included edge cases and basic cases for instructions.
- I have included every cycle state, taking the screenshot from multisim wave viewer.
- You can find this instructions inside "instruction.mem", which I have provided with project.
- To use the "instruction.mem", it needs to be placed in "simulation/modelsim"
- I relied on loading initial values to registers with "register.mem" on some tests, so even though the code looks same, they are different cases. You can see the difference in the screenshots.

LUI - SW - LW Test

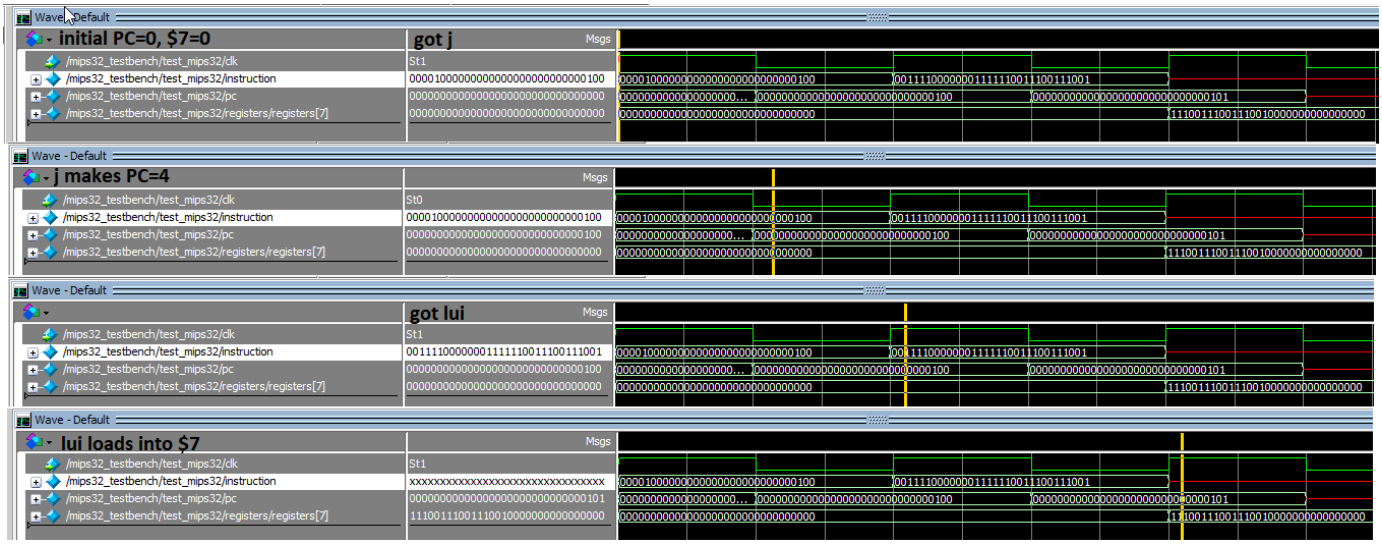
```
// lui sw lw test
// lui $7 1110011100111001
00111100000001111110011100111001
// sw $7 2($6)
```

```
10101100110001110000000000000010
// lw $5 2($6)
10001100110001010000000000000010
```



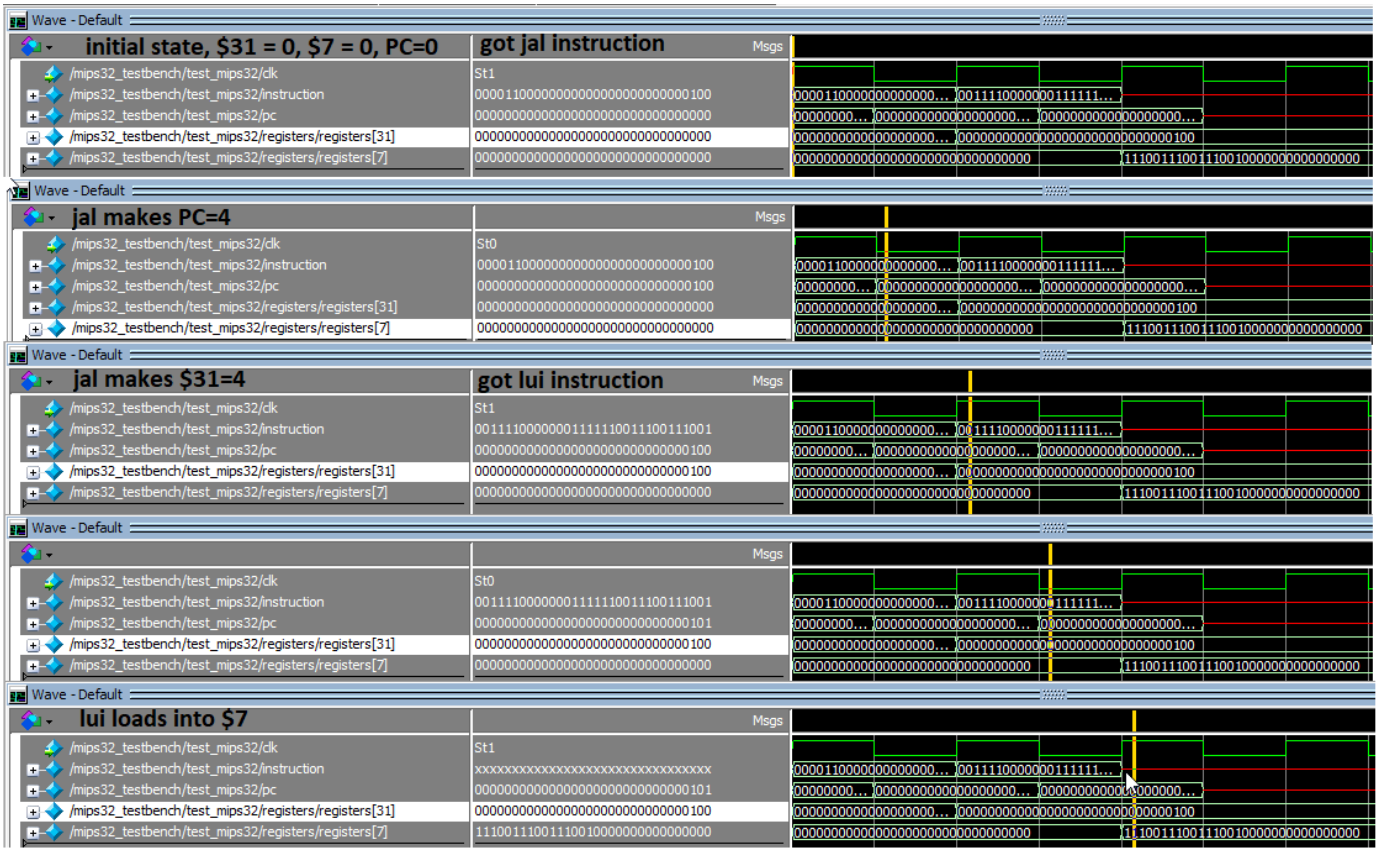
J Test

```
// j test
// j 4
00001000000000000000000000000100
// sw $2 2($1) (dummy will jump over)
10101100001000010000000000000010
// lw $3 2($1) (dummy will jump over)
10001100001000110000000000000010
// lw $3 2($1) (dummy will jump over)
10001100001000110000000000000010
// lui $7 1110011100111001
00111100000001111110011100111001
```



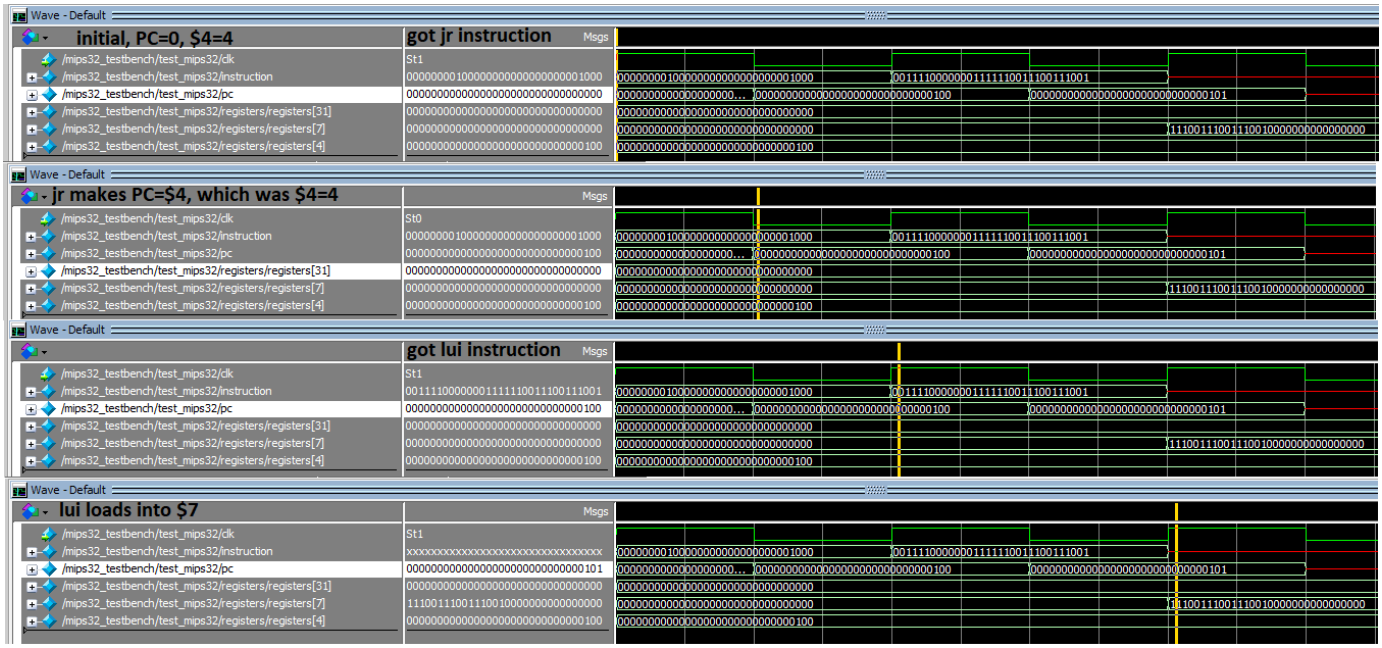
JAL Test

```
// jal test
// jal 4
00001100000000000000000000000100
// sw $2 2($1) (dummy will jump over)
10101100001000100000000000000010
// lw $3 2($1) (dummy will jump over)
10001100001000110000000000000010
// lw $3 2($1) (dummy will jump over)
10001100001000110000000000000010
// lui $7 1110011100111001
00111100000001111110011100111001
```



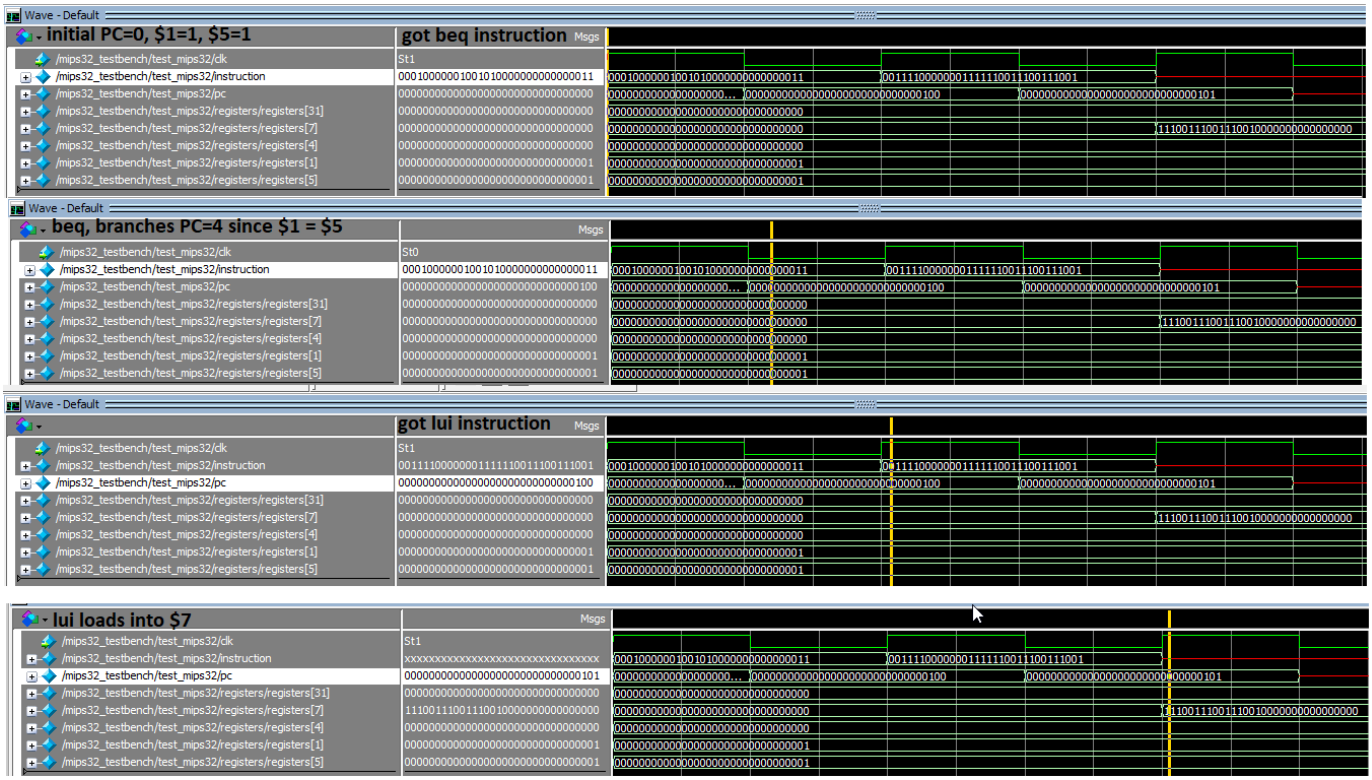
JR Test

```
// jr test
// jr $4
00000000100000000000000000001000
// sw $2 2($1) (dummy will jump over)
10101100001000100000000000000010
// lw $3 2($1) (dummy will jump over)
10001100001000110000000000000010
// lw $3 2($1) (dummy will jump over)
10001100001000110000000000000010
// lui $7 1110011100111001
00111100000001111110011100111001
```



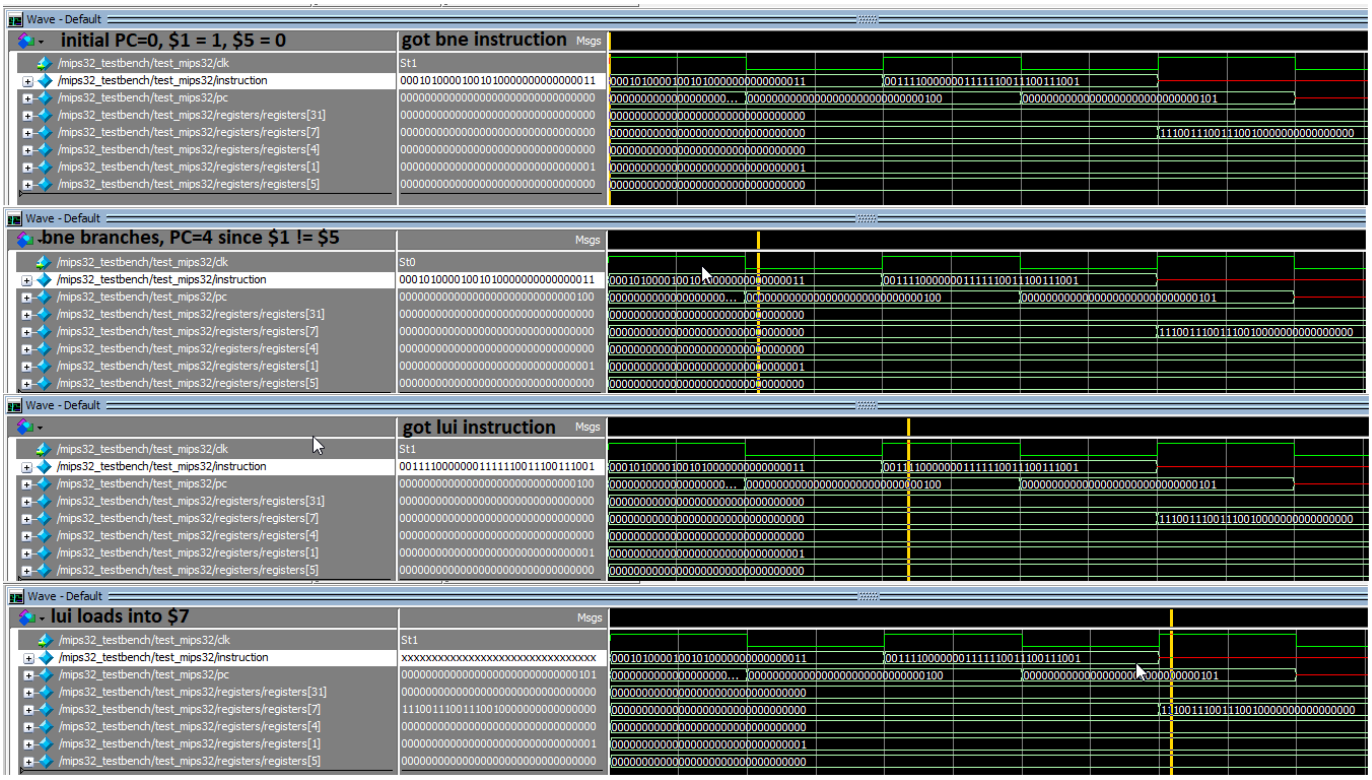
BEQ Test

```
// beq test
// beq $1 $5 to 4th instruction
00010000001001010000000000000011
// sw $2 2($1) (dummy will jump over)
10101100001000100000000000000010
// lw $3 2($1) (dummy will jump over)
10001100001000110000000000000010
// lw $3 2($1) (dummy will jump over)
10001100001000110000000000000010
// lui $7 1110011100111001
00111100000001111110011100111001
```



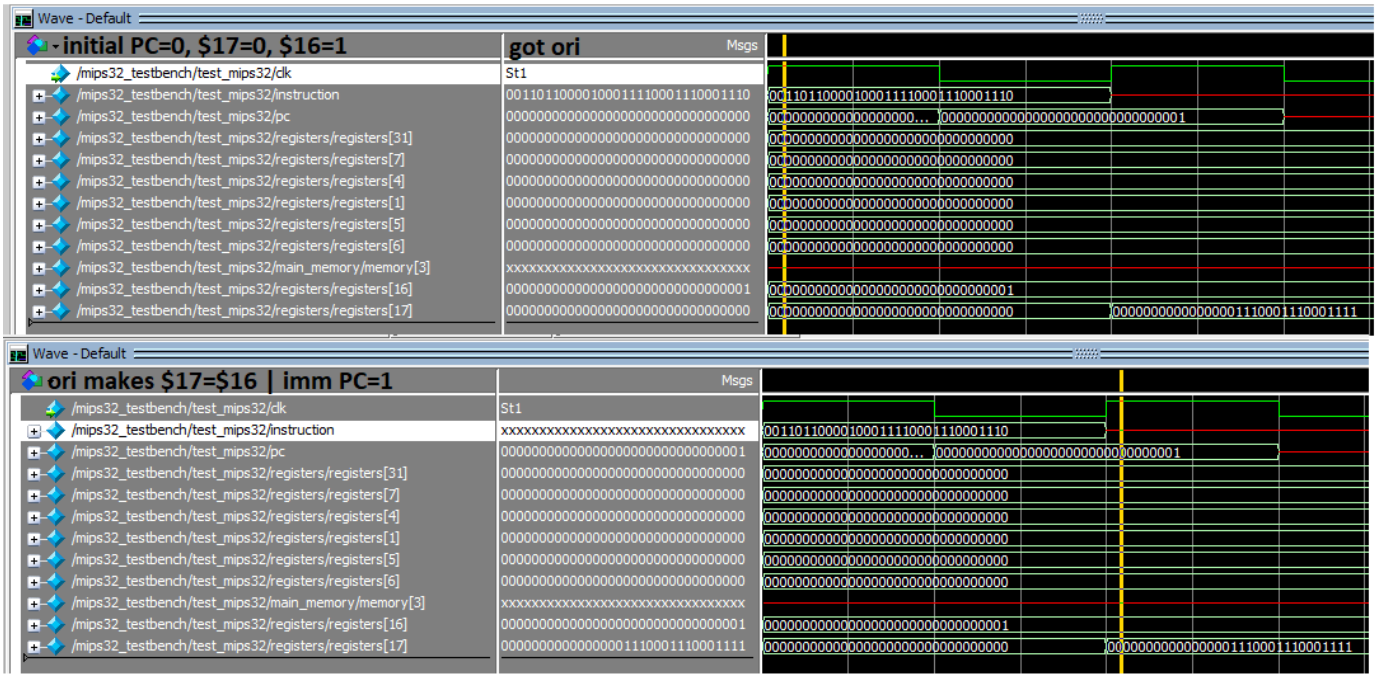
BNE Test

```
// bne test
// bne $1 $5 to 4th instruction
00010100001001010000000000000011
// sw $2 2($1) (dummy will jump over)
10101100001000100000000000000010
// lw $3 2($1) (dummy will jump over)
10001100001000110000000000000010
// lw $3 2($1) (dummy will jump over)
10001100001000110000000000000010
// lui $7 1110011100111001
00111100000001111110011100111001
```



ORI Test

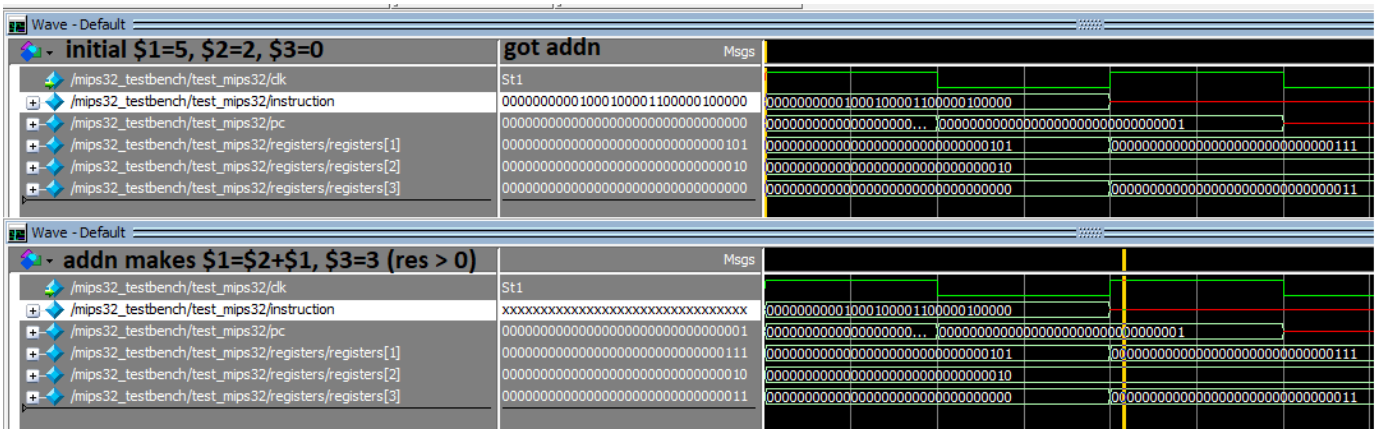
```
// ori test
// ori $17 $16 1110001110001110
00110110000100011110001110001110
```



ADDN Test

A test for basic addition, makes N value 3.

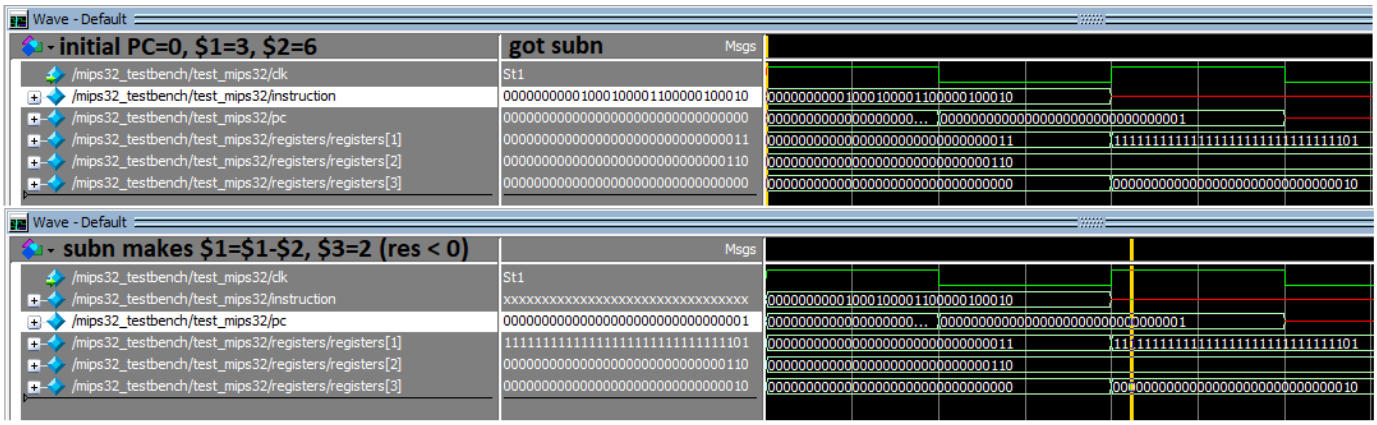
```
// addn test
// addn $3 $1 $2
00000000001000100001100000100000
```



SUBN Test

Here is a test for negative result, which making N value 2.

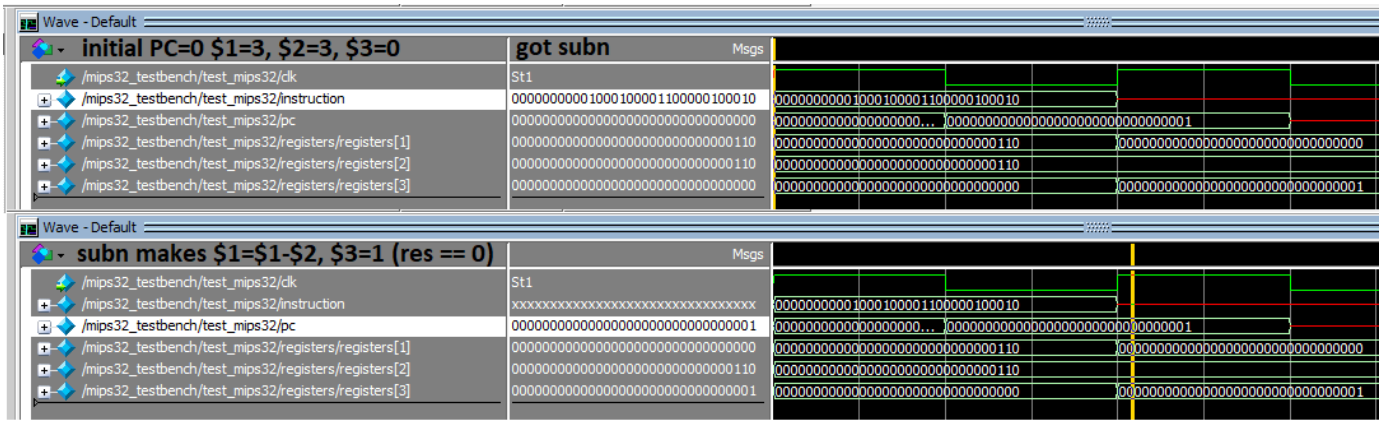
```
// subn test
// subn $3 $1 $2
00000000001000100001100000100010
```



SUBN Zero Test

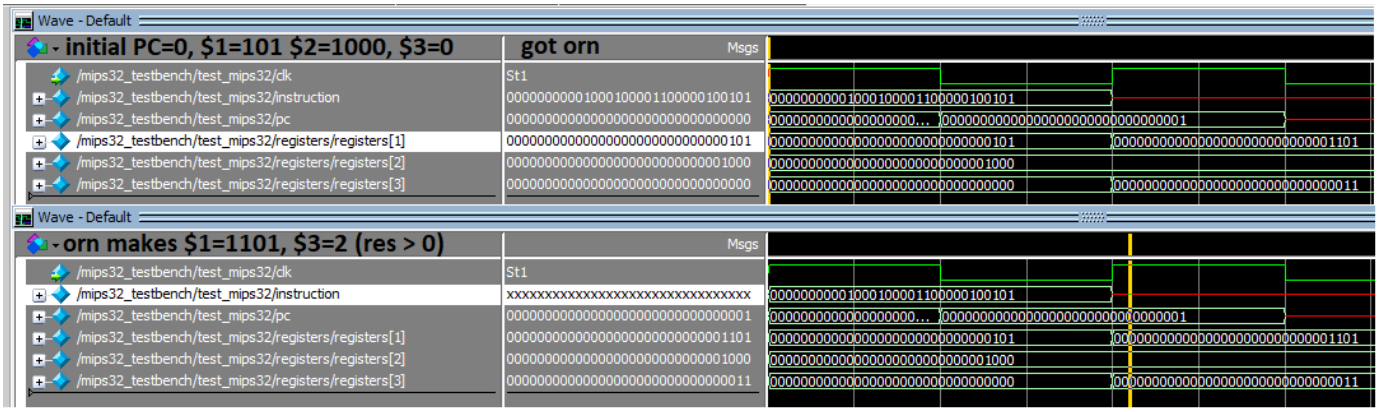
Here I changed the register contents to make result 0, thus making N value 1.

```
// subn test
// subn $3 $1 $2
00000000001000100001100000100010
```



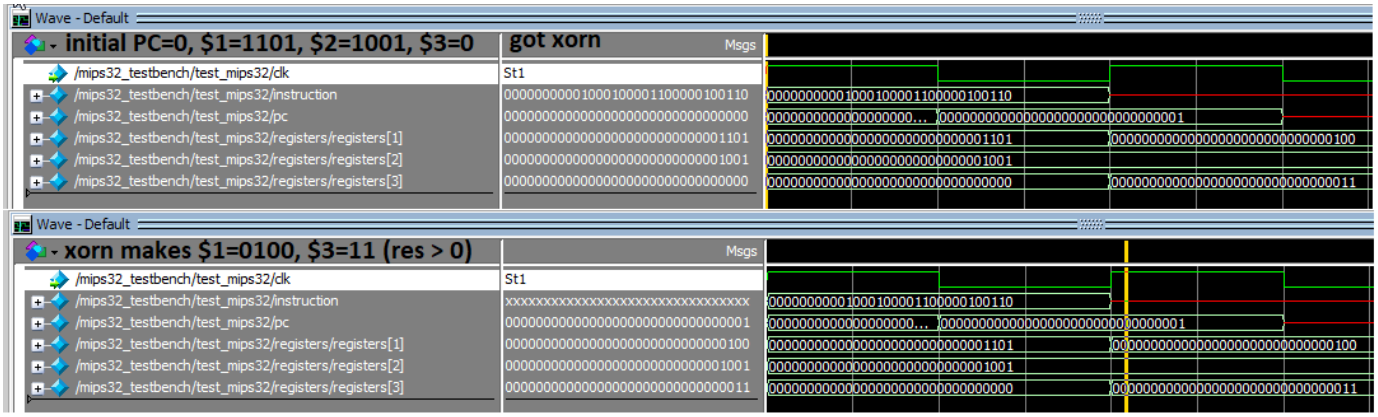
ORN Test

```
// orn test
// orn $3 $1 $2
00000000001000100001100000100101
```



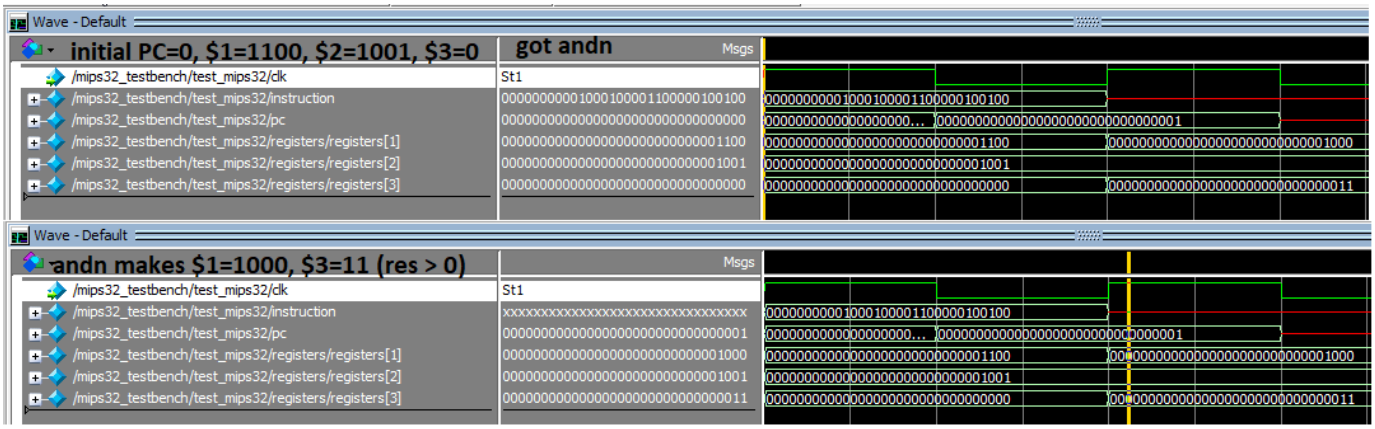
XORN Test

```
// xorn $3 $1 $2
00000000001000100001100000100110
```



ANDN Test

```
// andn $3 $1 $2
00000000001000100001100000100100
```


Wave - Default
initial PC=0, $1=1100, $2=1001, $3=0
got andn
Msgs
/mips32_testbench/test_mips32/clk
/mips32_testbench/test_mips32/instruction
/mips32_testbench/test_mips32/pc
/mips32_testbench/test_mips32/registers/registers[1]
/mips32_testbench/test_mips32/registers/registers[2]
/mips32_testbench/test_mips32/registers/registers[3]
St1
00000000001000100001100000100100
00000000000000000000000000000000
00000000000000000000000000001100
00000000000000000000000000001001
00000000000000000000000000000000
Wave - Default
andn makes $1=1000, $3=11 (res > 0)
Msgs
/mips32_testbench/test_mips32/clk
/mips32_testbench/test_mips32/instruction
/mips32_testbench/test_mips32/pc
/mips32_testbench/test_mips32/registers/registers[1]
/mips32_testbench/test_mips32/registers/registers[2]
/mips32_testbench/test_mips32/registers/registers[3]
St1
00000000000000000000000000000001
00000000000000000000000000001000
00000000000000000000000000001001
00000000000000000000000000000011